#3 EVM Execution Model

Lecture Notes for CS190N: Blockchain Technologies and Security October 6, 2025

This lecture builds a concrete and practical model of the Ethereum Virtual Machine. We start by grounding execution at the instruction level so that inputs, stack operations, memory, and returns are obvious from a trace. We then connect that execution model to persistent state by reading and writing storage, explaining why only storage changes survive across calls and how values move between calldata, stack, memory, and slots. Finally, we introduce gas metering as the mechanism that both enforces termination and prices scarce resources, and we learn to produce quick cost estimates that match real measurements. By the end, you will be able to read opcode traces, predict which steps are expensive, and explain how a short sequence of instructions turns inputs into durable state updates.

1 INTRODUCTION

Ethereum is a replicated state machine. A transaction is an input that advances the global state, and each full node computes the same transition. The engine that performs this computation is the Ethereum Virtual Machine, defined as a deterministic and sandboxed stack machine with a precise instruction set and state transition function [1]. These notes proceed in three parts: how instructions execute and move data (*Opcodes*), how persistent state is read and written (*Storage*), and how work is metered and priced (*Gas*). Examples are explained step by step so the notes stand alone.

2 OPCODES

At execution time the EVM maintains a program counter, a stack of up to 1024 items, a byte addressed memory region that starts empty for each call, and read only calldata provided by the caller. Words on the stack and in storage are 256 bits. Memory reads and writes operate on 32 byte words at byte offsets. None of these locations are persistent; they vanish when the call ends.

Before we touch examples, fix this picture in your head. Inputs arrive as a byte array named calldata, not as variables. Computation happens on the stack using fixed width words. Any output must be staged in memory. Returning data requires naming a memory offset and a length. This flow is the backbone of almost every trace you will read, and it matches the formal state machine in the Yellow Paper [1].

Stack warm-up. Stack moves introduce, duplicate, and reorder data without using memory or returns. The program below leaves `0xBB` on the stack using only `PUSH`/`DUP`/`SWAP`/`POP`.

```
# Leave 0xBB on the stack using only stack moves
PUSH1 0xAA        # [0xAA]
PUSH1 0xBB        # [0xBB, 0xAA]
SWAP1             # [0xAA, 0xBB]
DUP1              # [0xAA, 0xAA, 0xBB]
POP               # [0xAA, 0xBB]
POP               # [0xBB]
STOP
```

`PUSHx` places an immediate; `DUPx` duplicates the xth item (here `DUP1`); `SWAPx` exchanges positions (here `SWAP1`); `POP` discards the top.

Computation on the stack. The fastest work stays entirely on the stack. Arithmetic is modulo 2^{256}, so results wrap on overflow.

```
# Compute 2 + 3 and leave 5 on the stack
PUSH1 0x02         # [0x02]
PUSH1 0x03         # [0x03, 0x02]
ADD                # [0x05]  pops 0x03, 0x02; pushes 0x05
STOP
```

Do not miss the data flow discipline. `ADD` consumes the top two items and produces one result. If you needed to reuse either input, insert a `DUP1` before `ADD`. If the order were wrong, a `SWAP1` would repair it.

Control flow. Branches are explicit and their targets are constrained. The program below returns 1 if the first argument is nonzero, else returns 0. Only positions marked with `JUMPDEST` are legal jump targets; this rule keeps control flow unambiguous. As before, returning bytes goes through memory.

```
# If x != 0 then return 1 else return 0
PUSH1 0x04
CALLDATALOAD       # [x]
PUSH1 label_true   # [addrT, x]
JUMPI              # PC := addrT if x != 0

# false branch:
PUSH1 0x00
PUSH1 0x00
MSTORE
PUSH1 0x20
PUSH1 0x00
RETURN

# true branch:
JUMPDEST label_true
PUSH1 0x01
PUSH1 0x00
MSTORE
PUSH1 0x20
PUSH1 0x00
RETURN
```

To consolidate, Table 1 collects the instruction families you will meet in traces. Keep it open while reading examples.

Reading input and returning output. With stack rules, arithmetic, and control flow in place, we now add the full I/O path. The program reads a 32 byte argument from calldata and echoes it back. It demonstrates the path from calldata to stack to memory to return. Comments show the top of the stack on the left, so you can track values as they move.

Table 1. Frequently used opcodes

Category	**Instructions and role**
Stack	PUSHx place immediate, POP discard, DUPx duplicate the xth item, SWAPx swap top with xth.
Arithmetic and logic	ADD, SUB, MUL, DIV; bitwise AND, OR, XOR. Operate on 256 bit words.
Memory	MLOAD, MSTORE read or write 32 byte words at byte offsets. High addresses trigger expansion cost.
Calldata and return	CALLDATALOAD reads 32 bytes from calldata. RETURN sends bytes from memory by offset and length.
Control flow	JUMP, JUMPI to JUMPDEST; STOP, REVERT.

```
# Echo the first 32-byte argument from calldata
# Calldata layout: [4-byte selector][arg0][arg1]...
PUSH1 0x04          # stack: [0x04]   offset of arg0 after the selector
CALLDATALOAD        # stack: [arg0]   copy 32 bytes from calldata
PUSH1 0x00          # stack: [0x00, arg0]
MSTORE              # mem[0..31] = arg0, stack: []
PUSH1 0x20          # stack: [0x20]   length = 32
PUSH1 0x00          # stack: [0x00, 0x20]  offset = 0
RETURN              # return mem[0..31] to the caller
```

This trace establishes three habits. Inputs live in calldata and must be explicitly loaded; nothing implicitly appears on the stack. You cannot return directly from the stack; you must first place bytes in memory with MSTORE. RETURN always takes two numbers, an offset and a length, so the caller receives exactly those bytes.

Where this leaves us. You now have the vocabulary to read a trace: begin with stack moves, add arithmetic on the stack, layer explicit control flow, and finally tie in the calldata → memory → return path. Nothing here changes persistent state; that belongs to storage.

3 STORAGE

Storage is the EVM's persistent key value map from 256 bit keys to 256 bit values. Only storage changes survive after the call finishes. Reading a slot uses SLOAD. Writing uses SSTORE. Because storage mutations alter the global state that every full node must retain and serve to others, these operations are intentionally expensive.

Although this section focuses on storage, keep the transient locations in view. Table 2 summarizes lifetime, scope, and cost across the three places data can live. Refer to it when deciding where to perform work.

Writing and reading slot 0. The first listing persists a literal in slot 0. The second reads slot 0 and returns its 32 bytes. Note the ordering discipline before SSTORE: the value must be on top of the stack and the key under it, because SSTORE pops the value first and the key second.

```
# Persist the literal 0x2a in slot 0
PUSH1 0x2a
```

Table 2. Data locations in one view

Feature	**Stack**	**Memory**	**Storage**
Lifetime	Call only	Call only	Across transactions
Scope	Current call	Current call	Contract global state
Size model	1024 words of 32 bytes	Grows as needed	2^{256} slots
Relative cost	Very low	Low plus expansion	Very high
Typical use	Opcode inputs and outputs	Build return buffers	Persistent variables

```
PUSH1 0x00
SWAP1
SSTORE
STOP
```

```
# Read slot 0 and return it
PUSH1 0x00
SLOAD
PUSH1 0x00
MSTORE
PUSH1 0x20
PUSH1 0x00
RETURN
```

End to end: set slot 0 to the caller's 32 byte argument. This combines input handling and persistence. The stack snapshots in Table 3 show how values move.

```
# Calldata: [4-byte selector][x]
PUSH1 0x04
CALLDATALOAD
PUSH1 0x00
SWAP1
SSTORE
STOP
```

Design patterns that minimize storage cost. Stage intermediate results on the stack, use memory only when you must build or return bytes, and commit to storage once. For example, summing n inputs and writing one total performs 1 write instead of n. The next section quantifies why this works as well as it does.

4 GAS

Gas is the metering and pricing layer for EVM execution. It enforces termination and prices scarce resources. Stack moves and small arithmetic are very cheap. Memory costs include a small base and an expansion term that grows with the highest memory address touched in the call. Storage reads and writes are far more expensive because they change persistent state. Within a single transaction,

Table 3. Stack evolution for “store argument into slot 0”

Step	**Before**	**After**
`PUSH1 0x04`	[]	[0x04]
`CALLDATALOAD`	[0x04]	[x]
`PUSH1 0x00`	[x]	[0x00, x]
`SWAP1`	[0x00, x]	[x, 0x00]
`SSTORE`	[x, 0x00]	[]
`STOP`	[]	[]

the first access to a given address or storage slot is cold; later accesses to the same location are warm and cheaper. The semantics and schedule follow the protocol specifications and updates that refine the Yellow Paper over time [1].

Before estimating anything, keep the scale in Table 4 in mind. It shows why your first pass at optimization should be to remove unnecessary storage writes rather than micro tune stack juggling.

Table 4. Illustrative opcode costs, simplified

Opcode	Meaning	Approximate gas
`PUSH1`, `DUPx`, `SWAPx`	stack moves	about 3
`ADD`, `SUB`, `MUL`	arithmetic	about 3
`MLOAD`, `MSTORE`	memory read or write	base 3 plus expansion
`SLOAD`	read storage slot	about 100 warm or 2100 cold
`SSTORE`	write storage slot	about 2900 warm or 22100 cold new

Minimal estimate: compute 2 + 3 and store it in slot 0. We count only opcode costs to highlight the scale. The program computes a small sum on the stack, then persists it with one write.

```
PUSH1 0x02
PUSH1 0x03
ADD
PUSH1 0x00
SWAP1
SSTORE
STOP
```

Using Table 4, a cold write totals roughly

$$3 + 3 + 3 + 3 + 3 + 22100 \approx \mathbf{22115} \text{ gas.}$$

If slot 0 was already touched earlier in the same transaction, the write is warm and the estimate becomes

$$3 + 3 + 3 + 3 + 3 + 2900 \approx \mathbf{2915} \text{ gas.}$$

This gap is the practical reason to batch reads and commit a single write when possible.

Read path estimate. Reading and returning slot 0 costs a cold `SLOAD` of about 2100 or a warm one of about 100, plus a few units for stack moves, one `MSTORE` base, and the return copy. The headline is unchanged: the persistent operation dominates.

Memory expansion in practice. If a program writes to memory at address 0 then returns 32 bytes, the expansion term is tiny. If it writes first at a high address such as 0x4000, the cost increases because expansion depends on the highest address touched. This is why programs stage outputs near the start of memory unless there is a reason to reserve space.

Putting it together. Gas rewards designs that keep most work on the stack and in memory, and that minimize persistent writes. When in doubt, scan a trace for `SSTORE` and cold `SLOAD` first. They explain the majority of the cost.

5 SUMMARY

Execution is a small vocabulary used precisely. Opcodes move values from calldata to the stack, through arithmetic and control flow, into memory for return, and into storage when persistence is required. Storage is the only persistent location and is priced accordingly. Gas ties these choices to dollars and ensures termination. When reading a trace, find the storage operations first, then follow how the stack and memory carry inputs to the point where state changes. Keep Table 1, Table 2, and Table 4 open as quick references.

REFERENCES

[1] Gavin Wood. 2014. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* Technical Report. Ethereum Foundation. https://ethereum.github.io/yellowpaper/paper.pdf Yellow Paper, living document with periodic revisions.

A USEFUL RESOURCES

A.0.1 EVM Opcodes Interactive Reference. An interactive reference site containing all EVM opcodes with their gas costs, stack behavior, and descriptions. https://www.evm.codes/

A.0.2 EVM Playground. A powerful online tool for decompiling bytecode into opcodes and stepping through execution to visualize stack, memory, and storage state changes. https://www.evm.codes/playground