**UNIT IV**

|  |
| --- |
| **Smart Contracts and Ethereum 101:**   * Smart Contracts: * Definition * Ricardian contracts.   **Ethereum 101:**   * Introduction * Ethereum blockchain * Elements of the Ethereum blockchain * Precompiled contracts. |

**Smart Contracts**

**History**

Smart contracts were first theorized by Nick Szabo in the late 1990s, 20 years before the true potential and benefits of them were truly appreciated.  
The idea of smart contracts was implemented in a limited fashion in Bitcoin in 2009, where Bitcoin transactions can be used to transfer the value between users over a peer-to-peer network where users do not necessarily trust each other, and there is no need for a trusted intermediary.

**Definition**

* A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.
* A computer program that is written in a language that a computer or target machine can understand.
* Encompasses agreements between parties in the form of business logic.
* Smart contracts are automatically executed when certain conditions are met.
* **Enforceable**: All contractual terms are executed as defined and expected, even in the presence of adversaries.
* Smart Contracts are secure and unstoppable.
* Computer programs are required to be designed in such a fashion that they are fault-tolerant and executable in a reasonable amount of time.
* The program should be able to execute and maintain a healthy internal state, even if external factors are unfavourable.
* It will provide greater benefits in the long run if security and unstoppable properties are included in the smart contract.
* Certain inputs that need to be provided by people can and should also be automated via the use of Oracles.
* Smart contracts usually operate by managing their internal state using a state machine model.
* A smart contract might be more acceptable in legal situations if smart contract code is readable not only by machines but also by people.
* If humans and machines can both understand the code written in a smart contract.
* Smart contracts are inherently required to be deterministic in nature.
* This property will allow a smart contract to be run by any node on a network and achieve the same result.

**Properties**

A smart contract has the following four properties:

* Automatically executable
* Enforceable
* Semantically sound
* Secure and unstoppable
* The first two properties are required as a minimum, while the other two may not be required or implementable in certain scenarios and can be relaxed.

**Ricardian Contracts**

**History**

* Ricardian contracts were originally proposed in the *Financial Cryptography in 7 Layers* paper by Ian Grigg in the late 1990s.
* Contracts were used initially in a bond trading and payment system called Ricardo.
* The key idea is to write a document that is understandable and acceptable by both a court of law and computer software.
* Ricardian contracts address the challenge of the issuance of value over the Internet.

**Definition**

A Ricardian contract:

* Identifies the issuer.
* Captures all the terms and clauses of the contract in a document to make it acceptable as a legally binding contract.

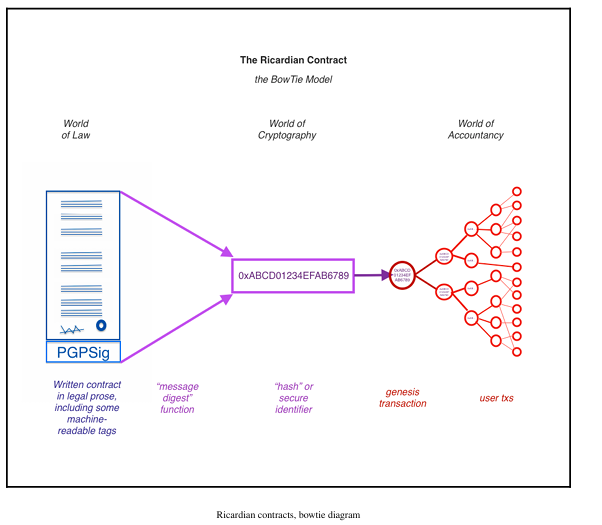
**Properties**

A Ricardian contract is a document that has several of the following properties:

* Contract offered by an issuer to holders.
* Valuable right held by holders and managed by the issuer.
* Easily readable by people (like a contract on paper).
* Readable by programs (parseable, like a database).
* Digitally signed.
* Carries the keys and server information.
* Allied with a unique and secure identifier.

**Implementation**

* Ricardian contracts are implemented by producing a single document containing:
  + The terms of the contract in legal prose.
  + Required machine-readable tags.
* The document is digitally signed by the issuer using their private key.
* The document is then hashed using a message digest function to produce a hash by which the document can be identified.
* The hash is further used and signed by parties during the performance of the contract to link each transaction, with the identifier hash serving as evidence of intent.  
  This is called the **bowtie model**.



**Ricardian Contracts**

* A Ricardian contract is different from a smart contract.
* A **smart contract** does not include any contractual document and is focused purely on the **execution** of the contract.
* A Ricardian contract is more concerned with the **semantic richness** and the production of a document that contains contractual legal prose.

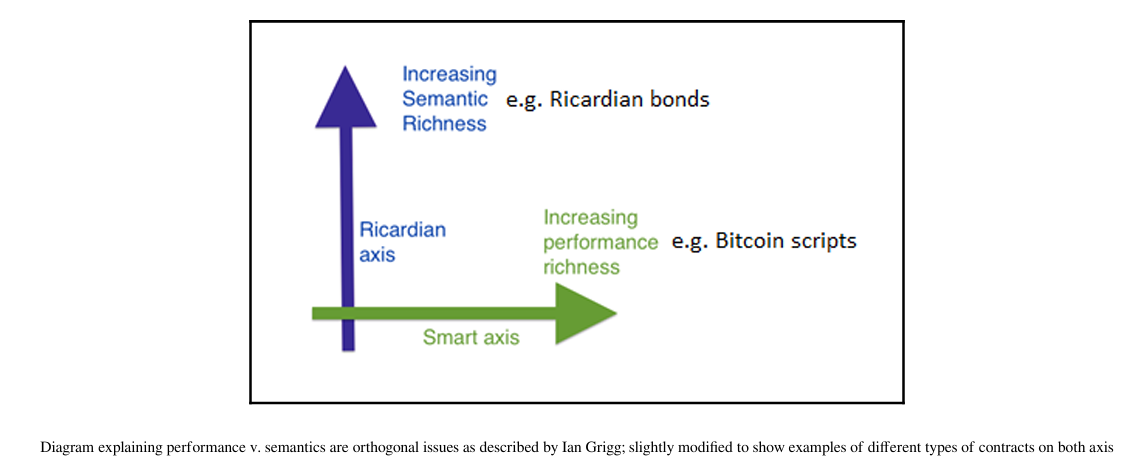
**Semantics of a Contract**

The semantics of a contract can be divided into two types:

1. **Operational Semantics**
   * Defines the actual execution, correctness, and safety of the contract.
2. **Denotational Semantics**
   * Concerned with the real-world meaning of the full contract.

**Smart Contracts vs. Ricardian Contracts**

* Some researchers have differentiated between **smart contract code** and **smart legal contracts**:
  + A **smart contract** is only concerned with the execution of the contract.
  + A **smart legal contract** encompasses both the **denotational** and **operational semantics** of a legal agreement.
* **Bitcoin** demonstrates a very simple implementation of a smart contract that is fully oriented towards the execution of the contract.
* In contrast, a **Ricardian contract** is more geared towards producing a document that is:
  + Understandable by humans.
  + Contains some parts that a computer program can understand.



**Ricardian Contracts**

* Smart contract is made up to have both performance and semantics embedded together.
* Ricardian contract can be represented as a tuple of three objects, namely **Prose**, **Parameters**, and **Code**:
  + **Prose** represents the legal contract in regular language.
  + **Code** represents the program that is a computer-understandable representation of legal prose.
  + **Parameters** join the appropriate parts of the legal contract to the equivalent code.
* Ricardian contracts have been implemented in many systems, such as:
  + CommonAccord
  + OpenBazaar
  + OpenAssets
  + Askemos

**Smart Contract Templates**

* Smart contracts can be implemented for any industry.
* Most current use cases are related to the financial industry.
* Recent work in the smart contract space, specific to the financial industry, has proposed the idea of **Smart Contract Templates**:
  + The idea is to build standard templates that provide a framework to support legal agreements for financial instruments.
  + **CLACK**, a common language for augmented contract knowledge, has been proposed, and research has begun to develop the language.
    - This language is intended to be very rich and provide a large variety of functions, ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

**Domain-Specific Languages (DSLs)**

* Contracts in the finance industry is not a new concept.
* Various DSLs are already in use in the financial industry to provide specific language for a specific domain:
  + DSLs are developed with limited expressiveness for a particular application or area of interest.
  + **DSLs vs GPLs (General-Purpose Programming Languages)**:
    - DSLs have a small set of features optimized for the intended domain.
    - Not used to build general-purpose large application programs.
* **Solidity** is one such language introduced with Ethereum blockchain to write smart contracts.
* **Serpent** is another language introduced with Ethereum, though it's not as widely used as Solidity.

**Graphical Domain-Specific Language**

* The idea of domain-specific languages for smart contract programming can be further extended to a **graphical DSL**, a smart contract modeling platform:
  + A **domain expert** (not a programmer) can use a graphical user interface and a canvas to define and draw the semantics and performance of a financial contract.
  + Once the flow has been drawn and completed, it can:
    - Be emulated first to test.
    - Then deployed from the same system to the target platform, which can be a blockchain.
* Research should also be conducted in developing **high-level DSLs** for programming smart contracts in a **user-friendly graphical user interface**, allowing non-programmers to design smart contracts.

**Oracles**

* Oracles are an important component of the smart contract ecosystem.
* Limitation with smart contracts:
  + They cannot access external data (e.g., stock prices of securities required by the contract to release dividend payments).
* **Oracles** provide external data to smart contracts.
* Oracle is an interface that delivers data from an external source to smart contracts.
* Depending on the industry and requirements, Oracles can deliver different types of data, ranging from:
  + Weather reports, real-world news, corporate actions, to data from Internet of Things (IoT) devices.
* Oracles are **trusted entities** that use a secure channel to transfer data to a smart contract.

**Capabilities of Oracles**

* Oracles are capable of digitally signing the data to prove the source's authenticity.
* Smart contracts can **subscribe** to the Oracles to:
  + Pull data, or
  + Allow Oracles to push the data to smart contracts.

**Concerns about Oracles**

* Oracles should not be able to manipulate the data they provide.
* Must provide **authentic data**.
* Even though Oracles are trusted, it may still be possible for data to be manipulated.
* Validation mechanisms (e.g., notary schemes) can ensure data integrity.

**Types of Oracles**

1. **Standard or Simple Oracles**
   * Smart contract designers can accept data from an Oracle provided by a large, reputable, trusted third party.
   * However, the issue of centralization remains.
2. **Decentralized Oracles**
   * Built using distributed mechanisms.
   * Oracles can source data from another blockchain driven by distributed consensus, ensuring data authenticity.
   * Example: Institutions running private blockchains can publish their data feed via an Oracle consumed by other blockchains.
3. **Hardware Oracles**
   * Introduced by researchers for real-world data from physical devices (e.g., IoT, telemetry).
   * Requires tamper-proof devices to ensure data integrity.

**Platforms for Oracles**

* Platforms now enable smart contracts to get external data using an Oracle.
* Methods used by Oracles depend on the type of blockchain used:
  + Example: On the Bitcoin blockchain, an Oracle can write data to a specific transaction via an **OP\_RETURN Opcode**, allowing smart contracts to monitor and read the data.
* **Examples of Oracle Services**:
  + [Oraclize.it](http://www.oraclize.it/)
  + [RealityKeys](https://www.realitykeys.com/)
  + [SmartContract.com](https://smartcontract.com/)
* Mechanisms like **TLSnotary** can be used to prove the authenticity of the data retrieved by Oracles.
  + This ensures that the data fed back to the smart contract is genuinely retrieved from the source.



**Smart Oracles**

* Smart Oracles are basically entities just like Oracles, but with the added capability of contract code execution.
* Smart Oracles proposed by **Codius** run using **Google Native Client**, which is a sandboxed environment for running untrusted x86 native code.

**Deploying Smart Contracts on a Blockchain**

* Smart contracts may or may not be deployed on a blockchain.
* However, it makes sense to deploy them on a blockchain due to the **distributed consensus mechanism** provided by blockchain.
* **Ethereum** is an example of a blockchain that natively supports the development and deployment of smart contracts.
* Smart contracts on the Ethereum blockchain are usually part of a larger application, such as **Decentralized Autonomous Organizations (DAOs)**.

**Smart Contracts on Bitcoin Blockchain**

* In the Bitcoin blockchain, the **lock\_time field** in a Bitcoin transaction can be seen as an enabler of a basic version of a smart contract.
  + **lock\_time field** enables a transaction to be locked until a specified time or after a number of blocks, enforcing a basic contract that a certain transaction can only be unlocked if certain conditions (e.g., elapsed time or number of blocks) are met.
  + This can be viewed as a basic smart contract.
* The Bitcoin scripting language, though limited, can be used to construct basic smart contracts.

**The DAO**

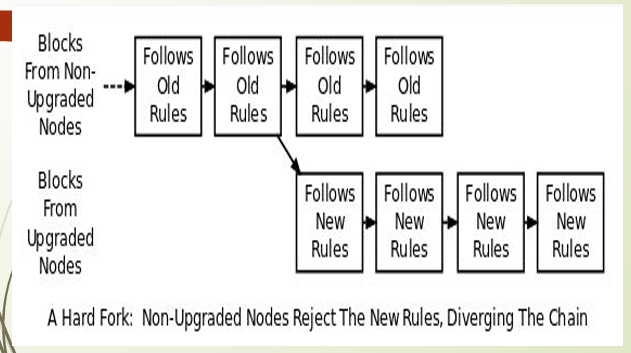
* DAO (Decentralized Autonomous Organization) is one of the highest crowdfunded projects, launched in **April 2016**.
  + It consisted of a set of smart contracts written to provide a platform for investment.
  + Due to a **bug in the code**, it was hacked in **June 2016**, and an equivalent of **$50 million** was siphoned out of the DAO into another account.
  + This incident resulted in a **hard fork** on Ethereum to recover from the attack.

**Key Insights on the DAO Incident**

* The notion of "code is law" or **unstoppable smart contracts** came under scrutiny.
* **Ethereum Foundation's Response**:
  + Introduced a hard fork to stop and change the execution of The DAO.
  + The hard fork went against the **true spirit of decentralization** and the notion of "code is law."

**Creation of Ethereum Classic**

* Resistance against the hard fork arose from some miners who continued mining the original chain.
* This resulted in the creation of **Ethereum Classic**, the original, non-forked Ethereum blockchain where "code is still law."

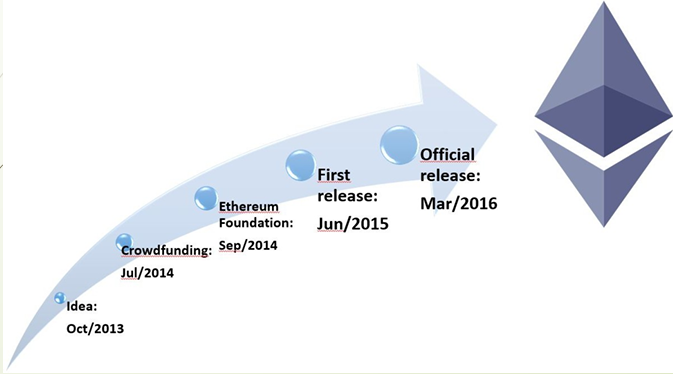


**Ethereum 101**

**Introduction**

* Ethereum was conceptualized by Vitalik Buterin in November 2013.  
  key idea proposed was the development of a Turing-complete  
  language  
  Allows the development of arbitrary programs (smart contracts)  
  for blockchain and decentralized applications.  
  This is in contrast to bitcoin,  
  where the scripting language is very limited and allows basic and  
  necessary operations only

**History of Ethereum – Timeline**



**Ethereum Clients and Releases**

**Ethereum Clients**

Various Ethereum clients have been developed using different languages.

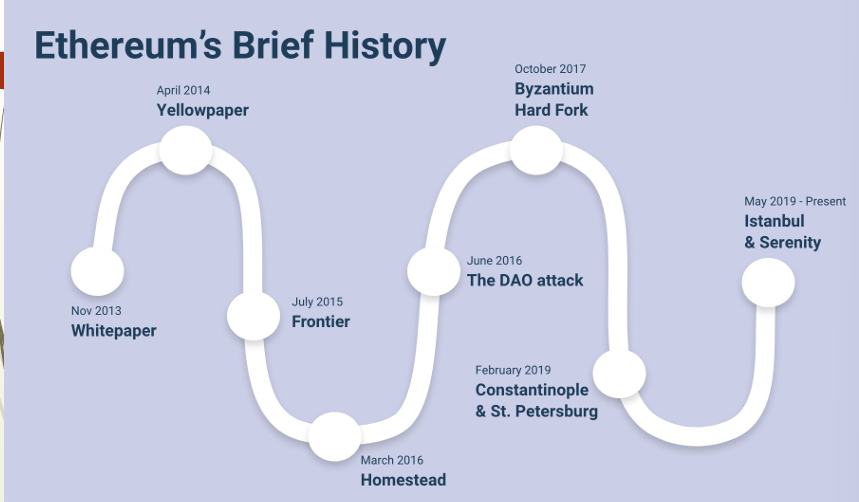
* **Most popular** are **go-Ethereum** and **Parity**.
  + go-Ethereum was developed using **Golang**.
  + Parity was built using **Rust**.

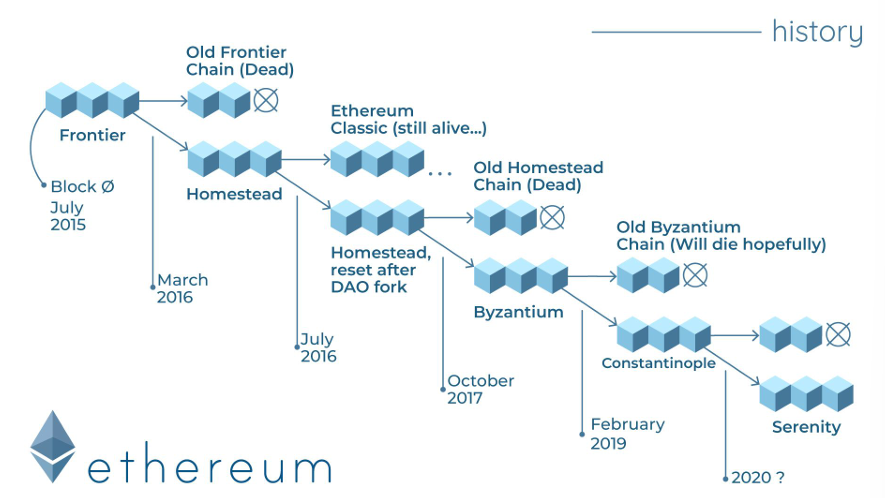
The **go-Ethereum client**, known as **geth**, is sufficient for all purposes.

* **Mist** is a user-friendly **Graphical User Interface (GUI)** wallet that runs geth in the background to sync with the network.

**Ethereum Releases**

* **Frontier**: The first release of Ethereum.
* **St. Petersburg**: The current release of Ethereum.
* **Istanbul**: The next release.
* **Serenity**:
  + Envisaged to implement a **Proof of Stake algorithm (Casper)**.
  + Targets other areas of research, including:
    - **Scalability**.
    - **Privacy**.
    - **Ethereum Virtual Machine (EVM) upgrade**.





**Future Vision**

The **Ethereum ecosystem** will undergo constant improvement and development.

* **Serenity** should not be considered a final version but a major milestone in Ethereum's continuous improvement journey.
* Further releases are planned but not yet named.

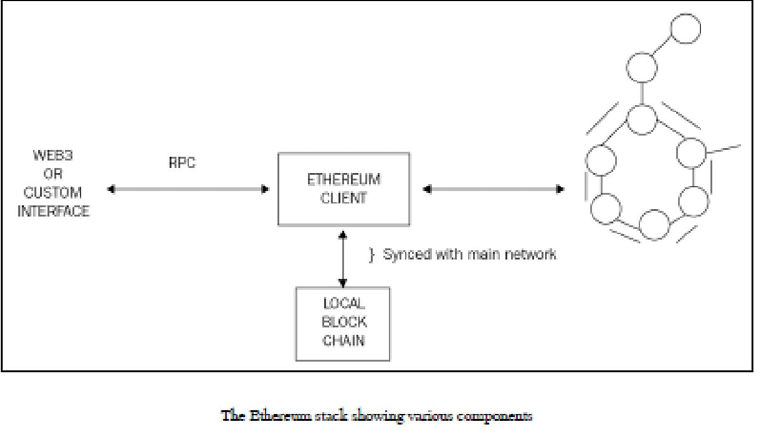
The **vision of Web 3.0** has been proposed and is actively discussed in the Ethereum community.

* Web 3.0 proposes a **semantic and intelligent web** evolving from Web 2.0 technology.
* With blockchain technology, the idea of a **decentralized web** has emerged:
  + Decentralization of major services such as **DNS**, **search engines**, and **identity**.
  + Ethereum is envisaged as a platform to help realize this vision.

**Ethereum Stack**

**Key Components**

1. **Ethereum Blockchain**:
   * At its core, Ethereum operates on the **P2P Ethereum network**.
2. **Ethereum Client (geth)**:
   * Runs on the nodes and connects to the peer-to-peer Ethereum network.
   * Downloads and stores the blockchain locally.
   * Provides functions like **mining** and **account management**.
   * Synchronizes the local blockchain copy regularly with the network.
3. **Web3.js Library**:
   * Allows interaction with **geth** via the **Remote Procedure Call (RPC)** interface.

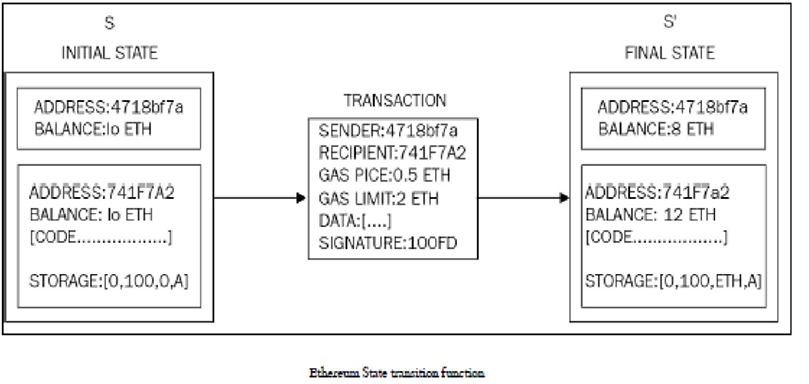


**Ethereum Blockchain**

Ethereum can be visualized as a **transaction-based state machine**:

* The **genesis state** transforms incrementally into a **final state** by executing transactions.
* This **final transformation** is accepted as the **absolute, undisputed version of the state**.

Transfer of 2 Ether from Address 4718bf7a to Address 741f7a2 is initiated



**Currency (ETH and ETC)**  
As an incentive to the miners, Ethereum also rewards its native currency called Ether, abbreviated as ETH.  
After the DAO hack a hard fork was proposed in order to mitigate the issue;  
There are now two Ethereum blockchains:

* Ethereum Classic and its currency is represented by ETC
* Hard-forked version is ETH, which continues to grow  
  ETC, has a dedicated community that is further developing ETC, which is the non-forked original version of Ethereum.

**Forks**  
With the release of homestead, due to major protocol upgrades, it resulted in a hard fork.  
Protocol was upgraded at block number 1,150,000,  
Resulting in the migration from the first version of Ethereum known as Frontier to the second version of Ethereum called Homestead.

**Forks**  
Recent unintentional fork that occurred on November 24, 2016, at 14:12:07 UTC was due to a bug in the geth client's journaling mechanism.  
Network fork occurred at block number 2,686,351.  
This bug resulted in geth failing to revert empty account deletions in the case of the empty out-of-gas exception.  
This means that from block number 2686351, the Ethereum blockchain is split into two, one running with parity clients and the other with geth.  
This issue was resolved with the release of geth version 1.5.3.

**Gas**  
Another key concept in Ethereum is that of gas.  
All transactions on the Ethereum blockchain are required to cover the cost of computation they are performing.  
Cost is covered by something called gas or crypto fuel,  
Gas as execution fee is paid upfront by the transaction originators.  
Fuel is consumed with each operation.

**Gas**  
Each operation has a predefined amount of gas associated with it.  
Each transaction specifies the amount of gas it is willing to consume for its execution.  
If it runs out of gas before the execution is completed, any operation performed by the transaction up to that point is rolled back.  
If the transaction is successfully executed, then any remaining gas is refunded to the transaction originator.

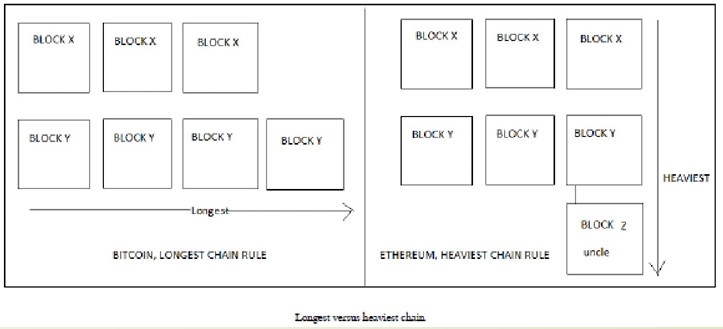
**The consensus mechanism**  
Consensus mechanism in Ethereum is based on the GHOST protocol originally proposed by Zohar and Sompolinsky in December 2013  
Ethereum uses a simpler version of GHOST protocol,  
where the chain that has most computational effort spent on it in order to build it is identified as the definite version.  
Another way is to find the longest chain, as the longest chain must have been built by consuming adequate mining effort.

**The consensus mechanism**  
Greedy Heaviest Observed Subtree (GHOST) was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to stale or orphan blocks.  
**Stale blocks**:  
Blocks that were propagated to the network and verified by some nodes as being correct but eventually being cast off as a longer chain achieved dominance, or Forking  
Orphan, or stale block, is created when two nodes find a block at the same time. Both say we’ve found the solution to this block and send off their block to be verified and included in others block chains.  
In GHOST, stale blocks are added in calculations to figure out the longest and heaviest chain of blocks.  
Stale blocks are called Uncles or Ommers in Ethereum.

**The consensus mechanism**  
GHOST includes stale blocks – or Uncles  
These are included in the calculation of which chain is longest or has the highest cumulative difficulty.  
Centralisation is solved by giving block rewards to stales of 87.5%  
Nephew (child of the Uncle block) also receives a reward of 12.5% of the block reward.

**The consensus mechanism**  
The Ethereum version of Ghost only goes down seven levels – or back seven levels in the height of the block chain.  
A block must specify its parents and its number of Uncles.  
An Uncle included in a block must be a direct child of the new block and less than seven blocks below it in terms of height  
It cannot be the direct ancestor of the block being formed.  
An Uncle must have a valid block header.  
An Uncle must be different from all other Uncles in previous blocks and the block being formed.  
For every Uncle included in the block the miner gets an additional 3.125% of a standard block reward.  
Miner of the Uncle receives 93.75% of a standard block reward.

* Blocks with less difficulty add less work
* Two blocks in the same difficulty period always add the same amount of work to the chain



**The world state**  
World state in Ethereum represents the global state of the Ethereum blockchain.  
Mapping between Ethereum addresses and account states.  
Addresses are 20 bytes long.  
This mapping is a data structure that is serialized using Recursive Length Prefix (RLP).  
RLP is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage, transmission over the network, and save the state in a Patricia tree.  
RLP function takes an item as an input, which can be a string or a list of items, and produces raw bytes that are suitable for storage and transmission over the network.  
RLP does not encode data; instead, its main purpose is to encode structures.

**Account state**  
Account state consists of four fields:

* nonce
* balance
* storageroot
* codehash

**Account state**  
**Nonce**

* Value that is incremented every time a transaction is sent from the address.
* In case of contract accounts, it represents the number of contracts created by the account.

**Balance**

* Contract accounts are one of the two types of accounts that exist in Ethereum.
* Represents the number of Weis, which is the smallest unit of the currency (Ether) in Ethereum held by the address.

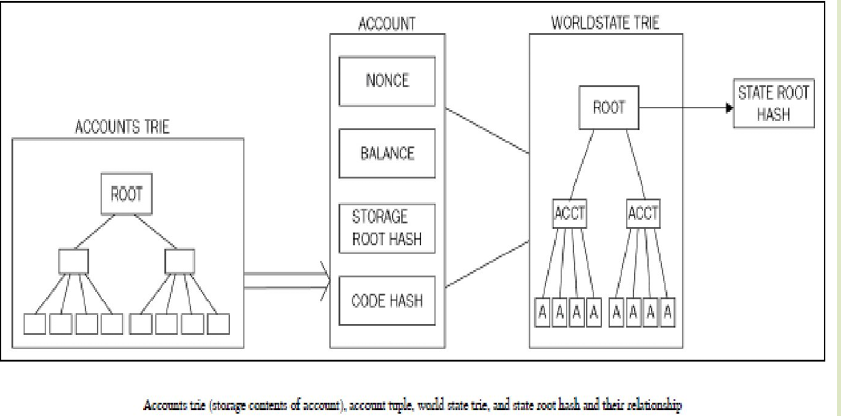
**The account state**  
**Storageroot**

* Represents the root node of a Merkle Patricia tree that encodes the storage contents of the account.

**Codehash**

* This is an immutable field that contains the hash of the smart contract code that is associated with the account.
* In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string.
* This code is invoked via a message call.

**World state and its relationship with accounts trie, accounts, and block header**



**World state and its relationship with accounts trie, accounts, and block header**  
It shows the account data structure in the middle of the diagram, which contains a storage root hash derived from the root node of the account storage trie shown on the left.  
Account data structure is then used in the world state trie, which is a mapping between addresses and account states.  
Finally, the root node of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the block header data structure, which is shown on the right-hand side of the diagram as state root hash.

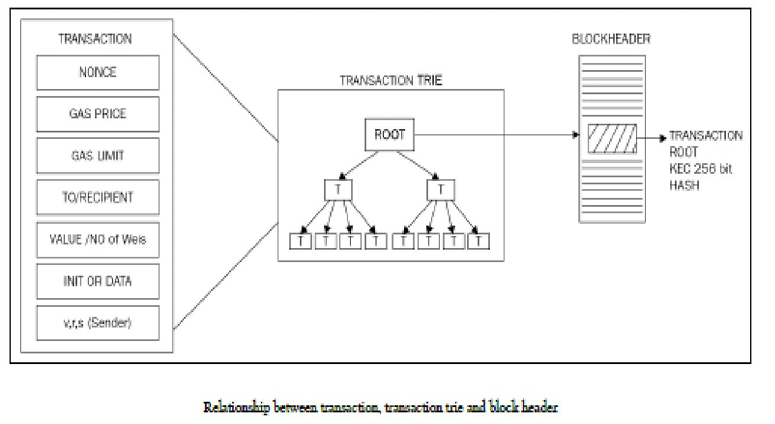
**World state and its relationship with accounts trie, accounts, and block header**  
Accounts trie is basically a Merkle Patricia tree used to encode the storage contents of an account.  
Contents are stored as a mapping between Keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

**Transactions**  
Transaction in Ethereum is a digitally signed data packet using a private key that contains instructions that, when completed, result in a message call or contract creation.  
Transactions can be divided into two types based on the output they produce:

* **Message call transactions:**  
  Transaction simply produces a message call that is used to pass messages from one account to another.
* **Contract creation transactions:**  
  Transactions result in the creation of a new contract.  
  When this transaction is executed successfully, it creates an account with the associated code.

**Transactions**  
Both transactions are composed of a number of common fields:

* **Nonce**  
  Nonce is a number that is incremented by one every time a transaction is sent by the sender.  
  It must be equal to the number of transactions sent and is used as a unique identifier for the transaction.  
  Nonce value can only be used once.
* **gasPrice**  
  gasPrice field represents the amount of Wei required in order to execute the transaction.
* **gasLimit**  
  gasLimit field contains the value that represents the maximum amount of gas that can be consumed in order to execute the transaction.  
  This is the amount of fee in Ether that a user is willing to pay for computation.
* **To**  
  To field is a value that represents the address of the recipient of the transaction.
* **Value**  
  Value represents the total number of Wei to be transferred to the recipient.  
  In the case of a contract account, this represents the balance that the contract will hold.
* **Signature**  
  Signature is composed of three fields, namely v, r, and s.  
  Represent the digital signature (R, S) and some information that can be used to recover the public key (V).  
  Sender of the transaction can also be determined.  
  Signature is based on ECDSA scheme and makes use of the SECP256k1 curve.
* **Init**  
  Init field is used only in transactions that are intended to create contracts.  
  Byte array of unlimited length that specifies the EVM code to be used in the account initialization process.  
  Code contained in this field is executed only once, when the account is created for the first time, and gets destroyed immediately after that.  
  Init also returns another code section called body, which persists and runs in response to message calls that the contract account may receive.  
  These message calls may be sent via a transaction or an internal code execution.
* **Data**  
  If the transaction is a message call, then the data field is used instead of init, which represents the input data of the message call.  
  It is also unlimited in size and is organized as a byte array.



**Transactions**  
Transaction is a tuple of the fields, which is then included in a transaction trie (modified Merkle-Patricia tree).  
Root node of transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block.  
Transactions can be found in either transaction pools or blocks.  
When a mining node starts its operation of verifying blocks, it starts with the highest paying transactions in the transaction pool and executes them one by one.  
When the gas limit is reached or no more transactions are left to be processed in the transaction pool, the mining starts.

**Transactions**  
In the mining process, the block is repeatedly hashed until a valid nonce is found that, once hashed with the block, results in a value less than the difficulty target.  
Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network.  
Similar to Bitcoin's mining process.  
The only difference is that Ethereum's Proof of Work algorithm is ASIC-resistant, known as Ethash, where finding a nonce requires large memory.

**Contract creation transaction**  
There are a few essential parameters that are required when creating an account:

**Parameters:**

* Sender: Original transactor
* Available gas
* Gas price
* Endowment: Amount (ether) allocated initially
* Byte array of arbitrary length
* Initialization EVM code
* Current depth of the message call/contract-creation stack

*Current depth means the number of items that are already there in the stack.*

**Contract creation transaction**  
Addresses generated as a result of contract creation transactions are 160-bit in length.  
Rightmost 160-bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce.  
Initially, nonce in the account is set to zero.  
Balance of the account is set to the value passed to the contract.  
Storage is also set to empty.  
Code hash is Keccak 256-bit hash of the empty string.

**Contract creation transaction**  
Account is initialized when the EVM code (Initialization EVM code) is executed.  
If there is an exception during code execution, such as not having enough gas, the state does not change.  
If the execution is successful, the account is created after the payment of appropriate gas costs.

The result of a contract transaction is either:

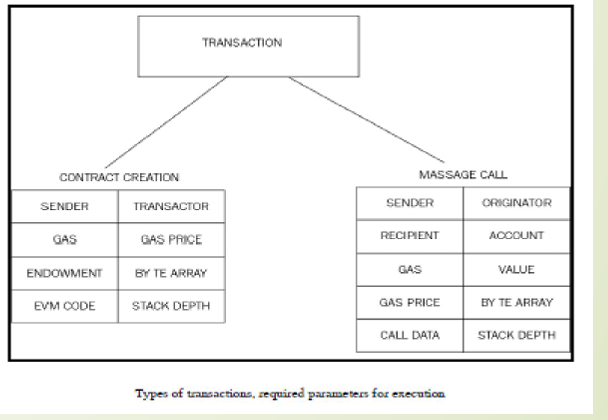
* A new contract with its balance
* No new contract is created with no transfer of value.

**Message call transaction**  
Message call requires several parameters for execution:

* Sender: Transaction originator
* Recipient: Account whose code is to be executed
* Available gas
* Value
* Gas price
* Arbitrary length byte array: Input data of the call
* Current depth of the message call/contract creation stack

**Message call transaction**  
Message calls result in state transition.  
Message calls also produce output data, which is not used if transactions are executed.  
If the message calls are triggered by VM code, the output produced by the transaction execution is used.

**Two types of transaction**



**Elements of the Ethereum blockchain - Ethereum virtual machine (EVM)**  
EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another.  
The word size of the virtual machine is set to 256-bit.  
Stack size is limited to 1024 elements and is based on the LIFO (Last In First Out) queue.  
EVM is a Turing-complete machine but is limited by the amount of gas required to run any instruction.  
Infinite loops that can result in denial-of-service attacks are not possible due to gas requirements.  
EVM also supports exception handling. In case exceptions occur, such as not having enough gas or invalid instructions, the machine would immediately halt and return the error to the executing agent.

**Ethereum virtual machine (EVM)**  
EVM is a fully isolated and sandboxed runtime environment.  
Code that runs on the EVM does not have access to any external resources, such as a network or file system.  
EVM is a stack-based architecture.  
EVM is big-endian by design, and it uses 256-bit wide words.  
The word size allows for Keccak 256-bit hash and elliptic curve cryptography computations.

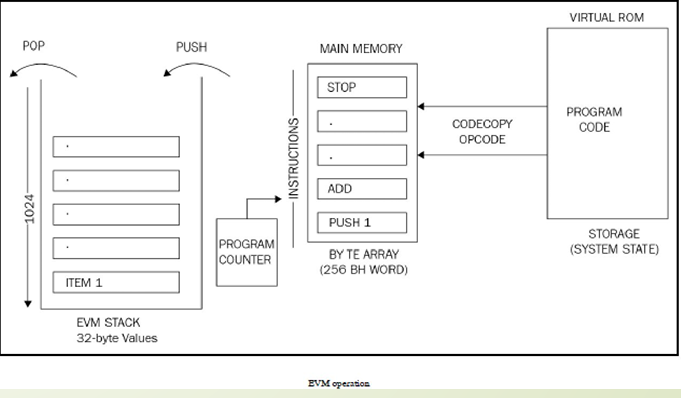
Two types of storage are available to contracts and the EVM:

1. **Memory**: A byte array.
   * When a contract finishes the code execution, the memory is cleared.
2. **Storage**: Permanently stored on the blockchain.
   * It is a key-value store.

**Ethereum virtual machine (EVM)**  
Memory is unlimited but constrained by gas fee requirements.  
Storage associated with the virtual machine is a word array that is nonvolatile and is maintained as part of the system state.  
Keys and values are 32 bytes in size in storage.

Program code is stored in a virtual read-only memory (virtual ROM) that is accessible using the CODECOPY instruction.  
The CODECOPY instruction is used to copy the program code into the main memory.

Initially, all storage and memory are set to zero in the EVM.



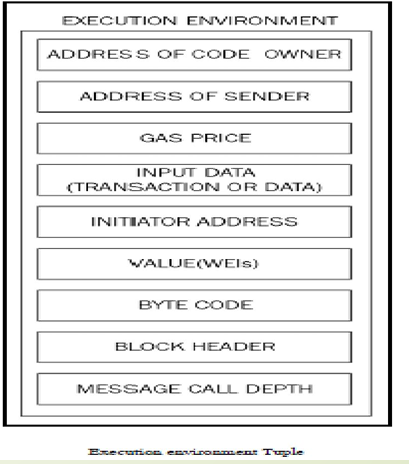
**Ethereum virtual machine (EVM)**  
Virtual ROM stores the program code that is copied into main memory using the CODECOPY instruction.  
Main memory is then read by the EVM by referring to the program counter and executes instructions step by step.  
The program counter and EVM stack are updated accordingly with each instruction execution.  
EVM optimization is an active area of research to achieve high performance.

**Execution environment**  
Key parameters are provided by the execution agent:

* **Address of the account** that owns the executing code.
* **Address of the sender** of the transaction and the originating address of this execution.
* **Gas price** in the transaction that initiated the execution.
* **Input data or transaction data** depending on the type of executing agent.
  + In the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.

**Execution environment - key parameters**

* **Address of the account** that initiated the code execution or transaction sender.
* **Value or transaction value**: Amount in Wei.
  + If the execution agent is a transaction, then it is the transaction value.
* **Code to be executed**: Presented as a byte array that the iterator function picks up in each execution cycle.
* **Block header** of the current block.
* **Number of message calls or contract creation transactions** currently in execution.
  + Number of CALLs or CREATEs currently in execution.



**Execution Environment**

In addition to the nine fields, the system state and the remaining gas are also provided to the execution environment.  
Execution results in producing:

* Resulting state
* Gas remaining after the execution
* Self-destruct or suicide set
* Log series
* Any gas refunds

**Gas**

* **Halting problem (infinite loop):** The main reason for introducing Gas.
  + **Problem:** Cannot determine whether a program will run infinitely based on compiled code.
  + **Solution:** Charge a fee per computational step to limit infinite loops and prevent flawed code from executing.
* Every transaction needs to specify an estimate of the gas it will spend.
  + Essentially, it measures how much one is willing to spend on a transaction, even if it is buggy.
* **Gas Price:**
  + Current market price of a unit of Gas (in Wei).
  + Check gas price here: [ethgasstation.info](https://ethgasstation.info/)
  + Always set by the user before a transaction.
* **Gas Limit:**
  + The maximum amount of Gas a user is willing to spend.
  + Helps regulate the network's load.
* **Gas Cost:**
  + Calculated as gasLimit \* gasPrice.
  + All blocks have a **Gas Limit**, which is the maximum Gas each block can use.

**Code Execution**

* Every node contains a virtual machine, similar to Java's, called the **Ethereum Virtual Machine (EVM)**:
  + Compiles code from a high-level language to bytecode.
  + Executes smart contract code and broadcasts state changes.
* Every full node on the blockchain processes all transactions and stores the entire state.

**Machine State**

The **Machine State** is maintained internally by the EVM and updated after each execution cycle.  
An iterator function runs in the virtual machine and outputs the results of a single state cycle.

**Machine State is a tuple consisting of the following elements:**

* **Available gas**
* **Program counter:** A positive integer up to 256.
* **Memory contents**
* **Active number of words in memory**
* **Contents of the stack**

**Machine State Exceptions**

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following occur:

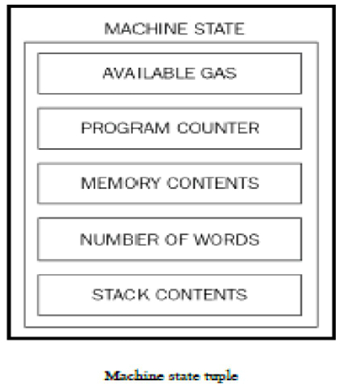
* Not enough gas required for execution.
* Invalid instructions.
* Insufficient stack items.
* Invalid destination of jump opcodes.
* Invalid stack size (greater than 1024).

**The Iterator Function**

The iterator function performs several critical tasks to set the next state of the machine and eventually the world state:

* Fetches the next instruction from a byte array where the machine code is stored in the execution environment.
* Adds/removes items from the stack using PUSH/POP instructions.
* Reduces gas according to the gas cost of the instructions/opcodes.
* Increments the program counter (PC).

**Machine state can be viewed as a tuple**



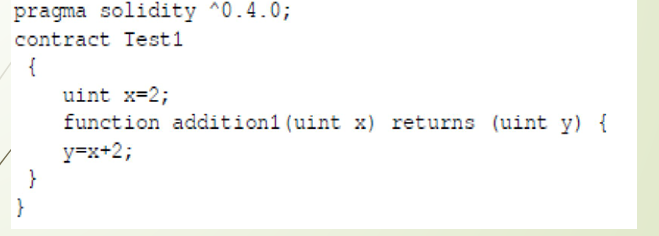
**Machine State**

* Machine state can be viewed as a tuple.
* The virtual machine is also capable of halting under normal conditions if the **STOP**, **SUICIDE**, or **RETURN** opcodes are encountered during the execution cycle.

**High-Level Languages for Ethereum**

* Code written in high-level languages such as **Serpent**, **LLL**, or **Solidity** is converted into bytecode that the EVM can understand for execution.
* **Solidity**:
  + A high-level language specifically developed for Ethereum.
  + It uses JavaScript-like syntax for writing smart contract code.
  + Once written, Solidity code is compiled into bytecode using the Solidity compiler, **solc**, which is understood by the EVM.
* **LLL**:
  + Stands for **Lisp-like Low-level language**.
  + It is another language used for writing smart contract code.
* **Serpent**:
  + A Python-like high-level language for writing Ethereum smart contracts.

**simple program in solidity**

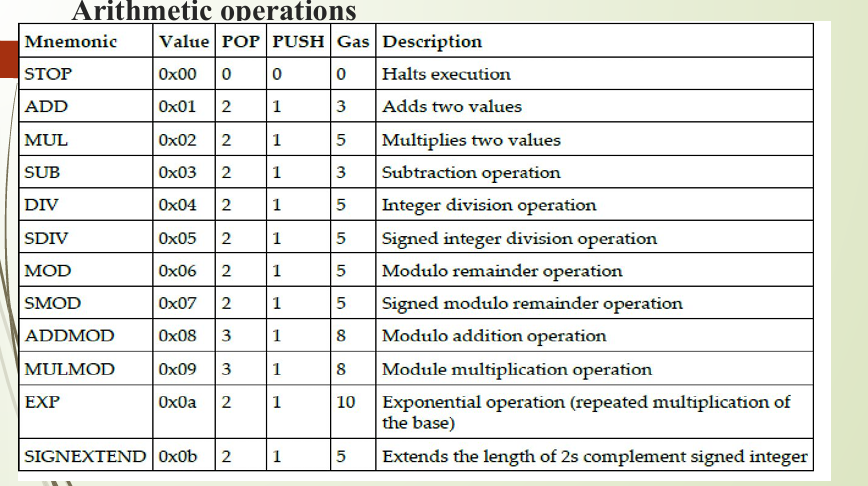


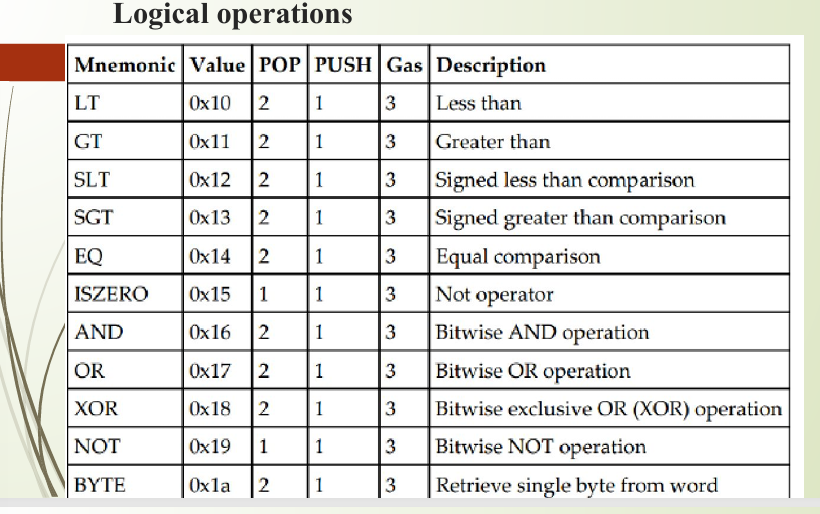
**Bytecode Conversion**

* Smart contract programs are converted into bytecode using tools like **Remix**, an online IDE for Ethereum development:
  + [Remix Ethereum IDE](https://remix.ethereum.org/)

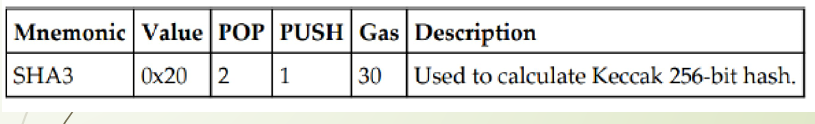
**Opcodes and Their Meaning**

* The Ethereum Virtual Machine (EVM) introduces **opcodes**, which represent the low-level instructions that the EVM executes.
* **Opcodes** are divided into multiple categories based on the operations they perform.

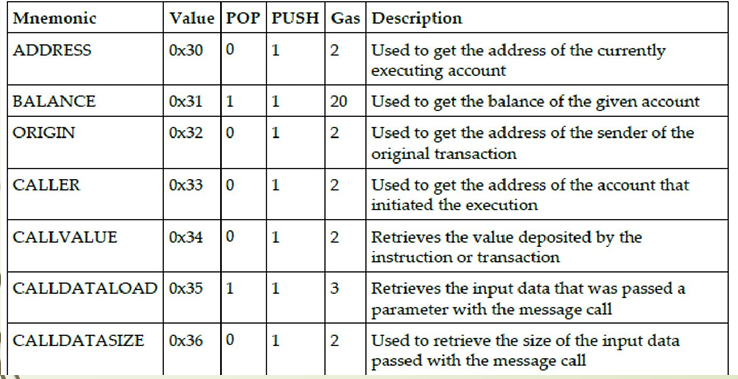




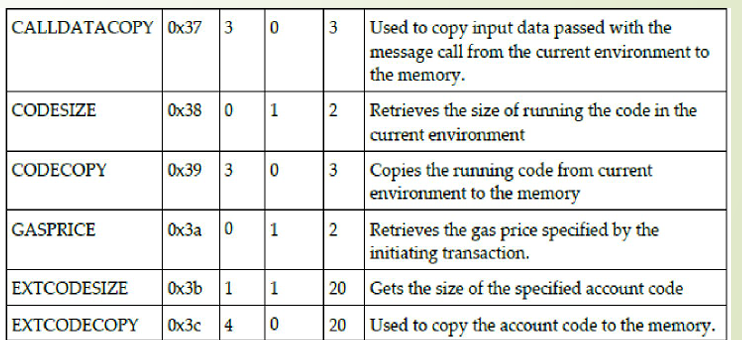
**Cryptographic operations**

****

**Environmental information**

****

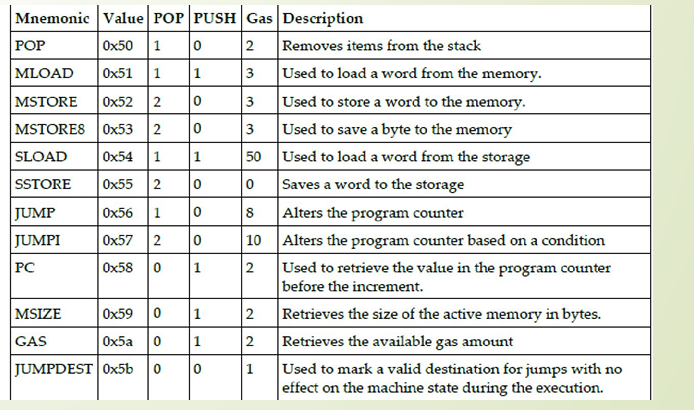
**Environmental information**

****

**Block Information**

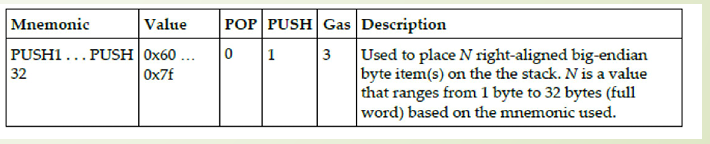
****

**Stack, memory, storage and flow operations**

****

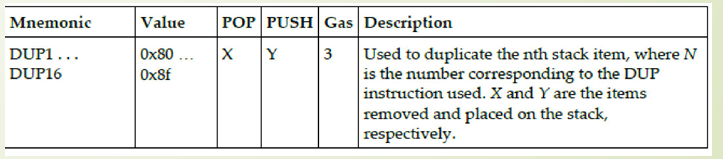
Push Operations

* PUSH operations are used to place items on the stack in the Ethereum Virtual Machine (EVM).
* The range of these instructions is from 0x60 to 0x7f.
* There are a total of 32 PUSH operations available in the EVM.
* The PUSH operation reads data from the byte array of the program code and places it on the stack.



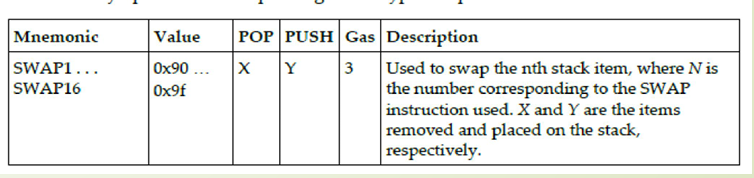
**Duplication Operations (DUP)**

* **Duplication operations (DUP)** are used to duplicate items from the stack in the Ethereum Virtual Machine (EVM).
* The range of **DUP operations** is from **0x80 to 0x8f**.
* There are a total of **16 DUP instructions** available in the EVM.
* **Stack behavior**:
  + Items are **removed from the stack** and **duplicated** according to the instruction's mnemonic.
  + For example:
    - **DUP1** removes 1 item from the stack and places 2 items on the stack.
    - **DUP16** removes 16 items from the stack and places 17 items on the stack.



**Exchange Operations (SWAP)**

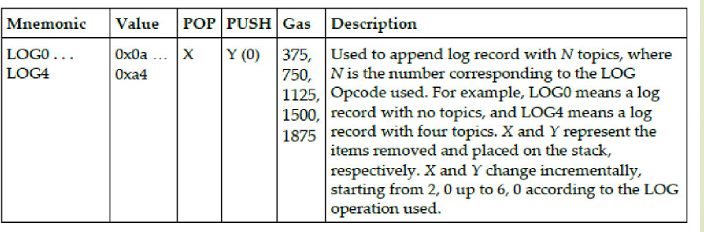
* **SWAP operations** are used to **exchange** stack items in the Ethereum Virtual Machine (EVM).
* The range of **SWAP operations** is from **0x90 to 0x9f**.
* There are a total of **16 SWAP instructions** available in the EVM.
* **Stack behavior**:
  + These operations **swap** the top stack item with another item located at a specific position in the stack.
  + For example:
    - **SWAP1** exchanges the top item of the stack with the item 1 position below.
    - **SWAP16** exchanges the top item of the stack with the item in the 16th position below.



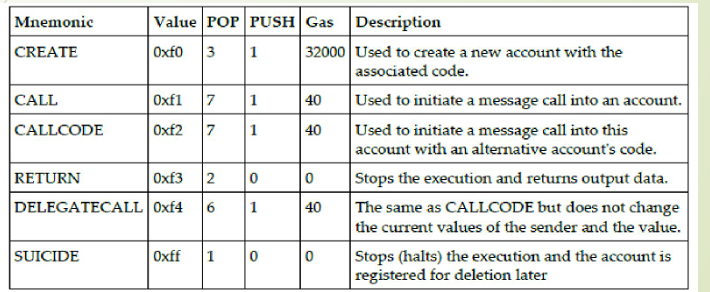
**Logging Operations**

Logging operations in the Ethereum Virtual Machine (EVM) allow the contract to **append log entries** to the **log series** field in the sub-state tuple. These logs can be useful for tracking contract events, providing transparency, and interacting with off-chain systems.

* **Range of Log Operations**: From **0x0a to 0xa4**.
* **Total Log Operations**: **Four operations** in total.



**System operations**  
System operations are used to perform various system-related operations, such as account creation, message calling, and execution control.



**Accounts**  
Accounts are one of the main building blocks of the Ethereum blockchain. State is created or updated as a result of the interaction between accounts. Operations performed between and on the accounts represent state transitions. State transition is achieved using Ethereum state transition function.

**Ethereum state transition function-Works as follows**

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.
2. Transaction fee is calculated and the sending address is resolved using the signature.
3. Sender's account balance is checked and subtracted accordingly, and nonce is incremented.
4. Error is returned if the account balance is not enough.
5. Provide enough ether (gas price) to cover the cost of the transaction. Charged per byte incrementally according to the size of the transaction.

**Ethereum state transition function-Works as follows**  
In the next step, the actual transfer of value occurs. Flow is from the sender's account to the receiver's account. Account is created automatically if the destination account specified in the transaction does not exist yet. If the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs out of gas.

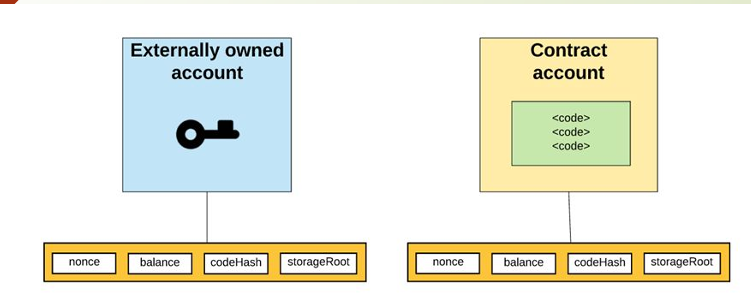
**Ethereum state transition function-Works as follows**  
In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back with the exception of fee payment, which is paid to the miners. Finally, the remainder of the fee is sent back to the sender as change, and the fee is paid to the miners accordingly. At this point, the function returns the resulting state.

**Types of accounts**

* **Externally owned accounts (EOAs)**  
  EOAs are similar to accounts that are controlled by a private key in Bitcoin.
* **Contract accounts**  
  Contract accounts are the accounts that have code associated with them along with the private key.

**Types of Account**

* **EOA** has ether balance, is able to send transactions, and has no associated code.
* **Contract Account (CA)** has ether balance, associated code, and the ability to get triggered and execute code in response to a transaction or a message. Code within contract accounts can be of any level of complexity. Code is executed by the EVM by each mining node on the Ethereum network. Contract accounts are able to maintain their own permanent state and can call other contracts. Serenity release - Distinction between externally owned accounts and contract accounts may be eliminated.



**External Account (EOA, Valid Ethereum Address)**

* Has an associated nonce (amount of transactions sent from the account) and a balance.
* **codeHash**: Hash of associated account code, i.e., a computer program for a smart contract (hash of an empty string for external accounts, EOAs).
* **Storage Root**: Root hash of Merkle-Patricia trie of associated account data.

**Contract Account**

* Ethereum accounts can store and execute code.
* Has an associated nonce and balance.
* **codeHash**: Hash of associated account code.
* **storageRoot**: Contains Merkle tree of associated storage data.

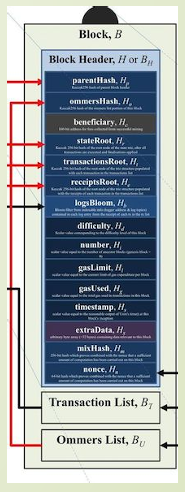
**Block**  
Blocks are the main building blocks of a blockchain. Ethereum blocks consist of various components:

* **Block header**
* **Transactions list**
* \*\*List of headers of Ommers or Uncles

The transaction list is simply a list of all transactions included in the block. The list of headers of Uncles is also included in the block. The most important and complex part is the block header.

**Ethereum Block**  
Blocks consist of 3 elements:

* **Transaction List**: List of all transactions included in a block.
* **Block Header**: Group of 15 elements.
* **Ommer List**: List of all Uncle blocks included.



**Uncles/Ommers**  
Sometimes valid block solutions don’t make the main chain. Any broadcast block (up to 6 previous blocks back) with valid PoW and difficulty can be included as an uncle. A maximum of two can be included per block. Uncle block transactions are not included, only the headers are included. This mechanism is aimed at decreasing centralization and rewarding work.

**Uncles/Ommers Rewards**:

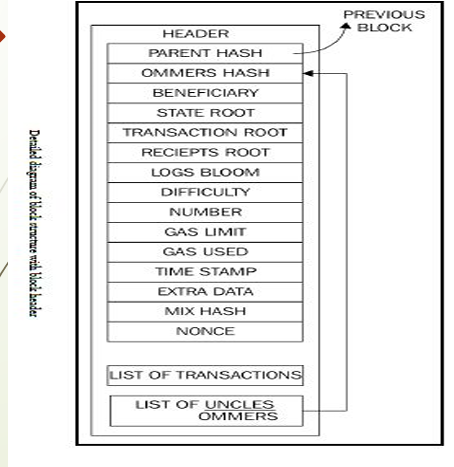
* Uncle headers can be included in the main block for 1/32 of the main block miner’s reward, which is given to the miner of the uncle block.
* Miners of uncle blocks receive a percentage of the main block reward according to the formula: (Un+(8−Bn))×58(Un + (8 - Bn)) \times \frac{5}{8}(Un+(8−Bn))×85​ Where Un is the uncle block number and Bn is the block number.  
  **Example**: (1333+8−1335)×58=3.75ETH(1333 + 8 - 1335) \times \frac{5}{8} = 3.75 ETH(1333+8−1335)×85​=3.75ETH

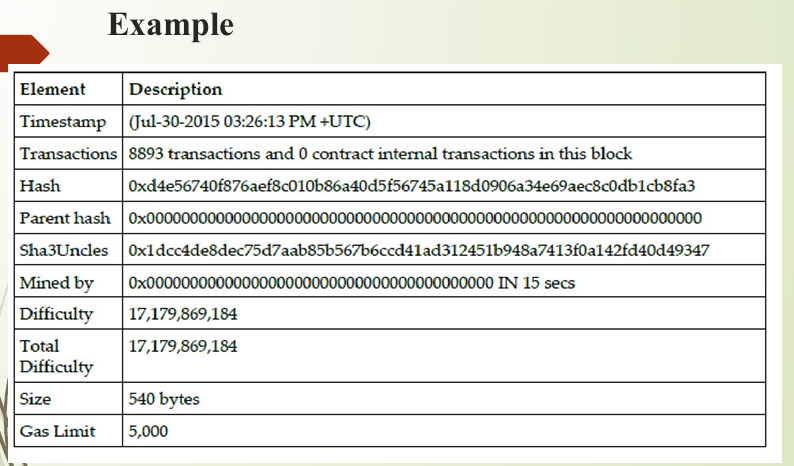
**Block Header**  
Block headers are the most critical and detailed components of an Ethereum block. The header contains valuable information, including:

* **Parent hash**: Keccak 256-bit hash of the parent (previous) block's header.
* **Ommers hash**: Keccak 256-bit hash of the list of Ommers (Uncles) blocks included in the block.
* **Beneficiary**: 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.
* **State root**: Keccak 256-bit hash of the root node of the state trie, calculated after all transactions have been processed and finalized.
* **Transactions root**: Keccak 256-bit hash of the root node of the transaction trie, representing the list of transactions included in the block.
* **Receipts root**: Keccak 256-bit hash of the root node of the transaction receipt trie. This trie contains the receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information.
* **Logs bloom**: A Bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt in the block.
* **Difficulty**: Difficulty level of the current block.
* **Number**: Total number of all previous blocks; the genesis block is block zero.
* **Gas limit**: The limit set on the gas consumption per block.
* **Gas used**: Total gas consumed by the transactions included in the block.
* **Timestamp**: Epoch Unix time of the block initialization.
* **Extra data**: Used to store arbitrary data related to the block.
* **Mixhash**: Contains a 256-bit hash that, once combined with the nonce, proves that adequate computational effort has been spent to create the block.
* **Nonce**: A 64-bit hash (a number) used to prove, in combination with the mixhash field, that adequate computational effort has been spent to create the block.

**Extra Points**

* **World state**: It is a mapping between addresses and account states.
* **Account state**: Stores the state of each of Ethereum’s accounts. It also contains the storageRoot of the account state trie, which holds the storage data for the account.
* **Transaction**: Represents a state transition in the system. It can be a funds transfer, a message call, or a contract deployment.







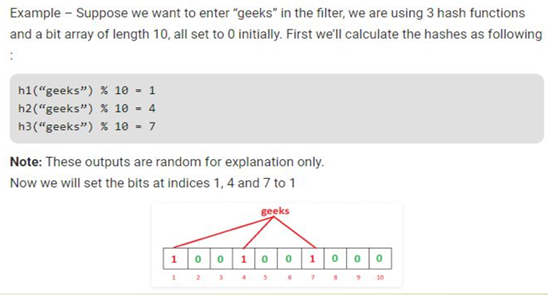
**Transaction Receipts**

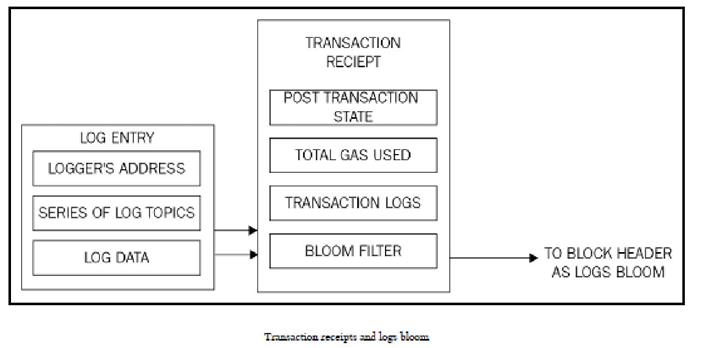
Transaction receipts are used as a mechanism to store the state after a transaction has been executed. These receipts record the outcome of the transaction execution and are produced after the execution of each transaction. All receipts are stored in an index-keyed trie, and the hash (Keccak 256-bit) of the root of this trie is placed in the block header as the receipts root. Each receipt consists of four elements:

**Four Elements of Transaction Receipts**

1. **Post-Transaction State**  
   This is the trie structure that holds the state after the transaction has been executed. It is encoded as a byte array.
2. **Gas Used**  
   Represents the total amount of gas used during the execution of the transaction. This value is taken immediately after the transaction execution is completed. The total gas used is a non-negative integer.
3. **Set of Logs**  
   This field shows the set of log entries created as a result of the transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.
4. **Bloom Filter**  
   A Bloom filter is created from the information contained in the set of logs. Each log entry is reduced to a hash of 256 bytes, which is then embedded in the header of the block as the logs bloom.
   * **Log Entry**: A log entry is composed of the logger's address, log topics, and log data.
   * **Log Topics**: These are encoded as a series of 32-byte data structures.
   * **Log Data**: This is made up of a few bytes of data.

**Bloom Filter**  
Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.  
For example, checking availability of username is set membership problem, where the set is the list of all registered username.





**Transaction Validation and Execution**

Transactions are executed after verifying the transactions for validity.  
Initial tests include:

* Transaction must be well-formed and RLP-encoded without any additional trailing bytes.
* Digital signature used to sign the transaction is valid.
* Transaction nonce must be equal to the sender's account's current nonce.
* Gas limit must not be less than the gas used by the transaction.
* Sender's account contains enough balance to cover the execution cost.

**Transaction Sub-State**

Transaction sub-state is created during the execution of the transaction and is processed immediately after the execution completes.  
Transaction sub-state is a tuple composed of three items:

**Suicide Set**

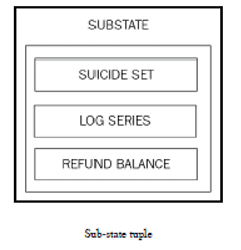
* Contains the list of accounts that are disposed of after the transaction is executed.

**Log Series**

* Indexed series of checkpoints that allow monitoring and notification of contract calls to entities external to the Ethereum environment.
* Works like a trigger mechanism executed every time a specific function is invoked or a specific event occurs.
* Logs are created in response to events occurring in the smart contract.

**Refund Balance**

* Total price of gas in the transaction that initiated the execution.
* Refunds are not immediately executed; instead, they are used to partially offset the total execution cost.



**Block Validation Mechanism**

An Ethereum block is considered valid if it passes the following checks:

* **Consistency with Uncles and Transactions**:
  + All Ommers (Uncles) satisfy the property that they are valid Uncles.
  + Proof of Work for Uncles must be valid.
* **Parent Block Validation**:
  + The previous block (parent) must exist and be valid.
* **Timestamp Validation**:
  + The current block's timestamp must be greater than the parent block's timestamp.
  + It must also be less than 15 minutes into the future.
  + All block times are calculated in epoch time.

If any of these checks fail, the block is rejected.

**Block Finalization**

Block finalization is performed by miners to validate the block's contents and apply rewards. The process involves four steps:

**1. Ommers Validation**

* Validate Ommers (Uncles).
* If mining, determine Ommers.
* Validate the headers of stale blocks to ensure they meet the maximum depth of six blocks from the current block.
* A block can include a maximum of two Uncles.

**2. State and Nonce Validation**

* Verify the state and nonce.
* If mining, compute a valid state and nonce.

**3. Transaction Validation**

* Validate all transactions.
* If mining, determine valid transactions.
* Ensure the total gas used matches the final gas consumption after processing the last transaction.

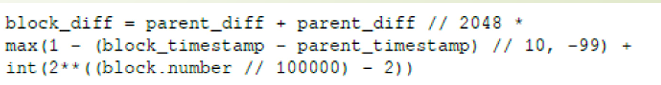
**4. Reward Application**

* Update the beneficiary's account with the reward balance.
* Miners also receive rewards for stale blocks (1/32 of the block reward).
* Uncles included in the block receive 7/8 of the total block reward.
* The current block reward is 5 Ether.

**Block Difficulty**

Block difficulty adjusts to maintain a consistent block generation time:

* If the time between two blocks decreases, the difficulty is increased.
* The difficulty adjustment algorithm was updated in Ethereum's Homestead release.



**Block Difficulty**

**Adjustment Based on Time Difference:**

* **If the time difference between the generation of the parent block and the current block is:**
  + **Less than 10 seconds: Difficulty increases.**
  + **Between 10 to 19 seconds: Difficulty remains unchanged.**
  + **20 seconds or more: Difficulty decreases, proportional to the time difference.**

**Difficulty Time Bomb ("Ice Age"):**

* **In addition to timestamp-based adjustment, the difficulty increases exponentially every 100,000 blocks.**
* **This mechanism, known as the difficulty time bomb, makes mining on the Proof-of-Work (PoW) chain increasingly difficult.**
* **The purpose is to encourage a transition to Ethereum's Proof-of-Stake (PoS) system (e.g., Casper).**
* **As block generation times grow unfeasibly long, mining on PoW becomes impractical.**

**Step-by-Step Difficulty Adjustment:**

1. **Calculate Time Difference:**
   * **Compute the difference between the parent block's timestamp and the new block's timestamp.**
2. **Divide by 10:**
   * **Divide the result by 10 and take the integer part to create ranges.**
   * **Example:**
     + **Time difference 1–9 → Output = 0.**
     + **Time difference 10–19 → Output = 1.**
     + **Time difference 20–29 → Output = 2, etc.**
3. **Create Three Ranges:**
   * **Subtract 1 from the previous result.**
   * **Output will be:**
     + **Positive when time difference is 0–9.**
     + **Zero when time difference is 10–19.**
     + **Negative when time difference is 20 or more.**
4. **Limit to -99:**
   * **If the result is less than -99, set it to -99.**
5. **Divide Parent Difficulty:**
   * **Divide the parent block's difficulty by the difficulty bound divisor (2048).**
6. **Multiply and Adjust:**
   * **Multiply the result of step 4 by step 5 to calculate the difficulty difference.**
   * **Add this to the parent difficulty.**
     + **Positive → New difficulty increases.**
     + **Negative → New difficulty decreases.**
7. **Apply Minimum Threshold:**
   * **Ensure the calculated difficulty is greater than the minimum threshold of 131,072.**
8. **Check Block Number for "The Bomb":**
   * **If the block number exceeds 200,000, apply the exponential increase:**
     + **Add 1 to the parent block number.**
     + **Divide by 100,000.**
     + **Subtract this result from 2.**
     + **Calculate the exponential increase: 2result2^{\text{result}}2result.**
9. **Add Exponential Increase:**
   * **Add the result to the difficulty calculated in step 7.**

**Ether**

* **Minted by Miners:**
  + **Reward for computational effort in securing the network by verifying transactions and blocks.**
* **Used For:**
  + **Paying for contract execution on the Ethereum Virtual Machine (EVM).**
  + **Purchasing gas, which is required for computations on the Ethereum blockchain.**
* **Transaction Fees:**
  + **Fees are charged for computations performed by the EVM.**

****

**Gas**  
Gas is required to be paid for every operation performed on the Ethereum blockchain. This mechanism ensures that infinite loops cannot cause the entire blockchain to stall due to the Turing-complete nature of the Ethereum Virtual Machine (EVM).

A transaction fee is charged as some amount of Ether and is deducted from the account balance of the transaction originator. The fee is paid to miners for including the transaction in the block. If the fee is too low, the transaction may never be picked up by miners; conversely, the higher the fee, the greater the chances that the transaction will be included in the block.

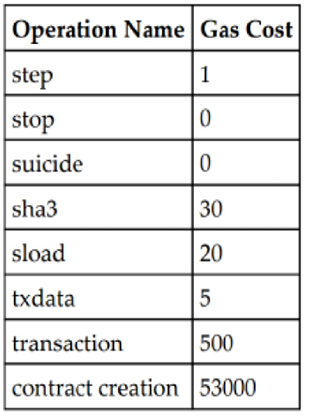
If a transaction with an appropriate fee is included in a block by miners but has too many complex operations to perform, it can result in an "out-of-gas" exception if the gas cost is insufficient. In this case, the transaction will fail but will still be recorded in the block, and the transaction originator will not receive any refund.

Transaction costs can be estimated using the following formula:  
**Total cost = gasUsed \* gasPrice**

* **gasUsed** is the total gas required for the transaction execution.
* **gasPrice** is specified by the transaction originator as an incentive to miners to include the transaction in the next block, and it is specified in Ether.

Each EVM opcode has a specific gas fee associated with it. The gas cost is an estimate because the actual gas usage may differ depending on factors like computation time or changes in the behavior of the smart contract. If the execution consumes more or less gas than originally anticipated, the actual gas used may vary.

If the transaction execution is successful and there is remaining gas, it is returned to the transaction originator.



**Example:**

The calculation of the SHA3 operation can be done as follows:

* SHA3 costs 30 gas.
* The current gas price is 25 GWei, which is equivalent to 0.000000025 Ether.

Now, multiplying both:

0.000000025×30=0.00000075 Ether0.000000025 \times 30 = 0.00000075 \, \text{Ether}0.000000025×30=0.00000075Ether

**Fee Schedule**

Gas is charged in three scenarios as a prerequisite to the execution of an operation:

* Computation of an operation
* Contract creation or message call
* Increase in the usage of memory

**Messages**

Messages are the data and value that are passed between two accounts. A message is a data packet passed between two accounts. The data packet contains data and value (amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (externally owned account) in the form of a transaction that has been digitally signed by the sender.

Contracts can send messages to other contracts. Messages only exist in the execution environment and are never stored. Messages are similar to transactions:

* Messages are produced by the contracts
* Transactions are produced by entities external (externally owned accounts)

**Components of a Message**

A message consists of the following components:

* Sender of the message
* Recipient of the message
* Amount of Wei to transfer
* Message to the contract address
* Optional data field (Input data for the contract)
* Maximum amount of gas that can be consumed

Messages are generated when CALL or DELEGATECALL opcodes are executed by the contracts.

**Calls**

A Call does not broadcast anything to the blockchain; instead, it is a local call to a contract function and runs locally on the node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. Calls are executed locally on a node and generally do not result in any state change. This is the act of passing a message from one account to another.

If the destination account has an associated EVM code, then the virtual machine will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object, then the call passes any data returned from the virtual machine operation. State is altered by transactions.

**Mining**

Mining is the process by which new currency is added to the blockchain. It incentivizes miners to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

**Functions of a Miner**

At a theoretical level, a miner performs the following functions:

* Listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed.
* Determines stale blocks called Uncles or Ommers and includes them in the block.
* Updates the account balance with the reward earned from successfully mining the block.
* Finally, a valid state is computed, and the block is finalized, which defines the result of all state transitions.

**Proof of Work and Proof of Stake**

The current method of mining is based on **Proof of Work**, which is similar to that of Bitcoin. When a block is deemed valid, it has to satisfy not only the general consistency requirements, but it must also contain the Proof of Work for a given difficulty.

Proof of Work is due to be replaced with the **Proof of Stake** algorithm with the release of Serenity. The algorithm named **Casper** has been developed to replace the existing Proof of Work algorithm in Ethereum. A security deposit, based on the economic protocol, requires nodes to place a security deposit before they can produce blocks. Nodes have been named **bonded validators** in Casper, whereas the act of placing the security deposit is named **bonding**.

**Ethash**

Ethash is the name of the Proof of Work algorithm used in Ethereum. Originally, this was proposed as the Dagger-Hashimoto algorithm. The core idea behind mining is to find a nonce that, once hashed, results in a predetermined difficulty level.

Initially, difficulty was low when Ethereum was new, and even CPU and single GPU mining were profitable to a certain extent, but that is no longer the case. Ethash is a memory-hard algorithm, which makes it difficult to be implemented on specialized hardware.

**Mining Centralization**

As in Bitcoin, **ASICs** have been developed, which have resulted in mining centralization over the years. However, memory-hard Proof of Work algorithms are one way of thwarting this threat. Ethereum implements Ethash to discourage ASIC development for mining. This algorithm requires choosing subsets of a fixed resource called **DAG** (Directed Acyclic Graph) depending on the nonce and block headers. DAG is around 2 GB in size and changes every 30,000 blocks. Mining can only start when DAG is completely generated the first time a mining node starts.

Time between every 30,000 blocks is around 5.2 days and is called an **epoch**. DAG is used as a seed by the Proof of Work algorithm called Ethash.

**Ethash Reward Scheme**

The current reward scheme is 5 Ether for successfully finding a valid nonce. In addition to receiving 5 Ether, the successful miner also receives the cost of the gas consumed within the block and an additional reward for including stale blocks (Uncles) in the block. A maximum of two Uncles are allowed per block and are rewarded 7/8 of the normal block reward.

To achieve a 12-second block time, block difficulty is adjusted at every block. Rewards are directly proportional to the miner's hash rate, which basically means how fast a miner can hash.

**Mining Setup**

Mining can be performed by simply joining the Ethereum network and running an appropriate client. The key requirement is that the node should be fully synced with the main network before mining can start.

**CPU Mining**

CPU mining is still valuable on the test network or even a private network to experiment with mining and contract deployment. Geth can be started with the mine switch in order to start mining:



CPU mining can also be started using the web3 Geth console. The Geth console can be started by issuing the following command:

|  |
| --- |
| geth attach |

After this, the miner can be started by issuing the following command, which will return true if successful, or false otherwise:

|  |
| --- |
| Miner.start(4) |

The preceding command will start the miner with four threads. To stop the miner, use the following command:

|  |
| --- |
| Miner.stop |

**GPU Mining**

GPU mining can be performed easily by running two commands:

|  |
| --- |
| geth –rpc |

Once Geth is up and running and the blockchain is fully downloaded, **Ethminer** can be run in order to start mining. Ethminer is a standalone miner that can also be used in farm mode to contribute to mining pools. It can be downloaded from the following GitHub repository: [Ethminer GitHub](https://github.com/Genoil/cpuminer-ethereum/releases).

To start GPU mining, use:

|  |
| --- |
| Ethminer -G |

**GPU Mining Requirements**

GPU mining requires an AMD or Nvidia graphics card and an applicable OpenCL SDK. For Nvidia chipsets, it can be downloaded from: [NVIDIA CUDA Downloads](https://developer.nvidia.com/cuda-downloads). For AMD chipsets, it is available at: AMD OpenCL SDK.

**Benchmarking**

CPU benchmarking:

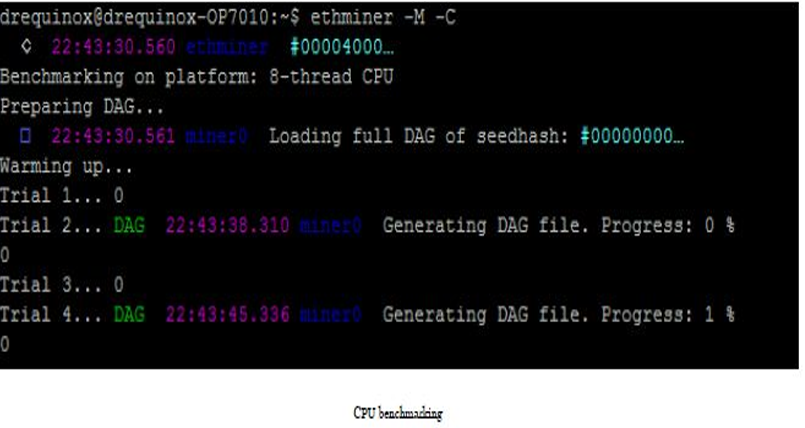
Mathematica

|  |
| --- |
| $ ethminer -M -C |

GPU benchmarking:

ruby

|  |
| --- |
| $ ethminer -M -G |



**GPU Mining**

The GPU device to be used can also be specified in the command line:

css

Copy code

$ ethminer -M -G --opencl-device 1

As GPU mining is implemented using OpenCL, AMD chipset-based GPUs tend to work faster compared to Nvidia GPUs. Due to the high memory requirements (DAG creation), **FPGAs** and **ASICs** will not provide any major advantage over GPUs, in order to discourage the development of specialized hardware for mining.

**Mining Rigs**

As difficulty increased over time for mining Ether, mining rigs with multiple GPUs were starting to be built by miners. A typical mining rig usually contains around five GPU cards, and all of them work in parallel for mining, thus improving the chances of finding valid nonces for mining.

Mining rigs can be built with some effort and are also available commercially from various vendors.

**Typical Mining Rig Configuration**

* **Motherboard**: Specialized motherboard with multiple PCI-E x1 or x16 slots.
* **SSD Hard Drive**: An SSD drive is recommended because of its much faster performance compared to the analog equivalent. It is mainly used to store the blockchain.
* **GPU**: The most important component of the rig, as it is the main workhorse used for mining. For example, it can be a Sapphire AMD Radeon R9 380 with 4 GB RAM.

**Operating System for Mining Rig**

Linux Ubuntu's latest version is usually chosen as the operating system for the rig. There is also another variant of Linux available, called **EthOS**.

Mining software such as **Ethminer** and **Geth** are installed. Remote monitoring and administration software is also installed, so that rigs can be monitored and managed remotely.

**Cooling Mechanisms**

Air conditioning or cooling mechanisms are necessary as running multiple GPUs can generate a lot of heat.

**Mining Pools**

Online mining pools that offer Ethereum mining are commonly used to increase the chances of mining rewards by pooling resources. In these pools, miners share their computational power and reward is distributed based on the contributed mining efforts.

ethminer -C –F http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc377 6075b3e60c20eb1459c@DrEquinox

**Clients and Wallets**

* **Geth**: A Go implementation of the Ethereum client.
* **Eth**: A C++ implementation of the Ethereum client.
* **Pyethapp**: A Python implementation of the Ethereum client.
* **Parity**: Built using Rust and developed by EthCore.

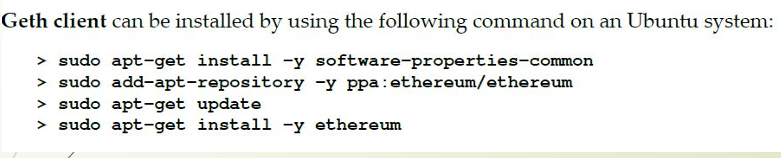
**Light Clients**

* **SPV Clients**: These clients download only a small subset of the blockchain. This allows low-resource devices, such as mobile phones, embedded devices, or tablets, to verify transactions.
* **Transaction Validation**: SPV clients validate the execution of transactions.
* **Light Clients**: SPV clients are also referred to as light clients.

**Example Wallet**

* **Jaxx Wallet**: A wallet available from Jaxx ([link to Jaxx](https://jaxx.io/)), which can be installed on iOS and Android devices. It provides SPV (Simple Payment Verification) functionality.

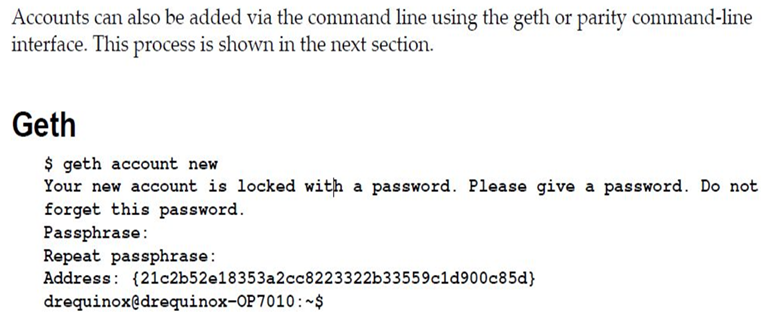
**Installation**

****

Geth can be launched simply by issuing the geth command at the command prompt

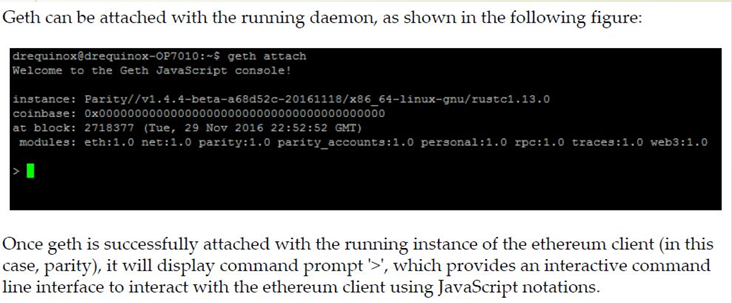
as it comes preconfigured with all the required parameters to connect to the live Ethereum network (mainnet):

> geth





The geth console geth JavaScript console can be used to perform various functions



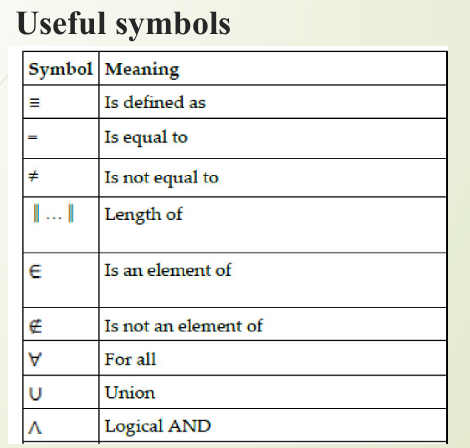
**he Yellow Paper**

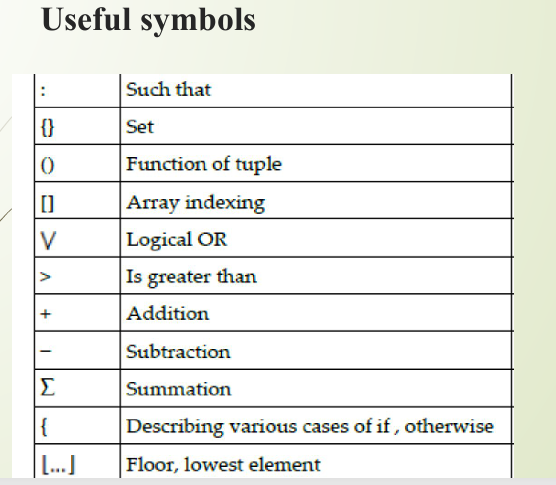
The **Ethereum Yellow Paper** was written by **Dr. Gavin Wood** and serves as a formal definition of the Ethereum protocol. It outlines the technical specifications and provides the framework for implementing an Ethereum client. Anyone can implement an Ethereum client by following the protocol specifications defined in the paper.

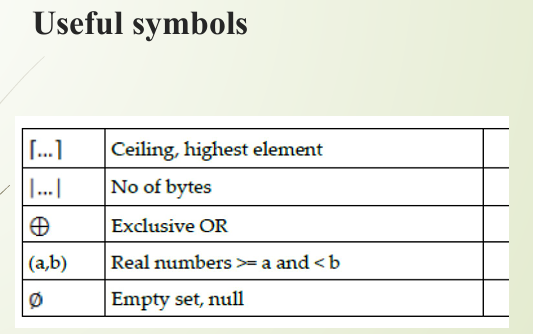
**Abstract**

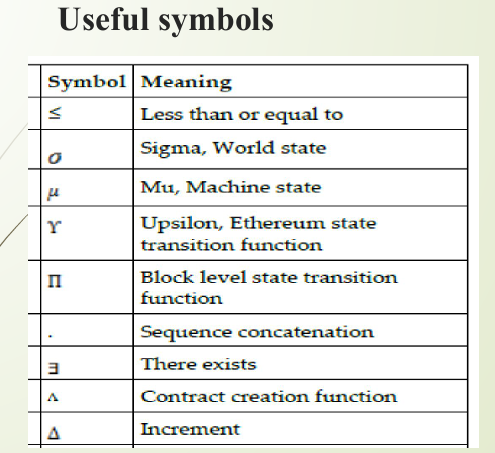
The **blockchain paradigm**, when coupled with cryptographically-secured transactions, has demonstrated its utility through various projects, with **Bitcoin** being one of the most notable examples. Each of these projects can be seen as a simple application on a decentralized, but singleton, compute resource. We can call this paradigm a **transactional singleton machine with shared-state**.

**Ethereum** implements this paradigm in a generalized manner. It provides a plurality of such resources, each with a distinct state and operating code, but able to interact with others through a message-passing framework. This paper discusses the design, implementation issues, opportunities, and the future hurdles we anticipate for Ethereum.









**The Ethereum Network**

The Ethereum network is a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism. Networks can be divided into three types based on requirements and usage:

**MainNet**

* **MainNet** is the current live network of Ethereum.
* The current version of MainNet is **Homestead**.
* **Latest release:** Muir Glacier / 1 January 2020.

**TestNet**

* **TestNet**, also called **Ropsten**, is the test network for the Ethereum blockchain.
* This blockchain is used to test smart contracts and DApps before being deployed to the production live blockchain.
* It allows experimentation and research.

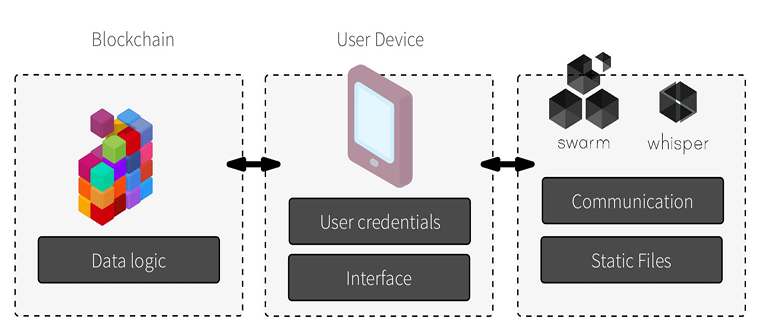
**Private Net(s)**

* A **Private Network** can be created by generating a new genesis block.
* This is usually the case in distributed ledger networks where a private group of entities starts their own blockchain and uses it as a permissioned blockchain.

**Supporting Protocols**

Various supporting protocols are developed to support the complete decentralized ecosystem:

1. **Whisper** and **Swarm** protocols.
2. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized to achieve a fully decentralized ecosystem.
   * This includes **decentralized storage** and **decentralized messaging**.
   * **Whisper** is a decentralized messaging protocol being developed for Ethereum.
   * **Swarm** is a decentralized storage protocol.

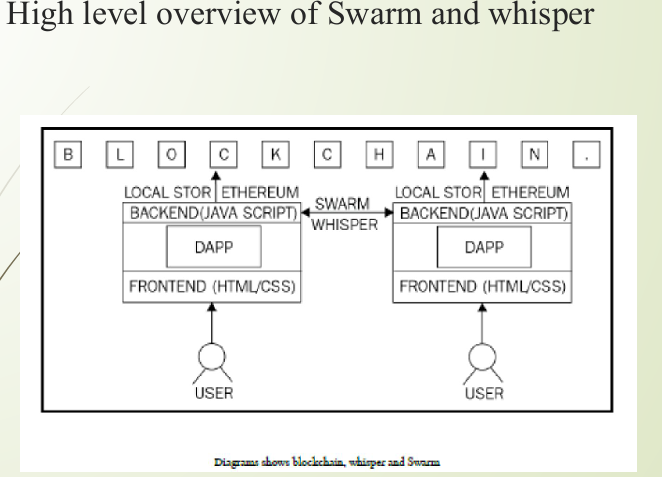


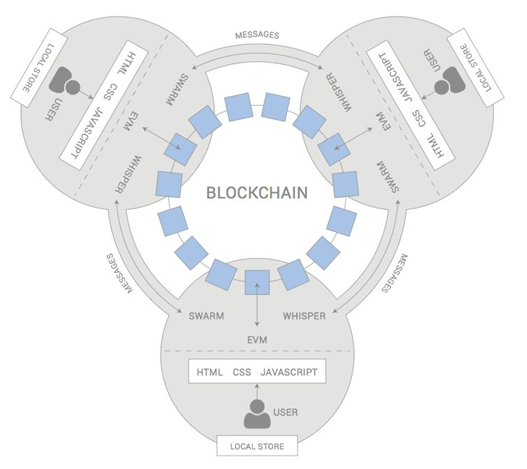
**Whisper**

* Whisper provides decentralized peer-to-peer messaging capabilities to the Ethereum network.
* It is a communication protocol that nodes use to communicate with each other.
* Data and routing of messages are encrypted within Whisper communications.
* It is used for smaller data transfers and in scenarios where real-time communication is not required.
* Whisper is also designed to provide a communication layer that cannot be traced, offering “dark communication” between parties.
* While the blockchain can be used for communication, it is expensive and consensus is not really required for messages exchanged between nodes.
* Whisper can be used as a protocol that allows decentralized communication.
* Whisper is available with Geth.

**Swarm**

* Swarm is being developed as a distributed file storage platform.
* It is a decentralized, distributed, and peer-to-peer storage network.
* Files in this network are addressed by the hash of their content. This contrasts with traditional centralized services, where storage is available only at a central location.
* Swarm is developed as a native base layer service for the Ethereum Web 3.0 stack.
* Swarm is integrated with **DevP2P**, the multiprotocol network layer of Ethereum.
* It aims to provide a DDOS (Distributed Denial of Service)-resistant and fault-tolerant distributed storage layer for Ethereum Web 3.0.





**Applications Developed on Ethereum**

* Various implementations of DAOs and smart contracts have been built on Ethereum.
* One notable example is the **DAO**, which was hacked and required a hard fork to recover the stolen funds.
  + DAO was designed to serve as a decentralized platform to collect and distribute investments.
* Another notable DApp is **Augur**, a decentralized prediction market implemented on Ethereum.

**Scalability and Security Issues**

* **Scalability**
  + Scalability is a fundamental challenge for any blockchain.
  + The Ethereum network has a current maximum capacity of **15 transactions per second**.
  + Without improvements, the industry's infrastructure will struggle to handle increasing demand.
  + Developing frameworks and scalability solutions requires significant time and effort.
  + Any proposed solution needs support from miners, developers, businesses, and other stakeholders, a process that can take months and may result in disagreements.
* **Security**
  + Security remains of paramount importance.
  + Issues like privacy and confidentiality have caused adaptability challenges, particularly in the financial sector.

**On-Chain Scaling**

* **Sharding**
  + Sharding splits a database horizontally to distribute the load.
  + It reduces network congestion and increases transactions per second by creating new chains called **shards**.
  + Validators are no longer required to process all transactions across the network, reducing their load.

**Off-Chain Scaling**

1. **Layer 2 Scaling**
   * Layer 2 solutions rely on servers or clusters, often referred to as nodes, validators, operators, or sequencers.
   * Transactions are submitted to these Layer 2 nodes instead of the main Ethereum network (Layer 1).
   * These nodes batch transactions and anchor them to Layer 1, securing them in the process.
2. **State Channels**
   * State channels use **multisig contracts** to enable participants to transact quickly off-chain while settling finality on the mainnet.
   * Benefits include reduced network congestion, lower fees, and fewer delays.
   * Two types of channels: **state channels** and **payment channels**.
3. **Sidechains**
   * Sidechains are independent, **EVM-compatible blockchains** running parallel to the Ethereum mainnet.
   * They are connected via **two-way bridges** and operate under their own rules for consensus and block parameters.
4. **Plasma**
   * Plasma chains are separate blockchains anchored to the main Ethereum chain.

**Why Are So Many Scaling Solutions Needed?**

* Multiple scaling solutions reduce congestion on specific parts of the network, avoiding single points of failure.
* These solutions can coexist and work together, exponentially improving transaction speed and throughput.
* Some scaling solutions do not rely on Ethereum's consensus algorithm, offering unique benefits.

**References**

* Imran Bashir, *Mastering Blockchain*, Packt Publishing.
* Various web materials.