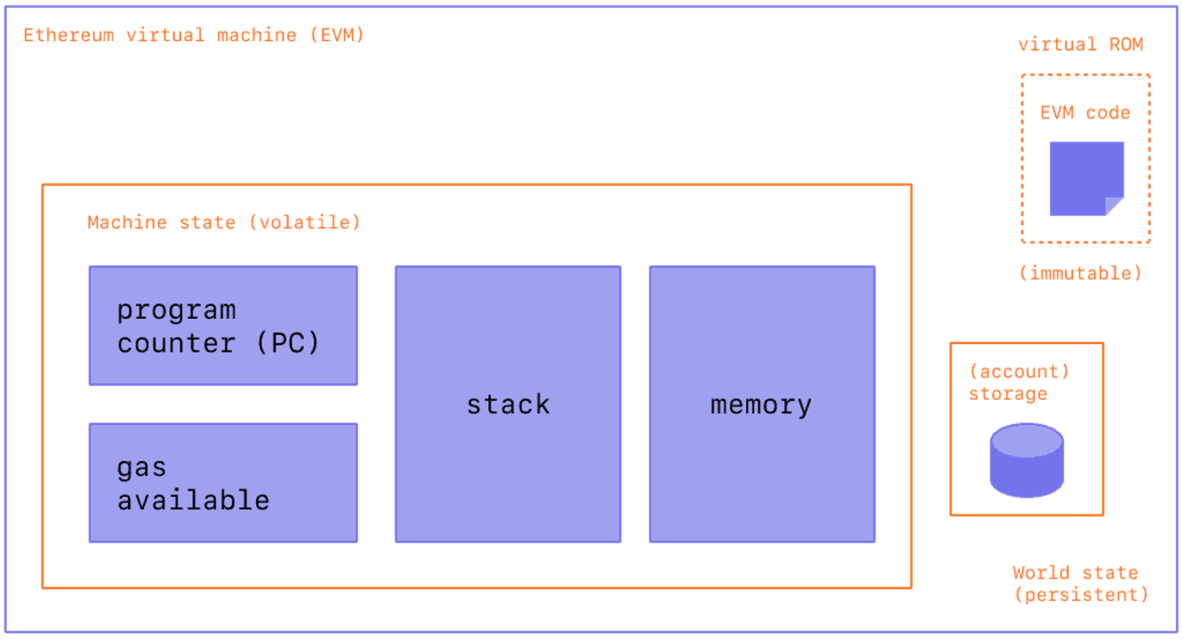
Storage & Memory

** Inspired by this tweet



- A specific instructions of each opcodes (`ADD` , `SSTORE` , `BALANCE` , or etc) is defined at **go-ethereum/core/vm/instructions.go**.
- All opcodes takes three parameters:
  1. **Program Counter** (`pc *uint64`): PC encodes which instruction should be next read by EVM.
  2. **EVMInterpreter**: A straightforward interpreter that executes EVM code.

```
type EVMInterpreter struct {
    evm   *EVM
    table *JumpTable

    hasher    crypto.KeccakState // Keccak256 hasher instance shared acr
```

```
    hasherBuf common.Hash        // Keccak256 hasher result array shared

    readOnly   bool   // Whether to throw on stateful modifications
    returnData []byte // Last CALL's return data for subsequent reuse
}
```

- `EVM` provides the necessary tools to run a contract on the give state with the provided context. For example, it contains StateDB, which gives access to the underlying state.

  ▼ EVM structure

```
// EVM is the Ethereum Virtual Machine base object and provides
// the necessary tools to run a contract on the given state with
// the provided context. It should be noted that any error
// generated through any of the calls should be considered a
// revert-state-and-consume-all-gas operation, no checks on
// specific errors should ever be performed. The interpreter make
// sure that any errors generated are to be considered faulty code
//
// The EVM should never be reused and is not thread safe.
type EVM struct {
    // Context provides auxiliary blockchain related information
    Context BlockContext
    TxContext
    // StateDB gives access to the underlying state
    StateDB StateDB
    // Depth is the current call stack
    depth int

    // chainConfig contains information about the current chain
    chainConfig *params.ChainConfig
    // chain rules contains the chain rules for the current epoch
    chainRules params.Rules
    // virtual machine configuration options used to initialise the
    // evm.
```

```
    Config Config
    // global (to this context) ethereum virtual machine
    // used throughout the execution of the tx.
    interpreter *EVMInterpreter
    // abort is used to abort the EVM calling operations
    abort atomic.Bool
    // callGasTemp holds the gas available for the current call. This
    // available gas is calculated in gasCall* according to the 63/64
    // applied in opCall*.
    callGasTemp uint64
}
```

- `JumpTable` contains the EVM opcodes supported at a given fork.

3. **ScopeContext**

```
type ScopeContext struct {
    Memory   *Memory
    Stack    *Stack
    Contract *Contract
}
```

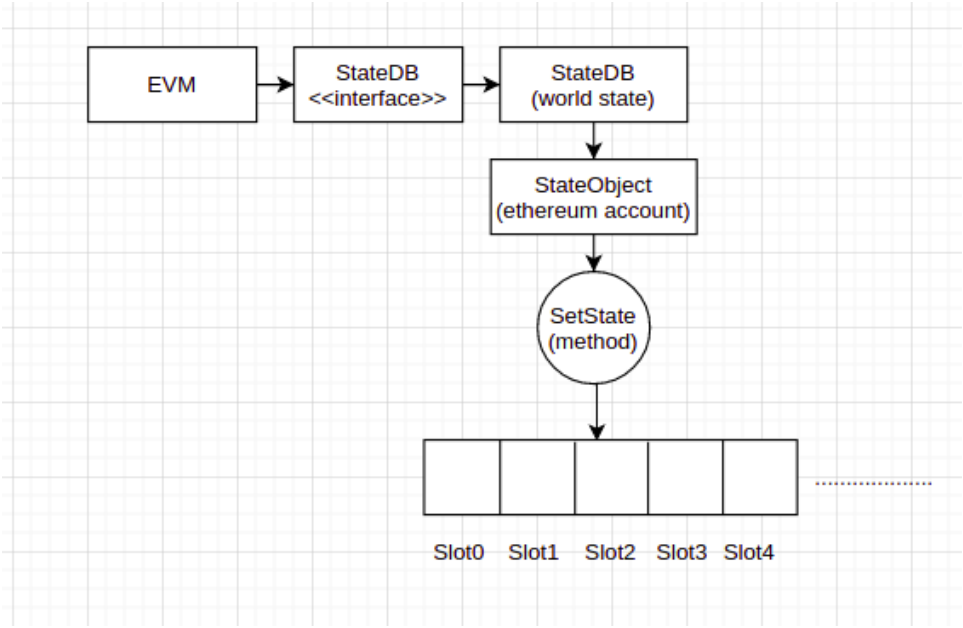
- `Contract` contains information about a compiled contract, along with its code and runtime code.

```
type Contract struct {
    Code        string          `json:"code"`
    RuntimeCode string          `json:"runtime-code"`
    Info        ContractInfo    `json:"info"`
    Hashes      map[string]string `json:"hashes"`
}

type ContractInfo struct {
    Source          string      `json:"source"`
    Language        string      `json:"language"`
    LanguageVersion string      `json:"languageVersion"`
```

```
    CompilerVersion string      `json:"compilerVersion"`
    CompilerOptions string      `json:"compilerOptions"`
    SrcMap          interface{} `json:"srcMap"`
    SrcMapRuntime   string      `json:"srcMapRuntime"`
    AbiDefinition   interface{} `json:"abiDefinition"`
    UserDoc         interface{} `json:"userDoc"`
    DeveloperDoc    interface{} `json:"developerDoc"`
    Metadata        string      `json:"metadata"`
}
```

1. `SSTORE`



Definition

- writes a (u)int256 to storage
- pays 20000 gas at most (refund & cold/hot storage logic)

When is it used?

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
contract Example {
    // Storage variable declared
    uint number;

    function setNumber() public {
        // opcode SSTORE is used to store value at storage variable
        number = 1;
    }
}
```

Geth Implementation

```
func opSstore(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext
    if interpreter.readOnly {
        return nil, ErrWriteProtection
    }
    loc := scope.Stack.pop()
    val := scope.Stack.pop()
    interpreter.evm.StateDB.SetState(scope.Contract.Address(), loc.Bytes32(),
    return nil, nil
}
```

- `SSTORE` pops 2 values from the stack, and uses them as location and value to store.
- It brings the instance of `StateDB` interface from interpreter struct, and save the state provided from the stack.

  `StateDB` structs are used to store anything within merkle trie. It's a general query interface to retrieve Contracts or Accounts.

  ▼ StateDB struct

  ```
  type StateDB struct {
      db         Database
      prefetcher *triePrefetcher
  ```

```
trie       Trie
hasher     crypto.KeccakState
snaps      *snapshot.Tree    // Nil if snapshot is not available
snap       snapshot.Snapshot // Nil if snapshot is not available

// originalRoot is the pre-state root, before any changes were made
// It will be updated when the Commit is called.
originalRoot common.Hash

// These maps hold the state changes (including the corresponding
// original value) that occurred in this **block**.
accounts       map[common.Hash][]byte                  // The mutated
storages       map[common.Hash]map[common.Hash][]byte  // Th
accountsOrigin map[common.Address][]byte               // The origi
storagesOrigin map[common.Address]map[common.Hash][]byte //

// This map holds 'live' objects, which will get modified while proce
// a state transition.
stateObjects         map[common.Address]*stateObject
stateObjectsPending  map[common.Address]struct{}           // State
stateObjectsDirty    map[common.Address]struct{}           // State c
stateObjectsDestruct map[common.Address]*types.StateAccount

// DB error.
// State objects are used by the consensus core and VM which are
// unable to deal with database-level errors. Any error that occurs
// during a database read is memoized here and will eventually be
// returned by StateDB.Commit. Notably, this error is also shared
// by all cached state objects in case the database failure occurs
// when accessing state of accounts.
dbErr error

// The refund counter, also used by state transitioning.
refund uint64

// The tx context and all occurred logs in the scope of transaction.
```

```
thash   common.Hash
txIndex int
logs    map[common.Hash][]*types.Log
logSize uint

// Preimages occurred seen by VM in the scope of block.
preimages map[common.Hash][]byte

// Per-transaction access list
accessList *accessList

// Transient storage
transientStorage transientStorage

// Journal of state modifications. This is the backbone of
// Snapshot and RevertToSnapshot.
journal        *journal
validRevisions []revision
nextRevisionId int

// Measurements gathered during execution for debugging purpose
AccountReads         time.Duration
AccountHashes        time.Duration
AccountUpdates       time.Duration
AccountCommits       time.Duration
StorageReads         time.Duration
StorageHashes        time.Duration
StorageUpdates       time.Duration
StorageCommits       time.Duration
SnapshotAccountReads time.Duration
SnapshotStorageReads time.Duration
SnapshotCommits      time.Duration
TrieDBCommits        time.Duration

AccountUpdated int
StorageUpdated int
```

```
    AccountDeleted int
    StorageDeleted int

    // Testing hooks
    onCommit func(states *triestate.Set) // Hook invoked when commit
}
```

▼ SetState

```
// statedb.go
func (s *StateDB) SetState(addr common.Address, key, value commo
    stateObject := s.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.SetState(key, value)
    }
}

// state_object.go
// SetState updates a value in account storage.
func (s *stateObject) SetState(key, value common.Hash) {
    // If the new value is the same as old, don't set
    prev := s.GetState(key)
    if prev == value {
        return
    }
    // New value is different, update and journal the change
    s.db.journal.append(storageChange{
        account:  &s.address,
        key:      key,
        prevalue: prev,
    })
    s.setState(key, value)
}

func (s *stateObject) setState(key, value common.Hash) {
```

```
    s.dirtyStorage[key] = value
}
```

`SetState` stores the given value if the value is **different** from the previously stored value.

- If the interpreter is set to be `readOnly` (maybe `view` instruction in Solidity), it returns `ErrWriteProtection` error.

2. `SLOAD`

Definition

- reads a (u)int256 from storage
- pays 2100 gas for cold storage / 100 gas for hot storage

When is it used?

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Example {
    // Storage variable initialized as 6
    uint number = 6;

    function getNumber() public returns (uint) {
        // opcode SSTORE is used to read value at storage variable
        return number;
    }
}
```

Geth implementation

```
func opSload(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext)
    loc := scope.Stack.peek()
    hash := common.Hash(loc.Bytes32())
    val := interpreter.evm.StateDB.GetState(scope.Contract.Address(), hash)
```

```
    loc.SetBytes(val.Bytes())
    return nil, nil
}
```

- `opSload` copys value from the top of stack, and uses its hash as location to get the state.

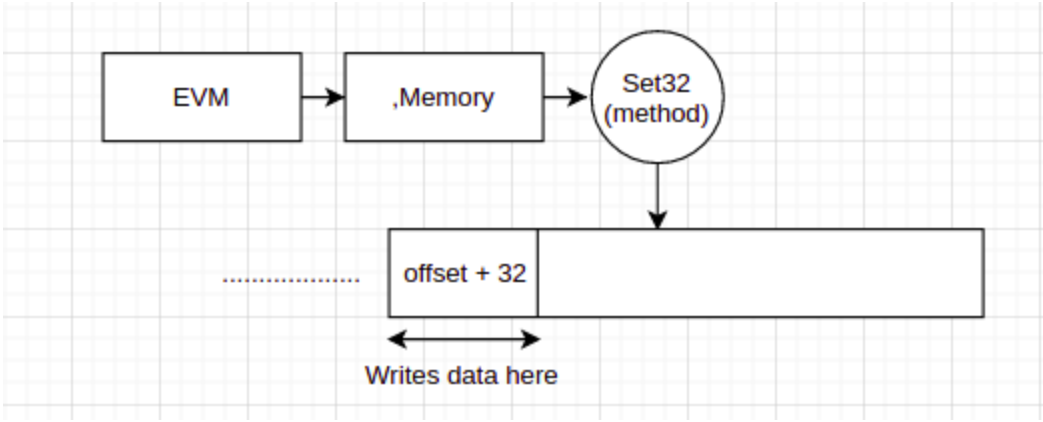
▼ `GetState` function

```
// statedb.go
func (s *StateDB) GetState(addr common.Address, hash common.Hash)
    stateObject := s.getStateObject(addr)
    if stateObject != nil {
        return stateObject.GetState(hash)
    }
    return common.Hash{}
}


// state_object.go

// GetState retrieves a value from the account storage trie.
func (s *stateObject) GetState(key common.Hash) common.Hash {
    // If we have a dirty value for this state entry, return it
    value, dirty := s.dirtyStorage[key]
    if dirty {
        return value
    }
    // Otherwise return the entry's original value
    return s.GetCommittedState(key)
}
```

3. `MSTORE`



Definition

- writes a (u)int256 to memory
- pays 16 gas per bytes (non-zero value) / 4 gas per bytes (zero value), but it increases quadratically after certain threshold.

When is it used?

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Example {

    function returnSix() public returns (uint) {
        // Allocate value to memory variable
        uint number = 6;
        return number;
    }

}
```

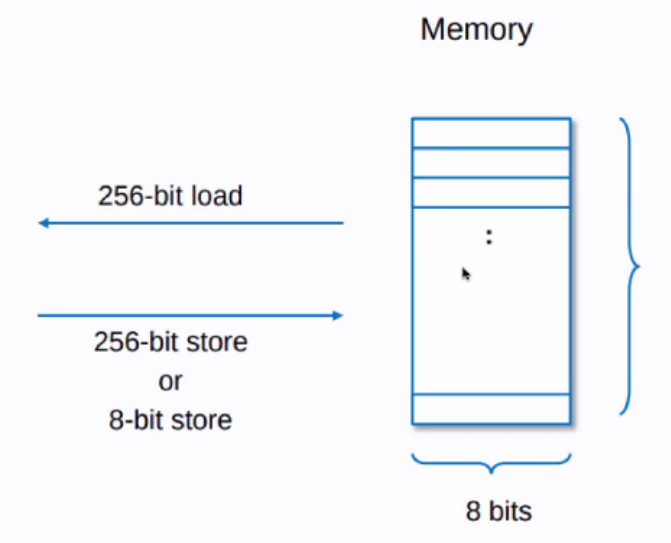
Geth Implementation

```
func opMstore(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContex
    // pop value of the stack
    mStart, val := scope.Stack.pop(), scope.Stack.pop()
    scope.Memory.Set32(mStart.Uint64(), &val)
```

```
    return nil, nil
}
```

- It pulls data and location from the stack, and copies it inside the memory.
- Certain offset is needed to be specified to store data in the memory location.
- Data inside memory is stored in 32 bytes.

  The structure of memory in Solidity:



- `Set32` first creates and zeros out the `offset + 32` data locations in order to create space inside the memory and then stores the data at that location.

4. `MLOAD`

Definition

- reads a (u)int256 from storage
- 3 gas per bytes

Where is it used?

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
contract Example {

    function returnSix() public returns (uint) {
        uint number = 6;
        // When returning, the value is read from memory
        return number;
    }

}
```

Geth Implementation