Uxntal Opcodes

Uxn has 64kb of memory, 16 devices, 2 stacks, and 36 opcodes with 3 modes each. The list below show the opcodes and their effect on a given stack **a b c**, where **PC**: Program Counter, **M**: Memory, **D**: Devices, and **rs**: Return Stack.

```
LIT a b c M[PC]   EQU a b?c             LDZ a b M[c8]      ADD a b+c
INC a b c+1       NEQ a b!c             STZ a {M[c8]=b}    SUB a b-c
POP a b           GTH a b>c             LDR a b M[PC+c8]   MUL a b*c
NIP a c           LTH a b<c             STR a {M[PC+c8]=b} DIV a b/c
SWP a c b         JMP a b {PC+=c}       LDA a b M[c16]     AND a b&c
ROT b c a         JCN a {(b8)PC+=c}     STA a {M[c16]=b}   ORA a b|c
DUP a b c c       JSR a b {rs.PC PC+=c} DEI a b D[c8]      EOR a b^c
OVR a b c b       STH a b {rs.c}        DEO a {D[c8]=b}    SFT a b>>c8l<<c8h

JMI PC=M[PC]      JCI (a8)PC=M[PC]      JSI {rs.PC} M[PC]
--2 a16 b16+c16   --r a b c {rs.b+rs.c} --k a b c b+c
```

To learn more about each opcode, see the {Uxntal Reference}.

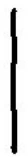
The Uxntal Manual

This documentation includes {hand gestures}, and {glyphs}, which might serve a dual purpose; both enabling the usage of the Uxntal language outside of the computer, as well as to help students to familiarize themselves with {hexadecimal} finger-counting and bitwise operations.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	BRK	INC	POP	NIP	SWP	ROT	DUP	OVR	EQU	NEQ	GTH	LTH	JMP	JCN	JSR	STH
10	LDZ	STZ	LDR	STR	LDA	STA	DEI	DEO	ADD	SUB	MUL	DIV	AND	ORA	EOR	SFT
20	JCI	INC2	POP2	NIP2	SWP2	ROT2	DUP2	OVR2	EQU2	NEQ2	GTH2	LTH2	JMP2	JCN2	JSR2	STH2
30	LDZ2	STZ2	LDR2	STR2	LDA2	STA2	DEI2	DEO2	ADD2	SUB2	MUL2	DIV2	AND2	ORA2	EOR2	SFT2
40	JMI	INCr	POPr	NIPr	SWPr	ROTr	DUPr	OVRr	EQUr	NEQr	GTHr	LTHr	JMPr	JCNr	JSRr	STHr
50	LDZr	STZr	LDRr	STRr	LDAr	STAr	DEIr	DEOr	ADDr	SUBr	MULr	DIVr	ANDr	ORAr	EORr	SFTr
60	JSI	INC2r	POP2r	NIP2r	SWP2r	ROT2r	DUP2r	OVR2r	EQU2r	NEQ2r	GTH2r	LTH2r	JMP2r	JCN2r	JSR2r	STH2r
70	LDZ2r	STZ2r	LDR2r	STR2r	LDA2r	STA2r	DEI2r	DEO2r	ADD2r	SUB2r	MUL2r	DIV2r	AND2r	ORA2r	EOR2r	SFT2r
80	LIT	INCk	POPk	NIPk	SWPk	ROTk	DUPk	OVRk	EQUk	NEQk	GTHk	LTHk	JMPk	JCNk	JSRk	STHk
90	LDZk	STZk	LDRk	STRk	LDAk	STAk	DEIk	DEOk	ADDk	SUBk	MULk	DIVk	ANDk	ORAk	EORk	SFTk
a0	LIT2	INC2k	POP2k	NIP2k	SWP2k	ROT2k	DUP2k	OVR2k	EQU2k	NEQ2k	GTH2k	LTH2k	JMP2k	JCN2k	JSR2k	STH2k
b0	LDZ2k	STZ2k	LDR2k	STR2k	LDA2k	STA2k	DEI2k	DEO2k	ADD2k	SUB2k	MUL2k	DIV2k	AND2k	ORA2k	EOR2k	SFT2k
c0	LITr	INCkr	POPkr	NIPkr	SWPkr	ROTkr	DUPkr	OVRkr	EQUkr	NEQkr	GTHkr	LTHkr	JMPkr	JCNkr	JSRkr	STHkr
d0	LDZkr	STZkr	LDRkr	STRkr	LDAkr	STAkr	DEIkr	DEOkr	ADDkr	SUBkr	MULkr	DIVkr	ANDkr	ORAkr	EORkr	SFTkr
e0	LIT2r	INC2kr	POP2kr	NIP2kr	SWP2kr	ROT2kr	DUP2kr	OVR2kr	EQU2kr	NEQ2kr	GTH2kr	LTH2kr	JMP2kr	JCN2kr	JSR2kr	STH2kr
f0	LDZ2kr	STZ2kr	LDR2kr	STR2kr	LDA2kr	STA2kr	DEI2kr	DEO2kr	ADD2kr	SUB2kr	MUL2kr	DIV2kr	AND2kr	ORA2kr	EOR2kr	SFT2kr

In the `a b -- c d` notation, "a b" represent the state of the stack before the operation, "c d" represent the state after the operation, with "b" and "d" on top of the stack, respectively.

Break



`BRK --` Ends the evalutation of the current {vector}. This opcode has no modes.

Jump Conditional Instant

`JCI cond8 --` Pops a byte from the working stack and if it is not zero, moves the PC to a relative address at a distance equal to the next short in memory, otherwise moves PC+2. This opcode has no modes.

Jump Instant

`JMI --` Moves the PC to a relative address at a distance equal to the next short in memory. This opcode has no modes.

Jump Stash Return Instant

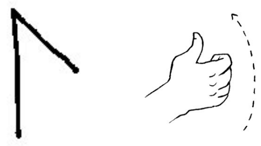
`JSI --` Pushes PC+2 to the return-stack and moves the PC to a relative address at a distance equal to the next short in memory. This opcode has no modes.

Literal

`LIT -- a` Pushes the next bytes in memory, and moves the PC+2. The LIT opcode always has the {keep mode} active. Notice how the `0x00` opcode, with the *keep* bit toggled, is the location of the literal opcodes. To learn more, see {literals}.

```
LIT 12          ( 12 )
LIT2 abcd       ( ab cd )
```

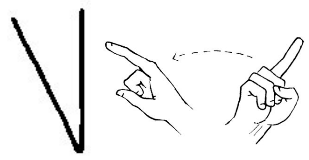
Increment



`INC a -- a+1` Increments the value at the top of the stack, by 1.

```
#01 INC         ( 02 )
#0001 INC2      ( 00 02 )
#0001 INC2k     ( 00 01 00 02 )
```

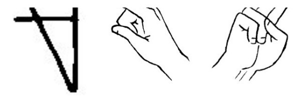
Pop



`POP a --` Removes the value at the top of the stack.

```
#1234 POP    ( 12 )
#1234 POP2   ( )
#1234 POP2k  ( 12 34 )
```

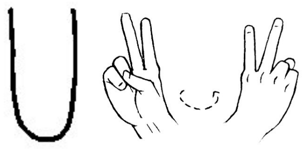
Nip



`NIP a b -- b` Removes the second value from the stack. This is practical to convert a small short into a byte.

```
#1234 NIP          ( 34 )
#1234 #5678 NIP2   ( 56 78 )
#1234 #5678 NIP2k  ( 12 34 56 78 56 78 )
```

Swap



`SWP a b -- b a` Exchanges the first and second values at the top of the stack.

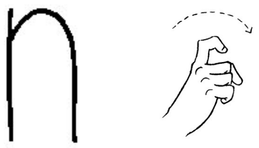
```
#1234 SWP          ( 34 12 )
#1234 SWPk         ( 12 34 34 12 )
#1234 #5678 SWP2   ( 56 78 12 34 )
#1234 #5678 SWP2k  ( 12 34 56 78 56 78 12 34 )
```

Rotate

`ROT a b c -- b c a` Rotates three values at the top of the stack, to the left, wrapping around.

```
#1234 #56 ROT            ( 34 56 12 )
#1234 #56 ROTk           ( 12 34 56 34 56 12 )
#1234 #5678 #9abc ROT2   ( 56 78 9a bc 12 34 )
#1234 #5678 #9abc ROT2k  ( 12 34 56 78 9a bc 56 78 9a bc 12 34 )
```

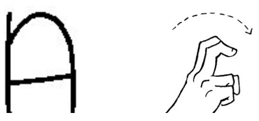
Duplicate



`DUP a -- a a` Duplicates the value at the top of the stack.

```
#1234 DUP   ( 12 34 34 )
#12 DUPk    ( 12 12 12 )
#1234 DUP2  ( 12 34 12 34 )
```

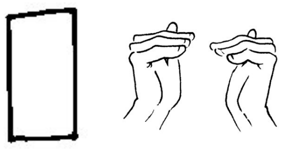
Over



`OVR a b -- a b a` Duplicates the second value at the top of the stack.

```
#1234 OVR          ( 12 34 12 )
#1234 OVRk         ( 12 34 12 34 12 )
#1234 #5678 OVR2   ( 12 34 56 78 12 34 )
#1234 #5678 OVR2k  ( 12 34 56 78 12 34 56 78 12 34 )
```

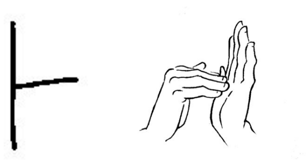
Equal



`EQU a b -- bool8` Pushes 01 to the stack if the two values at the top of the stack are equal, 00 otherwise.

```
#1212 EQU          ( 01 )
#1234 EQUk         ( 12 34 00 )
#abcd #ef01 EQU2   ( 00 )
#abcd #abcd EQU2k  ( ab cd ab cd 01 )
```

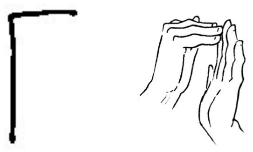
Not Equal



`NEQ a b -- bool8` Pushes 01 to the stack if the two values at the top of the stack are not equal, 00 otherwise.

```
#1212 NEQ          ( 00 )
#1234 NEQk         ( 12 34 01 )
#abcd #ef01 NEQ2   ( 01 )
#abcd #abcd NEQ2k  ( ab cd ab cd 00 )
```

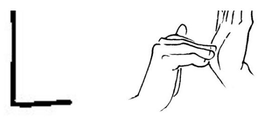
Greater Than



`GTH a b -- bool8` Pushes 01 to the stack if the second value at the top of the stack is greater than the value at the top of the stack, 00 otherwise.

```
#1234 GTH          ( 00 )
#3412 GTHk         ( 34 12 01 )
#3456 #1234 GTH2   ( 01 )
#1234 #3456 GTH2k  ( 12 34 34 56 00 )
```

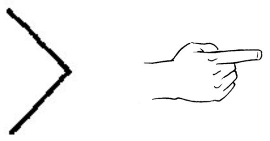
Lesser Than



`LTH a b -- bool8` Pushes 01 to the stack if the second value at the top of the stack is lesser than the value at the top of the stack, 00 otherwise.

```
#0101 LTH          ( 00 )
#0100 LTHk         ( 01 00 00 )
#0001 #0000 LTH2   ( 00 )
#0001 #0000 LTH2k  ( 00 01 00 00 00 )
```

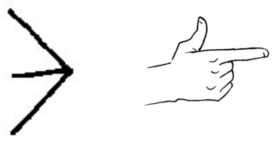
Jump



`JMP addr --` Moves the PC by a relative distance equal to the signed byte on the top of the stack, or to an absolute address in short mode.

```
,&skip-rel JMP BRK &skip-rel #01  ( 01 )
```

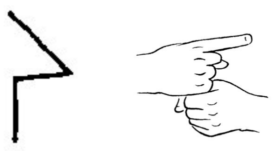
Jump Conditional



`JCN cond8 addr --` If the byte preceeding the address is not 00, moves the PC by a signed value equal to the byte on the top of the stack, or to an absolute address in short mode.

```
#abcd #01 ,&pass JCN SWP &pass POP  ( ab )
#abcd #00 ,&fail JCN SWP &fail POP  ( cd )
```

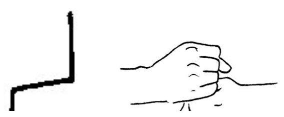
Jump Stash Return



`JSR addr --` Pushes the PC to the return-stack and moves the PC by a signed value equal to the byte on the top of the stack, or to an absolute address in short mode.

```
,&get JSR #01 BRK &get #02 JMP2r  ( 02 01 )
```

Stash



`STH a --` Moves the value at the top of the stack, to the return stack.

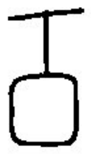
```
#01 STH LITr 02 ADDr STHr  ( 03 )
```

Load Zero-Page

`LDZ addr8 -- value` Pushes the value at an address within the first 256 bytes of memory, to the top of the stack.

```
|00 @cell $2 |0100 .cell LDZ ( 00 )
```

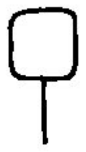
Store Zero-Page



`STZ val addr8 --` Writes a value to an address within the first 256 bytes of memory.

```
|00 @cell $2 |0100 #abcd .cell STZ2  { ab cd }
```

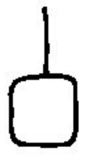
Load Relative



`LDR addr8 -- value` Pushes a value at a relative address in relation to the PC, within a range between -128 and +127 bytes, to the top of the stack.

```
,cell LDR2 BRK @cell abcd  ( ab cd )
```

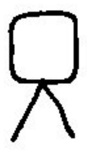
Store Relative



`STR val addr8 --` Writes a value to a relative address in relation to the PC, within a range between -128 and +127 bytes.

```
#1234 ,cell STR2 BRK @cell $2  ( )
```

Load Absolute



`LDA addr16 -- value` Pushes the value at a absolute address, to the top of the stack.

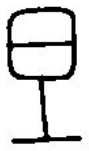
```
;cell LDA BRK @cell abcd ( ab )
```

Store Absolute

`STA val addr16 --` Writes a value to a absolute address.

```
#abcd ;cell STA BRK @cell $1 ( ab )
```

Device Input

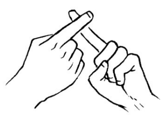


`DEI device8 -- value` Pushes a value from the device page, to the top of the stack. The target device might capture the reading to trigger an I/O event.

Device Output

`DEO val device8 --` Writes a value to the device page. The target device might capture the writing to trigger an I/O event.

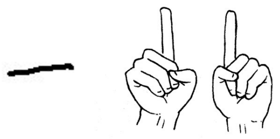
Add



`ADD a b -- a+b` Pushes the sum of the two values at the top of the stack.

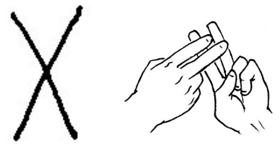
```
#1a #2e ADD       ( 48 )
#02 #5d ADDk      ( 01 5d 5f )
#0001 #0002 ADD2  ( 00 03 )
```

Subtract



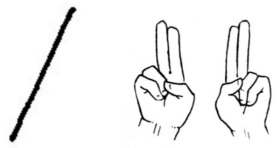
`SUB a b -- a-b` Pushes the difference of the first value minus the second, to the top of the stack.

Multiply



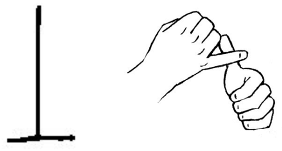
`MUL a b -- a*b` Pushes the product of the first and second values at the top of the stack.

Divide



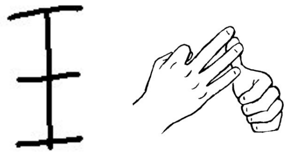
`DIV a b -- a/b` Pushes the quotient of the first value over the second, to the top of the stack.

And



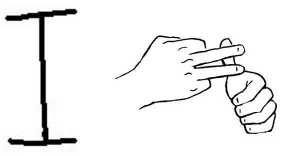
`AND a b -- a&b` Pushes the result of the bitwise operation AND, to the top of the stack.

Or



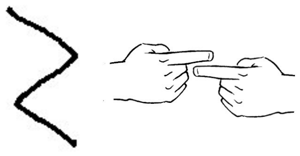
`ORA a b -- a|b` Pushes the result of the bitwise operation OR, to the top of the stack.

Exclusive Or



`EOR a b -- a^b` Pushes the result of the bitwise operation XOR, to the top of the stack.

Shift



`SFT a shift8 -- c` Shifts the bits of the second value of the stack to the left or right, depending on the control value at the top of the stack. The high nibble of the control value indicates how many bits to shift left, and the low nibble how many bits to shift right. The rightward shift is done first.

```
#34 #10 SFT        ( 68 )
#34 #01 SFT        ( 1a )
#34 #33 SFTk       ( 34 33 30 )
#1248 #34 SFTk2    ( 12 48 34 09 20 )
```

- [Rekka Bellum], illustration
- [Kira Oakley], contributor
- [Ismael Venegas Castello], contributor