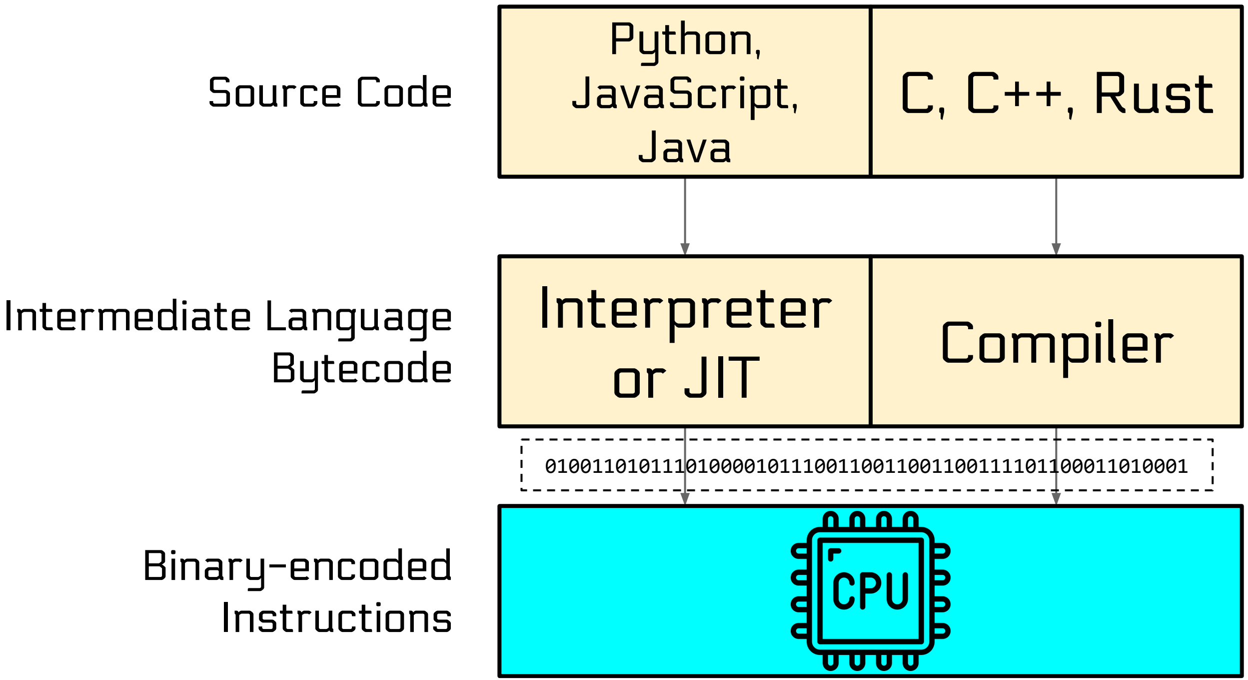
Assembly Crash Course

Data

Yan Shoshitaishvili
Arizona State University

All roads lead to the CPU



Binary?

Described mathematically by:
Thomas Harriot (pictured), Juan Caramuel y Lobkowitz, and/or Leibniz sometime in the 16th and 17th centuries.
But also known earlier: https://en.wikipedia.org/wiki/Binary_code

Decimal (base 10) has digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Binary (base 2) has digits 0, 1.

A binary digit is called a *bit*.

Numbers greater than 1 require multiple digits
(like numbers greater than 9 for base 10)

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000

Computers and Binary

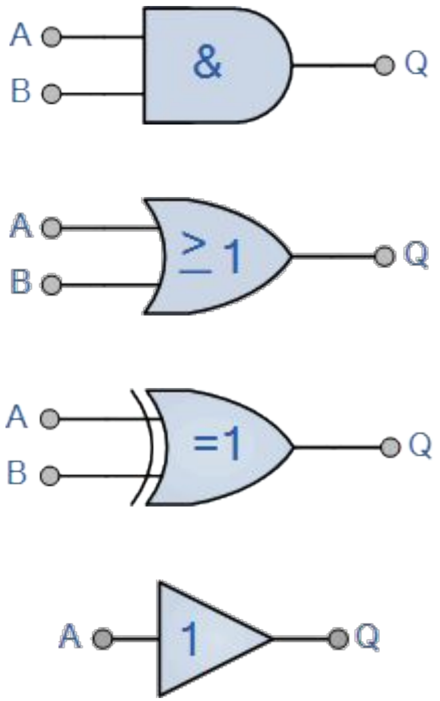
Why do computers speak binary? Consider the *logic gate*.

a. A, B, and Q represent either "on" or "off"
b. these concepts can be mapped to 1 and 0
c. "on" or "off" are relatively easy to check for
    i. binary: "is the lightbulb on"
    ii. other systems: "how bright is the lightbulb"

A few historical examples of *ternary* computers exist.

- Thomas Fowler's Calculating Machine https://en.wikipedia.org/wiki/Thomas_Fowler_(inventor)#Calculating_machine
- Setun: https://en.wikipedia.org/wiki/Setun
- QTC-1: https://ieeexplore.ieee.org/document/5195

But, binary is the standard.



http://www.electronics-tutorials.ws/logic/logic_1.html

Humans and Binary

Binary overwhelms the senses with a LOT of digits.

consider: 197_{10} is 11000101_2

compute: $11000101_2 - 10010011_2$ without writing it out

(it's $197_{10} - 147_{10} = 50_{10}$)

Decimal's "round" numbers don't align well to binary "round" numbers.

10000000_2 is 128_{10}
11000000_2 is 192_{10}
11100000_2 is 224_{10}
11110000_2 is 240_{10}

But if we use a base 2^X, we can represent X binary digits at once! Common bases:

Octal (base 2^3, or 8), commonly prefixed with `0`

Hexadecimal (base 2^4, or 16).

Caveat: how do we represent digits >10? A,B,C,D,E, and F!

Commonly prefixed with `0x`.

Decimal	Binary	Octal	Hex
0	0	00	0x0
1	1	01	0x1
2	10	02	0x2
3	11	03	0x3
4	100	04	0x4
5	101	05	0x5
6	110	06	0x6
7	111	07	0x7
8	**1000**	**010**	0x8
9	1001	011	0x9
10	1010	012	0xA
11	1011	013	0xB
12	1100	014	0xC
13	1101	015	0xD
14	1110	016	0xE
15	1111	017	0xF
16	**10000**	**020**	**0x10**
17	10001	021	0x11
18	10010	022	0x12
19	10011	023	0x13
20	10100	024	0x14
128	**10000000**	0200	**0x80**
192	**11000000**	0300	**0xc0**
224	**11100000**	0340	**0xe0**
240	**11110000**	0360	**0xf0**

Expressing Text

Bits in a computer typically do something useful.
Examples: encoding assembly instructions, whole programs, images, *text*...

Example: the earliest *extant* text encoding format is **ASCII**.
American Standard Code for Information Exchange.
Specified how to encode, in 7 bits, the English alphabet and common symbols.

For the most part:
Uppercase letters: `0x40 + LETTER_INDEX_IN_HEX`
Lowercase letters: `0x60 + LETTER_INDEX_IN_HEX`
Digit representations: `0x30 + DIGIT`
Characters lower than `0x20` (space) are "control characters":
`0x09` (tab), `0x0a` (newline), `0x07` (bell!)

ASCII has evolved into UTF-8, used on 98% of the web.
Leftmost bit (0x80) of letter signifies *extended* character (e.g., encoded in more than 8 bits).

	2	3	4	5	6	7
0:		0	@	P	`	p
1:	!	1	A	Q	a	q
2:	"	2	B	R	b	r
3:	#	3	C	S	c	s
4:	$	4	D	T	d	t
5:	%	5	E	U	e	u
6:	&	6	F	V	f	v
7:	'	7	G	W	g	w
8:	(	8	H	X	h	x
9:	)	9	I	Y	i	y
A:	*	:	J	Z	j	z
B:	+	;	K	[	k	{
C:	,	<	L	\	l	|
D:	-	=	M	]	m	}
E:	.	>	N	^	n	~
F:	/	?	O	_	o	DEL

Grouping Bits into Bytes

A standard-sized grouping of bits is called a *byte*.

Historically, somewhat tied to text encoding (e.g., # of bits to encode a letter).

Historical byte widths.
Nothing inherently good in any # of bits over any other # of bits (within reason).
I've encountered architectures with 6-bit, 7-bit, 8-bit, 9-bit, 12-bit, 16-bit, 18-bit, 31-bit, and 36-bit bytes!
The newest "real-world" architecture of these was from the late 1960s...

8-bit byte.
IBM invented 8-bit EBCDIC in 1963 for use on their terminals.
ASCII (also released in 1963!) replaced it, but the 8-bit byte stuck.
Every modern architecture uses 8-bit bytes.

Grouping Bytes into Words

Bytes are 8-bit, but modern architectures are (mostly) 64-bit...

Word.
Words are groupings of 8-bit bytes.
Architectures define the *word width*.
For historical reasons, the terminology is *really messed up*.

Nibble: half of a byte, 4 bits
Byte: 1 byte, 8 bits
Half word / "word": 2 bytes, 16 bits
Double word (dword): 4 bytes, 32 bits
Quad word (qword): 8 bytes, 64 bits

Note that the term Word on a 64-bit architecture can refer to either 16 or 16 bits!
Be precise.

Expressing Numbers

A 64-bit machine can reason about 64 bits at a time.

Caveat: in practice, even more. Modern x86 can use specialized hardware to crunch data 512 bits (64 *bytes)* at a time!

64 binary digits can express a large range of values!

Minimum: `0b0 = 0 = 0x0`
A cool number: `0b10100111001 = 1337 = 0x539`
A random number: `0b1011101000000011000100011110011111001011011000111100001001000 = 1675447075404019784 = 0x1740623cf96c7848`
Maximum: `0b1111111111111111111111111111111111111111111111111111111111111111 = 18446744073709551615 = 0xffffffffffffffff`

Sidebar: what happens if you add `1` to `0xffffffffffffffff`?

Integer overflow: `1 + 0xffffffffffffffff = 0x10000000000000000`
The 65th bit (1) doesn't fit!
The extra bit gets put in common *carry bit* storage by the CPU, and the result of the computation becomes `0`!
The inverse happens if we subtract `1` from `0`.

Expressing *Negative* Numbers

How to differentiate between positive and negative numbers?

One idea: sign bit (8-bit example):

Consider: `0b00000011 == 3`
If we use the leftmost bit as a sign bit: `0b10000011 == -3`
Drawback 1: `0b00000000 == 0 == 0b10000000`
Drawback 2: arithmetic operations have to be signedness-aware:

```
(unsigned) 0b00000000 - 1 = 0 - 1 = 255 == 0b11111111
(signed)   0b00000000 - 1 = 0 - 1 = -1  == 0b10000001
```

Clever (but crazy) approach: two's complement

One representation of zero: `0b00000000 == 0`
Negative numbers are represented as the large positive numbers that they would correlate to!

```
 0 - 1 == 0b11111111 == 255 == -1
-1 - 1 == 0b11111110 == 254 == -2
```

Advantage: arithmetic operations don't have to be sign-aware!

```
(unsigned) 0b00000000 - 1 = 0 - 1 = 255 == 0b11111111
(signed)   0b00000000 - 1 = 0 - 1 = -1  == 0b11111111
```

Bonus: sign-bit is still there (for easy testing for negative numbers)!
Note: smallest expressible negative number (for 8 bits): `0b10000000 = -128`

John von Neumann
First Draft of a Report on the EDVAC, 1945.

Anatomy of a Word

Consider `0xc001c475`: