PWN College

Session 1

Main Reference: https://pwn.college/

PWN College

- To learn about, and practice, core cybersecurity concepts
- Designed to take a "**white belt**" in cybersecurity to becoming a "**blue belt**", able to approach (simple) CTFs and wargames.

PWN Challenges

- Pwn challenges consist of challenges that test your skills in **bypassing security mechanisms** inside of systems.
- 95% of the time these challenges will be binary exploitation challenges where you are given a program with some kind of bug that you need to find and then exploit to **gain control of a system** or **make the binary print the flag** you are trying to find.
- You will usually be given the **required binaries** and some **network address** that belongs to a server you are attempting to exploit.

Fundamentals

- C programming.
- C compilation.
- x86_64 assembly.
- OS internals (system calls, etc).
- Linux operations (FS layout, permissions, shell scripting, etc).

Introduction

Computer Systems Security

System Security

- What is System Security?
  - Security of a system with respect to a specified property

- Example
  - Vending Machine

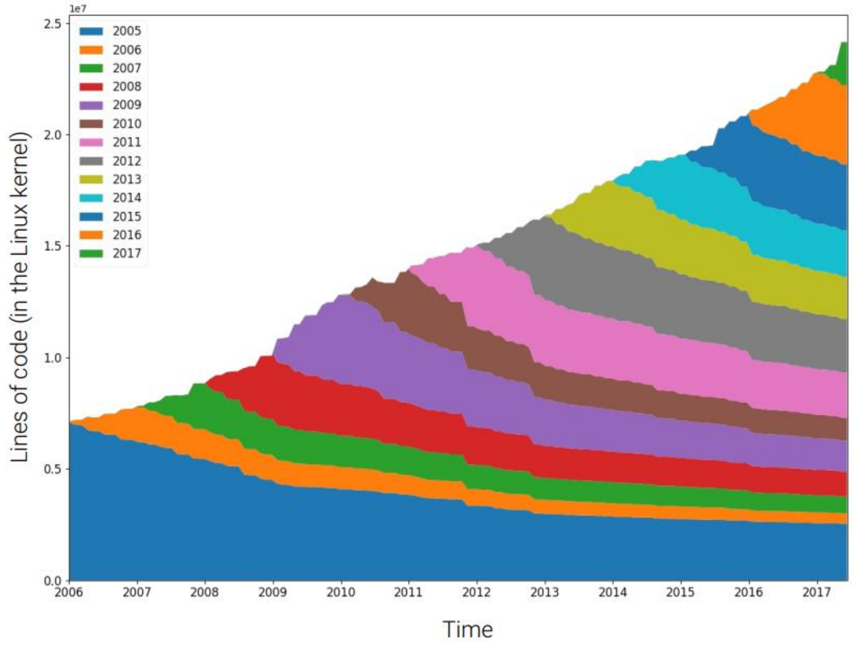
System Security

A chain is only as strong as its weakest link.

Computer System Security

- What is Computer System Security?
  - Protection of computer systems and information from harm, theft, and unauthorized use.

- Modern computer systems are complex.
- With great complexity comes great vulnerability.



Fundamentals

Computer Architecture

Assembly Code

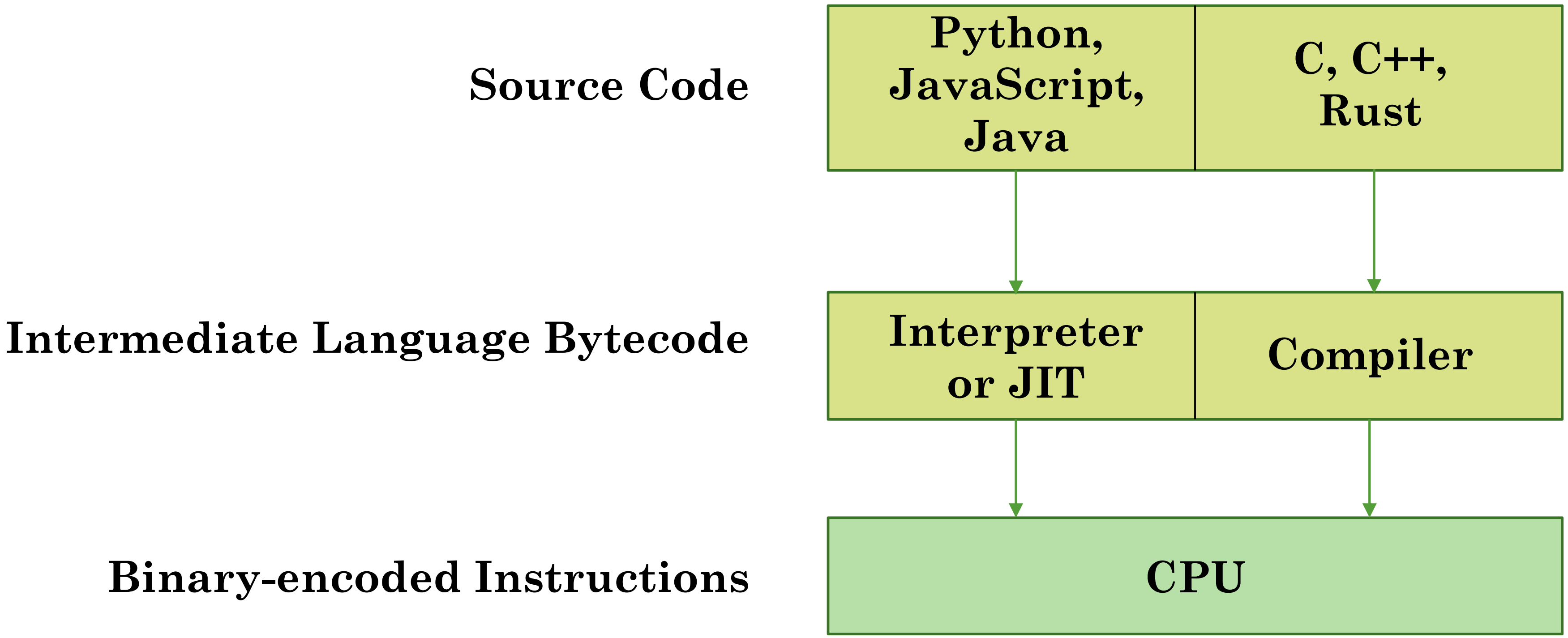
Introduction to Binary Files

Linux Process Loading

Linux Process Execution

All roads lead to the CPU

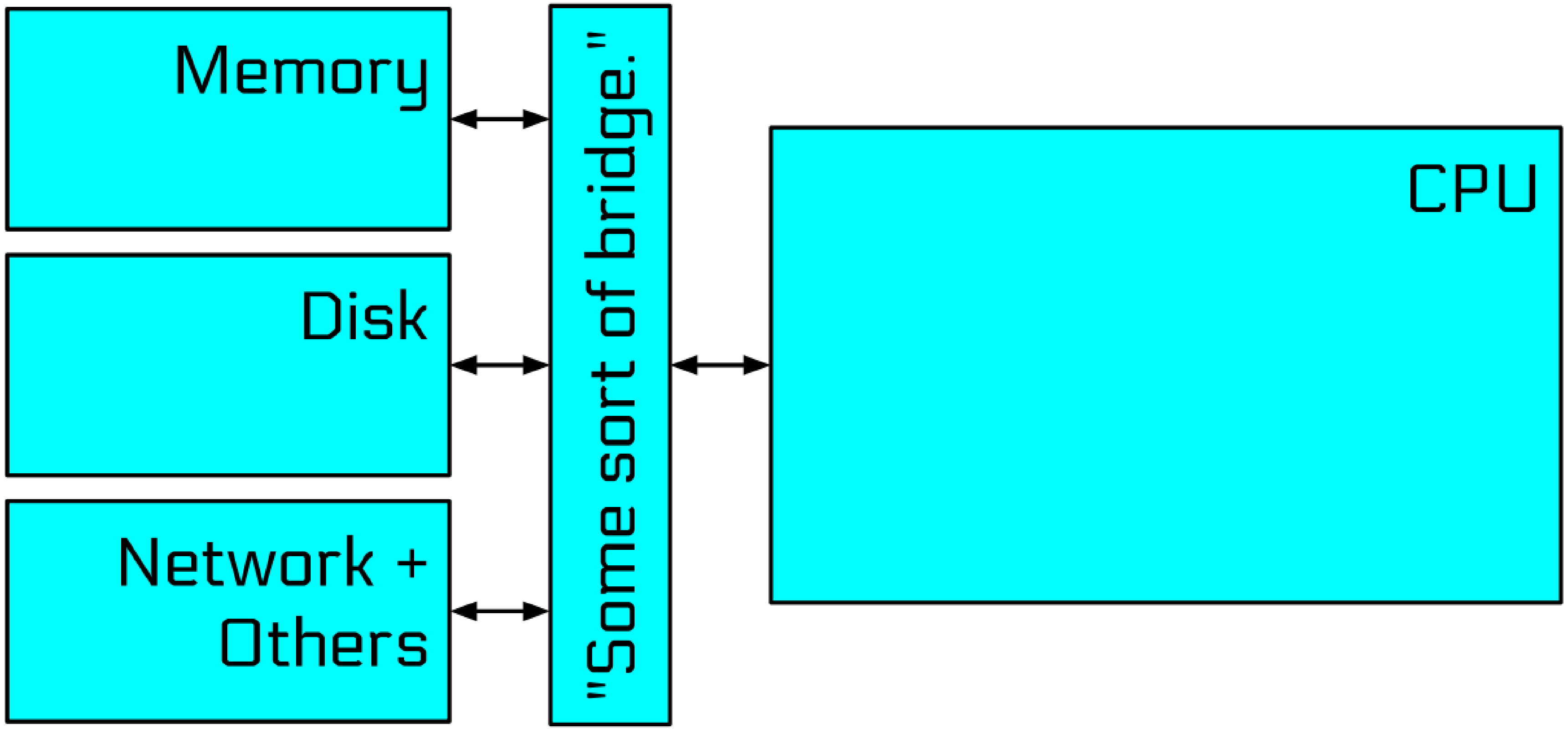
- Everything you write ends up being executed as binary encoded instructions on a CPU.



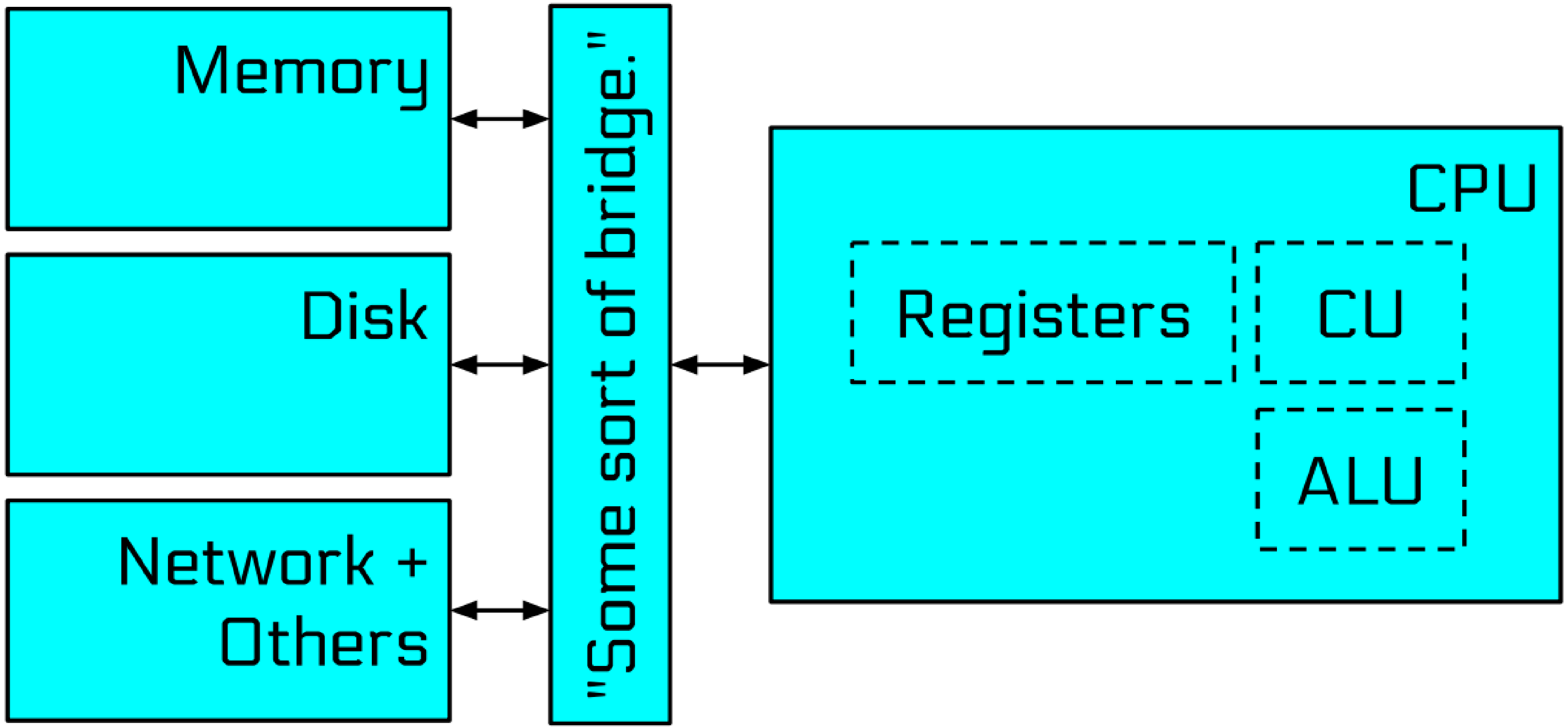
Logic Gates

- There are many logic gates at the center of your CPU.
  - Adder
  - Subtractor
  - Multiplexer
  - Encoder
  - Decoder
  - 7-Segment
  - BCD

Computer Architecture (very high level)



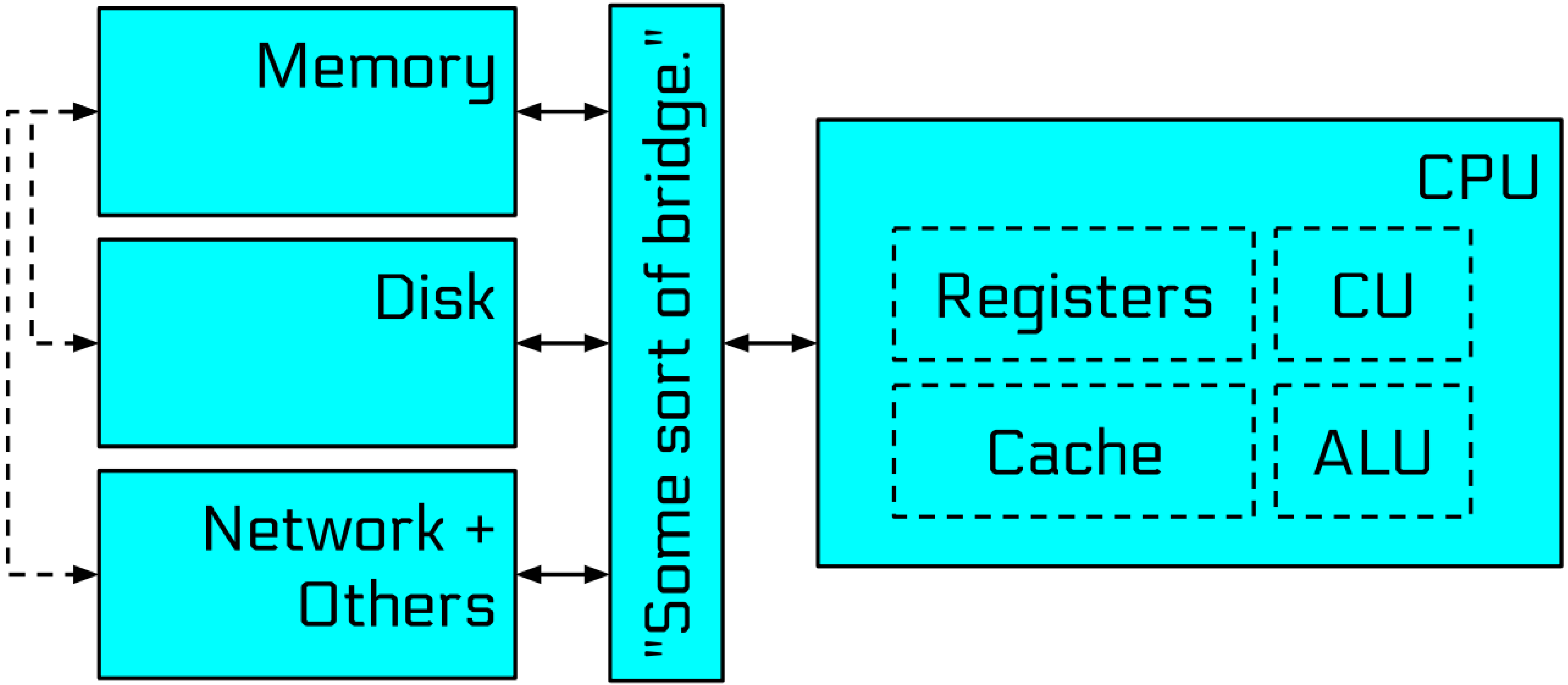
Computer Architecture (drilling down)



Computer Architecture (drilling down)

- **Registers**
  - Temporary storage area in the CPU that accept, quickly store, and transfer data and instruction that can be used and processed by the CPU immediately.
  - Internal registers: IR, MDR, MAR
  - User-accessible registers: DR, AR, PC,…
- **Control Unit**
  - A component of the CPU that directs the operation of the processor
- **Arithmetic Logic Unit**
  - An integrated circuit within a CPU or GPU that performs arithmetic and logic operations.
  - addition, subtraction, shifting operations, AND, OR, XOR, NOT

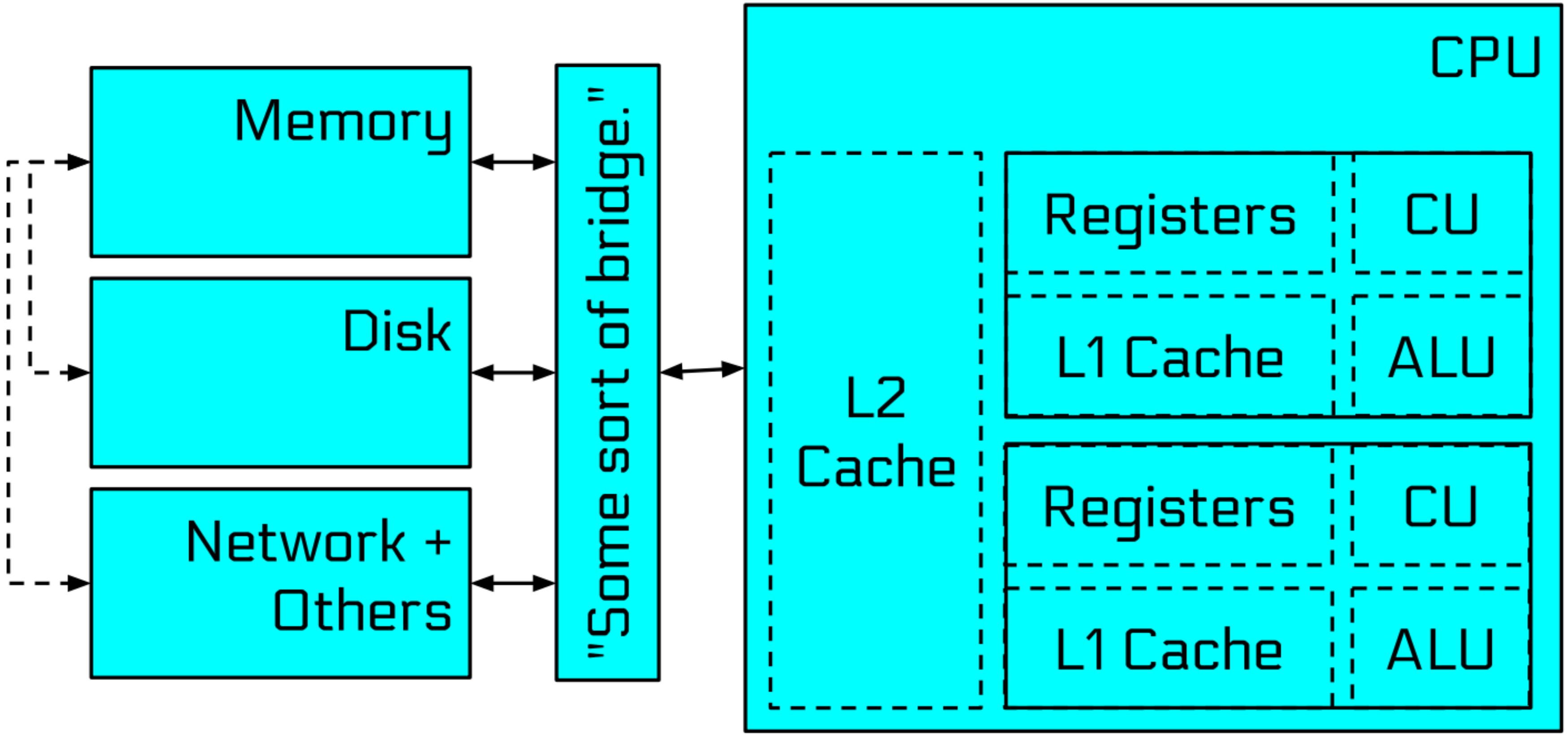
Computer Architecture (further down!)



Computer Architecture (further down!)

- Cache
  - A special storage space for temporary files that makes a device, browser, or app run faster and more efficiently.
- A series of caching layers stacked on top of each other.

Computer Architecture (as far as we'll go)



Computer Architecture (as far as we'll go)

- Multi-core CPU
  - Each of them has its own cache, its own registers,…
  - They share a common cache.

Fundamentals

Computer Architecture

Assembly Code

Introduction to Binary Files

Linux Process Loading

Linux Process Execution

Assembly

- The only true programming language, as far as a CPU is concerned.

- Concepts:
  - Registers
  - Instructions
  - Memory

Registers

- Registers are very fast, temporary stores for data.
- General Purpose Registers
  - Internal registers
  - Used to calculate data and store addresses.

- 8085 architecture:
  - 8-bit architecture
  - **Seven** general purpose 8-bit registers
  - a, c, d, b, e, h, l

Registers

- 8086 architecture:
  - 16-bit architecture
  - **Eight** general purpose 16-bit registers
  - ax, cx, dx, bx, **sp**, **bp**, si, di

- X86 architecture:
  - 32-bit architecture
  - **Eight** general purpose 32-bit registers
  - eax, ecx, edx, ebx, **esp**, **ebp**, esi, edi

Registers

- amd64 architecture:
  - 64-bit architecture
  - **Sixteen** general purpose 64-bit registers
  - rax, rcx, rdx, rbx, **rsp**, **rbp**, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- General purpose registers are used to hold general data. Some of them have also slightly specific purposes.
  - the stack pointers: sp, esp, rsp
  - the base pointers: bp, ebp, rbp
- Special register with special task:
  - Instruction Pointer: ip, eip, rip

Registers

- To see your system's architecture use command bellow:
  - uname –p
  - It prints processor type of system.

```
➜  ~ uname -p
x86_64
```

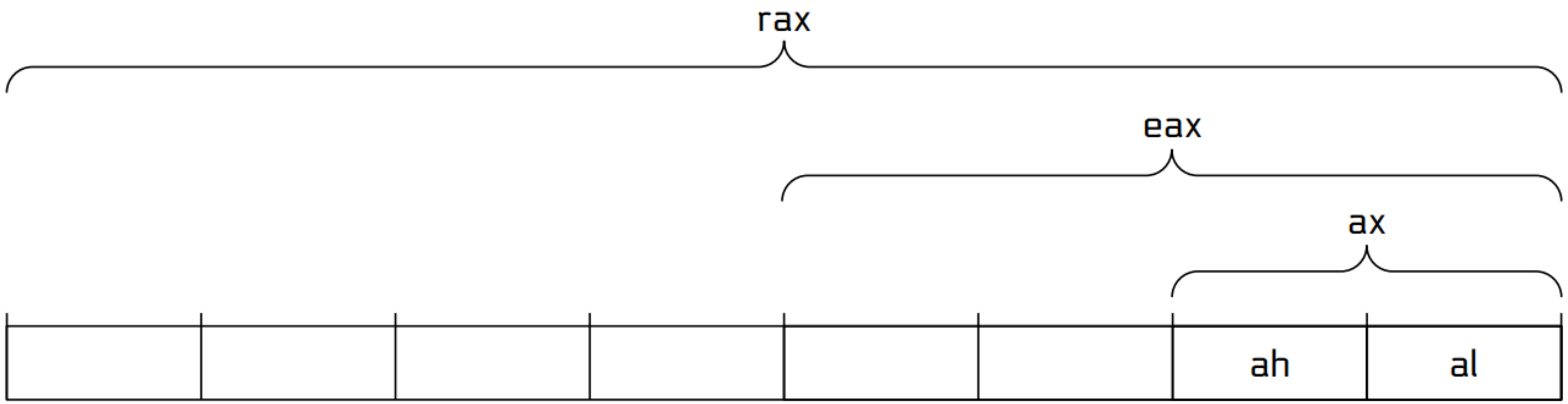
- AMD64 architecture is also known as x86_64, x86-64, x64, Intel64.

Registers

- ***a***: ‘*a*’
  - 8-bit
- ***ax***: ‘*a*’ extended
  - 16-bit
- ***eax***: extended ‘*a*’ extended
  - 32-bit
- ***rax***: really ‘*a*’ extended
  - 64-bit

Registers

- Registers can be accessed **partially**.



- Accessing ***eax*** will zero out the rest of ***rax***.
- Other partial access preserve untouched parts of the register.

Registers

- All partial accesses on amd64:

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

64	32	16	8H	8L
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

- Only in *a*, *c*, *d*, *b* the high byte of the low 16 bits can be accessed.

Instructions

- Instructions tell the CPU what to do.
- There is different types of instructions, but generally they have an operator and several operands.
- General form:
  - OPCODE OPERAND, OPERAND, ...
  - OPCODE: what to do
  - OPERANDS: what to do it on/with
- *Note*: Here we use **Intel** assembly syntax for **amd64** arch.
  - Data flows mostly right to left.

Instructions (data manipulation)

- Instructions can move and manipulate data in registers and memory.

```
mov rax, rbx
mov rax, [rbx+4]
mov [rbx+4], rax
add rax, rbx
mul rsi
inc rax
inc [rax]
```

; $rax = rbx$
; $rax = mem[rbx + 4] \rightarrow read$
; $mem[rbx + 4] = rax \rightarrow write$
; $rax = rax + rbx$
; $rax \times rsi \rightarrow overflow: rdx, output: rax$
; $rax = rax + 1$
; $mem[rax] = mem[rax] + 1$

Instructions (control flow)

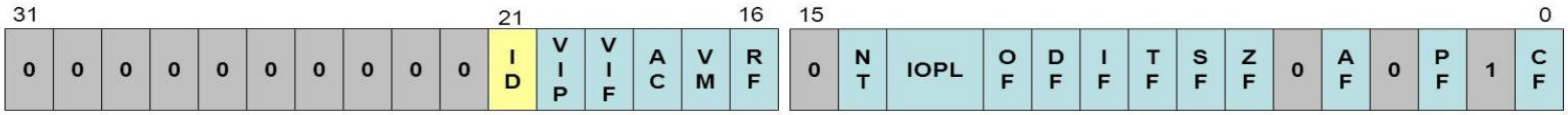
- Control flow is determined by conditional and unconditional jumps.
- Unconditional: call, jmp, ret
- Conditional:

$cmp\ rax, rbx$
$\boldsymbol{jb}\ some\_location$

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

Instructions (control flow)

- ‘flags’ register:
  - Some single bit flags
  - eflags (x86), rflags (amd64), aspr (arm)
- Updated by (x86/amd64):
  - arithmetic operations
  - cmp - subtraction (cmp rax, rbx)
  - test - and (test rax, rax)



Instructions (system calls)

- Almost all programs have to interact with the outside world.
- This is primarily done via system calls (**man syscalls**).
- Each system call is well-documented in **section 2** of the man pages (i.e., *man 2 open*).
- System calls (on amd64) are triggered by:
  1. set ***rax*** to the system call number.
  2. store arguments in ***rdi***, ***rsi***, etc.
  3. call the ***syscall*** instruction.
- We can trace process system calls using ***strace***.

Linux Systemcall Table: 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 new file descriptor of the open file.*
  - `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.*

System Calls

- Example:
  - We want to use syscall in order to call exit function.

```
.global _start
_start:
.intel_syntax noprefix
    mov rax, 60
    mov rdi, 42
    syscall
```

- Compile and run:

```
→  1- Fund_assembly gcc -nostdlib -o exit exit.s
→  1- Fund_assembly ./exit
→  1- Fund_assembly echo $?
42
```

  - ***echo $?*** will return the exit status of last command.

System Calls

- Dynamically Linked
  - *gcc -nostdlib -o exit exit.s*

```
→  1- Fund_assembly file exit
exit: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=187ae17dc3a2254388ba2b493f4b78e047b6d89f, not strip
ped
```

```
→  1- Fund_assembly strace ./exit
execve("./exit", ["./exit"], 0x7ffdc67b1040 /* 54 vars */) = 0
brk(NULL)                               = 0x55eb6f63f000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe40b77630) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f6802320000
arch_prctl(ARCH_SET_FS, 0x7f6802320a80) = 0
mprotect(0x55eb6eaad000, 4096, PROT_READ) = 0
exit(42)                                = ?
+++ exited with 42 +++
```

System Calls

- Statically Linked
  - *gcc -static -nostdlib -o exit exit.s*

```
➜  1- Fund_assembly file exit
exit: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, BuildID[sha1]=d6
9c0b1a7c5cf72bc11fb82a3c9582ef38af7ee8, not stripped
```

```
➜  1- Fund_assembly strace ./exit
execve("./exit", ["./exit"], 0x7fffe04245f0 /* 54 vars */) = 0
exit(42)                                = ?
+++ exited with 42 +++
```