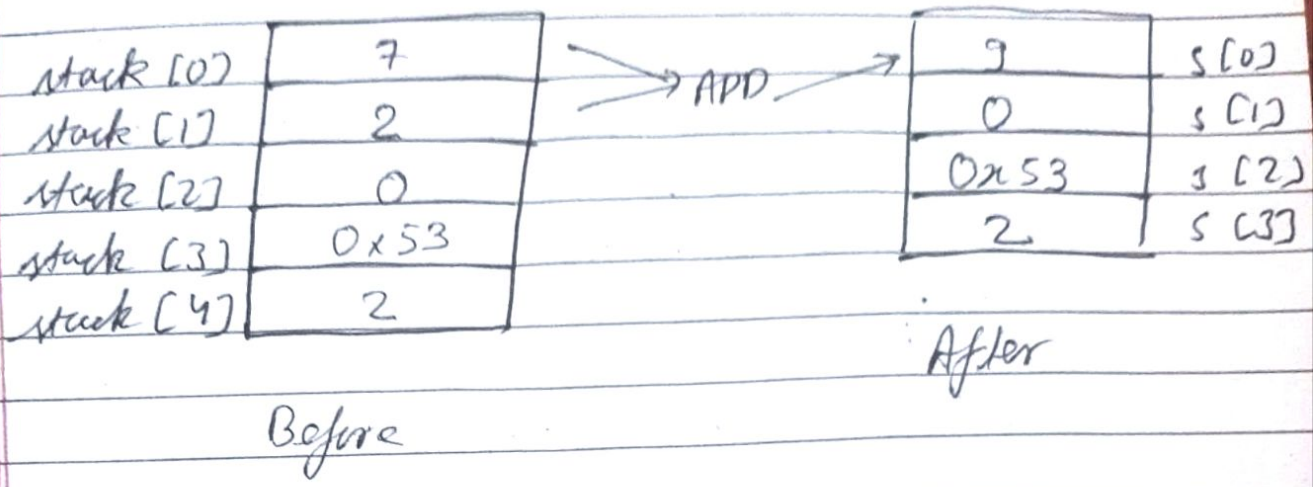
DATE: / /

Stack, Memory, Storage, Code, Calldata & Logs:

① The EVM is a stack based processor
② ∴ EVM opcodes pop information & push information onto the stack.

Example:



① Stack: EVM opcodes pop information from & push data onto the stack.
② Calldata: the data field of a txn.
3) Memory: Into store available for the duration of a txn.
4) Storage: Persistent data store.
5) Code: Existing code & static data storage.
6) Logs: Write-only logger event output.

Opcodes in the Yellow paper (Eth)

STOP	Halts execution
ADD	$\mu[0] = \mu[0] + \mu[1]$
MUL	$\mu[0] = \mu[0] \times \mu[1]$
SUB	$\mu[0] = \mu[0] - \mu[1]$
DIV	$\mu[0] = \begin{cases} 0 & \text{if } \mu[1] = 0 \\ \mu[0] \div \mu[1] & \text{otherwise} \end{cases}$
SDIV	Signed integer division op (truncated)

We will discuss the lyfecycle (inside of -bin file) and how it interacts with EVM.

Transaction Fields:

Nonce: the txn no for this account, starts with 0

Gas Price

Gas Limit

To

Value

Data

v, r, s : components of the txn sig.

Deployments:

For deployment, the "to" field has to be empty.

&

The Data field contains the init code to set-up the contract state & deploy the contract.

Function Call:

Data: Data that the code in the EVM processes & does stuff with

&

for Solidity contracts & other contracts that conform to the ABI

↳ Data: function to call + parameters.

→ Calldata [which is the data field during a txn]

Contracts are deployed using transactions where "To" address is not specified
"Data" is the init code fragment. This includes the contract binary. This is the compiler output in the *.bin file.

→ The init function is not stored on the blockchain.
→ The data of the transaction is treated as code for contract deployment.

Because it's one constructor, and would only ever need to be run once to set initial state, therefore it makes sense to not store it in the blockchain.

Sample .bin file

6080 60405234801561

PUSH 0x80 Since 60 is the opcode for push
PC: 0x0, opcode: PUSH1 0x80
PC: 0x2, " PUSH1 0x40
PC: 0x4, " MSTORE
PC: 0x5, " CALLVALUE
PC: 0x6, "
PC: 0x7, "
PC: 0x8, "
PC: 0xb, "
PC: 0xc, "
PC: 0xe, "
PC: 0xf, "
PC: 0x10, "

↳ The stack has a maximum depth of 1024 words.
↳ Stack items are 1 word wide ie 32 bytes.

PAGE NO.:
DATE:

Byte Code:

PUSH1: push the byte following this opcode onto the stack.
The top 31 bytes of the word are zero-filled.

MSTORE: Store a word in memory.
↳ Top two values of the stack
stack[0] is the location to write to
stack[1] is the value to write

Now the stack is empty.

∴ Now, stack[0] is the location to write to
&
stack[1] is the location to write.

This sets up the free memory pointer
Memory[0x40] = 0x80
⇓
Memory location 0x40 to value 0x80.
Means that the Solidity code can safely allocate memory starting at 0x80.

// If code allocates memory, then pointer should be updated.

Interesting pt/Note: Here in Memory[0x40] = 0x80
This thing is in bytes (1 byte) & this thing is in words (32 bytes)

JUMI [Jump if]

to
Jump to ~~&~~ the program counter stack[0] if stack[1] is not zero. Pop two values off the stack.

stack[0]: value sent with the tan

PUSH 1 0x00
DUP 1
stack[0]: 0
stack[01]: 0
stack[2] = value sent with the tan

REVERT
Halt execution and indicate a REVERT has occured. Use stack[0] as a memory location & stack[1] as a length of Revert reason.

⇓

~~JUMPDEST~~ So what essentially happened in the code from [CALLVALUE ... to REVERT] is that:
code snippet

REVERT if the constructor which is NOT payable has Wei in the contract deploy transaction.

JUMPDEST
If the Jump to program counter 0x10 was taken it would arrive here
Valid jump dest' are indicated by the JUMPDEST opcode.
stack[0] = value sent with tan.

{ since every Jump instruction has got to have a valid Jump destination }

[link it to msg.value ?]

CALLVALUE
Push onto the stack how much Wei was sent with the txn
This is the value txn field

stack[0]: value sent with txn

DUP1
Duplicates the top of the stack.
Now, stack[0]: value sent with txn
stack[1]: value sent with txn

DUP2 → pushes a copy of stack[1] onto the stack
DUP3 → pushes a copy of stack[2] onto the stack
⋮
all the way upto DUP31

ISZERO
Pop a word off the stack.
If the word is zero, push 1 onto the stack,
otherwise push 0.

stack[0]: 1 if value == 0 & 0 if value != 0
stack[1]: value sent with txn

PUSH2 0x0010
Push two bytes following the opcode onto the stack.
stack[0] = 0x10
stack[1] = 1 if value == 0 & 0 if value != 0
stack[2] = value sent with txn

POP
↳ Pop top value of the stack
↳ Stack is now empty.

since we were storing 3 in val 2 variable
PUSH 1 0x03
PUSH 1 0x01
SSTORE → Stores a value in storage

↳ Pops two values of the stack
↳ stack[0] is the location to write to
& stack[1] is the value to write
@ Storage[1] = 3

stack is now empty

PUSH 1	0xC1	Set-up the stack to prepare for the CODECOPY opcode & RETURN opcode
DUP 1		
PUSH 2	0x0024	
PUSH 1	0x00	

stack[0] = 0
stack[1] = 0x24
stack[2] = 0xC1
stack[3] = 0xC1

CODECOPY → Copy from code that is executing to memory.
s[0] → memory offset to write to
s[1] → code offset to read from
s[2] → length in bytes to copy

The stack now look like
S[0] - 0x01

So, basically, the entire operation from
PUSH 0x01 — — CODECOPY

Logic the contracts code that will be on the blockchain to memory

PUSH1 0x00 } RETURN
RETURN } End execution, return a result & indicate successful execution.

stack[0] is the starting offset (in memory) of the result.
stack[1] is the ending offset (in memory) of the result.

S[0] = 0
S[1] = 0x01

↳ For contract deployment,
store the return result into deployed contract's address.

In this example, the 193 bytes (0x01) of code at program counter offset 0x24 was copied to ~~0x00~~ memory [0x00 — 0x01] & it the no code that you'll find at the contract's address.

INVALID → Invalid operation marks the end of the init code

Function Calls

↳ public storage variables automatically have "getter" view calls created.

---

then pop stack[0] & stack[1]
&
push 1 onto the stack
else
pop S[0] & S[1]
&
push 0 onto the stack.

PUSH1 0x3c
JUMPI → Revert if the transaction data field is less than 4 bytes long
PUSH1 0x00

↓
Reason: function selector

Jump Table for functions

CALLDATALOAD → Push onto the stack the 32 bytes (of CALLDATA) starting at offset S[0]

Therefore
PUSH1 0x00
CALLDATALOAD } Push onto the stack the 32 bytes starting at offset 0x00

```
pragma solidity >= 0.4.23;

contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

```
PUSH1 0x80
PUSH1 0x40
MSTORE
CALLVALUE
DUP1
ISZERO
PUSH1 0x0f
JUMPI
PUSH1 0x00
DUP1
REVERT
JUMPDEST
POP
PUSH1 0x01
CALLDATASIZE
LT
```

Set up the free memory pointer

Revert if the tsn value isn't zero

Remove the value from the stack

Push the size of the tsn data field on the stack

Less than
if s[0] < s[1]

Note on function selector:

The first four bytes of the call data for a function call specifies the function to be called.

It is the first four bytes of Keccak-256 hash of the signature of the function.

The signature is defined as the canonical expression of the basic prototype without data location specifier i.e, the function name with the parenthesised list of param types.

Parameter types are split by a single comma - no spaces are used.

× ——— × ——— ×

If the function selector from the tan data field doesn't equal any of the functions then REVERT

```
JUMPDEST
PUSH1 0x00
DUP1
REVERT
```

Note:

LT followed by ISZERO is essentially, Greater than or equal to

STOP → End execution, indicate execution success, & don't return any data

SWAP2 → Swap S[0] & S[2]

PUSH1 0xc0 → shift the top 32 bits to the bottom 32 bits of the word.
SHR — Shift s[1] to the right s[0] times & pop s[0] off the stack.

&

PUSH1 0x00 ---- SHR

⇓

Load the function selector from the transaction data field.

DUP1 — Make a copy of stack[0], the function selector from the txn data field.

PUSH4 0x60fe47b1
EQ
PUSH1 0x41
JUMPI
— Jump to 0x41 if the fun selector from txn data field == 0x60fe47b1

DUP1
PUSH4 0x95cacbe0
EQ
PUSH1 0x5d
JUMPI
— Jump to 0x5d if the fun sel from the txn data field == 0x95cacbe0

DUP1
PUSH4 0xc82fdf36
EQ
PUSH1 0x75
JUMPI
— "