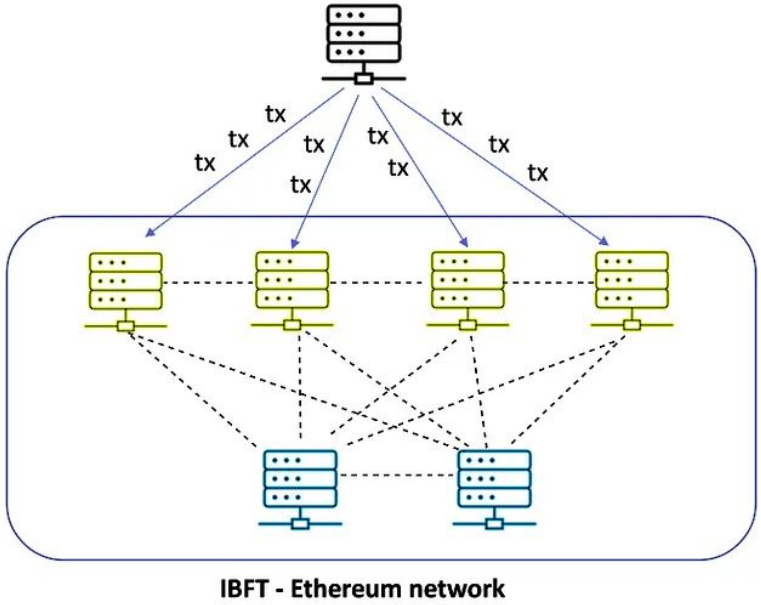
# Lecture 11

Gas Optimization

# Ethereum Refresh - Blockchain Architecture

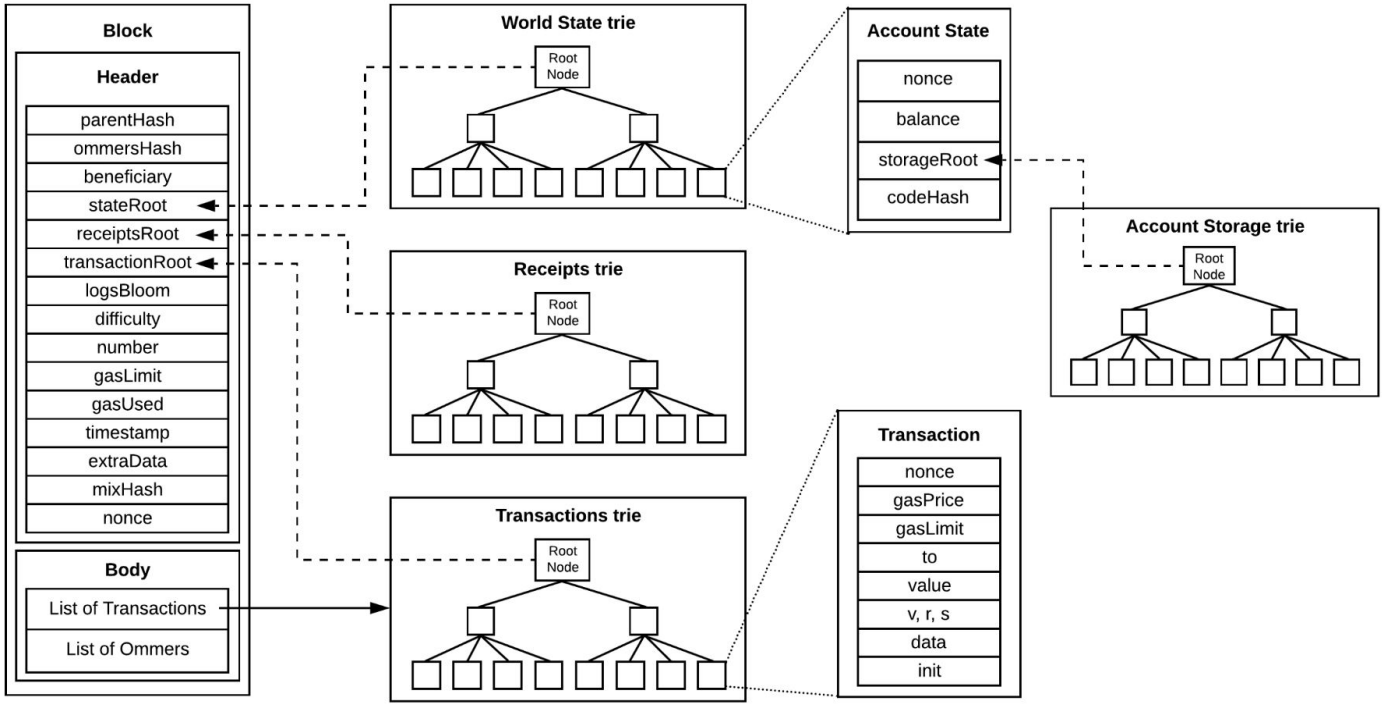



| icon | description |
|---|---|
| tx | Transaction |
| | Testing environment |
| | Regular node |
| | Validator node |
| | RPC connection |
| | P2P connection |

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

Source: Medium

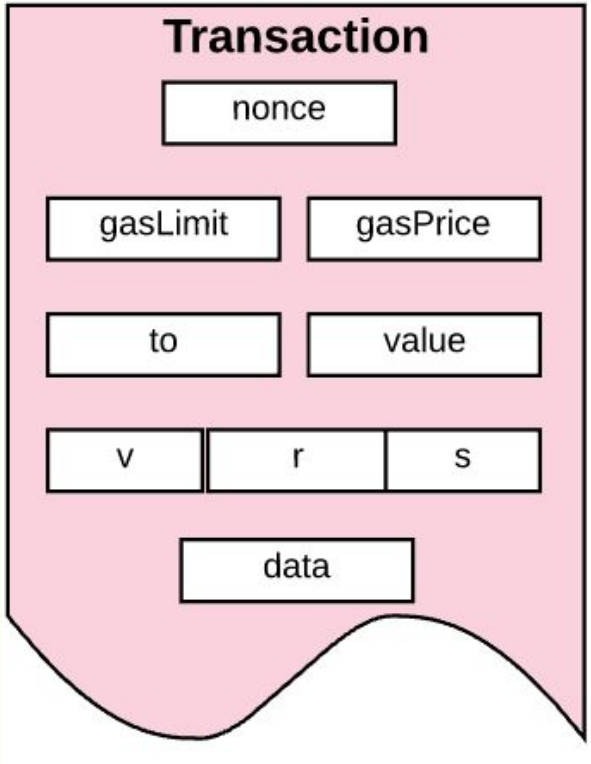
# Data Location - Storage, Code, Log (Lecture 3)




Block, transaction, account state objects and Ethereum tries

# Ethereum Refresh - Transactions

## Type 0 - Legacy

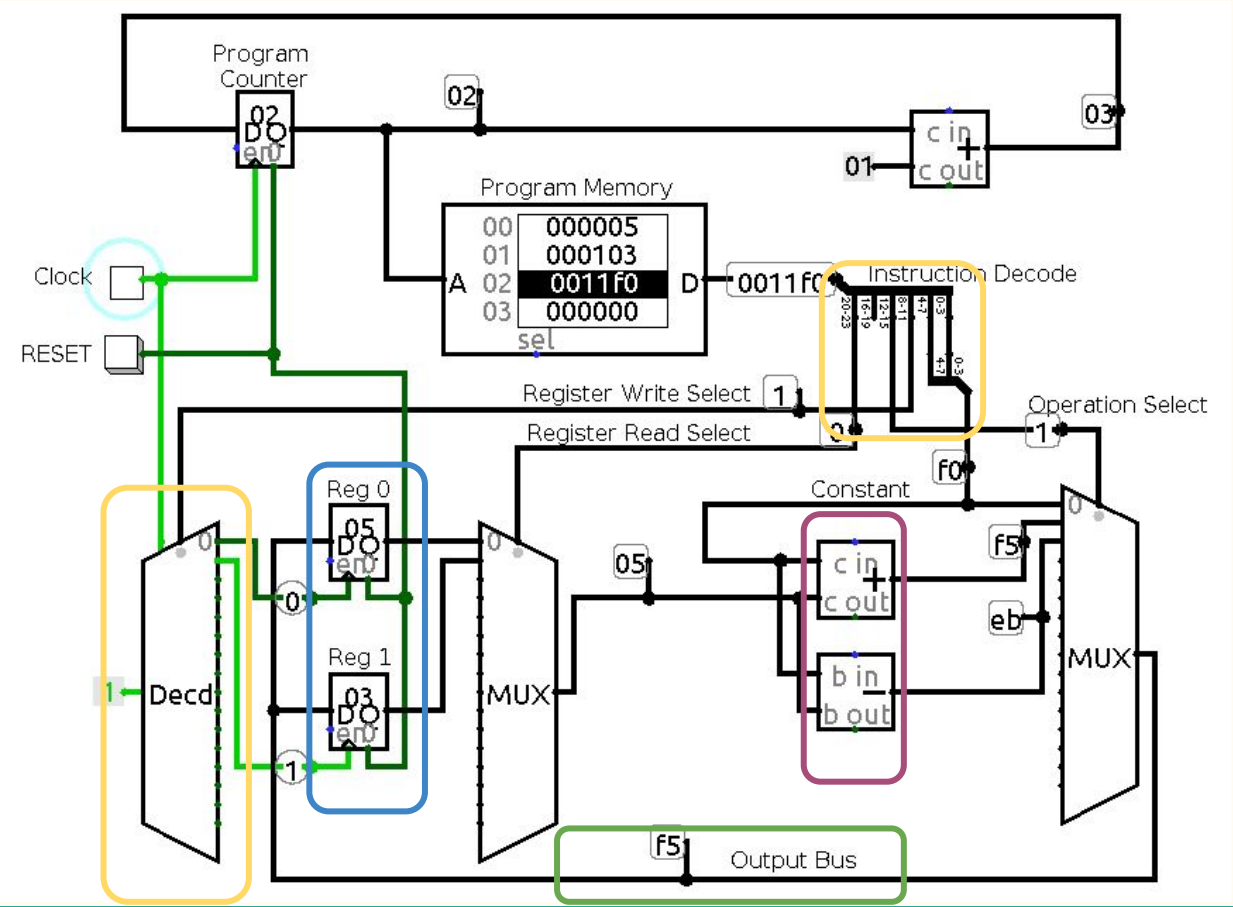



## Type 2 - Base + Priority Fee (Lecture 6)

**Signed Type 2 Transaction**

{
**nonce**: transaction ID,
**maxFeePerGas**: max acceptable base,
**maxPriorityFeePerGas**: max acceptable priority,
**gas**: gasLimit,
**to**: target address (wallet or smart contract),
**value**: eth value (to wallet or payable function),
**data**: bytecode to execute,
**v:** "0x26",
**r**: "0x223a7c9bcf20e",
**s**: "0x28cc7704971491663",
**hash**: keccak(transaction content)
}

# Ethereum Refresh - CPU execution (Lecture 1)



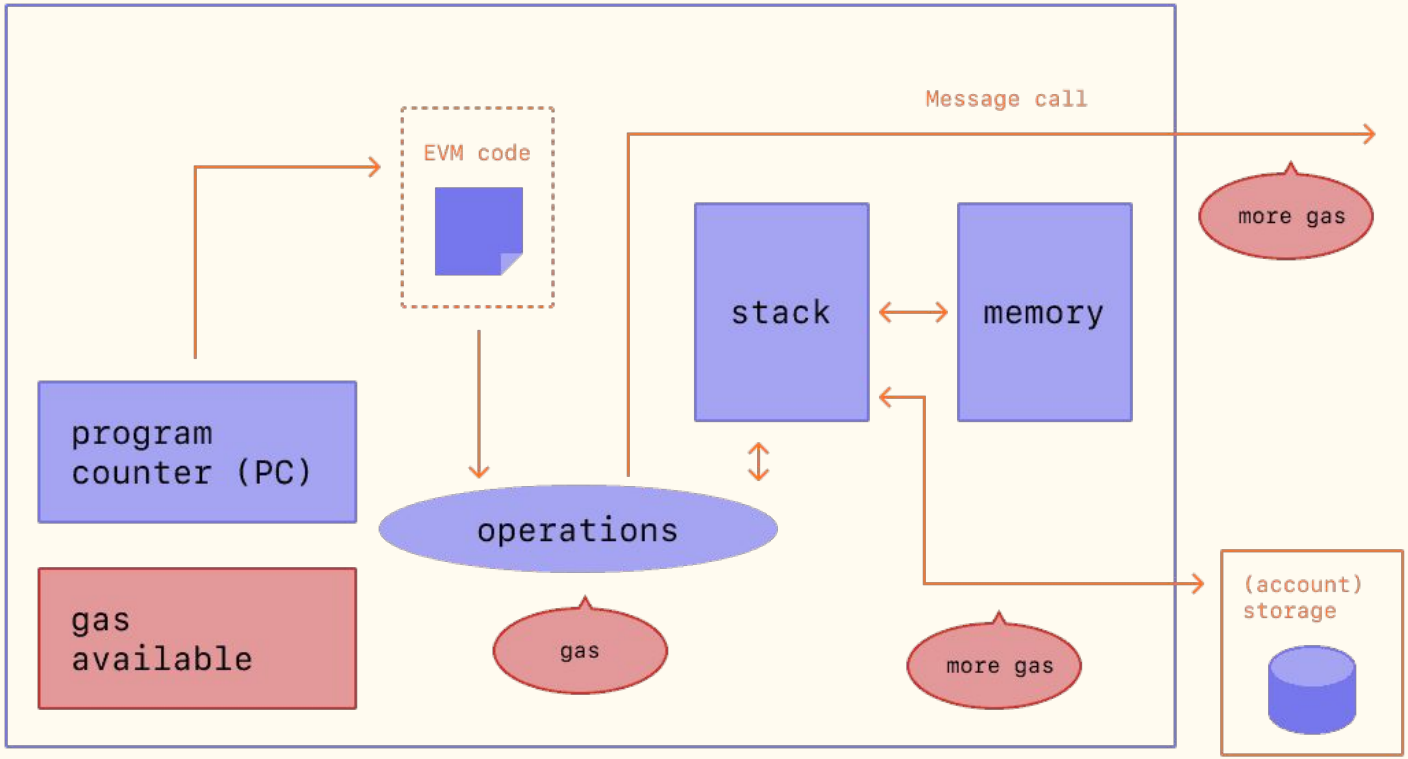

Decode Instructions into opcode and data

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)

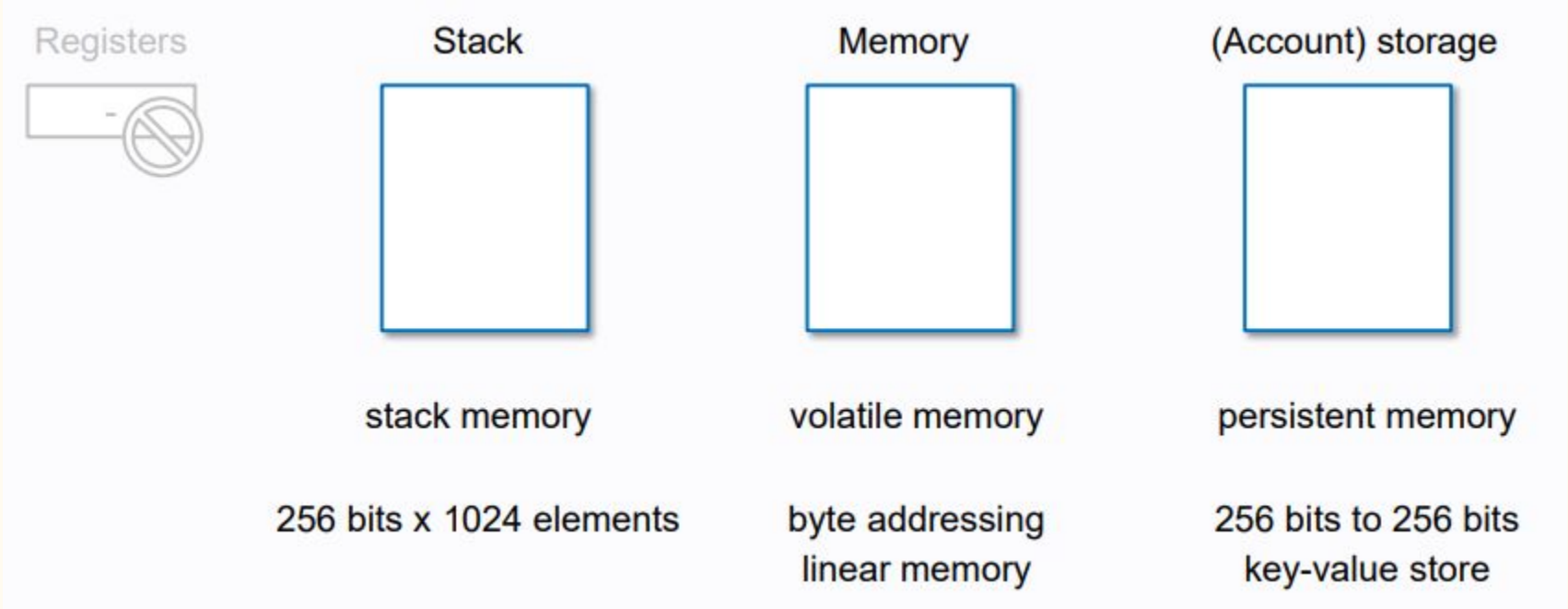


# EVM Memory

# 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

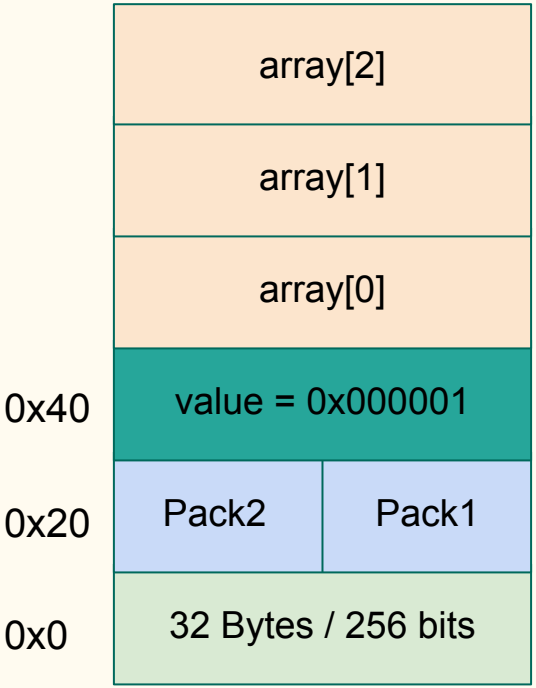
**Constructor:**
Hashed inside bytecode

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

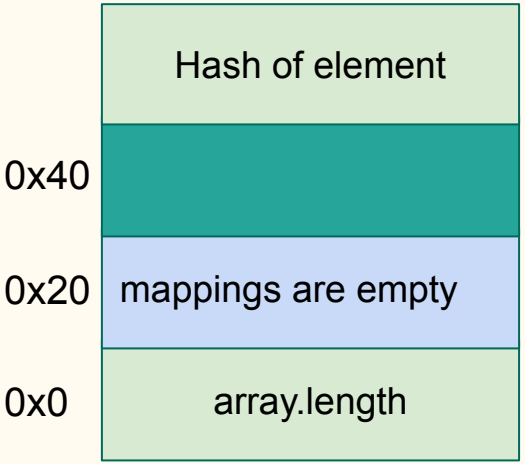
```
1  //SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.17;
3
4  contract christmas{
5      mapping(string => uint) private naughty_list;
6      string[] private names;
7
8      constructor (){
9          names = ["Annie", "Tim", "Mark"];
10         naughty_list["Annie"] = 0;
11         naughty_list["Tim"] = 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

**Value Type**




**Dynamic Type**



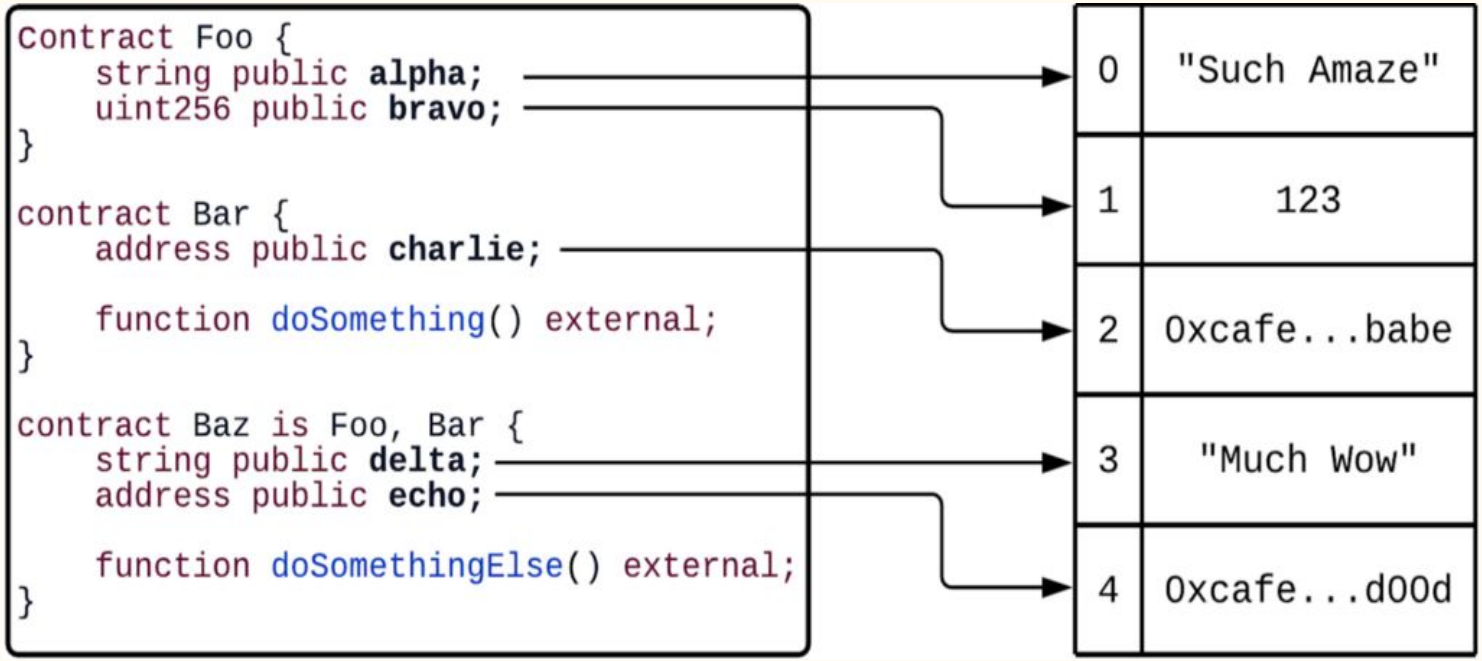

array = [235,12,0]
array location = keccak256(0)
map location = kck(h(key).p)

Full ruleset

**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? C3 linearization - very complex

# Contract Inheritance Linearization




Lecture 3:

DelegateCall maps the target storage onto the calling contract and can overwrite the storage state. This is called a storage collision

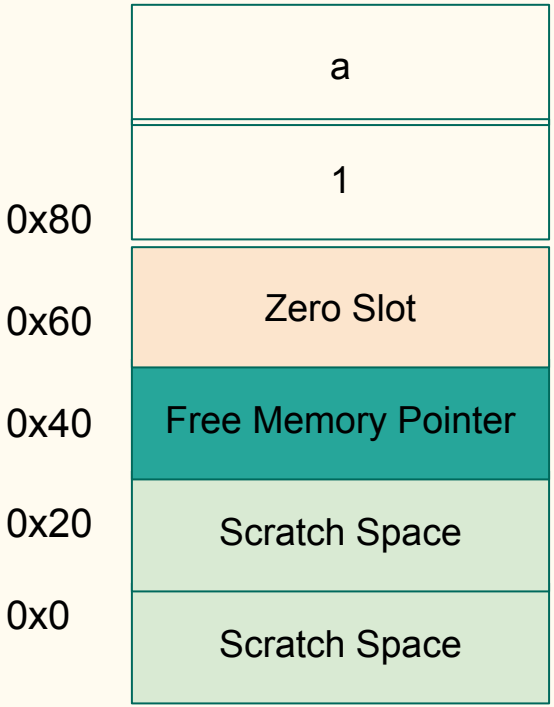
# Storage Example - USDC

```
619   * @dev ERC20 Token backed by fiat reserves
620   */
621  contract FiatTokenV1 is AbstractFiatTokenV1, Ownable, Pausable, Blacklistable {
622      using SafeMath for uint256;
623
624      string public name;
625      string public symbol;
626      uint8 public decimals;
627      string public currency;
628      address public masterMinter;
629      bool internal initialized;
630
631      mapping(address => uint256) internal balances;
632      mapping(address => mapping(address => uint256)) internal allowed;
633      uint256 internal totalSupply_ = 0;
634      mapping(address => bool) internal minters;
635      mapping(address => uint256) internal minterAllowed;
```

# Storage Example - USDC

| Slot | name: type [size] | | |
|---|---|---|---|
| 0 | | | owner: address [20] |
| 1 | | paused: bool [1] | pauser: address [20] |
| 2 | | | blacklister: address [20] |
| 3 | blacklisted: mapping [32] (this slot is left blank) | | |
| 4 | name: string [32] | | |
| 5 | symbol: string [32] | | |
| 6 | | | decimals: uint8 [1] |
| 7 | currency: string [32] | | |
| 8 | | masterMinter: address[20] | initialized: bool [1] |
| 9 | balances: mapping [32] (this slot is left blank) | | |
| 10 | allowed: mapping [32] (this slot is left blank) | | |
| 11 | totalSupply: uint256 [32] | | |
| 12 | minters: mapping [32] (this slot is left blank) | | |
| 13 | minterAllowed: mapping [32] (this slot is left blank) | | |

# Memory Layout




No packing!

a = 0x61000000000000 (high order alignment)

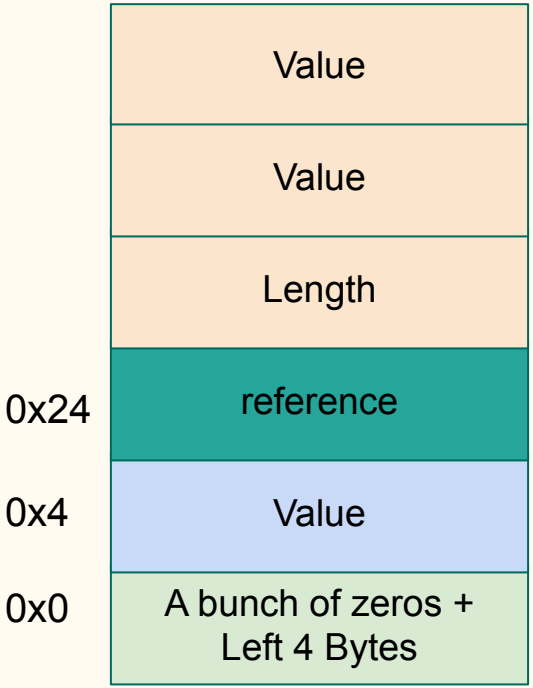
1 = 0x00000000000001 (low order alignment)

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 (Lecture 3)




Dynamic inputs(strings, arrays mappings) follow ABI encoding rules.

Each input must be padded to 32 Bytes

Primitives are directly stored (with padding)

This is the function signature. Matches input to function

# Gas costs and Testing

# Cost of transactions - Gas Calculations

**Gas exists to prevent denial of service attacks.**

gas => counted in units, defined per opcode

gwei => 10^9 gwei = 1 ETH (price)

gasLimit = max amount number of units

Ethereum Gas Tracker

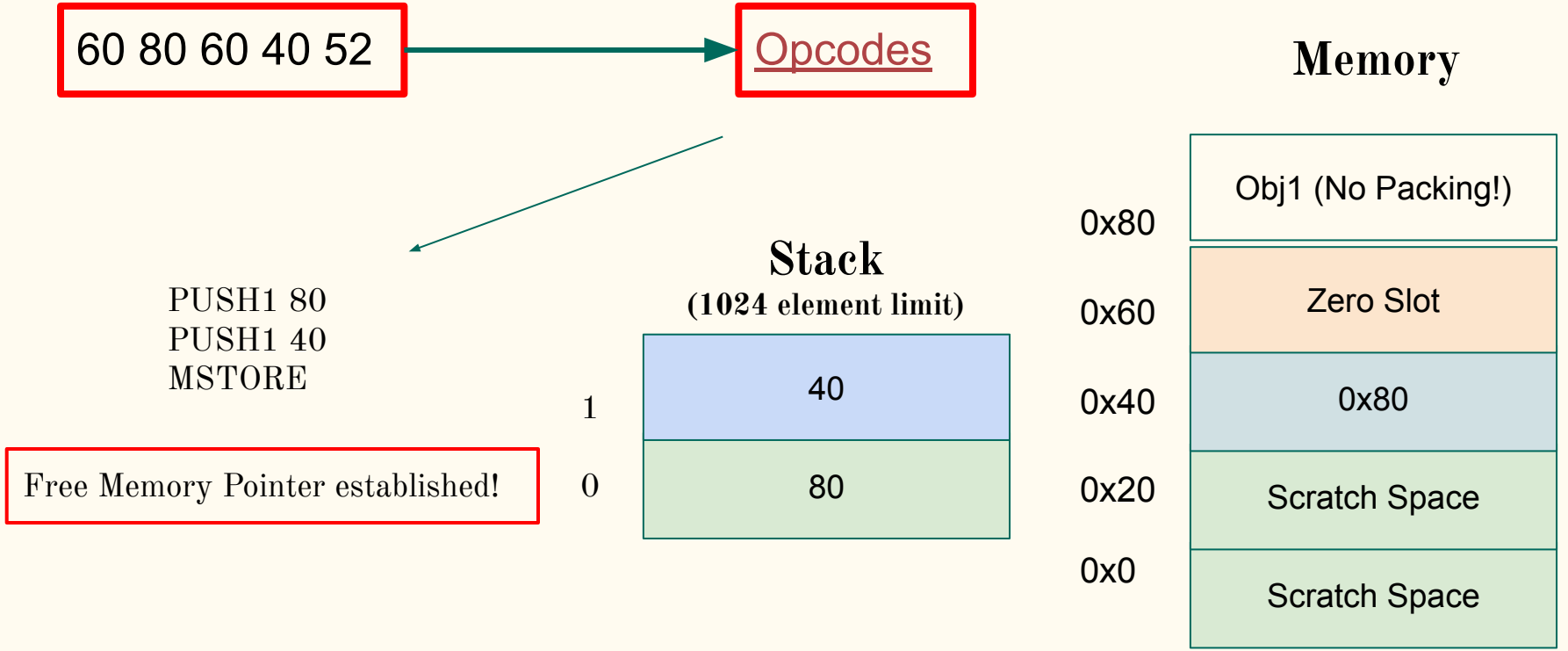
# Is 1 + 1 complex?

In order to execute 1 + 1:

- Set up the memory of the contract (Next slide)
- Copy the calldata of the user input into the stack
- Execute the addition
- Store the value in memory
- Start the return procedure to provide user with result

```
Ran 2 tests for test/Assembly.t.sol:AssemblyTest
[PASS] test_assembly() (gas: 8348)
[PASS] test_solidity() (gas: 8496)
```

# Solidity Contract Creation



# 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?

# Why does it matter? - Gas Cost Testing

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

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.54ms (5.51ms CPU time)

| src/Christmas.sol:Christmas Contract |                 |       |        |       |         |
|=======================================|=================|=======|========|=======|=========|
| Deployment Cost                       | Deployment Size |       |        |       |         |
|---------------------------------------|-----------------|-------|--------|-------|---------|
| 503035                                | 2363            |       |        |       |         |
|---------------------------------------|-----------------|-------|--------|-------|---------|
|                                       |                 |       |        |       |         |
|---------------------------------------|-----------------|-------|--------|-------|---------|
| Function Name                         | Min             | Avg   | Median | Max   | # Calls |
|---------------------------------------|-----------------|-------|--------|-------|---------|
| get_score                             | 24418           | 24418 | 24418  | 24418 | 1       |
|---------------------------------------|-----------------|-------|--------|-------|---------|
| increase_naughty_score                | 56081           | 56081 | 56081  | 56081 | 1       |
```

**71061 gwei = 0,00071061 ETH = 1,5 EUR**

For every naughty act of every child, Santa has to pay 1,5 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?

# 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


```
//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 - Variable packing, fixed memory

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;
}
```

Try to declare fixed blocks rather than dynamic memory. When it makes sense.

```
string[3] private names;
```

```
mapping(string => uint8) private naughty_list;
```

| Median |
| --- |
| 830 |
| 56037 |

| Median |
| --- |
| 836 |
| 57071 |

**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!

| Median |
| --- |
| 830 |
| 56037 |

| Median |
| --- |
| 836 |
| 53721 |

Succeeded in making contract cheaper, but now Santa can only track 3 children :(

# 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){
```

| Median |
| --- |
| 830 |
| 56037 |

| Median |
| --- |
| 686 |
| 55872 |

Both functions got cheaper!

# Technique 4 - Work in memory, avoid repetition in loops

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 repeating functions in loops! Your loops should have as much pre-computed variables as possible. The below code computes the name input hash once so that it is not computed every time the for loop repeats

| Median |
| --- |
| 686 |
| 55872 |

| Median |
| --- |
| 686 |
| 55543 |

```
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

| src/Christmas.sol:Christmas Contract | | | |
|---|---|---|---|
| Deployment Cost | Deployment Size | | |
| 415067 | 1956 | | |
| | | | |
| Function Name | Min | Avg | Median |
| get_score | 686 | 686 | 686 |
| increase_naughty_score | 55543 | 55543 | 55543 |

| src/ChristmasCheap.sol:ChristmasCheap Contract | | | |
|---|---|---|---|
| Deployment Cost | Deployment Size | | |
| 127473 | 447 | | |
| | | | |
| Function Name | Min | Avg | Median |
| get_score | 468 | 468 | 468 |
| increase_naughty_score | 43763 | 43763 | 43763 |

```solidity
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract ChristmasCheap{
    mapping(address => uint) private naughty_list;

    constructor (){
        naughty_list[address(1)] = 0;
        naughty_list[address(2)] = 0;
        naughty_list[address(3)] = 0;
    }

    function get_score(address user) external view returns(uint){
        return naughty_list[user];
    }

    function increase_naughty_score(address user) external {
    //Here, we do not need to check if the string name exists because
    // Solidity will automatically add the entry if it does not exist.
    naughty_list[user] += 10;
    }
}
```

Solidity represents a huge mind shift from traditional programming. It is hard to manipulate strings and mappings because Solidity is meant to support **transactions and addresses.**

Try to think like a bank and anonymous accounts are submitting transactions to you.

# 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
[PATH]
```

| src/Christmas.sol:Christmas Contract | | | | | |
|---|---|---|---|---|---|
| Deployment Cost | Deployment Size | | | | |
| 415067 | 1956 | | | | |
| | | | | | |
| Function Name | Min | Avg | Median | Max | # Calls |
| get_score | 686 | 686 | 686 | 686 | 1 |
| increase_naughty_score | 55543 | 55543 | 55543 | 55543 | 1 |

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.

Still cost the same after 10000 runs - Santa has a very scalable contract!

# 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.

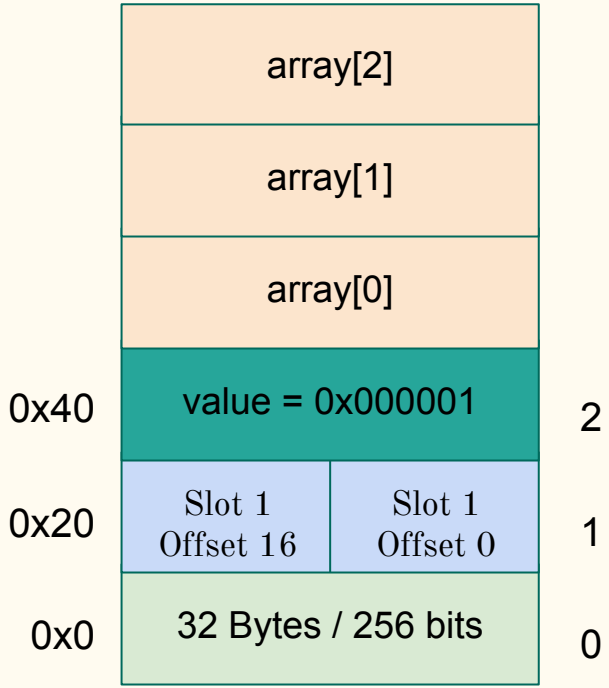
What do these loops compute? Is it more efficient written in Yul? Why?

# Yul - Storage

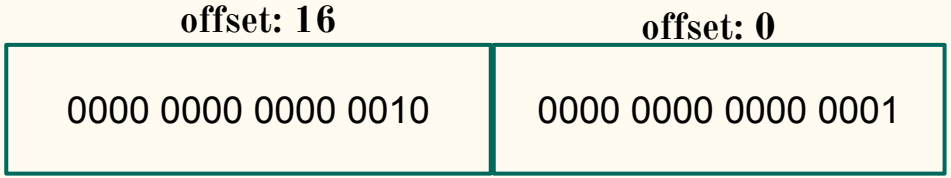
Think of storage manipulation in terms of **slots** rather than addresses. The first declared global variable goes into slot 0 and the next declared follows on.

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




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

Read 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 |

Writing 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 |