Chapter 13: The Ethereum Virtual Machine (EVM).

The EVM is the heart of the Ethereum protocol. The EVM is the part of Ethereum that handles smart contract deployment and execution.

At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store.

The Ethereum world computer is completely virtual.

The EVM instruction set offers most of the operations you might expect, including:
Arithmetic and bit-wise logic operations.
Execution context inquiries.
Stack, memory, and storage access.
Control flow operations.
Logging, calling, and other operators.

In addition to the typical bytecode operations, the EVM also has access to account information (e.g., address and balance) and block information (e.g., block number and current gas price).

The EVM is Quasi-Turing-Complete, because, as a world computer running computation, storing and retrieving data, someone might maliciously or accidentally, trigger a transactional loop that will run forever and halt the EVM, hence, the EVM being quasi, it allocates 'gas' and 'gas limits' for some specific transaction, and once the gas is used up to its limit, the transaction fails and reverts.

The EVM has a stack-based architecture, storing all in-memory values on a stack. It works with a word size of 256 bits (mainly to facilitate native hashing and elliptic curve operations) and has several addressable data components.

The EVM has 3 sections:
Immutable ROM (program).
Volatile Memory (memory).
Permanent storage (storage).

The available opcodes on the EVM can be divided into the following categories:

Arithmetic Opcodes.
Stack, memory and storage opcodes.
Process flow opcodes.
System opcodes.
Logic opcodes.
Environmental opcodes.
Block opcodes.

An EOA (wallet) will always have no code and an empty storage.

A key variable is the gas supply for this execution, which is set to the amount of gas paid for by the sender at the start of the transaction (see “Gas” for more details). As code execution progresses, the gas supply is reduced according to the gas cost of the operations executed. If at any point the gas supply is reduced to zero we get an “Out of Gas” (OOG) exception; execution immediately halts and the transaction is abandoned.

At this point, you can think of the EVM running on a sandboxed copy of the Ethereum world state, with this sandboxed version being discarded completely if execution cannot complete for whatever reason.

With new transactions, an instance of the EVM is created and states are updated with the level above, then, the gas is allocated for the execution and should it fail, the instantiation is discarded and should it pass, the states from the level above are updated.

Generating the raw opcode stream of a Solidity source file is easily achieved with the --opcodes command-line option. This opcode stream leaves out some information (the --asm option produces the full information), but it is sufficient for this discussion. For example, compiling an example Solidity file, Example.sol, and sending the opcode output into a directory named BytecodeDir is accomplished with the following command:

$ solc -o BytecodeDir --opcodes Example.sol
or:
$ solc -o BytecodeDir --asm Example.sol

Let's examine the first 4 instructions:
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE

1 byte = 0xAB
2 bytes = 0xABCD

And it keeps doubling on and on.

The first argument for MSTORE is the address of the word in memory where the value to be saved will be put.

The next opcode is CALLVALUE, which is an environmental opcode that pushes onto the top of the stack the amount of ether (measured in Wei) sent with the message call that initiated this execution.

In order to create a new contract, a special transaction is needed that has its to field set to the special 0x0 address and its data field set to the contract's initiation code. When such a contract creation transaction is processed, the code for the new contract account is not the code in the data field of the transaction. Instead, an EVM is instantiated with the code in the data field of the transaction loaded into its program code ROM, and then the output of the execution of that deployment code is taken as the code for the new contract account.

When compiling a contract offline, e.g., using solc on the command line, you can either get the deployment bytecode or the runtime bytecode.

Deployment bytecode: Bytecode used for deployment or initializing the EVM to create a new contract account. This contains the entire runtime bytecode.

Runtime bytecode: Bytecode ran whenever an account calls a transaction to a deployed contract. This is entirely contained in the deployment bytecode.

CALLDATASIZE gets the size in bytes of the data sent with the transaction.

The next instruction is JUMPI, which stands for "jump if". It works like so:
jumpi(label, cond) // Jump to "label" if "cond" is true

As we have already touched on, in simple terms, a system or programming language is Turing complete if it can run any program.

Gas is Ethereum's unit for measuring the computational and storage resources required to perform actions on the Ethereum blockchain.

When an EVM is needed to complete a transaction, in the first instance it is given a gas supply equal to the amount specified by the gas limit in the transaction. Every opcode that is executed has a cost in gas, and so the EVM's gas supply is reduced as the EVM steps through the program. Before each operation, the EVM checks that there is enough gas to pay for the operation's execution. If there isn't enough gas, execution is halted and the transaction is reverted.

If the EVM reaches the end of execution successfully, without running out of gas, the gas cost used is paid to the miner as a transaction fee, converted to ether based on the gas price specified in the transaction:

miner fee = gas cost * gas price

The gas remaining in the gas supply is refunded to the sender, again converted to ether based on the gas price specified in the transaction:

remaining gas = gas limit - gas cost
refunded ether = remaining gas * gas price

transaction fee = total gas used * gas price paid (in ether)

Gas cost is the number of units of gas required to perform a particular operation.

Gas price is the amount of ether you are willing to pay per unit of gas when you send your transaction to the Ethereum network.

SELFDESTRUT refunds 24,000 gas.
DELETE refunds 15,000 gas.

The block gas limit is the maximum amount of gas that may be consumed by all the transactions in a block, and constrains how many transactions can fit into a block.

Current Block Gas Limit is 30 million gas ~~ 1,425 transactions of 21,000 gas each.