This week

- Yul continued
- Best Practices
- Formal Methods and verification
- Useful Libraries

Class Question

1. How can you bribe the Solidity team to introduce an op code for you ?
2. According to this event, FTX Exchange transfered some tokens
   What do you think ?

Bit operations

TWOS COMPLEMENT

The two's complement of an N-bit number is defined as its complement with respect to 2^N the sum of a number and its two's complement is 2^N.
For instance, for the three-bit number 011_2, the two's complement is 101_2, because $011_2 + 101_2 = 1000_2 = 8_{10}$ which is equal to 2^3.

The two's complement is calculated by inverting the bits and adding one.

Bit operations are performed on the two's complement representation of the number. This means that, for example
`~int256(0) == int256(-1)`.

Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by a signed type will produce a compilation error.

Shifts can be "simulated" using multiplication by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

- `x << y` is equivalent to the mathematical expression `x * 2**y`.
- `x >> y` is equivalent to the mathematical expression `x / 2**y`, rounded towards negative infinity.

See this gist for examples

Recap of Yul

Good Introduction

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`

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)
}
```

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).

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.

memoryguard keyword

This function is available in the EVM dialect with objects. The caller of
`let ptr := memoryguard(size)`
(where `size` has to be a literal number) promises that they only use memory in either the range `[0, size)`
or the unbounded range starting at `ptr`.

Since the presence of a `memoryguard` call indicates that all memory access adheres to this restriction, it allows the optimizer to perform additional optimization steps

verbatim keyword

The set of `verbatim...` builtin functions lets you create bytecode for opcodes that are not known to the Yul compiler. It also allows you to create bytecode sequences that will not be modified by the optimizer.

The functions are `verbatim_<n>i_<m>o("<data>", ...)`, where

- `n` is a decimal between 0 and 99 that specifies the number of input stack slots / variables
- `m` is a decimal between 0 and 99 that specifies the number of output stack slots / variables
- `data` is a string literal that contains the sequence of bytes

If you for example want to define a function that multiplies the input by two, without the optimizer touching the constant two, you can use

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

Stand Alone Yul

As well as being included in Solidity files (using the assembly keyword), we can write stand alone Yul
This uses the Yul object specification
For Example

```
// A contract consists of a single object with sub-objects representing
// the code to be deployed or other contracts it can create.
// The single "code" node is the executable code of the object.
// Every (other) named object or data section is serialized and
// made accessible to the special built-in functions datacopy / dataoffset
/ datasize
// The current object, sub-objects and data items inside the current object
// are in scope.
object "Contract1" {
    // This is the constructor code of the contract.
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // first create "Contract2"
```

```
            let size := datasize("Contract2")
            let offset := allocate(size)
            // This will turn into codecopy for EVM
            datacopy(offset, dataoffset("Contract2"), size)
            // constructor parameter is a single number 0x1234
            mstore(add(offset, size), 0x1234)
            pop(create(offset, add(size, 32), 0))

            // now return the runtime object (the currently
            // executing code is the constructor code)
            size := datasize("runtime")
            offset := allocate(size)
            // This will turn into a memory->memory copy for Ewasm and
            // a codecopy for EVM
            datacopy(offset, dataoffset("runtime"), size)
            return(offset, size)
        }

        data "Table2" hex"4123"

        object "runtime" {
            code {
                function allocate(size) -> ptr {
                    ptr := mload(0x40)
                    if iszero(ptr) { ptr := 0x60 }
                    mstore(0x40, add(ptr, size))
                }

                // runtime code

                mstore(0, "Hello, World!")
                return(0, 0x20)
            }
        }

        // Embedded object. Use case is that the outside is a factory contract,
        // and Contract2 is the code to be created by the factory
        object "Contract2" {
            code {
                // code here ...
            }

            object "runtime" {
                code {
                    // code here ...
                }
            }

            data "Table1" hex"4123"
        }
    }
```

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)

Yul+ online compiler : https://yulp.fuel.sh/
There is a plugin for Yul+ in Remix also

EVM Playground

EVM Playground

- Can be used for assembly / Yul
- Also has details of pre compiled contracts

Solmate - Building Blocks

Example Yul+ ERC20 contract

```
object "ERC20" {
  code {
    // Experimental, in-work - some functions may not work

    enum Storage (balance, allowance) // storage index numbers

    // constructor(address owner, uint256 totalSupply)
    codecopy(64, sub(codesize(), 64), 64)

    // stipulate initial owner and total supply
    let owner := mload(64)
    let totalSupply := mload(96)

    // set initial owner balance at totalSupply
    mstore(0, owner, Storage.balance)
    sstore(keccak256(0, 64), totalSupply)

    // Goto runtime
    datacopy(0, dataoffset("Runtime"), datasize("Runtime"))
    return(0, datasize("Runtime"))
  }
  object "Runtime" {
    code {
        const _calldata := 128 // leave first 4 32 byte chunks for hashing,
returns etc..

        enum Storage (balance, allowance) // storage index numbers

        calldatacopy(_calldata, 0, calldatasize()) // copy all calldata to
memory

        switch mslice(_calldata, 4) // 4 byte calldata signature

        case sig"transfer(address owner, uint256 amount) returns (bool
success)" {
            mstruct transferCalldata(sig: 4, owner: 32, amount: 32)

            transferFrom(caller(),
                transferCalldata.owner(_calldata),
                transferCalldata.amount(_calldata))
        }

        case sig"transferFrom(address source, address destination, uint
amount) returns (bool success)" {
            mstruct transferFromCalldata(sig: 4, source: 32, destination:
32, amount: 32)

            transferFrom(transferFromCalldata.source(_calldata),
                transferFromCalldata.destination(_calldata),
                transferFromCalldata.amount(_calldata))
        }

        case sig"approve(address destination, uint256 amount) returns (bool
success)" {
            mstruct approveCalldata(sig: 4, destination: 32, amount: 32)

            sstore(mappingStorageKey2(caller(),
                approveCalldata.destination(_calldata),
                Storage.allowance), approveCalldata.amount(_calldata))

            mstore(0, approveCalldata.amount(_calldata))
            log3(0, 32,
                topic"event Approval(address indexed source, address
indexed destination, uint256 amount)",
                caller(),
                approveCalldata.destination(_calldata))

            mstore(0, true)
            return(0, 32)
        }

        case sig"balanceOf(address owner) view returns (uint256 balance)" {
            mstruct balanceOfCalldata(sig: 4, owner: 32)
            mstore(0,
sload(mappingStorageKey(balanceOfCalldata.owner(_calldata),
                Storage.balance)))
            return (0, 32)
        }

        case sig"allowance(address source, address owner) view returns
(uint256 allowance)" {
            mstruct allowanceCalldata(sig: 4, source: 32, owner: 32)

            mstore(0,
sload(mappingStorageKey2(allowanceCalldata.source(_calldata),
                allowanceCalldata.owner(_calldata),
                Storage.allowance)))
            return (0, 32)
        }

        case sig"name() view returns (string)" {
            // mstore(0, "Fake Dai Stablecoin") somethig like this, proper
but w/ encoding.
            // return(0, 32)
        }
        case sig"symbol() view returns (string)" {
            // mstore(0, "FDAI")
            // return(0, 32)
        }
        case sig"version() view returns (string)" {
            // mstore(0, "1")
            // return(0, 32)
        }
        case sig"decimals() view returns (string)" {
            // mstore(0, 18)
            // return(0, 32)
        }

        default { require(0) } // invalid method signature

        stop() // stop execution here..

        function transferFrom(source, destination, amount) {
            let balanceOfSource := sload(mappingStorageKey(source,
Storage.balance))
            let allowanceOfDestination := sload(mappingStorageKey2(source,
destination, Storage.balance))
            let allowanceOfSourceSender := sload(mappingStorageKey2(source,
caller(), Storage.allowance))

            // require(balanceOf[src] >= wad, "Dai/insufficient-balance");
            require(or(gt(balanceOfSource, amount), eq(balanceOfSource,
amount)))

            // if (src != msg.sender && allowance[src][msg.sender] !=
uint(-1)) {
            if and(neq(source, caller()), neq(allowanceOfSourceSender,
MAX_UINT)) {
                // require(allowance[src][msg.sender] >= wad,
"Dai/insufficient-allowance");
                require(gte(allowanceOfDestination, amount))

                // allowance[src][msg.sender] = sub(allowance[src]
[msg.sender], wad);
                sstore(mappingStorageKey2(source, destination,
Storage.balance),
                    sub(allowanceOfDestination, amount))
            }

            //  balanceOf[src] = sub(balanceOf[src], wad);
            sstore(mappingStorageKey(source, Storage.balance),
                sub(balanceOfSource, amount))

            // balanceOf[dst] = add(balanceOf[dst], wad);
            let balanceOfDestination :=
sload(mappingStorageKey(destination, Storage.balance))
            sstore(mappingStorageKey(destination, Storage.balance),
                add(balanceOfDestination, amount))

            mstore(0, amount)
            log3(0, 32, topic"event Transfer(address indexed source,
address indexed destination, uint amount)",
                source, destination)

            mstore(0, true)
            return(0, 32)
        }

        // Solidity Style Storage Key: mapping(bytes32 => bytes32)
        function mappingStorageKey(key, storageIndex) -> storageKey {
            mstore(0, key, storageIndex)
            storageKey := keccak256(0, 64)
        }

        // Solidity Style Storage Key: mapping(bytes32 => mapping(bytes32
=> bytes32)
        function mappingStorageKey2(key, key2, storageIndex) -> storageKey
{
            mstore(0, key, storageIndex, key2)
            mstore(96, keccak256(0, 64))
            storageKey := keccak256(64, 64)
        }
    }
  }
}
```

Resources

EVM Guide
EVM Chapter from Mastering Ethereum Book

Disassemblers

- *Porosity* is a popular open source decompiler.
- *Ethersplay* is an EVM plug-in for Binary Ninja, a disassembler.
- *IDA-Evm* is an EVM plugin for IDA, another disassembler.

Answer to class question
https://gist.github.com/hrkrshnn/84d2a9686a88ea355749dec2a5987ee9

Bad Advice about Assembly
