ZScript Standard Opcodes

Zenotta AG

To express redeeming conditions for transactions, Zenotta uses ZScript, a stack-based scripting language. In this document we provide a list of the standard opcodes that ZScript currently supports, along with examples. New opcodes will be added with future updates.

1 Constants

`OP_0 ... OP_16` push numbers from 0 to 16 onto the stack. Negative numbers are not allowed.

```
OP_1 OP_2 OP_3 -> [1 2 3]
```

2 Flow Control

`OP_NOP` does nothing.

```
OP_0 OP_NOP -> [0]
```

`OP_IF` checks if the top item on the stack is not 0 and executes the next block of instructions until `OP_ENDIF` or `OP_ELSE` is encountered. Skips the next block if the condition check is failed. Should have a closing `OP_ENDIF`. The top item is consumed.

```
OP_1 OP_IF OP_2 OP_ELSE OP_3 OP_ENDIF -> [2]
```

`OP_NOTIF` checks if the top item on the stack is 0 and executes the next block of instructions until `OP_ENDIF` or `OP_ELSE` is encountered. Skips the next block if the condition check is failed. Should have a closing `OP_ENDIF`. The top item is consumed.

```
OP_0 OP_NOTIF OP_2 OP_ELSE OP_3 OP_ENDIF -> [2]
```

`OP_ELSE` executes the next block of instructions if the previous `OP_IF` or `OP_NOTIF` was not executed. Should have a closing `OP_ENDIF`.

```
OP_0 OP_IF OP_2 OP_ELSE OP_3 OP_ENDIF -> [3]
```

`OP_ENDIF` ends either an `OP_IF` or `OP_NOTIF` block.

```
OP_0 OP_IF OP_2 OP_ENDIF -> []
```

`OP_VERIFY` removes the top item from the stack and ends execution with an error if it is 0.

```
OP_1 OP_VERIFY -> []
OP_0 OP_VERIFY -> fail
```

`OP_BURN` ends execution with an error. It can be used to create provably unspendable UTXOs.

```
OP_BURN -> fail
```

3 Stack

OP_TOALTSTACK moves the top item from the main stack to the top of the alt stack.

```
OP_1 OP_TOALTSTACK -> [] [1]
```

OP_FROMALTSTACK moves the top item from the alt stack to the top of the main stack.

```
OP_1 OP_TOALTSTACK OP_FROMALTSTACK -> [1] []
```

OP_2DROP removes the top two items from the stack.

```
OP_1 OP_2 OP_2DROP -> []
```

OP_2DUP duplicates the top two items on the stack.

```
OP_1 OP_2 OP_2DUP -> [1 2 1 2]
```

OP_3DUP duplicates the top three items on the stack.

```
OP_1 OP_2 OP_3 OP_3DUP -> [1 2 3 1 2 3]
```

OP_2OVER copies the second-to-top pair of items to the top of the stack.

```
OP_1 OP_2 OP_3 OP_4 OP_2OVER -> [1 2 3 4 1 2]
```

OP_2ROT moves the third-to-top pair of items to the top of the stack.

```
OP_1 OP_2 OP_3 OP_4 OP_5 OP_6 OP_2ROT -> [3 4 5 6 1 2]
```

OP_2SWAP swaps the top two pairs of items on the stack.

```
OP_1 OP_2 OP_3 OP_4 OP_2SWAP -> [3 4 1 2]
```

OP_IFDUP duplicates the top item on the stack if it is not 0.

```
OP_1 OP_IFDUP -> [1 1]
OP_0 OP_IFDUP -> [0]
```

OP_DEPTH pushes the stack size onto the stack.

```
OP_0 OP_1 OP_2 OP_3 OP_DEPTH -> [0 1 2 3 4]
```

OP_DROP removes the top item from the stack.

```
OP_1 OP_DROP -> []
```

OP_DUP duplicates the top item on the stack.

```
OP_1 OP_DUP -> [1 1]
```

OP_NIP removes the second-to-top item from the stack.

```
OP_1 OP_2 OP_NIP -> [2]
```

OP_OVER copies the second-to-top item to the top of the stack.

```
OP_1 OP_2 OP_OVER -> [1 2 1]
```

OP_PICK copies the $(n+1)$th-to-top item to the top of the stack, where n is the top item on the stack.

```
OP_1 OP_2 OP_3 OP_4 OP_3 OP_PICK -> [1 2 3 4 1]
```

OP_ROLL moves the $(n+1)$th-to-top item to the top of the stack, where n is the top item on the stack.

```
OP_1 OP_2 OP_3 OP_4 OP_3 OP_ROLL -> [2 3 4 1]
```

OP_ROT moves the third-to-top item to the top of the stack.

```
OP_1 OP_2 OP_3 OP_ROT -> [2 3 1]
```

OP_SWAP swaps the top two items on the stack.

```
OP_1 OP_2 OP_SWAP -> [2 1]
```

OP_TUCK copies the top item behind the second-to-top item on the stack.

```
OP_1 OP_2 OP_TUCK -> [2 1 2]
```

4 Splice

OP_CAT concatenates the two strings on top of the stack.

```
"hello" "world" OP_CAT -> ["helloworld"]
```

OP_SUBSTR extracts a substring from the third-to-top item on the stack.

```
"hello" 1 2 OP_SUBSTR -> ["el"]
```

OP_LEFT extracts a left substring from the second-to-top item on the stack.

```
"hello" 2 OP_LEFT -> ["he"]
```

OP_RIGHT extracts a right substring from the second-to-top item on the stack.

```
"hello" 2 OP_RIGHT -> ["llo"]
```

OP_SIZE computes the size in bytes of the string on top of the stack.

```
"hello" OP_SIZE -> ["hello" 5]
```

5 Bitwise Logic

OP_INVERT computes bitwise NOT of the number on top of the stack.

```
OP_0 OP_INVERT -> [18446744073709551615] // = 2^64 - 1
```

OP_AND computes bitwise AND between the two numbers on top of the stack.

```
OP_1 OP_2 OP_AND -> [0]
```

OP_OR computes bitwise OR between the two numbers on top of the stack.

```
OP_1 OP_1 OP_OR -> [1]
```

OP_XOR computes bitwise XOR between the two numbers on top of the stack.

```
OP_1 OP_1 OP_XOR -> [0]
```

OP_EQUAL substitutes the top two items on the stack with 1 if they are equal, with 0 otherwise.

```
OP_1 OP_1 OP_EQUAL -> [1]
OP_1 OP_2 OP_EQUAL -> [0]
```

OP_EQUALVERIFY computes OP_EQUAL and OP_VERIFY in sequence.

```
OP_1 OP_1 OP_EQUALVERIFY -> []
OP_1 OP_2 OP_EQUALVERIFY -> fail
```

6 Arithmetic

OP_1ADD adds 1 to the number on top of the stack. Ends execution with an error in case of overflow.

```
OP_1 OP_1ADD -> [2]
OP_0 OP_INVERT OP_1ADD -> fail // = (2^64 - 1) + 1
```

OP_1SUB subtracts 1 from the number on top of the stack. Ends execution with an error in case the result is negative.

```
OP_1 OP_1SUB -> [0]
OP_0 OP_1SUB -> fail // = -1
```

OP_2MUL multiplies by 2 the number on top of the stack. Ends execution with an error in case of overflow.

```
OP_2 OP_2MUL -> [4]
OP_0 OP_INVERT OP_2MUL -> fail // = (2^64 - 1) * 2
```

OP_2DIV divides by 2 the number on top of the stack.

```
OP_1 OP_2DIV -> [0]
```

OP_NOT substitutes the number on top of the stack with 1 if it is equal to 0, with 0 otherwise.

```
OP_0 OP_NOT -> [1]
OP_2 OP_NOT -> [0]
```

OP_ONOTEQUAL substitutes the number on top of the stack with 1 if it is not equal to 0, with 0 otherwise.

```
OP_2 OP_ONOTEQUAL -> [1]
OP_0 OP_ONOTEQUAL -> [0]
```

OP_ADD adds the two numbers on top of the stack. Ends execution with an error in case of overflow.

```
OP_3 OP_2 OP_ADD -> [5]
OP_0 OP_INVERT OP_2 OP_ADD -> fail // = (2^64 - 1) + 2
```

OP_SUB subtracts the number on top of the stack from the second-to-top number on the stack. Ends execution with an error in case the result is negative.

```
OP_3 OP_2 OP_SUB -> [1]
OP_2 OP_3 OP_SUB -> fail // = -1
```

OP_MUL multiplies the second-to-top number by the number on top of the stack. Ends execution with an error in case of overflow.

```
OP_3 OP_2 OP_MUL -> [6]
OP_0 OP_INVERT OP_3 OP_MUL -> fail // = (2^64 - 1) * 3
```

OP_DIV divides the second-to-top number by the number on top of the stack. Ends execution with an error in case of division by 0.

```
OP_3 OP_2 OP_DIV -> [1]
OP_3 OP_0 OP_DIV -> fail
```

OP_MOD computes the remainder of the division of the second-to-top number by the number on top of the stack. Ends execution with an error in case of division by 0.

```
OP_3 OP_2 OP_MOD -> [1]
OP_3 OP_0 OP_MOD -> fail
```

OP_LSHIFT computes the left shift of the second-to-top number by the number on top of the stack. Ends execution with an error in case of overflow.

```
OP_2 OP_1 OP_LSHIFT -> [4]
OP_1 <64> OP_LSHIFT -> fail
```

OP_RSHIFT computes the right shift of the second-to-top number by the number on top of the stack. Ends execution with an error in case of overflow.

```
OP_2 OP_1 OP_RSHIFT -> [1]
OP_1 <64> OP_RSHIFT -> fail
```

OP_BOOLAND substitutes the two numbers on top of the stack with 1 if they are both non-0, with 0 otherwise.

```
OP_1 OP_2 OP_BOOLAND -> [1]
OP_0 OP_1 OP_BOOLAND -> [0]
```

OP_BOOLOR substitutes the two numbers on top of the stack with 1 if they are not both 0, with 0 otherwise.

```
OP_0 OP_1 OP_BOOLOR -> [1]
OP_0 OP_0 OP_BOOLOR -> [0]
```

OP_NUMEQUAL substitutes the two numbers on top of the stack with 1 if they are equal, with 0 otherwise.

```
OP_1 OP_1 OP_NUMEQUAL -> [1]
OP_0 OP_1 OP_NUMEQUAL -> [0]
```

OP_NUMEQUALVERIFY computes OP_NUMEQUAL and OP_VERIFY in sequence.

```
OP_1 OP_1 OP_NUMEQUALVERIFY -> []
OP_0 OP_1 OP_NUMEQUALVERIFY -> fail
```

OP_NUMNOTEQUAL substitutes the two numbers on top of the stack with 1 if they are not equal, with 0 otherwise.

```
OP_0 OP_1 OP_NUMNOTEQUAL -> [1]
OP_1 OP_1 OP_NUMNOTEQUAL -> [0]
```

OP_LESSTHAN substitutes the two numbers on top of the stack with 1 if the second-to-top is less than the top item, with 0 otherwise.

```
OP_2 OP_3 OP_LESSTHAN -> [1]
OP_3 OP_2 OP_LESSTHAN -> [0]
```

OP_GREATERTHAN substitutes the two numbers on top of the stack with 1 if the second-to-top is greater than the top item, with 0 otherwise.

```
OP_3 OP_2 OP_GREATERTHAN -> [1]
OP_2 OP_3 OP_GREATERTHAN -> [0]
```

OP_LESSTHANOREQUAL substitutes the two numbers on top of the stack with 1 if the second-to-top is less than or equal to the top item, with 0 otherwise.

```
OP_2 OP_2 OP_LESSTHANOREQUAL -> [1]
OP_3 OP_2 OP_LESSTHANOREQUAL -> [0]
```

OP_GREATERTHANOREQUAL substitutes the two numbers on top of the stack with 1 if the second-to-top is greater than or equal to the top item, with 0 otherwise.

```
OP_2 OP_2 OP_GREATERTHANOREQUAL -> [1]
OP_2 OP_3 OP_GREATERTHANOREQUAL -> [0]
```

OP_MIN substitutes the two numbers on top of the stack with the minimum between the two.

```
OP_0 OP_1 OP_MIN -> [0]
```

OP_MAX substitutes the two numbers on top of the stack with the maximum between the two.

```
OP_0 OP_1 OP_MAX -> [1]
```

OP_WITHIN substitutes the three numbers on top of the the stack with 1 if the third-to-top is greater or equal to the second-to-top and less than the top item, with 0 otherwise.

```
OP_1 OP_1 OP_3 OP_WITHIN -> [1]
OP_3 OP_1 OP_3 OP_WITHIN -> [0]
```

7 Crypto

OP_SHA3 hashes the top item on the stack using SHA3-256.

```
"hello" OP_SHA3 -> ["3338be694f50c5f338814986cdf0686453a888b84f424d792af4b9202398f392"]
```

OP_HASH256 creates a standard address from a public key and pushes it onto the stack.

```
<pk> OP_HASH256 -> ["3c85fab8f52253e477abf4933cb196622d4cd57eb57fb236db6026017e576c9f"]
```

OP_CHECKSIG pushes 1 onto the stack if the signature is valid, 0 otherwise. It allows signature verification on arbitrary messages, not only transactions.

```
<msg> <sig> <pk> OP_CHECKSIG -> [1] / [0]
```

OP_CHECKSIGVERIFY runs OP_CHECKSIG and OP_VERIFY in sequence.

```
<msg> <sig> <pk> OP_CHECKSIGVERIFY -> [] / fail
```

OP_CHECKMULTISIG pushes 1 onto the stack if the m-of-n multi-signature is valid, 0 otherwise. It allows multi-signature verification on arbitrary messages, not only transactions. Ordering of signatures and public keys is not relevant.

```
<msg> <sig1> <sig2> OP_2 <pk1> <pk2> <pk3> OP_3 OP_CHECKMULTISIG -> [1] / [0]
```

OP_CHECKMULTISIGVERIFY runs OP_CHECKMULTISIG and OP_VERIFY in sequence.

```
<msg> <sig1> <sig2> OP_2 <pk1> <pk2> <pk3> OP_3 OP_CHECKMULTISIGVERIFY -> [] / fail
```