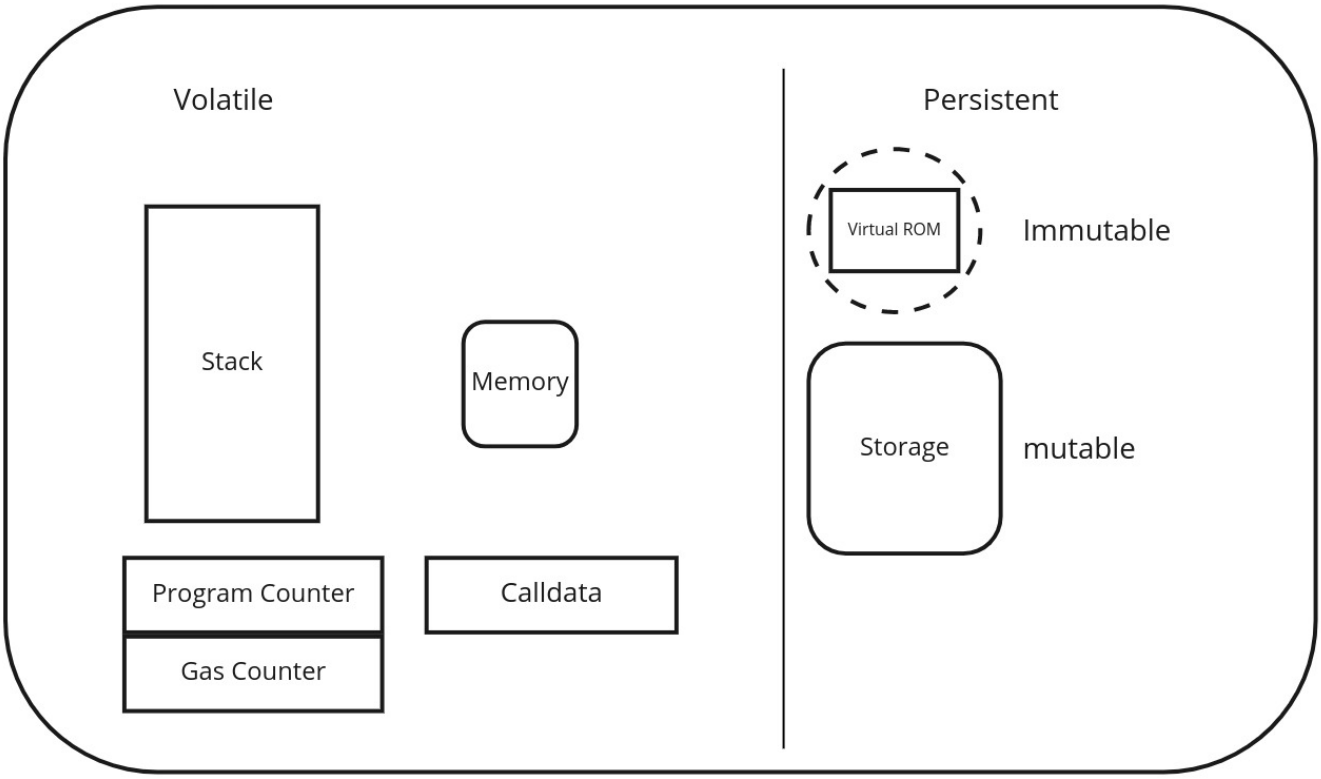
Solidity Assembly

EVM - A distributed state machine

Good In depth Video (https://www.youtube.com/watch?v=RxL_1AfV7N4)

The EVM

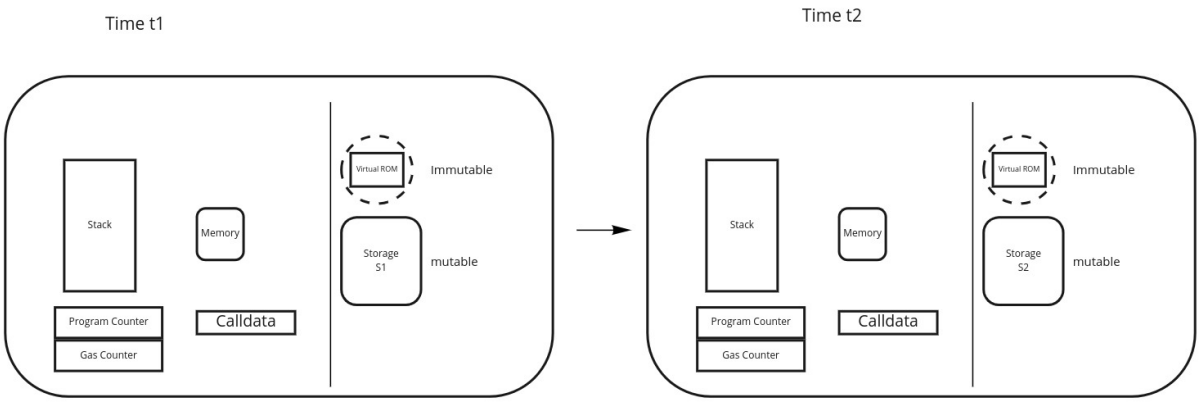


Data areas

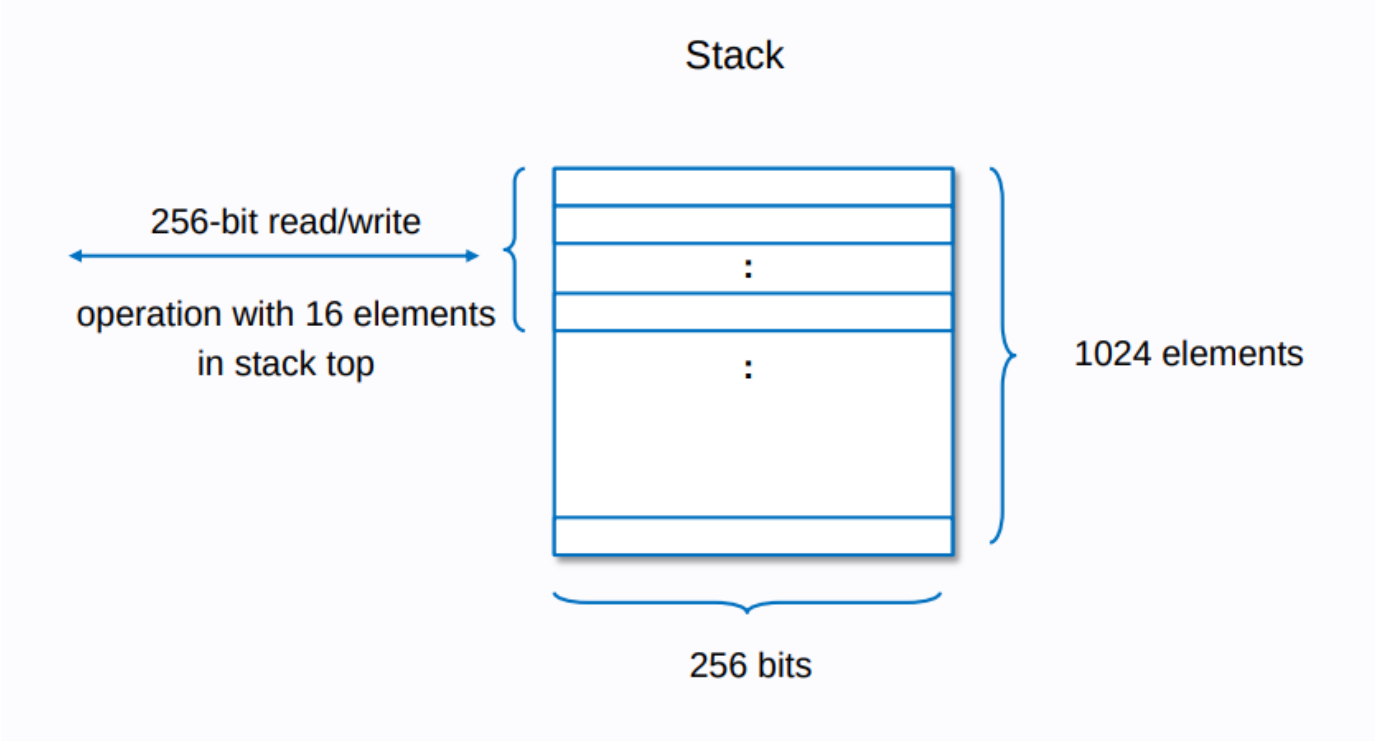
Data can be stored in

- Stack
- Calldata
- Memory
- Storage
- Code
- Logs

EVM State transition

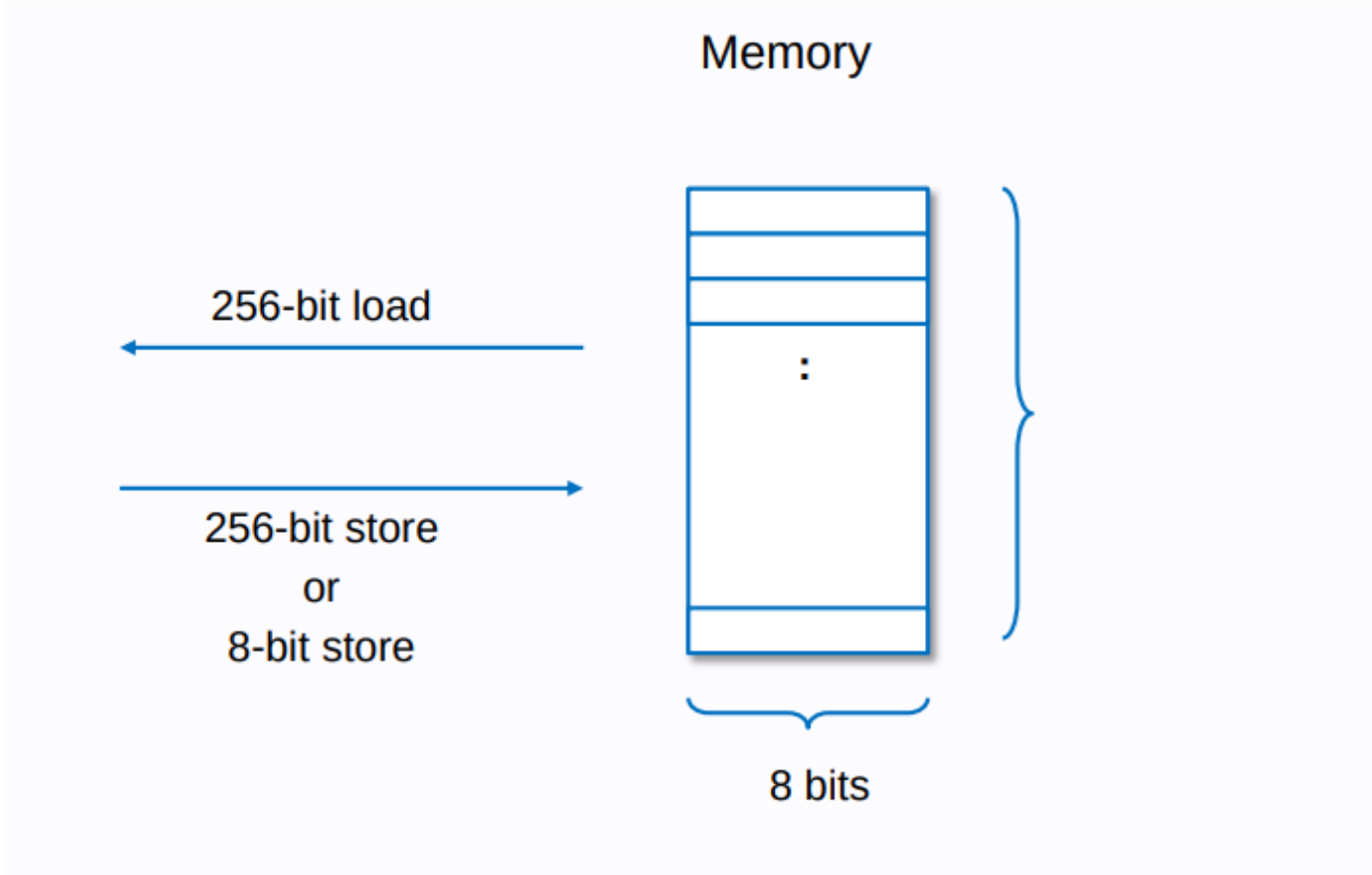


The Stack



The top 16 items can be manipulated or accessed at once (or stack too deep error)

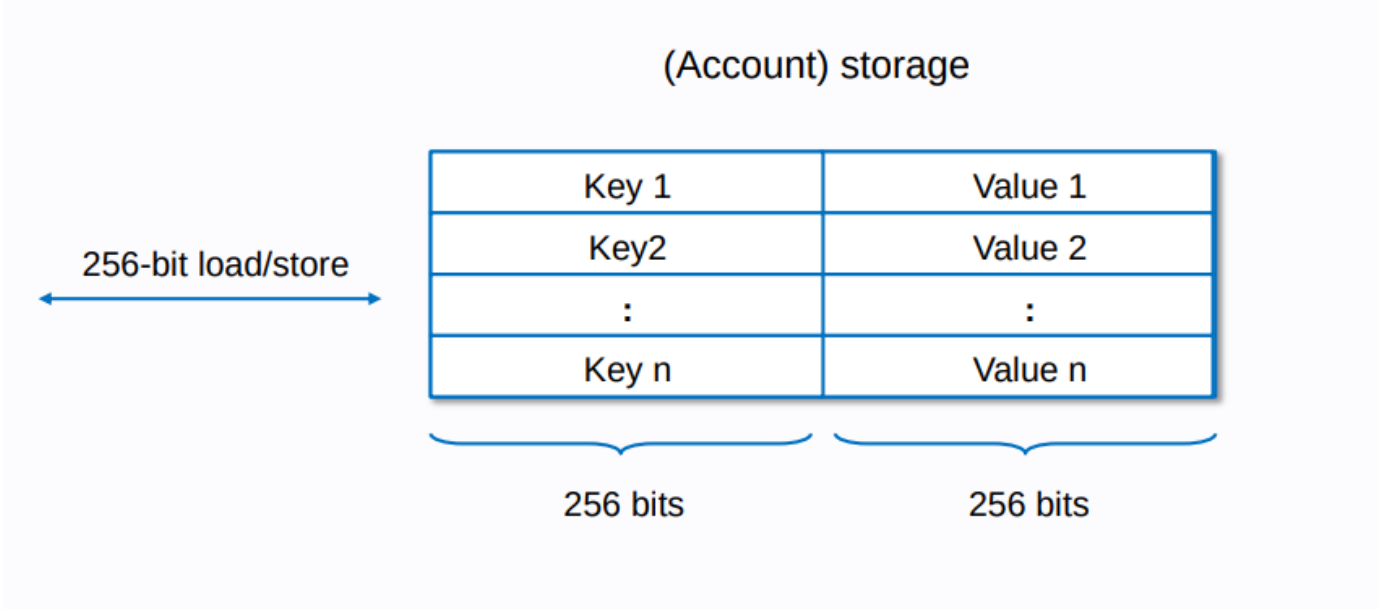
Memory



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. Since memory is contiguous, it does save gas to keep it packed and shrink its size, instead of having large patches of zeros.

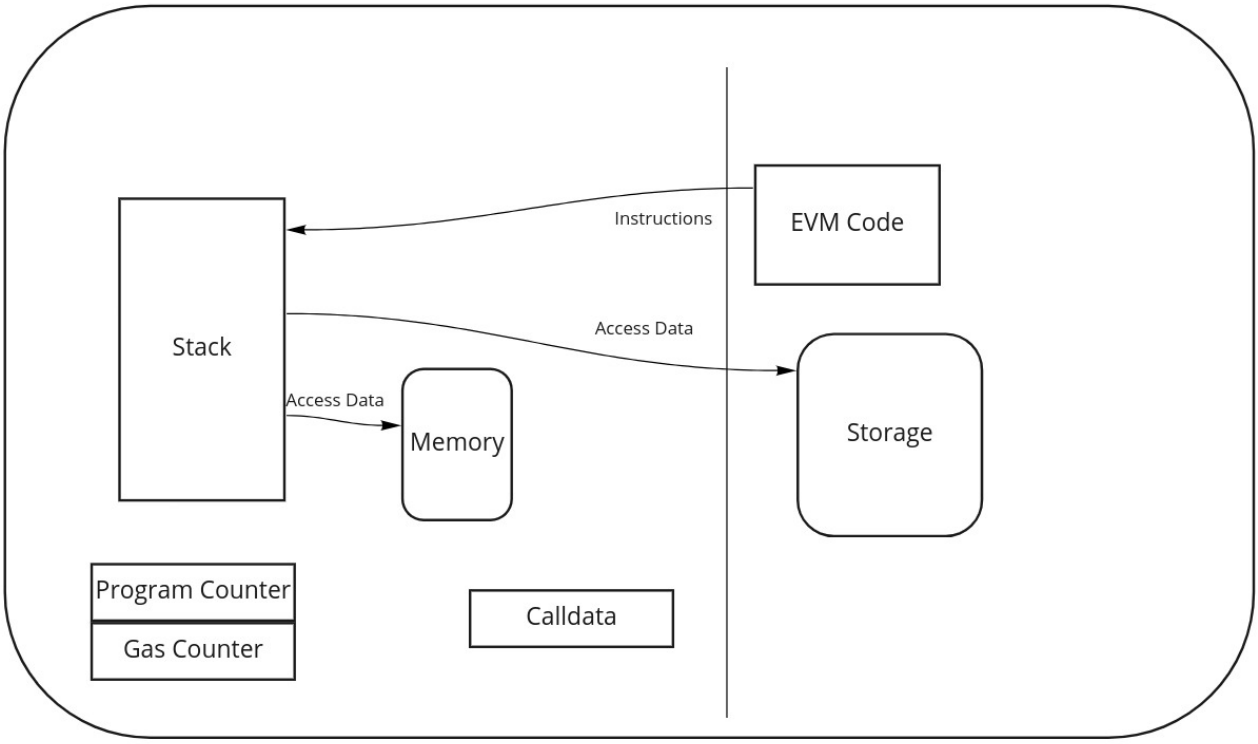
- MLOAD loads a word from memory into the stack.
- MSTORE saves a word to memory.
- MSTORE8 saves a byte to memory.

Storage



See Documentation (https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html)

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)

https://www.ethervm.io/ (https://www.ethervm.io/)

Opcodes

uint8	Mnemonic	Stack Input	Stack Output	Expression
00	STOP	-	-	STOP()
01	ADD	a b	a + b	a + b
02	MUL	a b	a * b	a * b
03	SUB	a b	a - b	a - b
04	DIV	a b	a // b	a // b

```
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 (https://en.wikipedia.org/wiki/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

```
PUSH 2
PUSH 4
ADD
```

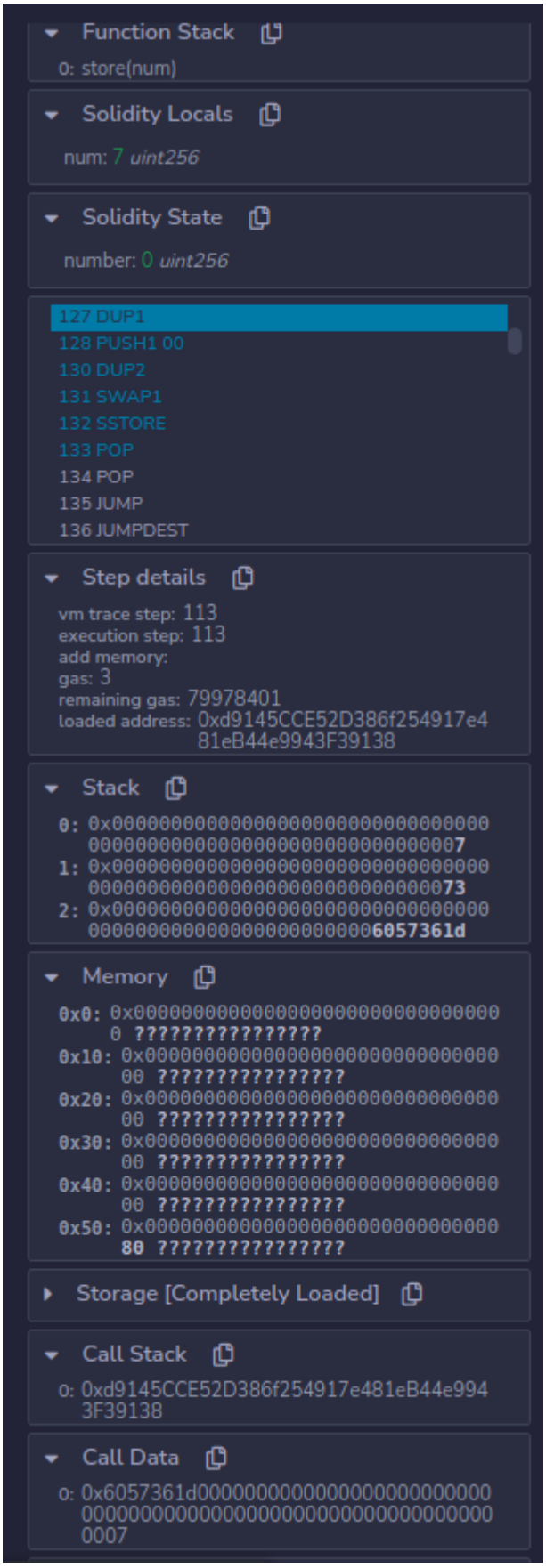
```
PUSH 2
|__2__|

PUSH 4
|__4__|
|__2__|

ADD
|__6__|
```

A more complex example

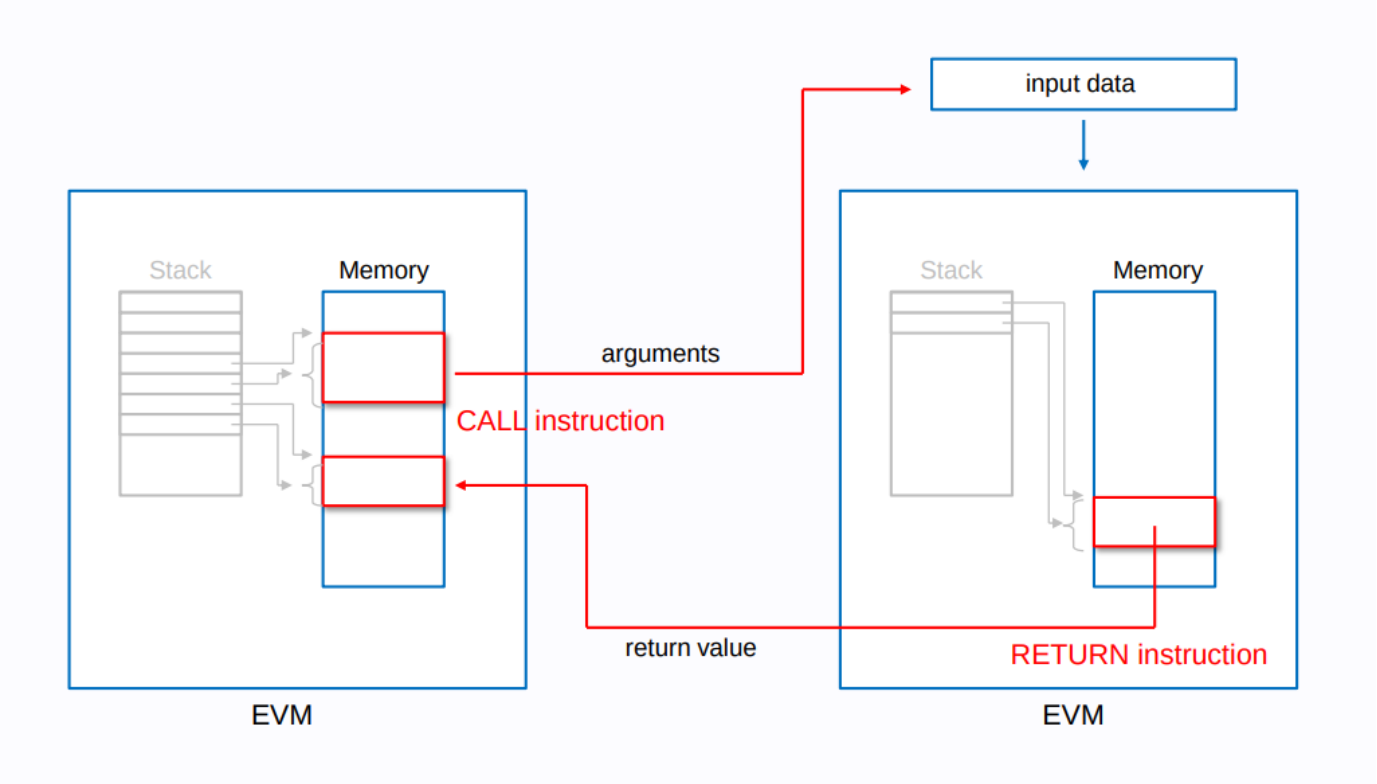
```
mstore(0x80, add(mload(0x80), 2))
2 0x80 mload add 0x80 mstore

                      >
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;
    }
```



External Calls



Assembly (Yul)

Yul language documentation (https://docs.soliditylang.org/en/v0.8.11/yul.html)

“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)
}
```

There is no explicit return keyword in Yul, to return a value simply assign it in the final statement (similar to other languages) Instead you can use the `leave` keyword.

Note there is an EVM OPCODE return which stops the current execution

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.

Yul+

Adds

- Memory structures (mstruct)
- Enums (enum)
- Constants (const)
- Ethereum standard ABI signature/topic generation (sig"function …", topic"event …)
- Booleans (true, false)
- Safe math (over/under flow protection for addition, subtraction, multiplication)
- Injected methods (mslice and require)

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
}
```

CREATE3 from 0xsequence

Repo (https://github.com/0xsequence/create3)
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);
  }
}
```

SLOAD2

Repo (https://github.com/0xsequence/sstore2)

SLOAD2 is a set of Solidity libraries for writing and reading contract storage paying a fraction of the cost, it uses contract code as storage, writing data takes the form of contract creations and reading data uses EXTCODECOPY.

Interesting Blog : How I Almost Cheesed the EVM (https://zefram.xyz/posts/how-i-almost-cheesed-the-evm/)

Tracking the nonce using balance

Usful Tools

https://ethervm.io/decompile (https://ethervm.io/decompile) - Decompiler
https://github.com/Arachnid/evmdis (https://github.com/Arachnid/evmdis) - EVM Dissasembler
https://www.trustlook.com/services/smart.html (https://www.trustlook.com/services/smart.html) - Decompiler