Lesson 4

Pre-compiled contracts

The EVM is not very efficient...

```
// SHA256 implemented as a native contract.

type sha256hash struct{}

// RequiredGas returns the gas required to execute the pre-compiled
contract.

//

// This method does not require any overflow checking as the input size gas
costs

// required for anything significant is so high it's impossible to pay for.

func (c *sha256hash) RequiredGas(input []byte) uint64 {

return uint64(len(input)+31)/32*params.Sha256PerWordGas +
params.Sha256BaseGas

}

func (c *sha256hash) Run(input []byte) ([]byte, error) {

h := sha256.Sum256(input)

return h[:], nil

}
```

Solidity Types

User Defined Types :https://docs.soliditylang.org/en/latest/types.html#user-defined-value-types

A user defined value type is defined using `type C is V`, where `C` is the name of the newly introduced type and `V` has to be a built-in value type (the “underlying type”).
The function `C.wrap` is used to convert from the underlying type to the custom type.
Similarly, the function `C.unwrap` is used to convert from the custom type to the underlying type.

Example from the docs

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// Represent a 18 decimal, 256 bit wide fixed point type using a user
defined value type.
type UFixed256x18 is uint256;

/// A minimal library to do fixed point operations on UFixed256x18.
library FixedMath {
uint constant multiplier = 10**18;

/// Adds two UFixed256x18 numbers. Reverts on overflow, relying on checked
/// arithmetic on uint256.
function add(UFixed256x18 a, UFixed256x18 b) internal pure returns
(UFixed256x18) {
return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
}
/// Multiplies UFixed256x18 and uint256. Reverts on overflow, relying on
checked
/// arithmetic on uint256.
function mul(UFixed256x18 a, uint256 b) internal pure returns
(UFixed256x18) {
return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
}
/// Take the floor of a UFixed256x18 number.
/// @return the largest integer that does not exceed `a`.
function floor(UFixed256x18 a) internal pure returns (uint256) {
return UFixed256x18.unwrap(a) / multiplier;
}
/// Turns a uint256 into a UFixed256x18 of the same value.
/// Reverts if the integer is too large.
function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
return UFixed256x18.wrap(a * multiplier);
}
}
```

Notice how `UFixed256x18.wrap` and `FixedMath.toUFixed256x18` have the same signature but perform two very different operations: The `UFixed256x18.wrap` function returns a `UFixed256x18` that has the same data representation as the input, whereas `toUFixed256x18` returns a `UFixed256x18` that has the same numerical value.

Function Types

Function types : https://docs.soliditylang.org/en/latest/types.html#function-types

```
contract Oracle {
 struct Request {
 bytes data;
 function(uint) external callback;
 }

 Request[] private requests;
 event NewRequest(uint);

 function query(bytes memory data, function(uint) external callback) public
{
 requests.push(Request(data, callback));
 emit NewRequest(requests.length - 1);
 }

 function reply(uint requestID, uint response) public {
 // Here goes the check that the reply comes from a trusted source
 requests[requestID].callback(response);
 }
}
```

Function Selectors

How does the EVM call the correct function in a contract ?

Function Selector

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

The compiler creates something like

```
method_id = first 4 bytes of msg.data
if method_id == 0x25d8dcf2 jump to 0x11
if method_id == 0xaabbccdd jump to 0x22
if method_id == 0xffaaccee jump to 0x33
other code
0x11:
code for function with method id 0x25d8dcf2
0x22:
code for another function
0x33:
code for another function
```

Encoding the function signatures and parameters

Example

```

  pragma solidity ^0.8.0;

  contract MyContract {
    Foo otherContract;


    function callOtherContract() public view returns (bool){
        bool answer = otherContract.baz(69,true);
        return answer;
    }
  }


  contract Foo {
      function bar(bytes3[2] memory) public pure {}
      function baz(uint32 x, bool y) public pure returns (bool r) {
      r = x > 32 || y;
      }
      function sam(bytes memory, bool, uint[] memory) public pure {}
  }
```

The way the call is actually made involves encoding the function selector and parameters

If we wanted to call *baz* with the parameters 69 and true , we would pass 68 bytes total, which can be broken down into:

1. the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature baz(uint32,bool).

   ```
   ***0xcdcd77c0:***
   ```

2. the first parameter, a uint32 value 69 padded to 32 bytes

   0x0000000000000000000000000000000000000000000000000000000000000045

3. the second parameter - boolean true, padded to 32 bytes

   0x0000000000000000000000000000000000000000000000000000000000000001

In total

0xcdcd77c000000000000000000000000000000000000000
000000000000000000000045000000000000000000000000000000
00000000000000000000000000000000000001

This is what you see in block explorers if you look at the inputs to functions

There are helper methods to put this together for you

```
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);
```

Alternatively you can then call functions in external contracts on a low level way via

```
bytes memory payload =
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);

(bool success, bytes memory returnData) =
address(contractAddress).call(payload);

require(success);
```

DATA LOCATION AND ASSIGNMENT BEHAVIOUR

Data locations are not only relevant for persistency of data, but also for the semantics of assignments:

- Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a local storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

restrictions on mappings

Mappings can only have a data location of `storage` and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.

Assembly

```
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

```
PUSH 2
PUSH 4
ADD
```

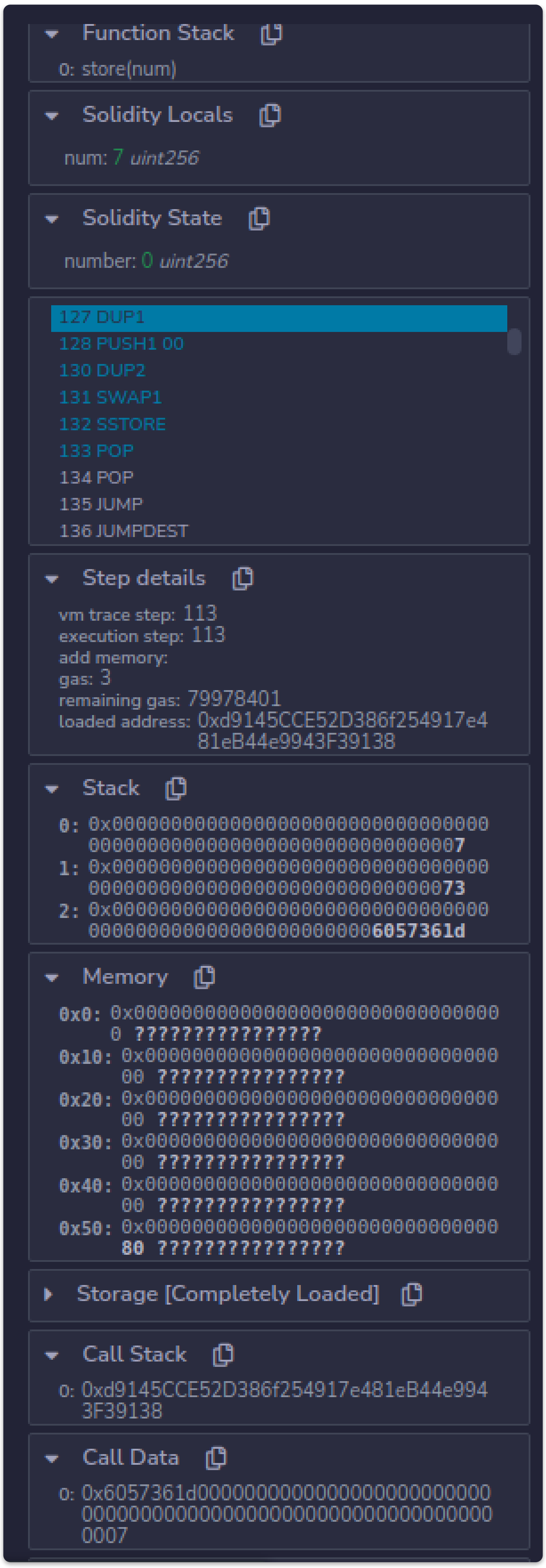
```
PUSH 2
|__2__|

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
|_____|
```

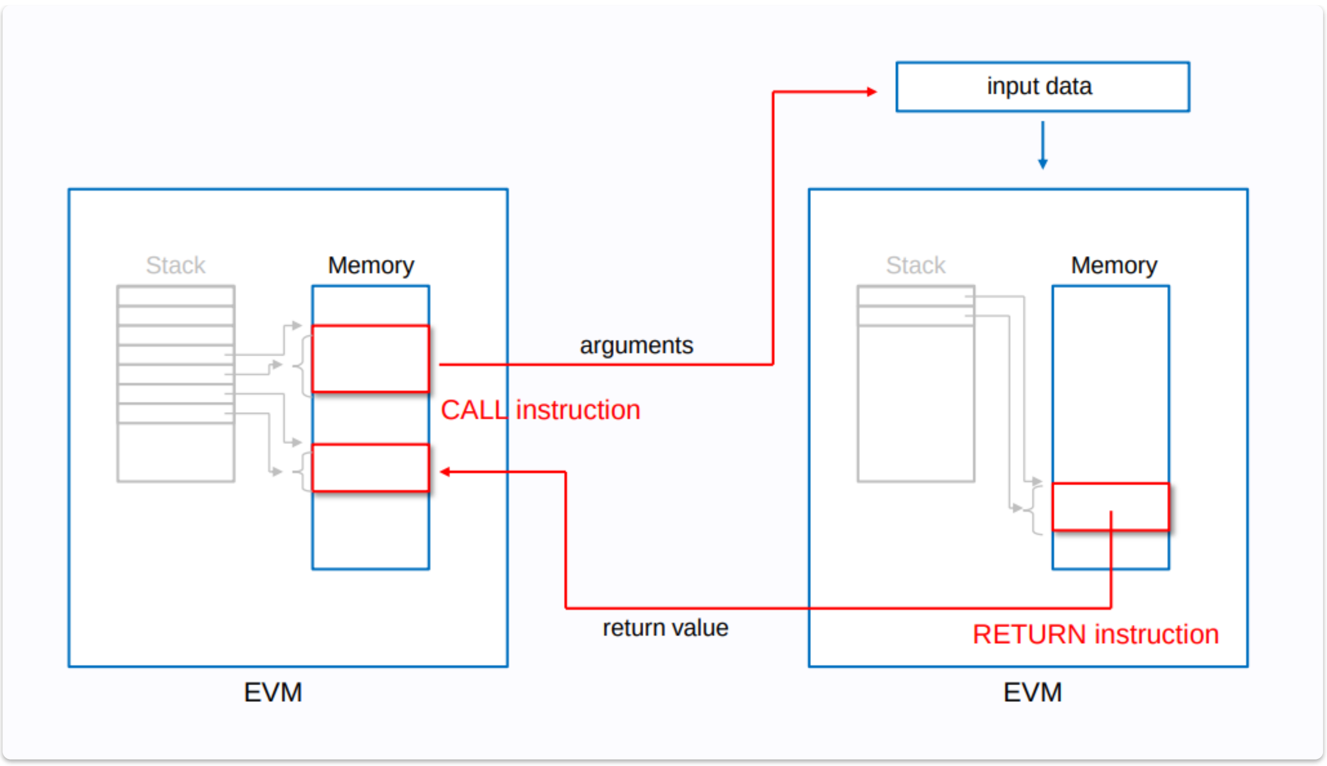
LOOKING AT A FUNCTION IN THE REMIX DEBUGGER

```
    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: 0x00000000000000000000000000000000
   00000000000000000000000000000007
1: 0x00000000000000000000000000000000
   00000000000000000000000000000073
2: 0x00000000000000000000000000000000
   000000000000000000000000006057361d

Memory
0x0: 0x0000000000000000000000000000000
     0 ????????????????
0x10: 0x000000000000000000000000000000
      00 ????????????????
0x20: 0x000000000000000000000000000000
      00 ????????????????
0x30: 0x000000000000000000000000000000
      00 ????????????????
0x40: 0x000000000000000000000000000000
      00 ????????????????
0x50: 0x000000000000000000000000000000
      80 ????????????????

Storage [Completely Loaded]

Call Stack
0: 0xd9145CCE52D386f254917e481eB44e994
   3F39138

Call Data
0: 0x6057361d0000000000000000000000000
   00000000000000000000000000000000000
   0007
```

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)
}
```

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
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

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.

Usful Tools

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