CSE251: System Programming

12. Cache (1)

Seongil Wi

Department of Computer Science and Engineering

HW3: Symbol Resolution

- Due: Apr. 24, 11:59 PM

- Main task: Your job is to use your knowledge of C, ELF, and linkers to recreate `hw3_a.c` and `hw3_b.c` so that their symbol tables match those shown in the hw3 handout

- You should submit a zip via BlackBoard

No Class on April 16 ☺

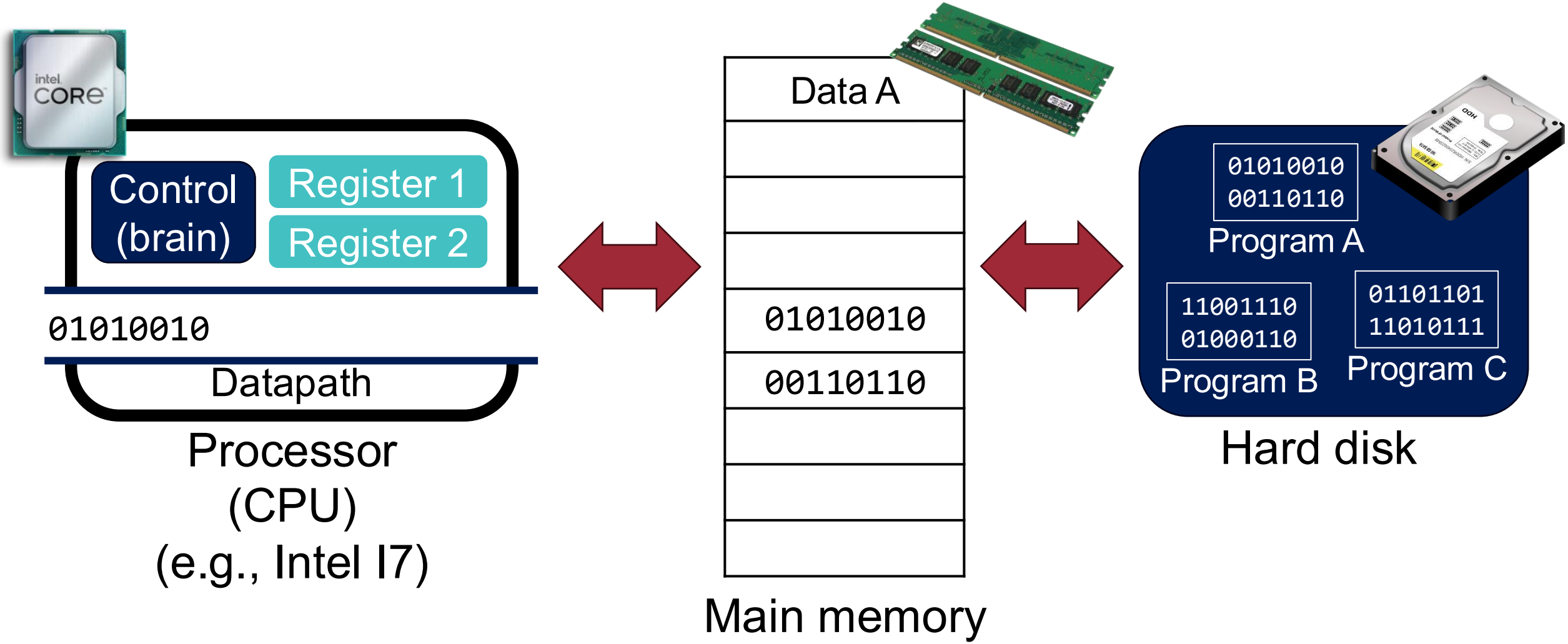
- There will be no class on April 16 to accommodate students' exam period and to provide time for working on HW3

No Class on Midterm Week

- Also, there will be no class on April 21 and 23

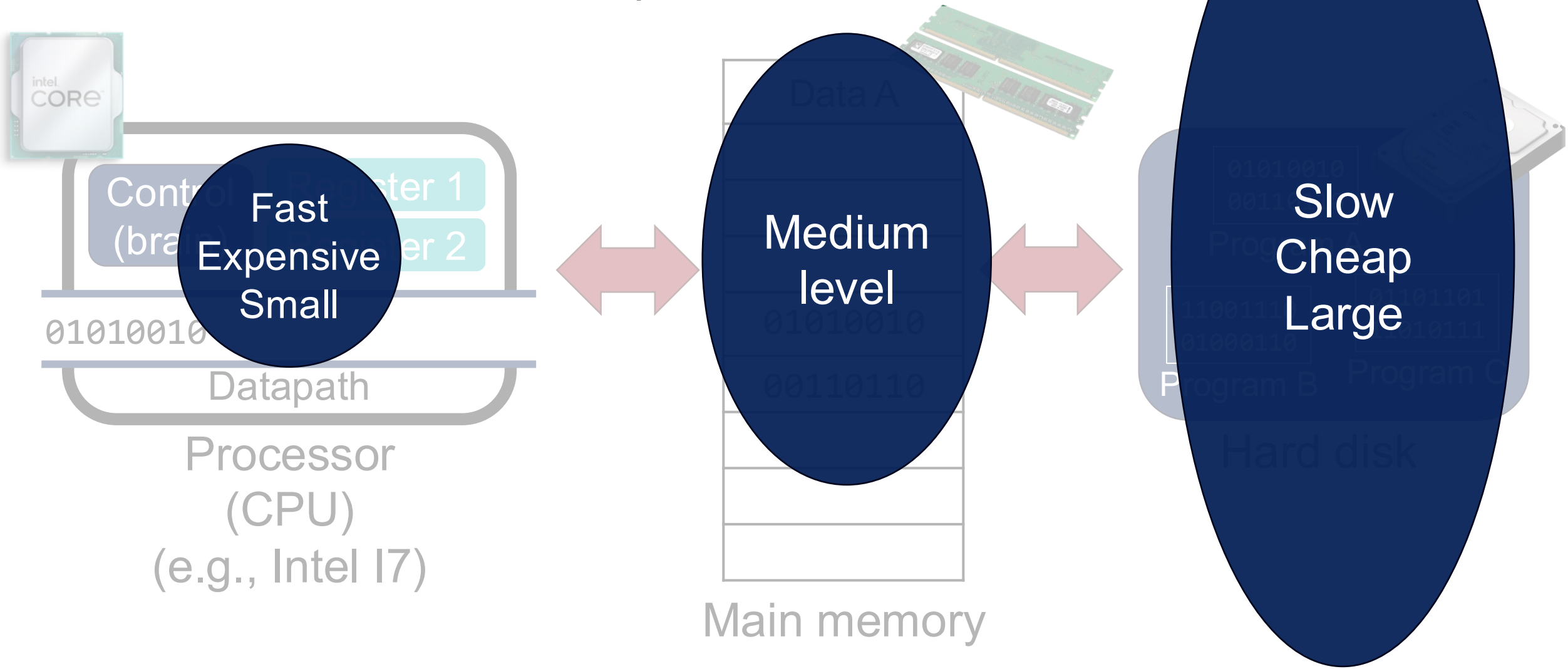
Recap: Memory Hierarchy

A structure that uses multiple levels of memories



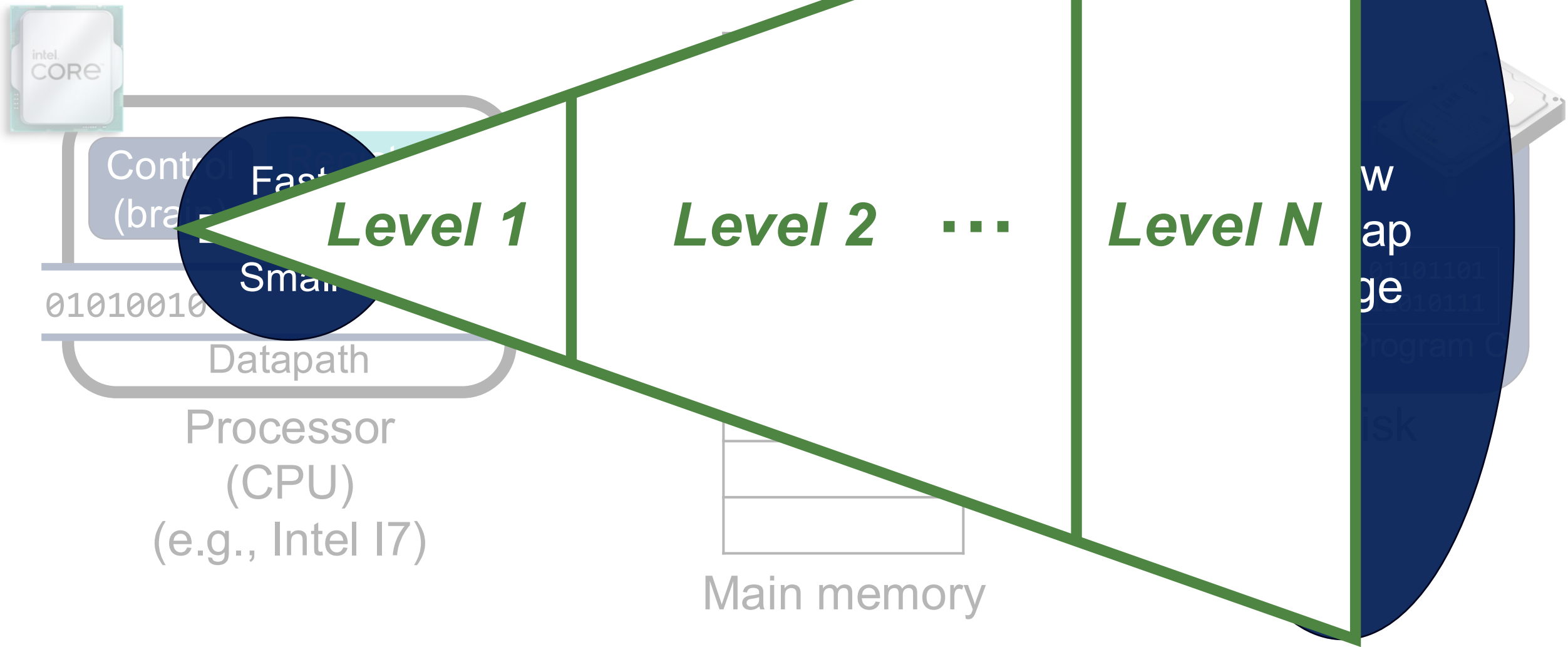
Recap: Memory Hierarchy

A structure that uses multiple levels of memories



Recap: Memory Hierarchy

A structure that uses multiple levels of mem



Recap: Locality (지역성)

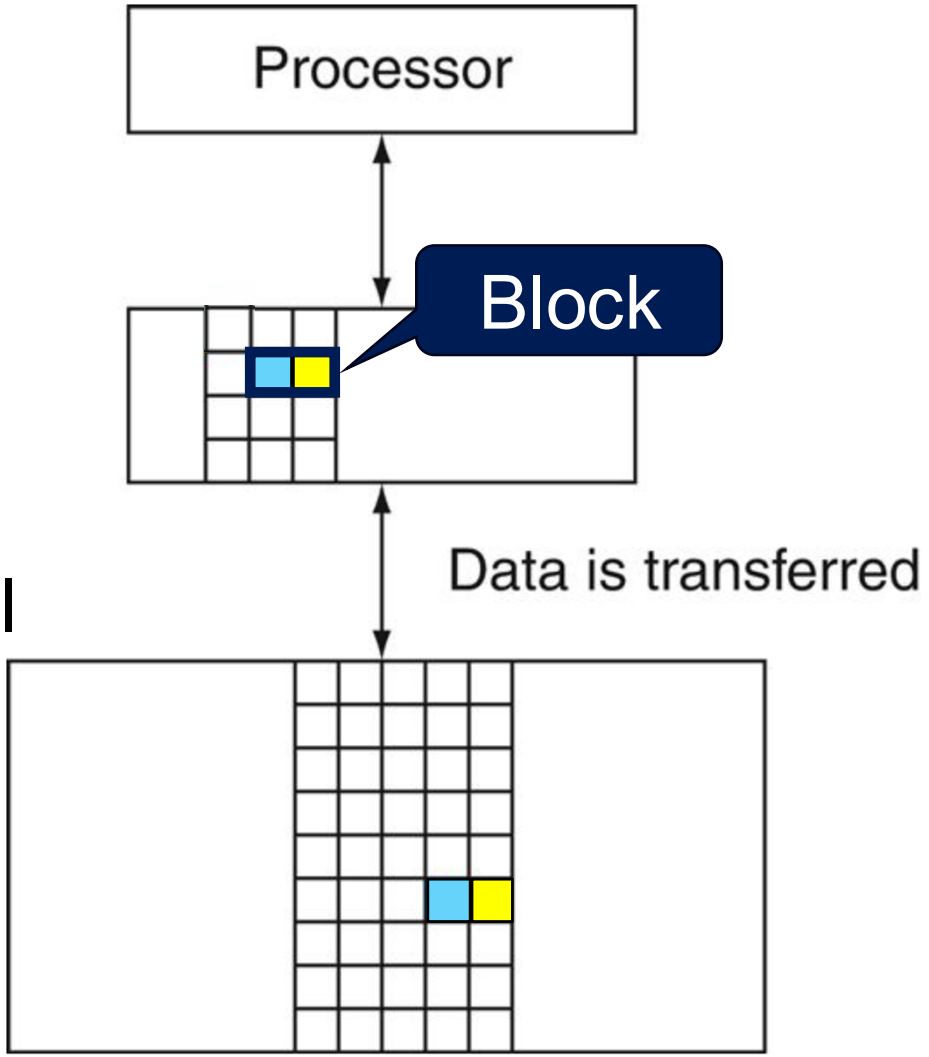
The tendency to access the same set of memory locations *repetitively* over a short period of time

1. **Temporal locality** (locality in time)
   - If an item is referenced, *the same item* will tend to be referenced again soon

2. **Spatial locality** (locality in space)
   - If an item is referenced, *nearby items* will tend to be referenced soon

Recap: Terms

- Block (a.k.a., line): unit of copying
  - Several words in cache memory
- **Hit**: data requested is in the upper level
- **Miss**: data requested is not in the upper level
  - Block copied from lower level



Today's Topic: Caches

Register 1

Register 2

On-chip cache (L1 cache)

Processor (CPU) (e.g., Intel I7)

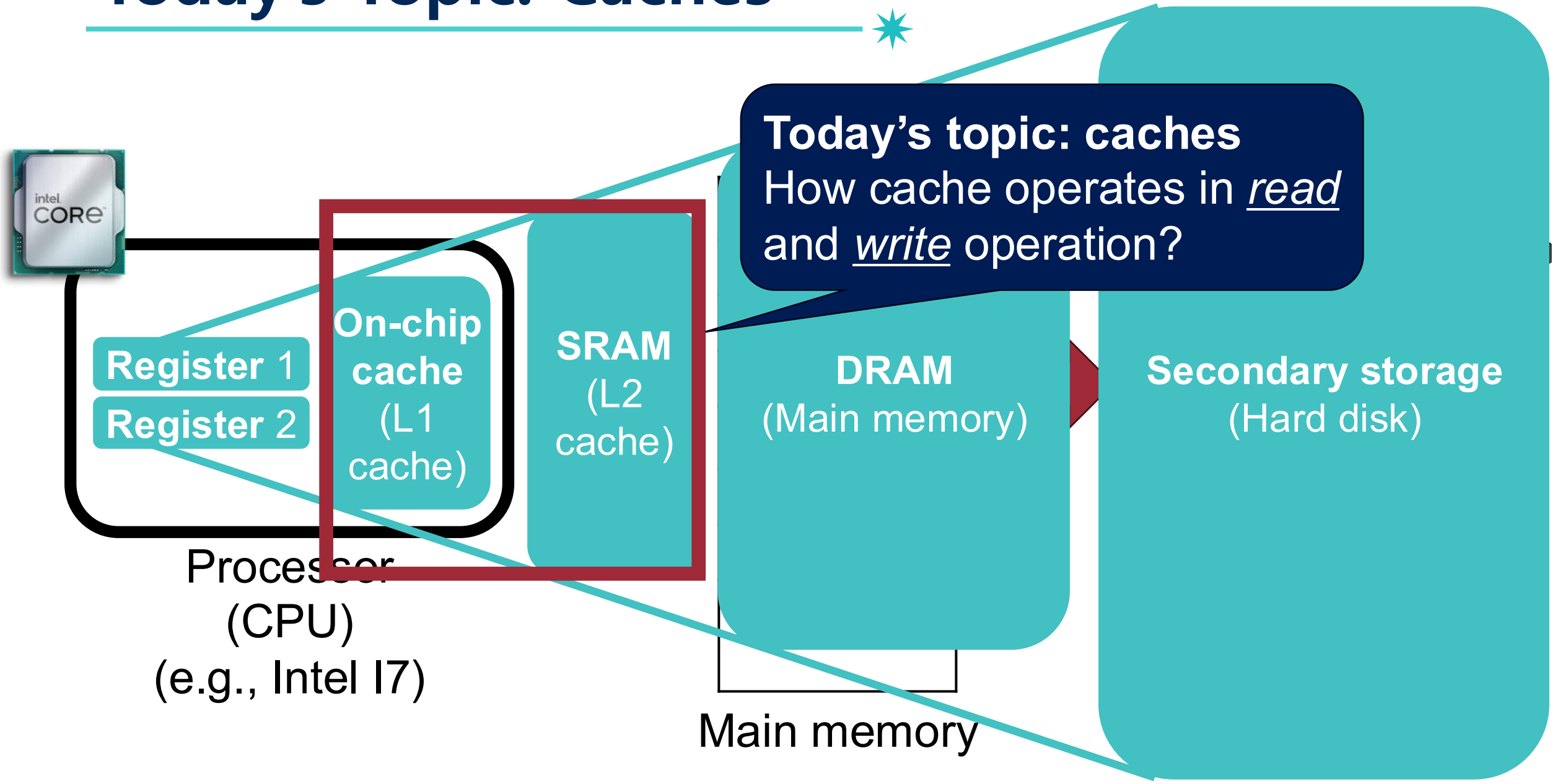
SRAM (L2 cache)

DRAM (Main memory)

Main memory

Secondary storage (Hard disk)

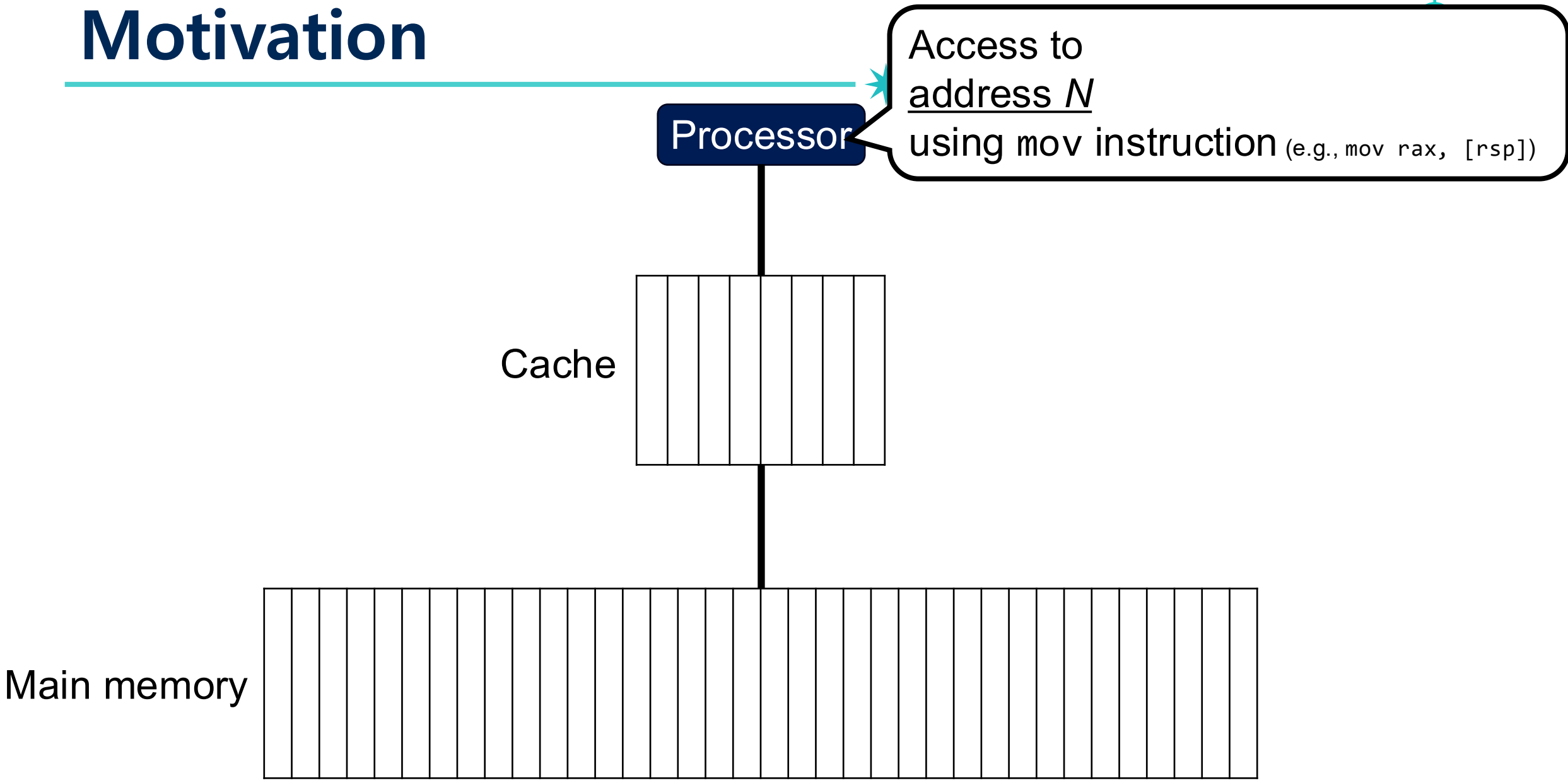
Today's Topic: Caches



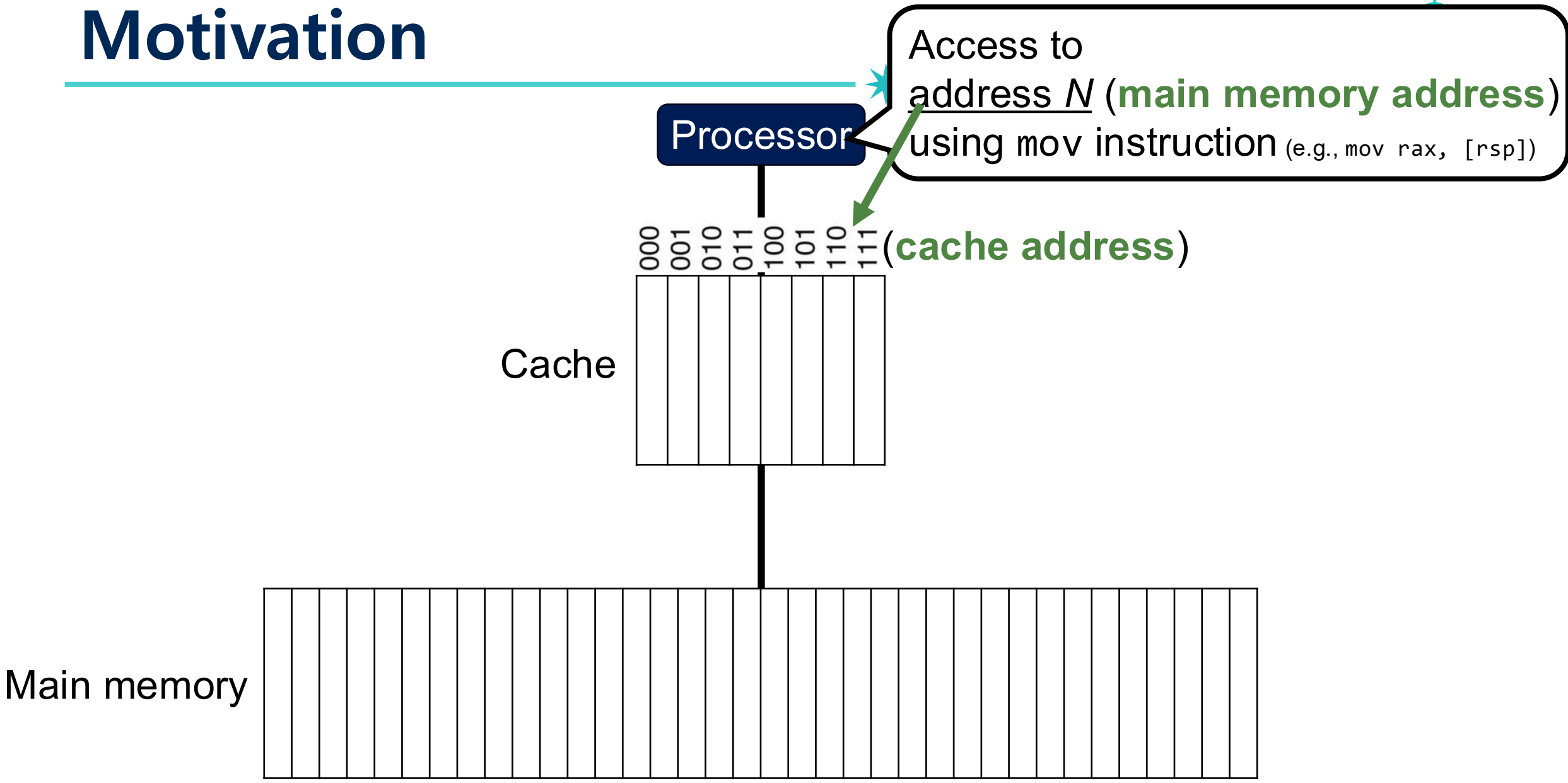
Caches

The level of the memory hierarchy closest to the CPU

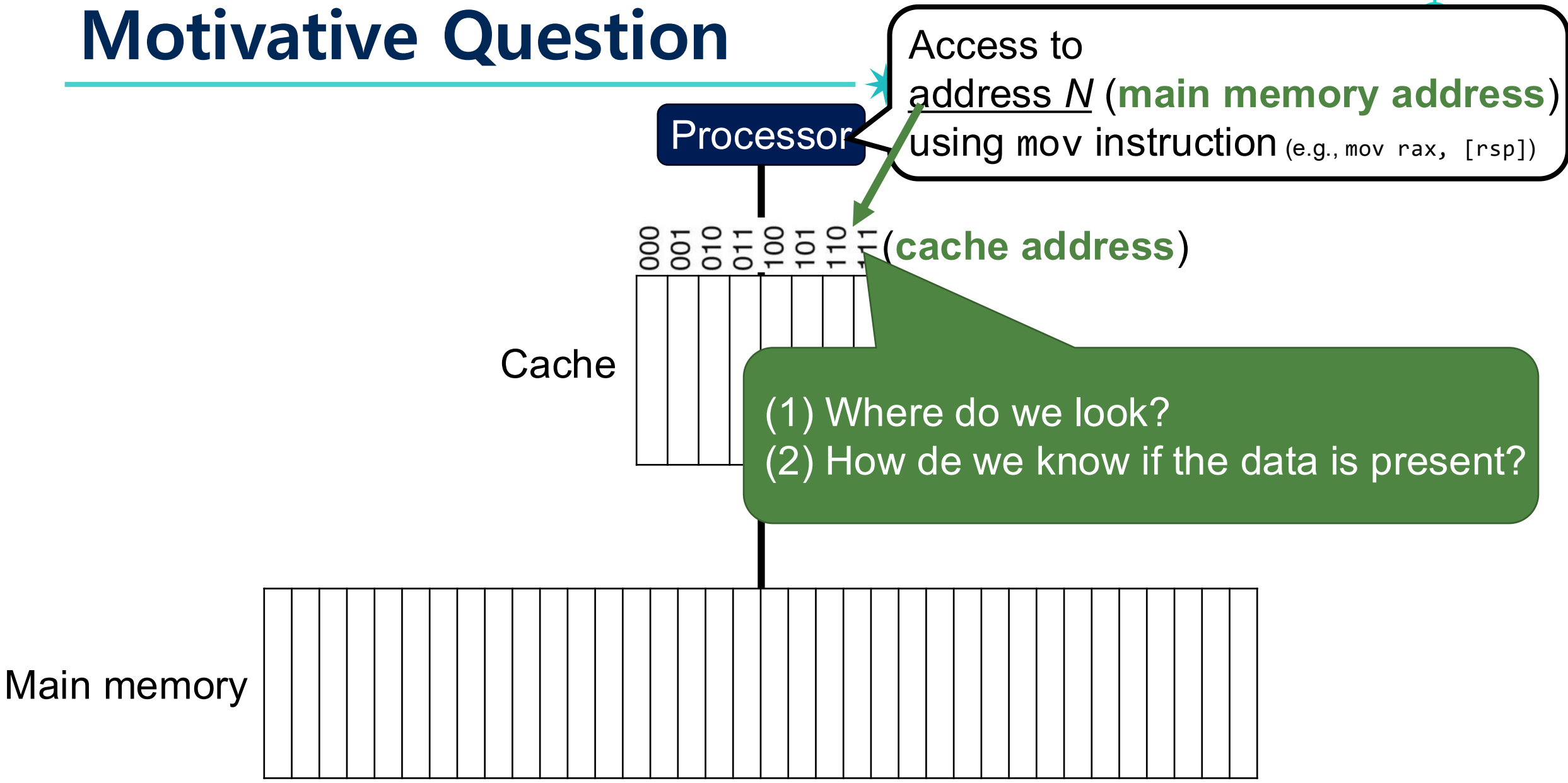
Motivation


Access to
address *N*
using `mov` instruction (e.g., `mov rax, [rsp]`)
Processor
Cache
Main memory

Motivation


Access to
address *N* (**main memory address**)
using `mov` instruction (e.g., `mov rax, [rsp]`)
Processor
000
001
010
011
100
101
110
111
(**cache address**)
Cache
Main memory

Motivative Question



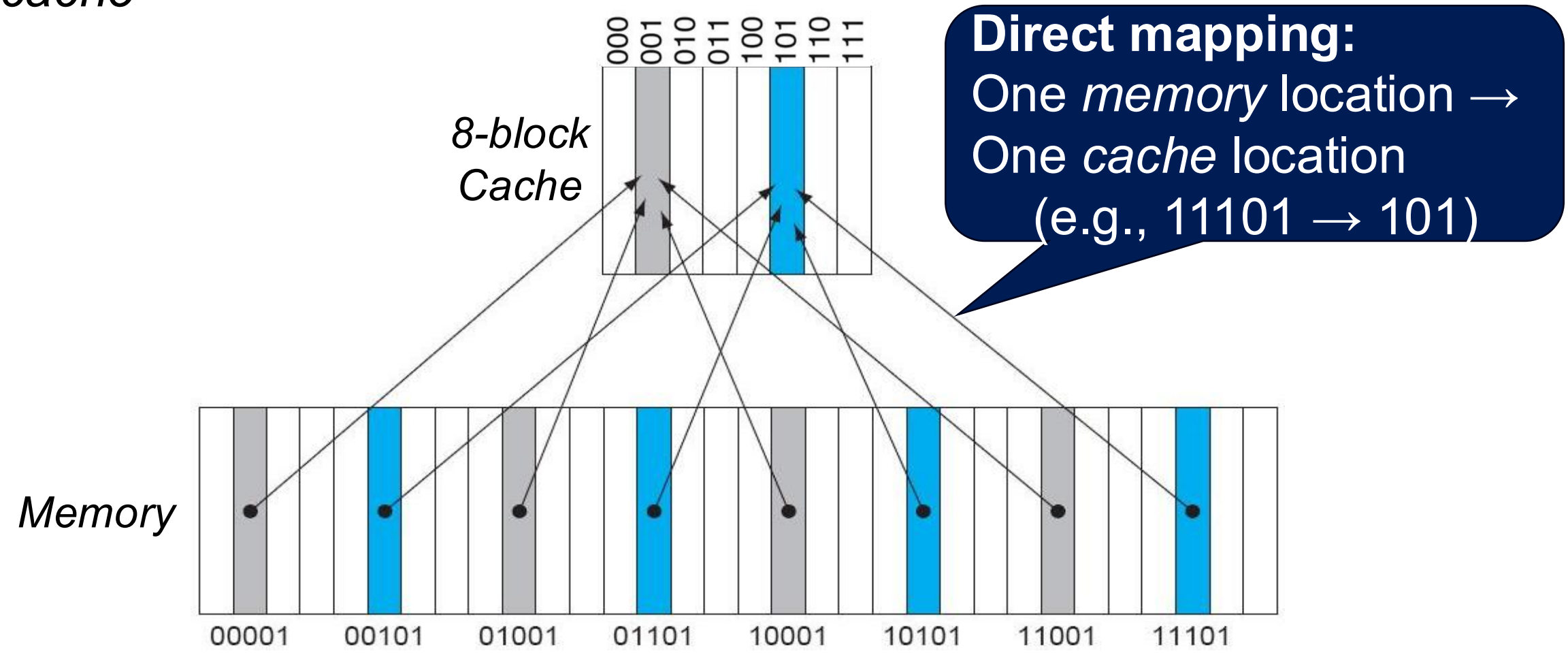
Direct Mapped Cache

Direct Mapped Cache

Each *memory* location is mapped to exactly one location in the *cache*

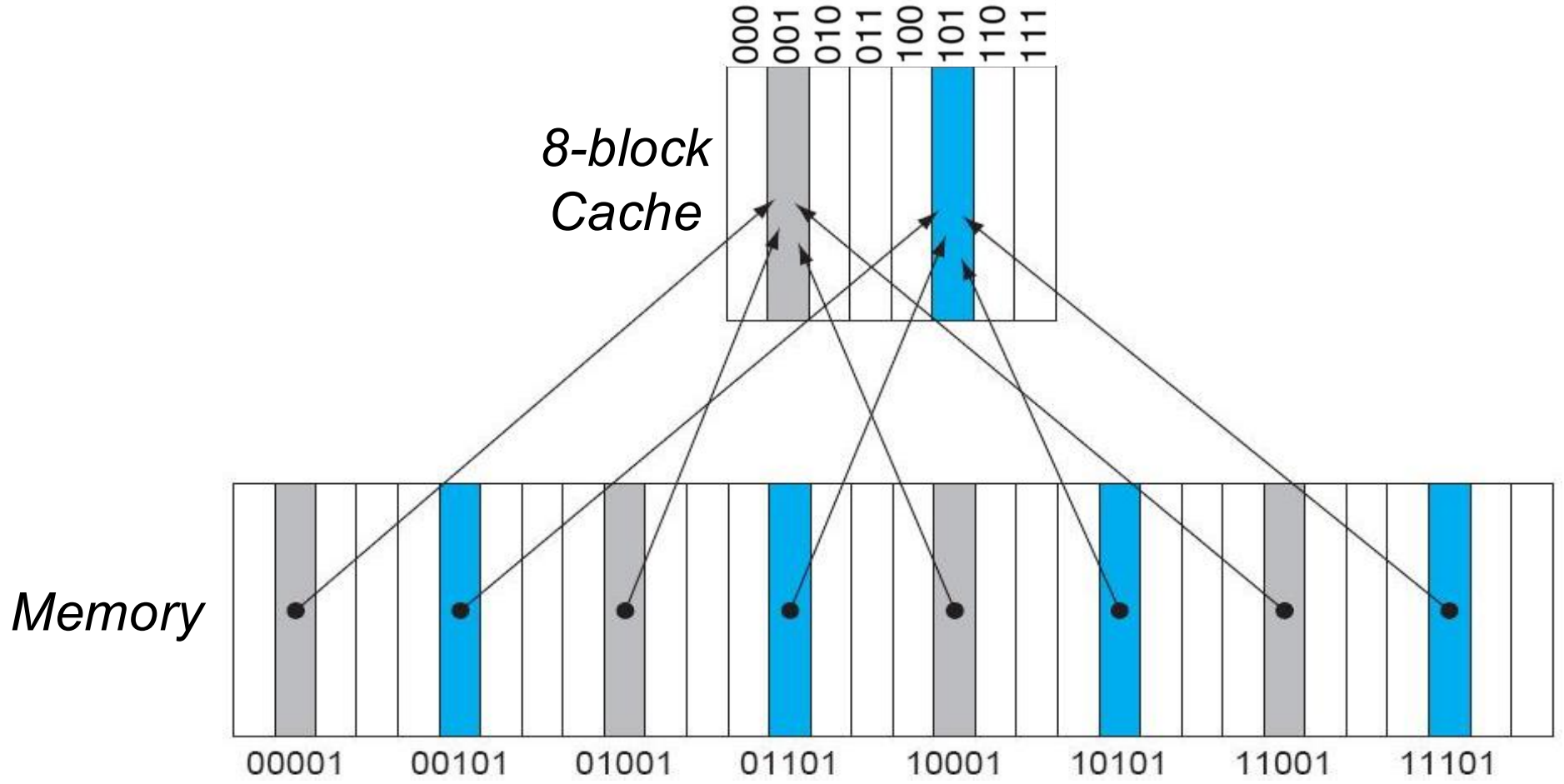
Direct Mapped Cache

Each *memory* location is mapped to exactly one location in the *cache*



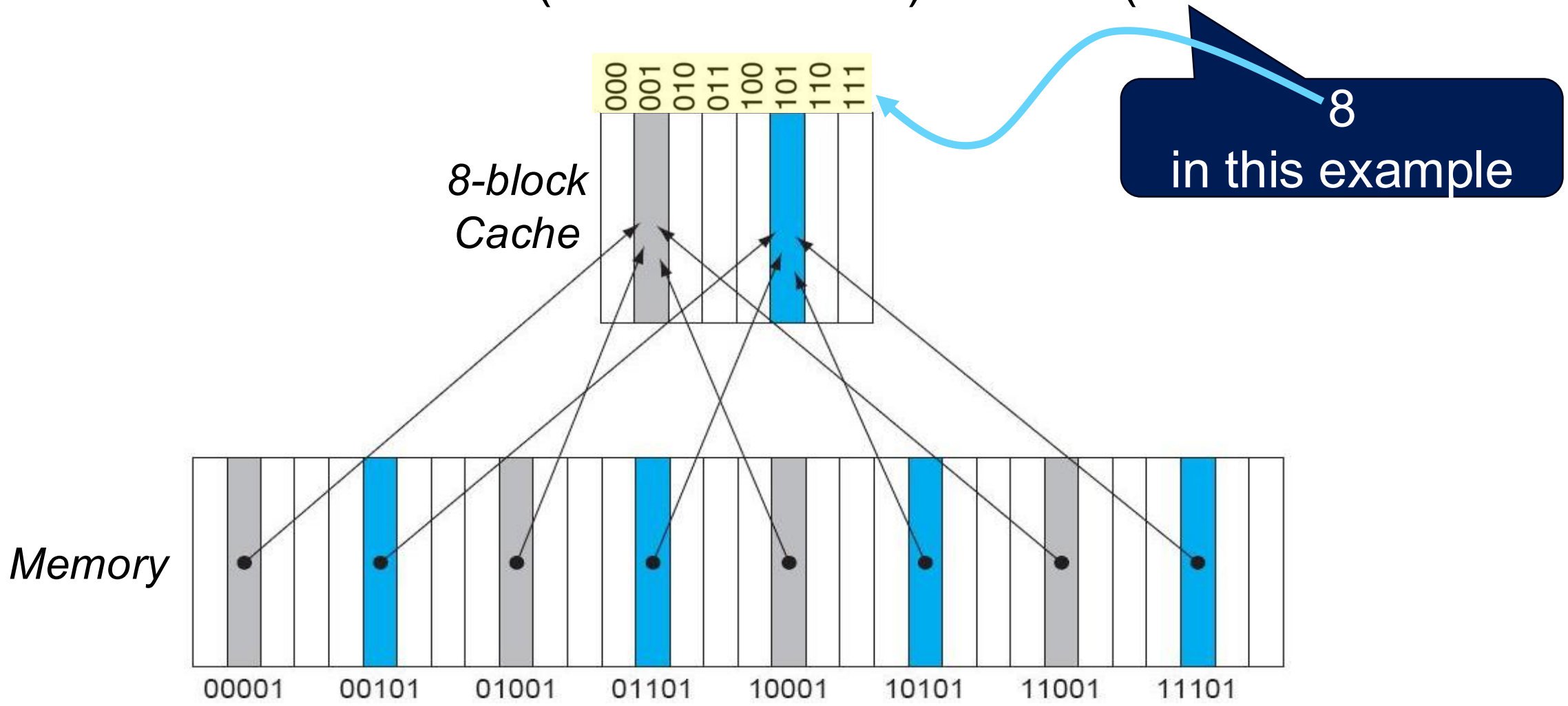
Mapping Algorithm

Location in the cache = (**Block address**) modulo (**# Blocks in cache**)



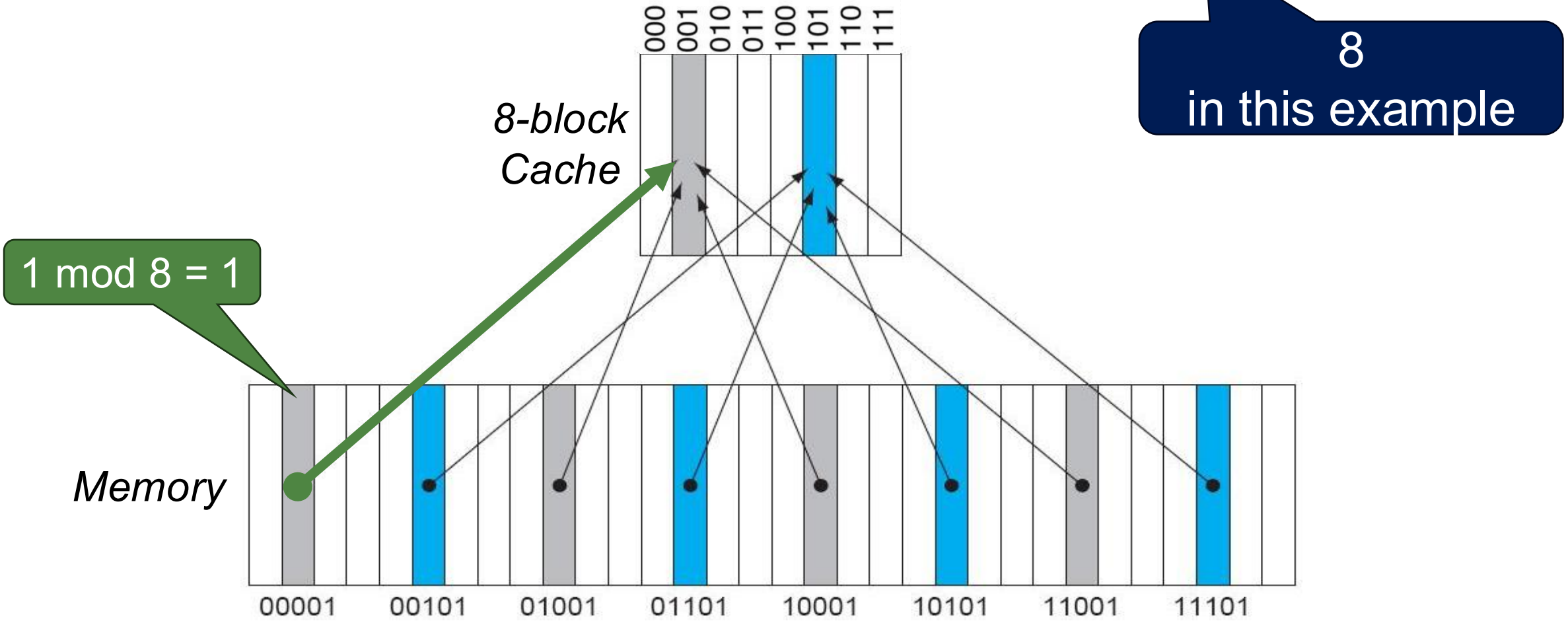
Mapping Algorithm

Location in the cache = (**Block address**) modulo (**# Blocks in cache**)



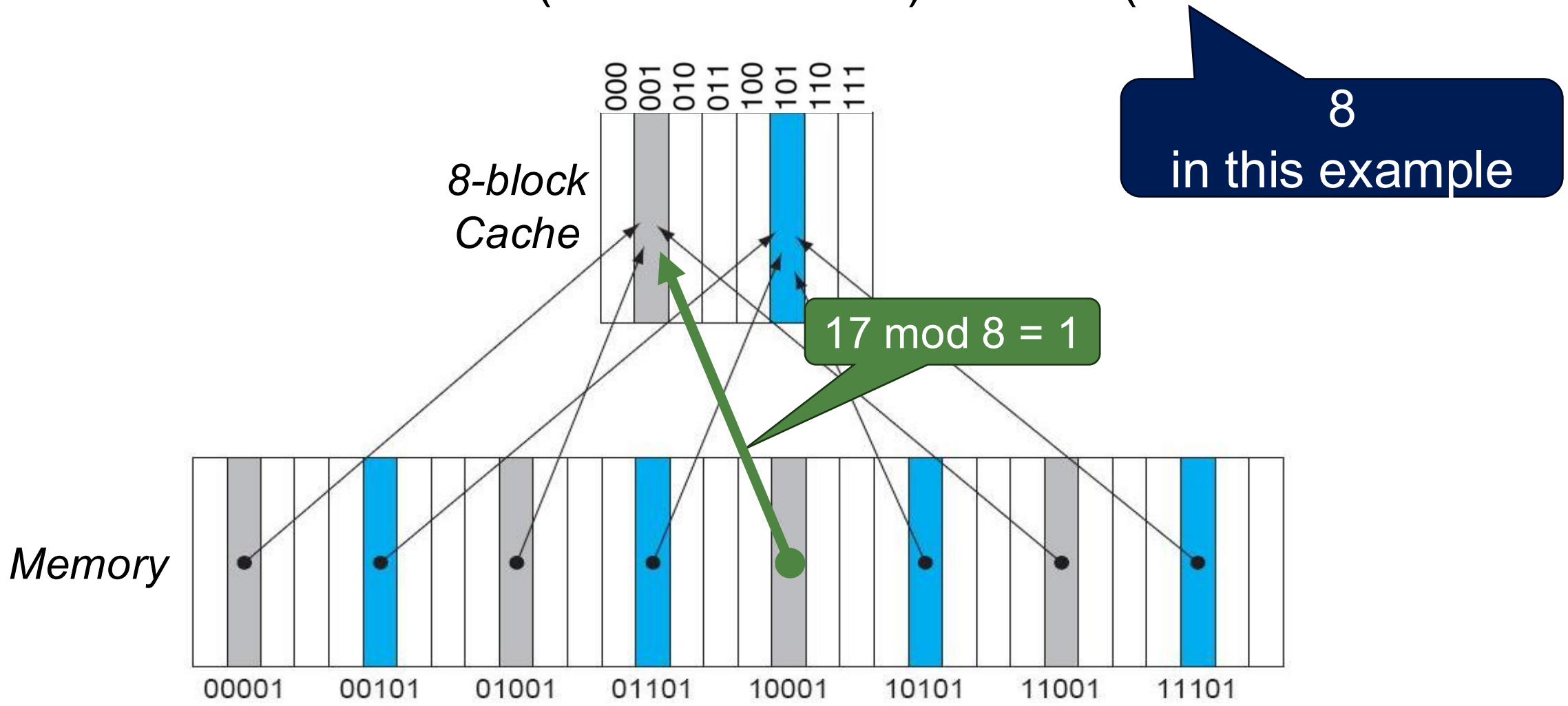
Mapping Algorithm

Location in the cache = (**Block address**) modulo (**# Blocks in cache**)



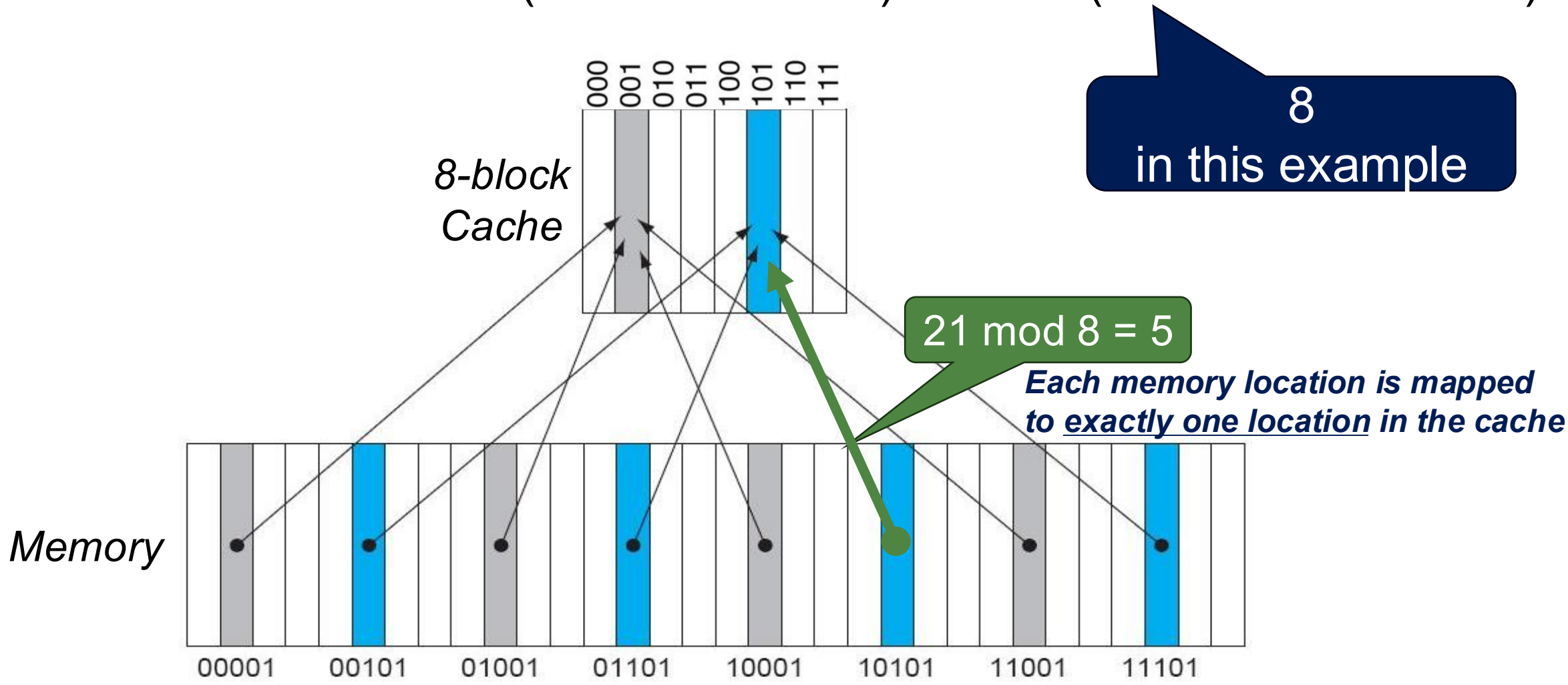
Mapping Algorithm

Location in the cache = (**Block address**) modulo (**# Blocks in cache**)



Mapping Algorithm

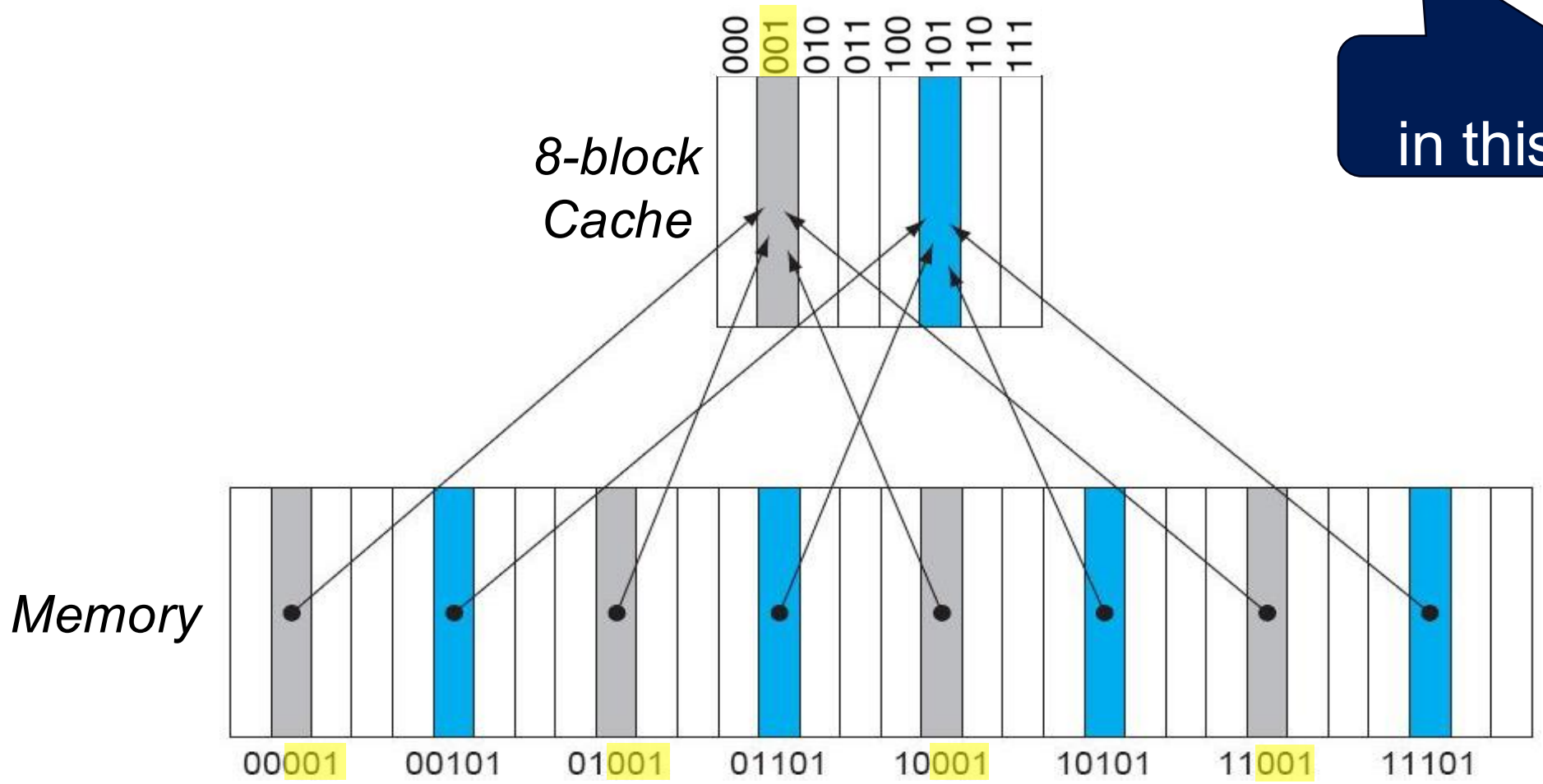
Location in the cache = (**Block address**) modulo (**# Blocks in cache**)



Mapping Algorithm: Binary View

Location in the cache = (**Block address**) modulo (**# Blocks in cache**)

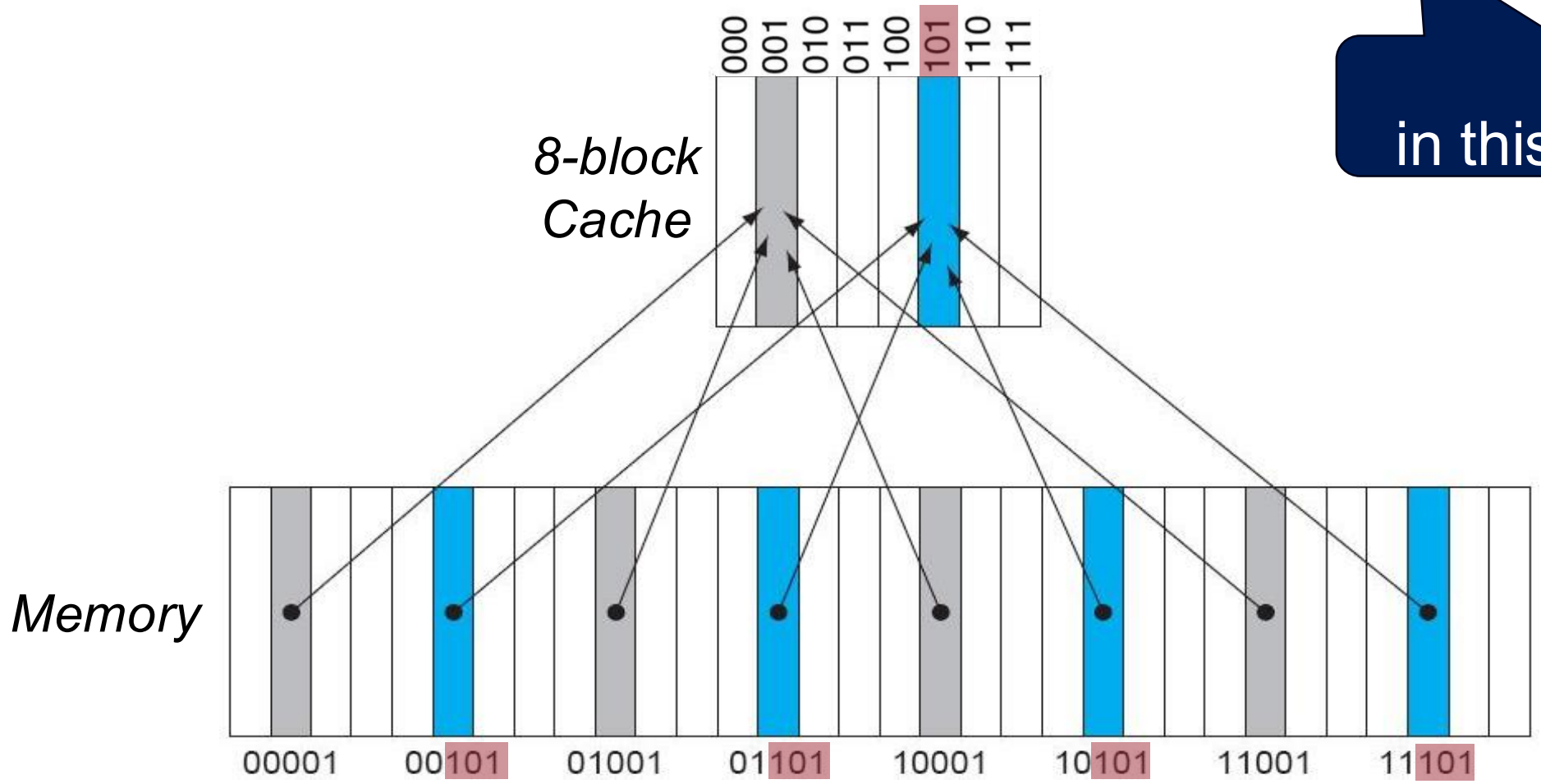
8
in this example



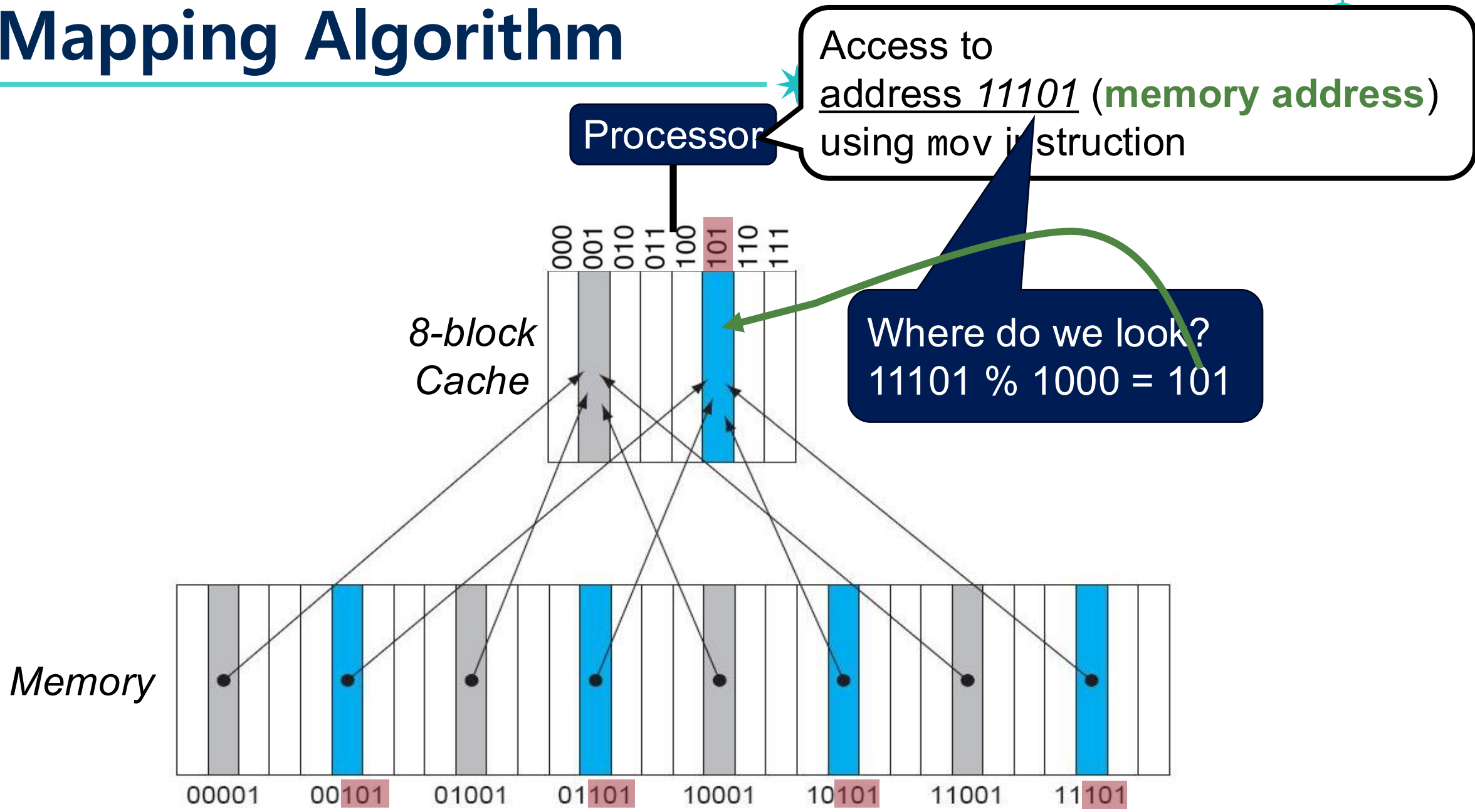
Mapping Algorithm: Binary View

Location in the cache = (**Block address**) modulo (**# Blocks in cache**)

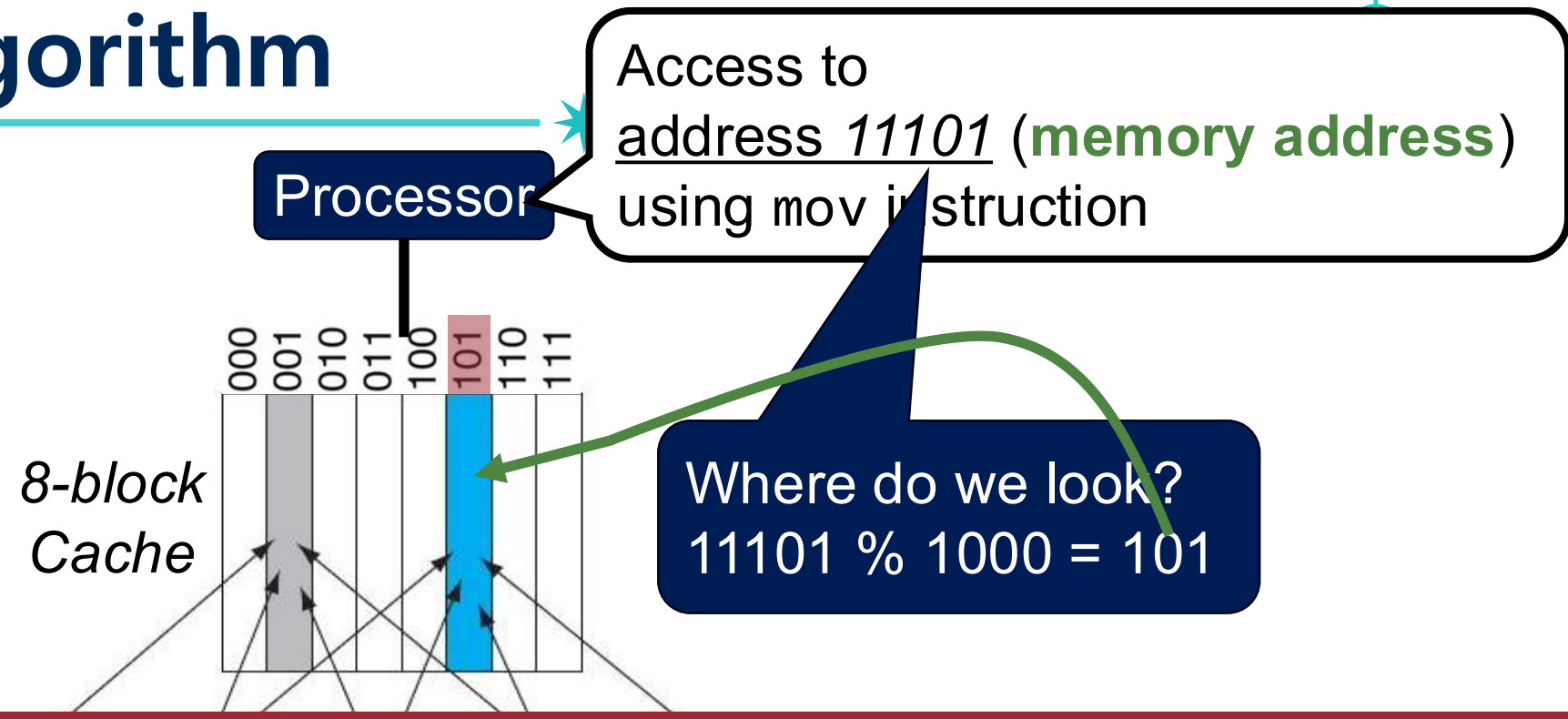
8
in this example



Mapping Algorithm



Mapping Algorithm

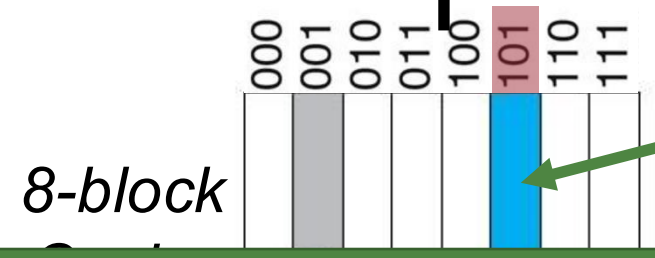


The value might not be from 11101 but from another location (e.g., 00101).
How can you confirm that it's the value from 11101?

Mapping Algorithm

Access to address *11101* (**memory address**) using `mov` instruction

Processor



Where do we look?
11101 % 1000 = 101

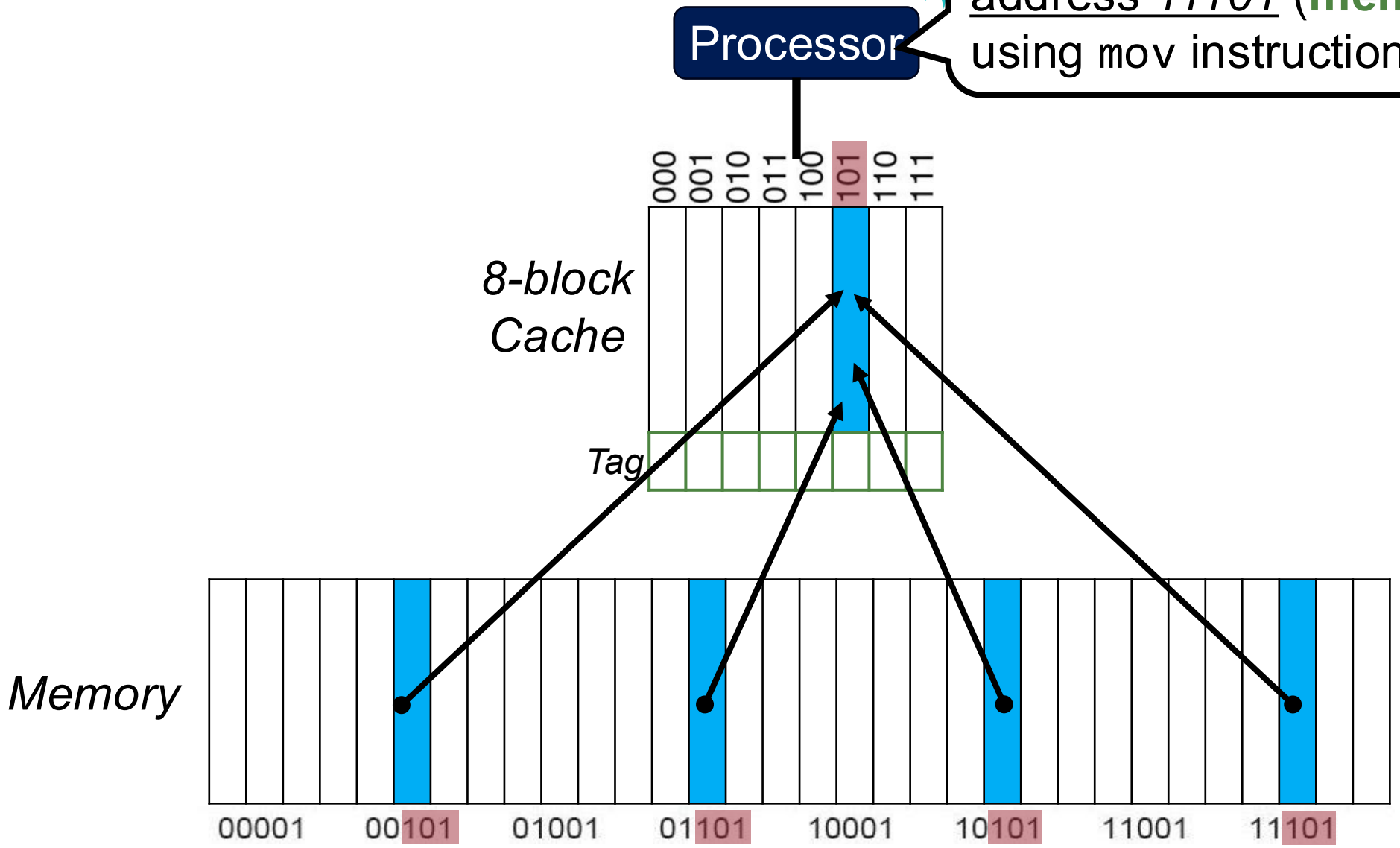
Idea: store the *high-order bits* in the cache to specify **the exact location**

The value might not be from 11101 but from another location (e.g., 00101).

How can you confirm that it's the value from 11101?

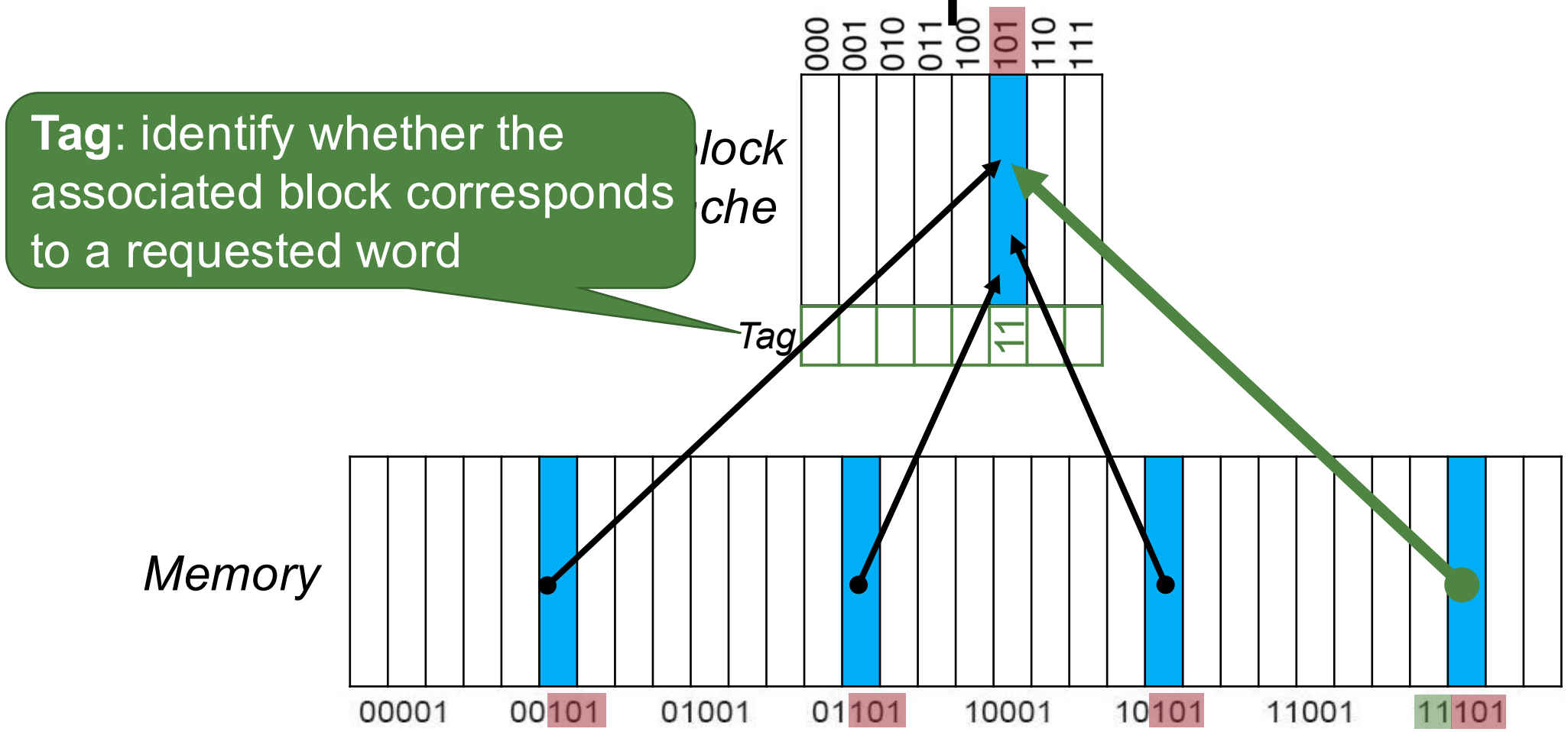
Additional Field: Tag Bits

Access to address *11101* (**memory address**) using `mov` instruction



Additional Field: Tag Bits

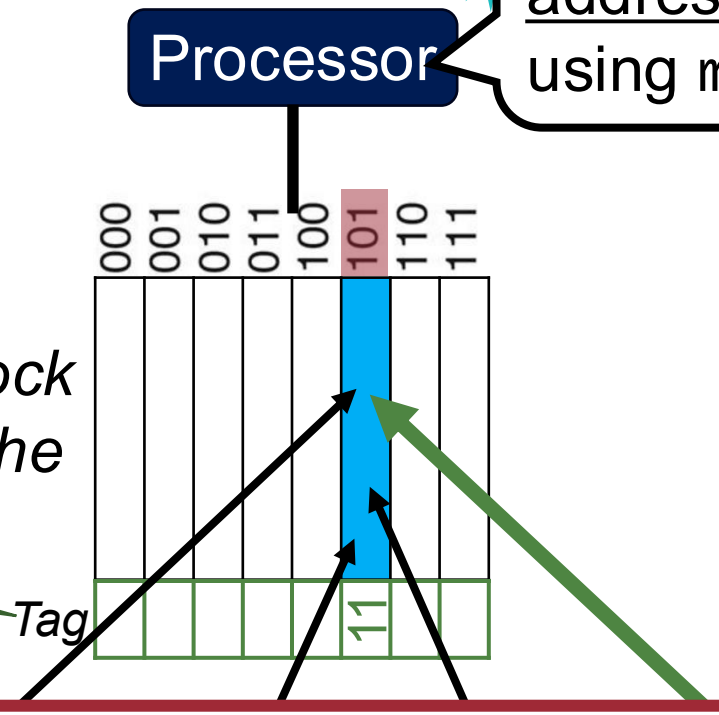
Access to address *11101* (**memory address**) using `mov` instruction



Tag: identify whether the associated block corresponds to a requested word

Additional Field: Tag Bits

Access to address *11101* (**memory address**) using `mov` instruction

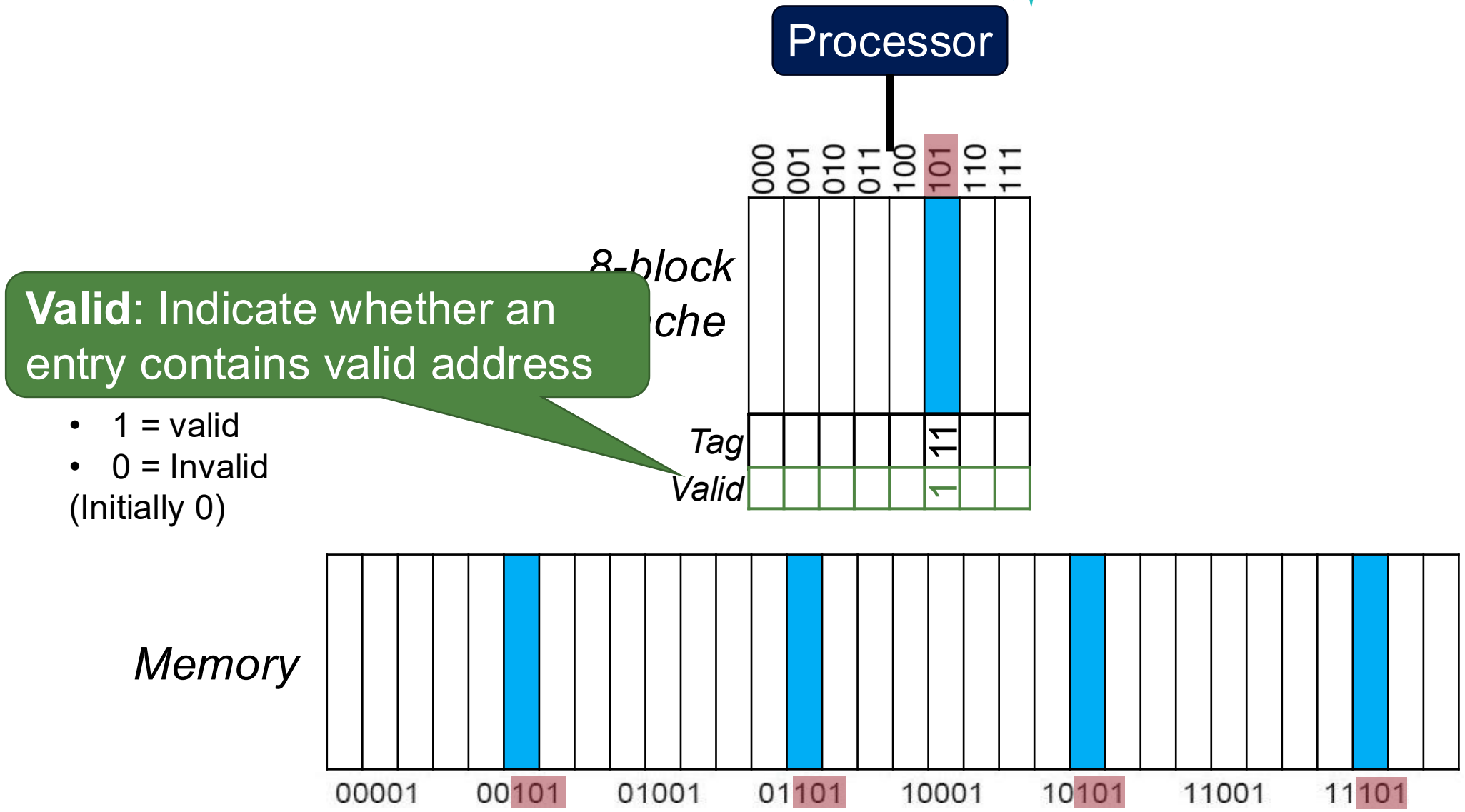


Tag: identify whether the associated block corresponds to a requested word

How do you know if the data being accessed is *valid or not*?

(When a processor starts up, the cache does not have valid data)

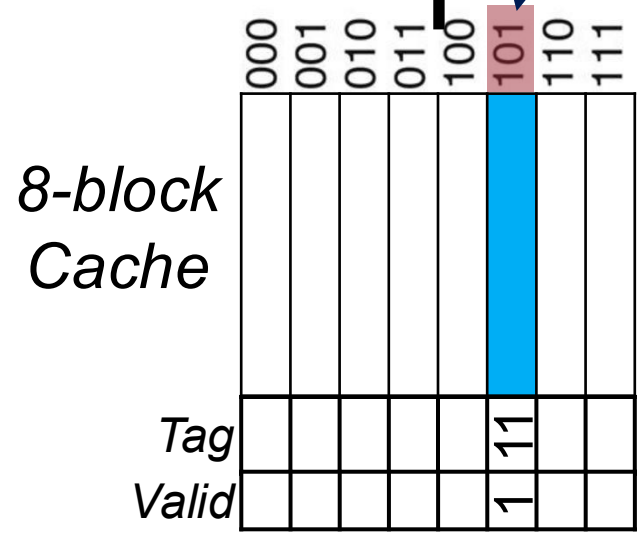
Additional Field: Valid Bit



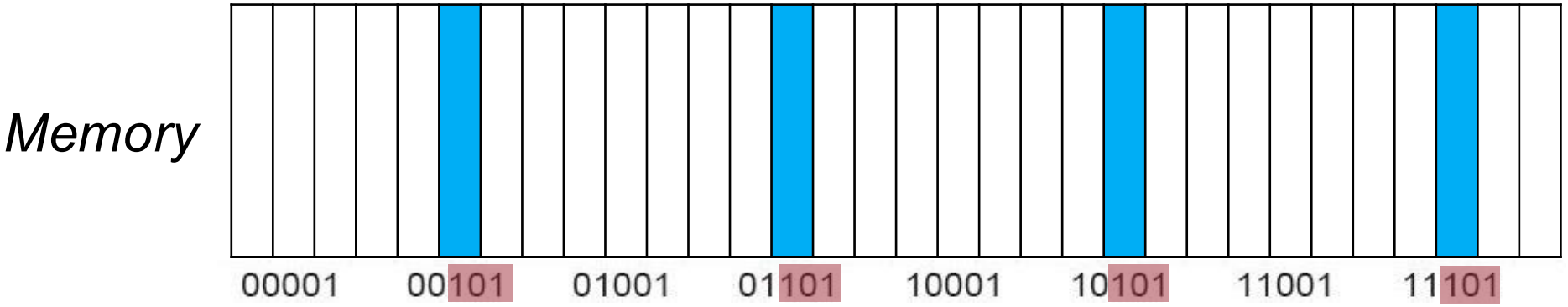
Lookup Workflow

Access to address *11101* (**memory address**) using mov instruction

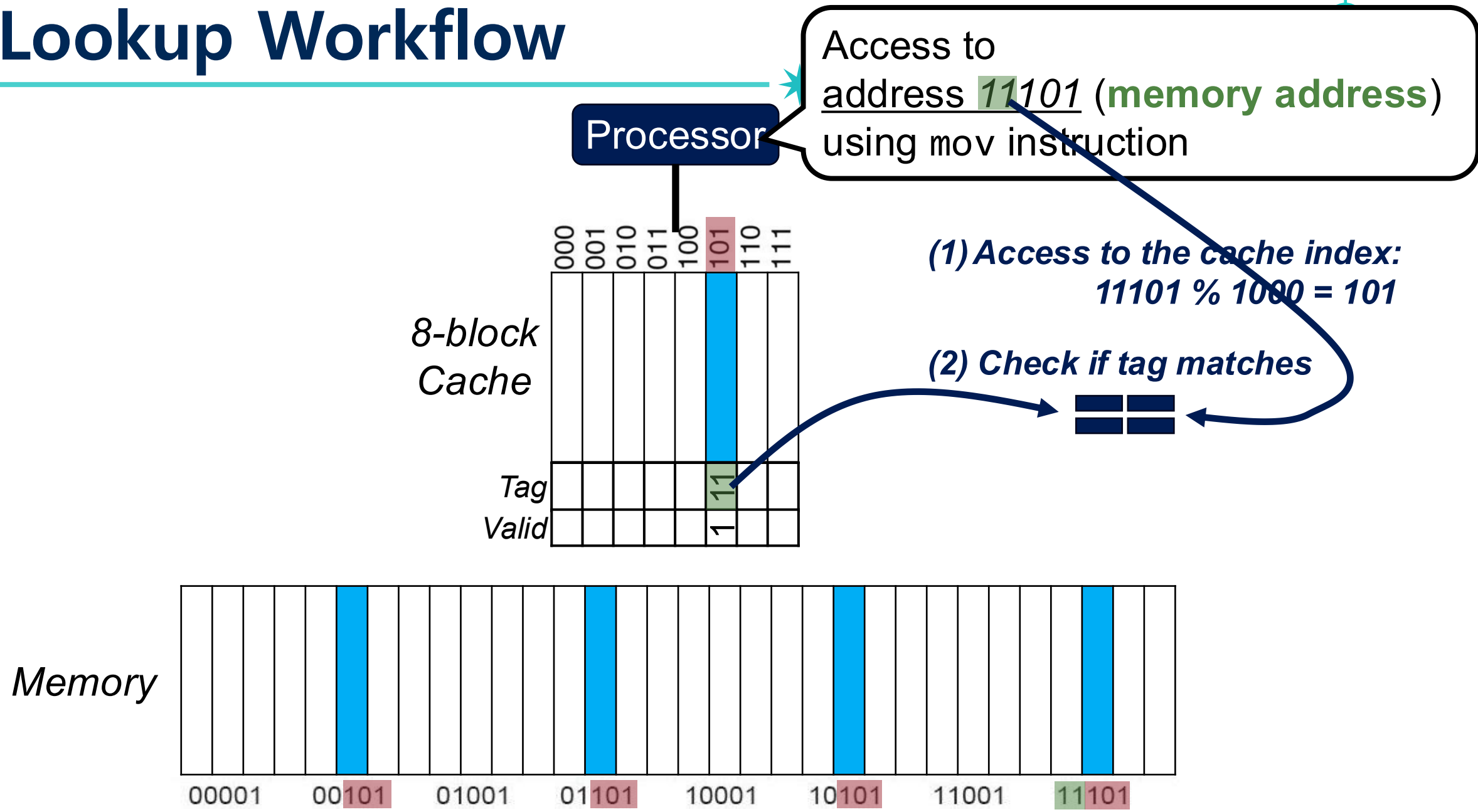
Processor



(1) Access to the cache index: 11101 % 1000 = 101



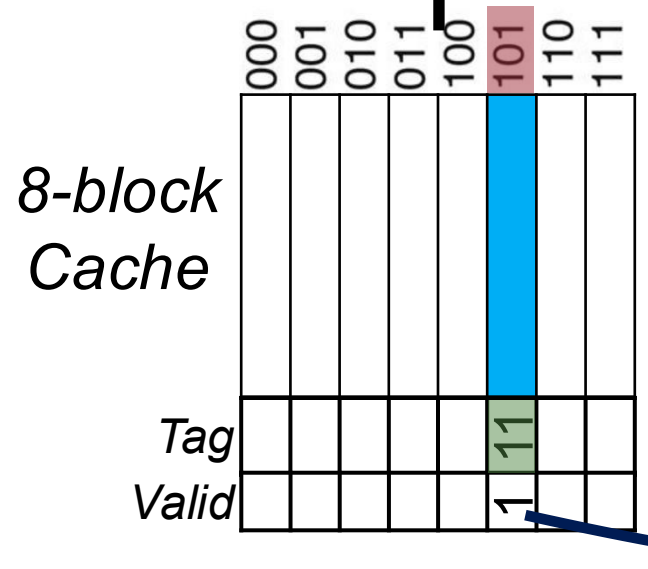
Lookup Workflow



Lookup Workflow

Access to address *11101* (**memory address**) using `mov` instruction

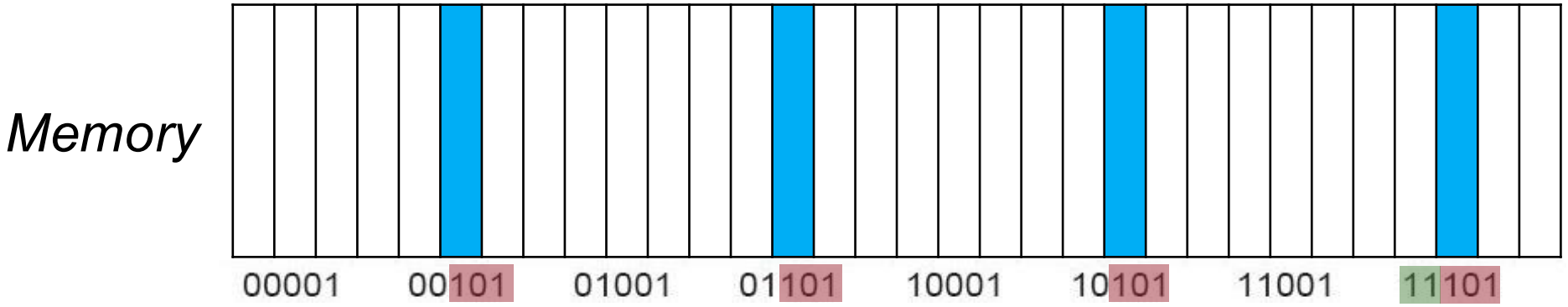
Processor



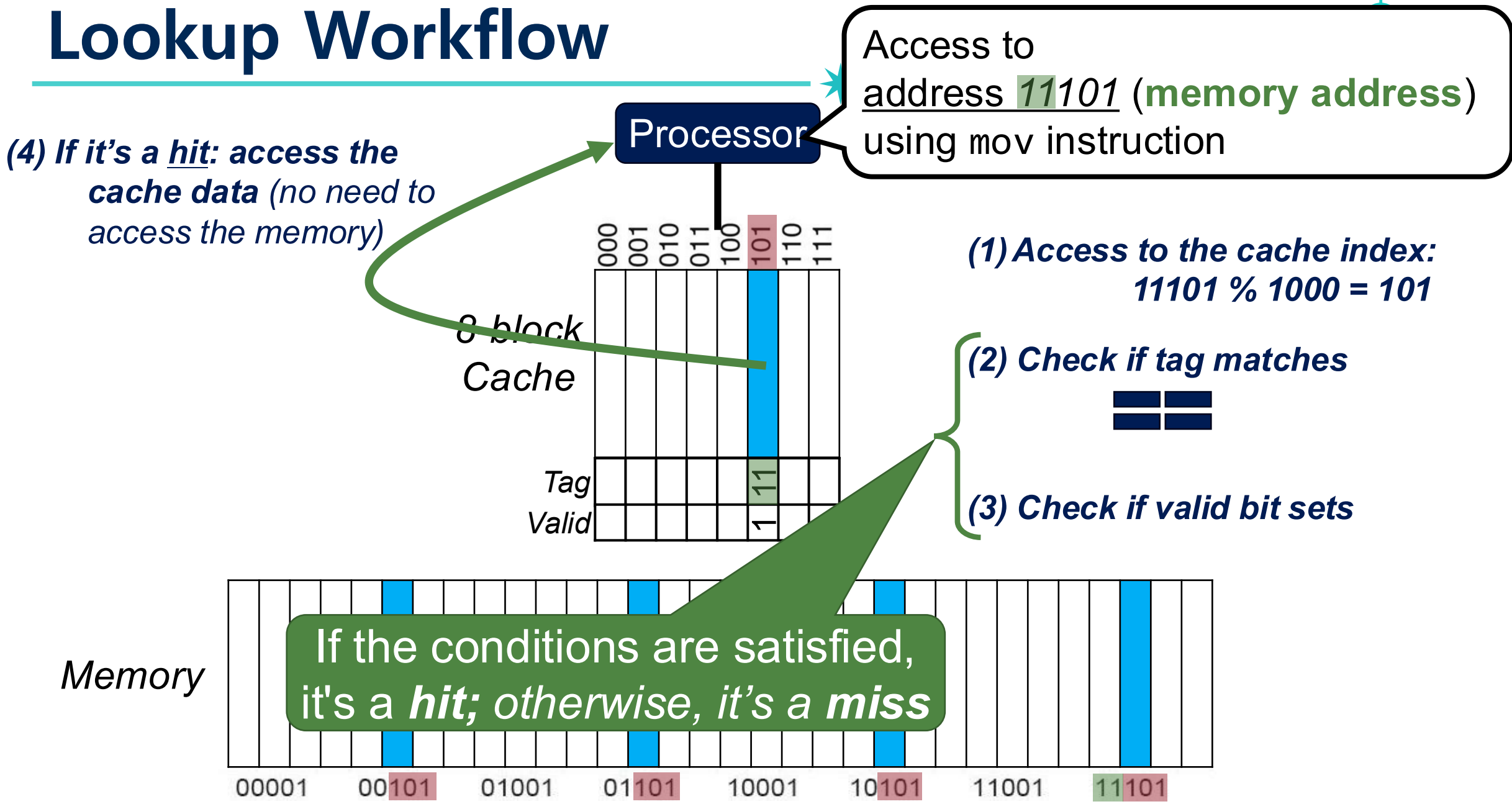
(1) Access to the cache index:
11101 % 1000 = 101

(2) Check if tag matches

(3) Check if valid bit sets



Lookup Workflow



Direct Mapped Cache Structure

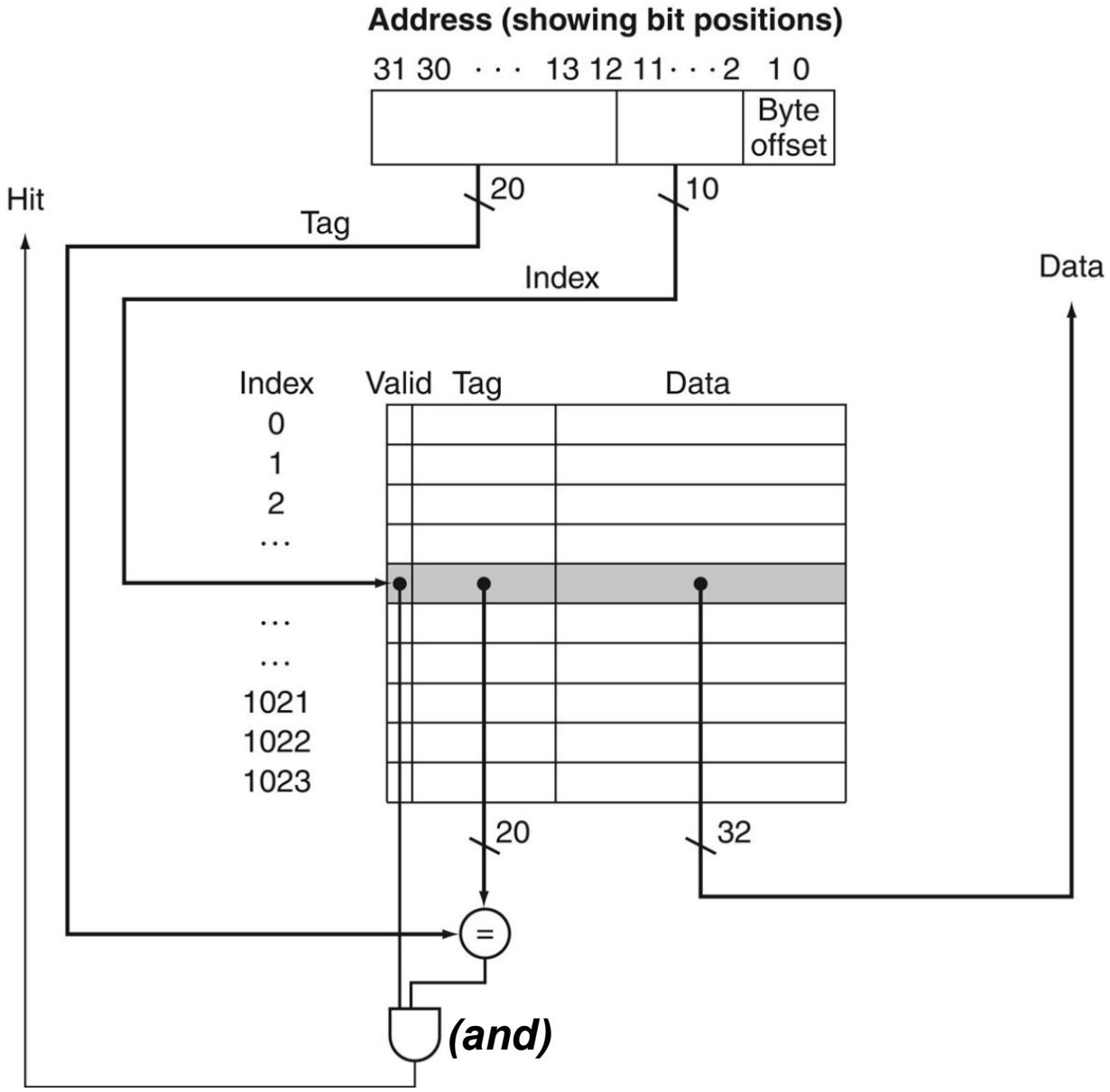
- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?

Direct Mapped Cache Structure

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

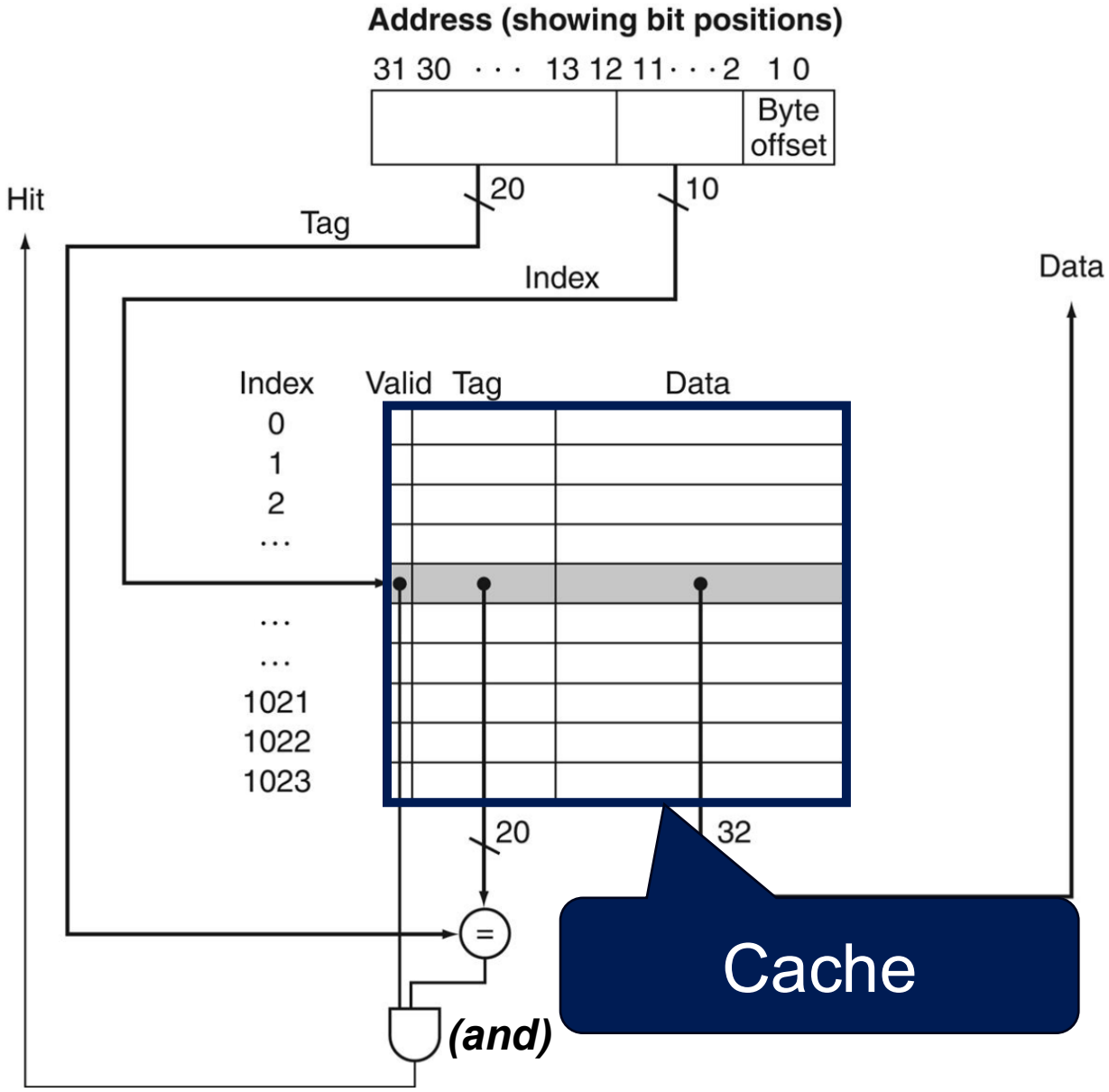
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

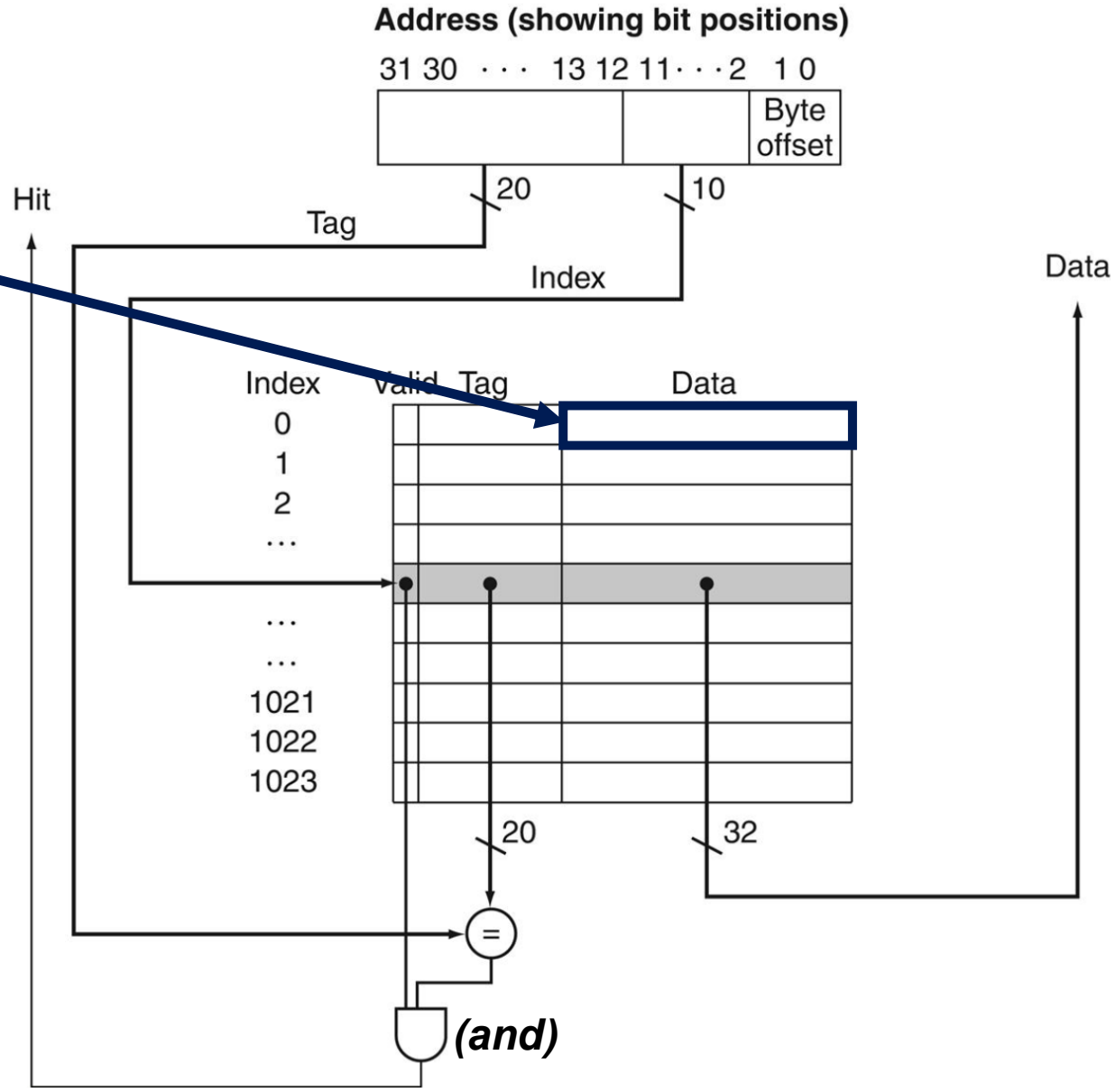
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

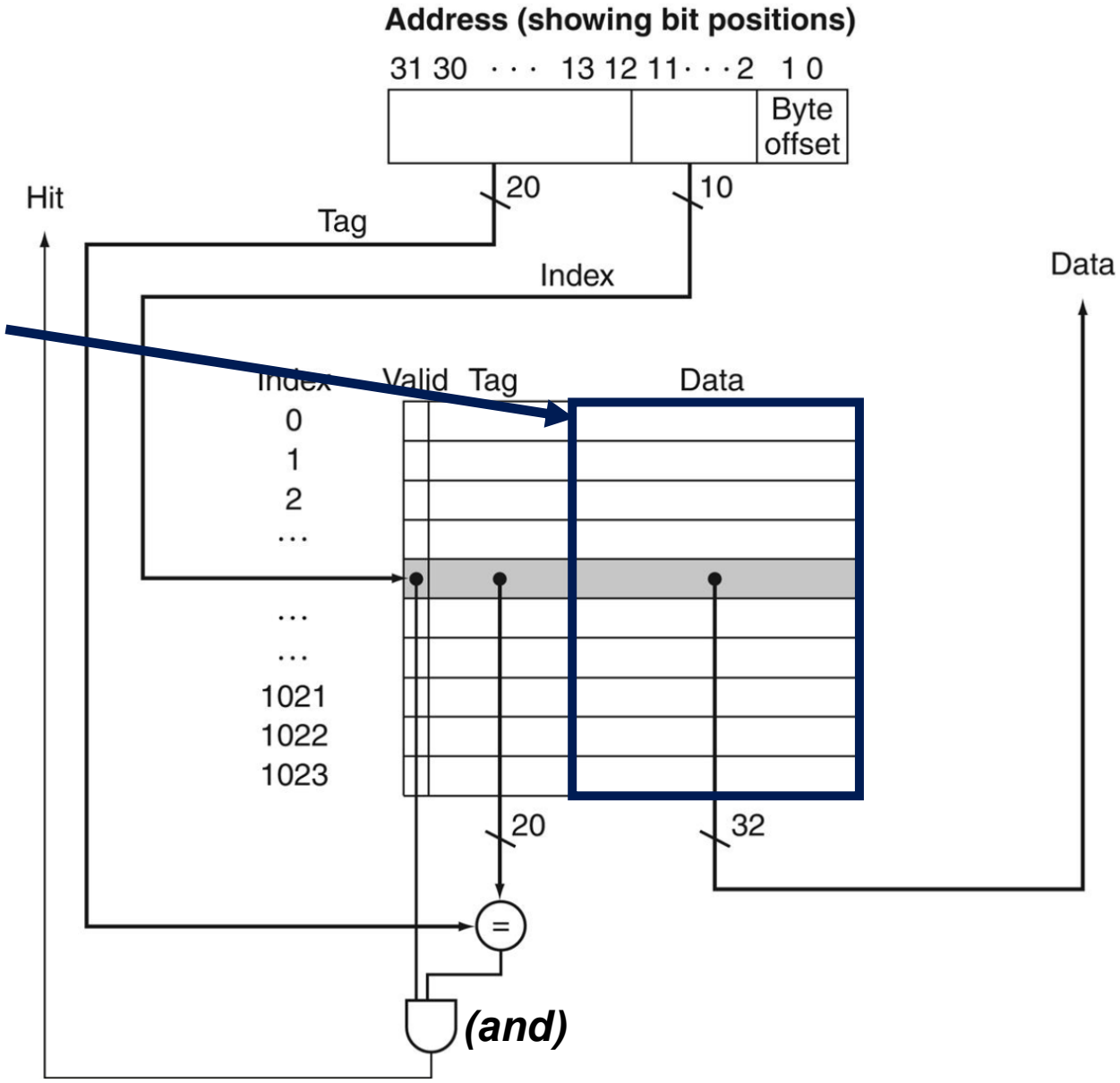
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

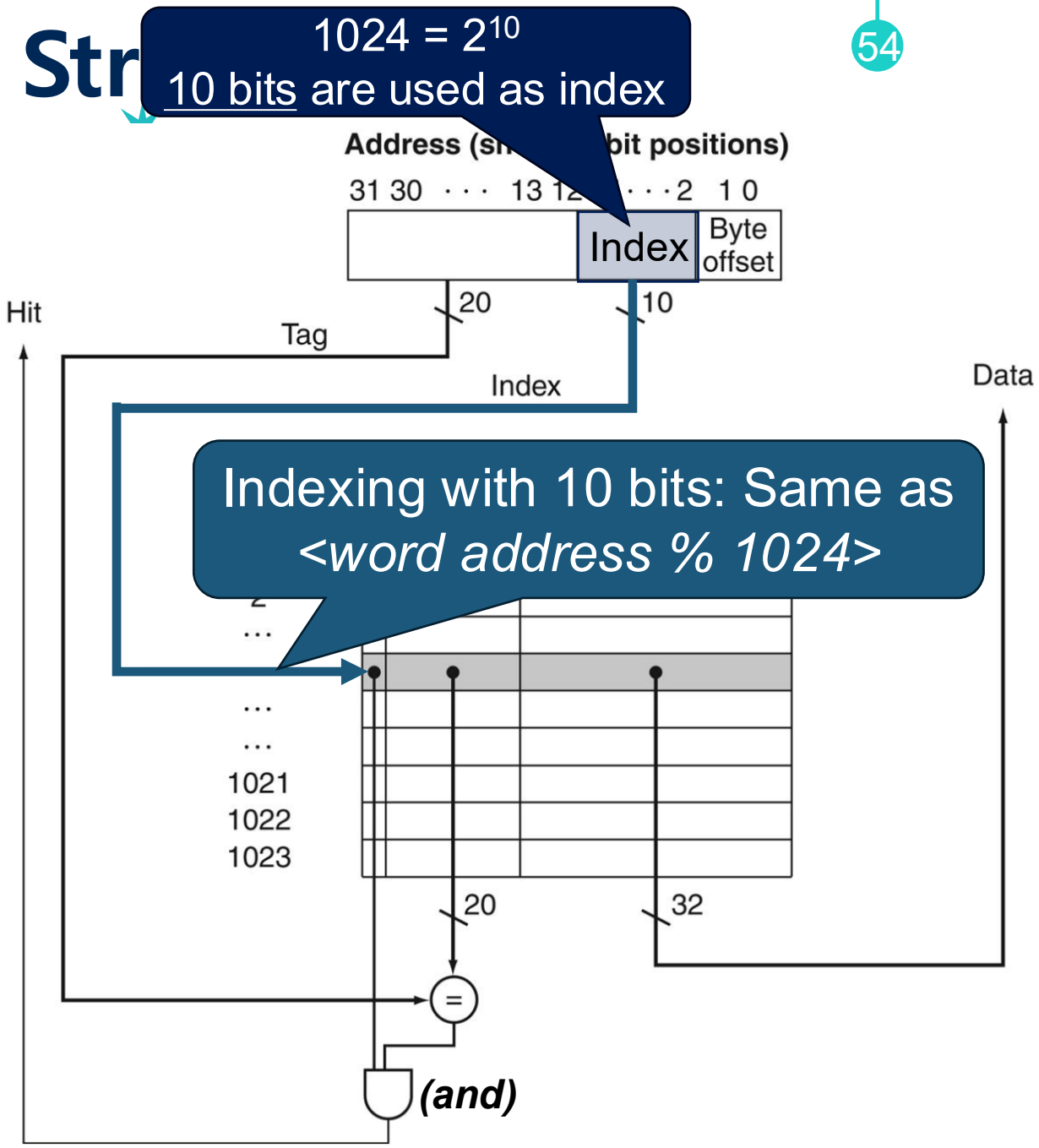
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Str

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

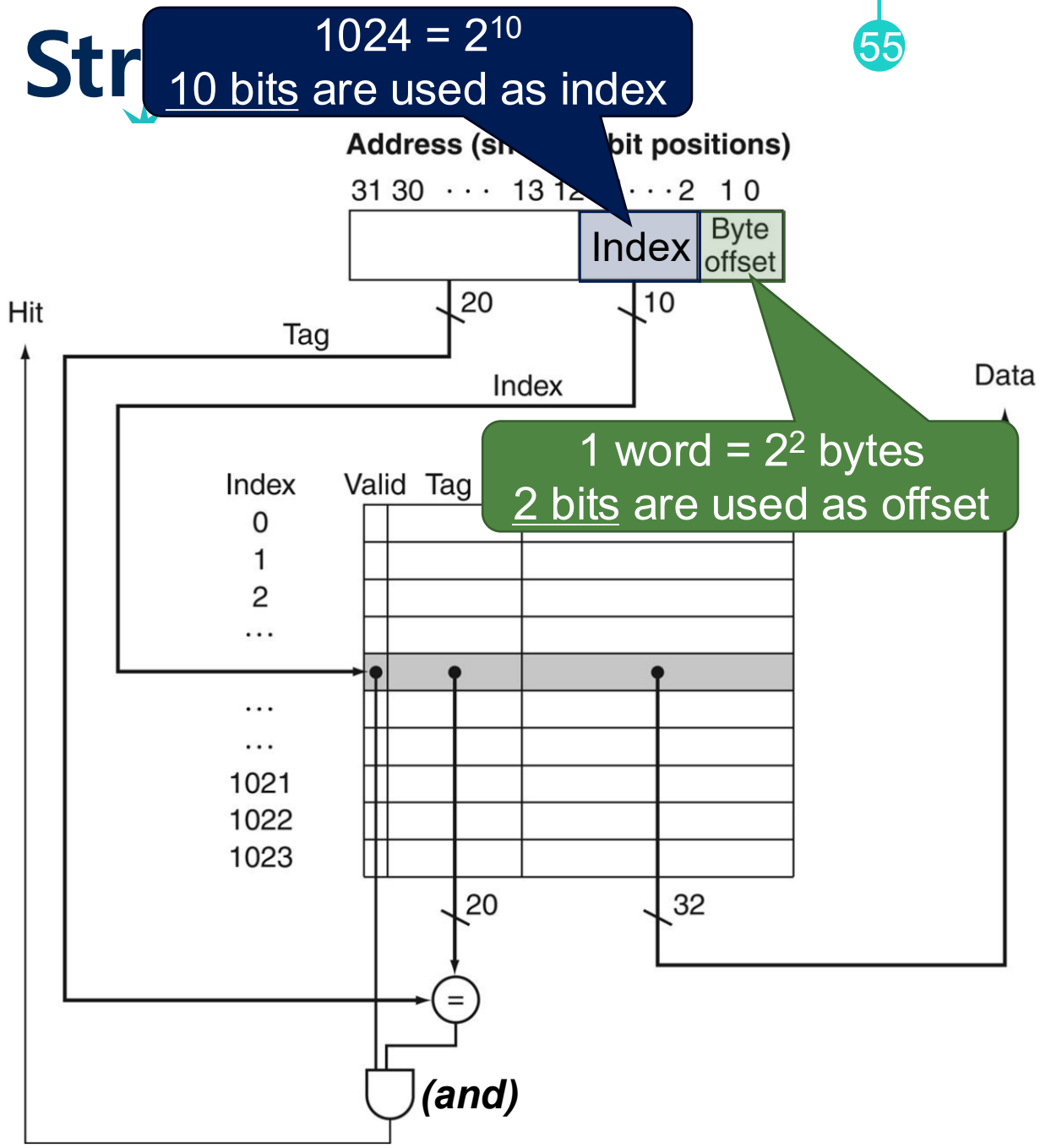
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Str

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

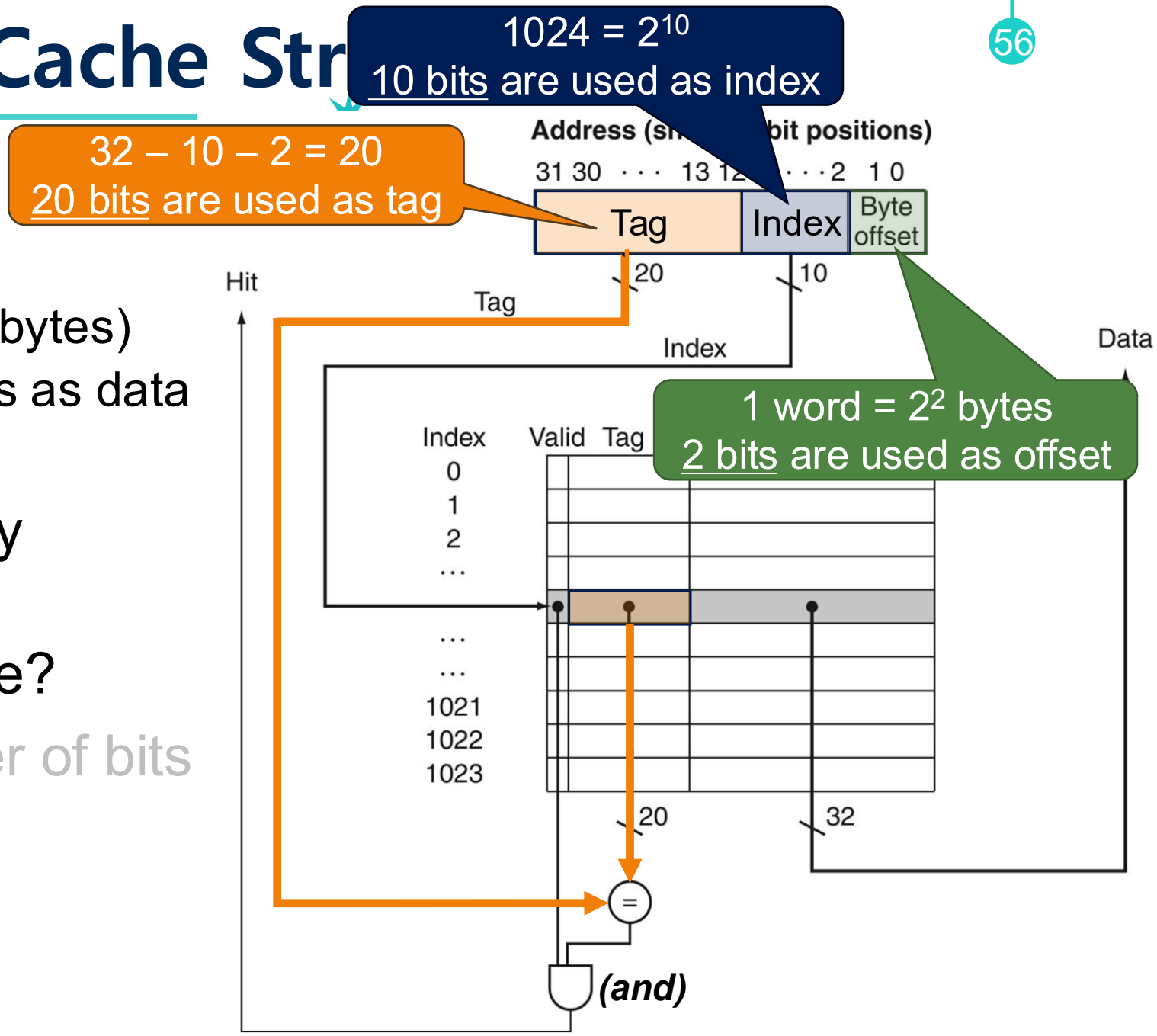
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Str

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

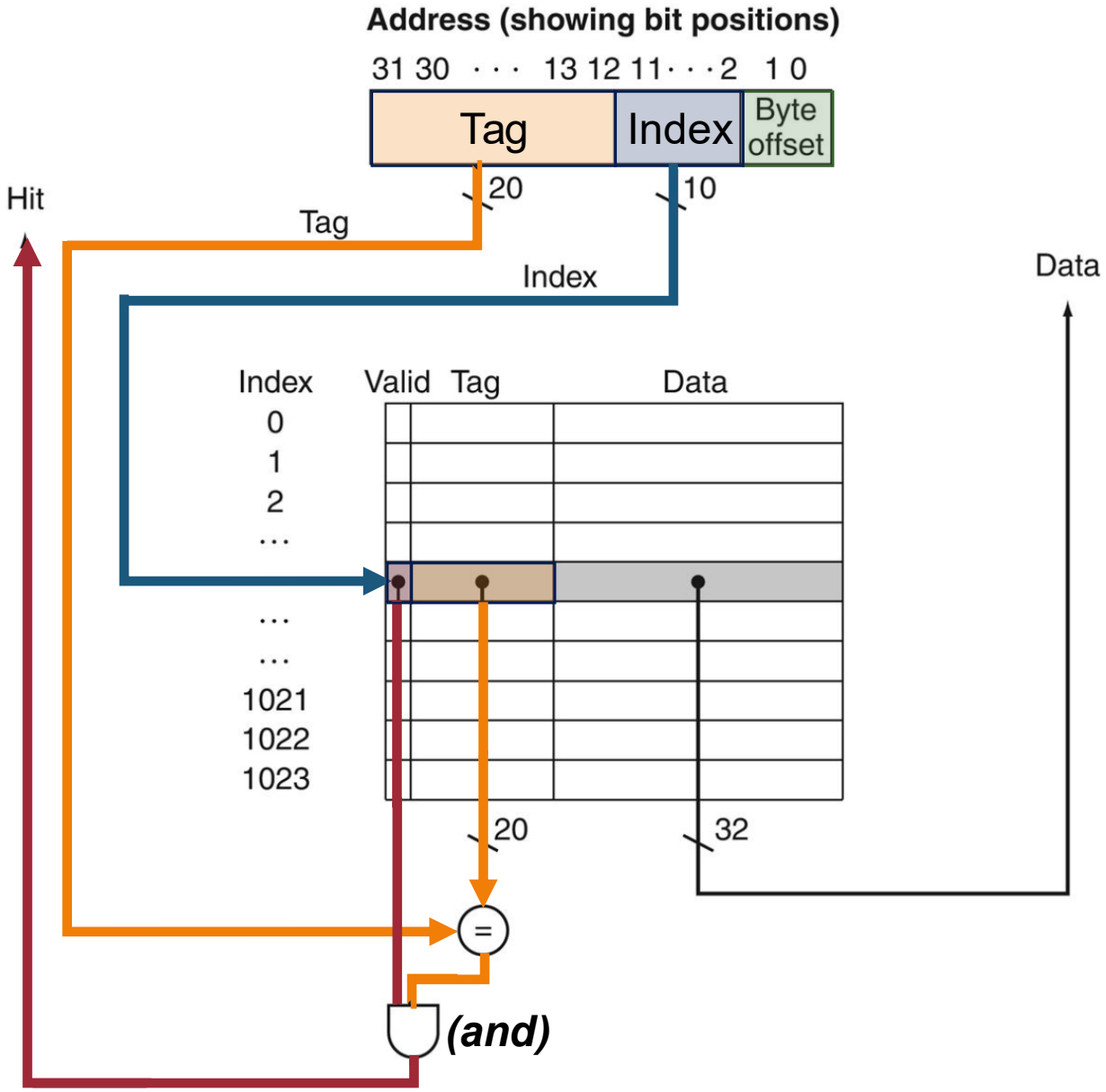
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?

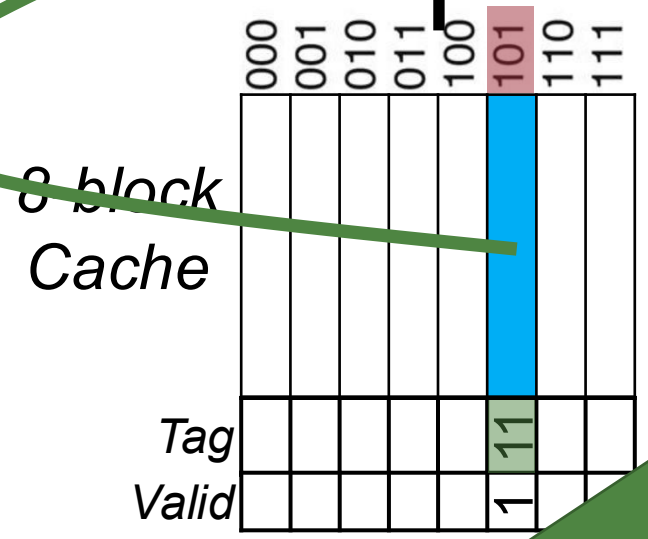


Recap: Hit

Access to address *11101* (**memory address**) using `lw` instruction

Processor

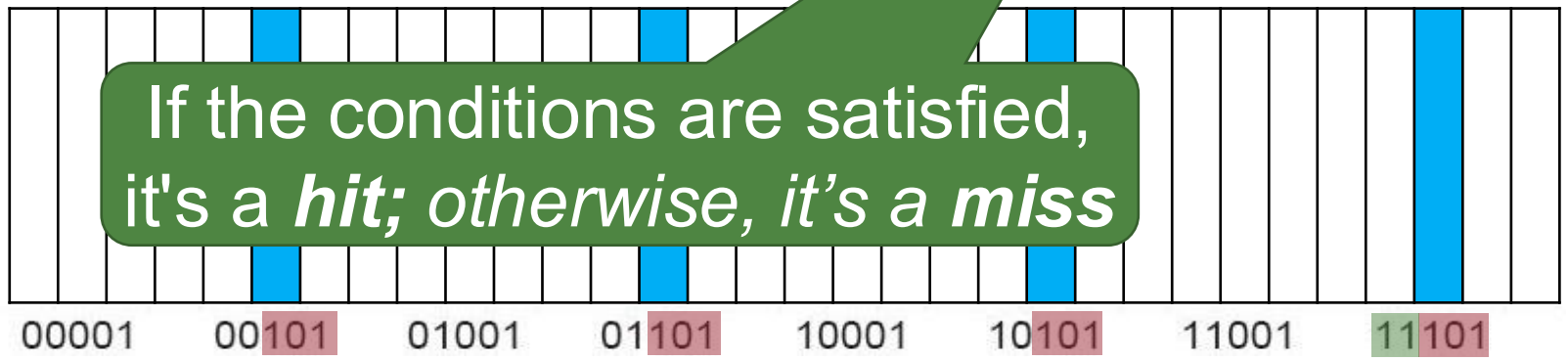
(4) If it's a hit: access the cache data *(no need to access the memory)*



(1) Access to the cache index: 11101 % 1000 = 101

(2) Check if tag matches

(3) Check if valid bit sets



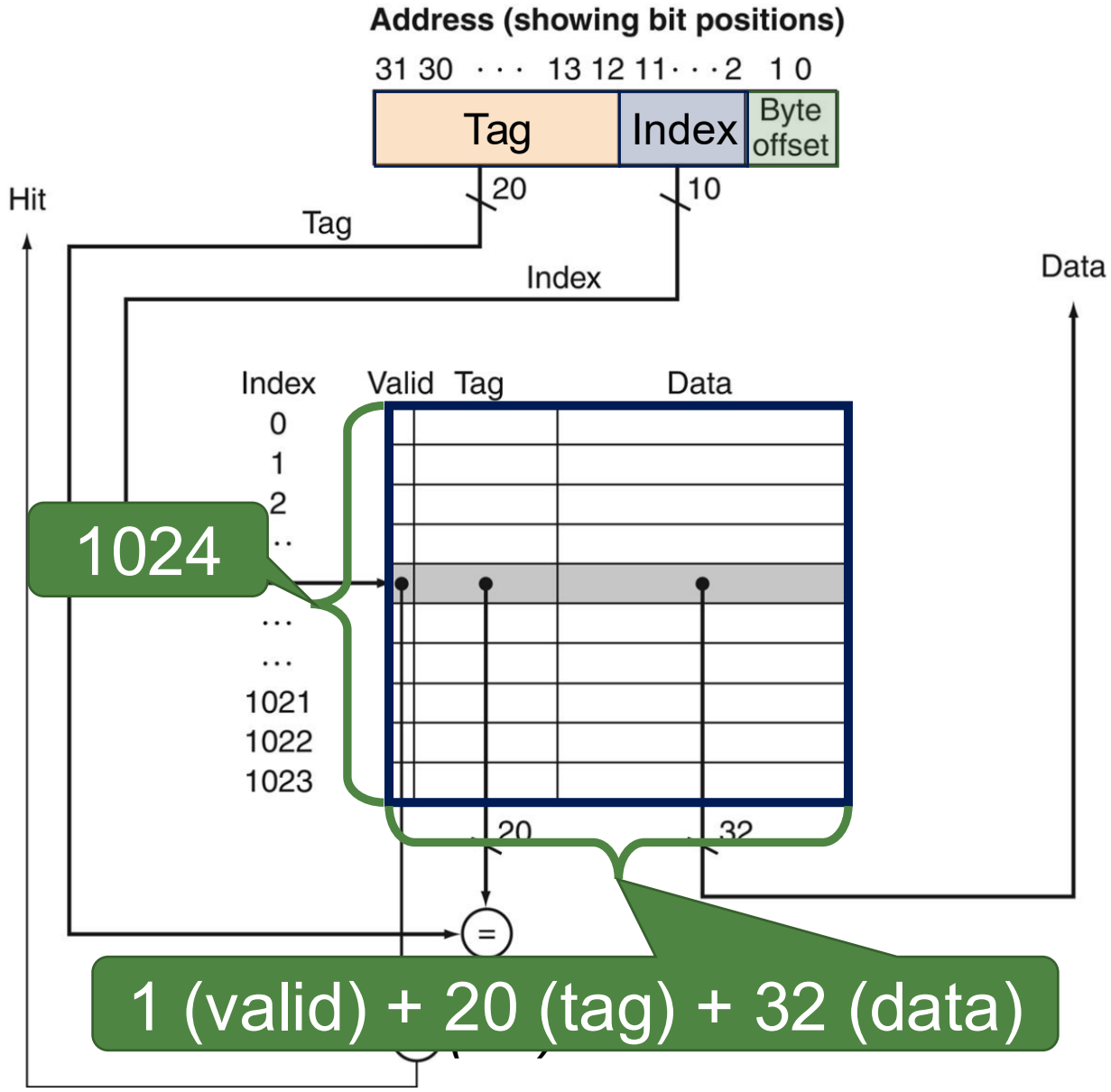
If the conditions are satisfied, it's a ***hit;*** *otherwise, it's a* ***miss***

Direct Mapped Cache Structure

- **Assumption**:
  - 1 word = 4 bytes
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?

1024 x (1+20+32) bits



Handling Cache Hits & Misses

Hits vs. Misses

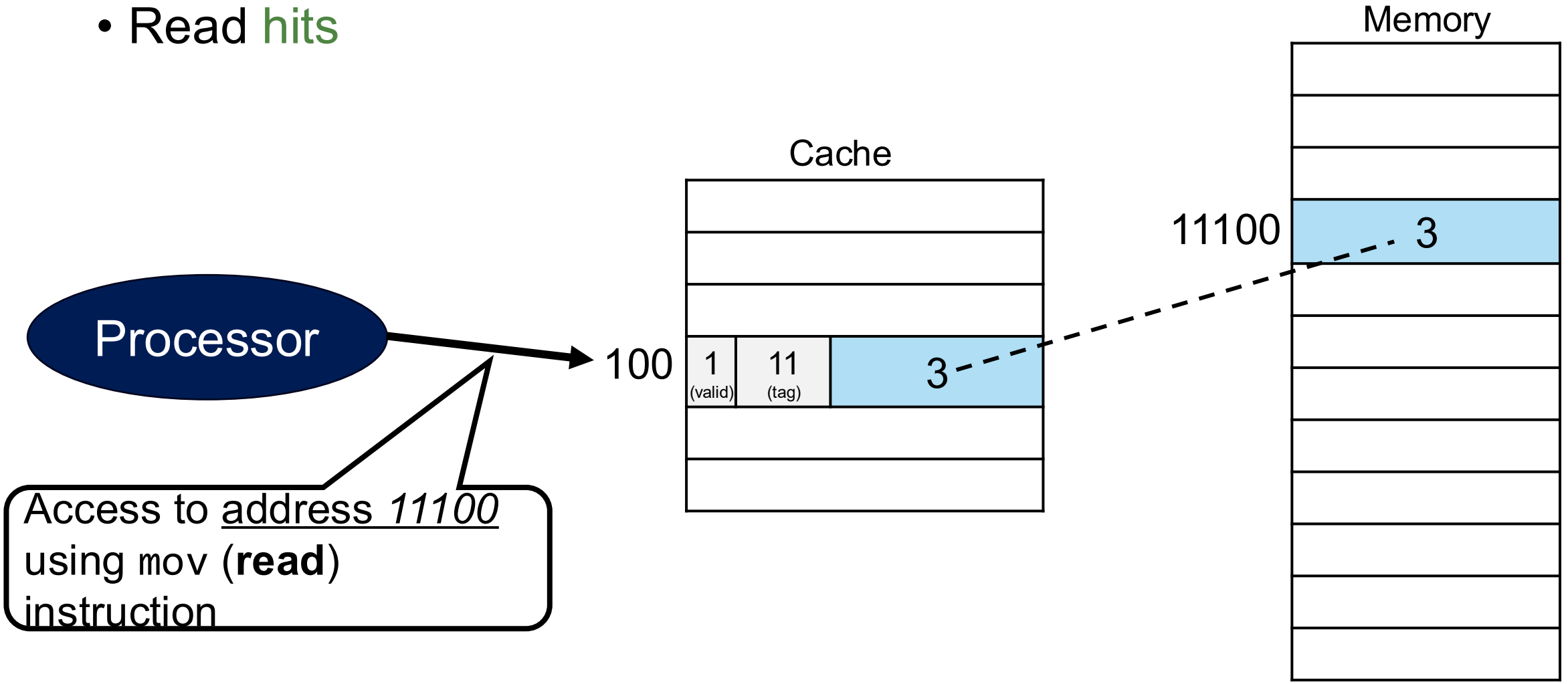
- Read hits
- Read misses
- Write hits
- Write misses

Hits vs. Misses

- Read hits
- Read misses
- Write hits
- Write misses

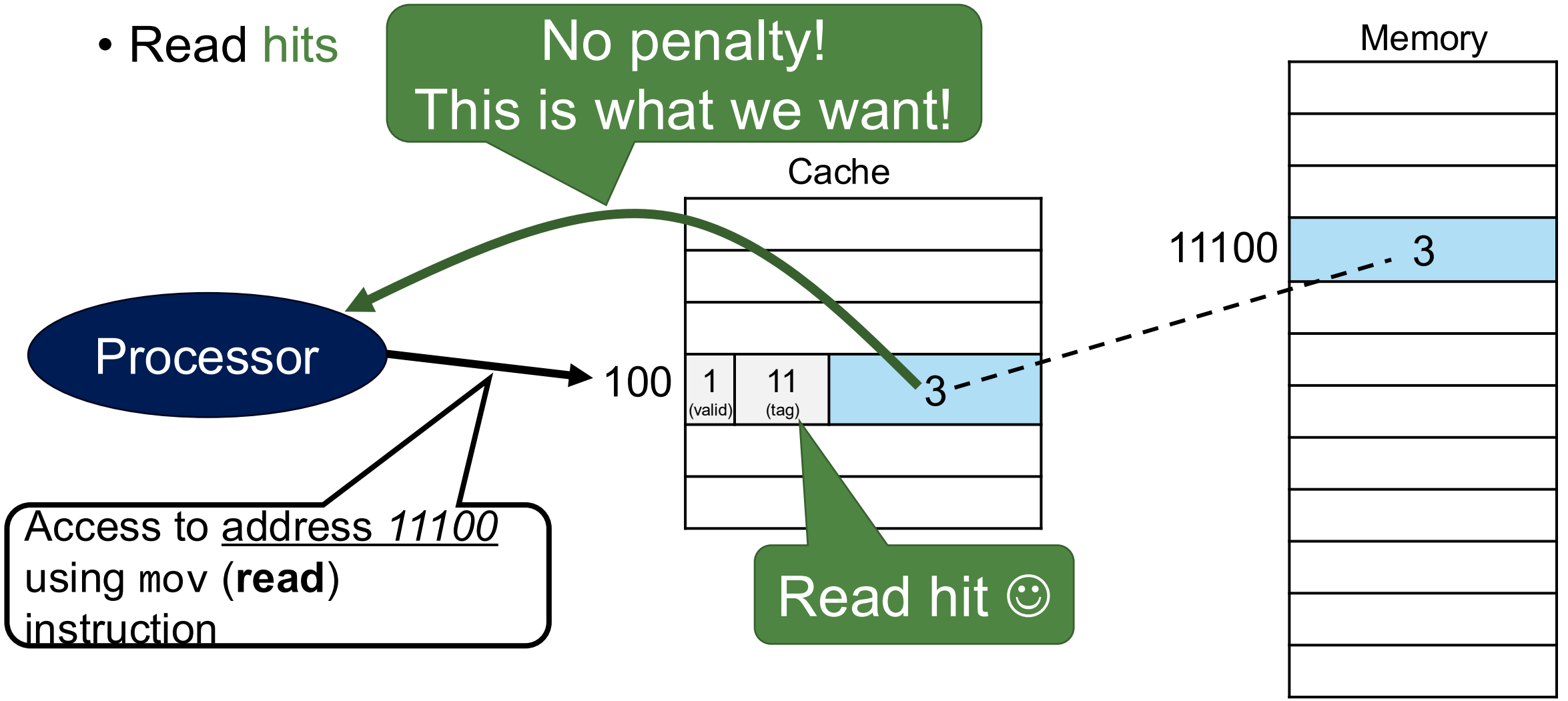
Hits vs. Misses: Read Hits

- Read hits



Hits vs. Misses: Read Hits

- Read hits



Summary: Hits vs. Misses

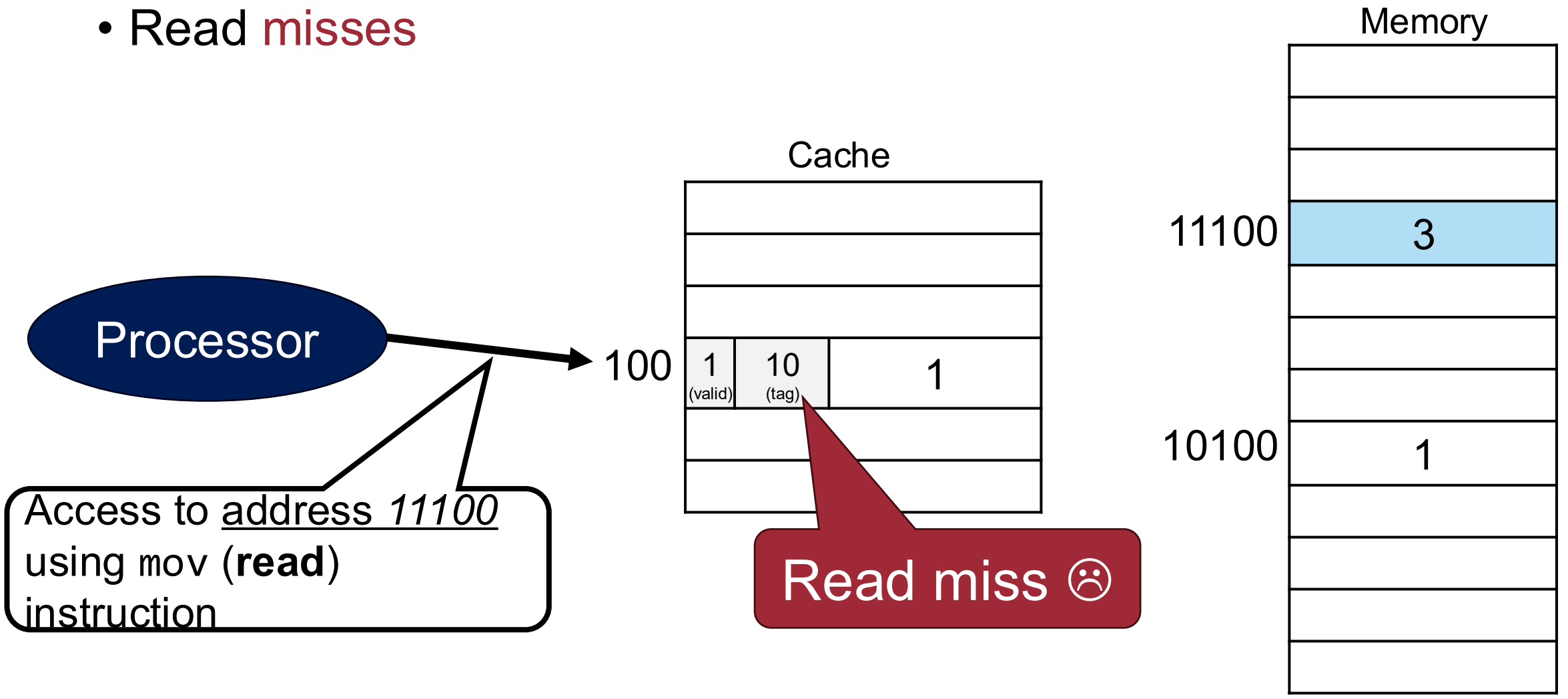
- Read hits
  - This is what we want!
- Read misses
- Write hits
- Write misses

Summary: Hits vs. Misses

- Read hits
  - This is what we want!
- Read misses
- Write hits
- Write misses

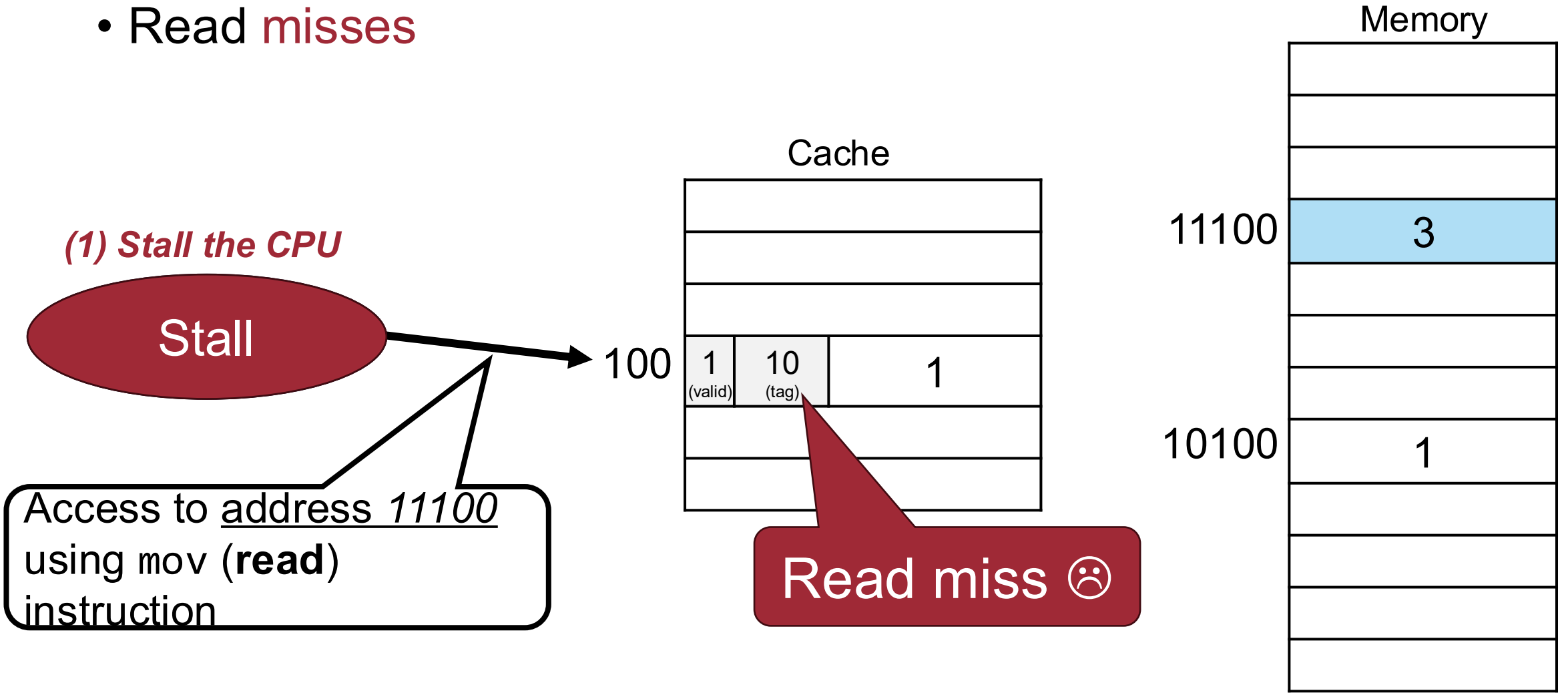
Hits vs. Misses: Read Misses

- Read misses



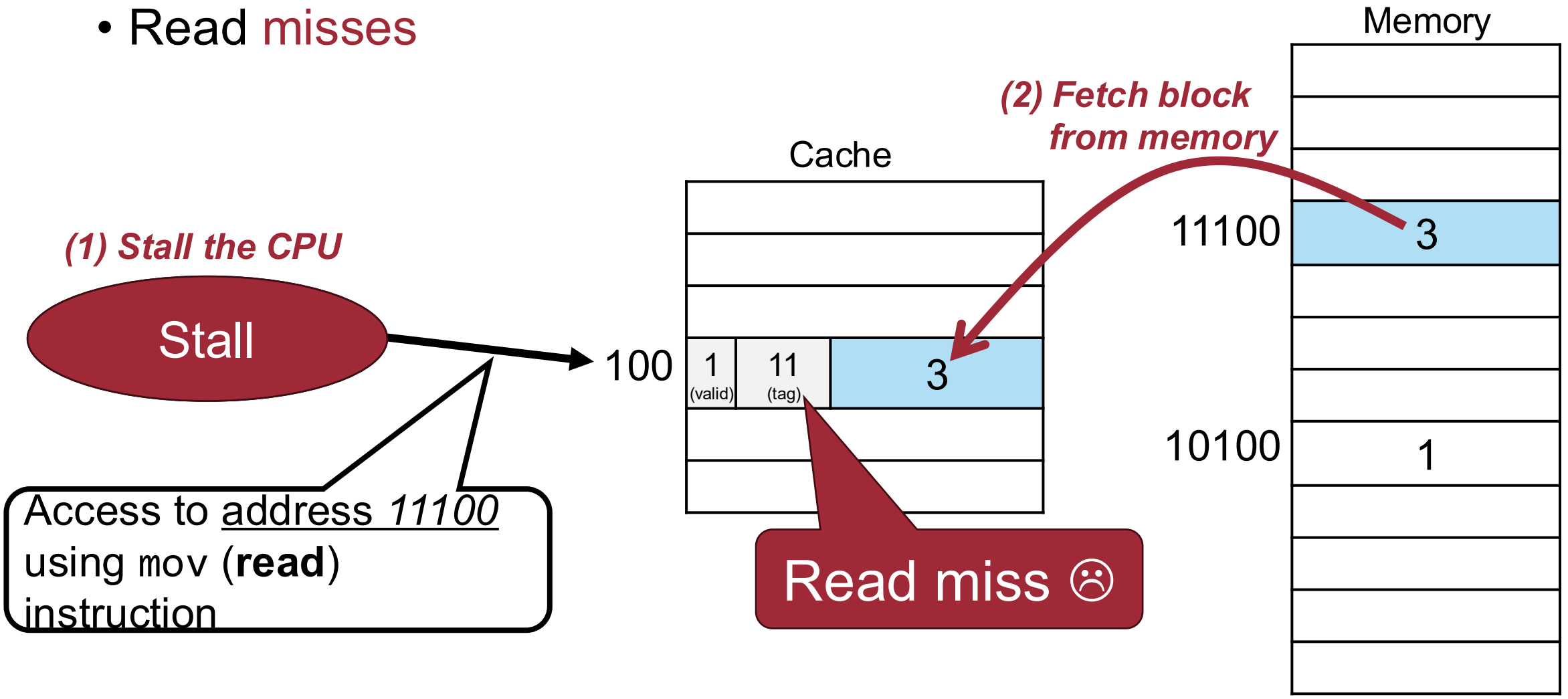
Hits vs. Misses: Read Misses

- Read misses



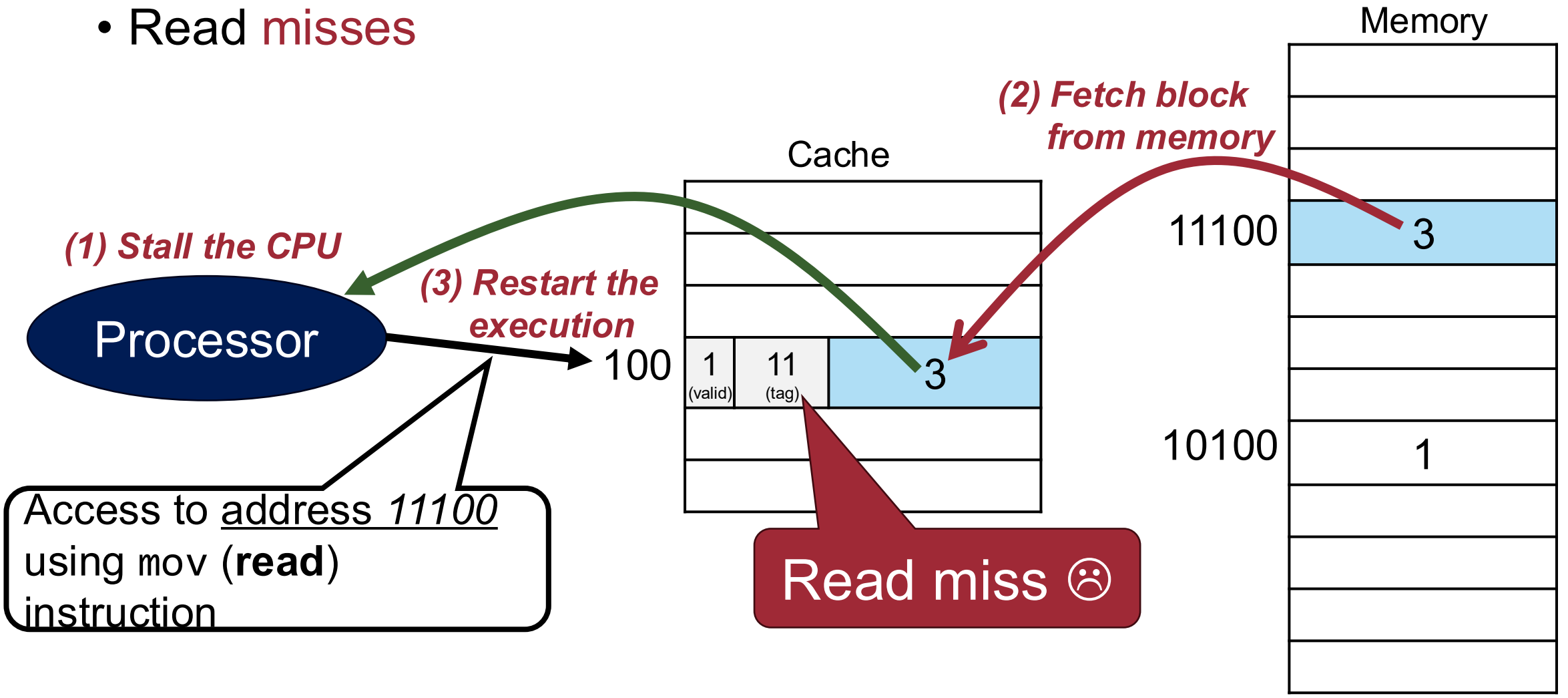
Hits vs. Misses: Read Misses

- Read misses



Hits vs. Misses: Read Misses

- Read misses



Hits vs. Misses: Read Misses

- Read misses

(1) **CPU stalls**: freezing the contents of all the registers while waiting for memory

(2) **Fetch block** from memory

(3) **Restart** the execution

Summary: Hits vs. Misses

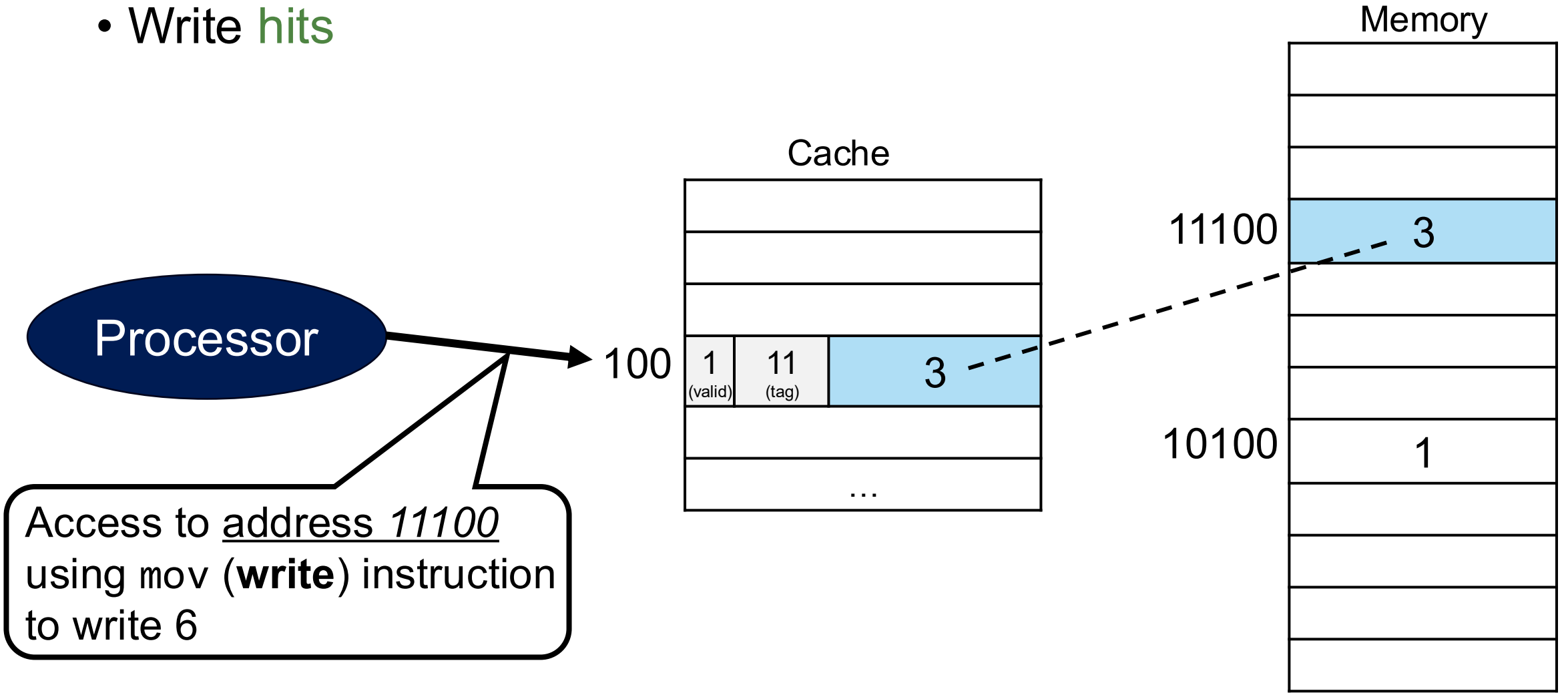
- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch block from memory, restart
- Write hits
- Write misses

Summary: Hits vs. Misses

- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch block from memory, restart
- Write hits
- Write misses

Hits vs. Misses: Write Hits

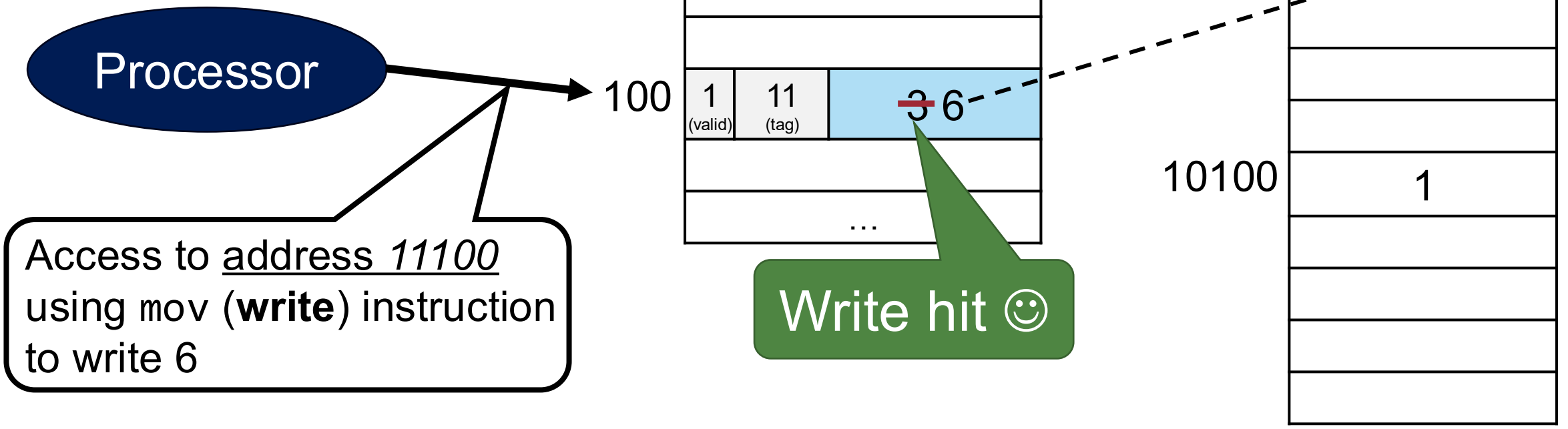
- Write hits



Hits vs. Misses: Write Hits

- Write hits

Problems?



Hits vs. Misses: Write Hits

- Write hits

Problems?

Memory

11100

3

Cache

100

1 (valid)

11 (tag)

~~3~~ 6

...

Processor

Access to address *11100* using `mov` (**write**) instruction to write 6

Write hit ☺

Cache and memory would be *inconsistent*

Write Policies

- Write hits
  - Cache and memory would be inconsistent! What can we do?

(1) Write through

(2) Write back

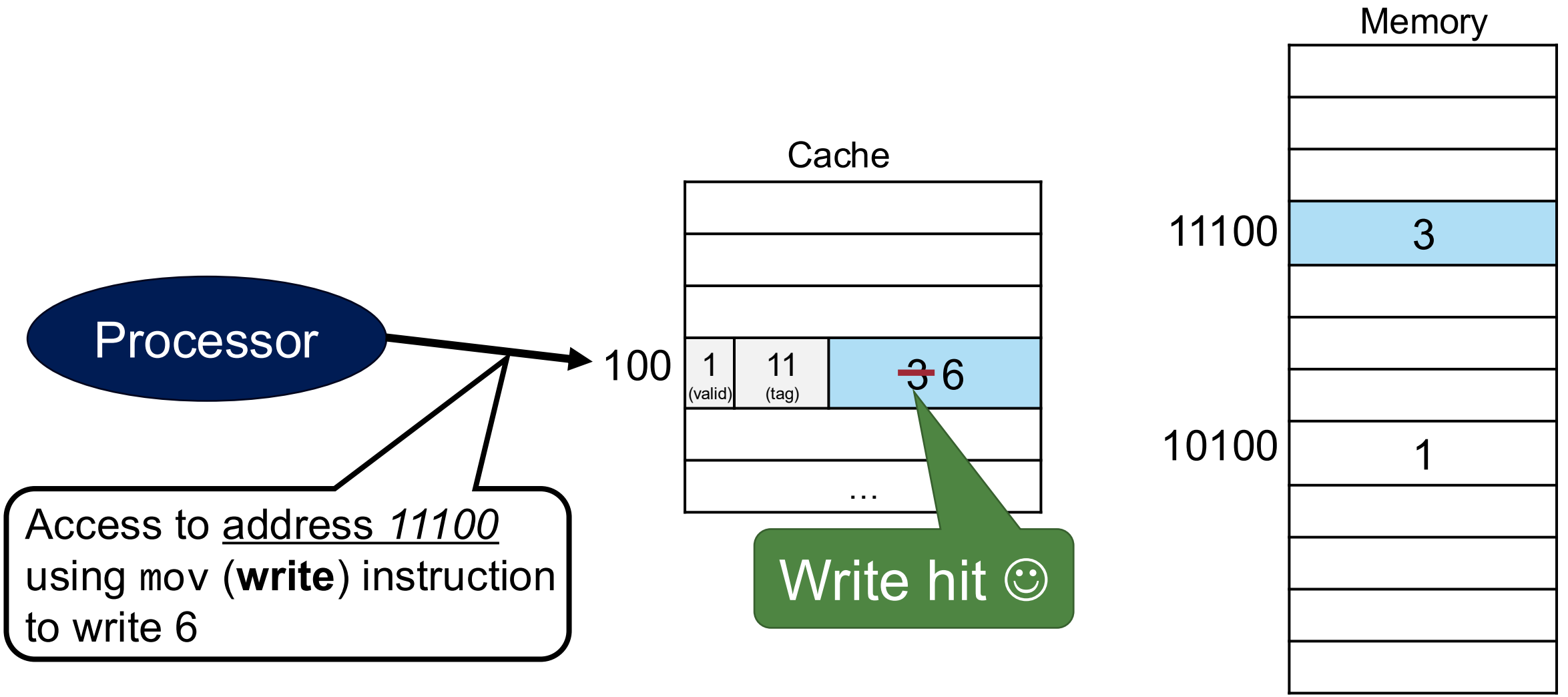
Write Policies

- Write hits
  - Cache and memory would be inconsistent! What can we do?

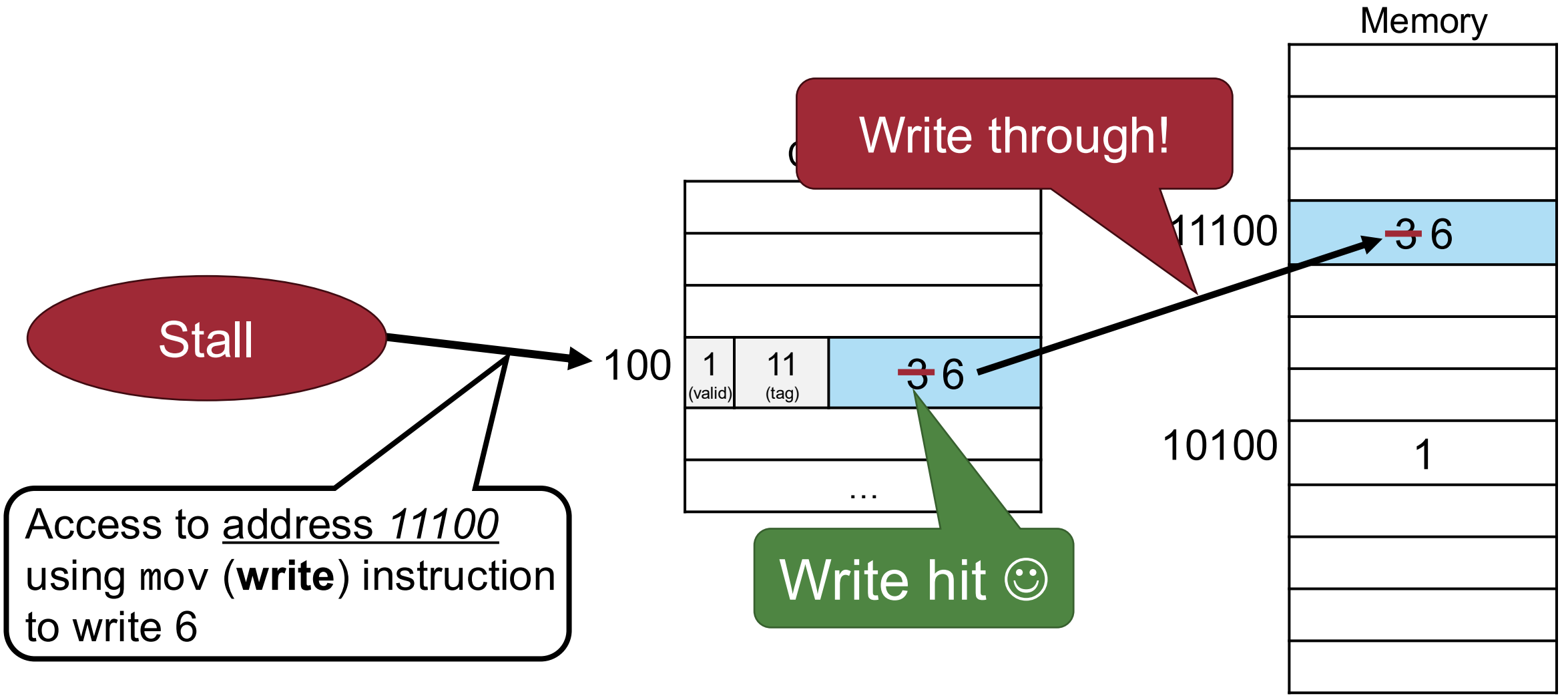
  (1) Write through: On each write hit, the information is written to both in the cache and in the memory

  (2) Write back

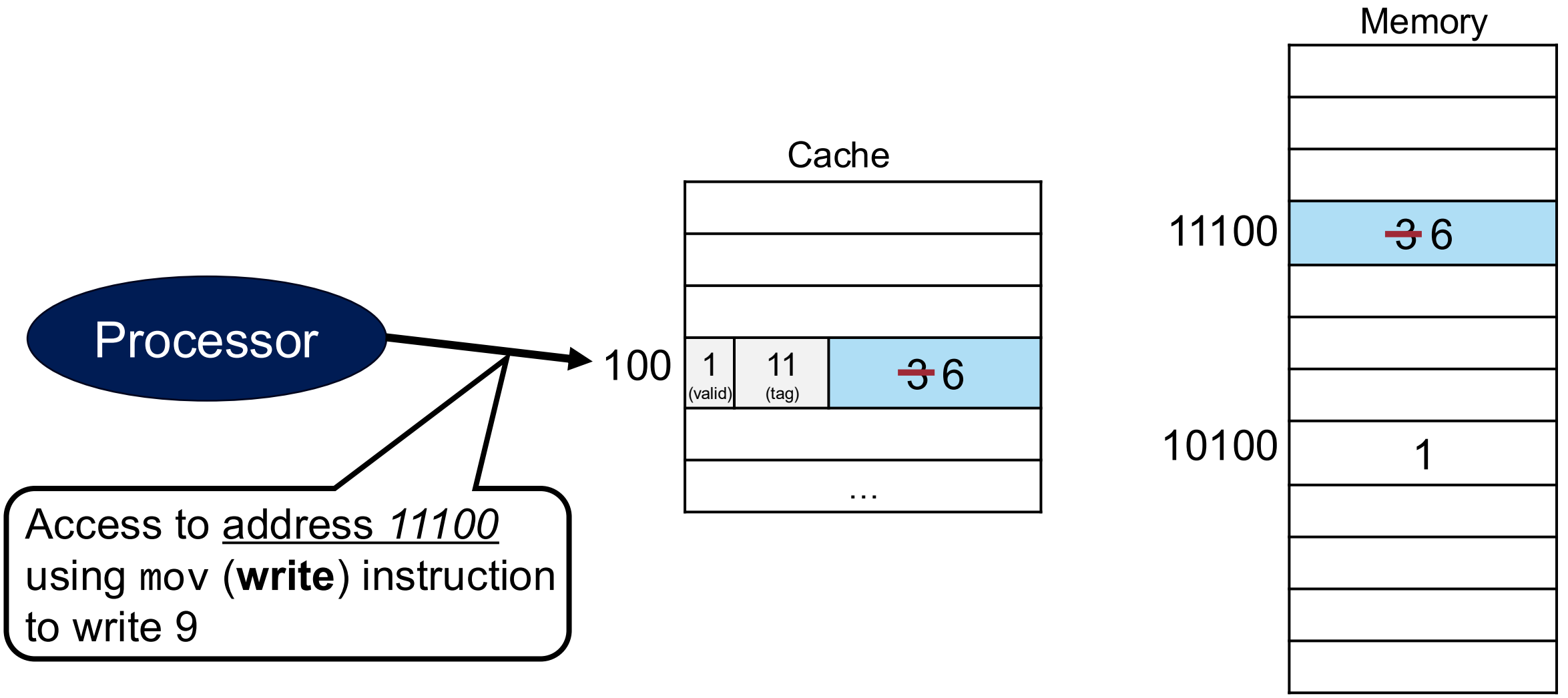
Write Through



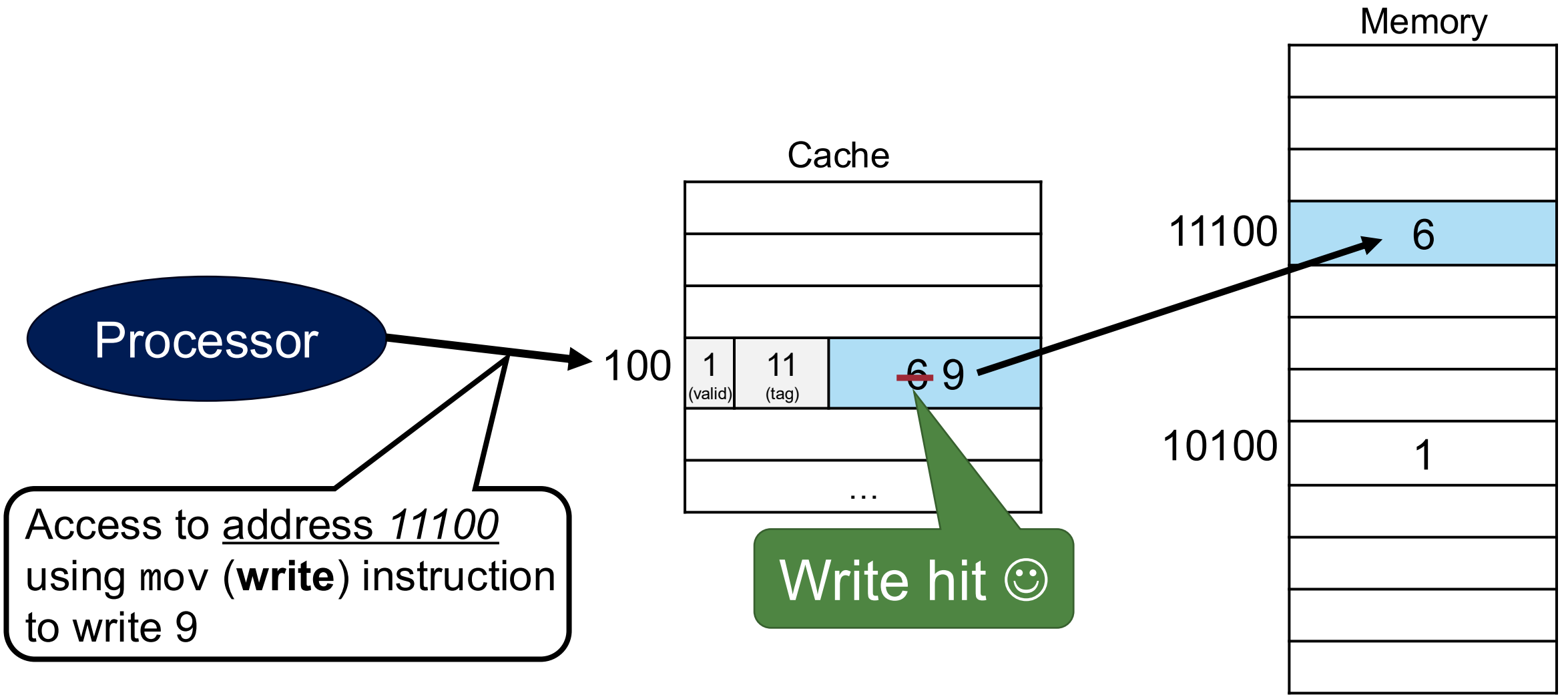
Write Through



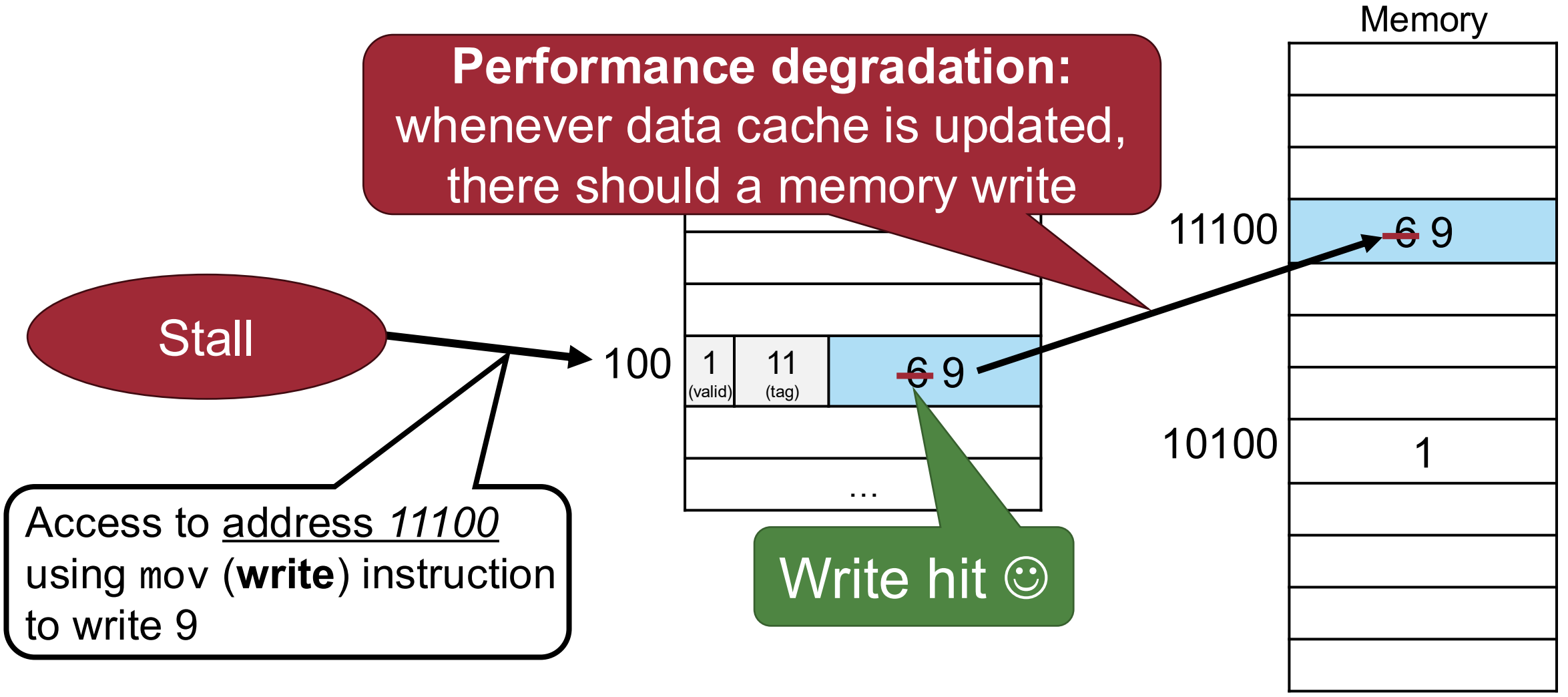
Write Through



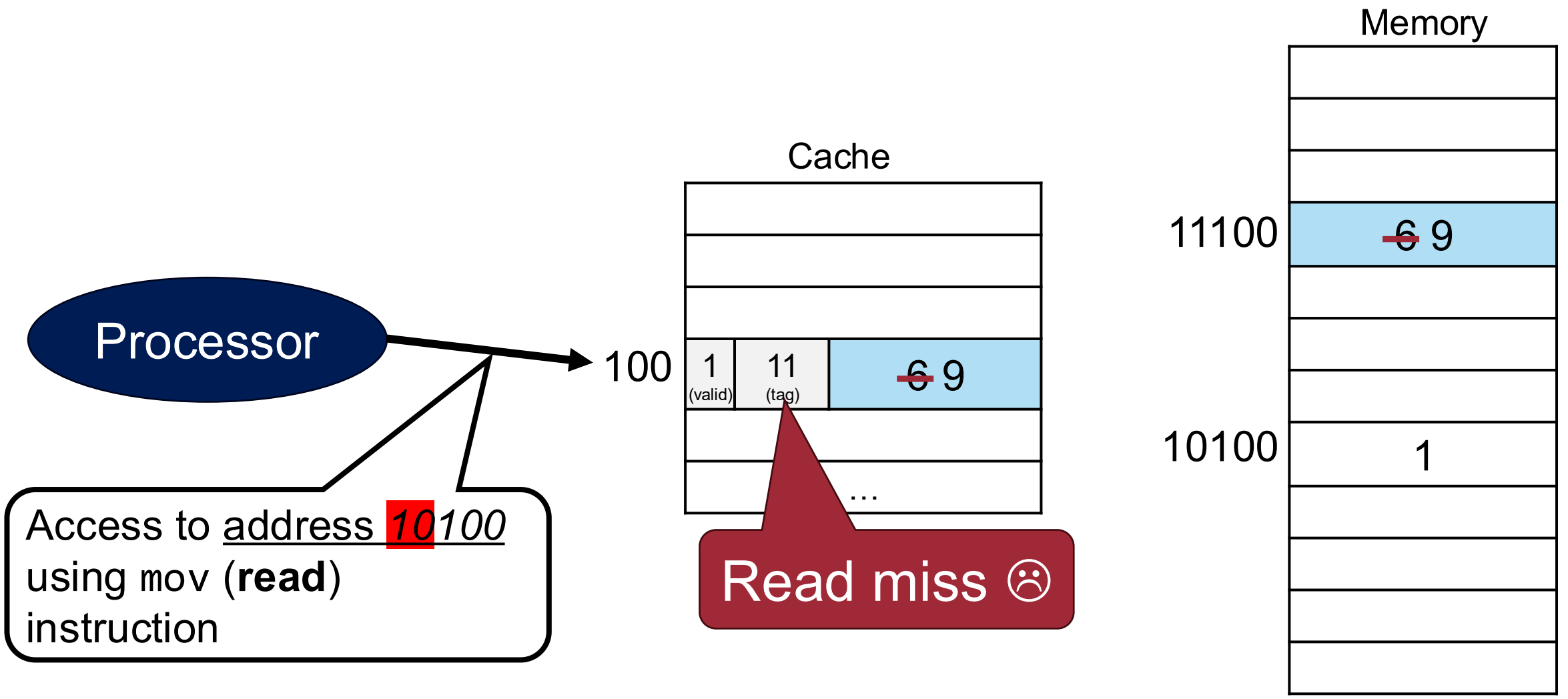
Write Through



Write Through

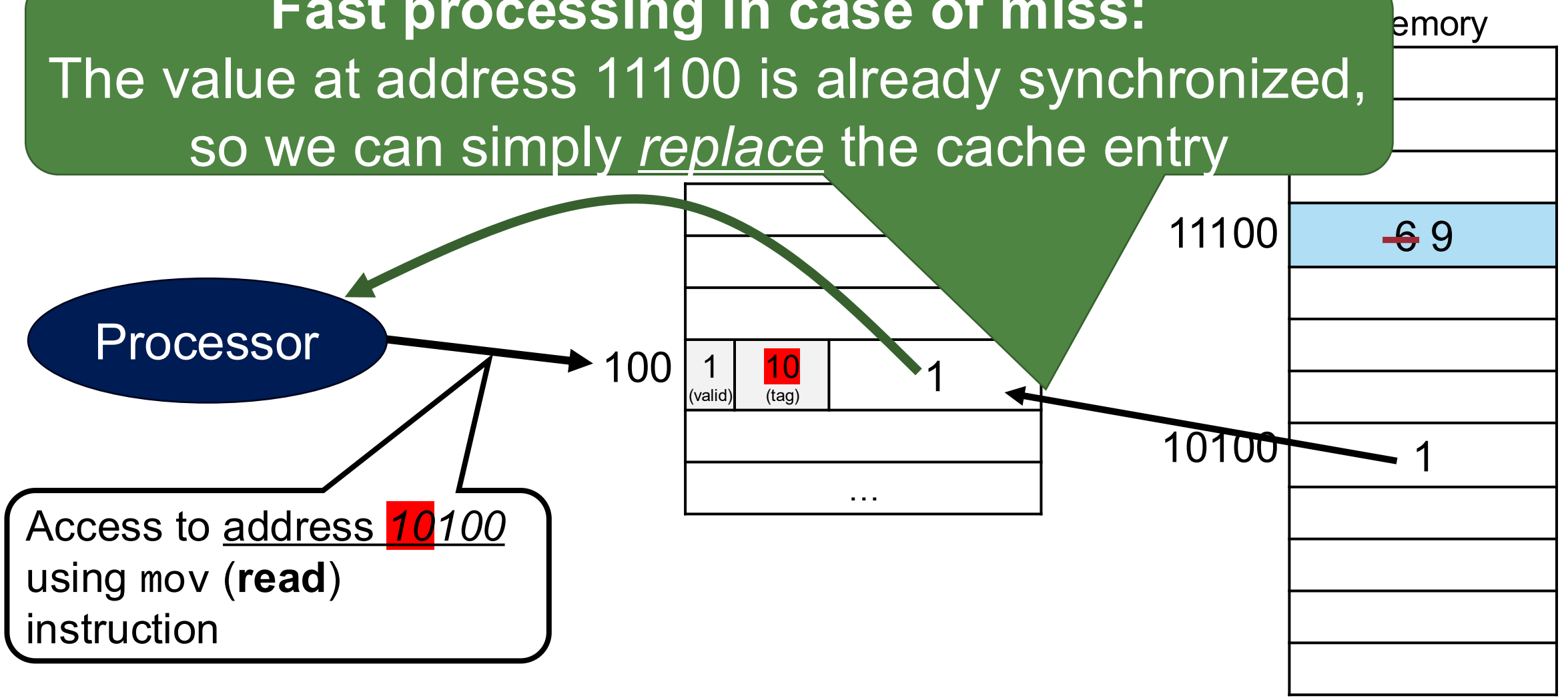


Write Through

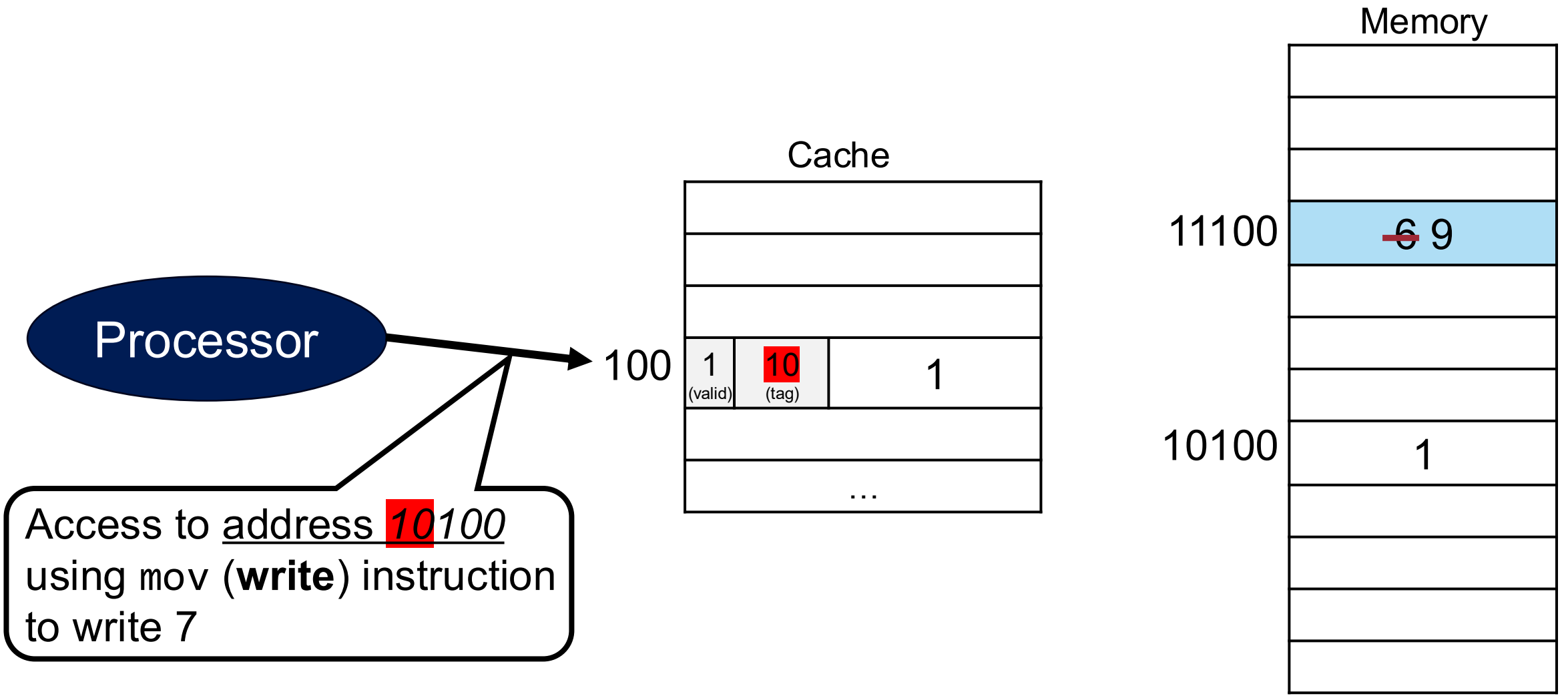


Write Through

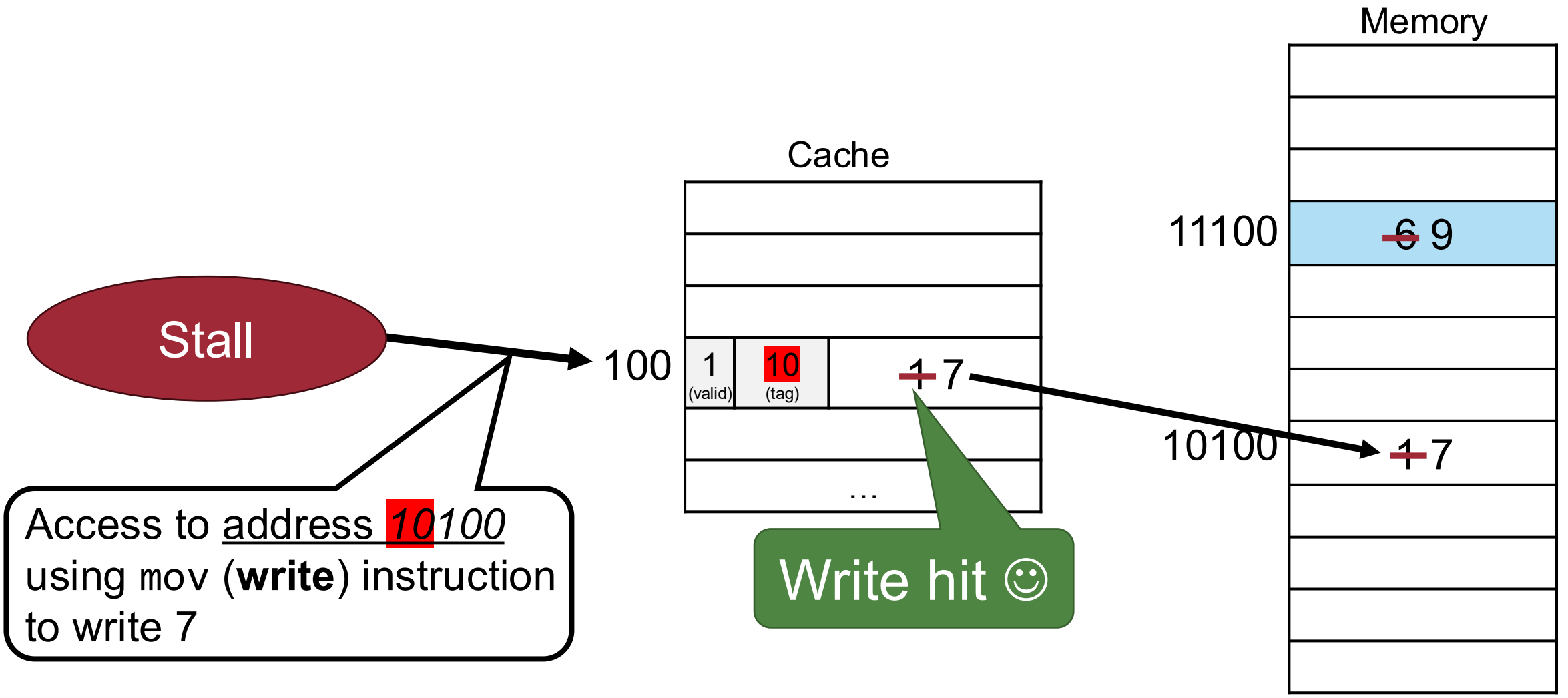
Fast processing in case of miss:
The value at address 11100 is already synchronized, so we can simply *replace* the cache entry



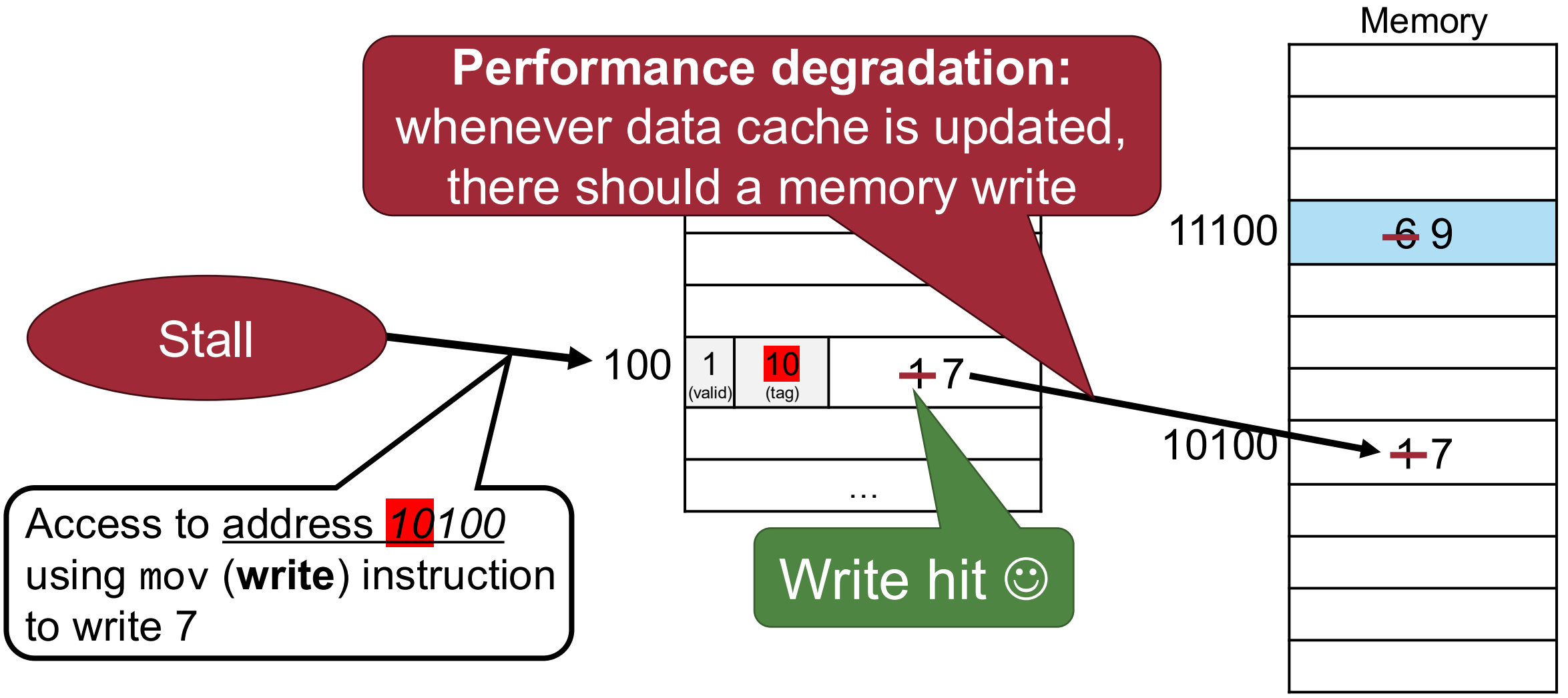
Write Through



Write Through



Main Disadvantage: Performance Degradation



Write Policies

- Write hits
  - Cache and memory would be inconsistent! What can we do?

  (1) Write through: On each write hit, the information is written to both in the cache and in the memory
    - Pros: faster processing in case of 'miss'
    - Cons: make writes take longer (whenever data cache is updated, there should a memory write)

  (2) Write back:

Write Policies

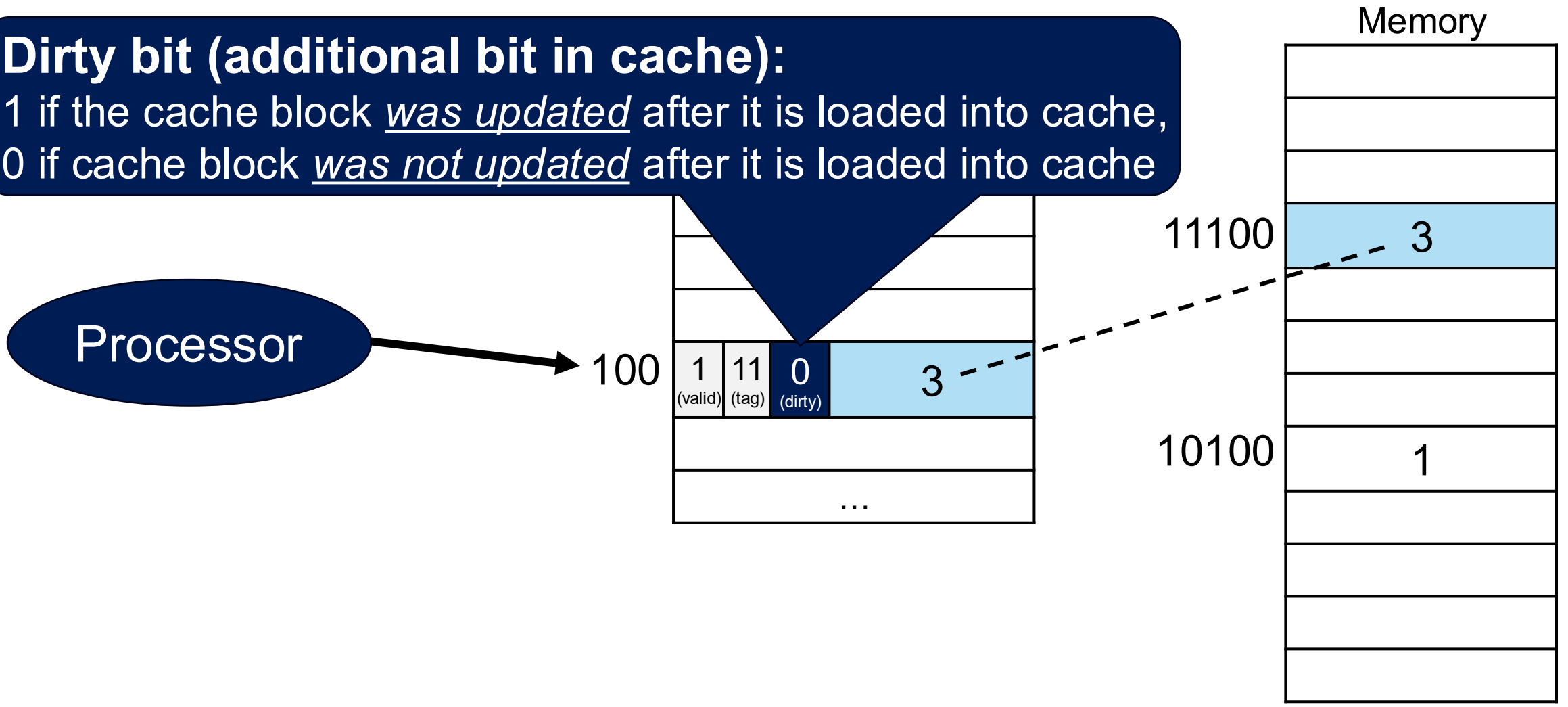
- Write hits
  - Cache and memory would be inconsistent! What can we do?

  (1) Write through: On each write hit, the information is written to both in the cache and in the memory
    - Pros: faster processing in case of 'miss'
    - Cons: make writes take longer (whenever data cache is updated, there should a memory write)

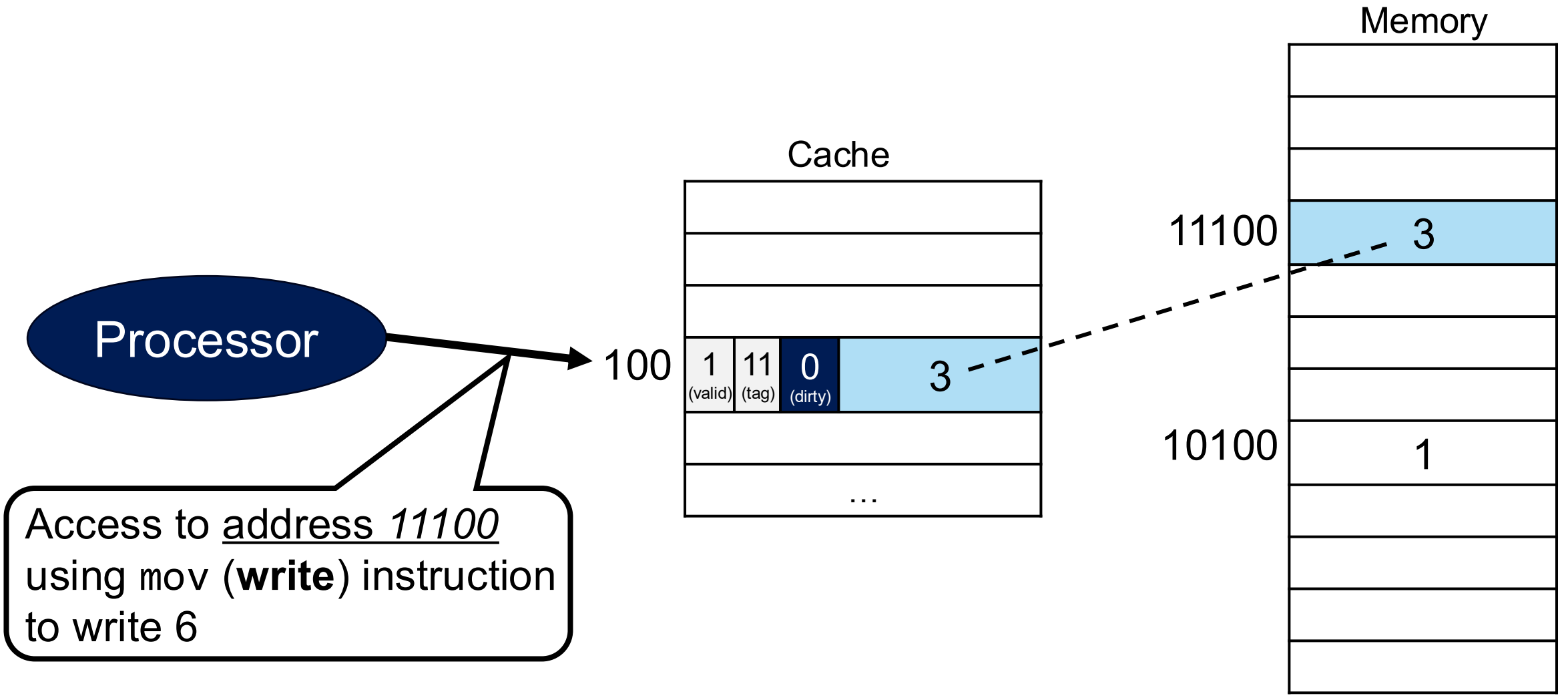
  (2) Write back: On each write hit, update only the block in cache. The modified cache block is written to memory ***only when it is replaced***
    - Keep track of whether each block is *dirty (additional bit)*

Write Back – Dirty Bit

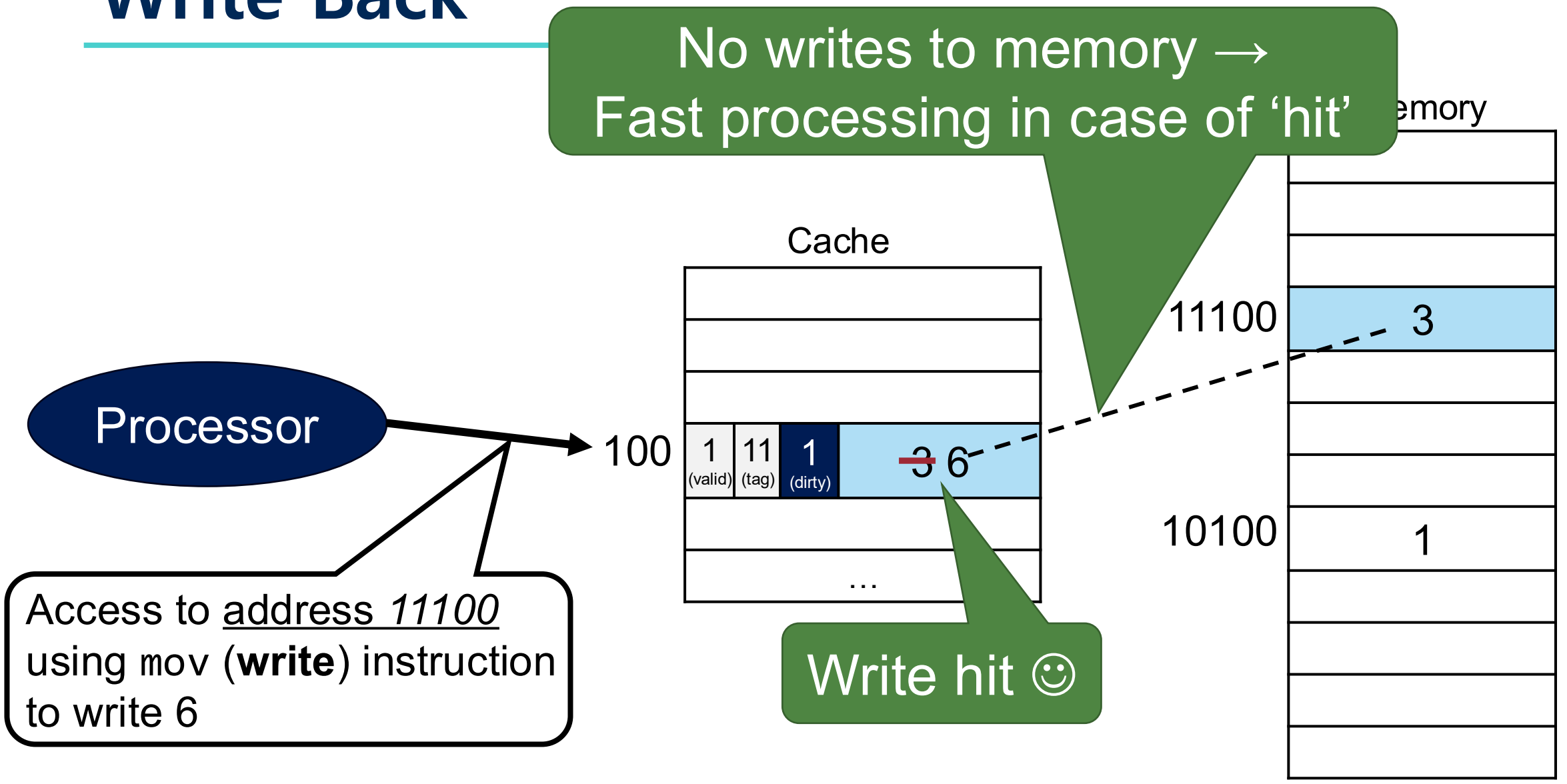
Dirty bit (additional bit in cache):
1 if the cache block *was updated* after it is loaded into cache,
0 if cache block *was not updated* after it is loaded into cache



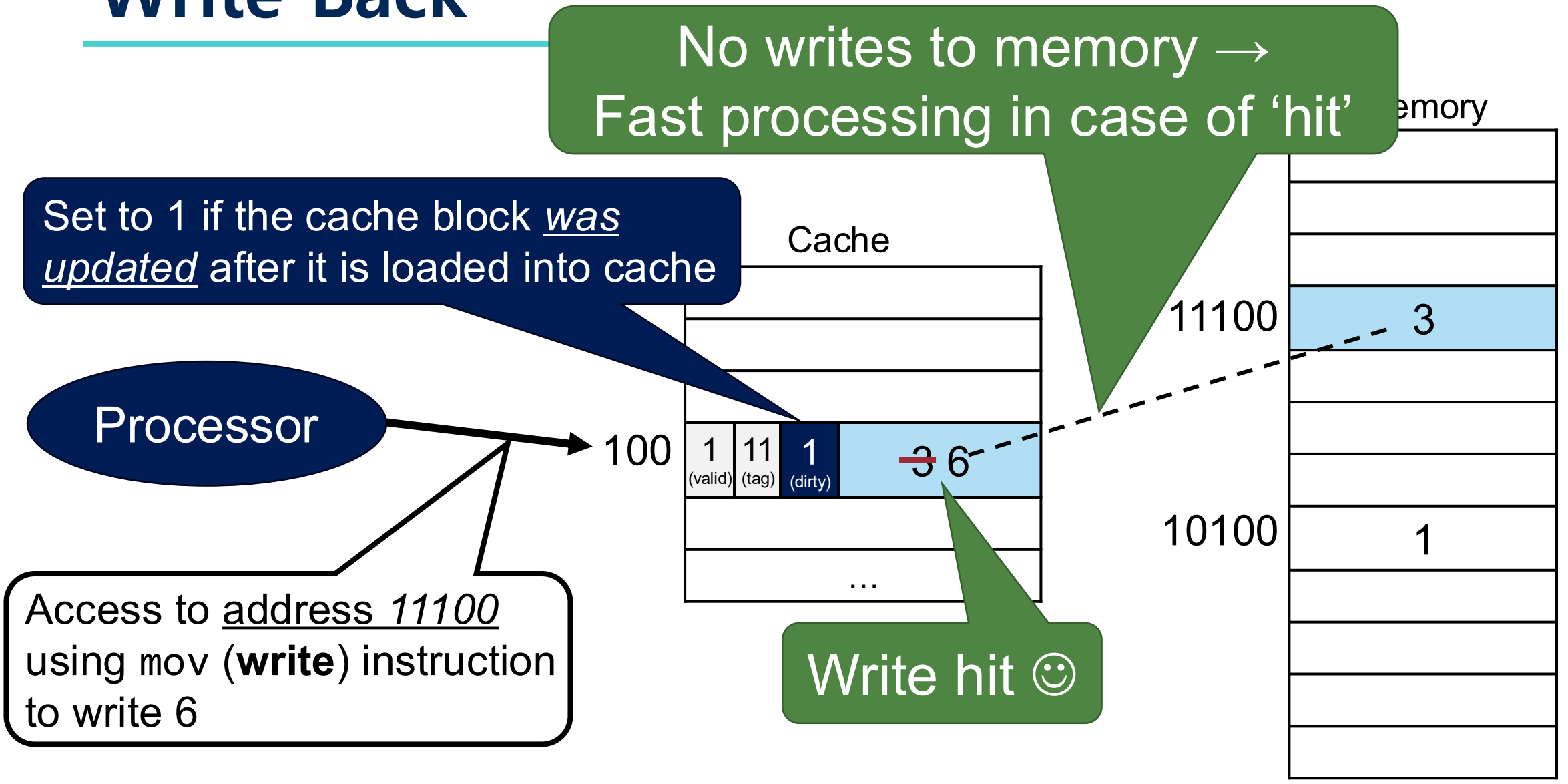
Write Back – Dirty Bit



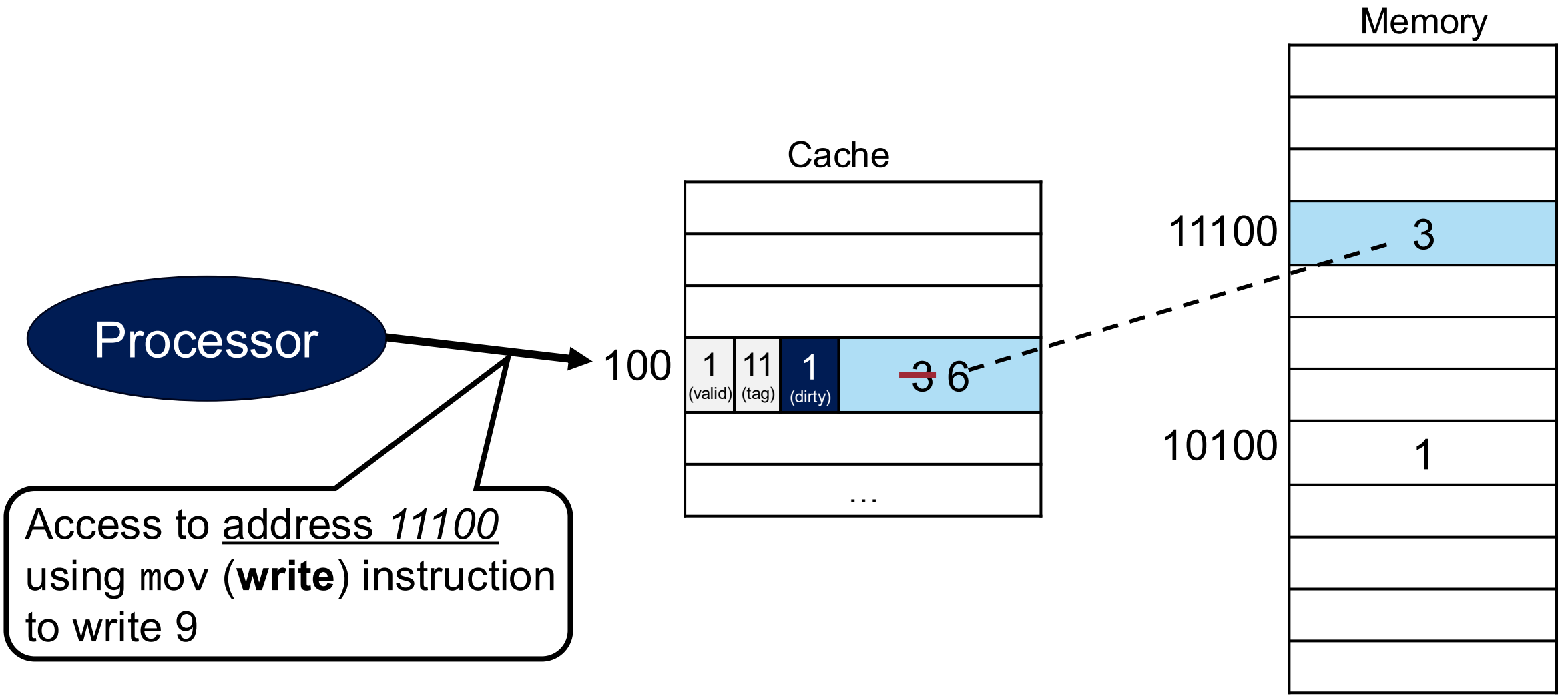
Write Back


No writes to memory →
Fast processing in case of 'hit'
Processor
Access to address 11100 using mov (write) instruction to write 6
Cache
100
1 (valid)
11 (tag)
1 (dirty)
3 6
...
Write hit ☺
emory
11100
3
10100
1

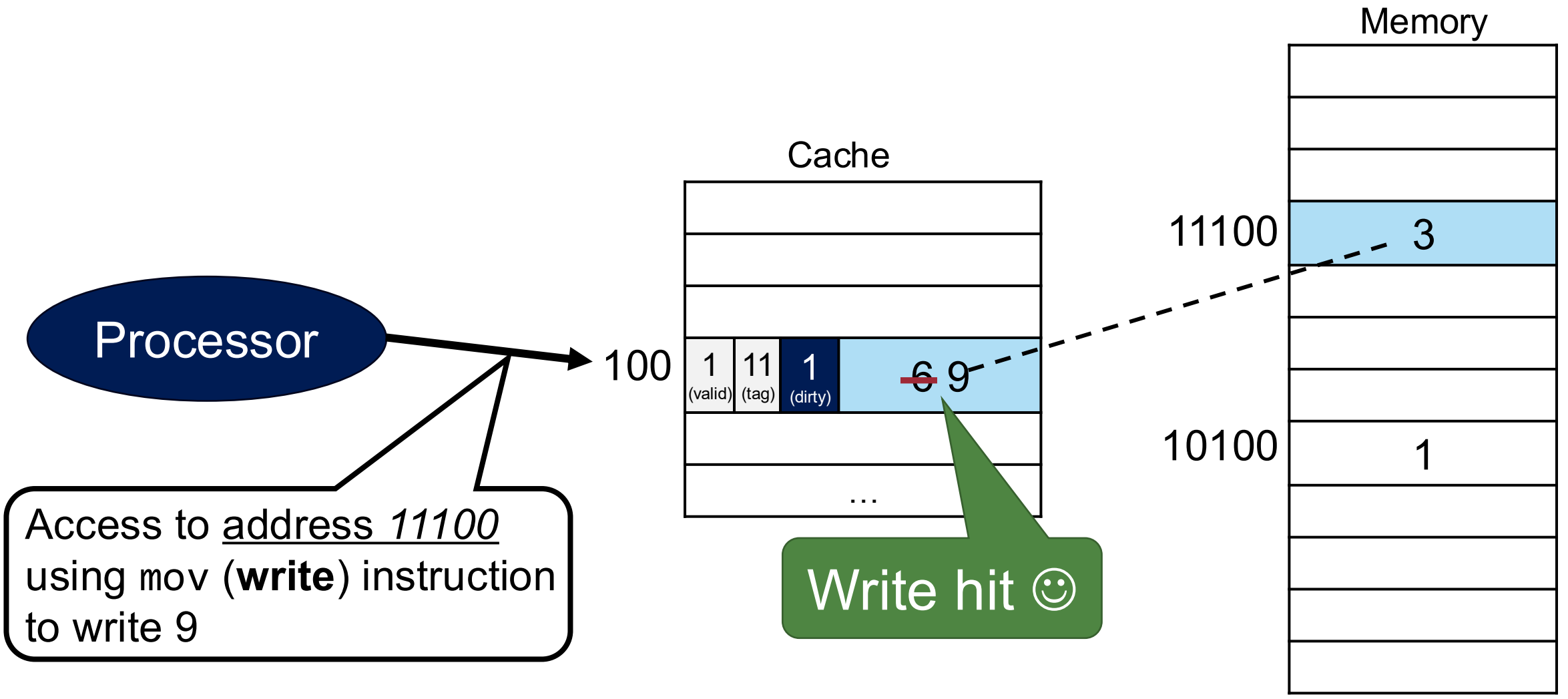
Write Back


No writes to memory →
Fast processing in case of 'hit'
Set to 1 if the cache block *was updated* after it is loaded into cache
Processor
Access to address *11100* using mov (**write**) instruction to write 6
Cache
100
1 (valid)
11 (tag)
1 (dirty)
3 6
...
Write hit ☺
emory
11100
3
10100
1

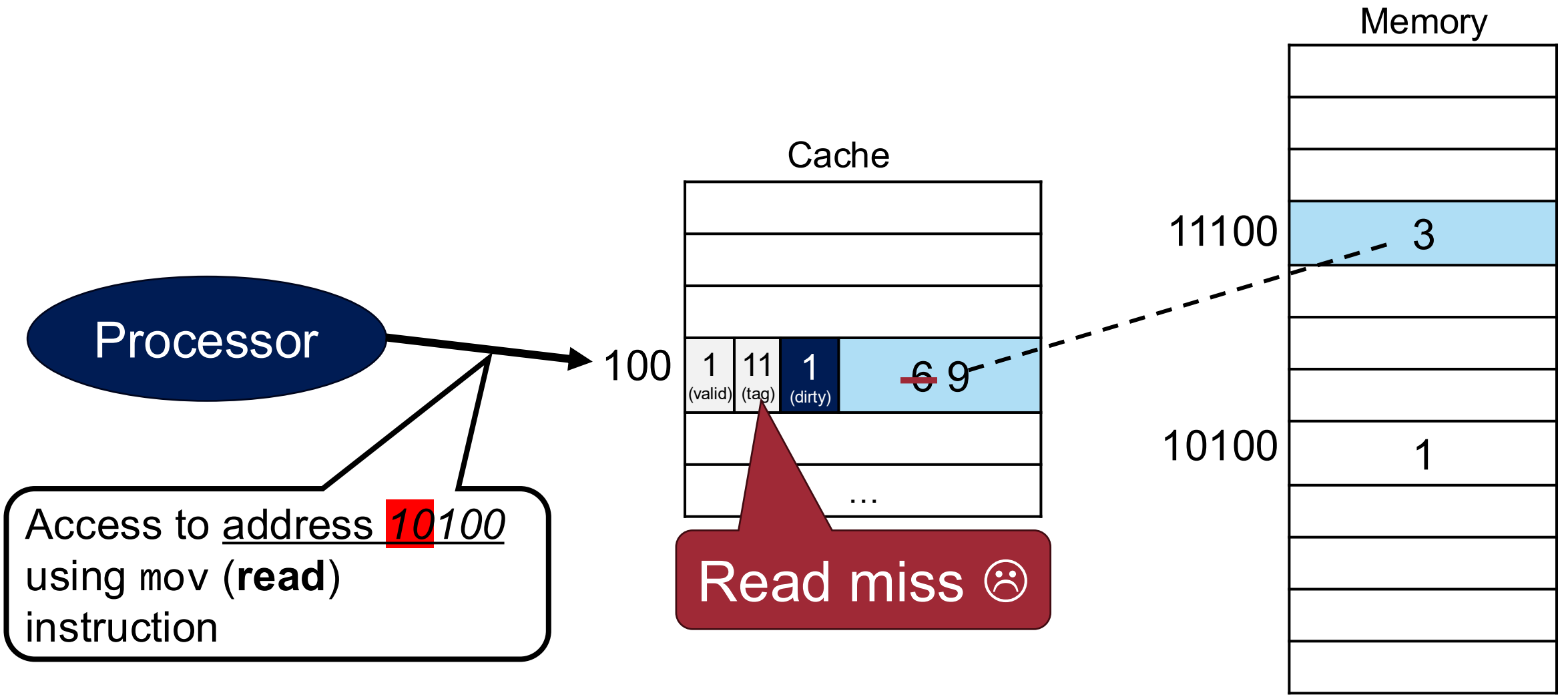
Write Back


Memory
Cache
Processor
100
1
(valid)
11
(tag)
1
(dirty)
3 6
...
11100
3
10100
1
Access to address 11100
using mov (write) instruction
to write 9

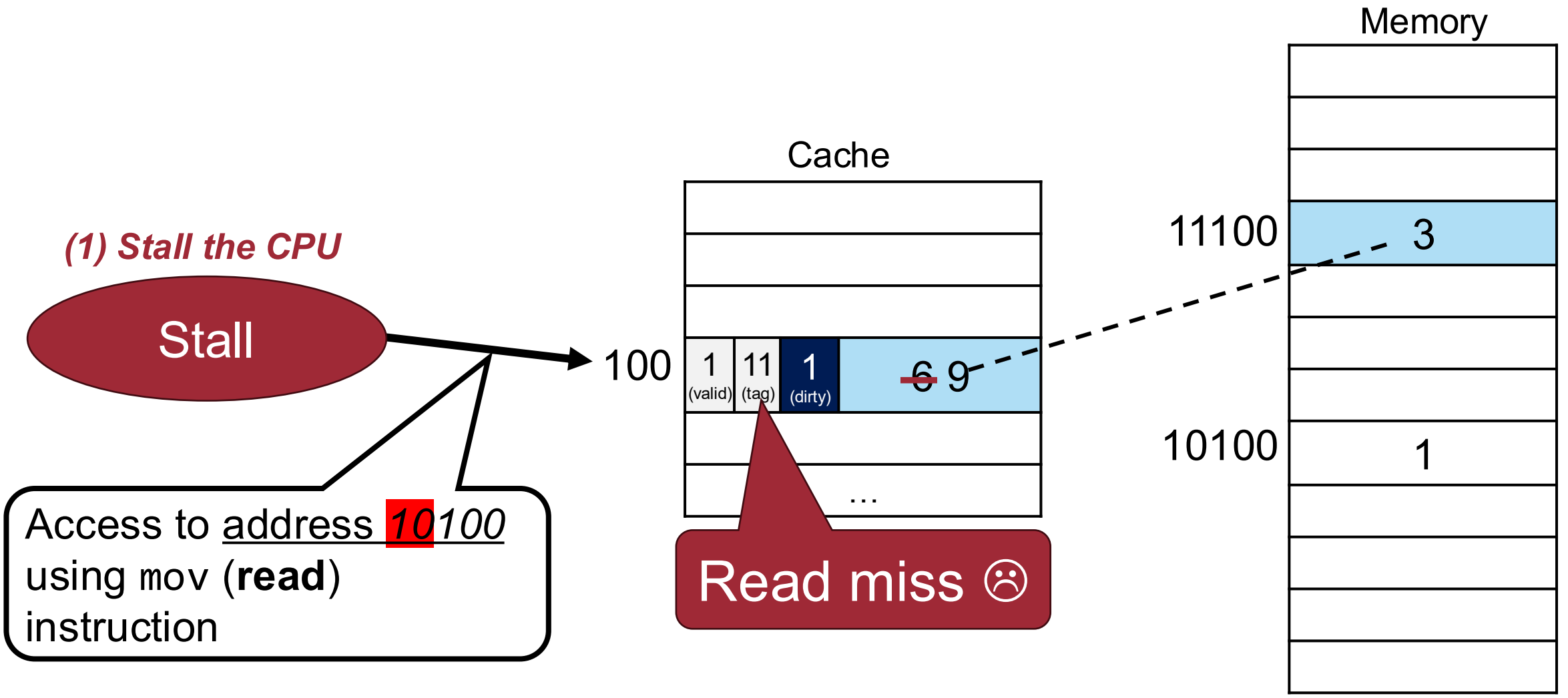
Write Back


Memory
Cache
Processor
100
1
(valid)
11
(tag)
1
(dirty)
6 9
...
11100
3
10100
1
Access to address 11100 using mov (write) instruction to write 9
Write hit ☺

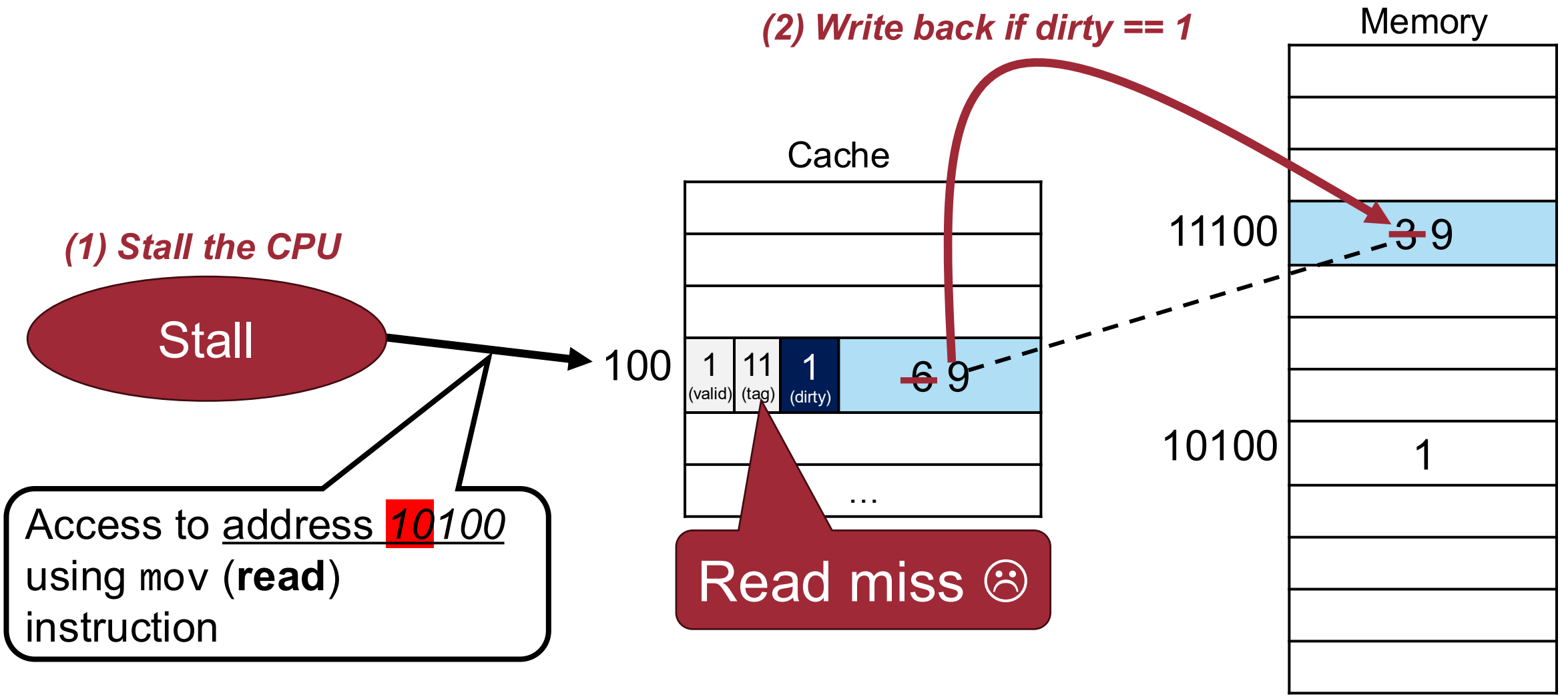
Write Back


Memory
Cache
Processor
100
1
(valid)
11
(tag)
1
(dirty)
6 9
...
11100
3
10100
1
Access to address 10100 using mov (read) instruction
Read miss ☹

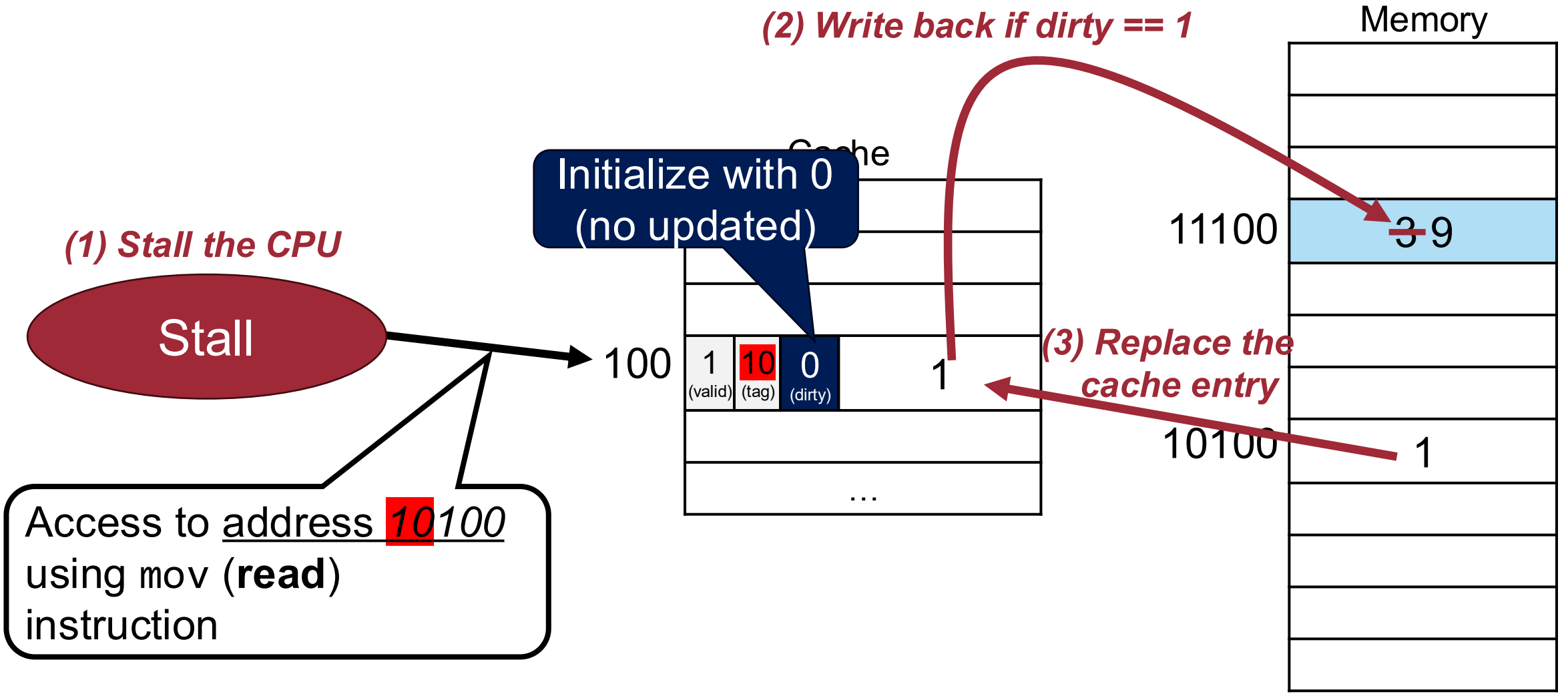
Write Back


(1) Stall the CPU
Stall
Access to address 10100 using mov (read) instruction
Cache
100
1 (valid)
11 (tag)
1 (dirty)
6 9
...
Read miss ☹
Memory
11100
3
10100
1

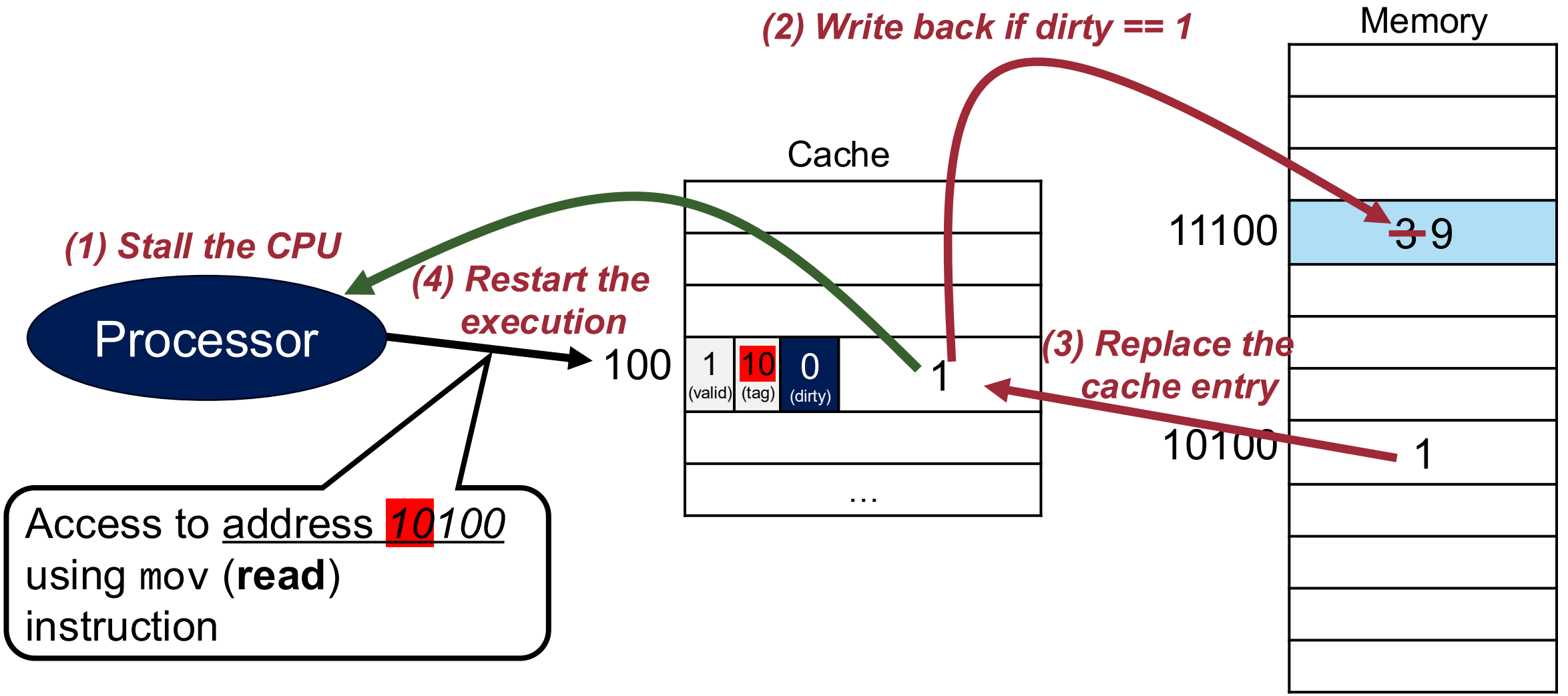
Write Back


(2) Write back if dirty == 1
Memory
Cache
(1) Stall the CPU
Stall
100
1 (valid)
11 (tag)
1 (dirty)
6 9
...
11100
3 9
10100
1
Access to address 10100 using mov (read) instruction
Read miss ☹

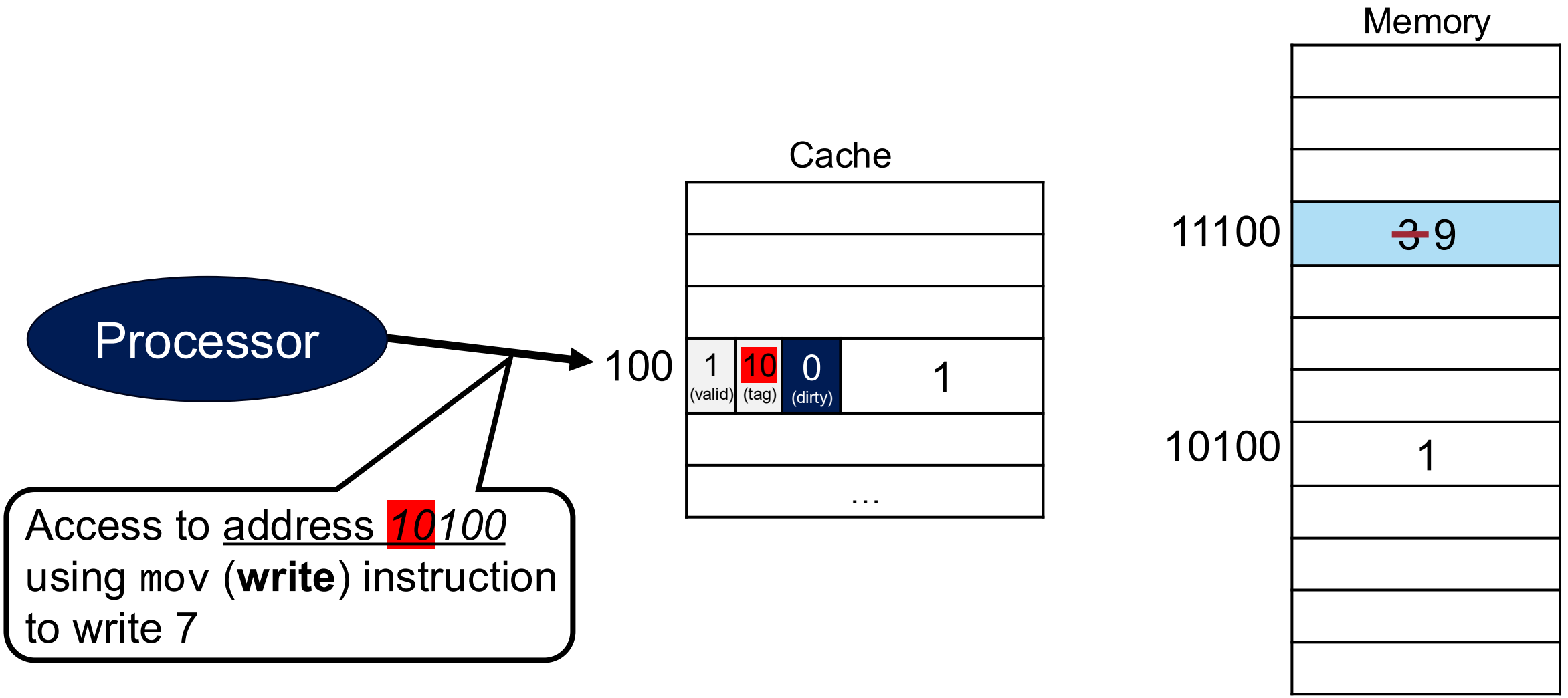
Write Back


(2) Write back if dirty == 1
Memory
Initialize with 0
(no updated)
(1) Stall the CPU
Stall
11100
3 9
100
1
(valid)
10
(tag)
0
(dirty)
1
(3) Replace the
cache entry
10100
1
...
Access to address 10100
using mov (read)
instruction

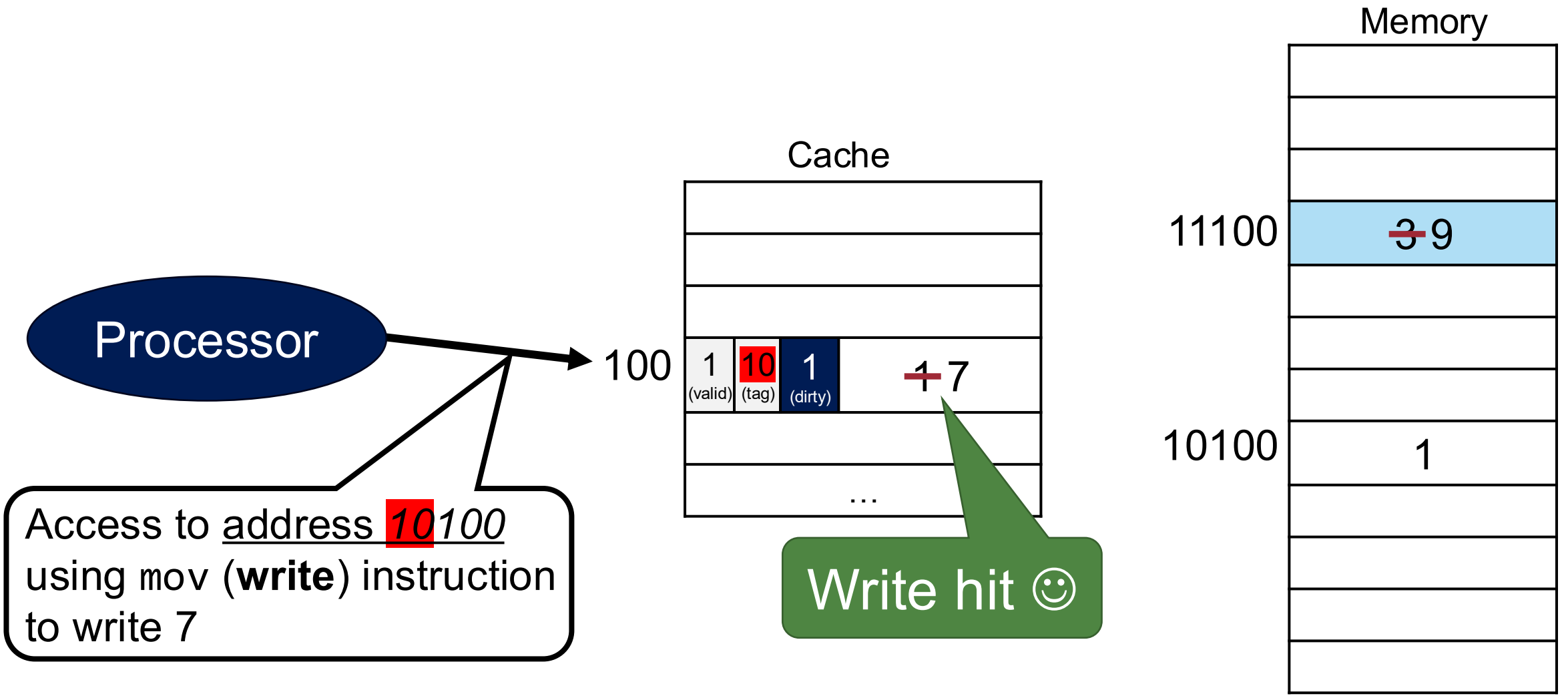
Write Back


(2) Write back if dirty == 1
Memory
Cache
(1) Stall the CPU
(4) Restart the execution
Processor
100
1 (valid)
10 (tag)
0 (dirty)
1
(3) Replace the cache entry
11100
3 9
10100
1
...
Access to address 10100 using mov (read) instruction

Write Back


Memory
Cache
Processor
100
1
(valid)
10
(tag)
0
(dirty)
1
...
11100
3 9
10100
1
Access to address 10100 using mov (write) instruction to write 7

Write Back


Memory
Cache
Processor
100
1
(valid)
10
(tag)
1
(dirty)
~~1~~ 7
...
11100
~~3~~ 9
10100
1
Access to address 10100 using mov (**write**) instruction to write 7
Write hit ☺

Write Policies

- Write hits
  - Cache and memory would be inconsistent! What can we do?

  (1) Write through: On each write hit, the information is written to both in the cache and in the memory
    - Pros: faster processing in case of 'miss'
    - Cons: make writes take longer (whenever data cache is updated, there should a memory write)

  (2) Write back: On each write hit, update only the block in cache. The modified cache block is written to memory ***only when it is replaced***
    - Keep track of whether each block is *dirty (additional bit)*.
    - Pros: faster processing in case of 'hit' (No repeated writes to memory)
    - Cons: slower in case of 'miss'

Summary: Hits vs. Misses

- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch block from memory, restart
- Write hits
  - (option #1) Write the data both in cache and memory (*write through*)
  - (option #2) Write the data only into the cache (*write back*)
- Write misses

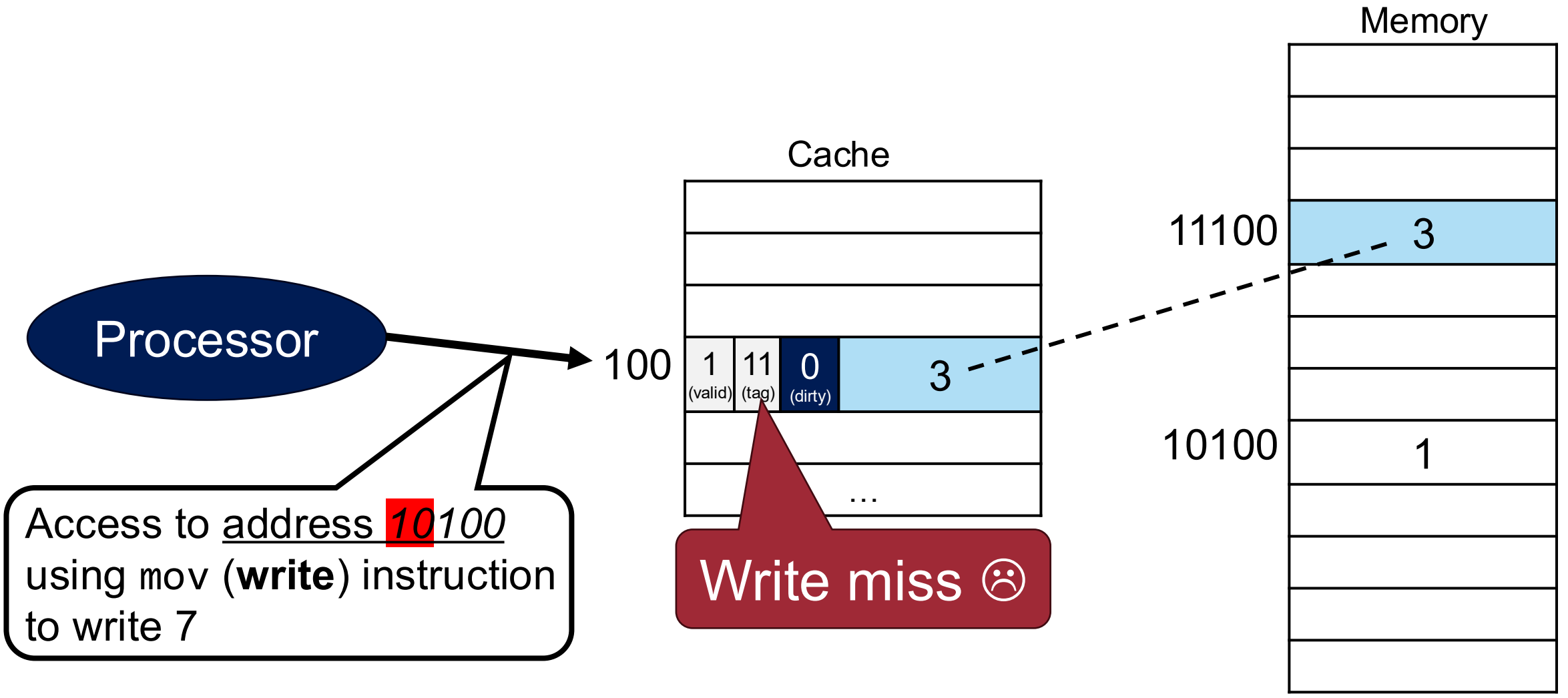
Write Misses

- Write misses
  - **Write allocate (common strategy)**
  - **No write allocation**:

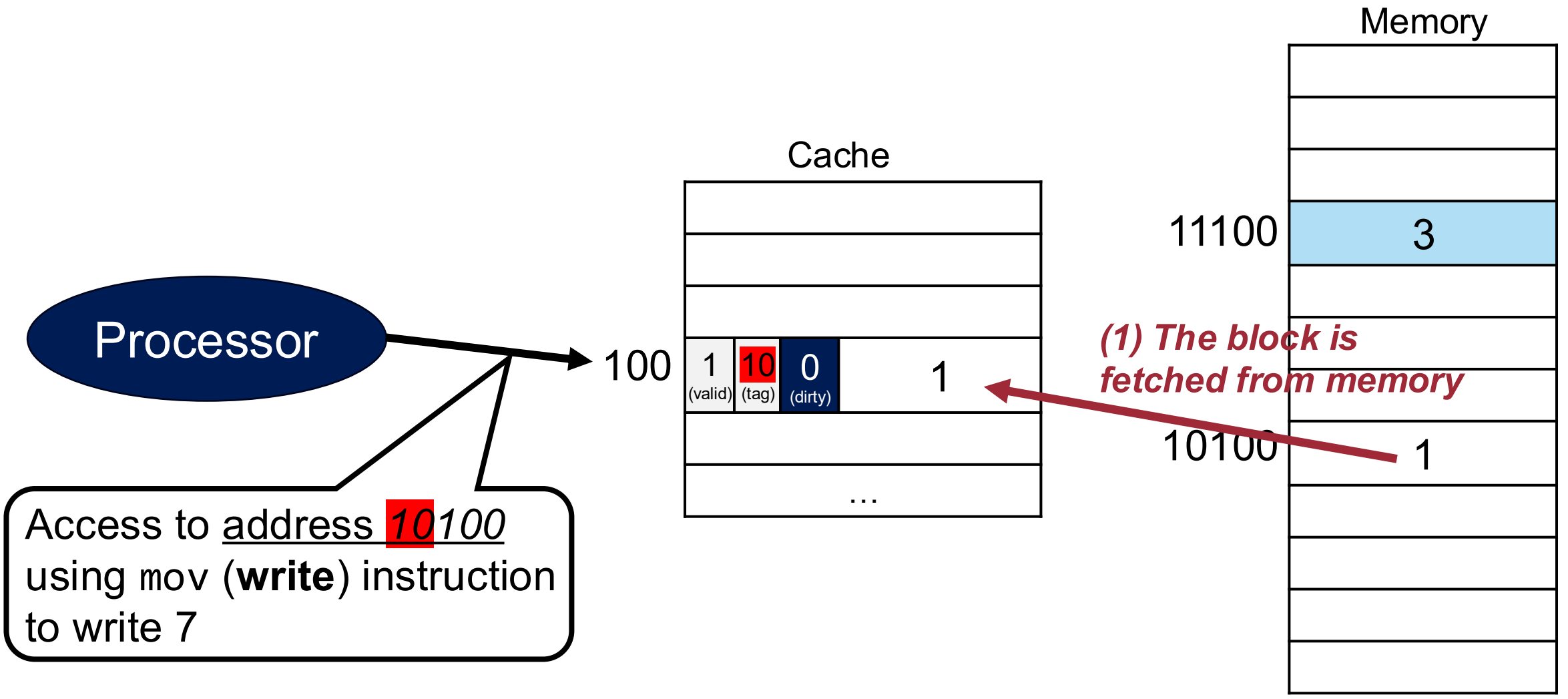
Write Misses

- Write misses
  - **Write allocate (common strategy):** allocate block on miss in the cache
    - The block is fetched from memory and then the appropriate portion of the block is overwritten (Similar with read miss)
  - **No write allocation**:

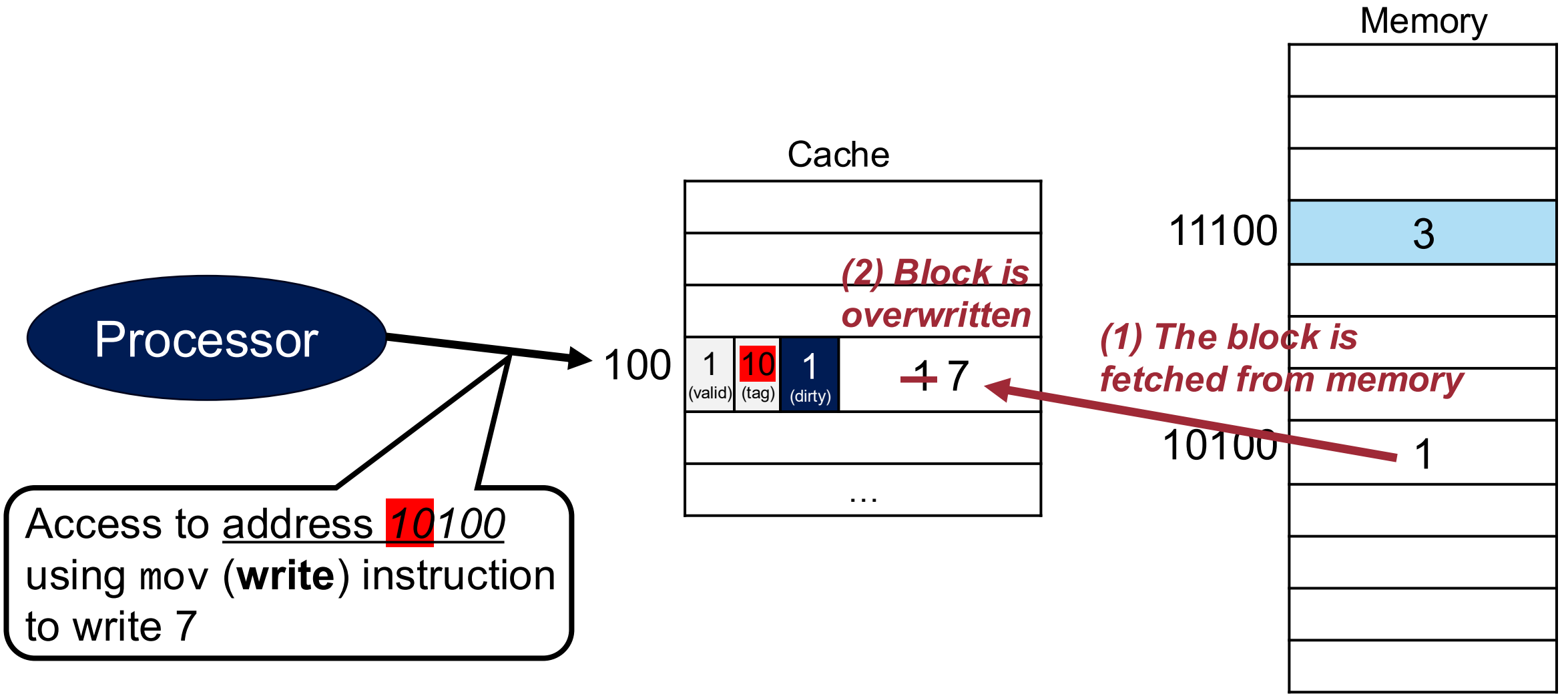
Write Misses: Write Allocate



Write Misses: Write Allocate



Write Misses: Write Allocate



Write Misses

- Write misses
  - **Write allocate (common strategy):** allocate block on miss in the cache
    - The block is fetched from memory and then the appropriate portion of the block is overwritten (Similar with read miss)
  - **No write allocation**: update the portion of the block in memory *in direct manner* but not put it in the cache

Summary: Cache Hits vs. Misses

- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch block from memory, restart
- Write hits
  - (option #1) Write the data both in cache and memory (*write through*)
  - (option #2) Write the data only into the cache (*write back*)
- Write misses
  - (option #1) Write allocate
  - (option #2) Write no allocate

Typical
- Write-through + No-write-allocate
- **Write-back + Write-allocate**

Question?