Demystifying Ethereum Assembly

A practical zero-to-one guide.

jtriley.eth
EVM Smart Contract Engineer

Section 1

Understanding The Ethereum Virtual Machine

Links for this Workshop

https://linktr.ee/evmassembly

Instruction Set

The EVM is a stack-based virtual machine with a relatively small instruction set. The instructions can be categorized by one of the following.

- Stack Instructions
- Arithmetic Instructions
- Comparison Instructions
- Bitwise Instructions
- Memory Instructions
- Context Instructions
  - Read
  - Write

Stack Instructions

Stack instructions involve manipulating the position of values on the stack.

- pushN value: pushes a value to the top of the stack where N is the byte size of the value.
- pop: pops a value from the top of the stack.
- swapN: swaps the value from the top of the stack with a value at stack index N.
- dupN: duplicates a value from the stack at index N and pushes it to the stack.

Stack Example

```
push1 0x01  // [one]
push1 0x02  // [two, one]

swap1       // [one, two]

dup1        // [one, one, two]

pop         // [one, two]
pop         // [two]
pop         // []
```

Arithmetic Instructions

Arithmetic instructions pop two or more values from the stack, performs an arithmetic operation, and pushes the result.

- add pushes the result of addition of two values.
- sub pushes the result of subtraction of two values.
- mul / smul pushes the result of multiplication of two values.
- div / sdiv pushes the result of the division of two values.
- mod pushes the result of the modulus of two values.
- exp pushes the result of exponentiation of two values.
- addmod / mulmod combines add with mod and mul with mod.

*smul and sdiv treat the values as “signed” integers.

Arithmetic Example

```
push1 0x01  // [one]
push1 0x02  // [two, one]

add         // [three]

push 0x02   // [two, three]
dup2        // [three, two, three]

mul         // [six, three]

div         // [two]

pop         // []
```

Comparison Instructions

Comparison pop one or two values from the stack, performs a comparison and based on the result, pushes either true (0) or false (1).

- lt / slt pushes true if the top stack value is less than the second.
- gt / sgt pushes true if the top stack value is greater than the second.
- eq pushes true if the top two stack values are equal.
- iszero pushes true if the top stack value is zero.

Comparison Example

```
push1 0x01  // [one]
push1 0x02  // [two, one]

// NOTE: false == zero and true == one

eq          // [false]

// NOTE: iszero can be used to invert a boolean (!bool)

iszero      // [true]

pop         // []
```

Bitwise Instructions

Bitwise instructions pop one or more values from the stack and performs bitwise operations on them.

- and performs bitwise AND on the top two stack values.
- or performs bitwise OR on the top two stack values.
- xor performs bitwise Exclusive OR on the top two stack values.
- not performs bitwise NOT on the top stack value.
- shr / shl performs a bit-shift right and left, respectively.

Bitwise Example

```
// one        | 0b 0001
// two        | 0b 0010
// three      | 0b 0011
// four       | 0b 0100
// five       | 0b 0101
// six        | 0b 0110
// seven      | 0b 0111
// eight      | 0b 1000
// nine       | 0b 1001
// ten        | 0b 1010
```

```
// using 4 bit binary format in comments here for clarity.
// one == 0b0001, two == 0b0010, three == 0b0011, etc.

push1 0x01  // [0b0001]
push1 0x02  // [0b0010, 0b0001]

// shift 0b0001 left by two bits.

shl         // [0b0100]

push1 0x02  // [0b0010, 0b0100]

// shift 0b0100 right by two bits.

shr         // [0b0001]

// flip the bits

not         // [0b1110]
```

Memory Instructions

Memory instructions read and write to a chunk of memory. Memory expands linearly and can be read / written to arbitrarily.

- mstore stores a 32 byte (256 bit) word in memory.
- mstore8 stores a one byte (8 bit) word in memory.
- mload loads a 32 byte word from memory.

Memory Example

```
push1 0x01  // [one]
push1 0x00  // [zero, one]

// store one in memory starting at slot zero

mstore      // []

push1 0x00  // [zero]

// load from memory starting at slot one

mload       // [one]

// MEMORY:
// | -----| -------------------------------------------------------------------|
// | slot | value                                                              |
// | ---- | ------------------------------------------------------------------ |
// | 0x00 | 0x0000000000000000000000000000000000000000000000000000000000000001 |
// | ...  | ...                                                                |
```

Context Instructions (Read)

The following is a non-comprehensive, short list of instructions that can read from the global state and execution context.

- caller pushes the address that called the current context.
- timestamp pushes the current block's timestamp.
- staticcall can make a read-only call to another contract.
- calldataload can load a chunk of the calldata in the current context.
- sload can read a piece of data from persistent storage on the current contract.

Context (Read) Example

```
// assume owner's address is stored at slot 0x00

caller      // [caller_address]

push1 0x00  // [zero, caller_address]
sload       // [owner_address, caller_address]

eq          // [is_caller_owner]
```

Context Instructions (Write)

The following is a non-comprehensive, short list of instructions that can write to the global state and the execution context.

- sstore can store data to persistent storage.
- logN can append data to the current transaction logs where N is the number of special, indexed values in the log.
- call can make a call to external code, which can also update the global state.
- create / create2 can deploy code to a new address, creating a new contract.

Context (Write) Example

```
// store timestamp at slot zero

timestamp       // [block_timestamp]
push1 0x00      // [zero, block_timestamp]
sstore          // []
```

```
// get the caller's address, store it at
// slot zero (0x00).

caller      // [caller_address]
push1 0x00  // [zero, caller_address]
sstore      // []

// store "true" in memory at an offset
// of zero (0x00).

push1 0x01  // [true]
push1 0x00  // [zero, true]
mstore      // []

// return data to the caller from memory
// starting at an offset of zero (0x00)
// and a size of 32 bytes (0x20).

push1 0x20  // [word_size]
push1 0x00  // [memory_offset, word_size]
return      // []
```

Instruction Set Review

The EVM has a fairly simple instruction set. This section did not cover every instruction, but rather it will serve as a foundation for understanding Yul in the following section.

To the left, there is a simple contract that will store the caller's address in persistent storage, then return "true" to indicate success.

Section 2

Yul Syntax

Yul Overview

Yul is a low level language that may be written in-line in Solidity, as a standalone language, and as a compilation target.

Built into the language are most EVM instructions callable as functions, basic control flow support, and functions.

Notice that the stack is largely abstracted away with the exception of a built-in pop function to drop variables.

Syntax Overview

Notice that object and code keywords are only used in stand-alone Yul files, not in-line Solidity.

Also notice Yul does not support else blocks. To create if { } else { } functionality, a switch statement may be used.

The for loop contains the iterator declaration, break condition, increment logic, then the body.

```
object "MyContract" {
  code {
    let assignmentSytax := funcSyntax(1, 2)


    function funcSyntax(a, b) -> c {
      c := add(a, b)
    }


    if eq(a, 0) {
      revert(0, 0)
    } // no else statement

    switch a
    case 1 {
      // handle a == 0
    }
    case 2 {
      // handle a == 1
    }
    default {
      // handle anything else
    }


    for { let i := 0 } lt(i, a) { i := add(i, 1) } {
      // iterate
    }
  }
}
```

Comparison to Mnemonic Bytecode

```
// get the caller's address, store it at
// slot zero (0x00).

caller      // [caller_address]
push1 0x00  // [zero, caller_address]
sstore      // []

// store "true" in memory at an offset
// of zero (0x00).

push1 0x01  // [true]
push1 0x00  // [zero, true]
mstore      // []

// return data to the caller from memory
// starting at an offset of zero (0x00)
// and a size of 32 bytes (0x20).

push1 0x20  // [word_size]
push1 0x00  // [memory_offset, word_size]
return      // []
```

```
sstore(0, caller())
mstore(0, 1)
return(0, 32)
```

Section 3

Yul in Solidity

Solidity Standards Overview

Solidity has created abstractions for standards that engineers and auditors must be aware of when dealing with in-line Yul.

- Calldata Layout
- Memory Layout
- Storage Layout
- Event Logging
- Errors

Calldata Layout

Per the Application Binary Interface (ABI) standardization, the calldata layout is as follows.

- The selector is the leftmost 4 bytes of a Keccak-256 hash of the function signature (name and argument types).
- Each argument is padded to 32 bytes.
- If an argument is of dynamic size, the 32 byte slot will be a pointer to the dynamic value later in the calldata.

Calldata Visualization

```
// Signature:
// function transfer(address,uint256) view returns (bool)

// Hash:
// 0xa9059cbb2ab09eb219583f4a59a5d0623ade346d962bcd4e46b11da047c9049b

// Selector:
// 0xa9059cbb

// Calldata Layout:
// 0x00 : 0xa9059cbb
// 0x04 : 0x000000000000000000000000c02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
// 0x24 : 0x000000000000000000000000000000000000000000000000000000000000000a
```

Memory Layout

Per the Solidity documentation, the first four slots of memory are reserved.

- 0x00 : scratchspace
- 0x20 : scratchspace
- 0x40 : free memory pointer
- 0x60 : zero slot

Dynamically sized arrays occupy one slot to point to the value in memory, one slot to indicate length, then one slot for each element.

Byte arrays and strings are similar, except their elements are tightly packed and aligned to the left.

Memory Example

```
uint256 constant bytesLen = 4;
bytes32 constant paddedBytes =
    0x1122334400000000000000000000000000000000000000000000000000000000;


function getBytes() pure returns (bytes memory) {
  bytes memory data;

  assembly {

    // load memory pointer
    let freeMemoryPtr := mload(0x40)

    data := freeMemoryPtr

    // store the length at the free memory pointer
    mstore(freeMemoryPtr, bytesLen)

    // increment the free memory pointer by 32 bytes
    freeMemoryPtr := add(freeMemoryPtr, 0x20)

    // store the 4 bytes at the new free memory pointer
    mstore(freeMemoryPtr, paddedBytes)

    // increment the free memory pointer by 32 bytes
    freeMemoryPtr := add(freeMemoryPtr, 0x20)

    // store the updated free memory pointer at the
    // free memory pointer slot
    mstore(0x40, freeMemoryPtr)
  }

  return data;
}
```

Storage (Statically Sized Variables)

Per Solidity documentation:

- Storage layout starts at slot 0.
- The data is stored in the right-most byte(s).
- If the next value can fit into the same slot (determined by type), it is right-aligned in the same slot, else it is stored in the next slot.
- Immutable and constant values are not in storage, therefore they do not increment the storage slot count.

Simple Storage Example

```
contract MyCon {
  uint256 a = 1;
  uint128 b = 2;
  uint128 c = 3;
  uint8 d = 4;
  uint8 e = 5;
}

// Storage Layout:
// 0x00 : 0x0000000000000000000000000000000000000000000000000000000000000001
// 0x01 : 0x0000000000000000000000000000000300000000000000000000000000000002
// 0x02 : 0x0000000000000000000000000000000000000000000000000000000000000504
```

Storage (Dynamically Sized Variables)

Per Solidity documentation:

- A mapping slot is the Keccak-256 hash of the key value concatenated with the storage slot.
- A dynamically sized array stores the current length in its slot, then its elements are stored sequentially starting at the Keccak-256 hash of the slot number.
- Byte arrays and strings are stored the same way as other dynamic arrays unless the length is 31 or less. Then it is packed into one slot and the right-most byte is occupied by two times the length.

Dynamic Storage Example

```
contract MyCon {
  string a = "aaaa";
  uint256[] data = [1, 2];
  mapping(address => uint256) account;
  constructor() {
    account[address(1)] = 3;
  }
}

// Storage Layout:
// 0x00..00 : 0x6161616100000000000000000000000000000000000000000000000000000008
// 0x00..01 : 0x0000000000000000000000000000000000000000000000000000000000000002
// 0x00..02 : 0x0000000000000000000000000000000000000000000000000000000000000000
// ...
// 0xb1..f6 : 0x0000000000000000000000000000000000000000000000000000000000000001
// 0xb1..f7 : 0x0000000000000000000000000000000000000000000000000000000000000002
// ...
// 0xe9..e0 : 0x0000000000000000000000000000000000000000000000000000000000000003
```

Storage (Inheritance)

Solidity uses C3 Linearization. In the context of storage, this means the following.

- Storage slots in a parent contract precedes the the child contract.
- When a child has multiple parents, the order of parent storage is set by the order of inheritance.
- This process is repeated recursively.
- Storage packing rules are in play when applicable.

Inherited Storage Example

```
contract Parent0 {
  uint256 a = 1;
}

contract Parent1 {
  uint256 b = 2;
}

contract Child is Parent0, Parent1 {
  uint256 c = 3;
}

// Storage Layout (Child):
// 0x00 : 0x0000000000000000000000000000000000000000000000000000000000000001
// 0x01 : 0x0000000000000000000000000000000000000000000000000000000000000002
// 0x02 : 0x0000000000000000000000000000000000000000000000000000000000000003
```

Event Logs

Per the ABI standardization, event logs follow the following rules.

- Events have up to four indexed topics.
- The first topic is always the Keccak-256 hash of the event signature.
- Non-indexed topics are logged by storing them in memory and passing to the log instruction a pointer to the start of the data and the length of the data.

Event Log Example

```
bytes32 constant myEventHash =
    0x06208c2a7c81ce0338f4e7c431f49476c450bad6ce52ef2b4e186db4a1fbcaf4;

uint256 constant a = 1;
uint256 constant b = 2;

contract MyLogger {
  event MyEvent(uint256 indexed a, uint256 b);

  function logInSolidity() public {
    emit MyEvent(a, b);
  }

  function logInYul() public {
    assembly {

      // store `b` in memory at slot zero.
      mstore(0, b)

      // log with two topics, the hash and `a`.
      // `b` is stored in slot zero and is 32 bytes in length.
      log2(0, 32, myEventHash, a)

    }
  }
}
```

Errors

Per the ABI standardization, errors consist of a four byte error selector and the error data. There are a few Solidity pre-defined errors, but since Solidity 0.8.4, developers can define custom errors by name and argument types.

Error Example

```
// Solidity Defined Errors:
error Panic(uint256 panicCode);
error Error(string message);

// Memory Layout to Revert with Errors:

// assert(false);
// 0x00 : 0x4e487b7100000000000000000000000000000000000000000000000000000000
// 0x20 : 0x0000000001000000000000000000000000000000000000000000000000000000

// require(false, "message");
// 0x00 : 0xf9fbd55400000000000000000000000000000000000000000000000000000000
// 0x20 : 0x0000002000000000000000000000000000000000000000000000000000000000
// 0x40 : 0x0000076d65737361676500000000000000000000000000000000000000000000
```

Section 4

Applied Yul + Solidity

Section 5

Demystifying Production Assembly

Further Resources

Educational Resources:

- EVM Codes: https://www.evm.codes/
- Yellow Paper: https://ethereum.github.io/yellowpaper/paper.pdf

Developer Tooling / Languages

- Huff Language: https://docs.huff.sh/
- Foundry Dev Environment: https://book.getfoundry.sh/
- Remix Browser IDE: https://remix.ethereum.org/

Thank you!

jtriley.eth
EVM Smart Contract Engineer
jtriley15@gmail.com
@jtriley_eth