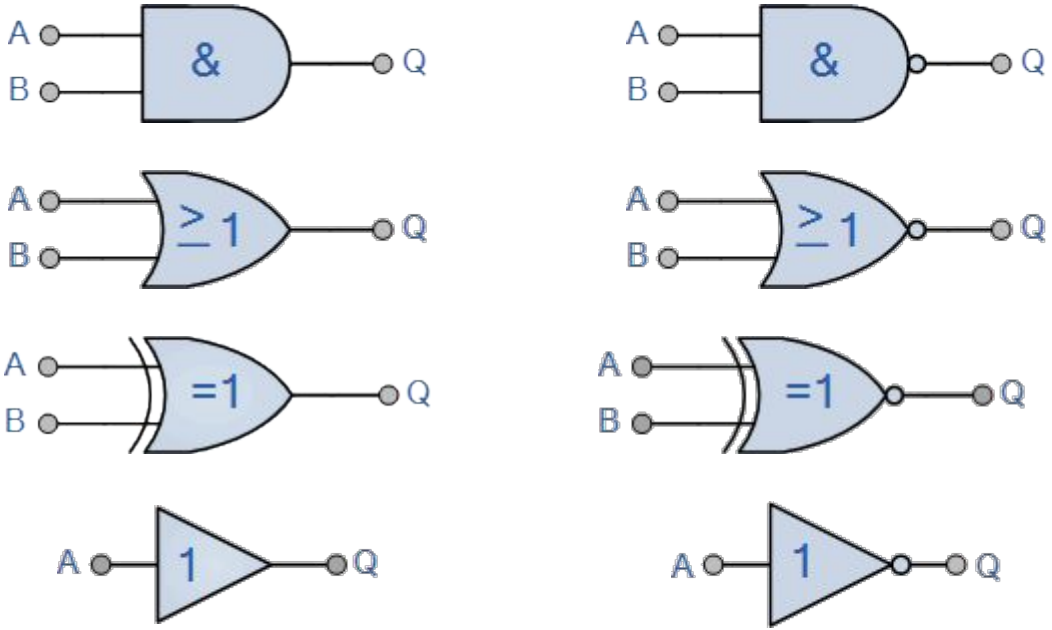
Assembly Crash Course

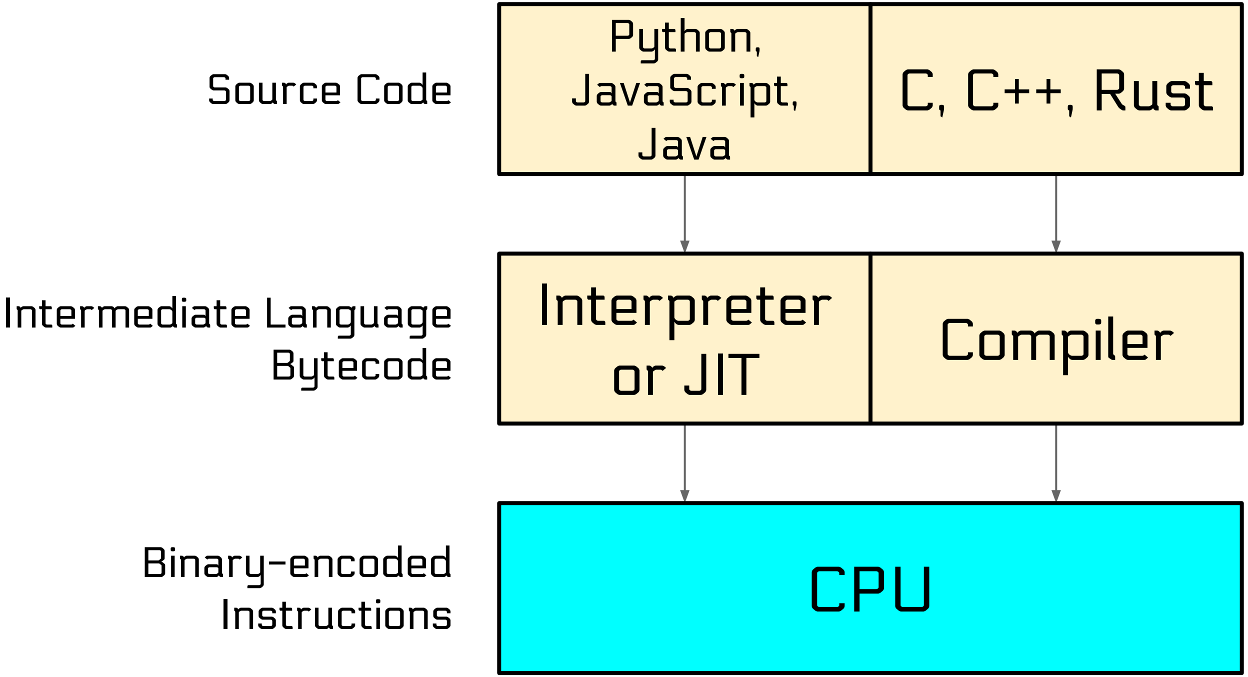
CSCI 297E: Ethical Hacking

Digging Deeper



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

All roads lead to the CPU



Source Code

Python, JavaScript, Java

C, C++, Rust

Intermediate Language
Bytecode

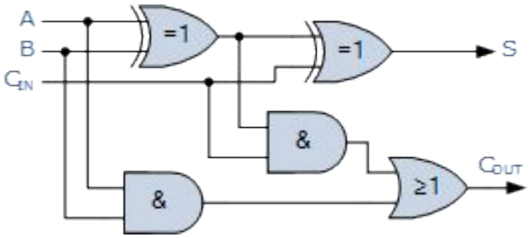
Interpreter or JIT

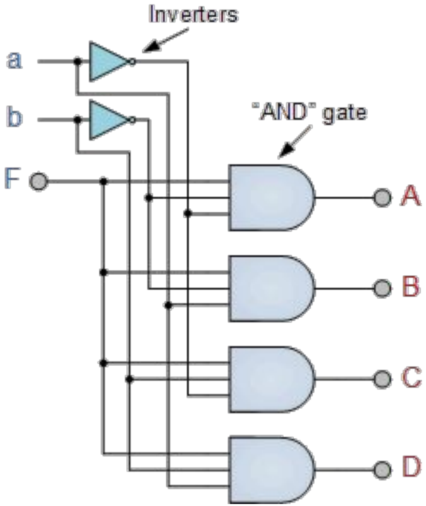
Compiler

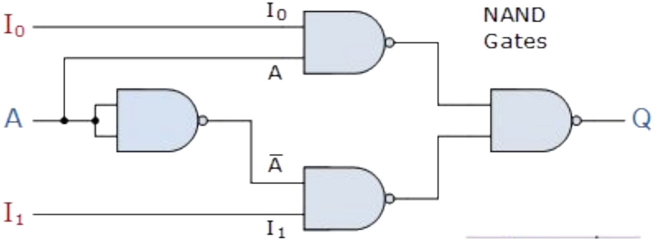
Binary-encoded
Instructions

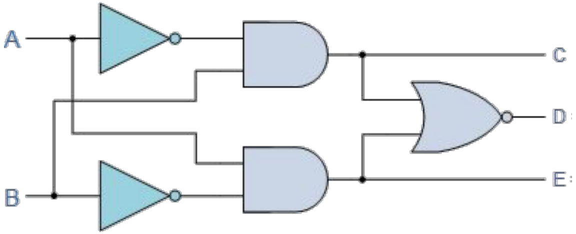
CPU

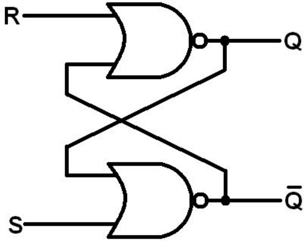
Too deep!





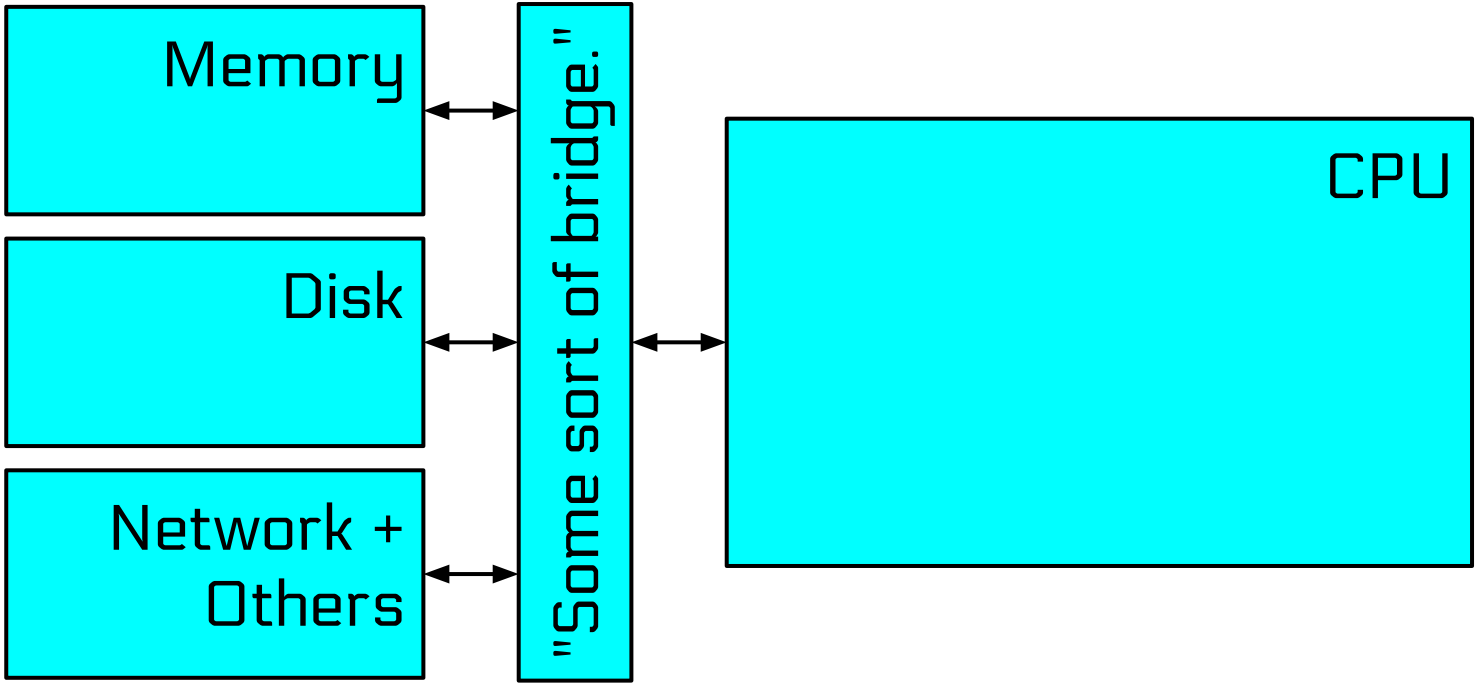






http://www.electronics-tutorials.ws/category/combination

Computer Architecture (very high level)



Memory

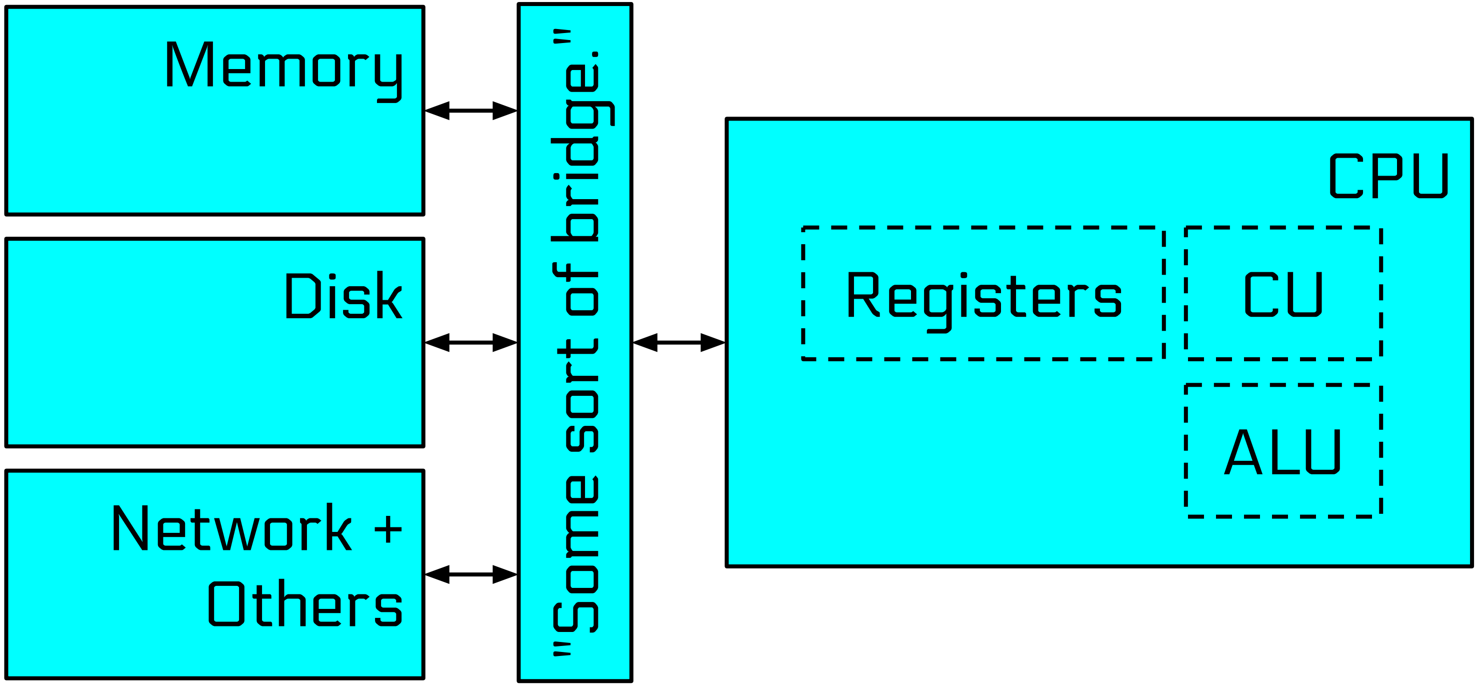
Disk

Network +
Others

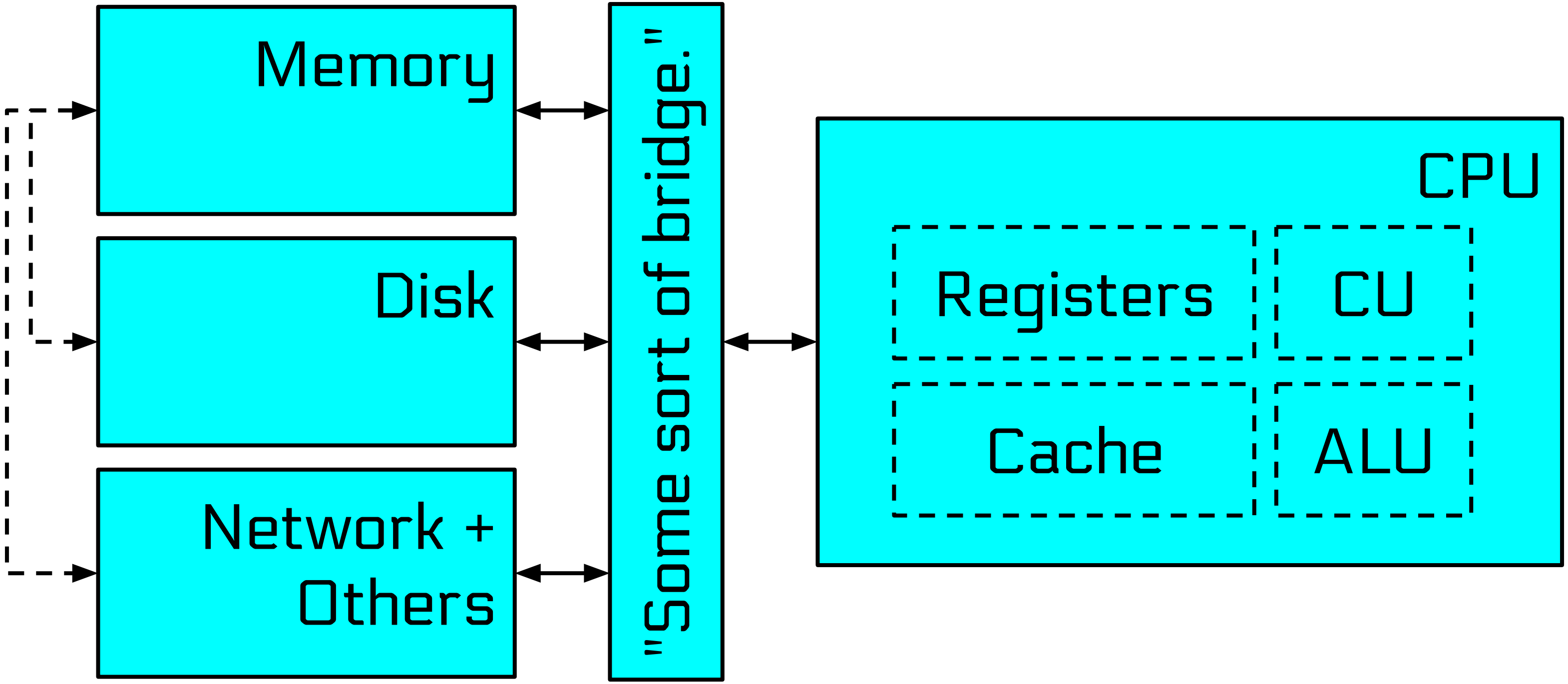
"Some sort of bridge."

CPU

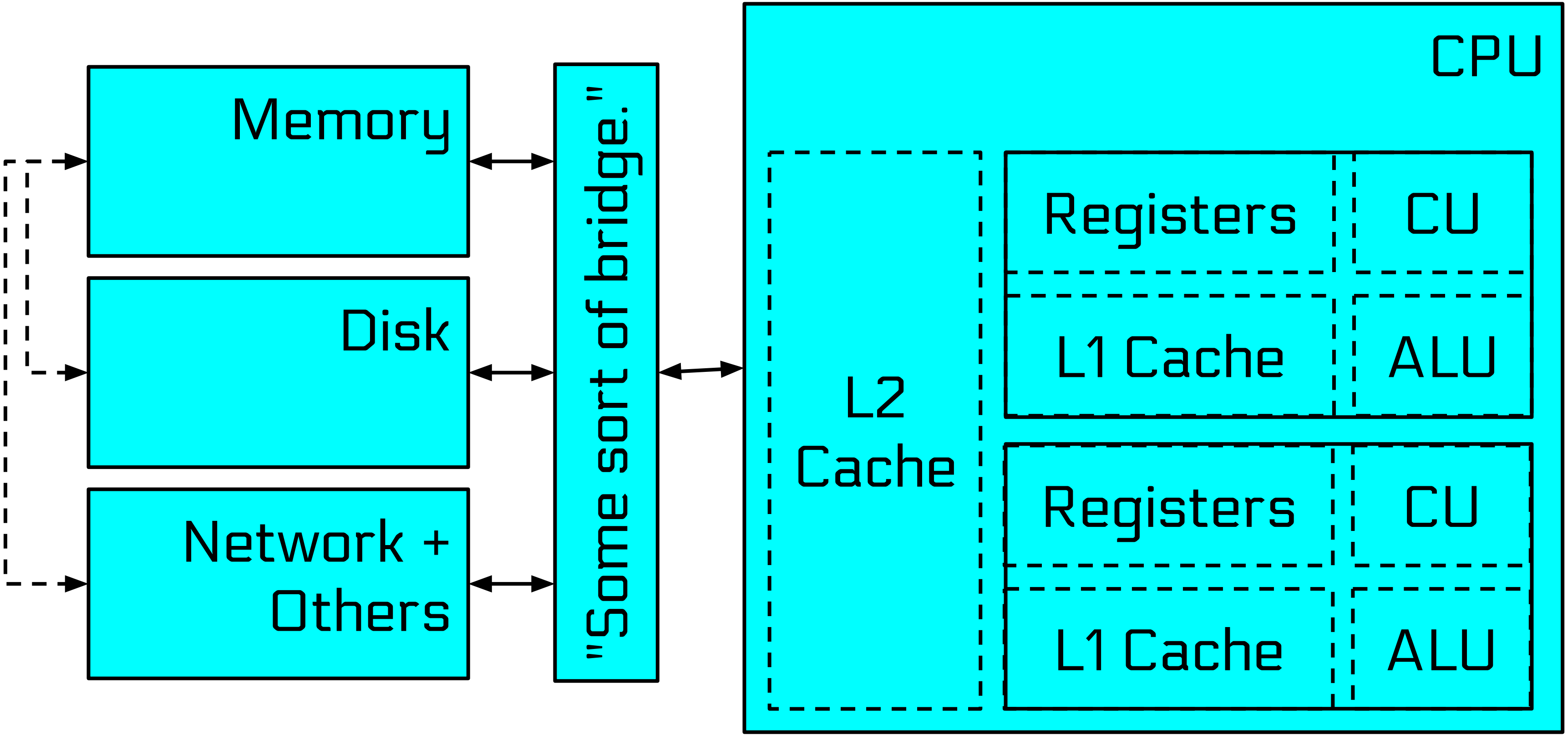
Computer Architecture (drilling down)

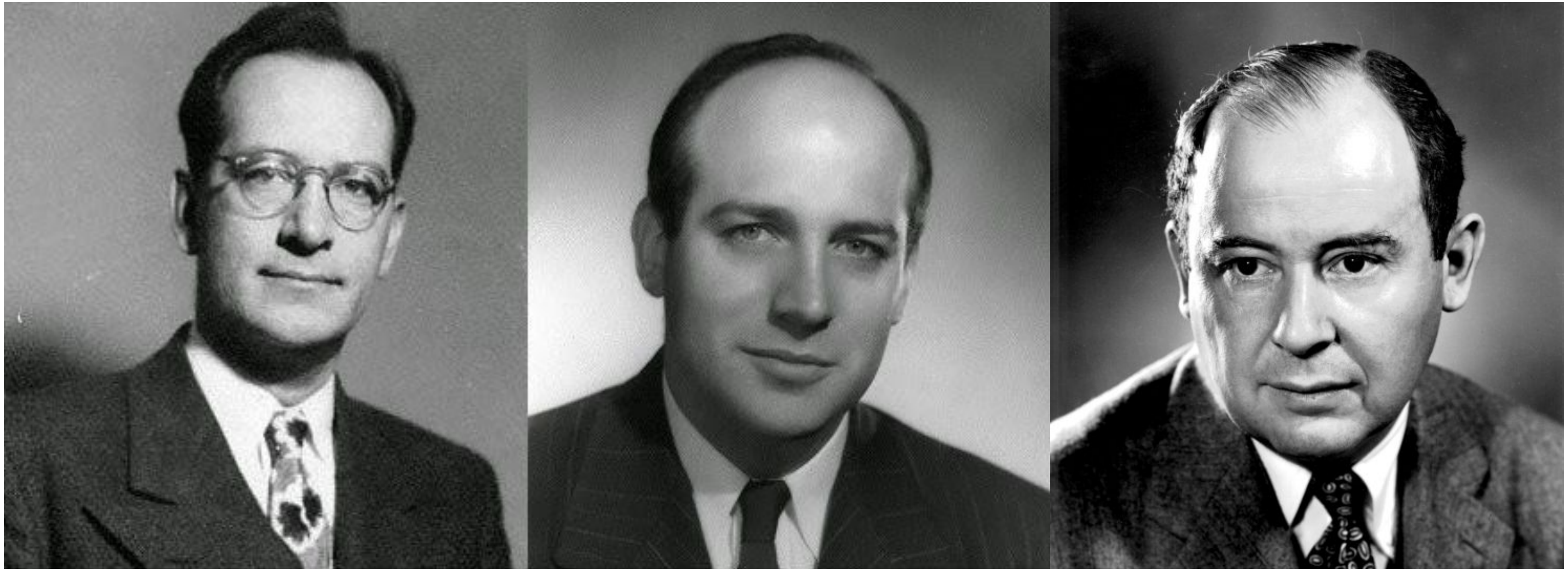


Computer Architecture (further down!)



Computer Architecture (as far as we'll go)

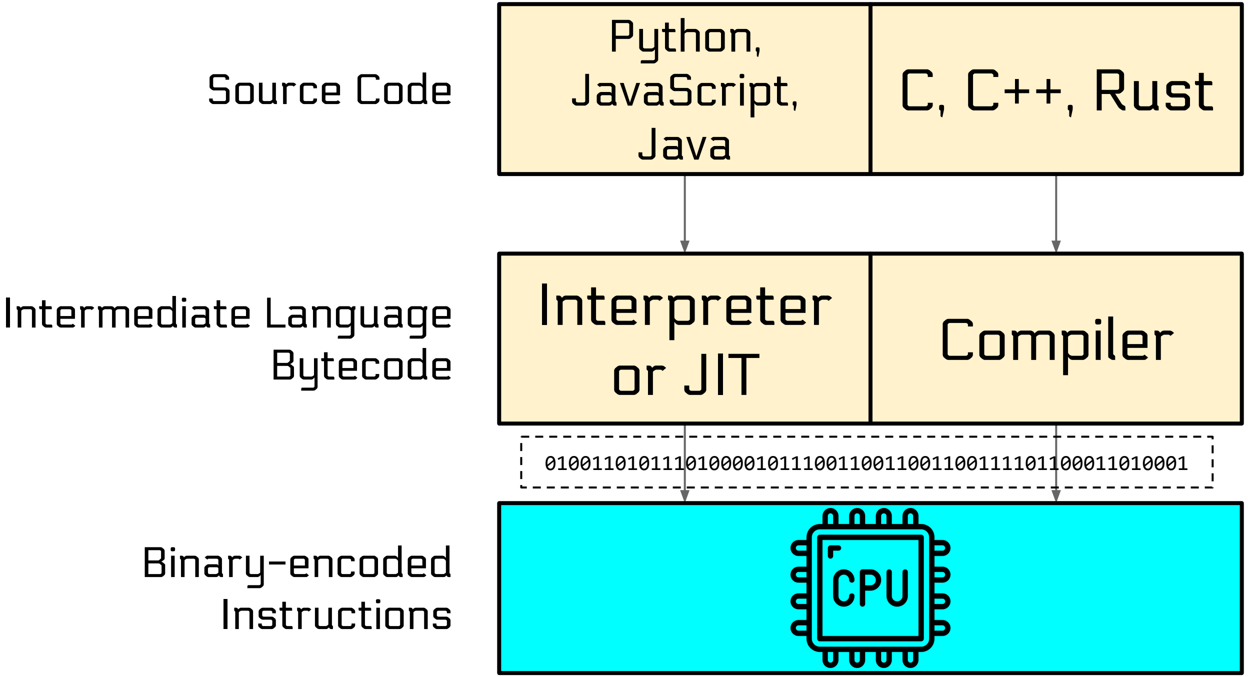




John Mauchly (Physicist), John Presper Eckert (Electrical Engineer), John Von Neumann (Mathematician)

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

All roads lead to the CPU



Source Code

Python, JavaScript, Java

C, C++, Rust

Intermediate Language Bytecode

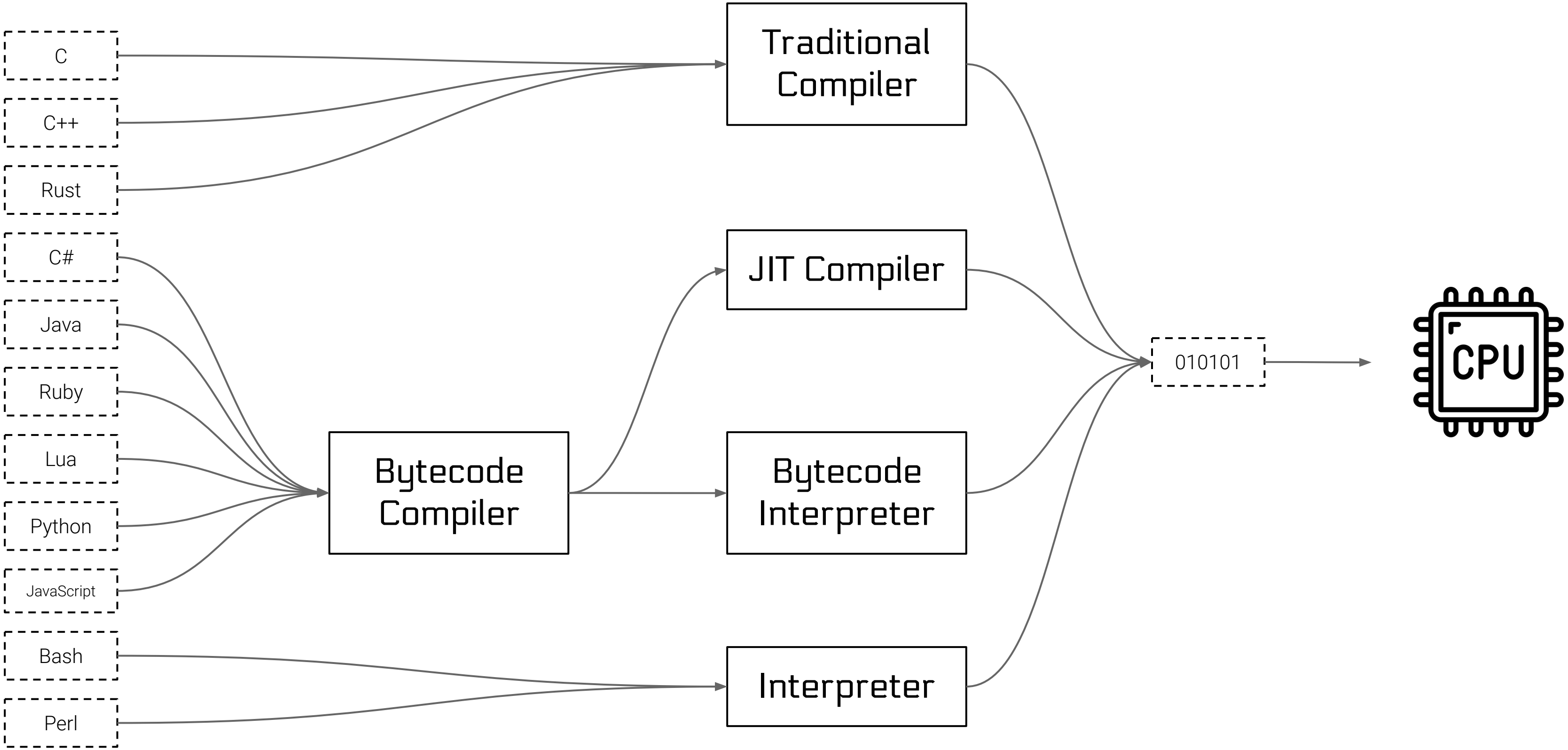
Interpreter or JIT

Compiler

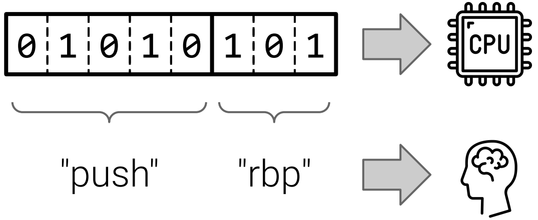
0100110101110100001011100110011001100111101100011010001

Binary-encoded Instructions

All Roads Lead to the CPU



Speaking Binary



Humans have a hard time with binary code...

So we created a text *representation* of the binary...

This representation is called **Assembly**.

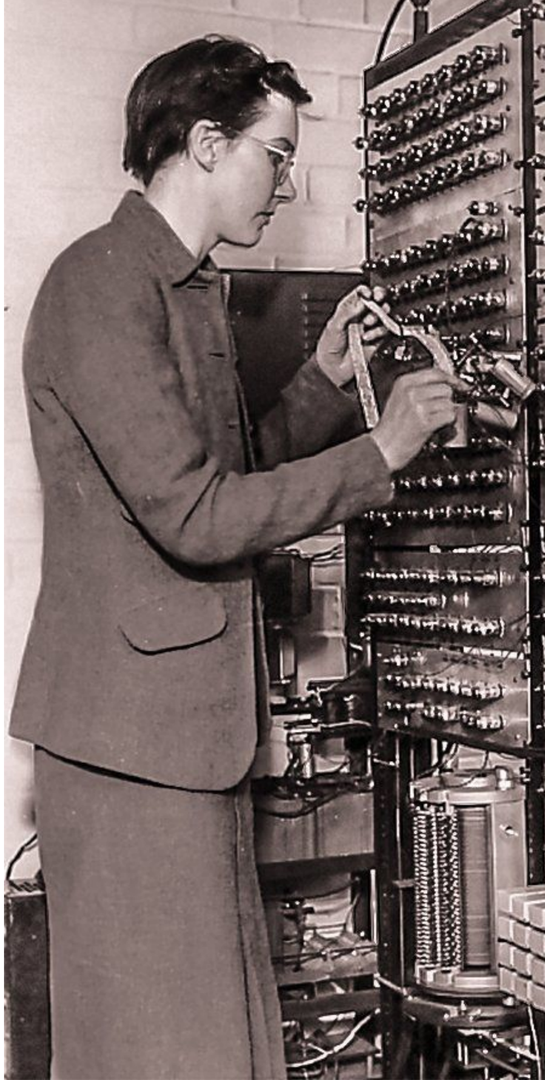
The binary and the assembly code is *equivalent**.

"Assembly"?

Assembly is named "Assembly" because it is *assembled* (*not* compiled) into binary code.

Invention:
Kathleen Booth,
Late 1940s/early 1950s,
For the APE(X)C (All-purpose Electronic (Rayon) Computer).

Adoption:
The second "stored-program computer" had an assembler,
Written by David Wheeler in 1948.



Assembly tells the CPU *what to do*

How do we tell people what to do? Sentences.

Let's look at an assembly "sentence" in terms of English grammar:

Sentence: we'll call this an "instruction" in assembly.
Verb: what do you want the instruction to do? We'll call this an "**operation**".
Noun: what do you want the instruction to do it *to*? We'll call this an "**operand**".

... that's it?
Simple!

Simplicity

Assembly is the **simplest** programming language.

It'd have to be, CPUs need to understand it!

You can master assembly in a week!

Nouns / Operands

What types of nouns might we deal with? Data!

For the most part, the CPU is concerned with three types of data:

data we directly give it as part of the instruction

data that is close at hand

data in storage

Verbs / Operations

What might you want to tell the computer to do with data?

Some ideas:
`add` some data together
`sub`tract some data
`mul`tiply some data
`div`ide some data
`mov`e some data into or out of storage
`c`o`mp`are two pieces of data with each other
`test` some other properties of data

Now you (almost) know some Assembly!

Assembly Dialects

Assembly is a direct translation of binary code ingested by the CPU...
... so it's very CPU architecture dependent.

Every architecture has its own variant:

x86 assembly
arm assembly
ppc assembly
mips assembly
risc-v assembly
pdp-11 assembly

The list goes on! Regardless of dialect, an assembly instruction looks like one of:

OPERATION
OPERATION OPERAND
OPERATION OPERAND OPERAND
OPERATION OPERAND OPERAND OPERAND

Dialects of Assembly Dialects

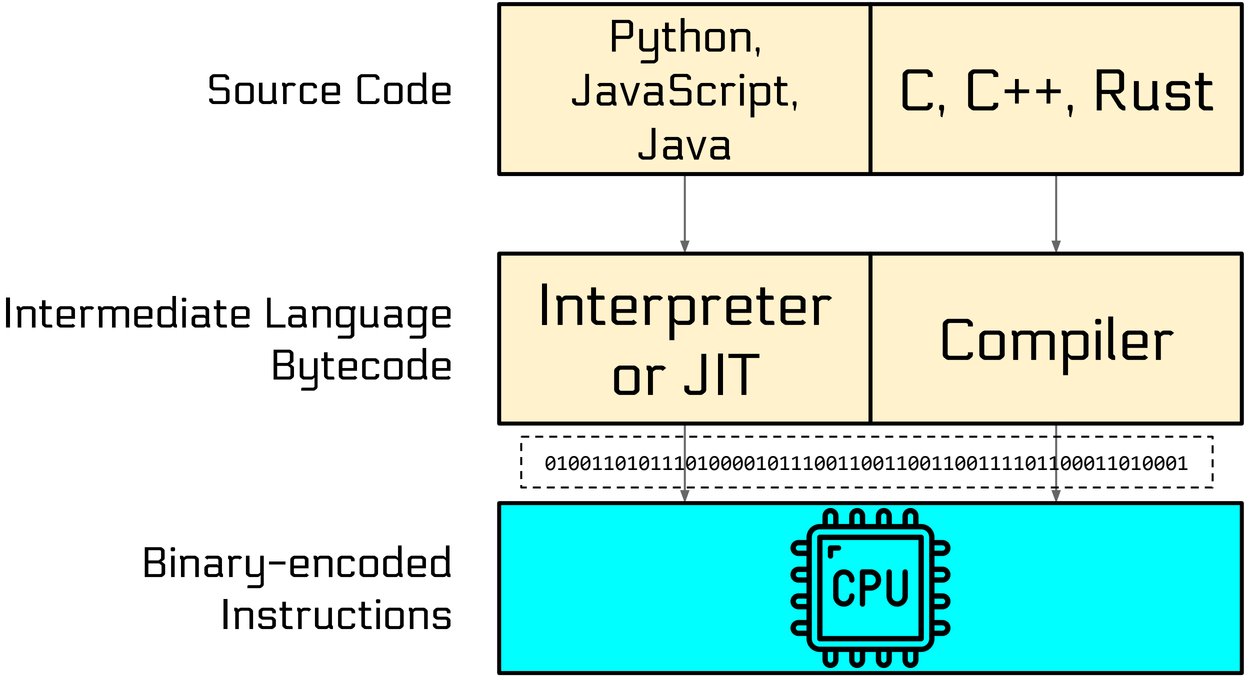
In the beginning (of x86), Intel created:
... the Intel 8085 CPU
... then the Intel 8086 CPU
... then the Intel 80186 CPU
... then the Intel 80286 CPU
... then the Intel 80386 CPU, which became modern x86
... and gave us a great Assembly dialect for all of them!

AT&T came along and created a (subjectively) TERRIBLE Assembly syntax for x86.
Why? No one knows.

tl;dr: there are two competing Assembly syntaxes for x86:
the right one (Intel) and the VERY WRONG one (AT&T).

Use Intel x86 syntax. They literally made the architecture.

All roads lead to the CPU



Source Code

Python,
JavaScript,
Java

C, C++, Rust

Intermediate Language
Bytecode

Interpreter
or JIT

Compiler

`0100110101110100001011100110011001100111101100011010001`

Binary-encoded
Instructions

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

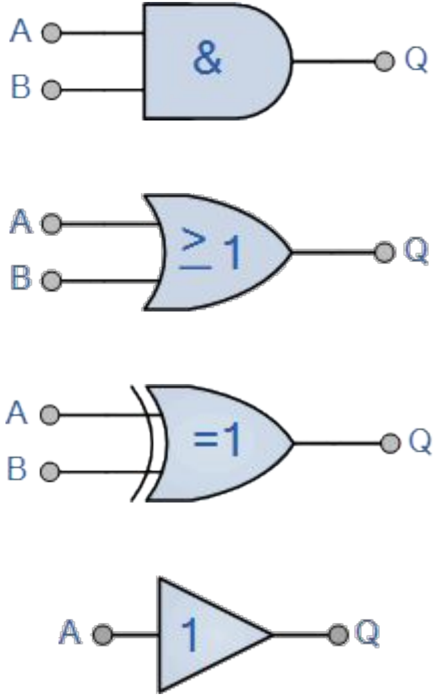
Decimal (base 10) has digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Binary (base 2) has digits 0, 1.

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 *ter*nary 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
```

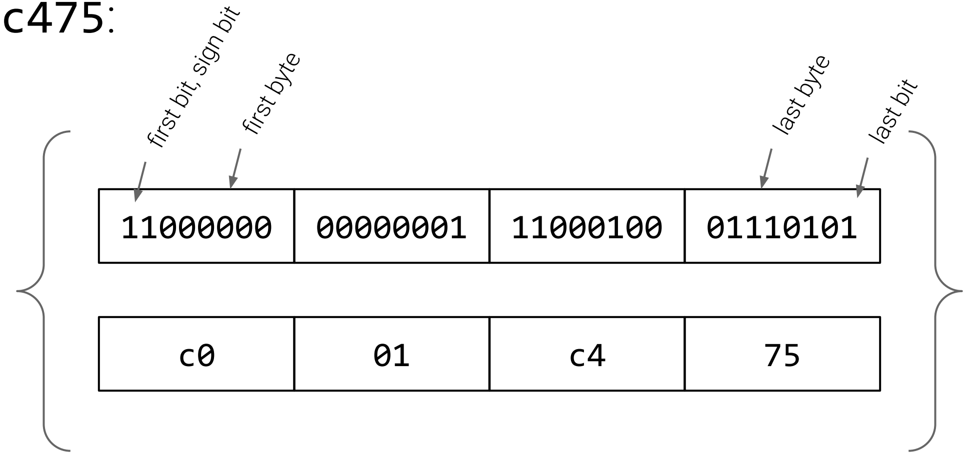
Advantage: arithmetic operations don't have to be sign-aware!
(unsigned) `0b00000000 - 1 = 0 - 1 = 255 == 0b11111111`
(signed) `0b00000000 - 1 = 0 - 1 = -1  == 0b11111111`

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`:



first bit, sign bit

first byte

last byte

last bit

11000000	00000001	11000100	01110101
c0	01	c4	75

"most significant" bits/bytes

"leftmost" bits/bytes

"high" bits/bytes

"least significant" bits/bytes

"rightmost" bits/bytes

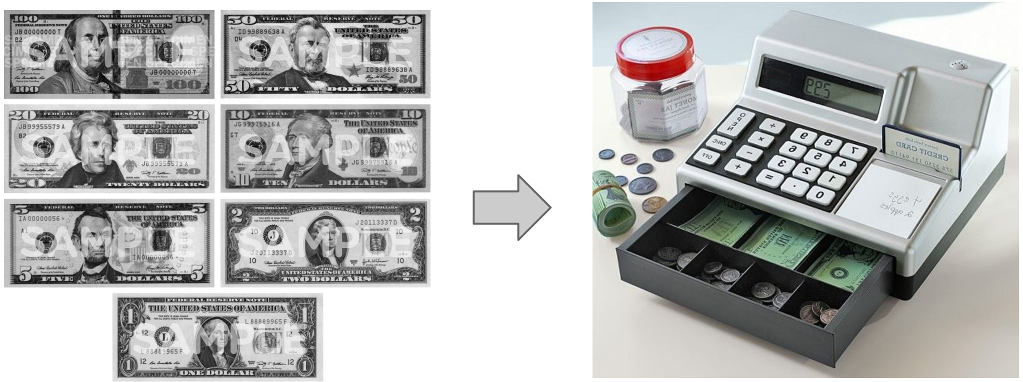
"low" bits/bytes

The Need for "Registers"

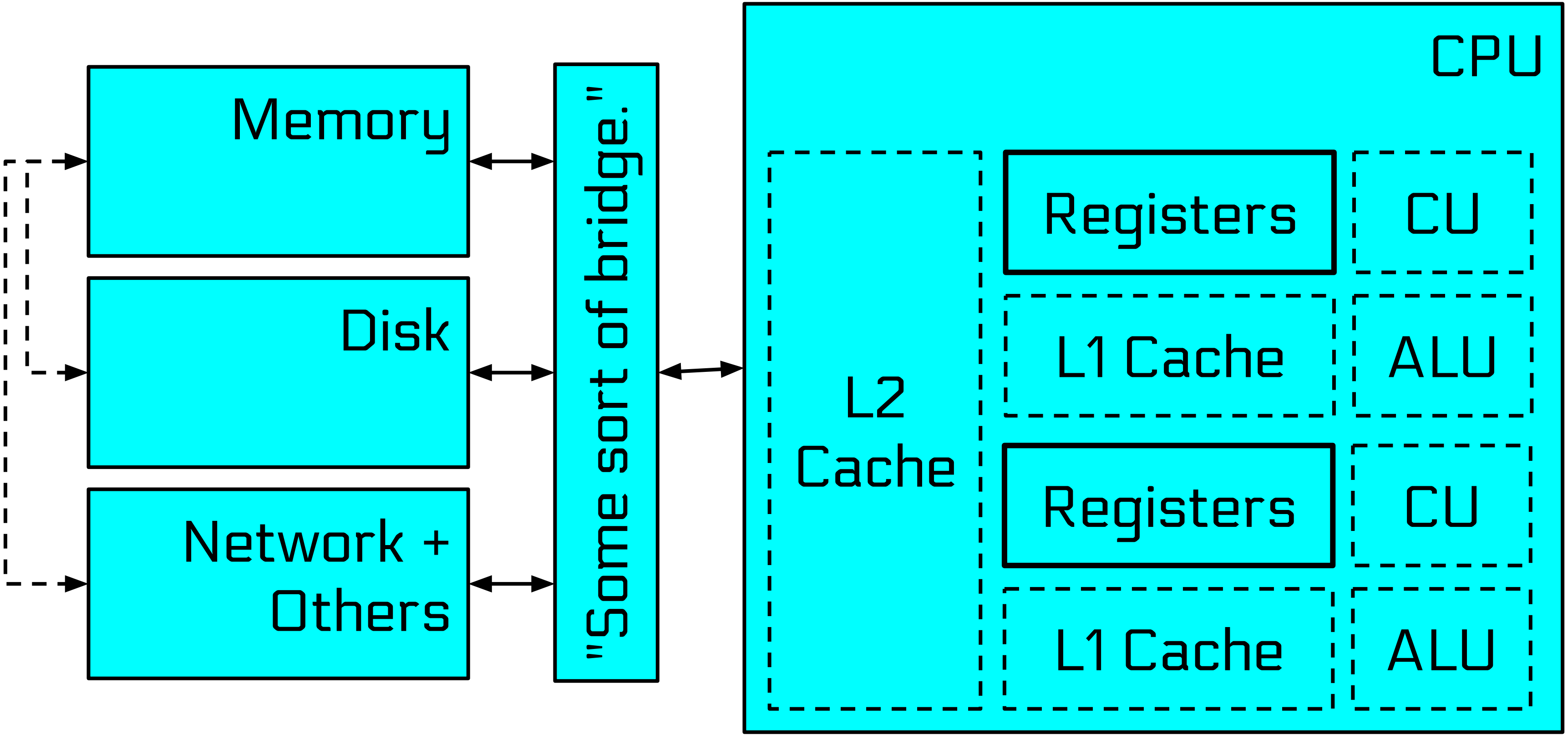
CPUs need to be *fast*.

To be fast, CPUs need rapid access to data they're working on.

This is done via the *Register File*.



Reminder: Computer Architecture



Registers

Registers are very fast, temporary stores for data.

You get several "general purpose" registers:

- 8085: a, c, d, b, e, h, l
- 8086: ax, cx, dx, bx, **sp**, **bp**, si, di
- x86: eax, ecx, edx, ebx, **esp**, **ebp**, esi, edi
- amd64: rax, rcx, rdx, rbx, **rsp**, **rbp**, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- arm: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, **r13**, **r14**

The address of the next instruction is in a register:

eip (x86), rip (amd64), r15 (arm)

Various extensions add other registers (x87, MMX, SSE, etc).

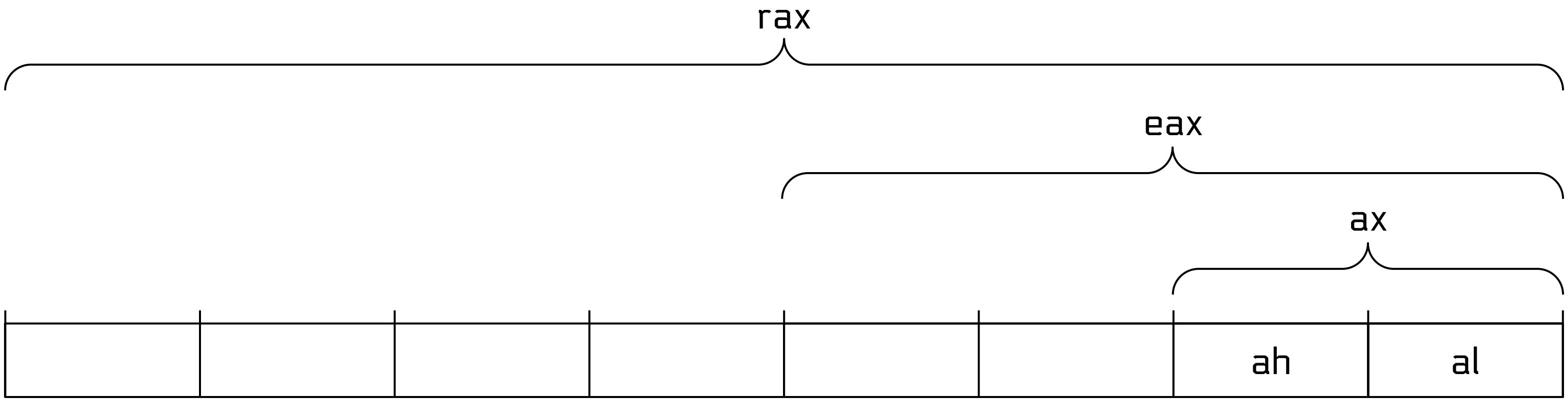
Register Size

Registers are (typically) the same size as the word width of the architecture.

On a 64-bit architecture (most) registers will hold 64 bits (8 bytes).

10110110	11011110	01111101	00000110	10110000	00111100	11110000	01000101

Partial Register Access



Registers can be accessed *partially*.

All partial accesses on amd64 (that I know of)

64	32	16	8H	8L
rax	eax	ax	ah	al
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rbx	ebx	bx	bh	bl
rsp	esp	sp		spl
rbp	ebp	bp		bpl
rsi	esi	si		sil
rdi	edi	di		dil
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

Setting Registers

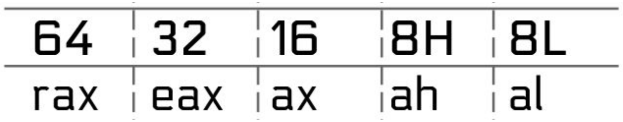
You load data into registers with... assembly! "`mov`" means "move".

```
mov rax, 0x539
mov rbx, 1337
```

Data specified directly in the instruction like this is called an **Immediate Value**.

You can also load data into partial registers:

```
mov ah, 0x5
mov al, 0x39
```



64	32	16	8H	8L
rax	eax	ax	ah	al

32-bit CAVEAT!

If you write to a 32-bit partial (e.g., `eax`), the CPU will *zero out* the rest of the register!
This was done for (believe it or not) performance reasons.

This sets `rax` to `0xffffffffffff0539`:

```
mov rax, 0xffffffffffffffff
mov ax, 0x539
```

This sets `rax` to `0x0000000000000539`:

```
mov rax, 0xffffffffffffffff
mov eax, 0x539
```

Shunting Data Around

You can also `mov` data between registers!

LINGUISTIC CAVEAT!
"mov" doesn't move the data, it copies it.

This sets both `rax` and `rbx` to 0x539 (1337).

```
mov rax, 0x539
mov rbx, rax
```

You can, of course, `mov` partials (32-bit clobber caveat applies)!
This sets rax to 0x539 and rbx to 0x39.

```
mov rax, 0x539
mov rbx, 0
mov bl, al
```

Extending Data...

Consider:

```
mov eax, -1
```

eax is now `0xffffffff` (both `4294967295` and `-1`) but...
rax is now `0x00000000ffffffff` (*only* `4294967295`)!

What if you wanted the operate on that `-1` in 64-bit land?

```
mov eax, -1
movsx rax, eax
```

`movsx` does a *sign-extending* move, preserving the Two's Complement value (i.e., copies the top bit to the rest of the register).

eax is now `0xffffffff` (both `4294967295` and `-1`) but...
rax is now `0xffffffffffffffff` (both `4294967295` and `-1`)!

Register Arithmetic

Once you have data in registers, you can *compute*!

For most arithmetic instructions, the first specified register stores the result.

Instruction	C / Math equivalent	Description
`add rax, rbx`	`rax = rax + rbx`	add rax to rbx
`sub ebx, ecx`	`ebx = ebx - ecx`	subtract ecx from ebx
`imul rsi, rdi`	`rsi = rsi * rdi`	multiple rsi to rdi, truncate to 64-bits
`inc rdx`	`rdx = rdx + 1`	increment rdx
`dec rdx`	`rdx = rdx - 1`	decrement rdx
`neg rax`	`rax = 0 - rax`	negate rax in terms of numerical value
`not rax`	`rax = ~rax`	negate each bit of rax
`and rax, rbx`	`rax = rax & rbx`	bitwise AND between the bits of rax and rbx
`or rax, rbx`	`rax = rax | rbx`	bitwise OR between the bits of rax and rbx
`xor rcx, rdx`	`rcx = rcx ^ rdx`	bitwise XOR (don't confuse ^ for exponent!)
`shl rax, 10`	`rax = rax << 10`	shift rax's bits left by 10, filling with 10 zeroes on the right
`shr rax, 10`	`rax = rax >> 10`	shift rax's bits right by 10, filling with 10 zeroes on the left
`sar rax, 10`	`rax = rax >> 10`	shift rax's bits right by 10, *with sign-extension to fill the now "missing" bits!*
`ror rax, 10`	`rax = (rax >> 10) | (rax << 54)`	rotate the bits of rax right by 10
`rol rax, 10`	`rax = (rax << 10) | (rax >> 54)`	rotate the bits of rax left by 10

Curious how these work? Play around with the rappel tool (https://github.com/yrp604/rappel)!

Some Registers are Special

You cannot directly read from or write to `rip`.
Contains the memory address of the next instruction to be executed (ip = Instruction Pointer).

You should be careful with `rsp`.
Contains the address of an region of memory to store temporary data (sp = Stack Pointer).

Some other registers are, by convention, used for important things.

More on this later in this module!

Other Registers Exist!

Modern x86 processors have a lot of other registers!

Registers for use by the Operating System itself (stay tuned for Kernel Security!).

Registers for *floating point* computation.

Registers for crunching large data fast.
32 512-bit "zmm" registers!

#

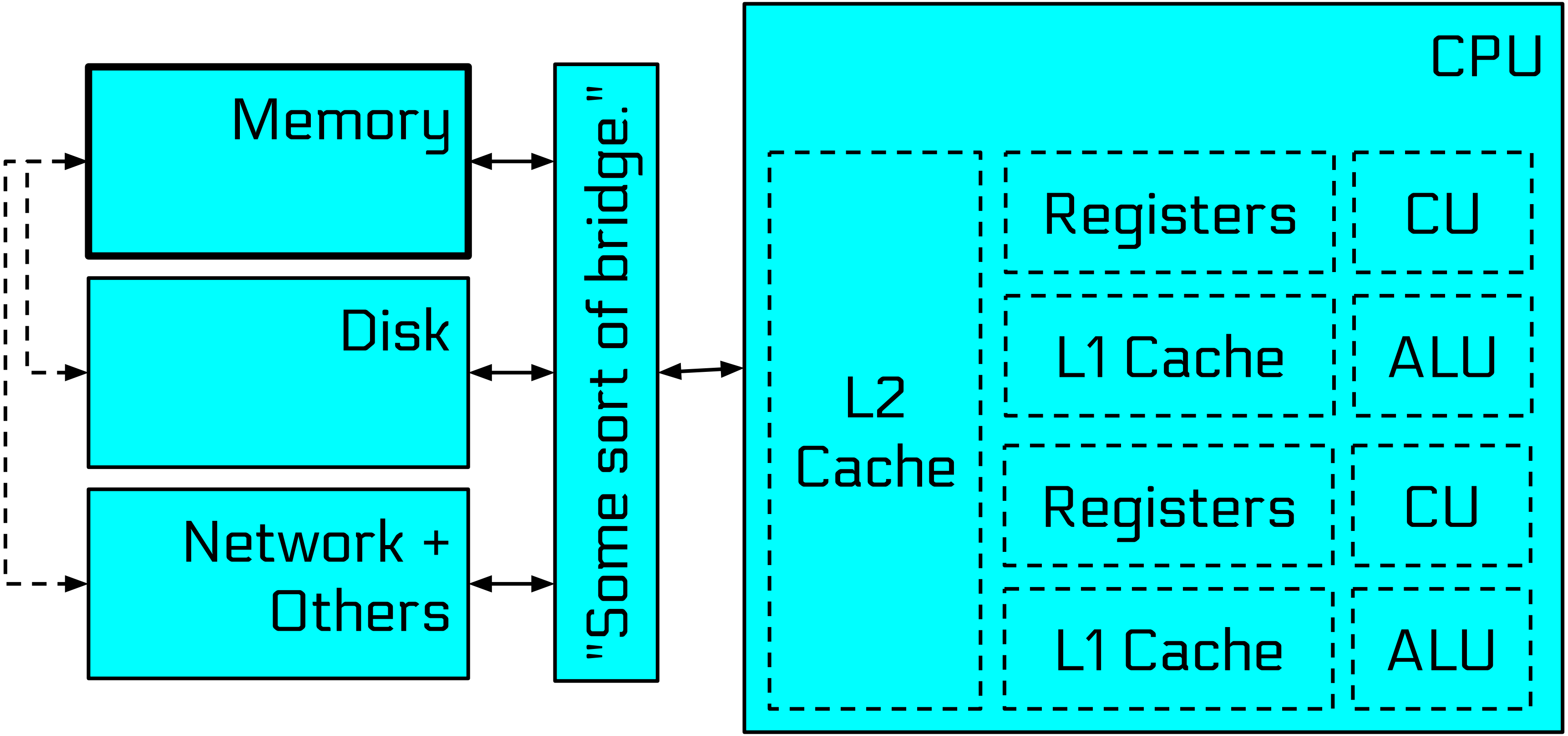
The Need for "Memory"

Registers are *expensive*, and we have a limited number of them.

We need a place to store lots of data and have *fairly fast* access to it when needed.

This place is system Memory.

Reminder: Computer Architecture



Memory: Process Perspective

Your process memory is used for A LOT:

Memory ↔ Registers
Memory ↔ Disk
Memory ↔ Network
Memory ↔ Video Card

There is too much memory to name every location (unlike registers).

Process memory is *addressed* linearly.
From: `0x10000` (for security reasons)
To: `0x7fffffffffff` (for architecture / OS purposes)

Each memory *address* references **one byte** in memory.
This means 127 *terabytes* of addressable RAM!

A Process' Memory

You don't have 127 TB of RAM... But that's okay, cause it's all ~~fake~~ ~~pretend~~ virtual!

Your process' memory starts out partially filled in by the Operating System.

Your process can ask for more memory from the Operating System (more on this later)!

Memory (stack)

The stack has several uses. For now, we'll talk about *temporary data storage*.

Registers and immediates can be **`push`**ed onto the stack to save values:

```
mov rax, 0xc001ca75
push rax
push 0xb0bacafe # WARNING: even on 64-bit x86, you can only push 32-bit immediates...
push rax
```

(Like mov, push leaves the value in the src register intact.)

Values can be **`pop`**ped back off of the stack (to any register!).

```
pop rbx # sets rbx to 0xc001ca75
pop rcx # sets rcx to 0xb0bacafe
```

Addressing the Stack

The CPU knows where the stack is because its address is stored in `rsp`.

`push 0xb0bacafe`

`pop rcx`

Historical oddity: the stack grows backwards toward smaller memory addresses!

push *decreases* rsp, **pop** *increases it*.

Accessing Memory

You can also move data between registers and memory with ... `mov`!

This will load the 64-bit value stored at memory address `0x12345` into `rbx`:

```
mov rax, 0x12345
mov rbx, [rax]
```

This will store the 64-bit value in `rbx` into memory at address `0x133337`:

```
mov rax, 0x133337
mov [rax], rbx
```

This is equivalent to push `rcx`:

```
sub rsp, 8
mov [rsp], rcx
```

Each addressed memory location contains one byte.

An 8-byte write at address `0x133337` will write to addresses
`0x133337` through `0x13333f`.

Controlling Write Sizes

You can use partials to store/load fewer bits!

Load 64 bits from addr `0x12345` and store the lower 32 bits to addr `0x133337`.

```
mov rax, 0x12345
mov rbx, [rax]
mov rax, 0x133337
mov [rax], ebx
```

Load 8 bits from addr `0x12345` to `bh`.

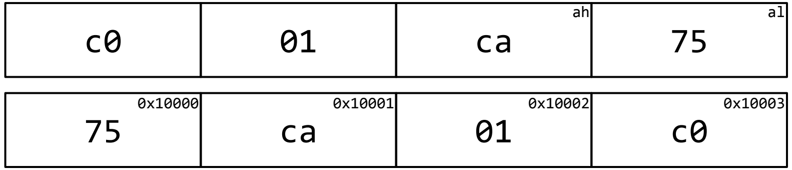
```
mov rax, 0x12345
mov bh, [rax]
```

Don't forget: changing 32-bit partials (e.g., by loading from memory) zeroes out the whole 64-register. Storing 32-bits to memory has no such problems, though.

Memory Endianess

Data on most modern systems is stored *backwards*, in *little endian*.

```
mov eax, 0xc001ca75 # sets rax to
mov rcx, 0x10000
mov [rcx], eax    # stores data as
mov bh, [rcx] # reads 0x75
```



c0	01	ah ca	al 75
0x10000 75	0x10001 ca	0x10002 01	0x10003 c0

Bytes are *only* shuffled for multi-byte stores and loads of registers to memory!
Individual bytes *never* have their bits shuffled.
Yes, writes to the stack behave just like any other write to memory.

Why little endian?

Intel created the 8008 for a company called Datapoint in 1972.
Datapoint used little endian for easier implementation of *carry* in arithmetic!
Intel used little endian in 8008 for compatibility with Datapoint's processes!
Every step in the evolution between 8008 and modern x86 maintained some level of binary compatibility with its predecessor.

https://en.wikipedia.org/wiki/Endianness

Address Calculation

You can do some limited calculation for memory addresses.

Use `rax` as an offset off some base address (in this case, the stack).

```
mov rax, 0
mov rbx, [rsp+rax*8] # read a qword right at the stack pointer
inc rax
mov rcx, [rsp+rax*8] # read the qword to the right of the previous one
```

You can get the calculated address with Load Effective Address (`lea`).

```
mov rax, 1
pop rcx
lea rbx, [rsp+rax*8+5] # rbx now holds the computed address for double-checking
mov rbx, [rbx]
```

Address calculation has limits.
`reg+reg*(2 or 4 or 8)+value` is as good as it gets.

RIP-Relative Addressing

`lea` is one of the few instructions that can directly access the rip register!

```
lea rax, [rip] # load the address of the next instruction into rax
lea rax, [rip+8] # the address of the next instruction, plus 8 bytes
```

You can also use **`mov`** to read directly from those locations!

```
mov rax, [rip] # load 8 bytes from the location pointed to by the address of the next instruction
```

Or even *write* there!

```
mov [rip], rax # write 8 bytes over the next instruction (CAVEATS APPLY)
```

This is useful for working with data embedded near your code!

This is what makes certain security features on modern machines *possible*.

Writing Immediate Values

You can also write immediate values. However, you must specify their size!

This writes a 32-bit 0x1337 (padded with 0 bits) to address 0x133337.

```
mov rax, 0x133337
mov DWORD PTR [rax], 0x1337
```

Depending on your assembler, it might expect `DWORD` instead of `DWORD PTR`.

Other Memory Regions

Other regions might be mapped in memory!

We previously talked about regions loaded due to directives in the ELF headers, but functionality such as `mmap` and `malloc` can cause other regions to be mapped as well.

These will feature prominently (and be discussed) in future modules.

#

Computers Make Decisions

```
if (authenticated) {
    leetness = 1337;
}
else {
    leetness = 0;
}
```

So far, we've just shunted data around.

But how do we make decisions?

What to Execute?

First, let's look at how computers execute instructions.

Recall: Assembly instructions are direct translations of binary code.

This binary code lives in *memory*.

Example:

	0x400800				
Program Binary Code	`pop rax`	`pop rbx`	`add rax, rbx`	`push rax`	

This is (in hex):

	0x400800	0x400801	0x400802	0x400805	
Program Binary Code	`58`	`5b`	`48 01 d8`	`50`	

Control Flow: Jumps

CPUs execute instructions in sequence *until told not to*.

One way to interrupt the sequence is with a `jmp` instruction:

```
  mov cx, 1337
  jmp STAY_LEET
  mov cx, 0
STAY_LEET:
  push rcx
```

	0x400800			STAY_LEET	
Program Binary Code	mov rcx, 0x1337	jmp STAY_LEET	mov rcx, 0	push rcx	

	0x400800	0x400804	0x400806	STAY_LEET 0x40080a	
Program Binary Code	66 b9 37 13	eb 04 (skip 4 bytes)	66 b9 00 00	51	

`jmp` skips X bytes and then resumes execution!
But that's still not enough for decisions...

Control Flow: *Conditional* Jumps!

Jumps can rely on conditions!

```
mov cx, 1337
jnz STAY_LEET
mov cx, 0
STAY_LEET:
push rcx
```

	0x400800			STAY_LEET	
Program Binary Code	`mov rcx, 0x1337`	`jmp STAY_LEET`	`mov rcx, 0`	`push rcx`	

	0x400800	0x400804	0x400806	STAY_LEET 0x40080a	
Program Binary Code	`66 b9 37 13`	`75 04`	`66 b9 00 00`	`51`	

je	jump if equal
jne	jump if not equal
jg	jump if greater
jl	jump if less
jle	jump if less than or equal
jge	jump if greater than or equal
ja	jump if above (unsigned)
jb	jump if below (unsigned)
jae	jump if above or equal (unsigned)
jbe	jump if below or equal (unsigned)
js	jump if signed
jns	jump if not signed
jo	jump if overflow
jno	jump if not overflow
jz	jump if zero
jnz	jump if not zero

`jnz` is "jump if not zero", but if **what** is not zero?

Control Flow: Conditions

Conditional jumps check Conditions stored in the "flags" register: `rflags`.

Flags are updated by:

Most arithmetic instructions.
Comparison instruction cmp (`sub`, but discards result).
Comparison instruction test (`and`, but discards result).

Main conditional flags:

Carry Flag: was the 65th bit 1?
Zero Flag: was the result 0?
Overflow Flag: did the result "wrap" between positive to negative?
Signed Flag: was the result's signed bit set (i.e., was it negative)?

Common patterns:

```
cmp rax, rbx; ja STAY_LEET # unsigned rax > rbx. 0xffffffff >= 0
cmp rax, rbx; jle STAY_LEET # signed rax <= rbx. 0xffffffff = -1 < 0
test rax, rax; jnz STAY_LEET # rax != 0
cmp rax, rbx; je STAY_LEET # rax == rbx
```

Thanks to Two's Complement, only the *jumps themselves* have to be signedness-aware.

je	jump if equal	ZF=1
jne	jump if not equal	ZF=0
jg	jump if greater	ZF=0 and SF=OF
jl	jump if less	SF!=OF
jle	jump if less than or equal	ZF=1 or SF!=OF
jge	jump if greater than or equal	SF=OF
ja	jump if above (unsigned)	CF=0 and ZF=0
jb	jump if below (unsigned)	CF=1
jae	jump if above or equal (unsigned)	CF=0
jbe	jump if below or equal (unsigned)	CF=1 or ZF=1
js	jump if signed	SF=1
jns	jump if not signed	SF=0
jo	jump if overflow	OF=1
jno	jump if not overflow	OF=0
jz	jump if zero	ZF=1
jnz	jump if not zero	ZF=0

Looping!

With our conditional jumps, we can implement a loop (think: `for`, `while`, etc)!

Example: this counts to 10!

```
mov rax, 0
LOOP_HEADER:
inc rax
cmp rax, 10
jb LOOP_HEADER
# now rax is 10!
```

With looping and conditional control flow, we have almost everything we need to write anything we want!

Control Flow: Function Calls!

Assembly code is split into functions with `call` and `ret`.

`call` pushes `rip` (address of the next instruction after the call) and jumps away!
`ret` pops `rip` and jumps to it!

Using a function that takes an `authenticated` value and returns `leetness`:

```
mov rdi, 0
call FUNC_CHECK_LEET
mov rdi, 1
call FUNC_CHECK_LEET
call EXIT

FUNC_CHECK_LEET:
  test rdi, rdi
  jnz LEET
  mov ax, 0
  ret
  LEET:
  mov ax, 1337
  ret

FUNC_EXIT:
  ???
```

```
int check_leet(int authed) {
  if (authed) return 1337;
  else return 0;
}

int main() {
  check_leet(0);
  check_leet(1);
  exit();
}
```

Calling Conventions

Callee and caller functions must agree on argument passing.

Linux x86: push arguments (in reverse order), then call (which pushes return address), return value in eax
Linux amd64: rdi, rsi, rdx, rcx, r8, r9, return value in rax
Linux arm: r0, r1, r2, r3, return value in r0

Registers are *shared* between functions, so calling conventions should agree on what registers are protected.

Linux amd64.
`rbx`, `rbp`, `r12`, `r13`, `r14`, `r15` are "callee-saved"
(the function you call keeps their values safe on the stack).
Other registers are up for grabs
(within reason; e.g., `rsp` must be maintained). Save their values (on the stack)!

Having Effects

```
exit();
```

How do we interact with the outside world?

Even something as simple as quitting the program?

System Calls

Remember system calls? It's an instruction that makes a *call* into the Operating System.
`syscall` triggers the system call specified by the value in `rax`.
arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`
return value in `rax`

Reading 100 bytes from stdin to the stack:

```
n = read(0, buf, 100);
mov rdi, 0 # the stdin file descriptor
mov rsi, rsp # read the data onto the stack
mov rdx, 100 # the number of bytes to read
mov rax, 0 # system call number of read()
syscall # do the system call
```

`read` returns the number of bytes read via `rax`, so we can easily `write` them out:

```
write(1, buf, n);
mov rdi, 1 # the stdout file descriptor
mov rsi, rsp # write the data from the stack
mov rdx, rax # the number of bytes to write (same as what we read in)
mov rax, 1 # system call number of write()
syscall # do the system call
```

Critical resource: https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

System Calls

System calls have very well-defined interfaces that very rarely change.

There are over 300 system calls in Linux. Here are some examples:

`int open(const char *pathname, int flags)` - returns a file new file descriptor of the open file (also shows up in /proc/self/fd!)
`ssize_t read(int fd, void *buf, size_t count)` - reads data from the file descriptor
`ssize_t write(int fd, void *buf, size_t count)` - writes data to the file descriptor
`pid_t fork()` - forks off an *identical* child process. Returns 0 if you're the child and the PID of the child if you're the parent.
`int execve(const char *filename, char **argv, char **envp)` - *replaces* your process.
`pid_t wait(int *wstatus)` - wait child termination, return its PID, write its status into *wstatus.

Look familiar?

Critical resource: https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

"String" Arguments

Some system calls take "string" arguments (for example, file paths).

A string is a bunch of contiguous bytes in memory, followed by a `0` byte.

Let's build a file path for `open()` on the stack:

```
mov BYTE PTR [rsp+0], '/' # write the ASCII value of / onto the stack
mov BYTE PTR [rsp+1], 'f'
mov BYTE PTR [rsp+2], 'l'
mov BYTE PTR [rsp+3], 'a'
mov BYTE PTR [rsp+4], 'g'
mov BYTE PTR [rsp+5], 0 # write the 0 byte that will terminate our string
```

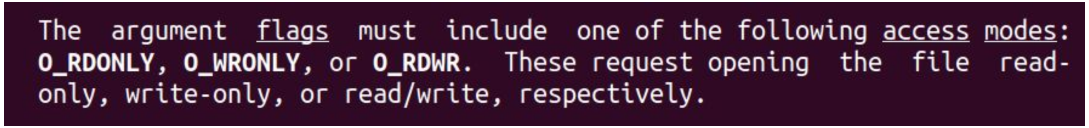
rsp	rsp+1	rsp+2	rsp+3	rsp+4	rsp+5
2f (/)	66 (f)	6c (l)	61 (a)	67 (g)	00 (\0)

Now, we can `open()` the `/flag` file!

```
mov rdi, rsp # read the data onto the stack
mov rsi, 0 # open the file read only (more on this later)
mov rax, 2 # system call number of open()
syscall # do the system call
```

`open()` returns the file descriptor number in `rax`

Constant Arguments



The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

Some system calls require archaic "constants".

Example: `open()` has a flags argument to determine how the file will be opened.

We can figure out the values of these arguments in C!

```
#include <stdio.h>
#include <fcntl.h>
int main() {
 printf("O_RDONLY is: %d\n", O_RDONLY);
}
```

```
yans@ramoth ~/pwn $ ./print_rdonly
O_RDONLY is: 0
yans@ramoth ~/pwn $ 
```

Quitting The Program

Finally, we can quit!

```
mov rdi, 42 # our program's return code (e.g., for bash scripts)
mov rax, 60 # system call number of exit()
syscall # do the system call
```

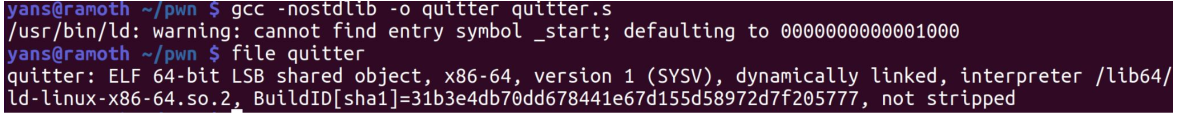
Goodbye, world!

From Assembly to Binary

We built a quitter... Now we have to put it in an Assembly file:

```
# .intel_syntax tells the assembler that we are using Intel assembly syntax
# noprefix tells it that we will not prefix all register names with "%" (cause that looks silly)
.intel_syntax noprefix
mov rdi, 42 # our program's return code (e.g., for bash scripts)
mov rax, 60 # system call number of exit()
syscall # do the system call
```

Assembly is named after the Assembler. Let's use the assembler!



```
yans@ramoth ~/pwn $ gcc -nostdlib -o quitter quitter.s
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000001000
yans@ramoth ~/pwn $ file quitter
quitter: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/
ld-linux-x86-64.so.2, BuildID[sha1]=31b3e4db70dd678441e67d155d58972d7f205777, not stripped
```

If that warning from ld annoys you, add this to the beginning of the program so that gcc doesn't have to guess at where your code starts:

```
.global _start
_start:
# then the rest of your code!
```

You've built your first assembly program!

Running the Program

Your program runs like any other...

```
# ./quitter
```

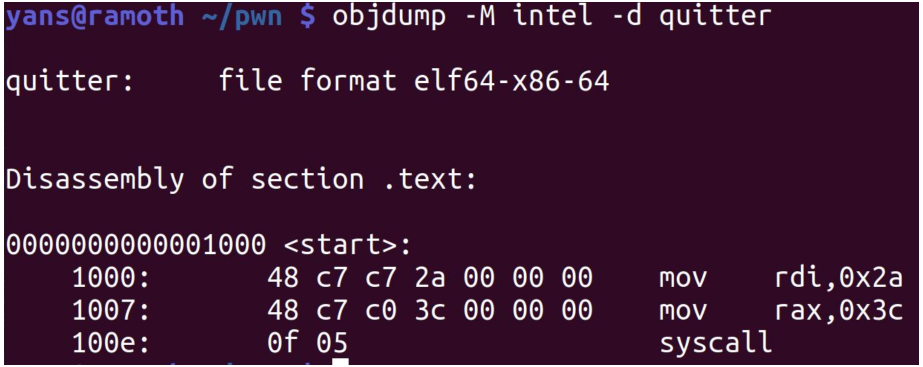
You can check its return code with bash's special $? variable!

```
# ./quitter
# echo $?
42
```

Reading Assembly

You can *disassemble* your program!

```
# objdump -M intel -d quitter
```



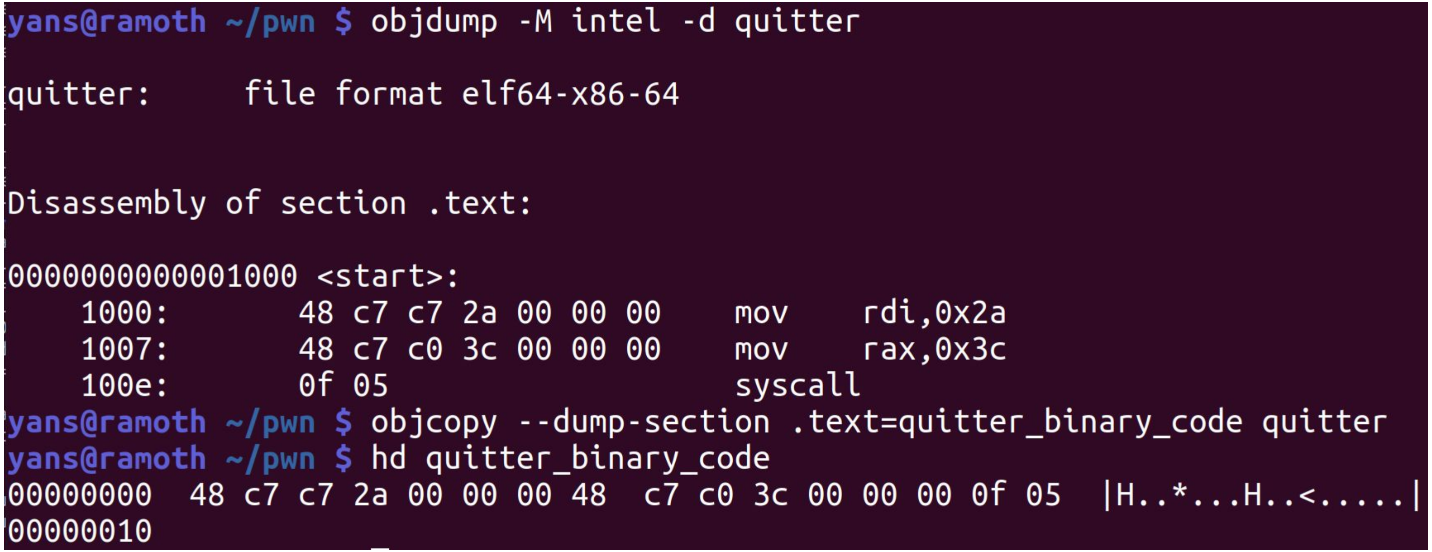
```
yans@ramoth ~/pwn $ objdump -M intel -d quitter

quitter:     file format elf64-x86-64


Disassembly of section .text:

0000000000001000 <start>:
    1000:       48 c7 c7 2a 00 00 00    mov    rdi,0x2a
    1007:       48 c7 c0 3c 00 00 00    mov    rax,0x3c
    100e:       0f 05                   syscall
```

Extracting the Binary Code

gcc builds your Assembly into a full ELF program.

You can extract *just* your binary code:

```
# objcopy --dump-section .text=quitter_binary_code quitter
```



```
yans@ramoth ~/pwn $ objdump -M intel -d quitter

quitter:     file format elf64-x86-64


Disassembly of section .text:

0000000000001000 <start>:
    1000:       48 c7 c7 2a 00 00 00    mov    rdi,0x2a
    1007:       48 c7 c0 3c 00 00 00    mov    rax,0x3c
    100e:       0f 05                   syscall
yans@ramoth ~/pwn $ objcopy --dump-section .text=quitter_binary_code quitter
yans@ramoth ~/pwn $ hd quitter_binary_code
00000000  48 c7 c7 2a 00 00 00 48  c7 c0 3c 00 00 00 0f 05  |H..*...H..<.....|
00000010
```

Bugs in the Program

Your program might have errors! This has been prophesied for centuries:

> *... an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.*
>
> - Ada Lovelace, Notes on the Analytical Engine, 1843

Debugging Bugs through the ages.

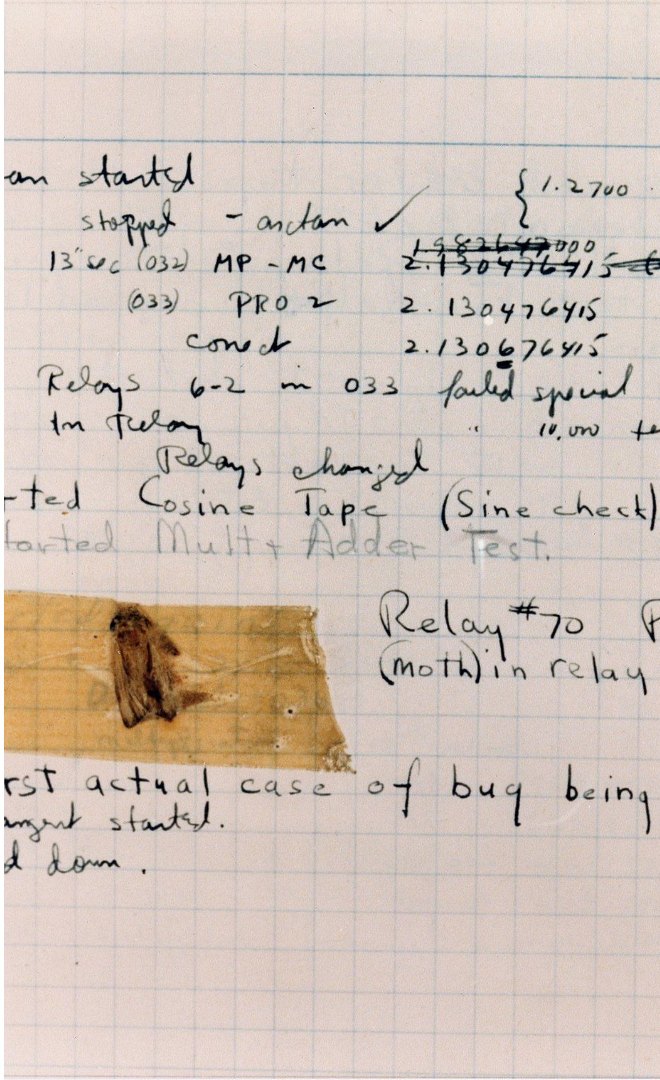
The term "bug" to mean "fault" dates back a long time:

> *... difficulties arise-this thing gives out and [it is] then that "Bugs"-as such little faults and difficulties are called-show themselves*
>
> - Thomas Edison, letter, 1878

Popularly attributed to Grace Hopper for the moth to the right.

To remove bugs from the program, you de-bug them!

https://en.wikipedia.org/wiki/Software_bug



an started {1.2700
stopped - arctan ✓
13" sc (032) MP - MC ~~1.982647000~~ ~~2.130476415~~
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special
In relay " 10,000 te
Relays changed
-ted Cosine Tape (Sine check)
tarted Mult+ Adder Test.
Relay #70 P
(moth) in relay
rst actual case of bug being
angent started.
d down.

Debugging

Debugging is done with *debuggers*, such as `gdb`.

Debuggers use (among other methods), a special *debug instruction*:

```
mov rdi, 42 // our program's return code (e.g., for bash scripts)
mov rax, 60 // system call number of exit()
int3 // trigger the debugger with a breakpoint!
syscall // do the system call
```

When the `int3` breakpoint instruction executes, the debugged program is interrupted and you can inspect its state!

Of course, the debugger itself can set breakpoints:

Overwrites the instruction at the breakpoint address with `int3`.
Emulates its effects when the breakpoint is executed instead!

Other Resources

GDB is your go-to debugging experience.
You WILL become very good friends with it.

strace lets you figure out how your program is interacting with the OS.
A great first stop for debugging.

Rappel lets you explore the effects of instructions.
Get it from https://github.com/yrp604/rappel or just use the pre-installed version in the dojo!
Easily installable via https://github.com/zardus/ctf-tools.

Documentation of x86:
Opcode listing by byte value: http://ref.x86asm.net/coder64.html
Instruction documentation: https://www.felixcloutier.com/x86/
Intel's x86_64 architecture manual: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf