Summary

Topics:

- Ethereum Virtual Machine (EVM)
- Memory Types
- Gas Charging
- OPCODES

Reading

Ethereum Documentation http://www.ethdocs.org/

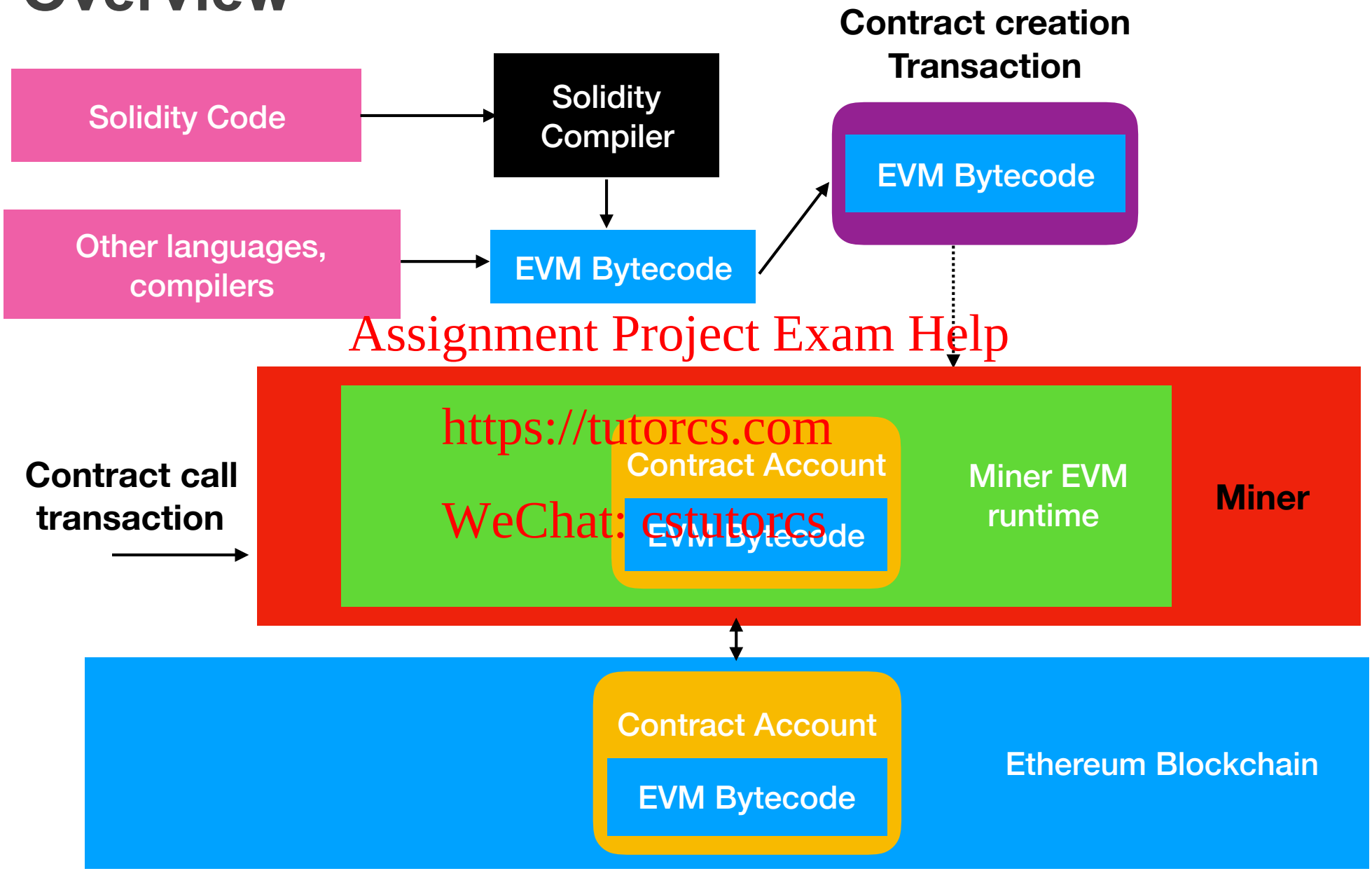
Ethereum Yellow Paper https://github.com/ethereum/yellowpaper
(The original "official" specification document for Ethereum. Very dense and terse on explanation, some errors and missing definitions. Author is no longer with the Ethereum Foundation, but retains copyright and control.)

KEVM https://github.com/kframework/evm-semantics
(Ethereum Foundation now treating this as the preferred spec. Formal, needs more expository material still.)

Ethereum Jello Paper https://jellopaper.org
A more readable interface to the KEVM repo

Beige Paper: https://github.com/chronaeon/beigepaper
An attempt at a more accessible introduction to the Yellow paper. (informal, incomplete)

Overview


Contract creation Transaction
Solidity Code
Solidity Compiler
EVM Bytecode
Other languages, compilers
EVM Bytecode
Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs
Contract call transaction
Contract Account
EVM Bytecode
Miner EVM runtime
Miner
Contract Account
EVM Bytecode
Ethereum Blockchain

Contract purposes

- Maintain a data store useful to users or other contracts
  (example: simulate a currency, maintain a membership list)
- Represent accounts with more complicated access controls, e.g., multi-sig
- Securely represent a contract or business process between multiple users
- Serve as a library of functions callable by other applications

Contract creation

Contract accounts on the blockchain are created by sending a transaction or message to the *null* address.

The payload is treated as code by the EVM and executed

The output of the execution is stored as the code of the contract account created

The address of the contract account created is determined from the *sender* of the transaction or message and the *nonce* of the sender.

Contract Interactions

Users may send a *transaction* to a contract

A contract may send a *message to another contract*

Messages are like function calls and return a *result*

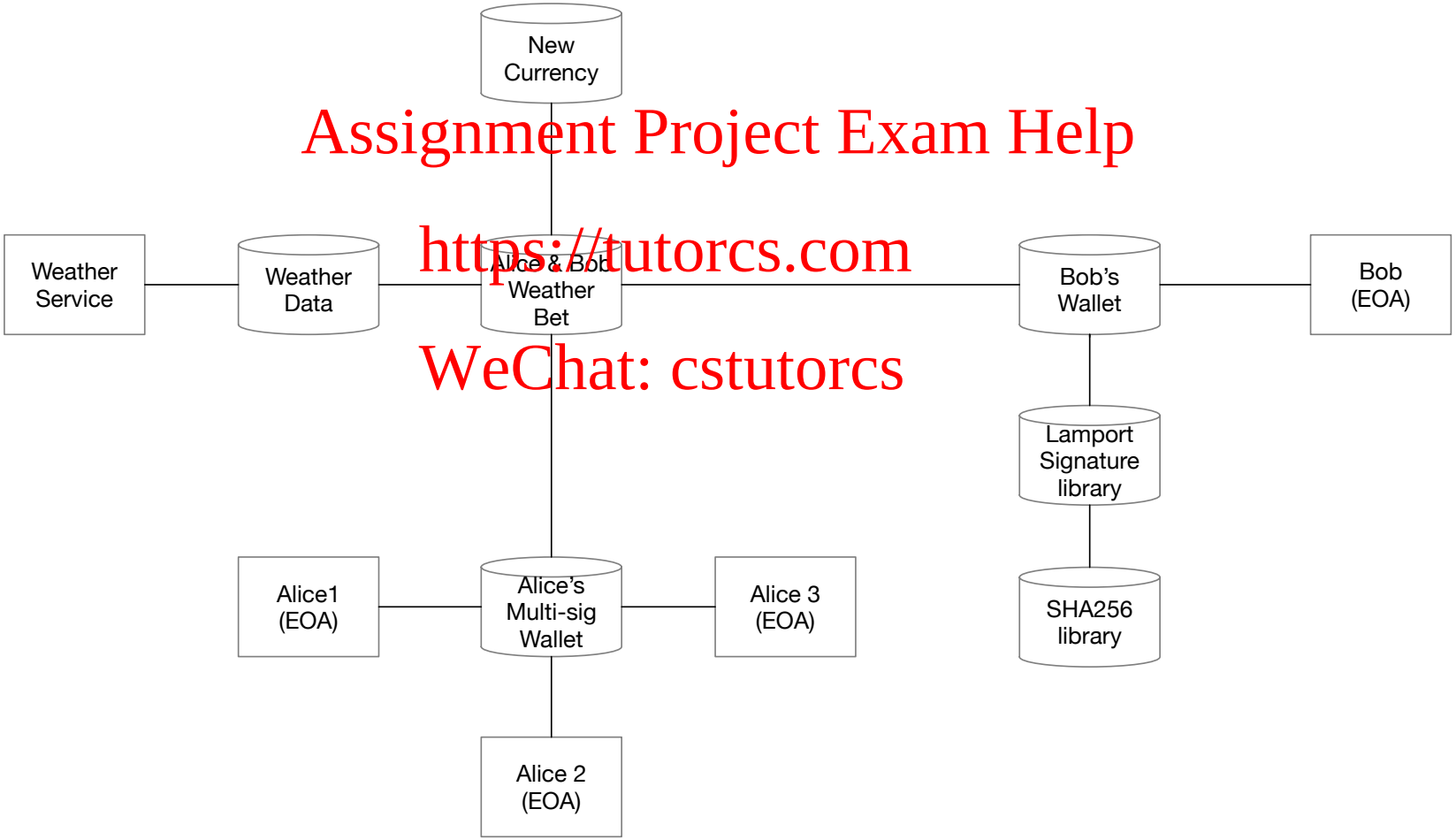
Transactions are signed using the private signature key associated to an externally owned account

This account is charged the total computation costs = *gasUsed* x *gasPrice*

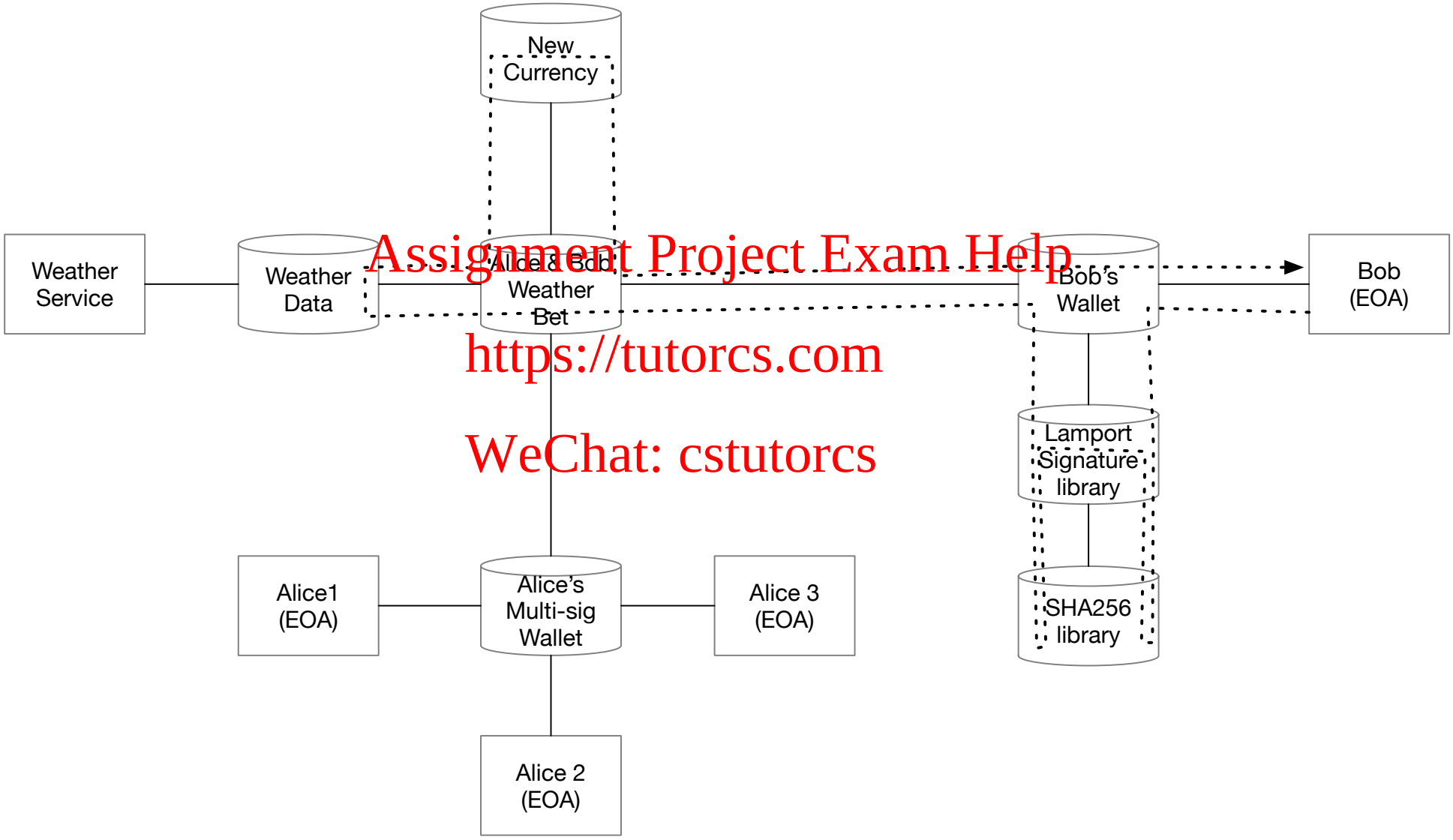
Contract Interactions

Example (from Homestead Documentation Section 1.8):

Alice and Bob bet some amount of value (in a new currency built on Ethereum) on the weather, using a weather oracle. Alice and Bob use special wallets for security.



Example: Bob wins the bet and calls the weather bet contract via his wallet contract to collect his winnings



Nonce Handling

When a transaction is called on a transaction account, the EVM:

- checks that the transaction nonce = sender's account nonce
  if not, the transaction is rejected
- increments the sender's account nonce
- executes the transaction
- if the transaction aborts for any reason, all state changes are reset,
  except that the sender's account nonce remains incremented
  (to record that the transaction was already processed)
- If the transaction causes any contract accounts to self-destruct, their deletion is scheduled to be done *after* the transaction has finished running.

Examples of Operation Cost Charges

Operation	Gas Cost	Explanation
arithmetic	3-8	
stop	0	free
stack swap,dup	3	
sha3	30	`add 6 * data_size_words + mem_expansion_cost`
sload	800	fetch from permanent storage
sstore	800-20,000	formula(zero?,changed value?)
call	9,700-25,700	formula(value sent?,new address?), add new memory cost
JUMPI	10	conditional jump
transaction	21,000	base transaction fee
contract creation	32,000	

Summary at: https://github.com/wolflo/evm-opcodes

Contract

Contracts on the blockchain are stored as *low level byte code*

When you are interacting with a contract (e.g., sending it money)
how can you know that it is going to do?

Options:

- Trust the people who claim to have created and deployed it (not really trustless!)
- Read and understand the byte code (hard work!)
- Decompile the byte code and read that (still hard)
- Obtain the source code from a registry (e.g., https://etherscan.io),
  read it and verify it compiles to the on-chain code.
  (requires maintenance of compiler versions and knowledge of exact compiler and compilation parameters used)
- Read a high level formal specification and use tools to check a formal proof that the bytecode implements it. (Work towards that in progress, but in early stages.)

EVM Memory Types

Code Store: virtual *read-only* memory (Like Harvard architecture, protects overwriting of code)

Program Counter: pointer into the code store

CallData: An array of bytes containing the input for the contract call being processed

Stack : Similar to Bitcoin Script Stack, there are no registers, so this is used for operation processing (entries are 256bit words, max depth = 1024)

Memory: short term memory used in EVM during contract execution
= array 256-bit word -> byte , all locations initially 0

Storage: long term storage (maintained on chain)
= array 256-bit word -> 256-bit word , all locations initially 0

Log: an (on chain) append only log used to return information about the computation to the caller

Call Stack: Execution Call Stack (max depth = 1024) - used for function calls to *different* contracts

Note: *everything* here is effectively *publicly visible! Code may constrain writes by checking signatures.*

Charging in more detail

Fees are charged as follows:

- per cost of the operation performed
- payment for a subordinate message call or contract creation
  (CREATE, CALL, CALLCODE operation charge)
- payment for an increase in memory usage
  (total fee is proportional to the smallest multiple of 32 bytes required for all reads and writes to be in range)
- storage:
  - long term storage is a cost to miners, so costs more than short term memory
  - in effect, storage used is paid for *up front*
  - a *refund* is given when a storage location is freed (includes contract suicide)

EVM OPCODES (Basic Operations)

Operations provided include:

- STOP
- Arithmetic Operations: ADD, MUL, SUB, DIV, EXP
- Modular Arithmetic: MOD, ADDMOD, MULMOD
- Comparisons: LT, GT, EQ, ISZERO
- Boolean: AND, OR, XOR, NOT, BYTE (retrieve byte from 256-bit word)
- Cryptographic: SHA3 (Keccak-256, actually)

EVM OPCODES (Block Information)

- Block information:

  BLOCKHASH (hash of one of the 256 most recent blocks)

- With input a block's hash:

  COINBASE (address of the beneficiary miner of the block)

  TIMESTAMP (of the block)

  NUMBER (of the block)

  DIFFICULTY (of the block)

  GASLIMIT (of the block)

EVM OPCODES (Account and Message Info)

Query Account:

ADDRESS (of currently executing account)

BALANCE (Balance of a given account)

Query Transaction/Message:

ORIGIN (sender of the original transaction)

CALLER (address of account that directly called this account)

CALLVALUE (amount of Wei deposit sent with the call)

GASPRICE (price being paid for gas used in the current call)

EVM OPCODES (Memory Management)

Stack, calldata, memory and storage operations:

POP (remove item from stack)
PUSH1, PUSH2, …., PUSH32 (push a specified number of bytes onto the stack)
DUP1, DUP2, … DUP16 (duplicate the n-th stack item)
SWAP1, SWAP2, ….., SWAP16 (exchange 1st and n-th item on the stack)

MLOAD (load word from memory) MSTORE (save word to memory)
MSIZE (get active memory size)

SLOAD (load word from storage) SSTORE (save word to storage)

CALLDATALOAD (load 32 bytes input data onto the stack)
CALLDATASIZE (get size of input data)
CALLDATACOPY (copy call data to memory)

EVM OPCODES (Control Flow)

Control Flow Operations:

- JUMP (to program location)
- JUMPI (jump to given location if nonzero, else increment program counter)
- PC (get value of program counter)

Note that because of the JUMP, JUMPI operations, loops can be represented, unlike Bitcoin Script.

EVM OPCODES (Contract Creation/Destruction)

System Operations:

CREATE (create a new contract account with associated code)

SELFDESTRUCT (halt execution and delete this contract account)

EVM OPCODES (Messages/Calls)

CALL (message call to an account, parameter are
gas, to, value, input offset, input size, output offset, output size)

STATICCALL (message call, readonly, prohibit recipient from changing state)

RETURN (halt execution and return output data)

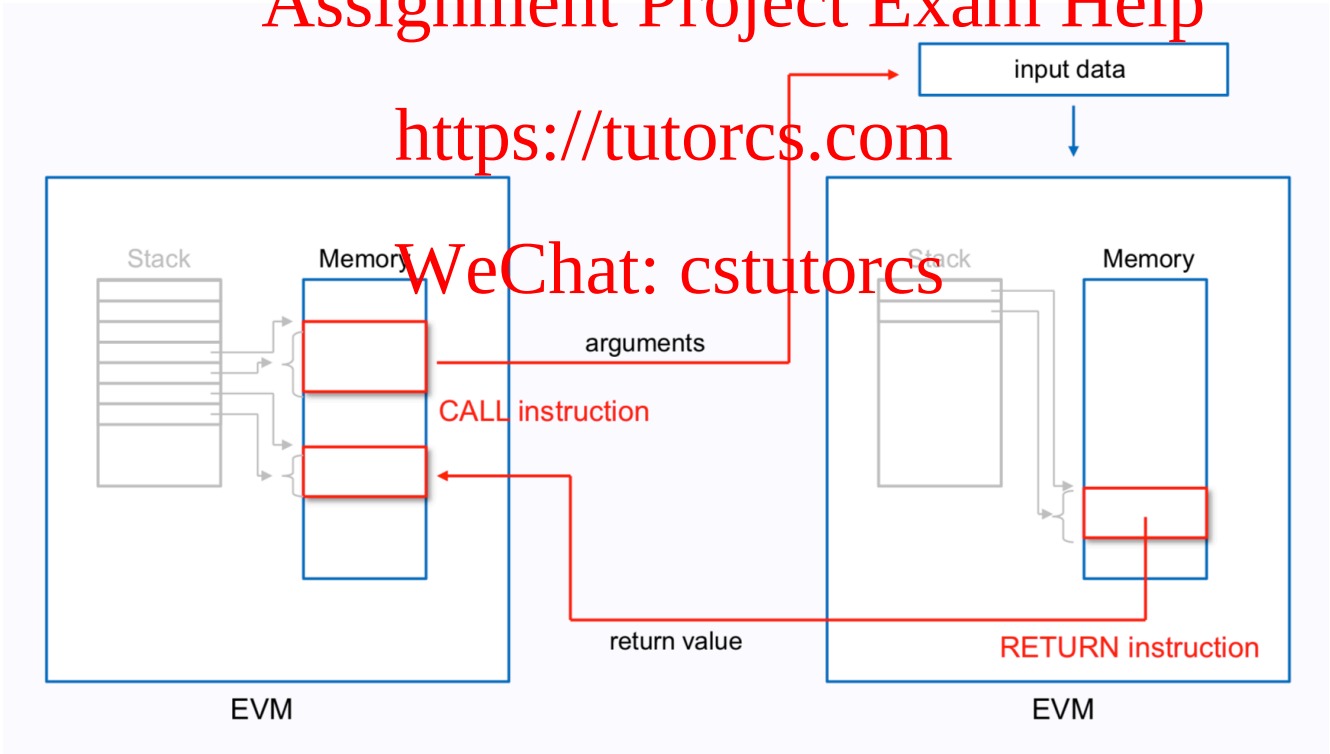


Image: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EVM OPCODES (Library calls)

The following opcodes are intended to support calls to smart contracts providing library code:

CALLCODE (message-call to *this* account, but run code from another account, with *msg.sender*=this account, and *msg.value* adjustable)

DELEGATECALL (message call into *this* account, but run code from another account, with the *same* values for *msg.sender* and *msg.value*)

DELEGATECALL is preferred - this is essentially a bug fix to CALLCODE

As usual with libraries, there is a security risk in running someone else's code with your own permissions: make sure you know what it does!

EVM OPCODES (Gas Management)

GAS (get amount of unspent gas for this computation)

If a called contract aborts, e.g., because of a stack overflow, or failed authorization check, the state is reset and the gas that was passed to it is completely consumed, even if it was not completely spent by the called contract.

Byzantium release adds opcode

REVERT - abort, reset the state and refund the caller with any unspent gas

EVM OPCODES (Logging)

Each block contains a *log*, a type of *append-only* memory where contracts that executed transactions can write information about the running of the contract code.

Contracts cannot *read* the log - it is intended for use only by outside observers of the blockchain.

For access by light clients, the log is stored using *Bloom filters.*

The log of a block is stored using a separate Merkle tree.

Opcodes:

LOG0, LOG1, … LOG4 (add a given number of topics to the log)

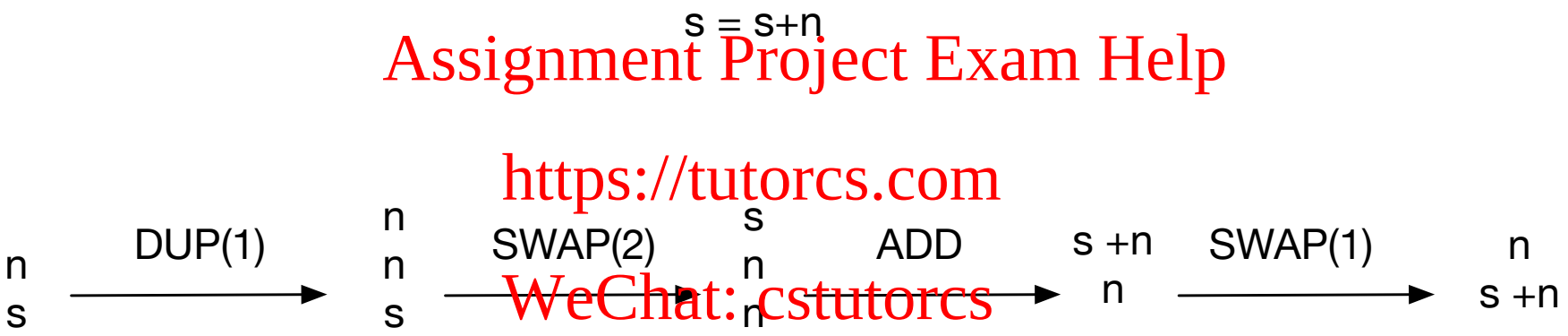
Example

From

“KEVM: A Complete Semantics of the Ethereum Virtual Machine” Everett Hildenbrandt , et al. 2017


```
s = 0 ;
n = 10 ;
while (n > 0) {
    s = s + n ;
    n = n - 1 ;
}
sstore(0, s) ;
```

```
// s = 0; n = 10
      PUSH(1, 0) ; PUSH(1, 10)
// loop: if n == 0 jump to end
    ; JUMPDEST ; DUP(1) ; ISZERO ; PUSH(1, 21) ; JUMPI
// s = s + n;
    ; DUP(1) ; SWAP(2) ; ADD
// n = n - 1
    ; SWAP(1) ; PUSH(1, 1) ; SWAP(1) ; SUB
// jump to loop
    ; PUSH(1, 4) ; JUMP
// end: store to account
    ; JUMPDEST ; POP ; PUSH(1, 0) ; SSTORE
```



Summary

- Ethereum Virtual Machine (EVM)
- Memory Types
- OPCODES
- Gas Charging