Ovato - A public distributed ledger protocol for a trustless financial ecosystem

Version 0.1.0

Architect
Ovato (OVO)
Ovato Pte Ltd
architect@ovato.com

Abstract: Since the inception of blockchain technology with Bitcoin, the world has seen numerous other protocols developed based on the foundations laid by Bitcoin and a consensus protocol built on the blockchain.

Introduction:

From barter, to state issued physical and digital assets, the world of currencies has come a long way on centralized models backed by governments and pseudo governments. Until 2009, most of the currency systems heavily realised on national centralization backed by trust of the custodial government. Bitcoin radically changed that perception and with the blockchain technology made it possible to imagine a global currency system backed by consensus and absent of national or centralized agenda.

Blockchain is a mutually distributed ledger that enables the process of creating an unalterable record of transactions and assets both publicly and privately, depending on the business requirements. Assets may be of a tangible - a house, a car, cash, land; or an intangible -intellectual property, such as patents, copyrights, or branding nature.

Blockchain brings everyone to the highest accountability. The dependencies or limitations like human or machine errors or proof of a completed transaction are mitigated by using a consensus mechanism that is unbiased and solves the previously unsolved Byzantine Generals'

Motivation

Although Bitcoin solved the issue of universal trust, its creators missed out on the point of adoption for the masses. They did not see this problem through and envisioned a slow process of mainstream adoption which would occur overtime purely by word of mouth. The core team mainly concentrates on making the Bitcoin protocol more and more efficient and not spending enough resources in getting it accepted as a currency.

Bitcoin is now categorised as a “Commodity, Digital Asset or even Digital Gold “and not as a currency. This is due to the large acceptance and proven fact as a way to exchange value globally, quickly and affordably.

Ovato wishes to create an ecosystem around its blockchain with its digital asset coin. By creating a suite of partnerships covering all forms of digital commerce prevalent in today’s centralized financial society, Ovato aims at adding utility to the coin which can be redeemed via broad section of distribution partners in order to create the necessary leverage for mass adoption.

Ovato Blockchain

Peer to Peer Network:

Distributed networks are generally characterised by a master-slave architecture where a majority of nodes on the network follow a leader and inherent leader election algorithms make sure the network is never devoid of one. Such networks heavily rely on the leader selection algorithm to work properly and without a master, the network falls apart.

Ovato Blockchain has no such hierarchy. All nodes participating in the network have the same set of powers to query or set any data. The participating nodes make a mesh network by connecting with one another. Much like the internet, the Ovato Blockchain network has a flat topology. Also, the topology is independent of geography. Any sort of hierarchy which can exist is an abstract one based on the hashing power of the connecting nodes, since hashing power governs the rate at which a node will mine.
However, nodes can wish to have limited functionality depending on the purpose they are serving. Over the period of time the blockchain can grow enormously in size and it won’t be feasible for a lot of nodes to store gigabytes worth of data. The types of nodes that can exist are as follows:

1. Full Node: Has a full blockchain database, P2P Network routing. These nodes have full authority to verify transactions they receive without any external bias.

2. SPV Node: Limited part of the blockchain database stored, P2P Network Routing

3. Mining Nodes: These are kind of full nodes, but have mining software along with those features. These nodes compete in the consensus problem

The types can be expanded upon more as the ecosystem grows, but these 3 types broadly define the set of functionalities node can wish to have. The Ovato reference implementation is a mining node with wallet capabilities as well.

Any new node wanting to join the network, first has to find one valid full node with the entire blockchain database. The Ovato foundation is running a set of seed nodes which any new node would try to connect to. The connection is completed by the initial handshake. The data sent across in this handshake process is as follows:

nVersion	The P2P protocol version of the node eg (80000)
subver	sub-version running on the new node (e.g., /Satoshi:0.8.3.0/)
nTime	Timestamp at which handshake is sent
nLocalServices	Local services support given by the node, currently NODE_NETWORK
addrYou	IP address of seed node as seen by new node
addrMe	IP address of the new node
BestHeight	Number of blocks in new node's blockchain, generally 0 if fresh node

Once the new node connects with any of the seeds, it updates its version of the blockchain. Till this point the node cannot authorise or create new transactions. Once the synchronisation is completed, the node becomes a full node.

A full node broadcasts and receives data via the Gossip Protocol [Appendix A]

Block Structure:

The Ovato Blockchain, very briefly, is a linked-list based of hashed pointers, with each element on this linked list housing a merkle tree. The top most block is linked back to the block before it and this chain extends all the way till the genesis block [Appendix A]. The length of this linked list in terms of block is called as the block height which is a crucial information shared at the time of initial handshake between nodes.

A Block contains the following data fields:

Field	Size	Description
Magic Number	4 bytes	value always 0xD9B4BEF9
Block Size	4 bytes	Size of block in bytes
Block Header	80 bytes	Consists of 6 fields
Transaction Counter	1-9 bytes	Positive Integer VarInt
Transactions	Variable	Non-empty list of transactions

The meta-data stored in the block headers are described as follows:

Field	Size	Description
Version	4 bytes	Protocol version of the software
Previous Block Hash	32 bytes	Hash Pointer to previous block
Merkle Root	32 bytes	The merkle root of transaction list
Timestamp	4 bytes	Approximate Timestamp of Block Creation
Difficulty Target	4 bytes	Proof of Work difficulty target for the created block
Nonce	4 bytes	Counter for Proof of Work COnsensus Algorithm

Transaction:

Transactions carry the trail of ownership of Ovato coin from address to another. A single transaction generally references a previous transaction output. The outputs nominate an address as the next owner and a valid signature is required to unlock these funds.

These outputs are spent when the owner of the addresses is able to provide a signature signifying that they are spending the transactions creating a new transaction with inputs as output of the previous transaction.

General Format of transaction:

Field	**Description**	**Size**
Version No	currently 1	4 bytes
Flag	If present, always 0001, and indicates the presence of witness data	optional 2-byte array
In-counter	positive integer VI = VarInt	1 - 9 bytes
List of inputs	the first input of the first transaction is also called "coinbase" (its content was ignored in earlier versions)	<in-counter>-many inputs
Out counter	positive integer VI = VarInt	1 - 9 bytes
List of outputs	the outputs of the first transaction spend the mined Bitcoins for the block	<out-counter>-many outputs
Witnesses	A list of witnesses, 1 for each input, omitted if flag above is missing	
Lock_time	if non-zero and sequence numbers are < 0xFFFFFFFF: block height or timestamp when transaction is final	4 bytes

Each transaction input consists of the following fields:

Field Size	Description	Data Type	Remarks
36	previous_output	outpoint	The previous output transaction reference, as an OutPoint structure
1+	Script length	var_int	Signature Script length
?	signature script	uchar[]	Computational Script for confirming transaction authorization
4	sequence	uint32_t	Transaction version as defined by the sender. Intended for "replacement" of transactions when information is updated before inclusion into a block.

The outPoint structure consists of the following fields:

Field Size	Description	Data Type	Remarks
32	hash	char[32]	The hash of referenced transaction
4	index	uint32_t	The index of the specific output in the transaction. The first output is 0, etc.

Transaction outputs consists of the following fields

Field Size	Description	Data Type	Remarks
8	value	int64_t	Transaction Value
1+	pk_script length	var_int	Length of the pk_script
?	pk_script	uchar[]	Usually contains the public key as an Ovato script setting up conditions to claim this output.

The pk_script is basically hex encoded locking script. Script is a Turing complete language used by Ovato for transaction verification. Any entity wishing to claim the Ovato coin in the output of a transaction has to provide an unlocking script for the locking script in the transaction. [Refer Appendix D]

Generally, the transactions can be programmed in a certain way by having such locking scripts. However, for normal use cases, Ovato uses P2PKH (Pay to Public Key Hash) and P2SH (Pay to Script Hash) type of transactions for most of the financial activities.

A special type of transaction known as the coinbase transaction exists in every block. This transaction pays the miner of that particular block the block reward worth of Ovato coins. Coinbase transaction heavily resembles a normal transaction, except the input transactions fields.

Field Size	Description	Data Type	Remarks
36 bytes	previous_output	outpoint	The previous output transaction reference, as an OutPoint structure. The tx hash has all bits zero and the index all bits are one: 0xFFFFFFFF
1+	Script length	var_int	Signature Script length
1-9 bytes	Coinbase Data	var_int	Length of coinbase data from 2 to 100 bytes
4 bytes	sequence	uint32_t	Set to 0xFFFFFFFF

Each transaction is identified by a hash which is generated as follows:

Raw txBody is a big hexadecimal string.

txBody = <nVersion><flags><inCounter><transactionInputList><outCounter><transactionOutputList><witnesses><sequence>

txId = littleEndian(sha256(sha256(txBody)))

Public Key Infrastructure:

Identity on the Ovato Blockchain is linked to 160-bit hash of the public portion of a public/private ECDSA [Appendix B] keypair. This hash is base58 encoded into strings known as addresses.

These elliptic curves used for generating the public/private keypair is the **secp256k1** curve defined by the equation

$$y^2 = x^3 + 7$$

The elliptic curve domain parameters over Fp associated with a Koblitz curve secp256k1 are specified by the sextuple T = (p,a,b,G,n,h) where the finite field Fp is defined by:

- p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
  $= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- The curve E: $y^2 = x^3+ax+b$ over Fp is defined by:
  a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007
- The base point G in compressed form is:
  G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
  and in uncompressed form is:
  G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8
- Finally, the order n of G and the cofactor are:
  n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
  h = 01

The address generation algorithm is as follows:

PubKeyHash = RIPEMD160(SHA256(PubKey))

Address = Base58Check(PubKeyHash)

The private keys which are required to sign the transactions are stored in the Wallet Import Format (WIF). The WIF generation algorithm is as follows:

WIF = Base58Check(PrivKey)

The Base58check method is as follows:

ExtendedKey = Prefix | Payload
Checksum = SHA256(SHA256(ExtendedKey))
Result = Base58(ExtendedKey | Checksum)

Ovato supports BIP 32, 43, 44 [Appendix C] for mnemonic words based Hierarchical Deterministic Wallet.

APPENDIX A
Gossip Protocol

A gossip protocol is a procedure or process of computer peer-to-peer communication that is based on the way epidemics spread. Some distributed systems use peer-to-peer gossip to ensure that data is routed to all members of an ad-hoc network. Some ad-hoc networks have no central registry and the only way to spread common data is to rely on each member to pass it along to their neighbors.

The term epidemic protocol is sometimes used as a synonym for a gossip protocol, as gossip spreads information in a manner similar to the spread of a virus in a biological community.

APPENDIX B
Elliptical Curve Digital Signatures

Diffie-Hellman key exchange protocol:

A large prime **p** and a primitive root **g** of **p** are made public. Party A chooses an exponent **a** between 0 and **p** - 1 at random. Party B does the same with an exponent **b.** Party A transmits $\mathbf{g^a}$ to B, and vice-versa. Both parties agree on $\mathbf{g^{ab}}$. The security of this protocol rests on two unproven (but reasonable) assumptions:

1. Any method of obtaining $\mathbf{g^{ab}}$ from $\mathbf{g^a}$ and $\mathbf{g^b}$ would be as hard as obtaining a from $\mathbf{g^a}$ (taking “discrete logarithms”).
2. If **p** - 1 did not have only small prime factors, that finding discrete logarithms was intractible (i.e. could not run in time polynomial in log **p**).

Elliptic Curves:

An elliptic curve defined over a field **F** is the curve defined by the following equation:

$$E: y^2 = x^3 + ax + b$$

where (**a** and **b** are elements of **F** (assumed to have characteristic # 2 or 3. There is a slightly more complicated formulation in those cases.). There is a natural law of composition on the points of **E** obtained by the “tangent and chord method”: Given two points **P** and **Q**, the straight line containing them intersects the curve in a third point **R** (if **P = Q** take the tangent to the curve at **P**). Define **P + Q** as being the point $\mathbf{(x(R)_1 - y(R))}$. This provides a commutative and associative law of composition, whose zero element is the point at infinity: **(infinity, infinity)** We denote the set of points (including the point at infinity) of the curve E with coordinates in the field F by **E(F)** The discriminant of the curve $\mathbf{A = 16(4a^3 - 27b^2)}$. The elliptic curve

$$\mathbf{E_L : y^2 = x^3 + L^4 ax + L^6 b}$$

is isomorphic to the curve **E** above by the substitution

$$(x,y) \rightarrow (x/L^2 , y/L^3)$$

We say the **E** is minimal if **a** and **b** are integers, and there is no integer **L +- 1** such that $L^4 \mid a$ and $L^6 \mid b$

A number of details need to be addressed in order to make this scheme practical:

1. The actual algorithm for multiplication on an elliptic curve.

   Each point is represented by the triple **(x, y, z)** which corresponds to the point $(x/z^2, y/z^3)$. This is a homogeneous representation with **x** having weight 2,**y** having weight 3, and **z** having weight 1. If this representation is used with the recursions in the first section, then it is easily checked that the only change is in the initialization. A simple induction shows that g_{2n} has weight $4n^2 - 4$, and that g_{2n+1} has weight $4n^2 + 4n$.

2. The choice of the parameters **A** and **B** for the elliptic curve.
3. The choice of the prime modulus **p**.
4. What information needs to be transmitted.

APPENDIX C
BIPS 32, 43, 44

BIP 32:

Conventions

In the rest of this text we will assume the public key cryptography used in Bitcoin, namely elliptic curve cryptography using the field and curve parameters defined by secp256k1 (http://www.secg.org/sec2-v2.pdf). Variables below are either:

- Integers modulo the order of the curve (referred to as n).
- Coordinates of points on the curve.
- Byte sequences.
- Addition (+) of two coordinate pair is defined as application of the EC group operation. Concatenation (||) is the operation of appending one-byte sequence onto another.

As standard conversion functions, we assume:

- point(p): returns the coordinate pair resulting from EC point multiplication (repeated application of the EC group operation) of the secp256k1 base point with the integer p.
- ser32(i): serialize a 32-bit unsigned integer i as a 4-byte sequence, most significant byte first.

- ser256(p): serializes the integer p as a 32-byte sequence, most significant byte first.
- serP(P): serializes the coordinate pair P = (x,y) as a byte sequence using SEC1's compressed form: (0x02 or 0x03) || ser256(x), where the header byte depends on the parity of the omitted y coordinate.
- parse256(p): interprets a 32-byte sequence as a 256-bit number, most significant byte first.

Child key derivation (CKD) functions

Given a parent extended key and an index i, it is possible to compute the corresponding child extended key. The algorithm to do so depends on whether the child is a hardened key or not (or, equivalently, whether i ≥ 231), and whether we're talking about private or public keys

Private parent key → private child key

The function CKDpriv((kpar, cpar), i) → (ki, ci) computes a child extended private key from the parent extended private key:

- Check whether i ≥ 231 (whether the child is a hardened key).
- If so (hardened child): let I = HMAC-SHA512(Key = cpar, Data = 0x00 || ser256(kpar) || ser32(i)). (Note: The 0x00 pads the private key to make it 33 bytes long.)
- If not (normal child): let I = HMAC-SHA512(Key = cpar, Data = serP(point(kpar)) || ser32(i)).
- Split I into two 32-byte sequences, IL and IR.
- The returned child key ki is parse256(IL) + kpar (mod n).
- The returned chain code ci is IR.
- In case parse256(IL) ≥ n or ki = 0, the resulting key is invalid, and one should proceed with the next value for i. (Note: this has probability lower than 1 in 2127.)
- The HMAC-SHA512 function is specified in RFC 4231.

Public parent key → public child key

The function CKDpub((Kpar, cpar), i) → (Ki, ci) computes a child extended public key from the parent extended public key. It is only defined for non-hardened child keys.

- Check whether i ≥ 231 (whether the child is a hardened key).
- If so (hardened child): return failure
- If not (normal child): let I = HMAC-SHA512(Key = cpar, Data = serP(Kpar) || ser32(i)).

- Split I into two 32-byte sequences, IL and IR.
- The returned child key Ki is point(parse256(IL)) + Kpar.
- The returned chain code ci is IR.
- In case parse256(IL) ≥ n or Ki is the point at infinity, the resulting key is invalid, and one should proceed with the next value for i.

Private parent key → public child key

The function N((k, c)) → (K, c) computes the extended public key corresponding to an extended private key (the "neutered" version, as it removes the ability to sign transactions).

- The returned key K is point(k).
- The returned chain code c is just the passed chain code.
- To compute the public child key of a parent private key:
- N(CKDpriv((kpar, cpar), i)) (works always).
- CKDpub(N(kpar, cpar), i) (works only for non-hardened child keys).
- The fact that they are equivalent is what makes non-hardened keys useful (one can derive child public keys of a given parent key without knowing any private key), and also what distinguishes them from hardened keys. The reason for not always using non-hardened keys (which are more useful) is security; see further for more information.

Public parent key → private child key

This is not possible.

The next step is cascading several CKD constructions to build a tree. We start with one root, the master extended key m. By evaluating CKDpriv(m,i) for several values of i, we get a number of level-1 derived nodes. As each of these is again an extended key, CKDpriv can be applied to those as well.

To shorten notation, we will write CKDpriv(CKDpriv(CKDpriv(m,3H),2),5) as m/3H/2/5.
Equivalently for public keys, we write CKDpub(CKDpub(CKDpub(M,3),2),5) as M/3/2/5. This results in the following identities:

N(m/a/b/c) = N(m/a/b)/c = N(m/a)/b/c = N(m)/a/b/c = M/a/b/c.
N(m/aH/b/c) = N(m/aH/b)/c = N(m/aH)/b/c.

However, N(m/aH) cannot be rewritten as N(m)/aH, as the latter is not possible. Each leaf node in the tree corresponds to an actual key, while the internal nodes correspond to the collections of keys that descend from them. The chain codes of the leaf nodes are ignored, and only their embedded private or public key is relevant. Because of this construction, knowing an extended

private key allows reconstruction of all descendant private keys and public keys, and knowing an extended public key allows reconstruction of all descendant non-hardened public keys.

Serialization format

Extended public and private keys are serialized as follows:

- 4 bytes: version bytes (mainnet: 0x0488B21E public, 0x0488ADE4 private; testnet: 0x043587CF public, 0x04358394 private)
- 1 byte: depth: 0x00 for master nodes, 0x01 for level-1 derived keys, ...
- 4 bytes: the fingerprint of the parent's key (0x00000000 if master key)
- 4 bytes: child number. This is ser32(i) for i in xi = xpar/i, with xi the key being serialized. (0x00000000 if master key)
- 32 bytes: the chain code
- 33 bytes: the public key or private key data (serP(K) for public keys, 0x00 || ser256(k) for private keys)

This 78-byte structure can be encoded like other Bitcoin data in Base58, by first adding 32 checksum bits (derived from the double SHA-256 checksum), and then converting to the Base58 representation. This results in a Base58-encoded string of up to 112 characters. Because of the choice of the version bytes, the Base58 representation will start with "xprv" or "xpub" on mainnet, "tprv" or "tpub" on testnet.

Note that the fingerprint of the parent only serves as a fast way to detect parent and child nodes in software, and software must be willing to deal with collisions. Internally, the full 160-bit identifier could be used.

Master key generation

The total number of possible extended keypairs is almost 2512, but the produced keys are only 256 bits long, and offer about half of that in terms of security. Therefore, master keys are not generated directly, but instead from a potentially short seed value.

- Generate a seed byte sequence S of a chosen length (between 128 and 512 bits; 256 bits is advised) from a (P)RNG.
- Calculate I = HMAC-SHA512(Key = "Bitcoin seed", Data = S)
- Split I into two 32-byte sequences, IL and IR.
- Use parse256(IL) as master secret key, and IR as master chain code. In case IL is 0 or ≥n, the master key is invalid.

BIP 43:

We propose the first level of BIP32 tree structure to be used as "purpose". This purpose determines the further structure beneath this node.

m / purpose' / *

Apostrophe indicates that BIP32 hardened derivation is used.

We encourage different schemes to apply for assigning a separate BIP number and use the same number for purpose field, so addresses won't be generated from overlapping BIP32 spaces.

Example: Scheme described in BIP44 should use 44' (or 0x8000002C) as purpose.

Note that m / 0' / * is already taken by BIP32 (default account), which preceded this BIP.

Not all wallets may want to support the full range of features and possibilities described in these BIPs. Instead of choosing arbitrary subset of defined features and calling themselves BIPxx compatible, we suggest that software which needs only a limited structure should describe such structure in another BIP and use different "purpose" value.

BIP 44:

This BIP defines a logical hierarchy for deterministic wallets based on an algorithm described in BIP-0032 (BIP32 from now on) and purpose scheme described in BIP-0043 (BIP43 from now on).

This BIP is a particular application of BIP43. We define the following 5 levels in BIP32 path:

m / purpose' / coin_type' / account' / change / address_index

Apostrophe in the path indicates that BIP32 hardened derivation is used. Each level has a special meaning, described in the chapters below.

Purpose

Purpose is a constant set to 44' (or 0x8000002C) following the BIP43 recommendation. It indicates that the subtree of this node is used according to this specification.

Hardened derivation is used at this level.

Coin type
One master node (seed) can be used for unlimited number of independent cryptocoins such as Bitcoin, Litecoin or Namecoin. However, sharing the same space for various cryptocoins has some disadvantages.

This level creates a separate subtree for every cryptocoin, avoiding reusing addresses across cryptocoins and improving privacy issues.

Coin type is a constant, set for each cryptocoin. Cryptocoin developers may ask for registering unused number for their project. The list of already allocated coin types is in the chapter "Registered coin types" below.

Hardened derivation is used at this level.

Account

This level splits the key space into independent user identities, so the wallet never mixes the coins across different accounts.

Users can use these accounts to organize the funds in the same fashion as bank accounts; for donation purposes (where all addresses are considered public), for saving purposes, for common expenses etc.

Accounts are numbered from index 0 in sequentially increasing manner. This number is used as child index in BIP32 derivation.

Hardened derivation is used at this level.

Software should prevent a creation of an account if a previous account does not have a transaction history (meaning none of its addresses have been used before). Software needs to discover all used accounts after importing the seed from an external source. Such an algorithm is described in "Account discovery" chapter.

Change

Constant 0 is used for external chain and constant 1 for internal chain (also known as change addresses). External chain is used for addresses that are meant to be visible outside of the wallet (e.g. for receiving payments). Internal chain is used for addresses which are not meant to be visible outside of the wallet and is used for return transaction change.

Public derivation is used at this level.

Index

Addresses are numbered from index 0 in sequentially increasing manner. This number is used as child index in BIP32 derivation.

Public derivation is used at this level.

APPENDIX D
Script Opcodes

This appendix has a list of all script words aka OPCODES, commands or functions. False is zero or negative zero (using any number of bytes) or an empty array, and True is anything else.

Constants

Word	Opcode	Hex	Input	Output	Description
OP_0, OP_FALSE	0	0x00	Nothing	(empty val)	An empty array of bytes is pushed onto the stack. (This is not a no-op: an item is added to the stack.)
N/A	1-75	0x01-0x4b	(special)	data	The next opcode bytes is data to be pushed onto the stack OP_PUSHDATA1 76 0x4c (special) data The next byte contains the number of bytes to be pushed onto the stack.
OP_PUSHDATA1	76	0x4c	(special)	data	The next byte contains the number of bytes to be pushed onto the stack.
OP_PUSHDATA2	77	0x4d	(special)	data	The next two bytes contain the number of bytes to be pushed onto the stack in little endian order.
OP_PUSHDATA4	78	0x4e	(special)	data	The next four bytes contain the number of bytes to be pushed onto the stack in little endian order.
OP_1NEGATE	79	0x4f	Nothing	-1	The number -1 is pushed onto the stack.
OP_1, OP_TRUE	81	0x51	Nothing	1	The number 1 is pushed onto the stack.
OP_2-OP_16	82-96	0x52-0x60	Nothing	2-16	The number in the word name (2-16) is pushed onto the stack.

Flow Control

<if_expressions> = <expression> if [statements] [else [statements]]* endif

Word	Opcode	Hex	Input	Output	Description
OP_NOP	97	0x61	Nothing	(empty val)	An empty array of bytes is pushed onto the stack. (This is not a no-op: an item is added to the stack.)
OP_IF	99	0x63	<if_expressions>	NA	If the top stack value is not False, the statements are executed. The top stack value is removed.
OP_NOTIF	100	0x64	<if_expressions>	NA	If the top stack value is False, the statements are executed. The top stack value is removed.
OP_ELSE	103	0x67	(<if_expressions>	NA	If the preceding OP_IF or OP_NOTIF or OP_ELSE was not executed then these statements are and if the preceding OP_IF or OP_NOTIF or OP_ELSE was executed then these statements are not.
OP_ENDIF	104	0x68	<if_expressions>	NA	Ends an if/else block. All blocks must end, or the transaction is invalid. An OP_ENDIF without OP_IF earlier is also invalid.
OP_VERIFY	105	0x69	True/False	Nothing / fail	Marks transaction as invalid if top stack value is not true. The top stack value is removed.
OP_RETURN	106	0x6a	Nothing	Fail	Marks transaction as invalid. Since Bitcoin 0.9, a standard way of attaching extra data to transactions is to add a zero-value output with a scriptPubKey consisting of OP_RETURN followed by data. Such outputs are provably unspendable and specially discarded from storage in the UTXO set, reducing their cost to the network. Standard relay rules allow a single output with OP_RETURN, that contains any sequence of push statements (or OP_RESERVED[1]) after the OP_RETURN provided the total scriptPubKey length is at most 83 bytes.

Stack

Word	Opcode	Hex	Input	Output	Description
OP_TOALTSTACK	107	0x6B	x1	(alt)x1	Puts the input onto the top of the alt stack. Removes it from the main stack.
OP_FROMALTSTACK	108	0x6B	(alt)x1	x1	Puts the input onto the top of the main stack. Removes it from the alt stack.
OP_IFDUP	115	0x73	x	x / x x	If the top stack value is not 0, duplicate it.
OP_DEPTH	116	0x74	Nothing	<Stack size>	Puts the number of stack items onto the stack.
OP_DROP	117	0x75	x	Nothing	Removes the top stack item
OP_DUP	118	0x76	x	x x	Duplicates the top stack item.
OP_NIP	119	0x77	x1 x2	x2	Removes the second-to-top stack item.
OP_OVER	120	0x78	x1 x2	x1 x2 x1	Copies the second-to-top stack item to the top.
OP_PICK	121	0x79	xn ... x2 x1 x0 <n>	xn ... x2 x1 x0 xn	The item n back in the stack is copied to the top.
OP_ROLL	122	0X7a	xn ... x2 x1 x0 <n>	... x2 x1 x0 xn	The item n back in the stack is moved to the top.
OP_ROT	123	0x7b	x1 x2 x3	x 2 x3 x1	The top three items on the stack are rotated to the left.
OP_SWAP	124	0x7c	x1 x2	x 2 x1	The top two items on the stack are swapped.
OP_TUCK	125	0x7d	x1 x2	x2 x1 x2	The item at the top of the stack is copied and inserted before the second-to-top item.
OP_2DROP	109	0x6d	x1 x2	Nothing	Removes the top two stack items.

OP_2DUP	110	0x6e	x1 x2	x1 x2 x1 x2	Duplicates the top two stack items.
OP_3DUP	111	0x6f	x1 x2 x3	x1 x2 x3 x1 x2 x3	Duplicates the top three stack items.
OP_2OVER	112	0x70	x1 x2 x3 x4	x1 x2 x3 x4 x1 x2	Copies the pair of items two spaces back in the stack to the front.
OP_2ROT	113	0x71	x1 x2 x3 x4 x5 x6	x3 x4 x5 x6 x1 x2	The fifth and sixth items back are moved to the top of the stack.
OP_2SWAP	114	0x72	x1 x2 x3 x4	x3 x4 x1 x2	Swaps the top two pairs of items.

Splice

Word	**Opcode**	**Hex**	**Input**	**Output**	**Description**
OP_SIZE	130	0x82	in	In size	Pushes the string length of the top element of the stack (without popping it).

BitWise Logic

Word	**Opcode**	**Hex**	**Input**	**Output**	**Description**
OP_EQUAL	135	0x87	x1 x2	True / False	Returns 1 if the inputs are exactly equal, 0 otherwise.
OP_EQUALVERIFY	136	0x88	x1 x2	Nothing / Fail	Same as OP_EQUAL, but runs OP_VERIFY afterward.

Arithmetic

Word	Opcode	Hex	Input	Output	Description
OP_1ADD	139	0x8b	in	out	1 is added to the input.
OP_1SUB	140	0x8c	in	out	1 is subtracted from the input.
OP_NEGATE	143	0x8f	in	out	The sign of the input is flipped.
OP_ABS	144	0x90	in	out	The input is made positive.
OP_NOT	145	0x91	in	out	If the input is 0 or 1, it is flipped. Otherwise the output will be 0.
OP_0NOTEQUAL	146	0x92	In	out	Returns 0 if the input is 0. 1 otherwise.
OP_ADD	147	0x93	a b	out	a is added to b
OP_SUB	148	0x94	a b	out	b is subtracted from a
OP_BOOLAND	154	0x9a	a b	out	If both a and b are not 0, the output is 1. Otherwise 0.
OP_BOOLOR	155	0x9b	a b	out	If a or b is not 0, the output is 1. Otherwise 0.
OP_NUMEQUAL	156	0x9c	a b	out	Returns 1 if the numbers are equal, 0 otherwise.
OP_NUMEQUALVERIFY	157	0x9d	a b	Nothing/Fal	Same as OP_NUMEQUAL, but runs OP_VERIFY afterward.
OP_NUMNOTEQUAL	158	0x9e	a b	out	Returns 1 if the numbers are not equal, 0 otherwise.
OP_LESSTHAN	159	0x9f	a b	out	Returns 1 if a is less than b, 0 otherwise.
OP_GREATERTHAN	160	0xa0	a b	out	Returns 1 if a is greater than b, 0 otherwise.
OP_LESSTHANOREQUAL	161	0xa1	a b	out	Returns 1 if a is less than or equal to b, 0 otherwise.

OP_GREATERTHANOREQUAL	162	0xa2	a b	out	Returns 1 if a is greater than or equal to b, 0 otherwise.
OP_MIN	163	0xa3	a b	out	Returns the smaller of a and b.
OP_MAX	164	0xa4	a b	out	Returns the larger of a and b.
OP_WITHIN	165	0xa5	x min max	out	Returns 1 if x is within the specified range (left-inclusive), 0 otherwise.

Crypto

Word	**Opcode**	**Hex**	**Input**	**Output**	**Description**
OP_RIPEMD160	166	0xa6	in	hash	The input is hashed using RIPEMD-160.
OP_SHA1	167	0xa7	in	hash	The input is hashed using SHA-1.
OP_SHA256	168	0xa8	in	hash	The input is hashed using SHA-256.
OP_HASH160	169	0xa9	in	hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_HASH256	170	0xaa	in	hash	The input is hashed two times with SHA-256.
OP_CODESEPARATOR	171	0xab	Nothing	Nothing	All of the signature checking words will only match signatures to the data after the most recently-executed OP_CODESEPARATOR.
OP_CHECKSIG	172	0xac	sig pubkey	True / False	The entire transaction's outputs, inputs, and script (from the most recently-executed OP_CODESEPARATOR to the end) are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.

OP_CHECKSIGVERIFY	173	0xad	sig pubkey	Nothing / Fail	Same as OP_CHECKSIG, but OP_VERIFY is executed afterward.
OP_CHECKMULTISIG	174	0xae	x sig1 sig2 ... <number of signatures> pub1 pub2 <number of public keys>	True / False	Compares the first signature against each public key until it finds an ECDSA match. Starting with the subsequent public key, it compares the second signature against each remaining public key until it finds an ECDSA match. The process is repeated until all signatures have been checked or not enough public keys remain to produce a successful result. All signatures need to match a public key. Because public keys are not checked again if they fail any signature comparison, signatures must be placed in the scriptSig using the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript. If all signatures are valid, 1 is returned, 0 otherwise. Due to a bug, one extra unused value is removed from the stack.
OP_CHECKMULTISIGVERIFY	175	0xaf	x sig1 sig2 ... <number of signatures> pub1 pub2 ... <number of public keys>	Nothing / Fail	Same as OP_CHECKMULTISIG, but OP_VERIFY is executed afterward.

Pseudo-words

Word	Opcode	Hex	Description
OP_PUBKEYHASH	253	0xfd	Represents a public key hashed with OP_HASH160.
OP_PUBKEY	254	0xfe	Represents a public key compatible with OP_CHECKSIG.
OP_INVALIDOPCODE	255	0xff	Matches any opcode that is not yet assigned.

Reserved Words

Word	Opcode	Hex	Description
OP_RESERVED	80	0x50	Transaction is invalid unless occuring in an unexecuted OP_IF branch
OP_VER	98	0x62	Transaction is invalid unless occuring in an unexecuted OP_IF branch
OP_VERIF	101	0x65	Transaction is invalid even when occurring in an unexecuted OP_IF branch
OP_VERNOTIF	102	0x66	Transaction is invalid even when occurring in an unexecuted OP_IF branch
OP_RESERVED1	137	0x89	Transaction is invalid unless occuring in an unexecuted OP_IF branch
OP_RESERVED2	138	0x8a	Transaction is invalid unless occuring in an unexecuted OP_IF branch
OP_NOP1, OP_NOP4-OP_NOP10	176,179-185	0xb0, 0xb3-0xb9	The word is ignored. Does not mark transaction as invalid.

References:
[1] V. Miller, “Uses of elliptic curves in cryptography”,Advances in Cryptology– Crypto’85, Lecture Notes in Computer Science,218(1986),Springer-Verlag,417-426
[2] Mastering Bitcoin
[3] BIPS 32, 43, 44
https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki
[4] Bitcoin Documentation