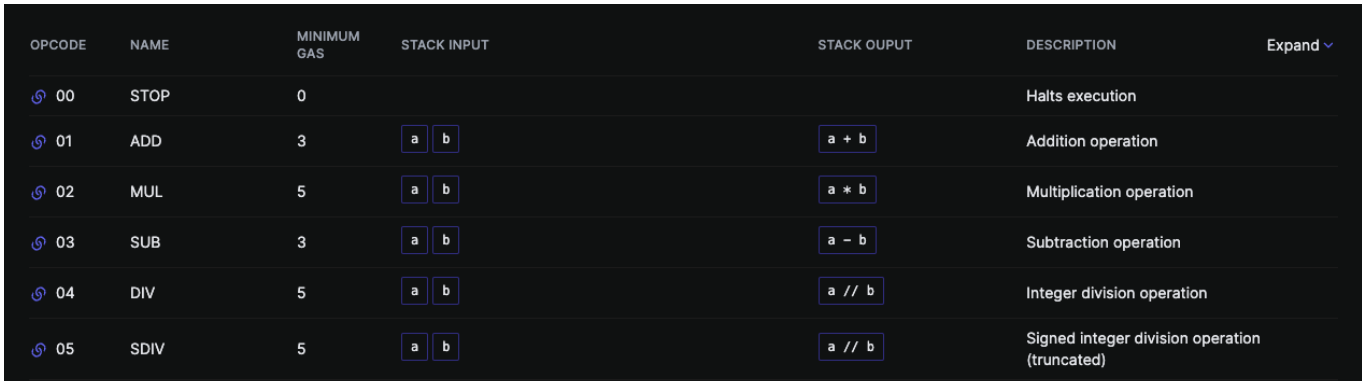
Operating Codes (Op Codes)

Each operation that is executed on the EVM has a cost, denoted in gas. There are 256 possible operating codes defined on the EVM beginning at 00 which is the **STOP** code, and ending with FF (denoted in hexadecimal) which is the **SELFDESTRUCT** code. The **STOP** opcode has a gas cost of 0, while the **SELFDESTRUCT** opcode has a gas cost of 5000.

Gas is measured in generic units and the total gas is provided prior to the transaction execution. As each operation is executed, the gas cost is deducted from the total gas provided to pay for the transaction.

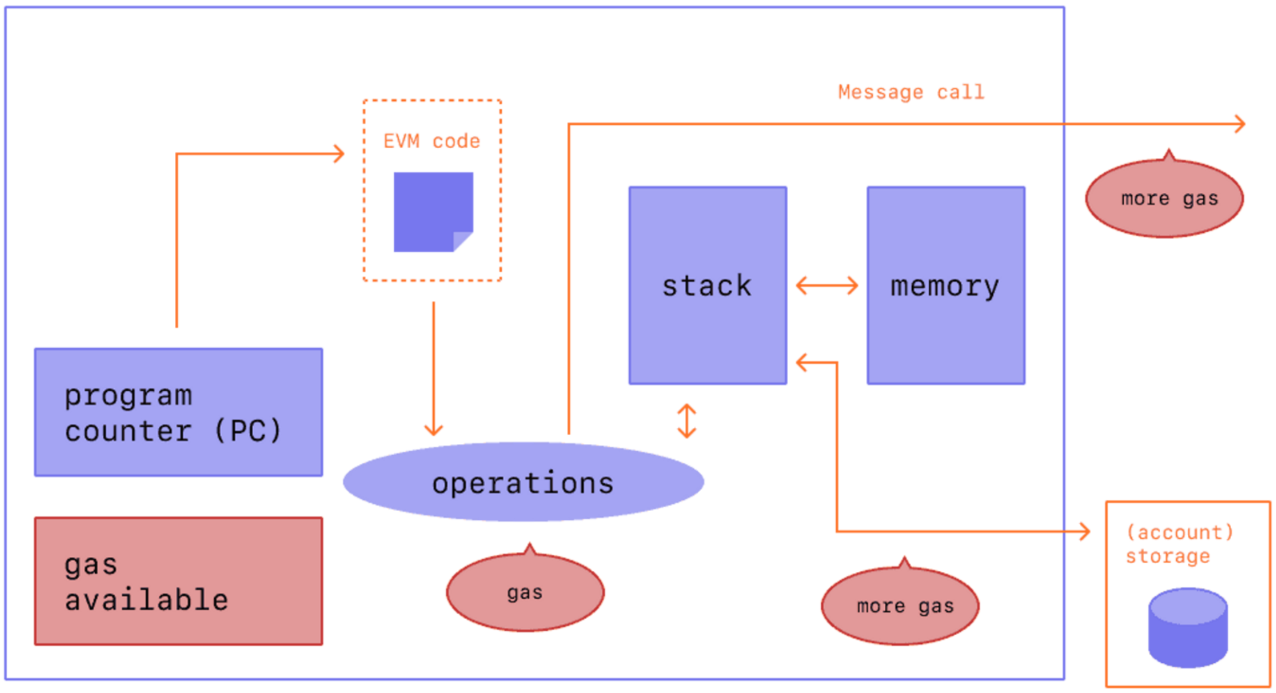


OPCODE	NAME	MINIMUM GAS	STACK INPUT	STACK OUPUT	DESCRIPTION
00	STOP	0			Halts execution
01	ADD	3	a b	a + b	Addition operation
02	MUL	5	a b	a * b	Multiplication operation
03	SUB	3	a b	a - b	Subtraction operation
04	DIV	5	a b	a // b	Integer division operation
05	SDIV	5	a b	a // b	Signed integer division operation (truncated)

Cost

This transaction fee is calculated by multiplying the total cost of all opcodes expected to be executed during the transaction, by the current gas price:

cost = gas units (limit) * gas price per unit



There are a few parameters to pay attention to when submitting a transaction

Base Fee: The minimum fee per unit of gas necessary for block inclusion. This fee is calculated based on the historical amount of gas used from previous transactions. With the London Upgrade, this base fee is burned when the block is mined

Priority Fee: This fee is a tip to the miners to incentivize the prioritization of transaction inclusion

Max Fee: This is the upper bound a user is willing to pay for their transaction to be executed. The gas returned is based on the following formula:

return = max fee - (base fee + priority fee)

Transactions

It is important to understand that if there is any error during the execution of these opcodes the entire state will revert to the state prior to execution, and ALL gas used during the execution of the transaction up to the error will be consumed.

This gas is not returned to the caller or initiator of the transaction. The gas is paid to miners/validators to perform each opcode, and as that opcode has run, the gas fee for that opcode is passed on.

In this way, gas is a consumable resource that does not get returned once consumed.

Reverts

It is also the case that a transaction may run out of gas during its execution. In this case, all gas is consumed, with nothing going back to the initiator of the transaction. The transaction will **revert**, and all state that was changed during this transaction will revert to just prior to the transaction. The transaction must have enough gas provided to cover all computations required to execute all opcodes associated with the request. If the transaction is successful, and there is leftover gas, this gas will be returned to the initiator of the transaction.

Ethereum Upgrades

Another important consideration is the upgrades that the Ethereum network has undergone over its lifetime. The most recent upgrade of the network changed its consensus mechanism.

The Merge

The upgrade from what is termed Eth1 to Eth2 changed the way that the network decided on the world state of the network. Eth1 used computational work done by "miners" (proof-of-work) to add new blocks to the Ethereum blockchain.

In Eth2, the consensus mechanism has transitioned to "proof-of-stake" where validators who each have staked 32 ether are randomly selected to contribute the next block to the blockchain.

London Upgrade

Another impactful change was the London upgrade (EIP-1559) which burns the gas base fee pushing ETH into a deflationary state as well as helping to make gas fees more predictable.

In addition, a priority fee or "tip" can be provided in the transaction fee, changing the calculation

Solidity

Ethereum is constantly evolving which is exciting but can also make it difficult to keep up to date on all the changes to the protocol. Solidity, the language used to program smart contracts on Ethereum, is also changing. Although Solidity still has not had a major release, minor release candidates are consistently being added. As of the time of this writing the current version of Solidity is 0.8.18. For more information on the Solidity language.

The Ethereum Virtual Machine is a key component of the Ethereum network, responsible for executing smart contracts, managing the state of the network, and ensuring that all nodes on the network have a consistent copy of the blockchain.

Gas Optimization

```
Sample Code

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

    // Function to increment count by 1
    function inc() public {
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {
        count -= 1;
    }

    function countUp(uint _increment) public {
        for (uint256 i = 0; i < _increment; i++) {
            inc();
        }
    }

    function countUp2(uint _increment) public {
        for (uint256 i ; i < _increment;) {
            inc();
            unchecked { i = i + 1; }
        }
    }
}
```

In our augmented Counter contract, there are two new functions that use a *for* loop to repeatedly call the *inc()* function. There are a few subtle differences between the functions *countUp()* and *countUp2()*. They both take in one uint256 parameter, and both call the public function *inc()*, but the declaration of the loop parameters and the incrementation is different.

Differences between *countUp()* and *countUp2()*

We notice that *countUp2()* does not initialize the incrementer *i*, and its *for* loop doesn't use the post incrementer, *i*++, but rather the *unchecked* keyword with an explicit assignment or *i + 1*.

Explanation of *unchecked* keyword

Some of these changes are specific to how the compiler optimizes instructions, however, the keyword unchecked is a special signal to the compiler to avoid an arithmetic check for overflows. With the current version of Solidity, the compiler will automatically check for integer overflow in arithmetic operations.

This extra checking is a security mechanism to help guard against overflow errors that have caused great havoc in the past. This extra checking costs more in gas, however, as there are more instructions used to execute the extra checks. We can tell the compiler, “we know what we are doing, this operation will never overflow, or we don’t care if it does, so skip the checks.” In doing so, we save a bit of gas.