---

title: Understanding the Yellow Paper's EVM Specifications description: Understanding the part of the Yellow Paper, the formal specifications for Ethereum, that explains the Ethereum virtual machine (EVM). author: "qbzzt" tags: ["evm"] skill: intermediate lang: en published: 2022-05-15

The Yellow Paper is the formal specification for Ethereum. Except where amended by the EIP process, it contains the exact description of how everything works. It is written as a mathematical paper, which includes terminology programmers may not find familiar. In this paper you learn how to read it, and by extension other related mathematical papers.

Which Yellow Paper? {#which-yellow-paper}

Like almost everything else in Ethereum, the Yellow Paper evolves over time. To be able to refer to a specific version, I uploaded the current version at writing. The section, page, and equation numbers I use will refer to that version. It is a good idea to have it open in a different window while reading this document.

Why the EVM? {#why-the-evm}

The original yellow paper was written right at the start of Ethereum's development. It describes the original proof-of-work based consensus mechanism that was originally used to secure the network. However, Ethereum switched off proof-of-work and started using proof-of-stake based consensus in September 2022. This tutorial will focus on the parts of the yellow paper defining the Ethereum Virtual Machine. The EVM was unchanged by the transition to proof-of-stake (except for the return value of the DIFFICULTY opcode).

9 Execution model {#9-execution-model}

This section (p. 12-14) includes most of the definition of the EVM.

The term *system state* includes everything you need to know about the system to run it. In a typical computer, this means the memory, content of registers, etc.

A Turing machine is a computational model. Essentially, it is a simplified version of a computer, which is proved to have the same ability to run computations that a normal computer can (everything that a computer can calculate a Turing machine can calculate and vice versa). This model makes it easier to prove various theorems about what is and what isn't computable.

The term Turing-complete means a computer that can run the same calculations as a Turing machine. Turing machines can get into infinite loops, and the EVM cannot because it would run out of gas, so it's only quasi-Turing-complete.

9.1 Basics {#91-basics}

This section gives the basics of the EVM and how it compares with other computational models.

A stack machine is a computer that stores intermediate data not in registers, but in a **stack**. This is the preferred architecture for virtual machines because it is easy to implement meaning that bugs, and security vulnerabilities, are a lot less likely. The memory in the stack is divided into 256-bit words. This was chosen because it is convenient for Ethereum's core cryptographic operations such as Keccak-256 hashing and elliptic curve computations. The maximum size of the stack is 1024 bytes. When opcodes are executed they are usually getting their parameters from the stack. There are opcodes specifically for reorganizing elements in the stack such as `POP` (removes item from top of stack), `DUP_N` (duplicated N'th item in stack), etc.

The EVM also has a volatile space called **memory** which is used to store data during execution. This memory is organized into 32-byte words. All memory locations are initialized to zero. If you execute this Yul code to add a word to memory, it will fill 32 bytes of memory by padding the empty space in the word with zeros, i.e. it creates one word - with zeros in locations 0-29, 0x60 to 30, and 0xA7 to 31.

```
yul mstore(0, 0x60A7)
```

`mstore` is one of three opcodes the EVM provides for interacting with memory - it loads a word into memory. The other two are `mstore8` which loads a single byte into memory, and `mload` which moves a word from memory to stack.

The EVM also has a separate non-volatile **storage** model that is maintained as part of the system state - this memory is

organized into word arrays (as opposed to word-addressable byte arrays in the stack). This storage is where contracts keep persistent data - a contract can only interact with its own storage. Storage is organized in key-value mappings.

Although it is not mentioned in this section of the Yellow Paper, it is also useful to know there is a fourth type of memory. **Calldata** is byte-addressable read-only memory used to store the value passed with the `data` parameter of a transaction. The EVM has specific opcodes for managing `calldata`. `calldatasize` returns the size of the data. `calldataload` loads the data into the stack. `calldatacopy` copies the data into memory.

The standard Von Neumann architecture stores code and data in the same memory. The EVM does not follow this standard for security reasons - sharing volatile memory makes it possible to change program code. Instead, code is saved to storage.

There are only two cases in which code is executed from memory:

- When a contract creates another contract (using `CREATE` or `CREATE2`), the code for the contract constructor comes from memory.
- During the creation of *any* contract, the constructor code runs and then returns with the code of the actual contract, also from memory.

The term exceptional execution means an exception that causes the execution of the current contract to halt.

9.2 Fees overview {#92-fees-overview}

This section explains how the gas fees are calculated. There are three costs:

Opcode cost {#opcode-cost}

The inherent cost of the specific opcode. To get this value, find the cost group of the opcode in Appendix H (p. 28, under equation (327)), and find the cost group in equation (324). This gives you a cost function, which in most cases uses parameters from Appendix G (p. 27).

For example, the opcode `CALLDATACOPY` is a member of group W_{copy}. The opcode cost for that group is $G_{verylow}+G_{copy}\times\lceil\mu_s[2]\div32\rceil$. Looking at Appendix G, we see that both constants are 3, which gives us $3+3\times\lceil\mu_s[2]\div32\rceil$.

We still need to decipher the expression $\lceil\mu_s[2]\div32\rceil$. The outmost part, $\lceil\ \langle value\rangle\ \rceil$ is the ceiling function, a function that given a value returns the smallest integer that is still not smaller than the value. For example, $\lceil2.5\rceil=\lceil3\rceil=3$. The inner part is $\mu_s[2]\div32$. Looking at section 3 (Conventions) on p. 3, μ is the machine state. The machine state is defined in section 9.4.1 on p. 13. According to that section, one of the machine state parameters is s for the stack. Putting it all together, it seems that $\mu_s[2]$ is location #2 in the stack. Looking at the opcode, location #2 in the stack is the size of the data in bytes. Looking at the other opcodes in group W_{copy}, `CODECOPY` and `RETURNDATACOPY`, they also have a size of data in the same location. So $\lceil\mu_s[2]\div32\rceil$ is the number of 32 byte words required to store the data being copied. Putting everything together, the inherent cost of `CALLDATACOPY` is 3 gas plus 3 per word of data being copied.

Running cost {#running-cost}

The cost of running the code we're calling.

- In the case of `CREATE` and `CREATE2`, the constructor for the new contract.
- In the case of `CALL`, `CALLCODE`, `STATICCALL`, or `DELEGATECALL`, the contract we call.

Expanding memory cost {#expanding-memory-cost}

The cost of expanding memory (if necessary).

In equation 324, this value is written as $C_{mem}(\mu_i')-C_{mem}(\mu_i)$. Looking at section 9.4.1 again, we see that μ_i is the number of words in memory. So μ_i is the number of words in memory before the opcode and μ_i' is the number of words in memory after the opcode.

The function C_{mem} is defined in equation 326: $C_{mem}(a)=G_{memory}\times a+\lfloor a^2\div512\rfloor$. $\lfloor x\rfloor$ is the floor function, a function that given a value returns the largest integer that is still not larger than the value. For example, $\lfloor2.5\rfloor=\lfloor2\rfloor=2$. When $a<\sqrt{512}$, $a^2<512$,

and the result of the floor function is zero. So for the first 22 words (704 bytes), the cost rises linearly with the number of memory words required. Beyond that point $\lfloor a^2 \div 512 \rfloor$ is positive. When the memory required is high enough the gas cost is proportional to the square of the amount of memory.

Note that these factors only influence the *inherent* gas cost - it does not take into account the fee market or tips to validators that determine how much an end user is required to pay - this is just the raw cost of running a particular operation on the EVM.

Read more about gas.

9.3 Execution environment {#93-execution-env}

The execution environment is a tuple, I, that includes information that isn't part of the blockchain state or the EVM.

| Parameter | Opcode to access the data | Solidity code to access the data | | ---------- | ---------- | ---------- | | I_a | ADDRESS | `address(this)` | | I_o | ORIGIN | `tx.origin` | I_p | GASPRICE | `tx.gasprice` | | I_d | CALLDATALOAD, etc. | `msg.data` | | I_s | CALLER | `msg.sender` | | I_v | CALLVALUE | `msg.value` | | I_b | CODECOPY | `address(this).code` | | I_H | Block header fields, such as NUMBER and DIFFICULTY | `block.number`, `block.difficulty`, etc. | | I_e | Depth of the call stack for calls between contracts (including contract creation) | | I_w | Is the EVM allowed to change state, or is it running statically |

A few other parameters are necessary to understand the rest of section 9:

| Parameter | Defined in section | Meaning | | --------- | ---------- | ---------- | | σ | 2 (p. 2, equation 1) | The state of the blockchain | | g | 9.3 (p. 13) | Remaining gas | | A | 6.1 (p. 8) | Accrued substate (changes scheduled for when the transaction ends) | | o | 9.3 (p. 13) | Output - the returned result in the case of internal transaction (when one contract calls another) and calls to view functions (when you are just asking for information, so there is no need to wait for a transaction) |

9.4 Execution overview {#94-execution-overview}

Now that have all the preliminaries, we can finally start working on how the EVM works.

Equations 137-142 give us the initial conditions for running the EVM:

| Symbol | Initial value | Meaning | | ---------- | ---------- | ---------- | | μ_g | g | Gas remaining | | μ_{pc} | 0 | Program counter, the address of the next instruction to execute | | μ_m | $(0, 0, ...)$ | Memory, initialized to all zeros | | μ_i | 0 | Highest memory location used | | μ_s | $()$ | The stack, initially empty | | μ_o | $\varnothing$ | The output, empty set until and unless we stop either with return data (RETURN or REVERT) or without it (STOP or SELFDESTRUCT). |

Equation 143 tells us there are four possible conditions at each point in time during execution, and what to do with them:

1. `Z(σ,μ,A,I)`. Z represents a function that tests whether an operation creates an invalid state transition (see exceptional halting). If it evaluates to True, the new state is identical to the old one (except gas gets burned) because the changes have not been implemented.
2. If the opcode being executed is REVERT, the new state is the same as the old state, some gas is lost.
3. If the sequence of operations is finished, as signified by a RETURN), the state is updated to the new state.
4. If we aren't at one of the end conditions 1-3, continue running.

9.4.1 Machine State {#941-machine-state}

This section explains the machine state in greater detail. It specifies that w is the current opcode. If μ_{pc} is less than $||I_b||$, the length of the code, then that byte ($I_b[\mu_{pc}]$) is the opcode. Otherwise, the opcode is defined as STOP.

As this is a stack machine, we need to keep track of the number of items popped out (δ) and pushed in (α) by each opcode.

9.4.2 Exceptional Halting {#942-exceptional-halt}

This section defines the Z function, which specifies when we have an abnormal termination. This is a Boolean function, so it uses ∨ for a logical or and ∧ for a logical and.

We have an exceptional halt if any of these conditions is true:

- $\mu_g < C(\sigma,\mu,A,I)$ As we saw in section 9.2, C is the function that specifies the gas cost. There isn't enough gas left to cover the next opcode.
- $\delta_w=\emptyset$ If the number of items popped for an opcode is undefined, then the opcode itself is undefined.
- $||\mu_s|| < \delta_w$ Stack underflow, not enough items in the stack for the current opcode.
- $w = JUMP \wedge \mu_s[0] \notin D(I_b)$ The opcode is JUMP and the address is not a JUMPDEST. Jumps are *only* valid when the destination is a JUMPDEST.
- $w = JUMPI \wedge \mu_s[1] \neq 0 \wedge \mu_s[0] \notin D(I_b)$ The opcode is JUMPI, the condition is true (non zero) so the jump should happen, and the address is not a JUMPDEST. Jumps are *only* valid when the destination is a JUMPDEST.
- $w = RETURNDATACOPY \wedge \mu_s[1]+\mu_s[2] > ||\mu_o||$ The opcode is RETURNDATACOPY. In this opcode stack element $\mu_s[1]$ is the offset to read from in the return data buffer, and stack element $\mu_s[2]$ is the length of data. This condition occurs when you try to read beyond the end of the return data buffer. Note that there isn't a similar condition for the calldata or for the code itself. When you try to read beyond the end of those buffers you just get zeros.
- $||\mu_s|| - \delta_w + \alpha_w > 1024$

Stack overflow. If running the opcode will result in a stack of over 1024 items, abort.

- $\neg I_w \wedge W(w,\mu)$ Are we running statically (¬ is negation and I_w is true when we are allowed to change the blockchain state)? If so, and we're trying a state changing operation, it can't happen.

The function $W(w,\mu)$ is defined later in equation 150. $W(w,\mu)$ is true if one of these conditions is true:

- $w \in \{CREATE, CREATE2, SSTORE, SELFDESTRUCT\}$ These opcodes change the state, either by creating a new contract, storing a value, or destroying the current contract.
- $LOG0 \leq w \wedge w \leq LOG4$ If we are called statically we cannot emit log entries. The log opcodes are all in the range between LOG0 (A0) and LOG4 (A4). The number after the log opcode specifies how many topics the log entry contains.
- $w=CALL \wedge \mu_s[2] \neq 0$ You can call another contract when you're static, but if you do you cannot transfer ETH to it.
- $w = SSTORE \wedge \mu_g \leq G_{callstipend}$ You cannot run SSTORE unless you have more than $G_{callstipend}$ (defined as 2300 in Appendix G) gas.

9.4.3 Jump Destination Validity {#943-jump-dest-valid}

Here we formally define what are the JUMPDEST opcodes. We cannot just look for byte value 0x5B, because it might be inside a PUSH (and therefore data and not an opcode).

In equation (153) we define a function, $N(i,w)$. The first parameter, i, is the opcode's location. The second, w, is the opcode itself. If $w \in [PUSH1, PUSH32]$ that means the opcode is a PUSH (square brackets define a range that includes the endpoints). If that case the next opcode is at $i+2+(w-PUSH1)$. For PUSH1 we need to advance by two bytes (the PUSH itself and the one byte value), for PUSH2 we need to advance by three bytes because it's a two byte value, etc. All other EVM opcodes are just one byte long, so in all other cases $N(i,w)=i+1$.

This function is used in equation (152) to define $D_J(c,i)$, which is the set of all valid jump destinations in code c, starting with opcode location i. This function is defined recursively. If $i \geq ||c||$, that means that we're at or after the end of the code. We are not going to find any more jump destinations, so just return the empty set.

In all other cases we look at the rest of the code by going to the next opcode and getting the set starting from it. $c[i]$ is the current opcode, so $N(i,c[i])$ is the location of the next opcode. $D_J(c,N(i,c[i]))$ is therefore the set of valid jump destinations that starts at

the next opcode. If the current opcode isn't a `JUMPDEST`, just return that set. If it is `JUMPDEST`, include it in the result set and return that.

9.4.4 Normal halting {#944-normal-halt}

The halting function *H*, can return three types of values.

- If we aren't in a halt opcode, return∅, the empty set. By convention, this value is interpreted as Boolean false.
- If we have a halt opcode that doesn't produce output (either`STOP` or `SELFDESTRUCT`), return a sequence of size zero bytes as the return value. Note that this is very different from the empty set. This value means that the EVM really did halt, just there's no return data to read.
- If we have a halt opcode that does produce output (either`RETURN` or `REVERT`), return the sequence of bytes specified by that opcode. This sequence is taken from memory, the value at the top of the stack ($\mu_s[0]$) is the first byte, and the value after it ($\mu_s[1]$) is the length.

H.2 Instruction set {#h2-instruction-set}

Before we go to the final subsection of the EVM, 9.5, let's look at the instructions themselves. They are defined in Appendix H.2 which starts on p. 29. Anything that is not specified as changing with that specific opcode is expected to stay the same. Variables that do change are specified with as \<something>′.

For example, let's look at the`ADD` opcode.

| Value | Mnemonic | δ | α | Description | | ----: | -------- | --- | --- | --- | | 0x01 | ADD | 2 | 1 | Addition operation. | | | | | | $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$ |

δ is the number of values we pop from the stack. In this case two, because we are adding the top two values.

α is the number of values we push back. In this case one, the sum.

So the new stack top ($\mu'_s[0]$) is the sum of the old stack top ($\mu_s[0]$) and the old value below it ($\mu_s[1]$).

Instead of going over all the opcodes with an "eyes glaze over list", This article explains only those opcodes that introduce something new.

| Value | Mnemonic | δ | α | Description | | ----: | --------- | --- | --- | --- | | 0x20 | KECCAK256 | 2 | 1 | Compute Keccak-256 hash. | | | | | | $\mu'_s[0] \equiv KEC(\mu_m[\mu_s[0] \ldots (\mu_s[0] + \mu_s[1] - 1)])$ | | | | | | $\mu'_i \equiv M(\mu_i, \mu_s[0], \mu_s[1])$ |

This is the first opcode that accesses memory (in this case, read only). However, it might expand beyond the current limits of the memory, so we need to update μ_i. We do this using the*M* function defined in equation 328 on p. 29.

| Value | Mnemonic | δ | α | Description | | ----: | -------- | --- | --- | -------------------------------- | | 0x31 | BALANCE | 1 | 1 | Get balance of the given account. | | | | | | ... |

The address whose balance we need to find is$\mu_s[0] \bmod 2^{160}$. The top of the stack is the address, but because addresses are only 160 bits, we calculate the value modulo 2^{160}.

If $\sigma[\mu_s[0] \bmod 2^{160}] \neq \varnothing$, it means that there is information about this address. In that case,$\sigma[\mu_s[0] \bmod 2^{160}]_b$ is the balance for that address. If $\sigma[\mu_s[0] \bmod 2^{160}] = \varnothing$, it means that this address is uninitialized and the balance is zero. You can see the list of account information fields in section 4.1 on p. 4.

The second equation, $A'_a \equiv A_a \cup \{\mu_s[0] \bmod 2^{160}\}$, is related to the difference in cost between access to warm storage (storage that has recently been accessed and is likely to be cached) and cold storage (storage that hasn't been accessed and is likely to be in slower storage that is more expensive to retrieve). A_a is the list of addresses previously accessed by the transaction, which should therefore be cheaper to access, as defined in section 6.1 on p. 8. You can read more about this subject in EIP-2929.

| Value | Mnemonic | δ | α | Description | | ----: | -------- | --- | --- | -------------------------------------- | | 0x8F | DUP16 | 16 | 17 | Duplicate 16th stack item. | | | | | | $\mu'_s[0] \equiv \mu_s[15]$ |

Note that to use any stack item, we need to pop it, which means we also need to pop all the stack items on top of it. In the case of DUP<n> and SWAP<n>, this means having to pop and then push up to sixteen values.

9.5 The execution cycle {#95-exec-cycle}

Now that we have all the parts, we can finally understand how the execution cycle of the EVM is documented.

Equation (155) says that given the state:

- σ (global blockchain state)
- μ (EVM state)
- A (substate, changes to happen when the transaction ends)
- I (execution environment)

The new state is (σ', μ', A', I').

Equations (156)-(158) define the stack and the change in it due to an opcode (μ_s). Equation (159) is the change in gas (μ_g). Equation (160) is the change in the program counter (μ_{pc}). Finally, equations (161)-(164) specify that the other parameters stay the same, unless explicitly changed by the opcode.

With this the EVM is fully defined.

Conclusion {#conclusion}

Mathematical notation is precise and has allowed the Yellow Paper to specify every detail of Ethereum. However, it does have some drawbacks:

- It can only be understood by humans, which means thatcompliance tests must be written manually.
- Programmers understand computer code. They may or may not understand mathematical notation.

Maybe for these reasons, the newer consensus layer specs are written in Python. There areexecution layer specs in Python, but they are not complete. Until and unless the entire Yellow Paper is also translated to Python or a similar language, the Yellow Paper will continue in service, and it is helpful to be able to read it.