CS4998: Blockchain Development

The Ethereum Virtual Machine
By Rodrigo Villar

> "The concept of an arbitrary state transition function as implemented by the Ethereum protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, Ethereum is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come."
> - Vitalik Buterin, *The Ethereum Whitepaper*

Introduction

At this point of the course, you're become familiar with the Solidity syntax and writing basic smart contracts. Therefore, it's appropriate that we finally cover the machine that runs your smart contract code: the Ethereum Virtual Machine (EVM). Understanding the EVM serves as a foundation for understanding the security and gas optimization practices that Solidity development requires. Furthermore, learning about the EVM will allow us to introduce *Yul*, a language used within Solidity that allows for low-level programming (in contrast to Solidity's high-level programming).

Before discussing the internals of the EVM, it is important to note that the EVM is an *architecture*. There does not exist an EVM; rather, there exists multiple implementations of the EVM (i.e. execution clients) which implement the necessary functionality for the client to compute data from the blockchain. Outside of this though, each execution client has the freedom to implement their version of the EVM as they wish (an obvious example of this is the language which an execution client is written in).

State and Transactions

Recall from the Ethereum primer that although we refer to Ethereum as being a blockchain, the blockchain is *not* the only component of Ethereum. Rather, the blockchain is a data structure that allows us to update and store [1] the state of Ethereum. Ethereum is better described as a distributed computer and like all computers, there are several components that define the architecture of a computer. In the following section, we will examine the components of the EVM that is used during state transitions.

At its most basic level, the EVM can be described as follows:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

where σ_t, σ_{t+1} are the previous and newly computed states, respectively. Υ is the state transition function which takes as input the current state and a transaction T. In reality, Υ is the EVM; given a transaction and the current state of the blockchain, the EVM in most cases modifies the state of Ethereum, in essence outputting a new state.

World State vs Machine State

Although it is clear that Ethereum is a *stateful* machine, we must differentiate between the two types of state found within Ethereum: the world state and the machine state. The machine state is discussed in detail in the components section, so we'll be focusing on the world state for the rest of this section.

[1] By storing, we mean the transactions that we can then use to derive the desired state σ_n

The world state is a mapping from account addresses (i.e. 160-bit values) to account states, usually implemented as a Merkle Patricia Trie. Within account states, we can find the following four attributes:

- Balance: how much ether an account has (denominated in Wei)
- Nonce: the current transaction count of an account
- Code Hash: if the account is a smart contract, the hash of its code will be found here
- Storage Hash: if the account is a smart contract, the hash of its storage mapping will be found here

Code and storage mapping are stored in separate Merkle Patricia Tries, and so by just storing a contract's code/storage hash, the EVM is able to fetch[2] the storage mapping/ROM necessary for execution.

Components

Before discussing the individual components of the EVM, it is important to discuss the two most frequently used data units: a byte refers to an 8-bit value while a word refers to a 32-byte value.

Note: stack, memory, the program counter, and the gas counter are all part of the *machine state*, while account storage is not part of the machine state.

Stack

Similar to the stack seen in CS2110, the stack is a *last-in first-out* data structure; each element of the stack is 32 bytes long. The stack has a maximum size of 1024; exceeding this size or accessing an nonexistent element within the stack will result in the transaction reverting.

Memory

Memory is byte-addressable that is, in theory [3], of size 2^{256} bytes. It is important to note that memory is local relative to a message; whenever the EVM switches to a different memory context, the EVM changes to a different memory array (either an all-zero array or the already existing array for the message). At initialization, all memory elements are set equal to 0.

Account Storage

Account storage is the storage mapping of the executing contract. Like all storage mapping, account storage maps 32 bytes values to 32 bytes values. Account storage is where we can find the state variables of the executing smart contract. Note however, that although account storage is part of the EVM environment, it is not part of the machine state since storage is persistent across transactions.

ROM/Contract Bytecode

Contract bytecode (ROM) can be best described via an example; consider the following contract:

[2]Note that this does not imply that in a given instance of the EVM, the EVM cannot directly access the state variable of an external contract. The EVM can only access an external contract's code and the length of said code.

[3]Doing so in practice would be *very very expensive*

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;

contract Basic {}
```

It's obvious that this contract does nothing, but there is still bytecode that is produced for this contract when compiled. This same bytecode[4] is what represents the contract on-chain. The contract bytecode contains all the logic that is used whenever a transaction calls said contract. Whenever a transaction whose "to" address is a smart contract, said contract's bytecode is loaded into the EVM's ROM; it is this code that is executed during a transaction.

As suggested in the name read-only memory (ROM), a smart contract's bytecode is *immutable* - once initialized, it cannot be changed.

Program Counter (PC)

The program counter tells the EVM what line of the contract bytecode to execute. This is explained in detail in the operations section.

Gas Counter

The gas counter tracks how much gas the current context of the EVM has. If there is not enough gas, the transaction will revert.

Operations

It would be quite boring if all the EVM did was load in the relevant data within doing anything. Therefore, the EVM contains operations that allow for the manipulation of data within the execution context. The following is the grouping of operations as listed in the Ethereum Yellowpaper:

- Stop and Arithmetic Operations: group of arithmetic operations that manipulate elements on the stack
- Comparison/Bitwise Operations: group of comparison and bitwise operations that manipulate elements on the stack
- Keccak: hashing function that hashes a segment of memory
- Environmental Operations: fetches values related to the current transaction, current message, current executing contract, bytecode of an external contract, or the return value of a previous call
- Block Operations: fetches values related to the current block
- Stack, Memory, Storage, and Flow Operations: manipulates the stack, memory, storage, and program counter
- Push, Duplication, and Exchange Operations: operations that manipulate the stack
- Logging Operations: operations relating to events
- System Operations: operations relating to contract creations, external contract calls, and reverting/returning

[4] In actuality, compilation produces creation-time and run-time bytecode for the contract. The creation-time bytecode is used to instantiate the contract while the run-time bytecode is what is actually used for transactions. This is evident in example #4 of Example of Computation.

Gas

As long as a transaction is correctly written, it can be executed by the EVM. However, there are some problems with allowing *any* correctly written transactions to be executed:

- What if the transaction loops forever? This wastes the computational energy of the node executing this transaction. Furthermore, there's no way for a node to determine whether if a transaction will loop forever or eventually terminate (otherwise, this would solve the Halting Problem!)
- Since the only requirement for executing a transaction is its correctness, any malicious actor could just spam the network with transactions that waste the computational resources of the executing nodes (i.e. sending transactions that just loop forever).

Problems such as the ones mentioned above led to the implementation of gas. For each operation, a user must pay a certain amount of gas which is denominated in Wei. The most basic example of this is transferring ether from one account to another. As an example, assume Alice wishes to send Bob 1 gwei. In addition to forking over the 1 wei to Bob, Alice would also need to pay the 21000 gas that is the minimum for each transaction. Therefore, Alice would need to pay $1 + (21000 \cdot x)$ gwei where x is the price of gas denominated in gwei.
For some operations, gas is not constant but rather dependent on the input passed. An example of this is the `SHA3` operation, whose cost of gas is dependent on the length of the value being hashed.

Examples of Computation

In this section, we will be over a few examples of EVM executions. Each example will progressively get more complex.

Example #1: Adding

Assume we have the following hexadecimal instructions:

0x6001603c0100

Converting our hexadecimal instructions into a human-readable format, we get the following:

Instruction Address	Instruction	Additional Arguments
0x00	PUSH1	0x01
0x02	PUSH1	0x3c
0x04	ADD	N/A
0x05	STOP	N/A

We now execute our instructions sequentially: taking the instruction at 0x00, we see that we have a PUSH1 operation. This means that we will push the following byte (in this case, 0x01) from ROM onto the stack.

Index	Element
1	0x01

Stack 1: EVM stack after executing first operation

Push operations are unique in that they are the only EVM operations to have their data stored in the ROM versus the stack/memory/storage. Stack 1 represents the state of the stack after pushing 0x01 on the stack.

We now execute the second instruction; since the prior PUSH1 operation and its associated data was of size 1 byte each, our next instruction will be found at address 0x02. Again, we have another PUSH1 operation with an additional argument 0x3c. Therefore, we will push the value 0x3c onto the stack. Figure 2 represents the state of the stack after pushing 0x3c onto the stack.

Index	Element
1	0x3c
2	0x01

Stack 2: EVM stack after executing second operation

We now focus on the ADD operation. ADD differs from the previous instructions in that it pops the first 2 elements from the stack, and pushes back their sum.

Index	Element
1	0x3d

Stack 3: EVM stack after executing third operation

Finally, we reach the last instruction STOP. STOP tells the EVM to halt execution; in our context, this implies we are done with our instructions!

Example #2: Introducing Memory and Hashing

In this example, we will introduce memory and show how to store values into memory. Furthermore, we will demonstrate the use of the SHA3 function to hash values from memory. Assume we have the following instructions in human-readable format (try deriving the hexadecimal version... its a long string!):

Instruction Address	**Instruction**	**Additional Arguments**
0x00	PUSH32	$0xf^{64}$
0x21	PUSH1	0x00
0x23	MSTORE	N/A
0x24	PUSH1	0x20
0x26	PUSH1	0x00
0x28	SHA3	N/A

(Exponentiation here refers to a n-length string where each character is the element being exponentiated. So in this case, f^{64} refers to a 64 character string of the character f. We still interpret these strings as hexadecimals).

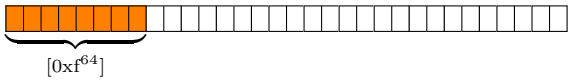
We execute the first two instructions which results in our stack looking as follows:

Index	Element
1	0x0
2	$0xf^{64}$

Stack 4: EVM stack after executing first two operations

We now come across an instruction we haven't seen before: MSTORE. MSTORE stores a 32-byte value in memory and requires two values: the offset (i.e. the address to store the 32-byte value) and

the value being stored. These two values are popped from the stack, where the first popped value is the offset and the second popped value is the value itself. Therefore, after the MSTORE operation, our stack is empty and the memory of the EVM has its addresses 0x0-0x1F filled by the string $0xf^{64}$. An visualization of how EVM memory looks after the MSTORE operation can be seen below, where non-zero values are in orange and zero values are in white:



Memory 1: EVM memory after MSTORE operation

After executing the next two PUSH1 operations, we come across an another instruction we haven't seen before: sha3. sha3 (in particular, keccak256) takes a string of arbitrary length and hashes it into a 32-byte value. sha3 pops two values from the stack: the first popped value (i) will represent the start address of the value to hash and the second popped value (l) will represent the length of the value to hash. This can be represented as the following: `value_to_hash = memory[i:i+l]`[5]. With this in mind, we get `hashed_value = sha3(memory[i:i+l])`. This hashed value gets pushed onto the stack; therefore, our stack and memory at the end of our computation is as follows:

Index	Element
1	0xa9c584056064687e149968cbab758a3376d22aedc6a55823d1b3ecbee81b8fb9

Stack 5: EVM stack after executing the SHA3 operation

Memory 2: EVM memory after SHA3 operation

Example #3: Control Flow

It would be pretty boring if our programs were only able to execute sequentially. Allowing our program counter to only increment by 1 means that things like if-else statements, for loops, and while loops wouldn't be able to be implemented! Therefore, via the `JUMP`, `JUMPI`, and `JUMPDEST` operations, the EVM allows the ROM to change the program counter. In this example, assume we're given the following operations:

[5]It is important to note that reading from memory does not erase the value stored

Instruction Address	Instruction	Additional Arguments
0x00	JUMPDEST	N/A
0x01	PUSH1	0x00
0x03	MLOAD	N/A
0x04	PUSH1	0x03
0x06	EQ	N/A
0x07	PUSH1	0x16
0x09	JUMPI	N/A
0x0a	PUSH1	0x00
0x0c	MLOAD	N/A
0x0d	PUSH1	0x01
0x0f	ADD	N/A
0x10	PUSH1	0x00
0x12	MSTORE	N/A
0x13	PUSH1	0x00
0x15	JUMP	N/A
0x16	JUMPDEST	N/A
0x17	STOP	N/A

These are a lot of instructions; however in Solidity, these machine instructions are equivalent to the following snippet of code[6]:

```
uint x = 0;
while(x < 3) {
    x += 1;
}
```

We begin with the `JUMPDEST` instruction; `JUMPDEST` does nothing[7] in practice; rather, it serves as marker for the EVM that the address `JUMPDEST` is located at is a valid destination for a `JUMP` or `JUMPI` instruction. If either of these two previously mentioned instructions changes the program counter to an address that doesn't have a `JUMPDEST` instruction associated with it, the current execution will revert. Instructions 0x01 and 0x03 loads the value stored in the first 32 bytes of memory and pushes it onto the stack. Therefore, our stack is as follows.

Index	Element
1	0x0

Stack 6: EVM stack after executing instructions 0x01, 0x03

Instructions 0x04-0x06 does the following: the value 0x03 is pushed onto the stack and afterwards, the `EQ` operation is called. `EQ` pops two values μ_1, μ_2 from the stack, and pushes back 1 if $\mu_1 == \mu_2$ or pushes back 0 otherwise. Since in this iteration we have that 0x3 $\neq$ 0x0, then the value 0x0 is pushed onto the stack.

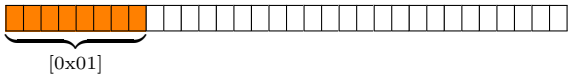
Instruction 0x07 pushes the value 0x016 onto the stack; this value may seem random, but as we'll see in a second, this is the address our program will jump to if $x \geq 3$. We now move onto instruction 0x09, the `JUMPI` operation. `JUMPI` pops off two values μ_1, μ_2 from the stack and does the following:

- If $\mu_2 \neq 0$, this implies that our jump condition holds and we jump to the address μ_1
- If $\mu_2 == 0$, this implies that our jump condition failed and we move to the next instruction (i.e. $pc = pc + 1$)

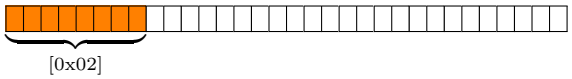
[6] I won't make any promises that your Solidity compiler will produce the same code as seen below, but as it'll become clear, the machine instructions is logically equivalent to the Solidity code

[7] JUMPDEST still consumes gas

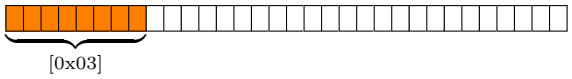
In this iteration, its clear that $\mu_2 == 0$ and so we move on to instruction 0x0a. After pushing 0x0 to the stack and executing `MSTORE`, we call `ADD` and store the sum in memory. Instructions 0x13 pushes the destination address of the `JUMP` operation. Notice that for a `JUMP` operation, we don't need to evaluate any sort of jump condition to see whether if we should jump or not. After executing instruction 0x16, we end up at instruction 0x00, and the entire process described above runs for another iteration.



Memory 3: EVM memory after the first iteration



Memory 4: EVM memory after the second iteration



Memory 5: EVM memory after the third iteration

Once $x = 3$, when we execute instruction 0x06, a value of 0x01 will be pushed onto the stack. After pushing the jump destination onto the stack, we call `JUMPI`, which will see that $\mu_2 \neq 0$, implying that the jump condition holds. In this case, we jump to instruction 0x16 which is followed by the `STOP` operation, thus terminating our while loop.

Example #4: Basic Contract Creation

Recall that when initializing a smart contract, a transaction must be sent to the `0x0` address; in this case, the transaction calldata acts as the ROM of the execution context. Assume that we have the following contract that we want to instantiate:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.17;
3
4  contract Basic {
5      uint256 age;
6
7      constructor() {
8          age = 5;
9      }
10 }
```

When compiled, the bytecode that is produced contains[8] *both* the contract creation code and the contract runtime code. Let the following table represent the contract creation code and the following bytecode represent the contract runtime code:

[8] If viewed as a string, the contract creation code comes before the runtime code

Instruction Address	Instruction	Additional Arguments
0x00	PUSH1	0x80
0x02	PUSH1	0x40
0x04	MSTORE	N/A
0x05	CALLVALUE	N/A
0x06	DUP1	N/A
0x07	ISZERO	N/A
0x08	PUSH1	0x0f
0x0a	JUMPI	N/A
0x0b	PUSH1	0x00
0x0d	DUP1	N/A
0x0e	REVERT	N/A
0x0f	JUMPDEST	N/A
0x10	POP	N/A
0x11	PUSH1	0x05
0x13	PUSH1	0x00
0x15	SSTORE	N/A
0x16	PUSH1	0x3f
0x18	DUP1	N/A
0x19	PUSH1	0x22
0x1b	PUSH1	0x00
0x1d	CODECOPY	N/A
0x1e	PUSH1	0x00
0x20	RETURN	N/A

Human Readable Form of Contract Creation Code

6080604052348015600f57600080fd5b506005600055603f8060226000396000f3

Contract Runtime Code

We now examine sections of the contract creation code to see what's going on during the initialization of the `Basic` smart contract:

- `0x00-0x04`: we are storing the free memory pointer (`0x80`) in memory address `0x40`.
- `0x05-0x0a`: we are checking if there was any wei sent with this transaction. When executing the `JUMPI` instruction here, we jump to instruction `0x0f` if no wei was sent, or continue with instruction `0x0b` otherwise.
  - `0x0b-0x0e`: wei was sent with this transaction and so we are reverting the transaction.
- `0x0f-0x15`: no wei was sent with the transaction. We are now storing the value `0x05` in storage slot `0x00` (i.e. we are storing `age = 5`).
- `0x16-0x1d`: we are grabbing bytes `0x22-0x3f` of the executing code (i.e. the transaction calldata) and storing it in in memory address `0x00` (we are storing the contract runtime code into memory).
- `0x1e-0x20`: we complete the contract initialization, returning the value `0x00-0x3f` from memory (this is our contract runtime code).

If you examine the `Basic` runtime code, you will see that because our contract does nothing in terms of behavior, any transaction call to the `Basic` contract will consist of the following:

- We store the free memory pointer (`0x80`) in memory address `0x40`.
- We revert.

Example #5: Function Selectors

In order for accounts (both EOAs and smart contracts) to interact with contracts, there must be a general framework for how both smart contracts and transaction calldata are encoded[9]; the Solidity Abstract Binary Interface (ABI) is this framework. In this example, we will focus on how transaction calldata is encoded to adhere to the ABI specification.
When interacting with a contract, there's always a function being called[10]. Therefore, the first segment of our transaction calldata will be dedicated to specifying the function that we wish to call. Assume that we want to call the following function:

```
transfer(address recipient, uint256 amount)
```

We first write `transfer` to its canonical form (i.e. removing unncessary spaces, labels such as `calldata`, `memory`, etc.):

```
transfer(address,uint256)
```

To some, it might seem as if we've hit a roadblock, considering that the canonical form of `transfer` needs to be converted into a hexadecimal value. However, we have a solution to this: we hash our canonical string (using keccak256) into a 32-byte value and extract the first 4 bytes. Therefore, `transfer` becomes the following value: `0xa9059cbb`. This 4-byte value is known as the *function selector*; it tells the smart contract that we are calling the function with ID `0xa9059cbb` (in this case, `transfer`).
We now want to add the arguments to our calldata; in contrast to computing the function selector, this is straightforward[11]. For each argument, we append it to our calldata string (padding it to a 32-byte value whenever necessary).

References

The following sources were (very) useful in the creation of this primer:

- https://ethervm.io/
- https://www.evm.codes/playground
- https://docs.soliditylang.org/en/v0.8.19/index.html
- https://www.4byte.directory/
- https://emn178.github.io/online-tools/keccak_256.html
- https://ethereum.github.io/yellowpaper/paper.pdf
- https://ethereum.org/en/developers/docs/evm/

[9]While this is the recommended way of encoding your smart contracts/transaction calldata, you are not required to adhere to it. The benefits of not adhering to the ABI format is that you gain, in a sense, privacy and gas optimizations. The downside is that no one will know how to interact with your contract unless you tell them otherwise.
[10]Except in the case of sending ether directly to a contract via its fallback/receive functions
[11]For static types, this is straightforward. For dynamic types such as string and uint256[], its not as straightforward.