Algorand Transaction Execution Approval Language, Opcodes

September 24, 2019

Abstract

Algorand allows transactions to be effectively signed by a small program. If the program evaluates to true then the transaction is allowed. This document defines the language opcodes and byte encoding.

Contents

1 Opcodes

Ops have a 'cost' of 1 unless otherwise specified.

1.1 err

- Opcode: 0x00
- Pops: *None*
- Pushes: *None*
- Error. Panic immediately. This is primarily a fencepost against accidental zero bytes getting compiled into programs.

1.2 sha256

- Opcode: 0x01
- Pops: ... *stack*, []byte
- Pushes: []byte

- SHA256 hash of value, yields [32]byte
- **Cost**: 7

1.3 keccak256

- Opcode: 0x02
- Pops: ... *stack*, []byte
- Pushes: []byte
- Keccac256 hash of value, yields [32]byte
- **Cost**: 26

1.4 sha512_256

- Opcode: 0x03
- Pops: ... *stack*, []byte
- Pushes: []byte
- SHA512_256 hash of value, yields [32]byte
- **Cost**: 9

1.5 ed25519verify

- Opcode: 0x04
- Pops: ... *stack*, {[]byte A}, {[]byte B}, {[]byte C}
- Pushes: uint64
- for (data, signature, pubkey) verify the signature of the data against the pubkey => {0 or 1}
- **Cost**: 1900

1.6 rand

- Opcode: 0x05
- Pops: *None*
- Pushes: uint64
- push random uint64 to stack
- **Cost**: 3

Random number generator based on the ChaCha20 algorithm. Seeded with the previous block's `Seed` value and the current transaction ID.

1.7 +

- Opcode: 0x08
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A plus B. Panic on overflow.

1.8 -

- Opcode: 0x09
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A minus B. Panic if B > A.

1.9 /

- Opcode: 0x0a
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A divided by B. Panic if B == 0.

1.10 *

- Opcode: 0x0b
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A times B. Panic on overflow.

It is worth noting that there are 10,000,000,000,000,000 micro-Algos in the total supply, or a bit less than 2^54. When doing rational math, e.g. (A * (N/D)) as ((A * N) / D) one should limit the numerator to less than 2^10 to be completely sure there won't be overflow.

1.11 <

- Opcode: 0x0c
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A less than B => {0 or 1}

1.12 >

- Opcode: 0x0d
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A greater than B => {0 or 1}

1.13 <=

- Opcode: 0x0e
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A less than or equal to B => {0 or 1}

1.14 >=

- Opcode: 0x0f
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A greater than or equal to B => {0 or 1}

1.15 &&

- Opcode: 0x10
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A is not zero and B is not zero => {0 or 1}

1.16 ||

- Opcode: 0x11
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A is not zero or B is not zero => {0 or 1}

1.17 ==

- Opcode: 0x12
- Pops: ... *stack*, {any A}, {any B}
- Pushes: uint64
- A is equal to B => {0 or 1}

1.18 !=

- Opcode: 0x13
- Pops: ... *stack*, {any A}, {any B}
- Pushes: uint64
- A is not equal to B => {0 or 1}

1.19 !

- Opcode: 0x14
- Pops: ... *stack*, uint64
- Pushes: uint64
- X == 0 yields 1; else 0

1.20 len

- Opcode: 0x15
- Pops: ... *stack*, []byte
- Pushes: uint64

- yields length of byte value

1.21 btoi

- Opcode: 0x17
- Pops: ... *stack*, []byte
- Pushes: uint64
- converts bytes as big endian to uint64

1.22 %

- Opcode: 0x18
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A modulo B. Panic if B == 0.

1.23 |

- Opcode: 0x19
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A bitwise-or B

1.24 &

- Opcode: 0x1a
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A bitwise-and B

1.25 ^

- Opcode: 0x1b
- Pops: ... *stack*, {uint64 A}, {uint64 B}
- Pushes: uint64
- A bitwise-xor B

1.26 ~

- Opcode: 0x1c
- Pops: ... *stack*, uint64
- Pushes: uint64
- bitwise invert value

1.27 intcblock

- Opcode: 0x20 {varuint length} [{varuint value}, ...]

- Pops: *None*
- Pushes: *None*
- load block of uint64 constants

`intcblock` loads following program bytes into an array of integer constants in the evaluator. These integer constants can be referred to by `intc` and `intc_*` which will push the value onto the stack.

1.28 intc

- Opcode: 0x21 {uint8 int constant index}
- Pops: *None*
- Pushes: uint64
- push value from uint64 constants to stack by index into constants

1.29 intc_0

- Opcode: 0x22
- Pops: *None*
- Pushes: uint64
- push uint64 constant 0 to stack

1.30 intc_1

- Opcode: 0x23
- Pops: *None*
- Pushes: uint64
- push uint64 constant 1 to stack

1.31 intc_2

- Opcode: 0x24
- Pops: *None*
- Pushes: uint64
- push uint64 constant 2 to stack

1.32 intc_3

- Opcode: 0x25
- Pops: *None*
- Pushes: uint64
- push uint64 constant 3 to stack

1.33 bytecblock

- Opcode: 0x26 {varuint length} [({varuint value length} bytes), ...]
- Pops: *None*

- Pushes: *None*
- load block of byte-array constants

`bytecblock` loads the following program bytes into an array of byte string constants in the evaluator. These constants can be referred to by `bytec` and `bytec_*` which will push the value onto the stack.

1.34 bytec

- Opcode: 0x27 {uint8 byte constant index}
- Pops: *None*
- Pushes: []byte
- push bytes constant to stack by index into constants

1.35 bytec_0

- Opcode: 0x28
- Pops: *None*
- Pushes: []byte
- push bytes constant 0 to stack

1.36 bytec_1

- Opcode: 0x29
- Pops: *None*
- Pushes: []byte
- push bytes constant 1 to stack

1.37 bytec_2

- Opcode: 0x2a
- Pops: *None*
- Pushes: []byte
- push bytes constant 2 to stack

1.38 bytec_3

- Opcode: 0x2b
- Pops: *None*
- Pushes: []byte
- push bytes constant 3 to stack

1.39 arg

- Opcode: 0x2c {uint8 arg index N}
- Pops: *None*
- Pushes: []byte

- push LogicSig.Args[N] value to stack by index

1.40 arg_0

- Opcode: 0x2d
- Pops: *None*
- Pushes: []byte
- push LogicSig.Args[0] to stack

1.41 arg_1

- Opcode: 0x2e
- Pops: *None*
- Pushes: []byte
- push LogicSig.Args[1] to stack

1.42 arg_2

- Opcode: 0x2f
- Pops: *None*
- Pushes: []byte
- push LogicSig.Args[2] to stack

1.43 arg_3

- Opcode: 0x30
- Pops: *None*
- Pushes: []byte
- push LogicSig.Args[3] to stack

1.44 txn

- Opcode: 0x31 {uint8 transaction field index}
- Pops: *None*
- Pushes: any
- push field from current transaction to stack

Most fields are a simple copy of a uint64 or byte string value. `XferAsset` is the concatenation of the AssetID Creator Address (32 bytes) and the big-endian bytes of the uint64 AssetID Index for a total of 40 bytes.

`txn` Fields:

Index	Name	Type
0	Sender	[]byte
1	Fee	uint64
2	FirstValid	uint64

Index	Name	Type
3	LastValid	uint64
4	Note	[]byte
5	Receiver	[]byte
6	Amount	uint64
7	CloseRemainderTo	[]byte
8	VotePK	[]byte
9	SelectionPK	[]byte
10	VoteFirst	uint64
11	VoteLast	uint64
12	VoteKeyDilution	uint64
13	Type	[]byte
14	TypeEnum	uint64
15	XferAsset	[]byte
16	AssetAmount	uint64
17	AssetSender	[]byte
18	AssetReceiver	[]byte
19	AssetCloseTo	[]byte
20	GroupIndex	uint64

TypeEnum mapping:

Index	Name
0	unknown
1	pay
2	keyreg
3	acfg
4	axfer
5	afrz

1.45 global

- Opcode: 0x32 {uint8 global field index}
- Pops: *None*
- Pushes: any
- push value from globals to stack

`global` Fields:

Index	Name	Type
0	Round	uint64
1	MinTxnFee	uint64
2	MinBalance	uint64

Index	Name	Type
3	MaxTxnLife	uint64
4	TimeStamp	uint64
5	ZeroAddress	[]byte
6	GroupSize	uint64

1.46 gtxn

- Opcode: 0x33 {uint8 transaction group index}{uint8 transaction field index}
- Pops: *None*
- Pushes: any
- push field to the stack from a transaction in the current transaction group

for notes on transaction fields available, see `txn`

1.47 load

- Opcode: 0x34 {uint8 position in scratch space to load from}
- Pops: *None*
- Pushes: any
- copy a value from scratch space to the stack

1.48 store

- Opcode: 0x35 {uint8 position in scratch space to store to}
- Pops: ... *stack*, any
- Pushes: *None*
- pop a value from the stack and store to scratch space

1.49 bnz

- Opcode: 0x40 {0..0x7fff forward branch offset, big endian}
- Pops: ... *stack*, uint64
- Pushes: *None*
- branch if value is not zero

For a bnz instruction at `pc`, if the last element of the stack is not zero then branch to instruction at `pc + 3 + N`, else proceed to next instruction at `pc + 3`. Branch targets must be well aligned instructions. (e.g. Branching to the second byte of a 2 byte op will be rejected.)

1.50 pop

- Opcode: 0x48
- Pops: ... *stack*, any

- Pushes: *None*
- discard value from stack

1.51 dup

- Opcode: 0x49
- Pops: ... *stack*, any
- Pushes: any, any
- duplicate last value on stack