NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Ethereum

Operation of the "World Computer"

Dr Sourav SEN GUPTA
Lecturer, SCSE, NTU

Operating Framework

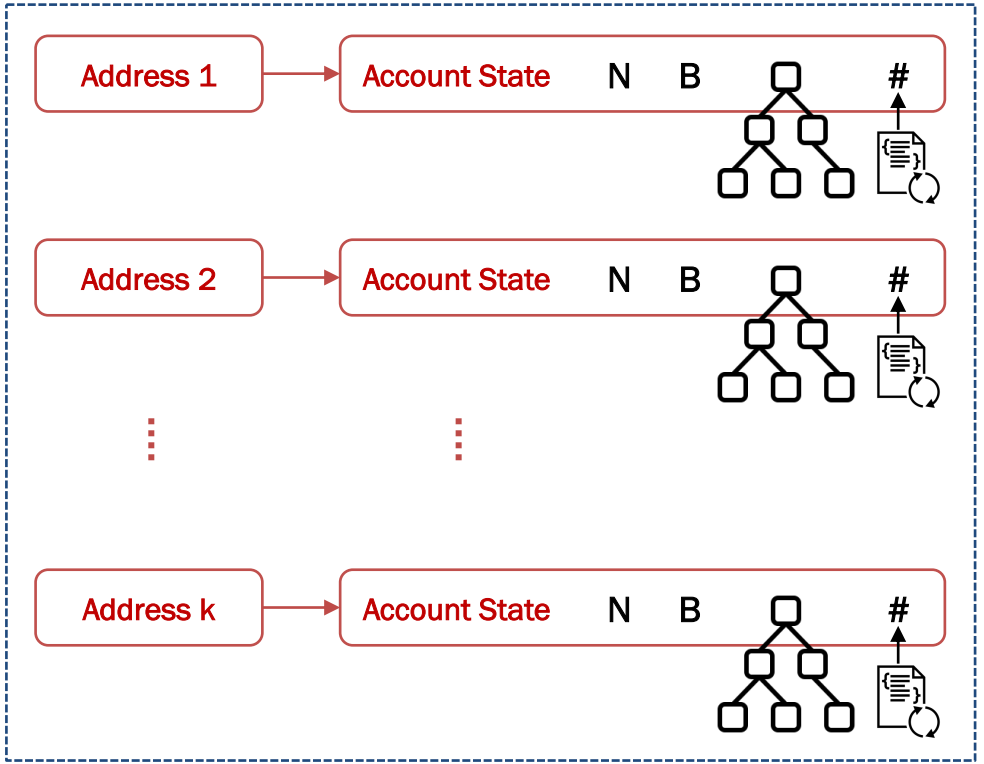
Ethereum Virtual Machine

Recall : Ethereum “World State”

The **world state** of Ethereum is a **map** between Addresses and corresponding Account States.

- Addresses are 160-bit identifiers
- States **not stored** on Blockchain
- Mapping maintained in a trie DB
- Root Hash identifies “world state”

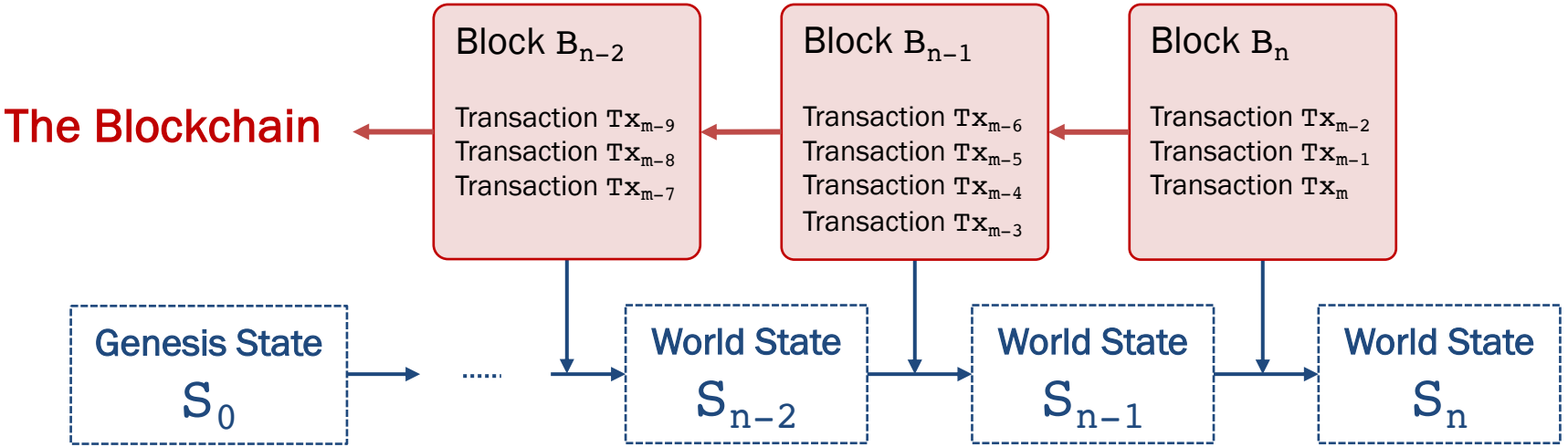
State : Collection of (Code + Data)



Recall : State-Transition Chain

Ethereum state machine was initiated with a "**genesis state**" in the beginning.

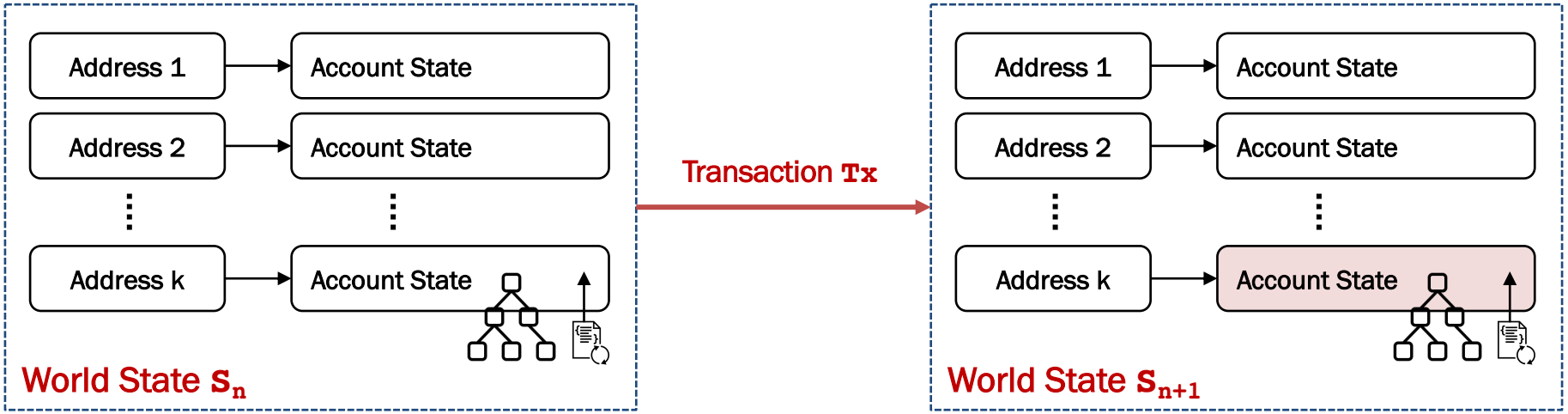
At each **epoch**, previous state updates to the next state through **Transactions**.



Recall : State Transition

Transactions act as **atomic digitally-signed instructions** on the **world state.**

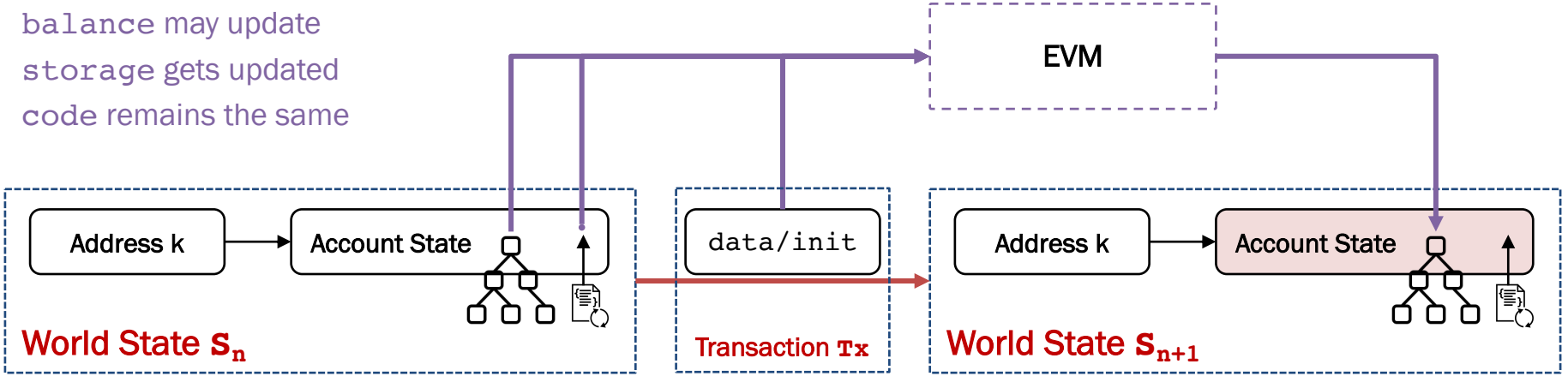
Transactions and resultant messages update one or more Account States.



Ethereum Virtual Machine (EVM)

Contract bytecode is executed on Ethereum Virtual Machine (EVM), based on the Account State and the input `data/init` specified in the Transaction `Tx`.

`nonce` gets updated
`balance` may update
`storage` gets updated
`code` remains the same

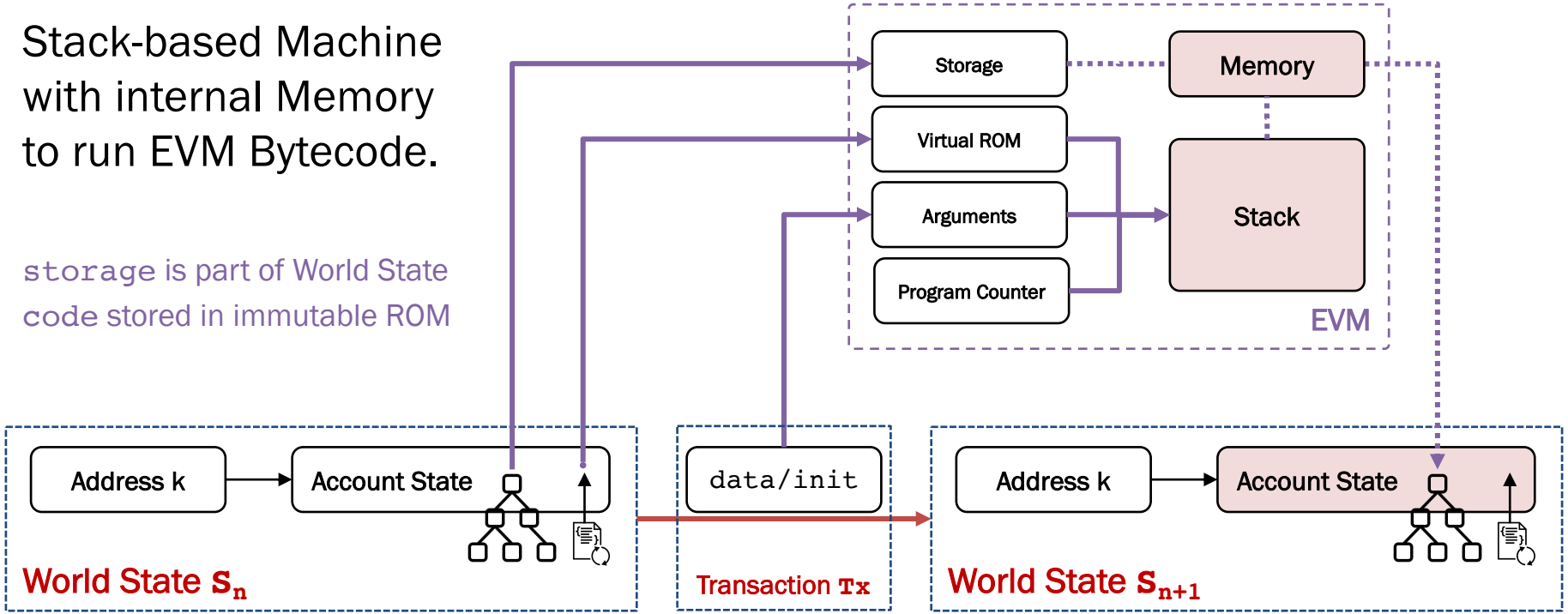


EVM Architecture

Stack-based Machine with internal Memory to run EVM Bytecode.

`storage` is part of World State

`code` stored in immutable ROM

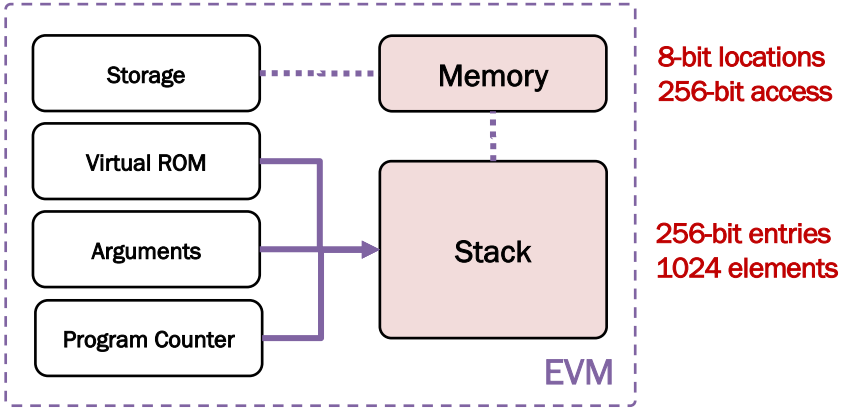


EVM Instruction Set

EVM Code : Low-level Stack-based Bytecode

Opcodes allowed in Contract Bytecode

- Operations : Arithmetic and Logic
- Access : Stack, Memory, Storage
- Control : Process Flow Operations
- Enquiries : Execution Environment
- System : Logging/Calling Operators

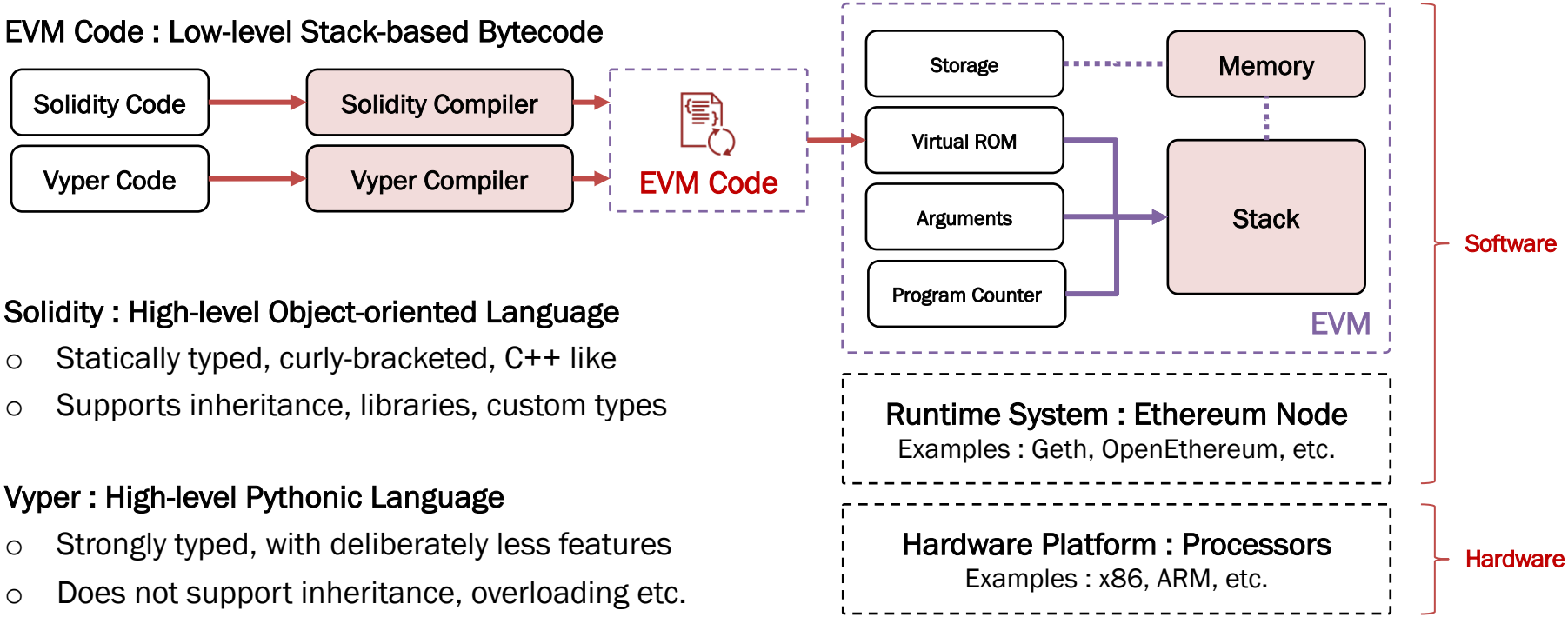


EVM has access to account information (address, balance, etc.) from the “World State” and basic block information (block number, gas price etc.) from Ethereum Blockchain.

ref: Ethereum Virtual Machine Opcodes (https://www.ethervm.io)

EVM Compilers

EVM Code : Low-level Stack-based Bytecode



Solidity : High-level Object-oriented Language

- Statically typed, curly-bracketed, C++ like
- Supports inheritance, libraries, custom types

Vyper : High-level Pythonic Language

- Strongly typed, with deliberately less features
- Does not support inheritance, overloading etc.

Turing Completeness

Turing Complete programming languages/systems support the execution of any program.

Concerns with a Turing Complete language for Ethereum

- Turing complete languages allow the execution of programs that may run forever.
- It is not possible to tell whether a program will run forever by checking the code.
- The only way to check if such a program may run forever is to actually execute it.
- This will be a huge problem in Ethereum, as the entire World State will be stalled.

Solution #1

Deliberately use a Turing Incomplete language as in Bitcoin : Loss for the “World Computer”.

Solution #2

Make the language *quasi*-Turing Complete, with upper bounds on the “cost of computation”.

Cost of Computation

Gas and Payment

Recall : Ethereum Transaction

Digitally signed instruction constructed and sent by Externally Owned Account.

`nonce`	:	Number of transactions sent by the Sender
`to`	:	160-bit address of the Recipient's Account
`value`	:	Number of Wei to transfer to the Recipient
`v,r,s`	:	Values for Sender's Signature and Identity
`data/init`	:	Byte array for Input Data or Code Creation
`gasPrice`	:	Number of Wei to be paid for the Execution
`gasLimit`	:	Maximum amount of gas for the Execution

Transactions accomplish either Ether Transfer, Message or Contract Creation.

Gas Cost for Execution

Every operation triggered by Transactions requires **computation and storage** resources.

Gas is the internal *unit* to measure resources spent by the miners to run the operation.

Economics and Security

- Each operation costs a fixed amount of gas, based on computation and storage.
- Gas acts as a buffer between market volatility of Ether and incentives for miners.
- Most importantly, gas acts as a deterrent for malicious denial-of-service attacks.

Examples[1] Gas = 3 for operations like ADD, SUB, AND, XOR, PUSH, DUP, etc.
Gas = 5 for operations like MUL, DIV, MOD, SELFBALANCE, etc.
Gas = 21000 for every transaction; 32000 for contract-creation.

[1] reading : Ethereum Yellow Paper (https://ethereum.github.io/yellowpaper/paper.pdf)

Gas Accounting in Ethereum

`gasLimit` : Maximum amount of gas for the Execution of a Transaction

- Specified by the creator of the transaction, estimating the total gas cost.
- Must be greater than the overall “intrinsic gas” used by the transaction.
- EVM deducts gas used for each operation before running the next one.
- If there is no gas to run the next operation, the transaction is aborted.
- Even if the transaction is aborted, 21000 gas is charged for the work.

Example[2] Transaction with 70-byte `init` (10 zero bytes, 60 non-zero bytes) to create a contract.
Gas for transaction : 21000; Gas price for contract creation = 32000 (on top of the rest);
Gas for zero bytes of data/code = 10 x 4 = 40; Gas for non-zero bytes = 60 x 68 = 4080.
May set `gasLimit` = 80000 for this estimate of 21000 + 32000 + 40 + 4080 = 57120.

[2] reading : How does Ethereum work anyway? (https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway)

Gas Cost vs Gas Price

`gasPrice` : Number of Wei to be paid for the Execution

- Specified by the creator of the transaction, estimating miners' strategy.
- Transaction creator's account debited `gasLimit*gasPrice` in Ether.
- Final `gasCost` is computed by miner after execution of the transaction.
- `gasCost*gasPrice` goes to the miner as the fee for this transaction.
- Balance `(gasLimit-gasCost)*gasPrice` is "refunded" to creator.

Example Transaction with 70-byte init (10 zero bytes, 60 non-zero bytes) to create a contract.
You set `gasLimit` = 80000 for an estimate of 21000 + 32000 + 40 + 4080 = 57120.
Suppose you set `gasPrice` = 20 gwei and the final `gasCost` is 70000 after execution.
Pre-purchase = 80000 x 20 gwei = 0.0016 eth, Refund = 0.0002 eth, Fee = 0.0014 eth.

Computing on Ethereum

Smart Contracts

Recall : Contract Accounts

State : { nonce, balance, storage, code }

Action : Send Messages

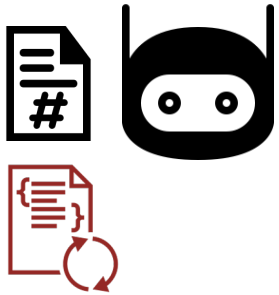
- to transfer Ether to some EOA
- to trigger some Contract Code
- to create some new Contract

Action : Execute Contract Code

- on receiving “calls” from others
- to update storage and balance
- to send Messages as required

Address : 0x2a0c0D…50c208

Nonce : 257
Balance : 38223 Ether (in Wei)
Storage : storageRoot (trie)
Code : codeHash (contract)



```
pragma solidity ^0.4.16;
mapping (address =>
          mapping (address => uint256))
          public tokens;
contract Token { … }
contract Exchange { … }
```

ref : https://etherscan.io/address/0x2a0c0dbecc7e4d658f48e01e3fa353f44050c208

Contracts in a Nutshell

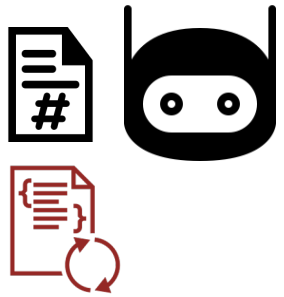
Programs — Simply pieces of code. Not a legal “contract” in this context.

Immutable — Once deployed, the code of a smart contract cannot change.
To modify a smart contract is to deploy a “new instance” of it.

Deterministic — Execution produces the same outcome for everyone who runs.
Unless the context of the transaction or world state is different.

EVM Context — Contracts can access own state and the context of transaction.
Can also access some information about the most recent blocks.

Decentralized — EVM runs as a local instance on every Ethereum node, identically.
After consensus, a single version of truth prevails for the execution.



```
{
  nonce,
  balance,
  storage,
  code
}
```

Example : (unsafe) Faucet Contract

Gives out Ether to any Address that requests for some Ether.

Features of the Faucet

- `contract` object with scope `{…}`
- `external` function `receive(…)` accepts any incoming refill amount.
- `public` function `withdraw(…)` allows anyone to withdraw amount.
- `msg.sender.transfer(…)` transfers the amount to requestor.

```
1  // SPDX-License-Identifier: CC-BY-SA-4.0
2
3  // Version of Solidity compiler this program was written for
4  pragma solidity 0.6.4;
5
6  // Our first contract is a faucet!
7  contract Faucet {
8      // Accept any incoming amount
9      receive() external payable {}
10
11     // Give out ether to anyone who asks
12     function withdraw(uint withdraw_amount) public {
13         // Limit withdrawal amount
14         require(withdraw_amount <= 100000000000000000);
15
16         // Send the amount to the address that requested it
17         msg.sender.transfer(withdraw_amount);
18     }
19 }
```

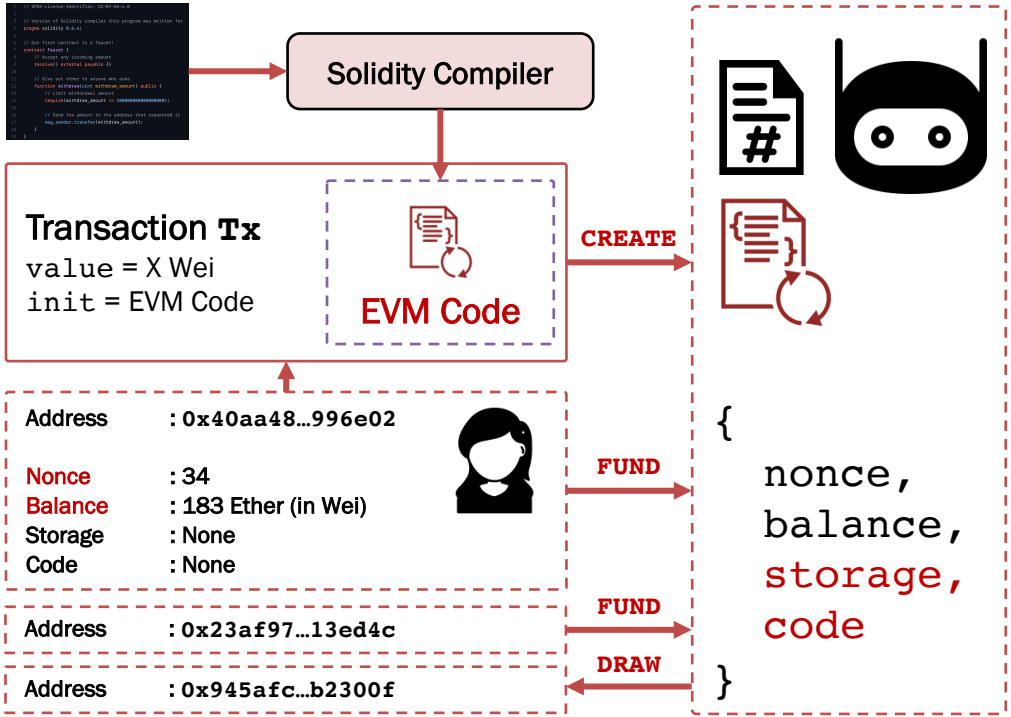
ref : https://github.com/ethereumbook/ethereumbook/

Example : Contract Lifecycle

Typical lifecycle of Contracts

- **Write** the Contract (Solidity)
- **Compile** it to EVM Bytecode
- **Deploy** via Contract CREATE
- **View** in Blockchain Explorer
- Fund the deployed Contract
- Withdraw from the Contract

Immutable **code** instantiated and executed on Ethereum Blockchain.



ref : https://github.com/ethereumbook/ethereumbook/