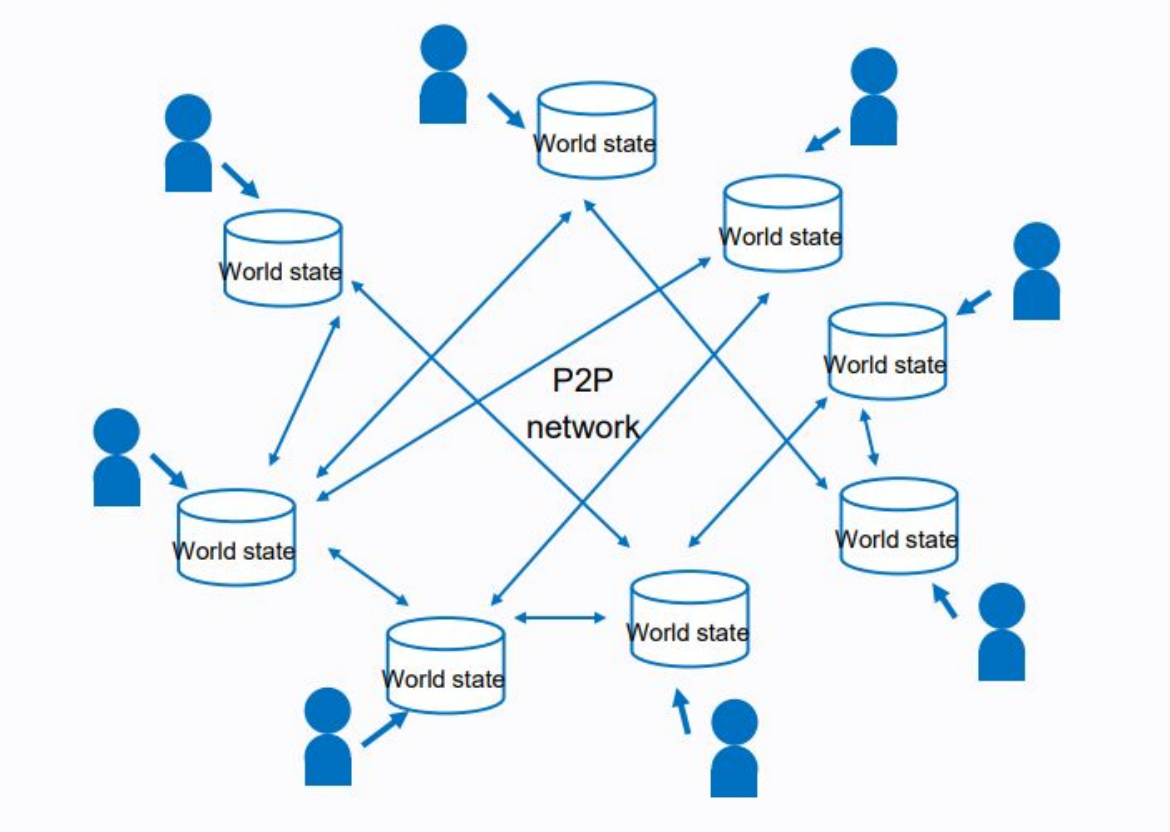
Lecture 11

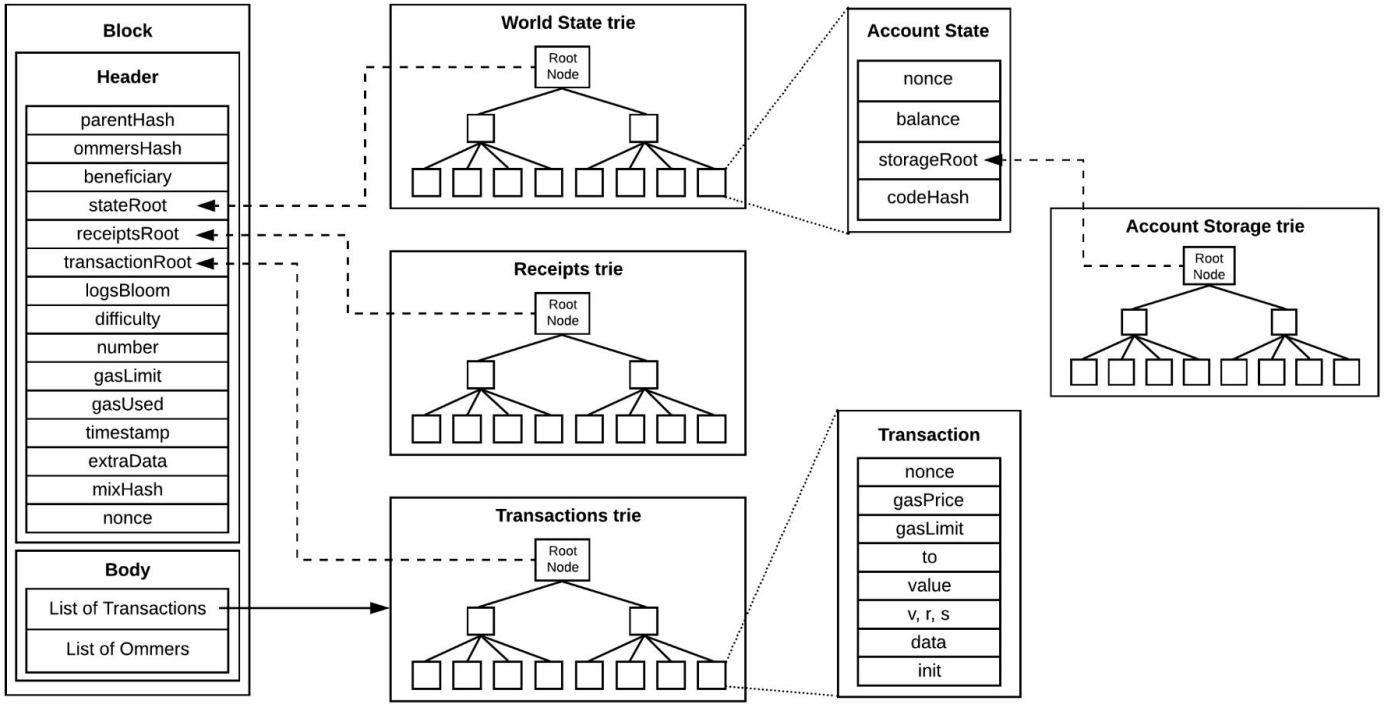
Gas Optimization

Ethereum Refresher



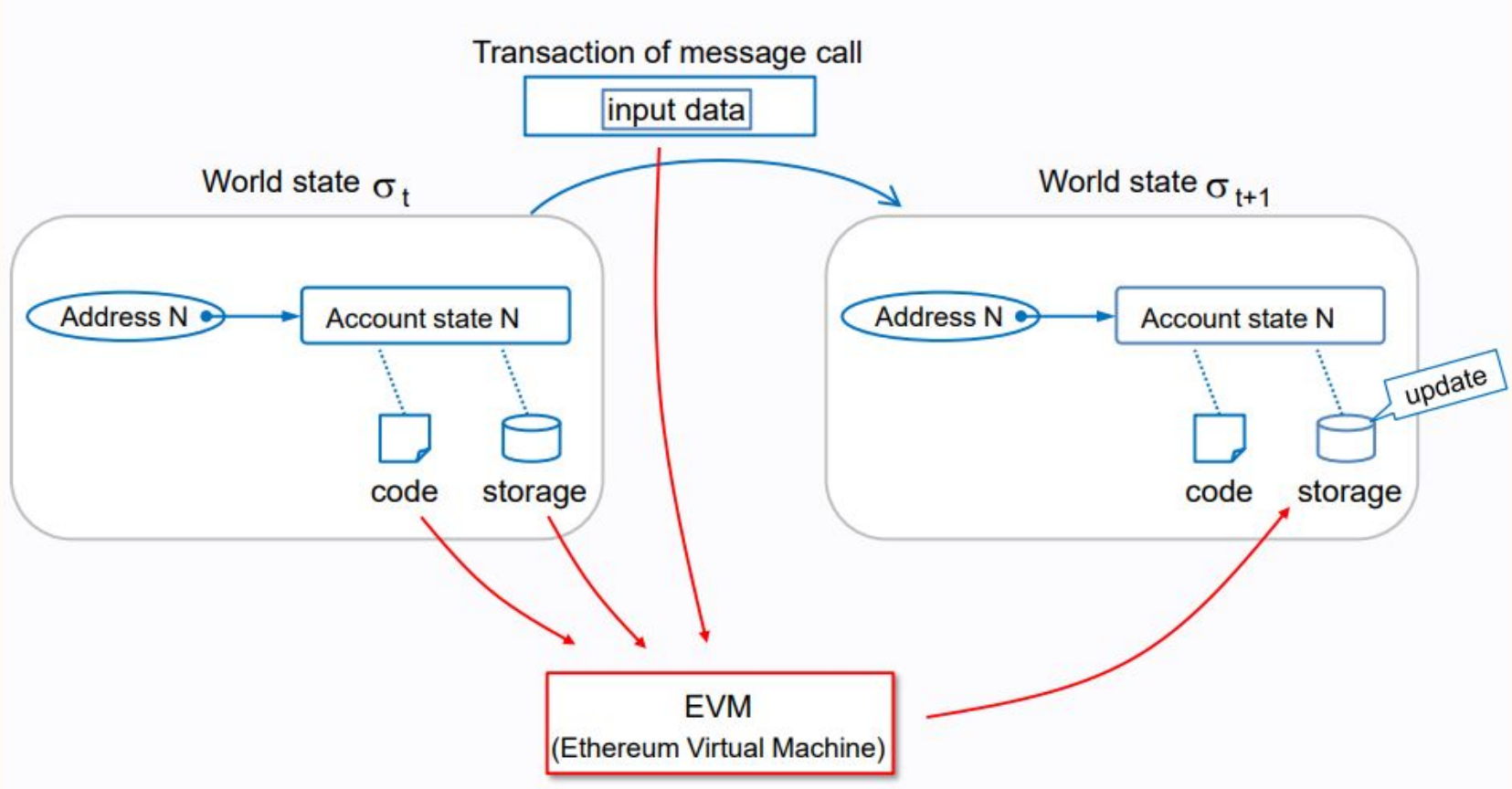
- Every miner / validator keeps a copy of the world state.
- The world state is propagated through the consensus mechanism
  - Contrast IPFS/Torrent's mechanism?
- The world state is updated through atomic transactions
- Transactions are selected based on gas fees paid

Data Location - Storage, Code, Log (Lecture 3)

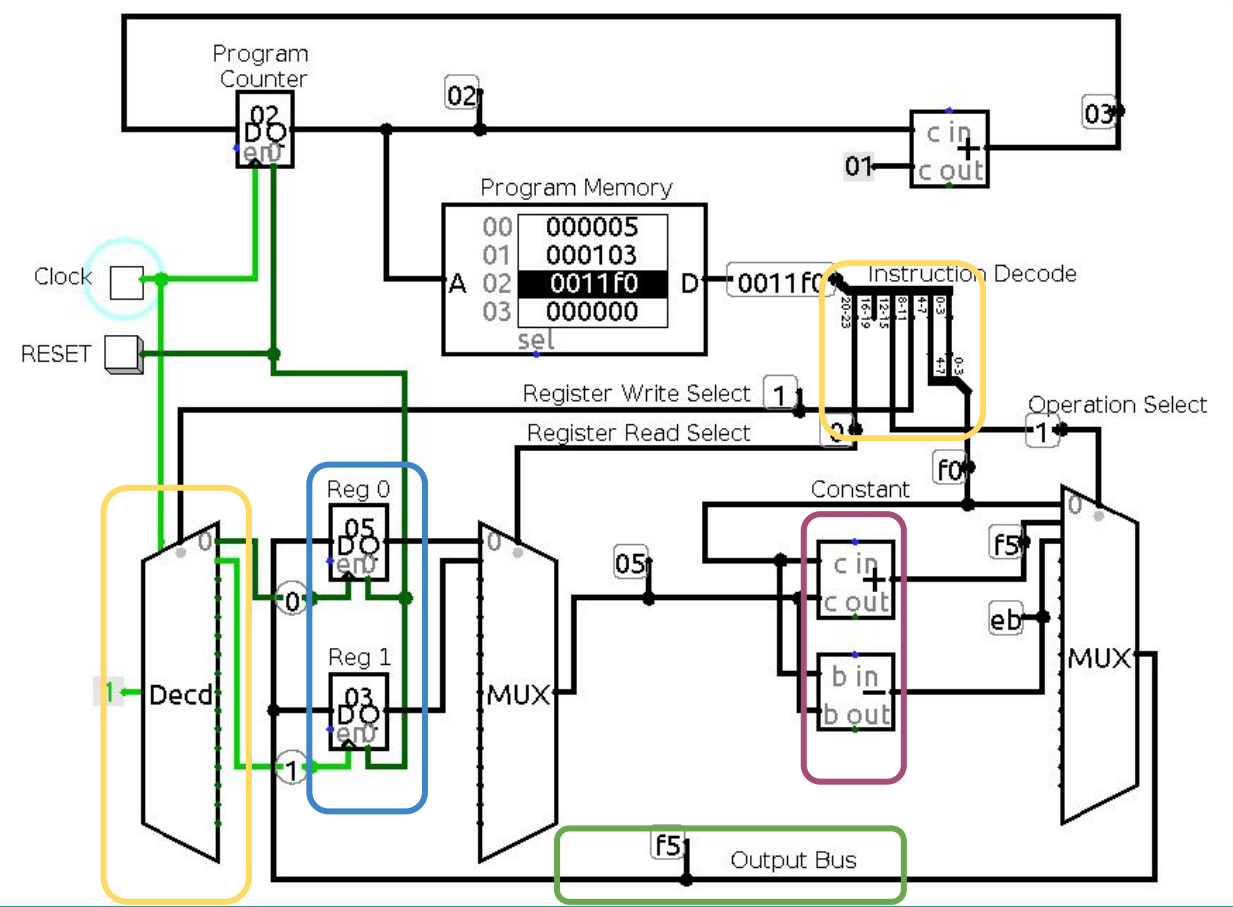


Block, transaction, account state objects and Ethereum tries

Ethereum Refresh - EVM Architecture



Ethereum Refresh - CPU execution (Lecture 1)



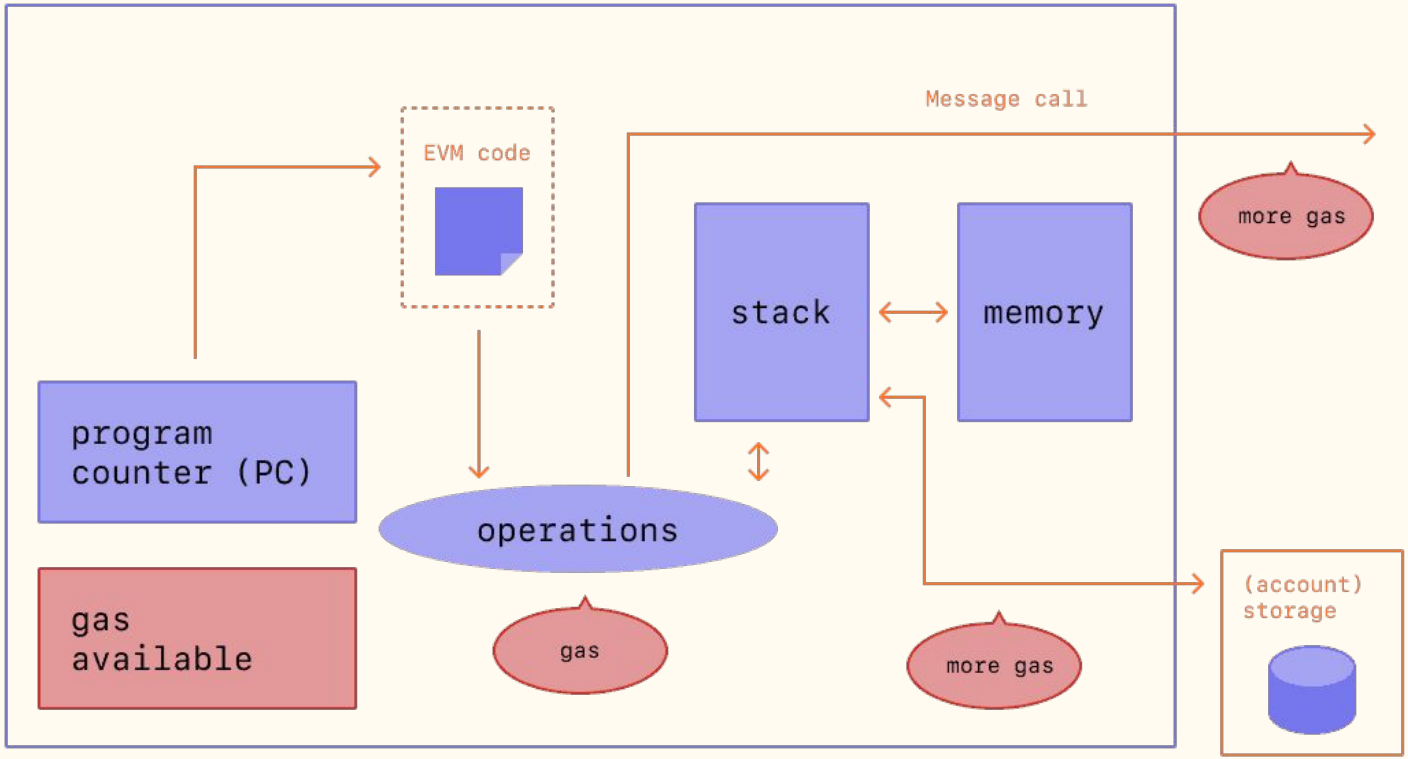
Decode Instructions into opcode and data

Registers to hold **program** essentials: Data, loops state, pointers

Algorithmic Logic Unit

Ram read write, towards more permanent storage. Indexed by **Addresses**

Ethereum Refresh - EVM architecture (Lecture 1)

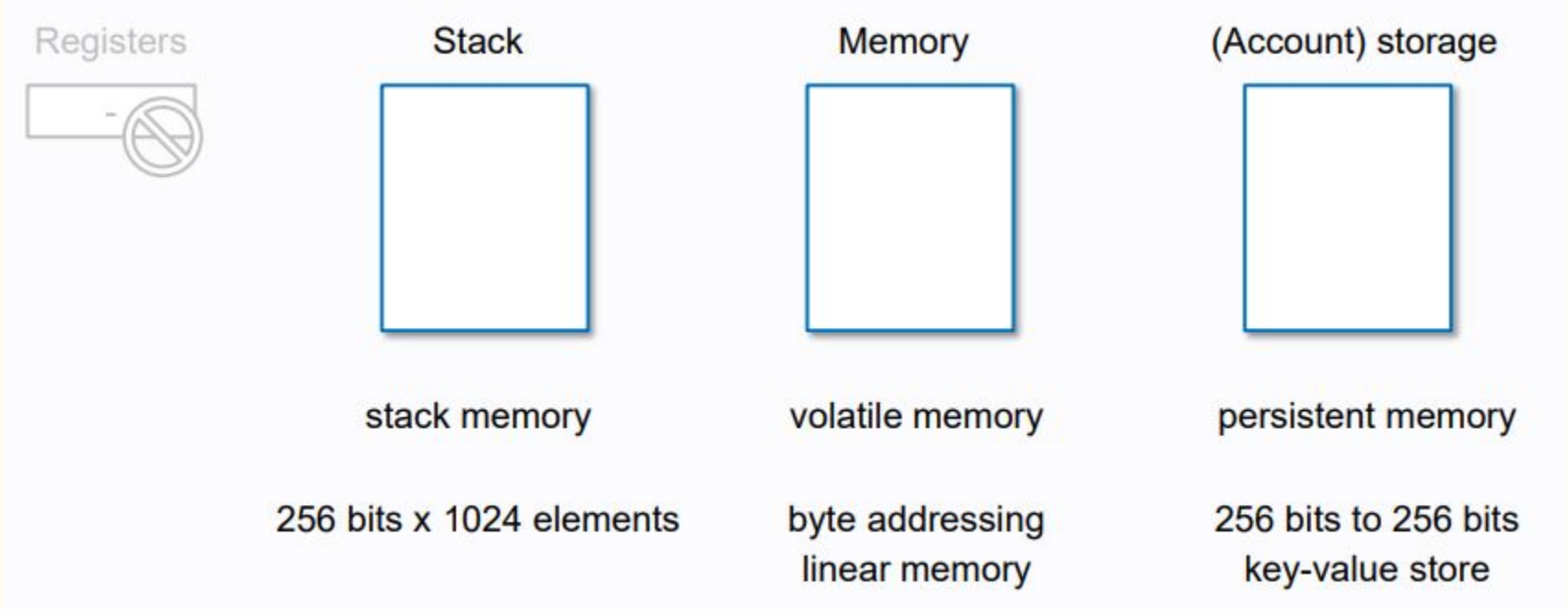


EVM Memory

6 types of memory

Memory Type	Size	Usage
Storage	Large	For all global variable which needs to be stored onchain. Very Expensive
Memory	Medium	For all local variables that only live for the duration of the contract execution.
Stack	Small	Immediate execution
Calldata	Tiny	Immutable user inputs
Bytecode	Small / Medium	A hash of contract bytecode
Logs	Onchain	Emitted events. onchain logs

Data Location - Memory, Stack, Calldata



Memory types - can you identify?

Santa's naughty list

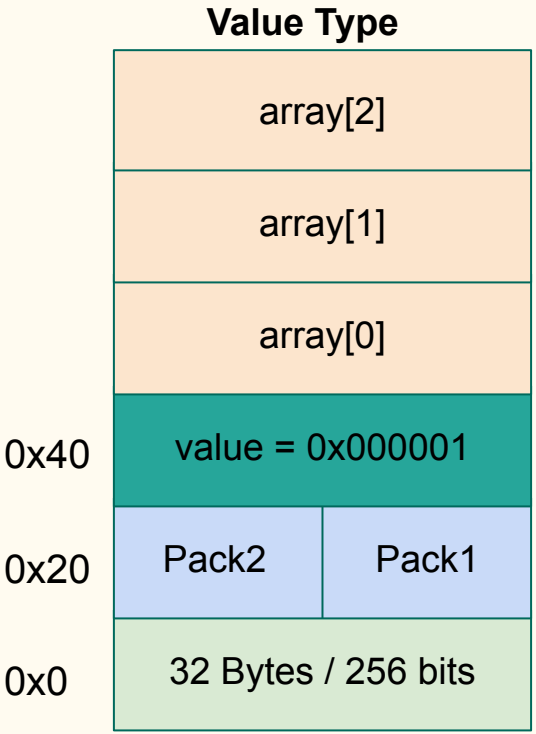
Storage:
naughty_list
names

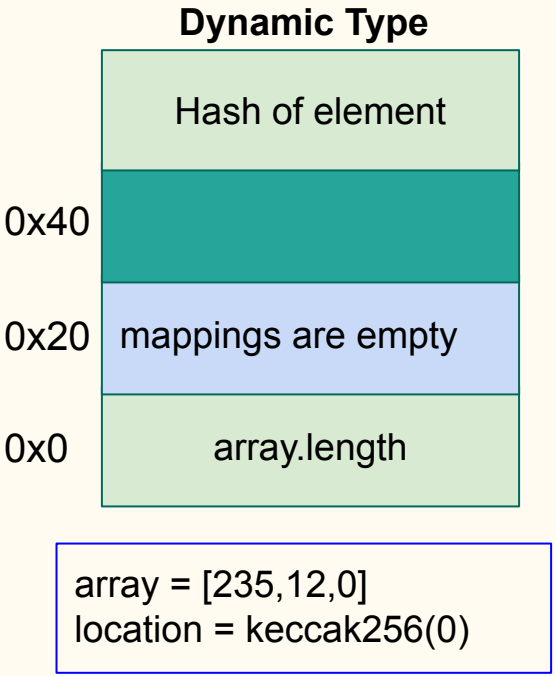
Constructor:
Hashed inside bytecode

Memory:
name - user input
i - index in for loop

```
1  //SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.17;
3
4  contract christmas{
5      mapping(string => uint) private naughty_list;
6      string[] private names;
7
8      constructor (){
9          names = ["Annie", "Tim", "Mark"];
10         naughty_list["Annie"] = 0;
11         naughty_list["Tim"] = 0;
12         naughty_list["Mark"] = 0;
13     }
14
15     function increase_naughty_score(string memory name) public {
16         for(uint i = 0; i < names.length; i++){
17             if (keccak256(abi.encodePacked(names[i]))== keccak256(a
18                 naughty_list[name] += 10;
19             } else {
20                 naughty_list[name] = 10;
21             }
22         }
23     }
24 }
25
```

Storage Layout





Points to remember:

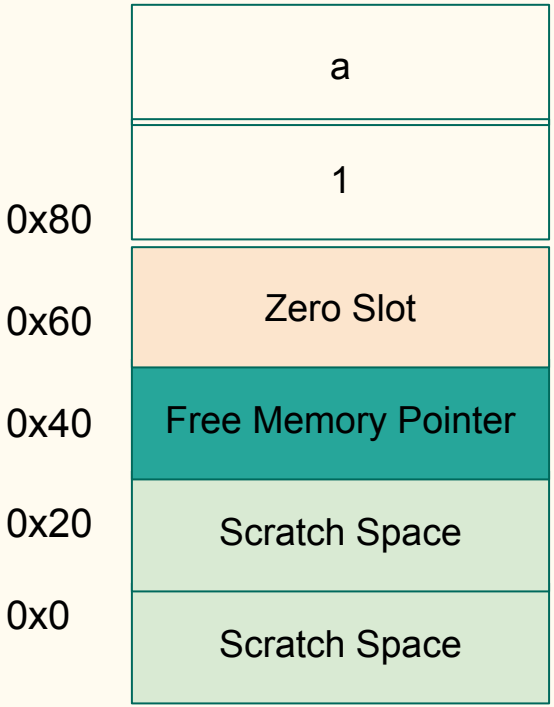
- Declared arrays and structs are assigned a block together and considered a value type
- Packed blocks may incur additional gas when not updated at the same time
- Recursive hashing possible
- What about inheritance?

See it in action - USDC

Do you recall what are the global variables of an ERC20 token? (Lecture 4)

USDC storage interactive representation

Memory Layout



No packing!

a = 0x61000000000000 (high order alignment)

1 = 0x00000000000001 (low order alignment)

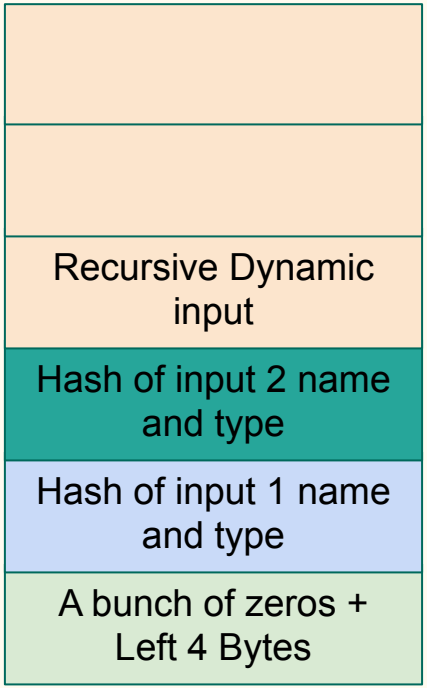
Always 0. Optimized initialization

Pointer to the next free slot address where a new variable can be added.

Use for intermediate executions in inline-assembly

Calldata Layout

keccak256(function name, address, input name, type)



Dynamic inputs(strings, arrays mappings) work the same way as storage for - They point to different locations in memory of their elements

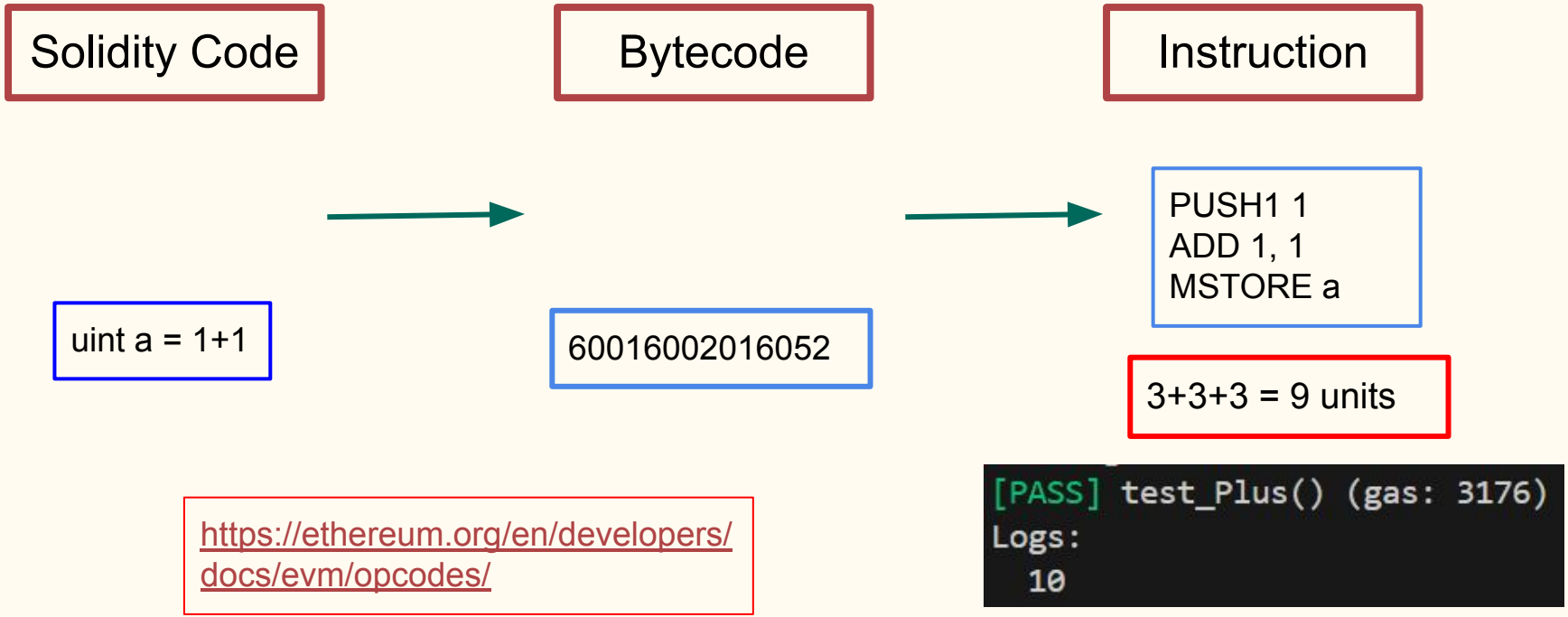
Each input must be padded to 32 Bytes

Hash of the type, name and value of the function input

This is the function signature. Matches input to function

Gas costs and Testing

Is 1 + 1 complex?



Solidity Contract Creation

60 80 60 40 52

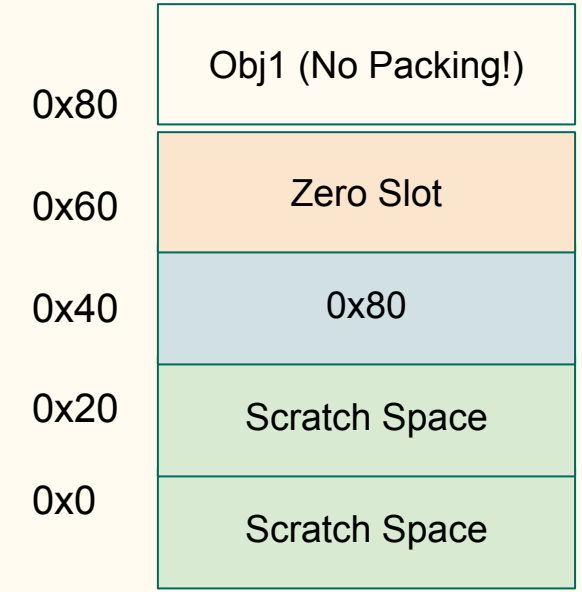
PUSH1 80
PUSH1 40
MSTORE

Free Memory Pointer established!

Stack
(1024 element limit)

1	40
0	80

Memory



Why does it matter? - Gas Cost Testing

https://www.npmjs.com/package/hardhat-gas-reporter

Solc version: 0.8.18	Optimizer enabled: false	Runs: 200	Block limit: 30000000 gas	
Methods				
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**eur (avg)**
christmas	increase_naughty_score	58389	83733	71061	2	-
Deployments					**% of limit**	
christmas		-	-	666290	2.2 %	-

666290 gwei = 0,000724 ETH = 1,19 EUR

For every naughty act of every child, Santa has to pay 1,2 EUR. There are potentially millions of naughty in children in the world doing naughty things everyday! This is a very expensive list for Santa to keep. Can he get it cheaper?

The Cost of Memory

CREATE	32000 + memory expansion + per-byte bytecode hash cost
MLOAD	3 + offset cost (how many slots from start)
MSTORE	3 + offset cost
SLOAD	Cold Access (1st time): 2100 ; Warm Access: 100
SSTORE	highly variable

Cost of SSTORE - Setup simple checks

gas_cost = 0
gas_refund = 0 - provided for cleaning state. Only available in sstore

If gas_left <= 2300:
    throw OUT_OF_GAS_ERROR (can not sstore with < 2300 gas for backwards compatibility)
If (context_addr, target_storage_key) not in touched_storage_slots (cold access/SLOAD):
    gas_cost += 2100
If new_val == current_val (no-op):
    gas_cost += 100

Cost of SSTORE - A Zero Game

```
Else new_val != current_val:
    If current_val == orig_val ("clean slot"):
        If orig_val == 0 (zero ->  zero -> nonzero):
            gas_cost += 20000
        Else orig_val != 0 ( nonzero -> nonzero ->  nonzero):
            gas_cost += 2900
            If new_val == 0 (nonzero -> nonzero -> zero):
                gas_refund += 4800
    Else current_val != orig_val ("dirty slot", already updated in current execution context):
        gas_cost += 100
        If orig_val != 0 (execution context started with a nonzero value in slot):
            If current_val == 0 ( nonzero -> zero - > nonzero):
                gas_refund -= 4800
            Else if new_val == 0 (nonzero -> different nonzero -> zero):
                gas_refund += 4800
            If new_val == orig_val (slot is reset to the value it started with):
                If orig_val == 0 (zero -> nonzero ->  zero):
                    gas_refund += 19900
                Else orig_val != 0 (nonzero ->different nonzero -> orig nonzero):
                    gas_refund += 2800
```

Gas Optimization techniques

Reducing transaction costs

Santa's naughty list

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

contract christmas{
    mapping(string => uint) public naughty_list;
    string[] public names;

    constructor (){
        names = ["Annie", "Tim", "Mark"];
        naughty_list["Annie"] = 0;
        naughty_list["Tim"] = 0;
        naughty_list["Mark"] = 0;
    }

    function get_score(string memory name) public view returns(uint){
        return naughty_list[name];
    }

    function increase_naughty_score(string memory name) public {
        bool not_in_list = true;

        for(uint i = 0; i < names.length; i++){
            if (keccak256(abi.encodePacked(names[i]))== keccak256(abi.encodePacked(name))){
                not_in_list = false;
                naughty_list[names[i]] += 10;
            }
        }

        if (not_in_list){
            names.push(name);
            naughty_list[name] = 10;
        }
    }
}
```

According to some Christmas traditions, Santa keeps a list of naughty children. If you were very naughty, there will be no presents under the Christmas tree for you. In fact, there will be some coal if you were extra naughty.

Santa is now keeping his naughty list on the blockchain! How expensive is it for him?

Cost of transactions - Gas Calculations

```
require("@nomicfoundation/hardhat-toolbox");
require("hardhat-gas-reporter");

module.exports = {
  solidity: "0.8.18",
  gasReporter: {
    currency: "EUR",
    gasPrice: 15,
    enabled: true
  }
};
```

Gas exists to prevent denial of service attacks.
gas units = units per opcode
gas price = gwei per unit gas you are willing to pay
gas limit = max amount willing to pay

Solc version: 0.8.18	Optimizer enabled: false	Runs: 200	Block limit: 30000000 gas
Methods	15 gwei/gas		1667.06 eur/eth
Contract	Method	Min	Max	Avg	# calls	eur (avg)
christmas	increase_naughty_score	58433	83777	71105	2	1.78
Deployments					% of limit	
christmas		-	-	781291	2.6 %	19.54

Technique 1 - restrict scope in functions and variables

- Everytime a public global variable is defined, Solidity automatically creates a getter and setter function for that variable.
- function scope
  - Public functions have arguments copied into memory
  - External functions have args copied into calldata
  - private / internal is very cheap

Optimizer enabled: false	Runs: 200	Block limit: 30000000 gas
15 gwei/gas		1670.23 eur/eth

Min	Max	Avg	# calls	eur (avg)
58389	83733	71061	2	1.78
			% of limit	
-	-	666290	2.2 %	16.69

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

contract christmas{
    mapping(string => uint) private naughty_list;
    string[] private names;

    constructor (){
        names = ["Annie", "Tim", "Mark"];
        naughty_list["Annie"] = 0;
        naughty_list["Tim"] = 0;
        naughty_list["Mark"] = 0;
    }

    function get_score(string memory name) external view returns(uint){
        return naughty_list[name];
    }

    function increase_naughty_score(string memory name) external {
        bool not_in_list = true;

```

Technique 2 - Global variable packing

Always declare globals in a way that fits into 32 Bytes/256 bits

```
contract Integers{
  uint16 a;
  uint b;
  uint16c;
}
```

→

```
contract Integers{
  uint16 a;
  uint16 c;
  uint b;
}
```

We just made our contract more expensive!!!

- mappings and dynamic arrays use hashes and are not stored sequentially like structs
- we had to add an extra check function since we had the score a lot smaller. Checks are good though!

```
contract christmas{
    mapping(string => uint8) private naughty_list;
    string[] private names;

    constructor (){
        names = ["Annie", "Tim", "Mark"];
        naughty_list["Annie"] = 0;
        naughty_list["Tim"] = 0;
        naughty_list["Mark"] = 0;
    }

    function increase_naughty_score(string memory name) external {
        bool not_in_list = true;

        for(uint i = 0; i < names.length; i++){
            if (keccak256(abi.encodePacked(names[i]))== keccak256(abi.encodePacked(name))){
                not_in_list = false;
                if(naughty_list[names[i]] >= 245){
                    return;
                }
                naughty_list[names[i]] += 10;
            }
        }
```

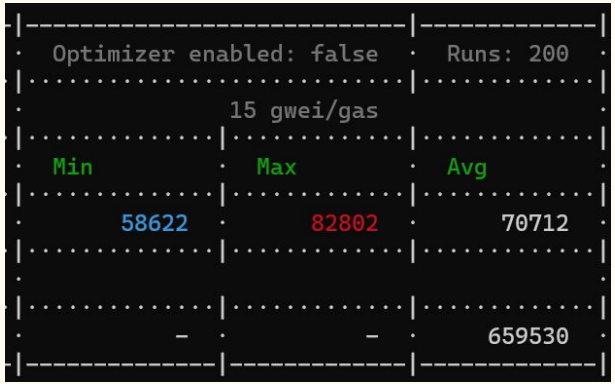
Optimizer enabled: false	Runs: 200	
15 gwei/gas		
Min	Max	Avg
59494	83787	71641
-	-	703980

Technique 3 - Use calldata as much as possible

Recap: Calldata is much smaller and immutable as compared to memory. Therefore it is much cheaper. Try to not mutate user inputs when creating functions. If you do need to change the variable value inside the function, then use memory.

```
15      function increase_naughty_score(string calldata name) external {
```

```
34      function get_score(string calldata name) external view returns(uint){
```

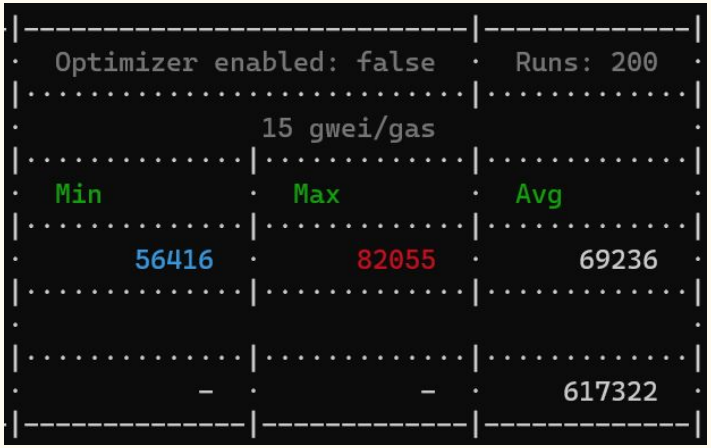


Technique 4 - Work in memory, avoid loops and repetition

SLOAD is much more expensive than MLOAD. if you need to work on your global variables, copy it into memory. Reduce the number of read and writes to storage.

Avoid dynamic arrays if possible. Rather have a large continuous block of memory than pointers. Alternatively, change arrays to mappings or combine them. Note, strings are just a dynamic Byte array.

Avoid repeating functions in loops! Your loops should have as much pre-computed variables as possible.



Optimizer enabled: false	Runs: 200
15 gwei/gas	
Min	Max	Avg
56416	82055	69236
-	-	617322

```
function increase_naughty_score(string calldata name) public {
    bool not_in_list = true;
    bytes32 inputHash = keccak256(abi.encodePacked(name));

    for(uint i = 0; i < names.length; i++){
        if (keccak256(abi.encodePacked(names[i]))== inputHash){
            not_in_list = false;
            naughty_list[name] += 10;
        }
    }
```

Technique 5 - Respect Solidity's way of thinking

Solidity represents a huge mind shift from traditional programming. It is hard to manipulate strings and mappings because Solidity is meant to support transactions. Try to think like a bank and anonymous accounts are submitting transactions to you.

```
1   //SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.17;
3
4   contract christmas{
5       mapping(address => uint) private naughty_list;
6
7       function get_score(address user) external view returns(uint){
8           return naughty_list[user];
9       }
10
11      function increase_naughty_score(address user) public {
12          naughty_list[user] += 10;
13      }
14  }
```

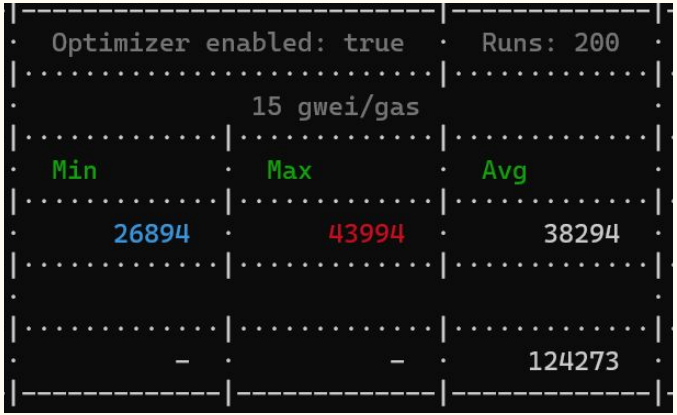
Solc version: 0.8.18	Optimizer enabled: false	Runs: 200	Block limit: 30000000 gas
Methods	15 gwei/gas		1656.13 eur/eth
Contract	Method	Min	Max	Avg	# calls	eur (avg)
christmas	increase_naughty_score	27192	44292	38592	3	0.96
Deployments				% of limit	
christmas		-	-	192181	0.6 %	4.77

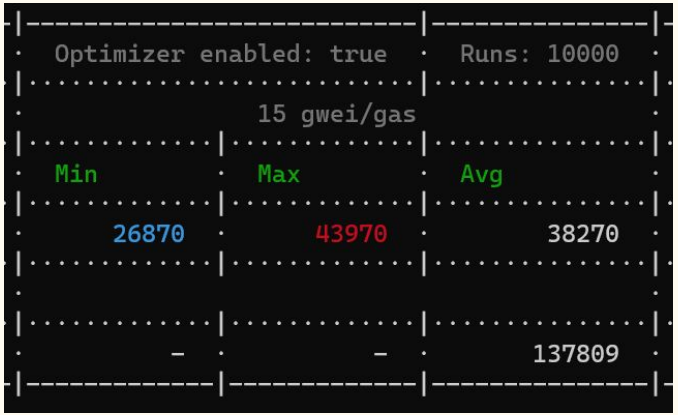
Technique 6 - Enable the compiler optimizer

The solidity optimizer will do a number of operations to automatically make your code more efficient

- Code sanitization through dependency graph
  - unused, duplicate variables
- Opcode Based optimization
  - CommonSubexpressionEliminator
  - Inline Assembly memory management

forge build –optimize –optimizer-runs 10000





The number of runs indicate the number of times your contract will be called.
The optimiser will try to reduce function call cost at the expense of deployment cost.
We have a very scalable contract here!

Forge Gas Reporting

Forge is able to produce a variety of gas reports:

- forge test --gas-report
  - cost of contract deployment
  - min / max / avg gas costs
  - Number of times the function was called in the test suite (frequency check)
- forge snapshot
  - generates a .gas-snapshot file in your project
  - shows the gas cost of each test
  - comparisons possible between different snapshots possible with the -diff and -check commands
  - filtering possible to reach gas cost goals

https://book.getfoundry.sh/forge/gas-tracking

Inline Assembly

Programming in Opcodes

Yul - Operations

Instruction	Explanation
let	This is required before defining a variable. Since all values are bytes, there is no need to assign a value type.
:=	Solidity equivalent: x = y
add(x,y)	Solidity equivalent: x + y
sub(x,y)	Solidity equivalent: x - y
mul(x,y)	Solidity equivalent: x * y
div(x,y)	Solidity equivalent: x / y (or 0 if y equals 0)
mod(x,y)	Solidity equivalent: x % y (or 0 if y equals 0)
lt(x,y)	Solidity equivalent: x < y
gt(x,y)	Solidity equivalent: x > y
eq(x,y)	Solidity equivalent: x == y
iszero(x)	Solidity equivalent: x == 0

Yul - Loops

For Loop

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

While Loop

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

There are no while loops. They are for loops with less inputs.

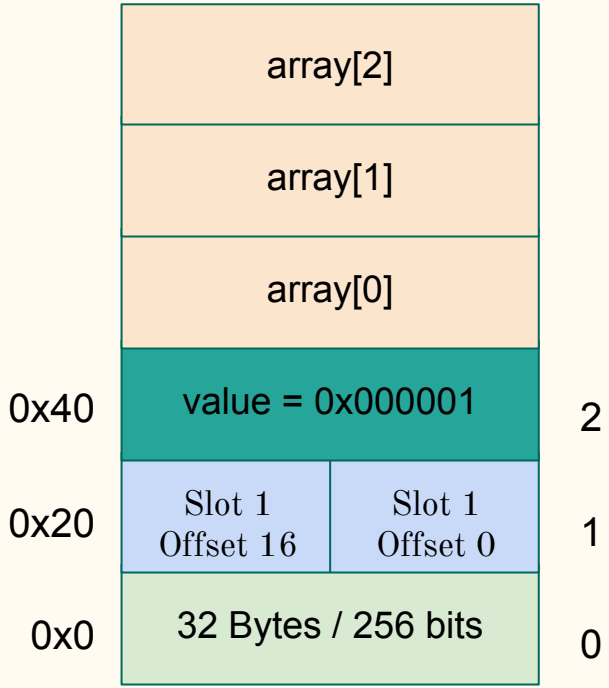
What do these loops compute? Is it more efficient written in Yul? Why?

Yul - Storage

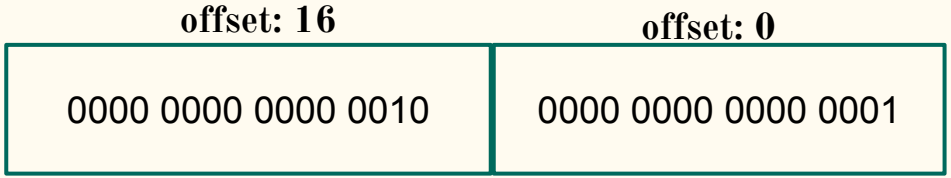
Think of storage manipulation in terms of **slots** rather than addresses. The first declared global variable goes into slot 0 and the next declared follows on.

Recap on storage mechanics:
Fixed arrays - continuous after pointer
Dynamic arrays - pointer location filled with length. Data storage is continuous at keccak256(pointer, length)
Mappings - pointer location empty. Data stored at keccak256(pointer, key)

Instruction	Explanation
sload(p)	Loads the variable in slot p from storage.
sstore(p,v)	Assigns storage slot p value v.
v.slot	Returns the storage slot of variable v.
v.offset	Returns the index in bytes of where variable v begins in a storage slot.



Yul - Packed Storage



Get left block (2) => shr(offset, slot) => 0000 0000 0000 0000 0000 0000 0000 00010

Get right block (1) => use a mask => 0000 0000 0000 0000 1111 1111 1111 1111
and(mask, sload(slot, offset)) => 0000 0000 0000 0010 0000 0000 0000 0001

Instruction	Explanation
and(x, y)	bitwise "and" of x and y
or(x, y)	bitwise "or" of x and y
xor(x, y)	bitwise "xor" of x and y
shl(x, y)	a logical shift left of y by x bits
shr(x, y)	a logical shift right of y by x bits

Loading into packed storage gets a bit more complicated. You need to use different types of masks combined to insert the value correctly.

Masks can also be OR (rare) or XOR (used in binary addition cases)

Yul - Memory

Instruction	Explanation
mload(p)	Similar to `sload()`, but we are saying load the next 32 bytes after p
mstore(p, v)	Similar to `sstore()`, but we are saying store value v in p plus 32 bytes
mstore8(p, v)	Similar to `mstore()`, but only for a single byte
msize()	Returns the largest accessed memory index
pop(x)	Discard value x
return(p, s)	End execution, and return data from memory locations p - v
revert(p, s)	End execution without saving state changes, and return data from memory