CSE261: Computer Architecture

16. Memory Hierarchy (2): Caches #1

Seongil Wi

Department of Computer Science and Engineering

Recap: Memory Hierarchy

A structure that uses multiple levels of memories

Control (brain)

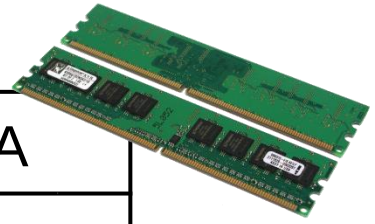
Register 1

Register 2

01010010

Datapath

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



Data A

01010010

00110110

Main memory

01010010
00110110

Program A

11001110
01000110

Program B

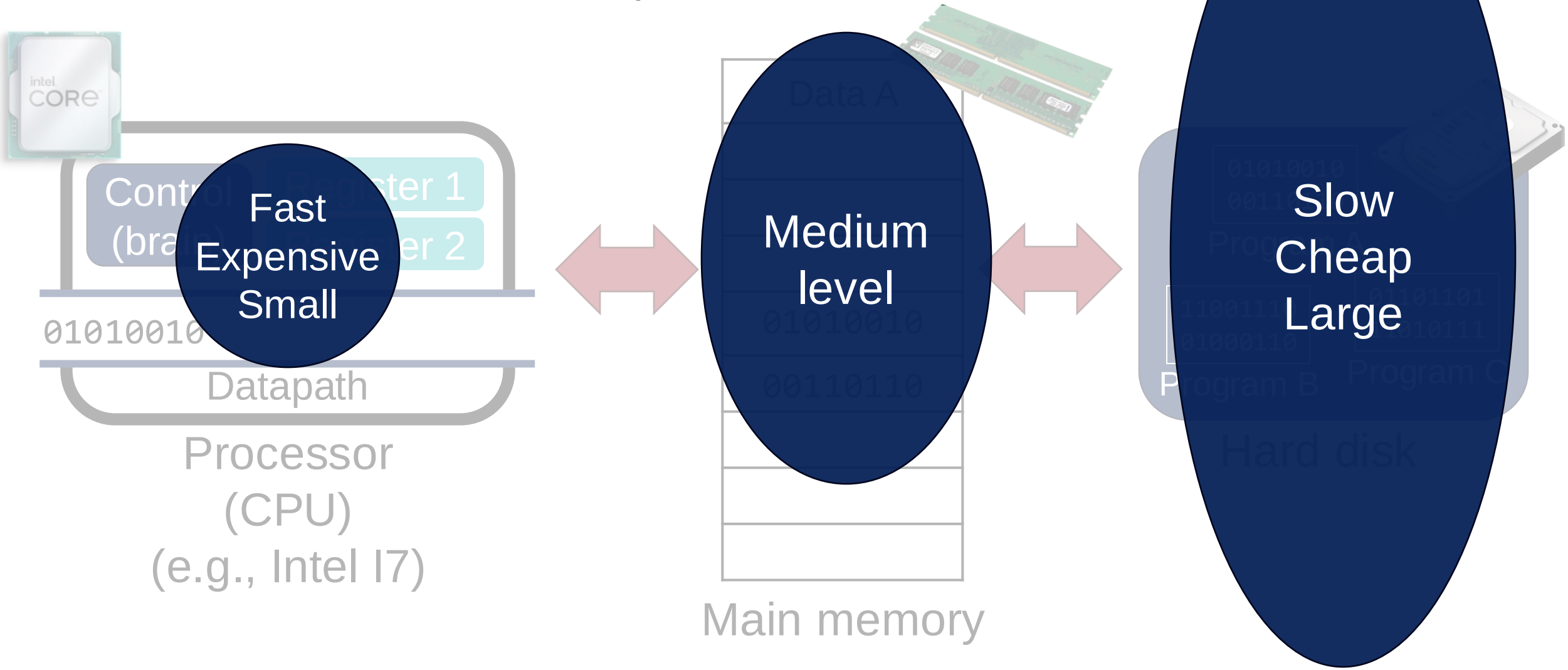
01101101
11010111

Program C

Hard disk

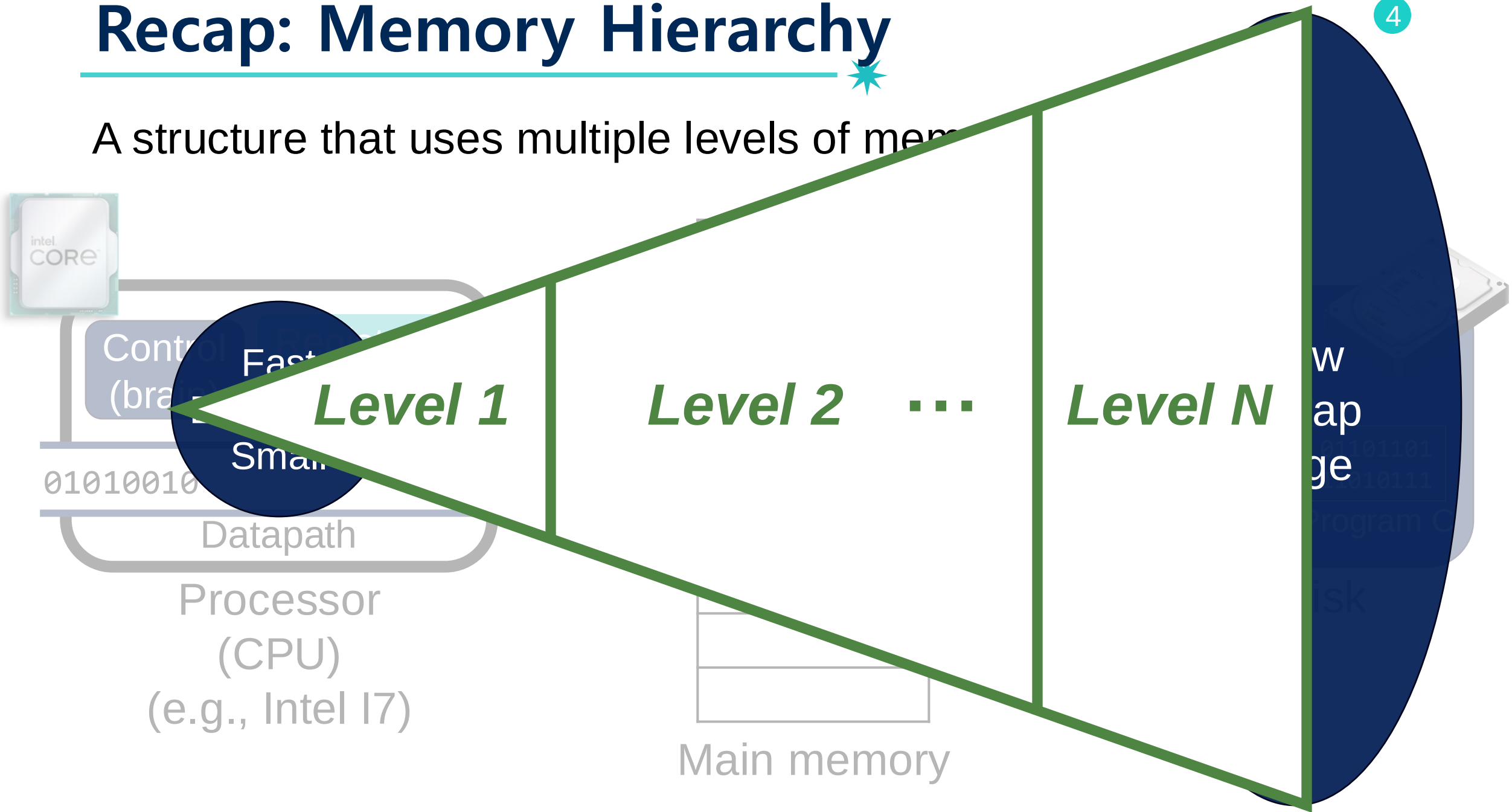
Recap: Memory Hierarchy

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 <u>same set of memory locations</u> <u>*repetitively*</u> over a short period of time

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

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

Recap: Temporal Locality Example

```
a = b + c
d = 2*a + 1
```

Temporal locality: items accessed recently are likely to be accessed again soon!

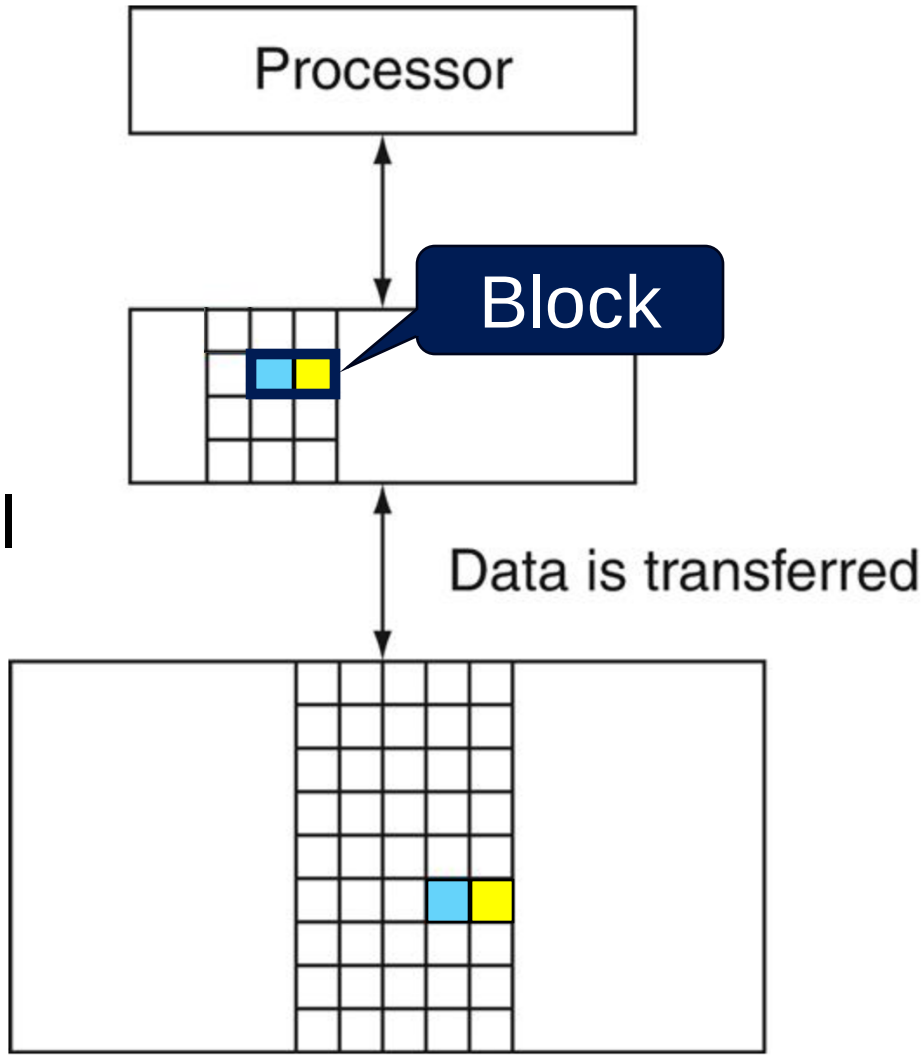
Recap: Spatial Locality Example

```
for (i=0; i<10; i++)
    sum = sum + a[i]
```

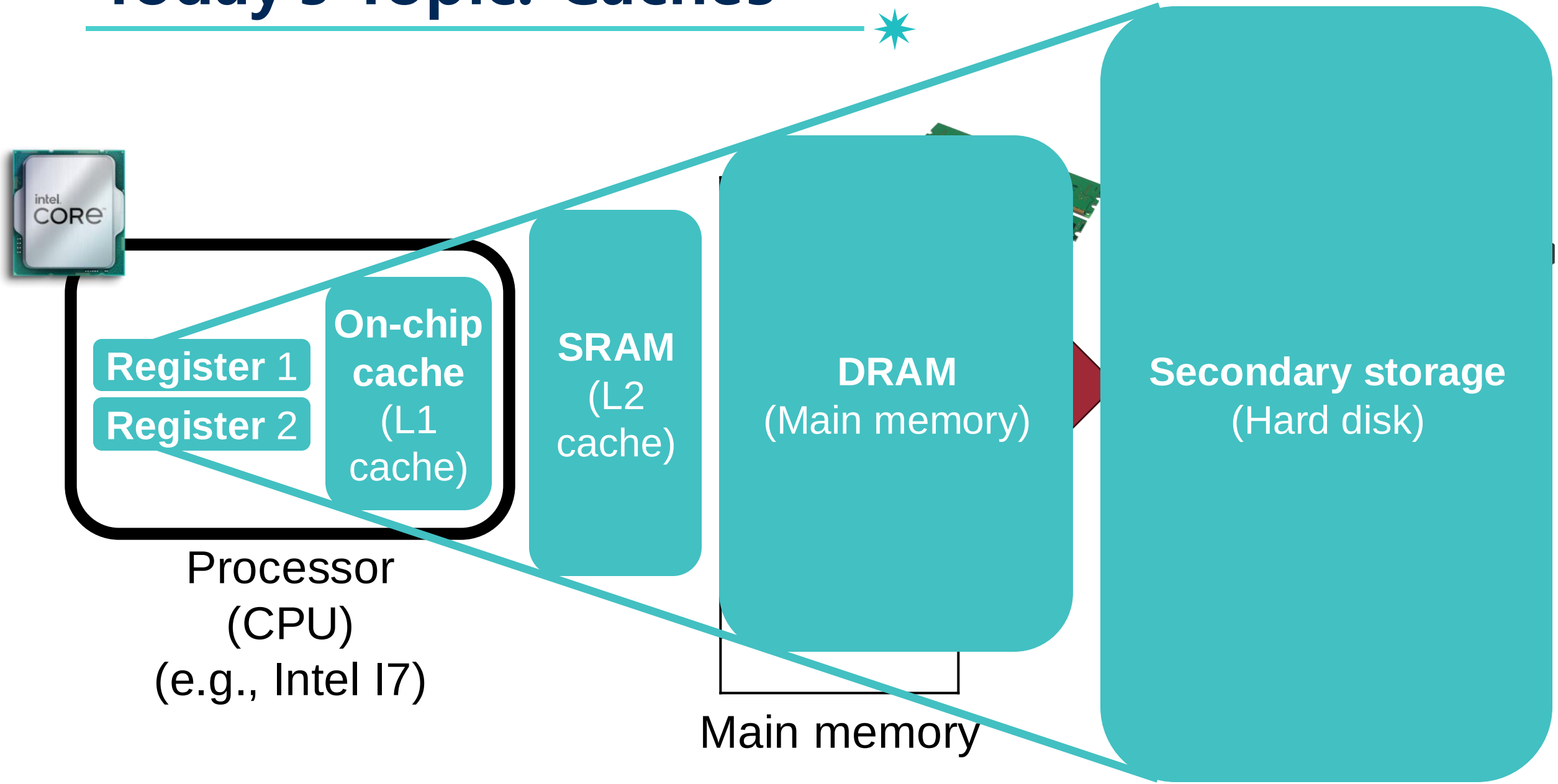
Spatial locality: Items near those accessed recently are likely to be accessed soon

Recap: Terms

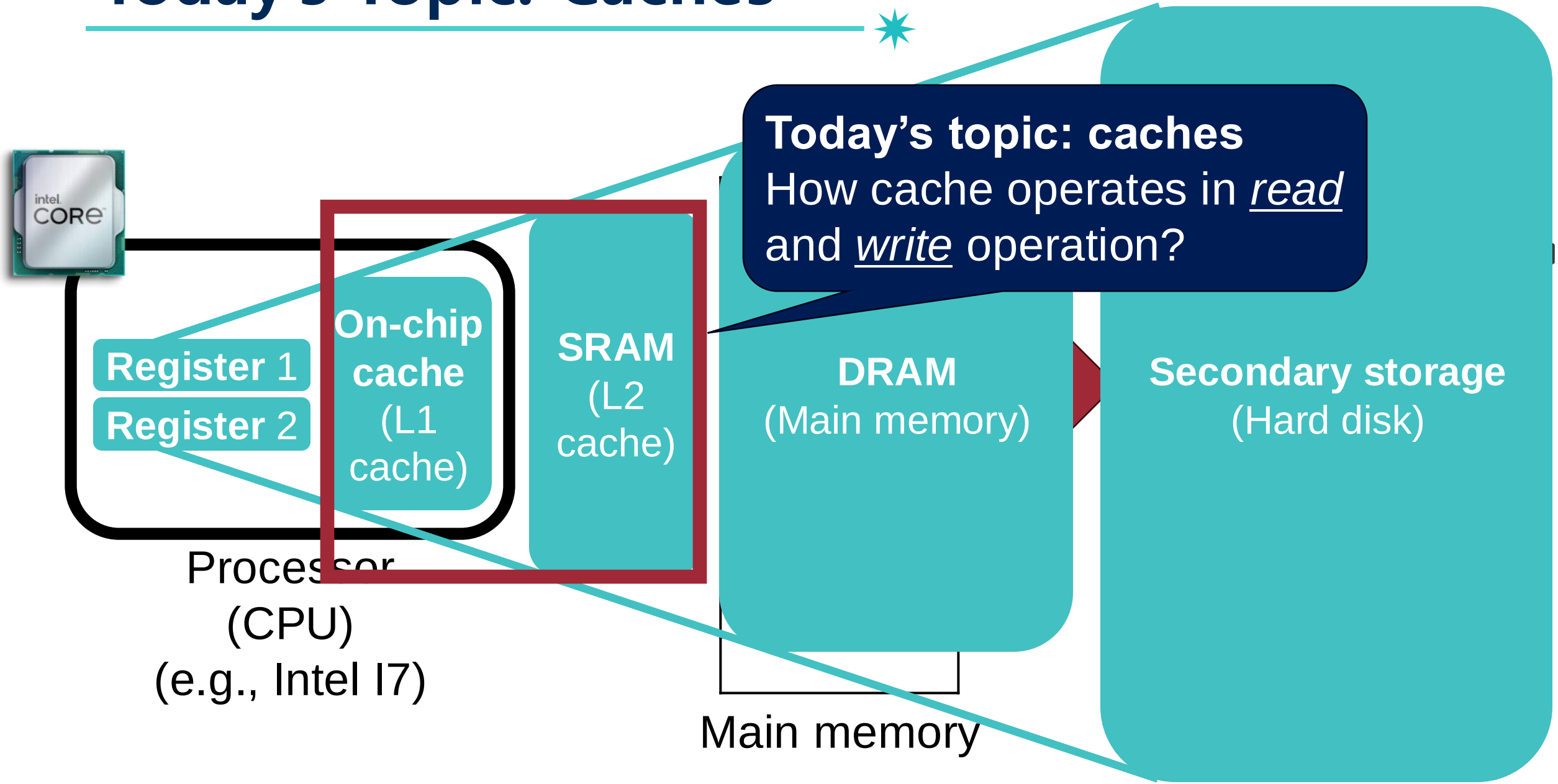
- Block (a.k.a., line): unit of copying
  - Several words in cache memory
- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower leve
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio



Today's Topic: Caches



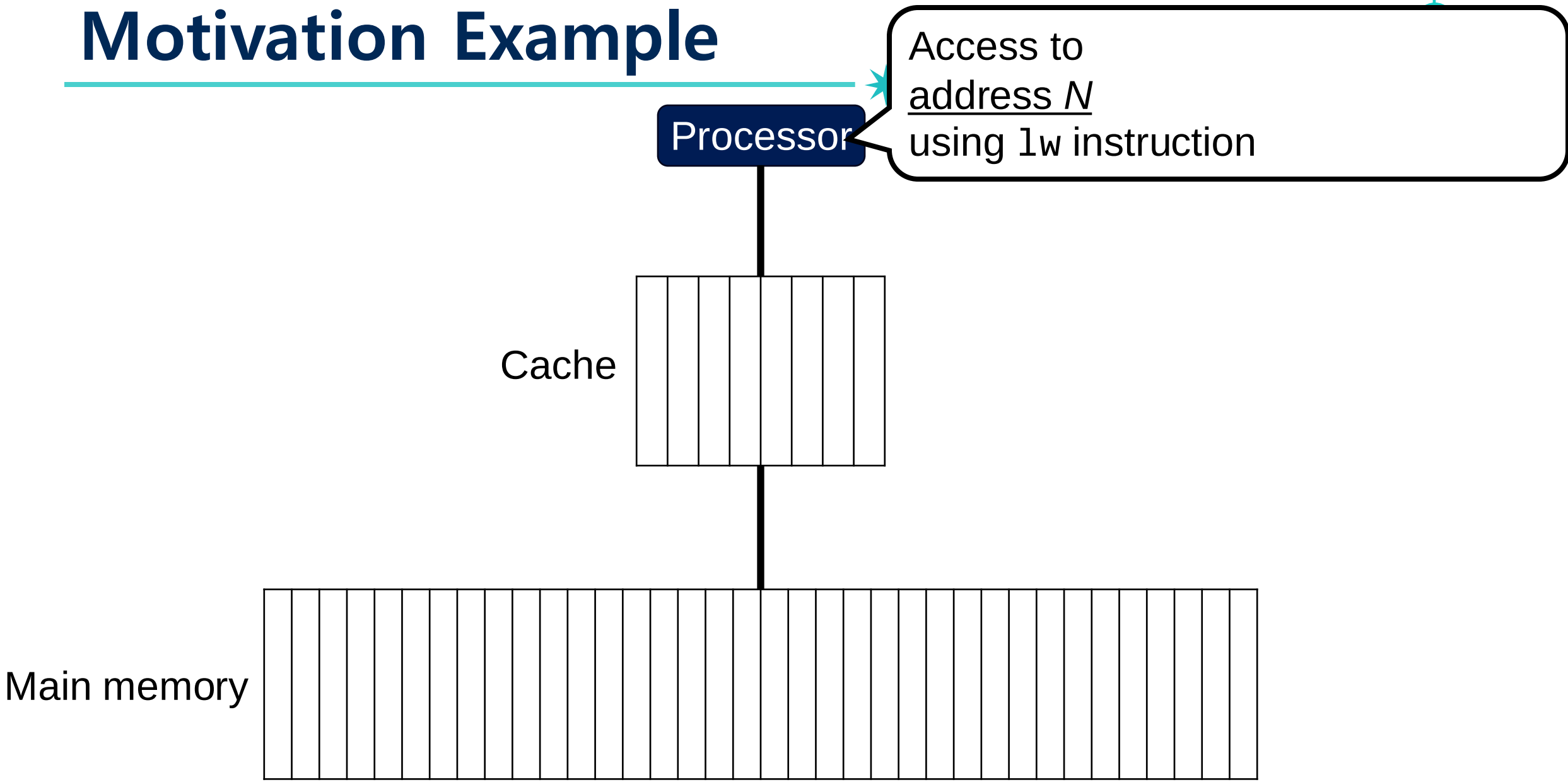
Today's Topic: Caches



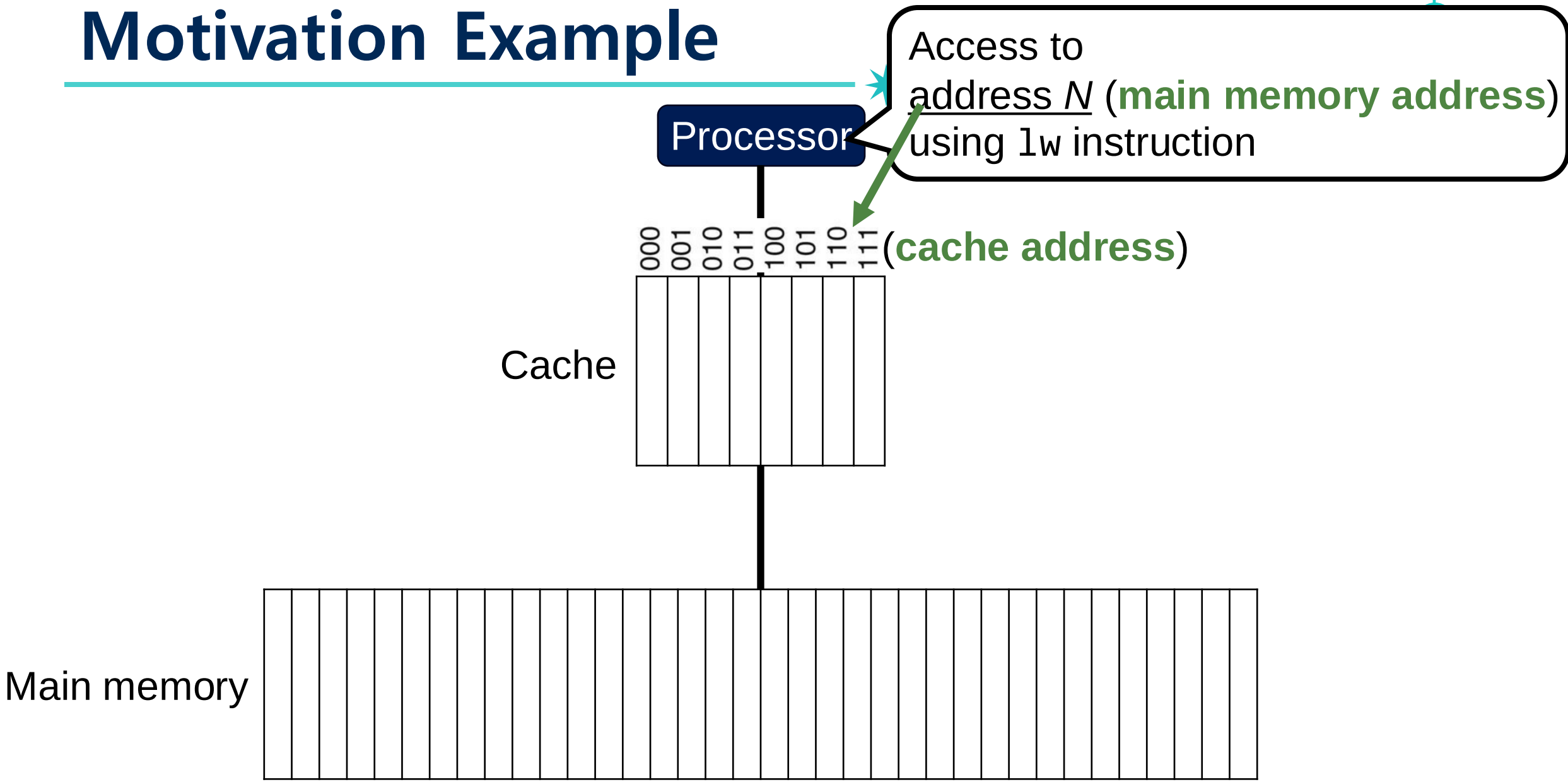
Caches

The level of the memory hierarchy closest to the CPU

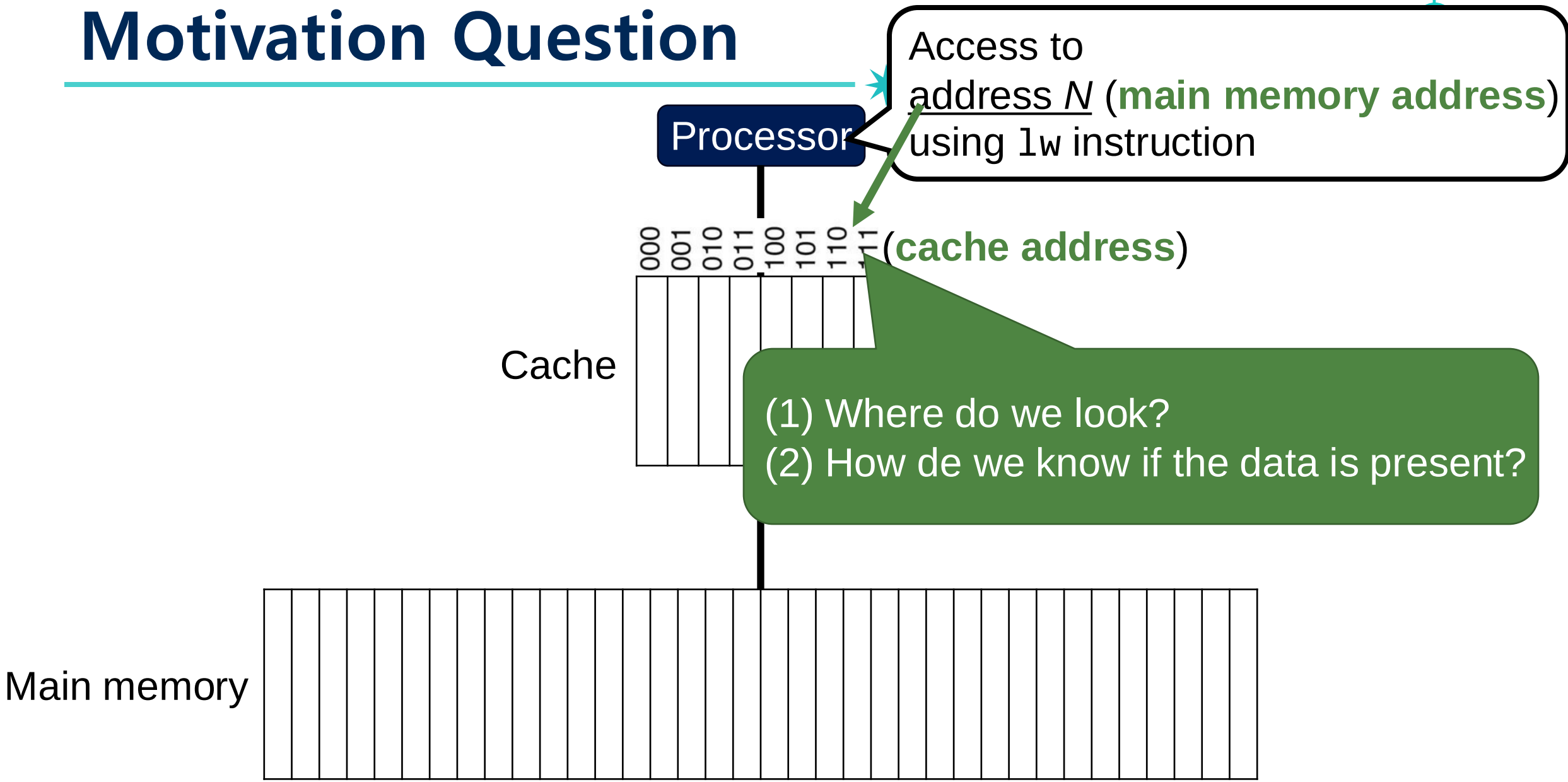
Motivation Example



Motivation Example



Motivation Question



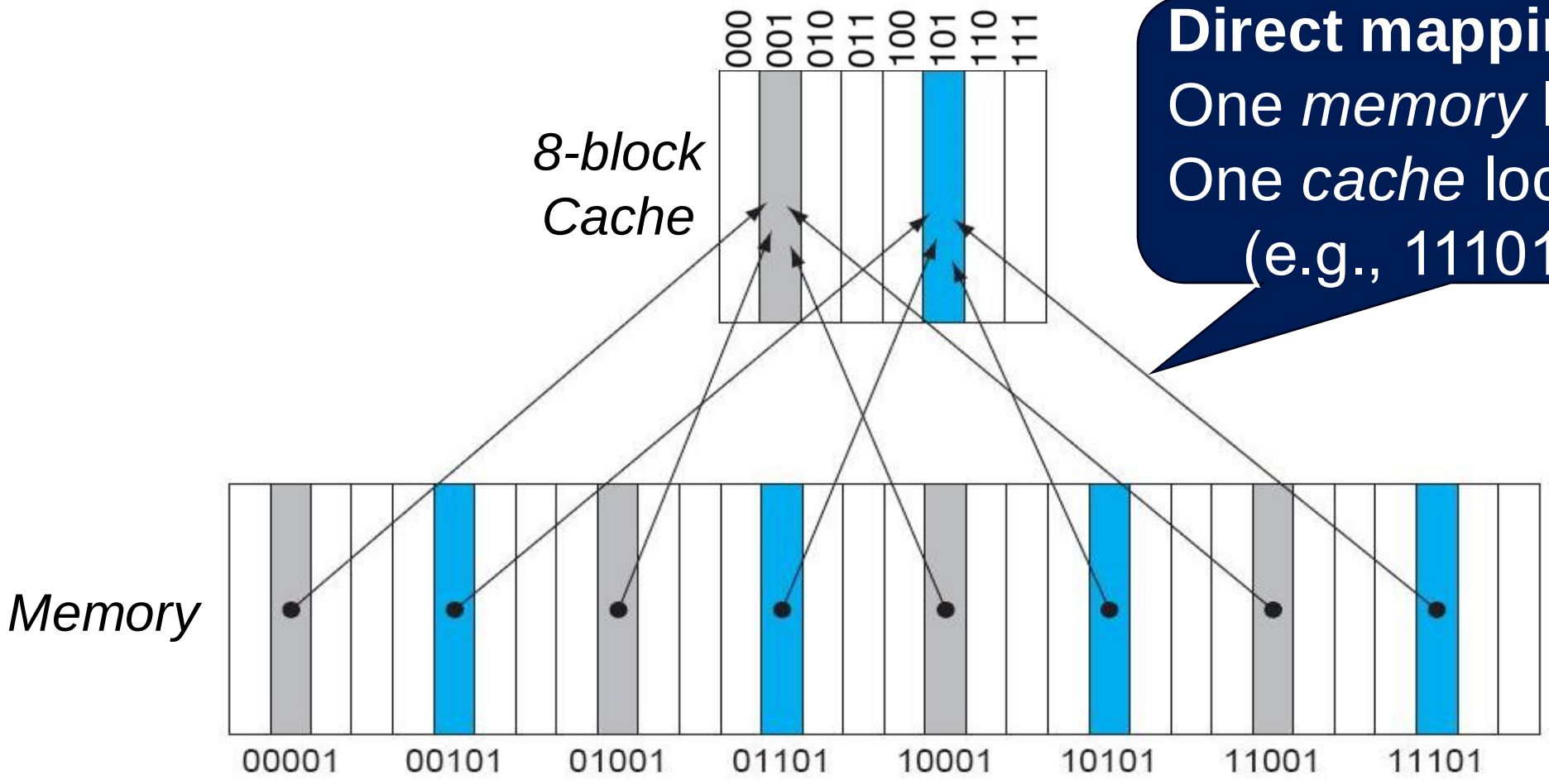
Direct Mapped Cache

Direct Mapped Cache

Each *memory* location is mapped to <u>exactly one location</u> in the *cache*

Direct Mapped Cache

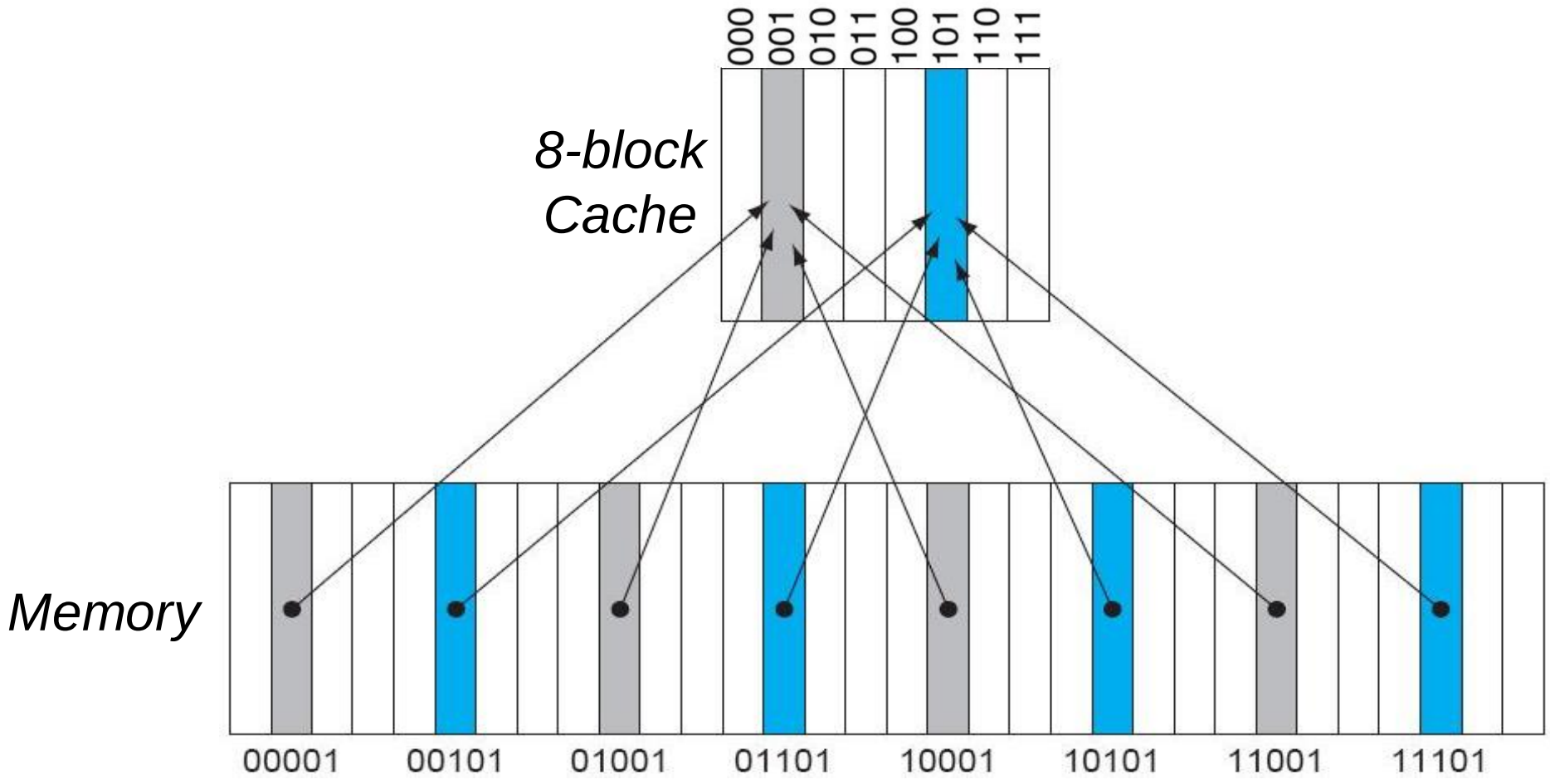
Each *memory* location is mapped to <u>exactly one location</u> in the *cache*



Direct mapping:
One *memory* location → One *cache* location
(e.g., 11101 → 101)

Mapping Algorithm

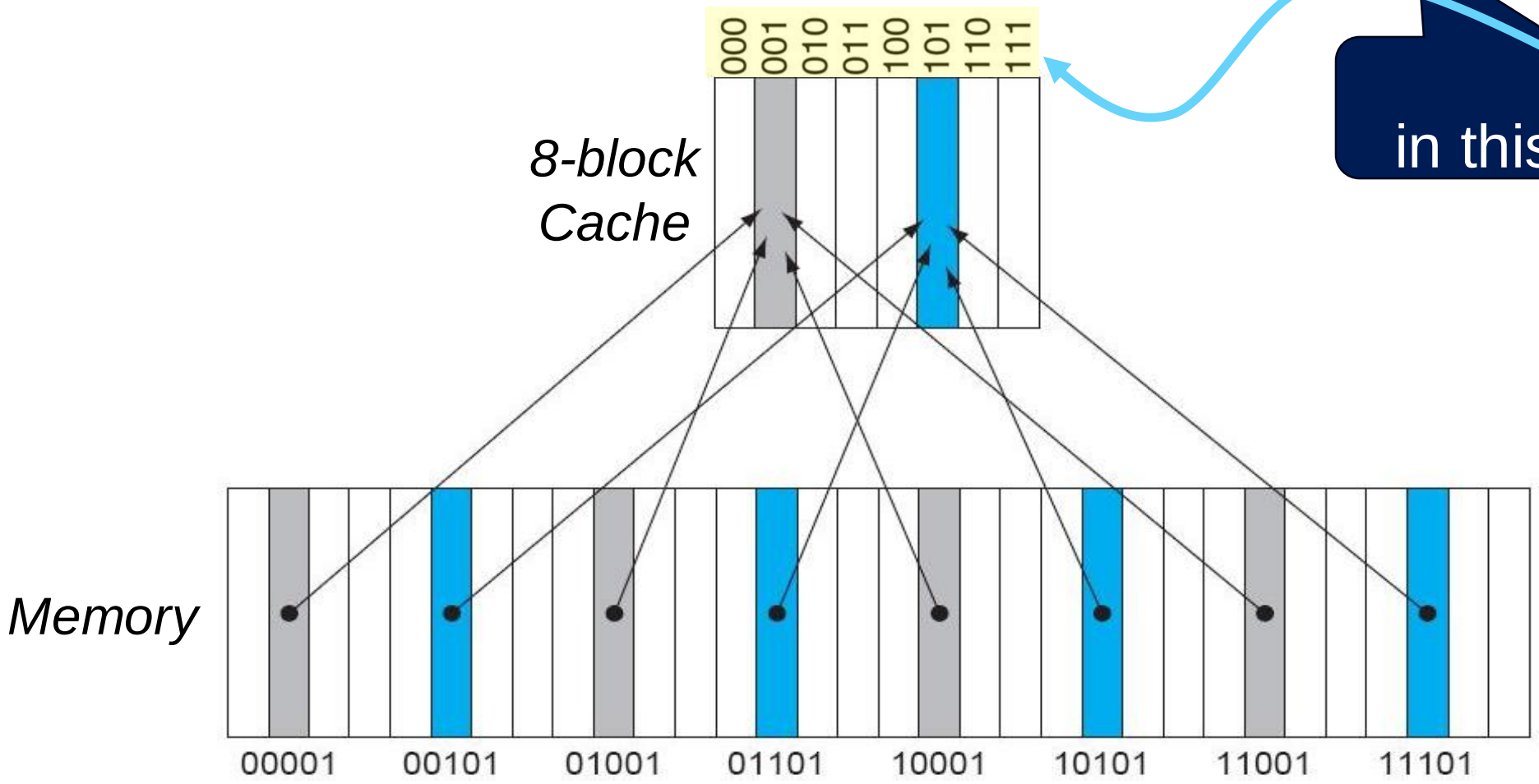
Location in the cache = (**Block address**) modulo (**# Blocks in cache**)



Mapping Algorithm

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

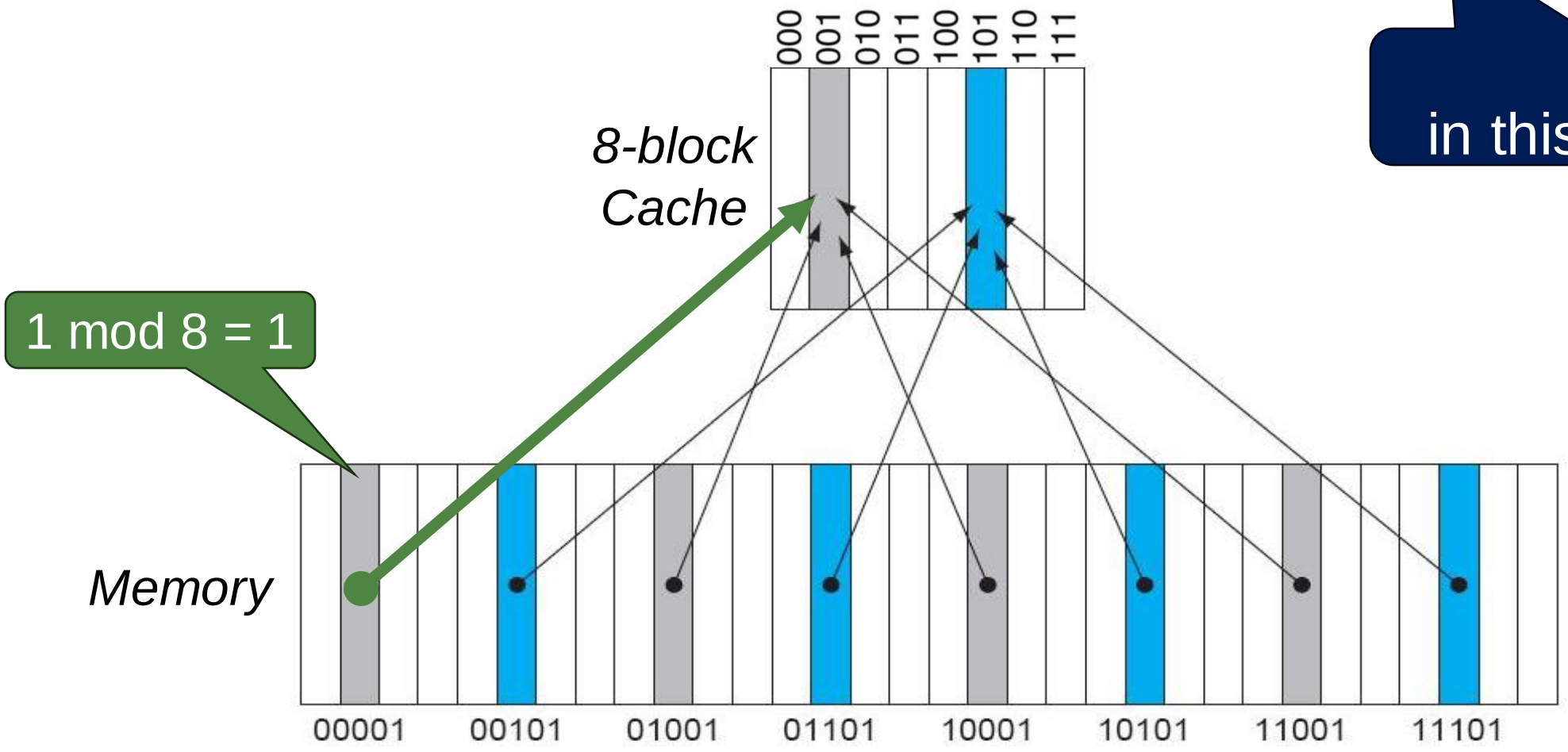
8
in this example



Mapping Algorithm

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

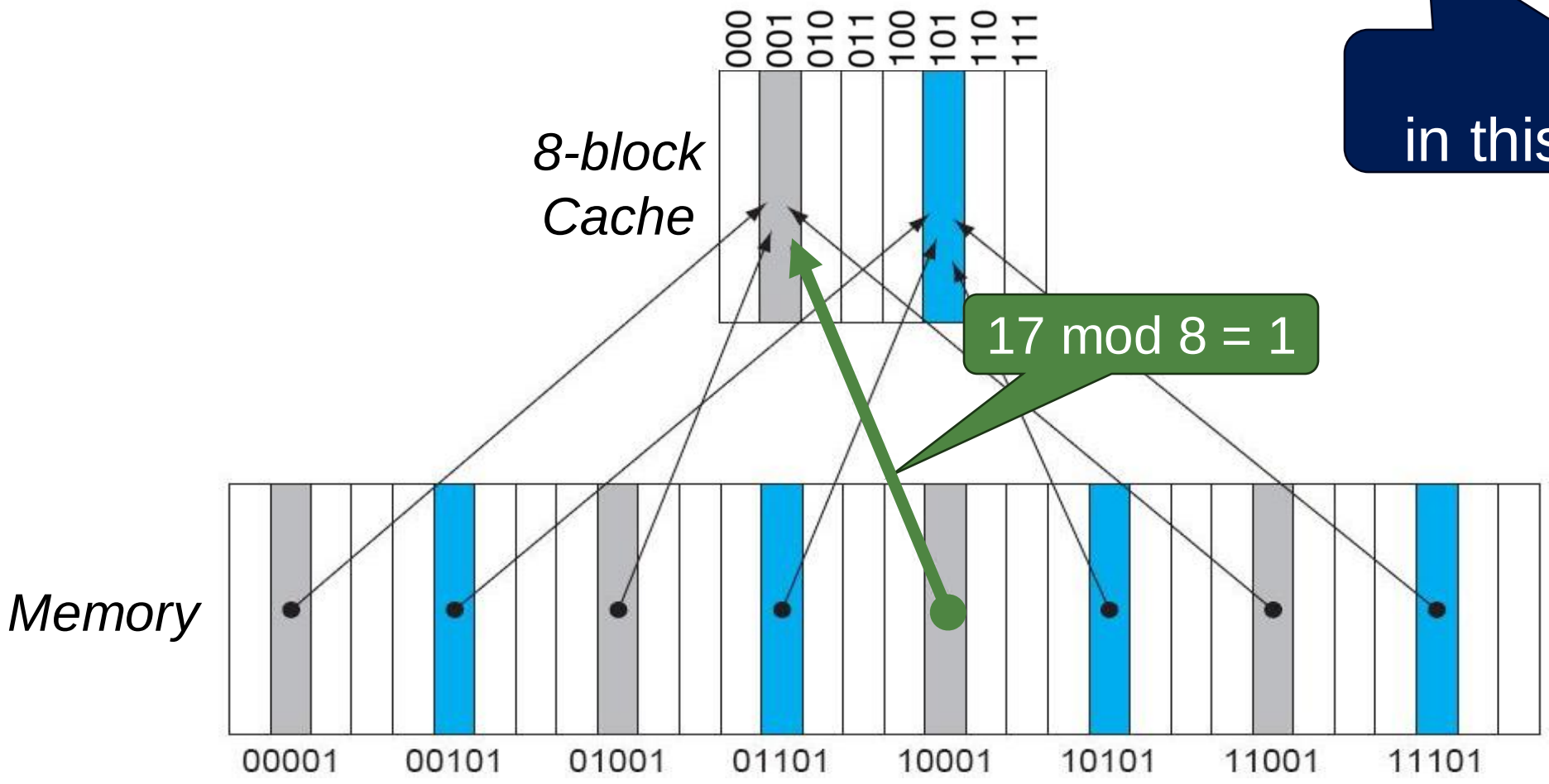
8
in this example



Mapping Algorithm

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

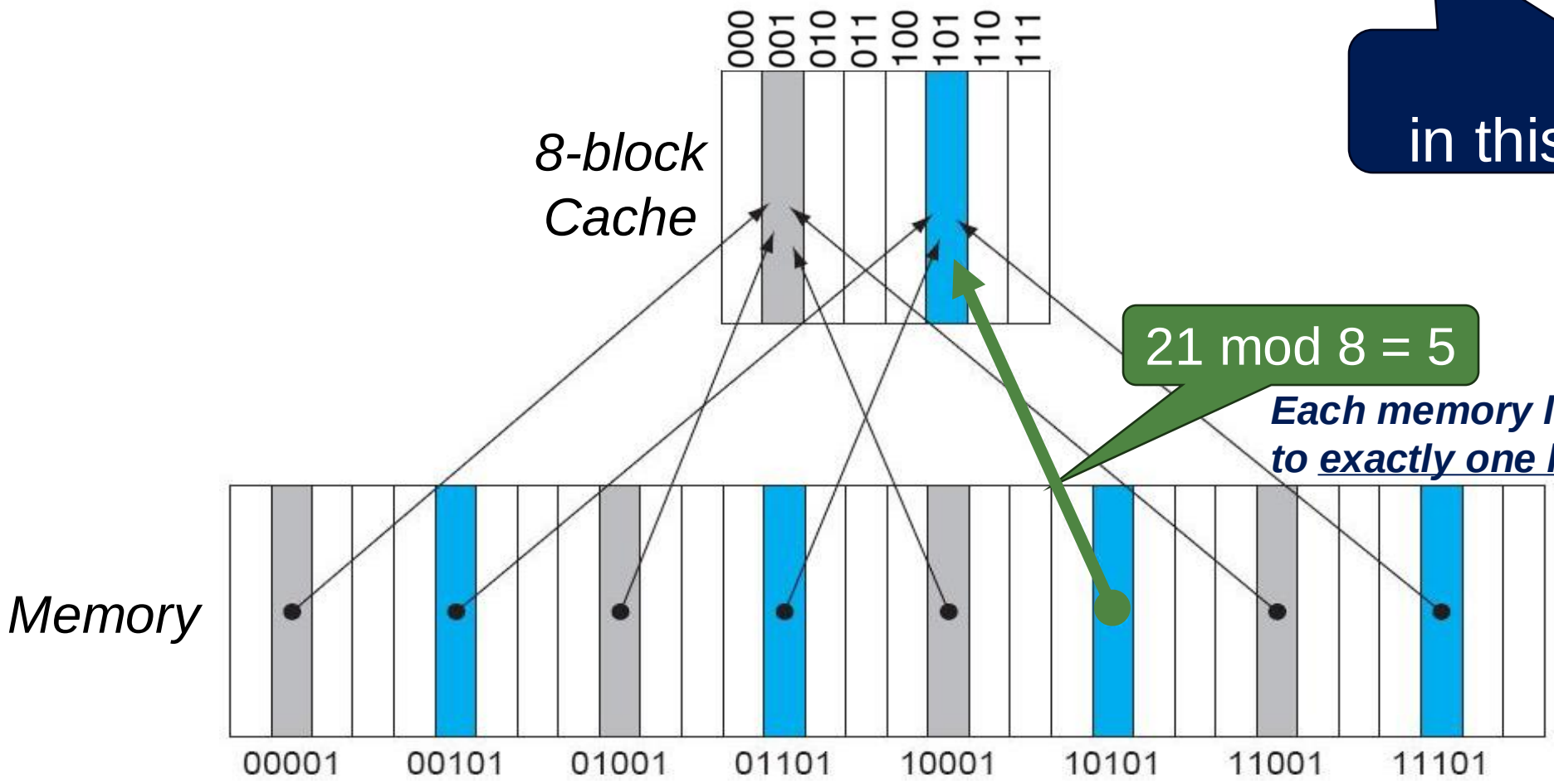
8
in this example



Mapping Algorithm

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

8
in this example



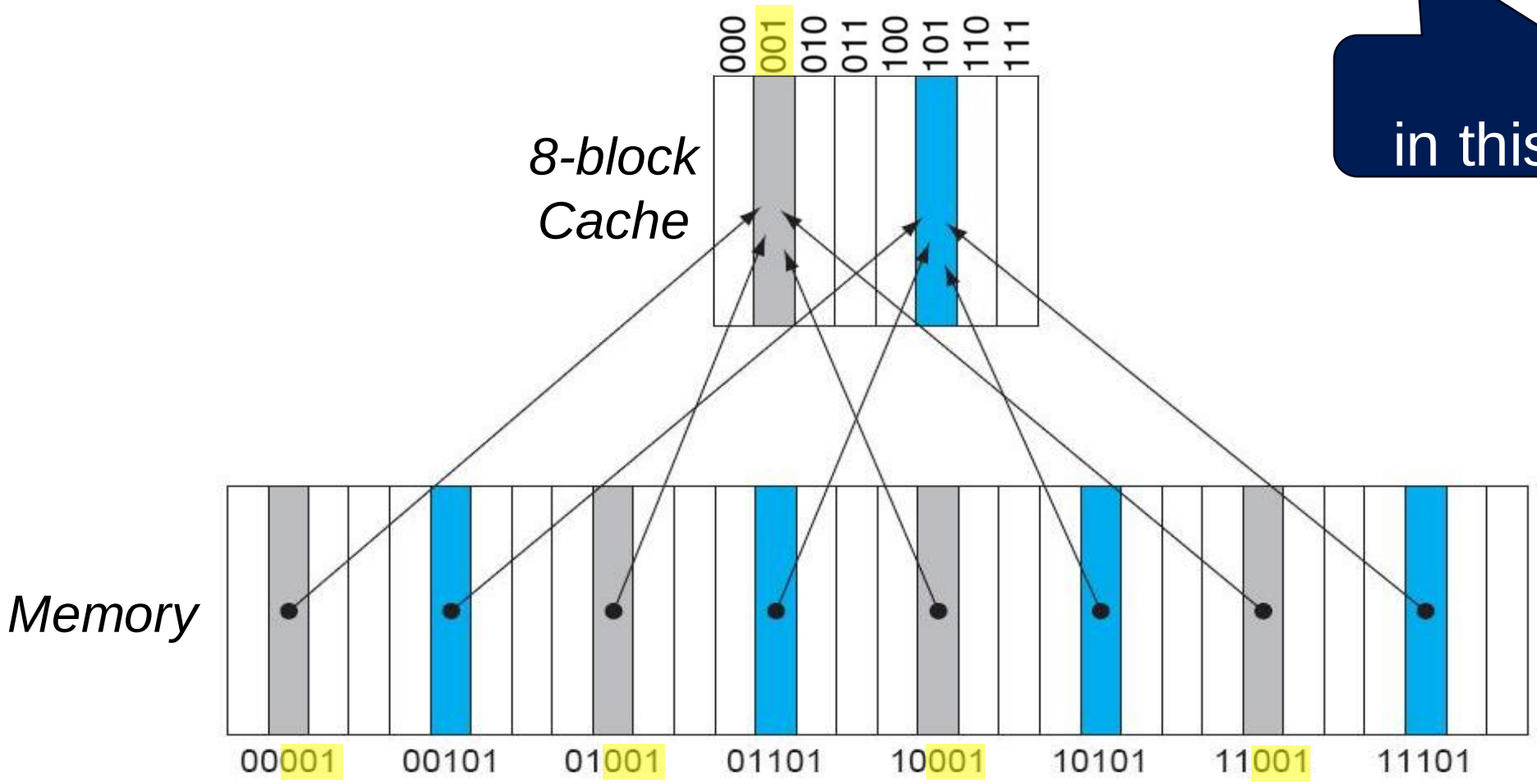
21 mod 8 = 5

Each memory location is mapped to <u>exactly one location</u> in the cache

Mapping Algorithm: Binary View

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

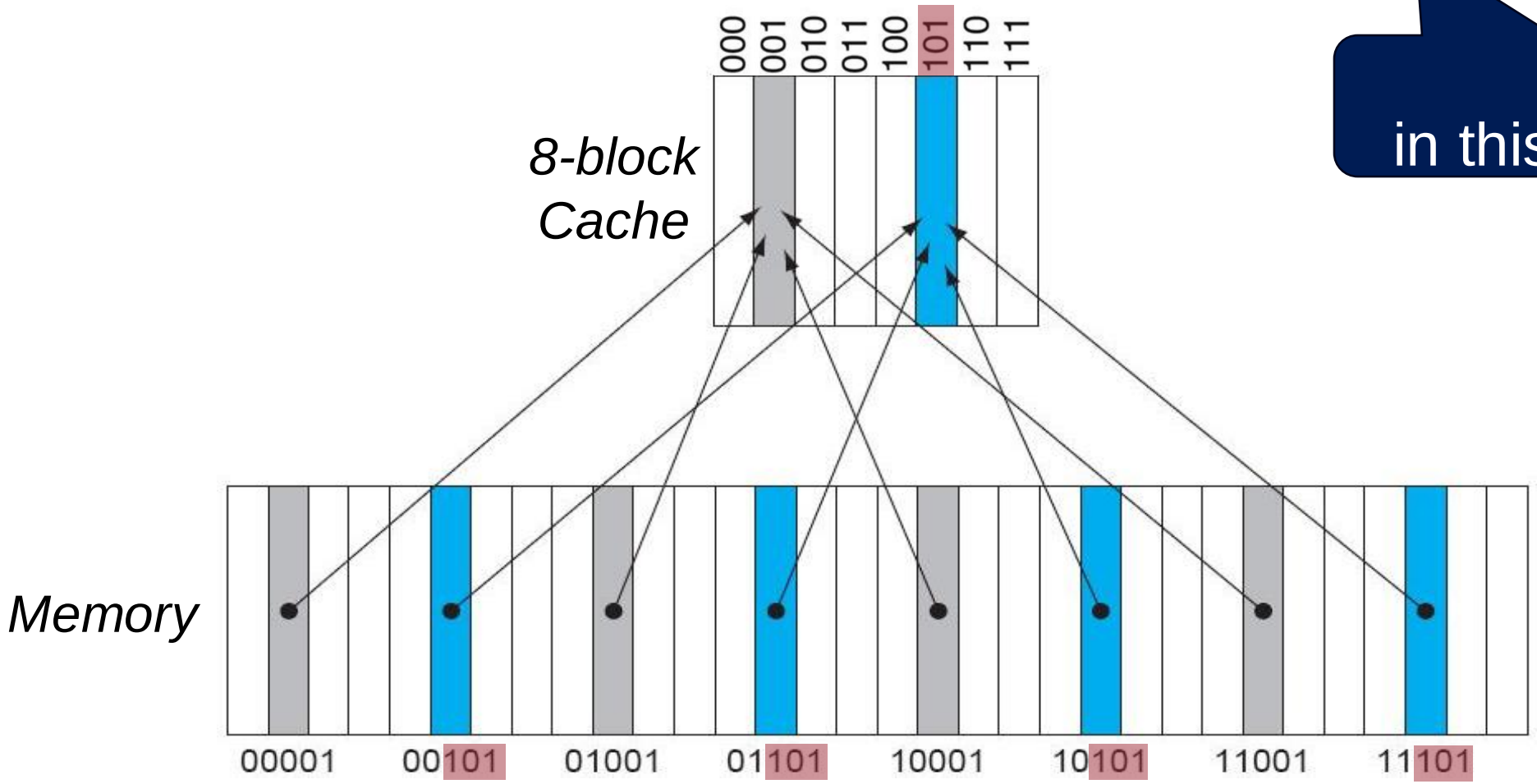
8
in this example



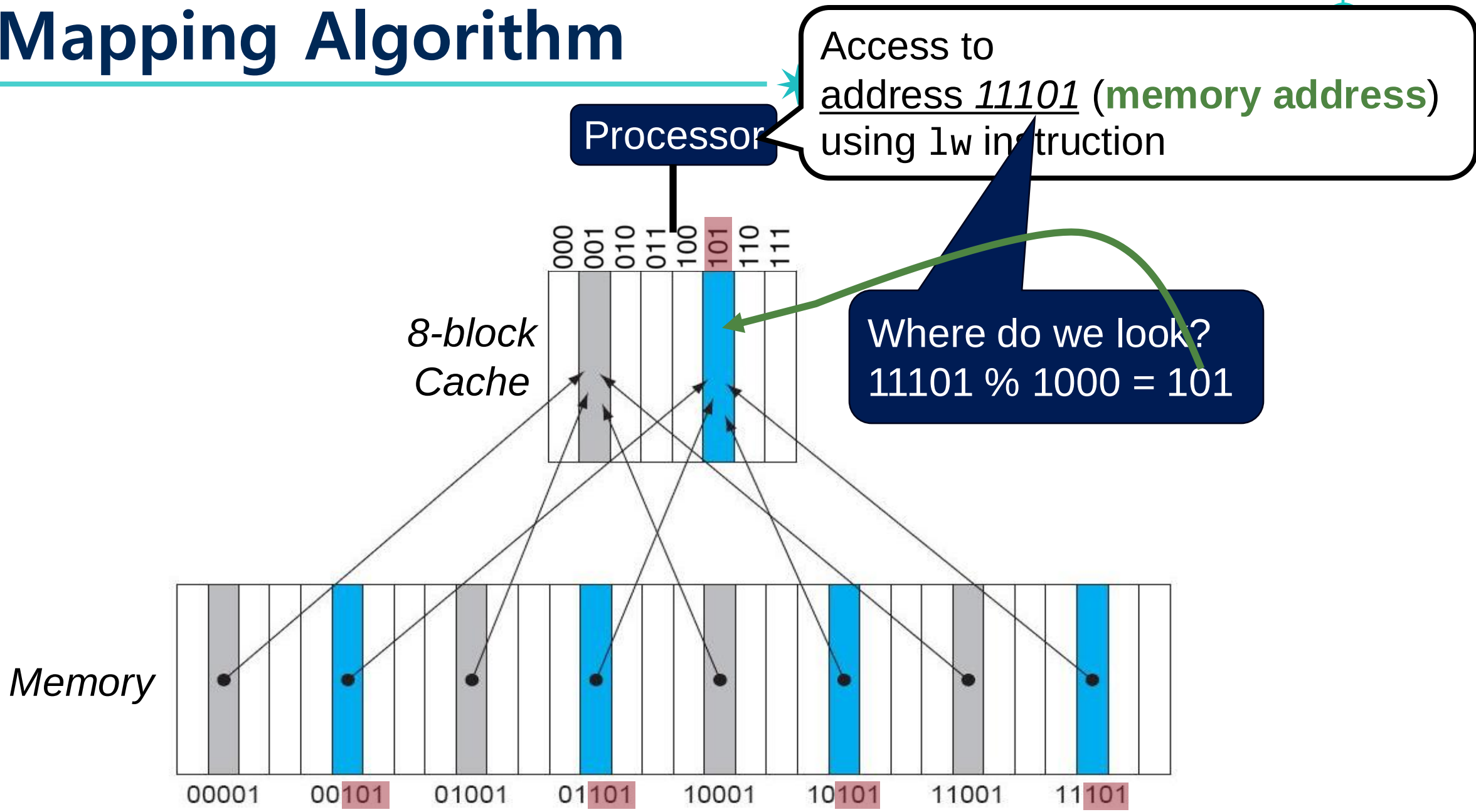
Mapping Algorithm: Binary View

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

8
in this example

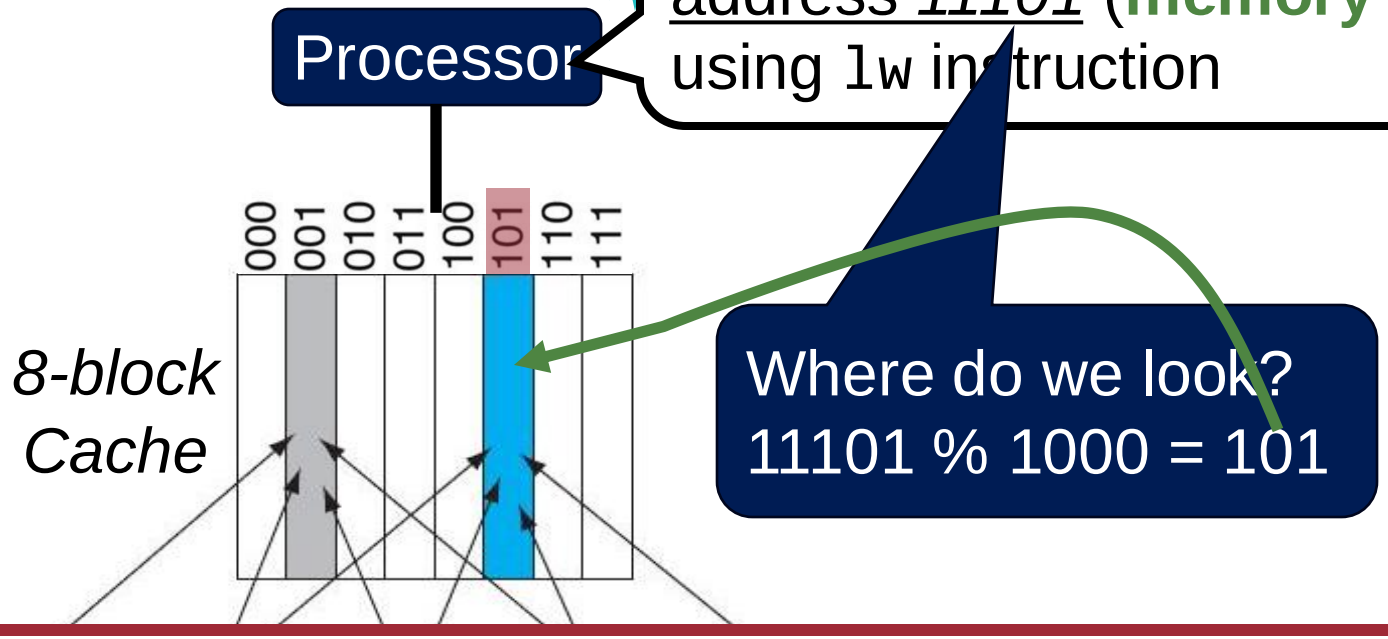


Mapping Algorithm



Mapping Algorithm

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

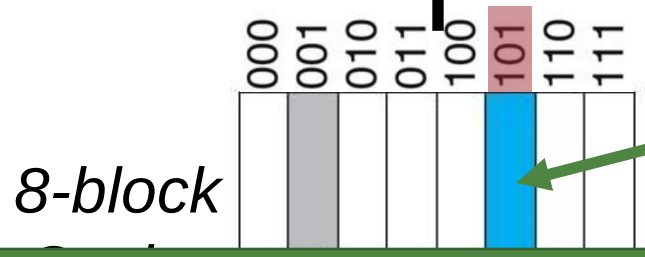


The value might not be from 11101
but from another location (e.g., 00101).
How can you confirm that it's the value from 11101?

Mapping Algorithm

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

Processor



8-block

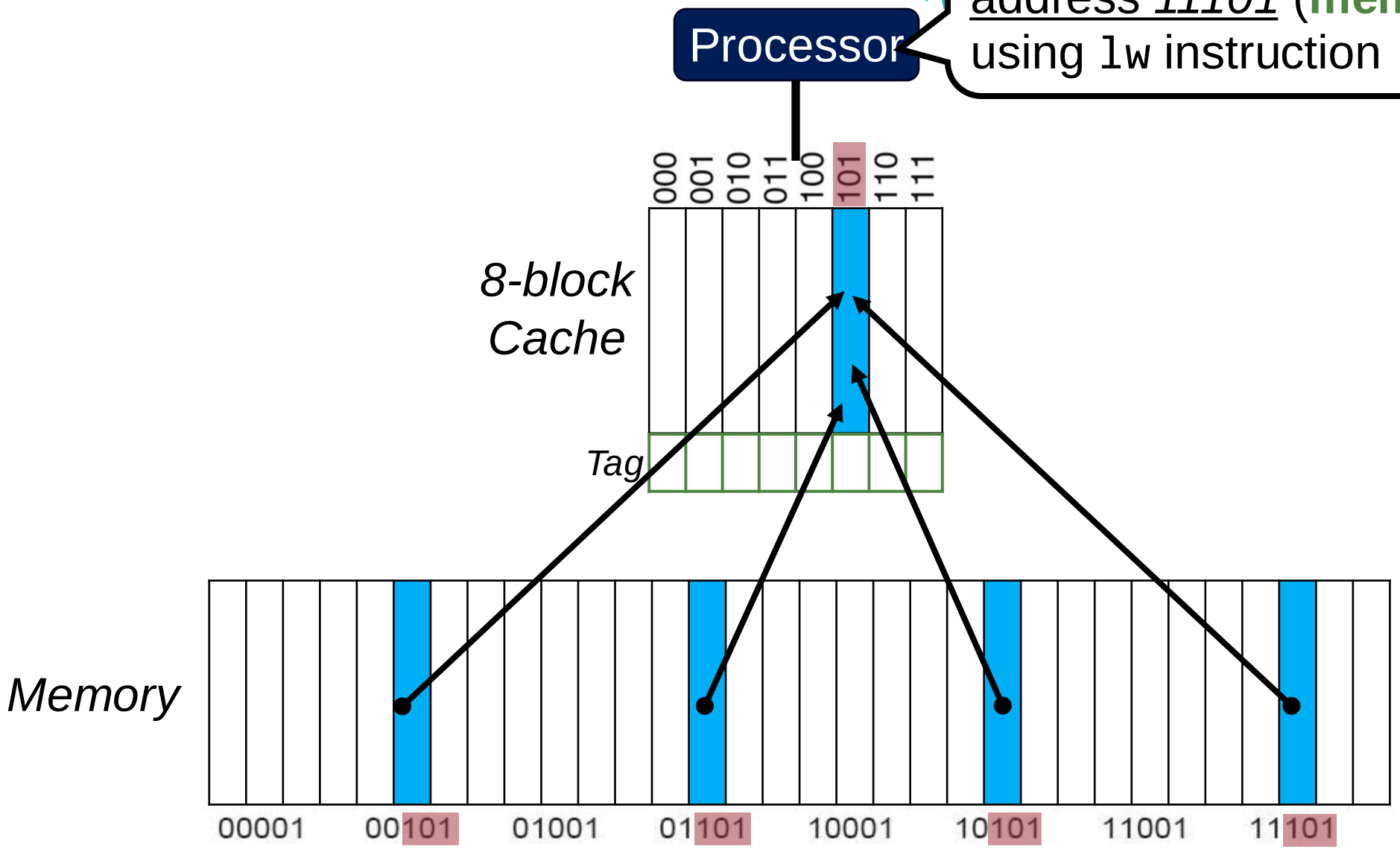
Where do we look?
11101 % 1000 = 101

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?

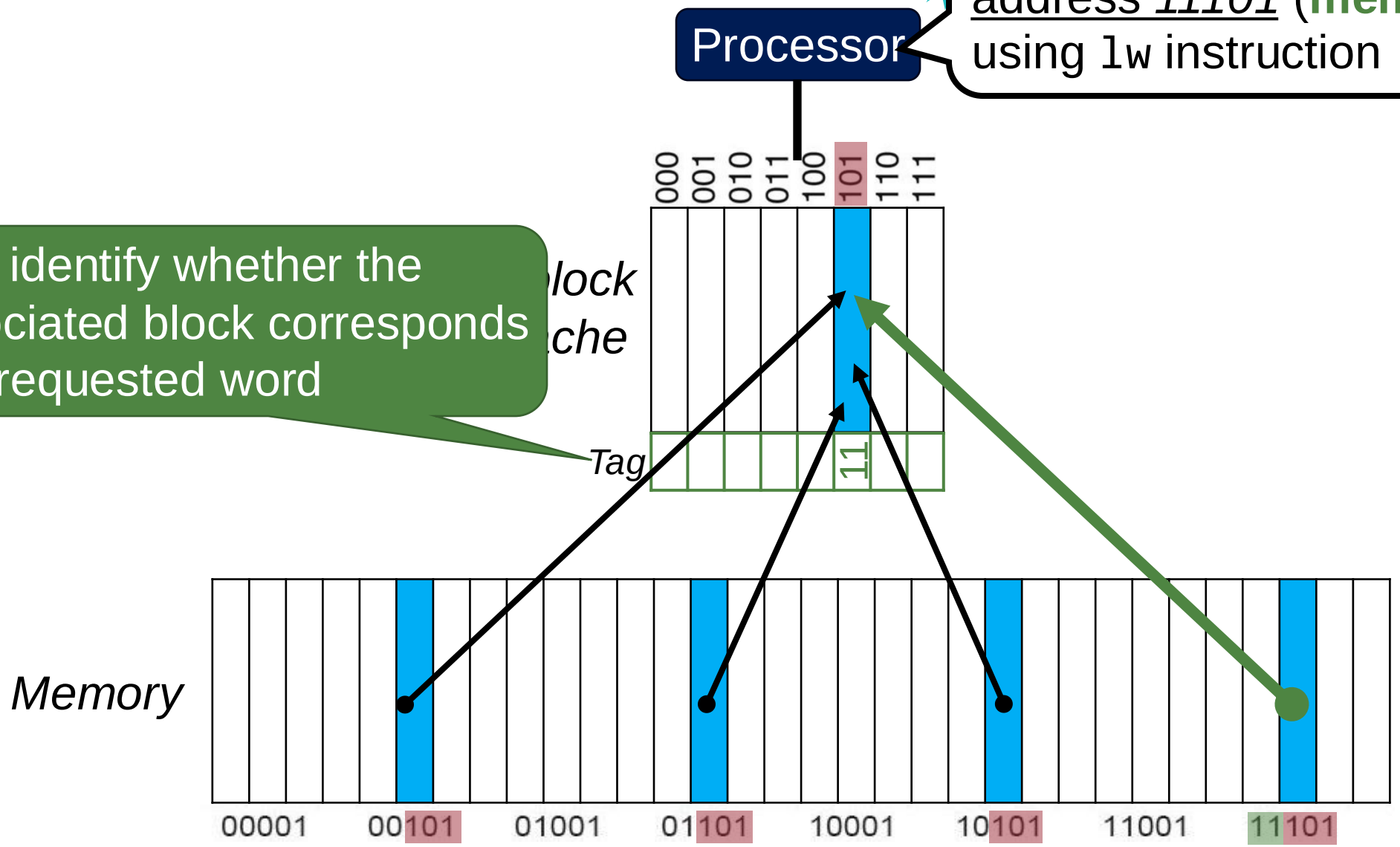
Additional Field: Tag Bits

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



Additional Field: Tag Bits

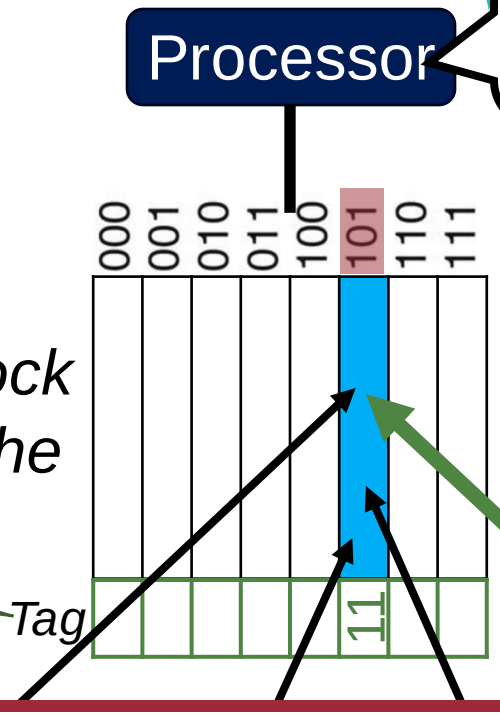
Access to address *11101* (**memory address**) using `lw` instruction



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

Additional Field: Tag Bits

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

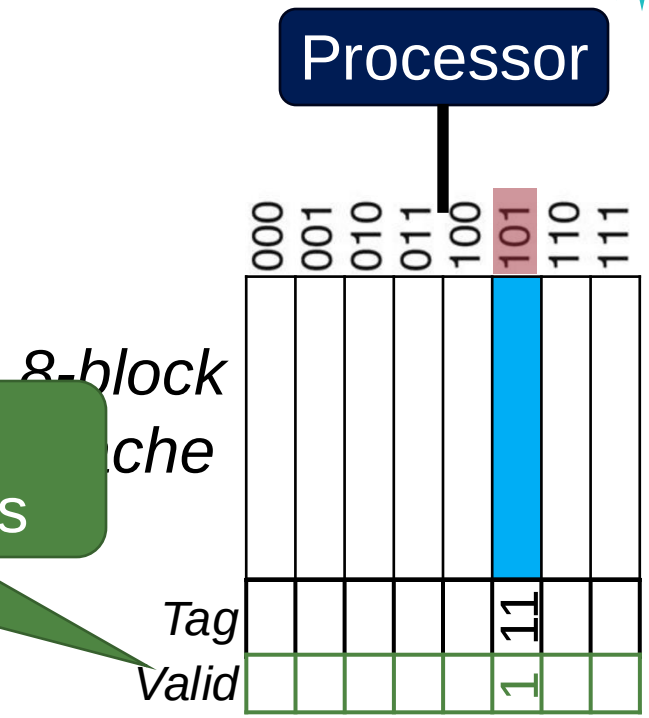


Tag: identify whether the associated block corresponds to a requested word

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)

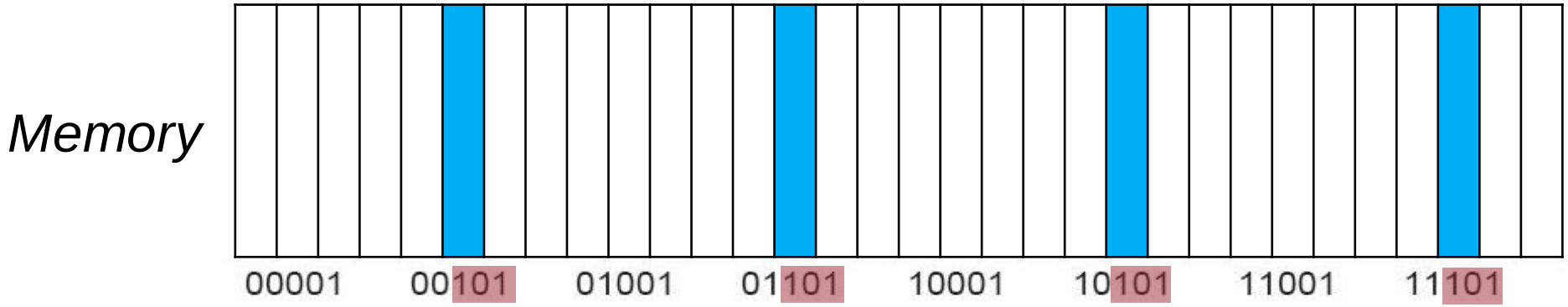
Additional Field: Valid Bit



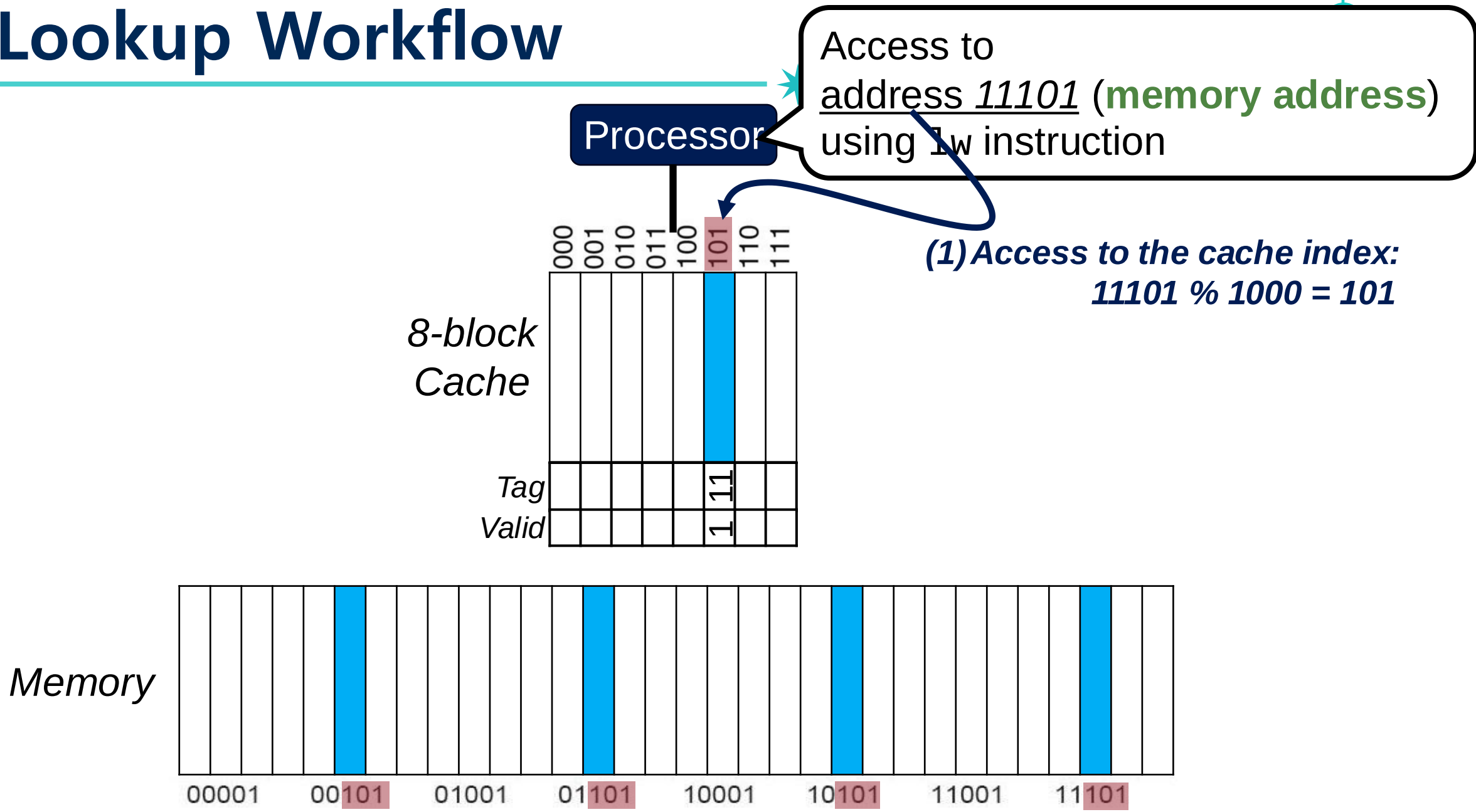
Valid: Indicate whether an entry contains valid address

- 1 = valid
- 0 = Invalid

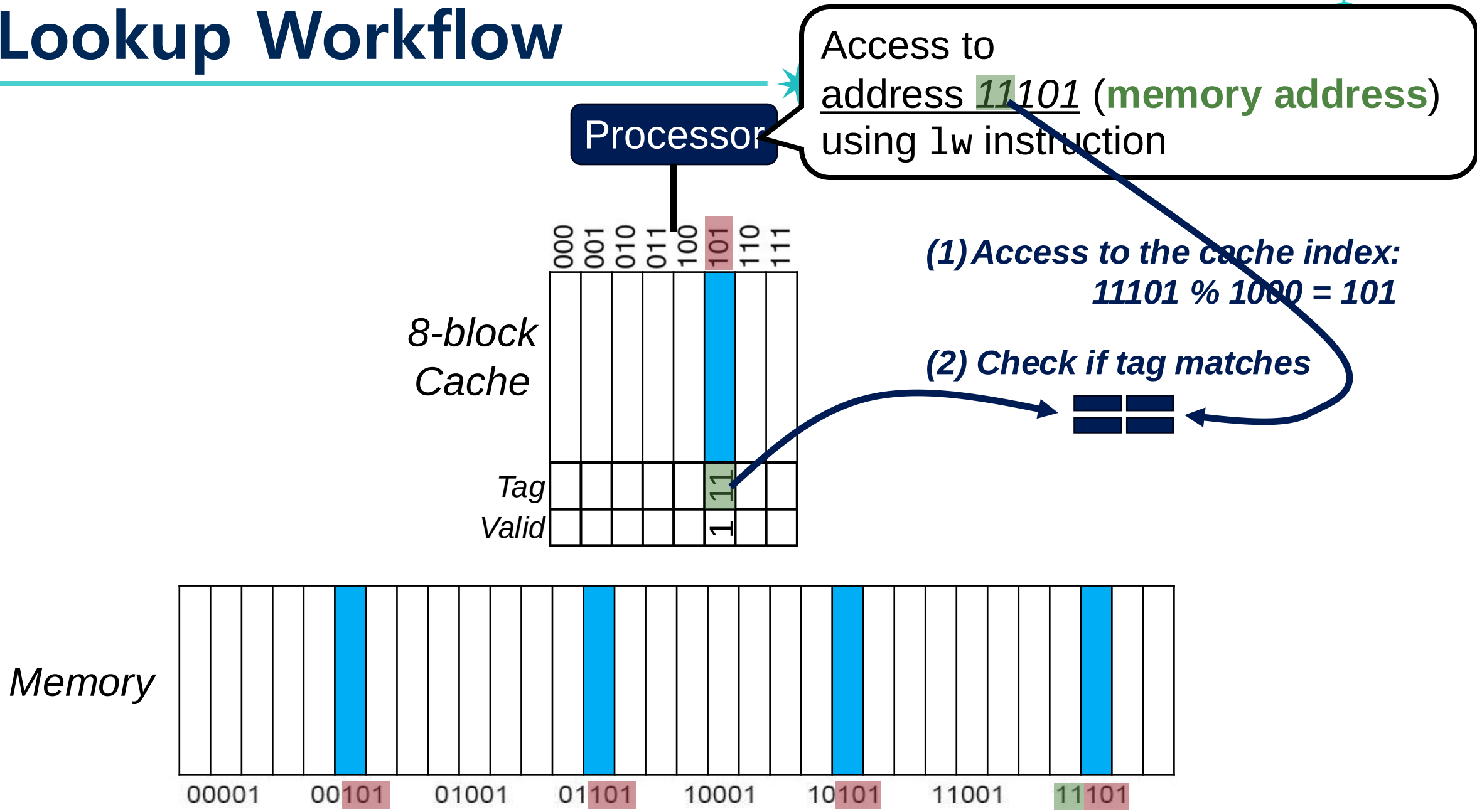
(Initially 0)



Lookup Workflow

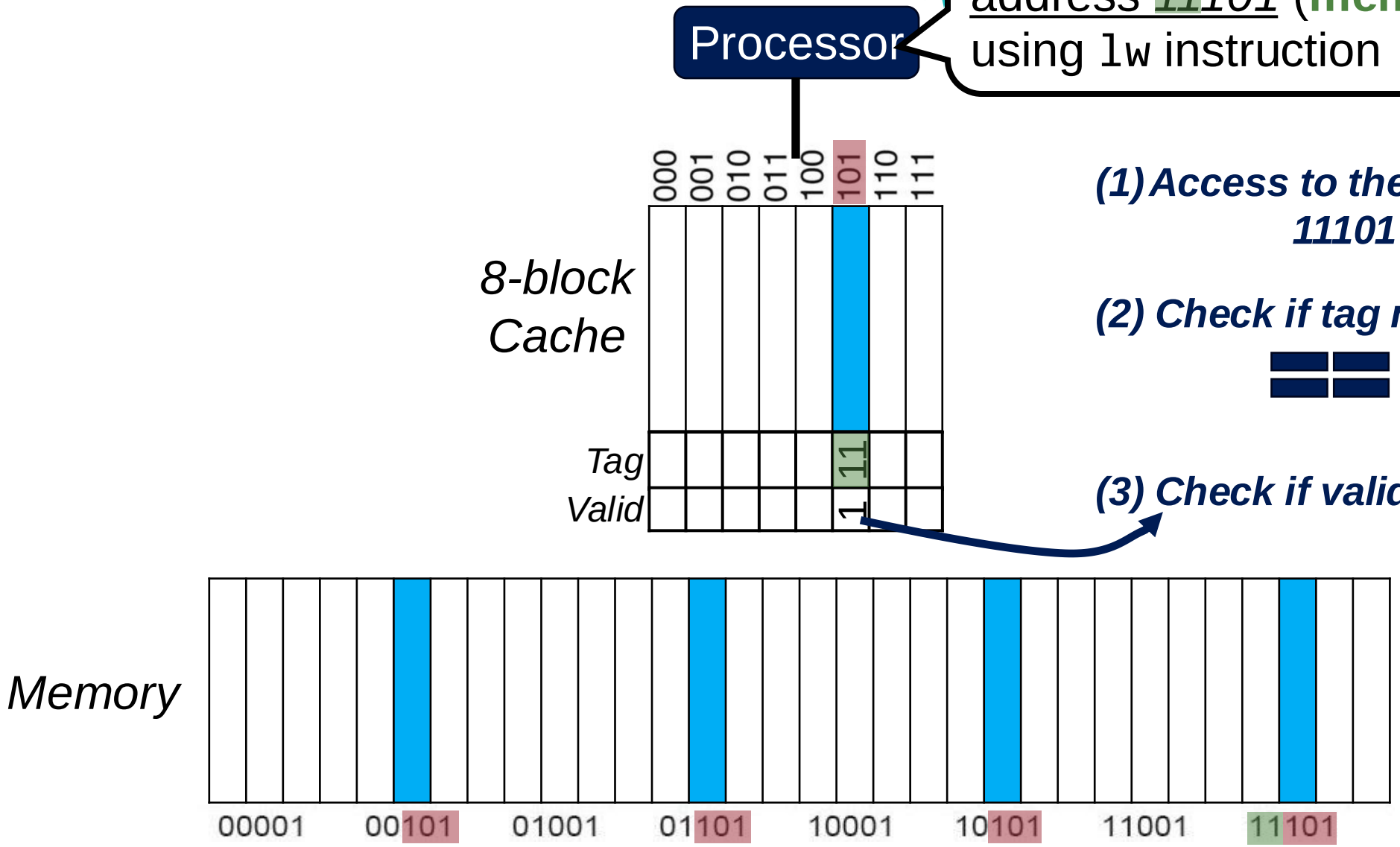


Lookup Workflow



Lookup Workflow

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



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

(2) Check if tag matches

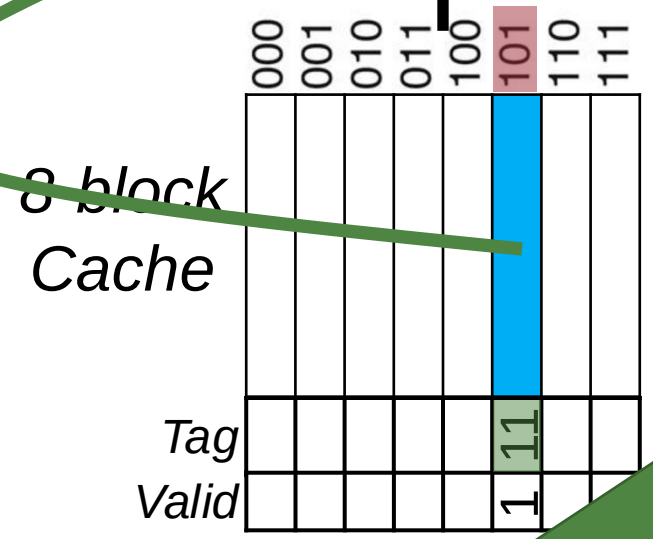
(3) Check if valid bit sets

Lookup Workflow

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

Processor

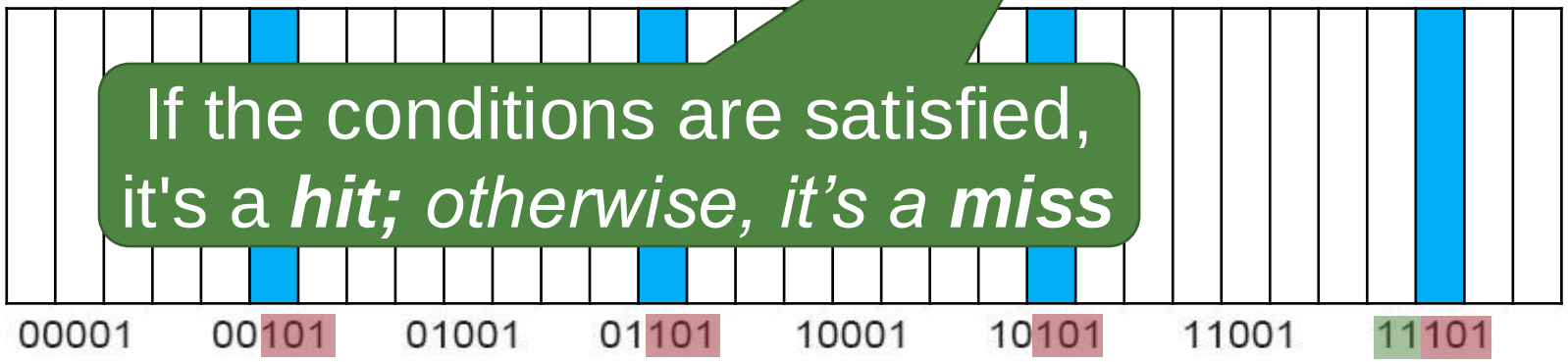
(4) If it's a hit: access the cache data *(no need to access the memory)*



(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 Example

Memory access sequence:

(1) 10110
(2) 11010
(3) 10000
(4) 00011
(5) 10010
(6) 10110

The initial state of the cache after power-on

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110
(2) 11010
(3) 10000
(4) 00011
(5) 10010
(6) 10110

a. The initial state of the cache after power-on

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110
(2) 11010
(3) 10000
(4) 00011
(5) 10010
(6) 10110

b. After handling a miss of address 10110

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110

Miss (2) 11010

(3) 10000

(4) 00011

(5) 10010

(6) 10110

b. After handling a miss of address 10110

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110

Miss (2) 11010

(3) 10000

(4) 00011

(5) 10010

(6) 10110

c. After handling a miss of address 11010

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory(11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110

Miss (2) 11010

Miss (3) 10000

(4) 00011

(5) 10010

(6) 10110

d. After handling a miss of address 10000

Index	Valid	Tag	Data
000	Y	10_{two}	Memory(10000_{two})
001	N		
010	Y	11_{two}	Memory(11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110

Miss (2) 11010

Miss (3) 10000

Miss (4) 00011

(5) 10010

(6) 10110

e. After handling a miss of address 00011

Index	Valid	Tag	Data
000	Y	10_{two}	Memory(10000_{two})
001	N		
010	Y	11_{two}	Memory(11010_{two})
011	Y	00_{two}	Memory(00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110
Miss (2) 11010
Miss (3) 10000
Miss (4) 00011
Miss (5) 10010
(6) 10110

e. After handling a miss of address 00011

Index	Valid	Tag	Data
000	Y	10_{two}	Memory(10000_{two})
001	N		
010	Y	$10 \neq 11_{two}$	Memory(11010_{two})
011	Y	00_{two}	Memory(00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

Miss (1) 10110
Miss (2) 11010
Miss (3) 10000
Miss (4) 00011
Miss (5) 10010
(6) 10110

f. After handling a miss of address 10010

Index	Valid	Tag	Data
000	Y	10_{two}	Memory(10000_{two})
001	N		
010	Y	10_{two}	Memory(10010_{two})
011	Y	00_{two}	Memory(00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Direct Mapped Cache Example

Memory access sequence:

- Miss (1) 10110
- Miss (2) 11010
- Miss (3) 10000
- Miss (4) 00011
- Miss (5) 10010
- Hit (6) 10110

Index	Valid	Tag	Data
000	Y	10_{two}	Memory(10000_{two})
001	N		
010	Y	10_{two}	Memory(10010_{two})
011	Y	00_{two}	Memory(00011_{two})
100	N		
101	N		
110	**Y**	**10_{two}**	Memory(10110_{two})
111	N		

Direct Mapped Cache Structure

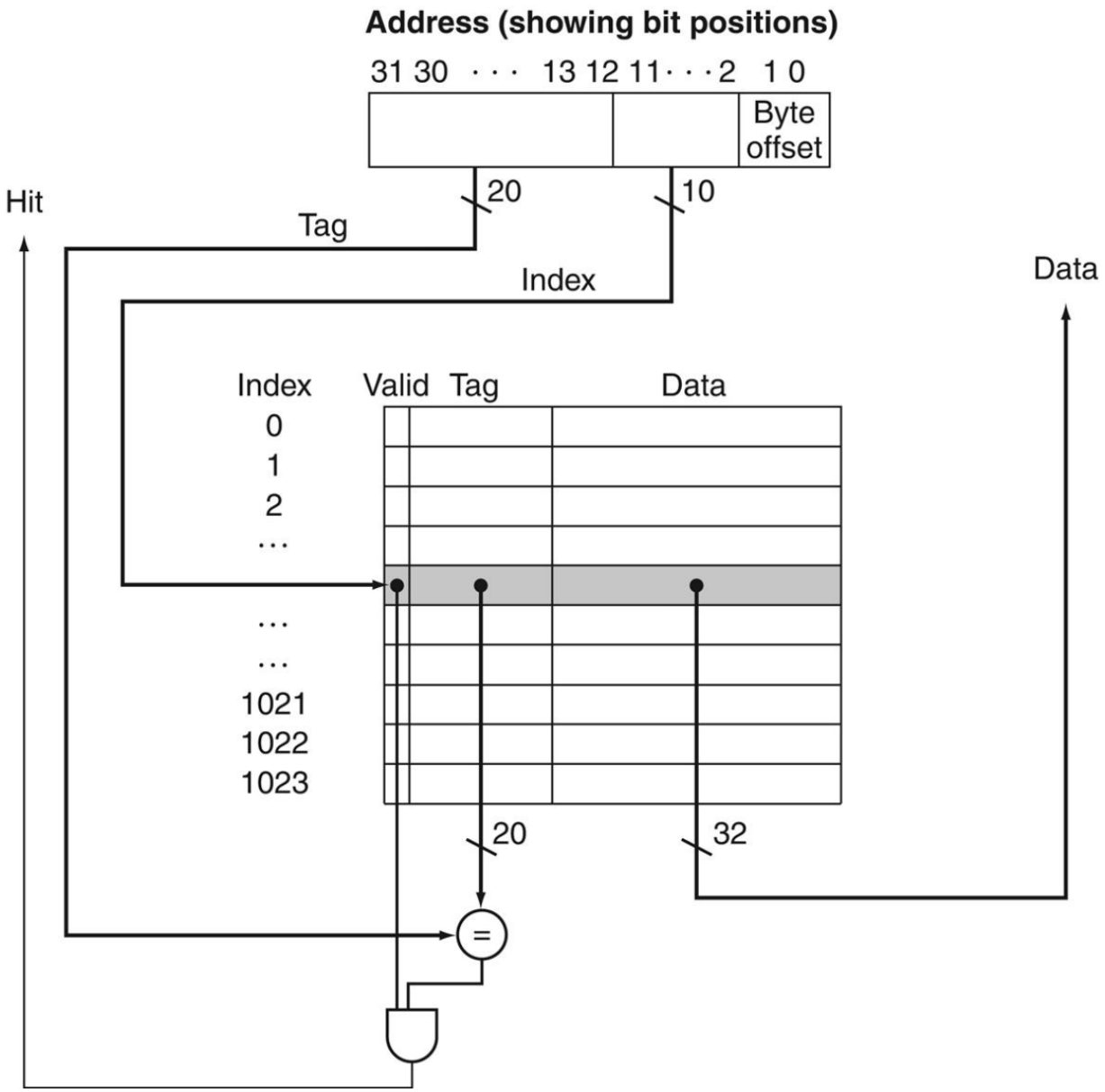
- **Assumption**:
  - Each block is 1 word (4 bytes)
  - Cache holds 1024 words as data

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

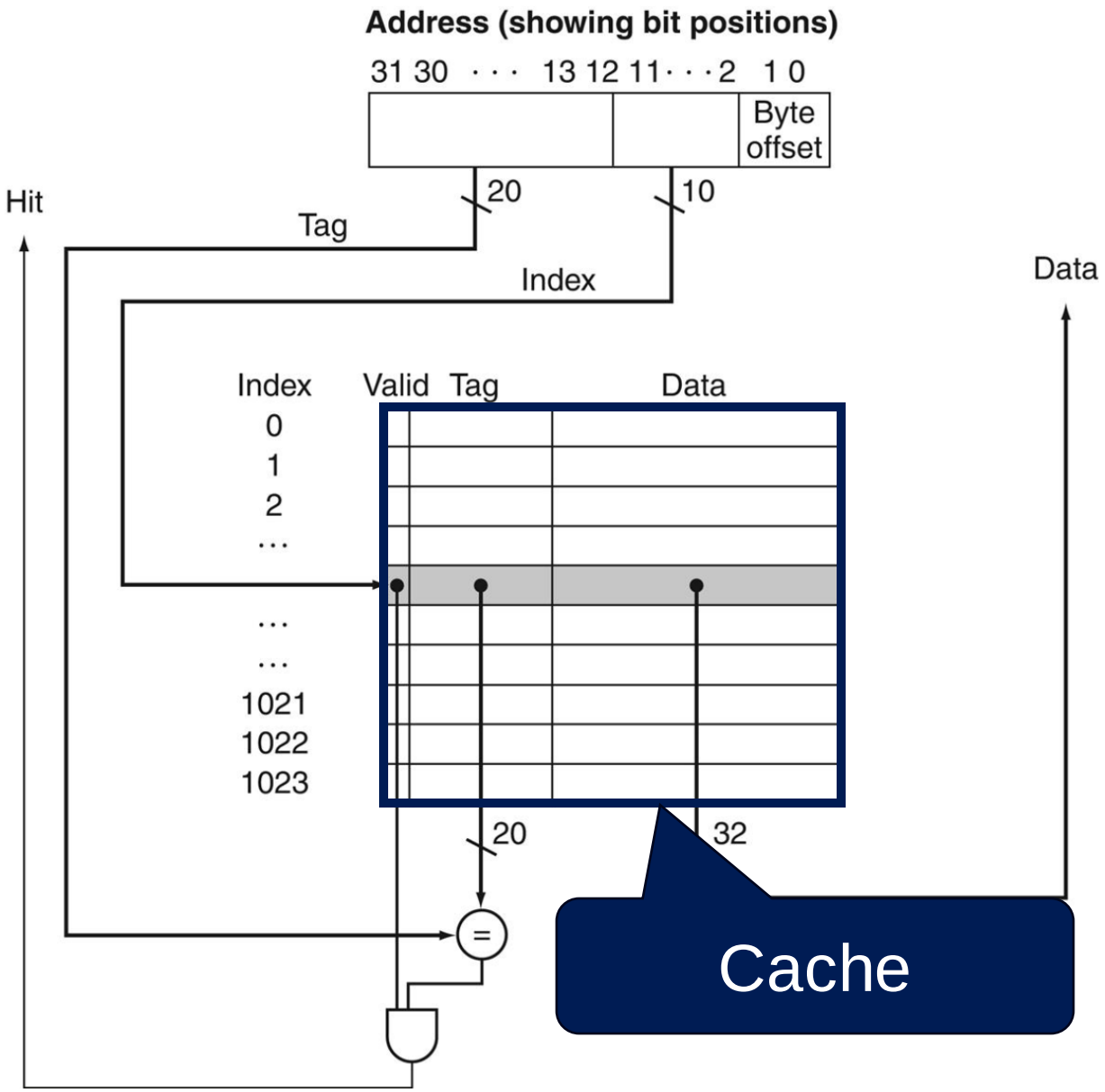
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

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

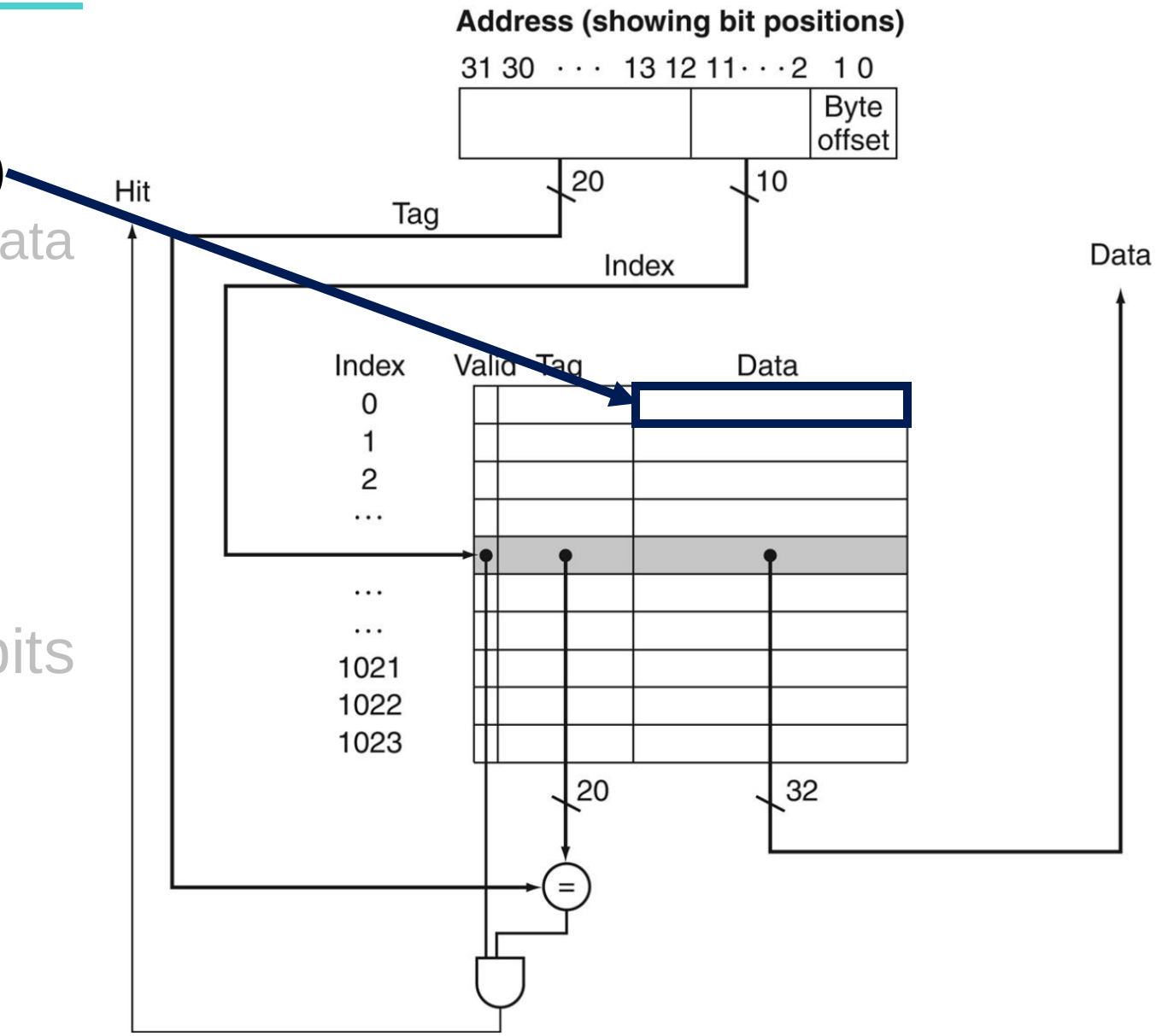
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

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

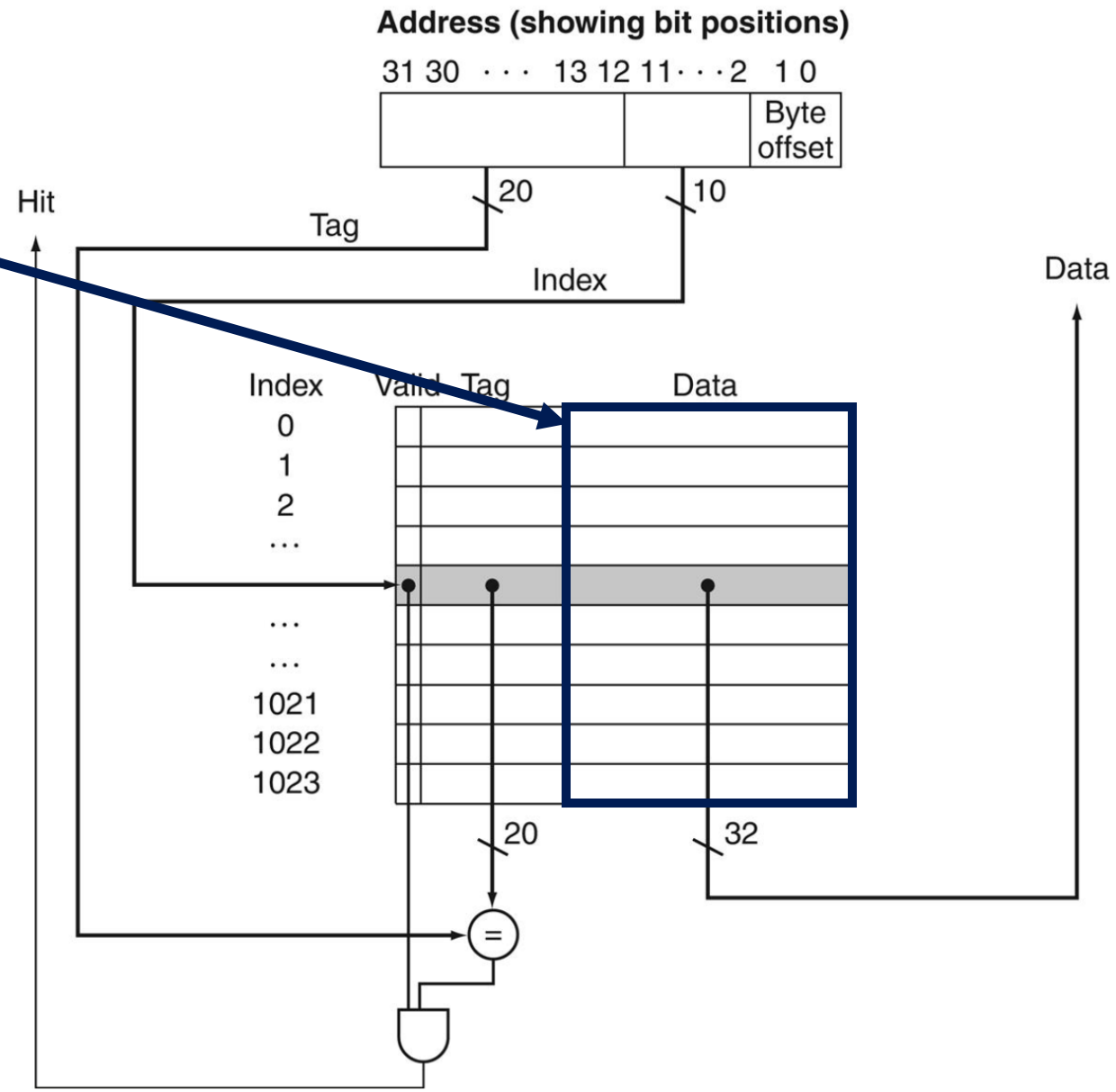
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

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

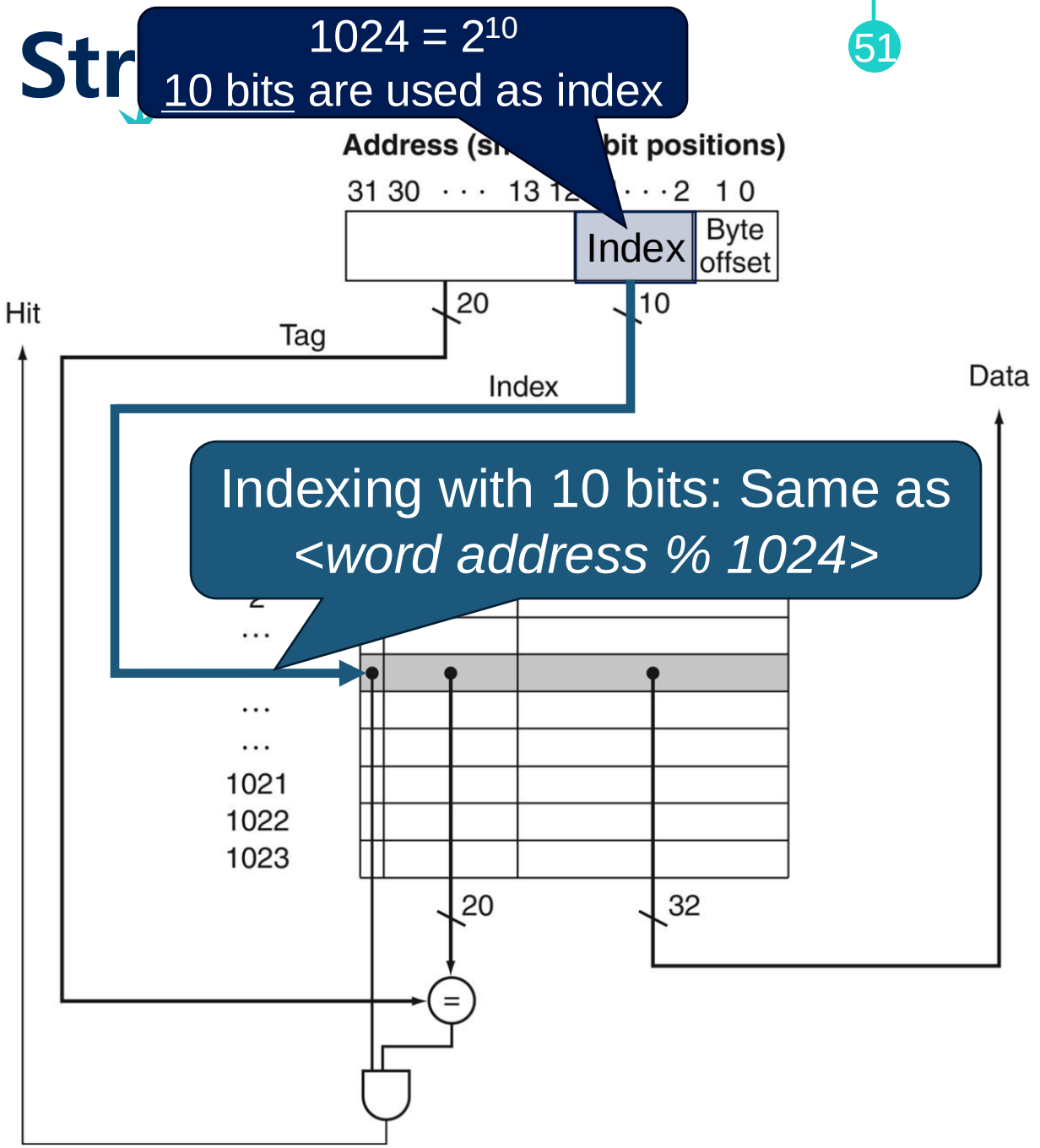
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Str

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

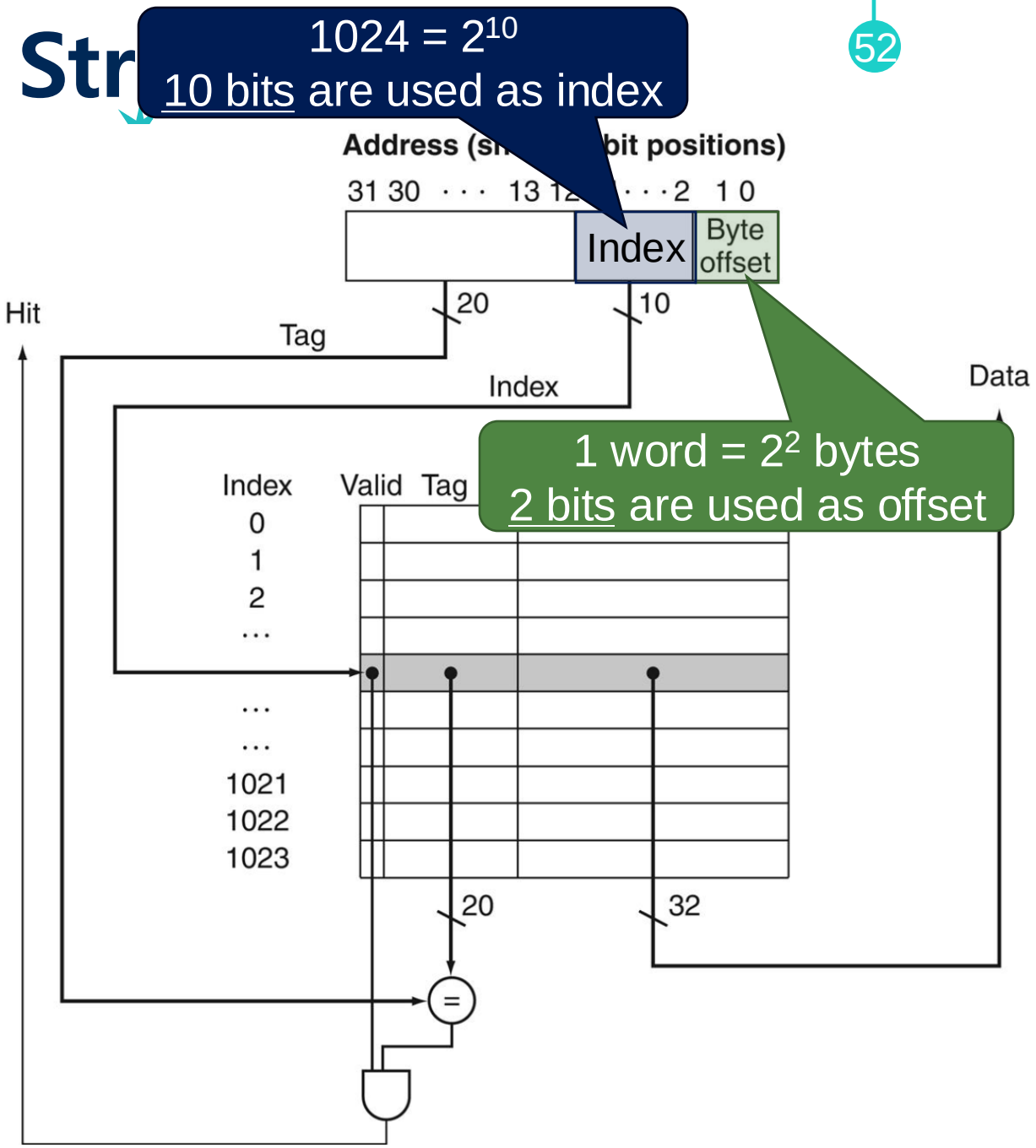
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Str

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

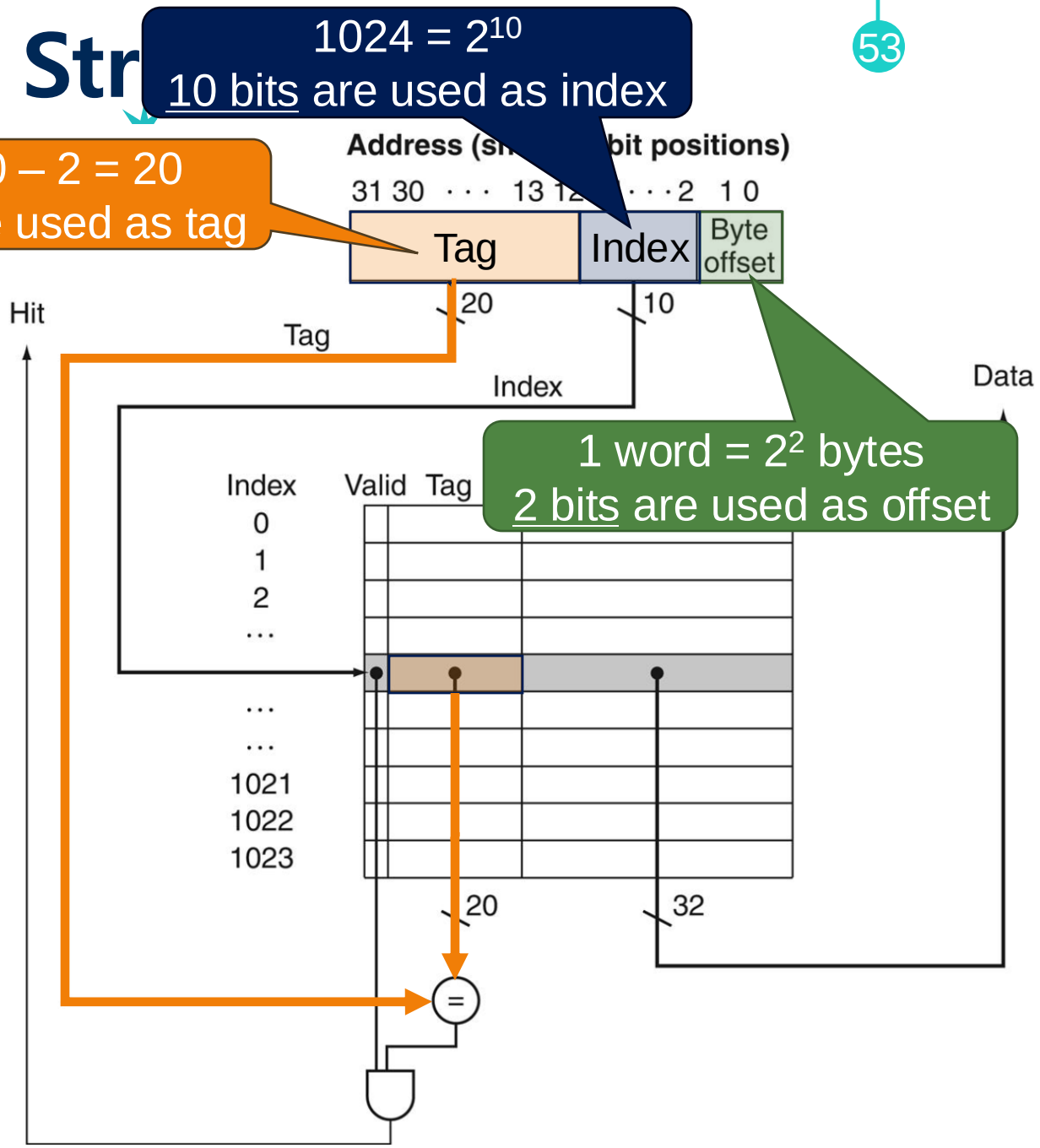
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Str

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

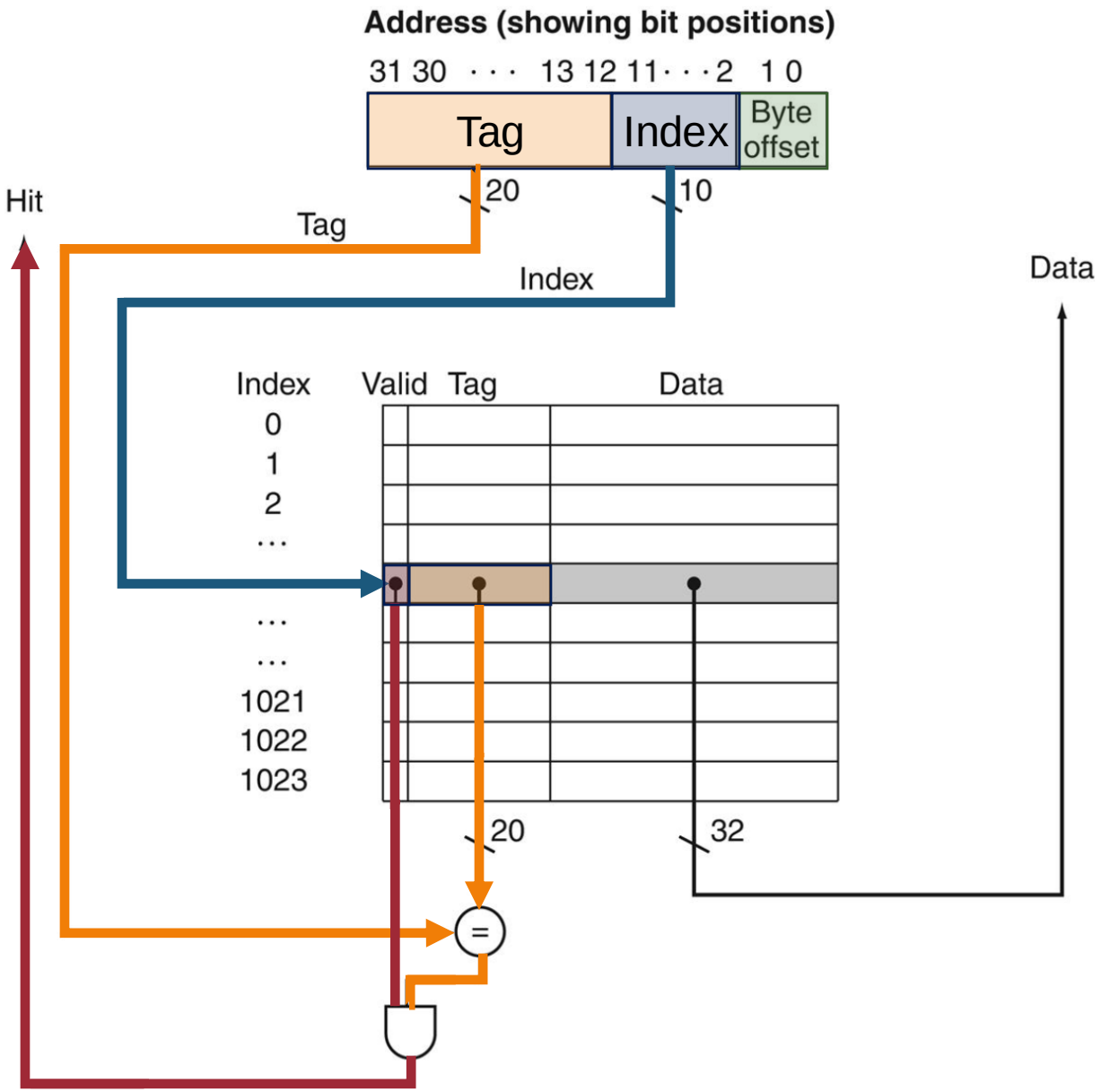
- How should the memory address be structured?
- What is the tag field size?
- What is the total number of bits in cache?



Direct Mapped Cache Structure

- **Assumption**:
  - 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?

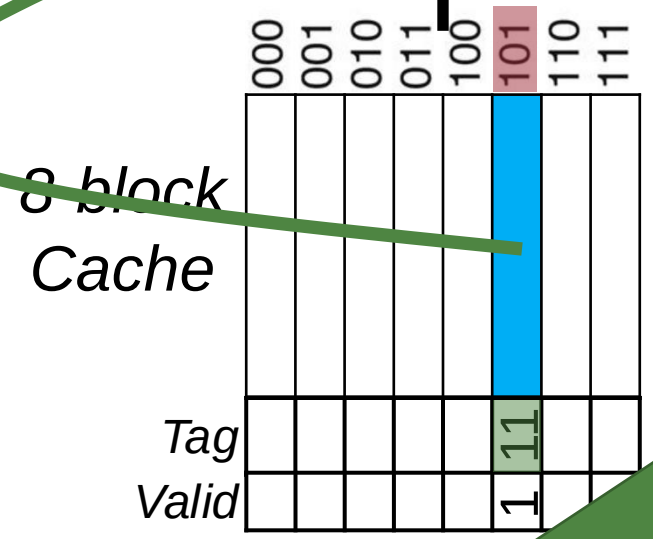


Recap: Hit

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

Processor

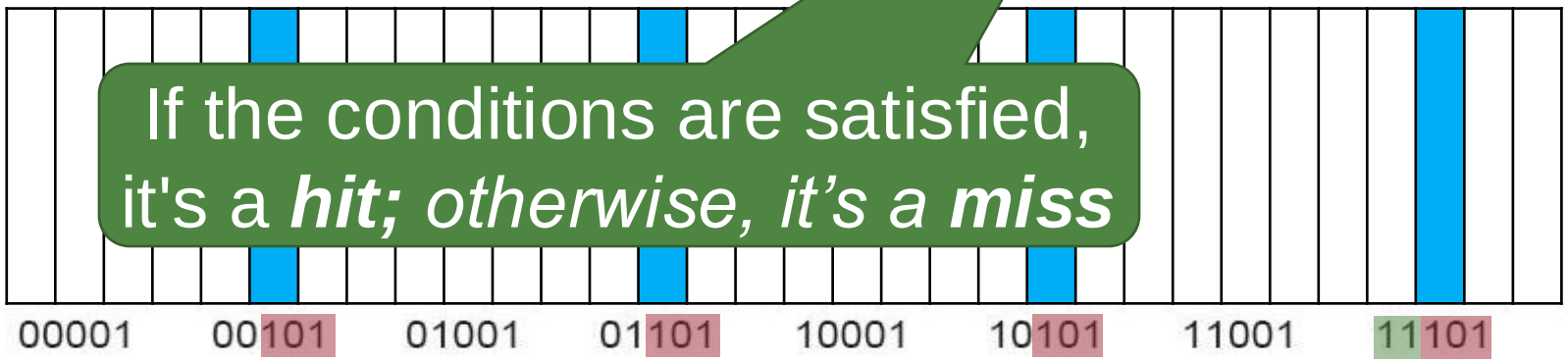
(4) If it's a hit: access the cache data *(no need to access the memory)*



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

(2) Check if tag matches

(3) Check if valid bit sets



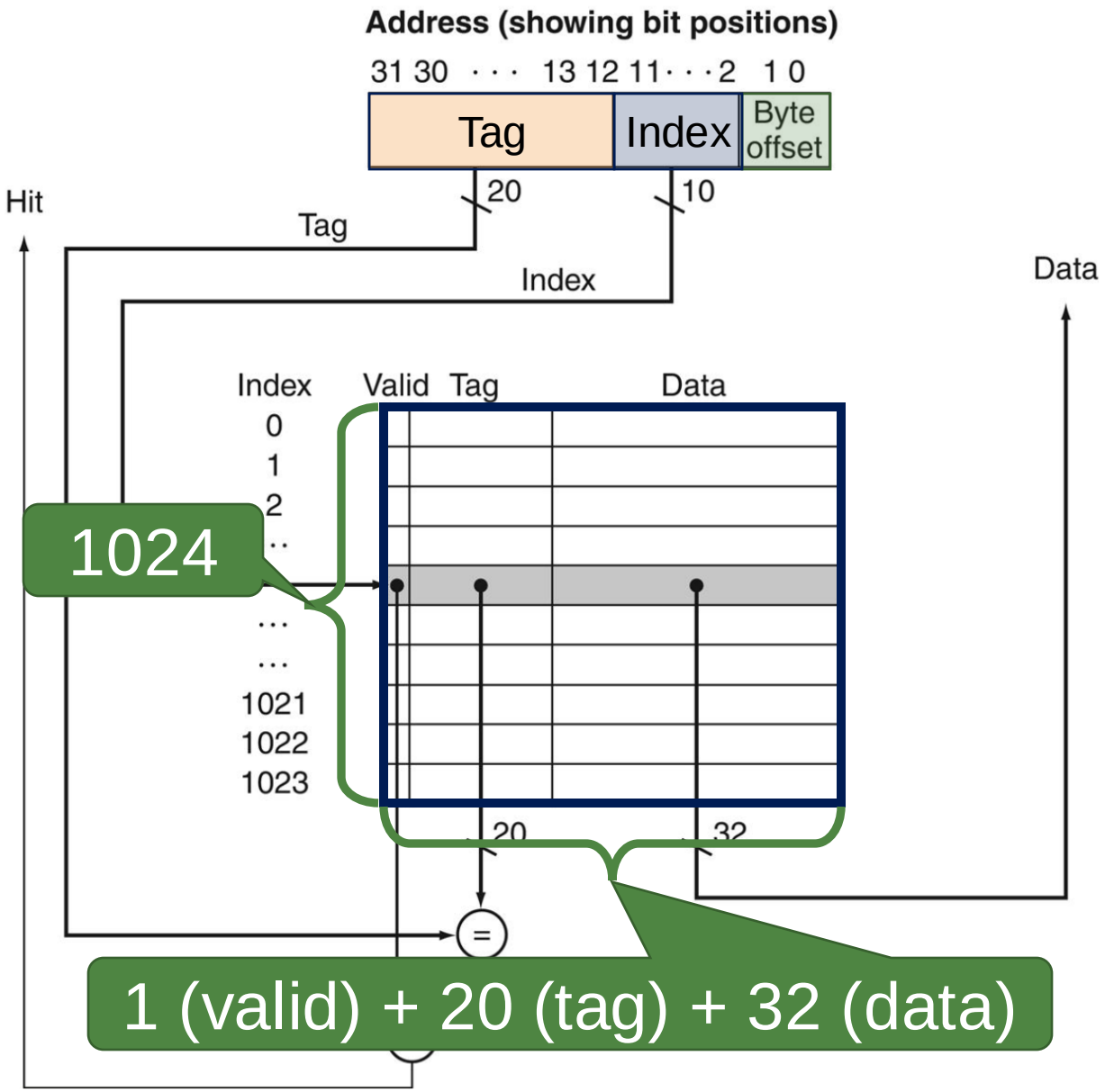
If the conditions are satisfied, it's a ***hit;*** *otherwise, it's a* ***miss***

Direct Mapped Cache Structure

- **Assumption**:
  - 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

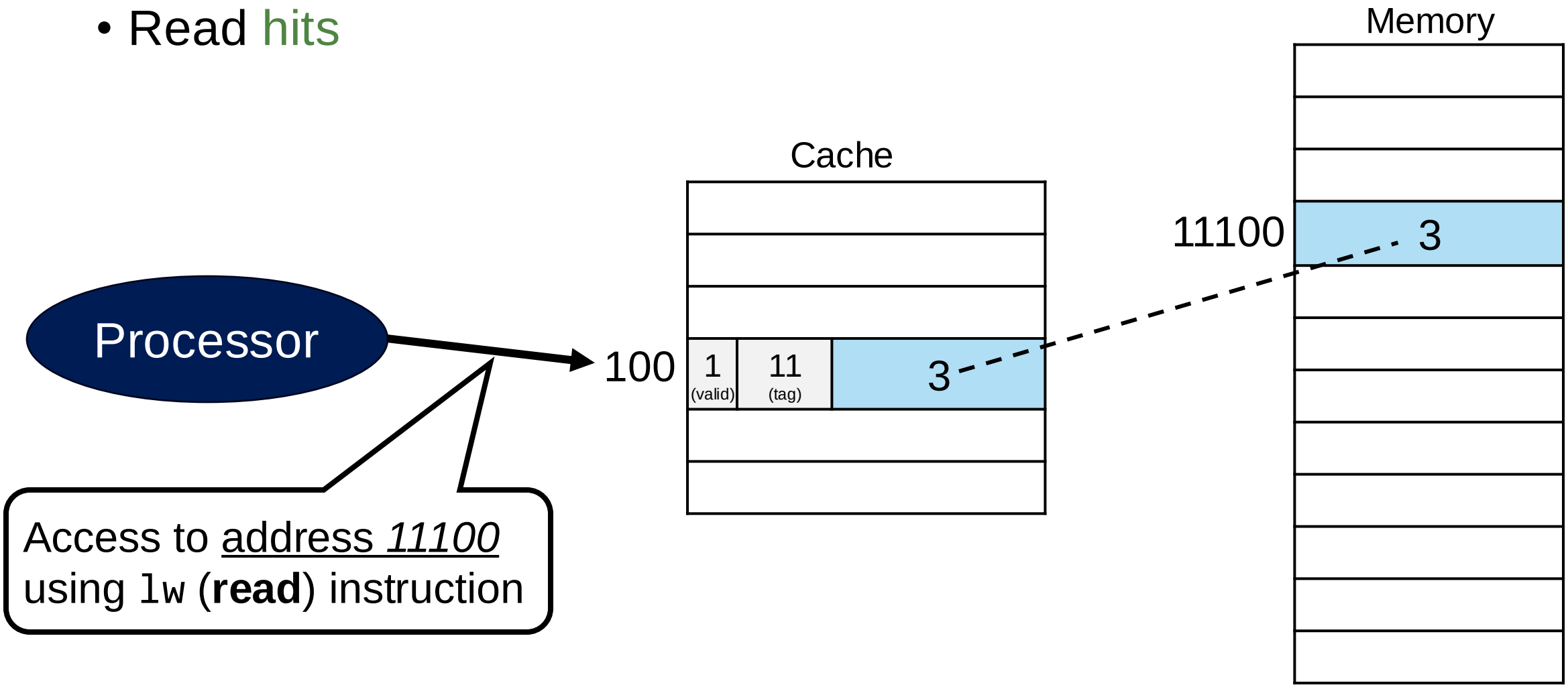
- Read hits
- Read misses
- Write hits
- Write misses

Hits vs. Misses

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

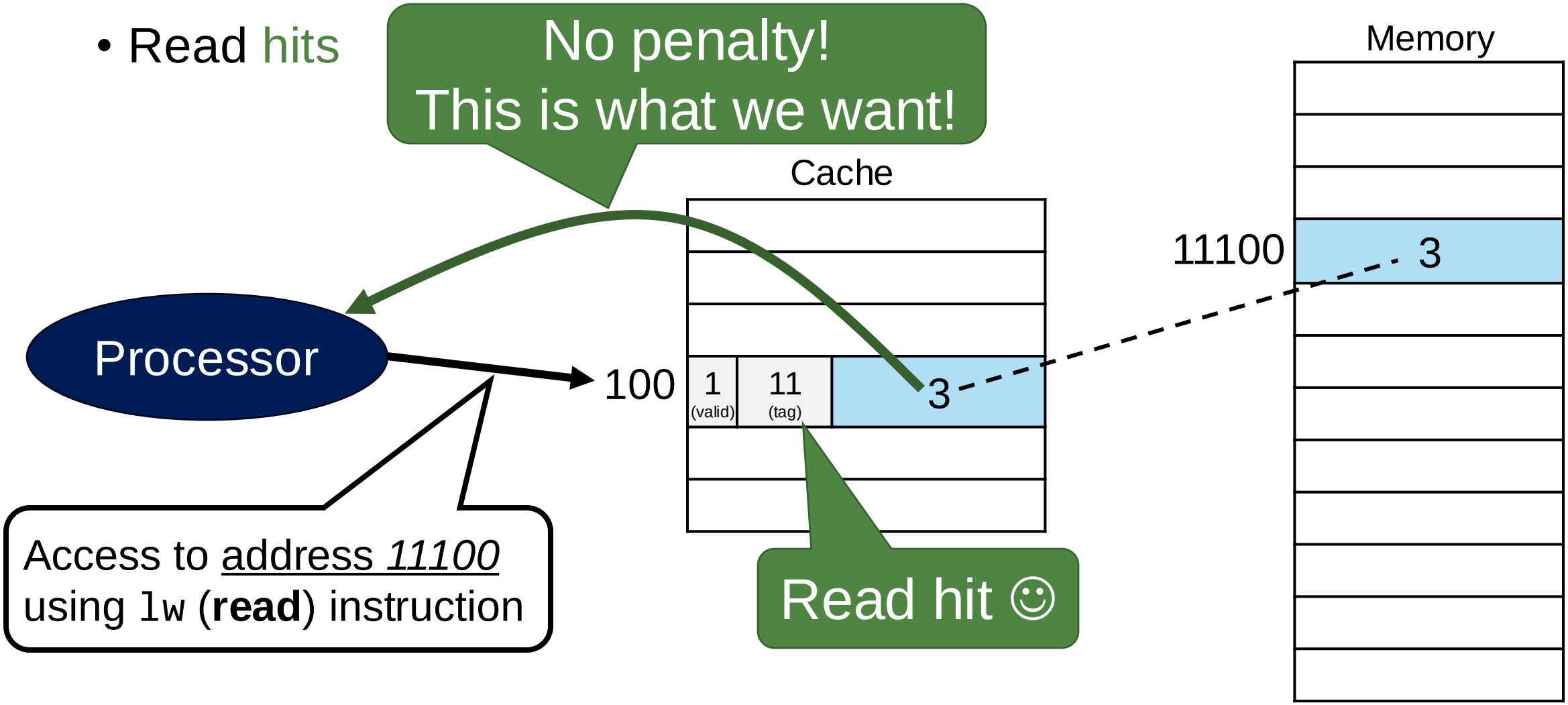
Hits vs. Misses: Read Hits

- Read hits



Hits vs. Misses: Read Hits

- Read hits



Summary: Hits vs. Misses

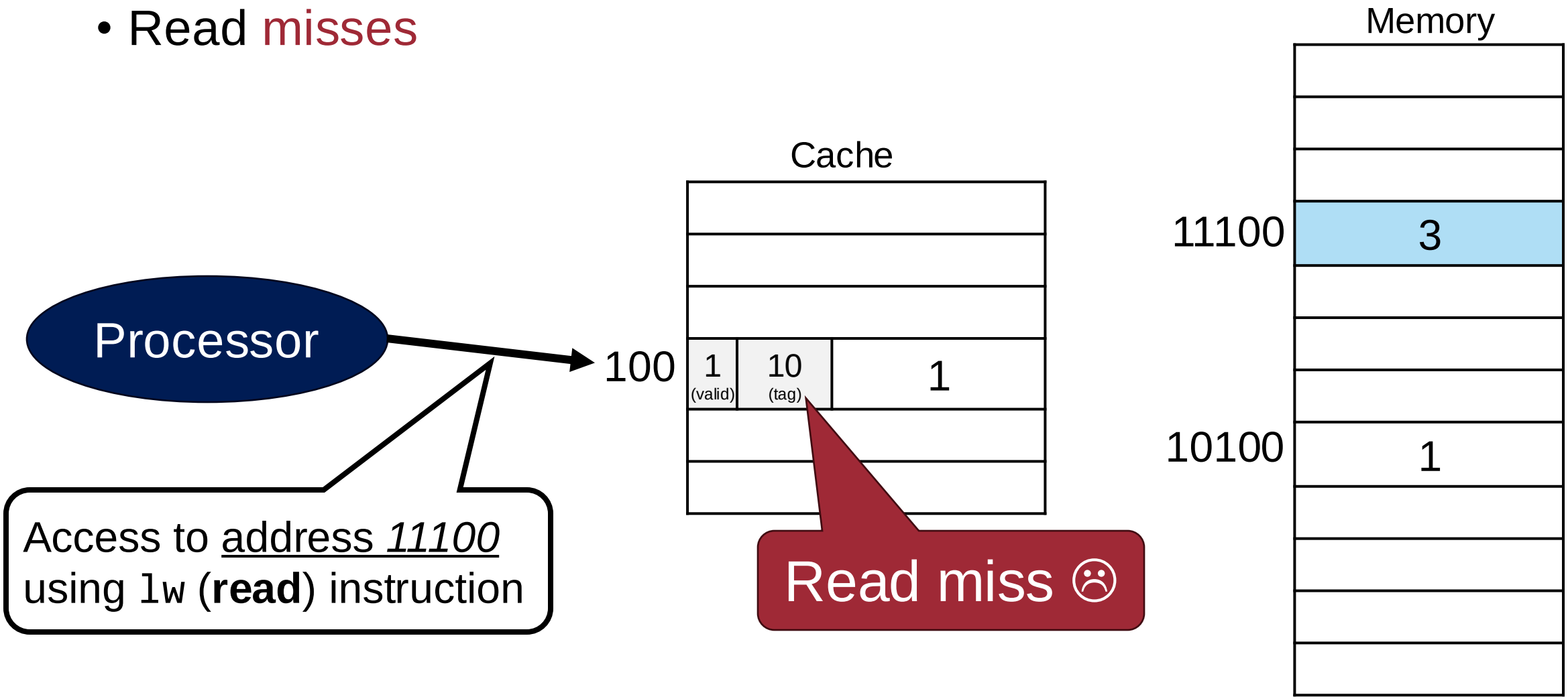
- Read hits
  - This is what we want!
- Read misses
- Write hits
- Write misses

Summary: Hits vs. Misses

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

Hits vs. Misses: Read Misses

- Read misses



ead Misses

(1) Stall the CPU

Stall

Access to <u>address *11100*</u> using `lw` (**read**) instruction

Cache

100	1 (valid)	10 (tag)	1

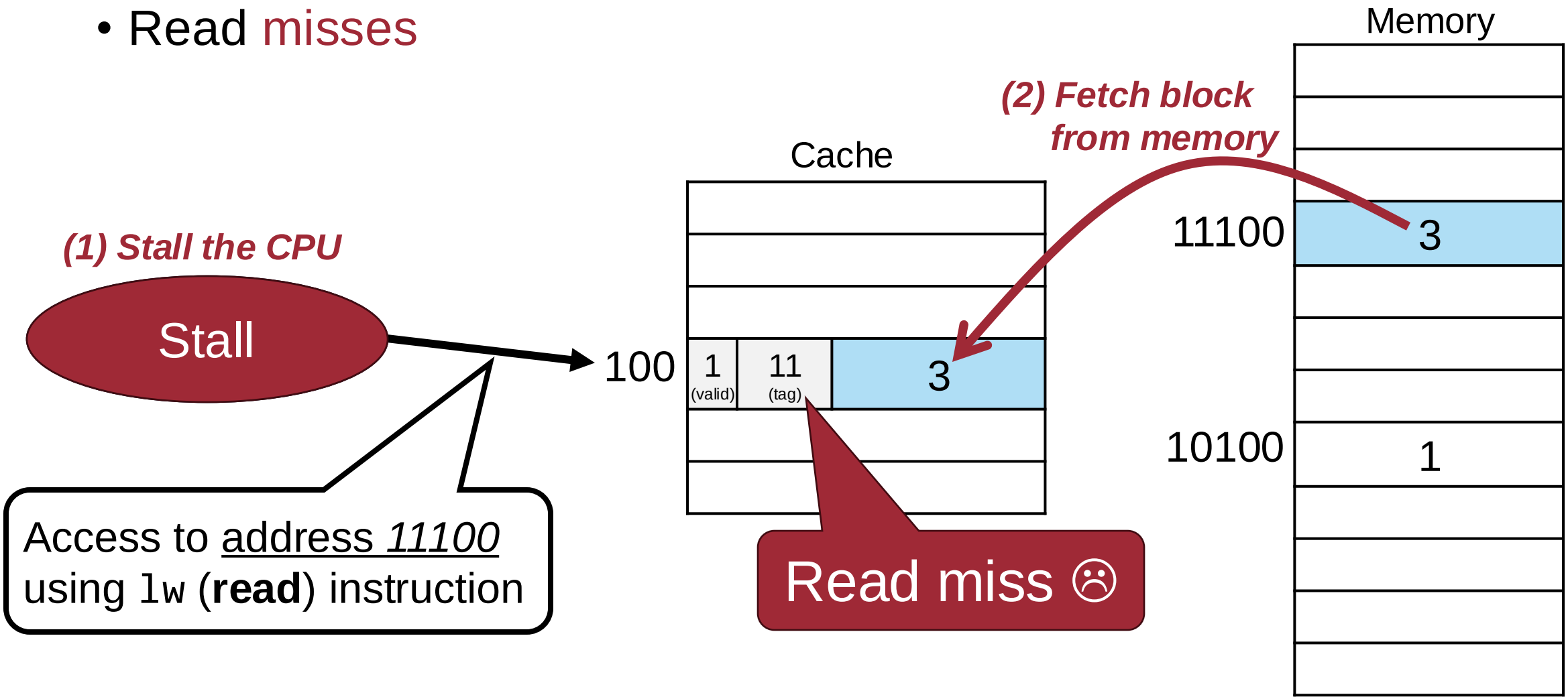
Read miss ☹

Memory

11100	3
10100	1

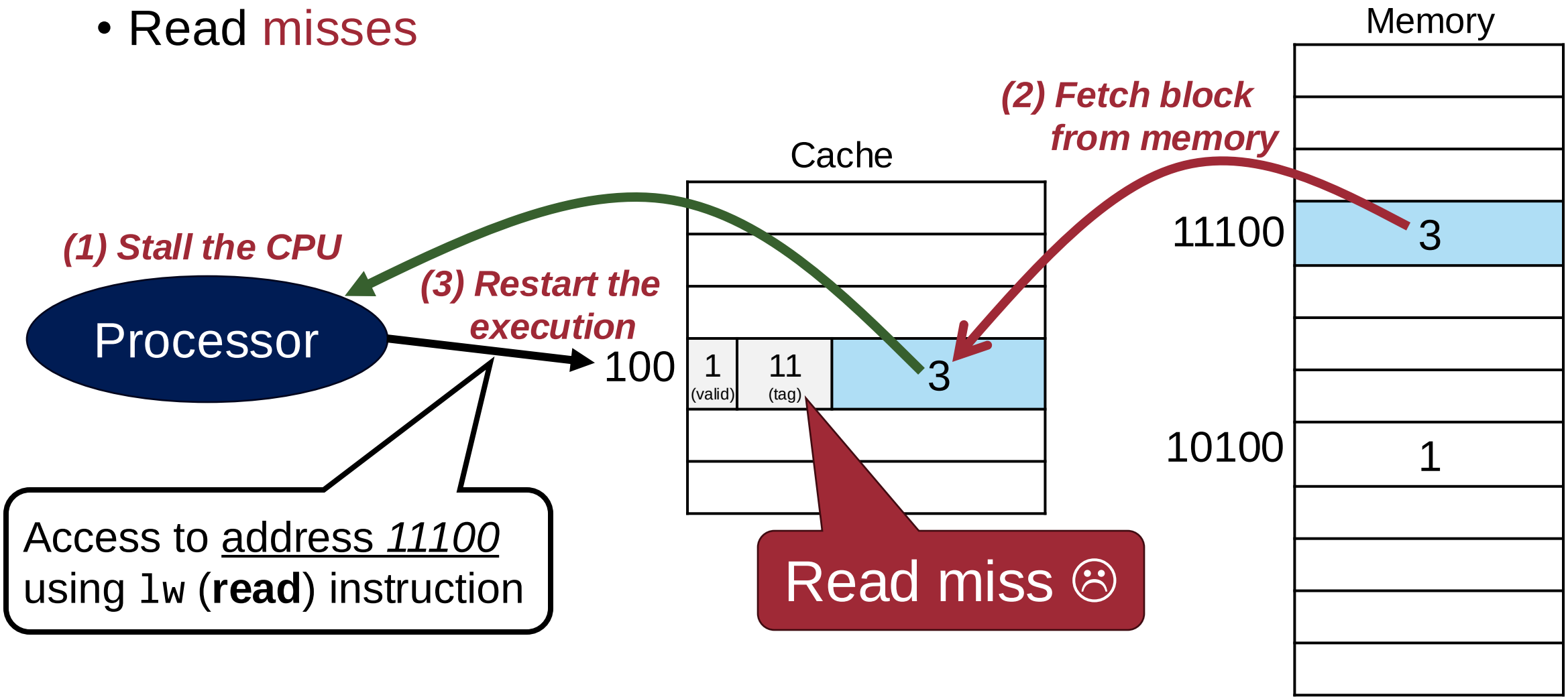
Hits vs. Misses: Read Misses

- Read misses



Hits vs. Misses: Read Misses

- Read misses



Hits vs. Misses: Read Misses

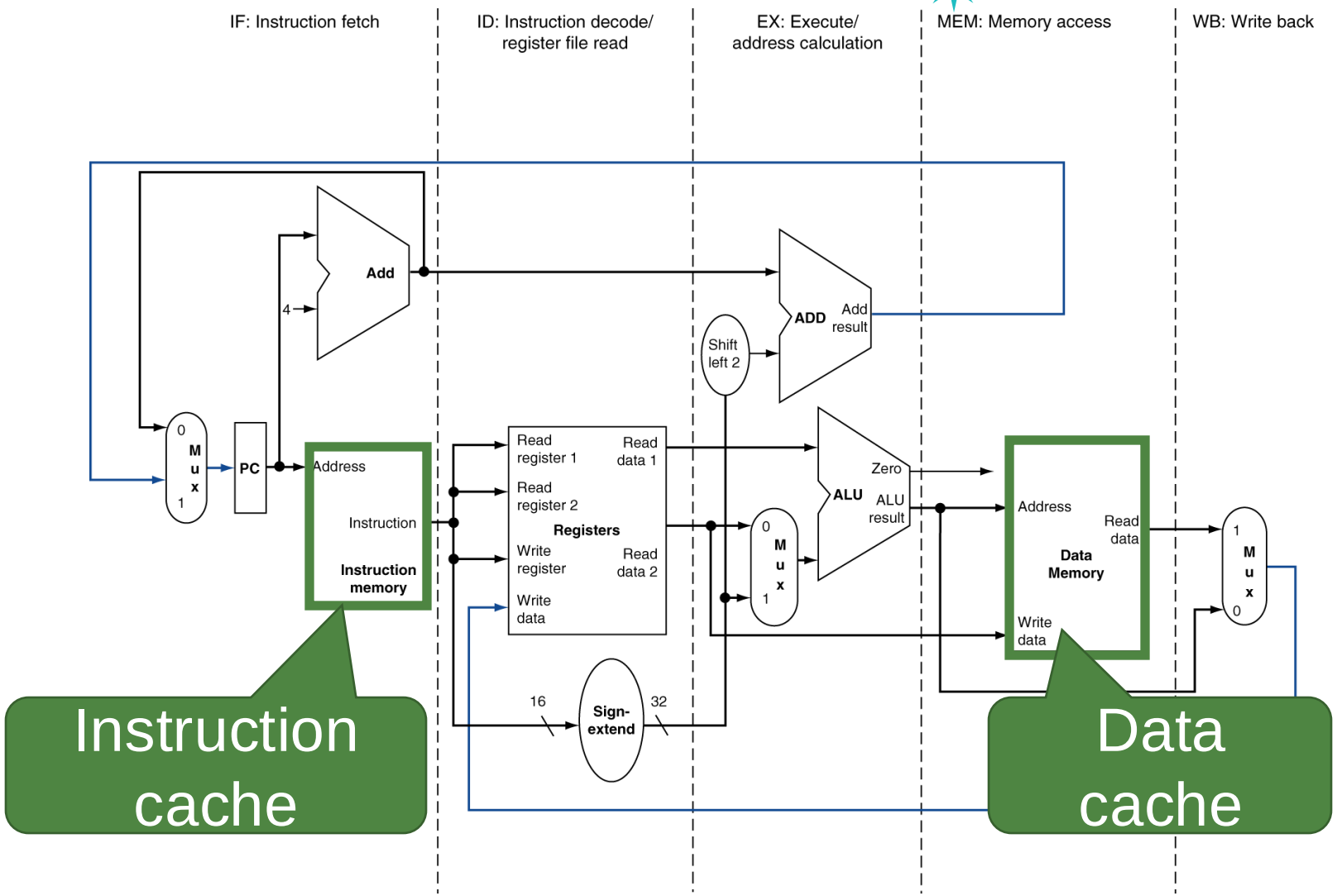
- Read misses

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

(2) Fetch block from memory

(3) Restart the execution

Two Different Caches



Hits vs. Misses: Read Misses

- Read misses

  (0) (For the instruction cache) Set PC-4

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

  (2) Fetch block from memory

  (3) Restart the execution (= Re-fetch instruction)

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

Cases to consider for the instruction cache

Cases to consider for the data cache

Summary: Hits vs. Misses

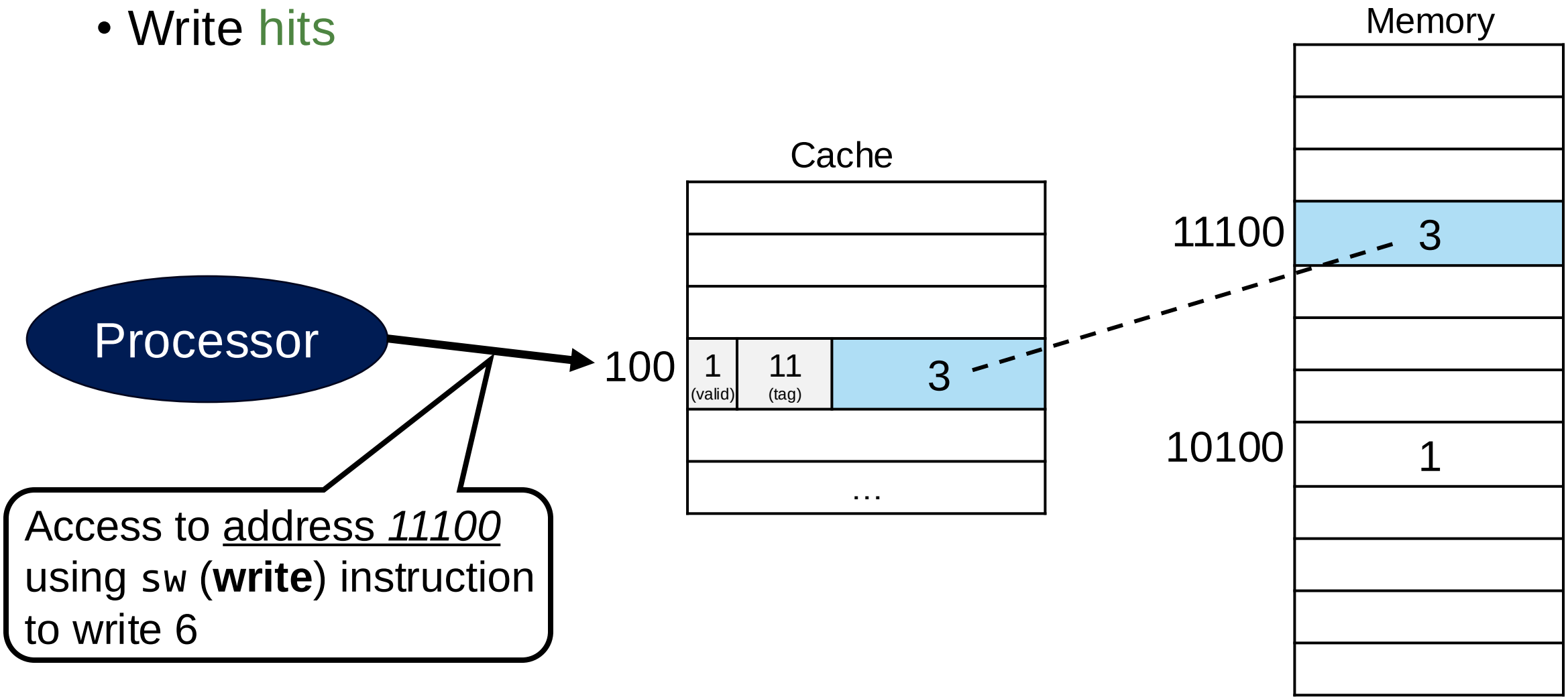
- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch block from memory, restart
- Write hits
- Write misses

Cases to consider for the instruction cache

Cases to consider for the data cache

Hits vs. Misses: Write Hits

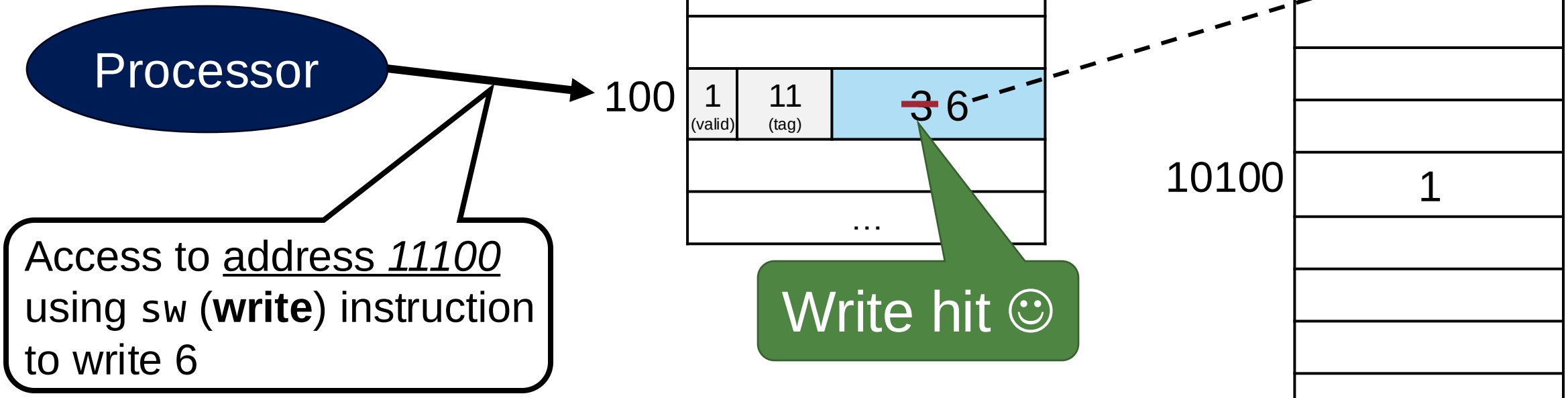
- Write hits



Hits vs. Misses: Write Hits

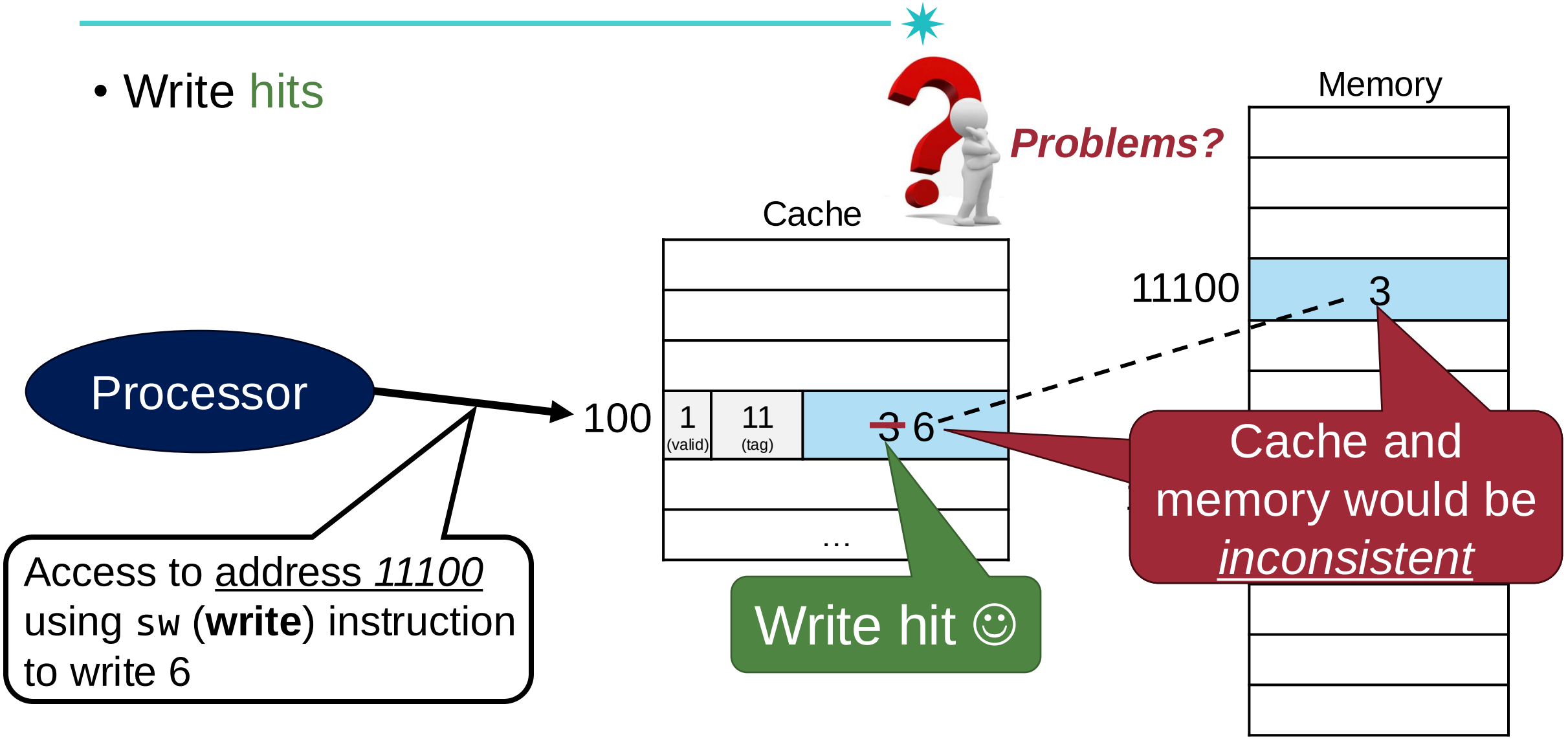
- Write hits

Problems?



Hits vs. Misses: Write Hits

- Write hits



Write Policies

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

(1) Write through

(2) Write back

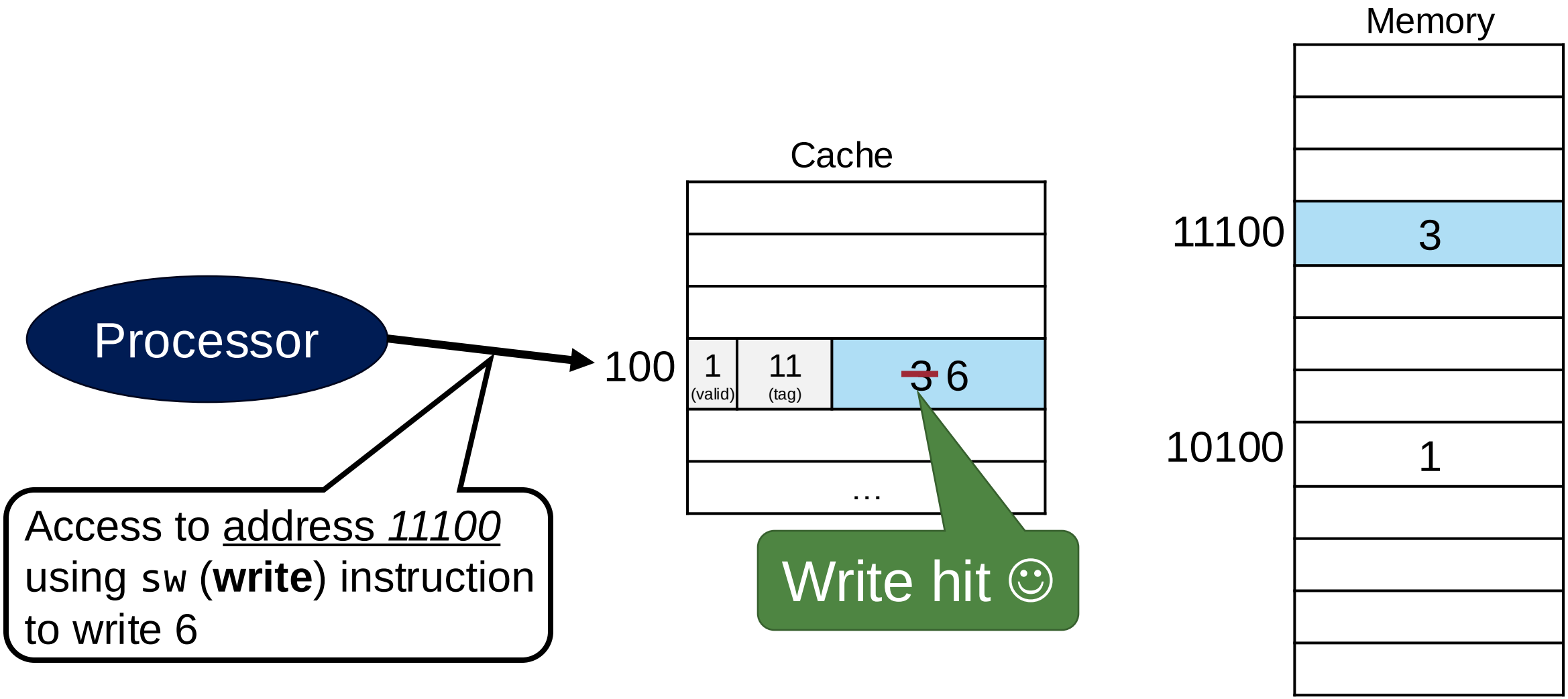
Write Policies

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

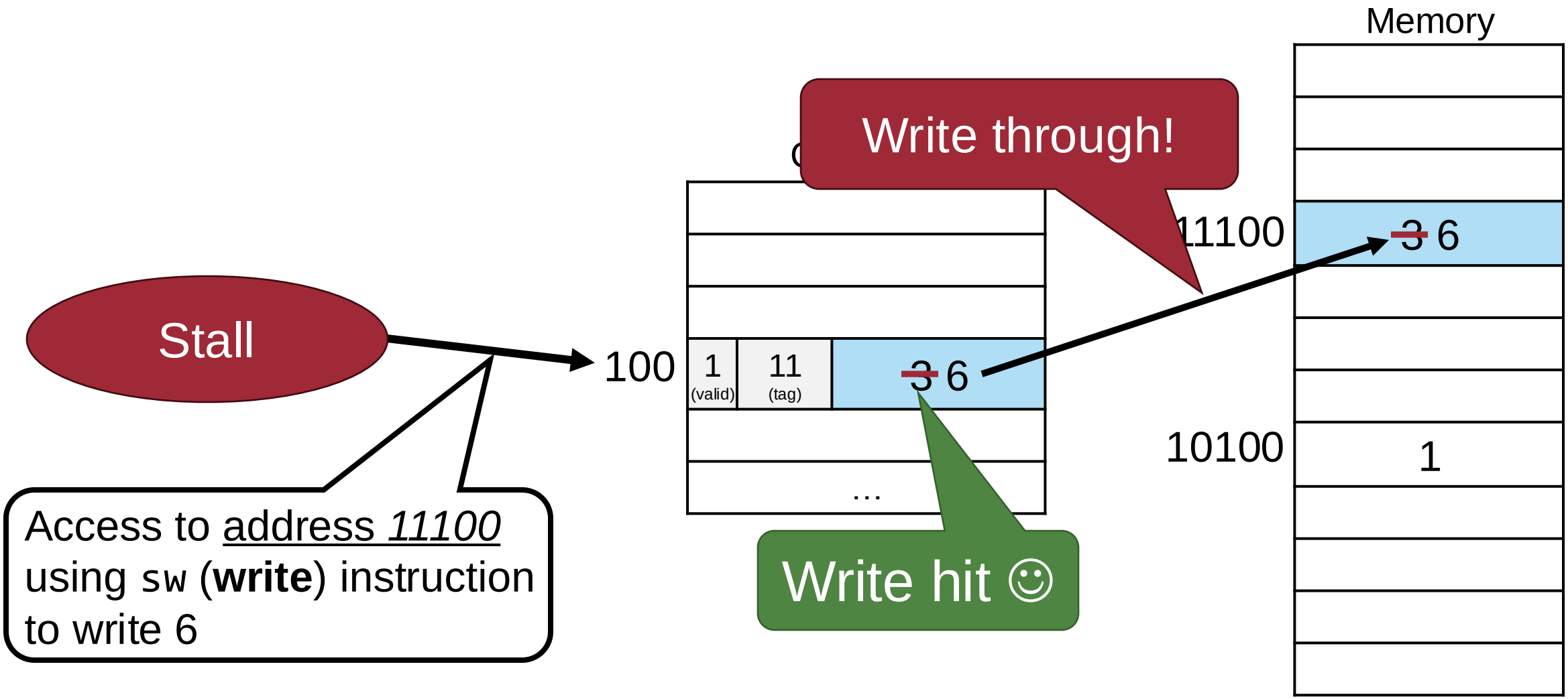
(1) Write through: On each write hit, the information is written to <u>both in the cache and in the memory</u>

(2) Write back

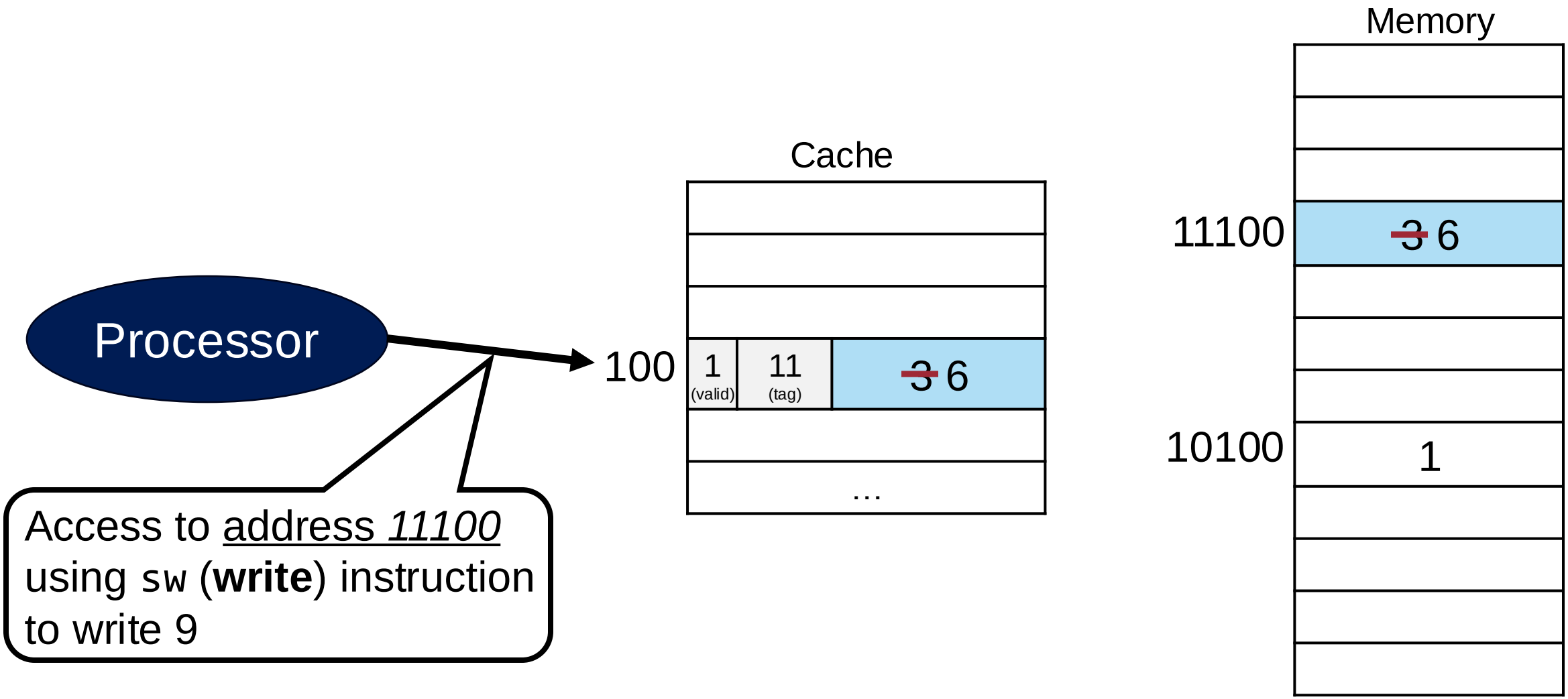
Write Through



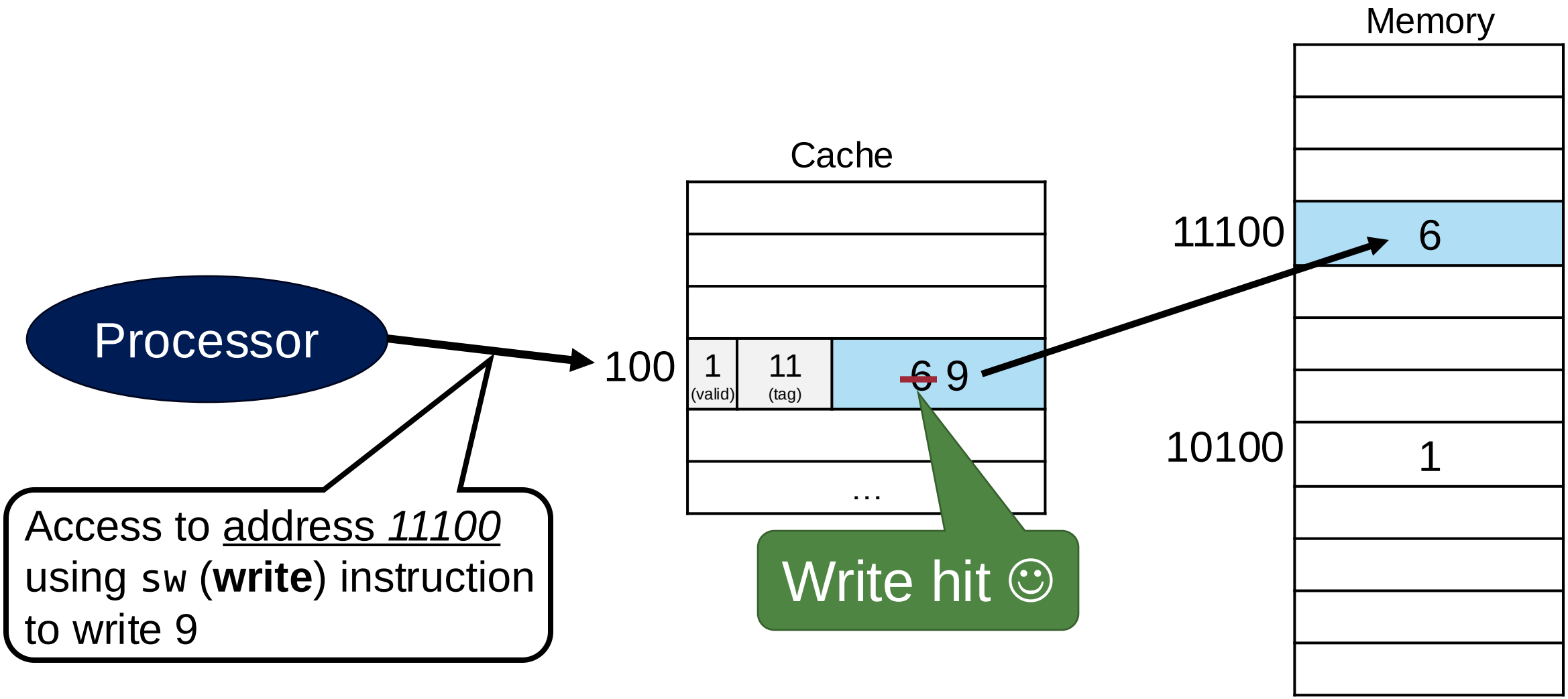
Write Through



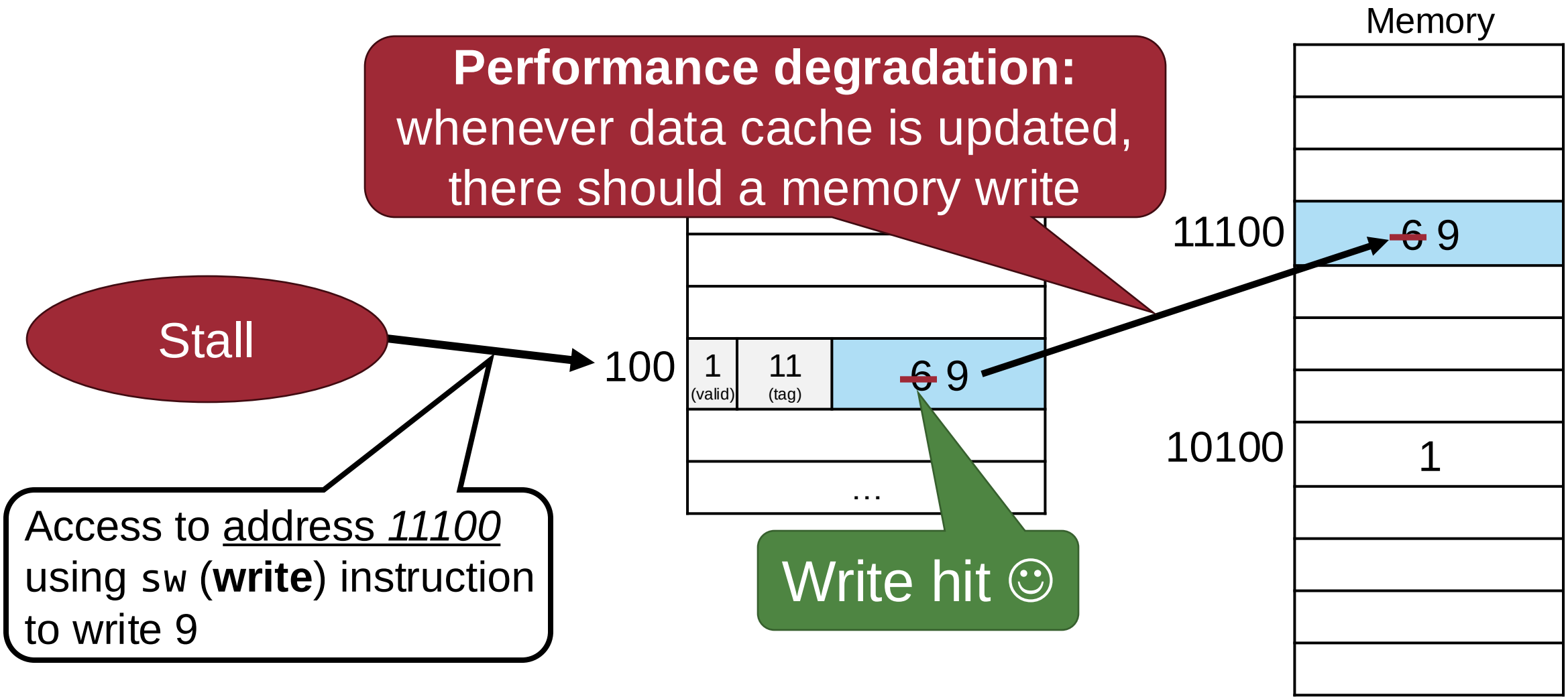
Write Through



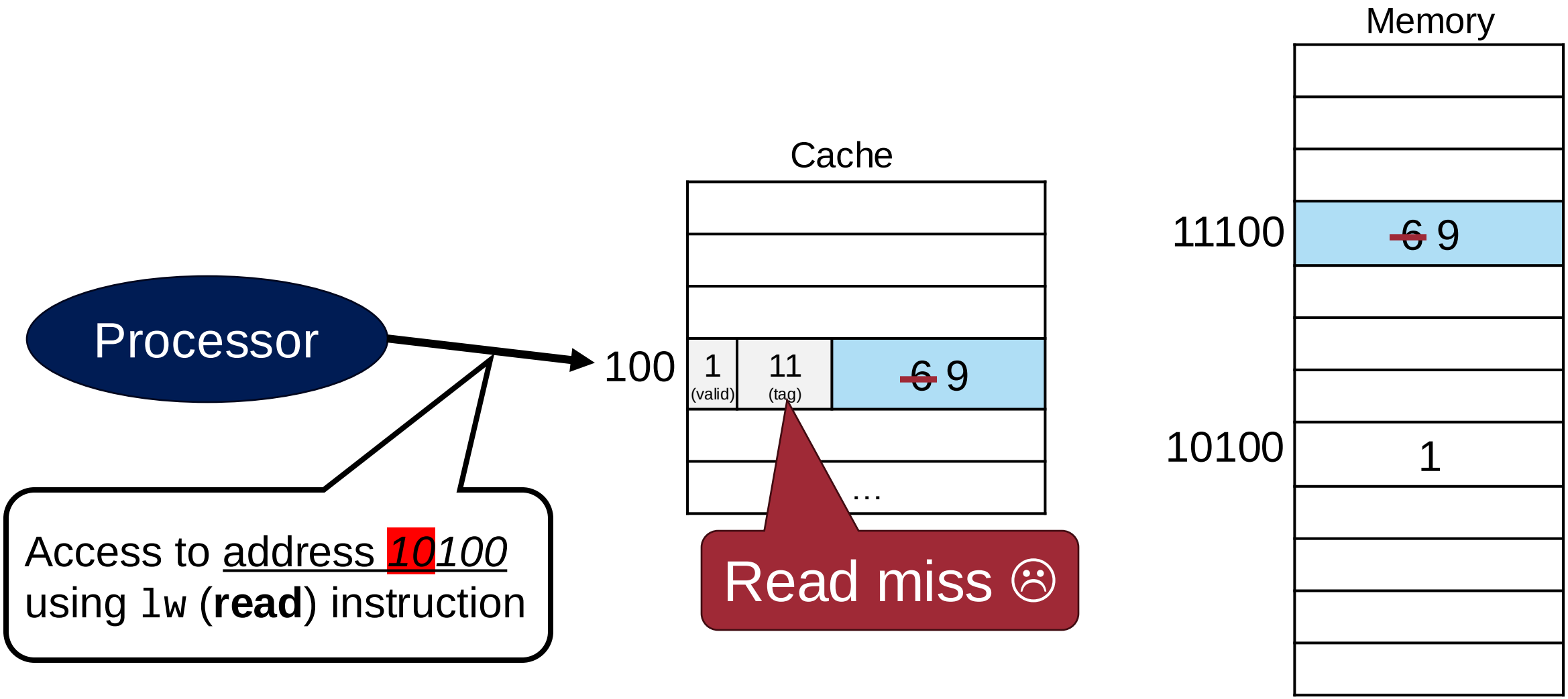
Write Through



Write Through

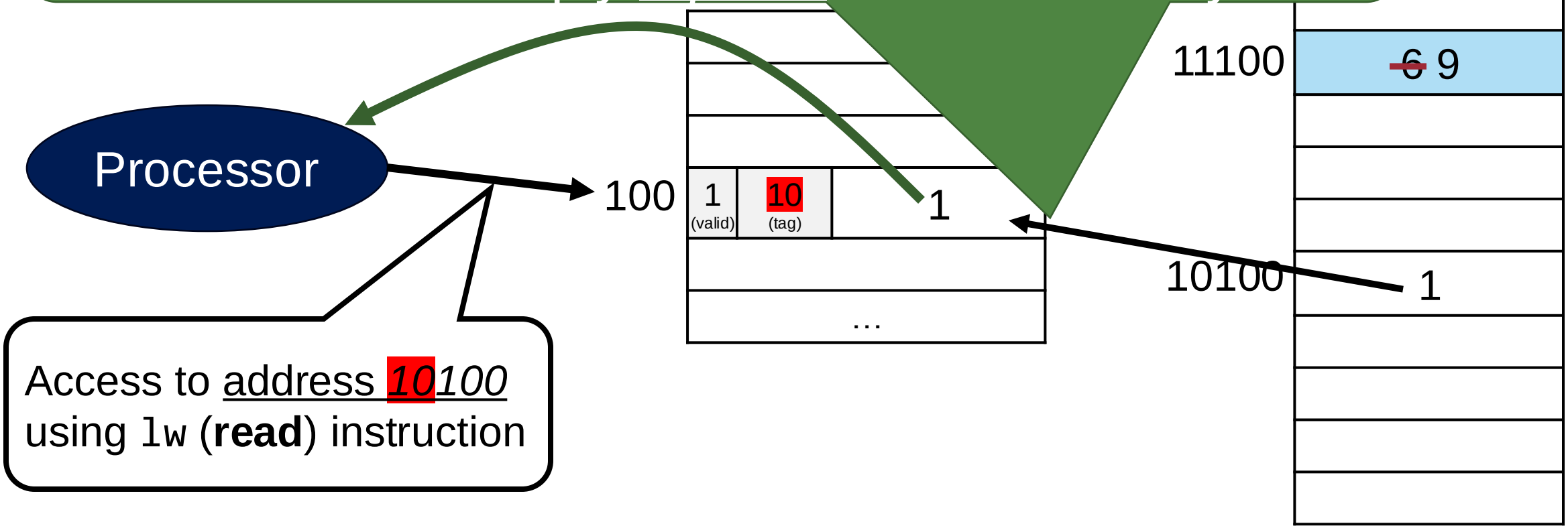


Write Through



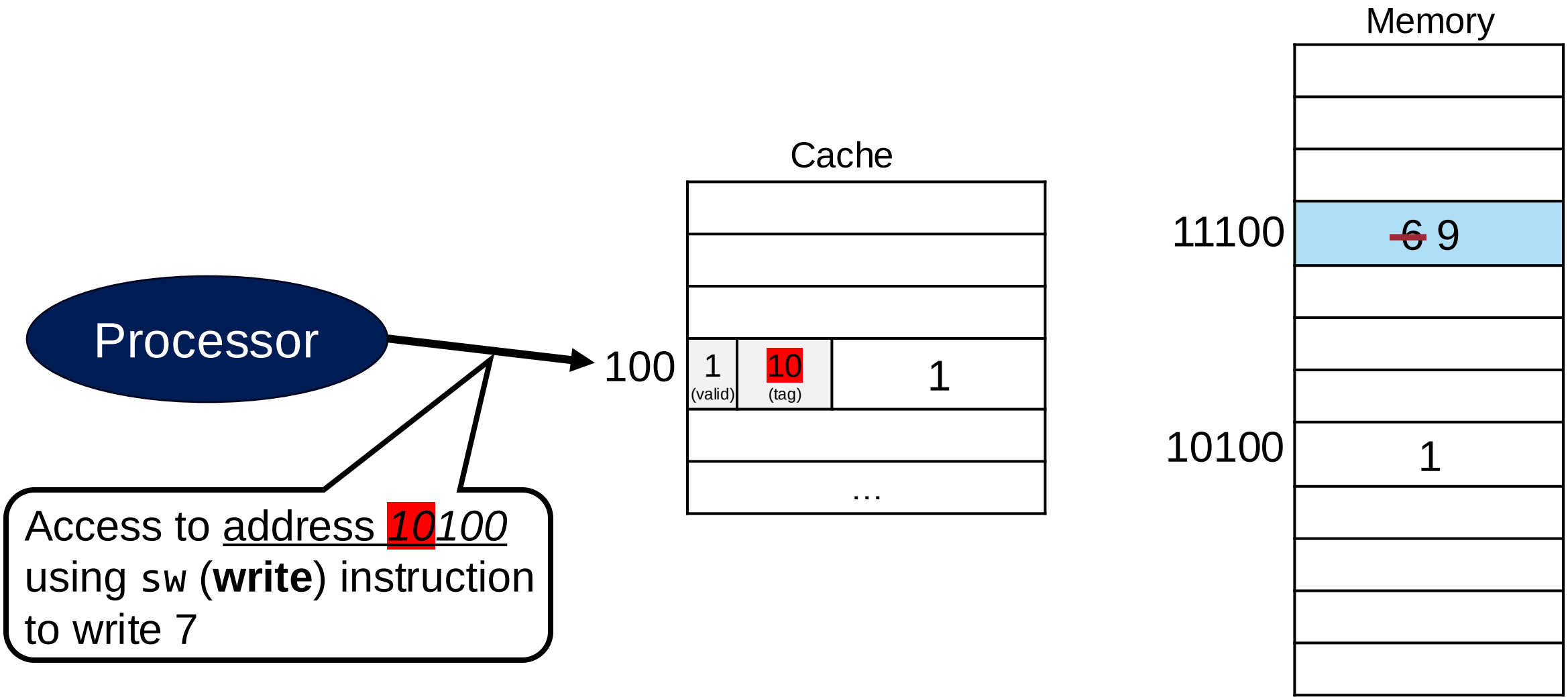
Write Through

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

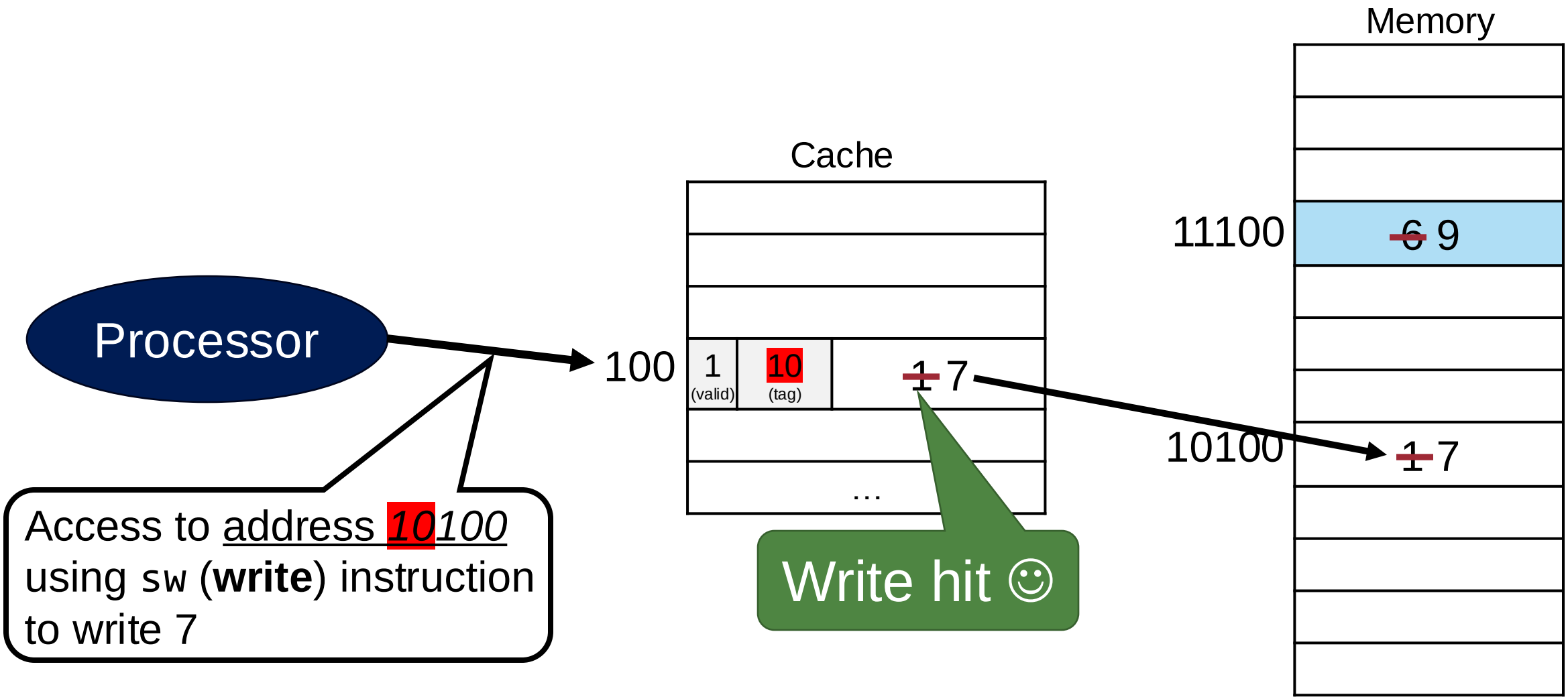


Access to address *10100* using `lw` (**read**) instruction

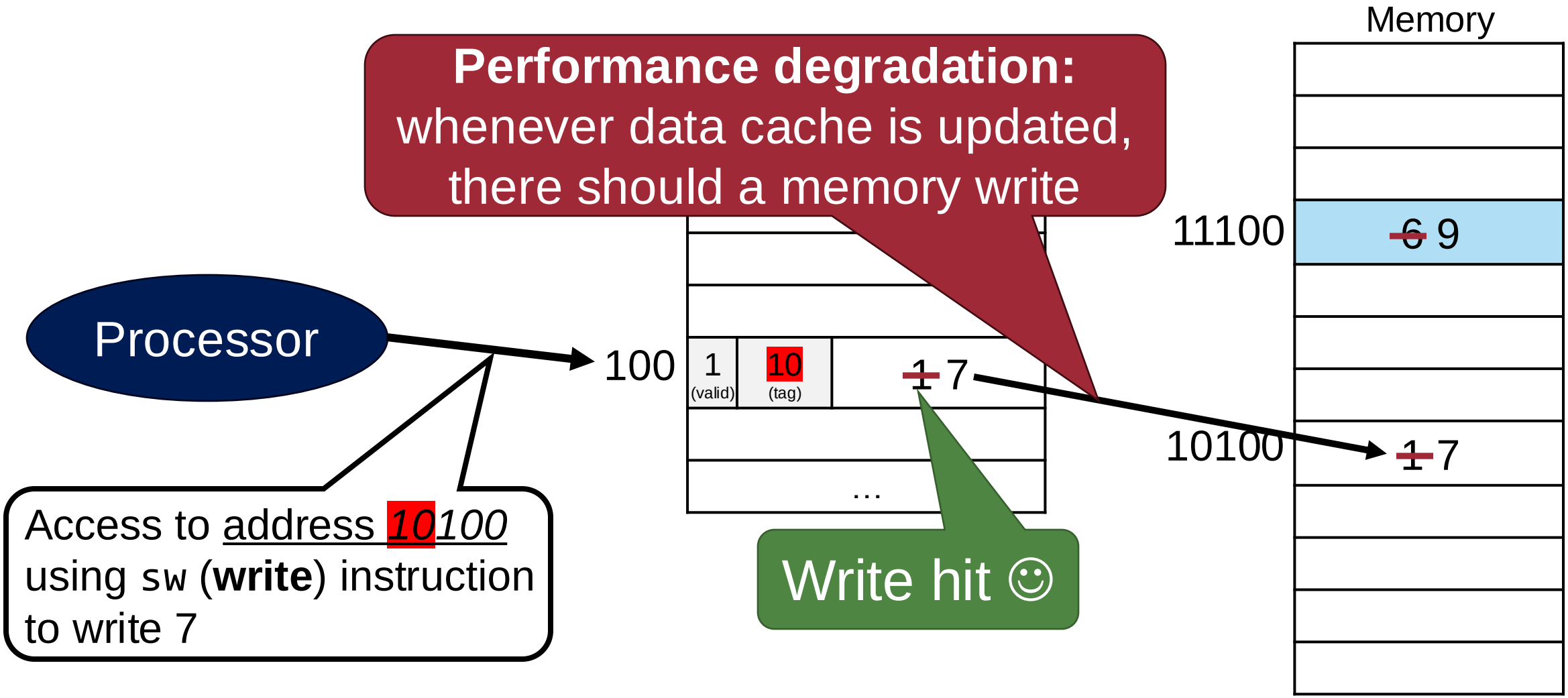
Write Through



Write Through

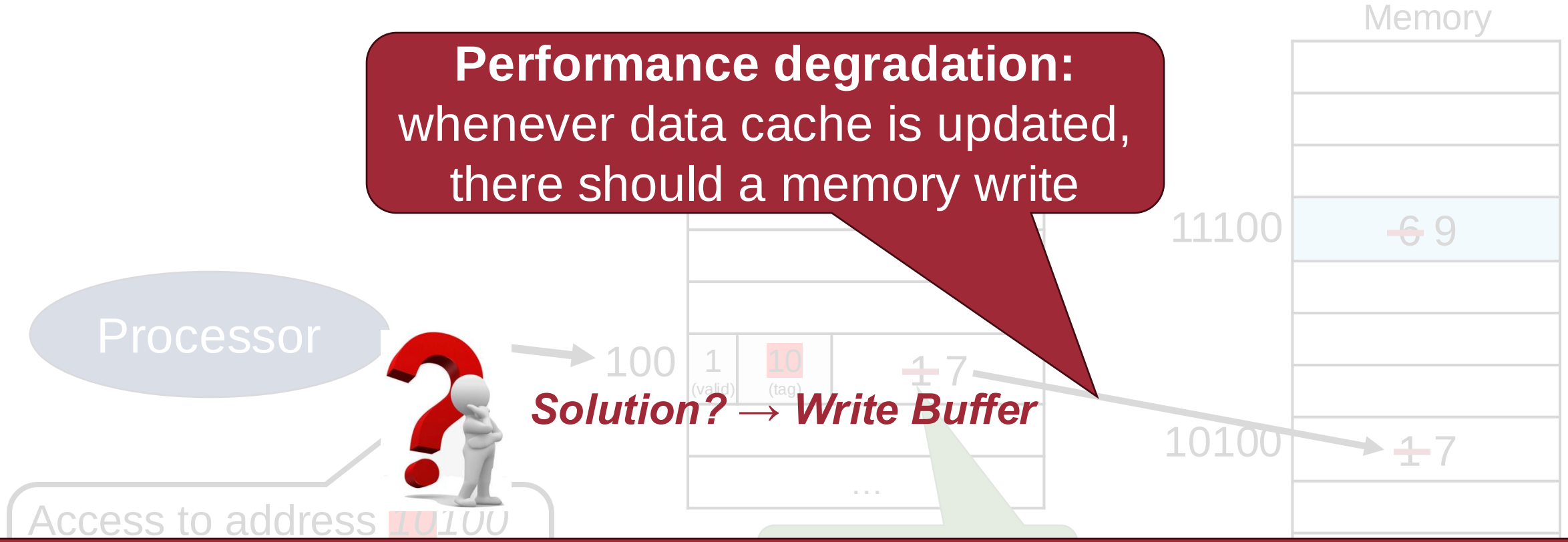


Main Disadvantage: Performance Degradation



Main Disadvantage: Performance Degradation

Performance degradation:
whenever data cache is updated,
there should a memory write

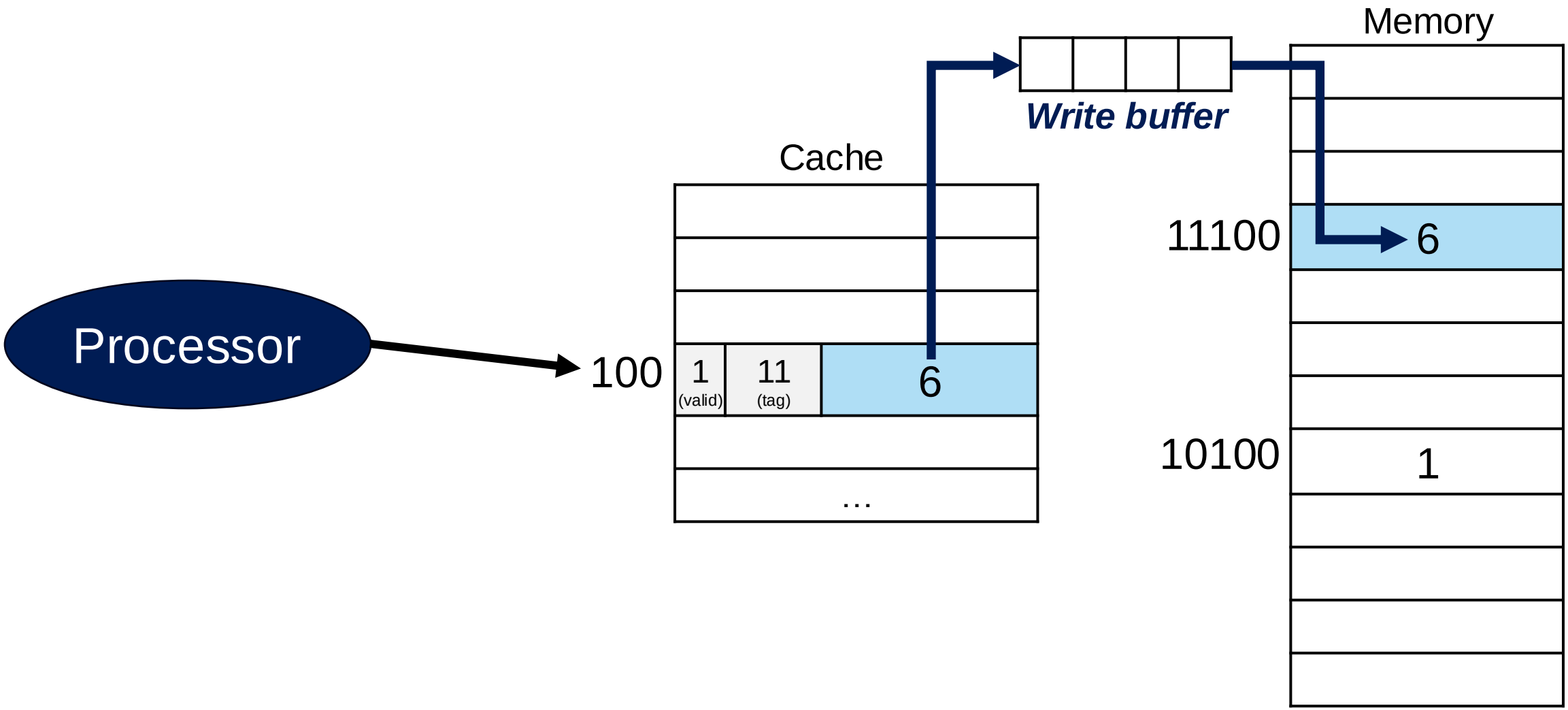


Solution? → Write Buffer

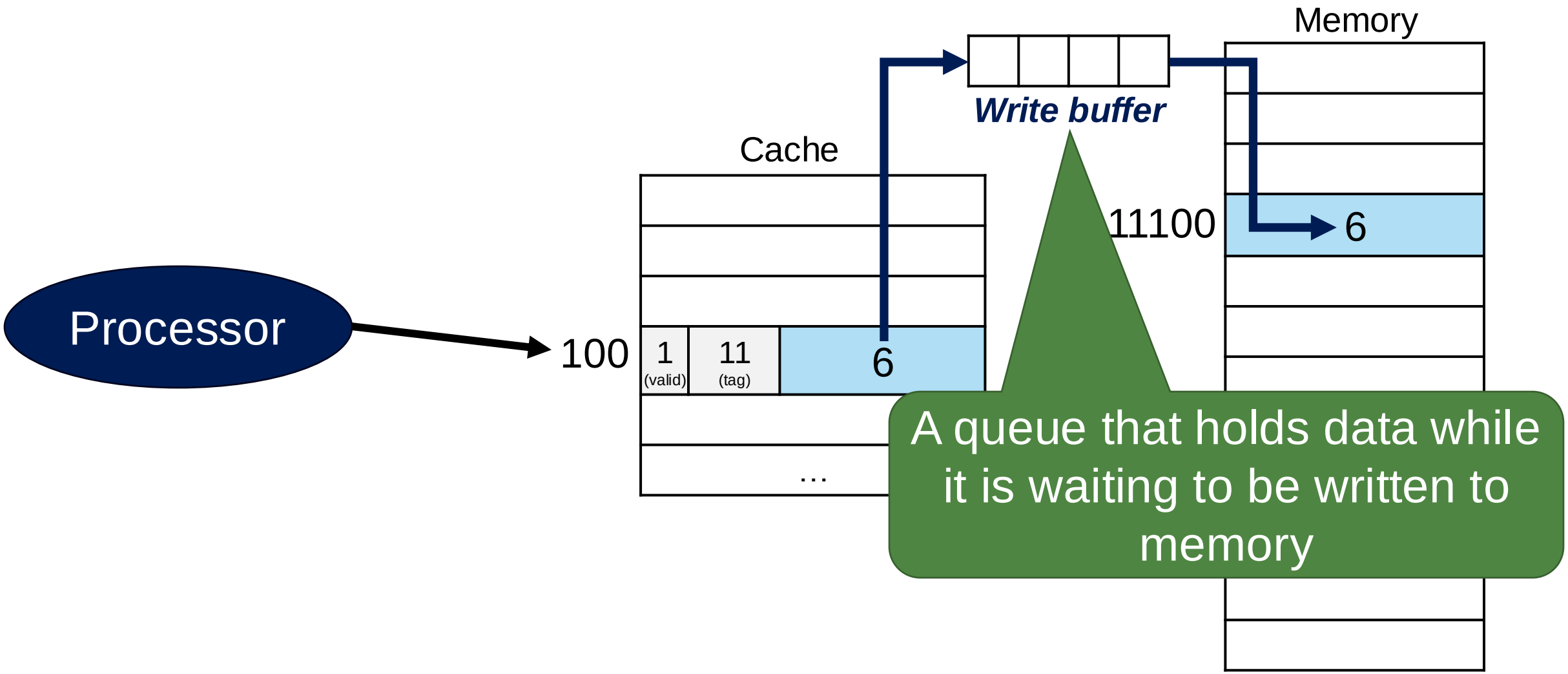
If base CPI = 1, 10% of instructions are stores,
write to memory takes 100 extra cycles

CPI = 1 + 0.1 x 100 = 11

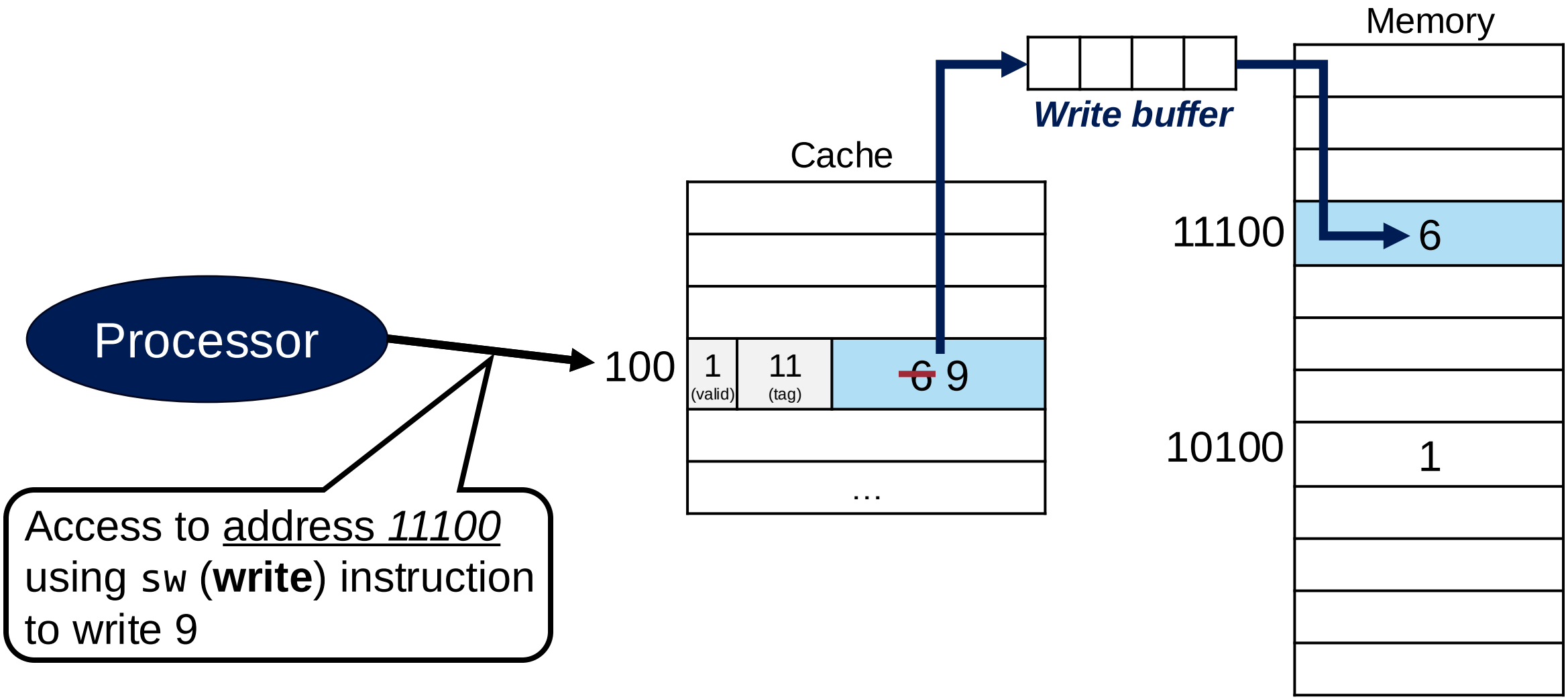
Simple Idea: Write Buffer + Write Through



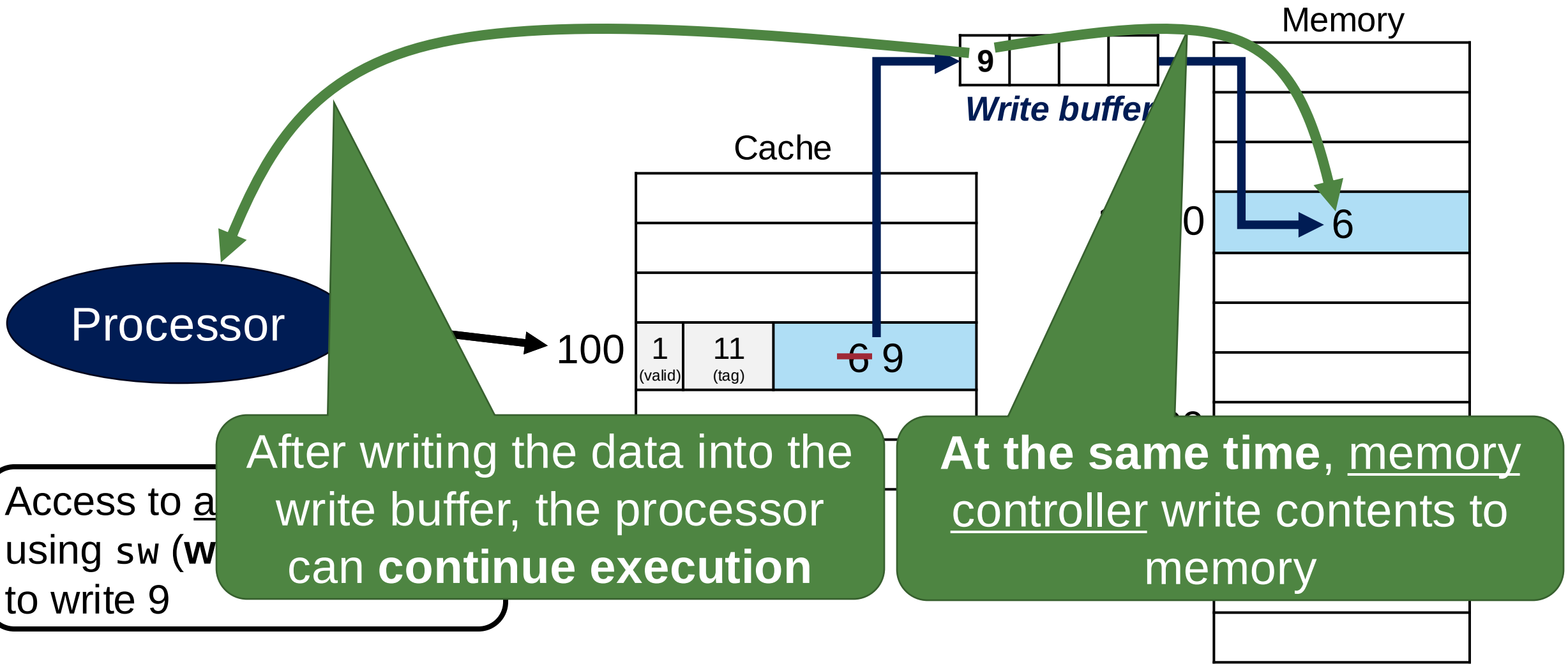
Simple Idea: Write Buffer + Write Through



Simple Idea: Write Buffer + Write Through



Simple Idea: Write Buffer + Write Through



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 <u>both in the cache and in the memory</u>
    - Pros: faster processing in case of 'miss'
    - Cons: make writes take longer (whenever data cache is updated, there should a memory write)
      - ✓Solution: write buffer

  (2) Write back: Next lecture!

Question?