Software Security II

CSE 565: Fall 2024
Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Disclaimer

- We don’t claim any originality of the slides. The content is developed heavily based on
  - Slides from lectures by Yan Shoshitaishvili @ ASU security team (pwn.college)
  - Slides from Prof Ziming Zhao’s past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html)
  - Slides from Prof Hongxin Hu’s past offering of CSE565

Announcement

- HW3 and Project3 **due Today (Tue, Nov 12), 23:59 pm**.
- In-class bonus quiz next lecture (This Thursday)
  - Will be posted in UBLearns (so please bring your laptop to the class).
  - Available only during lecture time.
  - Serve as extra points:
    - E.g, if the bonus quizzes account for 10 pts, then the *total final point grades* will be 110, but the *letter grade cutoff* remains the same: i.e., you still only need 90 total pts to get an A, 75 to get a B, etc.

Review of Last Lecture

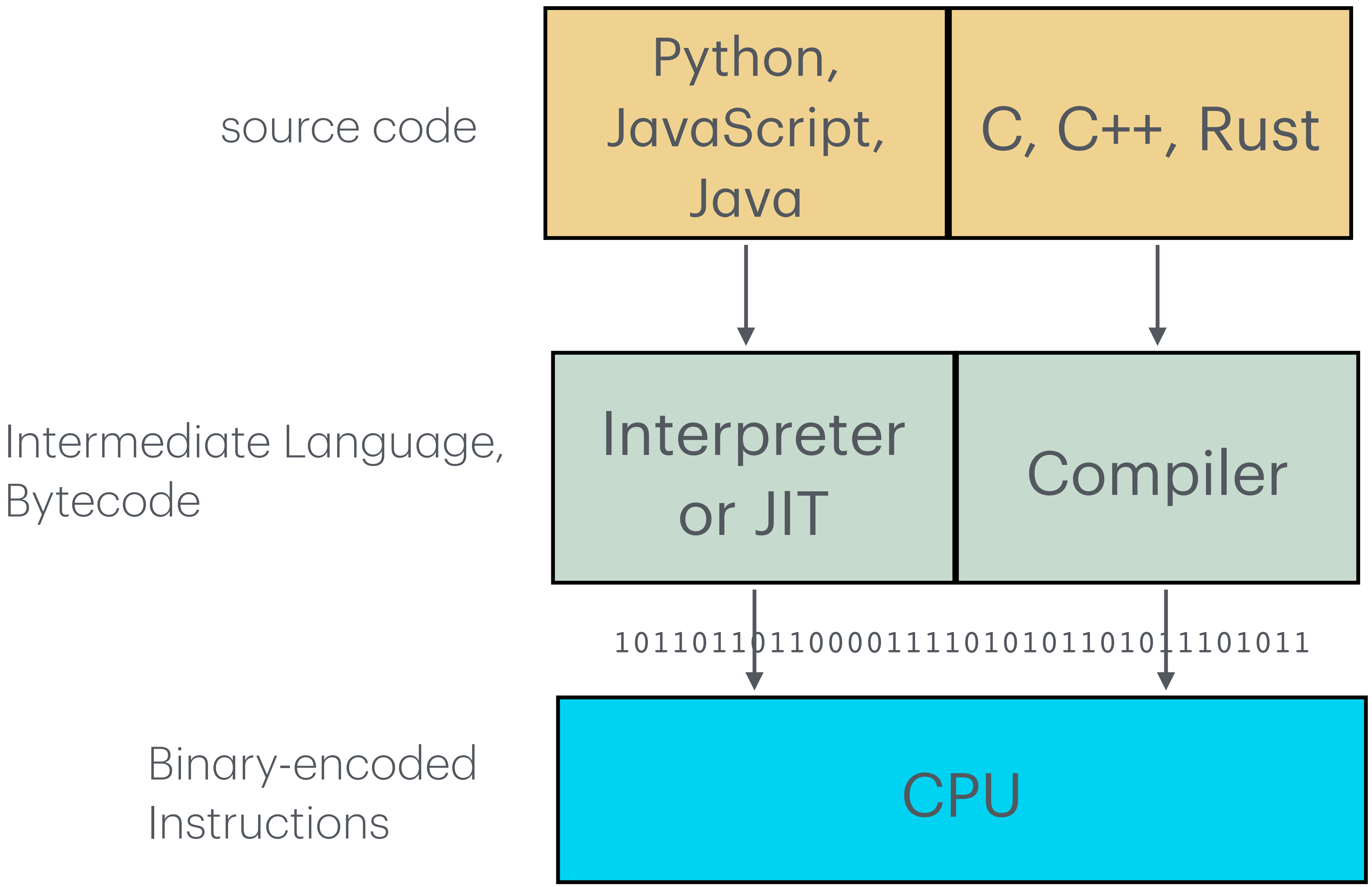
- Background
  - ELF: Executable and Linkable Format
    - Tells the OS how to execute a program
    - Program Headers: necessary for execution. Specifies the interpreter and how to load the executable into memory
    - Section Headers: Good for debugging
  - Linux process Loading & Execution
    - Dynamic-Linked ELF: Kernel load (interpreter & executable) -> Interpreter load shared libraries -> run
    - Syscalls

Today's topic

- (X86) Assembly Crash Course
- "SoK: Eternal War in Memory"

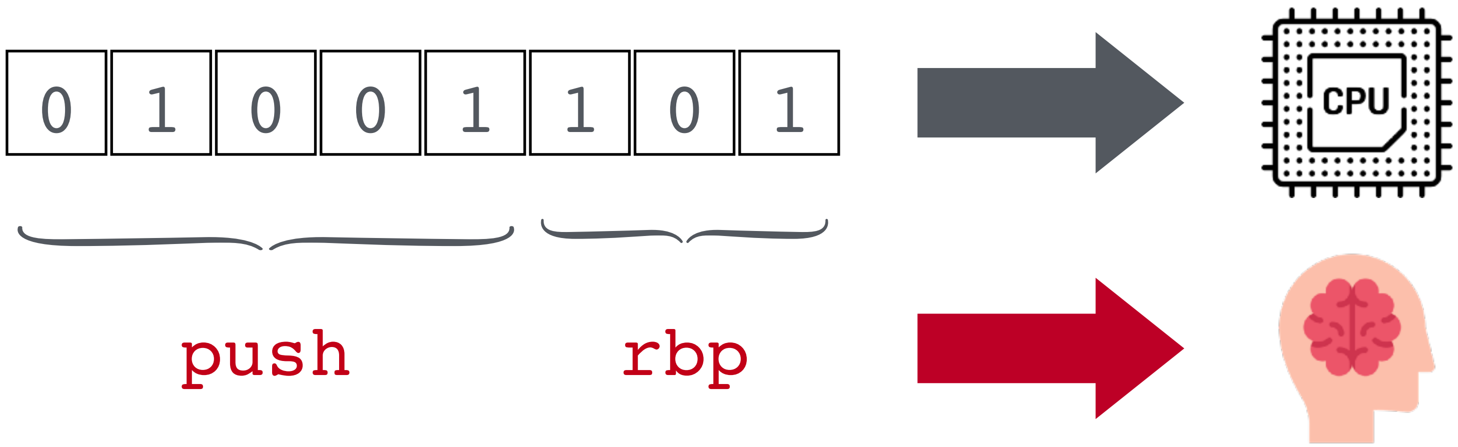
(X86) Assembly Crash Course

All roads lead to the CPU



Assembly Binary

- Binary code is not easy for human to read
- So we create a *text representation* of the binary: **Assembly**
- The binary and the assembly code is **equivalent**



Assembly language

- Assembly is the simplest programming language: It directly tells CPU what to do
  - **Instruction**: *Operations* with suitable *operands*
  - **Operand**: Data
    - Directly given (constant)
    - Close at hand (register)
    - In storage (memory)
  - **Operation**:
    - `add` / `sub`tract / `mul`tiply / `div`ide some data together
    - `mov`e some data into or out of storage
    - `cmp`are two pieces of data with each other
    - `test` some other properties of data

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 ← our focus
  - **arm** assembly
  - **mips** assembly
  - **risc-v** assembly
  - ...
- Regardless of dialect, an assembly instruction always looks like (one of) the following:
  - `OPERATION`
  - `OPERATION OPERAND`
  - `OPERATION OPERAND OPERAND`
  - `OPERATION OPERAND OPERAND OPERAND`

Assembly Dialects

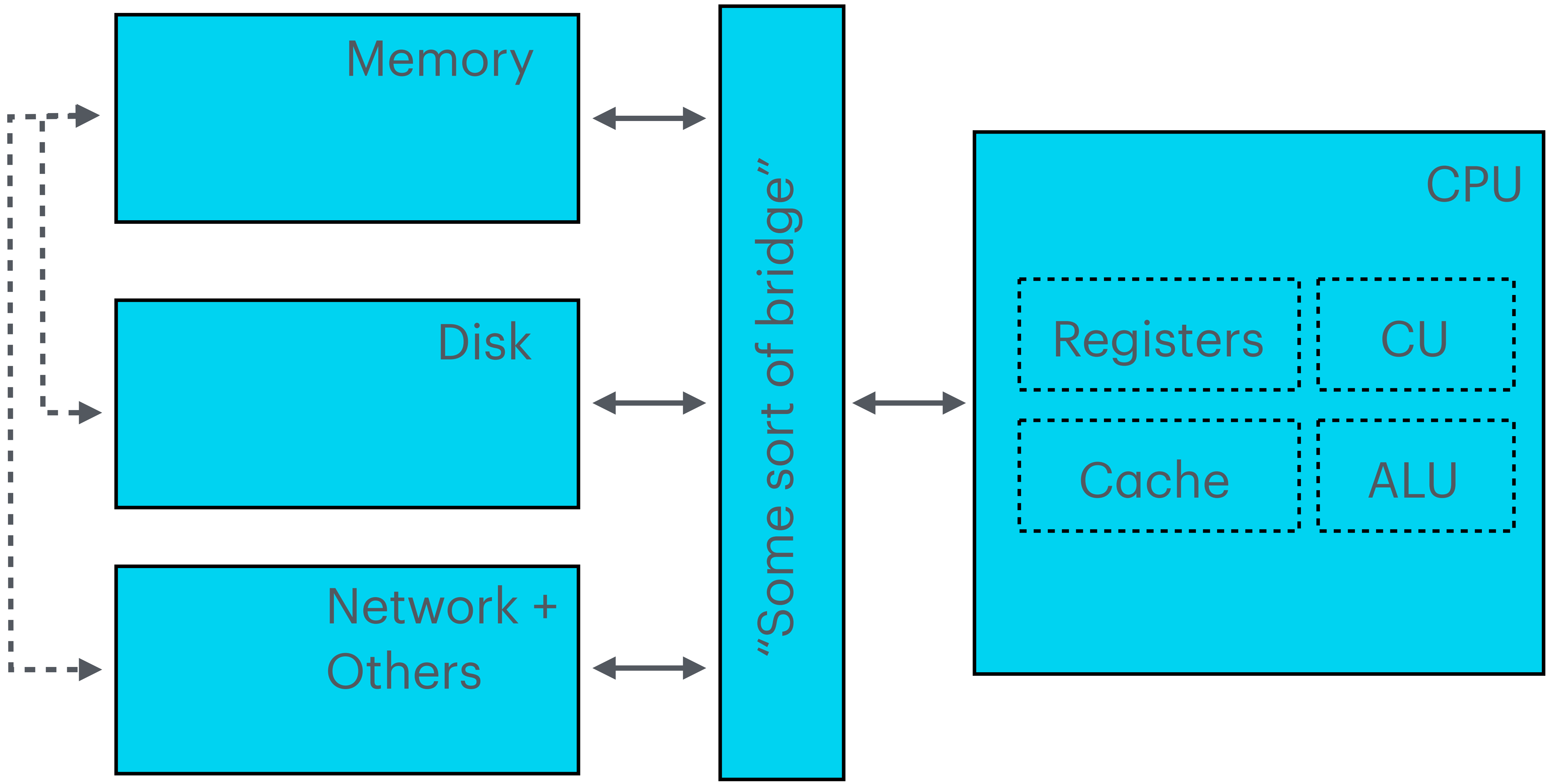
- ~~Interestingly~~ Annoyingly, there are two common dialects for x86
- (**Preferred**) Intel syntax: “`mov ax, 2`”
  - Made by the creator of x86 arch
  - More common in the MS world. Fully supported by GNU toolchains nowadays
  - Much cleaner than the AT&T syntax
- AT&T syntax: “`movb $2, %ax`”
  - Nobody knows why AT&T invented this alternative
  - (Unfortunately) more common in the Linux world

Read Assembly

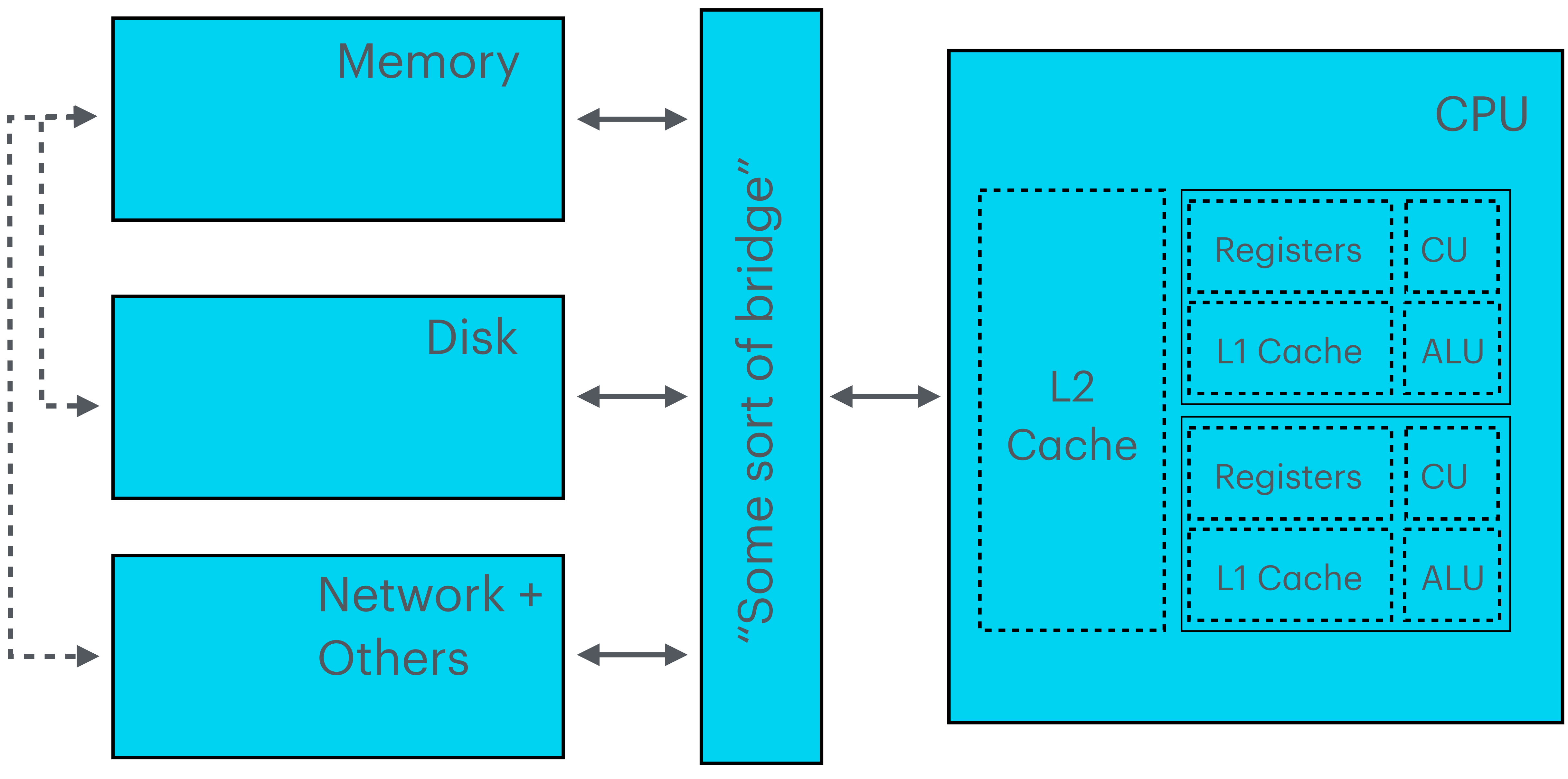
- Disassemble binary code
  - `objdump -d -M intel ./cat | head -40`

```
seed@seed-vm ~/Programs> objdump -d -M intel ./cat | head -40

./cat:     file format elf64-littleaarch64


Disassembly of section .init:

00000000000006e8 <_init>:
objdump: unrecognised disassembler option: intel
 6e8:   d503201f        nop
 6ec:   a9bf7bfd        stp     x29, x30, [sp, #-16]!
 6f0:   910003fd        mov     x29, sp
 6f4:   94000040        bl      7f4 <call_weak_fn>
 6f8:   a8c17bfd        ldp     x29, x30, [sp], #16
 6fc:   d65f03c0        ret

Disassembly of section .plt:

0000000000000700 <.plt>:
 700:   a9bf7bf0        stp     x16, x30, [sp, #-16]!
 704:   90000090        adrp    x16, 10000 <__FRAME_END__+0xf55c>
 708:   f947c211        ldr     x17, [x16, #3968]
 70c:   913e0210        add     x16, x16, #0xf80
 710:   d61f0220        br      x17
 714:   d503201f        nop
 718:   d503201f        nop
```

Computer Architecture (very high level)



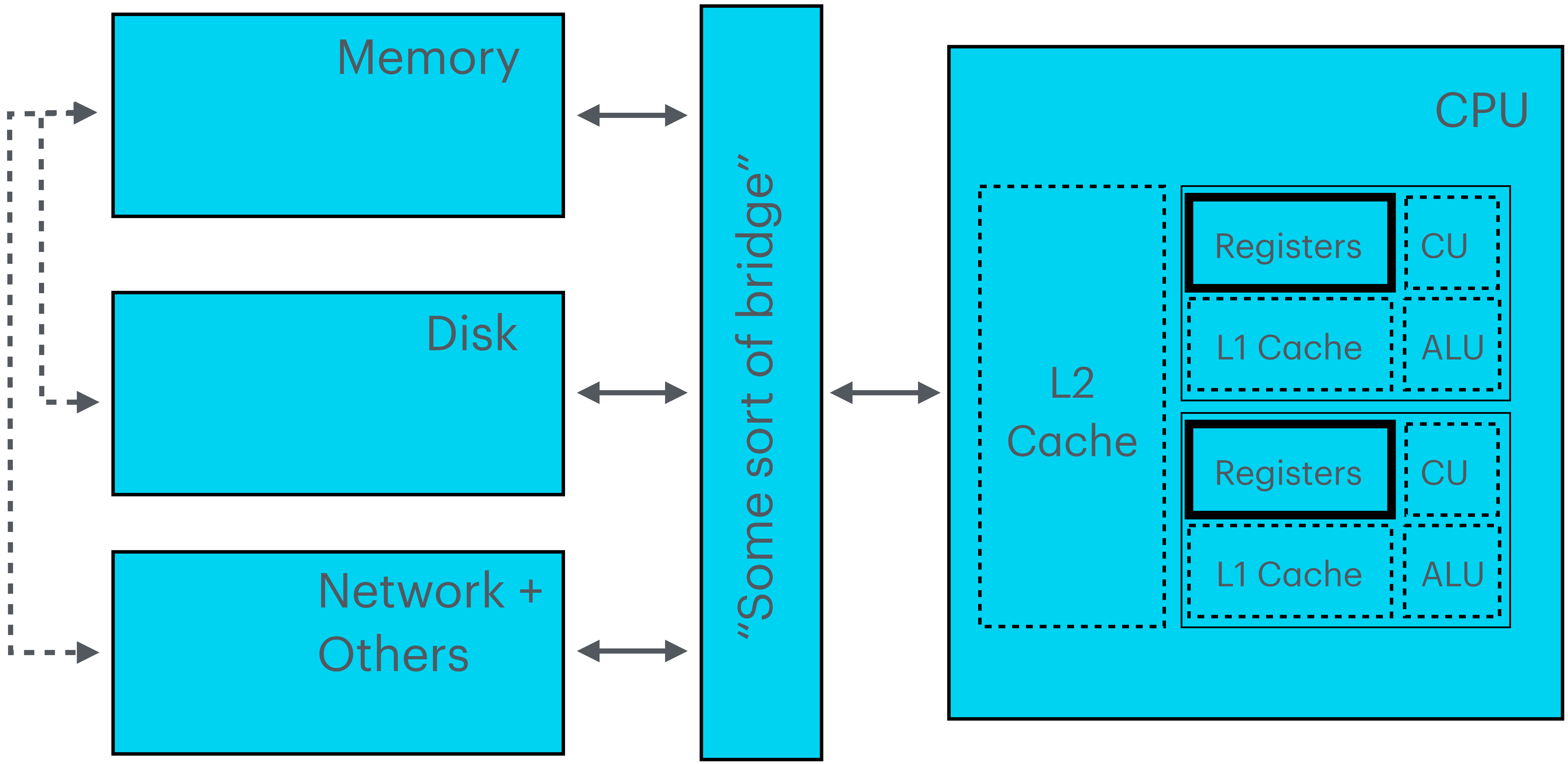
Computer Architecture (very high level)



(X86) Assembly Crash Course

Registers

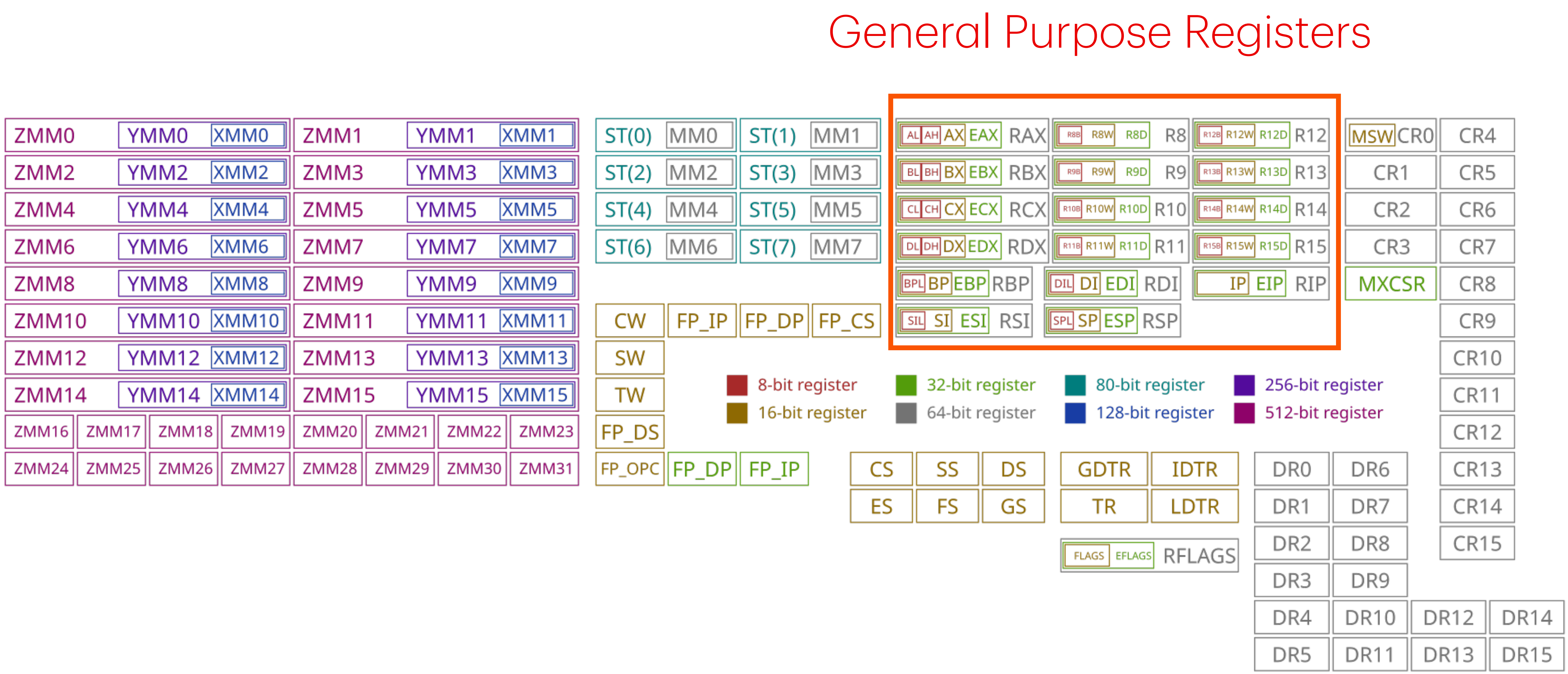
Registers


Memory
Disk
Network +
Others
"Some sort of bridge"
CPU
L2
Cache
Registers
CU
L1 Cache
ALU
Registers
CU
L1 Cache
ALU

Registers

- CPU need to be *fast*: need rapid access to data they're working on.
- Registers are very fast, temporary stores for data.
- There are several *general purpose* registers
  - 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, r15`
- 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)

All X86 registers

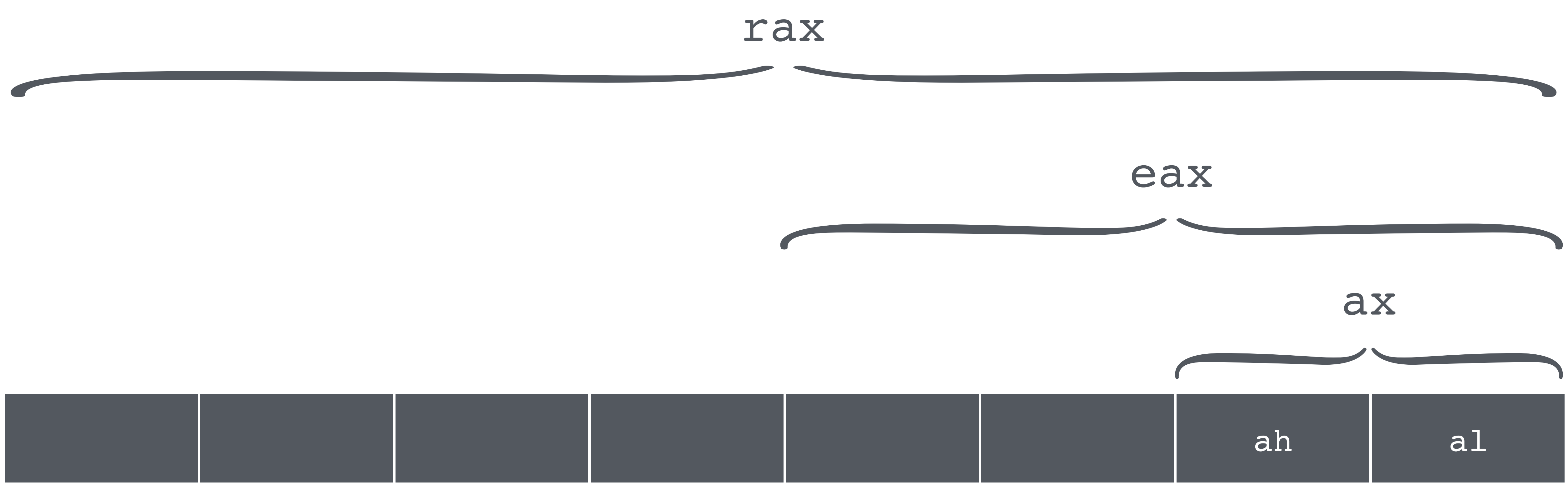


Register Size

- Registers are (typically) the same size as the word width of the architecture.
- On a 64-bit arch (most) registers will hold 64 bits (8 bytes)

10110110	11011110	01101101	10110110	10110110	00110100	11110111	00000110

Partial Register Access



Registers can be accessed *partially*.

Partial Register Access (on amd64)

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

Setting Registers

- Load data into registers with **`mov`**

```
mov rax, 0x39
mov rbx, 1337
```

  - Data specified directly in the instruction is called an immediate value

- 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 mov to a 32-bit partial, the CPU will zero out the rest of the register.

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

rax

`0xffffffffffff0539`

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

`0x0000000000000539`

"Moving" Data Around

- You can also `mov` data between registers
  - "`mov`" does *not* move the data: it *copies* the data
- This sets both `rax` and `rbx` to 0x539

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

- You also can mov partials (32-bit caveat applies): this sets `rax` to 0x539 and `rbx` to 0x39

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

Extending Data

- Due to the 32-bit zeroing, moving a negative number may result in unwanted result:

```
mov eax, -1
```

→

$$\underbrace{\boxed{\texttt{0x00000000}}\,\overbrace{\boxed{\texttt{0xffffffff}}}^{\texttt{eax = -1}}}_{\texttt{rax = 4294967295}}$$

- What if you want to operate on that -1 on 64 bit?

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

→

$$\underbrace{\boxed{\texttt{0xffffffff}}\,\overbrace{\boxed{\texttt{0xffffffff}}}^{\texttt{eax = -1}}}_{\texttt{rax = -1}}$$

- **`movsx`** does sign-extending, preserving two's complement

Register Arithmetic

- You can compute with data in registers.
  - In most cases, the first register stores the result.

Instruction	Math equivalent	Description
`add rax, rbx`	`rax = rax + rbx`	add `rbx` to `rax`
`sub ebx, ecx`	`ebx = ebc - ecx`	subtract `ecx` from `ebx`
`imul rsi, rdi`	`rsi = rsi * rdi`	multiply `rdi` to `rsi`, truncate to 64 bits
`inc rdx`	`rdx = rdx + 1`	increment `rdx`
`dec rdx`	`rdx = rdx - 1`	decrement `rdx`
`neg rax`	`rax = -rax`	(numerically) negate `rax`
`not rax`	`rax = ~rax`	flip each bit of `rax`
`and rax, rbx`	`rax = rax & rbx`	bitwise AND of `rax` and `rbx`
`or rax, rbx`	`rax = rax | rbx`	bitwise OR of `rax` and `rbx`
`xor rcx, rdx`	`rcx = rcx ^ rdx`	bitwise XOR of `rax` and `rbx`
`shl rax, 10`	`rax = rax << 10`	leftshit `rax` by 10 bits, filingl 0 on the right
`shr rax, 10`	`rax = rax >> 10`	rightshift `rax` by 10 bits, filling 0 on the left
`sar rax, 10`	`rax = rax >> 10`	rightshift `rax` by 10 bits, with sign extension
`ror rax, 10`	`rax = (rax >> 10) | (rax << 54)`	rightward rotate `rax` by 10 bits
`rol rax, 10`	`rax = (rax << 10) | (rax >> 54)`	leftward rotate `rax` by 10 bits

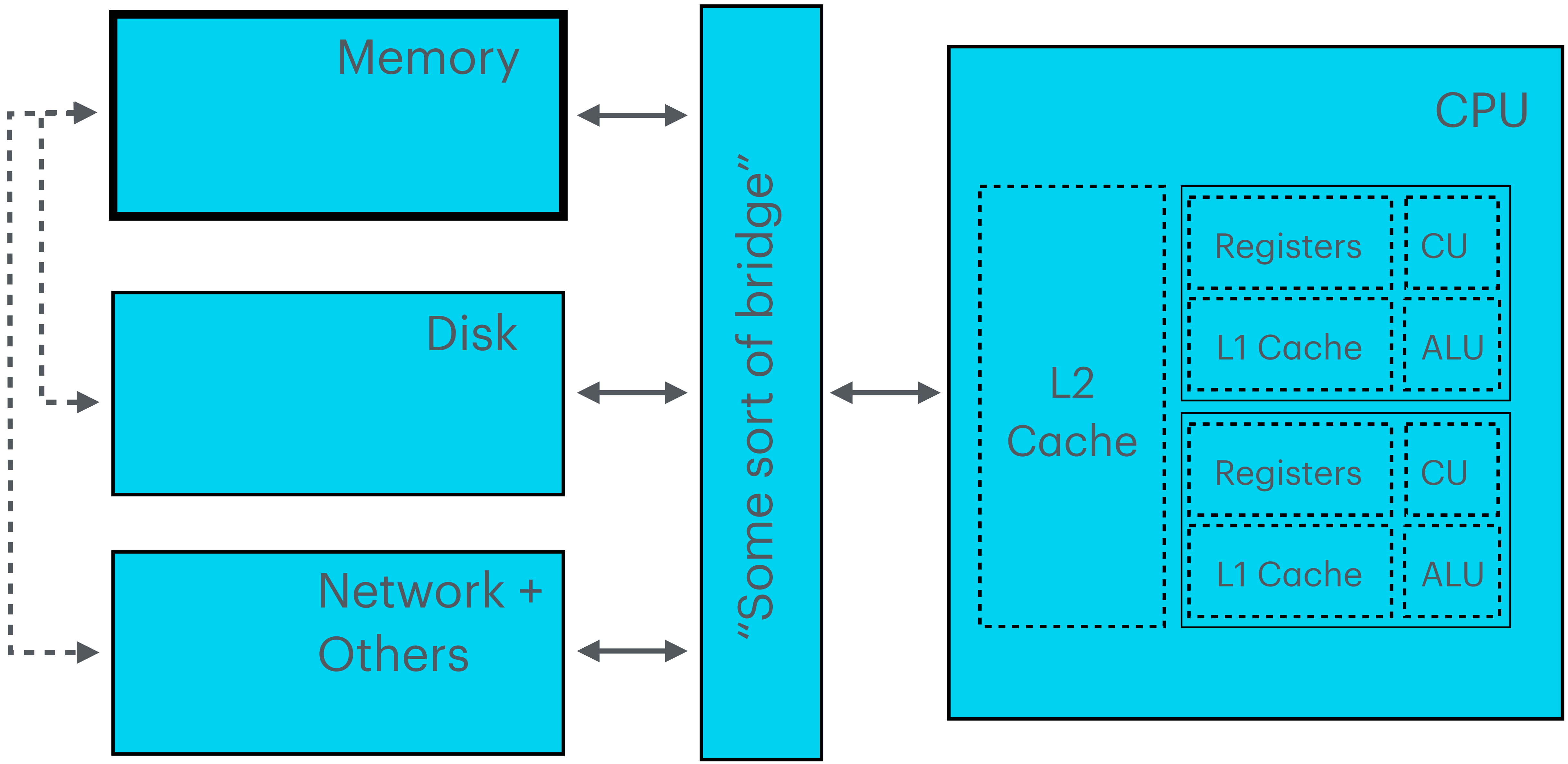
Special Registers

- You cannot directly read or write `rip`
  - It contains the mem address of the next instruction to be executed (ip = **I**nstruction **P**ointer)
- You should be careful with `rsp`
  - It contains the addr of an mem region to store temporary data (sp = **S**tack **P**ointer)
- Some other registers are, by convention, used for important things.

(X86) Assembly Crash Course

Memory

Memory



Memory
Disk
Network +
Others
"Some sort of bridge"
CPU
L2
Cache
Registers
CU
L1 Cache
ALU
Registers
CU
L1 Cache
ALU

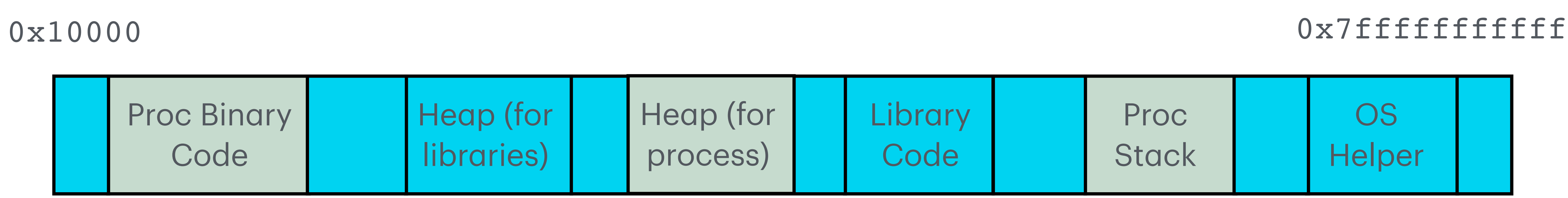
Memory: Process Perspective

- Process memory is used for a lot
  - Memory ↔ Registers
  - Memory ↔ Disk
  - Memory ↔ Network
  - Memory ↔ Video Card
  - ...
- There's too much memory to name every location (unlike registers)
- Memory is addressed *linearly*
  - **From** `0x10000` (for security reasons)
  - **To** `0x7fffffffffff` (47 bits, for arch / OS purpose)
- Each memory address references one byte in memory.
  - 64-bit arch means 127 TB of addressable RAM

Memory: Process Perspective

- You don't have 127 TB of RAM, but that's ok, since it's all *virtual*
- A process' memory starts out partially filled-in by the OS

- The process can ask for more memory from the OS



Memory: Stack

- The stack can be used for temporary data storage and grows backwards
- Registers and immediates can be **push**ed onto the stack to save values

```
mov rax, 0x1234abcd
push rax
push 0xb0b2cafe
push rax
```

- Values can be popped back off of the stack (to any register)

```
pop rbx  # sets rbx to 0x1234abcd
pop rcx  # sets rcx to 0xb0b2cafe
```

Addressing the Stack

- The memory address of the stack's top is stored in **`rsp`**

- Stack grows backwards: **`push`** decreases **`rsp`**, **`pop`** increases it.

Accessing Memory

- You can also move data between registers and memory with **`mov`**
- Example: load the 64-bit value stored at mem addr **`0x12345`** into **`rbx`**

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

- Example: store the 64-bit value in **`rbx`** into mem at addr **`0x133337`**

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

- Example: equivalent to push **`rcx`**

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

- Remember: Each addressed memory location contains one byte
  - An 8-byte write at **`0x133337`** will write to addr **`0x133337`** through **`0x13333e`**

Controlling Write Size

- You can use partial registers 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
```

- Store 8 bits from **`ah`** to addr **`0x12345`**

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

- Remember: changing 32-bit partials zeroes out the whole 64-bit register.
  - Storing 32 bits to memory has no such problems, though

Controlling Write Size

- You can use partial registers 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 **`ah`**

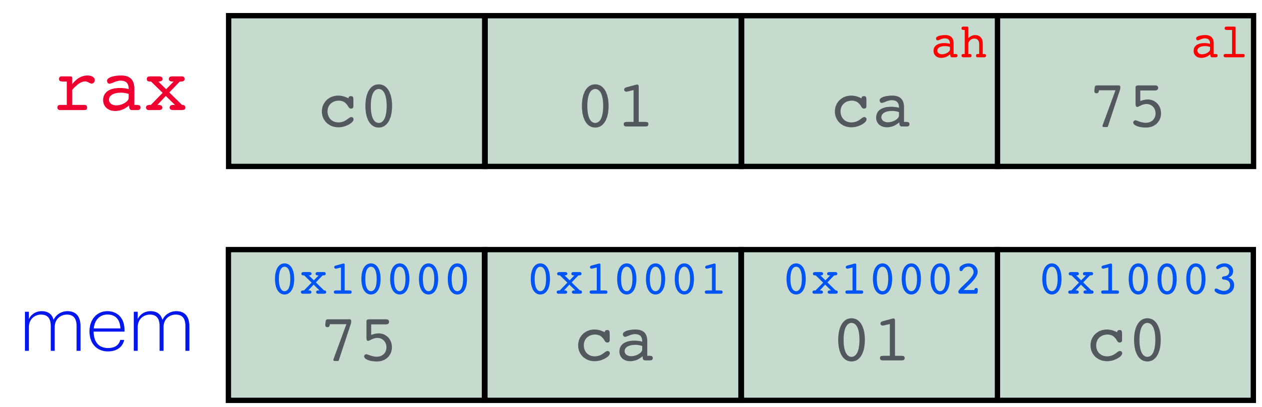
```
mov rbx, 0x12345
mov ah, [rbx]
```

- Remember: changing 32-bit partials zeroes out the whole 64-bit register.
  - Storing 32 bits to memory has no such problems, though

Memory Endianess

- Data on most modern systems is stored in little endian: least-significant byte of a word at the smallest address.

```
mov eax, 0xc001ca75
mov rcx, 0x10000
mov [rcx], eax
mov bh, [rcx]
```



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

Address Calculation

- Limited calculations are allowed for mem addr
- Use `rax` as an offset off some base addr (in this case., the stack)

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

- Get the calculated addr with Load Effective Addr (`lea`)

```
mov rax, 1
pop rcx
lea rbx, [rsp+rax*8+5] # rbx now holds the computed addr
mov rbx, [rbx]
```

- Address calculation has limits
  - `reg+reg*(2 or 4 or 8)+value` is all you can do

RIP-relative Addressing

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

```
lea eax, [rip] # load the addr. of the next instr. to rax
lea rax, [rip+8] # the addr. of the next instr. plus 8 bytes
```

- You can also use `mov` to read from those locations.

```
mov rax, [rip] # load 8 bytes from the addr of the next instr
```

- Or even write there:

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

- RIP-relative addressing is particularly useful for working with *data* embedded near the code
  - This is what makes certain security features on modern machines *possible*

Writing Immediate Values to Memory

- You can also write immediate values to mem, but you must *specify their size*.

- This writes a 32-bit `0x1337` (padded with 0) to addr `0x133337`

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

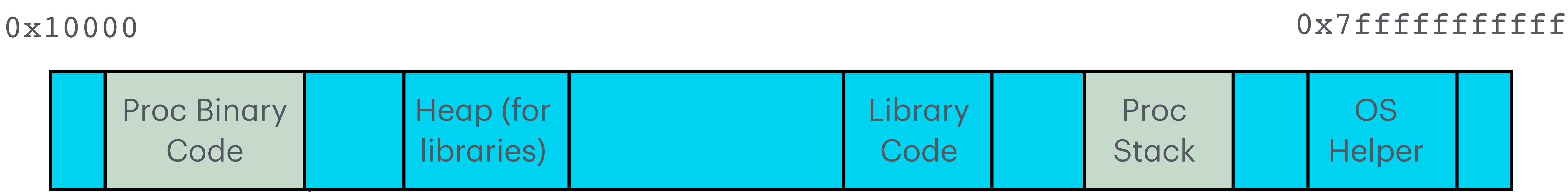
- Depending on your assembler, it might expect `DWORD` instead of `DWORD PTR`

(X86) Assembly Crash Course

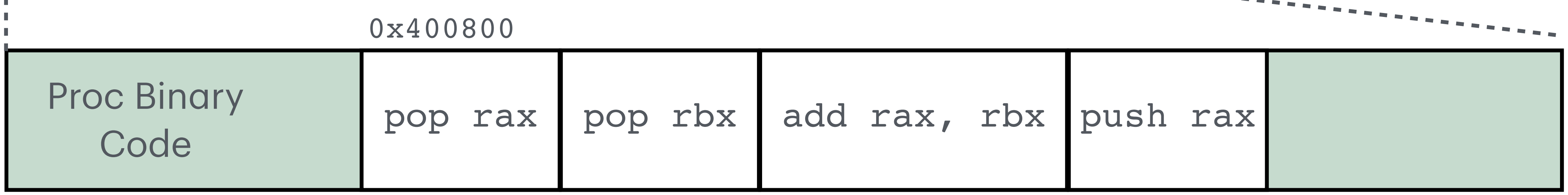
Control Flow

What to Execute

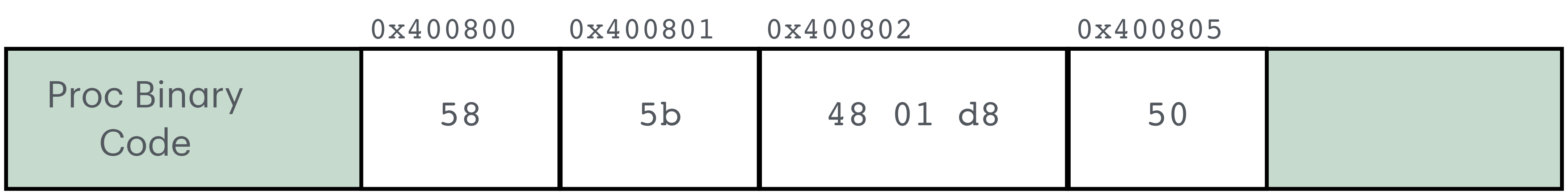
- Recall: Assembly instr. are direct translations of binary code, which lives in *memory*.



- Example:



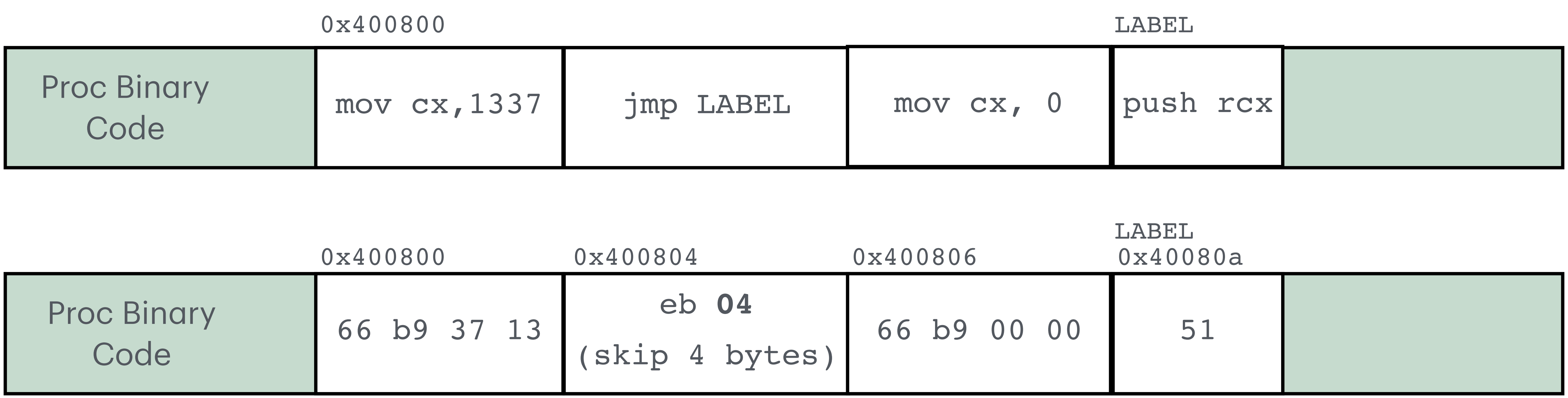
- In hex:



Jumps

- CPUs execute instructions in sequence *until told not to*.
- One way to interrupt the seq is with a **`jmp`** instruction: skip X bytes and resume execution.

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



Conditional Jumps

- Jumps can rely on conditions

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

`je/jne`	jump if equal / inequal
`jg/jl`	jump if greater / less
`jle/jge`	jump if <= / >=
`ja/jb`	jump if > / < (unsigned)
`jae/jbe`	jump if >= / <= (unsigned)
`js/jns`	jump if signed / unsigned
`jo/jno`	jump if overflow / not overflow
`jz/jnz`	jump if zero / nonzero

	0x400800			LABEL	
Proc Binary Code	mov cx,1337	jnz LABEL	mov cx, 0	push rcx	

	0x400800	0x400804	0x400806	LABEL 0x40080a	
Proc Binary Code	66 b9 37 13	75 **04**	66 b9 00 00	51	

Conditional Jumps

- Conditional jumps check conditions stored in the flags register: **`rflags`**
- Flags can be updated by:
  - Most arithmetic instr.
  - Comparison instr. **`cmp`** (**`sub`**)
  - Comparison instr. **`test`** (**`and`**)
- Main conditional flags
  - Carry (**`CF`**): was the 65th bit 1?
  - Zero (**`ZF`**): was the result 0?
  - Overflow (**`OF`**): did the result overflow?
  - Signed (**`SF`**): was the result's signed bit set?

`je/jne`	jump if `ZF=1` / `ZF=0`
`jg/jl`	jump if `ZF=0 AND SF=OF` / `SF != OF`
`jle/jge`	jump if `ZF=1 OR SF != OF` / `SF = OF`
`ja/jb`	jump if `CF = 0 AND ZF = 0` / `CF = 1`
`jae/jbe`	jump if `CF = 0` / `CF = 1 OR ZF = 1`
`js/jns`	jump if `SF = 1` / `SF = 0`
`jo/jno`	jump if `OF = 1` / `OF = 0`
`jz/jnz`	jump if `ZF = 1` / `ZF = 0`

- Common patterns

```
cmp rax, rbx: ja LABEL
cmp rax, rbx; jle LABEL
test rax, rax; jnz LABEL # rax !=0
cmp rax, rbx; je LABEL # rax == rbx
```

Looping

- With conditional jumps, we can implement loops easily
- Example: counts to 10

```
mov rax, 0
LOOP_HEADER:
inc rax
cmp rax, 10
jb LOOP_HEADER
```

Function Calls

- Assembly code is split into functions with `call` and `ret`
  - `call` pushes `rip` (next instr. pointer) to stack and jumps away
  - `ret` pops rip and jumps to it.
- Using a function that takes an authed value and returns different vals

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

FUNC_CHECK_AUTH:
   test rdi, rdi
   jz AUTH
   mov ax, 0
   ret
   AUTH:
   mov ax, 1337
   ret

FUNC_EXIT:
   ???
```

```
int check_auth(int authed) {
  if (authed) return 1337;
  else return 0;
}
int main() {
  check_auth(0);
  check_auth(1);
  exit();
}
```

Calling Conventions

- Callee and caller functions must agree on argument passing
  - **Linux x86**: `push` arguments (in reverse order), then `call` (which pushed return address), return value in `eax`.
  - **Linux amd64**: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, return val in `rax`
  - **Linux arm**: `r0`, `r1`, `r2`, `r3`, return val in `r0`
- **Linux amd64**
  - `rbx`, `rbp`, `r12`, `r13`, `r14`, `r15` are "callee-saved": the function you call will restore these registers to their same initial state.
  - Other registers are up for grabs. Save their values.

(X86) Assembly Crash Course

System Call

System Calls

- Instructions that make a *call* into the OS
  - `syscall` triggers the system call specified by the value in `rax`
  - arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`
  - return value in `rax`
- Example: 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
```

https://x64.syscall.sh/

System Calls

- Example: 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`, and we can `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
mov rax, 1 # system call number of write()
syscall # do the system call
```

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`.
  - `long syscall(long syscall, ...)` – invoke specified `syscall`.

“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.
- Example
  - 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], ‘C‘
mov BYTE PTR [rsp+2], ‘S‘
mov BYTE PTR [rsp+3], ‘E‘
mov BYTE PTR [rsp+4], 0 # write the 0 byte that terminates the string
```

rsp	rsp+1	rsp+2	rsp+3	rsp+4
2f(/)	43(C)	53(S)	45(E)	00(\0)

  - Then `open()` the `/CSE` file
    - `mov rdi, rsp # read the data onto the stack`
    - `mov rsi, 0 # open the file read-only`
    - `mov rax, 2 # system call number of open()`
    - `syscall # do the system call`
  - `open()` returns the file descriptor number in `rax`

Quitting the Program

- Lastly, we can quit

```
mov rdi, 42 # program's return code
mov rax, 60 # system call number of exit()
syscall # do the system call
```

Eternal War in Memory

SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei[*‡], Dawn Song[‡]
[†]*Stony Brook University*
[‡]*University of California, Berkeley*
[*]*Peking University*

***Abstract*—Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of safety in these languages allows attackers to alter the program's behavior or take full control over it by hijacking its control flow. This problem has existed for more than 30 years and a vast number of potential solutions have been proposed, yet memory corruption attacks continue to pose a serious threat. Real world exploits show that all currently deployed protections can be defeated.**

This paper sheds light on the primary reasons for this by describing attacks that succeed on today's systems. We systematize the current knowledge about various protection techniques by setting up a general model for memory corruption attacks. Using this model we show what policies can stop which attacks. The model identifies weaknesses of currently deployed techniques, as well as other proposed protections enforcing stricter policies.

We analyze the reasons why protection mechanisms implementing stricter polices are not deployed. To achieve wide adoption, protection mechanisms must support a multitude of features and must satisfy a host of requirements. Especially important is performance, as experience shows that only solutions whose overhead is in reasonable bounds get deployed.

A comparison of different enforceable policies helps designers of new protection mechanisms in finding the balance

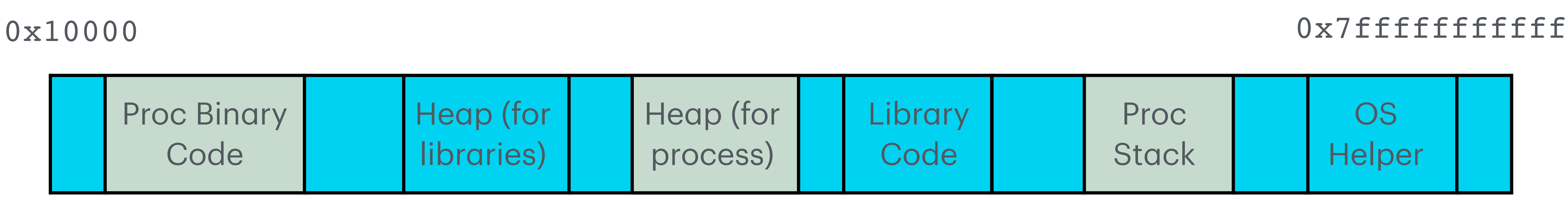
try to write safe programs. The memory war effectively is an arms race between offense and defense. According to the MITRE ranking [1], memory corruption bugs are considered one of the top three most dangerous software errors. Google Chrome, one of the most secure web browsers written in C++, was exploited four times during the Pwn2Own/Pwnium hacking contests in 2012.

In the last 30 years a set of defenses has been developed against memory corruption attacks. Some of them are deployed in commodity systems and compilers, protecting applications from different forms of attacks. Stack cookies [2], exception handler validation [3], Data Execution Prevention [4] and Address Space Layout Randomization [5] make the exploitation of memory corruption bugs much harder, but several attack vectors are still effective under all these currently deployed basic protection settings. Return-Oriented Programming (ROP) [6], [7], [8], [9], [10], [11], information leaks [12], [13] and the prevalent use of user scripting and just-in-time compilation [14] allow attackers to carry out practically any attack despite all protections.

A multitude of defense mechanisms have been proposed

Memory corruption

- Software vulnerability that caused by accessing the memory in unintended ways. Prevalent in low-level languages like C or C++.
  - Attacker manipulate a program's internal state by forcing it to *read* or *write* data to memory locations beyond the intended boundaries.
    - Program code is stored in memory: direct attack
    - Control flow can depend on data in memory: var used in **`if`**, function **`ret`**urn addr, etc
    - Library codes are also in memory: used as gadget



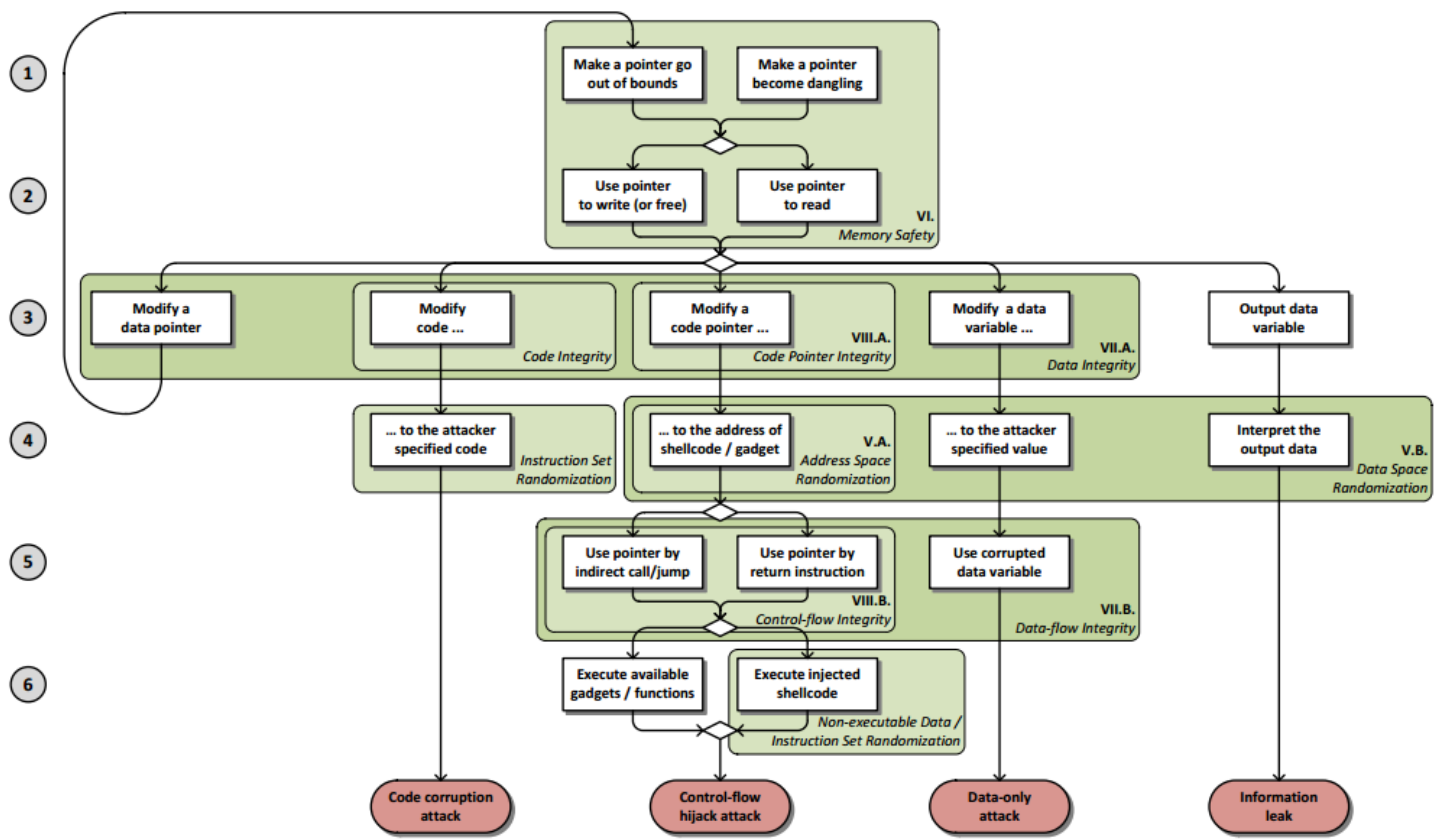


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Temporal & Spatial Error

- **Spatial Error**: out-of-bounds accessing. (E.g. pointers pointing beyond the end of an array.)
  - E.g.: User input interpreted as an address

    ```
    printf("%s\n", err_msg); // leak arbitrary memory contents
    ```

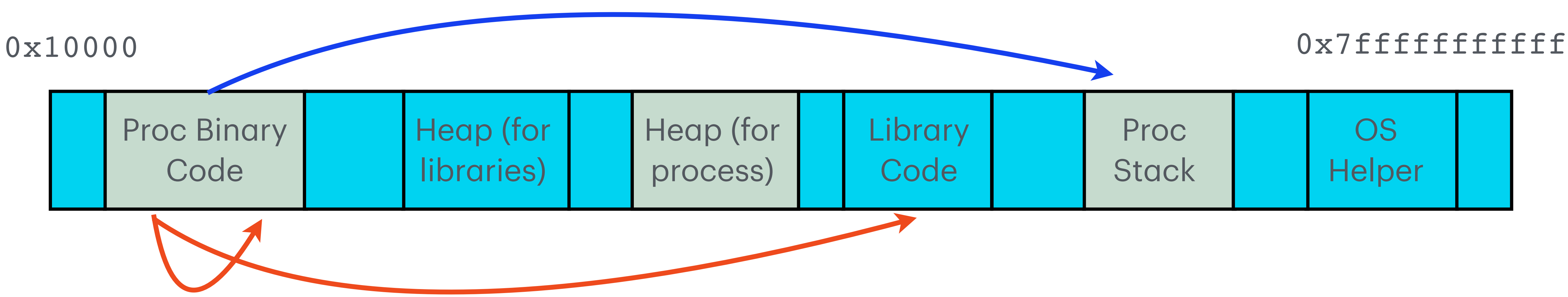
  - E.g. Format string bug

    ```
    printf(user_input); // input "%3$x" leaks the 3rd integer on the stack
    ```

- **Temporal Error**: accessing a deleted object (a dangling pointer)
  - The dangling pointer points to a new obj controlled by attacker
  - When the pointer is dereferenced, the new obj will be interpreted in the way of the old obj.

Control-flow hijack

- Attacker overrides a **`ret`** address or **`jmp`** address to direct execution to a code segment of their choice
  - Return to code injected by attacker ("shellcode")
    - Prevented by *Non-Executable Data* policy
  - Return to existing code in memory: return-to-libc attack; Return Oriented Programming (ROP); Jump Oriented Programing (JOP)



What's next

- (Stack-based) Buffer Overflow Attack
  - Modify function `ret` addr / stack frame pointer `rbp` & `rsp` / flag var value stored on stack
  - Format string vulnerability
- Heap Exploitation
  - Use-After-Free
- Return-Oriented Programming
  - Code injection; Ret-to-libc; Finding gadgets in existing code;

Questions?