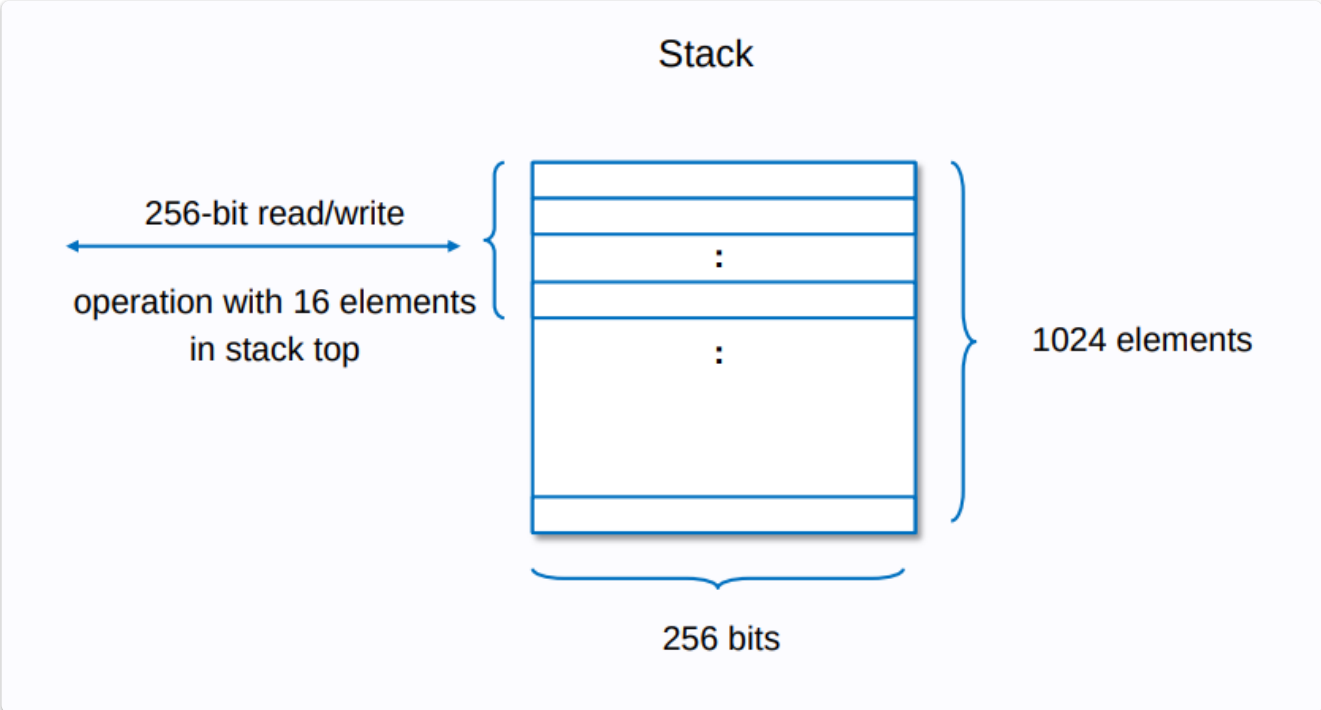
Lesson 5

Topics this week

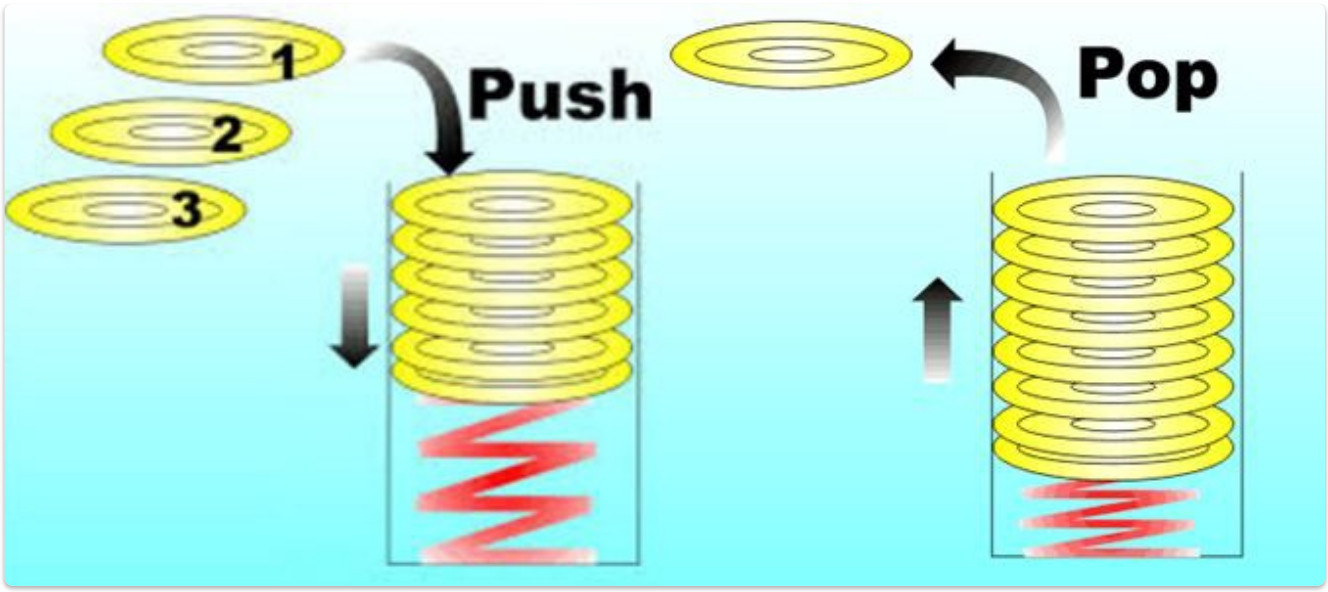
Monday : Assembly 1
Tuesday : Assembly 2
Wednesday : Layer2 and rollups
Thursday : Gas Optimisation

Quick review of lesson 3

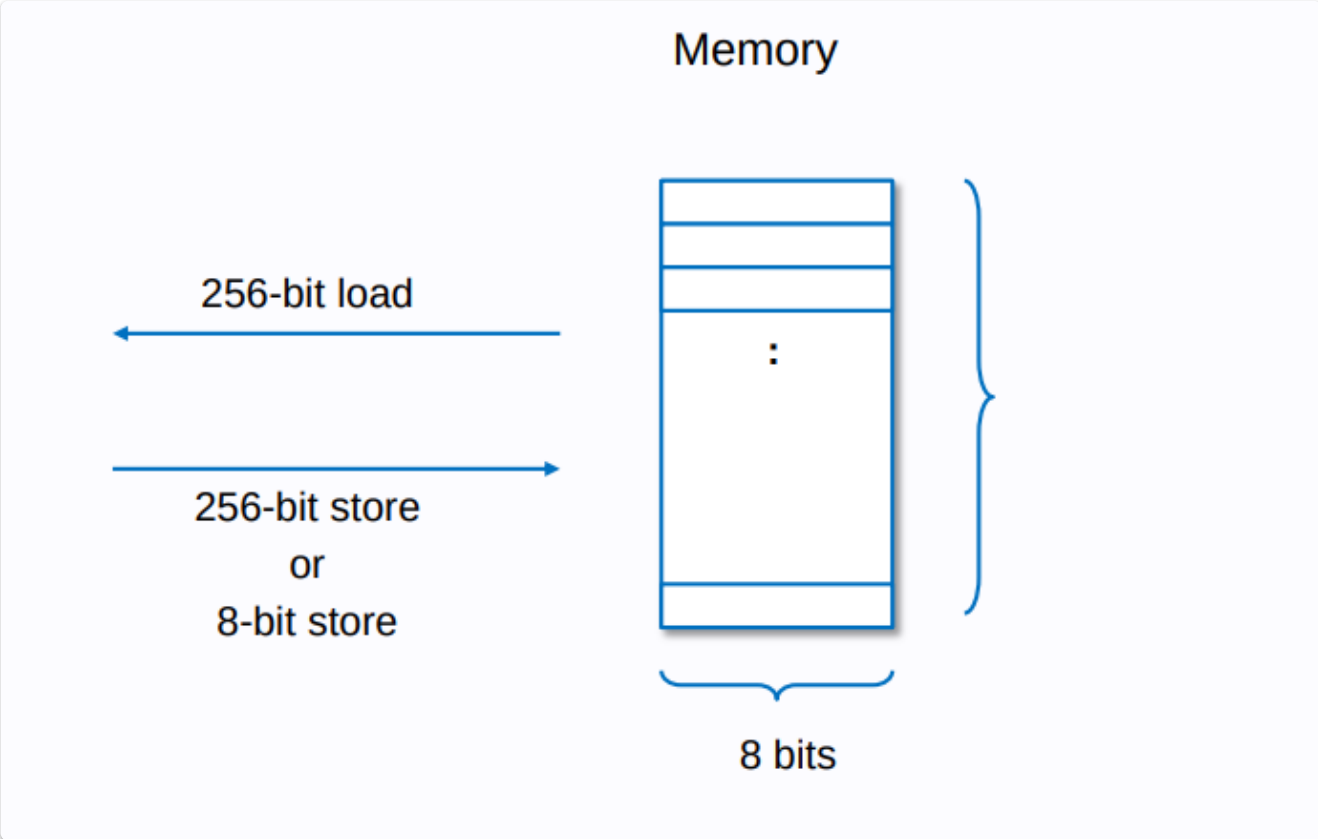
The Stack



The top 16 items can be manipulated or accessed at once (or stack too deep error)
The stack is where the operations take place

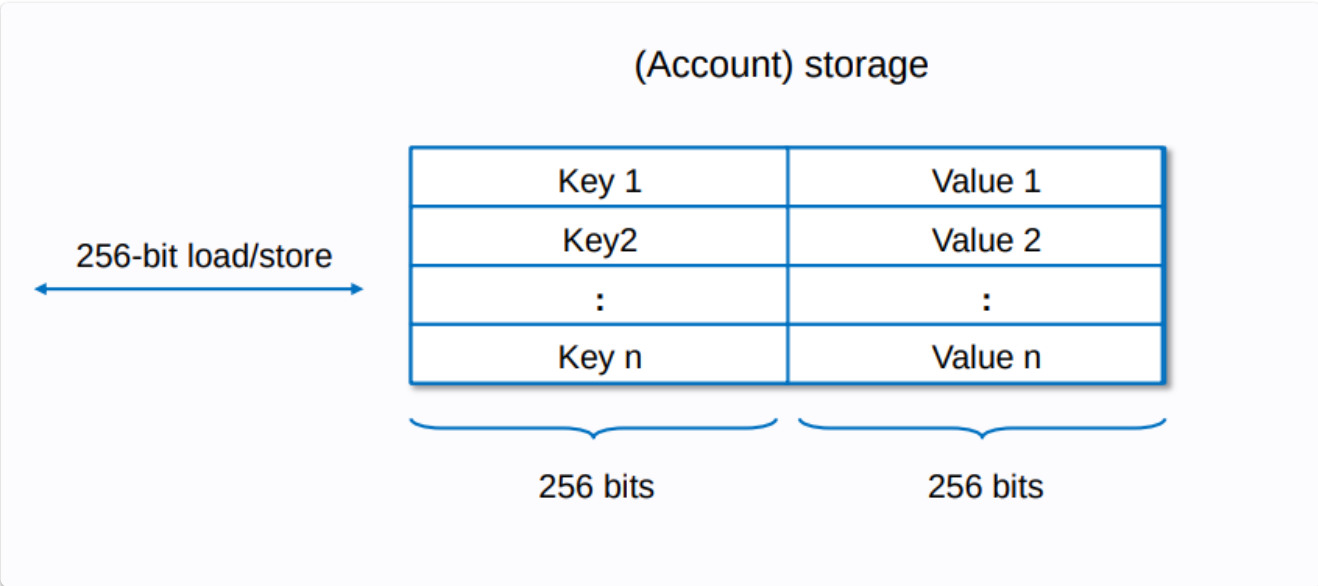

1
2
3
Push
Pop

Memory



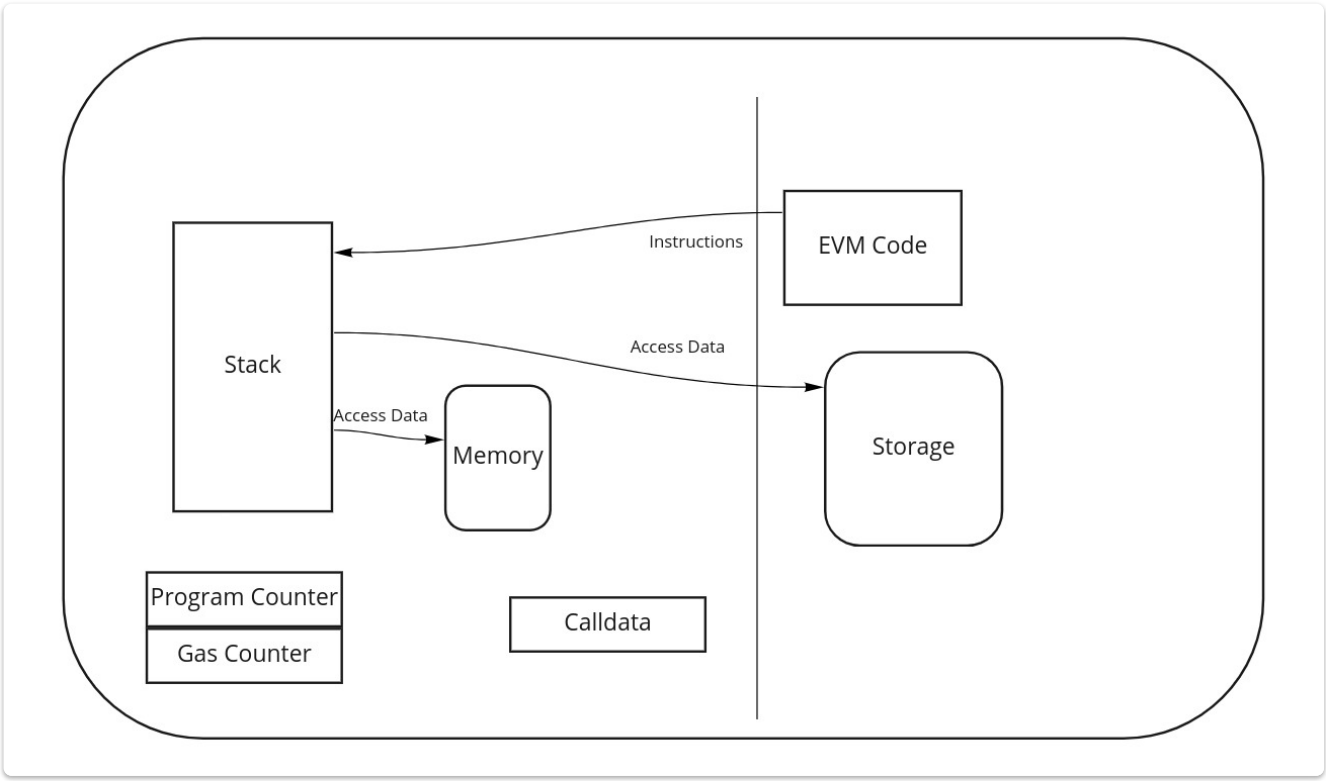
Memory is a byte-array. Memory starts off zero-size, but can be expanded in 32-byte chunks by simply accessing or storing memory at indices greater than its current size.

Storage



See Documentation

Code Execution



OpCodes

- Stack-manipulating opcodes (POP, PUSH, DUP, SWAP)
- Arithmetic/comparison/bitwise opcodes (ADD, SUB, GT, LT, AND, OR)
- Environmental opcodes (CALLER, CALLVALUE, NUMBER)
- Memory-manipulating opcodes (MLOAD, MSTORE, MSTORE8, MSIZE)
- Storage-manipulating opcodes (SLOAD, SSTORE)
- Program counter related opcodes (JUMP, JUMPI, PC, JUMPDEST)
- Halting opcodes (STOP, RETURN, REVERT, INVALID, SELFDESTRUCT)

List of EVM Opcodes

Assembly

A good reference is Jean's article

```
a + b      // Standard Notation (Infix)
a b add    // Reverse Polish Notation

mstore(0x80, add(mload(0x80), 2))      // how we write the code
2 0x80 mload add 0x80 mstore           // How it is represented in bytecode
```

Reverse Polish Notation

Examples

```
mstore(0x40,0x80) = Store 0x80 at memory location 0x40
results in


00:  6080 PUSH1 0x80
02:  6040 PUSH1 0x40
04:  52   MSTORE
```

This is in fact setting up the free memory pointer, it means that memory after address 0x80 is free

---

THE STACK ADDING 2 AND 4

For example

```
PUSH 2
PUSH 4
ADD
```

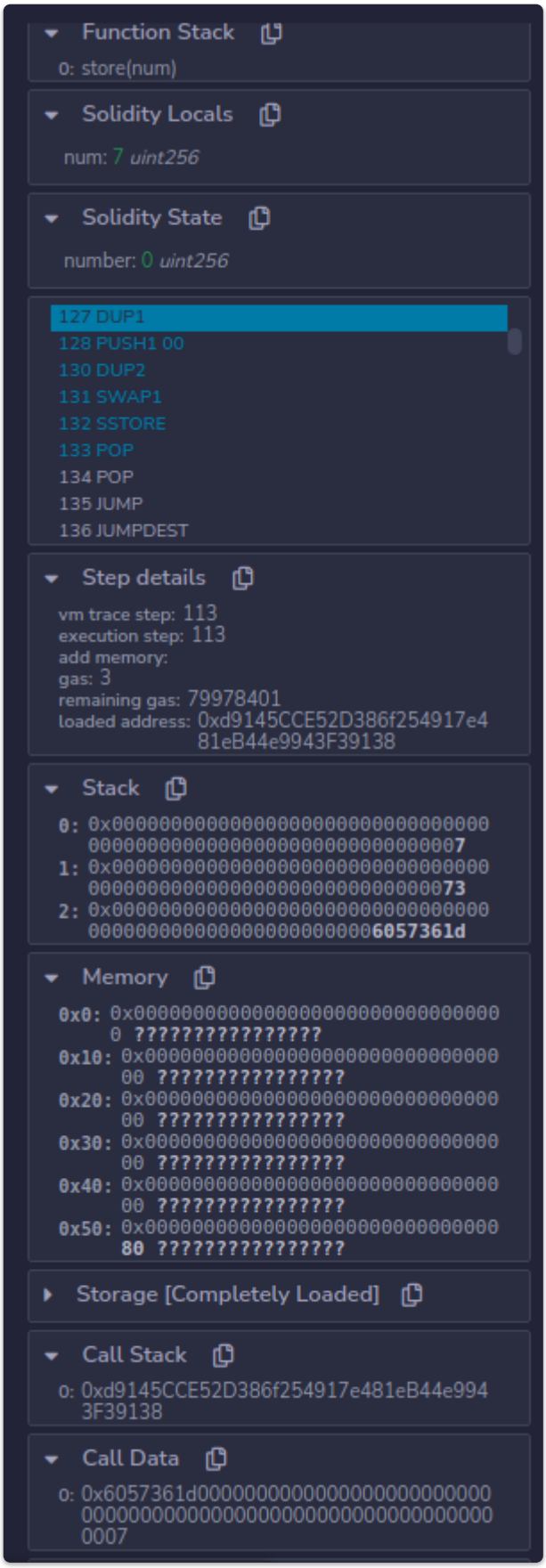
```
PUSH 2
|__2__|

PUSH 4
|__4__|
|__2__|

ADD
|__6__|
```

A MORE COMPLEX EXAMPLE

```
mstore(0x80, add(mload(0x80), 2))
2 0x80 mload add 0x80 mstore

PUSH 2
|__2__|

PUSH 0x80
|_0x80|
|__2_|

MLOAD
|__5__|
|__2__|

ADD
|__7__|

PUSH 0x80
|_0x80|
|__7__|

MSTORE
|_____|
```

Looking at a function in the remix debugger

```
    function store(uint256 num) public {
        number = num;
    }
```


Function Stack
0: store(num)
Solidity Locals
num: 7 uint256
Solidity State
number: 0 uint256
127 DUP1
128 PUSH1 00
130 DUP2
131 SWAP1
132 SSTORE
133 POP
134 POP
135 JUMP
136 JUMPDEST
Step details
vm trace step: 113
execution step: 113
add memory:
gas: 3
remaining gas: 79978401
loaded address: 0xd9145CCE52D386f254917e4
81eB44e9943F39138
Stack
0: 0x0000000000000000000000000000000000000000000000000000000000000007
1: 0x0000000000000000000000000000000000000000000000000000000000000073
2: 0x000000000000000000000000000000000000000000000000000000006057361d
Memory
0x0: 0x0000000000000000000000000000000000000000 ??????????????????
0x10: 0x0000000000000000000000000000000000000000 ??????????????????
0x20: 0x0000000000000000000000000000000000000000 ??????????????????
0x30: 0x0000000000000000000000000000000000000000 ??????????????????
0x40: 0x0000000000000000000000000000000000000000 ??????????????????
0x50: 0x0000000000000000000000000000000000000000 ??????????????????
Storage [Completely Loaded]
Call Stack
0: 0xd9145CCE52D386f254917e481eB44e9943F39138
Call Data
0: 0x6057361d0000000000000000000000000000000000000000000000000000000000000007

Bytecode and Deployment

This contract (See gist)

```
contract Deploy1{
        uint256 value1;
        constructor(){
                value1 = 17;
        }
        function read() view public returns (uint256 result){
        return value1;
        }
}
```

Has this bytecode

```
"object":
"6080604052348015610010576000080fd5b506011600081905555060b6806100276000396000f3f
e6080604052348015600f57600080fd5b506004361060285760003560e01c806357de26a414602
d575b600080fd5b60336047565b604051603e9190605d565b60405180910390f35b60008054905
090565b6057816076565b82525050565b60006020820190506070600083018460505565b9291505
0565b60008190591905056fea2646970667358221220872b5d4b9f200afddd5ed3c424f6b3b99
5bf467e212ec4c313f65365aeadf8e964736
f6c63430008060033",

"opcodes": "PUSH1 0x80 PUSH1 0x40 MSTORE
CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST
POP PUSH1 0x11 PUSH1 0x0 DUP2 SWAP1 SSTORE POP PUSH1 0xB6 DUP1 PUSH2
0x27 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN INVALID PUSH1 0x80 PUSH1 0x40
MSTORE CALLVALUE DUP1 ISZERO PUSH1 0xF JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH1 0x28 JUMPI PUSH1 0x0
CALLDATALOAD PUSH1 0xE0 SHR DUP1 PUSH4 0x57DE26A4 EQ PUSH1 0x2D JUMPI
JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0x33 PUSH1 0x47 JUMP
JUMPDEST PUSH1 0x40 MLOAD PUSH1 0x3E SWAP2 SWAP1 PUSH1 0x5D JUMP
JUMPDEST PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH1
0x0 DUP1 SLOAD SWAP1 POP SWAP1 JUMP JUMPDEST PUSH1 0x57 DUP2 PUSH1
0x76 JUMP JUMPDEST DUP3 MSTORE POP POP JUMP JUMPDEST PUSH1 0x0 PUSH1
0x20 DUP3 ADD SWAP1 POP PUSH1 0x70 PUSH1 0x0 DUP4 ADD DUP5 PUSH1 0x50
JUMP JUMPDEST SWAP3 SWAP2 POP POP JUMP JUMPDEST PUSH1 0x0 DUP2 SWAP1
POP SWAP2 SWAP1 POP JUMP INVALID LOG2 PUSH5 0x6970667358 0x22 SLT
KECCAK256 DUP8 0x2B 0x5D 0x4B SWAP16 KECCAK256 EXP REVERT 0xDD 0x5E
0xD3 0xC4 0x24 0xF6 0xB3 0xB9 SWAP6 0xBF CHAINID PUSH31
0x212EC4C313F65365AEADF8E964736F6C6343000806003300000000000000000 "
```

Init Code

This is the initialisation (init) code

```
6080604052348015610010576000 80fd5b506011600081905555060b6806100276000396000f3fe
```

This code runs the constructor and then loads the rest of the byte code into memory and returns it.

The first part sets up the free memory pointer

```
PUSH1 0x80
PUSH1 0x40
MSTORE
```

The next part checks for wei being sent with the transaction and would revert since the constructor is not payable.

```
CALLVALUE
DUP1
ISZERO
PUSH2 0x10
JUMPI
PUSH1 0x0
DUP1
REVERT
```

```
JUMPDEST  = Valid jump destination

POP - stack is now empty

Now store the value 0x11 in storage slot 0
PUSH1 0x11
PUSH1 0x0
DUP2
SWAP1
SSTORE

Now we prepare for the codecopy, we need a location and a length
Stack 0 is the memory offset to write to
Stack 1 is the code offset to read from
Stack 2 is the length of the copy
```

```
POP
PUSH1 0xB6
DUP1
PUSH2 0x27
PUSH1 0x0
```

After that the stack looks like this

```
[ "0x0000000000000000000000000000000000000000000000000000000000000000",
"0x0000000000000000000000000000000000000000000000000000000000000027",
"0x00000000000000000000000000000000000000000000000000000000000000b6",
"0x00000000000000000000000000000000000000000000000000000000000000b6" ]
```

Now we copy the rest of the bytecode (the non init bytecode)

```
CODECOPY
PUSH1 0x0
RETURN
```

When we return from this type of transaction,
Stack 0 is the starting offset of the result (in memory)
Stack 1 is the ending offset of the result (in memory)

When we return, the stack looks like this

```
[ "0x0000000000000000000000000000000000000000000000000000000000000000",
"0x00000000000000000000000000000000000000000000000000000000000000b6" ]
```

```
INVALID - this marks the end of the init code
```

See this video for a similar walkthrough.

Question

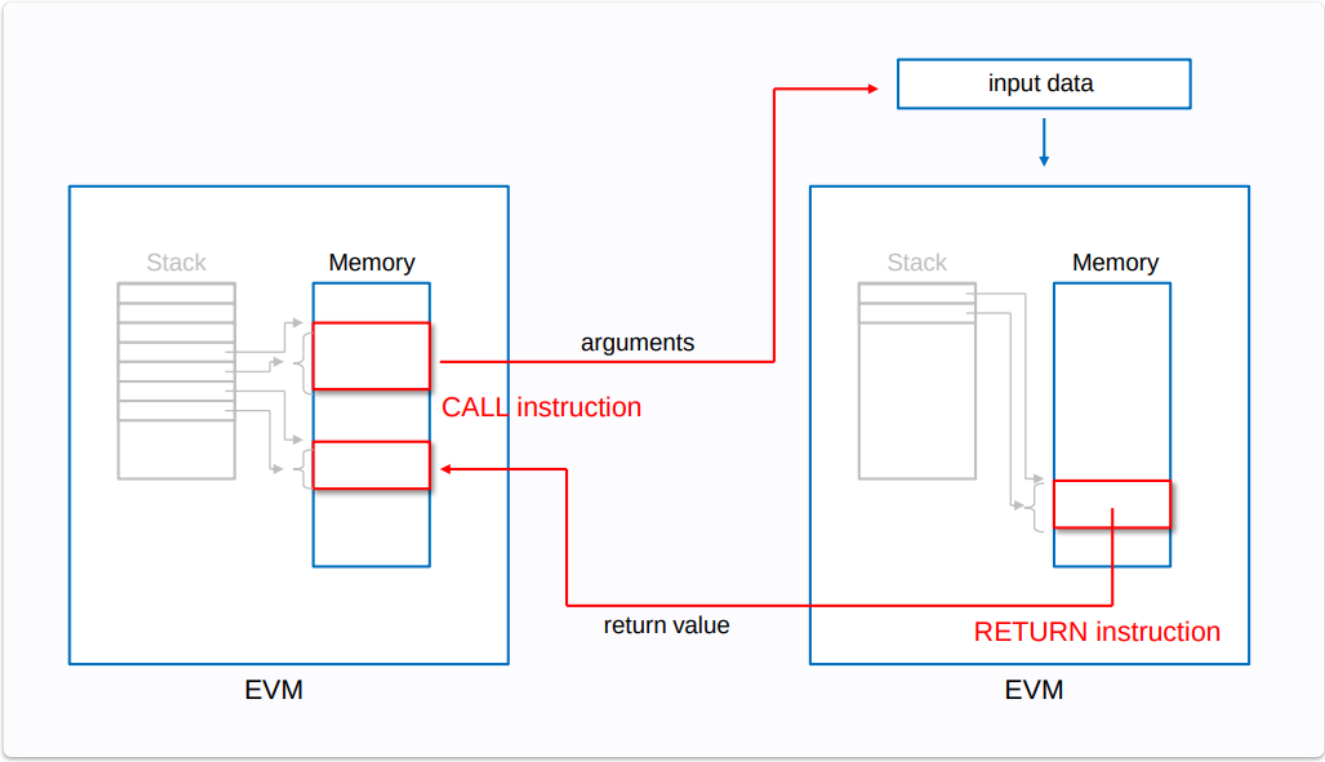
After the init code has run, where is the bytecode ?

From the yellow paper
"init: An unlimited size byte array specifying the EVM-code for the account initialisation procedure.
Init is an EVM-code fragment; it returns the body, a second fragment of code that executes each time the account receives a message call (either through a transaction or due to the internal execution of code).
Init is executed only once at account creation and gets discarded immediately thereafter.

Transaction fields for a contract creation

- transaction type
- nonce,
- gas price,
- gas limit,
- recipient,
- transfer value,
- transaction signature values,
- account initialization (Init)

EXTERNAL CALLS



Yul

Yul language documentation

"Yul does not expose the complexity of the stack itself. The programmer or auditor should not have to worry about the stack."

"Yul is statically typed. At the same time, there is a default type (usually the integer word of the target machine) that can always be omitted to help readability."

DIALECTS

Currently, there is only one specified dialect of Yul. This dialect uses the EVM opcodes as built in functions and defines only the type u256,

QUICK SYNTAX OVERVIEW

Yul can contain

- literals, i.e. 0x123, 42 or "abc" (strings up to 32 characters)
- calls to builtin functions, e.g. add(1, mload(0))
- variable declarations, e.g. let x := 7, let x := add(y, 3) or let x (initial value of 0 is assigned)
- identifiers (variables), e.g. add(3, x)
- assignments, e.g. x := add(y, 3)
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`
- if statements, e.g. `if lt(a, b) { sstore(0, 1) }`
- switch statements, e.g. `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- for loops, e.g. `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- function definitions, e.g. `function f(a, b) -> c { c := add(a, b) }`

Blocks

Blocks of code are delimited by `{}` and the variable scope is defined by the block.

For example

```
function addition(uint x, uint y) public pure returns (uint) {
     assembly {
          let result := add(x, y)    // x + y
          mstore(0x0, result)        // store result in memory
          return(0x0, 32)            // return 32 bytes from memory
      }
 }
```

VARIABLE DECLARATIONS

To assign a variable, use the let keyword

```
 let y := 4
```

Under the hood, the let keyword

- Creates a new stack slot
- The new slot is reserved for the variable.
- The slot is then automatically removed again when the end of the block is reached.

Since variables are stored on the stack, they do not directly influence memory or storage, but they can be used as pointers to memory or storage locations in the built-in functions `mstore`, `mload`, `sstore` and `sload`

ACCESSING VARIABLES

We can access variables that are defined outside the block, if they are local to a soldity function

```
function assembly_local_var_access() public pure {
    uint b = 5;
    assembly {                 // defined inside  an assembly block
        let x := add(2, 3)
        let y := 10
        z := add(x, y)
    }
    assembly {                // defined outside an assembly block
        let x := add(2, 3)
        let y := mul(x, b)
    }
}
```

LITERALS

These work the same way as in Solidity, but there is a maximum length of 32 bytes for string literals.

```
 let a := 0x123             // Hexadecimal
 let b := 42                // Decimal
 let c := "hello world"     // String
```

Control structures

IF STATEMENTS (THERE IS NO ELSE)

```
assembly {
    if lt(a, b) { sstore(0, 1) }
    }
```

Single line statements still requires braces

SWITCH STATEMENTS

```
assembly {
    let x := 0
    switch calldataload(4)  // function selector
    case 0 {
        x := calldataload(0x24) // load 32 bytes from 0x24
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

Note

- Control does not flow from one case to the next
- If all possible values of the expression type are covered, a default case is not allowed.

LOOPS

```
function for_loop_assembly(uint n, uint value)
public pure returns (uint) {

     assembly {

       for { let i := 0 } lt(i, n) { i := add(i, 1) } {
           value := mul(2, value)
       }

       mstore(0x0, value)
       return(0x0, 32)

   }

}
```

WHILE LOOPS

There isn't a while loop per se, but we can modify a for loop to behave as a while loop

```
assembly {
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

FUNCTIONS

```
assembly {

    function allocate(length) -> pos {
        pos := mload(0x40)
        mstore(0x40, add(pos, length))
    }
    let free_memory_pointer := allocate(64)
}
```

An assembly function definition has the function keyword, a name, two parentheses () and a set of curly braces { ... }.
It can also declare parameters. Their type does not need to be specified as we would in Solidity functions.

```
assembly {
    function my_assembly_function(param1, param2) {
        // some code ...
    }
}
```

Return values can be defined by using "->"

```
assembly {

    function my_assembly_function(param1, param2) -> my_result {

        // param2 - (4 * param1)
        my_result := sub(param2, mul(4, param1))

    }
    let some_value = my_assembly_function(4, 9)
}
```

FUNCTION VISIBILITY

We do not have the public / internal / private idea that we have in Solidity. Assembly functions are no part of the external interface of a contract.
Functions are only visible in the block that they are defined in.

Conversion

During assignments and function calls, the types of the respective values have to match. There is no implicit type conversion. Type conversion in general can only be achieved if the dialect provides an appropriate built-in function that takes a value of one type and returns a value of a different type.

Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

0x00 - 0x3f (64 bytes): scratch space for hashing methods

0x40 - 0x5f (32 bytes): currently allocated memory size (aka. free memory pointer)

0x60 - 0x7f (32 bytes): zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to 0x80 initially).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Common examples

CHECKING IF AN ADDRESS IS A CONTRACT

```
function isContract(address _addr) private returns (bool isContract){
  uint32 size;
  assembly {
    size := extcodesize(_addr)
  }
  return (size > 0);
}
```

CALLING A PRECOMPILED CONTRACT

```
function callBn256Add(bytes32 ax, bytes32 ay, bytes32 bx, bytes32 by)
public returns (bytes32[2] memory result) {
    bytes32[4] memory input;

    input[0] = ax;
    input[1] = ay;
    input[2] = bx;
    input[3] = by;
    assembly {
        let success := call
        (50000, 0x06, 0, input, 0x80, result, 0x40)

        switch success
        case 0 {
            revert(0,0)
        }
        mstore(0x0, result)
        return(0x0, 64)
    }


}
```

ANOTHER EXTERNAL CALL

```
assembly {
let x := mload(0x40)
//Find empty storage location using "free memory pointer"
mstore(x,sig) //Place signature at begining of empty storage
mstore(add(x,0x04),a) //Place first argument directly next to signature
mstore(add(x,0x24),b) //Place second argument next to first, padded to 32
bytes

let success := call(
                    5000, //5k gas
                    addr, //To addr
                    0,    //No value
                    x,    /Inputs are stored at location x
                    0x44, //Inputs are 68 bytes long
                    x,    //Store output over input (saves space)
                    0x20) //Outputs are 32 bytes long

c := mload(x) //Assign output value to c
mstore(0x40,add(x,0x44)) // Set storage pointer to empty space
}
```

CODECOPY and EXTCODECOPY

CODECOPY copies bytes from this contract into memory
EXTCODECOPY copies bytes from another contract into memory

CREATE and CREATE2

The CREATE opcode creates a contract (for example from the `new` keyword)
The created contract address is found from

```
new_address = hash(sender, nonce)
```

CREATE2 was introduced later (Constantinople hard fork) to allow a deterministic address

```
new_address = hash(0xFF, sender, salt, bytecode)
```

You can call CREATE2 from Solidity (See docs)

DeFi projects often use this to make address of their contracts on mainnet and test networks the same.

CREATE3 from 0xsequence

Repo
Example

```
//SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import "@0xsequence/create3/contracts/Create3.sol";


contract Child {
  function hola() external view returns (string) {
    return "mundo";
  }
}

contract Deployer {
  function deployChild() external {
    Create3.create3(keccak256(bytes("<my salt>")), type(Child).creationCode);
  }
}
```

Also see EIP-3171

Usful Tools

https://www.evm.codes/ - OpCodes and Playground
https://ethervm.io/decompile - Decompiler
https://github.com/Arachnid/evmdis - EVM Dissasembler
https://www.trustlook.com/services/smart.html - Decompiler