polygon zkEVM

Technical Document

zk-EVM Architecture and Assembly Specification v.1.0

March 14, 2023

Contents

1 Introduction

Integrity is doing the right thing even when no one is watching. - C.S. Lewis. Computational integrity (CI) refers to the assurance that a computation is well done executed and the results of that execution are accurate, reliable, and trustworthy. This document describes the assembly language created by Polygon, designed specifically with Ethereum Virtual Machine (EVM) features to represent blockchain transactional-based computations that could be executed with probable CI.

Ethereum is a decentralized, general-purpose blockchain computer where programs are represented as smart contracts and state transitions are triggered by users through the execution of transactions. One of the key components of Ethereum is the EVM. The EVM is a runtime environment that executes smart contracts on the Ethereum network. It provides a secure and isolated environment for executing code, and it ensures that the code is executed in a predictable and deterministic manner.

When executing a transaction on the Ethereum network, a fee must be paid. The fee is proportional to the complexity of the computation and the demand on the network. The increasing demand for the Ethereum network, combined with its limited capacity, has caused the fees to rise to the point where they may impact the practical usability of the network. To address this issue, several layer 2 (L2) solutions have emerged in the market to improve the usability of Ethereum.

Polygon's zkEVM is a layer 2 network that implements a special instance of the Ethereum Virtual Machine (EVM). Although the network has a different architecture and state from Ethereum layer 1 (L1), communication with the Polygon zkEVM is done through a JSON-RPC interface that is fully compatible with Ethereum RPC, allowing all Ethereum-compatible applications and tools to be natively compatible with Polygon zkEVM. However, it's important to note that this is a separate instance with a distinct state from Ethereum L1, and as such, balances in accounts may not be the same and L1 smart contracts cannot be directly accessed through L2 transactions. Nevertheless, a bridge and cross-chain messaging mechanism enables the exchange of data between both networks (refer to the technical documents regarding the zkEVM bridge).

1.1 Ethereum Virtual Machine Basics

1.2 Introduction

The Ethereum blockchain is a digital ledger that keeps track of all transactions and interactions that occur on the Ethereum network. Ethereum can store and execute smart contracts, which can perform a variety of tasks and operations on the network, in addition to recording transactions. At any given time, a collection of data defines the current state of the Ethereum blockchain. The Ethereum state includes account balances, smart contract code, smart contract storage and other information relevant to the operation of the network. The Ethereum's state is maintained by each network node.

Some key features of the EVM:

- **Deterministic**: This means it will always produce the same output given the same input. This feature is critical for ensuring the dependability and predictability of smart contract execution.

- **Sandboxed**: This means that transactions processed by smart contracts run in an environment that is isolated from the rest of the system, making it impossible for them to access or modify data outside this environment. This contributes to security by preventing unauthorized access to sensitive data.

- **Stack-based**: This means that it employs a last-in first-out memory data structure for the basic processing of the operations, with data being pushed onto a stack and popped off as needed.

The EVM is made up of several key components that work together to execute smart contracts on the Ethereum blockchain and provide the previous features.

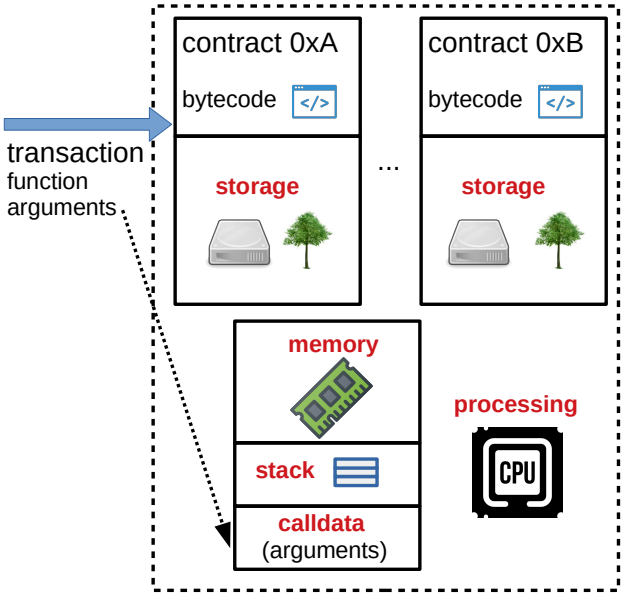

Figure 1: EVM Components Involved in the Processing of a Transaction.

As shown in Figure 1, the main components of the EVM involved in the processing of a transaction are the following:

- **Smart Contract Bytecode**: This is the low-level code that is executed by the EVM. This bytecode is a series of opcodes, or machine-level instructions. Each opcode in the EVM bytecode corresponds to a specific operation, such as arithmetic, conditional branching, or memory manipulation. The EVM executes the bytecode in a step-by-step fashion, with each opcode being processed in sequence. In general, smart contracts are written in a high-level programming language, such as Solidity, and then compiled into EVM bytecode.
- **Processing Environment**: The processing environment is responsible for executing smart contracts. It provides a runtime environment for the contract bytecode to execute in and manages the memory and storage that the contract uses.
- **Stack**: The EVM is a stack-based machine, which means that it uses a stack data structure to execute its operations.
- **Memory**: The EVM also has a memory component that allows smart contracts to store and retrieve data. Memory is organized as a linear array of bytes, and data is accessed by specifying the memory location.
- **Calldata**: The transaction that invokes a smart contract contains a set of parameters and values required for the smart contract to perform its function. These parameters and values are passed to the smart contract as calldata. Calldata is read-only, which means that the smart contract cannot modify it during execution. This is due to the

fact that the input data is part of the transaction that is stored on the blockchain, and any changes to the input data result in a different transaction hash and a different state of the blockchain.

- **Storage**: Smart contracts can also store data in the EVM's storage component. Storage is a persistent key-value store that is associated with each contract and can be used to store state information.

The EVM is a variant of the von Neumann architecture that uses a single shared memory for both data and instructions. The smart contract's bytecode is stored in memory in the EVM, and the program counter (PC) keeps track of the current instruction being executed. The stack is used to hold values that are needed for immediate use, such as function parameters, local variables, and return values. The stack is typically used for storing small values, such as integers and booleans while, the memory is used for storing large data structures, such as arrays and strings.

On the other hand, the EVM has its own instruction set or list of available opcodes, which is a set of low-level commands that are used to manipulate data in the stack, memory, and storage components. The instruction set includes operations such as arithmetic, bit manipulation, and control flow. In addition, to prevent spam and denial-of-service attacks, the EVM employs a gas system. Gas is a unit of measurement for the computational resources required to execute a smart contract, and each operation in the instruction set has its own gas cost.

1.3 EVM Stack

The EVM is a stack-based machine, which means that it uses a stack data structure to execute its operations. When an operation is performed, it uses the values that are currently on the top of the stack, and then pushes the result back onto the stack. Some of the main stack operations in the EVM are:

- `PUSH`: Pushes a value onto the stack. The opcode is followed by a byte indicating the number of bytes to be pushed onto the stack, and the actual bytes to be pushed. For example, the opcode "PUSH2 0x0123" pushes the value 0x0123 onto the stack.
- `POP`: Removes the top value from the stack and discards it.
- `DUP`: Duplicates the top value on the stack and pushes the duplicate onto the stack.
- `SWAP`: Swaps the top two values on the stack.
- `ADD`, `SUB`, `MUL`, `DIV`, `MOD`: These opcodes perform arithmetic operations on the top two values of the stack, and push the result back onto the stack.
- `AND`, `OR`, `XOR`, `NOT`: These opcodes perform bitwise logic operations on the top two values of the stack, and push the result back onto the stack.
- `EQ`, `LT`, `GT`: These opcodes perform comparison operations on the top two values of the stack, and push the result back onto the stack as a boolean.
- `SHA3`: Computes the SHA3 hash of the top value on the stack, and pushes the hash onto the stack.
- `JUMP`, `JUMPI`: These opcodes modify the program counter, allowing the program to jump to a different part of the code.

The EVM stack is limited to 1024 elements. If a contract attempts to push more elements onto the stack than this limit, a stack overflow error will occur, causing the transaction to fail.

1.4 EVM Memory

The memory in the EVM is used for storing large data structures, such as arrays and strings. This memory is a linear array of bytes that is used by smart contracts to store and retrieve data. The size of the memory is dynamically allocated at runtime, meaning that the amount of memory available to a smart contract can grow depending on its needs. The EVM memory is byte-addressable, which means that each byte in the memory can be individually addressed using a unique index. The size of the words in the EVM is 256 bits (32 bytes), which means that data is typically loaded and stored in 32-byte chunks. The EVM also provides instructions for loading and storing smaller chunks of data, such as individual bytes or 16-bit words. The EVM memory is non-persistent, which means that it is cleared whenever a smart contract execution completes. This means that if a contract wants to store data permanently, it must use the EVM storage component instead. It's also worth noting that the use of memory in the EVM is subject to gas costs. This is because accessing and modifying memory requires computational resources, which are paid for in the form of gas. The current memory limit for smart contracts on the Ethereum network is 2^{16} (or 65,536) pages. This means that the maximum amount of memory that a contract can use is 2 megabytes.

When a contract calls another contract, a new execution environment is created with its own memory space. The parent contract's memory space is saved, and the new contract's memory space is initialized. The new contract can then make use of its memory as needed. When the called contract's execution is complete, the memory space is released and the parent contract's saved memory is restored. It's worth noting that if a smart contract does not actually use the memory it has been allocated, that memory cannot be reclaimed or reused in the execution context of another contract. The opcodes related to memory are the following:

- `MLOAD`: This opcode is used to load a 32-byte word from memory into the stack. It takes a memory address as its input and pushes the value stored at that address onto the stack.
- `MSTORE`: This opcode is used to store a 32-byte word from the stack into memory. It takes a memory address and a value from the stack as its input, and stores the value at the specified address.
- `MSTORE8`: This opcode is similar to MSTORE, but it stores a single byte of data instead of a 32-byte word. It takes a memory address and a byte value from the stack as its input, and stores the byte at the specified address.
- `MSIZE`: This opcode returns the size of the current memory area in bytes.

1.5 EVM Storage

The EVM storage is a persistent key-value store that is associated with each smart contract. Storage is organized as a large array of 32-byte words and each word is identified by a unique 256-bit key, which is used to access and modify the value stored in that word. Because the storage is non-volatile, data stored in it will persist even after the contract execution is completed. Accessing and modifying storage is a relatively expensive operation in terms of gas costs. The EVM storage is implemented using a modified version of the Merkle Patricia Tree data structure, which allows for efficient access and modification of the storage data. A Patricia tree is a specific type of trie that is designed to be more space-efficient than a standard trie, by storing only the unique parts of the keys in the tree. Patricia trees are particularly useful in scenarios where keys share common prefixes, as they allow for more efficient use of memory and faster lookups compared to standard tries. The opcodes to manipulate the storage of a smart contract are the following:

- SLOAD: This opcode loads a 256-bit word from storage at a given index and pushes it onto the stack.
- SSTORE: This opcode stores a 256-bit word to storage at a given index. The value to be stored is popped from the stack, and the index is specified as the next value on the stack.

1.6 Transaction Processing

An Ethereum transaction is processed by first decoding it to obtain relevant fields such as the recipient address, the amount of Ether being transferred, and the data payload. To encode and decode transaction data, the RLP (Recursive Length Prefix) is used. Transactions are digitally signed using ECDSA (Elliptic Curve Digital Signature Algorithm). An important feature of ECDSA is that the signing public key can be recovered from the transaction signature without requiring the user to explicitly provide it in the transaction. The associated Ethereum account, which is identified by a 20-byte (160-bit) address, is then computed using the public key. The address is derived from the public key associated with the account by computing the Keccak-256 hash of the public key and taking the last 20 bytes of this hash.

When processing a transaction, the EVM begins by creating a context with an empty stack and memory space. The bytecode instructions are then executed, with values pushed and popped on the stack as needed. The EVM also uses a program counter to keep track of which instruction to execute next. Each opcode has a fixed number of bytes, so the program counter increments by the appropriate number of bytes after each instruction is executed. The stack elements have a size of 32 bytes each. This means that each value pushed onto the stack by an opcode, as well as each value popped off the stack by an opcode, is 32 bytes in size. The 32-byte size limit is a fundamental design choice in Ethereum and is based on the size of the EVM word. The EVM word is the basic unit of storage and processing in the EVM, and it is defined as a 256-bit (32-byte) unsigned integer. Since the EVM word is the smallest unit of data that can be processed by the EVM, the stack elements are also designed to be 32 bytes in size.

To summarize, the EVM sequentially executes the opcodes in the bytecode, following the program counter, and manipulates 32-byte values on the stack and in memory as needed to perform the desired computations and store the desired values to persist.

1.7 EVM interpreter

EVM interpreter is a software component that can process and execute Ethereum transactions.

Ethereum smart contracts can be written in various programming languages, such as Solidity, Vyper, Fe, or Yul, but are ultimately compiled into a sequence of EVM opcodes, expressed as bytecodes, that can be interpreted by the EVM interpreter. Ethereum opcodes are the low-level instruction set for EVM and represent the basic operations that EVM can perform during the execution of a smart contract triggered by a transaction.

The list of Ethereum op codes includes over 200 different operations , ranging from general arithmetic and logical operations to more advanced and blockchain eviroment specific operations like calls to other contracts, contract creation, and storage management. Some of the most commonly used op codes include:

- **ADD, SUB, MUL, DIV:** Basic arithmetic operations.
- **CALL, DELEGATECALL, CALLCODE:** Calling other contracts.
- **PUSH, POP:** Stack management operations.

- **JUMP, JUMPI:** Conditional jumps for making decisions.

- **SLOAD, SSTORE:** Storage management operations.

- **MLOAD, MSTORE:** Memory management operations.

1.8 zkASM and the ROM

The zero-knowledge Assembly (zkASM) is an assembly language designed by Polygon to describe computations that can be executed by a "special" virtual machine. This virtual machine has the ability to not only compute an output from a computation description and a set of inputs, but also generate a succinct cryptographic CI proof of a fixed length. This proof can be verified using a fixed and, more importantly, a low amount of energy and time. To achieve this "special" behavior, this virtual machine relies on Zero Knowledge technology.

Therefore, for any computation described in zkASM and executed with this "special" virtual machine its CI can be verified using fewer computational resources than were required for the original computation. The trick is that the proof verification algorithm can be implemented using a Ethereum smart contracts language and deployed on Ethereum L1, so for any computation expressed in zkASM its CI can be efficiently verified on Ethereum L1 using a fixed and low amount of gas, thereby inheriting Ethereum L1 security while avoiding network overhead.

In Ethereum L1, transactions are grouped into blocks. Each block contains an ordered sequence of transactions that are executed in a deterministic manner over Ethereum state, resulting in a new version of the Ethereum L1 state. Unlike Ethereum L1, in Polygon's zkEVM, the data structure that contains an ordered set of transactions that represents a state transition, is called a batch.

The ROM is a zkASAM program that is an EVM interpreter for Polygon zkEVM's batches architecture, all EVM opcodes are implemented on it aswell the batch interpretation and transaction execution logic in such a way that by the use of the ROM, our "special" virtual machine, given an Polygon's zkEVM L2 State and a batch of transactions, can execute those transactions, compute the resulting L2 State and generate a CI proof of the state transition. In order to verify the CI proofs, and provide data availability to the batches data, a sophisticated protocol has been designed and deployed on Ethereum L1 (refer to the technical documents regarding the zkEVM state management). The advantage of this design is that it enables the creation of a highly efficient Ethereum network (Polygon zkEVM) on top of Ethereum L1, inheriting its security, moreover most of the Ethereum ecosistem will be natively compatible with this new network. ROM stands for Read-Only Memory, due to its analogy with computer memories. Indeed the system can be viewed as a silicon processor capable of interpreting a set of instructions and a ROM memory containing a firmware (a piece of low-level software that is infrequently subjected to changes) written with that set of instructions which implements a special EVM interpreter for Polygon zkEVM L2 architecture. To operate, the processor executes the ROM's program which takes as inputs a batch of L2 transactions to be executed and the previous L2 State, and produces a new state and a CI proof as output. The system composed of the zk "special" virtual machine and the ROM is called the zkEVM batch prover.

Figure 1 shows a high level overview of the zkEVM batch prover.

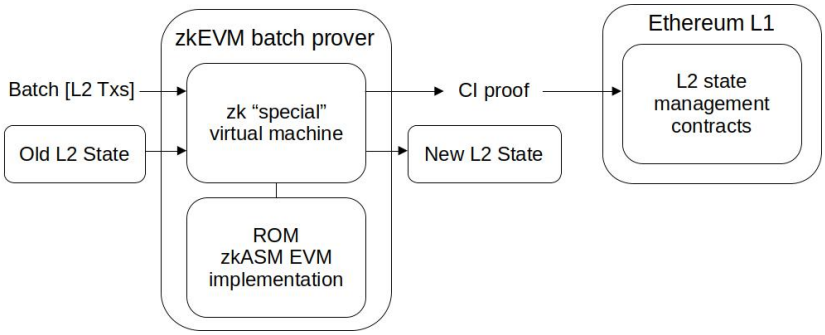

Figure 2: zkEVM batch prover structure.

In order to avoid futures misunderstood, it would be helpful to define and distinguish between the zkASM assembly instructions set and the EVM opcodes at this point.

- **zkASM instructions:** Set of instructions created and developed by Polygon to target a "special" zero-knowledge virtual machine that can execute computations with probable CI.
- **EVM opcodes:** Set of instructions designed to target the EVM, used to define smart contract's computations.

Although zkASM instructions and EVM opcodes are different types of instructions, the Polygon zkEVM ROM contains a piece of code written in zkASM instructions to implement each EVM opcode.

2 The ROM

As explained above, the ROM is a zkASM program that can be executed by a "special" virtual machine and allows to execute Polygon L2 State transitions with CI by inheriting Ethereum L1 security. This section aims to provide a detailed description of the ROM, allowing for a better understanding of its operation. The code of the ROM found at 0xPolygonHermez/zkevm-rom GitHub repository.

2.1 ROM executions public parameters

To achieve CI of a specific ROM execution, the resulting proof generated by the zkEVM batch prover must be successfully verified by L1 contracts. However, a zero-knowledge proof does not reveal any information about the specific computation being verified. Therefore, to allow the L1 smart to verify that a specific CI proofs corresponds to a specific state transition, a few "public" parameters of the computation are disclosed. The L1 verifier contract will verify the proof using this parameters and the verification will succeed only if the public parameters are those ones used to generate the proof by the prover. The disclosure of these parameters ensures that the proof being verified corresponds to a specific state transition, meaning that the execution of a specific batch over a specific state leads to a specific new state. L1 contracts provide data availability of the L2 batches, so that the prover is bound to on-chain data to fill the public parameters and generate a valid CI proof, and the verifier can access that data in a trustless manner during the proof verification.

The verification will succeed only if the public parameters match to those used to generate the CI proof by the prover. Each of the public parameters is listed and detailed below:

- **oldStateRoot:** L2 State Merkle Root of the L2 State before the state transition that wants to be proven. Ensures the integrity of the old L2 State on which transactions are executed.
- **oldAccInputHash:** Unique cryptographic identifier of the previous batch in the batch chain, the batch whose execution led to the L2 state before the state transition that wants to be proven, ensures the correct position of the state transaction in the batches sequence.
- **oldBatchNum:** Unique batch index of the previous batch, the batch whose execution led to the L2 state before the state transition that wants to be proven.
- **newStateRoot:** L2 State Merkle Root of the L2 State after the state transition that wants to be proven. Ensures the integrity of the old L2 State that results of the state transition.
- **newAccInputHash:** Unique cryptographic identifier of the batch whose execution is being proved, ensures the integrity of the batch.
- **newBatchNum:** Unique batch index of the the batch whose execution is being proved.
- **localExitRoot:** L2 Bridge contract's Exit Merkle Tree(EMT) at the end of the batch execution, ensures the integrity of the bridging transactions going out form the L2.
- **chainID:** Unique chain ID of Polygon zkEVM network, ensures that the computation can only be proven for a specific network.
- **forkID:** Unique identifier of the version of the ROM being used, ensures that the computation can only be proven for a specific version of the ROM code.

2.2 ROM main.zkasm

The main.zkasm is the zkASM code of the ROM where the batch processing and execution is described. The entry point of the ROM is represented by the **start** instruction.

ROM's main.zkasm code is divided in the following 6 sections:

- **A**: Load initial registers into memory.
- **B**: Set batch global variables.
- **C**: Loop parsing RLP transactions.
- **D**: Loop processing transactions.
- **E**: Batch asserts & computations:.
- **F**: Finalize execution.

2.2.1 A: Load initial registers into memory

The ROM code describes a general computation to process and execute a batch of L2 transactions, but the specific batch to process must be given as well as all the values that can vary between different ROM executions. We refer to these values as ROM's input variables.

In the first lines of the ROM, all those inputs are loaded into the memory to be used later. Note that each input has a proper memory variable to be stored.

```
STEP => A
0                                   :ASSERT ; Ensure it is the beginning of the execution

CTX                                 :MSTORE(forkID)
CTX - %FORK_ID                      :JMPNZ(failAssert)

B                                   :MSTORE(oldStateRoot)
C                                   :MSTORE(oldAccInputHash)
SP                                  :MSTORE(oldNumBatch)
GAS                                 :MSTORE(chainID)

${getGlobalExitRoot()}              :MSTORE(globalExitRoot)
${getSequencerAddr()}               :MSTORE(sequencerAddr)
${getTimestamp()}                   :MSTORE(timestamp)
${getTxsLen()}                      :MSTORE(batchL2DataLength) ; less than 300.000 bytes. Enforced by the smart
       contract

B => SR ;set initial state root

; Increase batch number
SP + 1                              :MSTORE(newNumBatch)
```

The first four lines ensure that this fragment of code is only executed at the beginning of the execution, i.e. they asserts that the **STEP** register is equal to 0, and that the version of the rom is correct, i.e ensures that the ROM's constant **FORK_ID** equals to the **frok_id** input variable. The following lines store the values of the input variables in memory variables. Note that most of them correspond to the execution's public parameters.

2.2.2 B: Set batch gobal variables

Batches are stored in Ethereum L1 following a specific data structure. The ROM uses the values in that data structure to identify the batch and ensure its integrity (refer to the technical documents regarding zkEVM state management to lean about batch data structure). The next section of the main.zkasm will load the batch data in the memory to be used later.

In the next three lines, the program will verify whether **globalExitRoot** is equal to 0. If it is, the execution flow will jump to the **skipSetGlobalExitRoot** line in the ROM. Note that the program first loads the value of **globalExitRoot** into register A and the value of 0 into register B before executing the combination of **EQ** and **JMPC(skipSetGlobalExitRoot)** instructions. The **EQ** instruction checks if A is equal to B, and if it is, the program flow will jump accordingly.

```
$${eventLog(onStartBatch, C)}

$ => A                                  :MLOAD(globalExitRoot)
0 => B
$                                       :EQ, JMPC(skipSetGlobalExitRoot)
```

The section of main.zkasm shown below stores the **GlobalExitRoot** value in the **globalExitRootMap** of the **PolygonZkEVMGlobalExitRootL2.sol** contract instance, which allows users to claim bridged assets in L2 (refer to the technical documents regarding the zkEVM bridge). 0 is not a valid value for a **GlobalExitRoot**, therefore in this case, the previously explained jump will be executed and this part will be skipped. **globalExitRootMap** mapping entries has **GlobalExitRoot** as keys and the batch's timestamp as values.

```
1   setGlobalExitRoot:
2   0 => HASHPOS
3   $ => E                                  :MLOAD(lastHashKIdUsed)
4   E+1 => E                                :MSTORE(lastHashKIdUsed)
5
6   32 => D
7   A                                       :HASHK(E)
8   %GLOBAL_EXIT_ROOT_STORAGE_POS           :HASHK(E) ; Storage position of the global exit root map
9   HASHPOS                                 :HASHKLEN(E)
10  $ => C                                  :HASHKDIGEST(E)
11
12  %ADDRESS_GLOBAL_EXIT_ROOT_MANAGER_L2 => A
13  %SMT_KEY_SC_STORAGE => B
14
15  ; read timestamp given the globalExitRoot
16  ; skip overwrite timestamp if it is different than 0
17  ; Since timestamp is enforced by the smart contract it is safe to compare only 32 bits in 'op0' with JMPNZ
18  $ => D                                  :SLOAD, JMPNZ(skipSetGlobalExitRoot)
19
20  $ => D                                  :MLOAD(timestamp)
21  $ => SR                                 :SSTORE ; Store 'timestamp' in storage position 'keccak256(globalExitRoot, 0)'
```

Let's analyze the lines above in depth. Specific Solidity mapping values are stored in a specific contract storage slot computed as Keccak("key", "mapping slot"). Therefore, in order to compute the storage slot where the timestamp of a specific **globalExitRoot** will be stored, it is mandatory to perform a Keccak hash operation.

The first four lines are a consequence of how the hashes are performed in zkASM. First, the **HASPOS** register is set to zero in order to set the pointer of the hashing input array to its 0 position. Then, the **lastHashKIdUsed** memory variable is loaded into register E and incremented by one. **lastHashKIdUsed** contains the index of the last hash operation performed, therefore our hash operation will be next to it and the new value of register **E** will be used as its index.

The goal of line 6 is to set the length in bytes of the next entry in the hash input array that will be taken, by design from the register **D**. In this case, since both Keccak arguments are 32 bytes in length, the number 32 is set. In the following two lines, the two Keccak arguments are loaded into the hash inputs array: first, the **GlobalExitRoot** that was already in the **A** register, and then the slot address of the mapping in the contract's storage. Then, in line 9, the computation of the hash is triggered by giving **HASHPOS** as the length of the hash input array. Note that **HASHPOS** will be 64 since it auto-increments its value each time a byte is pushed into the hash input array via **HASHK** instruction, and we have pushed 32 bytes of **GlobalExitRoot** and 32 bytes of the slot address. In the following line the hash digest will be loaded to register C.

Next, in lines 12 and 13, the address of the **PolygonZkEVMGlobalExitRootL2.sol** contract instance and the type of Polygon zkEVM's state tree leaf that will contain the mapping value (3-Contract storage slot value) will be loaded into registers A and B, respectively. Note that at this moment, zkASM storage operations can be performed on the zkEVM state tree leaf that holds the 32-byte storage slot that corresponds to the value of the specific **GlobalExitRoot** in the mapping **globalExitRootMap** of the **PolygonZkEVMGlobalExitRootL2.sol** contract instance.

Line 18 will check if the storage slot is already set, i.e., if it is different from zero, and will skip lines 20 and 21 in that case to avoid overwriting a **GlobalExitRoot** that has already been set. Lines 20 and 21 will load the timestamp value from the memory variable to register D and then store it with the **SSTORE** instruction. Note that the **SR** is also updated with the latest Polygon zkEVM's state root value in the line 21.

The following 8 lines will save the previously computed state tree root to the memory variable batchSR. Then, they will load the number index of the last transaction executed in the L2 from the leaf that contains it in the state tree, and store it in the **txCount** memory variable.

```
skipSetGlobalExitRoot:
SR                                  :MSTORE(batchSR)
; Load current tx count
%LAST_TX_STORAGE_POS => C
%ADDRESS_SYSTEM => A
%SMT_KEY_SC_STORAGE => B
$ => D              :SLOAD
D                   :MSTORE(txCount)
```

TODO: EXPLAIN WHY MUST BE CHECKED THE KECCAK COUNTERS.

```

; Compute necessary keccak counters to finish batch
$ => A              :MLOAD(batchL2DataLength)
; Divide the total data length + 1 by 136 to obtain the keccak counter increment.
; 136 is the value used by the prover to increment keccak counters
A + 1                               :MSTORE(arithA)
136                                 :MSTORE(arithB), CALL(divARITH); in: [arithA, arithB] out: [arithRes1: arithA/
      arithB, arithRes2: arithA%arithB]
$ => B                              :MLOAD(arithRes1)
; Compute minimum necessary keccaks to finish the batch
B + 1 + %MIN_CNT_KECCAK_BATCH => B  :MSTORE(cntKeccakPreProcess)
%MAX_CNT_KECCAK_F - CNT_KECCAK_F - B    :JMPN(handleOOCKatRLP)
```

2.2.3 C: Loop parsing RLP transactions

In a batch, the transactions are represented as a byte array where each transaction is encoded using the Ethereum pre-EIP-115 or EIP-115 formats and following the RLP (Recursive-length prefix) standard. The encoded transaction is concatenated with the values v, r, and s of the signature. The section of main.zkasm that follows iterates through each transaction in the batch. For each transaction, a new zkasm memory context is created and the transaction data is parsed to extract the transaction values which are stored in memory variables for later use. Also each transaction data will be pushed to an specific keccak operation given by batchHashDataId input buffer. The hash digest of all transaction data is used to calculate the accumulated hash of the batch in question.

The transactions must have one of the following structure:

- EIP-155: rlp(nonce, gasprice, g asLimit, to, value, data, chainid, 0, 0,)vrs.
- pre-EIP-155: rlp(nonce, gasprice, gasLimit, to, value, data)vrs.

```
E+1 => E                            :MSTORE(lastHashKIdUsed)
0                                   :MSTORE(batchHashPos)
E                                   :MSTORE(batchHashDataId)
$ => A                              :MLOAD(lastCtxUsed)
A                                   :MSTORE(ctxTxToUse) ; Points at first context to be used when processing
      transactions

$${var p = 0}

txLoopRLP:
$ => A          :MLOAD(lastCtxUsed)
A+1 => CTX      :MSTORE(lastCtxUsed)

$ => A          :MLOAD(batchL2DataLength)
$ => C          :MLOAD(batchL2DataParsed)
C - A           :JMPN(loadTx_rlp, endCheckRLP)
endCheckRLP:
:JMP(txLoop)
```

The first three lines will prepare the Keccak hash instance to compute the hash digest of all transaction data in the batch. It takes and stores in the memory a **batchHashDataId** based on the last hash ID used. Also, **batchHashPos** will be set to zero since it will be the pointer of the Keccak input that will be incremented with each transaction addition to the hash input.

Then the variable **ctxTxToUse** will be set to the last context used value to create new contexts ahead of older ones for each transaction in the batch.

The lines 9 to 15 are executed in a loop for each transaction in the batch. For each transaction, first, a new memory context will be created. The variable **lastCtxUsed** will act as the loop index and will also give a specific context number to each transaction. Note that in each iteration, it will be incremented by 1. Then lines 13 to 15 will check if is the last iteration of the loop by comparing length of parsed batch with non parsed batch. In that case, the loop will break. If not, the **loadTx_rlp** code will be executed, it can be found at the ROM's GitHub repository. It contains the logic to parse a transaction of the batch and store each transaction data value in a specific memory variable of the transaction's memory context. It also pushes the data to the hash of all transactions input.

2.2.4 D: Loop processing transactions

The section of main.zkasm that follows iterates again through all transactions in the batch, executing each one of them and applying the changes to the Polygon zkEVM state tree.

```
1   txLoop:
2   $ => A          :MLOAD(pendingTxs)
3   A-1 => A        :MSTORE(pendingTxs), JMPN(processTxsEnd)
4
5   $ => A          :MLOAD(ctxTxToUse) ; Load first context used by transaction
6   A+1 => CTX      :MSTORE(ctxTxToUse),JMP(processTx)
7
8   processTxEnd:
9   :CALL(updateSystemData)
10  processTxFinished:
11  $${eventLog(onFinishTx)}   :JMP(txLoop)
12
13  processTxsEnd:
```

The lines 1 to 11 are executed in loop for each transaction. the variable **pendingTxs** will be the loop index. Note that now in each iteration it decrements by 1, starting from the last value set in the **pendingTxs** variable (for each transaction parsed by the former executed **loadTx_rlp** code, the **pendingTxs** variable is incremented by one). Line 3 will check if all transactions in the batch are already processed by checking if the **pendingTxs** variable is less than 0. In that case, the loop will break. If not, the **process-tx** code will be executed, it can be found at the ROM's GitHub repository. It contains the logic to process an Ethereum transaction adapted to Polygon zkEVM infrastructure. All the verifications that would be done in the Ethereum network, such as transaction signature verification, chain ID, etc., are also implemented in the **process-tx** code. Note that thanks to the former ROM's section (C), all transaction data can now be accessed through zkASM memory opcodes without the need to parse the bytes string of the batch's transactions again.

In each iteration of the loop, after processing a specific transaction, the subroutine **updateSystemData** is called (Line 9). Its code can be found on the ROM's GitHub repository in the utils.zkasm file. The system contract is a special contract in Polygon zkEVM L2 whose storage contains information about the network. The **updateSystemData** subroutine is meant to update the total processed transaction counter and the state root mapping in the system contract.

2.2.5 E: Batch computations

The section of main.zkasm that follows performs the last computations that have to be done for each batch.

First, it reads the **LocalExitRoot** of the **PolygonZkEVMGlobalExitRootL2.sol** contract instance and stores it in **newLocalExitRoot**, which is a variable meant to contain the computation public parameter **LocalExitRoot**. Since **LocalExitRoot** is a public parameter, once the CI proof will be successfully verified by L1 contracts, this value will be sent to the **PolygonZkEVMGlobalExitRoot.sol** L1 contract instance to

trigger the computation of the new bridge's Global exit root and enable to claim bridge transactions in L1.

```
;; Get local exit root
; Read 'localExitRoot' variable from GLOBAL_EXIT_ROOT_MANAGER_L2 and store
; it to the 'newLocalExitRoot' input
%ADDRESS_GLOBAL_EXIT_ROOT_MANAGER_L2  => A
%SMT_KEY_SC_STORAGE => B
%LOCAL_EXIT_ROOT_STORAGE_POS => C
$ => A                                          :SLOAD
A                                               :MSTORE(newLocalExitRoot)
```

The next segment will ensure that the length of the input of the Keccak hash of all transaction data matches with the length of the byte array of the transactions given as computation input, i.e., all the transactions of given as computation input are included in the hash computation. Then computes the keccak hash and stores the hash digest in the **batchHashData** memory variable. This value will ensure the integrity of the batch's transactions data queried from L1.

```
;; Transactions size verification
; Ensure bytes added to compute the 'batchHashData' matches the number of bytes loaded from input
$ => A                          :MLOAD(batchHashPos)
$                               :MLOAD(batchL2DataLength), ASSERT

;; Compute 'batchHashData'
A => HASHPOS
$ => E                          :MLOAD(batchHashDataId)

HASHPOS                         :HASHKLEN(E)
$ => A                          :HASHKDIGEST(E)

A                               :MSTORE(batchHashData)
```

The last segment will compute the accumulated hash of the batch and store it in the **newAccInputHash** variable, which is meant to contain the computation public parameter **newAccInputHash**. The accumulated hash will ensure the integrity of the batch data (transactions, timestamp, globalExitRoot) and that of all its predecessors, as well as the order in which they have been sequenced. The L1 verification contract will check that this value equals that one in its storage to ensure that the exact data queried from L1 has been used to perform the off-chain computations of the batch.

```
;; Compute 'newAccInputHash'
0 => HASHPOS

32 => D
$ => A                          :MLOAD(oldAccInputHash)
A                               :HASHK(0)

$ => A                          :MLOAD(batchHashData)
A                               :HASHK(0)

$ => A                          :MLOAD(globalExitRoot)
A                               :HASHK(0)

8 => D
$ => A                          :MLOAD(timestamp)
A                               :HASHK(0)

20 => D
$ => A                          :MLOAD(sequencerAddr)
A                               :HASHK(0)

HASHPOS                         :HASHKLEN(0)

$ => C                          :HASHKDIGEST(0)
C                               :MSTORE(newAccInputHash)
$${eventLog(onFinishBatch)}
```

2.2.6 F: Finalize execution

The final section of main.zkasm will perform the final steps of ROM's execution, which is to load the values of computation results in the proper registers and set the initial

values of some required registers. Then it will jump to the final wait. All of these steps are required to be in the code due to the system's design.

```
    ; Set output registers
    $ => D                          :MLOAD(newAccInputHash)
    $ => E                          :MLOAD(newLocalExitRoot)
    $ => PC                         :MLOAD(newNumBatch)

    ; Set registers to its initials values
    $ => CTX                        :MLOAD(forkID)
    $ => B                          :MLOAD(oldStateRoot)
    $ => C                          :MLOAD(oldAccInputHash)
    $ => SP                         :MLOAD(oldNumBatch)
    $ => GAS                        :MLOAD(chainID)
    finalizeExecution:
    :JMP(finalWait)
```

3 zkEVM Architecture

3.1 Registries

In order to replicate the EVM opcodes, zkEVM introduces six state related generic registers named `A`, `B`, `C`, `D` and `E`. However, since zkEVM operates over a finite field of almost 64 bits, each register is split into 8 limbs of 32 bits each:

$$\begin{aligned}
&\mathtt{A}_0, \ldots, \mathtt{A}_7 \\
&\mathtt{B}_0, \ldots, \mathtt{B}_7 \\
&\mathtt{C}_0, \ldots, \mathtt{C}_7 \\
&\mathtt{D}_0, \ldots, \mathtt{D}_7 \\
&\mathtt{E}_0, \ldots, \mathtt{E}_7
\end{aligned}$$

with $\mathtt{A}_i, \mathtt{B}_i, \mathtt{C}_i, \mathtt{D}_i, \mathtt{E}_i \in \{0, \ldots, 2^{32} - 1\}$. When storing a value in a register, the least significant bits are placed in the lowest limb, starting from the 0-th one. That is, if we want to allocate a 32 bits value in the `A`-th register, we will fill only $\mathtt{A}_0$ with it. It we want to allocate a 64 bits value, we will fill $\mathtt{A}_0$ and $\mathtt{A}_1$ and so on. For example, if one want to store the value `0x12345678` in the `A` register, the least significant byte `0x78` would be placed in $\mathtt{A}_0$ and the most significant byte `0x12` in $\mathtt{A}_3$.

Apart from the generic registers related with the state of the zkEVM, there are additional registers in zkEVM that are used for various purposes. Here is a brief description of each of these registers:

- `SR`: This is the status register and is used to indicate the current status of the processor. For example, it might indicate whether an arithmetic operation resulted in a carry or overflow, or whether an instruction encountered an error.
- `CTX`: This is the context register and is used to store the context of the current execution environment. For example, it might store information about the current smart contract being executed or the current transaction.
- `SP`: This is the stack pointer register and is used to point to the top of the stack. Every time a number is pushed onto the stack or removed from it, it is either increased or decreased.
- `PC`: This is the EVM program counter register. The Program Counter (`PC`) encodes which instruction, stored in the code, should be next read by the EVM. The program counter is usually incremented by one byte, to point to the following instruction,

with some exceptions. For instance, the PUSHx instruction forces the PC to skip their parameter because it is longer than a single byte. The JUMP instruction modifies the program counter to a location determined by the top of the stack rather than increasing the PC's value.

- GAS: This is the gas register and is used to store the amount of gas remaining for the current transaction. Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network.
- RR: This is the return register and is used to store the address to return to after a function call or jump.
- zkPC: This is the zkEVM program counter register. Similarly as explained in the PC register, it encodes which instruction of the zkEVM is being executed. This register will be crucial in order to ensure that the program that is being executed matches the program that wants to be proved.
- STEP: This is the step register and is used to store the number of instructions executed so far in the current transaction.
- MAXMEM: This is the maximum memory register and is used to store the maximum amount of memory that can be allocated for the current transaction.
- HASHPOS: This is the hash position register and will contain the index of the next position of the bytes array of the input of the hash that we will start to fill. More information on that can be found in the hash-related instructions section.
- ROTL_C: This is the C-rotate left (read-only) register and is used to flag a left rotation of the C register by 4 bytes. This has the effect of moving the 4 most significant bytes of the C register to the 4 least significant bytes, and moving the 28 least significant bytes to the 28 most significant bytes. Later on, this rotated C register can be assigned elsewhere. For example:

```
1       ROT_C + 2 => A
```

  increases by 2 units the rotated value of C and assigns it into the register A.
- RCX: This is the repeat count register RCX, which is used in the repeat instruction. The repeat instruction allows for a certain instruction to be executed multiple times, based on the value stored in the RCX register. The RCX register is decremented by one each time the instruction is executed, until it reaches zero. The use of the RCX register in the repeat instruction allows for efficient execution of repetitive tasks, as it avoids the need for explicit loops in the code.

Each of these registers serves a specific purpose in the operation of zkEVM and is used by the various instructions and operations defined in the zkEVM specification.

3.2 Binary

The zkEVM contains a specific state machine in order to perform several 256-bits operations. Each one of them are programmed in order to operate between the registers A and B. The implemented operations are the following:

- ADD (+). This operation adds two 256-bit strings. Invoked using :ADD instruction.
- SUB (−). This operation subtracts two 256-bit strings. Invoked using :SUB instruction.

- `LT` ($<$). This operation checks if a 256-bit string is smaller than another 256-bit string considering a codification of the binary strings without sign. Invoked using `:LT` instruction.
- `SLT` ($<$). This operation checks if a 256-bit string is smaller than another 256-bit string but considering a codification of the binary strings with sign (the codification used by the EVM is complement to two). Invoked using `:SLT` instruction.
- `EQ` ($=$). This operation checks if two 256-bit strings are equal. Invoked using `:EQ` instruction.
- `AND` ($\wedge$). This operation computes the bit-wise "add" of the two strings. Invoked using `:AND` instruction.
- `OR` ($\vee$). This operation computes the bit-wise "or" of the two strings. Invoked using `:OR` instruction.
- `XOR` ($\oplus$). This operation computes the bit-wise "xor" of the two strings. Invoked using `:XOR` instruction.

To understand how the `ADD`, `SUB`, `LT` and `SLT` operations work, we need to understand in first place how the EVM codes 256-bit strings to signed and unsigned integers. Figure 3 shows these codifications for 3-bit strings but the idea can be easily extended to 256-bit strings.

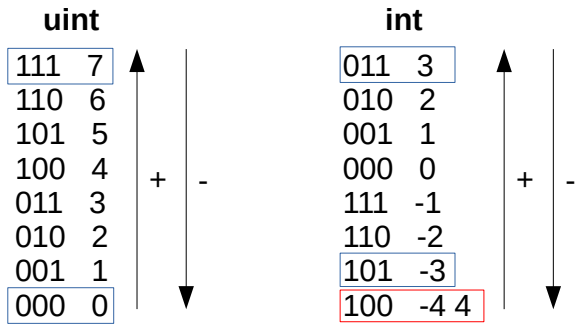

Figure 3: Codifications of 3-bit strings for signed and unsigned integers as used by the EVM.

Adding two strings is performed bit by bit using the corresponding carry. For example, let's add the 3-bit strings `0b001` and `0b101`:

- We start with an initial `carry` $= 0$ and adding the least significant bits:
  $1 + 1 +$ `carry` $= 1 + 1 + 0 = 0$ with the next carry being equal to 1.
- Then, we add the next bits using the previous carry:
  $0 + 0 +$ `carry` $= 0 + 0 + 1 = 1$ with the next carry being equal to 0.
- Finally, we add the most significant bits:
  $0 + 1 +$ `carry` $= 0 + 1 + 0 = 1$ with the final carry being equal to 0.
- As a result: `0b001` $+$ `0b101` $=$ `0b110` with `carry` $= 0$.

The sum `0b001` $+$ `0b101` $=$ `0b110`, for unsigned integers is $1 + 5 = 6$, while for signed integers encoded with complement to two this sum is $1 + (-3) = (-2)$. In other words, we can do the same binary sum for both signed integers and for unsigned integers.

The operations `LT` and `SLT` are different however. When comparing unsigned integers (LT), the natural order for comparisons is applied, e.g. 110 (6) > 010 (2). When comparing signed integers (`SLT`), we must take into account the most significant bit that acts as the sign. If the most significant bit of the two strings being compared is the same, the the natural order applies, e.g. 110 (-2) > 101 (-3). However, if the strings being compared have a different most significant bit, then the order must be flipped (bigger numbers start with 0), e.g. 001 (1) > 110 (-2). Finally, notice that with unsigned integers, there is a caveat since 4 and -4 have the same codification.

On the other hand, the `AND`, `OR` and `XOR` operations are bit-wise operations, that is to say, the operation is done bit by bit. As a result, there are not any carries to be considered when operating a pair of bits. As we will see, this is going to make the checks easier to implement for bit-wise operations. Table 4 depicts the truth tables of `AND`, `OR` and `XOR` operators, respectively.

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4: Truth Tables of bit-wise operations

Notice that we do not consider the **NOT** operation. This is because the **NOT** operation can be easily implemented with the `XOR` operation just by taking the 256-bit string and doing an `XOR` with `0xff...ff`.

3.3 Arithmetic

The Arithmetic State Machine (SM) is one of the six secondary state machines receiving instructions from the **Main SM** Executor. The main purpose of the Arithmetic State Machine is carry out elliptic curve arithmetic operations, such as Point Addition and Point Doubling as well as 256-bits modular operations. The selected curve E is the one with equation $y^2 = x^3 + 7$ over the field $\mathbb{F}_p$ with:

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1.$$

More specifically, thee Arithmetic SM is responsible for the execution of the following operations:

- **Field Arithmetic**: Here, y_2 and y_3 are the result of performing field arithmetic over x_1, y_1 and x_2. That is:

$$x_1 \cdot y_1 + x_2 = y_2 \cdot 2^{256} + y_3. \tag{1}$$

  Note that if y_1 is set to 1 then Eq. (1) represents field addition, and similarly if x_2 is set to 0 then Eq. (1) represents field multiplication.

- **Elliptic Curve Addition**: Given two points $P = (x_1, y_1), Q = (x_2, y_2)$ from E with $x_1 \neq x_2$, the point $P + Q = (x_3, y_3)$ is computed as follows:

$$\begin{aligned} x_3 &= s^2 - x_1 - x_2, \\ y_3 &= s(x_1 - x_3) - y_1. \end{aligned}$$

where:

$$s = \frac{y_2 - y_1}{x_2 - x_1}$$

- **Elliptic Curve Doubling**: Given a point $P = (x_1, y_1)$ from E such that $P \neq \mathcal{O}$, the point $P + P = 2P = (x_3, y_3)$ is computed as follows:

$$\begin{aligned} x_3 &= s^2 - 2x_1, \\ y_3 &= s(x_1 - x_3) - y_1. \end{aligned}$$

where:

$$s = \frac{3x_1^2}{2y_1}.$$

Motivated by the implemented operations, the Arithmetic SM is composed of 6 registers $x_1, y_1, x_2, y_2, x_3, y_3$. Each of these registers is decomposed in 16 sub-registers of 16-bit (2 byte) capacity, making a total of 256 bits per register. We also need to provide s and q_0, q_1, q_2, which are also elements of (the finite field) 256 bits.

The Arithmetic State Machine, combined with the Binary State Machine is used to implement opcodes like **signed and unsigned integer division**, **signed and unsigned module reducing**, **modular operations** or **exponentiation**.

3.4 Keccak-Related State Machines

The KECCAK State Machine is in charge of validating the correct computation of EVM's KECCAK-256 hash. Unlike POSEIDON, due to its bit-wise nature, it is very inneficient in our set up and, therefore, it has been the focus for performing several optimizations. The strategy taken it is to use its circuit-like construction together with a PLONK-ish design in order to perform several KECCAK-f permutations at the same time. This State Machine is a complex one, because it has to deal with the Sponge Construction byte-wise and, later on, translate this execution bit-wise in order to compute all the KECCAK-f permutations in a parallel way.

Sponge Construction

The **sponge construction** is a simple iterated construction for building a function

$$F : \mathbb{Z}_2^* \to \mathbb{Z}_2^l$$

with variable-length input and arbitrary output length based on a fixed-length permutation

$$f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$$

operating on a fixed number b of bits. Here b is called the **width**. The array of b bits that f keeps transforming is called the **state**. The state array is split in two chunks of r and c bits respectively. We call r the **bitrate** (or rate) and c the **capacity**. We will understand later on the motivation for this splitting.

Let us describe how the sponge construction works:

1. First of all, the input string is padded with a reversible **padding rule**, in order to achieve a length divisible by r. Subsequently, it is cut into blocks of r bits. We also initialize the b bits of the state to zero.

2. (*Absorbing Phase*) In this phase, the r-bit input blocks are XORed into the first r bits of the state, interleaved with applications of the function f. We proceed until processing all blocks of r-bits. Observe that the last c bits corresponding to the capacity value does not absorb any input from the outside.

3. (*Squeezing Phase*) In this phase, the first r bits of the state are returned as output blocks, interleaved with applications of the function f. The number of output blocks is chosen at will by the user. Observe that the last c bits corresponding to the capacity value are never output during this phase. Actually, if the output exceeds the specified length, we will just truncate it in order to fit.

We depict an schema of the sponge construction in Figure 5.

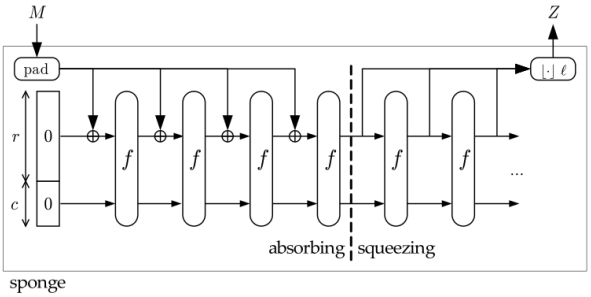

Figure 5: Schema of Sponge Construction.

The elements that completely describe a single instance of a sponge construction are: the fixed-length permutation f, the padding rule **pad** and the rate value r.

EVM Hash Function Specification

The EVM makes use of KECCAK-256 hash function, which is constructed using KECCAK[512] sponge construction. Let us, therefore, define the KECCAK[c] sponge construction. This sponge operates with a width of 1600 bits and a rate of $1600-c$. In the case of KECCAK[512], the rate chunk is composed of 1088 bits (or equivalently, 136 bytes) and the capacity chunk has 512 bits (or equivalently, 64 bytes). The permutation used in KECCAK[c] is KECCAK-p[1600, 24] (See [**?**]). The last ingredient we need to define in order to completely specify the hash function is the padding rule. In KECCAK[c], the padding pad10*1 is used. If we define $j = (-m-2) \mod r$, where m is the length of the input in bits, then the padding we have to append to the original input message is

$$P = 1 \,||\, 0^j \,||\, 1.$$

Thus, given an input bit string M and a output length d, KECCAK[c](M, d) outputs a d bit string following the previous sponge construction description.

It should be noted that this construction does **not** follow the FIPS-202 based standard (a.k.a SHA-3). According to [**?**], NIST changed the SHA3 padding to

$$\text{SHA3-256}(M) = \text{KECCAK[512]}(M \,||\, 01, 256).$$

The difference is the additional 01 bits appended to the original message, which were not present in the orignal KECCAK specification.

zkEVM Keccak-related State Machines Pipeline

Unlike the other state machines previously explained, rather than implementing the Keccak-256 hash function as a single state machine, the zkEVM does so in a framework of four state machines. They are as follows:

Figure 6: Pipeline for Keccak-related State Machines.

- The **Padding KK** SM is used for padding purposes, as well as validation of hash-related computations pertaining to the **Main SM**'s queries. As depicted in the above figure, the **Padding KK SM** is **Main SM**'s gateway to the Keccak hashing state machines.
- The **Padding KK Bit** SM converts between two string formats, the bytes of the **Padding KK SM** to the bits of the **Keccak-f SM**, and vice-versa.
- The **Bits2Field SM** is used specifically for parallelizing **Keccak-f SM** implementation. It acts as a multiplexer between the **Padding KK Bit SM** and the **Keccak-f SM**. This state machine is called **Bits2Field** because it initially ensured the correct packing of bits from 4444 different blocks of the **Padding KK Bit SM** into a single field element.
- The **Keccak-F SM** computes string hashes at the request of the **Main SM**. Although the **Keccak-f SM** is a binary circuit, it operates on a 44bits-by-44bits basis rather than a bit-by-bit basis. This equates to running four 4444 hashing circuits in parallel.

3.5 Poseidon-Related State Machines

The **Poseidon State Machine** is a secondary state machine that receives instructions from the **Main State Machine** of the zkProver. It uses the Poseidon hash function to generate hash values in response to requests from the **Storage SM** (which we will see later on) and instructions from the **Main SM** Executor. Poseidon Actions are the directives that the **Poseidon SM** receives from one of the two SMs. It performs the Poseidon Actions as a secondary SM and also verifies that the output hash values were accurately calculated.

Poseidon (See [**?**]) is a hash function designed to minimize prover and verifier complexities when zero-knowledge proofs are generated and validated. The previously defined KECCAK-256 cryptographic hash require large circuits as they are not tailored to finite fields used in ZK proof systems (actually, KECCAK-256 works well in binary fields, and we will see later on that this fact introduces a lot of complexity in the constrain design). For this reason, zkEVM uses Poseidon hash as the main internal hash function.

More concretely, we will now specify the specific instance of Poseidon that zkEVM uses. We will work over the field $\mathbb{F}_p$ where $p = 2^{64} - 2^{32} + 1$. The state width of the Poseidon permutation is of 8 field elements (observe that we are changing the paradigm, working with whole field elements instead of working bit-wise) meanwhile we will work with a capacity of 4-field elements.

The permutation used in Poseidon is a round function. A typical round function consists of three operations; an addition of a round-key $ARC(\cdot)$, a non-linear function S (i.e., a substitution box or S-box), and a linear function L which is often an affine transformation (in particular, an MDS matrix M). Some rounds are partial rounds because they use only one S-box instead of the full number of S-boxes, one for each element of the state. For security purposes against certain cryptanalytic attacks, outer rounds are full rounds, while the inner rounds are partial rounds.

Denote the number of rounds by $R = R_F + R_P$ where R_F is the number of full rounds and R_P is the number of partial rounds. Also, let $M(\cdot)$ denote the linear diffusion layer. Then, the figure blow depicts the Poseidon's permutation.

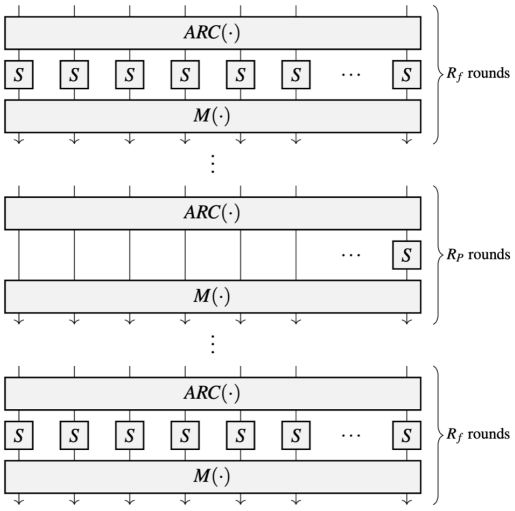

Figure 7: HADES-based Poseidon's permutation.

The Poseidon S-box layer that we will use is the 7-power S-Box, i.e.

$$SB(x) = x^7,$$

The Poseidon instance also requires to specify the number of full and partial rounds of the permutation. In our case, we will use

$$R_F = 8 \text{ (number of full rounds)} , \quad R_P = 22 \text{ (number of partial rounds)}$$

Only one squeezing iteration will be effectuated, with an output of the first 4 field elements of the state (which consists of approximately 256-bits, but no more than that). The Round Constants and the MDS matrix are completely specified using the previous parameters.

3.6 Memory-Related State Machines

Memory State Machine

The memory of the EVM (Ethereum Virtual Machine) is a volatile read-write memory that is used to store temporary data during the execution of transactions of smart contract functions. That is, data in memory is populated during transaction's execution but it does not persist between transactions. The memory is an array of 256-bit (32 bytes) words that can be accessed through *addresses at byte level*, that is to say, each byte in the memory has a different address. Memory has addresses of 32 bits and initially, all memory locations are composed by bytes set to zero. Now, let's see the layout in memory of the following two words 0xc417...81a7 and 0x88d1...b723. Table 8 shows this layout.

Observe that each word has 32 bytes and that the words are stored in Big-Endian form, i.e. the most significant bytes are set in the lower addresses. The EVM provides three opcodes to interact with the memory area. We have an opcode to read and an opcode to write 32-byte words providing an offset:

- `MLOAD`: It receives an offset and returns the 32 bytes in memory starting at that offset.
- `MSTORE`: It receives an offset and saves 32 bytes from the offset address of the memory.

ADDRESS	BYTE
0	0xc4
1	0x17
⋮	⋮
30	0x81
31	0xa7
32	0x88
33	0xd1
⋮	⋮
62	0xb7
63	0x23

Figure 8: Layout in memory of `0xc417...81a7` and `0x88d1...b723`.

Considering our previous memory contents, if we perform an `MLOAD` with an offset of `1`, we would obtain the following word: `0x17...a788`. On the other hand, if we do an `MSTORE` with an offset of `1` with the word `0x74f0...ce92`, we would modify the content of the memory as shown in Table 9.

ADDRESS	BYTE
0	0xc4
1	**0x74**
2	**0xf0**
...	...
31	**0xce**
32	**0x92**
33	0xd1
...	...
62	0xb7
63	0x23

Figure 9: Layout in memory after the introduction of `0x74f0...ce92`.

When the offset is not a multiple of 32 (or `0x20`), as in the previous example, we have to use bytes from two different words when doing `MLOAD` or `MSTORE`.

Finally, the EVM provides a write memory operation that just writes a byte:

- `MSTORE8`: It receives an offset and saves one byte on that address of the memory.

Notice that `MSTORE8` always uses only one word.

The **Memory SM** is in charge of proving the memory operations in the execution trace. As mentioned, read and write operations use addresses at byte level in the EVM. However, doing the proofs byte by byte would consume many values in the trace of this state machine.

Instead, in this machine, we operate addressing words (32 bytes). For example, if we have the memory layout from Table 8, then we would have the memory layout of Table 10 with addresses that point to 32-byte words.

The **Memory SM** uses this latter layout, the 32-byte word access, to check reads and writes. However, as previously mentioned, the EVM can read and write with offsets at a byte level. As a result, we will need to check the relationship between byte access and

ADDRESS	32-BYTE WORD
0	0xc417...81a7
1	0x88d1...b723

Table 10: Layout in the memory state machine.

32-byte word access. For these checks, we have another state machine called **Memory Align SM** that is discussed below.

Memory Align State Machine

The **Memory SM** checks memory reads and writes using a 32-byte word access, while the EVM can read and write 32-byte words with offsets at a byte level. Table 11 shows an example of possible byte-addressed and 32-byte-addressed memory layouts for the same content (three words).

ADDRESS	BYTE
0x00	0xc4
0x01	0x17
0x02	0x4f
...	...
0x1e	0x81
0x1f	0xa7
0x20	0x88
0x21	0xd1
0x22	0x1f
...	...
0x3e	0xb7
0x3f	0x23
0x40	0x6e
0x41	0x21
0x42	0xff
...	...
0x5e	0x54
0x5f	0xf9

ADDRESS	32 – BYTE WORD
0x00	0xc4174f...81a7
0x01	0x88d11f...b723
0x02	0x6e21ff...54f9

Table 11: Sample memory layouts for byte and 32-byte access.

The relationship between the 32-byte word addressable layout and the byte addressable layout is called "memory alignment" and the **Memory Align SM** is the state machine that checks the correctness of this relationship. Notice that, in the general case, `MLOAD` operation requires reading bytes of two different words. Considering that the content of the memory is the one shown at Table 11, since the EVM is addressed at a byte level, if we want to check a read from the EVM of a word starting at the address `0x22`, the value that we should obtain is the following:

$$\texttt{val} = \texttt{0x1f} \cdots \texttt{b7236e21}.$$

We denote the content of the words affected by an EVM memory read as $\texttt{m}_0$ and $\texttt{m}_1$. In our example, these words are the following:

$$\texttt{m}_0 = \texttt{0x88d11f}\cdots\texttt{b723}, \quad \texttt{m}_1 = \texttt{0x6e21ff}\cdots\texttt{54f9}.$$

We define a read block as the string concatenating the content of the words affected by the read: $\texttt{m}_0 \mid \texttt{m}_1$. Figure 12 shows the affected read words $\texttt{m}_0$ and $\texttt{m}_1$ that form the affected read block and the read value `val` for a read from the EVM at address `0x22` in our example memory of Table 11.

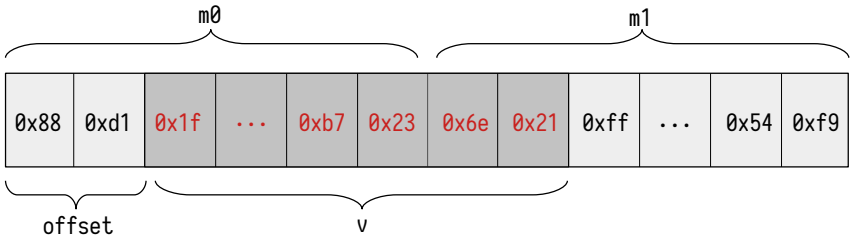

Figure 12: Schema of `MLOAD` example.

Let us now introduce the flow at the time of validating a read. Suppose that we want to validate that if we perform an `MLOAD` operation at the address `0x22`, we get the previous value $\texttt{0x1f}\cdots\texttt{7236e21}$. At this point, the main state machine will perform several operations. First of all, it will have to query for the values $\texttt{m}_0$ and $\texttt{m}_1$. Henceforth, it must call the **Memory SM** in order to validate the previous queries.

Observe that it is easy to extract the memory positions to query from the address `0x22`. In fact, if a is the memory position of the `MLOAD` operation, then $\texttt{m}_0$ is always stored at the memory position $\lfloor \frac{a}{32} \rfloor$ and $\texttt{m}_1$ is stored at the memory position $\lfloor \frac{a}{32} \rfloor + 1$. In our example, $a = \texttt{0x22} = 34$. Hence, $\texttt{m}_0$ is stored at the position $\lfloor \frac{32}{34} \rfloor = \texttt{0x01}$ and $\texttt{m}_1$ is stored at the position $\lfloor \frac{32}{34} \rfloor + 1 = \texttt{0x02}$.

Secondly, we should extract the correct `offset`. The `offset` represents an index between 0 and 31 indicating the number of bytes we should offset from the starting of $\texttt{m}_0$ to correctly place `val` in the block. In our case, the `offset` is 2. Similarly as before, it is easy to obtain the offset from a. In fact, the it is equal to $a \mod 32$. Now, the **Main SM** will check against the Memory Align State Machine that `val` is a correct read given the affected words $\texttt{m}_0$ and $\texttt{m}_1$ and the `offset`. That is, we should check that the value `val` can be correctly split into $\texttt{m}_0$ and $\texttt{m}_1$ using the provided `offset`.

Similarly, `MSTORE` instruction requires, in general, writing bytes in two words. The idea is very similar, but we are provided with a value `val` that we want to write into a specific location of the memory. We will denote by $\texttt{w}_0$ and $\texttt{w}_1$ the words that arise from $\texttt{m}_0$ and $\texttt{m}_1$ after the corresponding write.

Following our previous example, suppose that we want to write

$$\texttt{val} = \texttt{0xe201e6}\ldots\texttt{662b}$$

in the address `0x22` of the byte-addressed Ethereum memory. We are using the same $\texttt{m}_0$ and $\texttt{m}_1$ (and since we are writting into the same address as before) and they will transition into (see Figure 13):

$$\texttt{w}_0 = \texttt{0x88d1e201e6}\ldots, \quad \texttt{w}_1 = \texttt{0x662bff}\ldots\texttt{54f9}.$$

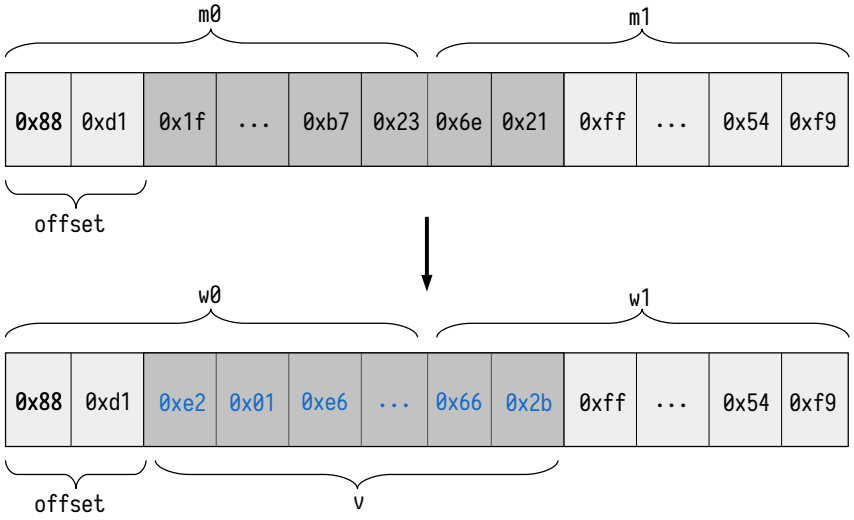

Figure 13: Schema of `MSTORE` example.

Just as before, the main state machine will need to perform several operations. We will be given an address `addr`, an offset value `offset` and a value to be wrote `val`. Identically as before, the **Main SM** will be in charge of reading the zkEVM memory to find $\mathtt{m}_0$ and $\mathtt{m}_1$ from the given address and offset. Of course, the validity of this query should be performed with a specific Plookup into the **Memory SM**, just as before.

Now, the **Main SM** can compute $\mathtt{w}_0$ and $\mathtt{w}_1$ from all the previous values in a uniquely way. The way of validating that we are providing the correct $\mathtt{w}_0$ and $\mathtt{w}_1$ is to perform a Plookup into the **Memory Align SM**. That is, we will check that the provided values $\mathtt{w}_0$ and $\mathtt{w}_1$ are correctly constructed from the provided `val`, $\mathtt{m}_0$, $\mathtt{m}_1$ and `offset` values.

Finally, the last opcode `MSTORE8` works similarly, *but it only affects one word* $\mathtt{m}_0$. Moreover, we can only write one byte and hence, only the less significant byte of `val` will be considered into the write. Observe that, in this opcode, $\mathtt{m}_1$ and $\mathtt{w}_1$ are unconstrained.

3.7 Storage SM

The **Storage SM** is in charge of validate operations over the key-value storage structure present in the EVM. The **Storage SM** is an essential building block for designing a more generic virtual machine that can check the correctness of state transitions resulting from executing smart contract transactions. The keys are expressed with 4 elements of $\mathbb{F}_p$, while the values are expressed using 8 elements in $\mathbb{F}_p$, where each of these eight elements have 0's at their most 32 significant bits.

The operations over the **Storage SM** are the typical Create, Read, Update and Delete (CRUD). A specific behavior of the **Storage SM** is that, in read operations, it returns zero if a key is not found. The structure used to build the key-value storage that can be proven with zero knowledge is a specific variant of a Merkle Patricia Tree (MPT).

In general, a Merkle tree is a data structure where every *leaf* of the tree contains the cryptographic hash of a value and every *non leaf*, which we also denote as *branch*, contains the concatenated hashes of its children. Merkle trees allow to link a set of values to a unique hash called the *root* of the tree and the efficient and secure verification of containment of large sets of key-values. In our case, we will use the key to unequivocally

determine the position of the leaf in the Merkle tree. Furthermore, we are going to use the previously explained Poseidon hash function because it is a function that can be verified with zero-knowledge primitives in a more friendly way than other standard hash functions like KECCAK. Moreover, two different instances of Poseidon hash function (achieved modifying the initial value for the capacity) should be used for hashing leafs and branches to prevent possible attacks.

The mechanics of the particular MPT proposed and its operations are described in detail later but essentially, proving operations over this hash structure involves to types of checks:

- For read operations, we need to show that the value of a related key is included at the tree at the correct position.
- For write operations, we have to prove that modifications lead to the appropriate new root.

3.8 Counters

Counters are a mechanism to control that the total number of steps do not exceed the maximum polynomial size. The purpose of counters is to limit the number of steps that can be taken during the processing of a transaction, in order to ensure that the computation terminates within a certain time bound, defined by the polynomial size, currently stored in a constant

```
CONST %TOTAL_STEPS = 2**23
```

to prevent the cryptographic proof system from failing due to a denial-of-service attack. Additionally, it is important to consider that there is a fixed minimum number of steps required to complete the processing of a transaction:

```
CONST %MIN_STEPS_FINISH_BATCH = 200
```

As a result, the total number of steps is determined by the following variable:

```
CONST %MAX_CNT_STEPS = %TOTAL_STEPS - %MIN_STEPS_FINISH_BATCH
```

Since there are operations that are more often used than others, it is a really good practice to limit the steps for each of the state machines individually. The values assigned to these counters are typically derived through a combination of design considerations and empirical testing. Design considerations may include factors such as the expected input rate or the maximum number of cycles the system can handle before failure. Empirical testing may involve running simulations or testing the system in a real-world environment to determine appropriate counter values.

```
CONST %MAX_CNT_ARITH = %TOTAL_STEPS / 32
CONST %MAX_CNT_BINARY = %TOTAL_STEPS / 16
CONST %MAX_CNT_MEM_ALIGN = %TOTAL_STEPS / 32
CONST %MAX_CNT_KECCAK_F = (%TOTAL_STEPS / 155286) * 44
CONST %MAX_CNT_PADDING_PG = (%TOTAL_STEPS / 56)
CONST %MAX_CNT_POSEIDON_G = (%TOTAL_STEPS / 30)
```

4 zkASM Language

4.1 Basic Syntax

This section is devoted to explain the basic syntax of zkASM from a high-level point of view. Advanced syntax is totally dependent of the use case (e.g. the design of a zkEVM) and will be explained in more detail in Section 5.

Comments are made with the semicolon **;** symbol:

```
; This is a totally useful comment
```

Multi-line comments are also supported via the following syntax:

```
/**
* Totally
* useful
* comment
*/
```

One can subdivide the zkASM code into multiple files and import code with the `INCLUDE` keyword. This is what we refer to as the modularity of the zkASM.

```
; File: main.zkasm

INCLUDE "utils.zkasm"
INCLUDE "constants.zkasm"
; -- code --
```

Each line of the code is, with some exception, considered as a clock of the state machine, named as **step**. There are several kinds of steps:

- *An assignment.* Assignments involve assigning a value to one or more registers within the same clock cycle. The synatx to invoke an assignment is shown below and its discussed in more detail in Section 4.2.

```
0 => A, B, C, ...
```

- *A list of instructions.* Another type of step is a list of instructions, which can be used to sequentially execute multiple operations among the registers. Each instruction in the list is executed in the same clock cycle. Section 5 provides more information on how to use instruction lists. To invoke an ordered list of instructions, use the following syntax:

```
            :INS1, INS2, INS3, ...
```

- *A combination of both, an assignment and a list of operations.* More information on this will be added on Sections 4.3 and 5. However, we propose below an use-case of this combined kind. The code below assigns the output of retrieving some value from the memory to the register `A`. The interesting part here is the way we retrieve the output of the load instruction, which is called a Free Input and will be explained in detail in Section 4.3.

```
$ => A^^I^^I^^I:MLOAD(address)
```

4.2 Assignments

An assignment is a statement in zkASM that is used to set a certain value to a register. Assignments in zkASM are denoted using the `=>` operator. For example, the following piece of code assigns the value 0 to the register `A`:

```
0 => A
```

However, in zkASM there are a lot of options when dealing with assignments. Let us deep into each of them. First of all, one invoke an assignment to several (and not only one) registers using colons. For example, the code below assigns the value 0 to registers `A`, `B` and `C`:

```
0 => A, B, C
```

Now, let us define all possible assignable elements. We have just seen that we can assign a constant value, like 0. However, we can also assign other elements. More concretely, we can input linear combinations (with scalars being numbers or big numbers) of the following kinds of elements:

- Free Input tags (`${}`, `$`). See Section 4.3.
- Registers (`A`, `B`, `C`, `D`, `E`, `SP`, `RR`, ...).
- Counters (`CNT_ARITH`, `CNT_BINARY`, `CNT_KECCAK_F`, `CNT_MEM_ALIGN`, `CNT_PADDING_PG`, `CNT_POSEIDON_G`).
- Numbers or big numbers (`0`, `0xFF`, `0n`, `0xFFn`, ...).
- Numbers' exponentiations (`2 ** 2`, `2 ** 0xFF`, ...). This numbers' exponentiations are being treated in turn as numbers or big numbers.
- Constants (see Section 4.4). Take into account that usage of `CONST` and `CONSTL` at the same time is not allowed in the same operation.

Below, we show several examples of valid assignments

```
-A + B => A, B
-2**2*A + B => A, B
2*A + 3*B => A, B
$ => A, B^^I^^I^^I^^I:MLOAD(addr)
${executorMethod()} => A, B
${A >> 2} => C, D
${A & 0x03} => C, D
```

4.3 Free Inputs and Commands

Free Inputs are values captured by the compiler which are not directly checked to be correct during execution. Free Inputs are introduced using the dollar operator `${}` under braces. We will call this statements **Free Inputs tags**. There are two ways we can introduce Free Inputs using the dollar operator:

- *Receiving values from instructions via free inputs.* Some instructions, like `MLOAD` defined in Section 5 sends its corresponding output through Free Inputs. For example, the program below saves the value stored in the memory slot corresponding to the address `someAddr` into the register `A` through a Free Input.

```
        $ => A^^I^^IMLOAD(someAddr)
```

- *Using a already implemented executor function.* The compiler can capture functions programmed inside the executor. The syntax to invoke them is the following one

```
${ executorMethod(params) } => A
```

where `executorMethod(params)` is a function which is programmed in the executor and can take several registers as parameters. An example on this can be found in the elliptic curves operations defined in the instructions section 5:

```
{xAddPointEc(A,B,C,D)} => E
```

Moreover, we can also use plain Javascript inside a zkASM program to introduce Free Inputs. This special feature can be invoked using the double dollars operator `$${}` under braces. We will call this statements **Commands**. For example, one can define temporal variables that can be accessed and modified by Free Input tags at any point of the program. For example, the code below defines a temporal variable `_someTmpVar` which will be equal to the sum of the values stored in registers `A` and `C`. Posteriorly, this variable will be captured by a Free Input tag, shifted 256 bits and reassigned to the `A` register. Note that we can use expressions under Free Input tags.

```
$${ var _someTmpVar = A + C }
${ _someTmpVar >> 256 } => A
```

4.4 Constants Definition

As in many programming languages, one can define constants in zkASM to avoid magic numbers. In order to explore all the possibilities that the compiler brings us, let us define the concept of a **numerical expression**.

A numerical expression (`nexpr`) can be defined recursively as

- **Arithmetic expressions** among numerical expressions: `nexpr` + `nexpr`, `nexpr` − `nexpr`, `nexpr` `*` `nexpr`, `nexpr` `**` `nexpr`, `nexpr` `%` `nexpr`, `nexpr` `/` `nexpr`.
- A **shift** of numerical expressions: `nexpr` ≪ `nexpr` or `nexpr` ≫ `nexpr`.
- A **bit-wise operation** of numerical expressions: `nexpr` ∧ `nexpr`, `nexpr` `|` `nexpr` or `nexpr` `&` `nexpr`.
- A **boolean operation** of numerical expressions: `nexpr` `<` `nexpr`, `nexpr` `>` `nexpr`, `nexpr` `<=` `nexpr`, `nexpr` `>=` `nexpr`, `nexpr` `==` `nexpr`, `nexpr` `!=` `nexpr`, `nexpr` `&&` `nexpr`, `nexpr` `||` `nexpr`.
- The **negation** of a numerical expression: `!nexpr`.
- A **ternary operator** over numerical expressions: `nexpr` `?` `nexpr` `:` `nexpr`.
- Numerical expressions can also involve **parenthesis**: (`nexpr`).

Since we defined `nexpr` recursively, we should define the base cases:

- A numerical value: `0` or `0x01`.
- A long numerical value: `0n` or `0x04n`.
- A constant identifier, that we will define below.

To define a constant we will use the following syntax:

```
CONST %CONSTID = nexpr
```

where `CONSTID` is a unique identifier for the constant we are defining. The way to invoke a constant inside can be shown in the example below, where we assign the value of the constant with identifier `SOMECONST` to the register `A`:

```
CONST %SOMECONST = 5
%SOMECONST => A
```

Observe that the definitions of a numerical expression allows us to define complex constants like:

```
CONST %NUMBERA = 10
CONST %NUMBERB = 2
CONST %EXP = %NUMBERA ** %NUMBERB
CONST %TERNARY = %EXP == 100 ? 1 : 0
```

Another feature that implements the compiler is the ability to choose between an expression or a constant depending on the existence of such a constant. For example, in the following piece of code:

```
CONST %NUMBERA = %NONEXISTINGCONST ?? 10
```

the constant `%NUMBERA` will take the value 10. However, in the code below:

```
CONST %EXISTINGCONST = 2
CONST %NUMBERA = %EXISTINGCONST ?? 10
```

the constant `%NUMBERA` will take the value 2.

4.5 Variables Definition

4.6 Some Examples

This section serves as a compendium of useful examples.

```
    opADD:
    SP - 2          :JMPN(stackUnderflow)
    SP - 1 => SP
    $ => A          :MLOAD(SP--)
    $ => C          :MLOAD(SP)

    ; Add operation with Arith
    A               :MSTORE(arithA)
    C               :MSTORE(arithB)
                    :CALL(addARITH)
    $ => E          :MLOAD(arithRes1)
    E               :MSTORE(SP++)
    1024 - SP       :JMPN(stackOverflow)
    GAS-3 => GAS    :JMPN(outOfGas)
    :JMP(readCode)
```

Let us explain in detail how the `ADD` opcode gets interpreted by us. Recall that at the beginning the stack pointer is pointing to the next ”empty” address in the stack:

- First, we check if the stack is filled ”properly” in order to carry on the `ADD` operation. This means that, as the `ADD` opcode needs two elements to operate, it is checked that these two elements are actually in the stack:

```
        SP - 2          :JMPN(stackUnderflow)
```

If less than two elements are present, then the `stackUnderflow` function gets executed.

- Next, we move the stack pointer to the first operand, load its value and place the result in the `A` register. Similarly, we move the stack pointer to the next operated, load its value and place the result in the `C` register.

```
SP - 1 => SP
$ => A          :MLOAD(SP--)
$ => C          :MLOAD(SP)
```

- Now its when the operation takes place. We perform the addition operation by storing the value of the registers `A` and `C` into the variables `arithA` and `arithB` and then we call the subrutine `addARITH` that is the one in charge of actually performing the addition.

```
A               :MSTORE(arithA)
C               :MSTORE(arithB)
:CALL(addARITH)
$ => E          :MLOAD(arithRes1)
E               :MSTORE(SP++)
```

Finally, the result of the addition gets placed into the register `E` and the corresponding value gets placed into the stack pointer location; moving it forward afterwise.

- A bunch of checks are performed. It is first checked that after the operation the stack is not full and then that we do not run out of gas.

```
1024 - SP       :JMPN(stackOverflow)
GAS-3 => GAS    :JMPN(outOfGas)
:JMP(readCode)
```

Last but not least, there is an instruction indicating to move forward to the next intruction.

5 zkASM instructions set

5.1 Memory Related Instructions

We refer to the memory as a volatile read-write data storage that exists only during the execution of a zkASM program. The memory is divided into different contexts of `0x40000` words. Each word is 256 bits in length, so each context is 8 MB in size.

Each context is divided into the following three blocks:

- **VARS:** With a relative offset of `0x00000` and a height of `0x10000` words (`2MB`), contains the local context variables pre-defined in the language. The list of all context variables can be found at `vars.zkasm`.
- **STACK:** With a relative offset of `0x10000` and a height of `0x10000` words (`2MB`), contains the stack of the the EVM. **STACK** is defined once per context.
- **MEMORY:** With a relative offset of `0x20000` and a height of `0x20000` words (`4MB`), contains the free memory that can be freely used. **MEMORY**, like **STACK**, is also defined once per context.

Therefore, for a given slot in memory, its pointer is computed as:

$$\texttt{memoryAddress} = \texttt{0x40000} \cdot \texttt{CTX} + \texttt{isStack} \cdot (\texttt{0x10000} + \texttt{SP}) + \texttt{isMem} \cdot (\texttt{0x20000} + \texttt{offset})$$

where:

- `CTX`: This integer variable refers to the memory context being accessed in the EVM's memory.
- `isStack`: this boolean value indicates whether the memory operation being performed is related to the EVM's stack. The EVM uses a stack-based architecture, meaning that operations are performed by pushing and popping values on and off the stack.
- `SP`: This variable refers to the current position of the stack pointer in the EVM's stack. The stack pointer is used to keep track of the current top of the stack. More information on `SP` will be added below.
- `isMem`: This boolean value indicates whether the memory operation being performed is related to the EVM's memory. Observe that `isStack` and `isMem` can not be 1 at the same time.
- `offset`: This variable likely refers to the offset or location within the current memory context being accessed.

Observe that, following the above description of the memory, the former set of variables completely determine a memory slot. Figure ?? shows the structure of the memory during a zkASM execution.

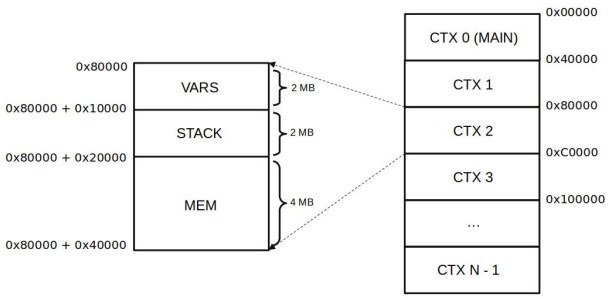

Figure 14: Schema of contexts and memory regions of the zkEVM.

In order to allow runtime interaction with memory, zkASM has a couple of instructions to read and write values to it.

Let us first define how can be access to a specific address of the memory. Memory acess will be parametrized by 2 integer parameters: the address `addr` and the relative address `relAddr`.

- `addr`: Address would be able to define a memory access up to memory region level. That is, `addr` will contain information of which context access the memory and if the access is directed to the system variables **VARS**, the **STACK** or the **MEMORY**.
- `relAddr`: However, `addr` is not enough. Hence, `relAddr` will point to a specific memory slot inside a concrete memory region of a context. We should take into account that `relAddr` can never be negative. Moreover, `relAddr` is also bounded

from above: if we are accessing to the **MEMORY**, `relAddr` should be strictly less than `0x40000` (because memory measures `4MB`) and if we are accessing to the **STACK** or to the system variables **VARS**, `relAddr` should be strictly less than `0x10000` (because both memory regions measure `2MB`). Then, the wanted memory slot will be equal to `addr` + `relAddr`.

Let us know how to specify `addr` and `relAddr` in zkASM language. We can specify `addr` with 3 keywords: `SYS` (which corresponds to **VARS** memory region), `STACK` (which corresponds to **STACK** memory region) and `MEM` (which corresponds to **MEMORY** memory region). Invoking `STACK` will increase `addr` by `0x10000` and similarly, invoking `MEM` will increase `addr` by `0x20000`. However, since **VARS** is the first memory region, it will not produce an increase in `addr`. Moreover, `addr` will also increase depending on the current context. More specifically, a total amount of `0x40000` will be increased per context, following our memory description above.

To specify `relAddr` we can use 2 registers: `E` and `RR`. If `E` is chosen, `relAddr` will increase a total amount of E_0 units. Similarly, if `RR` is chosen, `relAddr` will increase by `RR`. Moreover, we can add a numeric offset to increase a fixed amount of units `addr` + `relAddr`.

The syntax will be, in each case:

```
addr:relAddr
; or
addr:relAddr+offset
; or
addr:relAddr-offset
```

Below, we specify concrete examples on how to specify addresses:

```
SYS:E
SYS:E+1
STACK:RR
MEM:E
MEM:E-2
MEM:RR+1
```

To end up, we can also access to addresses by means of global and local variables, defined elsewhere in the assembly code. This variables will have a unique identifier that we will use in order to access to it.

5.1.1 MLOAD

`MLOAD` is the zkASM instruction used to read a value from a specific address in memory. It takes the pointer address of the memory slot to be read as a parameter. We can store the value that is read in a register of our choosing by using the free input (`$`) assignment.

Suppose that we want to read the memory value stored in the address `someAddr` in a certain context. The address `someAddr` is defined by the global variable:

```
VAR GLOBAL someAddr
```

The following zkASM code stores the corresponding value into the register `A`:

```
$ => A          :MLOAD(someAddr)
```

5.1.2 MSTORE

`MSTORE` is the zkASM instruction used to write a value to a specific address in memory. It takes the pointer address of the memory slot to be read as a parameter, the register that contains the value to be writed must be also specified.

The following example shows how to store in the memory the value of the current context. Note that as the MSTORE parameter, it is specified the variable that contains the pointer to where current context is stored in the memory. The value stored will be taken from CTX register:

```
CTX      :MSTORE(currentCTX)
```

5.1.3 Dealing with the STACK

A stack machine is a machine in which temporary values for computations are moved to and from a push down stack. Operations over the stack are the typical: PUSH, POP, DUP, SWAP, etc. Since the EVM is a stack-based virtual machine, we reserve an address space to create a stack within the memory of the zkEVM. The classical pointer called STACK POINTER (SP) contains the address of the **next free position on the** STACK. A POP from the STACK can be implemented as:

```
SP -1 => SP
$ => A^^I^^I:MLOAD(SP)
```

where we decrement SP to reposition it on the last element of the stack and then we load this element into registry A. Similarly, a PUSH into the STACK can be implemented as:

```
0^^I^^I:MSTORE (SP++)
```

which saves a 0 at the top of the stack and increments SP. An important note about both the stack and the memory is that the stack pointer and the memory are per context.

5.1.4 MEM_ALIGN_RD

Although the memory word in zkASM is 256 bits long, in roder to mimic the regular Ethereum Virtual Machine (EVM) memory behavior, zkASM has a specific instructions for accessing memory at the byte level. The instruction MEM_ALIGN_RD enables reading 32 bytes starting from an offset of any byte in memory. In this way, two memory registers are read and a the following transformation is applied to virtually obtain a new 32-byte word as a result of the reading.

$$\texttt{val} = \Big[\texttt{m}_0 \ll 8 \cdot \texttt{offset}\Big] \| \Big[\texttt{m}_1 \gg 256 - 8 \cdot \texttt{offset}\Big]$$

Here, the symbol $\|$ denotes string concatenation. The registers must be set as follows before call the MEM_ALIGN_RD instruction. We can store the value that is read in a register of our choosing by using the free input ($) assignment.

Register	MEM_ALIGN_RD parameters
A	Memory Slot of $\texttt{m}_0$
B	Memory Slot of $\texttt{m}_1$
C	offset

Table 15: MEM_ALIGN_RD instruction parameters.

The following example shows how to read 32 bytes that are stored occupying part of two consecutive zkASM memory words. The read value will be stored in register A:

```
$ => A          :MLOAD(someAddr)
$ => B          :MLOAD(someAddr+1)
16 => C

$ => A          :MEM_ALIGN_RD
```

Figure ?? shows how the 32-bytes value will be read for the MEM_ALIGN_RD given example.

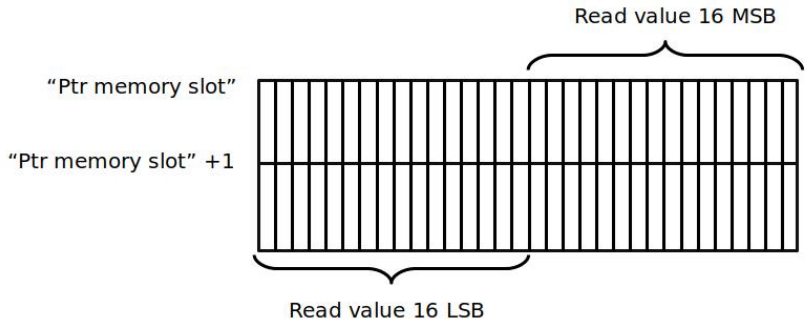

Figure 16: Example of how values are read from the memmory using MEM_ALIGN_RD with an offset of 16.

5.1.5 MEM_ALIGN_WR

MEM_ALIGN_WR is equivalent to MEM_ALIGN_RD but for writing a 32-byte value. In this case, we have to specify two memory slots that will be written after applying the following transformation to the value to be stored. The registers that contains the value to be stored must also be specified.

$$w_0 = \left[m_0 \ \& \ \left(2^{256} - 2^{256-8\cdot \mathtt{offset}}\right)\right] \| \left[\mathtt{val} \ll 8 \cdot \mathtt{offset}\right]$$
$$w_1 = \left[m_1 \ \& \ \left((2^{256} - 1) \gg 8 \cdot \mathtt{offset}\right)\right] \| \left[\mathtt{val} \ll 8 \cdot \mathtt{offset}\right]$$

Register	MEM_ALIGN_WR parameters
A	Memory Slot of m_0
B	Memory Slot of m_1
C	offset
D	w_0
E	w_1
op	Value to be written

Table 17: MEM_ALIGN_WR instruction parameters.

The following example shows how to write 32 bytes that are stored occupying part of two consecutive zkASM memory words. The value to be stored will be taken from free input register:

```
$ => A          :MLOAD(MEM:E)
$ => B          :MLOAD(MEM:E+1)

${memAlignWR_W0(A,mem.bytesToStore,C)} => D                  ; no trust calculate W0
${memAlignWR_W1(B,mem.bytesToStore,C)} => E                  ; no trust calculate W1
$               :MEM_ALIGN_WR,MLOAD(bytesToStore)
```

5.1.6 MEM_ALIGN_WR8

MEM_ALIGN_WR8 allows writing only 8 bits of a specific memory slot. In this case, we have to specify the memory slot to be written, the register that contains the byte to be stored, and the offset value that situates the byte in a specific position of the 32-byte word. The value will be written after applying the following transformation:

$$w_0 = \Big[m_0 \ \&\ (\texttt{maskByte} \gg 8 \cdot \texttt{offset})\Big] \| \Big[(\texttt{bits}\ \&\ \texttt{0xFF}) \ll 8 \cdot (31 - \texttt{offset})\Big]$$

where maskByte equals $2^{256} - 1$.

Register	MEM_ALIGN_WR8 parameters
A	Memory Slot of m_0
C	offset
D	w_0
op	Value to be written

Table 18: MEM_ALIGN_WR8 instruction parameters.

The following example shows how to write 1 bytes stored in the byte 4 of a specific storage slot. The value to be stored will be taken from B register:

```
4 => C
$ => A          :MLOAD(someAddr)
${memAlignWR8_W0(A,B,C)} => D  ; no trust calculate W0
B               :MEM_ALIGN_WR8 ; only use LSB of B, rest of bytes could be non zero
```

5.2 Storage Related Instructions

Polygon zkEVM, like Ethereum L1, has a storage component for storing persistent on-chain data, which includes the balances of all accounts, their nonces, and the state of all deployed smart contracts along with their codes. The data that forms the state is represented as cryptographic trie, but while Ethereum L1 uses a modified Patricia tree with Keccak256 as the hash operation, Polygon zkEVM uses a binary sparse Merkle tree with Poseidon as the hash operation (refer to the technical documents regarding the zkEVM bridge annex A to learn more about sparse Merkle trees).

Poseidon is a hash function that's specifically designed for use in zero-knowledge applications, as it's meant to operate with values of a prime field and it has been proven to be much more performant than Keccak256 in zero-knowledge constructions like those used in Polygon zkEVM. Moreover Poseidon hash has an input named capacity, which can be used as an extra input value.

Barely, the Polygon zkEVM state tree is a key-value structure in which the integrity can be ensured by a 256-bit value known as the state root. Each entry in the tree is a leaf and directly stores a 256-bit value. Additionally, the index position of that leaf in the tree corresponds to the 256-bits of the key. As can be seen in Figure X, since the keys are 256-bits in length, the tree has 32 levels and a total capacity of 2^{256} leaves.

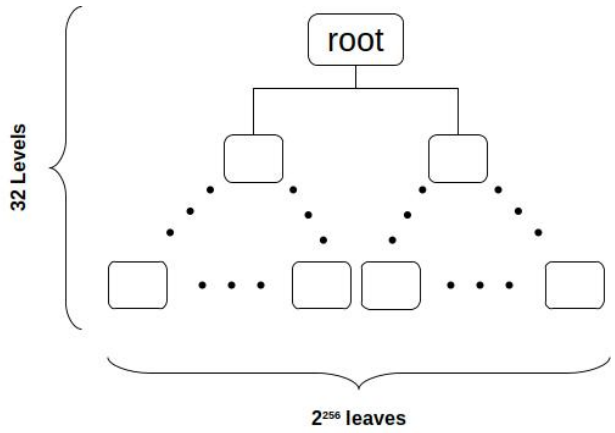

Figure 19: Polygon zkEVM state trie.

Since five different types of values can be stored, a distinction must be made among the five types of leaves. Table X shows the relation between leaf type and the corresponding data types that they contain.

Leaf type	Data type
0	`Account balance`
1	`Account nonce`
2	`Contract code hash`
3	`Contract storage slot value`
4	`Contract code length`

Table 20: Polygon zkEVM state tree leaf types.

For each leaf entered in the tree its key is computed as follows:

$$\mathsf{key} = \mathsf{Poseidon}(\mathsf{key\_seed})$$

where `key_seed` is a 32-bytes integer constructed as follows:

$$\mathsf{key\_seed} = (\mathsf{account\_address} \,||\, \mathtt{0x00000000} \,||\, \mathsf{leaf\_type} \,||\, \mathtt{0x00000000})$$

being $\mathsf{account\_address} \in \{0, 1, \ldots, 2^{160} - 1\}$ is a 20-bytes integer and $\mathsf{leaf\_type} \in \{0, 1, \ldots, 2^{32} - 1\}$ a 4-bytes integer.

The capacity of the Poseidon's instance used when computing keys is always zero except in the case where the leaf corresponds to a contract storage slot value (leaf type 3), in which case the capacity is directly set to the storage slot pointer. It's important to note that because the contract storage slot pointer is not encoded in the 32-byte input, if we do not use the capacity, all storage slots for the same contract would lead to the same state tree leaf.

Te value sored in the leaf has the following structure:

$$(v_0, \ldots, v_7)$$

codifying a 256-bits unsigned integer, where $v_i \in \mathbb{F}_p$ are bounded to 32-bits each.

In order to allow runtime interaction with the Polygon zkEVM state tree zkASM has a couple of instructions to read and write values on it.

5.2.1 SLOAD

SLOAD is the zkASM instruction used to read a value from a leaf in the state tree. It takes Poseidon's "Input" and "Capacity" parameters, along with the leaf type, to compute the key of the leaf to be read. The registers must be set as follows before call the SLOAD instruction.

Register	SLOAD parameters
A	Account address
B	Leaf type
C	Contract storage slot pointer (capacity)

Table 21: SLOAD instruction parameters.

We can store the value that is read in a register of our choosing by using the free input ($) assignment.

The following example shows how to read the balance of a specific account, the value that is read will be stored in the E register:

```
1  someAccountAddr => A
2  0 => B
3  0 => C
4  $ => E          :SLOAD
```

The following example shows how to read a storage slot of a specific contract, the value that is read will be stored in the E register:

```
1  someAccountAddr => A
2  3 => B
3  storageSlotPtr => C
4  $ => E          :SLOAD
```

5.2.2 SSTORE

SSTORE is the zkASM instruction used to store a value to a leaf in the state tree. It takes Poseidon's "Input" and "Capacity" parameters, along with the leaf type, to compute the key of the leaf to be writen, in addition takes the value to be written. The registers must be set as follows before call the SSTORE instruction.

Register	SSTORE parameters
A	Account address
B	Leaf type
C	Contract storage slot pointer (capacity)
D	Value to write

Table 22: SSTORE instruction parameters.

The following example shows how to write a storage slot of a specific contract:

```
1  someAccountAddr => A
2  3 => B
3  storageSlotPtr => C
4  value => D
5  A               :SSTORE
```

5.3 Binary-Related Instructions

The arithmetic operators are used to perform arithmetic mathematical operations on numeric data stored in registers.

5.3.1 ADD

`ADD` is used to sum the content of the registers `A` and `B`, the result will be treated as a free input. The following example shows how to use `ADD` instruction.

```
val1 => A
val2 => B

$ => C              :ADD ; [ val1 + val2 => C]
```

5.3.2 SUB

`SUB` is used to subtract de content fo register `B` to `A`, the result will be treated as a free input. The following example shows how to use `SUB` instruction.

```
val1 => A
val2 => B

$ => C              :SUB ; [ val1 - val2 => C]
```

5.3.3 LT

`LT` instruction is used to compare the values of the registers `A` and `B` as **unsigned integers**. The output of the operation will be 1 if `A` is actually lower than `B` (that is, $A < B$) and 0 otherwise (that is, $A \geq B$). The output of the instruction will be treated as a free input. The next lines of code show an example on how to use `LT` instruction:

```
valA => A
valB => B

$ => C^^I^^I^^I:LT ; [1 if A < B, 0 if A <= B]
```

5.3.4 SLT

`SLT` instruction is used to compare the values of the registers `A` and `B` as **signed integers**, explained in Section 3.2. The output of the operation will be 1 if `A` is actually lower than `B` (that is, $A < B$) and 0 otherwise (that is, $A \geq B$). The output of the instruction will be treated as a free input. The next lines of code show an example on how to use `SLT` instruction:

```
valA => A
valB => B

$ => C^^I^^I^^I:SLT ; [1 if A < B, 0 if A <= B]
```

5.3.5 EQ

`EQ` instruction is used to compare the equality relationship between the values of the registers `A` and `B`. The output of the operation will be 1 if `A` is equal to `B` (that is, $A = B$) and 0 otherwise (that is, $A \neq B$). The output of the instruction will be treated as a free input. The next lines of code show an example on how to use `EQ` instruction:

```
valA => A
valB => B

$ => C^^I^^I^^I:EQ ; [1 if A = B, 0 if A != B]
```

5.3.6 AND

`AND` instruction is used to perform the bit-wise `AND` operation between registers `A` and `B`, as explained in Section 3.2. The output of the instruction will be treated as a free input. The next lines of code show an example on how to use `AND` instruction:

```
valA => A
valB => B

$ => C^^I^^I^^I:AND
```

For sake of completeness, let us propose a more concrete example, where we assign the value `0xDBn` to `A` and the value `0x86n` to `B`. The result of the bit-wise `AND` operation is going to be `0x82n` because

$$\texttt{A} = \texttt{0b11011011}, \quad \texttt{B} = \texttt{0b10000110} \Longrightarrow \texttt{C} = \texttt{0b10000010}.$$

```
0xDBn => A
0x86n => B

$ => C^^I^^I^^I:AND ; C = 0x82n
```

5.3.7 OR

`OR` instruction is used to perform the bit-wise `OR` operation between registers `A` and `B`, as explained in Section 3.2. The output of the instruction will be treated as a free input. The next lines of code show an example on how to use `OR` instruction:

```
valA => A
valB => B

$ => C^^I^^I^^I:OR
```

For sake of completeness, let us propose a more concrete example, where we assign the value `0xDBn` to `A` and the value `0x86n` to `B`. The result of the bit-wise `OR` operation is going to be `0xDFn` because

$$\texttt{A} = \texttt{0b11011011}, \quad \texttt{B} = \texttt{0b10000110} \Longrightarrow \texttt{C} = \texttt{0b11011111}.$$

```
0xDBn => A
0x86n => B

$ => C^^I^^I^^I:OR ; C = 0xDFn
```

5.3.8 XOR

`XOR` instruction is used to perform the bit-wise `XOR` operation between registers `A` and `B`, as explained in Section 3.2. The output of the instruction will be treated as a free input. The next lines of code show an example on how to use `XOR` instruction:

```
valA => A
valB => B

$ => C^^I^^I^^I:XOR
```

For sake of completeness, let us propose a more concrete example, where we assign the value `0xDBn` to `A` and the value `0x86n` to `B`. The result of the bit-wise `XOR` operation is going to be `0x5Dn` because

$$\texttt{A} = \texttt{0b11011011}, \quad \texttt{B} = \texttt{0b10000110} \Longrightarrow \texttt{C} = \texttt{0b01011101}.$$

```
0xDBn => A
0x86n => B

$ => C^^I^^I^^I:XOR ; C = 0x5Dn
```

5.4 Arithmetic-Related Instructions

5.4.1 ARITH

The `ARITH` instruction allows to check field operations. More specifically, it checks a combination of an addition and a product, as explained in Section 3.3. Before calling the `ARITH` instruction, registers `A`, `B`, and `C` must be set. The equation that follows will be evaluated using the values of these 3 registers. It is necessary to specify where the result of the evaluation will be stored. If the evaluation results in an overflow of the output register, the overflow value will be stored in register `D`. More specifically, the equation that checks the `ARITH` instruction is the following one:

$$\texttt{D} \cdot 2^{256} + \texttt{op} = \texttt{A} \cdot \texttt{B} + \texttt{C}$$

The following example shows how to use `ARITH` instruction. The result of the evaluation will be stored in register `A`, and if there is an overflow, it will be stored in register `D`:

```
valA => A
valB => B
valC => C
A            :ARITH ; [ valA * valB + valC => [D,A]]
```

5.4.2 ARITH_ECADD_DIFFERENT

The `ARITH_ECADD_DIFFERENT` instruction allows to perform additions $P + Q$ over the elliptic curve defined in Section 3.3. This instruction can not perform doublings, since the input points to be added are supposed to be different. This is not explicitly check, but since the doubling formula differs a lot from the distinct point addition formula, the result will be wrong if $P = Q$. The input parameters of the instruction are specified in the table below:

Register	ARITH_ECADD_DIFFERENT parameters
A	x_1, x coordinate of P
B	y_1, y coordinate of P
C	x_2, x coordinate of Q
D	y_2, y coordinate of Q
E	x_3, x coordinate of $P + Q$
op	y_3, y coordinate of $P + Q$

Table 23: `ARITH_ECADD_DIFFERENT` instruction parameters.

An example on how to use the `ARITH_ECADD_DIFFERENT` instruction can be seen in the code blow. Observe that we make use of the executor implemented functions

`xAddPointEc(A,B,C,D)` and `yAddPointEc(A,B,C,D)` which compute the x and the y coordinate of $P + Q$ being $P = (A, B)$ and $Q = (C, D)$ whenever $P \neq Q$. After this is computed, the x coordinate of $P + Q$ is is stored into the memory slot given by the address `addX` and, similarly, the y coordinate of $P + Q$ is pushed into the memory slot given by the address `addY`. If we have used incorrect values for the coordinates of $P + Q$, an executor error will pop. This will also be captured when the proof of the batch is generated, since the instruction invocation fills the polynomials of the Arithmetic State Machine correctly.

```
$ => A  ^^I^^I^^I^^I^^I:MLOAD(Px)
$ => B  ^^I^^I^^I^^I^^I:MLOAD(Py)
$ => C  ^^I^^I^^I^^I^^I:MLOAD(Qx)
$ => D  ^^I^^I^^I^^I^^I:MLOAD(Qy)
${xAddPointEc(A,B,C,D)} => E  ^^I:MSTORE(addX)
${yAddPointEc(A,B,C,D)} ^^I^^I:ARITH_ECADD_DIFFERENT, MSTORE(addY)
```

5.4.3 ARITH_ECADD_SAME

The `ARITH_ECADD_SAME` instruction allows to perform point doublings $2P$ over the elliptic curve defined in Section 3.3. The input parameters of the instruction are specified in the table below:

Register	ARITH_ECADD_DIFFERENT parameters
A	x_1, x coordinate of P
B	y_1, y coordinate of P
E	x_3, x coordinate of $2P$
op	y_3, y coordinate of $2P$

Table 24: `ARITH_ECADD_DIFFERENT` instruction parameters.

An example on how to use the `ARITH_ECADD_SAME` instruction can be seen in the code blow. Observe that we make use of the executor implemented functions `xDblPointEc(A,B)` and `yDblPointEc(A,B)` which compute the x and the y coordinate of $2P$ being $P = (A, B)$. After this is computed, the x coordinate of $2P$ is is stored into the memory slot given by the address `doublePx` and, similarly, the y coordinate of $2P$ is pushed into the memory slot given by the address `doublePy`. If we have used incorrect values for the coordinates of $2P$, an executor error will pop. This will also be captured when the proof of the batch is generated, since the instruction invocation fills the polynomials of the Arithmetic State Machine correctly.

```
$ => A  ^^I^^I^^I^^I^^I:MLOAD(Px)
$ => B  ^^I^^I^^I^^I^^I:MLOAD(Py)

${xDblPointEc(A,B)} => E  ^^I:MSTORE(doublePx)
${yDblPointEc(A,B)} ^^I^^I:ARITH_ECADD_SAME, MSTORE(doublePy)
```

5.5 Execution Control Flow Related Instructions

In order to allow to conditional branch execution of the zkASM programs, 4 different instruction has been included in zkASM instructios set.

5.5.1 JMP

`JMP` is an unconditional jump instruction that always causes a jump in the program's execution flow, regardless of any conditions. It takes an address of the ROM as a parameter

to continue the execution flow. To avoid using numeric pointers for jumps, zkASM allows jump destinations to be aliased with custom names. The compiler resolves these aliases and substitutes them with pointers later on.

The following code shows the general usage of `JMP` instruction:

```

; ...
; Executed Code
; ...

        :JMP(destinationLabel)

; ...
; Non Executed Code
; ...

destinationLabel:

; ...
; Executed Code
; ...
```

Moreover, we can also parametrize the destination of a jump-like instruction using either the first limb of the register $\texttt{E}_0$ or the register `RR`, using the syntax below:

```
        :JMP(RR)
        :JMP(E)
```

For example, the code below will produce a jump of 5 units in the execution flow:

```
5 => E
            :JMP(E)
```

The former syntax will be also available for all the other kind of jumps specified in below sections, including the ones having an else clause.

5.5.2 JMPN

`JMPN` is a conditional jump instruction that causes a jump in the program's execution flow if a specified register contains a negative number. It takes the address of the ROM as a parameter to continue the execution flow. The register that contains the value to be evaluated must also be specified.

In the following example, the execution flow will be redirected to `stackUnderflow` in the case that the evaluation of $\texttt{SP} - 2$ leads to a negative number.

```
; check stack underflow
SP - 2          :JMPN(stackUnderflow)
```

Conditional jumps can also receive an else-clause label. The synatx is very similar:

```
A^^I^^I^^I^^I:JMPN(ifClauseLabel, elseClauseLabel)


ifClauseLabel:
        ; do something


elseClauseLabel:
        ; do something different
```

If the value stored in the `A` register is negative, then the execution of the program will continue under the `ifClauseLabel` label. However, if the value of `A` is bigger or equal than 0, then the execution will continue from the label `elseClauseLabel`.

5.5.3 JMPC

JMPC is a conditional jump instruction that causes a jump in the program's execution flow if a specified condition is evaluated as true. It is used along with the following comparative instructions.

- EQ: Evaluates if register A value is equal to register B value.
- LT: Evaluates if register A value is less than register B value.
- SLT: Evaluates if register A value is less than register B value also comparing negative values.

In the following example, the execution flow will be redirected to absIsNeg in the case that the value contained in register A (namely val) is a negative number.

```
val => A
0 => B
$          :SLT, JMPC(absIsNeg)
```

5.5.4 JMPZ

JMPZ is a conditional jump instruction that causes a jump in the program's exectuion flow if a specified register contains a 0. In the following example, the execution flow will be redirected to readCode in the case that the value contained in register A (namely val) is 0.

```
val => A
A                  :JMPZ(readCode)
```

5.5.5 JMPNC and JMPNZ

In addition to all the previously defined execution jumps, there are two more jumps: JMPNC and JMPNZ. This kind of jumps works the other way around JMPC and JMPZ does. More concretely, in the code below, the execution will jump to the label someLabel if and only if the value stored in the register A **is not** zero. Otherwise, if the condition is satisfied, the execution will proceed normally.

```

A^^I^^I^^I:JMPNZ(someLabel)


someLabel:
    ; do something
```

Hence, the first argument appearing in the instruction denotes now the else-clause. In fact, we can also adopt an if-else structure in this kind of negated instructions:

```

A^^I^^I^^I:JMPNZ(elseLabel, ifLabel)

elseLabel:
    ; do something

ifLabel:
    ; do something
```

In the above piece of code, if the value of A is not zero, then the execution will jump to the label elseLabel and, otherwise, it will jump to ifLabel.

5.5.6 ASSERT

`ASSERT` is used to ensure that a given register has the same value as register `A`. A failing assertion, meaning that the values are unequal, will stop execution and throw error during runtime. Additionally, an execution that contains a failing assertion cannot generate a valid CI proof.

The following code will compare $\texttt{val}_1$ with $\texttt{val}_2$, and if they are not equal the execution will be immediately stopped:

```
val1 => A
val2 => B
B    ^^I^^I:ASSERT
```

5.5.7 Subroutines (CALL and RETURN)

Subroutines allow breaking down the code into smaller sections that can be called by using only a `CALL` instruction. A subroutine is designed to be reusable and can be called by other parts of the program. The code of a subroutine always ends with a `RETURN` instruction. When a subroutine is called, control is transferred from the main program to the subroutine. The subroutine then executes its code and, when it's finished, control is returned to the point in the main program immediately following the point where the subroutine was called.

An example of subroutine can be the `ecrecover_tx` subrutine used in the zkEVM ROM, it is used to recover the signer of a specific ethereum transaction. zkASM code of `ecrecover_tx` subrutine can be found here.

The following zkASM code shows how to use the `ecrecover_tx` subroutine. Once the subroutine is executed, the code will continue on the following line and the recovered address will be in register `A`:

```
0xd9eba16ed0ecae432b71fe008c98cc872bb4cc214d3220a36f365326cf807d68n => A ; Tx hash
0xddd0a7290af9526056b4e35a077b9a11b513aa0028ec6c9880948544508f3c63n => B ; r
0x265e99e47ad31bb2cab9646c504576b3abc6939a1710afc08cbf3034d73214b8n => C ; s
0x1cn => D ; v
                        :CALL(ecrecover_tx)
```

5.5.8 References

The jump-like instructions we presented earlier provide programmers with the ability to modify the execution flow by jumping to various parts of the code. Similarly, subroutines enable the invocation of other code segments located in different files, allowing for a return to the point of invocation to continue with the original flow. However, what if a programmer wants to jump to a label in a different `.zkasm` file without returning to the original code? This is where **References** come in handy.

References allow programmers to combine jump-like instructions with subroutines, which in turn allows for the jumping to a label in another `.zkasm` file without the need to return to the original code. This results in greater flexibility and modularity in code organization and execution.

To use references, the syntax is as follows:

```
:JMP(@someLabel + RR)
```

Here, `someLabel` refers to a label located in another `.zkasm` file. The instruction above jumps to the line of code that corresponds to the value stored in the register `RR` under the `someLabel` label. Additionally, it is possible to parameterize the specific line of code to

jump to by adding the value of the first limb of the register $\texttt{E}_0$ to the reference, as shown below:

```
    :JMP(@someLabel + E)
```

References can also be used for other types of jumps, including conditional jumps with an else condition. For example:

```
    :JMPN(@someLabel + RR)
    :JMPZ(@someLabel + E)
    :JMPC(@someLabel + RR, elseLabel)
    ; ...
```

Moreover, **References** are also useful even if the tag we are pointing is actually inside the same `.zkasm` file. This is because, as we have seen before, we can parametrize how many lines we ought to jump **after** some label using the registers `E` and `RR`, which we can not using only tag identifiers.

5.5.9 REPEAT

Although jumps are enough in order to build program loops, a `REPEAT` instruction has been introduced in order to easily repeat a certain line of code. The `REPEAT` instruction makes use of the `RCX` register in order to parametrize the number of times the code should be repeated. To illustrate how to use the `REPEAT` instruction, we are going to propose an example:

```
10 => A
14 => RCX
A + 2 => A  :REPEAT(RCX)

40  ^^I^^I:ASSERT
```

The previous code assigns 10 to the `A` register and 14 to the repeat counter `RCX`. After that, invokes an addition by two units of the `A` register, together with the `REPEAT` instruction with parameter `RCX`. This will make the line of code

```
A + 2 => A
```

to repeat a total amount of 15 times, the first one written explicitly in the code and the other 14 times produced by the `REPEAT` instruction. This is something the user should take into account. After that, the `A` register will contain the value

$$10 + 15 \cdot 2 = 40.$$

Hence, an `ASSERT` can be invoked against the value 40, since `A` should be $\texttt{op} = 40$.

5.6 Hash Related Instructions

Both implementations in the zkEVM of each of the hashes is exactly the same, henceforth what we explain in this section for the `KECCAK-256` hash can be applied also for the `Poseidon` hash. There are 4 instructions referent to `KECCAK-256` hashes in the zkEVM assembly language: `HASHK`, `HASHK1`, `HASHKLEN` and `HASHKDIGEST`. Each of them has a different purpose:

- `HASHK`: This instruction is in charge of consecutively keep introducing bytes into the input of the hash. Via this instruction we can introduce a maximum amount of 32 bytes at the same time.

- `HASHK1`: This instruction is does actually the same that the previous instruction but only can introduce 1 byte at the time.
- `HASHKLEN`: This instruction is the one that actually performs the hash but actually do not retrieve its digest.
- `HASHKDIGEST`: This instruction retrieves the previous hash digest performed using the `HASHKLEN` instruction.

Similarly, there are 4 instructions referent to `Poseidon` hashes in the zkEVM assembly language: `HASHP`, `HASHP1`, `HASHPLEN` and `HASHPDIGEST`. Each of them mirrors the same instruction explained in the `KECCAK-256` case.

5.6.1 `HASHK`

Since hash functions can hash an arbitrarily large amount of data but our registers are limited to 32 bytes, we need a procedure to sequentially keep introducing bytes in order to be hashed together. This is what this instruction is providing: it allows to append from 1 to 32 bytes to the current input of the hash. The following registers will be relevant in this instruction: D_0 and `HASHPOS`. The former will contain the desired bytes we want to append and the later will contain the index of the next position of the bytes array of the input of the hash that we will start to fill. That is, this register will contain the total input bytes that we have previously introduced up to this precise moment.

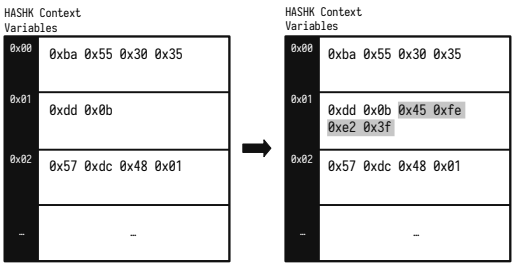

Figure 25: Schema of the `HASHK` instruction.

The typical use of the `HASHK` instruction in the zkASM language is the following one:

```
op^^I:HASHK(addr)
```

The `op` placeholder will usually be a register or a register operation like the following one:

```
A + 1^^I:HASHK(addr)
```

We can perform several hashes at the same time, each of them being stored in its corresponding address. Then, we can keep filling each of the addresses' bytes without perturbing the other ones. We can specify the address using the register `E`, which is the one used to store addresses 256-bits, or using a hard coded number. The value 0 is usually used as an address, which is reserved for storing specific hashes.

```
A + 1^^I:HASHK(E)
```

The former instruction will append the bytes of the current value of `A` + 1 into the input of the hash function we want to perform within the bytes attached to the address `E`.

To formalise what the `HASHK` instruction does, let $(\texttt{op}_0, \texttt{op}_1, \ldots, \texttt{op}_{31})$ be the byte decomposition of the `op` variable. We will denote $\texttt{trunc}_{\texttt{D}_0}(\texttt{op})$ by the byte decomposition of `op` truncated at the $\texttt{D}_0$ position. More precisely,

$$\texttt{trunc}_{\texttt{D}_0}(\texttt{op}) = (\texttt{op}_0, \texttt{op}_1, \ldots, \texttt{op}_{\texttt{D}_0 - 1}).$$

Let $\texttt{hashk}[\texttt{addr}] = (h_0, \ldots, h_{\texttt{HASHPOS}})$ be the current array of bytes that we are willing to hash at a certain address `addr`. The `HASHK` instruction will append the $\texttt{trunc}_{\texttt{D}_0}(\texttt{op})$ array into the `hashk` one, so that the next state of the (temporal) input of the hash will become

$$\texttt{hashk}[\texttt{addr}]' = (h_0, \ldots, h_{\texttt{HASHPOS}}, \texttt{op}_0, \texttt{op}_1, \ldots, \texttt{op}_{\texttt{D}_0 - 1})$$

At the end of this operation, we increase the value of the `HASHPOS` register in $\texttt{D}_0$:

$$\texttt{HASHPOS}' = \texttt{HASHPOS} + \texttt{D}_0.$$

Let us propose the following simple example: suppose that we want to hash a single byte concatenated with the first 31 bytes of a 32-bytes integer, each of them contained in the registers `A` and `B` respectively. To that we will use the address `0x03` stored in the register `E`. First of all, we should ensure that our current hash position is 0, because we are actually starting a new hash.

```
0x03 => E
0 => HASHPOS
```

At this moment

$$\texttt{hashk}[\texttt{0x03}] = \emptyset.$$

Later on, we will start adding the single byte of `A` into the hash input. Observe that we should assign the length 1 into the register `D` because we need to specify the length value in bytes when using the `HASHK` instruction.

```
1 => D
A^^I^^I^^I^^I:HASHK(E)
```

Now, we update the array

$$\texttt{hashk}[\texttt{0x03}] = (a)$$

where a denotes the current value of the register `A`. Moreover, `HASHPOS` increased in 1:

$$\texttt{HASHPOS}' = \texttt{HASHPOS} + 1.$$

Now, we do the same with the register `B`

```
31 => D
B^^I^^I^^I^^I:HASHK(E)
```

Finally, the corresponding hash array is the following

$$\texttt{hashk}[\texttt{0x03}] = (a, b_0, b_1, \ldots, b_{30})$$

where $(b_0, \ldots, b_{30}) = \texttt{trunc}_{31}(\texttt{B})$ are the first 31 bytes of the register `B`, which is actually the string we want to hash. Moreover, `HASHPOS` increased in 31:

$$\texttt{HASHPOS}' = \texttt{HASHPOS} + 31.$$

5.6.2 HASHK1

The instruction HASHK1 performs in the same way that HASHK but the register D_0 is not relevant here, because the size of the input string is always of 1 byte.

5.6.3 HASHKLEN

As commented before, this instruction is actually the one that computes the hash digest and stores it internally, to later on be acquired via the HASHKDIGEST instruction. This instruction also uses the first 32 bytes of the op intermediate value in order to specify the length within all the bytes stored in the specified address that will be hashed. Therefore, the total amount of bytes we can hash is 2^{32}.

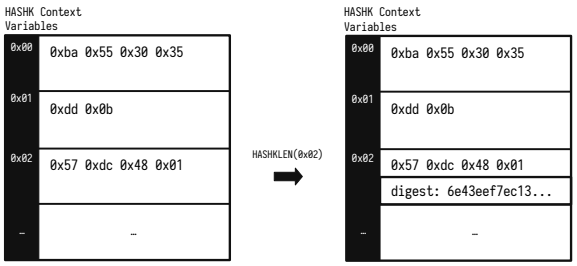

Figure 26: Schema of the HASHKLEN instruction.

Following the previous example, the line of zkASM that we need to execute in this step is the following one:

```
HASHPOS^^I:HASHKLEN(E)
```

Recall that HASHPOS value at the current state is 32 because we want to hash a total amount of 32 bytes.

More specifically, if $(h_1, \ldots, h_k)$ is the input array attached to a specific address addr (that is, hashk[addr] $= (h_1, \ldots, h_k)$ with the previous notation), the instruction

```
len^^I^^I:HASHKLEN(addr)
```

internally stores the digest

$$d = \text{KECCAK-256}(h_1, \ldots, h_k).$$

Observe that we should have that $k =$ len. Otherwise, the HASHKLEN instruction will get an error.

5.6.4 HASHKDIGEST

This usual way we are invoking this instruction is the following

```
$ => REG^^I:HASHKDIGEST(E)
```

Meaning that we are storing the digest of the hash attached to the address E into the register REG. The hash digest is introduced as a free input using the $ => operator. In our

example, if we want to assign the digest of the hash to the register `D`, we would use the following line

```
$ => D^^I^^I:HASHKDIGEST(E)
```