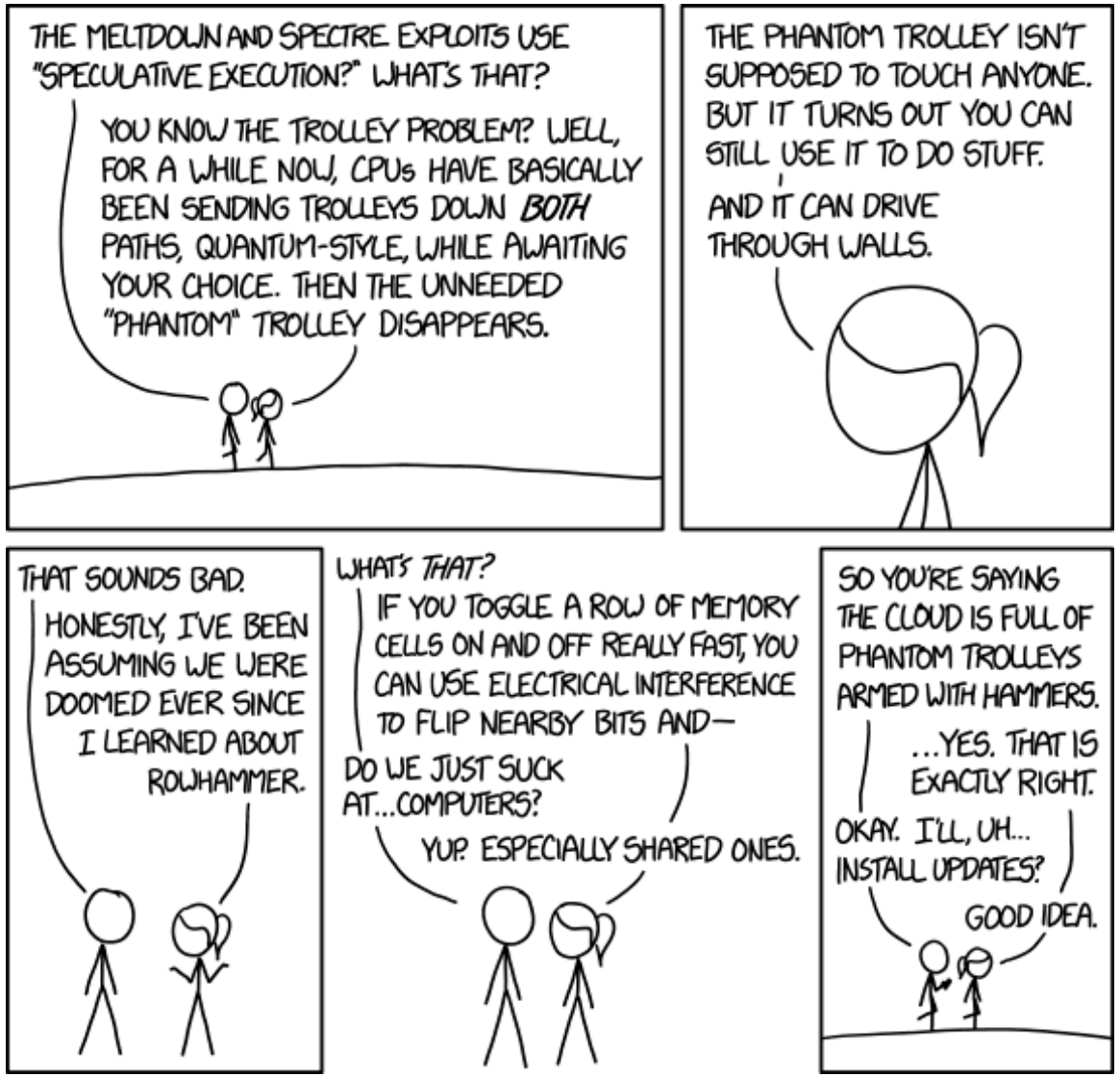


https://xkcd.com/1938/

Side Channels

System Security (CM0625, CM0631) 2022-23
Università Ca' Foscari Venezia

Matteo Busi
matteo.busi@unive.it
https://matteobusi.github.io/

An appetizer [Van Bulck et al., 2018]

https://youtu.be/8ZF6kX6z7pM

[Van Bulck et al., 2018] Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." *USENIX Security 2018*.

An appetizer [Van Bulck et al., 2018]

https://youtu.be/8ZF6kX6z7pM

[Van Bulck et al., 2018] Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." *USENIX Security 2018*.

People talk a lot about…

- Side channel attacks
- Constant-time programs
- Speculative execution
- Speculative leaks
- …

People talk a lot about…

- Side channel attacks
- Constant-time programs
- Speculative execution
- Speculative leaks
- …

What's all this jargon?

We'll spend the next 1.5 hours trying to decode the jargon

Computer Architecture 101

Remember how CPUs work?

Computer Architecture 101

Remember how CPUs work?

Computer Architecture 101

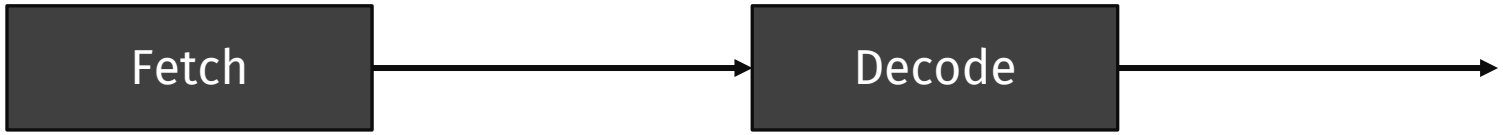
Remember how CPUs work?

Computer Architecture 101

Remember how CPUs work?

Computer Architecture 101

Remember how CPUs work?

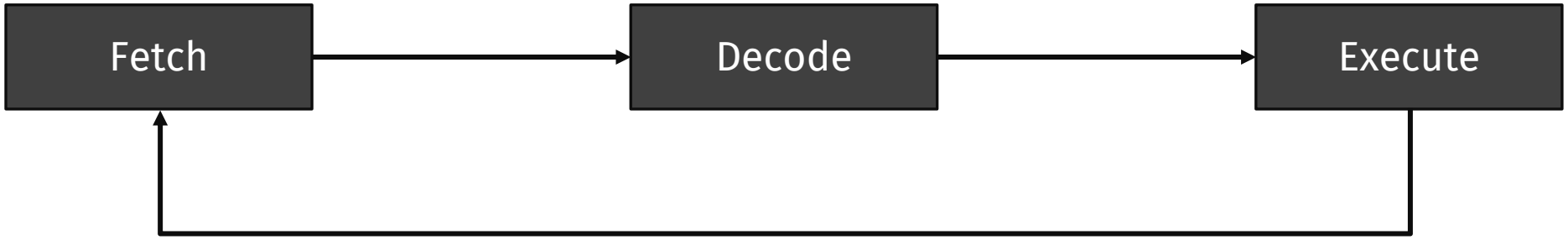


Computer Architecture 101

Remember how CPUs work?

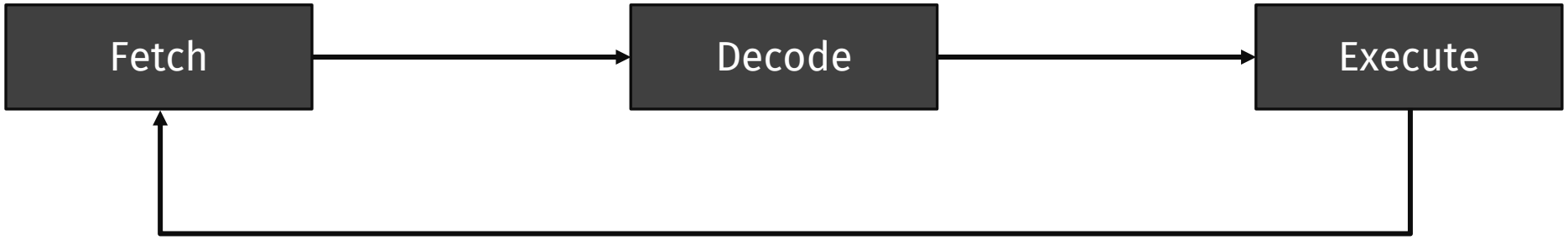
Computer Architecture 101

Remember how CPUs work?



Computer Architecture 101

Remember how CPUs work?



Problem: not very efficient (≥1 cycle per stage)

How to make processors faster?

- **First idea:** make cycles “shorter”, i.e., **increase** the processor’s clock

How to make processors faster?

- **First idea:** make cycles “shorter”, i.e., **increase** the processor’s clock
  - We can reach about 4 GHz
  - **Very good**, but ~2004 we reached the peak
  - Also: power-consumption issues, too much heat, …

How to make processors faster?

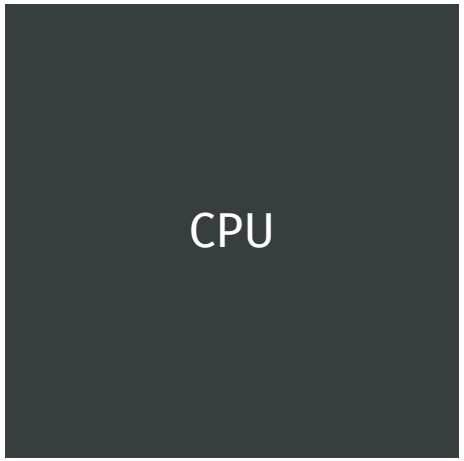
- **First idea:** make cycles “shorter”, i.e., **increase** the processor’s clock
  - We can reach about 4 GHz
  - **Very good**, but ~2004 we reached the peak
  - Also: power-consumption issues, too much heat, …
- People started thinking about alternatives!
  - **Idea:** make the avg case faster
  - **Solutions:** caches, speculation, multi-core, multi-thread, …

Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:

Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:

Caches

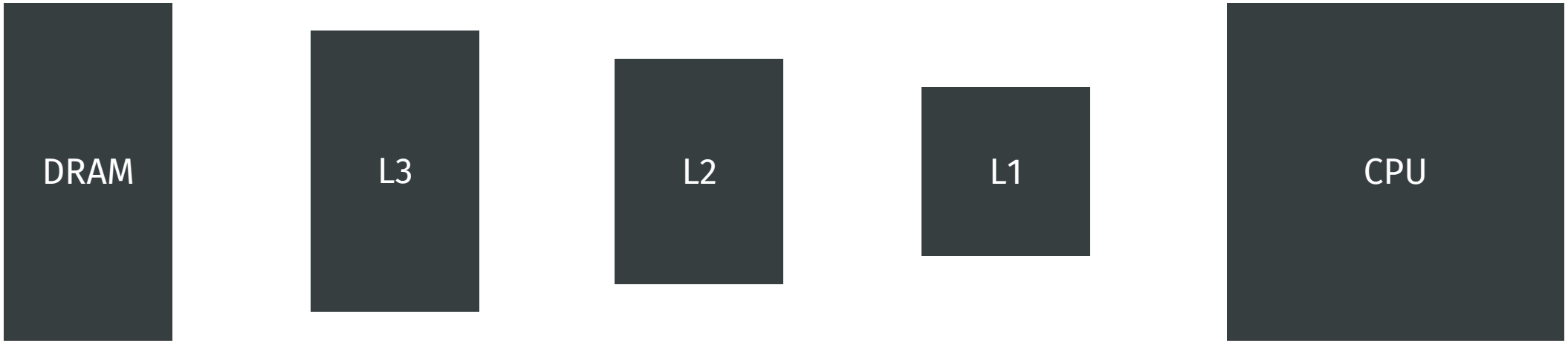
- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:

Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:

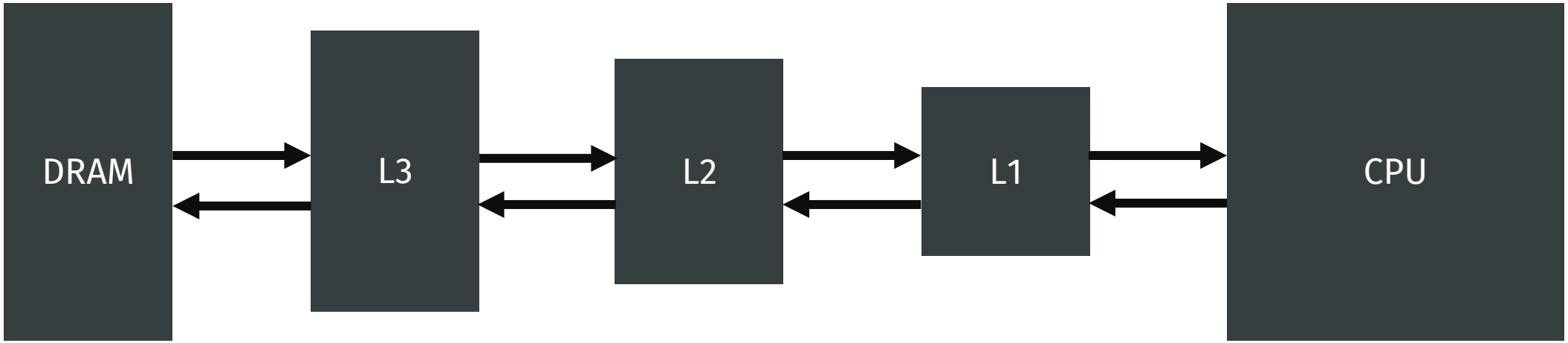
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



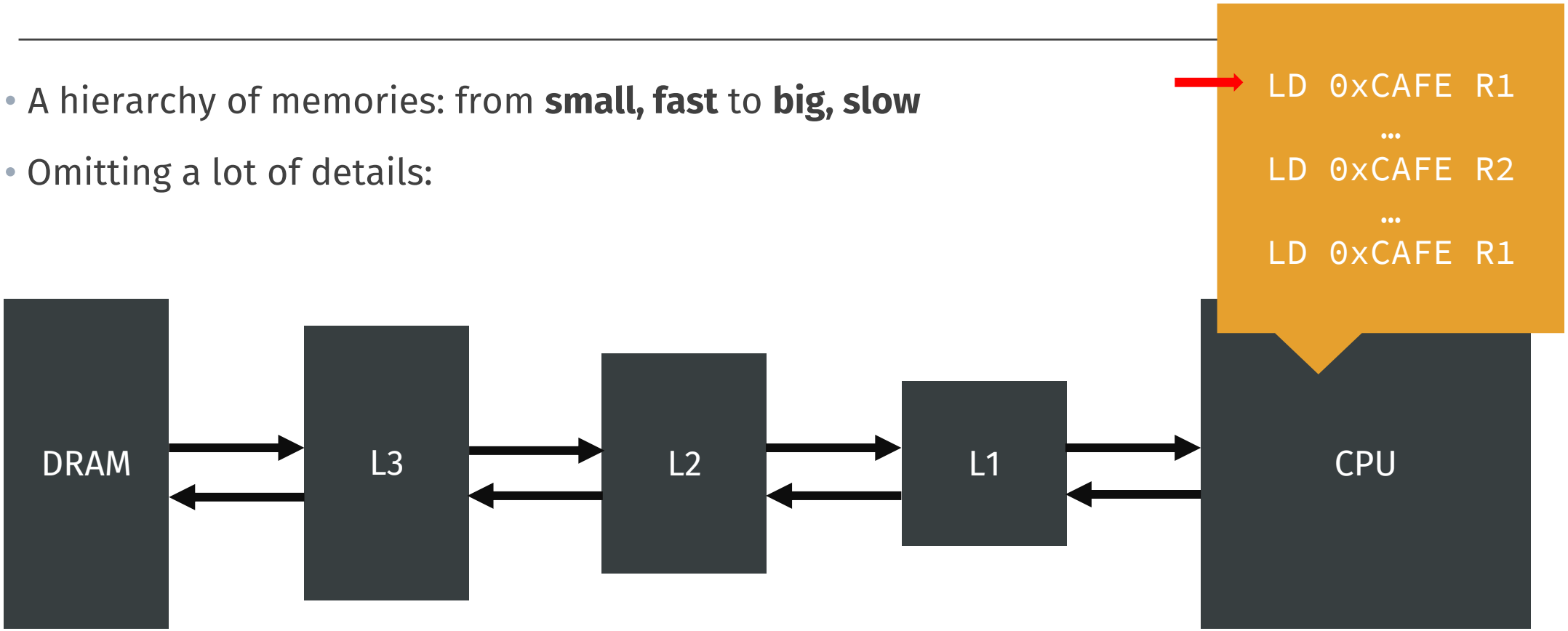
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



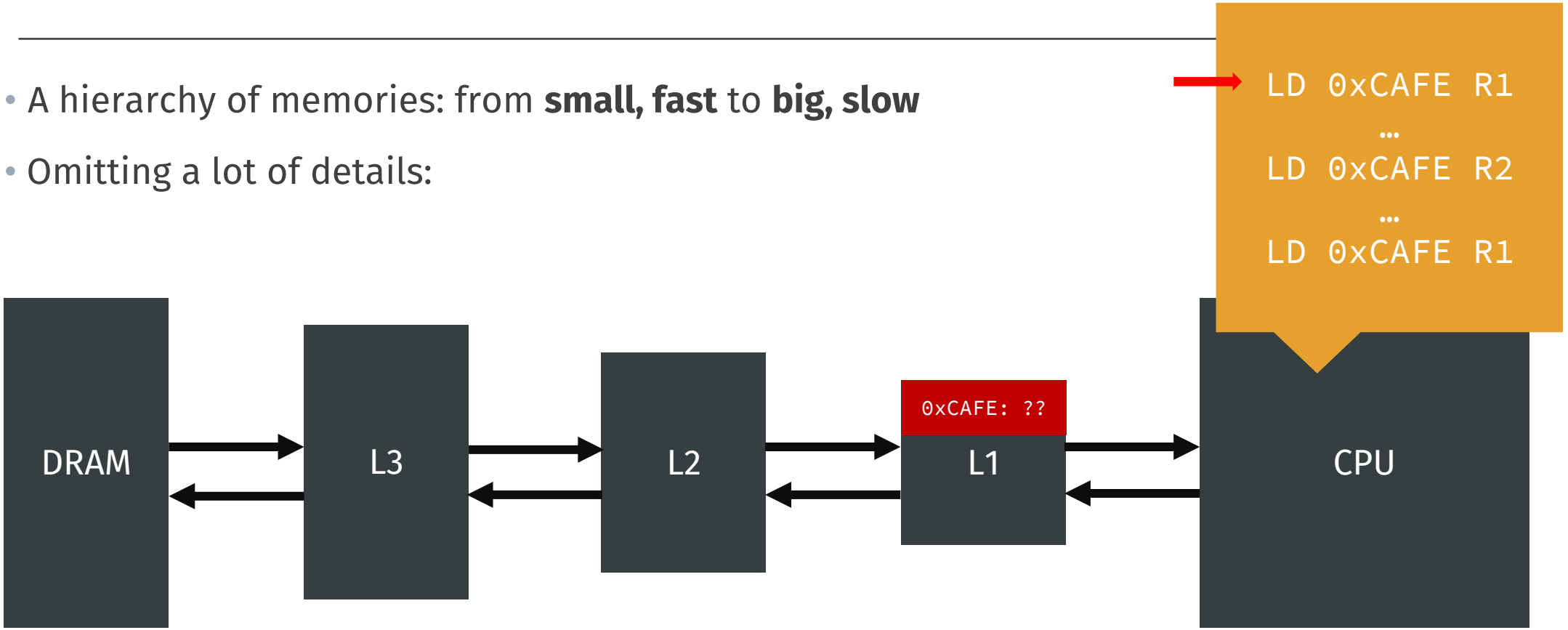
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



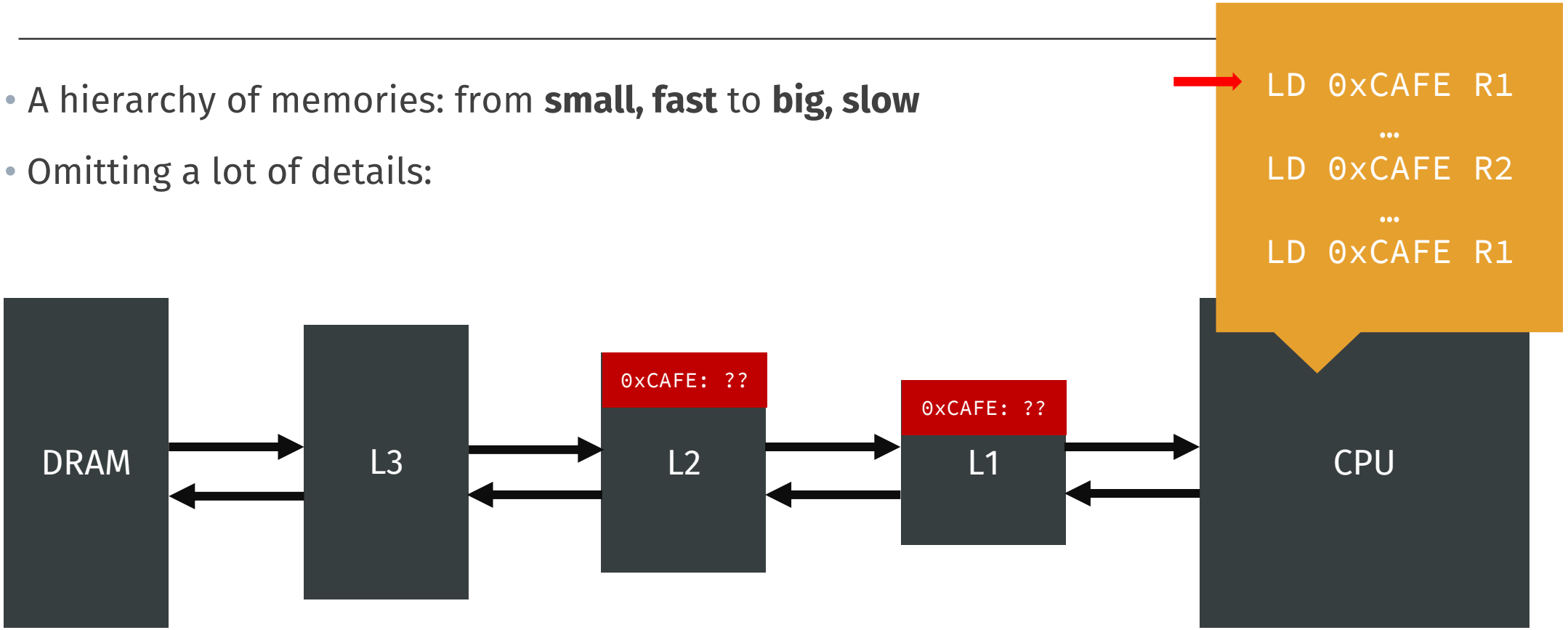
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



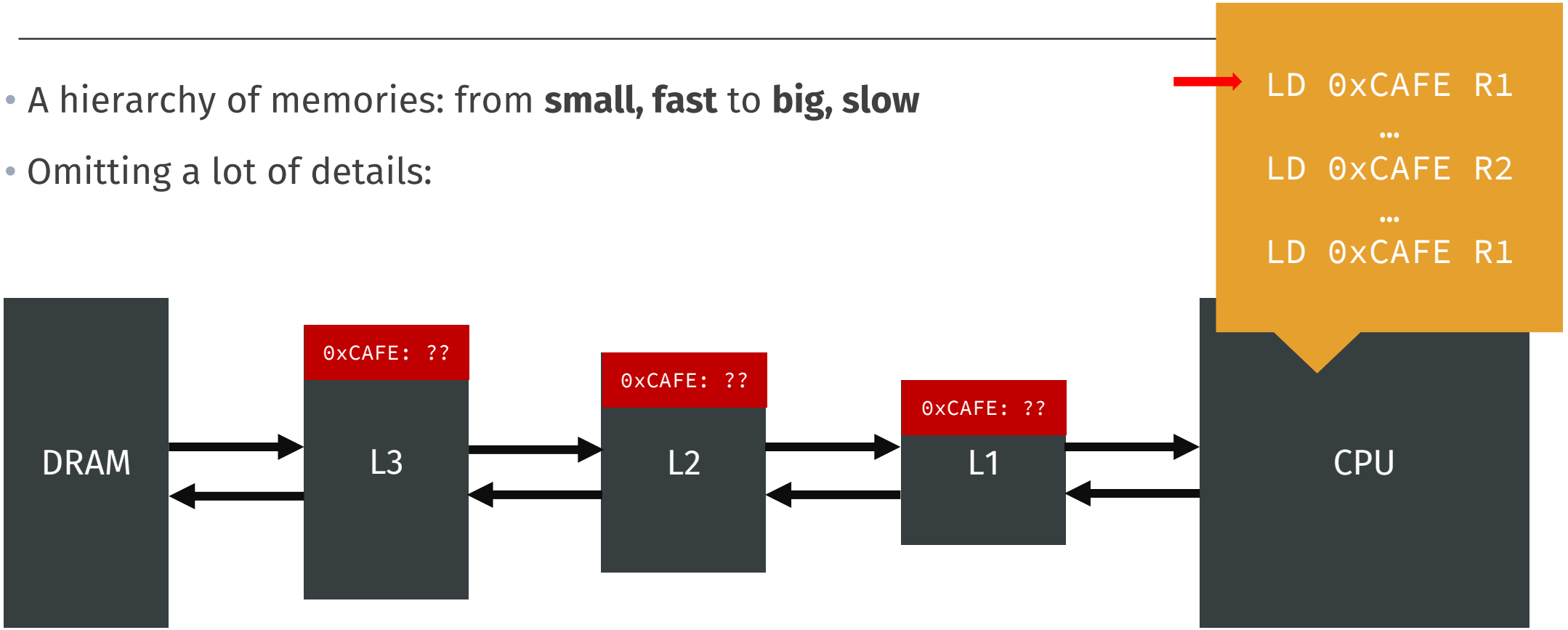
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



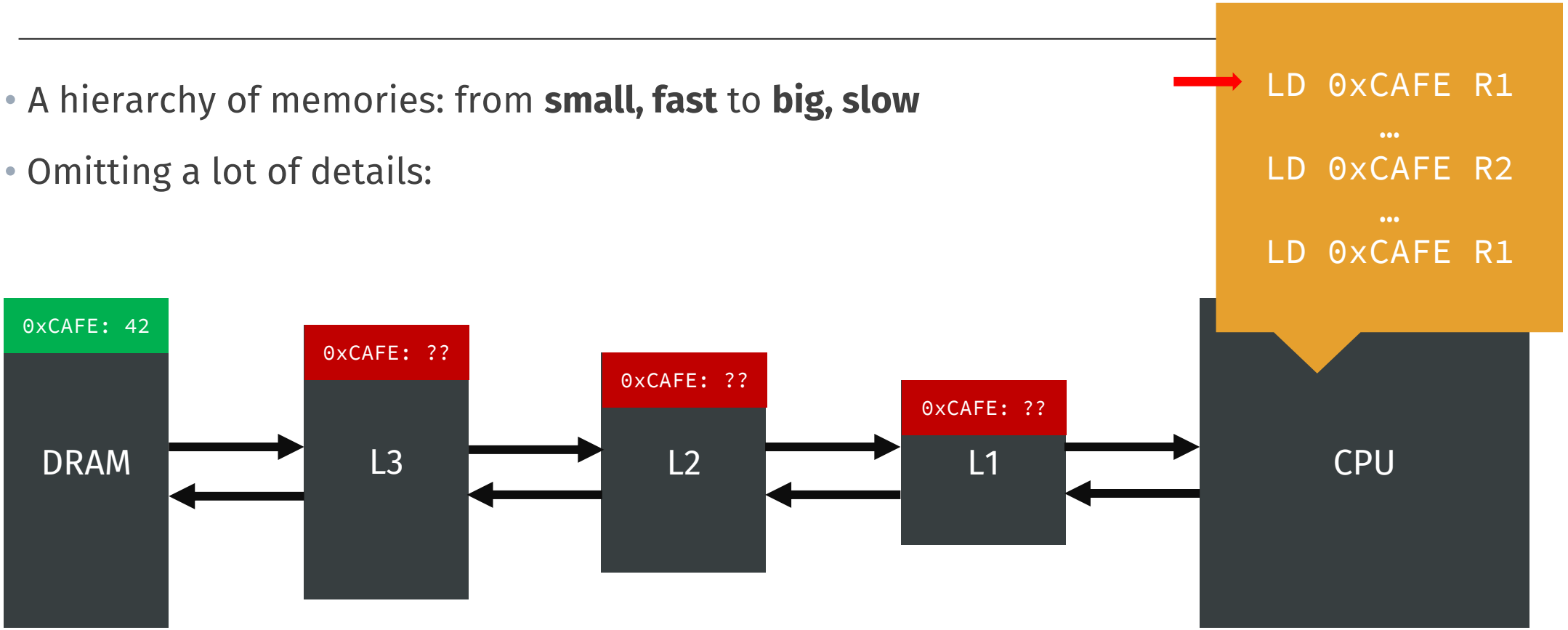
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



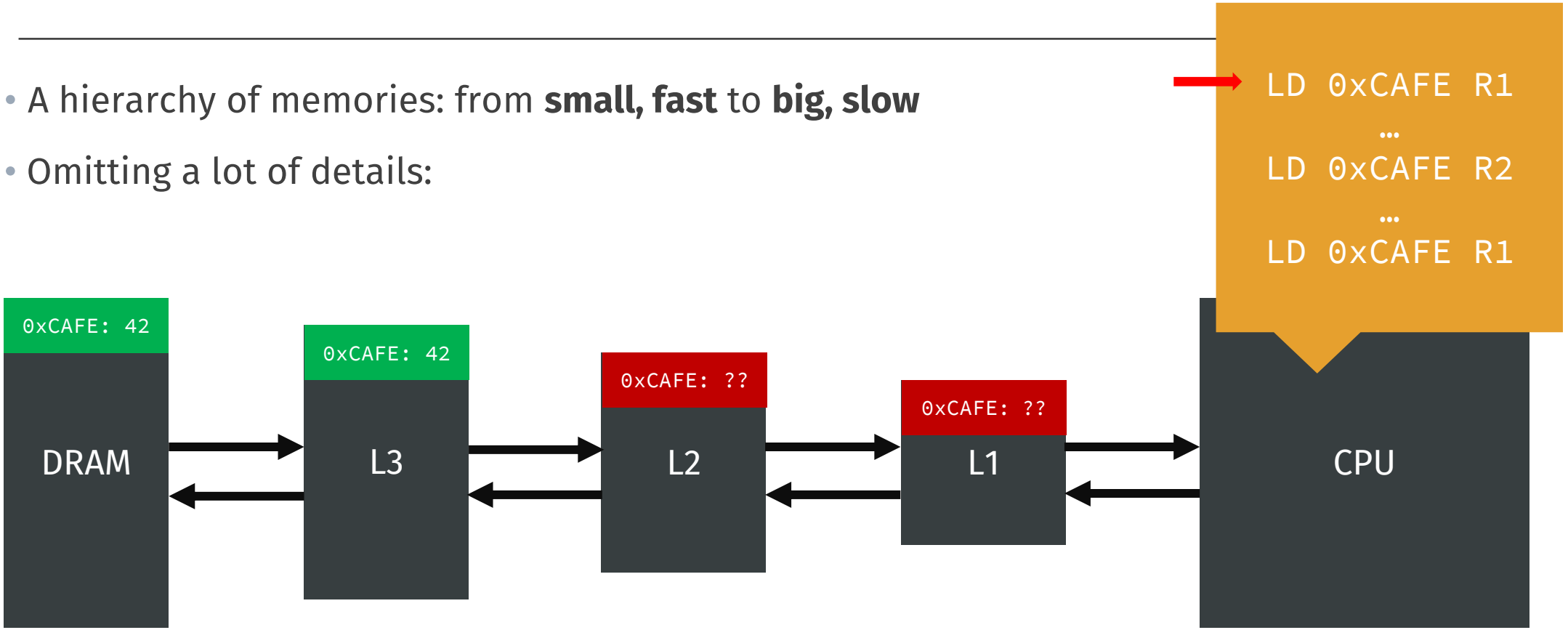
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



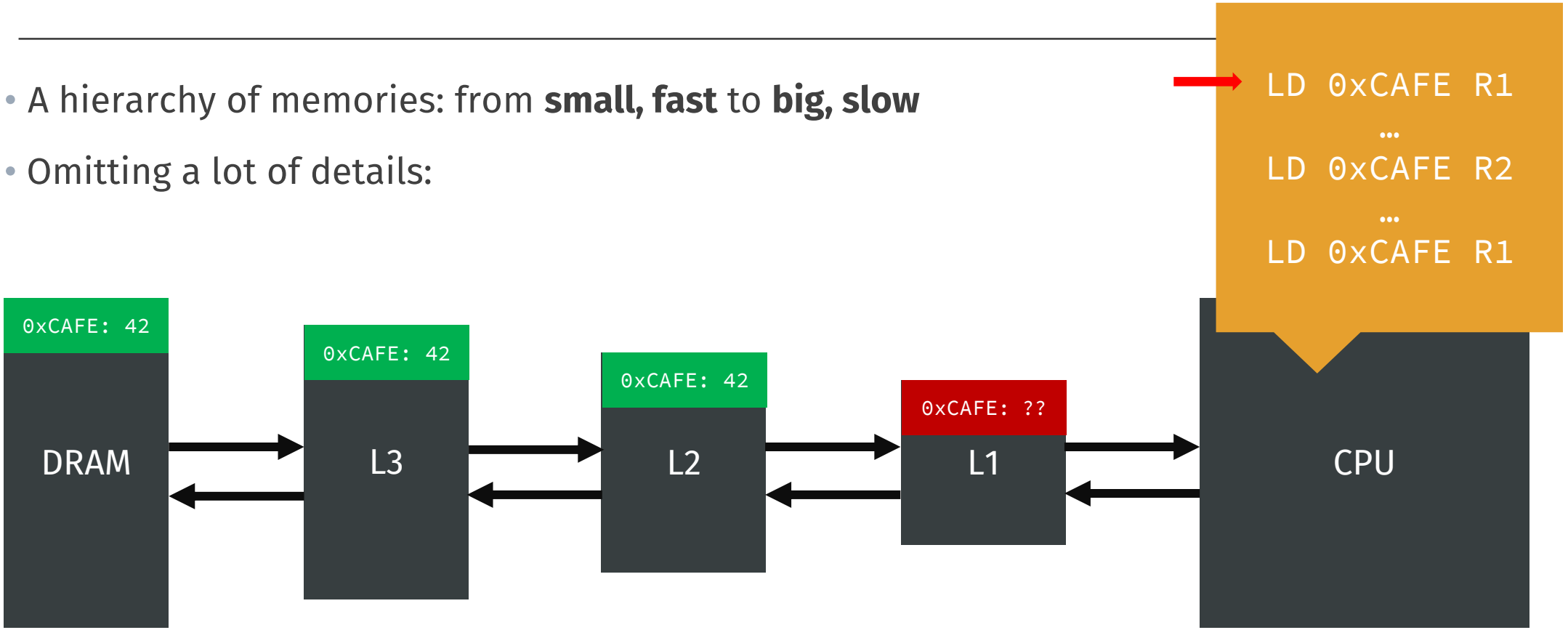
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



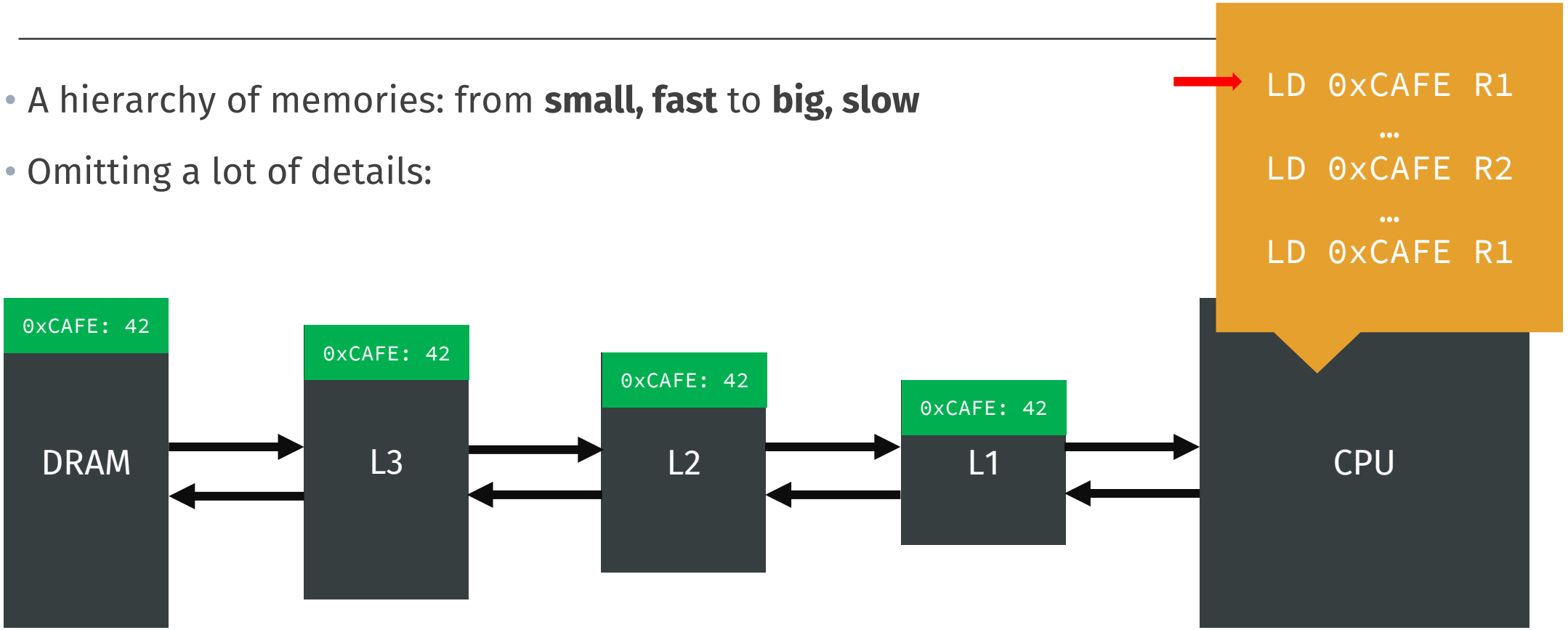
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



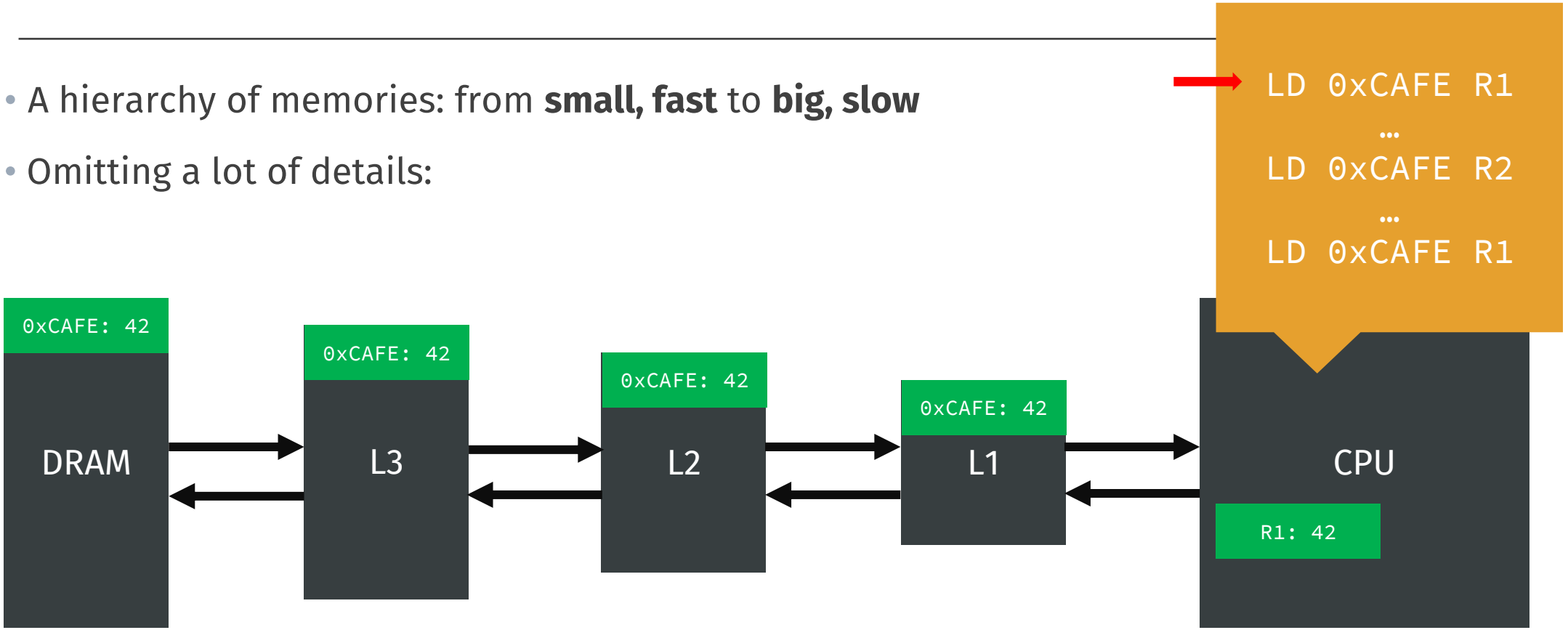
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



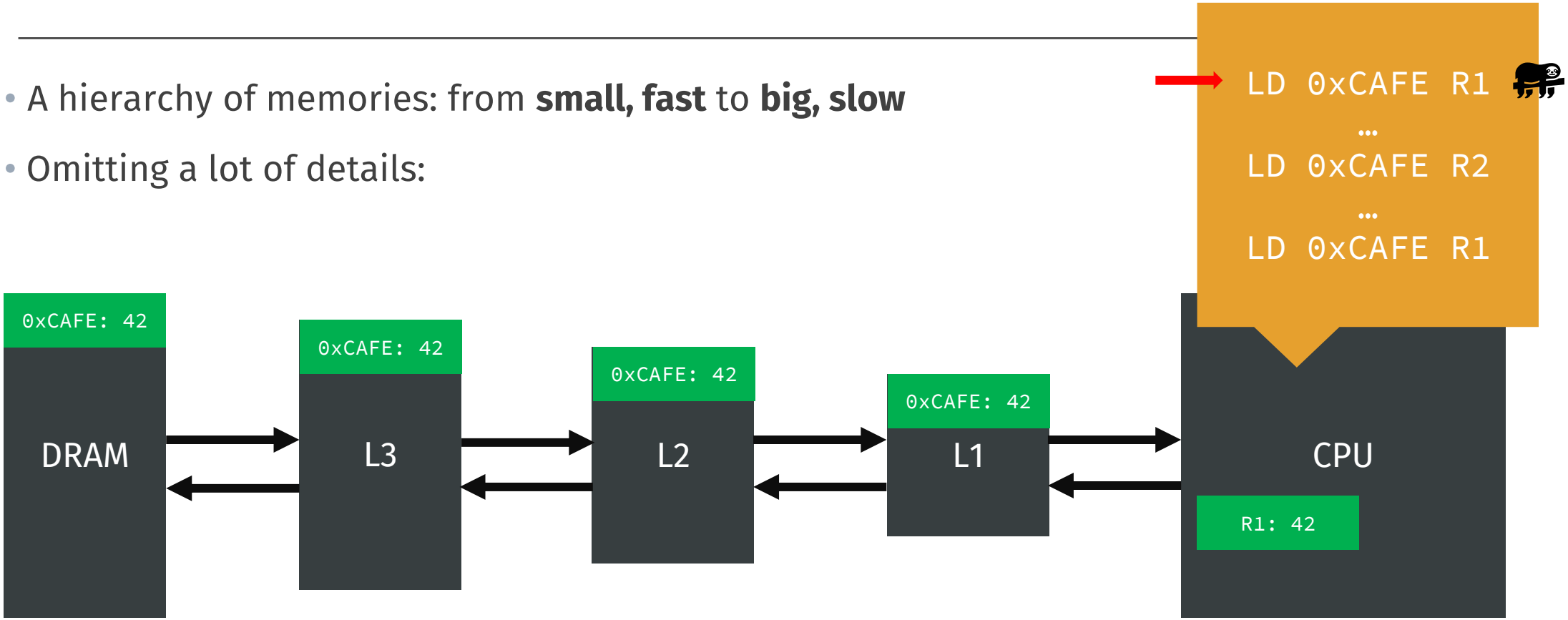
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



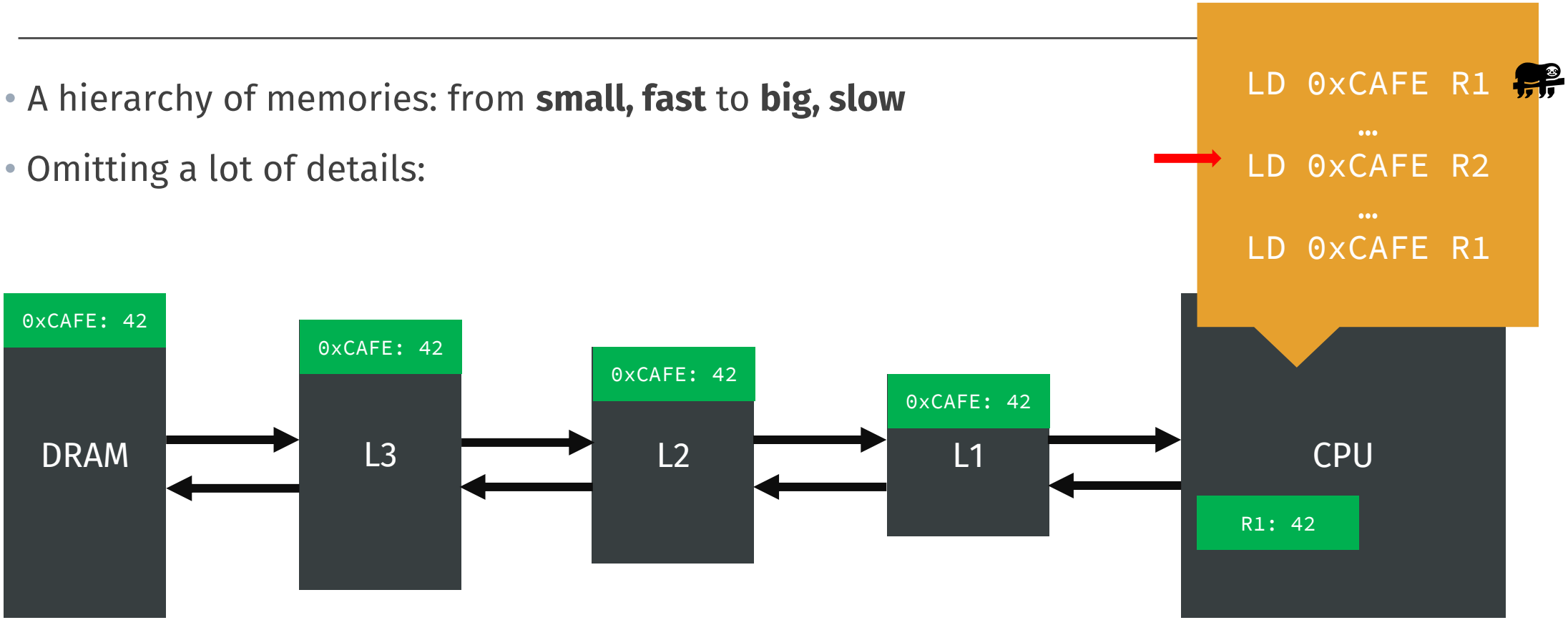
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



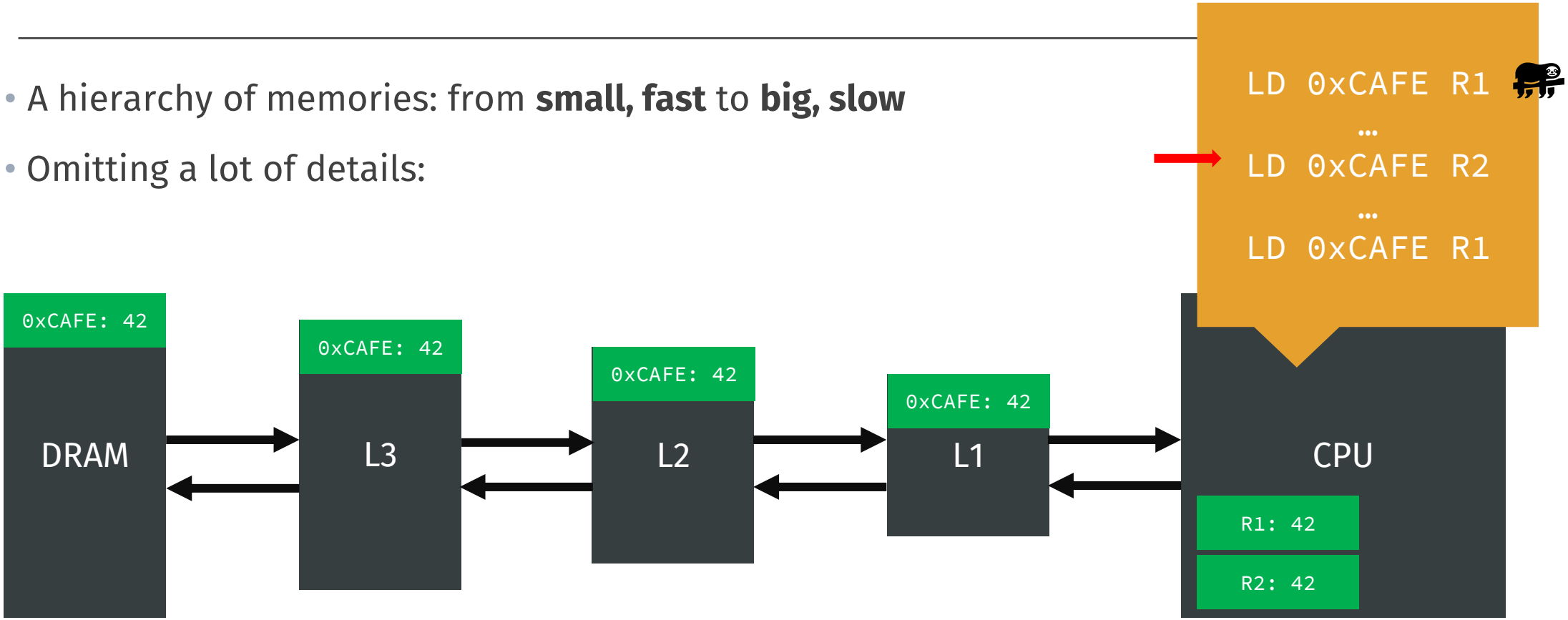
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



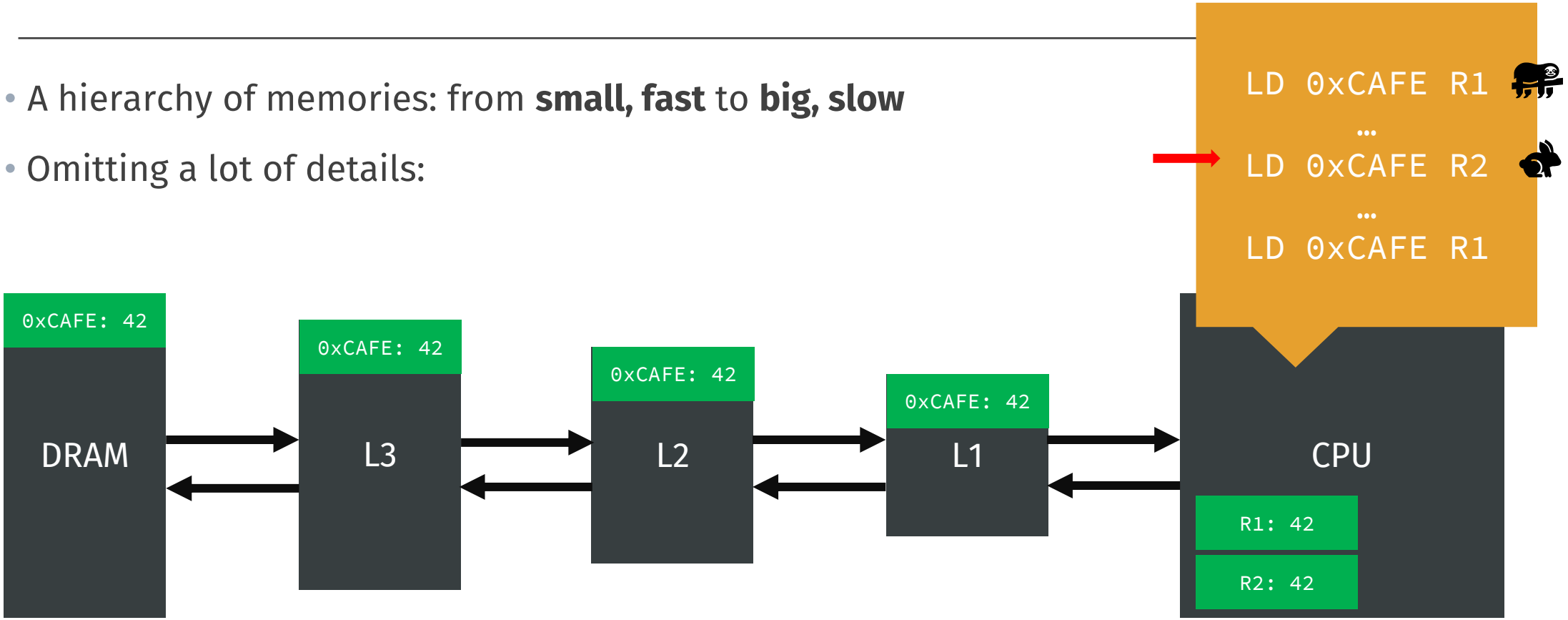
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



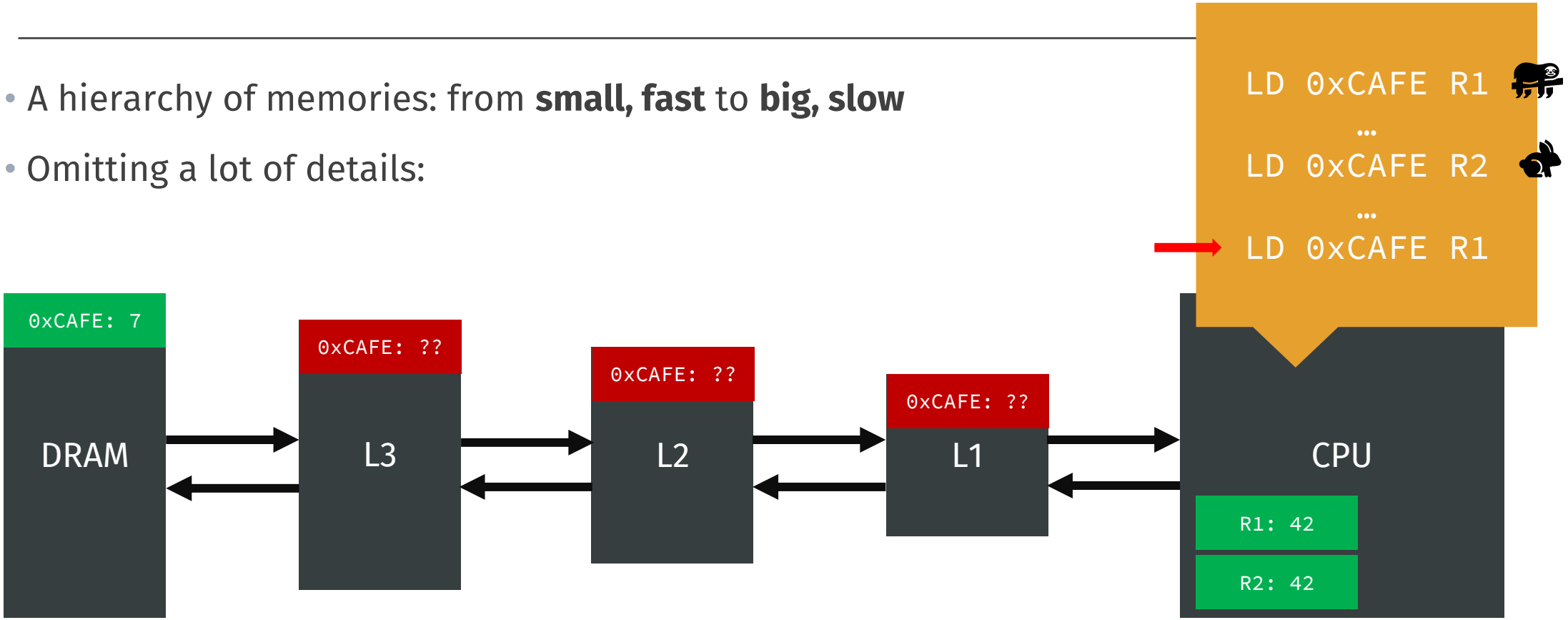
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



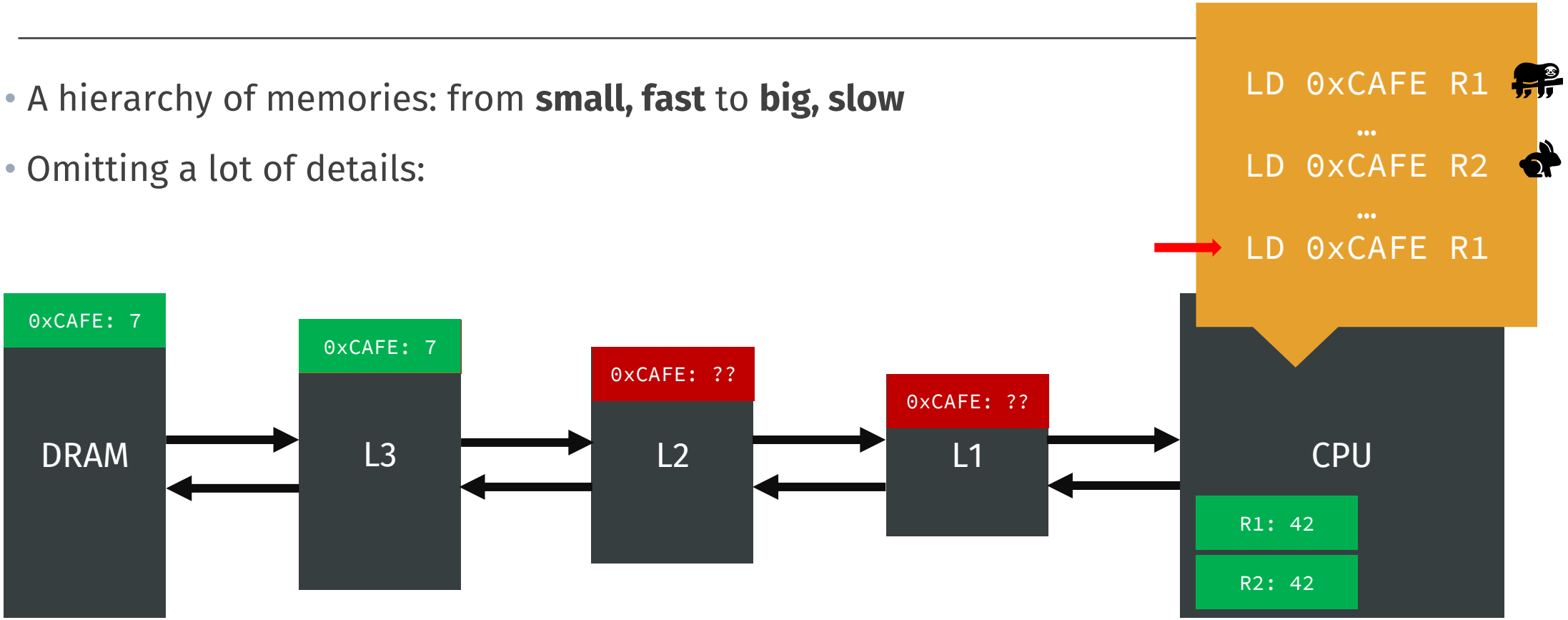
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



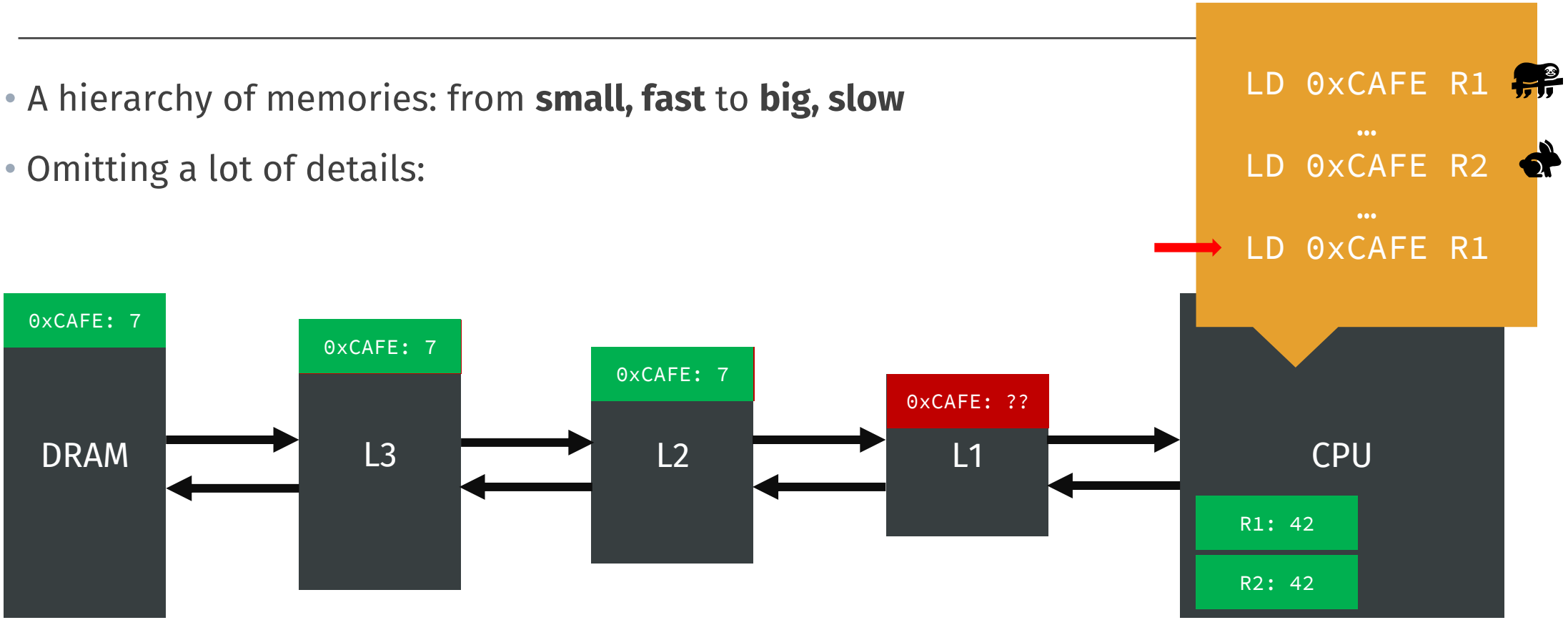
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



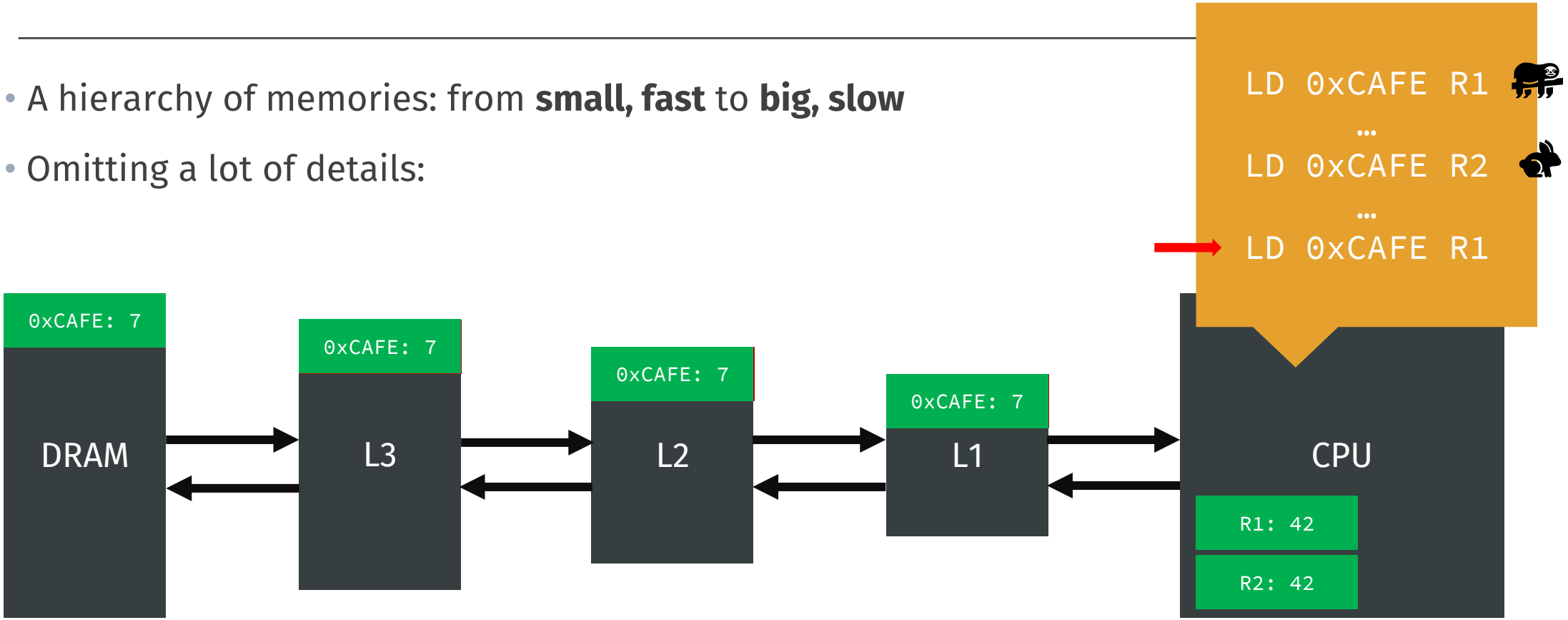
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



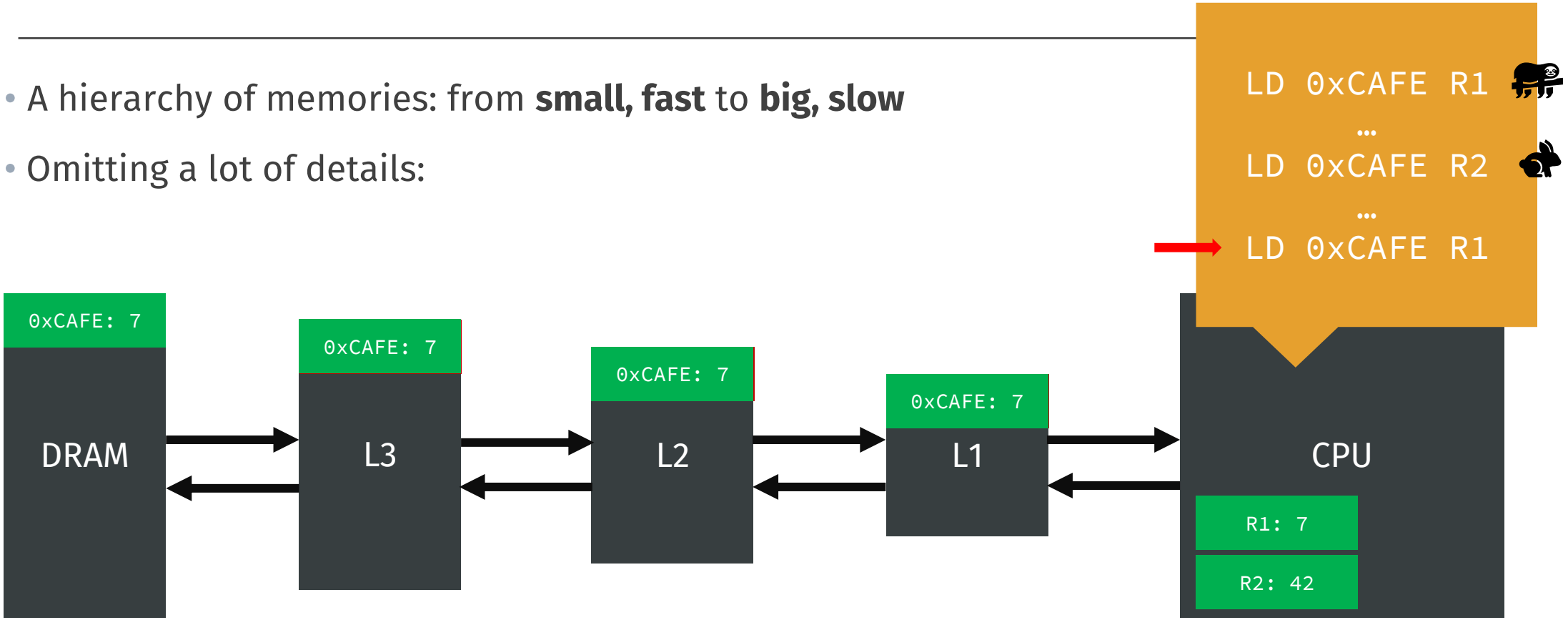
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



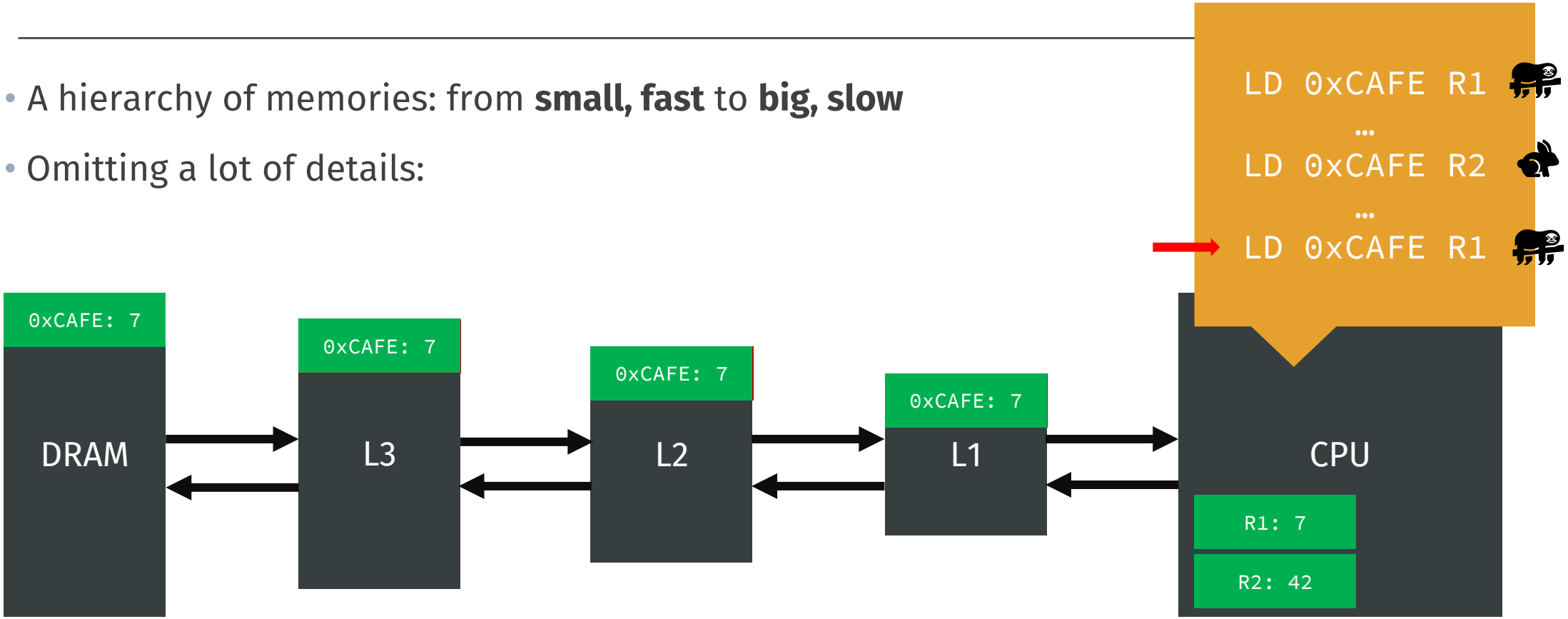
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Omitting a lot of details:



Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢

Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory, the CPU speculates on what's **probably right** to do and does that
  - When the value arrives, if it was the wrong thing: **rollback**

Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory, the CPU speculates on what's **probably right** to do and does that
  - When the value arrives, if it was the wrong thing: **rollback**
- For that, the CPU is equipped with a **branch predictor**

Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory, the CPU speculates on what's **probably right** to do and does that
  - When the value arrives, if it was the wrong thing: **rollback**
- For that, the CPU is equipped with a **branch predictor**
  - **Roughly:** should the CPU take the next branch?

Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory, the CPU speculates on what's **probably right** to do and does that
  - When the value arrives, if it was the wrong thing: **rollback**
- For that, the CPU is equipped with a **branch predictor**
  - **Roughly:** should the CPU take the next branch?
  - The CPU **trains** its branch predictor by **observing** what happened before

Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory, the CPU speculates on what's **probably right** to do and does that
  - When the value arrives, if it was the wrong thing: **rollback**
- For that, the CPU is equipped with a **branch predictor**
  - **Roughly:** should the CPU take the next branch?
  - The CPU **trains** its branch predictor by **observing** what happened before
  - Modern branch predictors are complex
    - but **for our purposes just one bit telling if we should take the branch**!

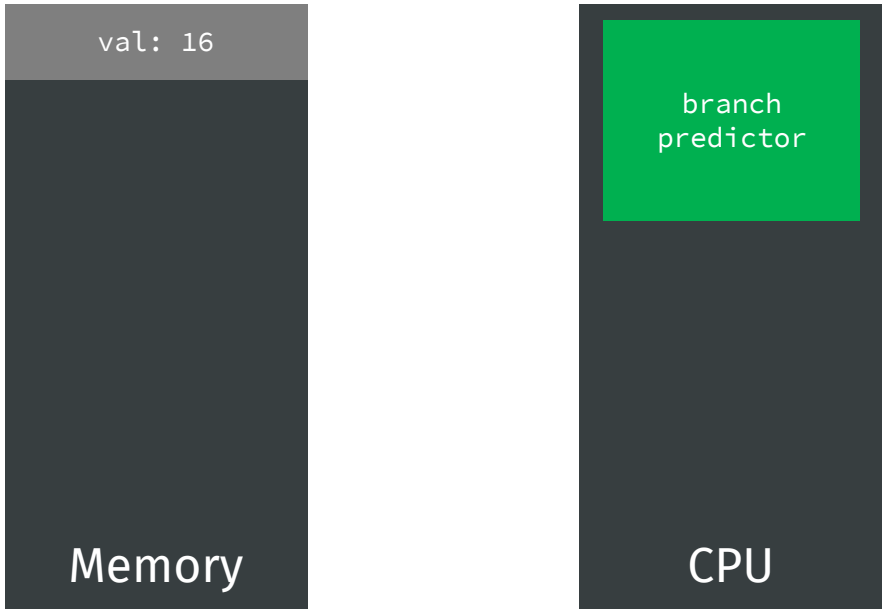
Speculation: when the CPU is right

Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```

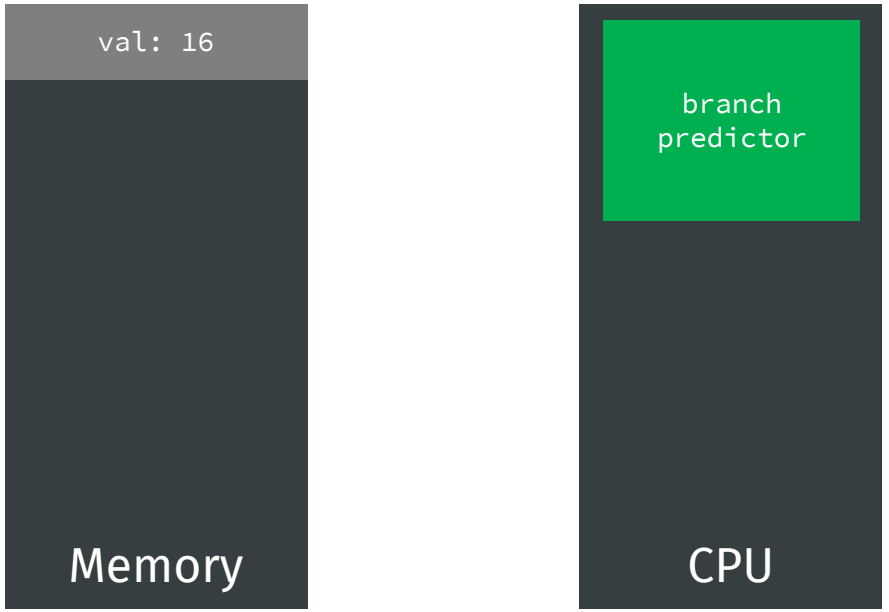
Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



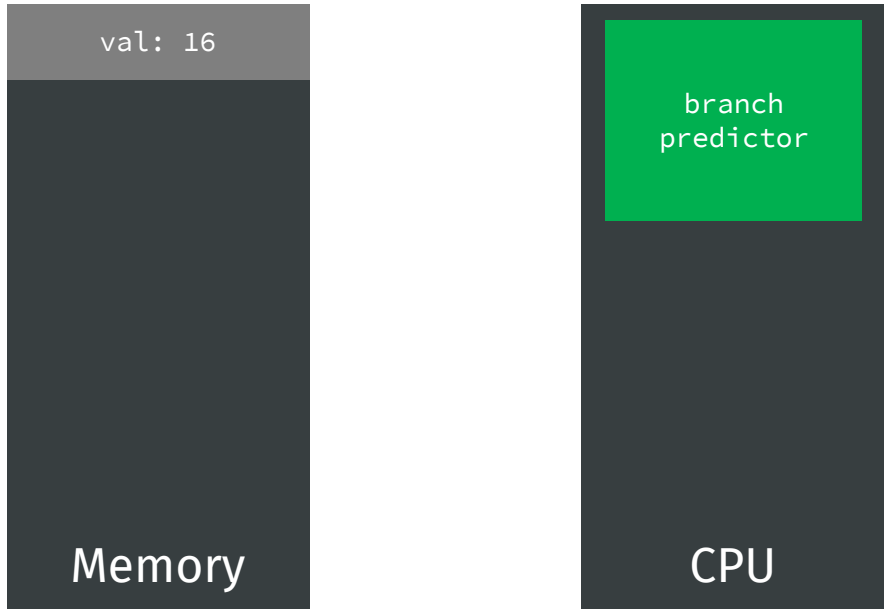
Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



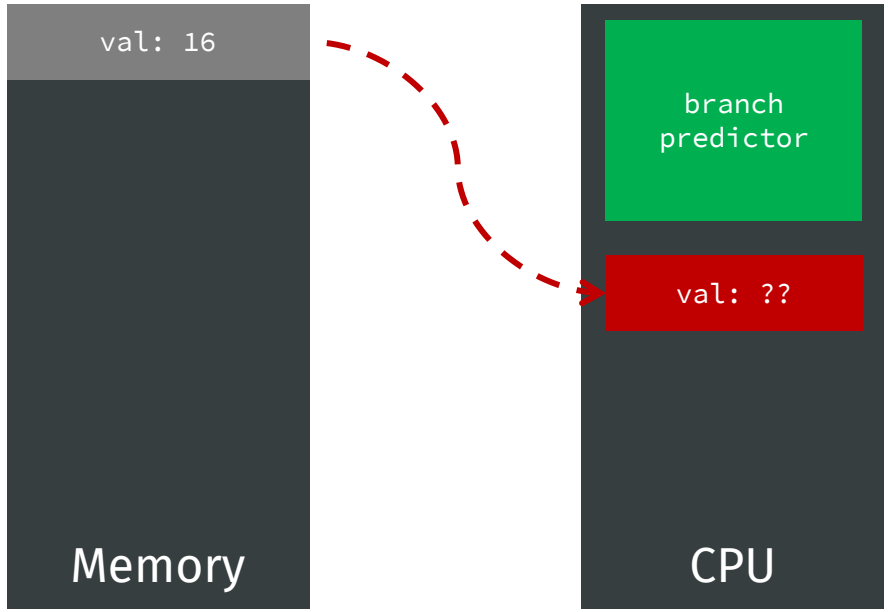
Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



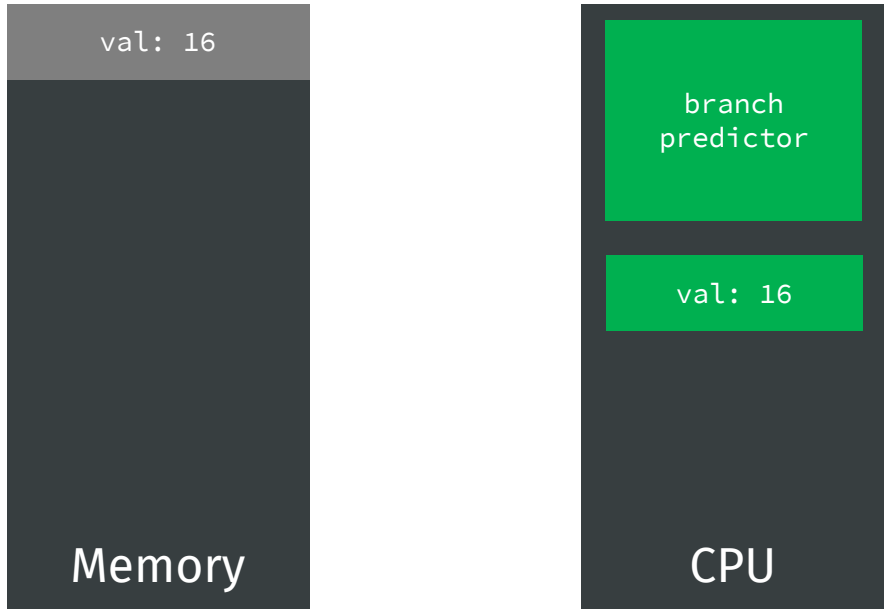
Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



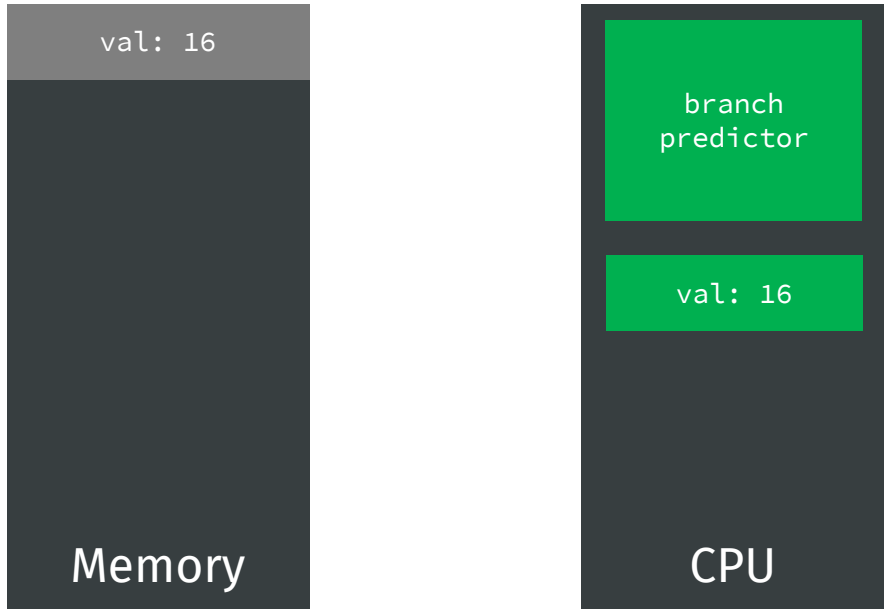
Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



Speculation: when the CPU is right

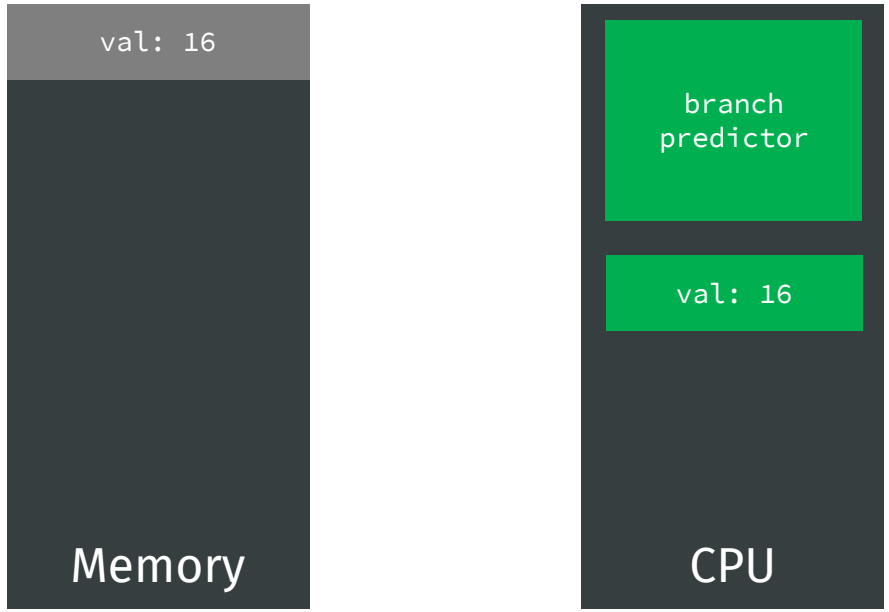
```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



Speculation: when the CPU is right

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```

The CPU was right: commit the changes!

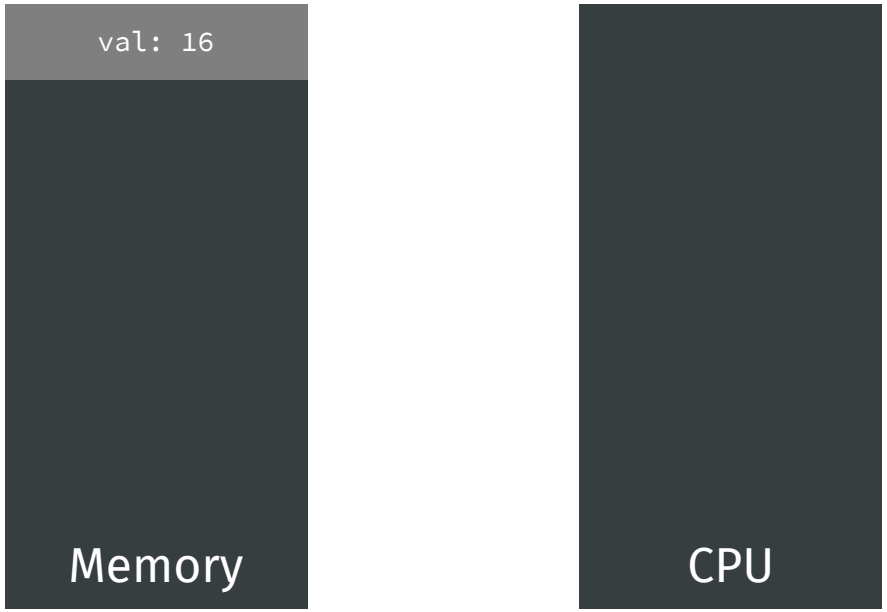


Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```

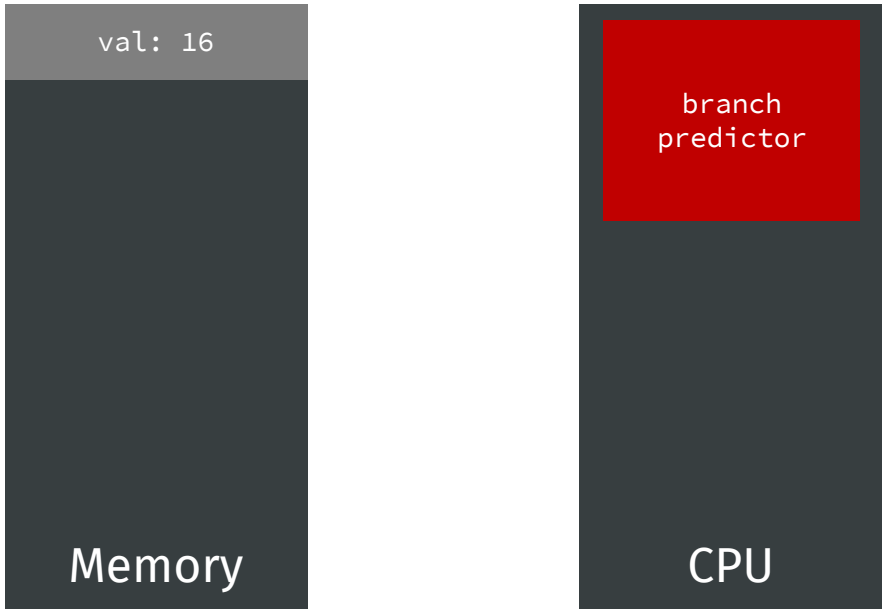
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



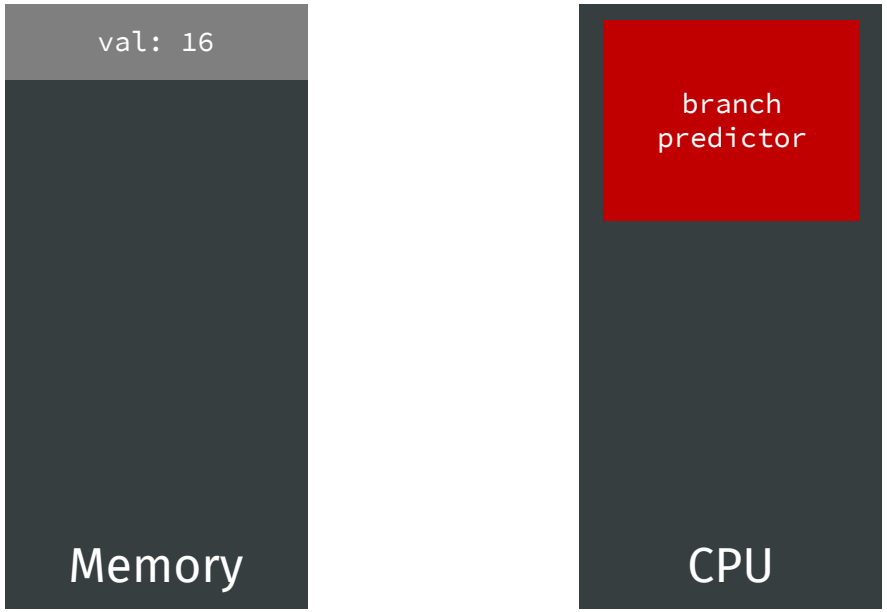
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



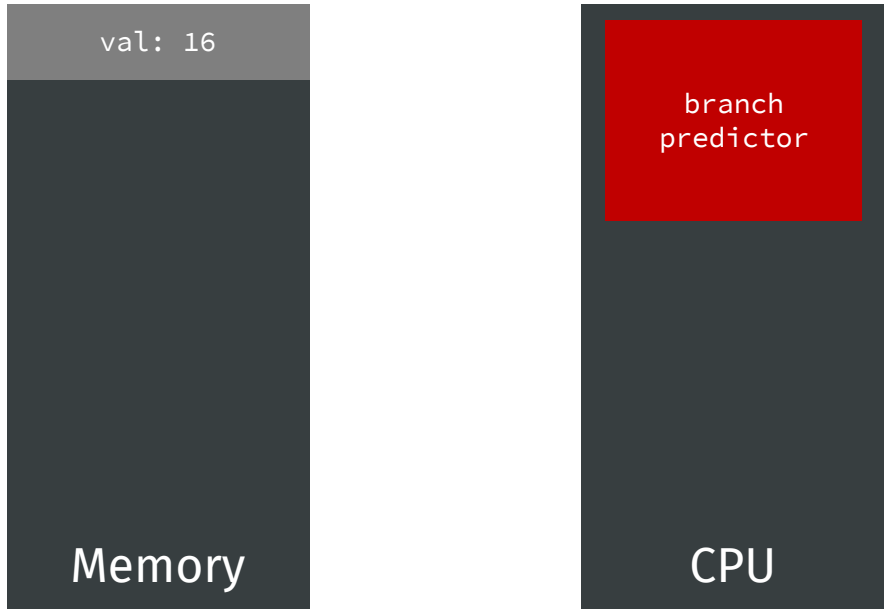
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



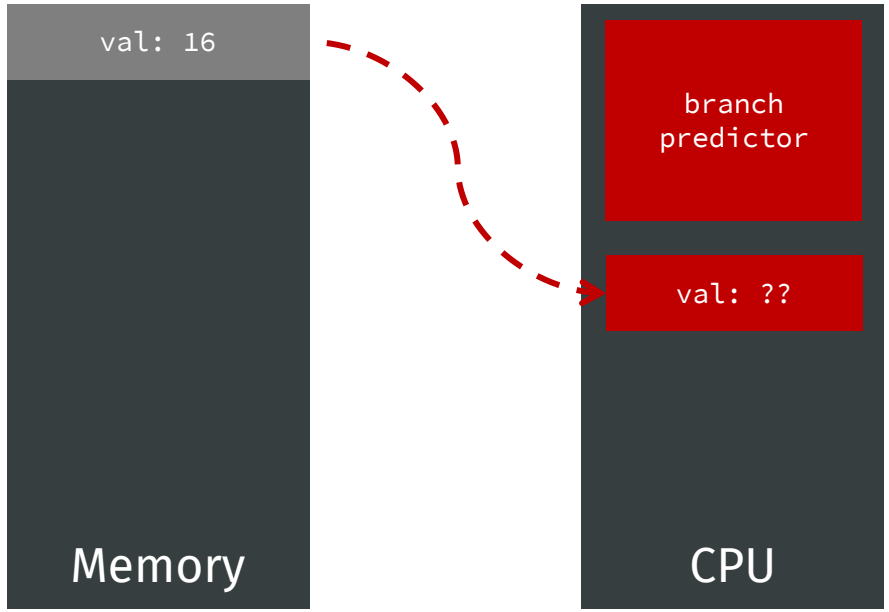
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



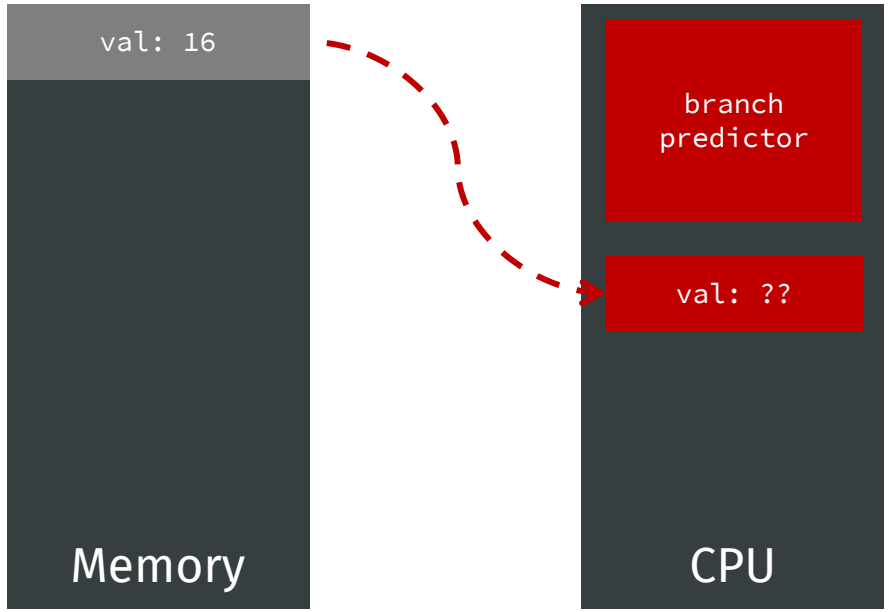
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



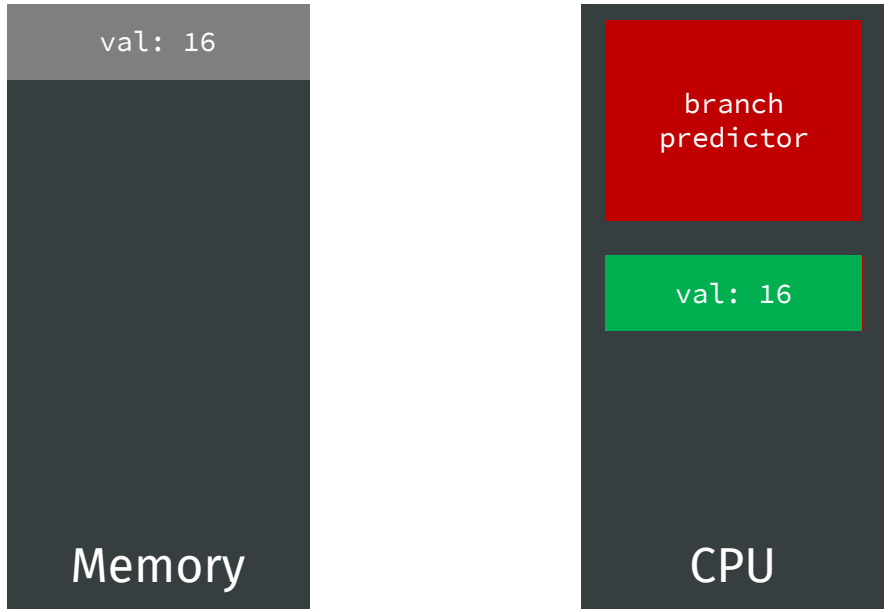
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



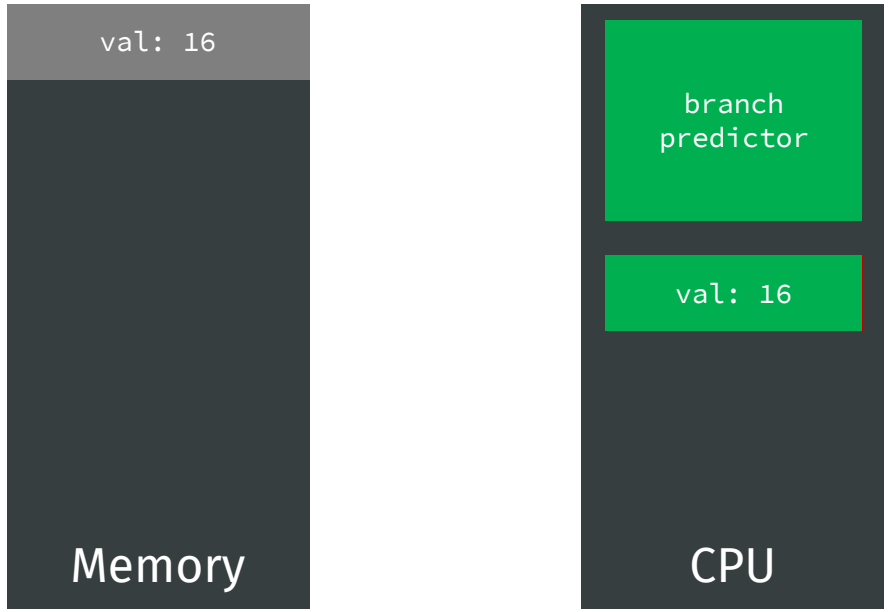
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



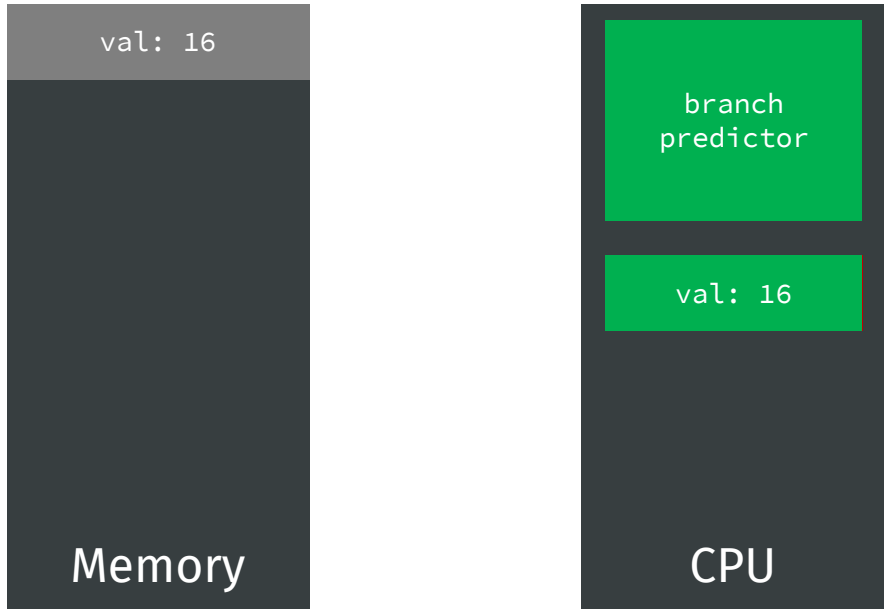
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



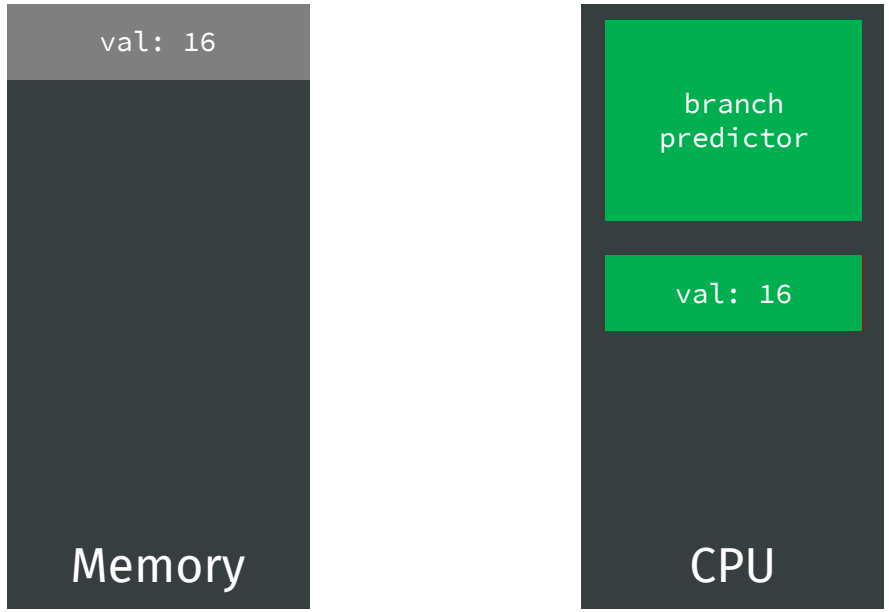
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();  rollback!
```



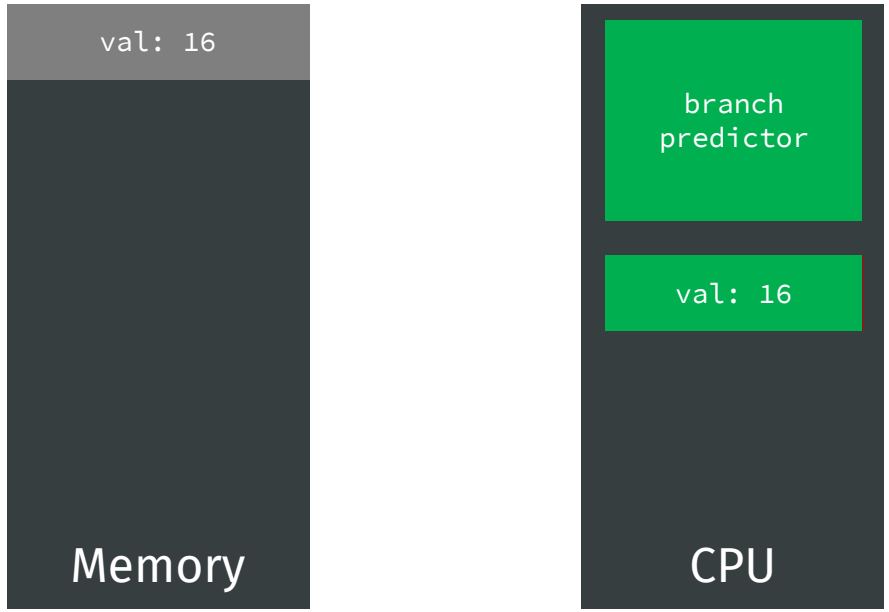
Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



Side channels...

Designers are trying to optimize the avg case!

- In the **worst** case the execution will be **slow**
- In the **best** case the execution will be **very fast**

[Lawson, 2009] Lawson, Nate. "Side-channel attacks on cryptographic software." *IEEE Security & Privacy* 7, no. 6 (2009): 65-68.

Side channels...

Designers are trying to optimize the avg case!

- In the **worst** case the execution will be **slow**
- In the **best** case the execution will be **very fast**

Consider an attacker that just **observes** the *execution* of a program

- If it can learn some information from its observation, then there is a side channel!
- **Informally:** programs that resist against attackers able to measure time are said **constant-time programs**
  - **Actually:** there are better (i.e., implementation-independent) definitions

[Lawson, 2009] Lawson, Nate. "Side-channel attacks on cryptographic software." *IEEE Security & Privacy* 7, no. 6 (2009): 65-68.

Side channels...

Designers are trying to optimize the avg case!

- In the **worst** case the execution will be **slow**
- In the **best** case the execution will be **very fast**

Consider an attacker that just **observes** the *execution* of a program

- If it can learn some information from its observation, then there is a side channel!
- **Informally:** programs that resist against attackers able to measure time are said **constant-time programs**
  - **Actually:** there are better (i.e., implementation-independent) definitions

More precisely:

"A **side channel** is any observable side effect of computation that an attacker could measure and possibly influence." [Lawson, 2009]

[Lawson, 2009] Lawson, Nate. "Side-channel attacks on cryptographic software." *IEEE Security & Privacy* 7, no. 6 (2009): 65-68.

Your first side-channel attack

FLUSH+RELOAD [Yarom&Falkner, 2014]

[Yarom&Falkner, 2014] Y. Yarom, K. Falkner. "FLUSH+RELOAD}: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *USENIX Security* 2014.

FLUSH+RELOAD [Yarom&Falkner, 2014]

- **Flush+Reload** exploits **caches** to **leak secrets**
  - Very impacting and difficult to fix
  - Other similar attacks: **Prime+Probe, Flush+Flush**

[Yarom&Falkner, 2014] Y. Yarom, K. Falkner. "FLUSH+RELOAD}: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *USENIX Security* 2014.

FLUSH+RELOAD [Yarom&Falkner, 2014]

- **Flush+Reload** exploits **caches** to **leak secrets**
  - Very impacting and difficult to fix
  - Other similar attacks: **Prime+Probe, Flush+Flush**
- Relatively easy to carry out:
  - Take an attacker that can measure the execution time of a program, choose a victim
  - Find a line from the cache shared between the attacker and the victim and **flush** it
  - Run the victim
  - Measure the time it takes to perform a memory read at the address corresponding to the evicted cache line (**Reload**)
    - If the victim accessed the shared line the access will be fast (data was cached!)

[Yarom&Falkner, 2014] Y. Yarom, K. Falkner. "FLUSH+RELOAD}: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *USENIX Security* 2014.

FLUSH+RELOAD [Yarom&Falkner, 2014]

- **Flush+Reload** exploits **caches** to **leak secrets**
  - Very impacting and difficult to fix
  - Other similar attacks: **Prime+Probe, Flush+Flush**
- Relatively easy to carry out:
  - Take an attacker that can measure the execution time of a program, choose a victim
  - Find a line from the cache shared between the attacker and the victim and **flush** it
  - Run the victim
  - Measure the time it takes to perform a memory read at the address corresponding to the evicted cache line (**Reload**)
    - If the victim accessed the shared line the access will be fast (data was cached!)
    - Otherwise, the read will be slow

[Yarom&Falkner, 2014] Y. Yarom, K. Falkner. "FLUSH+RELOAD}: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *USENIX Security* 2014.

FLUSH+RELOAD

FLUSH+RELOAD

1. “Flush” phase:

FLUSH+RELOAD

1. “Flush” phase:
    1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim

FLUSH+RELOAD

1. “Flush” phase:
    1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim

```
N = // SECRET
…
y = A[N * 512];
```

FLUSH+RELOAD

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)

```
N = // SECRET
…
y = A[N * 512];
```

FLUSH+RELOAD

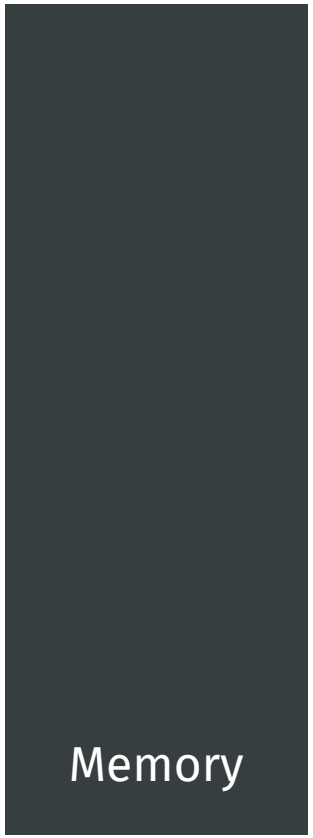
1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache

```
N = // SECRET
…
y = A[N * 512];
```

FLUSH+RELOAD

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache

```
N = // SECRET
…
y = A[N * 512];
```

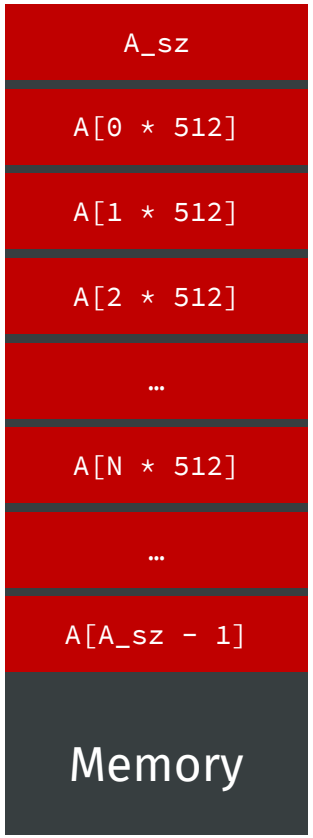


FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache



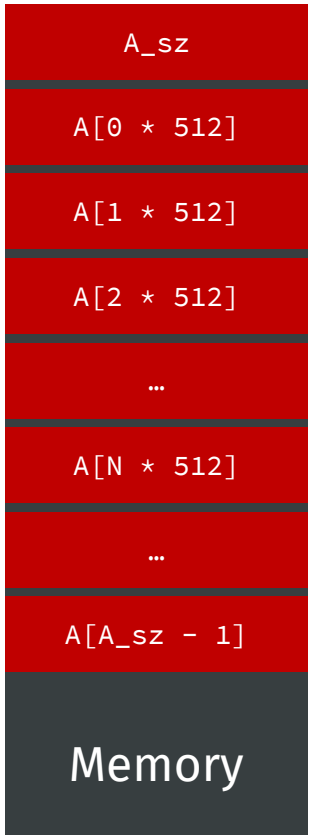

```
N = // SECRET
…
y = A[N * 512];
```

FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache
2. Run the victim – or the victim may already be running – and wait for gadget’s execution




```
N = // SECRET
…
y = A[N * 512];
```

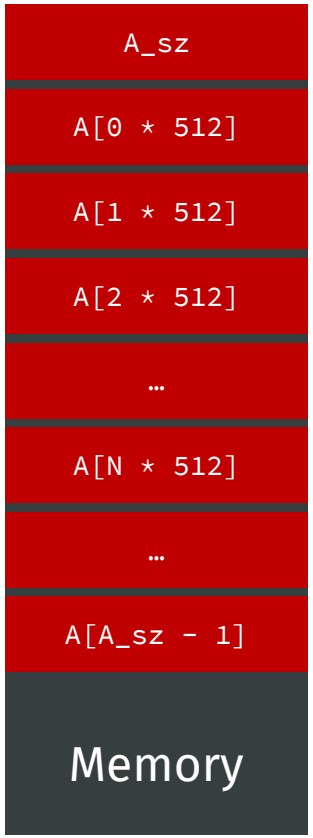
FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache
2. Run the victim – or the victim may already be running – and wait for gadget’s execution
3. “Reload” phase

```
N = // SECRET
…
y = A[N * 512];
```

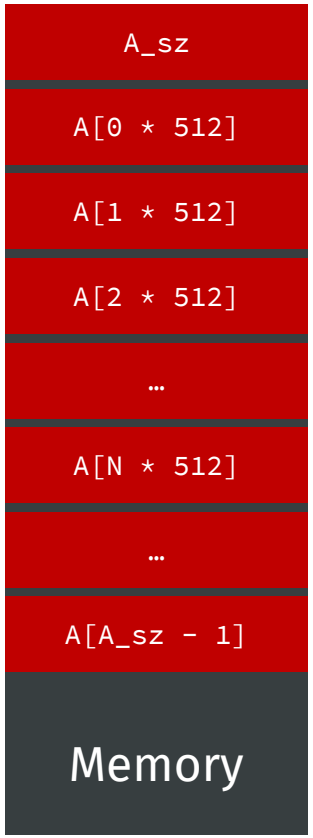


FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache
2. Run the victim – or the victim may already be running – and wait for gadget’s execution
3. “Reload” phase
   1. After gadget’s execution, `A[N * 512]` is loaded in the cache



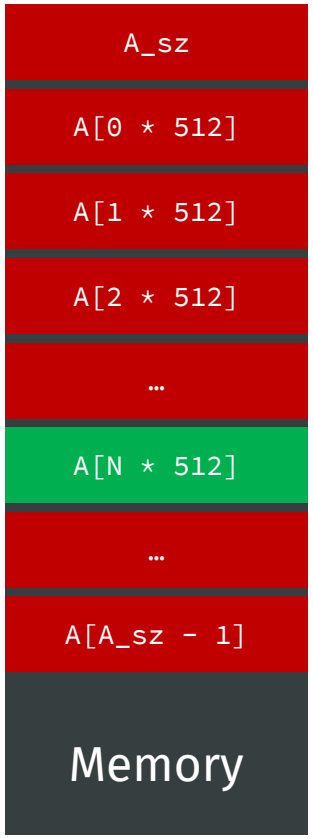

```
N = // SECRET
…
y = A[N * 512];
```

FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache
2. Run the victim – or the victim may already be running – and wait for gadget’s execution
3. “Reload” phase
   1. After gadget’s execution, `A[N * 512]` is loaded in the cache




```
N = // SECRET
…
y = A[N * 512];
```

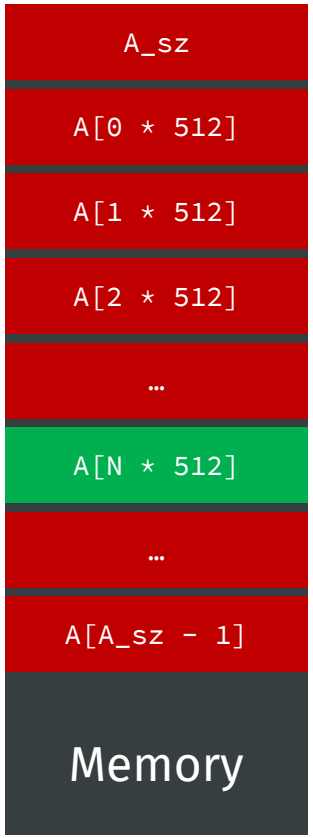
FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache
2. Run the victim – or the victim may already be running – and wait for gadget’s execution
3. “Reload” phase
   1. After gadget’s execution, `A[N * 512]` is loaded in the cache
   2. The attacker scans A, measuring access time for each `A[i * 512]`:

```
N = // SECRET
…
y = A[N * 512];
```



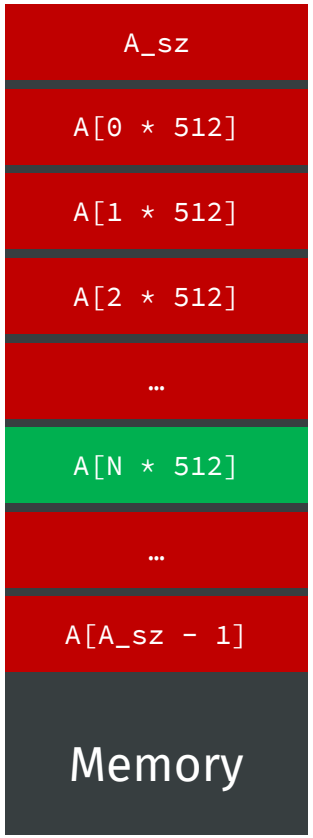
FLUSH+RELOAD

Not in cache

In cache

1. “Flush” phase:
   1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
   2. Make sure attacker and victim **share** memory pages (e.g., via page sharing/dedup)
   3. **Flush** A from the cache
2. Run the victim – or the victim may already be running – and wait for gadget’s execution
3. “Reload” phase
   1. After gadget’s execution, `A[N * 512]` is loaded in the cache
   2. The attacker scans A, measuring access time for each `A[i * 512]`:
      - for `i = N` the access will be fast, the **secret is leaked**!

```
N = // SECRET
…
y = A[N * 512];
```



Mitigating FLUSH+RELOAD

- Detecting **FLUSH+RELOAD**: Anomalous cache hit/misses patterns using HW performance counters
- Mitigating **FLUSH+RELOAD**:
  - **Software-based:** fix victims avoiding secret-dependent memory accesses; disable page sharing/de-duplication; limit access to high-resolution timers
  - **Hardware-based:** limit `clflush` access; make caches non-inclusive

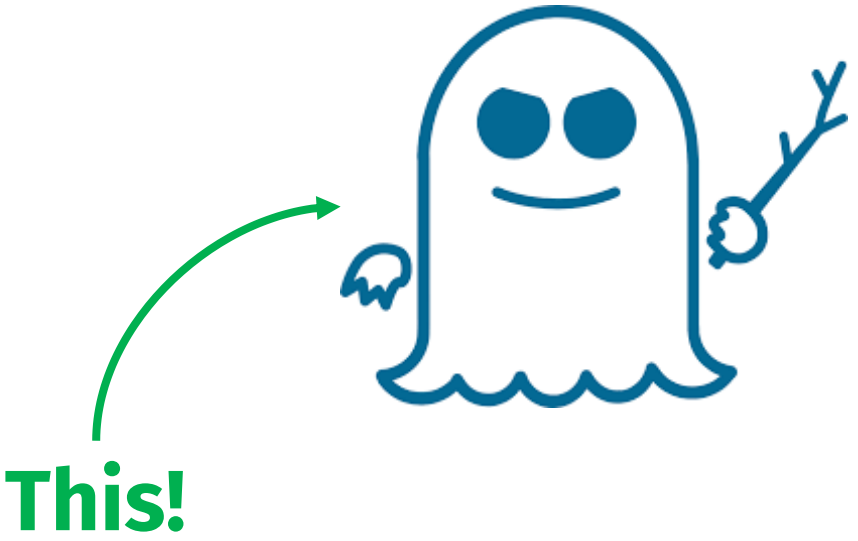
What about speculation?

What about speculation?

The **Spectre** attack!

What about speculation?

The **Spectre** attack!



What about speculation?

The **Spectre** attack!

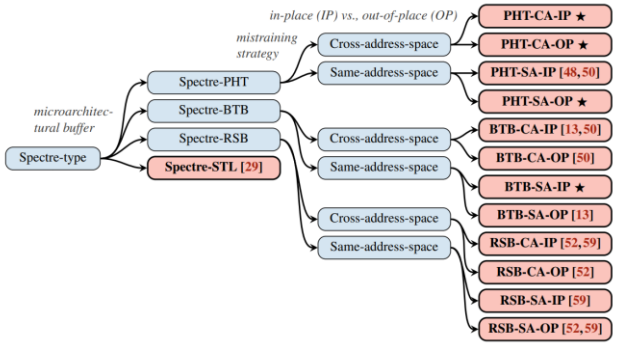
Not this!

This!

Spectre [Kocher et al., 2019]

Spectre is an attack exploiting **speculative execution** to **leak secrets**

- It is part of a recent wave of **micro-architectural attacks**, i.e., attacks using side-channels induced by the micro-architecture of a processor (e.g., cache, timers, virtual memory, …)
- "**Spectre** […] **transiently** bypasses software-defined security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths." [Canella et al., 2019] (← bit outdated, but a good survey on Spectre!)
- Many variants, here an updated version https://transient.fail/



[Kocher et al., 2019] P. Kocher et al. "Spectre attacks: Exploiting speculative execution." IEEE S&P 2019.

[Canella et al., 2019] C. Canella et al. "A systematic evaluation of transient execution attacks and defenses." *USENIX Security* 2019. URL: https://arxiv.org/abs/1811.05441v3

Spectre v1/v1.1 (special-cases of PHT)

We focus on Spectre v1 and v1.1

- Also called **Spectre-PHT**: the attacker **mistrains** the branch predictor by poisoning the **P**attern **H**istory **T**able
- How? **For example**
  - The attacker**:**
    1. Looks for a piece of “vulnerable code” (**code gadget**) including a condition (next slide)
    2. Runs that code multiple times with an input s.t. the condition always holds (so training the branch predictor to take the “true” branch)
    3. Then, runs the code with a specially-crafted input making the condition false: now the CPU mis-speculates and executes the “true” branch with wrong data (!!!)
    4. Finally, it looks for information left behind after the CPU discovered it mis-speculated and rolled-back the computation (e.g., contents of caches)

Spectre v1

Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim

Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim

```
if (x < A_sz)
  y = B[A[x] * 512];
```

Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor

```
if (x < A_sz)
  y = B[A[x] * 512];
```

Spectre v1

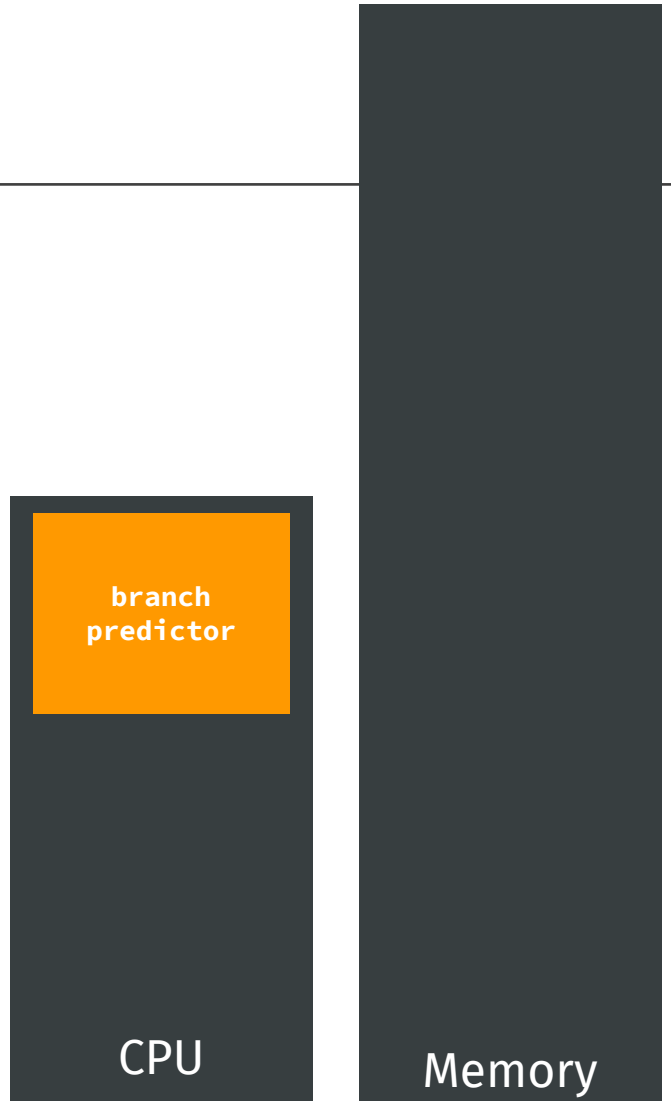
1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A

```
if (x < A_sz)
  y = B[A[x] * 512];
```

Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A

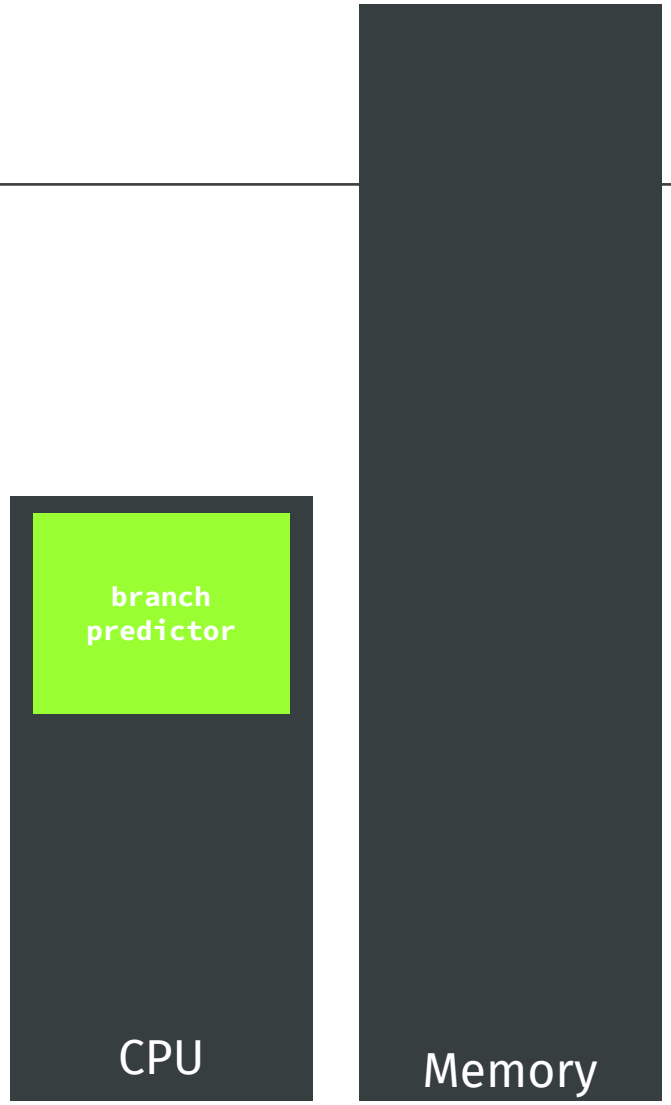
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A

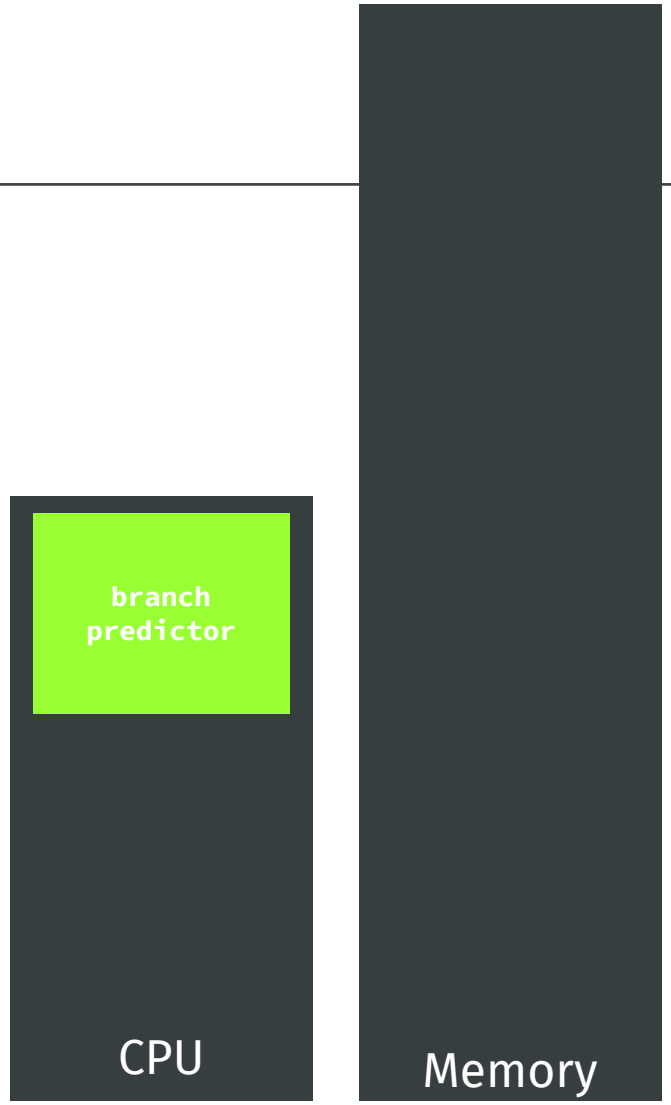
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A

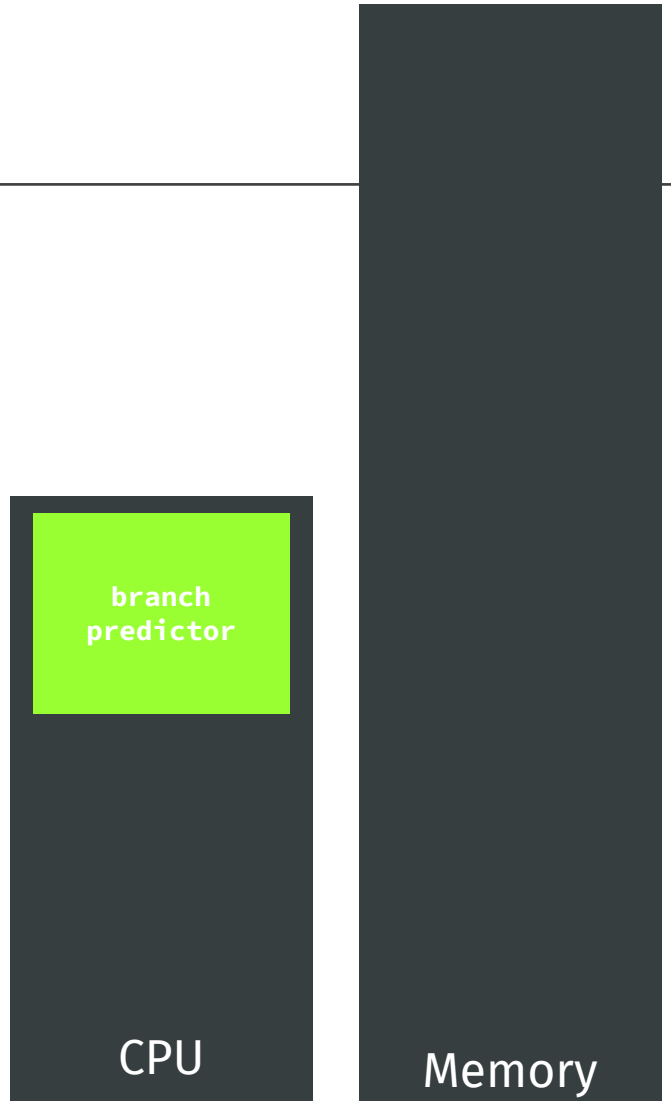
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose B and A_sz not in cache, A[N] (**the secret**!) in cache

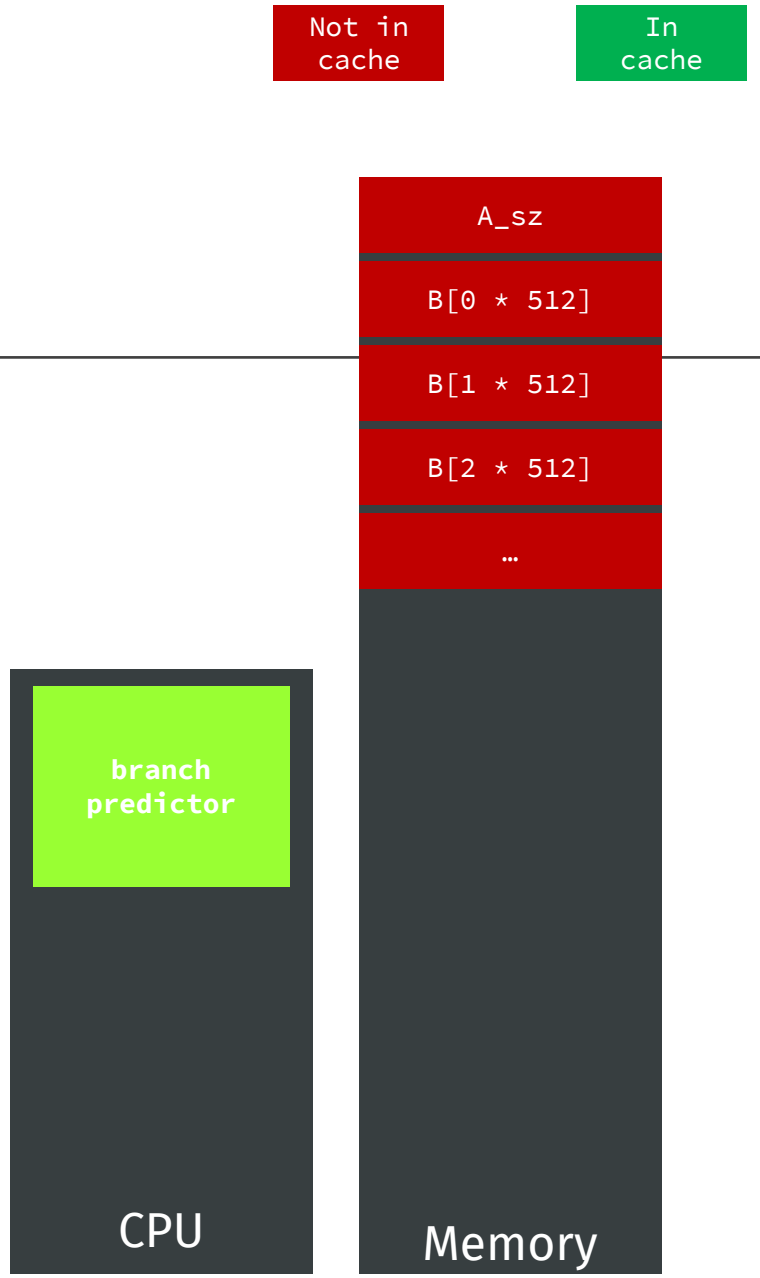
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose B and A_sz not in cache, A[N] (**the secret**!) in cache

```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

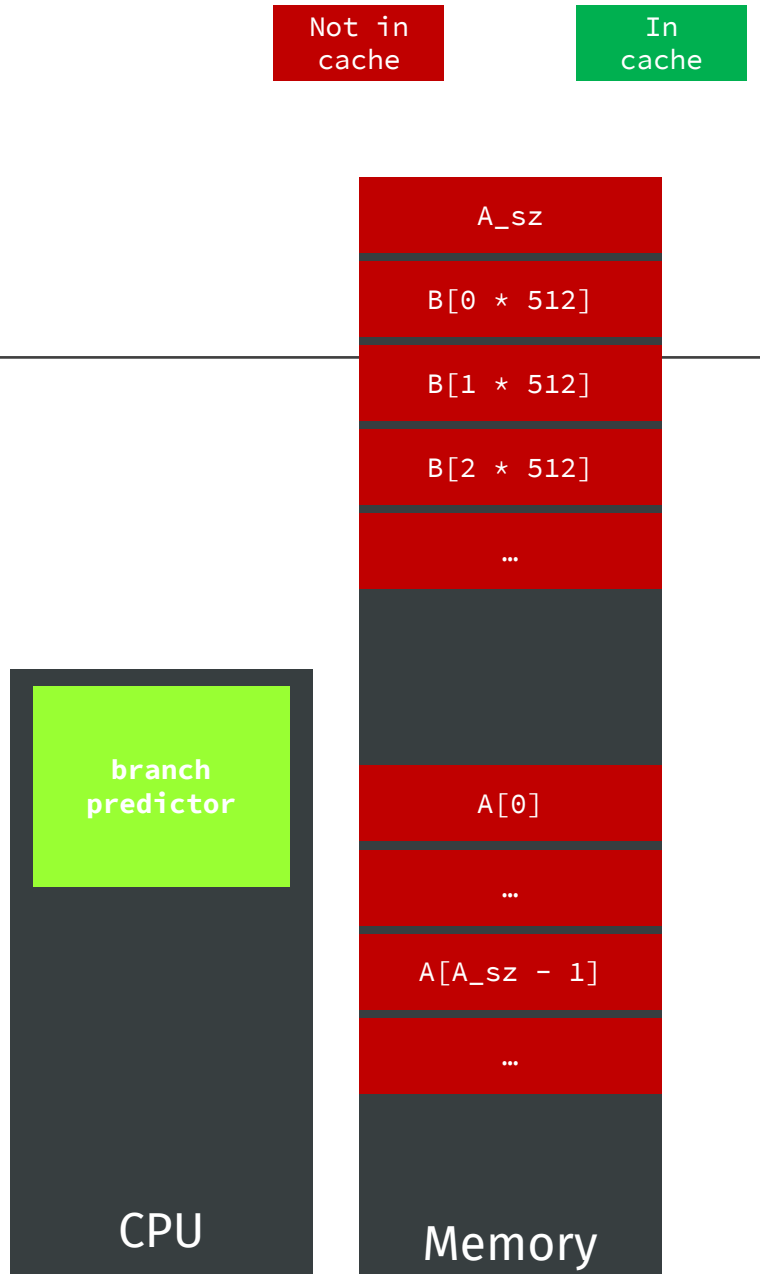
1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose B and A_sz not in cache, A[N] (**the secret**!) in cache

```
if (x < A_sz)
  y = B[A[x] * 512];
```

Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose B and A_sz not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with x = N > A_sz

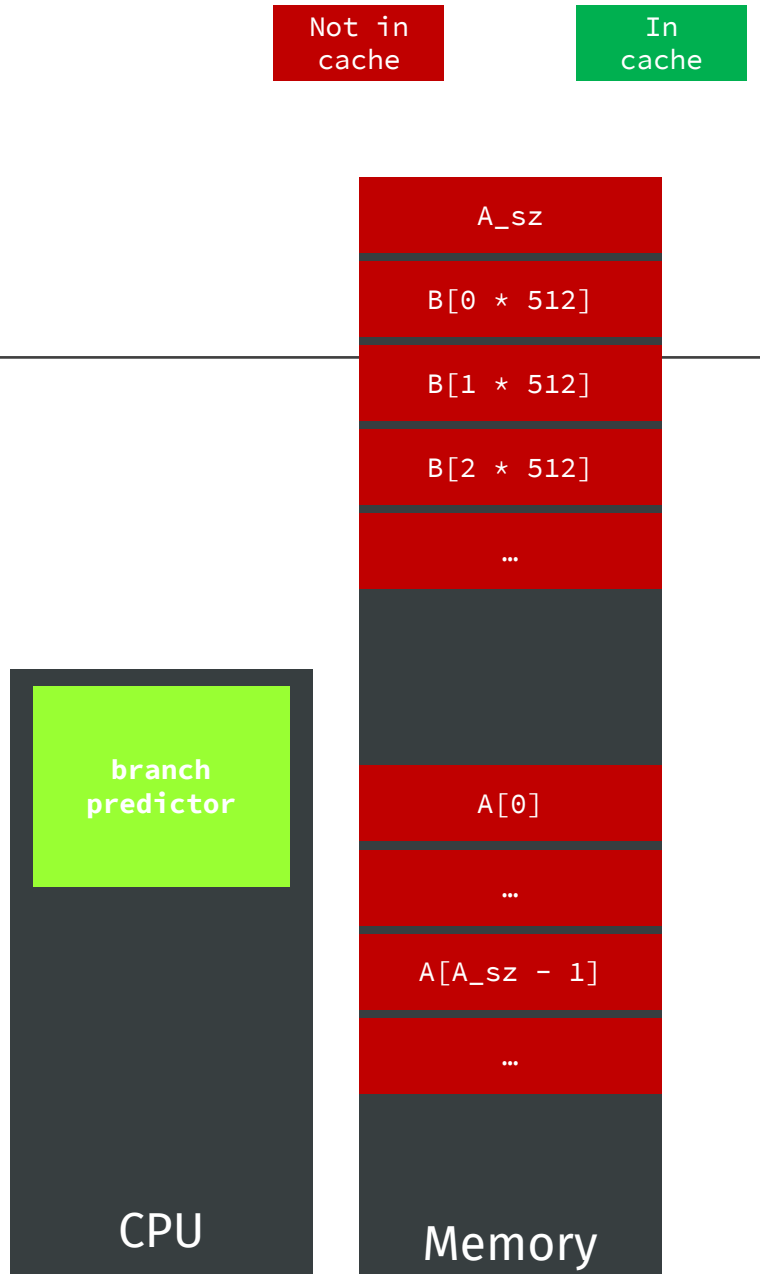
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)

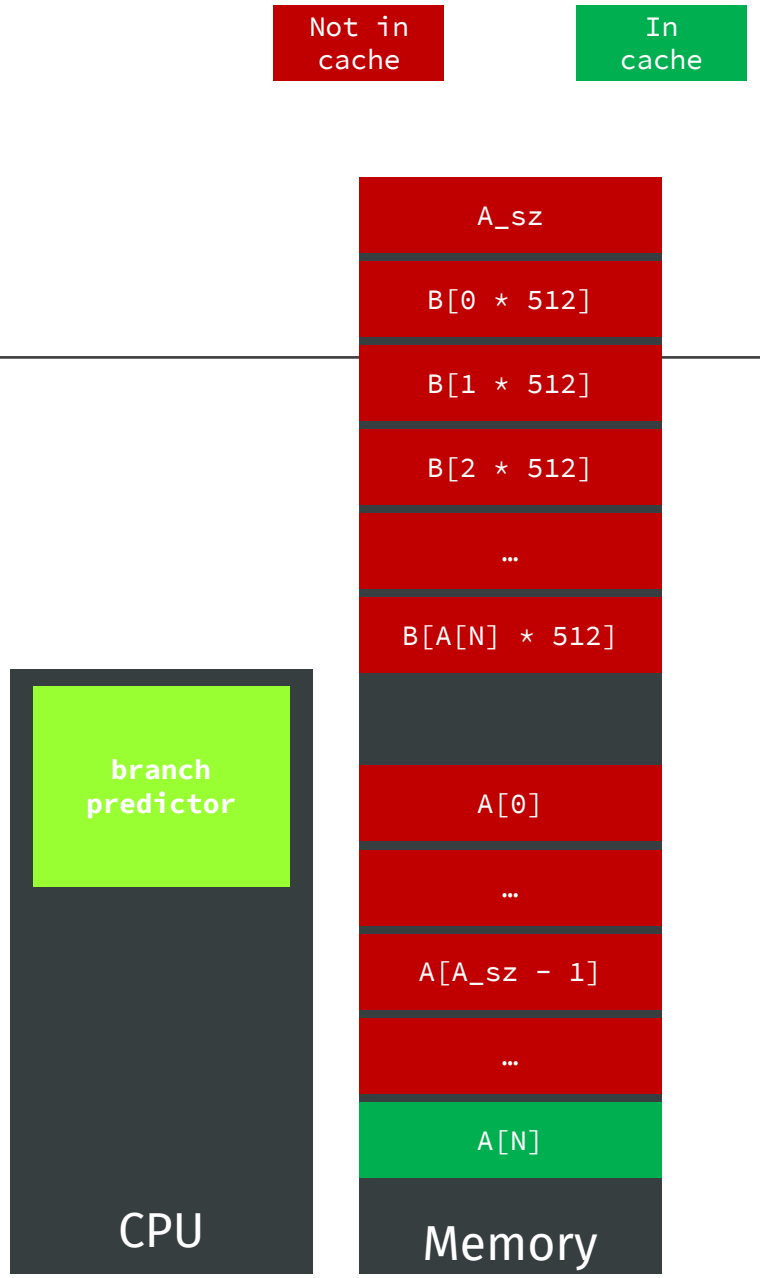
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)

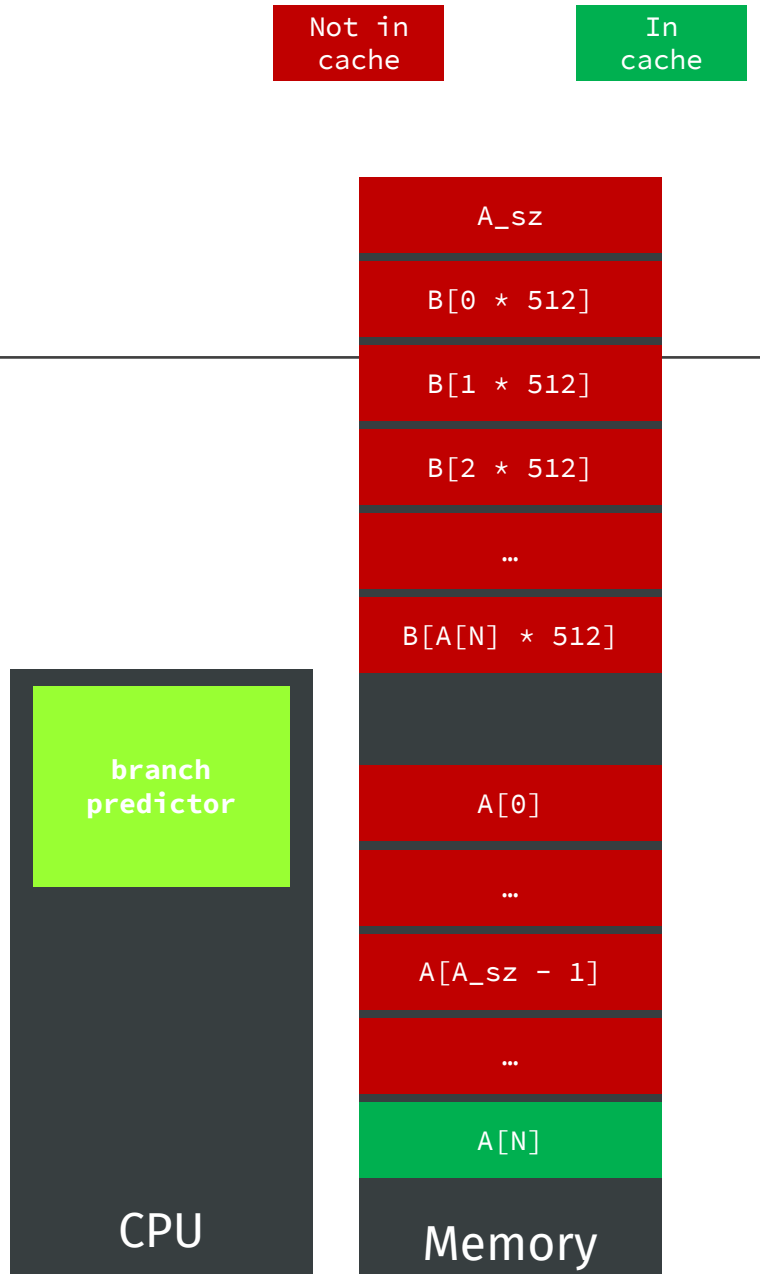
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)
7. At that point, B[A[N] * 512] is loaded in the cache

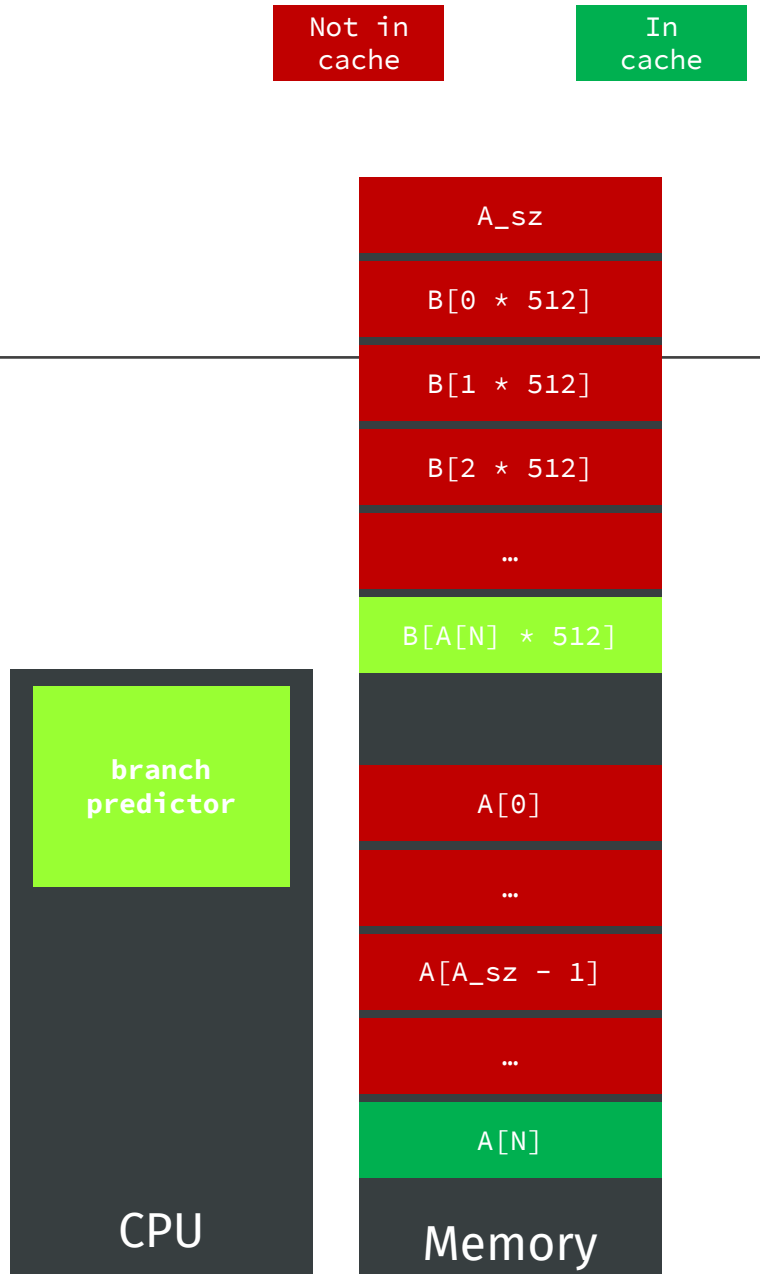
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)
7. At that point, B[A[N] * 512] is loaded in the cache

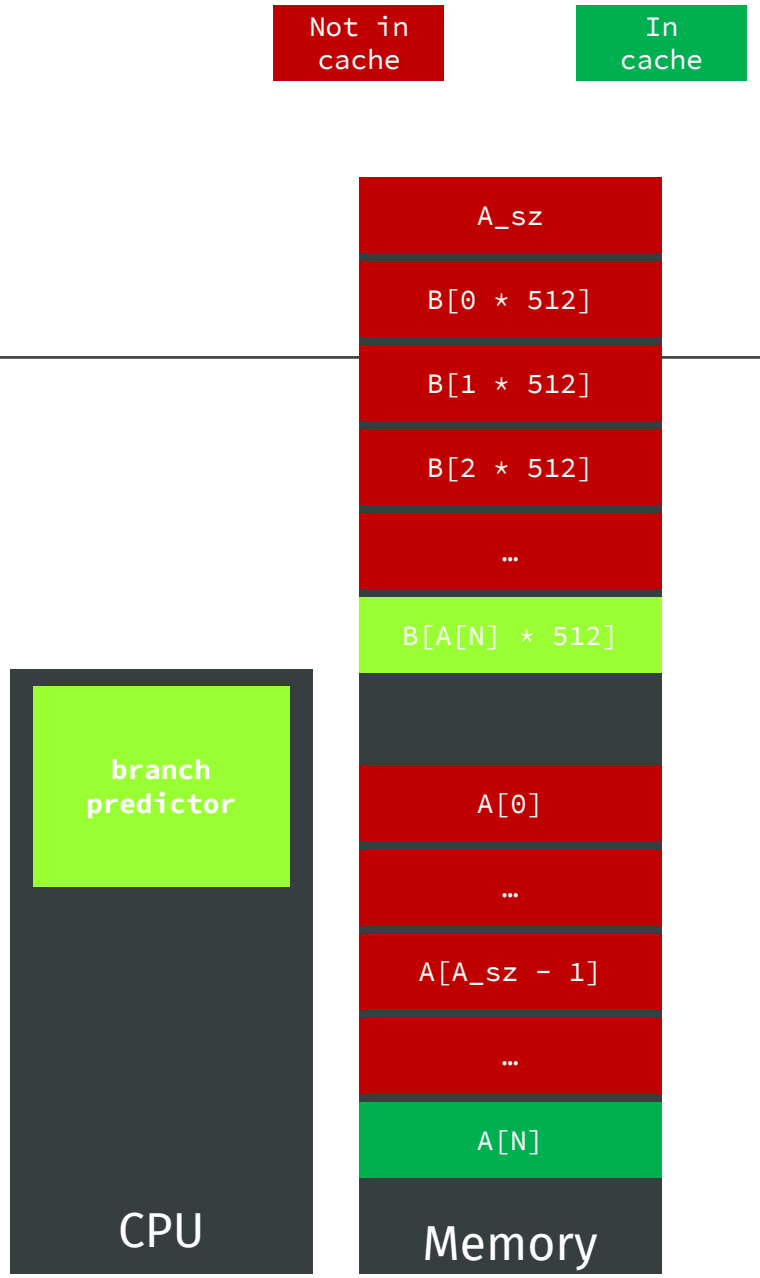
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)
7. At that point, B[A[N] * 512] is loaded in the cache
8. CPU discovers that the condition was false: **rollback!**

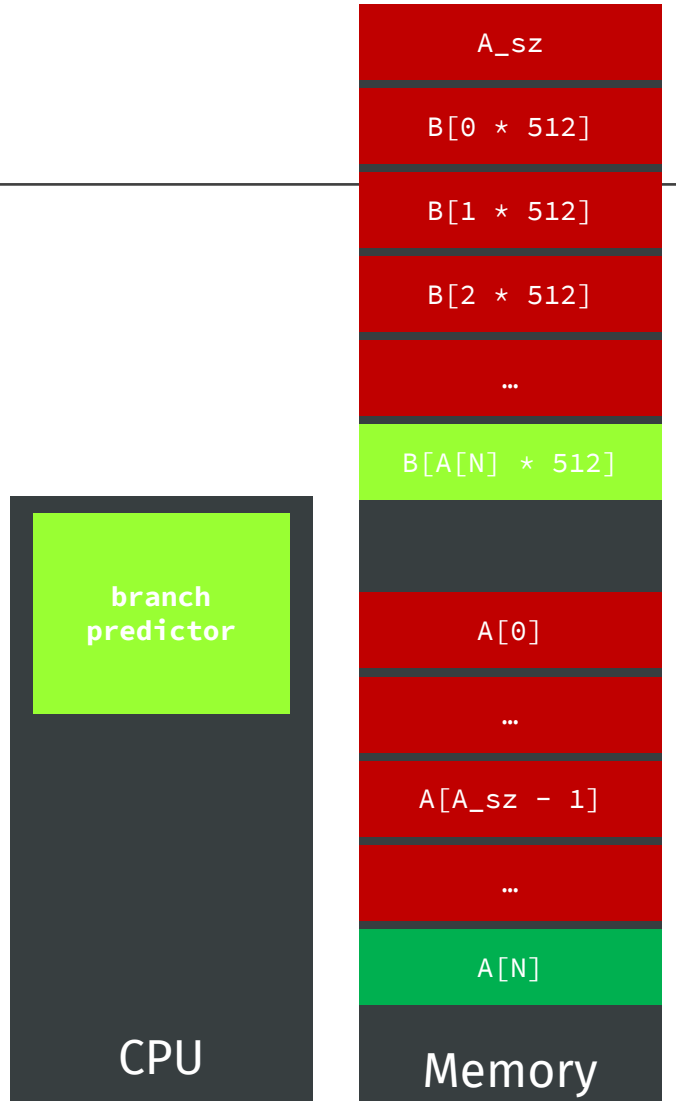
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)
7. At that point, B[A[N] * 512] is loaded in the cache
8. CPU discovers that the condition was false: **rollback!**
9. When control returns to caller (attacker) it just goes over `B[i * 512]` and measures access time:

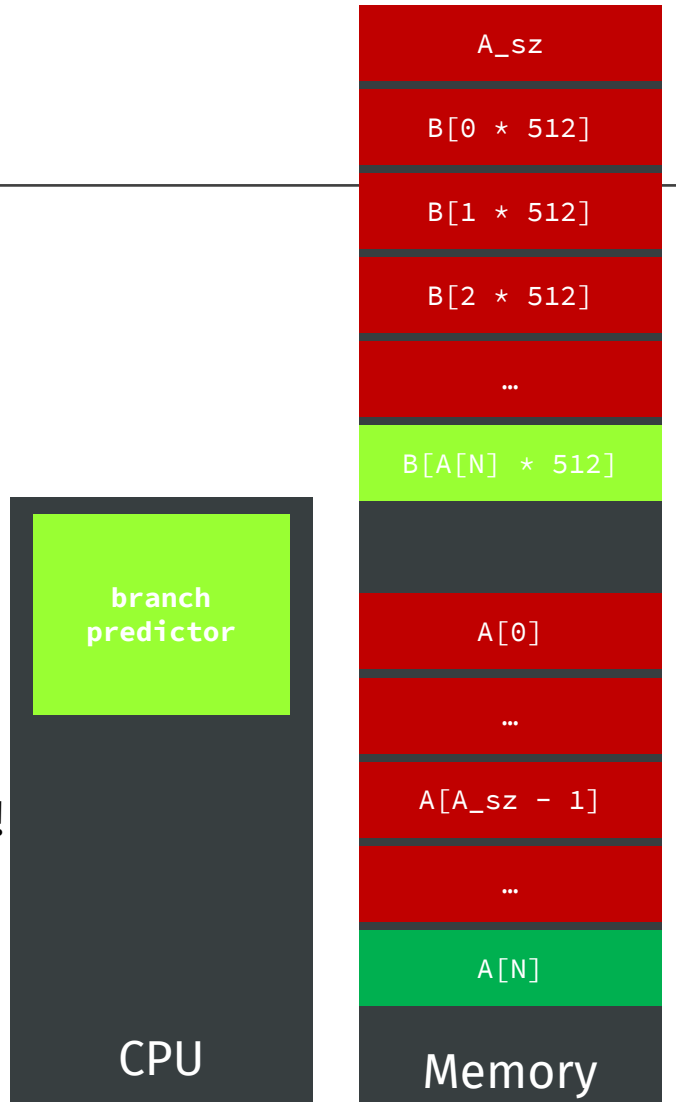
```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Choose a “malicious” index (out-of-bound) x = N for A
4. Suppose `B` and `A_sz` not in cache, A[N] (**the secret**!) in cache
5. Call the gadget with `x = N > A_sz`
6. CPU mis-speculates and accesses `A[N]` (fast!)
7. At that point, B[A[N] * 512] is loaded in the cache
8. CPU discovers that the condition was false: **rollback!**
9. When control returns to caller (attacker) it just goes over
   `B[i * 512]` and measures access time:
   - for `i = A[N]` the access will be fast, the secret is leaked!

```
if (x < A_sz)
  y = B[A[x] * 512];
```



Spectre v1.1

Spectre v1.1

1. Find a **code gadget** in the victim

Spectre v1.1

1. Find a **code gadget** in the victim

```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

Spectre v1.1

1. Find a **code gadget** in the victim
2. Mis-train the branch predictor

```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

Spectre v1.1

1. Find a **code gadget** in the victim
2. Mis-train the branch predictor
3. Assume:
   - `A, A_sz, B, x = N` as before
   - `A[N]` contains a secret

```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

Spectre v1.1

z points here


```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

1. Find a **code gadget** in the victim
2. Mis-train the branch predictor
3. Assume:
   - `A, A_sz, B, x = N` as before
   - `A[N]` contains a secret
   - `y` s.t. `A[y]` points on the return address on the stack
   - `z` points to an instruction accessing `B[A[N] * 512]`

Spectre v1.1

z points here


```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

1. Find a **code gadget** in the victim
2. Mis-train the branch predictor
3. Assume:
   - `A, A_sz, B, x = N` as before
   - `A[N]` contains a secret
   - `y` s.t. `A[y]` points on the return address on the stack
   - `z` points to an instruction accessing `B[A[N] * 512]`
4. If function `bar` returns during mis-speculation, the CPU loads `B[A[N] * 512]` in cache

Spectre v1.1

z points here

```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

1. Find a **code gadget** in the victim
2. Mis-train the branch predictor
3. Assume:
   - `A, A_sz, B, x = N` as before
   - `A[N]` contains a secret
   - `y` s.t. `A[y]` points on the return address on the stack
   - `z` points to an instruction accessing `B[A[N] * 512]`
4. If function `bar` returns during mis-speculation, the CPU loads `B[A[N] * 512]` in cache
5. Despite rollback, when control returns to caller (attacker) it just goes over `B[i * 512]` and measures access time:

Spectre v1.1

z points here

```
void foo () {
if (x < A_sz)
  y = B[A[x] * 512];
}
…

void bar () {
if (y < A_sz)
  A[y] = z;
}
```

1. Find a **code gadget** in the victim
2. Mis-train the branch predictor
3. Assume:
   - `A, A_sz, B, x = N` as before
   - `A[N]` contains a secret
   - `y` s.t. `A[y]` points on the return address on the stack
   - `z` points to an instruction accessing `B[A[N] * 512]`
4. If function `bar` returns during mis-speculation, the CPU loads `B[A[N] * 512]` in cache
5. Despite rollback, when control returns to caller (attacker) it just goes over `B[i * 512]` and measures access time:
   - for `i = A[N]` the access will be fast, the secret is leaked again!

Mitigating Spectre

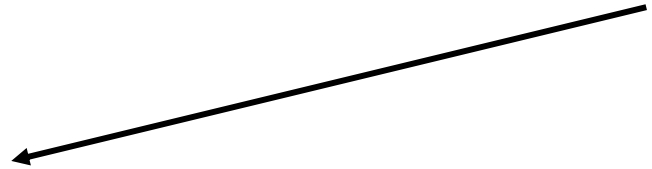
- Mitigations for Spectre are hard
- Also, the number of different variants does not help

Mitigating Spectre

- Mitigations for Spectre are hard
- Also, the number of different variants does not help
- One simple idea is to **stop speculation** from accessing to data it should not:

Mitigating Spectre

- Mitigations for Spectre are hard
- Also, the number of different variants does not help
- One simple idea is to **stop speculation** from accessing to data it should not:



Fences

Mitigating Spectre

- Mitigations for Spectre are hard
- Also, the number of different variants does not help
- One simple idea is to **stop speculation** from accessing to data it should not:

Mitigating Spectre: fences

A fence is a processor instruction that inhibits speculation

Mitigating Spectre: fences

A fence is a processor instruction that inhibits speculation

- On x86_64, we use `lfence` on all branches to inhibit speculation;
- Despite the name, `lfence` serializes execution completely, and not just loads

Mitigating Spectre: fences

A fence is a processor instruction that inhibits speculation

- On x86_64, we use `lfence` on all branches to inhibit speculation;
- Despite the name, `lfence` serializes execution completely, and not just loads
- In pseudo-C:

```
if (x < A_sz) {
  __asm__(“lfence”);
  y = B[A[x] * 512];
}
else {
  __asm__(“lfence”);
  /* … */
}
```

Mitigating Spectre: fences

```
if (x < A_sz) {
  __asm__(“lfence”);
  y = B[A[x] * 512];
}
else {
  __asm__(“lfence”);
  /* … */
}
```

Mitigating Spectre: fences

- The **Good:**
  - It works (mostly :)
  - Probably nothing else!

```
if (x < A_sz) {
  __asm__(“lfence”);
  y = B[A[x] * 512];
}
else {
  __asm__(“lfence”);
  /* … */
}
```

Mitigating Spectre: fences

- The **Good:**
  - It works (mostly :)
  - Probably nothing else!
- The **Bad:**
  - **Highly** inefficient (it stops speculation **completely**)
  - Requires re-compilation

```
if (x < A_sz) {
  __asm__(“lfence”);
  y = B[A[x] * 512];
}
else {
  __asm__(“lfence”);
  /* … */
}
```

Mitigating Spectre: fences

- The **Good:**
  - It works (mostly :)
  - Probably nothing else!
- The **Bad:**
  - **Highly** inefficient (it stops speculation **completely**)
  - Requires re-compilation

```
if (x < A_sz) {
  __asm__(“lfence”);
  y = B[A[x] * 512];
}
else {
  __asm__(“lfence”);
  /* … */
}
```

- To make this a bit better one could use static analysis to decide where fences should go
  - E.g., **Microsoft MSVC** does that by detecting known problematic patterns or **Blade tool**

Mitigating Spectre: SLH https://llvm.org/docs/SpeculativeLoadHardening.html

Mitigating Spectre: SLH https://llvm.org/docs/SpeculativeLoadHardening.html

- Speculative Load Hardening (SLH) is a bit more sophisticated than fences

Mitigating Spectre: SLH https://llvm.org/docs/SpeculativeLoadHardening.html

- Speculative Load Hardening (SLH) is a bit more sophisticated than fences
- **Idea:**
  - It may be more efficient to introduce an **artificial** data dependency between the condition of a jump and the pointer

Mitigating Spectre: SLH

https://llvm.org/docs/SpeculativeLoadHardening.html

- Speculative Load Hardening (SLH) is a bit more sophisticated than fences
- **Idea:**
  - It may be more efficient to introduce an **artificial** data dependency between the condition of a jump and the pointer
  - Simplifying, for Spectre v1 (assuming "_ ? _ : _" to be implemented w/o branching, e.g., using a `cmov`)

```
uint mask = ALL_ONES;
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

Mitigating Spectre: SLH

```
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

Mitigating Spectre: SLH

- The **Good:**
  - It works (with some changes also for other Spectre variants)
  - More efficient than fences
  - No need of static analysis code (but still could help)

```
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

Mitigating Spectre: SLH

```
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

- The **Good:**
  - It works (with some changes also for other Spectre variants)
  - More efficient than fences
  - No need of static analysis code (but still could help)
- The **Bad:**
  - Still slow (`mask` must be known when accessing to `B`, also when speculating!)
  - Must be carefully implemented
  - Still requires re-compilation

Mitigating Spectre: other ideas

Other possible fixes:

- Make secrets non reachable:
  - Just like **Chrome** and **Firefox**: isolate secrets in different processes
- Reduce the bandwidth of side-channels:
  - e.g., via shadow micro-arch state that can be discarded, less accurate timers, adding noise, …

Mitigating Spectre: other ideas

Other possible fixes:

- Make secrets non reachable:
  - Just like **Chrome** and **Firefox**: isolate secrets in different processes
- Reduce the bandwidth of side-channels:
  - e.g., via shadow micro-arch state that can be discarded, less accurate timers, adding noise, …

Is Spectre fixed?

Mitigating Spectre: other ideas

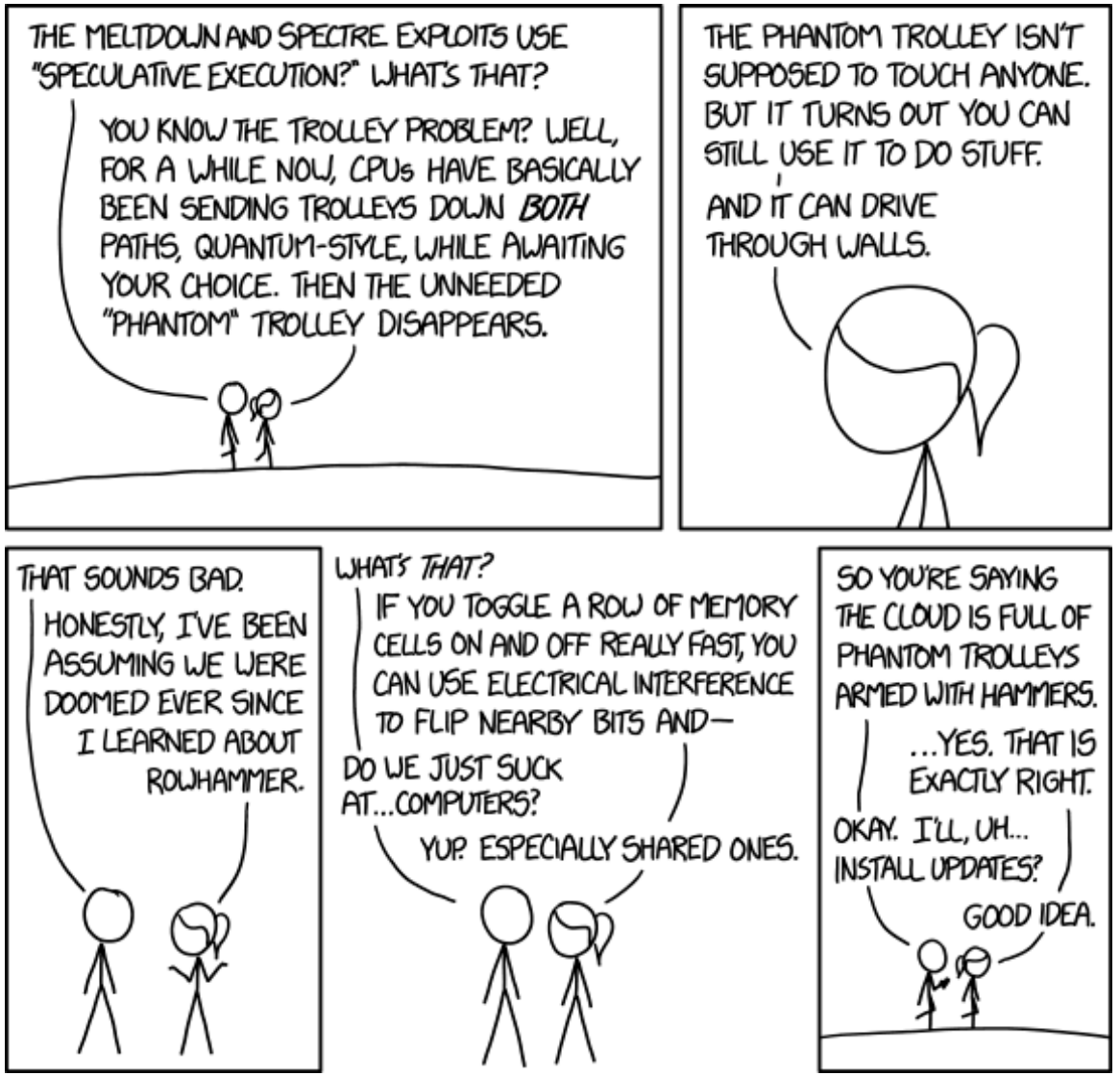
Other possible fixes:

- Make secrets non reachable:
  - Just like **Chrome** and **Firefox**: isolate secrets in different processes
- Reduce the bandwidth of side-channels:
  - e.g., via shadow micro-arch state that can be discarded, less accurate timers, adding noise, …

Is Spectre fixed?

Only if you are willing to re-compile/re-design your system and make it slow 😗

The End



https://xkcd.com/1938/