The Ethereum Virtual Machine (EVM)

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256-bit. The stack size is limited to 1,024 elements and is based on the **Last In, First Out** (**LIFO**) queue. The EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial-of-service attacks are not possible due to gas requirements. The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

The EVM is an entirely isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources such as a network or filesystem. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on Ethereum blockchain.

EVM is a stack-based architecture. The EVM is big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

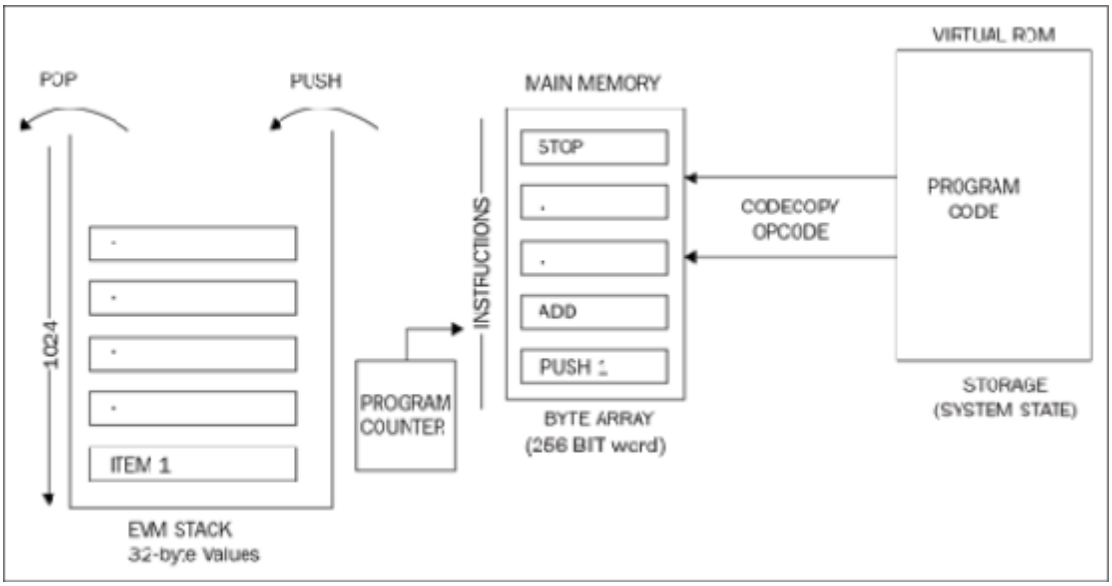
There are three main types of storage available for contracts and the EVM:

- **Memory**: The first type is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. write operations to the memory can be of 8 or 256 bits, whereas read operations are limited to 256-bit words. Memory is unlimited but constrained by gas fee requirements.
- **Storage**: The other type is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.
- **Stack**: EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1024 elements and supports the word size of 256 bits.

The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage. The program code is stored in **virtual read-only memory** (**virtual ROM**) that is accessible using the CODECOPY instruction. The CODECOPY instruction copies the program code into the main memory. Initially, all storage and memory are set to zero in the EVM.

The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into the main memory using the CODECOPY instruction. The main memory is then read by the EVM by referring to the program counter and executes instructions step by

step. The program counter and EVM stack are updated accordingly with each instruction execution:



EVM operation

The preceding diagram shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained, and is incremented with instructions being read from the main memory. The main memory gets the program code from the virtual ROM/storage via the CODECOPY instruction.

EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance. Research and development on **Ethereum WebAssembly** (**ewasm**)—an Ethereum-flavored iteration of WebAssembly—is already underway. **WebAssembly** (**Wasm**) was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group. Wasm aims to be able to run machine code in the browser that will result in execution at native speed.

Another intermediate language called YUL, which can compile to various backends such as the EVM and ewasm, is under development.

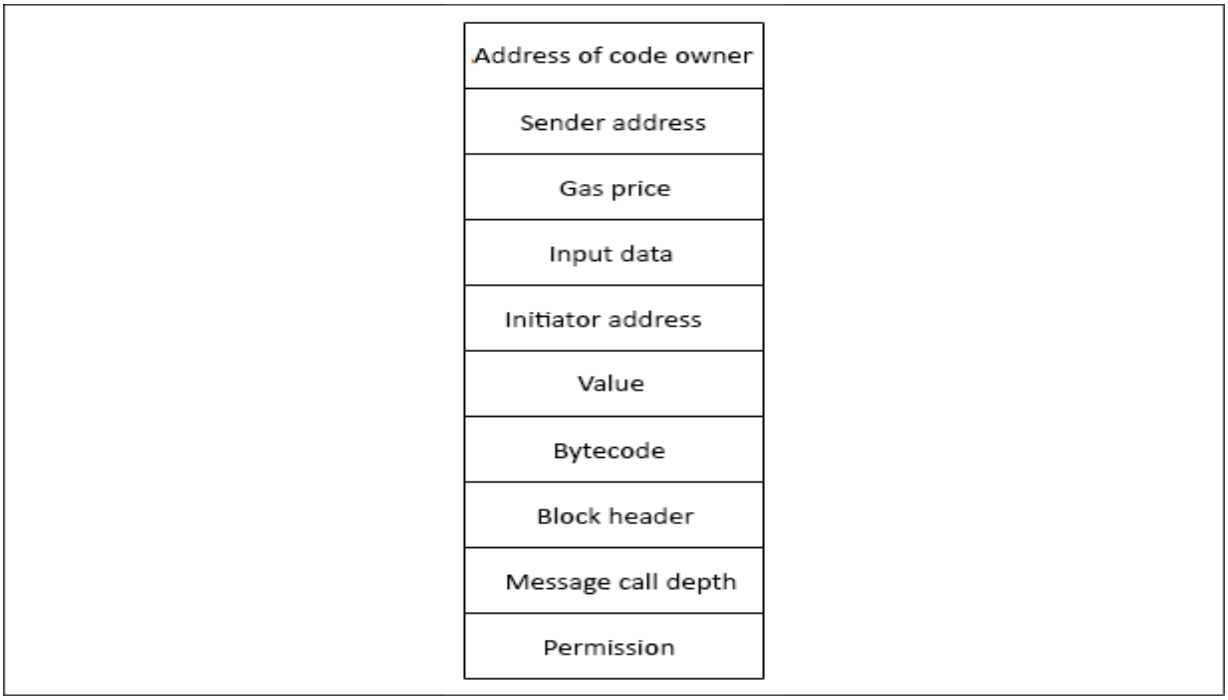
Execution environment

There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent; for example, a transaction. These are listed as follows:

- The system state.
- The remaining gas for execution.
- The address of the account that owns the executing code.

- The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).
- The gas price of the transaction that initiated the execution.
- Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
- The address of the account that initiated the code execution or transaction sender. This is the address of the sender in case the code execution is initiated by a transaction; otherwise, it is the address of the account.
- The value or transaction value. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
- The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
- The block header of the current block.
- The number of message calls or contract creation transactions (CALLs, CREATEs or CREATE2s) currently in execution.
- Permission to make modifications to the state.

The execution environment can be visualized as a tuple of ten elements, as follows:



The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

The machine state is also maintained internally, and updated after each execution cycle of the EVM. An iterator function (detailed in the next section) runs in the EVM, which outputs the results of a single cycle of the state machine.

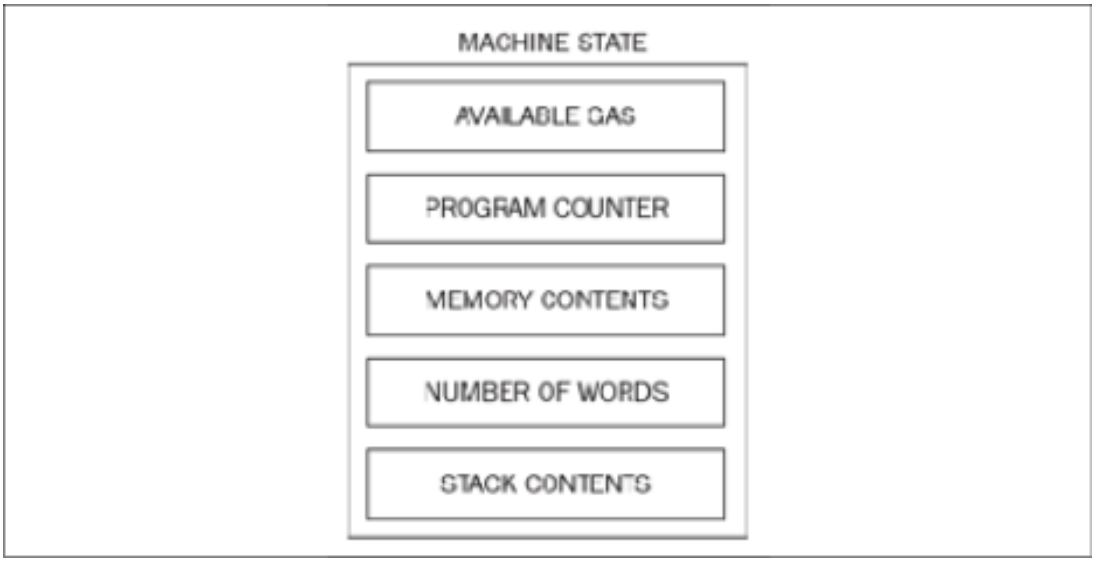
The machine state is a tuple that consists of the following elements:

- Available gas
- The program counter, which is a positive integer of up to 256
- The contents of the memory (a series of zeroes of size 2^{256})
- The active number of words in memory (counting continuously from position 0)
- The contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions should occur:

- Not having enough gas required for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump opcodes
- Invalid stack size (greater than 1,024)

Machine state can be viewed as a tuple, as shown in the following diagram:



Machine state tuple

The iterator function

The iterator function mentioned earlier performs various vital functions that are used to set the next state of the machine and eventually the world state. These functions include the following:

- It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
- It adds/removes (PUSH/POP) items from the stack accordingly.
- Gas is reduced according to the gas cost of the instructions/opcodes. It increments the **Program Counter** (**PC**).

The EVM is also able to halt in normal conditions if STOP, SUICIDE, or RETURN opcodes are encountered during the execution cycle.