Microarchitecture Security

CSE 565: Fall 2024 Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Disclaimer

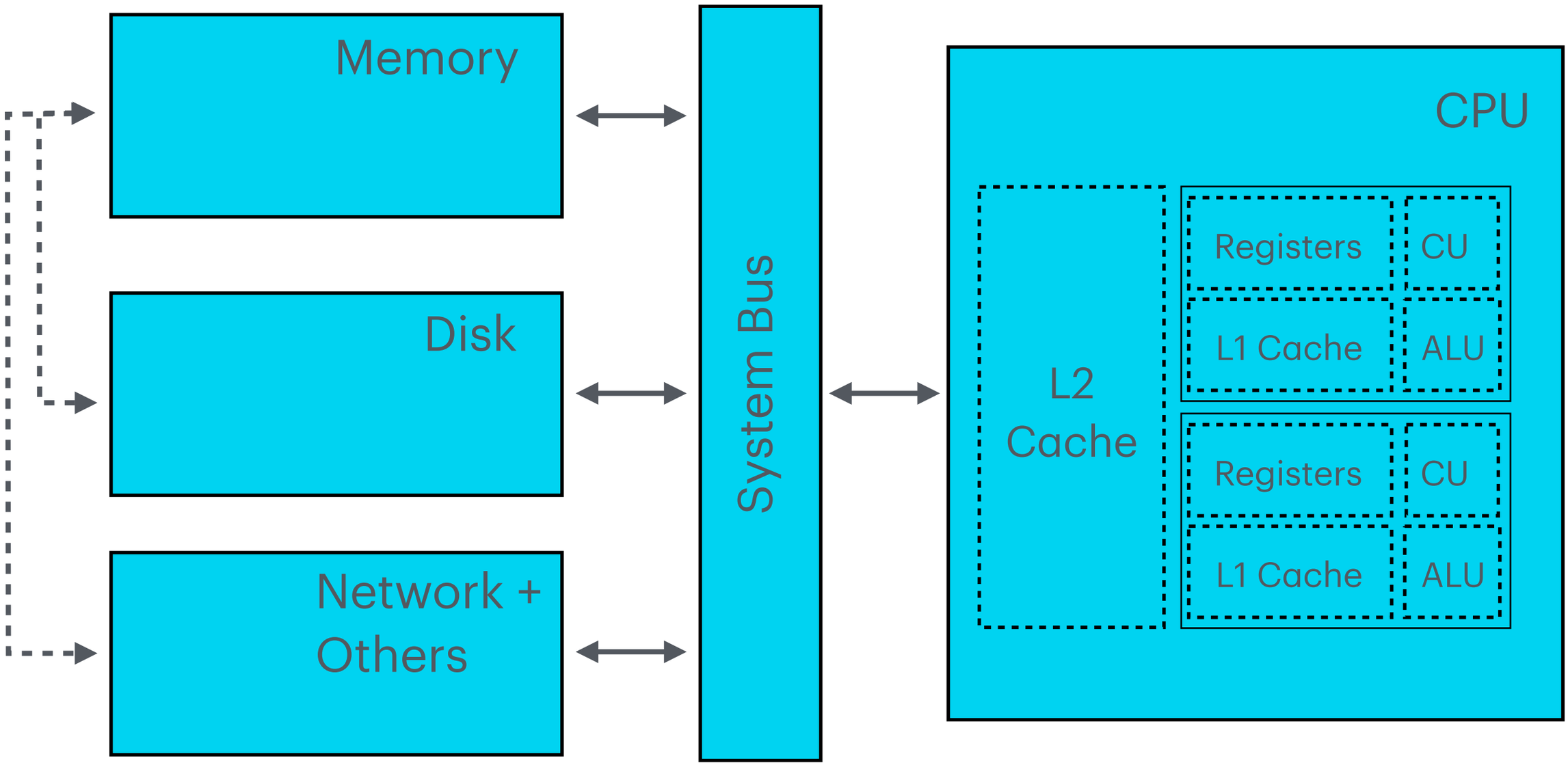
- We don’t claim any originality of the slides. Most of the content is borrowed from
  - Slides from lectures by Robert Wasinger’s great lecture on microarchitecture exploitation (https://pwn.college/software-exploitation/speculative-execution/)
  - Slides from Prof Ziming Zhao’s past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html) and CSE518 (https://zzm7000.github.io/teaching/2023fallcse410518/index.html)

Announcement

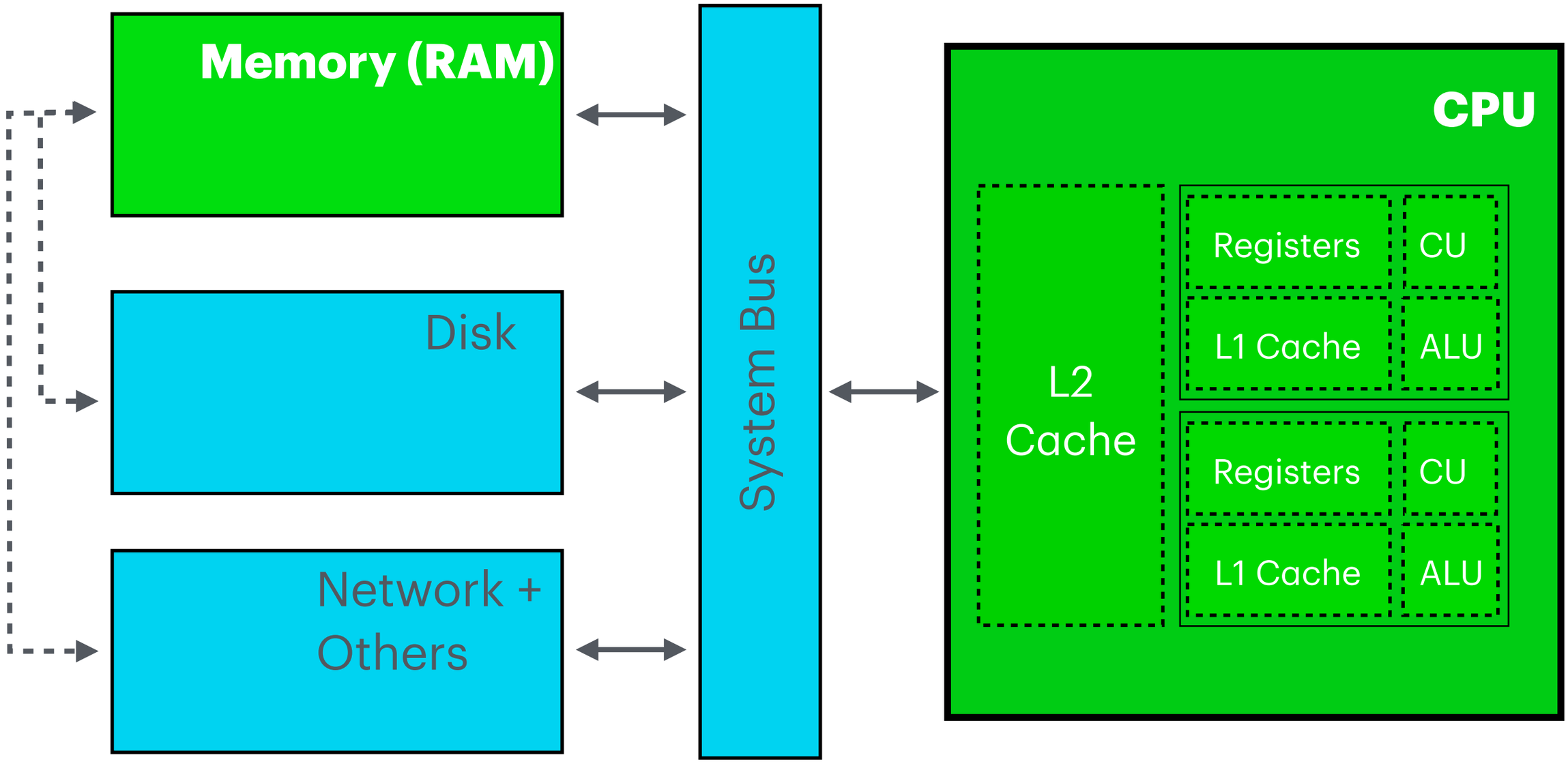
- Assignment 4 will be due **Sat Nov 30, 23:59**.

CPU Microarchitecture

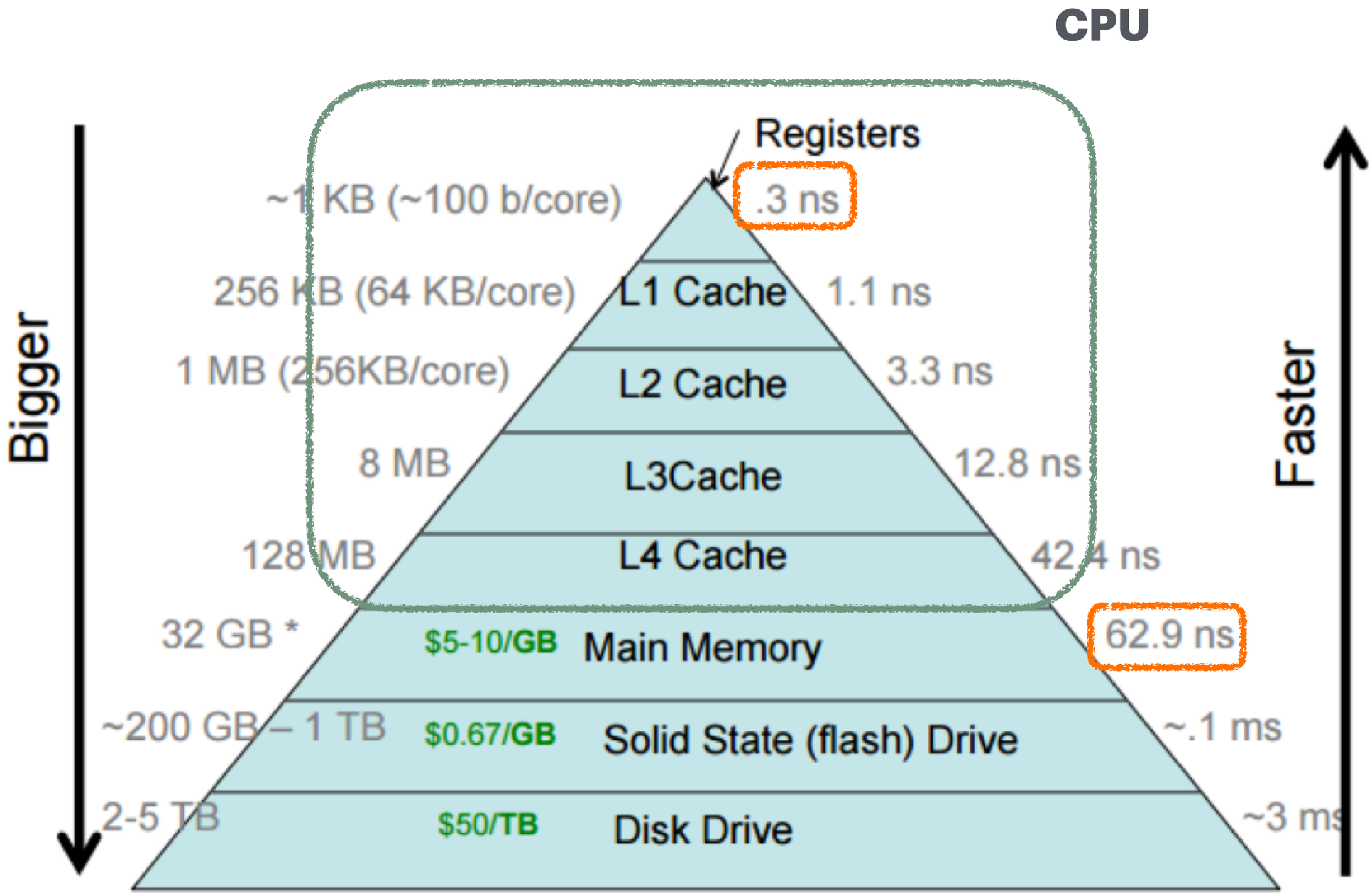
Computer Architecture



Computer Architecture

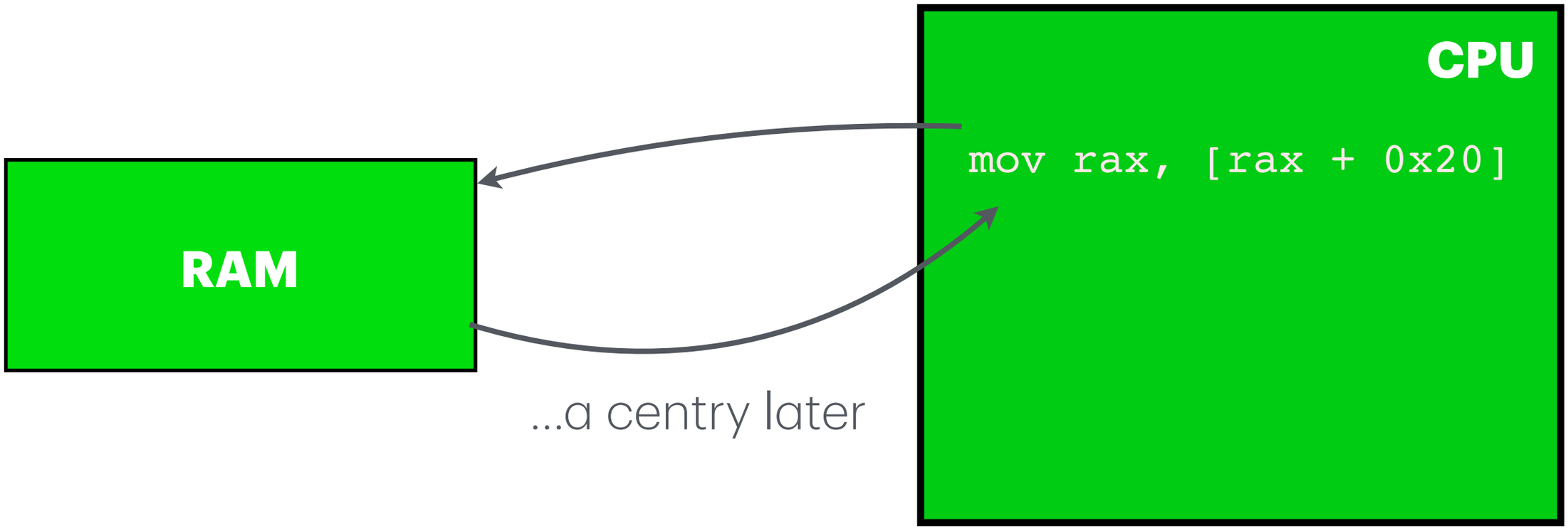


The Memory Hierarchy



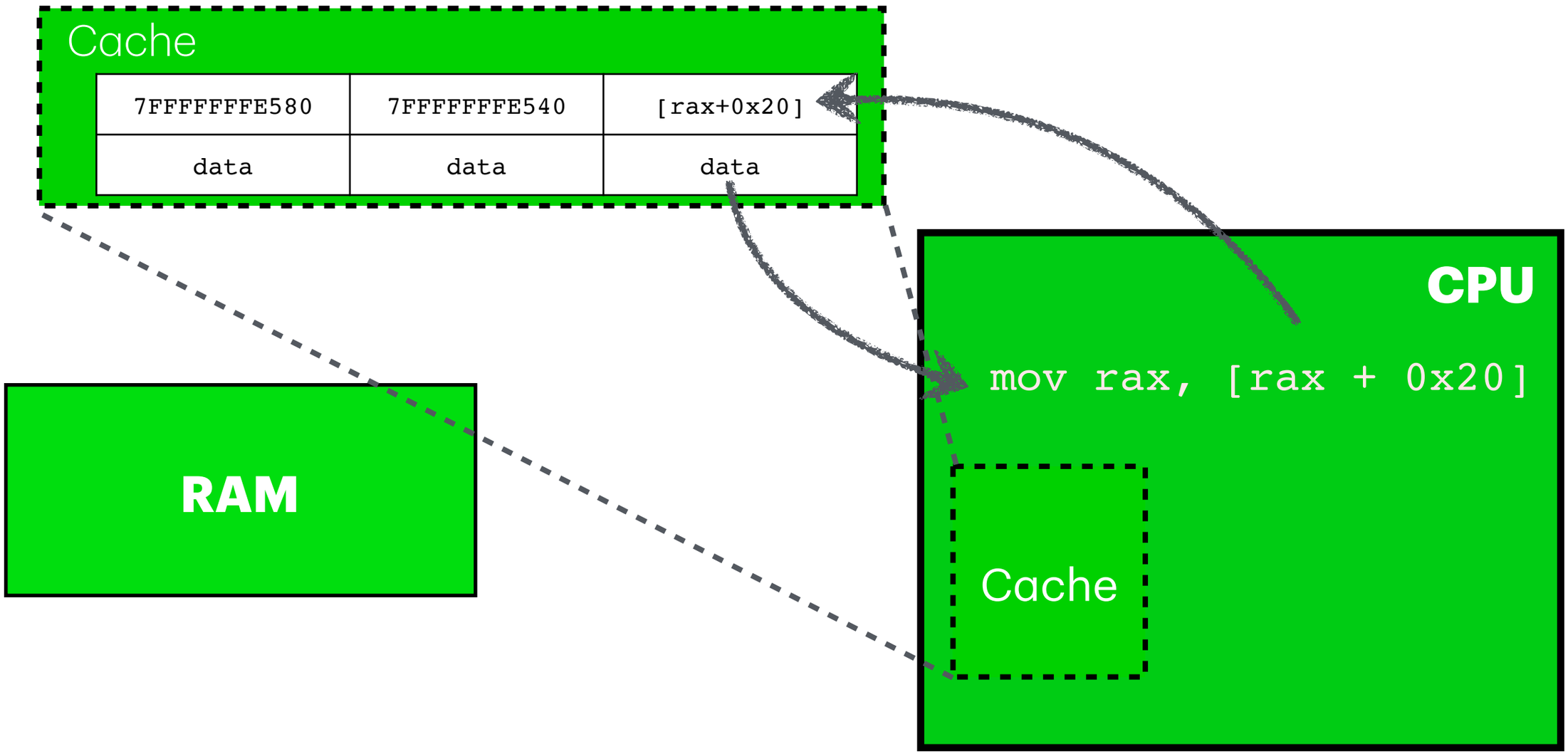
What happens inside the CPU?

- RAM is **“always”** accessed when used by an instruction
  - The CPU is much *faster* than memory access (0.5 ns v.s. 60 ns).
  - The CPU spends a *long* time waiting..



What happens inside the CPU? Caching!

- RAM is ~~**“always”**~~ accessed when used by an instruction? Not really.
  - The instruction may not touch RAM at all



What happens inside the CPU?

- Instructions are accessed in-order, as **`rip`** advances

```
mov rdi, 0
mov rax, [rax + 0x20]
cmp rax, 0
jz label
  mov rdi, 1
label:
  mov [rsi + 0x30], rdi
```

What happens inside the CPU?

- Instructions are accessed in-order, as **`rip`** advances

```
mov rdi, 0
mov rax, [rax + 0x20]
cmp rax, 0
jz label
  mov rdi, 1
label:
  mov [rsi + 0x30], rdi
```

Accessing RAM (slow)

Branching

What happens inside the CPU? Speculation!

- While waiting on other components the CPU can "speculate"
  - Guess branch to be taken
  - If incorrect, redo computation from branch
  - If correct, continue ahead with work already done
- This is occurs in the CPU hardware, but can be detected!

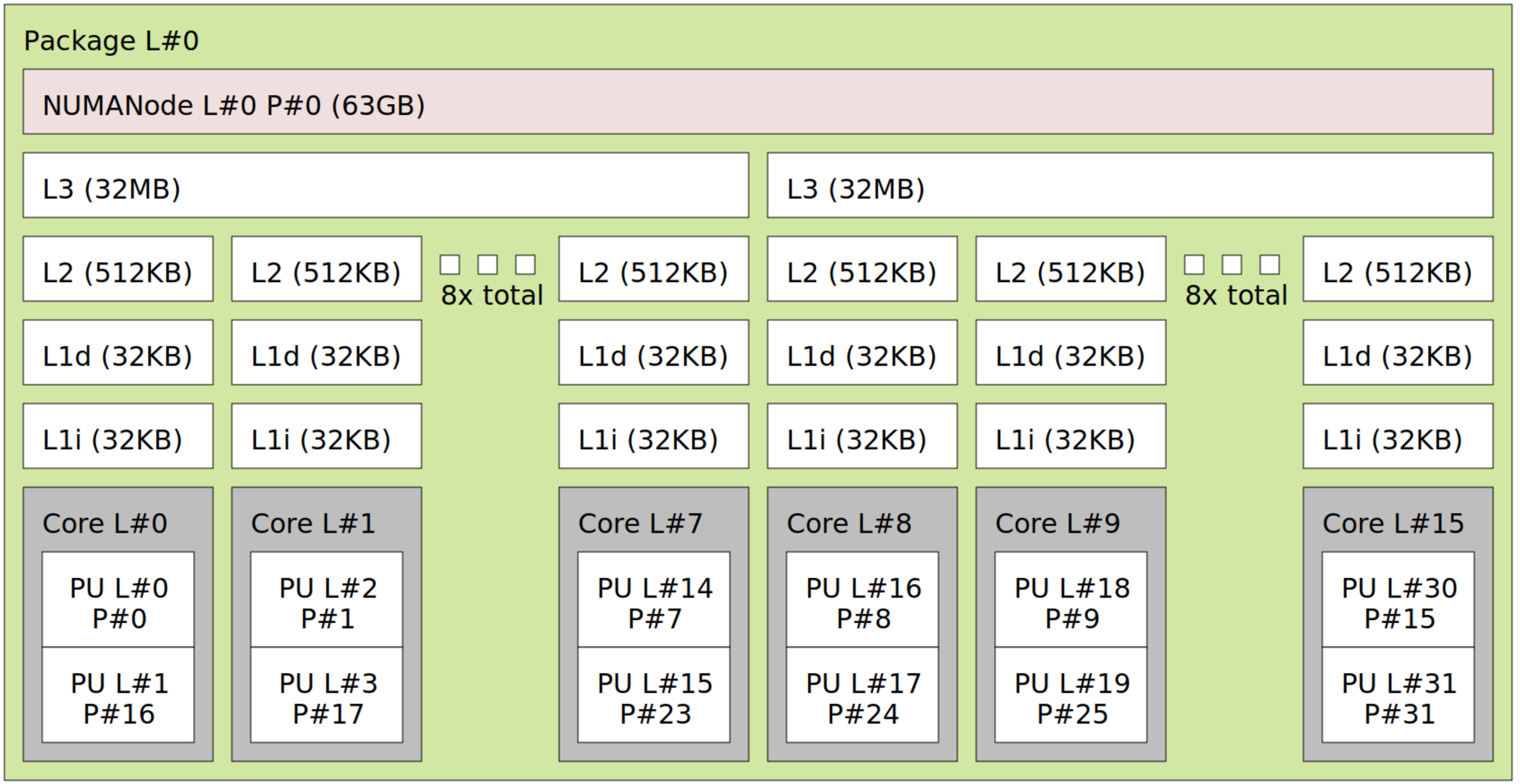
What happens inside the CPU? Speculation!

- Exact behavior varies on CPU
  - Chip specifics: Vendor, Model, etc
  - OS Security features enabled
- CPU details can be view in Linux via:
  - `cat /proc/cpuinfo`
  - `lscpu`

CPU Microarchitecture

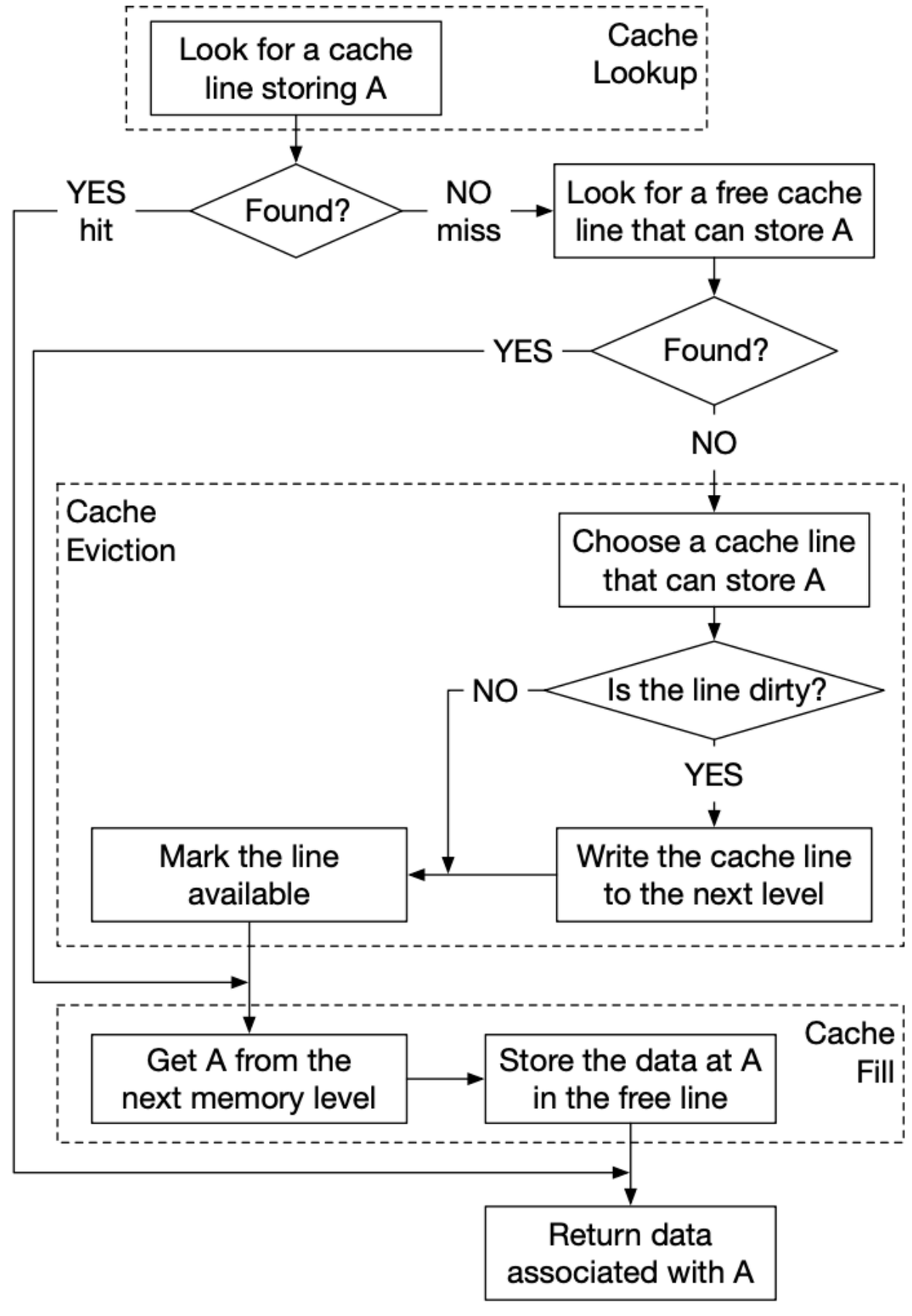
Cache

The Processor Die



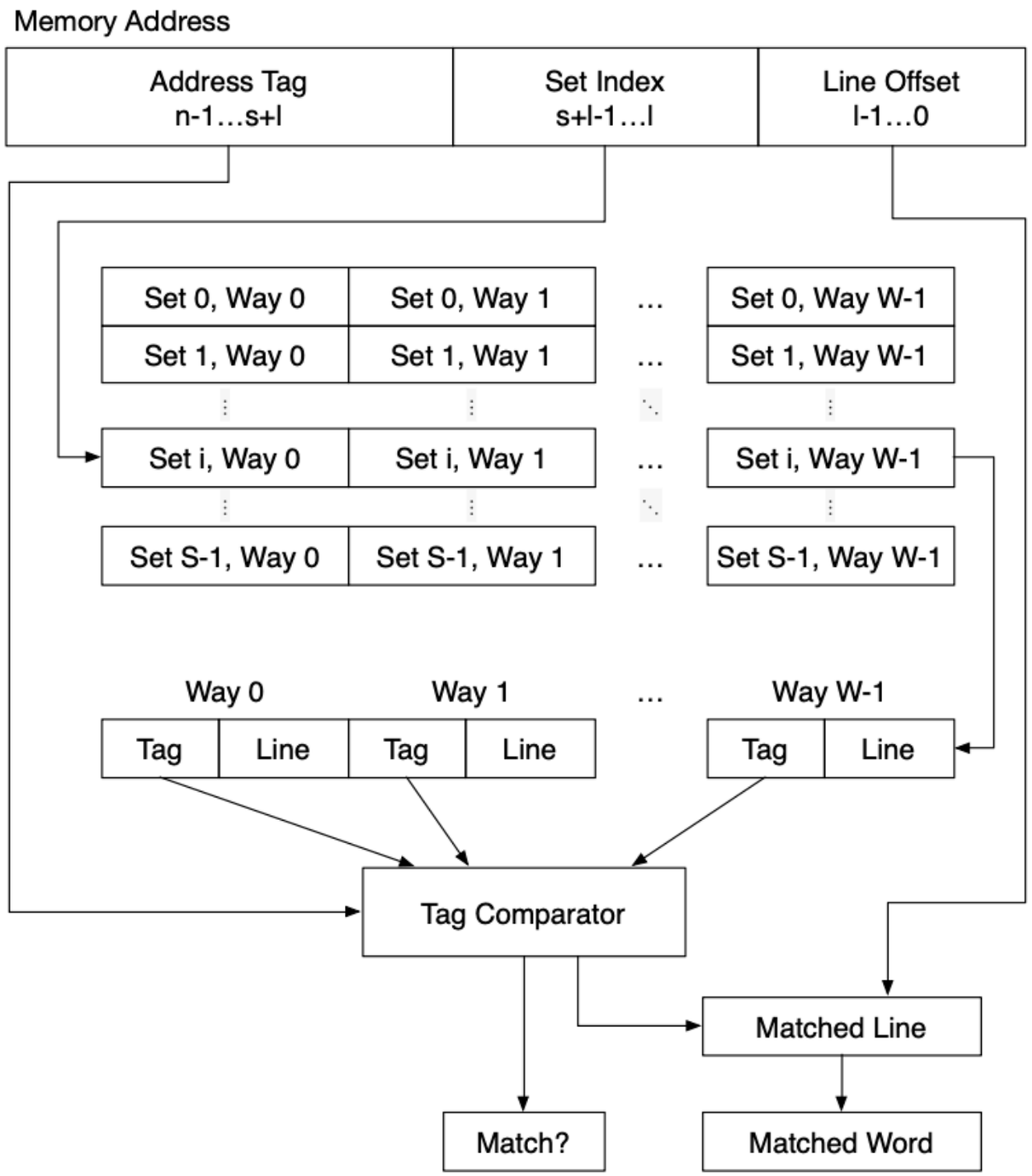
Cache policy

- **Cache lookup**: determine if the data exists in cache
  - Cache hit: data found.
  - Cache miss: Not found
    1. Cache eviction: allocate in-cache space for the data
    2. Cache fill: fetch data from the memory



Cache line

- **Cache line**: unit of cache organization
  - data: a copy of a continuous range of the RAM
  - tag: identify the memory addr that the data comes from



Caching as a side channel

- Requirements
  - Ability to influence target execution
  - Access to memory region used by target
    - Could be in-process
    - Could be shared memory
    - Could be kernel memory

Caching as a side channel

- General Strategy
  - Flush cache
  - Run victim code
  - Time access to same memory address
  - Compare timings
- This can be used to detect whether a region of memory was accessed!

Memory	Size	Access Time
Core Registers	1 KB	no latency
L1 D-Cache	32 KB	4 cycles
L2 Cache	256 KB	10 cycles
L3 Cache	8 MB	40-75 cycles
DRAM	16 GB	60 ns

Flush-and-Reload

Influencing CPU Behavior - Caching

- Extremely low level, but still to influence by code: **`#include <emmintrin.h>`**
  - Programs can be “pinned” to a cpu core via **`sched_setaffinity`**
  - Timing should be taken directly from the cpu via **`__rdtsc`**
  - Cache entry can be evicted with **`_mm_clflush(addr)`**
- Corresponding assembly versions: **`rdtsc`**, **`clflush`**

Influencing CPU Behavior - Speculating

- **Fences** force all prior memory stores/loads to be truly completed
  - `_mm_lfence` - wait for all memory loads to be completed
  - `_mm_sfence` - wait for all memory stores to be completed
  - `_mm_mfence` - wait for all memory operations to be completed

Caching as a side channel

- Requirements
  - Ability to influence target execution
  - Access to memory region used by target
    - Could be in-process
    - Could be shared memory
    - Could be kernel memory

Caching as a side channel

- General Strategy
  - **Flush** cache
  - Run victim code
  - **Time** access to the *same* memory address (“Reload”)
  - Compare timings
- This can be used to detect whether a region of memory was accessed!

Memory	Size	Access Time
Core Registers	1 KB	no latency
L1 D-Cache	32 KB	4 cycles
L2 Cache	256 KB	10 cycles
L3 Cache	8 MB	40-75 cycles
DRAM	16 GB	60 ns

Flush and Reload - Flush the Cache

- Flush cache by calling **`_mm_clflush(addr)`** on target addresses

Cache

7FFFFFFFE580	7FFFFFFFE540	[rax+0x20]
data	data	data

RAM

CPU execution engine

Victim Assembly

```
➞ cmp rdi, rsi
  jne skip
  mov rax, [rax + 0x20]
  ret
  skip:
  nop
```

Flush and Reload - Flush the Cache

- Once `[rax+0x20]` is flushed, we can trigger the victim to run

Cache

7FFFFFFFE580	7FFFFFFFE540	
data	data	

RAM

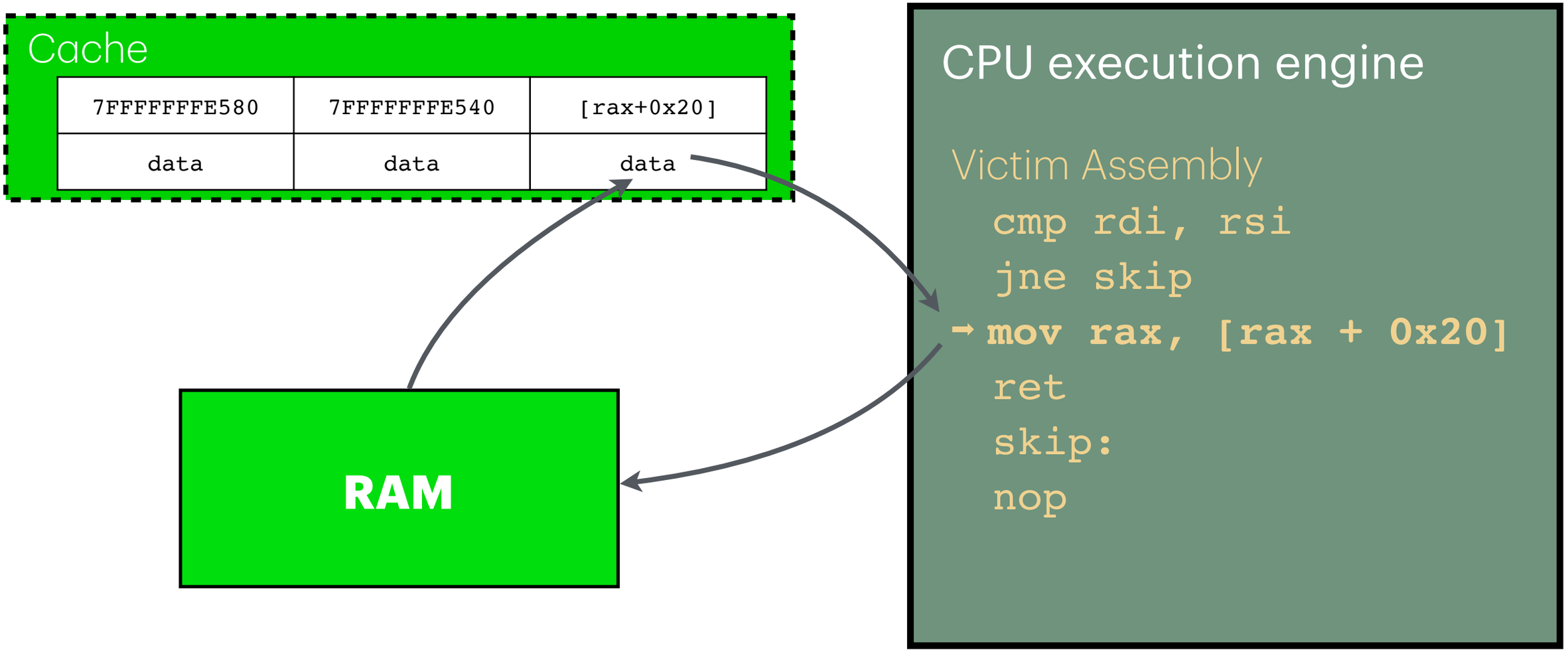
CPU execution engine

Victim Assembly

```
➡ cmp rdi, rsi
  jne skip
  mov rax, [rax + 0x20]
  ret
  skip:
  nop
```

Flush and Reload - Run victim code

- If the victim access an address the entry will be put in the cache



Flush and Reload - Run victim code

- If the victim does not access the address, nothing changes

Cache

7FFFFFFFE580	7FFFFFFFE540	
data	data	

RAM

CPU execution engine

Victim Assembly

```
  cmp rdi, rsi
  jne skip
  mov rax, [rax + 0x20]
  ret
→ skip:
  nop
```

Flush and Reload - Time Mem Access

- Attacker can time their independent memory access to learn the result

Cache

7FFFFFFFE580	7FFFFFFFE540	?
data	data	?

RAM

CPU execution engine

Attacker Pseudocode

```
start_timer()
mov rax, [rax + 0x20]
end_timer()
```

What's leaked

- Attacker knows that *certain memory address has been accessed* by the victim

- But there can be more: attacker can learn the *actual content* in the memory location

CPU execution engine

Victim Assembly

```
cmp rdi, rsi
jne skip
mov rax, [rax + 0x20]
ret
skip:
nop
```

What's leaked

- Attacker can learn the *actual content* in the memory location.
  - A (not working) example:

Victim code:

```
char secret;
…
// read/write secret
…
```

Attacker code:

```
char *p = &secret;
char probe[256 * 4096];
// flush the probe array
…
probe[*p * 4096] += 1;
…
for (i=0; i<256; ++i) {
   // time the access to each probe[i*4096]
   // secret=i with the fastest acc time
}
```

What's leaked

- Attacker can learn the *actual content* in the memory location **?**
- **Issues**:
  - Usually the attacker and victim code are running as different processes, i.e., living in independent virtual memory spaces.
    - So attacker has no way to gain **`&secret`**
    - Exceptions: shared memory; kernel space.
- Even if the attacker can somehow know **`&secret`**, he has no right to access it.
  - **`probe[*p * 4096] += 1`** will simply result in segment fault

CPU Microarchitecture

Out-of-order Execution

Transient Instructions

- Recall: instructions are speculatively executed

```
mov rdi, 0
mov rax, [rax + 0x20]
cmp rax, 0
jz label
  mov rdi, 1
label:
  mov [rsi + 0x30], rdi
```

Accessing RAM (slow)

Branching

Transient Instructions

- There are many slow patterns
  - Floating point operations
  - Dependent instruction chains
  - Uncached memory access
  - Keeping a physical core busy with multiple processes
  - ...

Transient Instructions

- Transient instructions are instructions that can only “run” during speculation
- Transient instructions never execute during normal program operation

```
mov rax, 0
mov rdi, 0x1337bee0
mov rax, [rax]                  <-- SegFault: accessing addr 0x000
mov rdi, BYTE PTR [rdi + 0x0f]  <-- Transient instruction
```

Transient Instructions

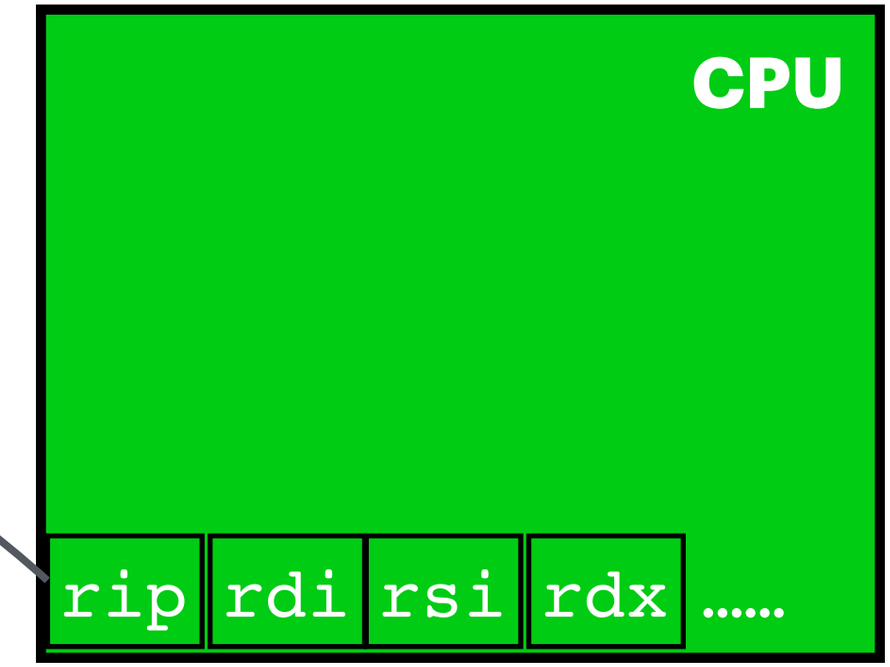
- Transient instructions do not impact the architectural cpu state
  - Not visible during execution
  - CPU registers are not impacted
  - For all “practical” purposes, these instructions didn’t run!

On the surface: sequential execution

- We reason about the cpu executing in-order
- We reason about the cpu state using architectural registers

- Pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [secret_value]
mov rdx, [rdi + rsi * 0x1000]
```

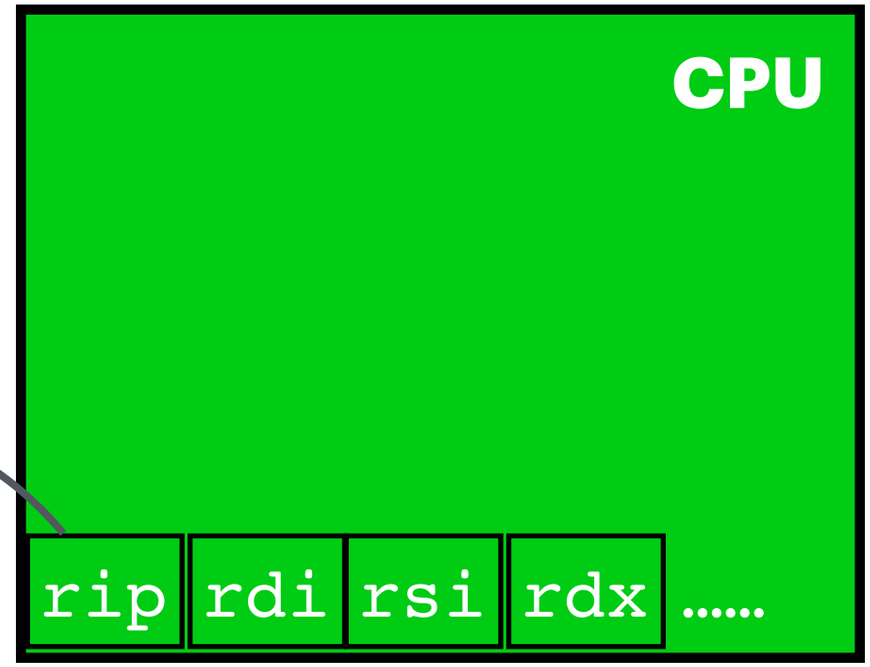


On the surface: sequential execution

- We reason about the cpu executing in-order
- We reason about the cpu state using architectural registers

- Pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [secret_value]
mov rdx, [rdi + rsi * 0x1000]
```

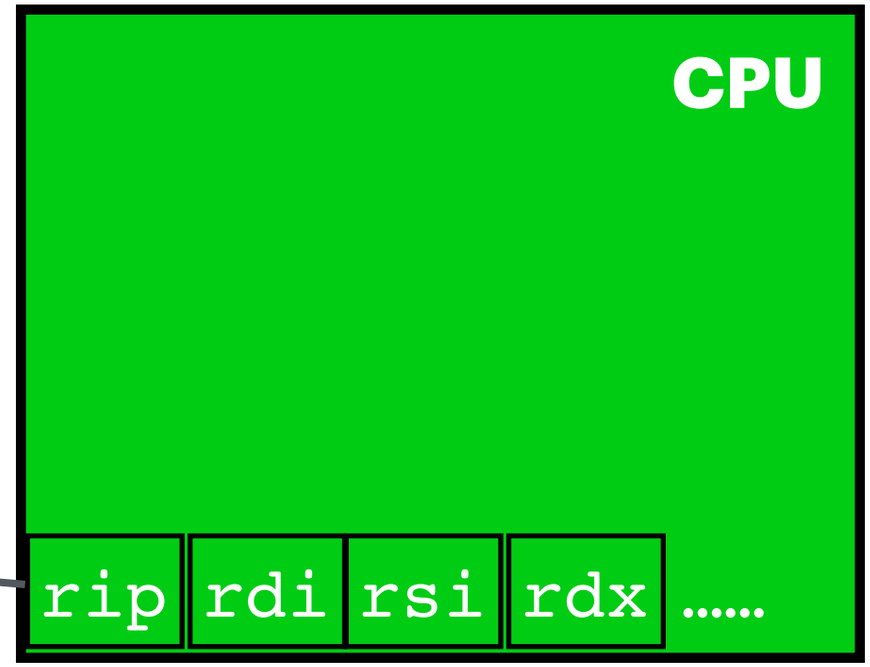


On the surface: sequential execution

- We reason about the cpu executing in-order
- We reason about the cpu state using architectural registers

- Pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [secret_value]
mov rdx, [rdi + rsi * 0x1000]
```

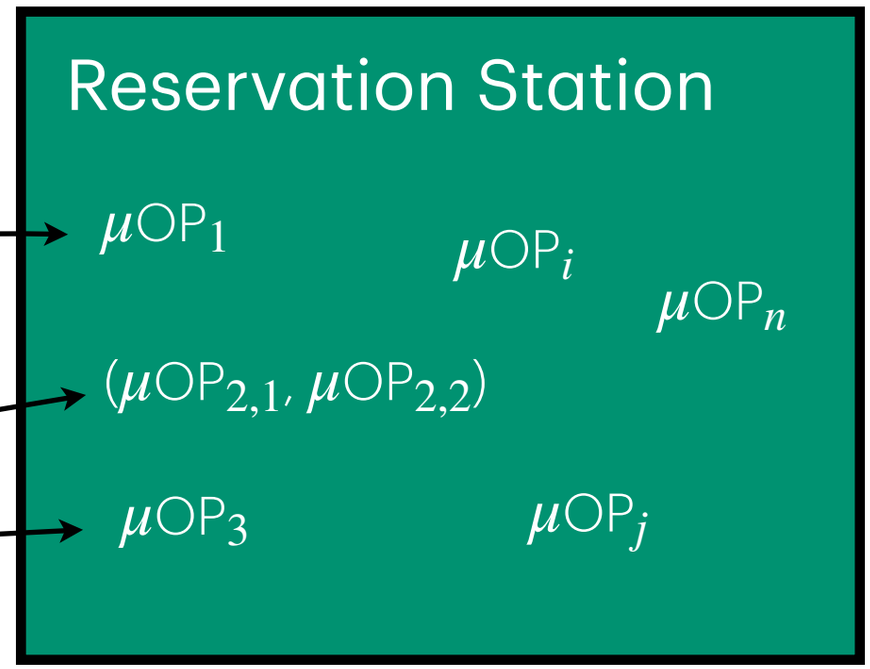


Micro-ops

- In actuality, assembly instructions get translated to **micro-ops** and passed to the **reservation station** component , which serves as a "staging area"

- Pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [secret_value]
mov rdx, [rdi + rsi * 0x1000]
```



Out-of-Order Execution

Reorder Buffer (ROB)

Scheduler

Reservation Station

μOP_b waiting for A

μOP_2 waiting for Z

μOP_1 waiting for X

μOP_3 waiting for Y

μOP_X computing X

μOP_Y computing Y

μOP_c

μOP_d

μOP_e

Execution Units

Out-of-Order Execution

Reorder Buffer (ROB)

Scheduler

Reservation Station

μOP$_b$ waiting for A

μOP$_2$ waiting for Z

μOP$_1$ waiting for X

μOP$_3$ waiting for Y

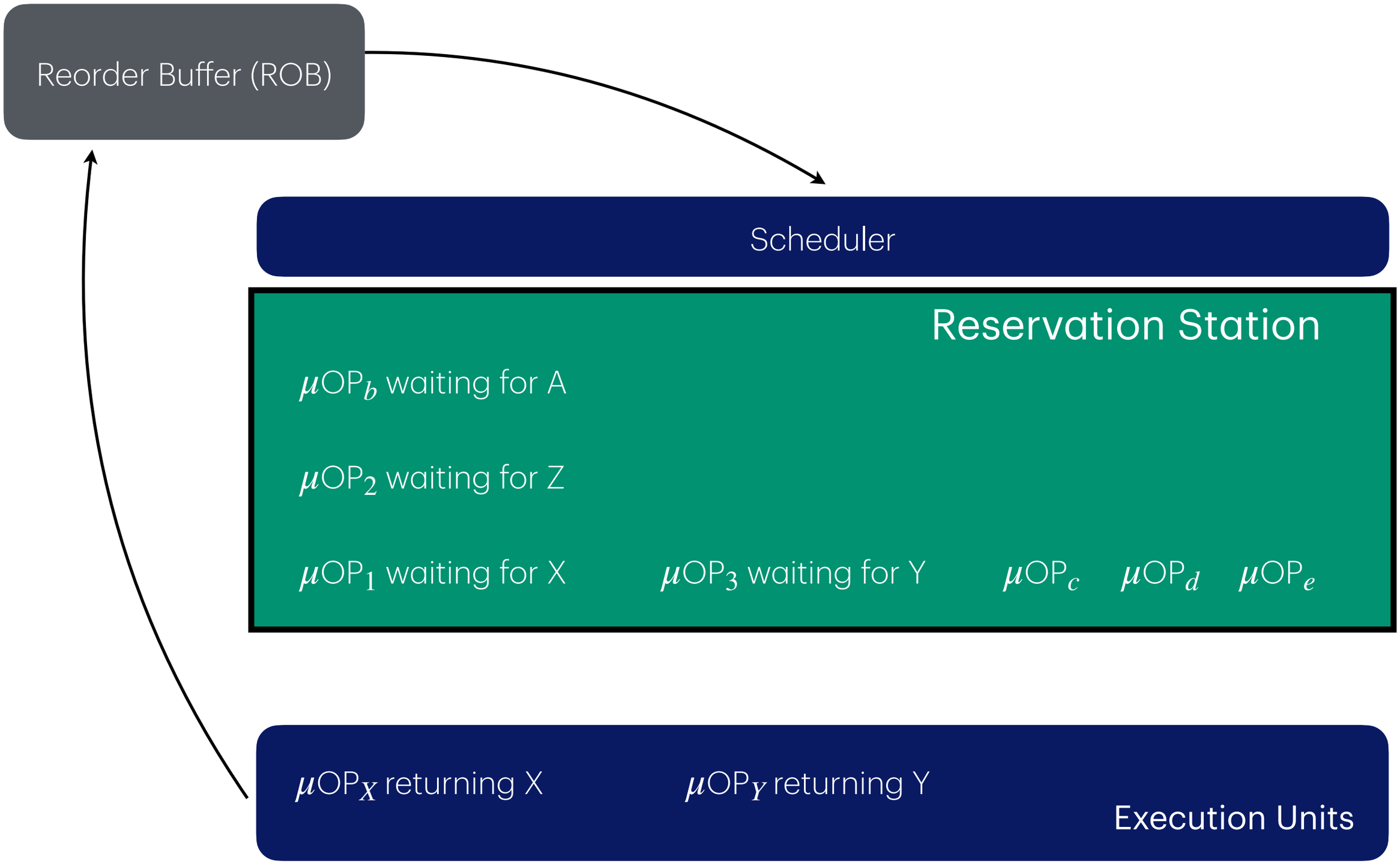
μOP$_c$ μOP$_d$ μOP$_e$

μOP$_X$ computing X

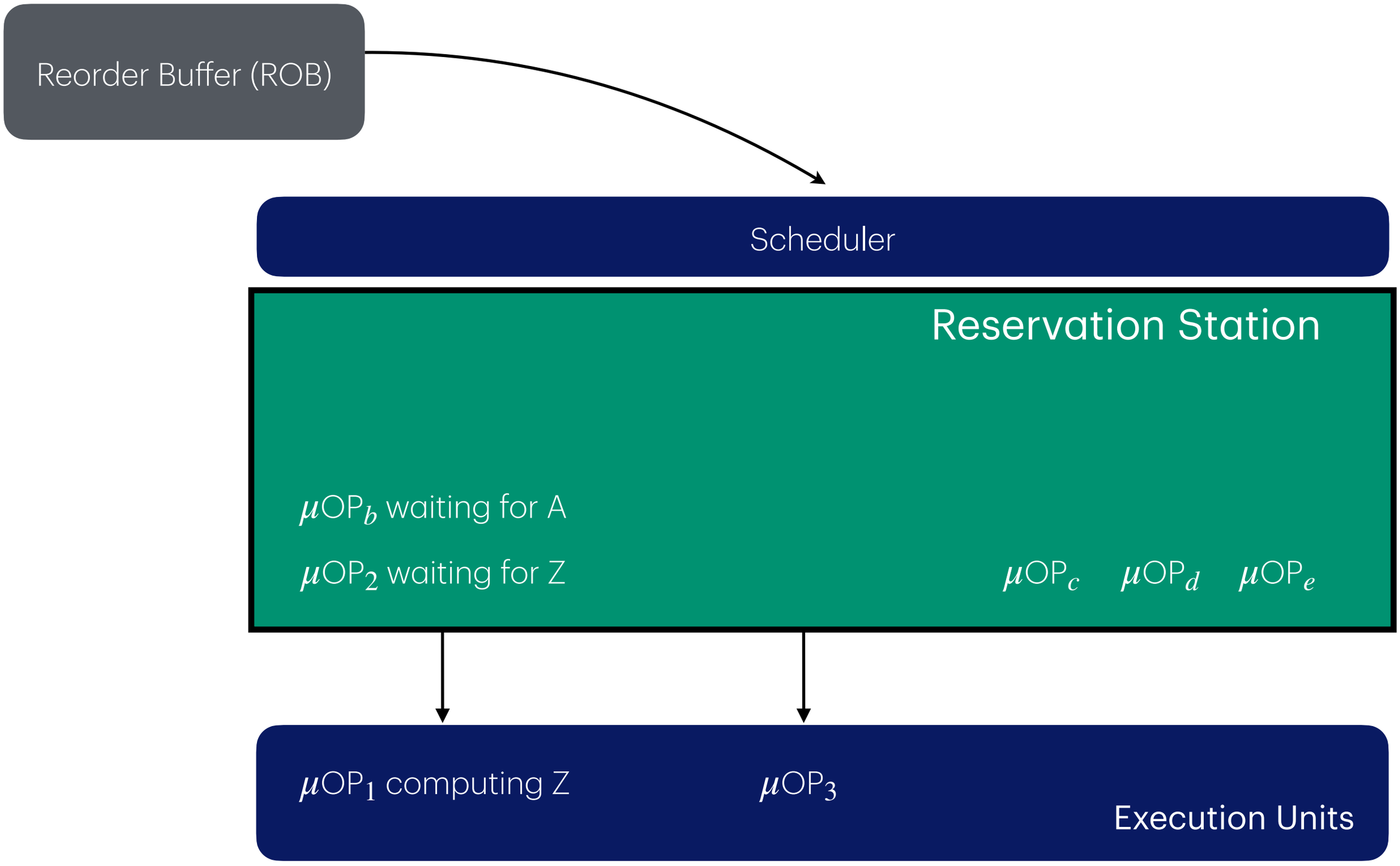
μOP$_Y$ computing Y

Execution Units

Out-of-Order Execution

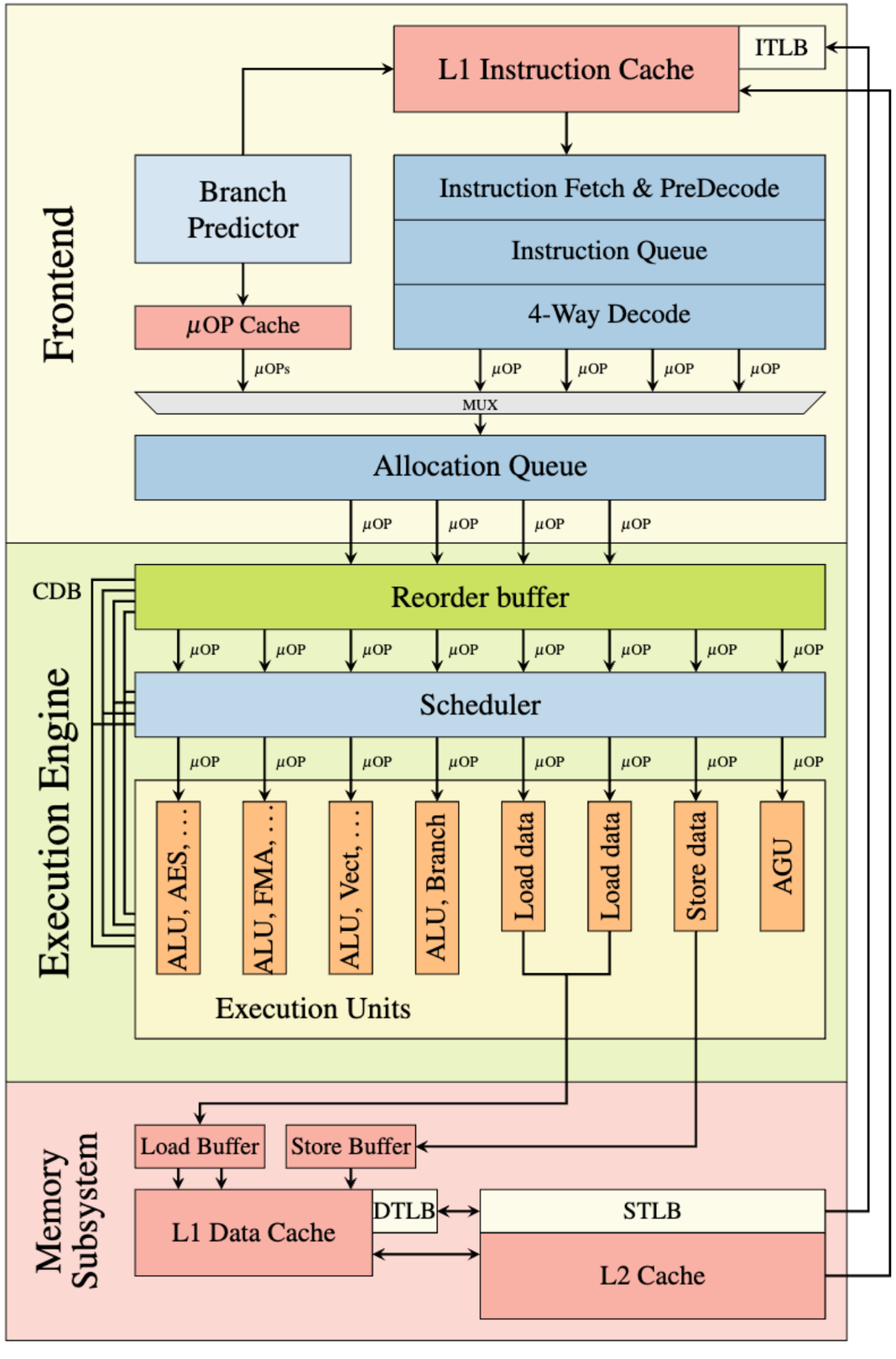


Out-of-Order Execution



Out-of-Order Execution

- **Out-of-Order Execution Engine**
  - Micro-ops received from the decode queue are written into Reorder Buffer (ROB)
- Scheduler (Reservation Station): select micro-ops in ROB for execution, and write the results back to ROB.
- ROB *commits* / *discards* the execution results of micro-ops.



Out-of-Order Execution

- Instructions are *fetched* and *decoded* by an in-order "front end"
- Instructions are *dispatched* to an out-of-order "backend"
  - Allocated an entry in the **ROB** (Re-Order Buffer), Reservation Stations
- Re-Order Buffer defines an execution window of out-of-order processing
  - These can be quite large – over 200 entries in contemporary designs

Out-of-Order Execution

- Instructions wait only until their dependencies are available
  - Later instructions may execute prior to earlier instructions
  - ROB allows for more physical registers than defined by the ISA

- Instructions complete (“retire”) in-order
  - When an instruction is the oldest in the machine, it is “retired”
  - State becomes *architecturally* visible (updates the architectural register file)

Why this complications?

- Adding this layer of abstraction allows the CPU to:
  - Intermediate results are stored in ROB: Execute many sequential instructions *without* updating a physical "register"
  - Reorder actions based upon true dependencies
  - Simultaneously have multiple versions of the same "register"!
  - Results are committed to physical register *in program order*
- This results in performance improvements while being architecturally invisible
- Or is it?

Meltdown

Meltdown: according to CPU architecture

Userland

Attacker pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [kernel_address]
mov rdi,[rdi + rsi * 0x1000]
```

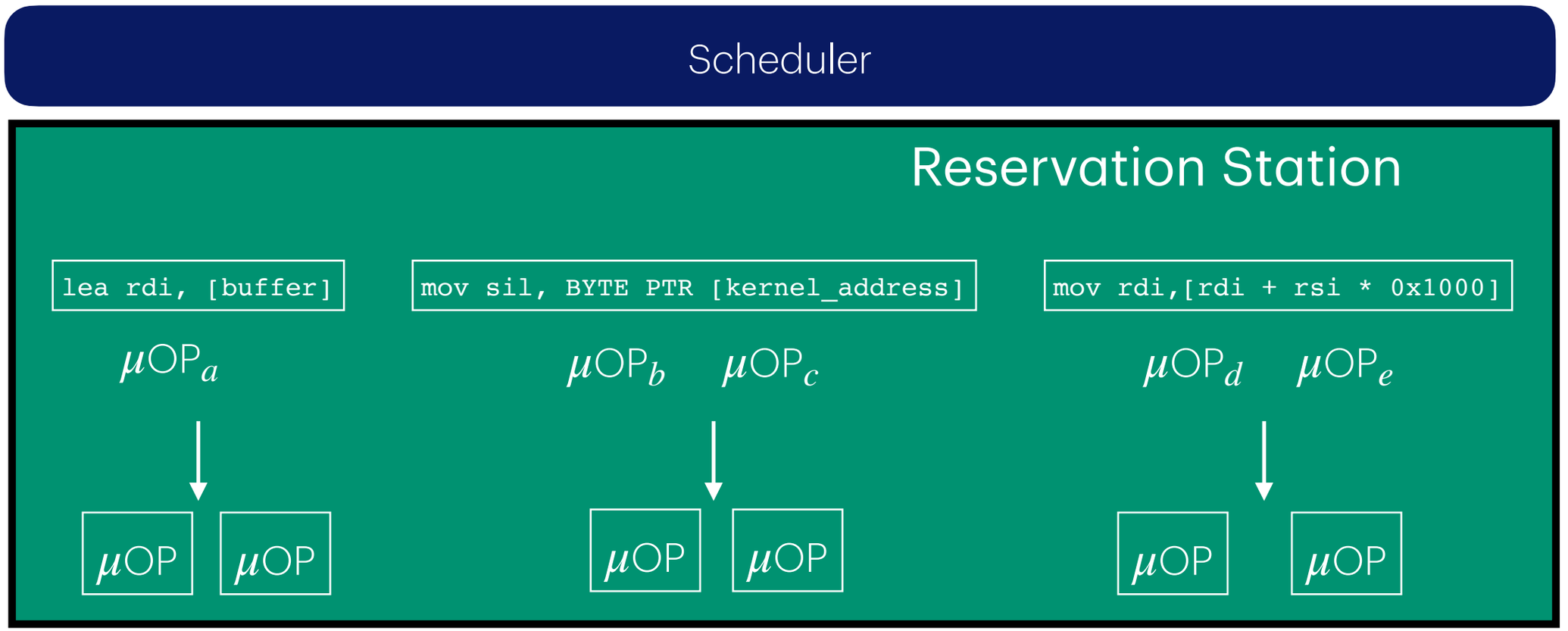
Segment fault

Transient instruction

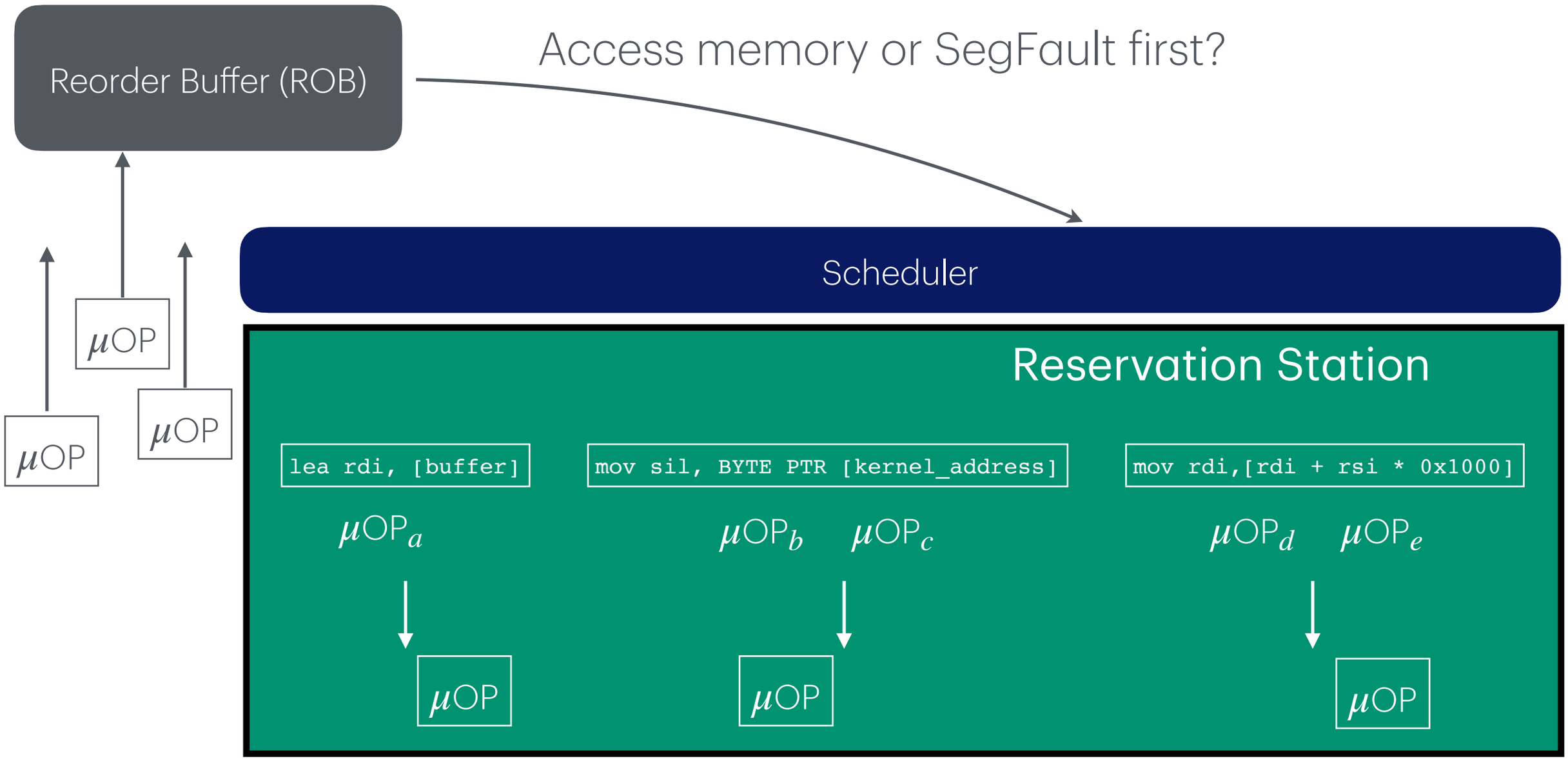
Meltdown: in micro-ops, a race condition!

Reorder Buffer (ROB)

Scheduler



Meltdown: in micro-ops, a race condition!



What is the prize?

- The transient instruction loads memory into the cache

Attacker pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [kernel_address]
mov rdi,[rdi + rsi * 0x1000]
```

Cache

7FFFFFFFE580	7FFFFFFFE540	**[rdi+rsi*0x1000]**
data	data	**data**

- What can this cache entry location tell us about the secret value stored?

What is the prize?

- **`rdi`** is the base address of an accessible buffer
  - Attacker can do flush-and-reload to it freely.
- **`rsi`** is a leaked byte from the kernel

Attacker pseudocode

```
lea rdi, [buffer]
mov sil, BYTE PTR [kernel_address]
mov rdi,[rdi + rsi * 0x1000]
```

What is the prize?

- Meltdown
  - Breaks the most fundamental isolation between user applications and the OS.
- Has been proven to be able to leak memory:
  - From the kernel to userland
  - From one process to another
  - From (mostly) any physical address (via the kernel)
  - With zero requirements other than a vulnerable CPU!

Mitigations

- **Kernel Page Table Isolation (KPTI)**:
  - Separates kernel and user memory spaces completely to prevent unauthorized access to kernel memory.
  - Introduced in major operating systems like Linux, Windows, and macOS.
  - Has non-trivial performance overhead.
- Microcode Updates:
  - Issued by CPU manufacturers to improve speculative execution controls and limit out-of-bounds memory access.

CPU Microarchitecture

Branch prediction

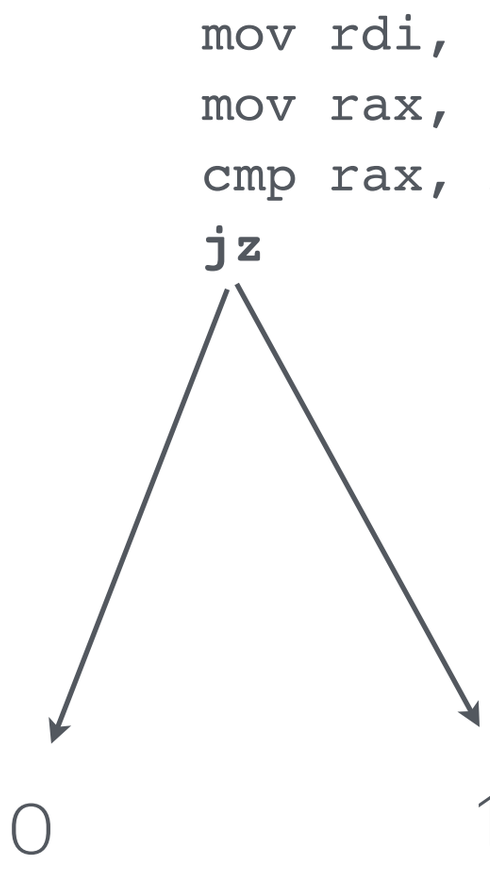
Past results may indicate future performance

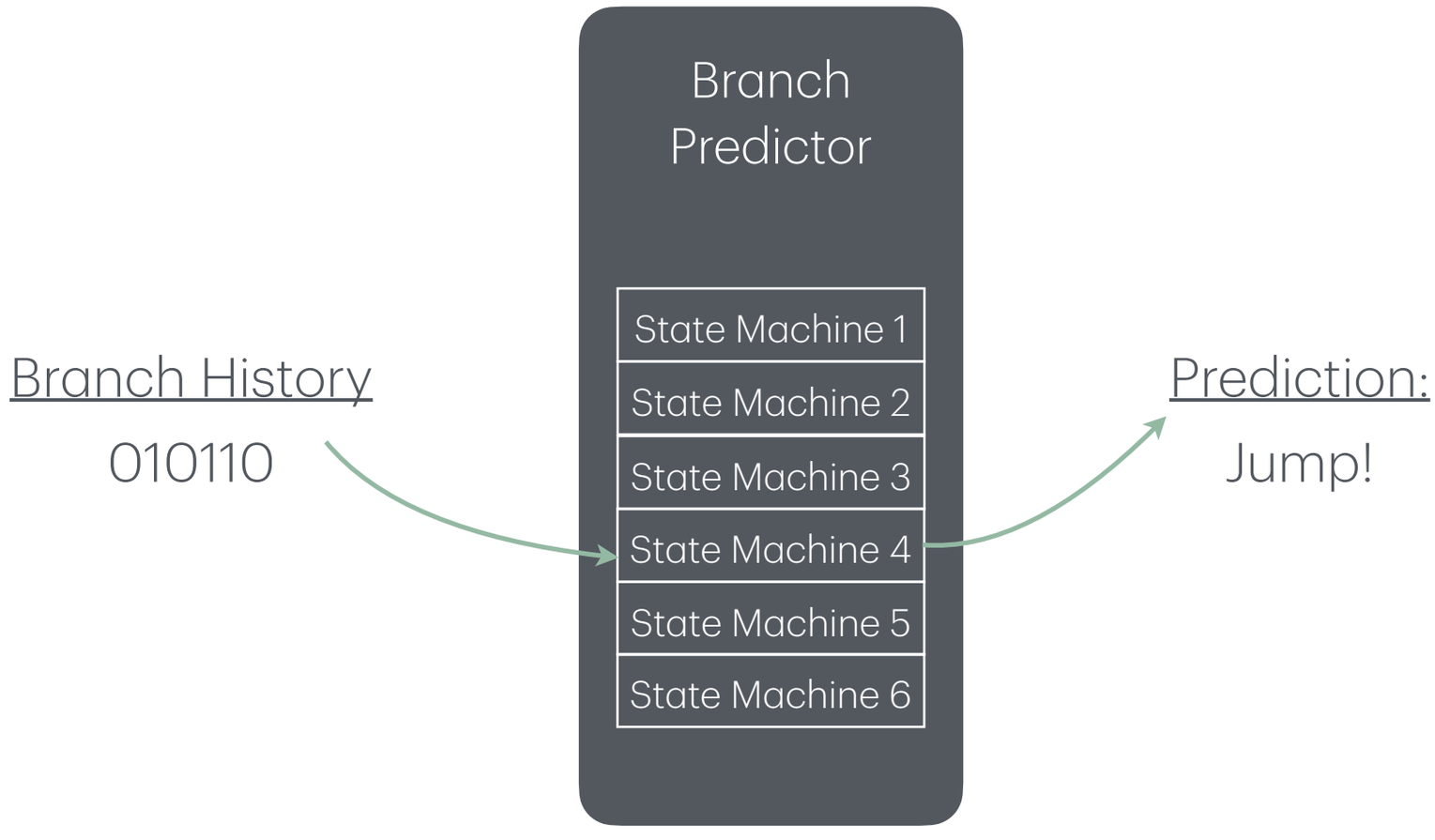
- The CPU branch predictor uses past execution to make predictions
  - Individual branch history at specific virtual addresses
  - Global branch history of all branching on core
- By repeatedly branching in one direction, the predictor can be *biased*

CPU Branch Predictors

Example: two-level predictor

```
mov rdi, [rdi]
mov rax, [rax]
cmp rax, rdi
jz
```

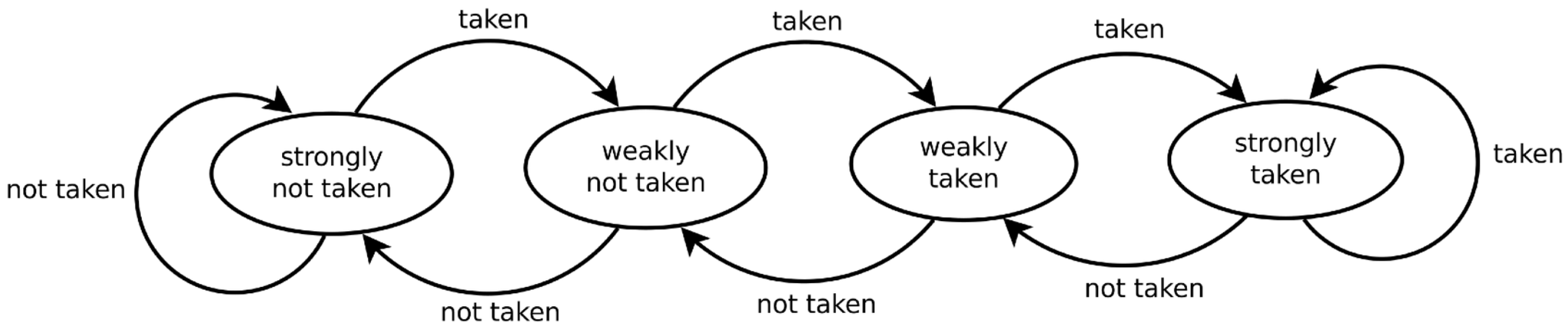




Depending on the recent history, the predictor will invoke a simple state machine to predict the outcome.

CPU Branch Predictors: 2-bit counter

State machine used in the predictor.



How to train your branch predictor

- Training is physical core specific
  - In the same process, possibly different context (kernel vs userland)
  - Across address-spaces (cross process)

- In many cases, training does not take many runs to start showing results!

Speculative Execution

- Implemented as a variation of Out-of-Order Execution
  - Uses the same underlying structures already present such as ROB, etc.
- Instructions that are speculative are specially tagged in the ROB
  - They must not have an architecturally visible effect on the machine state
  - Do not update the architectural register file *until* speculation is committed
  - Stores to memory are tagged in the ROB and will not hit the store buffers
- Exceptions caused by instructions will not be raised until instruction retirement
  - Tag the ROB to indicate an exception (e.g. privilege check violation on load)
  - If the instruction never retires, then no exception handling is invoked

Speculative Execution

- Once the branch condition is successfully resolved:
  1. If predicted branch correct, speculated instructions can be retired
     - Once instructions are the oldest in the machine, they can retire normally
     - They become *architecturally visible* and stores ultimately reach memory
     - Significant performance benefit from executing the speculated path
  2. If predicted branch incorrect, speculated instructions MUST be discarded
     - They exist only in the ROB, remove/fix, and discard store buffer entries
     - They do not become *architecturally visible*
     - Performance hit incurred from flushing the pipeline/undoing speculation

Spectre

Spectre V1

```
if (x < array1_size) {
  y = probe[array1[x] * 4096];
}
```

- This block of code never executes if **`x > array1_size`**.
- Spectre V1 relies on the fact the CPU can speculate it does.

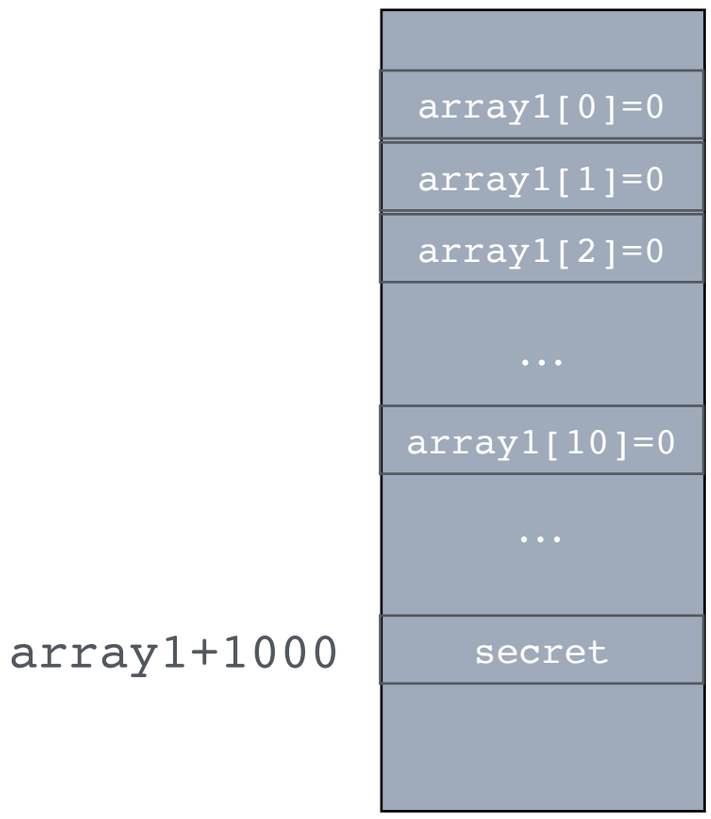
Spectre V1: Training

```
array1_size = 10
x = 1
```



```
if (x < array1_size) {
  y = probe[array1[x] * 4096];
}
```

Memory



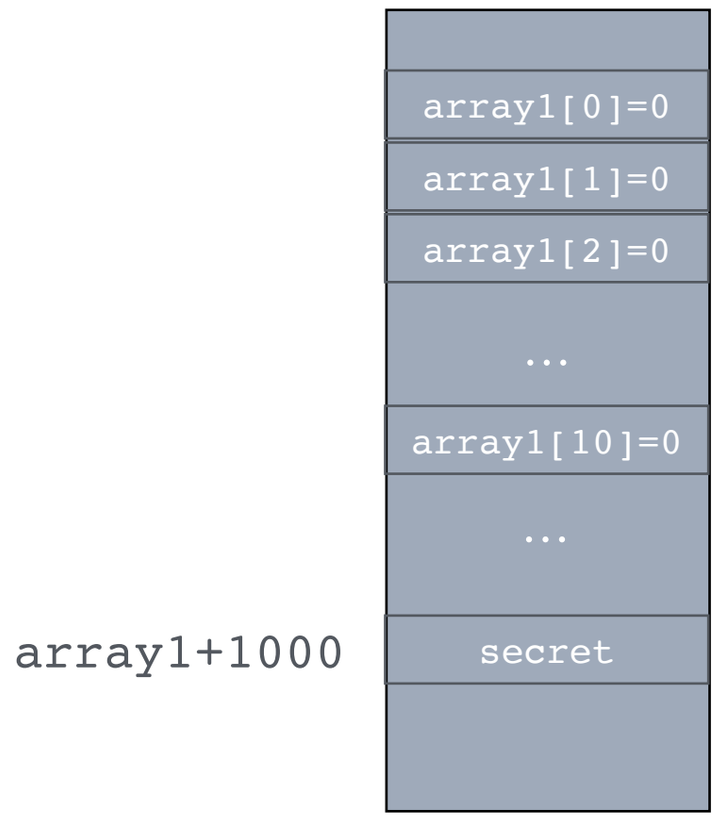
Repeatedly passing the check *trains* the branch predictor to enter this block

Spectre V1: Exploiting

```
array1_size = 10
x = 1000
```



```
if (x < array1_size) {
  y = probe[array1[x] * 4096];
}
```



Speculatively, **`array1`** will be indexed out of bounds by a transient instruction. This allows arbitrary memory access while speculating.

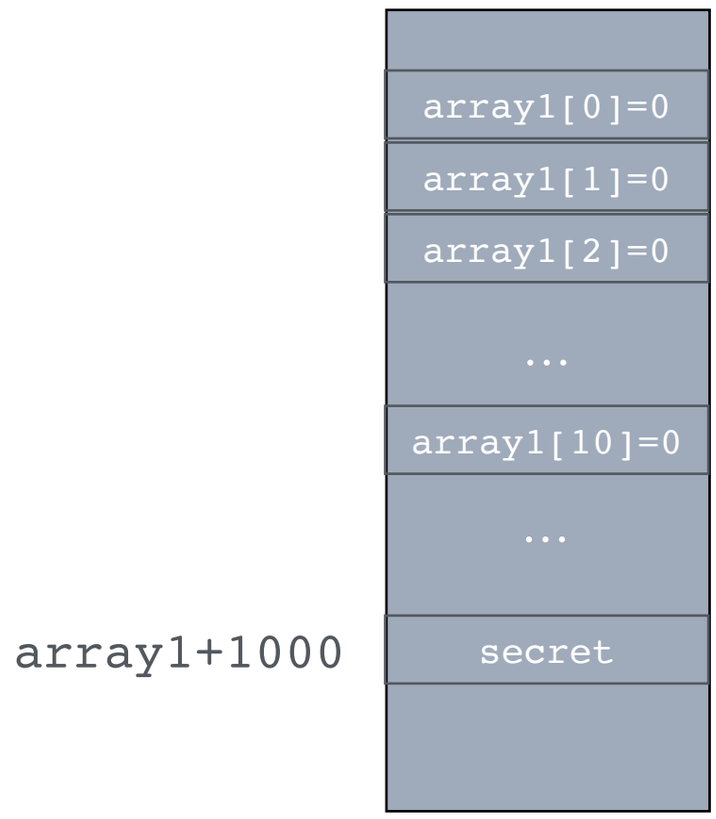
Spectre V1: Exploiting

```
array1_size = 10
x = 1000
```



```
if (x < array1_size) {
  y = probe[secret * 4096];
}
```

Memory



Speculatively, **`array1`** will be indexed out of bounds by a transient instruction. This allows arbitrary memory access while speculating.

Spectre V1

```
if (x < array1_size) {
  y = probe[array1[x] * 4096];
}
```

- If **`array1_size`** takes a long time to access, the CPU can *speculate*:
  - When the CPU speculates, it predicts a branch
  - Incorrectly predicting results in transient instructions being executed
  - Transient instructions can influence the cache

Spectre V2

- This example jumps to an address defined in memory at 0xdeadbeef

```
mov rdi, [rdi]
mov rsi, 0xdeadbeef
call QWORD PTR [rsi]
```

Any "slow" instruction that encourages speculation

- The CPU will attempt to predict where to jump and continue execution.

- Training this predictor can cause arbitrary instructions to be executed as transient instructions!

Spectre V2

- The attacker influences branch predictor by calling:
  - Valid addresses in a different context
  - Different virtual addresses that access the same physical address
  - The branch predictor will predict jumping to these new locations
- Executing again in a different context may trigger jumping to a spectre gadget

Spectre V2: Gadgets

Instructions that when speculatively executed, influence microarchitecture state

Gadget

```
y = array2[array1[x] * 4096];
```

```
mov rdi, [rbx+rdx]
mov dl,byte ptr [rdi]
```

Influencing **`rbx`** or **`rdx`** changes what address will be speculatively accessed

Here, the data at **`rbx+rdx`** can be leaked via a cache side-channel

Mitigations

- **Retpoline** (Return Trampoline):
  - Software-based mitigation that prevents branch target injection by ensuring speculative execution does not use indirect branch predictions.
- Bounds Checking and Speculation Barriers:
  - Insert speculative execution barriers (e.g., `lfence` instruction on Intel CPUs) to prevent speculation past certain points.
- Enhanced Branch Prediction Isolation:
  - Microcode updates to restrict branch prediction history sharing across contexts (e.g., between processes or threads).

Questions?