=nil;'s zkEVM1

A Secure Updatable Type-1 zkEVM

Alisa Cherniaeva — =nil; Foundation — a.cherniaeva@nil.foundation

Maksim Nikolaev — =nil; Foundation — maksim.n@nil.foundation

Mikhail Komarov — =nil; Foundation — nemo@nil.foundation

December 12, 2023

Abstract

zkEVMs has proven themselves to be a worthwile way to prove a more consistent (than Ethereum) database's state transitions to Ethereum to avoid bringing its own economic security to "L2" and simply "borrow" it from Ethreum. Circuits of zkEVMs, though, have been identified to possess critical security vulnerabilities because of the way they're being developed. The inherent potential weaknesses in their design and implementation raise concerns about the integrity of the overall zkRollup concept.

To address these security flaws, the transition from high-level code to Type-1 zkEVM circuits, facilitated by compiling production-used EVM (e.g. evmone) through zkLLVM circuit compiler, emerges as a promising solution. This process of compilation offers improved security and auditability to zkEVM circuits, ensuring a more robust, trustworthy and bytecode EVM-compatible Ethereum execution environment. By leveraging zkLLVM to transform high-level code into Type-1 zkEVM circuits, potential vulnerabilities are mitigated, thereby bolstering the security and reliability of these circuits.

This paper aims to highlight the significance of this transition in fortifying the security of zkEVM circuits by introducing a zkEVM compiled from high-level language-based production-used EVM implementation which incrases the auditability and through this, security of a zkEVM produced as a result.

1 Introduction

The Ethereum Virtual Machine (EVM) plays a crucial role in Ethereum by managing the deployment and execution of EVM applications. Whenever a transaction leads to the execution of an application, the EVM is instantiated, equipped with all necessary information related to the current replication packet in progress and the specific transaction. The EVM's role is to update the Ethereum state by calculating legitimate state transitions based on the execution of an EVM application code, in accordance with the specifications outlined in the Ethereum protocol.

A zero-knowledge Ethereum Virtual Machine (zkEVM) proves the execution of EVM applications in a way that's compatible with existing Ethereum infrastructure and augment it with a very small and efficiently verifiable proof that all transactions are valid and after executing all those transactions the updated state is correct.

A zkEVM is a promising solution to Ethereum's scalability problem, but building it is non-trivial problem for many reasons.

- A calculation that will be proven using SNARK must be represented as a circuit. But some EVM functions (for example, Keccak hash function) are unfriendly to be presented in this form.
- A zkEVM must be able to provide proofs for any sequence of valid transactions, so we cannot say in advance how many and what opcodes will be called. This significantly complicates the design of circuits for the proof system, requiring overhead.
- There must be a universal proof-verification algorithm, the execution of which as a contract will be relatively cheap in terms of gas.
- Circuits are difficult to maintain because their structure is very complex. They must be subject to external audit periodically to avoid errors.
- A minor update of the EVM operation logic can lead to serious difficulties in updating the zkEVM.

As of today there is no ideal solution that would cope with all technical difficulties.

This specification proposes a Type-1 zkEVM based on the general-purpose modular proof system Placeholder and the zkLLVM compiler. Our approach allows us to obtain a system that is easy to update when EVM changes and new algorithms and techniques are added to SNARK. At the same time, fine-tuning parameters and the possibility of parallelization make it possible to obtain performance sufficient for large-scale practical use.

1.1 Ethereum's State

This section contains a description of the basic components of the Ethereum network. The description is based on the [1, 2, 3].

1.1.1 Notation

$\mathbb{N}_{\mathrm{n}} = \{P : P \in \mathbb{N} \wedge P < 2^n\}$
$\mathbb{B}$ – string of bytes.
`KEC` – Keccak hash function.

1.1.2 World State

The world state $\boldsymbol{\sigma}$ is a mapping between addresses (160-bit identifiers) and account states. An Ethereum account represents an entity with a balance of ether (ETH). There are two different types of accounts in Ethereum: externally owned accounts (EOA) and contract accounts. Both account types are able to receive, store, and send ETH and tokens, as well as interact with deployed applications. EOAs are under the control of users, whereas contract accounts are governed by program code executed by the EVM. Transactions from an EOA to a contract account can activate code, leading to the execution of various actions such as token transfers or the creation of new contracts. The basic structure of a transaction is illustrated in Figure 1.

Field	Notation	Value	Description
nonce	$\boldsymbol{\sigma}[a]_{\mathrm{n}}$	$\mathbb{N}_{256}$	A counter that indicates the number of transactions sent from an EOA or the number of contracts created by a contract account.
balance	$\boldsymbol{\sigma}[a]_{\mathrm{b}}$	$\mathbb{N}_{256}$	The number of Wei owned by this address.
storageRoot	$\boldsymbol{\sigma}[a]_{\mathrm{s}}$	$\mathbb{B}_{32}$	A hash of the root node of a trie that encodes the storage contents of the account.
codeHash	$\boldsymbol{\sigma}[a]_{\mathrm{c}}$	$\mathbb{B}_{32}$	This hash corresponds to the code of an account within the EVM.

Table 1: *The structure of an account*

An account is *valid* when:

$$\mathtt{VALID}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a]_{\mathrm{n}} \in \mathbb{N}_{256} \wedge \boldsymbol{\sigma}[a]_{\mathrm{b}} \in \mathbb{N}_{256} \wedge \boldsymbol{\sigma}[a]_{\mathrm{s}} \in \mathbb{B}_{32} \wedge \boldsymbol{\sigma}[a]_{\mathrm{c}} \in \mathbb{B}_{32} = 1 \tag{1}$$

An account is *empty* when it has no code, zero nonce and zero balance:

$$\mathtt{EMPTY}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a]_{\mathrm{c}} = \mathtt{KEC}\big(()\big) \wedge \boldsymbol{\sigma}[a]_{\mathrm{n}} = 0 \wedge \boldsymbol{\sigma}[a]_{\mathrm{b}} = 0 \tag{2}$$

An account is *dead* when its account state is non-existent or empty:

$$\mathtt{DEAD}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a] = \varnothing \vee \mathtt{EMPTY}(\boldsymbol{\sigma}, a) \tag{3}$$

1.1.3 Transaction

Transactions are signed messages originated by an EOA. The transaction message's structure is RLP-serialized and contains the data, mentioned in Figure 2.

Field	Leg.	EIP-2930	EIP-1559	Not.	Value	Description
type	+	+	+	T_{x}	$\{0,1,2\}$	0 (legacy), 1 (EIP-2930) or 2 (EIP-1559)
nonce	+	+	+	T_{n}	$\mathbb{N}_{256}$	a sequentially incrementing counter which indicates the transaction number from the account
gasLimit	+	+	+	T_{g}	$\mathbb{N}_{256}$	the maximum amount of gas units that can be consumed by the transaction
to	+	+	+	T_{t}	$\mathbb{B}_{20}$ or $\mathbb{B}_{0}$	The recipient of a transaction (EOA or a contract address)
value	+	+	+	T_{v}	$\mathbb{N}_{256}$	amount of ETH to transfer from sender to recipient
r, s	+	+	+	T_{r}, T_{s}	$\mathbb{N}_{256}$	the signature of the transaction and used to determine the sender of the transaction
accessList	−	+	+	$T_{\mathbf{A}}$	$\{\mathbb{N}_{256}, \{\mathbb{N}_{256}\}_j\}_i$	List of access entries to warm up
chainId	−	+	+	T_{c}	β	the network chain ID
yParity	−	+	+	T_{y}	$\{0,1\}$	Signature Y parity
w	+	−	−	T_{w}	$\mathbb{N}_{256}$	A scalar value encoding Y parity and possibly chain ID
maxFeePerGas	−	−	+	T_{m}	$\mathbb{N}_{256}$	the maximum fee per unit of gas willing to be paid
maxPriorityFeePerGas	−	−	+	T_{f}	$\mathbb{N}_{256}$	the maximum price of the consumed gas to be included as a tip to the validator
gasPrice	+	+	−	T_{p}	$\mathbb{N}_{256}$	the number of Wei to be paid per unit of *gas* for all computation
init	+	+	+	$T_{\mathbf{i}}$	$\mathbb{B}$	the EVM-code for the account initialisation procedure
data	+	+	+	$T_{\mathbf{d}}$	$\mathbb{B}$	the input data of the message call

Table 2: *The structure of a transaction*

$$L_{\mathrm{T}}(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{x}} = 0 \\ (T_{\mathrm{c}}, T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathbf{A}}, T_{\mathrm{y}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{x}} = 1 \\ (T_{\mathrm{c}}, T_{\mathrm{n}}, T_{\mathrm{f}}, T_{\mathrm{m}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathbf{A}}, T_{\mathrm{y}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{x}} = 2 \end{cases} \tag{4}$$

where

$$\mathbf{p} \equiv \begin{cases} T_{\mathbf{i}} & \text{if } T_{\mathrm{t}} = \varnothing \\ T_{\mathbf{d}} & \text{otherwise} \end{cases} \tag{5}$$

1.1.4 EVM

The specific rules for changing Ethereum's state from replication packet to replication packet are defined by the Ethereum Virtual Machine (EVM). The EVM operates as the runtime environment for applications: when an application begins its execution, the EVM creates an execution context that includes various data structures and state variables that are outlined below.

The application code is presented as a byte array. Each array byte is an instruction opcode or an immediate operand. The program counter (pc) (initially 0) identifies the next instruction to execute. Executing an instruction consumes gas in the EVM, and this ensures that no infinite computation can occur. Gas (g) is the fuel left for future computations.

The EVM has a simple stack-based architecture. The Stack ($\mathbf{s}$) serves for storing temporary values during the execution of applications. The stack operates with a maximum of 1024 256-bit words (initially empty). These elements may include control flow information, storage addresses, and the results and parameters for application instructions.

Memory ($\mathbf{m}$) is a 256-bit addressable, contiguous dynamically sized array of bytes (initially empty). Memory is volatile and only available during the current program execution. Memory expands on-demand when a value is read or written to a given location. Values can be read from/written to memory using the instructions MLOAD, MSTORE, MLOAD8 or MSTORE8. The active number of words in memory (counting continuously from position 0) is denoted as i.

The machine state $\boldsymbol{\mu}$ is defined as the tuple $(g, pc, \mathbf{m}, i, \mathbf{s})$.

Storage is a persistent (it is retained between calls) key-value store that maps 256-bit words to 256-bit words. Storage can be read/written using the instructions SLOAD or SSTORE (which allow writing and reading 32 bytes). All locations in storage are well-defined initially as zero.

In addition to the system state $\boldsymbol{\sigma}$ and machine state $\boldsymbol{\mu}$ an execution environment also includes elements that mentioned in table 3.

Element of Execution Environment	Notation
the address of the account which owns the code that is executing	I_{a}
the sender address of the transaction that originated this execution	I_{o}
the price of gas paid by the signer of the transaction that originated this execution. This is defined as the effective gas price	I_{p}
the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data	$I_{\mathbf{d}}$
the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender	I_{s}
the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value	I_{v}
the current replication packet header	I_{H}
the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATE(2)s being executed at present)	I_{e}
the permission to make modifications to the state	I_{w}

Table 3: *The Execution Environment variables*

2 Preliminaries

This section describes the precursor set of technologies leading to zkEVM1.

2.1 Placeholder Proof System

Initiated in 2021, Placeholder [4] represents a modular proof system that incorporates a range of cryptographic primitives, including commitment schemes, lookup tables, and gate generation techniques.

In this section we focused on the Placeholder options that are relevant to zkEVM1. Overall zkEVM1 circuit is defined by fixed parameters and various sets of constraints (*Basic*, *Copy*, *Lookup*) on the *Execution trace*. The Execution trace stores values used during computations. It is represented by a rectangular matrix $\boldsymbol{\mathcal{T}}$ (which we'll refer to as *Table*) with $\mathsf{N}_{\mathsf{rows}}$ rows and $\mathsf{N}_{\mathsf{col}}$ columns:

$$\boldsymbol{\mathcal{T}} = \left[\vec{\tau}_0^T, \dots, \vec{\tau}_{\mathsf{N}_{\mathsf{col}}-1}^T\right].$$

2.1.1 Lookup Tables

The Lookup argument assumes a important role in the zkEVM construction, presenting a mechanism for sharing non-fixed records across multiple circuits with minimal overhead compared to a singular circuit approach. This technique necessitates only the incorporation of an extra circuit, specifically designed to provide evidence of the construction of the lookup table. In this way, zkEVM leverages the flexibility of the Lookup argument to efficiently manage and share dynamic data among distinct circuits, thereby enhancing the overall versatility and scalability of the construction.

Generalization Of Plookup [5] There are two components to the lookup argument, similar to the original PLONK argument: permutation and assertion check. We keep them as is. Initially, the prover rearranges $\vec{a}$ and $\vec{l}$ in a way that makes the verification of inclusion lookup queries into $\vec{l}$ a relatively straightforward task. Subsequently, the prover presents a permutation argument for the permuted columns. Finally, they demonstrate that the values from the permuted $\vec{a}$ form a subset of the values from the permuted $\vec{l}$. However, we add some flexibility to the original PLONK argument by allowing the prover to use any number lookup queries inside one lookup protocol. Moreover, the prover can compress several lookup tables inside one column of the execution trace. Vice versa, the prover can split one lookup table into several columns.

2.1.2 Gate Generation

Here, we have two approaches, TurboPlonk or IVC Plonkish. Both methodologies require the utilization of custom gates, main component in the formulation of constraints. These constraints serve as expressions for the values within a table for a specific row, potentially spanning several adjacent ones. In the context of these constraints, let $\mathbf{o}$ represent the set of offsets for the row indices involved in the constraint, typically taking the form $-1, 0, 1$. Each j-th constraint, where $0 \leq j < \mathsf{C_{bs}}$, is defined by a multivariate polynomial $\mathcal{C}'_j$ of total degree $\mathsf{C_{dg}}$ over the table values. This expression is given as:

$$\mathcal{C}'_j(\{\vec{w}_{0,i+o'}\}_{o' \in \mathbf{o}}, \ldots, \{\vec{w}_{\mathsf{N_{wt}}-1,i+o'}\}_{o' \in \mathbf{o}}) = 0, \text{ where } i \text{ -- number of row}, i \in [\mathsf{N_{rows}}] \tag{6}$$

Selectors detect inclusion/exclusion of a Basic Constraint check within a row. These selectors are integrated into the assertion process, collectively forming what is referred to as a "Gate." Each gate encompasses one or more constraints, and every row must satisfy all gates stipulated by the circuit. This robust framework ensures the integrity and compliance of each row with the specified set of constraints, contributing to the overall efficacy of the circuit.

2.1.3 Commitment Schemes

In selecting commitment schemes, we opt for a LPC and batched KZG scheme. Specifically, the LPC scheme is utilized for the initial layer, and KZG is employed for the subsequent two layers.

KZG Polynomial commitment schemes KGZ, introduced in [6], uses a triple of groups (G_1, G_2, G_3) with an efficiently computable non-degenerate bilinear pairing $e : G_1 \times G_2 \to G_3$. Let P_i be generators of G_i for $i = 1, 2, 3$. We denote $x \cdot P_i$ by $[x]_i$ for $i = 1, 2$ and any $x \in \mathbb{F}_p$. A trusted setup Gen generates srs which contains powers of a random field element $\alpha \in \mathbb{F}_p$: $(P_1, \alpha \cdot P_1, ..., \alpha^{d-1} \cdot P_1, P_2, \alpha \cdot P_2)$. The value of α must remain secret. For any polynomial $f \in \mathbb{F}_p^{<d}[X]$, $f = \sum_{i=0}^{d-1} c_i X^i$ commitment to f defined by $\mathsf{Commit}(f) = [f(\alpha)]_1$ that can be calculated using srs:

$$[f(\alpha)]_1 = \left(\sum_{i=0}^{d-1} c_i \cdot \alpha^i \right) \cdot P_1 = \sum_{i=0}^{d-1} c_i \cdot \mathsf{srs}_i.$$

To prove that $f(z) = s$, the $\mathsf{EvalProof}$ simply outputs a commitment $\pi = [h(\alpha)]_1$ to the quotient polynomial $h = (f(X) - s)/(X - z)$. A correctly generated proof will satisfy $e(\pi, [\alpha]_2 - [z]_2) = e(h(\alpha) \cdot P_1, (\alpha - z)P_2) = e((f(\alpha) - s) \cdot P_1, P_2)$. The proof is accepted by the verifier ($\mathsf{EvalVerify}$) if and only if $e([f(\alpha)]_1 - [s]_1, [1]_2) = e(\pi, [\alpha - z]_2)$. For the performance of the Placeholder, we use a version of the protocol that allows it to query multiple committed polynomials at multiple points at a time. An efficient batch version of the KZG is described in [7].

LPC We use a scheme, which is based on LPC [8], which is generalization of polynomial commitment scheme. An (ε, k)-list polynomial commitment scheme for some metric $\Delta : F[X] \times F[X] \to [0, 1]$ and all $\delta > 0$ consists of the following:

- $\mathrm{Gen}(1^\lambda) \to \mathrm{pp}$ generates public parameters,
- $\mathrm{Com} : F < d[X] \to C$ commitment c to some f,
- An IOP system (P, V) with $\varepsilon(\delta)$ soundness and $k(\delta)$ rounds of interaction for the relation $R_\delta(\mathrm{pp}) := \left\langle \left(d, N, \{z_i, y_i\}_{i=1}^N, c\right); f \right\rangle \exists g \in F < d[X], \Delta(f, g) < \delta, \forall i \in [N], g(z_i) = y_i, \mathrm{Com}(g) = c$ for which (P, V) are both provided with degree bound d, and a set of point-evaluation pairs $\{(z_i, y_i)\}_{i=1}^N$ and commitment $c \in C$, while P is also provided with a representation of $f \in F[X]$. Both P and V have access to an oracle for $\mathrm{Com}(\cdot)$.

2.1.4 WIP

Also, several approaches that can significantly increase the efficency of generating proofs for zkEVM are in the implementation stage.

Lookup Singularity Lookup techniques are actively used in the design of zkEVM. In particular, to prove data consistency in different tables given in 3. The need to commit these tables does not allow them to be large.

Lookup Singularity is the idea that we can efficiently define circuits using lookup arguments only. Lasso [9] is a lookup technique that allows the use of enormous tables 2^{128} or larger (if they are structured). In this case, the execution of each instruction is replaced by a single lookup.

IVC Executing zkEVM requires proving a sequence of "similar" operations. One promising approach for this case is the use of IVC. Incrementally-verifiable computation (IVC) is a cryptographic tool that allows for the generation of proofs verifying the correct execution of "long-running" computations:

$$
\begin{gathered}
t-\text{step computation (nondeterministic): for } \; z_0, z_t, F: \\
\exists z_1, \ldots, z_{t-1}, w_0, \ldots, w_{t-1}: \; \forall i \in \{0, \ldots, t-1\}: \; F(z_i, w_i) = z_{i+1}
\end{gathered}
$$

A number of works ([10, 11, 12]) propose a folding technique that brings an elegant way to implement IVC. The folding scheme allows you to combine several NP instances of the same type into one that is a randomized sum of them, and then folds this claim about the randomized sum.

However, the arithmetization must be sufficiently expressive to construct an efficient zkEVM, so it is preferable to use PLONKish arithmetization instead of R1CS. We rely on Protostar ([13]) in our implementation. This approach extends the PLONK relation by using d-homogenous polynomials and introducing the slack vector $\mathbf{E}$:

$$
\sum_{j=0}^{d} \mu^{d-j} \cdot f_j(\mathbf{pi}, \mathbf{w}, [r]_{i=1}^{k}) = \mathbf{E},
$$

where $\mathbf{pi}$ – public input, $\mathbf{w}$ – witness of the instance.

2.2 zkLLVM

zkLLVM ([14]) is a circuit compiler designed to translate high-level mainstream languages, such as C++ and Rust, into representations suitable for provable computation protocols, namely circuits for proof systems.

The architecture of zkLLVM offers distinct advantages over other methods of circuit development:

1. It allows users to directly compile their algorithm into circuits, without needing a custom Domain Specific Language (DSL) and duplicating source code.

2. It omits any intermediary layer, such as a specific zkVM, between the original algorithm and the resulting circuits. This absence translates to no additional overhead in the circuit size (and consequently, the proving time).

3. Due to direct access to the inner circuit representation, zkLLVM facilitates the generation of optimized low-level verifier code tailored for specific virtual machines. For example, `=nil;` utilizes the zkLLVM transpiler[1] for the EVM Placeholder verifier.

4. As an LLVM-based compiler, zkLLVM boasts compatibility with any LLVM IR-based extension. Consequently, several developments dedicated to LLVM IR have emerged in the zkLLVM ecosystem.

zkLLVM is based on LLVM framework because of several reasons:

1. Wide variety of language frontends available for LLVM which make zkLLVM capable of parsing those languages out of the box.

2. Modular architecture which allows to introduce circuit-specific adjustments to the memory model used as the closes memory model which allows to map data within the constraint table is a coherent memory model which is also widely used within various LLVM backends.

[1] https://github.com/NilFoundation/zkllvm-transpiler

3. Extensibility of an LLVM framework enables composability of various backends with various frontends or even of various backends with various backends. This means with proper backends/frontends being implemented zkLLVM is capable of achieving formal verifiability of its circuits produced with combining its proof system backend with an LLVM-based KFramework [2]) or an FHE-enabled backend.

3 Three layers of zkEVM

As mentioned earlier, the zkEVM comprises numerous circuits, each representing a distinct EVM state condition. These circuits can be aggregated through a wrapping strategy. The wrapping strategy is straightforward: all sub-circuits are collectively verified within the encompassing circuit, and their tables are shared uniformly among them. We can split this zkEVM circuit into several steps:

1. **Preprocessing** The non-fixed shared public tables are generated and padded to the chosen size. In this step, we assign values to the lookup columns in the execution table. We aim to do this in the most compact manner, utilizing rectangles in constant rows, for EVM circuit. The rows of these rectangles are determined by selectors, and the columns are based on the lookup table description. It's important to note that this is a preparation step for the prover process. The lookups tables size is restricted by the 2^{18} rows, which is the maximum number of rows that can be processed by the prover.

2. **Low-level Layer** We collect all low-level circuits proofs separately from each other along with their shared public input. In this layer, we employ a LPC to introduce flexibility in choosing the field size. The total number of rows is capped at 2^{18}, with a blowup factor of 2^3 and a maximum gate degree of 9. The total number of inner FRI rounds remains unchanged at 40. We use the general Plookup optimization across 50 distinct lookup gates.

3. **Root Layer** We verify the low-level circuits proofs and generate two new proofs for the root circuits. These root circuits are divided based on the prover's performance. Each circuit operates independently, with shared lookup tables being the only common element. Thus, this design allows for straightforward aggregation without additional complexities. The aggregation circuit utilizes a batched KZG commitment, featuring 2^{22} rows and a maximum gate degree of 9, with no lookup tables involved.

4. **Proof Layer** We avoid using lookups and generate a single proof for the whole zkEVM circuit based on verification cost. The final circuit comprises 2^{16} rows with a maximum gate degree of 9, and it does not involve any lookup tables. All properties the same as in the previous layer. The total verification cost is $500,000$ gas.

3.1 Preprocessing

The Preprocessing is responsible for generating all non-fixed shared public tables.

1. Read-Write Table validates the integrity of all random read-write access records. Here we group all records by type of the data target (*Storage, Memory, Stack, etc*).

2. Storage Table is used to check the validity of storage operations. The corresponding circuit checks the correctness of read and write operations with the Merkle Patricia Tree data structure.

3. Keccak Table is used to store the results of executing the Keccak hash function. The corresponding circuit checks the correctness of the table entries.

4. Tx Table contains transactions fields. The corresponding circuit checks the correctness of these transactions in accordance with the EVM specification.

5. Bytecode Table contains the bytecode that must be executed in the EVM. The corresponding circuit checks that the bytecode stored in the contract matches the bytes in the table.

6. Copy Table contains a list of records with data copying operations between bytecode, memory, log, etc.

[2] https://kframework.org/index.html

7. Block Table contains block's header fields. The corresponding circuit checks the hash code of these fields.

8. Withdrawal Table validates that the merkle patricia trie identified by the root withdrawalsRoot contains all the withdrawals.

9. ECDSA Table contains the results of executing the ECDSA signature scheme. The corresponding circuit checks the correctness of the table entries.

Low-Level Layer

We describe all components in detail in this section. Firstly, we describe the part that can't be generated by zkLLVM, as it uses unique selectors constraints and custom gates.

1. Generate a **RW** proof for Read-Write Table. This involves initial grouping of records based on their unique indices, followed by a sorting process dictated by the order of access, encapsulated by the special counter `ReadWriteCounter`. Thus, we use the following constraints:

   - Check grouping of records by their data storing target type.
   - Sort by address and `ReadWriteCounter` in ascending order.
   - By using selectors we check storage records for account existence and verify storage records.

   So, this part would works analougly to the PSE solution:

3.1.1 Start

- 1.0. `field_tag`, `address` and `id`, `storage_key` are 0
- 1.1. `rw counter` increases if it's not first row
- 1.2. `value` is 0
- 1.3. `initial_value` is 0
- 1.4. `state root` is the same if it's not first row

3.1.2 Memory

- 2.0. `field_tag` and `storage_key` are 0
- 2.1. `value` is 0 if first access and `READ`
- 2.2. Memory address is in 32 bits range
- 2.3. `value` is byte
- 2.4. `initial_value` is 0
- 2.5. `state root` is the same

3.1.3 Stack

- 3.0. `field_tag` and `storage_key` are 0
- 3.1. First access is WRITE
- 3.2. Stack pointer is less than 1024
- 3.3. Stack pointer increases 0 or 1 only
- 3.4. `initial_value` is 0
- 3.5. `state root` is the same

3.1.4 Storage

- 4.0. `field_tag` is 0
- 4.1. MPT lookup for last access to (address, storage_key)

3.1.5 Call Context

- 5.0. `address` and `storage_key` are 0
- 5.1. `field_tag` is in CallContextFieldTag range
- 5.2. `value` is 0 if first access and READ
- 5.3. `initial value` is 0
- 5.4. `state root` is the same

3.1.6 Account

- 6.0. `id` and `storage_key` are 0
- 6.1. MPT storage lookup for last access to (address, field_tag)

3.1.7 Tx Refund

- 7.0. `address`, `field_tag` and `storage_key` are 0
- 7.1. `state root` is the same
- 7.2. `initial_value` is 0
- 7.3. First access for a set of all keys are 0 if `READ`

3.1.8 Tx Access List Account

- 8.0. `field_tag` and `storage_key` are 0
- 8.1. `state root` is the same
- 8.2. First access for a set of all keys are 0 if `READ`

3.1.9 Tx Access List Account Storage

- 9.0. `field_tag` is 0
- 9.1. `state root` is the same
- 9.2. First access for a set of all keys are 0 if `READ`

3.1.10 Tx Log

- 10.0. `is_write` is 1
- 10.1. `state root` is the same

3.1.11 Tx Receipt

- 11.0. `address` and `storage_key` are 0
- 11.1. `field_tag` is boolean (according to EIP-658)
- 11.2. `tx_id` increases by 1 and `value` increases as well if `tx_id` changes
- 11.3. `tx_id` is 1 if it's the first row and `tx_id` is in 11 bits range
- 11.4. `state root` is the same

2. **Storage Proof** generate a proof of existence for all storage and account records. It containts verifying of Mercle Tree Patricia paths by lookups to keccak table. A Merkle Patricia Tree (MPT), also known as a Trie, is a data structure used in Ethereum to efficiently store and retrieve key-value pairs in a cryptographically secure manner. It is an extension of the traditional Merkle Tree and Patricia Trie structures. MPT circuit checks that the update of the trie state happened correctly.

   To verify the inclusion or absence of a key-value pair, you need the authentication path and the root hash of the Merkle Tree.

   The circuit checks the transition from $value$ to $value'$ at key that led to the change of trie root from $root$ to $root'$. To prove the correctness of two paths: $path : (key, value) \to root$ and $path' : (key, value') \to root'$ we have to prove that the hash of all child nodes on the path appears at the correct position of the parent node.

3. **Keccak Proof** generate a proof of calculation for all Keccak records. It contains a lot of custom gates and produces Keccak shared public input. There are two parameters. r - bitrate, which defines padding of the input and the size of a padded message chunk during *Absorbing* step. c - capacity, which defines the number of other bits that don't interact with input; this part gives more security to the hash. $b = r + c$ - bit length of the inner state of the hash. We use $b = 1600, r = 1088, c = 512$ bits as in Ethereum.
   There is used a vector of 24 round constants RC.

Algorithm 1 Keccak-f[r,c](M)

Padding:
$P = M||0x01||0x00^*$, so that $len(P)\%r = 0$; $P = P \oplus 0x80$. Absorbing:
$S[x, y] = 0$, $\forall (x, y) \in [0, 4]$.
for $P_i \in P$: where $len(P_i) = r$.
$S[x, y] = S[x, y] \oplus P[x + 5y]$, $\forall (x, y) | x + 5y < r/w$.
for $i \in [0, 24)$:
$S = Round[r + c](S, RC[i])$ Squeezing:
$Z =$ empty string
while output is requested:
$Z = Z||S$
for $i \in [0, 24)$:
$S = Round[r + c](S, RC[i])$

There is used a 5×5 matrix of cyclic shifts r.

Algorithm 2 Round[b](A, RC)

θ-step:
$C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$, $\forall x \in [0, 5)$.
$D[x] = C[x - 1] \oplus ROT(C[x + 1], 1)$, $\forall x \in [0, 5)$.
$A[x, y] = A[x, y] \oplus D[x]$, $\forall (x, y) \in [0, 5) \times [0, 5)$.
ρ/π-step:
$B[y, 2x + 3y] = ROT(A[x, y], r[x, y])$, $\forall (x, y) \in [0, 5) \times [0, 5)$.
ξ-step:
$A[x, y] = B[x, y] \oplus (\overline{B[x + 1, y]}\ AND\ B[x + 2, y])$, $\forall (x, y) \in [0, 5) \times [0, 5)$.
ι-step:
$A[0, 0] = A[0, 0] \oplus RC$.
Return A

4. **Tx Proof** generate a proof of existence for all transaction records. The transaction proof validates the signature of each transaction, ensures that the Merkle Patricia Trie identified by the root transactionsRoot includes all and only the intended transactions, and facilitates convenient access to the transaction data for the EVM proof through the transactions table.

   (a) `txSignData: bytes = rlp([nonce, gas_price, gas, to, value, data, chain_id, 0, 0])`
   (b) `txSignHash: word = keccak(txSignData)`
   (c) `sig_parity: {0, 1} = sig_v - 35 - chain_id / 2`
   (d) `ecdsa_recover(txSignHash, sig_parity, sig_r, sig_s) = pubKey` or equivalently `verify(txSignHash, sig_r, sig_s, pubKey) = true`
   (e) `fromAddress = keccak(pubKey)[-20:]`

5. **Bytecode Proof** generate a proof of existence for all bytecode records.

3.1.12 Constraints for $i = \texttt{first}$ or $i = \texttt{last}$

$$w[i][\texttt{"tag"}] = \texttt{"Header"}$$

3.1.13 Constraints for $(w[i][\texttt{"tag"}] = \texttt{"Header"}) \wedge (i \neq \texttt{last})$

$$\begin{aligned}
&w[i][\texttt{"index"}] = 0 \\
&w[i][\texttt{"value"}] = w[i][\texttt{"length"}]
\end{aligned}$$

3.1.14 Constraints for $(w[i][\texttt{"tag"}] = \texttt{"Byte"}) \wedge (i \neq \texttt{last})$

$$\begin{aligned}
&\texttt{push_data_size_table_lookup}(w[i][\texttt{"value"}], w[i][\texttt{"push_data_size"}]) \\
&w[i][\texttt{"is_code"}] = (w[i][\texttt{"push_data_left"}] = 0)
\end{aligned}$$

3.1.15 Constraints for $(w[i][\texttt{"tag"}] = \texttt{"Header"}) \wedge (w[i+1][\texttt{"tag"}] = \texttt{"Header"}) \wedge (i \neq \texttt{last})$

$$\begin{aligned}
&w[i][\texttt{"length"}] = 0 \\
&w[i][\texttt{"hash"}] = \texttt{EMPTY_HASH}
\end{aligned}$$

3.1.16 Constraints for $(w[i][\texttt{"tag"}] = \texttt{"Header"}) \wedge (w[i+1][\texttt{"tag"}] = \texttt{"Byte"}) \wedge (i \neq \texttt{last})$

$$\begin{aligned}
&w[i+1][\texttt{"length"}] = w[i][\texttt{"length"}] \\
&w[i+1][\texttt{"index"}] = 0 \\
&w[i+1][\texttt{"is_code"}] = 1 \\
&w[i+1][\texttt{"hash"}] = w[i][\texttt{"hash"}] \\
&w[i+1][\texttt{"value_rlc"}] = w[i+1][\texttt{"value"}]
\end{aligned}$$

3.1.17 Constraints for $(w[i][\texttt{"tag"}] = \texttt{"Byte"}) \wedge (w[i+1][\texttt{"tag"}] = \texttt{"Byte"}) \wedge (i \neq \texttt{last})$

$$\begin{aligned}
&w[i+1][\texttt{"length"}] = w[i][\texttt{"length"}] \\
&w[i+1][\texttt{"index"}] = w[i][\texttt{"index"}] + 1 \\
&w[i+1][\texttt{"hash"}] = w[i][\texttt{"hash"}] \\
&w[i+1][\texttt{"value_rlc"}] = w[i][\texttt{"value_rlc"}] * \texttt{randomness} + w[i+1][\texttt{"value"}] \\
&(w[i][\texttt{"is_code"}] = 0) \vee (w[i+1][\texttt{"push_data_left"}] = w[i][\texttt{"push_data_size"}]) \\
&(w[i][\texttt{"is_code"}] = 1) \vee (w[i+1][\texttt{"push_data_left"}] = w[i][\texttt{"push_data_left"}] - 1)
\end{aligned}$$

3.1.18 Constraints for $(w[i][\texttt{"tag"}] = \texttt{"Byte"}) \wedge (w[i+1][\texttt{"tag"}] = \texttt{"Header"}) \wedge (i \neq \texttt{last})$

$$\begin{aligned}
&w[i][\texttt{"index"}] + 1 = w[i][\texttt{"length"}] \\
&\texttt{keccak256_table_lookup}(w[i][\texttt{"hash"}], w[i][\texttt{"length"}], w[i][\texttt{"value_rlc"}])
\end{aligned}$$

3.1.19 Constraints for $i = \texttt{last}$

$$
\begin{aligned}
&w[i][\texttt{"length"}] = 0\\
&w[i][\texttt{"hash"}] = \texttt{EMPTY\_HASH}
\end{aligned}
$$

6. **Copy Proof** generate a proof of existence for all copy records.

3.1.20 Common constraints

$$
\begin{aligned}
&w[i][\texttt{"is\_first"}] \in \{0, 1\}\\
&w[i][\texttt{"is\_last"}] \in \{0, 1\}\\
&(w[i][\texttt{"q\_step"}] = 0) \Rightarrow (w[i][\texttt{"is\_first"}] = 0)\\
&(w[i][\texttt{"q\_step"}] = 1) \Rightarrow (w[i][\texttt{"is\_last"}] = 0)\\
&rw\_diff = (w[i][\texttt{"tag"}] = \texttt{"Memory"}) \vee (w[i][\texttt{"TxLog"}] = 0 \wedge w[i][\texttt{"Padding"}] = 0)\\
&(w[i][\texttt{"is\_last"}] = 0) \Rightarrow w[i+1][\texttt{"rw\_counter"}] = w[i][\texttt{"rw\_counter"}] + rw\_diff\\
&(w[i][\texttt{"is\_last"}] = 0) \Rightarrow w[i+1][\texttt{"rw\_inc\_left"}] = w[i][\texttt{"rwc\_inc\_left"}] - rw\_diff\\
&w[i][\texttt{"rlc\_acc"}] = w[i+1][\texttt{"rlc\_acc"}]\\
&(w[i][\texttt{"is\_last"}] = 1) \Rightarrow w[i][\texttt{"rwc\_inc\_left"}] = rw\_diff\\
&(w[i][\texttt{"is\_last"}] = 1 \wedge w[i][\texttt{"is\_rlc\_acc"}] = 1) \Rightarrow w[i][\texttt{"rlc\_acc"}] = i
\end{aligned}
$$

3.1.21 Transition constraints for all rows except the last two rows

$$
\begin{aligned}
&w[i][\texttt{"id"}] = w[i+2][\texttt{"id"}]\\
&w[i][\texttt{"tag"}] = w[i+2][\texttt{"tag"}]\\
&w[i][\texttt{"src\_addr\_end"}] = w[i+2][\texttt{"src\_addr\_end"}]\\
&w[i][\texttt{"addr"}] + 1 = w[i+2][\texttt{"addr"}]
\end{aligned}
$$

3.1.22 Constraints for $q\_step = 1$

$$
\begin{aligned}
&\texttt{lookup}\,(Type, Type[1])\\
&(w[i][\texttt{"is\_last"}] = 0) \Rightarrow (w[i+1][\texttt{"bytes\_left"}] = w[i][\texttt{"bytes\_left"}] - 1)\\
&(w[i][\texttt{"is\_rlc\_acc"}] = 0) \Rightarrow (w[i][\texttt{"value0"}] = w[i][\texttt{"value1"}])\\
&(w[i][\texttt{"is\_rlc\_acc"}] = 1 \wedge w[i][\texttt{"is\_first"}] = 1) \Rightarrow (w[i][\texttt{"value0"}] = w[i][\texttt{"value1"}])\\
&(w[i][\texttt{"Padding"}] = 1) \Rightarrow w[i][\texttt{"Value"}] = 0\\
&(w[i][\texttt{"addr"}] \geq w[i][\texttt{"src\_addr\_end"}]) \Rightarrow w[i][\texttt{"Padding"}] = 1
\end{aligned}
$$

3.1.23 Constraints for $q\_step = 0$

$$
\begin{aligned}
&(w[i][\texttt{"q\_step"}] = 0 \wedge w[i][\texttt{"is\_rlc\_acc"}] = 0 \wedge w[i][\texttt{"is\_last"}] = 0) \Rightarrow\\
&w[i+2][\texttt{"Value"}] = w[i][\texttt{"Value"}] * r + w[i+1][\texttt{"Value"}]
\end{aligned}
$$

7. **Block Proof**
The proof is used to verify the hash code of the block header values. Namely, the following pieces of information are fed to the input of the hash function.

- `parentHash`
- `ommersHash`
- `beneficiary`
- `stateRoot`
- `transactionsRoot`
- `receiptsRoot`
- `logsBloom`
- `difficulty`
- `number`
- `gasLimit`
- `gasUsed`
- `timestamp`
- `extraData`
- `mixHash`
- `nonce`
- `baseFeePerGas`

It also serves as a lookup table for the higher level circuit to access header fields.

8. **Withdrawal Proof** generate a proof of correctness for all withdrawal records. Namely, the circuit verifies the followings:

   - `withdrawalsData: bytes = rlp([withdrawal_index, validator_index, address, amount])`
   - `withdrawalDataHash: word = keccak(withdrawalsData)`
   - `withdrawalsRoot: word = mpt(withdrawalDataHash)`
   - `withdrawal_index`, `validator_index` and `amount` are all uint64 values.
   - `amount_wei = amount * 1e9` and increases validator's balance by `amount_wei`

   Also there are some general constraints:

   - `WithdrawalID` is increased monotonically and sequentially for each withdrawal.
   - MPT root is used to lookup MPT table.

9. **Public Inputs Proof** consolidates all the data that is used to generate the final proof. For the array of this data, the Kechchak hash function is computed. The result should match the public input of zkEVM. For such a check, the Keccak table is used, described in section 3. Additionally, the proof verifies that the values in other tables (data in transaction table, state_root, etc) were taken from the correct sections of the input data.

10. **ECDSA Proof** serves to verify the correct execution of the ECDSA scheme operations. Namely, Given a signature $(T_\mathrm{r}, T_\mathrm{s})$, a message hash, and a secp256k1 public key Q, it checks that

$$T'_\mathrm{r} = T_\mathrm{r},$$

where

$$T'_\mathrm{r} = u_1 \cdot P + u_2 \cdot Q,$$
$$u_1 = (e \cdot w) \mod n,$$
$$u_2 = (r \cdot w) \mod n,$$

P – base point of elliptic curve, $w = T_\mathrm{s}^{-1}$.

The basis of the test is to prove operations on an elliptic curve. More specifically, check constraints are used for the following operations:

- Addition The gates uses basic group law formulae. Let $P = (x_1, y_1), Q = (x_2, y_2), R = (x_3, y_3)$ and $R = P + Q$. Then:
  - $(x_2 - x_1) \cdot s = y_2 - y_1$
  - $s^2 = x_1 + x_2 + x_3$
  - $y_3 = s \cdot (x_1 - x_3) - y_1$

  For point doubling $R = P + P = 2P$:

- $2s \cdot y_1 = 3x_1^2$
- $s^2 = 2x_1 + x_3$
- $y_3 = s \cdot (x_1 - x_3) - y_1$

- Scalar multiplication
  - $P = [2]T$
  - for i from $n-1$ to 0:
    (a) $Q = k_i \,?\, T : -T$
    (b) $P = P + Q + P$

Some of components can be generated by zkLLVM, as they use already existing primitives.

EVM Proof generate a real proof of instructions executions within the Ethereum Virtual Machine (EVM). We make the assumption of a constrained assignments table, which is produced by a compiler based on gas limitations. The table is constrained by the total number of blocks, where each block represents the execution of an opcode.

The construction of each replication packet involves the following components:

1. The first part encompasses data linked to a contract: `codehash, gas, root, stack pointer, and the number of operations`.
2. The second part consists of selector equations used to choose the appropriate constraint system.
3. The third part denotes the maximal total number of rows required for the larger constraint system.

zkLLVM is responsible for generating the second and third regions for a block based on selected sets of instructions. Consequently, zkLLVM possesses the capability to replicate not only zkEVM but any zkVM, provided that it has a sufficient set of primitives for the desired virtual machine.

4 Root Layer

Here we describe the second layer of the zkEVM, which is responsible for aggregating all the sub-circuits and tables together.

1. It verifies EVM Proof, State Proof, MPT Proof, Keccak Proof and Tx Proof together in the first proof.
2. It verifies Bytecode Proof, Copy Proof, Block Proof, PublicInputs Proof, Withdrawal Proof and ECDSA proof together in the second proof.

The second layer circuit is composed of the following primitive circuits:

1. **The arithmetization of the copy constraints**

Algorithm 3 Permutation Argument Verification

(a) $\beta_1, \gamma_1 = \texttt{transcript}.\textbf{get\_challenge}()$

(b) $\texttt{transcript}.\textbf{append}(V_{P,\texttt{comm}})$,

(c) Denote :

$$g_{\texttt{perm}}(y) \coloneqq \prod_{i=0}^{N_{\texttt{perm}}+N_{PI}-1}(f_i(y) + \beta \cdot S_{id_i}(y) + \gamma)$$
$$h_{\texttt{perm}}(y) \coloneqq \prod_{i=0}^{N_{\texttt{perm}}+N_{PI}-1}(f_i(y) + \beta \cdot S_{\sigma_i}(y) + \gamma)$$

(d) Calculate:

$$F_0(y) = L_0(y)(1 - V_P(y))$$
$$F_1(y) = (1 - (q_{\texttt{last}}(y) + q_{\texttt{blind}}(y))) \cdot (V_P(\omega y) \cdot h_{\texttt{perm}}(y) - V_P(y) \cdot g_{\texttt{perm}}(y))$$
$$F_2(y) = q_{\texttt{last}}(y) \cdot (V_P(y)^2 - V_P(y))$$

The values $f_i(y), S_{id_i}(y), S_{\sigma_i}(y), V_P(y), L_0(y), q_{\texttt{last}}(y), q_{\texttt{blind}}(y), V_P(\omega y)$ are input to circuit. The part of permutation argument circuit for calculating $g_{\texttt{perm}}(y)$ and $h_{\texttt{perm}}(y)$ has $\lceil \frac{(N_{\texttt{perm}}+N_{PI}-1)}{6} \rceil \cdot 2$ rows. Each row has a following construction:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	β	γ	$f_i(y)$	$f_{i+1}(y)$	$f_{i+2}(y)$	$f_{i+3}(y)$	$f_{i+4}(y)$	$f_{i+5}(y)$	$S_i(y)$	$S_{i+1}(y)$	$S_{i+2}(y)$	$S_{i+3}(y)$	$S_{i+4}(y)$	$S_{i+5}(y)$	acc_{perm}

The S_i is $S_{id_i}(y)$ or $S_{\sigma_i}(y)$ and acc_{perm} is a product of previous acc_{perm} and $((f_j(y) + \beta \cdot S_j(y) + \gamma))$ for $j \in \{i, ..., i+5\}$. The acc_{perm} equal to 1 in the first row. All unused cells for S_i have to be equal to 1 as well. If we arrange the cells in such a way that the calculation of $g_{\texttt{perm}}(y)$ would be below and the calculation of $h_{\texttt{perm}}(y)$ is above then we can easily construct the row for calculation $F_0(y), F_1(y) and F_2(y)$:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	$F_0(y)$	$F_1(y)$	$F_2(y)$	$V_P(y)$	$L_0(y)$	$q_{\texttt{last}}(y)$	$q_{\texttt{blind}}(y)$	$V_P(\omega y)$	$g_{\texttt{perm}}(y)$	$h_{\texttt{perm}}(y)$	–	–	–	–	–

2. **Arithmetization of the lookup constraints**

Algorithm 4 Lookup Argument Verification

(a) $\theta = \texttt{transcript}.\textbf{get\_challenge}()$

(b) $\texttt{transcript}.\textbf{append}(A_{\texttt{perm,comm}})$, $\texttt{transcript}.\textbf{append}(S_{\texttt{perm,comm}})$, $\texttt{transcript}.\textbf{append}(V_{L,\texttt{comm}})$

(c) For $i = 0, \ldots, N_{\texttt{lookup}} - 1$:

   i. $\texttt{lookup\_gate}_i(y) := q_{l_i}(y) \cdot (\theta^{\nu_i} A_{0_i}(\omega^{d_{0_i}} y) + \cdots + \theta^{k_i - 1 + \nu_i} A_{k_i - 1}(\omega^{d_{k_i - 1}} y))$

   ii. $\texttt{table\_value}_i(y) := q_{l_i}(y) \cdot (\theta^{\nu_i} S_{0_i}(y) + \cdots + \theta^{k_i - 1 + \nu_i} S_{k_i - 1}(y))$

(d) Construct the input lookup compression and table compression:

$$A_{\texttt{compr}}(y) := \textstyle\sum_{0 \le i < N_{\texttt{lookup}}} \texttt{lookup\_gate}_i(y)$$
$$S_{\texttt{compr}}(y) := \textstyle\sum_{0 \le i < N_{\texttt{lookup}}} \texttt{table\_value}_i(y)$$

(e) $\beta, \gamma = \texttt{transcript}.\textbf{get\_challenge}()$

(f) Denote :

$$g_L(y) = (A_{\texttt{compr}}(y) + \beta) \cdot (S_{\texttt{compr}}(y) + \gamma)$$
$$h_L(y) = (A_{\texttt{perm}}(y) + \beta) \cdot (S_{\texttt{perm}}(y) + \gamma)$$

(g) Calculate:

$$F_3(y) = L_0(y)(1 - V_L(y))$$
$$F_4(y) = (1 - (q_{\texttt{last}}(y) + q_{\texttt{blind}}(y))) \cdot (V_L(\omega y) \cdot h_L(y) - V_L(y) \cdot g_L(y)$$
$$F_5(y) = q_{\texttt{last}}(y) \cdot (V_L(y)^2 - V_L(y))$$
$$F_6(y) = L_0(y)(A_{\texttt{perm}}(y) - S_{\texttt{perm}}(y))$$
$$F_7(y) = (1 - (q_{\texttt{last}}(y) + q_{\texttt{blind}}(y))) \cdot (A_{\texttt{perm}}(y) - S_{\texttt{perm}}(y)) \cdot (A_{\texttt{perm}}(y) - A_{\texttt{perm}}(\omega^{-1} y))$$

The values $q_{l_i}(y), A_r(\omega^{d_r} y), S_r(y), V_L(y), L_0(y), q_{\texttt{last}}(y), q_{\texttt{blind}}(y), V_L(\omega y), A_{\texttt{perm}}(y), S_{\texttt{perm}}(y), A_{\texttt{perm}}(\omega^{-1} y)$ are input to circuit. The part of lookup argument circuit for calculating $A_{\texttt{compr}}(y)$ and $S_{\texttt{compr}}(y)$ has $\lceil \frac{\sum_{i=0}^{N_{\texttt{lookup}}-1} k_i}{4} \rceil$ rows. Each row has a following construction:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	Θ	$A_r(y)$	$S_r(y)$	$q_{l_i}(y)$	$A_{r+1}(y)$	$S_{r+1}(y)$	$q_{l_i}(y)$	$A_{r+2}(y)$	$S_{r+2}(y)$	$q_{l_i}(y)$	$A_{r+3}(y)$	$S_{r+3}(y)$	$q_{l_i}(y)$	$A_{\texttt{compr}}(y)$	$S_{\texttt{compr}}(y)$

The $A_r(\omega^{d_r}y)$ and $S_r(y)$, where $r \in \{o_i, .., k_i - 1\}$. The values $A_{\texttt{compr}}(y), S_{\texttt{compr}}(y)$ are a sum of previous $A_{\texttt{compr}}(y), S_{\texttt{compr}}(y)$ and $q_{l_i}(y) \cdot \theta^{\nu_i} A_{0_i}(\omega^{d_{0_i}}y) + \dots + q_{l_i}(y) \cdot \theta^{k_i-1+\nu_i} A_{k_i-1}(\omega^{d_{k_i-1}}y))$ for $j \in \{i, ..., i+3\}$. The acc_{perm} equal to 0 in the first row. All unused cells for A_r and S_r have to be equal to 0 as well. Now we can easily construct the last row for calculation $F_3(y), F_4(y), F_5(y), F_6(y) and F_7(y)$:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	$F_3(y)$	$F_4(y)$	$F_5(y)$	$V_P(y)$	$L_0(y)$	$q_{\texttt{last}}(y)$	$q_{\texttt{blind}}(y)$	$V_P(\omega y)$	$A_{\texttt{perm}}(y)$	$S_{\texttt{perm}}(y)$	$A_{\texttt{compr}}(y)$	$S_{\texttt{compr}}(y)$	$F_6(y)$	$F_7(y)$	$A_{\texttt{perm}}(\omega^{-1}y)$

3. **Arithmetization of the custom gates**

Algorithm 5 Quotient Polynomial Check

(a) $Z(y) = y^{N_{\text{rows}}} - 1$

(b) $T(y) = T_0(y) + y^d T_1 + \dots + y^{total_d eg - d * N_T} T_{N_T}(y)$

(c) $\sum_{i=0}^{9} \alpha_i F_i(y) = Z(y)T(y)$

The values $y, T_0(y), \dots, T_{N_T}(y)$ are input to circuit. The first step requires $\lceil \frac{\log_4(N_{\text{rows}})}{14} \rceil$ rows. Each row has a following construction:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	acc	$acc \cdot y$	$acc \cdot y^2$	$acc \cdot y^3$	$acc \cdot y^4$	$acc \cdot y^5$	$acc \cdot y^6$	$acc \cdot y^7$	$acc \cdot y^8$	$acc \cdot y^9$	$acc \cdot y^{10}$	$acc \cdot y^{11}$	$acc \cdot y^{12}$	$acc \cdot y^{13}$	$acc \cdot y^{14}$

The acc equal to 0 in the first row. We suppose that in the general case one row is sufficient.

The second step requires $exponentiation_circuit.rows \cdot (N_T - 1) + \lceil \frac{N_T * 2 - 1}{12} \rceil$ rows.

4. **Exponentiation Circuit** The last rows have a following contruction:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	y'	$T_i(y)$	y'	$T_{i+1}(y)$	y'	$T_{i+2}(y)$	y'	$T_{i+3}(y)$	y'	$T_{i+3}(y)$	y'	$T_{i+4}(y)$	acc	$next_acc$	$-$

The values $next_acc$ are a sum of acc and $y' \cdot T_i(y) + \dots + y' \cdot T_{i+4}(y)$. The acc equal to 0 in the first row and $next_acc$ from previous row for the next row. All unused cells for y' and $T_i(y)$ have to be equal to 0. Now the value $next - acc$ in the last row is $T(y)$.

The third step requires one row:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
j	$Z(y)$	$T(y)$	α	α^6	$F_0(y)$	$F_1(y)$	$F_2(y)$	$F_3(y)$	$F_4(y)$	$F_5(y)$	$F_6(y)$	$F_7(y)$	$F_8(y)$	$--$	$-$

5. **Arithmetization of the commitment scheme**
The following algorithm is a binary-expansion version of the Miller loop.

Algorithm 6 The Miller Loop

Input$t = \sum_{i=0}^{L} c_i 2^i$, $c_i \in \{0,1\}$, $c_L = 1$; $P \in E(\mathbb{F}_p)$; $Q \in E'(\mathbb{F}_{p^2})$ Output$f \in \mathbb{F}_{p^{12}}$
$f \longleftarrow 1$ $T \longleftarrow Q$ For$i \longleftarrow L-1$ To 0 $f \longleftarrow f^2 \cdot \mathit{LineFunction}(T,T,P)$ $T \longleftarrow T+T$ If$c_i = 1$ $f \longleftarrow f \cdot \mathit{LineFunction}(T,Q,P)$ $T \longleftarrow T+Q$

The part of Miller loop that manipulates points from the curve $E'(\mathbb{F}_{p^2})$ is actually identical to the computation of the scalar product $[t]Q$ (or $[-t]Q$ since we ignore the sign) by means of a double-and-add process.

Algorithm 7 `LineFunction`

Input$Q_1 = (x_1, y_1) \in (F_{p^2})^2$, $Q_2 = (x_2, y_2) \in (F_{p^2})^2$, $P = (x,y) \in (\mathbb{F}_p)^2$ Output$f' \in \mathbb{F}_{p^{12}}$
tcpUntwist Q_1, Q_2 $Q_1 \longleftarrow (x_1/v, y_1/(wv))$ $Q_2 \longleftarrow (x_2/v, y_2/(wv))$ tcpNow $Q_1, Q_2 \in (\mathbb{F}_{p^{12}})^2$
eIf$Q_1 = Q_2$ $l \longleftarrow \frac{3x_1^2}{2y_1}$ $f' \longleftarrow l(x-x_1)+y_1-y$ eIf$x_1 = x_2$ **and** $y_1 = -y_2$ $f' \longleftarrow x - x_1$ $l \longleftarrow (y_2-y_1)/(x_2-x_1)$ $f' \longleftarrow l(x-x_1)+y_1-y$

Table 4: *Final Exponentiation circuit outline, part 1*

f	Input, result of the Miller loop.
f^{-1}	Gates centered on this row assure the computation of f^{-1} and f^{p^6} which are basically unary operations
f^{p^6}	in $\mathbb{F}_{p^{12}}$.
$f' = f^{p^6-1}$	Computed by a multiplication gate centered on the previous row.
$(f')^{p^2}$	Computed by a unary operation gate centered on the previous row.
$f'' = (f')^{p^2+1}$	Computed by a multiplication gate centered on the previous row.
$\vdots$ $(f'')^{(1-t)/3}$	A block of gates for raising to power $(1-t)/3$.
$\vdots$ $((f'')^{(1-t)/3})^{-t}$	A block of gates for raising to power $-t$.
$(f'')^{(t-1)/3}$	Assured by copy constraints.
$g = ((f'')^{(1-t)/3})^{1-t}$	Computed by a multiplication gate centered on previous row.

6. **Arithmetization of the transcript**
Therefore, each permutation state is represented by 3 elements and each row contains 5 states.
Denote i-th permutation state by $T_i = (T_{i,0}, T_{i,1}, T_{i,2})$.

Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	$T_{0,0}$	$T_{0,1}$	$T_{0,2}$	$T_{1,0}$	$T_{1,1}$	$T_{1,2}$	$T_{2,0}$	$T_{2,1}$	$T_{2,2}$	$T_{3,0}$	$T_{3,1}$	$T_{3,2}$	$T_{4,0}$	$T_{4,1}$	$T_{4,2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i+10$	$T_{50,0}$	$T_{50,1}$	$T_{50,2}$	$T_{51,0}$	$T_{51,1}$	$T_{51,2}$	$T_{52,0}$	$T_{52,1}$	$T_{52,2}$	$T_{53,0}$	$T_{53,1}$	$T_{53,2}$	$T_{54,0}$	$T_{54,1}$	$T_{54,2}$
$i+11$	$T_{55,0}$	$T_{55,1}$	$T_{55,2}$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

State change constraints:

$$\mathtt{STATE}(i+1) = \mathtt{STATE}(i)^{\alpha} \cdot \mathtt{MDS} + \mathtt{RC}$$

Denote the index of the first state in the row by `start` (e.g. `start` $= 50$ for 10-th row). We can expand the previous formula to:

Table 5: *Final Exponentiation circuit outline, part 2*

g	The last row of the previous part of the circuit.
g^{-1}	Gates centered on this row assure the computation of
g^{p^3}	g^{-1} and g^{p^3}.
g	Assured by copy constraints.
$\vdots$ g^{-t}	A block of gates for raising to power $-t$.
$\vdots$ g^{t^2}	A block of gates for raising to power $-t$.
g^{-1}	Assured by copy constraints.
g^{t^2-1}	Computed by a multiplication gate centered on previous row.
$\vdots$ $g^{-t(t^2-1)}$	A block of gates for raising to power $-t$.
$g^{t(t^2-1)}$	Computed by inversion gate centered on this row.
f''	Assured by copy constraints.
$f''g^{t(t^2-1)}$	Computed by a multiplication gate centered on previous row.
g^{p^3}	Assured by copy constraints.
$f''g^{p^3}g^{t(t^2-1)}$	Computed by a multiplication gate centered on previous row.
g^{t^2-1}	Assured by copy constraints.
$(g^{t^2-1})^{p^2}$	Computed by a unary operation gate centered on previous row.
$f''g^{p^3}g^{t(t^2-1)}$	Assured by copy constraints.
$f''g^{p^3}(g^{t^2-1})^{p^2}g^{t(t^2-1)}$	Computed by a multiplication gate centered on previous row.
g^{-t}	Assured by copy constraints.
g^{t}	Computed by inversion gate centered on previous row.
$(g^{t})^{p}$	Computed by a unary operation gate centered on previous row.
$f''g^{p^3}(g^{t^2-1})^{p^2}g^{t(t^2-1)}$	Assured by copy constraints.
$f''g^{p^3}(g^{t^2-1})^{p^2}(g^{t})^{p}g^{t(t^2-1)}$	Computed by a multiplication gate centered on previous row.

- For i from `start` to $\texttt{start} + 5$:
  - $T_{i+1,0} = T_{i,0}^7 \cdot \texttt{MDS}[0][0] + T_{i,1}^7 \cdot \texttt{MDS}[0][1] + T_{i,2}^7 \cdot \texttt{MDS}[0][2] + \texttt{RC}_{i+1,0}$
  - $T_{i+1,1} = T_{i,0}^7 \cdot \texttt{MDS}[1][0] + T_{i,1}^7 \cdot \texttt{MDS}[1][1] + T_{i,2}^7 \cdot \texttt{MDS}[1][2] + \texttt{RC}_{i+1,1}$
  - $T_{i+1,2} = T_{i,0}^7 \cdot \texttt{MDS}[2][0] + T_{i,1}^7 \cdot \texttt{MDS}[2][1] + T_{i,2}^7 \cdot \texttt{MDS}[2][2] + \texttt{RC}_{i+1,2}$

Notice that the constraints above include the state from the next row ($\texttt{start} + 5$).

4.1 Proof Layer

The third layer is constructed using the same primitive components as the second layer. The key distinction lies in utilizing the second layer as input for the third layer. As a result, the final proof is characterized by a reduced verification cost, facilitated by employing a Keccak for transcript and employing grinding techniques.

5 Opcodes Gates for zkLLVM Low-Level Circuit

We refrain from creating subcircuits for individual opcodes. Instead, we leverage the flexibility of the gates technique within zkLLVM to map the logic of opcodes onto the EVM circuit. This mapping is accomplished by employing general non-native arithmetics, allowing for the integration of finite fields and 256 bits aritmetic. Our non-native approach encompasses support for all primitive operations, and even more complex functionalities can be expressed using this comprehensive set. Notably, memory-related opcodes are handled akin to the PSE solution, utilizing lookup constraints. The order of lookup constraints and the placement of corresponding cells in the execution trace are managed by zkLLVM.

Flexible Gates The current section gives a brief description of the Flexible Gate technique. This is one of the optimizations of the underlying Placeholder proof system. The technique allows you to modify gates and individual gate constraints to provide greater circuit "density". Since some zkEVM circuits are generated using the zkLLVM compiler, we must optimize its behavior so that there are as few free zones in the execution trace as possible. To solve this problem, the compiler, based on the proof system parameters (number of columns, maximum gate degree, etc.), can combine gates for optimization.

Let $\{\mathcal{F}_w^d\}$ be a trace constraint, which can be expressed in polynomial, permutation and lookup form such as

$$\mathcal{F}_w^d(C_{i,j}, S_i) = 0, \; C_{i,j} \in \mathbb{F}, S_i \in \{0, 1\}.$$

Let M be a metric function, which take as input a set of the primitives $\{P\}$ and return $\{\mathcal{F}_w^d\}$ as output.

Thus, Flexible circuits technique has a following algorithm:

1. zkLLVM: computational sequence $\rightarrow \{P\}$.
2. M: $\{R\} \rightarrow \{\mathcal{F}_w^d\}$.
3. Evaluate each primitive P_k for parameters w_k, d_k, i, j_k.
4. Combine all evaluations from previous step in one circuit.

General Non-native Arithmetics Now we present a general mechanism for working with non-native arithmetics. This approach is based on the Chinese Remainder Theorem (CRT). This theorem asserts that we can calculate an equation modulo two prime numbers and be confident that it holds for the multiplication of these numbers.

Let $\mathbb{F}_n$ be an non-native field, where n is a some power of two. In order to provide computations over non-native $\mathbb{F}_n$ we use constraints over native field $\mathbb{F}_k$. Without loss of generality, let $k < n$ be a prime number. We can always find such a k that meets these requirements. Additionally, we compute an integer t, such that $2^t \cdot k \geq n^2 + n$. Now, we want to check equality:

$$a \cdot b = n \cdot q + r, r = a \cdot b \mod n$$

Each positive integer a, b, q, r is divided into N limbs, where the sizes of limbs are 20 bits respectively, where a chunk $bits_{last} < 20$ is the least significant bits. To check that a, b, q and r are less than p, we use range proofs. For this purpose, a lookup table with one column is used. The first column contains all integers in the range $[0, 2^{20})$.

1. The limbs $a_0, a_1, ..., a_{N-1}$ are range-constrained by the lookup table.
2. The value $a_{N-1} \cdot 2^{20-bits_{last}}$ are range-constrained by the lookup table.

Then we constrain the equation modulo n and 2^t as follows:

1. $(a \cdot b) \mod k = (p \cdot q + r) \mod k$
2. The new limbs for a, b, q, and r are constructed in such a way that they do not exceed $1/4\ t$. Let $a'_0, a'_1, a'_2, a'_3, b'_0, b'_1, b'_2, b'_3, q'_0, q'_1, q'_2, q'_3, r'_0, r'_1, r'_2, r'_3$ be the new limbs.
3. Let p' be $-p \mod 2^t$. The limbs p'_0, p'_1, p'_2 and p'_3 are circuits parameters.
4. Compute the following limbs:
   (a) $t_0 = a'_0 \cdot b'_0 + p'_0 \cdot q'_0$
   (b) $t_1 = a'_1 \cdot b'_0 + a'_0 \cdot b'_1 + p'_0 \cdot q'_1 + p'_1 \cdot q'_0$
   (c) $t_2 = a'_2 \cdot b'_0 + a'_0 \cdot b'_2 + a'_1 \cdot b'_1 + p'_0 \cdot q'_2 + p'_2 \cdot q'_0 + p'_1 \cdot q'_1$
   (d) $t_3 = a'_3 \cdot b'_0 + a'_0 \cdot b'_3 + a'_1 \cdot b'_2 + a'_2 \cdot b'_1 + p'_0 \cdot q'_3 + p'_3 \cdot q'_0 + p'_1 \cdot q'_2 + p'_2 \cdot q'_1$
   (e) $t_4 = a'_3 \cdot b'_1 + a'_1 \cdot b'_3 + a'_2 \cdot b'_2 + p'_1 \cdot q'_3 + p'_3 \cdot q'_1 + p'_2 \cdot q'_2$
5. $u_0 = t_0 - r'_0 + t_1 \cdot 2^{1/4t} - r'_1 \cdot 2^{1/4t} = v_0 \cdot 2^{1/2t}$
6. $u_1 = t_2 - r'_2 + t_3 \cdot 2^{1/4t} - r'_3 \cdot 2^{1/4t} + t_4 \cdot 2^{1/2t} + v_0 = v_1 \cdot 2^{1/2t+O_{bits}}$, where O_{bits} is the number of overflow bits.
7. The value v_0 has to be less than $2^{1/4t}$ and $v_1 \leq 2^{1/4t}$. So, we add range constraints for v_0 and v_1.

The algorithm outlined above can be adapted for any primitive operation in a similar way. Furthermore, it can be transformed to operate with non-native prime fields by employing a more intricate lookup table.

References

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," 2023.

[2] ethereum.org, "Ethereum development documentation," 2023. https://ethereum.org/en/developers/docs/.

[3] A. Antonopoulos and G. D, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2018.

[4] A. Cherniaeva, I. Shirobokov, and M. Komarov, "Placeholder proof system," 2023. https://cms.nil.foundation/uploads/placeholder_reasearch_2023_c5cd673c14.pdf.

[5] A. Gabizon and Z. J. Williamson, "plookup: A simplified polynomial protocol for lookup tables." Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[6] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology - ASIACRYPT 2010* (M. Abe, ed.), (Berlin, Heidelberg), pp. 177–194, Springer Berlin Heidelberg, 2010.

[7] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, "Efficient polynomial commitment schemes for multiple points and polynomials." Cryptology ePrint Archive, Paper 2020/081, 2020. https://eprint.iacr.org/2020/081.

[8] A. Kattis, K. Panarin, and A. Vlasov, "Redshift: Transparent snarks from list polynomial commitments." Cryptology ePrint Archive, Paper 2019/1400, 2019. https://eprint.iacr.org/2019/1400.

[9] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with lasso." Cryptology ePrint Archive, Paper 2023/1216, 2023. https://eprint.iacr.org/2023/1216.

[10] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes." Cryptology ePrint Archive, Paper 2021/370, 2021. https://eprint.iacr.org/2021/370.

[11] A. Kothapalli and S. Setty, "Supernova: Proving universal machine executions without universal circuits." Cryptology ePrint Archive, Paper 2022/1758, 2022. https://eprint.iacr.org/2022/1758.

[12] A. Kothapalli and S. Setty, "Hypernova: Recursive arguments for customizable constraint systems." Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573.

[13] B. Bünz and B. Chen, "Protostar: Generic efficient accumulation/folding for special sound protocols." Cryptology ePrint Archive, Paper 2023/620, 2023. https://eprint.iacr.org/2023/620.

[14] N. Kaskov and M. Komarov, "zkllvm circuit compiler," May 2023.

[15] Privacy Scale Explorations, "zkevm specification," 2021. https://github.com/privacy-scaling-explorations/zkevm-specs.git.

[16] A. Arun, S. Setty, and J. Thaler, "Jolt: Snarks for virtual machines via lookups." Cryptology ePrint Archive, Paper 2023/1217, 2023. https://eprint.iacr.org/2023/1217.

A Opcodes

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted. The zero-th power of zero 0^0 is defined to be one.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x03	SUB	2	1	Subtraction operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] - \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x04	DIV	2	1	Integer division operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated). $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = -2^{255} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = -1 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1]) \lfloor \lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when -2^{255} is negated.
0x06	MOD	2	1	Modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{otherwise} \end{cases}$
0x07	SMOD	2	1	Signed modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0])(\lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \rvert \bmod \lvert \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert) & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x08	ADDMOD	3	1	Modulo addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x0a	EXP	2	1	Exponential operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]^{\boldsymbol{\mu}_{\mathbf{s}}[1]}$
0x0b	SIGNEXTEND	2	1	Extend length of two's complement signed integer. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{t}} & \text{if} \quad i \leqslant t \quad \text{where } t = 256 - 8(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \\ \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}} & \text{otherwise} \end{cases}$

$\boldsymbol{\mu}_{\mathbf{s}}[x]_{\mathrm{i}}$ gives the ith bit (counting from zero) of $\boldsymbol{\mu}_{\mathbf{s}}[x]$

		10s: Comparison & Bitwise Logic Operations		
Value	**Mnemonic**	δ	α	**Description**
0x10	LT	2	1	Less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x13	SGT	2	1	Signed greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x14	EQ	2	1	Equality comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x15	ISZERO	1	1	Simple not operator. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = 0 \\ 0 & \text{otherwise} \end{cases}$
0x16	AND	2	1	Bitwise AND operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}}$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} \vee \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}}$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} \oplus \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}}$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i - 248 + 8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{if} \quad i \geq 248 \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ For the Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).
0x1b	SHL	2	1	Left shift operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \times 2^{\boldsymbol{\mu}_{\mathbf{s}}[0]}) \bmod 2^{256}$
0x1c	SHR	2	1	Logical right shift operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lfloor \boldsymbol{\mu}_{\mathbf{s}}[1] \div 2^{\boldsymbol{\mu}_{\mathbf{s}}[0]} \rfloor$
0x1d	SAR	2	1	Arithmetic (signed) right shift operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lfloor \boldsymbol{\mu}_{\mathbf{s}}[1] \div 2^{\boldsymbol{\mu}_{\mathbf{s}}[0]} \rfloor$

				Where $\boldsymbol{\mu}'_{\mathbf{s}}[0]$ and $\boldsymbol{\mu}_{\mathbf{s}}[1]$ are treated as two's complement signed 256-bit integers, while $\boldsymbol{\mu}_{\mathbf{s}}[0]$ is treated as unsigned.

20s: KECCAK256

Value	Mnemonic	δ	α	Description
0x20	KECCAK256	2	1	Compute Keccak-256 hash. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{KEC}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

30s: Environmental Information

Value	Mnemonic	δ	α	Description
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{a}}$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{b}} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{o}}$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{s}}$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{v}}$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZE	0	1	Get size of input data in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{d}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ This pertains to the input data passed with the message call instruction or transaction.
0x38	CODESIZE	0	1	Get size of code running in current environment.

				$\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{b}} \rVert$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{b}} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo.
0x3a	GASPRICE	0	1	Get price of gas in current environment. This is the *effective gas price* defined in section **??**. Note that as of the *London* hard fork, this value no longer represents what is received by the miner, but rather just what is paid by the sender. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{p}}$
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \lVert \mathbf{b} \rVert & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$ where $\mathtt{KEC}(\mathbf{b}) \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{b}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] + i < \lVert \mathbf{b} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ where $\mathtt{KEC}(\mathbf{b}) \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}}$ We assume $\mathbf{b} \equiv ()$ if $\boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}] = \varnothing$. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[2] + i$ are not subject to the 2^{256} modulo. $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$
0x3d	RETURNDATASIZE	0	1	Get size of output data from the previous call from the current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert \boldsymbol{\mu}_{\mathbf{o}} \rVert$
0x3e	RETURNDATACOPY	3	0	Copy output data from the previous call to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{o}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert \boldsymbol{\mu}_{\mathbf{o}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$
0x3f	EXTCODEHASH	1	1	Get hash of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \mathtt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}) \\ \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}} & \text{otherwise} \end{cases}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$

40s: "Block" Information

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete replication packets. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{\mathrm{H_p}}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ where P is the hash of a replication packet of a particular number, up to a maximum

				age. 0 is left on the stack if the looked for replication packet number is greater than or equal to the current replication packet number or more than 256 replication packets behind the current replication packet. $P(h, n, a) \equiv \begin{cases} 0 & \text{if} \quad n > H_{\mathrm{i}} \vee a = 256 \vee h = 0 \\ h & \text{if} \quad n = H_{\mathrm{i}} \\ P(H_{\mathrm{p}}, n, a+1) & \text{otherwise} \end{cases}$ and we assert the header H can be determined from its hash h unless h is zero (as is the case for the parent hash of the genesis replication packet).
0x41	COINBASE	0	1	Get the current replication packet's beneficiary address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hc}}$
0x42	TIMESTAMP	0	1	Get the current replication packet's timestamp. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hs}}$
0x43	NUMBER	0	1	Get the current replication packet's number. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hi}}$
0x44	DIFFICULTY	0	1	Get the current replication packet's difficulty. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hd}}$
0x45	GASLIMIT	0	1	Get the current replication packet's gas limit. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hl}}$
0x46	CHAINID	0	1	Get the chain ID. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \beta$
0x47	SELFBALANCE	0	1	Get balance of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}$
0x48	BASEFEE	0	1	Get the current replication packet's base fee. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hf}}$

50s: Stack, Memory, Storage and Flow Operations

Value	Mnemonic	δ	α	Description
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$ $A'_{\mathbf{K}} \equiv A_{\mathbf{K}} \cup \{(I_{\mathrm{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0])\}$

				$C_{\text{SLOAD}}(\boldsymbol{\mu}, A, I) \equiv \begin{cases} G_{\text{warmaccess}} & \text{if} \quad (I_{\text{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0]) \in A_{\mathbf{K}} \\ G_{\text{coldsload}} & \text{otherwise} \end{cases}$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_{\text{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $A'_{\mathbf{K}} \equiv A_{\mathbf{K}} \cup \{(I_{\text{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0])\}$ $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ and A'_{r} are specified by EIP-2200 as follows. We remind the reader that the checkpoint ("original") state $\boldsymbol{\sigma}_0$ is the state if the current transaction were to revert. Let $v_0 = \boldsymbol{\sigma}_0[I_{\text{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$ be the original value of the storage slot. Let $v = \boldsymbol{\sigma}[I_{\text{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$ be the current value. Let $v' = \boldsymbol{\mu}_{\mathbf{s}}[1]$ be the new value. Then: $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv \begin{cases} 0 & \text{if} \quad (I_{\text{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0]) \in A_{\mathbf{K}} \\ G_{\text{coldsload}} & \text{otherwise} \end{cases} + \begin{cases} G_{\text{warmaccess}} & \text{if} \quad v = v' \ \vee \ v_0 \neq v \\ G_{\text{sset}} & \text{if} \quad v \neq v' \ \wedge \ v_0 = v \ \wedge \ v_0 = 0 \\ G_{\text{sreset}} & \text{if} \quad v \neq v' \ \wedge \ v_0 = v \ \wedge \ v_0 \neq 0 \end{cases}$ $A'_{\text{r}} \equiv A_{\text{r}} + \begin{cases} R_{\text{sclear}} & \text{if} \quad v \neq v' \ \wedge \ v_0 = v \ \wedge \ v' = 0 \\ r_{\text{dirtyclear}} + r_{\text{dirtyreset}} & \text{if} \quad v \neq v' \ \wedge \ v_0 \neq v \\ 0 & \text{otherwise} \end{cases}$ where $r_{\text{dirtyclear}} \equiv \begin{cases} -R_{\text{sclear}} & \text{if} \quad v_0 \neq 0 \ \wedge \ v = 0 \\ R_{\text{sclear}} & \text{if} \quad v_0 \neq 0 \ \wedge \ v' = 0 \\ 0 & \text{otherwise} \end{cases}$ $r_{\text{dirtyreset}} \equiv \begin{cases} G_{\text{sset}} - G_{\text{warmaccess}} & \text{if} \quad v_0 = v' \ \wedge \ v_0 = 0 \\ G_{\text{sreset}} - G_{\text{warmaccess}} & \text{if} \quad v_0 = v' \ \wedge \ v_0 \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\text{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{\text{pc}}$. See section **??**.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\text{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{\text{pc}} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{\text{pc}}$. See section **??**.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\text{pc}}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_{\text{i}}$
0x5a	GAS	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\text{g}}$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{\mathrm{pc}} + 1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if} \quad x < \lVert I_{\mathbf{b}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \ldots (\boldsymbol{\mu}_{\mathrm{pc}} + 2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\lVert x \rVert - 1}))$ with c as defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place 32-byte (full word) item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \ldots (\boldsymbol{\mu}_{\mathrm{pc}} + 32)\big)$ where $\boldsymbol{c}$ is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).

80s: Duplication Operations

Value	Mnemonic	δ	α	Description
0x80	DUP1	1	2	Duplicate 1st stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x81	DUP2	2	3	Duplicate 2nd stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$
⋮	⋮	⋮	⋮	⋮
0x8f	DUP16	16	17	Duplicate 16th stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[15]$

90s: Exchange Operations

Value	Mnemonic	δ	α	Description
0x90	SWAP1	2	2	Exchange 1st and 2nd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathbf{s}}[1] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x91	SWAP2	3	3	Exchange 1st and 3rd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[2]$ $\boldsymbol{\mu}'_{\mathbf{s}}[2] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
⋮	⋮	⋮	⋮	⋮
0x9f	SWAP16	17	17	Exchange 1st and 17th stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[16]$ $\boldsymbol{\mu}'_{\mathbf{s}}[16] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$

a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate's log series:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_{\mathrm{a}}, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
and to update the memory consumption counter:
$\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
The entry's topic series, $\mathbf{t}$, differs accordingly:

Value	**Mnemonic**	δ	α	**Description**
0xa0	LOG0	2	0	Append log record with no topics. $\mathbf{t} \equiv ()$
0xa1	LOG1	3	0	Append log record with one topic. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2])$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0xa4	LOG4	6	0	Append log record with four topics. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4], \boldsymbol{\mu}_{\mathbf{s}}[5])$

f0s: System operations

Value	**Mnemonic**	δ	α	**Description**
0xf0	CREATE	3	1	Create a new account with associated code.

$\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] \dots (\boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] - 1)]$
$\zeta \equiv \varnothing$

$$(\boldsymbol{\sigma}', g', A', z, \mathbf{o}) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, A, I_{\mathrm{a}}, I_{\mathrm{o}}, L(\boldsymbol{\mu}_{\mathrm{g}}), I_{\mathrm{p}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \mathbf{i}, I_{\mathrm{e}} + 1, \zeta, I_{\mathrm{w}}) & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \leqslant \boldsymbol{\sigma}[\\ & \wedge\, I_{\mathrm{e}} < 1024 \\ (\boldsymbol{\sigma}, L(\boldsymbol{\mu}_{\mathrm{g}}), A, 0, ()) & \text{otherwise} \end{cases}$$

$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except} \quad \boldsymbol{\sigma}^*[I_{\mathrm{a}}]_{\mathrm{n}} = \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{n}} + 1$
$\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} - L(\boldsymbol{\mu}_{\mathrm{g}}) + g'$
$\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$
where $x = 0$ if $z = 0$, i.e., the contract creation process failed, or $I_{\mathrm{e}} = 1024$
(the maximum call depth limit is reached) or $\boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}$ (balance of the caller
is too low to fulfil the value transfer); and otherwise $x =$ $\mathtt{ADDR}(I_{\mathrm{a}}, \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{n}}, \zeta, \mathbf{i})$, the
address of the newly created account (**??**).
$\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[2])$

$$\boldsymbol{\mu}'_{\mathbf{o}} \equiv \begin{cases} () & \text{if} \quad z = 1 \\ \mathbf{o} & \text{otherwise} \end{cases}$$

Thus the operand order is: value, input offset, input size.

Value	**Mnemonic**	δ	α	**Description**
0xf1	CALL	7	1	Message-call into an account.

$\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[3] \dots (\boldsymbol{\mu}_{\mathbf{s}}[3] + \boldsymbol{\mu}_{\mathbf{s}}[4] - 1)]$

$$(\boldsymbol{\sigma}', g', A', x, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, A^*, I_{\mathrm{a}}, I_{\mathrm{o}}, t, t, C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \\ \quad I_{\mathrm{p}}, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_{\mathrm{e}} + 1, I_{\mathrm{w}}) & \quad I_{\mathrm{e}} < 1024 \\ (\boldsymbol{\sigma}, C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), A, 0, ()) & \text{otherwise} \end{cases}$$

$n \equiv \min(\{\boldsymbol{\mu}_{\mathbf{s}}[6], \|\mathbf{o}\|\})$
$\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[5] \dots (\boldsymbol{\mu}_{\mathbf{s}}[5] + n - 1)] = \mathbf{o}[0 \dots (n - 1)]$
$\boldsymbol{\mu}'_{\mathbf{o}} = \mathbf{o}$
$\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} - C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + g'$
$\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$
$A^* \equiv A \quad \text{except} \quad A^*_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{t\}$
$t \equiv \boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 2^{160}$
$\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4]), \boldsymbol{\mu}_{\mathbf{s}}[5], \boldsymbol{\mu}_{\mathbf{s}}[6])$
where $x = 0$ if the code execution for this operation failed, or if

				$\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}$ (not enough funds) or $I_\mathrm{e} = 1024$ (call depth limit reached); $x = 1$ otherwise. Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A)$ $C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + G_{\mathrm{callstipend}} & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) & \text{otherwise} \end{cases}$ $C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_\mathrm{g} - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A)), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if} \quad \boldsymbol{\mu}_\mathrm{g} \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu} \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$ $C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv C_{\mathrm{aaccess}}(t, A) + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{\mathrm{callvalue}} & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{\mathrm{newaccount}} & \text{if} \quad \mathtt{DEAD}(\boldsymbol{\sigma}, t) \wedge \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A', x, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, A^*, I_\mathrm{a}, I_\mathrm{o}, I_\mathrm{a}, t, C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \\ \quad I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e} + 1, I_\mathrm{w}) & \quad I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), A, 0, ()) & \text{otherwise} \end{cases}$ Note the change in the fourth parameter to the call Θ from the 2nd stack value $\boldsymbol{\mu}_\mathbf{s}[1]$ (as in CALL) to the present address I_a. This means that the recipient is in fact the same account as at present, simply that the code is overwritten.
0xf3	RETURN	2	0	Halt execution returning output data. $H_{\text{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section **??**. $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$
0xf4	DELEGATECALL	6	1	Message-call into this account with an alternative account's code, but persisting the current values for *sender* and *value*. Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is $\boldsymbol{\mu}_\mathbf{s}[2]$. As a result, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. Otherwise it is equivalent to CALL except: $(\boldsymbol{\sigma}', g', A', x, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, A^*, I_\mathrm{s}, I_\mathrm{o}, I_\mathrm{a}, t, C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), & \text{if} \quad I_\mathrm{e} < 1024 \\ \quad I_\mathrm{p}, 0, I_\mathrm{v}, \mathbf{i}, I_\mathrm{e} + 1, I_\mathrm{w}) & \\ (\boldsymbol{\sigma}, C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), A, 0, ()) & \text{otherwise} \end{cases}$ Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ. This means that the recipient is in fact the same account as at present, simply

				that the code is overwritten *and* the context is almost entirely identical.
0xf5	CREATE2	4	1	Create a new account with associated code. Exactly equivalent to CREATE except: The salt $\zeta \equiv \boldsymbol{\mu}_{\mathbf{s}}[3]$.
0xfa	STATICCALL	6	1	Static message-call into an account. Exactly equivalent to CALL except: The argument $\boldsymbol{\mu}_{\mathbf{s}}[2]$ is replaced with 0. The deeper argument $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$, $\boldsymbol{\mu}_{\mathbf{s}}[5]$ and $\boldsymbol{\mu}_{\mathbf{s}}[6]$ are respectively replaced with $\boldsymbol{\mu}_{\mathbf{s}}[2]$, $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$ and $\boldsymbol{\mu}_{\mathbf{s}}[5]$. The last argument of Θ is $\perp$.
0xfd	REVERT	2	0	Halt execution reverting state changes but returning data and remaining gas. $H_{\text{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)]$ The effect of this operation is described in (**??**). For the gas calculation, we use the memory expansion function, $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. $A'_{\mathbf{s}} \equiv A_{\mathbf{s}} \cup \{I_{\mathrm{a}}\}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{r\}$ $\boldsymbol{\sigma}'[r] \equiv \begin{cases} \varnothing & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \ \wedge \ \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} = 0 \\ (\boldsymbol{\sigma}[r]_{\mathrm{n}}, \boldsymbol{\sigma}[r]_{\mathrm{b}} + \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}, \boldsymbol{\sigma}[r]_{\mathbf{s}}, \boldsymbol{\sigma}[r]_{\mathrm{c}}) & \text{if} \quad r \neq I_{\mathrm{a}} \\ (\boldsymbol{\sigma}[r]_{\mathrm{n}}, 0, \boldsymbol{\sigma}[r]_{\mathbf{s}}, \boldsymbol{\sigma}[r]_{\mathrm{c}}) & \text{otherwise} \end{cases}$ where $r = \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}$ $\boldsymbol{\sigma}'[I_{\mathrm{a}}]_{\mathrm{b}} = 0$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{\text{selfdestruct}} + \begin{cases} 0 & \text{if} \quad r \in A_{\mathbf{a}} \\ G_{\text{coldaccountaccess}} & \text{otherwise} \end{cases}$ $+ \begin{cases} G_{\text{newaccount}} & \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}, r) \wedge \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \neq 0 \\ 0 & \text{otherwise} \end{cases}$