

Off-Whitepaper

Ethereum

Micah Dameron

*Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.*

The Zen of Python

Abstract

The goal of this paper is to create and expand concepts from Ethereum about which, notwithstanding any earlier documentation, there may be some justified confusion. We use pseudocode rather than mathematical notation to describe Ethereum's operation, because pseudocode has many advantages when describing *Abstract State Machines*,^a like Ethereum. This paper takes an approach to describing Ethereum that focuses on clarity and approachability. Our prime source is the Ethereum *Yellowpaper*, but supplemental knowledge and crucial points from other sources have also been used for the reader's benefit.

^aE. Borger and S. Robert F., *Abstract state machines: A method for high-level system design and analysis*. 1, pp. 3-8. Springer, 2003.

Acknowledgements

Thank you to the Ethereum founders for creating a product worth writing about in minute detail. Thanks to the Ethereum Foundation for maintaining this product in its basic integrity. Thanks to the ConsenSys Mesh for supporting my work on this project, by contributing your vast knowledge and expertise. Finally, thank you to Dr. Gavin Wood for your technical astuteness and creative genius; your Yellowpaper has given Ethereum a soul worth decoding.

Contents

1. Native Currency & Mining	5
1.1. Mining	5
1.1.1. Total Difficulty	5
1.2. GHOST Protocol	5
2. Memory and Storage	5
2.1. Data Structures	5
2.1.1. Merkle-Patricia Trees	5
2.1.2. World State	5
2.1.3. The Block	6
2.1.4. Block Header	6
2.1.5. Block Footer	6
2.2. State Database	6
2.2.1. Tries	6
2.2.2. Tree Terminology ¹	7
2.2.3. RLP	7
2.2.4. Create Account	7
2.2.5. Account State	8
2.3. Bloom Filter	8
2.3.1. Transaction Receipts	8
3. Processing and Computation	8
3.1. Verification	8
3.2. Transaction	8
3.2.1. Transactions Root	8
3.3. State Transition Function	9
3.3.1. Ethash	9
3.4. Verification	9
3.4.1. Ommers	9
3.5. Transactions	9
3.6. Serialization	9
3.7. Deserialization	9
3.8. Execution	10
3.8.1. Code Deposit	10
3.8.2. Intrinsic Validity	10
3.8.3. Execution Model	10
3.9. Substate	11
3.10. Provisional State	11
3.10.1. Message Calls	11
3.10.2. Contract Creation	12
3.10.3. Account Creation	12
3.11. Halting	12
3.12. Gas	12
3.12.1. Machine State	12

3.12.2. Exceptional Halting	13
3.12.3. EVM Code	13
3.12.4. Opcodes/EVM Assembly	13
A. Opcodes	14
A.1. 0x10's: Comparisons and Bitwise Logic Operations	14
A.2. 0x20's: SHA3	14
A.3. 0x30's: Environmental Information	14
A.4. 0x40's: Block Data	15
A.5. 0x50's: Stack, memory, storage, and flow operations.	15
A.6. 0x60-70's: Push Operations	16
A.7. 0x80's: Duplication Operations	17
A.8. 0x90's: Swap Operations	17
A.9. 0xa0's: Logging Operations	18
A.10.0xf0's: System Operations	18
B. Higher Level Languages	19
B.1. Lower-Level Lisp	19
B.2. Solidity	19
B.3. Serpent	19
B.4. Viper	19
References	20
Glossary	21
Acronyms	22
Index	23

1. Native Currency & Mining

Because Ethereum is turing-complete, there needs to be a *network cost unit* to mitigate the possibility of abusing the network with excessive computational expenditures. The smallest unit of currency in Ethereum is the Wei, which is equal to $\Xi 10^{-18}$, where Ξi stands for 1 Ether. All currency transactions in Ethereum are counted in Wei. There is also the Szabo, which is $\Xi 10^{-6}$, and the Finney, which is $\Xi 10^{-3}$.

Unit	Ether	Wei
Ether	$\Xi 1.000000000000000000$	1,000,000,000,000,000,000
Finney	$\Xi 0.001000000000000000$	1,000,000,000,000,000
Szabo	$\Xi 0.000001000000000000$	1,000,000,000,000
Wei	$\Xi 0.000000000000000001$	1

1.1. Mining

Apply Rewards

Notation : `apply_rewards`

Description : The third process in `block_finalization` that sends the mining reward to an account's address.

1.1.1. Total Difficulty

The *Total Difficulty* of a block is defined recursively by a function which calculates the difficulty of all blocks prior to the header in the present block.

Pseudocode	Definition
<code>total(difficulty)</code>	Total difficulty at <i>this block</i> .
<code>block(difficulty)</code>	<i>This block's</i> difficulty.

1.2. GHOST Protocol

Notation : `ghost`

Description : Stands for greedy heaviest object subtree; the GHOST Protocol determines the probable correctness of the next block based on the number of miners attaching to it.

2. Memory and Storage

2.1. Data Structures

2.1.1. Merkle-Patricia Trees

Notation : `merkle_p_t`

Description : Generalized Merkle DHT

As blockchain technologies move beyond the "1.0" model of every node processing every transaction, and a more diverse ecosystem including "light clients" that achieve security by downloading only a small portion of the blockchain and extracting the rest of the data on-demand through hash-based authentication comes into play, and particularly in the long term as scalability models essentially turn `_every_` node into a light client, there arises the need to develop a strong, robust and effective networking infrastructure to handle the load. Ideally, the core technology should be built to be maximally generalized, so that the same core code and network can be used for multiple blockchain, as well as non-blockchain, applications.

2.1.2. World State

Also known as *Actual State*, this is a MAPPING of addresses and account states through the use of RLP. The mapping is stored as a Merkle-Patricia **trie** in a DATABASE BACKEND.^a that maintains a mapping of bytearrays to bytearrays.^b The cryptographic internal data going back to the **root node** represents the *State* of the Blockchain at any given root, i.e. at any given *time*.^c As a whole, the state is the sum total of database relationships in the **state database**. The state is an inert position on the chain, a position between prior state and post state; a block's frame of

^aThe database backend is accessed by users through an external application, most likely an Ethereum client; see also: **state database**

^bA bytearray is specific set of bytes [data] that can be loaded into memory. It is a structure for storing binary data, e.g. the contents of a file.

^cThis permanent data structure makes it possible to easily recall any previous state with its root hash keeping the resources off-chain and minimizing on-chain storage needs.

reference, and a defined set of relationships to that frame of reference.

2.1.3. The Block

A block is made up of 17 different elements. The first 15 elements are part of what is called the *block header*.

2.1.4. Block Header

Notation : header

Description : The information contained in a block besides the transactions list. This consists of:

1. **Parent Hash** – This is the Keccak-256 hash of the parent block's header.
2. **Ommers Hash** – This is the Keccak-256 hash of the ommer's list portion of this block.
3. **Beneficiary** – This is the 20-byte address to which all block rewards are transferred.
4. **State Root** – This is the Keccak-256 hash of the root node of the state trie, after a block and its transactions are finalized.
5. **Transactions Root** – This is the Keccak-256 hash of the root node of the trie structure populated with each transaction from a Block's transaction list.
6. **Receipts Root** – This is the Keccak-256 hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block.
7. **Logs Bloom** – This is the bloom filter composed from indexable information (log address and log topic) contained in the receipt for each transaction in the transactions list portion of a block.
8. **Difficulty** – This is the difficulty of this block – a quantity calculated from the previous block's difficulty and its timestamp.
9. **Number** – This is a quantity equal to the number of ancestor blocks behind the current block.
10. **Gas Limit** – This is a quantity equal to the current maximum gas expenditure per block.
11. **Gas Used** – This is a quantity equal to the total gas used in transactions in this block.
12. **Timestamp** – This is a record of Unix's time at this block's inception.
13. **Extra Data** – This byte-array of size 32 bytes or less contains extra data relevant to this block.
14. **Mix Hash** – This is a 32-byte hash that verifies a sufficient amount of computation has been done on this block.
15. **Nonce** – This is an 8-byte hash that verifies a sufficient amount of computation has been done on this block.
16. **Ommers Block Headers** – These are the same components listed above for any ommers.

2.1.5. Block Footer

17. **Transaction Series** – This is the only non-header content in the block.

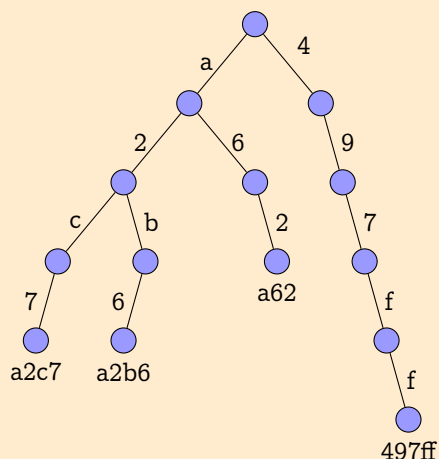
2.2. State Database

2.2.1. Tries

Notation : trie

Description : Also known as a search tree, a radix tree, or a prefix tree, a trie is tree structure for organizing sequential data hierarchically. Members of a previous generation can spawn infinitely many members of the next generation, but the data cannot be read out of sequence. In other words a parent can spawn any number of children, but cannot skip a generation. This ensures that all the data in a trie are related to each other properly.

The following is a search tree beginning with hexadecimal values a and 4:



2.2.2. Tree Terminology¹

Root – The top (first) node in a tree.

Child – A node directly connected to another node when moving away from the Root.

Parent – The converse notion of a child.

Siblings – A group of nodes with the same parent.

Descendant – A node reachable by repeated proceeding from parent to child.

Ancestor – A node reachable by repeated proceeding from child to parent.

Leaf ^a – A node with no children.

Branch ^b – A node with at least one child.

Degree – The number of subtrees of a node.

Edge – The connection between one node and another.

Path – A sequence of nodes and edges connecting a node with a descendant.

Level – The level of a node is defined by 1 + (the number of connections between the node and the root).

Node Height – The height of a node is the number of edges on the longest path between that node and a leaf.

Tree Height – The height of a tree is the height of its root node.

Depth – The depth of a node is the number of edges from the tree's root node to the node.

Forest – A forest is a set of $n \geq 0$ disjoint trees.

^aa.k.a. External Node

2.2.3. RLP

Notation : rlp

Description : RLP encodes arrays of nested binary data to an arbitrary depth; it is the main serialization method for data in Ethereum. RLP encodes mainly structure and does not pay heed to what type of data it is encoding.

Positive RLP integers are represented with the most significant value stored at the lowest memory address (big endian) and without any leading zeroes. As a result, the RLP integer value for 0 is represented by an empty byte-array. If a non-empty deserialized integer begins with leading zeros it is invalid.²

The global state database is encoded as RLP for fast traversal and inspection of data. In structure it constitutes a mapping between *addresses* and *account states*. Since it is stored on node operator's computers, the tree can be traversed speedily and without network delay. RLP encodes values as byte-arrays, or as sequences of further values.³

This means that:

```

if   rlp(x)           = bytearray
then rlp(bytearray)  = true
elif rlp(x)           = value
then rlp(value)      = true
elif rlp(x)           = null
then rlp(x)          = false

```

1. If the RLP-serialized byte-array contains a single byte integer value less than 128, then the output is exactly equal to the input.

In other words:

2.2.4. Create Account

Notation : create

Description : Account creation definitively accours in the YP with contract creation. Does account creation also apply to regular account creation, or does it only apply to contract accounts? Also, what about protocol accounts?

Related : `init`

2.2.5. Account State

Notation : `body`

Description : The EVM-code fragment that executes each time an account receives a message call.

Description : The account state is made up of four variables:

1. **nonce** The number of transactions sent from this address, or the number of contract creations made by the account associated with this address.
2. **balance** The number of Wei owned by this address.
3. **storage_root** A 256-bit (32-byte) hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account.
4. The storage root aspect of an account's state is the hash of the trie
5. **code_hash** The hash of the EVM code of this account's contract.

The account state is the state of any particular account during some specified world state.

Nonce The **nonce** aspect of an ACCOUNT'S STATE is the number of transactions sent from, or the number of contract-creations by, the address of that account.^a

Storage Root The **storage root** aspect of an ACCOUNT'S STATE is the hash of the trie^b

Code Hash The **code hash** aspect of an ACCOUNT'S STATE is the HASH OF THE EVM CODE of this account. Code hashes are STORED in the **state database**. Code hashes are permanent and they are executed when the address belonging to that account RECEIVES a message call.

Balance The amount of **Wei** OWNED by this account.

- Key/value pair stored inside the root hash.
- L_I^* is defined as the element-wise transformation of the base function
- The *element-wise transformation of the base-function* refers to all of the key/value pairs in L_I
- L_I refers to a particular **trie**.

2.3. Bloom Filter

Notation : `logs_bloom`

Description : The Bloom Filter is composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.

2.3.1. Transaction Receipts

3. Processing and Computation

3.1. Verification

Notation : `verification`

Description : The process in The EVM that verifies Ommer Headers.

3.2. Transaction

Description : A single cryptographically signed instruction sent to the Ethereum network. There are two types of transactions: MESSAGE CALLS and CONTRACT CREATIONS. Transactions are ubiquitous on the Ethereum network, and represent several common fields.

3.2.1. Transactions Root

Notation : `listhash`

Alternatively: Transactions Root

^a σ is the world state at a certain given time, and n is the number of transactions or contract creations by that account.

^bA particular path from root to leaf in the **state database** that encodes the STORAGE CONTENTS of the account.

Description : The K256 hash of the root node^a that precedes the transactions in the `transactions_list` section of a Block.

1. **Nonce** – The number of transactions sent by the sender.
2. **Gas Price** – The number of Wei to pay the network for unit of gas.
3. **Gas Limit** – The maximum amount of gas to be used in while executing a transaction.
4. **To** – The 20-character recipient of a message call.^b
5. **Value** The number of Wei to be transferred to the recipient of a message call.^c
6. **v, r, s**
- 7.
- 8.

3.3. State Transition Function

State Transitions come about through a what is known as the State Transition Function; this is an abstraction of several operations in Ethereum which comprise the overall act of computing changes to the *machine state* prior to adding them to the *world state*, that is, through them being finalized and rewards applied to a given miner. `apply_rewards` and `block_beneficiary` are here. subsectionMining

Block Beneficiary The 160-bit (20-byte, or 20-character) address to which all fees collected from the successful mining of a block are transferred.

Apply Rewards The third process in `block_finalization` that sends the mining reward to an account's address. A scalar value corresponding to the difficulty level of a current block. This can be calculated from the previous block's difficulty level and the timestamp.

3.3.1. Ethash

GHOST Protocol

^aA root node is a type of progenitor node

^bIn the case of a contract creation this is 0x00000000000000000000.

^cIn the case of a contract creation, an endowment to the newly created contract account.

3.4. Verification

Verifies Ommers headers

3.4.1. Ommers

Ommershash

3.5. Transactions

The basic method for Ethereum accounts to interact with eachother. Transactions lie at the heart of Ethereum, and are entirely responsible for the dynamism and flexibility of the platform. Transactions are the bread and butter of state transitions, that is of block additions, which contain all of the computation performed in one block. Each transaction applies the execution changes to the *machine state*, a temporary state which consists of all the temporary changes in computation that must be made before a block is finalized and added to the world state.

Notation : sender

Description : A function that maps transactions to their sender using ECDSA of the SECP-256k1 curve, (excepting the latter three signature fields) as the datum to sign. The sender of a given transaction can be represented: `transaction.sender`

3.6. Serialization

3.7. Deserialization

Notation : big_endian

Description : This function expands a positive-integer value to a big-endian byte array of minimal length. When accompanied by a `·` operator, it signals sequence concatenation. The `big_endian` function accompanies RLP serialization and deserialization.

3.8. Execution

Description : The execution of a transaction defines the state transition function: `stf`. However, before any transaction can be executed it needs to go through the initial tests of intrinsic validity.

3.8.1. Code Deposit

Notation : `code_deposit`

Description : If the initialization code completes successfully, a final contract-creation cost is paid, the code-deposit cost, c , proportional to the size of the created contract's code.

3.8.2. Intrinsic Validity

The criteria for intrinsic validity are as follows:

- The transaction follows the rules for *well-formed RLP* (recursive length prefix.)
- The *signature* on the transaction is valid.
- The *nonce* on the transaction is valid, i.e. it is equivalent to the sender account's current nonce.
- The *gas_limit* is greater than or equal to the *intrinsic_gas* used by the transaction.
- The sender's account balance contains the cost required in up-front payment.

Accordingly, the post-transactional state of Ethereum is expressed thus:

```
transaction(post.state) = stf(present.state,
transaction)
```

While the amount of gas used in the execution is expressed: `stf(gas_used)` and the accrued log items belonging to the transaction are expressed: `stf(logsbloom, content)(logsbloom, set)` Information concerning the result of a transaction's execution is stored in the transaction receipt `tx_receipt`. The set of log events which are created through the execution of the transaction, `logs_set` in addition to the bloom filter which contains the actual information from those log events `logs_bloom` are located in the transaction receipt. In addition, the post-transaction state `post_transaction(state)` and the amount of gas used in the block containing the transaction

receipt `post(gas_used)` are stored in the transaction receipt. Thusly the transaction receipt is a record of any given execution.

A valid transaction execution begins with a permanent change to the state: the nonce of the sender account is increased by one and the balance is decreased by the `collateral_gas`^a which is the amount of gas a transaction is required to pay prior to its execution. The original transactor will differ from the sender if the message call or contract creation comes from a contract account executing code.

After a transaction is executed, there comes a PROVISIONAL STATE:

Notation : `pv_state`

Description : Used to define the PRE-FINAL STATE, the PROVISIONAL STATE. Gas used for the execution of individual EVM opcodes prior to their potential addition to the `world_state` creates the provisional state. `productive_gas`, and an associated substate `substate_a`.

Code execution always depletes gas. If gas runs out, an out-of-gas error is signaled (`oog`) and the resulting state defines itself as an empty set; it has no effect on the world state. This describes the transactional nature of Ethereum. In order to affect the WORLD STATE, a transaction must go through completely or not at all.

3.8.3. Execution Model

Description : The stack-based *virtual machine* which lies at the heart of the Ethereum and performs the actions of a computer. This is actually an instantial runtime that executes several substates, as EVM computation instances, before adding the finished result, all calculations having been completed, to the final state via the finalization function.

In addition to the system state σ , and the remaining gas for computation g , there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple I :

^aDesignated "intrinsic_gas" in the Yellowpaper

- `account_address`, the address of the account which owns the code that is executing.
- `sender_address` the sender address of the transaction that originated this execution.
- `originator_price` the price of gas in the transaction that originated this execution.
- `input_data`, a byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- `account_address` 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.
- `newstate_value` the value, in Wei, passed to this account if the execution agent is a transaction, this would be the transaction value.³
- `code.array` the byte array that is the machine code to be executed.³
- `samestate_header` the block header of the present block.
- `the stack depth` the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATEs being executed at present).³

3.9. Substate

3.10. Provisional State

Description : A smaller, temporary state that is generated during transaction execution. It contains three sets of data:

- The accounts tagged for self-destruction following the transaction's completion. `self_destruct(accounts)`
- The `logs_series`, which creates checkpoints in EVM code execution for frontend applications to explore, and is made up of `the_logs_set` and `logs_bloom` from the `tx_receipt`.
- The refund balance.^a

3.10.1. Message Calls

Description :

Notation : `message_call`

Description : A message call can come from a transaction or internally from contract code execution. It contains the field `DATA`, which consists of user data input to a message call. Messages allow communication between accounts (whether contract or external,) and are a carryover from established concepts in Computer Science, most notably the *MPI: Message-Passing Framework*. Messages can come in the form of `msg_calls` which give output data. If an account has EVM code in it (a contract account,) this code gets executed when the account receives a message call. Message calls and contract creations are both *transactions*, but contract creations are never considered the same as message calls. Message calls always transfer some amount of value to an account. If the message call is an account creation transaction then the value given is taken on the role of an endowment toward the new account. Every time an account receives a message call it returns the body, something which is triggered by the `init` function. A message call can come through a transaction, or through the internal execution of code. Message call transactions only contain data. They are separate from regular, standard *transactions*.

Universal Gas Message calls always have a universally agreed-upon cost in gas. There is a strong distinction between contract creation transactions and message call transactions. Computation performed, whether it is a contract creation or a message call, represents the currently legal valid state. There can be no invalid transactions from this point.³ There is also a message call/contract creation *stack*. This stack has a depth, depending on how many transactions are in it. Contract creations and message calls have entirely different ways of executing, and are entirely different in their roles in Ethereum. The concepts can be conflated. Message calls can result in computation that occurs in the next state rather than the current one. If an account that is currently executing receives a message call, no code will execute, because the account might exist but has no code in it yet. To

^aThe `SSTORE` operation increases the amount refunded by resetting contract storage to zero from some non-zero state.

execute a message call transactions are required:

- Sender
- Transaction_Originator
- Recipient
- Account (usually the same as the recipient)
- Available_Gas
- Value
- Gas_Price
- An arbitrary length byte-array. `arb_array`
- Present_Depth of the message call/contract creation stack.

Notation : `data`

Description : User data input to a `message_call`, structured as an unlimited size byte-array.

3.10.2. Contract Creation

Notation : `init`

Description : When `INIT` is executed it returns the `BODY`. `Init` is executed only once at `ACCOUNT_CREATION`, and permanently discarded after that. Contract creation transactions are equal the recursive length prefix of an empty byte-sequence.

3.10.3. Account Creation

3.11. Halting

Execution Environment

Notation : `ERE`

Description : The environment under which an Autonomous Object executes in the EVM: the EVM runs as a part of this environment.

Notation : `big_endian_f`

Description : `BIG_ENDIAN_FUNCTION` This function expands a positive-integer value to a big-endian byte array of minimal length. When accompanied by a `·` operator, it signals sequence concatenation. The `big_endian` function accompanies RLP serialization and deserialization.

3.12. Gas

Description : The fundamental network cost unit converted to and from Ether as needed to complete the transaction while it is sent. Gas is arbitrarily determined at the moment it is needed, by the block and according to the miners decision to charge certain fees.

Miner Choice Miners choose which gas prices they want to accept.

Gasprice

Notation : `gas_limit`

Description : A value equal to the current limit of gas expenditure per block, according to the miners.

Gaslimit Any unused gas is refunded to the user.

Gasused

Description : A value equal to the total gas used in transactions in this block.

3.12.1. Machine State

The machine state is a tuple consisting of five elements:

1. `gas_available`
2. `program_counter`
3. `memory_contents` A series of zeroes of size 2^{256}
4. `memory_words.count`
5. `stack_contents`

There is also, [`to_execute`]: the current operation to be executed

3.12.2. Exceptional Halting

An exceptional halt may be caused by four conditions existing on the stack with regard to the next opcode in line for execution:

```
if
out_of_gas = true
or
bad_instruction = true
or
bad_stack_size = true
or
bad_jumpdest = true
then throw exception
else exec opcode x
then init control_halt
```

Exceptional halts are reserved for opcodes that fail to execute. They are not caused through an opcode's actual execution.

- The amount of remaining gas in each transaction is extracted from information contained in the `machine_state`
- A simple iterative recursive loop³ with a boolean value:
 - `true` indicating that in the run of computation, an exception was signaled
 - `false` indicating in the run of computation, exceptions were signaled. If this value remains false for the duration of the execution until the set of transactions becomes a series (rather than an empty set.) This means that the machine has reached a controlled halt.

Substate

Notation : substate

Description : A smaller, temporary state that is generated during transaction execution. It contains three sets of data:

- The accounts tagged for self-destruction following the transaction's completion.
`self_destruct(accounts)`

- The `logs_series`, which creates checkpoints in EVM code execution for frontend applications to explore, and is made up of `the_logs_set` and `logs_bloom` from the `tx_receipt`.
- The refund balance.^a

The substate is an emergent, ever-changing ball of computational energy that is about to be applied to the main state. It is the *meta state* by which transactions are decided valid and to be added to the blockchain.

3.12.3. EVM Code

The bytecode that the EVM can natively execute. Used to explicitly specify the meaning of a message to an account.

Notation : contract

Description : A piece of EVM Code that may be associated with an Account or an Autonomous Object.

3.12.4. Opcodes/EVM Assembly

The human readable version of EVM code.

^aThe `SSTORE` operation increases the amount refunded by resetting contract storage to zero from some non-zero state.

A. Opcodes

A.1. 0x10's: Comparisons and Bitwise Logic Operations

Data	Opcode	Gas	Input	Output	Description
0x00	STOP	0	0	0	Halts execution.
0x01	ADD	3	2	1	Addition operation.
0x02	MUL	5	2	1	Multiplication operation.
0x03	SUB	3	2	1	Subtraction operation.
0x04	DIV	5	2	1	Integer division operation.
0x05	SDIV	5	2	1	Signed integer division operation (truncated.)
0x06	MOD	5	2	1	Modulo remainder operation.
0x07	SMOD	5	2	1	Signed modulo remainder operation.
0x08	ADDMOD	8	3	1	Modulo addition operation.
0x09	MULMOD	8	3	1	Modulo multiplication operation.
0x0a	EXP	10	2	1	Exponential operation.
0x0b	SIGNEXTEND	5	2	1	Extend the length of two's complementary signed integer.
0x10	LT	3	2	1	Less-than comparison.
0x11	GT	3	2	1	Greater-than comparison.
0x12	SLT	3	2	1	Signed less-than comparison.
0x13	SGT	3	2	1	Signed greater-than comparison.
0x14	EQ	3	2	1	Equality comparison.
0x15	ISZERO	3	1	1	Simple not operator.
0x16	AND	3	2	1	Bitwise AND operation.
0x17	OR	3	2	1	Bitwise OR operation.
0x18	XOR	3	2	1	Bitwise XOR operation.
0x19	NOT	3	1	1	Bitwise NOT operation.
0x1a	BYTE	3	2	1	Retrieve single byte from word.

A.2. 0x20's: SHA3

Data	Opcode	Gas	Input	Output	Description
0x20	SHA3	30	2	1	Compute a Keccak-256 hash.

A.3. 0x30's: Environmental Information

Data	Opcode	Gas	Input	Output	Description
0x30	ADDRESS	2	0	1	Get the address of the currently executing account.
0x31	BALANCE	400	1	1	Get the balance of the given account.
0x32	ORIGIN	2	0	1	Get execution origination address. This is always the original sender of a transaction, never a contract account.

0x33	CALLER	2	0	1	Get caller address. This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	2	0	1	Get deposited value by the instruction/transaction responsible for this execution.
0x35	CALLDATALOAD	3	1	1	Get input data of the current environment.
0x36	CALLDATASIZE	2	0	1	Get size of input data in current environment. This refers to the optional data field that can be passed with a message call instruction or transaction.
0x37	CALLDATACOPY	3	3	0	Copy input data in the current environment to memory. This refers to the optional data field passed with the message call instruction or transaction.
0x38	CODESIZE	2	0	1	Get size of code running in the current environment.
0x39	CODECOPY	3	3	0	Copy the code running in the current environment to memory.
0x3a	GASPRICE	2	0	1	Get the price of gas in the current environment. This is the gas price specified by the originating transaction.
0x3b	EXTCODESIZE	700	1	1	Get the size of an account's code.
0x3c	EXTCODECOPY	700	4	0	Copy an account's code to memory.
0x3d	RETURNDATASIZE	2	0	1	
0x3e	RETURNDATACOPY	3	3	0	

A.4. 0x40's: Block Data

Data	Opcode	Gas	Input	Output	Description
0x40	BLOCKHASH	20	1	1	Get the hash of one of the 256 most recent blocks. ^a
0x41	COINBASE	2	0	1	Look up a block's beneficiary address by its hash.
0x42	TIMESTAMP	2	0	1	Look up a block's timestamp by its hash.
0x43	NUMBER	2	0	1	Look up a block's number by its hash.
0x44	DIFFICULTY	2	0	1	Look up a block's difficulty by its hash.
0x45	GASLIMIT	2	0	1	Look up a block's gas limit by its hash.

A.5. 0x50's: Stack, memory, storage, and flow operations.

Data	Opcode	Gas	Input	Output	Description
------	--------	-----	-------	--------	-------------

^aA value of 0 is left on the stack if the block number is more than 256 in number behind the current one, or if it is a number greater than the current one.

0x50	POP	2	1	0	Removes an item from the stack.
0x51	MLOAD	3	1	1	Load a word from memory.
0x52	MSTORE	3	2	0	Save a word to memory.
0x53	MSTORE8	3	2	0	Save a byte to memory.
0x54	SLOAD	200	1	1	Load a word from storage.
0x55	SSTORE	0	2	0	Save a word to storage.
0x56	JUMP	8	1	0	Alter the program counter.
0x57	JUMPI	10	2	0	Conditionally alter the program counter.
0x58	PC	2	0	1	Look up the value of the program counter prior to the increment resulting from this instruction.
0x59	MSIZE	2	0	1	Get the size of active memory in bytes.
0x5a	GAS	2	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction.
0x5b	JUMPDEST	1	0	0	Mark a valid destination for jumps. ^a

A.6. 0x60-70's: Push Operations

Data	Opcode	Gas	Input	Output	Description
0x60	PUSH1	-	0	1	Place a 1-byte item on the stack.
0x61	PUSH2	-	0	1	Place a 2-byte item on the stack.
0x62	PUSH3	-	0	1	Place a 3-byte item on the stack.
0x63	PUSH4	-	0	1	Place a 4-byte item on the stack.
0x64	PUSH5	-	0	1	Place a 5-byte item on the stack.
0x65	PUSH6	-	0	1	Place a 6-byte item on the stack.
0x66	PUSH7	-	0	1	Place a 7-byte item on the stack.
0x67	PUSH8	-	0	1	Place a 8-byte item on the stack.
0x68	PUSH9	-	0	1	Place a 9-byte item on the stack.
0x69	PUSH10	-	0	1	Place a 10-byte item on the stack.
0x6a	PUSH11	-	0	1	Place a 11-byte item on the stack.
0x6b	PUSH12	-	0	1	Place a 12-byte item on the stack.
0x6c	PUSH13	-	0	1	Place a 13-byte item on the stack.
0x6d	PUSH14	-	0	1	Place a 14-byte item on the stack.
0x6e	PUSH15	-	0	1	Place a 15-byte item on the stack.
0x6f	PUSH16	-	0	1	Place a 16-byte item on the stack.
0x70	PUSH17	-	0	1	Place a 17-byte item on the stack.
0x71	PUSH18	-	0	1	Place a 18-byte item on the stack.
0x72	PUSH19	-	0	1	Place a 19-byte item on the stack.
0x73	PUSH20	-	0	1	Place a 20-byte item on the stack.
0x74	PUSH21	-	0	1	Place a 21-byte item on the stack.
0x75	PUSH22	-	0	1	Place a 22-byte item on the stack.

^aThis operation has no effect on the machine_state during execution.

0x76	PUSH23	-	0	1	Place a 23-byte item on the stack.
0x77	PUSH24	-	0	1	Place a 24-byte item on the stack.
0x78	PUSH25	-	0	1	Place a 25-byte item on the stack.
0x79	PUSH26	-	0	1	Place a 26-byte item on the stack.
0x7a	PUSH27	-	0	1	Place a 27-byte item on the stack.
0x7b	PUSH28	-	0	1	Place a 28-byte item on the stack.
0x7c	PUSH29	-	0	1	Place a 29-byte item on the stack.
0x7d	PUSH30	-	0	1	Place a 30-byte item on the stack.
0x7e	PUSH31	-	0	1	Place a 31-byte item on the stack.
0x7f	PUSH32	-	0	1	Place a 32-byte item on the stack.

A.7. 0x80's: Duplication Operations

Data	Opcode	Gas	Input	Output	Description
0x80	DUP1	-	1	2	Duplicate the 1st item in the stack.
0x81	DUP2	-	2	3	Duplicate the 2nd item in the stack.
0x82	DUP3	-	3	4	Duplicate the 3rd item in the stack.
0x83	DUP4	-	4	5	Duplicate the 4th item in the stack.
0x84	DUP5	-	5	6	Duplicate the 5th item in the stack.
0x85	DUP6	-	6	7	Duplicate the 6th item in the stack.
0x86	DUP7	-	7	8	Duplicate the 7th item in the stack.
0x87	DUP8	-	8	9	Duplicate the 8th item in the stack.
0x88	DUP9	-	9	10	Duplicate the 9th item in the stack.
0x89	DUP10	-	10	11	Duplicate the 10th item in the stack.
0x8a	DUP11	-	11	12	Duplicate the 11th item in the stack.
0x8b	DUP12	-	12	13	Duplicate the 12th item in the stack.
0x8c	DUP13	-	13	14	Duplicate the 13th item in the stack.
0x8d	DUP14	-	14	15	Duplicate the 14th item in the stack.
0x8e	DUP15	-	15	16	Duplicate the 15th item in the stack.
0x8f	DUP16	-	16	17	Duplicate the 16th item in the stack.

A.8. 0x90's: Swap Operations

Data	Opcode	Gas	Input	Output	Description
0x90	SWAP1	-	2	2	Exchange the 1st and 2nd stack items.
0x91	SWAP2	-	3	3	Exchange the 1st and 3rd stack items.
0x92	SWAP3	-	4	4	Exchange the 1st and 4th stack items.
0x93	SWAP4	-	5	5	Exchange the 1st and 5th stack items.
0x94	SWAP5	-	6	6	Exchange the 1st and 6th stack items.
0x95	SWAP6	-	7	7	Exchange the 1st and 7th stack items.
0x96	SWAP7	-	8	8	Exchange the 1st and 8th stack items.
0x97	SWAP8	-	9	9	Exchange the 1st and 9th stack items.
0x98	SWAP9	-	10	10	Exchange the 1st and 10th stack items.
0x99	SWAP10	-	11	11	Exchange the 1st and 11th stack items.
0x9a	SWAP11	-	12	12	Exchange the 1st and 12th stack items.

0x9b	SWAP12	-	13	13	Exchange the 1st and 13th stack items.
0x9c	SWAP13	-	14	14	Exchange the 1st and 14th stack items.
0x9d	SWAP14	-	15	15	Exchange the 1st and 15th stack items.
0x9e	SWAP15	-	16	16	Exchange the 1st and 16th stack items.
0x9f	SWAP16	-	17	17	Exchange the 1st and 17th stack items.

A.9. 0xa0's: Logging Operations

Data	Opcode	Gas	Input	Output	Description
0xa0	LOG0	375	2	0	Append log record with 0 topics.
0xa1	LOG1	750	3	0	Append log record with 1 topic.
0xa2	LOG2	1125	4	0	Append log record with 2 topic.
0xa3	LOG3	1500	5	0	Append log record with 3 topic.
0xa4	LOG4	1875	6	0	Append log record with 4 topic.

A.10. 0xf0's: System Operations

Data	Opcode	Gas	Input	Output	Description
0xf0	CREATE	32000	3	1	Create a new contract account. Operand order is: value, input offset, input size.
0xf1	CALL	700	7	1	Message-call into an account. The operand order is: gas, to, value, in offset, in size, out offset, out size.
0xf2	CALLCODE	700	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL, except the recipient is the same account as at present, but the code is overwritten.
0xf3	RETURN	0	2	0	Halt execution, then return output data. This defines the output at the moment of the halt.
0xf4	DELEGATECALL	700	6	1	Message-call into this account with an alternative account's code, but with persisting values for sender and value. DELEGATECALL takes one less argument than CALL. This means that the recipient is in fact the same account as at present, but that the code is overwritten <i>and</i> the context is almost entirely identical.
0xf5	CALLBLACKBOX	40	7	1	-
0xfa	STATICCALL	40	6	1	-
0xfd	REVERT	0	2	0	-
0xfe	INVALID	-	1	0	Designated invalid instruction.

0xff	SELFDESTRUCT	5000	1	0	Halt execution and register the account for later deletion.
------	--------------	------	---	---	---

B. Higher Level Languages

B.1. Lower-Level Lisp

The Lisp-Like low level language: a human-writable language used for authoring simple contracts and trans-compiling to higher-level languages.

B.2. Solidity

A language similar in syntax to Javascript, and the most commonly used language for creating smart contracts in Ethereum.

B.3. Serpent

B.4. Viper

References

- [1] W. contributors, *Tree (data structure)* — *wikipedia, the free encyclopedia*, [Online; accessed 15-December-2017], 2017. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Tree_\(data_structure\)&oldid=813972413](https://en.wikipedia.org/w/index.php?title=Tree_(data_structure)&oldid=813972413) (cit. on p. 7).
- [2] E. Foundation, *Ethereum whitepaper*, <https://github.com/ethereum/wiki/wiki/White-Paper>, 2017 (cit. on p. 7).
- [3] D. G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, <https://github.com/ethereum/yellowpaper>, 2017 (cit. on pp. 7, 11, 13).

Glossary

addresses 20 character strings, specifically the right-most 20 characters of the Keccak-256 hash of the RLP-derived mapping which contains the sender's address and the nonce of the block.. [21](#)

balance A value which is intrinsic to accounts; the quantity of Wei in the account. All EVM operations are associated with changes in account balance. [21](#)

beneficiary The 20-character (160-bit) address to which all fees collected from the successful mining of this block be transferred. [21](#)

block header All the information in a block besides transaction information. [21](#)

Contract A piece of EVM Code that may be associated with an Account or an Autonomous Object. [21](#)

Cryptographic hashing functions Hash functions make secure blockchains possible by establishing universal inputs for which there is typically only one possible output.. [21](#)

Ethereum Runtime Environment The environment which is provided to an Autonomous Object executing in the EVM. Includes the EVM but also the structure of the world state on which the relies for certain I/O instructions including CALL & CREATE. [21](#)

Ethereum Virtual Machine A sub-process of the *State Transition Function* which initializes and executes all of the transactions (ergo computations) in a block, prior to their finalization into the state.. [21](#)

EVM Assembly The human readable version of EVM code. [21](#)

EVM Code The bytecode that the EVM can natively execute. Used to formally specify the meaning

and ramifications of a message to an Account. [21](#)

Gas The fundamental network cost unit; gas is paid for exclusively by Ether. [21](#)

leaf node the bottom-most node in a particular tree, of blocks, one half of the “key” the other half being the root node, which creates the path between. [21](#)

Message Data (as a set of bytes) and Value (specified in Wei) that is passed between two accounts.. [21](#)

Recursive Length Prefix Recursive Length Prefix. [21](#)

root node the uppermost node in a particular tree, of blocks, representing a single world state at a particular time. [5](#), [21](#)

serialization Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the machine state of an object in order to be able to recreate it when needed. [billwagner]. [21](#)

state machine The term *State Machine* is reserved for any simple or complex process that moves deterministically from one discrete state to the next.. [21](#)

state database A database stored off-chain, [i.e. on the computer of some user running an Ethereum client] which contains a trie structure mapping bytearrays [i.e. organized chunks of binary data] to other bytearrays [other organized chunks of binary data]. The *relationships* between each node on this trie constitute a MAP, a.k.a. a MAP-PING of all previous *world states* which a client might need to reference. [5](#), [8](#), [21](#)

storage root One aspect of an ACCOUNT'S STATE: this is the hash of the trie^a that decides the STORAGE CONTENTS of the account. [21](#)

^aA particular path from root to leaf in the state database

Storage State The information particular to a given account that is maintained between the times that the account's associated EVM Code runs. [21](#)

transaction A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous Object. Transactions are recorded into each block of the blockchain. Transactions are regarded as a single unit of work, and must be processed completely or not at all.[[Ngondi2016](#)]. [21](#)

trie A tree-structure for organizing data, the position of data in the tree contains the particular path from root to leaf node that represents the key (the path from root to leaf is "one" key) you are searching the trie structure for. The data of the key is contained in the trie relationships that emerge from related nodes in the trie structure. [5](#), [8](#), [21](#)

Acronyms

ERE Ethereum Runtime Environment. [21](#)

EVM Ethereum Virtual Machine. [21](#)

RLP Recursive Length Prefix. [21](#)

Index

abstract

state-machines, [1](#)

apply rewards, [5](#)

block difficulty, [5](#)

Ether, [5](#)

Finney, [5](#)

GHOST protocol, [5](#)

mining, [5](#)

native currency, [5](#)

network cost unit, [5](#)

pseudocode, [1](#)

Szabo, [5](#)

total difficulty, [5](#)

Wei, [5](#)

Yellowpaper, [1](#)