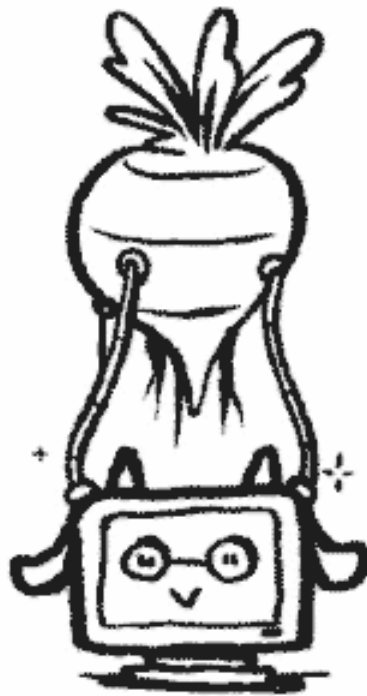


# THE CPU

At the heart of  
uxn is the cpu.  
It is said to be  
a beet.

The beet performs  
operations with  
instructions written  
in TAL.

> < & || =



+ - \* %

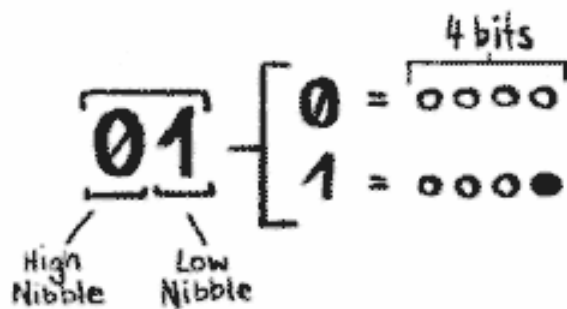
# BINARY ENCODING

The smallest unit I know are bits.  
A bit can either be 0 or 1.



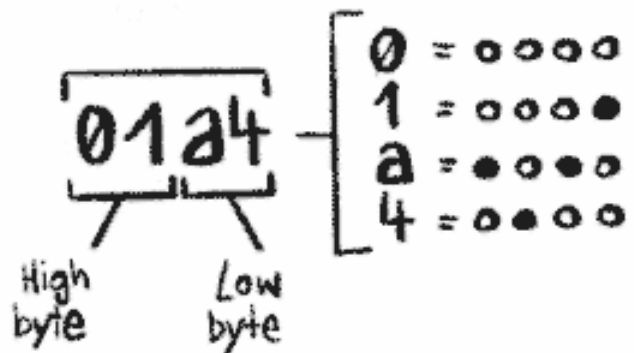
## BYTE

A byte is made of 8 bits.



## SHORT

A short is made of 2 bytes.



4 bits are called  
a "Nibble".



## BINARY TO HEX CONVERSION

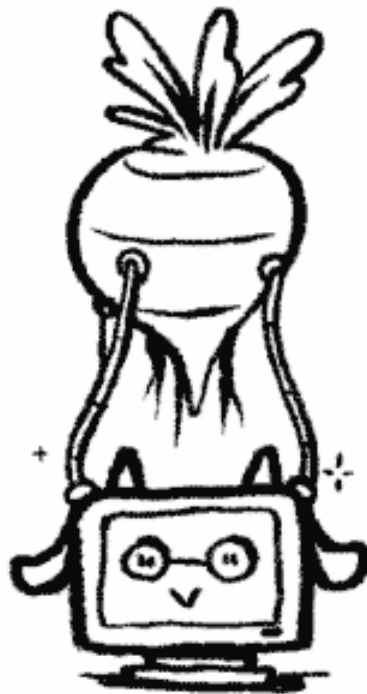
0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f



# THE CPU

At the heart of  
uxn is the cpu.  
It is said to be  
a beet.

The beet performs  
operations with  
instructions written  
in TAL.



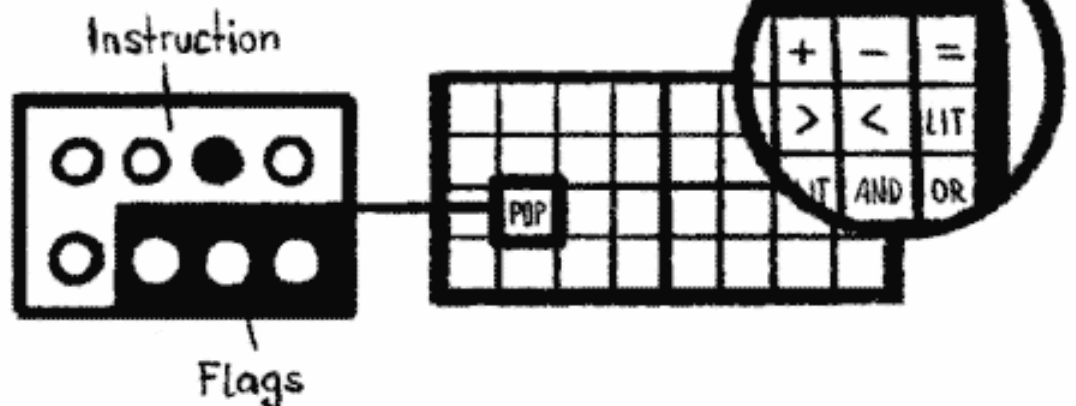
> < & || =

+ - \* %

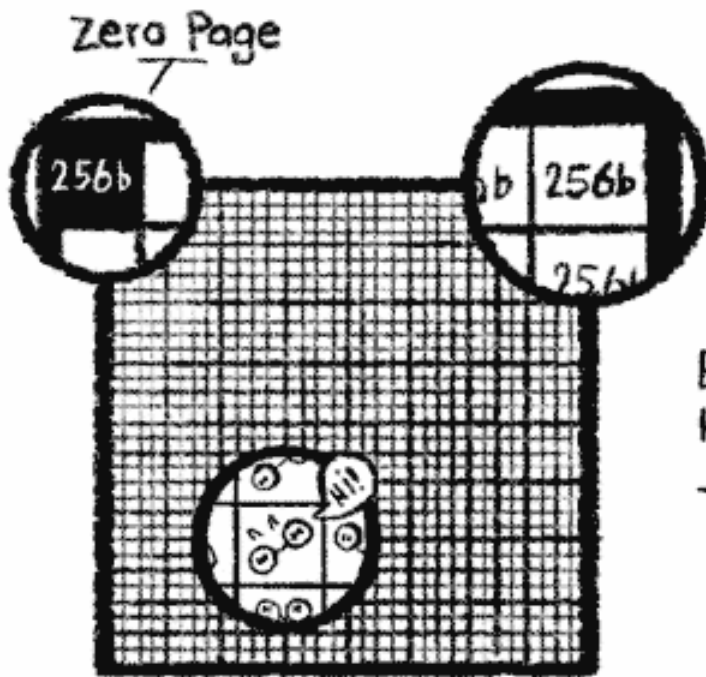
## OPCODES

Opcodes are instructions  
uxn can perform.

Each instruction is encoded  
in a single 8-bit word.



# MEMORY



## MAIN MEMORY 64 kb

Each byte in the main memory has an address of 16 bits.

The zero page is for data storage during runtime and can be addressed by 8 bits.

## I/O MEMORY 256 bytes



Varvara takes care of all devices such as screen, mouse, keyboard, audio, filesystem, etc.

### DEVICES

0	1	2	3
4	5	6	7
8	9	a	b
c	d	e	f

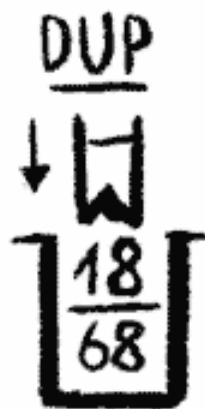
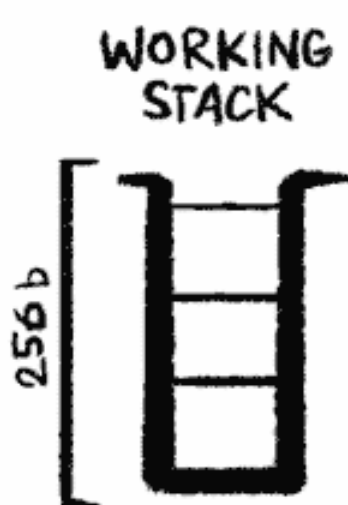
### PORTS

0	1	2	3
4	5	6	7
8	9	a	b
c	d	e	f

Each byte stores the address of a device. Each device has 16 ports.

# THE STACK

The stack is used to perform code operations.



Some instructions push bytes down onto the stack, others pop them off.

## REVERSE POLISH NOTATION a.k.a. "postfix"

A mathematical notation in which operators follow their operands.



INFIX	POSTFIX
$1 + 48$	$1\ 48\ +$
$(3 + 5) / 2 + 48$	$3\ 5\ +\ 2\ /\ 48\ +$

# RUNES, LABELS & MACROS



## RUNES

Runes are special characters that indicate element types of TAL.

%	Macro	&	Sublabel	;	Absolute Addr.
	Absolute Pad	#	Literal Hex	:	Raw Addr.
\$	Relative Pad	.	Zero Page Addr.	'	Raw Char
@	Label	,	Relative Addr.	"	Raw Word

## LABELS

Labels provide a readable link to devices and their ports.



## MACROS



Custom definitions that allow grouping and re-using instructions.

# THE INSTRUCTION CYCLE



The uxn cpu reads one byte at a time from the main memory.

Main memory address

Hex code for the letter "h"

Device 1: standard I/O

10100 LIT 68 LIT 18 DEO

Absolute pad

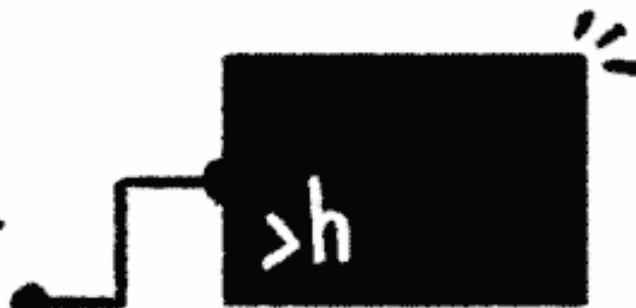
"Literal" instruction byte

Port 8: write

"Device Out" instruction



Once I read a byte I decode it as an instruction and perform it.



# ARITHMETICS

ADDITION  
#01 #24 ADD 25

SUBTRACTION  
#29 #24 SUB 85

MULTIPLICATION  
#02 #36 MUL 6C

DIVISION  
#fd #12 DIV 09



It's easier to read the stack horizontally.

01/24

To calculate shorts use the shortmode flag.



SHORT MODE ADDITION  
#0241 #1320 ADD2 1561



# BITWISE SHIFTING



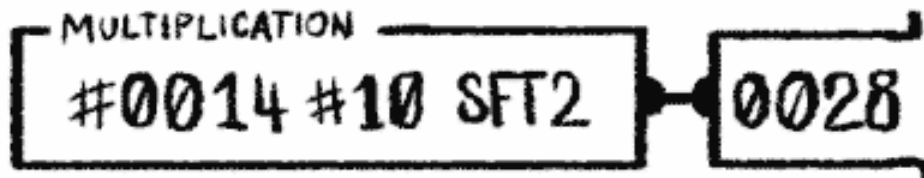
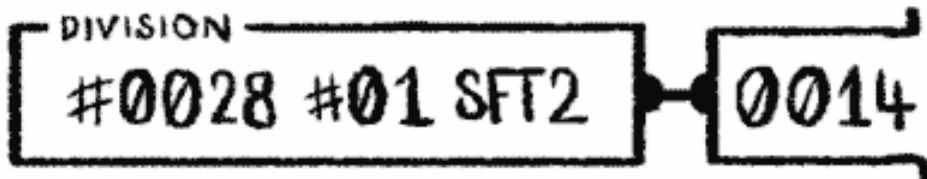
Bitwise shifting moves a bit  
to the left or to the right.



It can be used for  
multiplications or divisions!

28 ○ ○ ● ○ ● ○ ○ ○

14 ○ ○ ○ ● ○ ● ○ ○



14 ○ ○ ○ ● ○ ● ○ ○

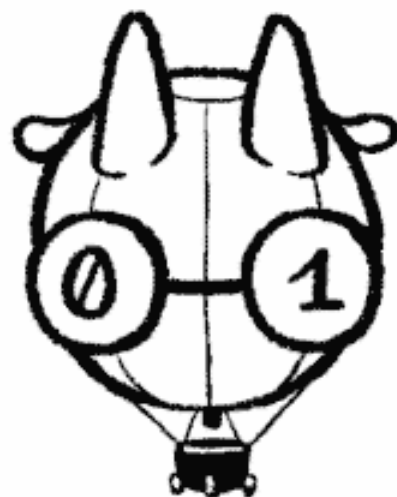
28 ○ ○ ● ○ ● ○ ○ ○

Bits can be shifted by more than one position.

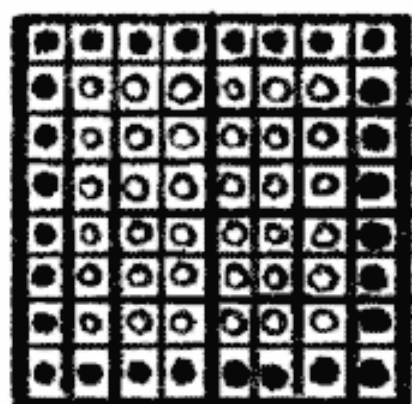
# SPRITE ENCODING

1 BIT PER PIXEL

A 1bpp tile is a set of 8 bytes that encode the state of its 8x8 pixels.



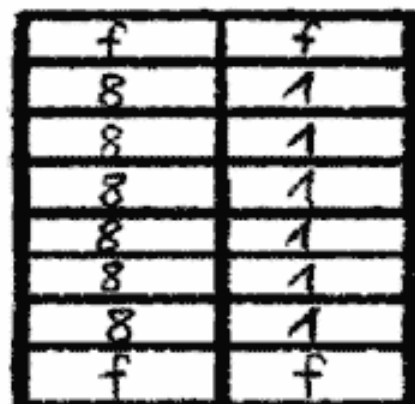
Each pixel can be on (1) or off (0).



DOTS



BINARY



HEX

SPRITE DEFINITION

```
@square ff81 8181 8181 81ff
```



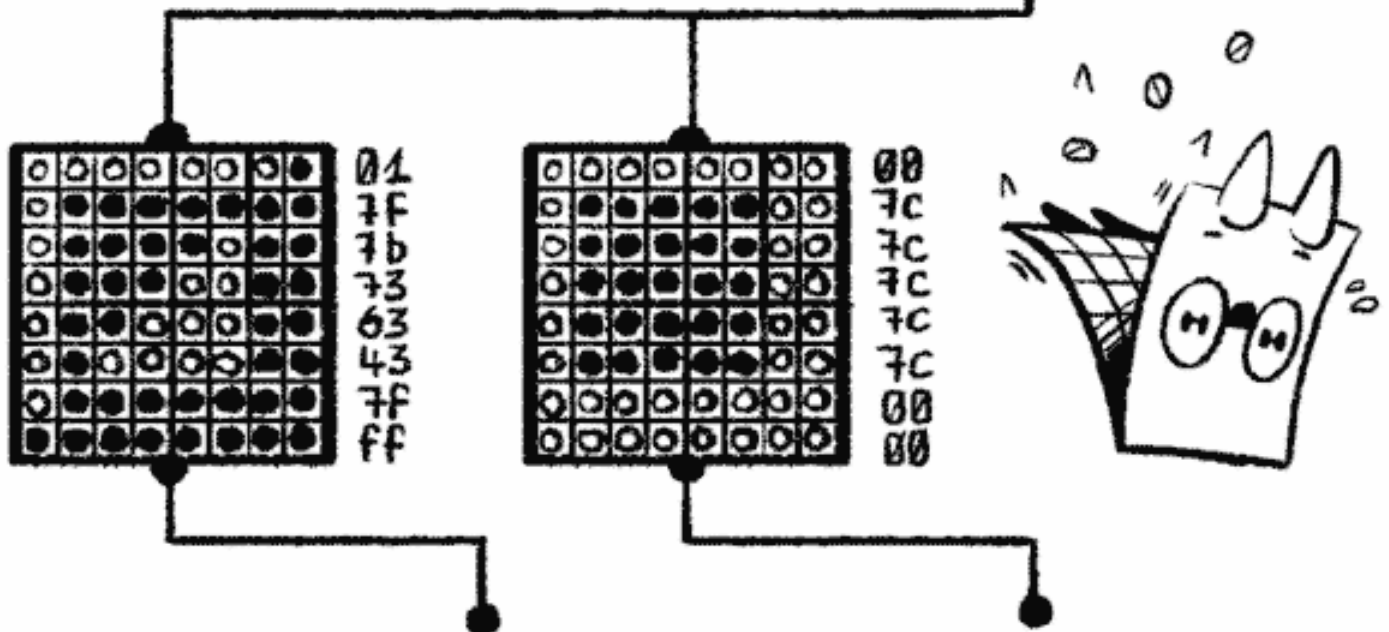
# 2 BITS PER PIXEL

Each pixel can have one of four possible colors.

0	0	0	0	0	0	0	1
0	3	3	3	3	3	1	1
0	3	3	3	3	2	1	1
0	3	3	3	2	2	1	1
0	3	3	2	2	2	1	1
0	3	2	2	2	2	1	1
0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

0 = 00  
 1 = 01  
 2 = 10  
 3 = 11

00	00	00	00	00	00	00	01
00	11	11	11	11	11	01	01
00	11	11	11	11	10	01	01
00	11	11	11	10	10	01	01
00	11	11	10	10	10	01	01
00	11	10	10	10	10	01	01
00	01	01	01	01	01	01	01
01	01	01	01	01	01	01	01



LOW BITS: @sprite 017F 7b73 6343 7FFF  
 HIGH BITS: 007c 7c7c 7c7c 0000

# DRAWING SPRITES

## 1. SETUP DEVICES



### SYSTEM

# vector....\$2  
# pad.....\$6  
# r.....\$2  
# g.....\$2  
# b.....\$2

### SCREEN

# vector.....\$2  
# width.....\$2  
# height.....\$2  
# pad.....\$6  
# x,y.....\$2  
# addr.....\$2  
# pixel.....\$1  
# sprite.....\$1

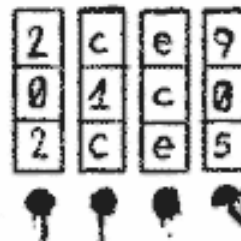
### DEVICES

```
100 @System [ #element #bytes ... ]  
120 @Screen [ #element #bytes ... ]
```

## 2. SET COLORS

### COLORS

```
10100  
# 2ce9 .System/r  
# 01c0 .System/g  
# 2ce5 .System/b
```



### 3. SET COORDINATES

```
COORDINATES  
#0008 .Screen/x DEO2  
#0008 .Screen/y DEO2
```

### 4. SET ADDRESS

```
ADDRESS  
;sprite .Screen/addr DEO2
```



### 5. DRAW THE SPRITE!

```
#01 .Screen/sprite DEO
```

Depending on whether you're drawing 1bpp or 2bpp sprites the high and the low nibble encode colors, layers, flipping and display modes.

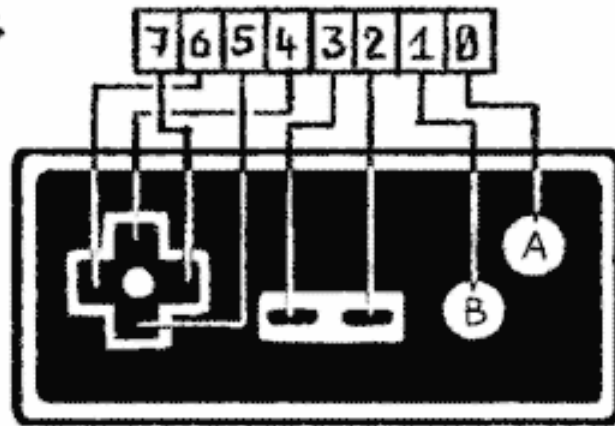


# INPUT

CONTROLLER DEVICE SETUP

```
180 @Controller [ &vector $2 &button $1 &key $1 ]
```

## BUTTONS



- 0 A
- 1 B
- 2 Select/Shift
- 3 Start/Home
- 4 Up
- 5 Down
- 6 Left
- 7 Right



The controller vector jumps to the ion-controller label address of a key pressed or released event.

LISTENING TO KEY EVENTS

```
ion-controller .Controller/vector DEO2
```

## KEYS

The key byte stores the ascii code of the keyboard key that is being pressed.



# COMPARISON

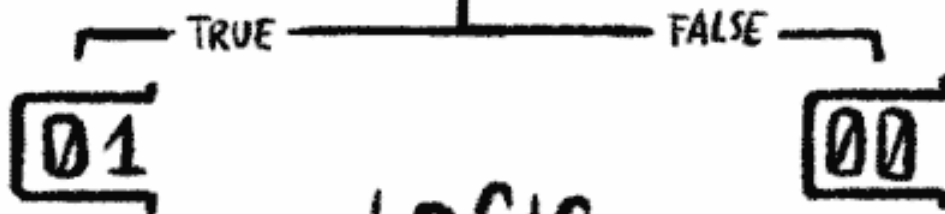
EQU  $a == b$   
NEQ  $a != b$   
GTH  $a > b$   
LTH  $a < b$



I push 01 in the stack when the condition is true. Otherwise I push 00 in the stack.

Is the pressed key equal to 'a'?

.Controller/key DEI LIT 'a' EQU



## LOGIC

There are three bitwise logic operators:

AND

$a \& b$



OR

$a | b$



EOR

$a \wedge b$



