CSE261: Computer Architecture

17. Memory Hierarchy (3): Caches #2

Seongil Wi

Department of Computer Science and Engineering

Notification: Final Exam

- Date: Dec. 19 (Thursday)
- Class Time (1h 15m), Closed book

- T/F problems + Computation problems + Descriptive problems

- Scope: All the material learned *after the midterm*

Notification: HW3

- Implementing a cache simulator
- Due: Dec 15, 11:59 PM

- Since the final exam period is approaching, it is recommended to complete this assignment *as soon as possible*

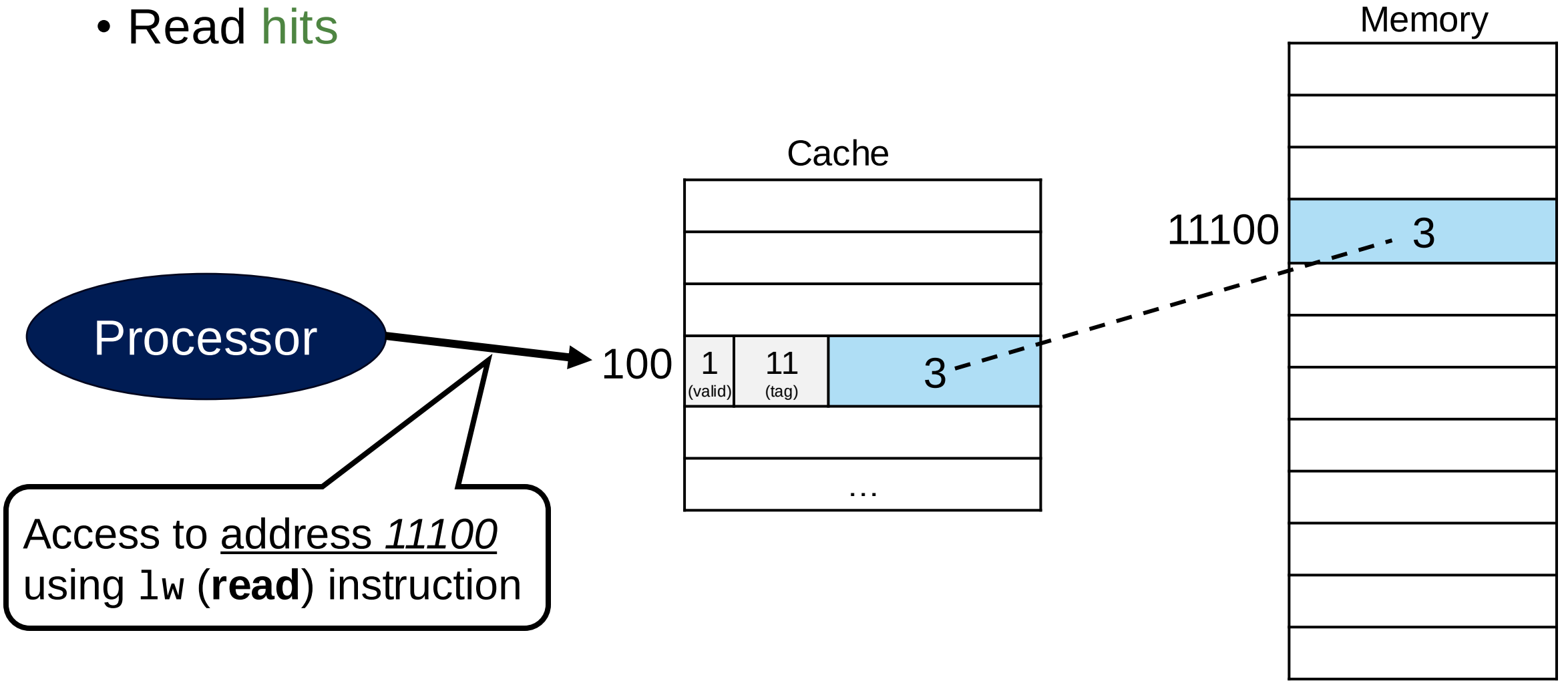
Handling Cache Hits & Misses

Hits vs. Misses

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

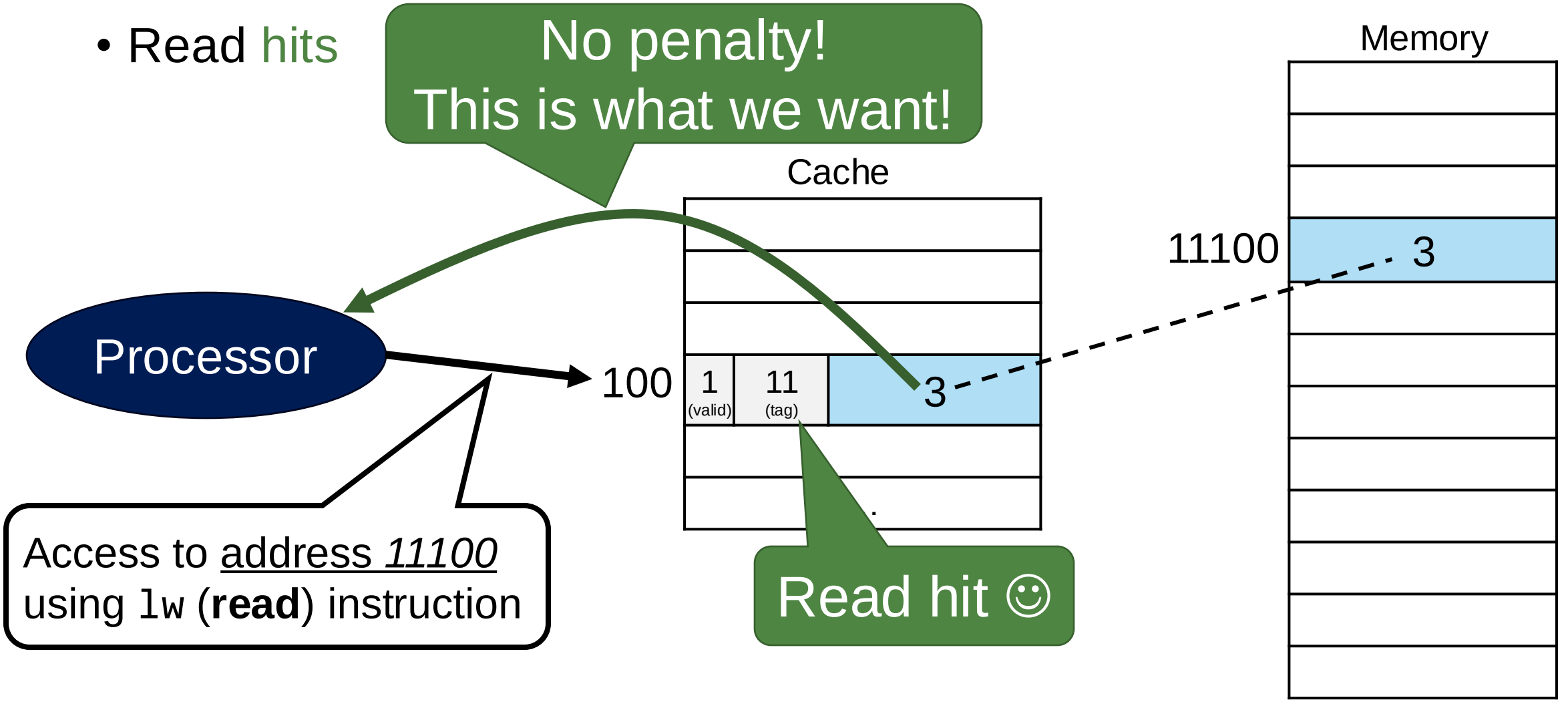
Hits vs. Misses: Read Hits

- Read hits



Hits vs. Misses: Read Hits

- Read hits

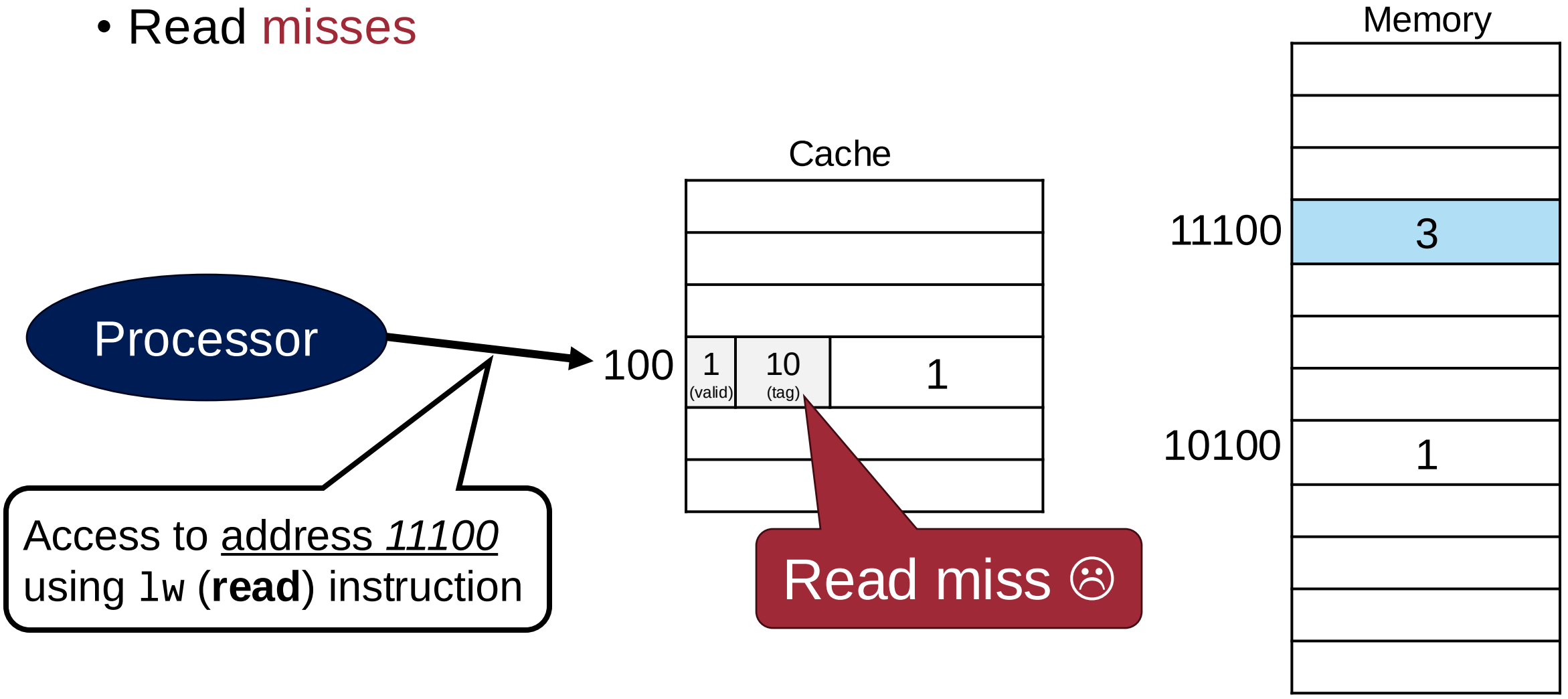


Summary: Hits vs. Misses

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

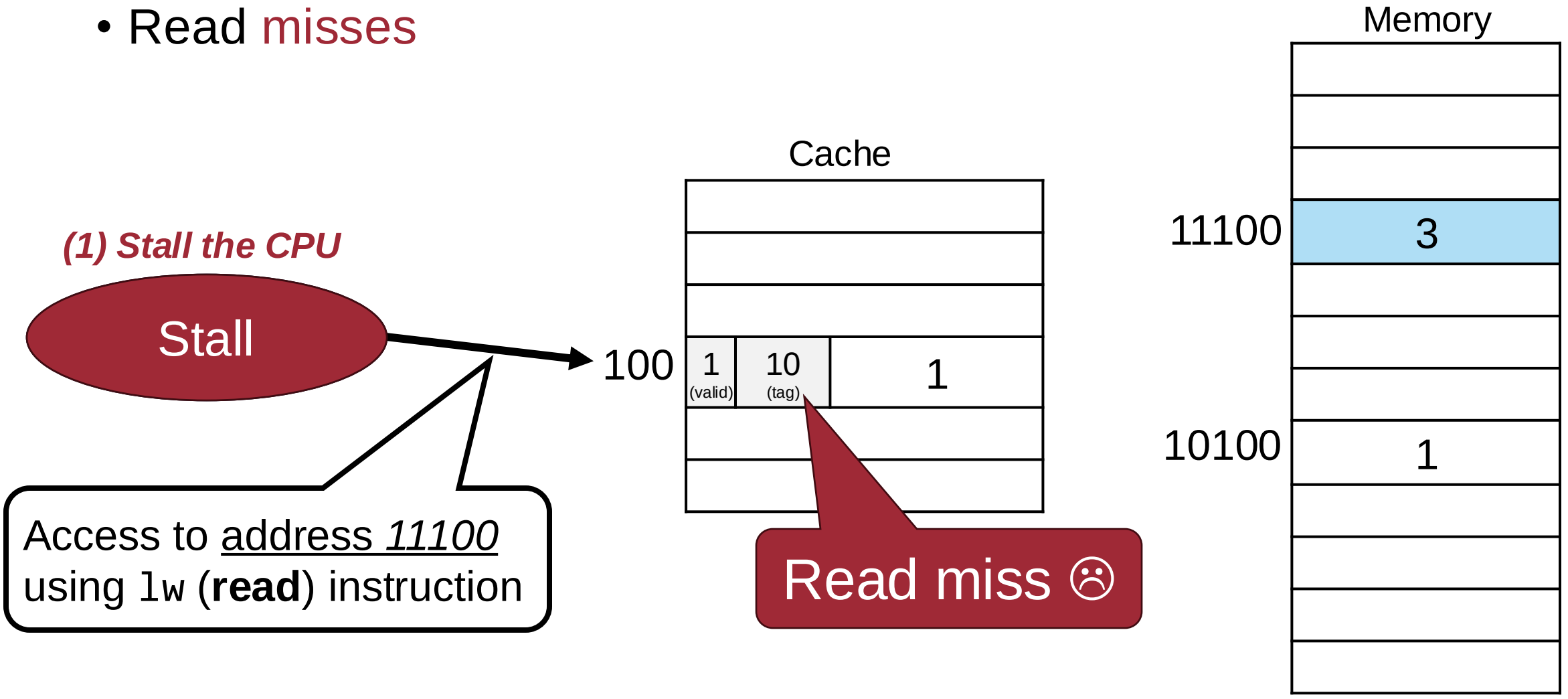
Hits vs. Misses: Read Misses

- Read misses



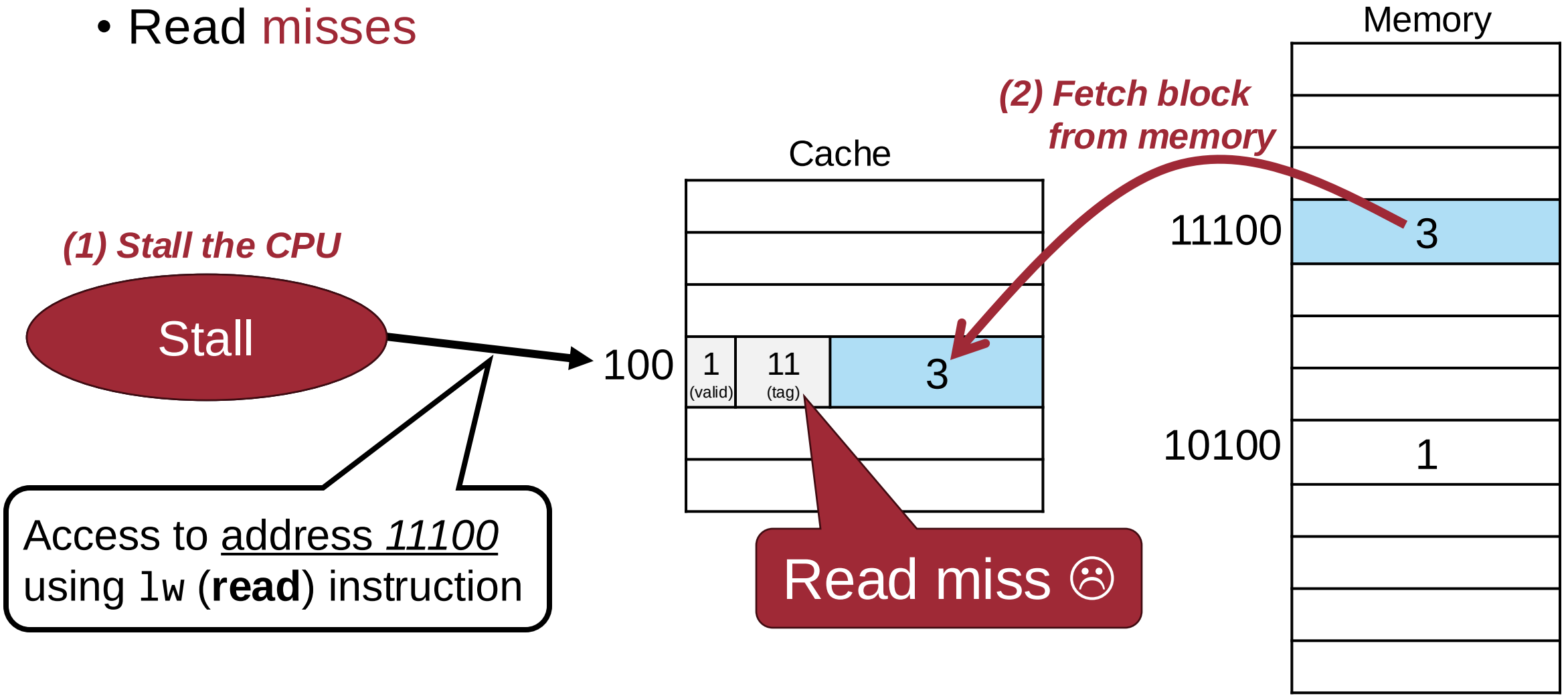
Hits vs. Misses: Read Misses

- Read misses



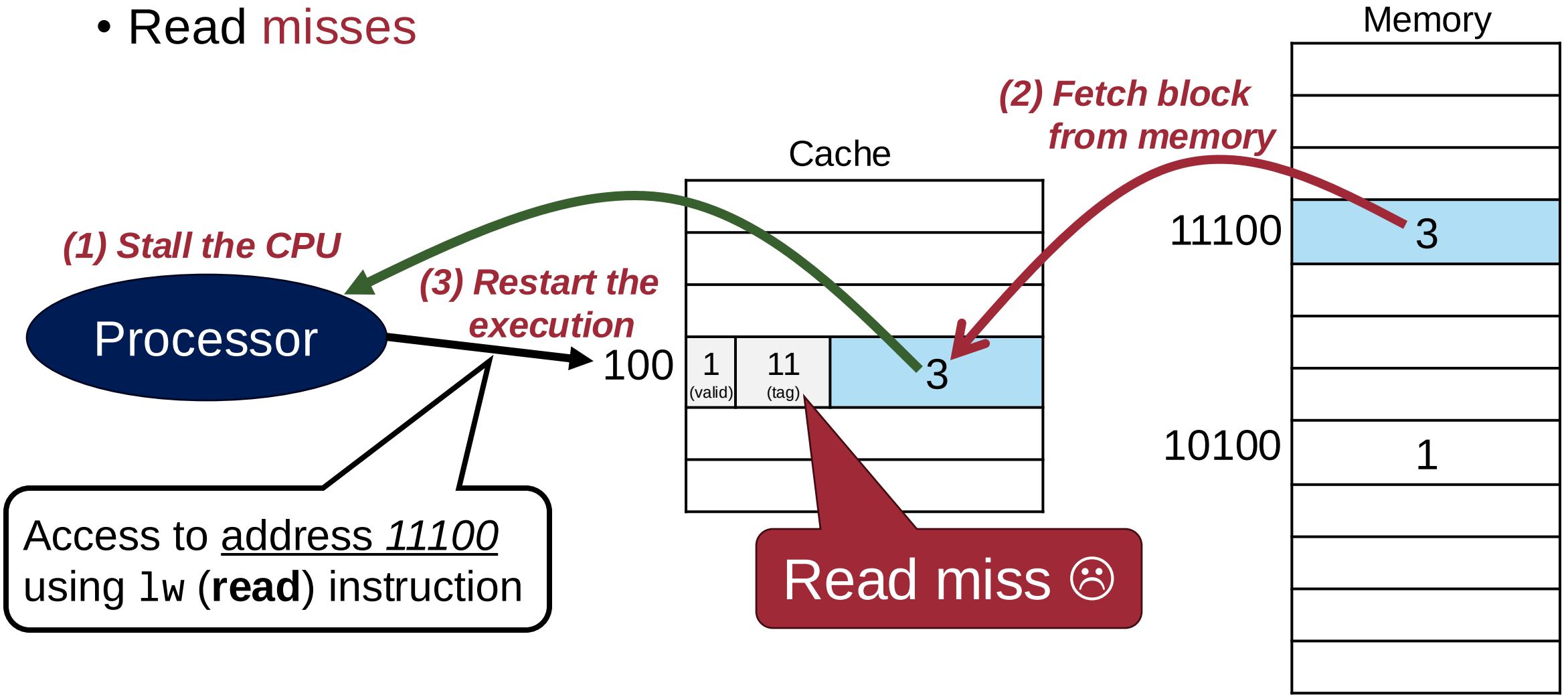
Hits vs. Misses: Read Misses

- Read misses



Hits vs. Misses: Read Misses

- Read misses



Summary: Hits vs. Misses

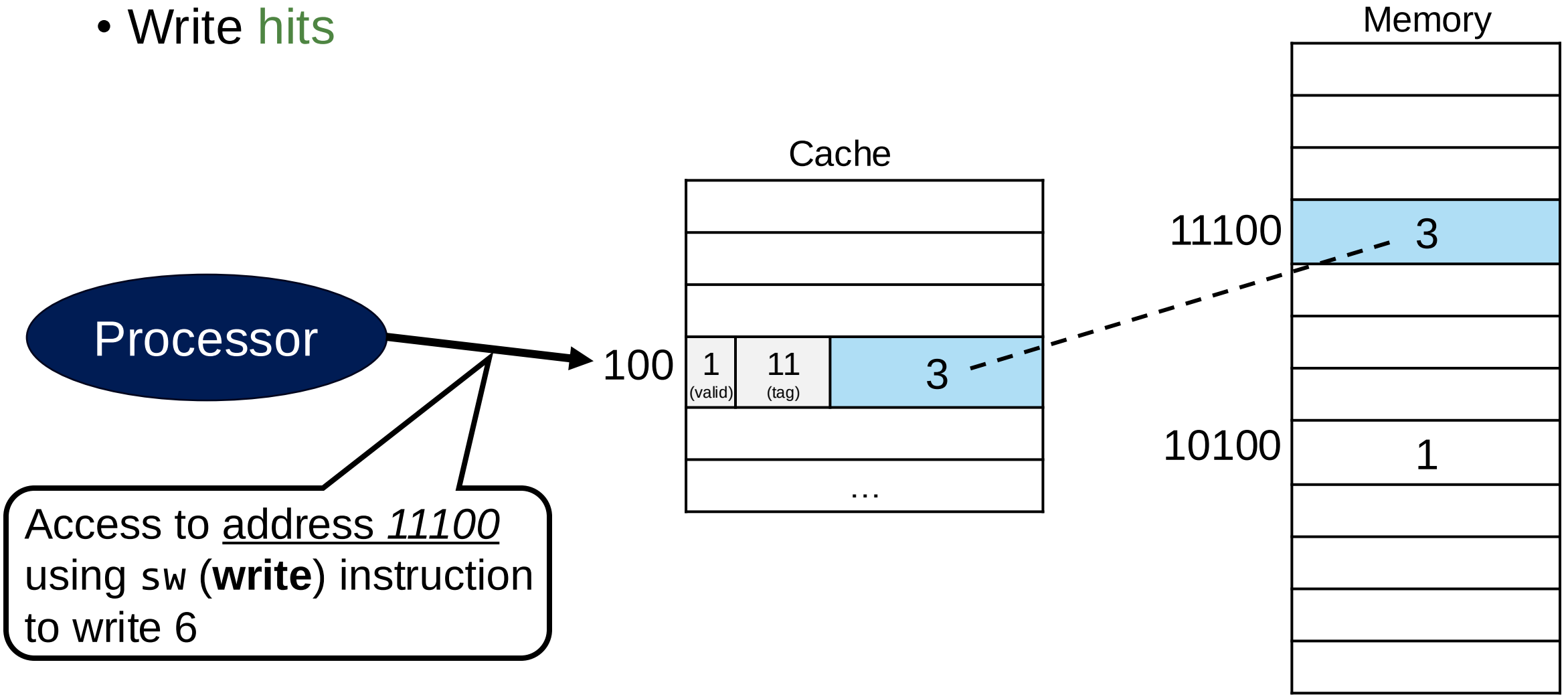
- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch block from memory, restart
- Write hits
- Write misses

Cases to consider for the instruction cache

Cases to consider for the data cache

Hits vs. Misses: Write Hits

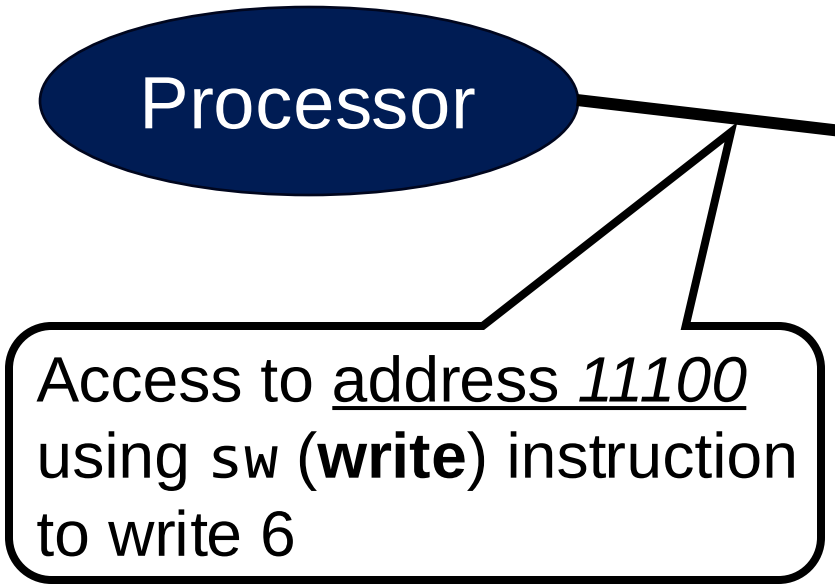
- Write hits

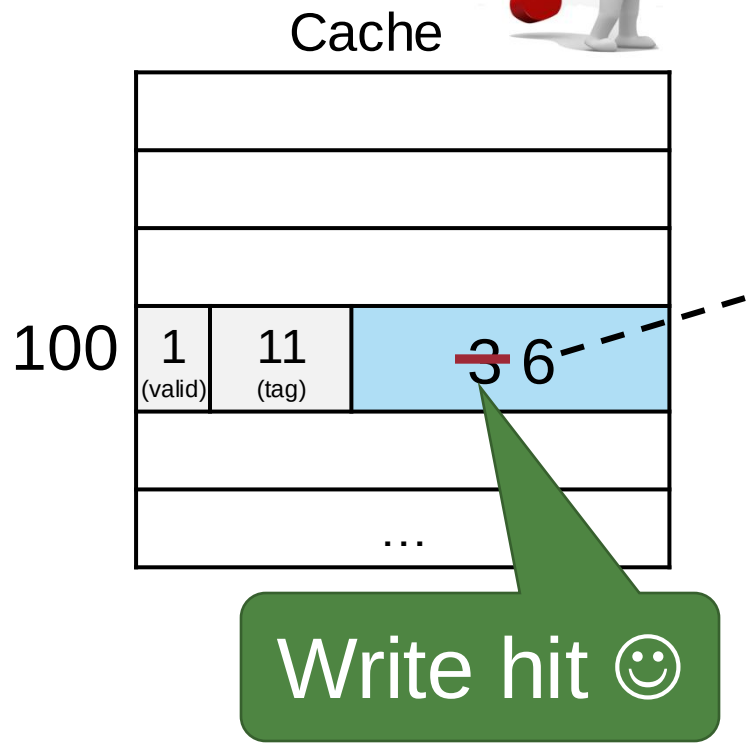


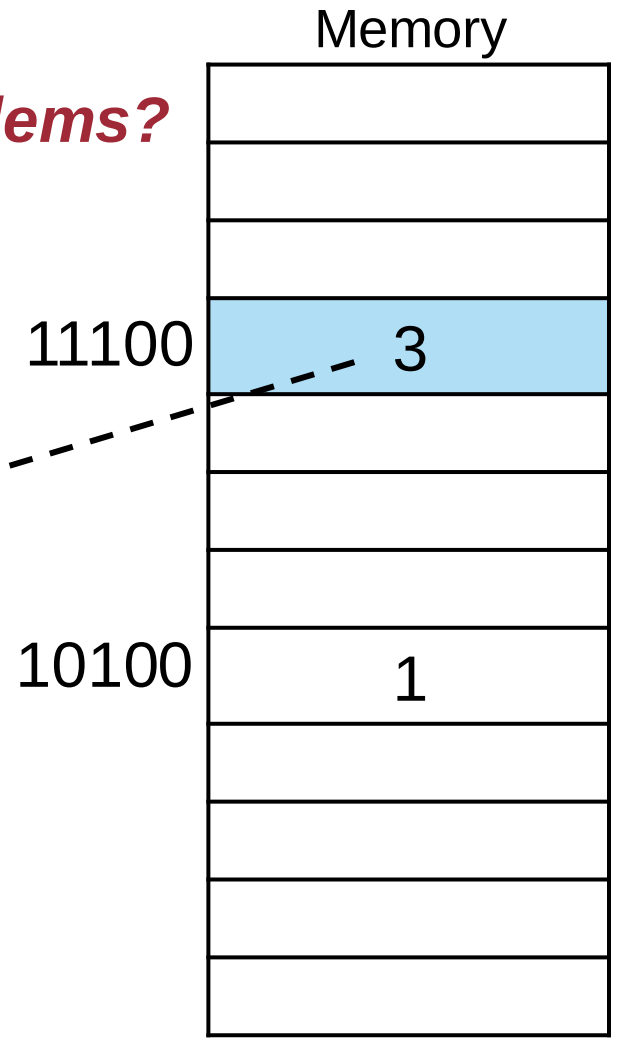
Hits vs. Misses: Write Hits

- Write hits

Problems?







Hits vs. Misses: Write Hits

- Write hits

Problems?

Memory

11100 3

Cache

100

1 (valid)	11 (tag)	~~3~~ 6
		…

Processor

Access to address *11100* using `sw` (**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

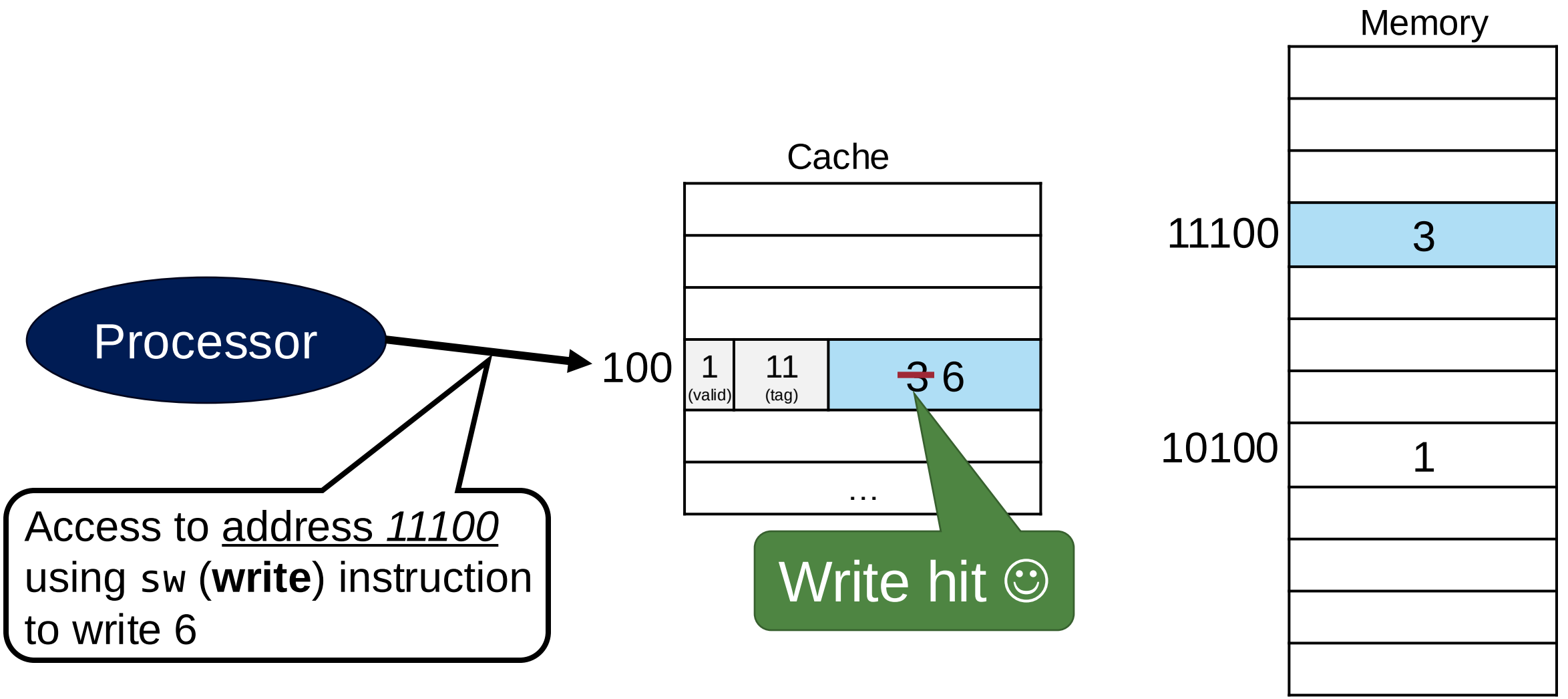
Write Policies

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

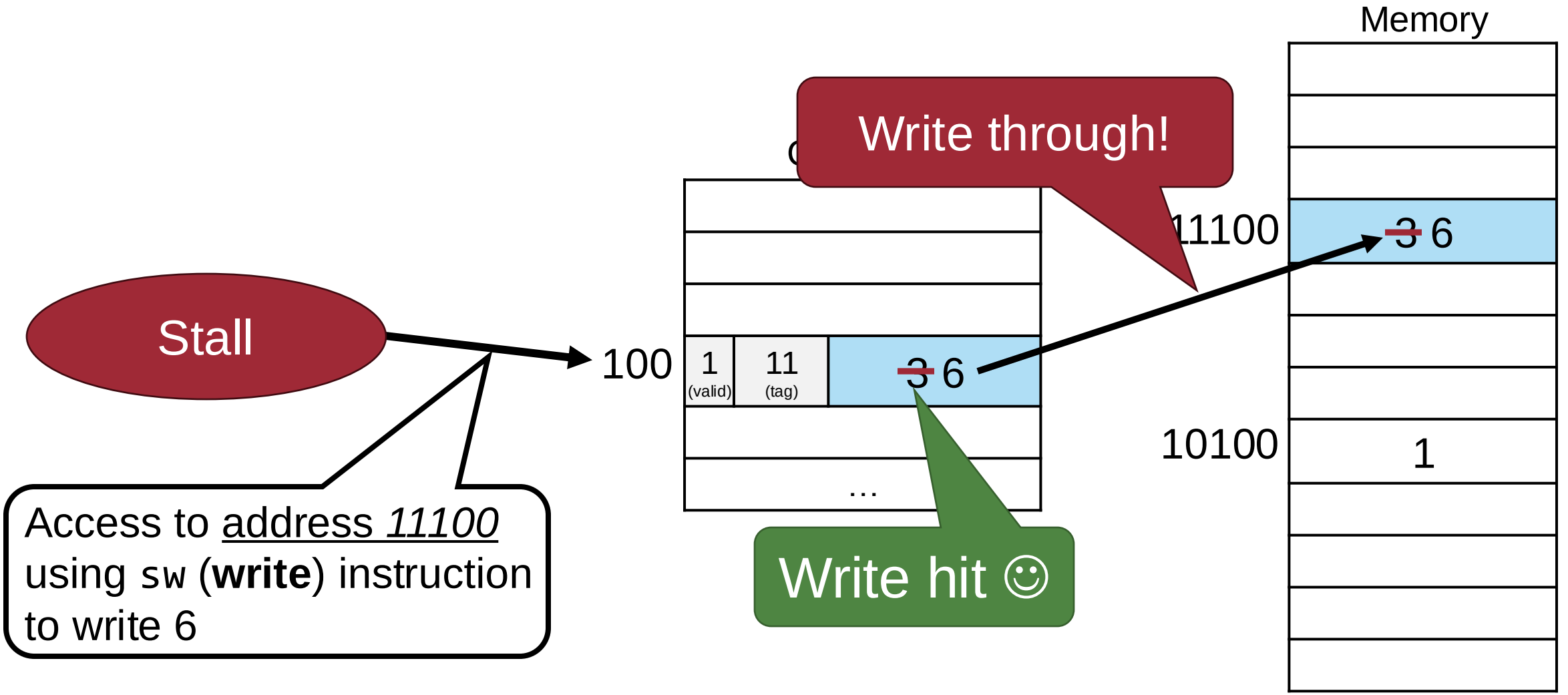
  (1) Write through: On each write hit, the information is written to both in the cache and in the memory

  (2) Write back

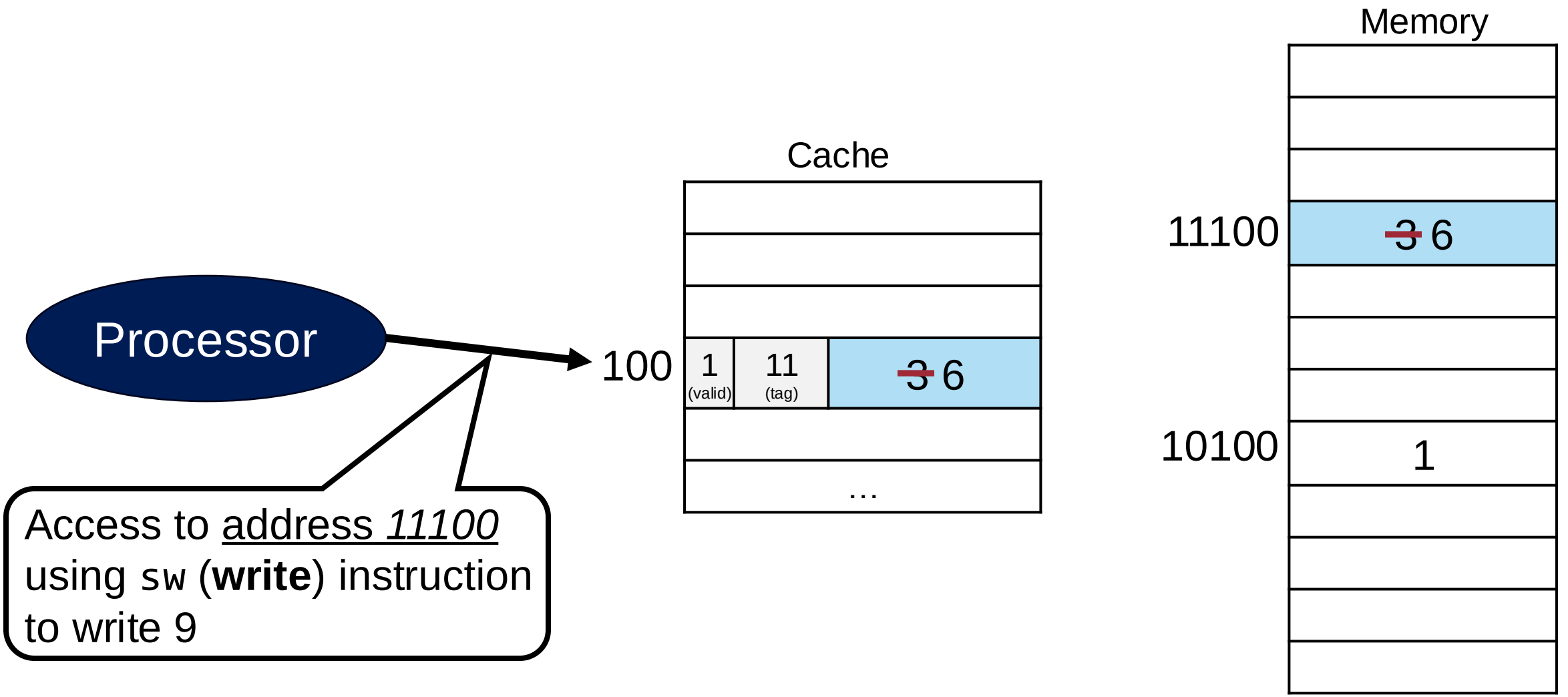
Write Through



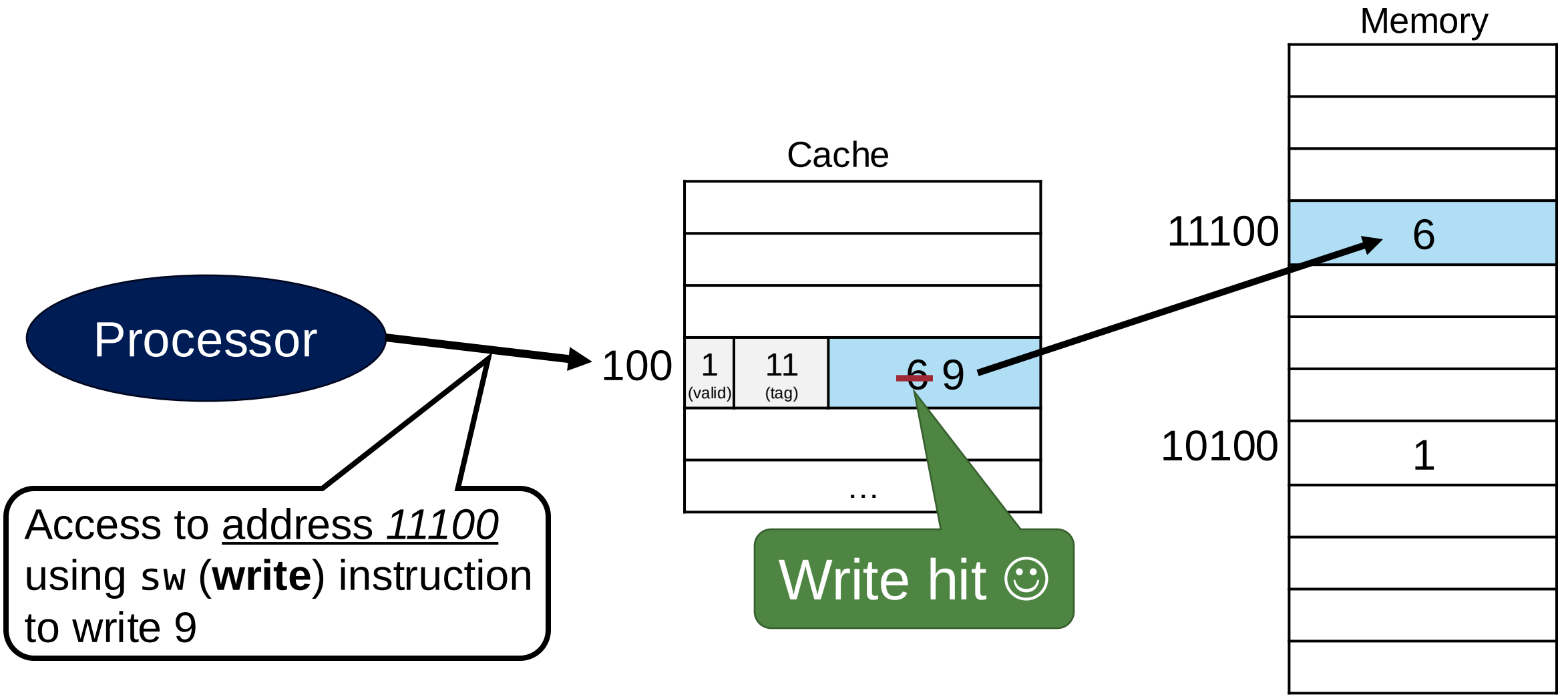
Write Through



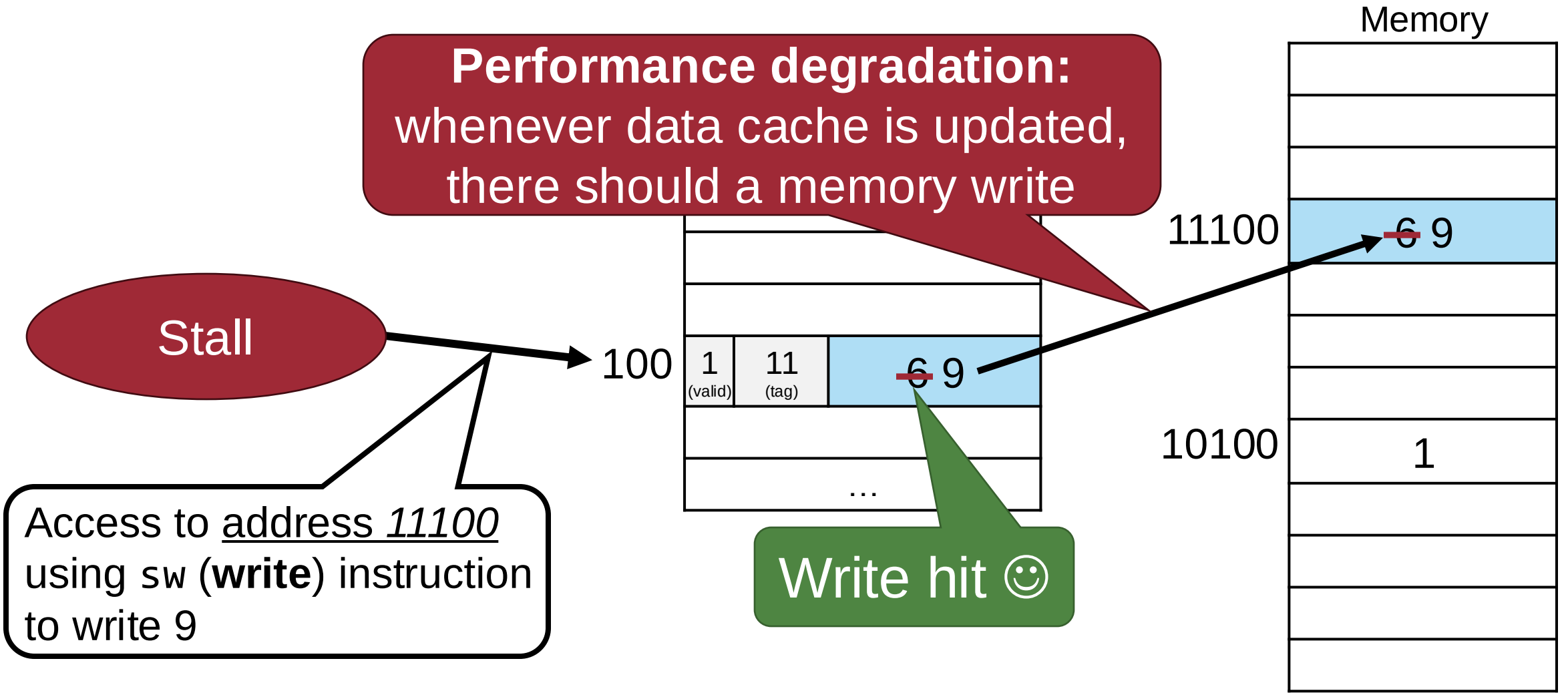
Write Through



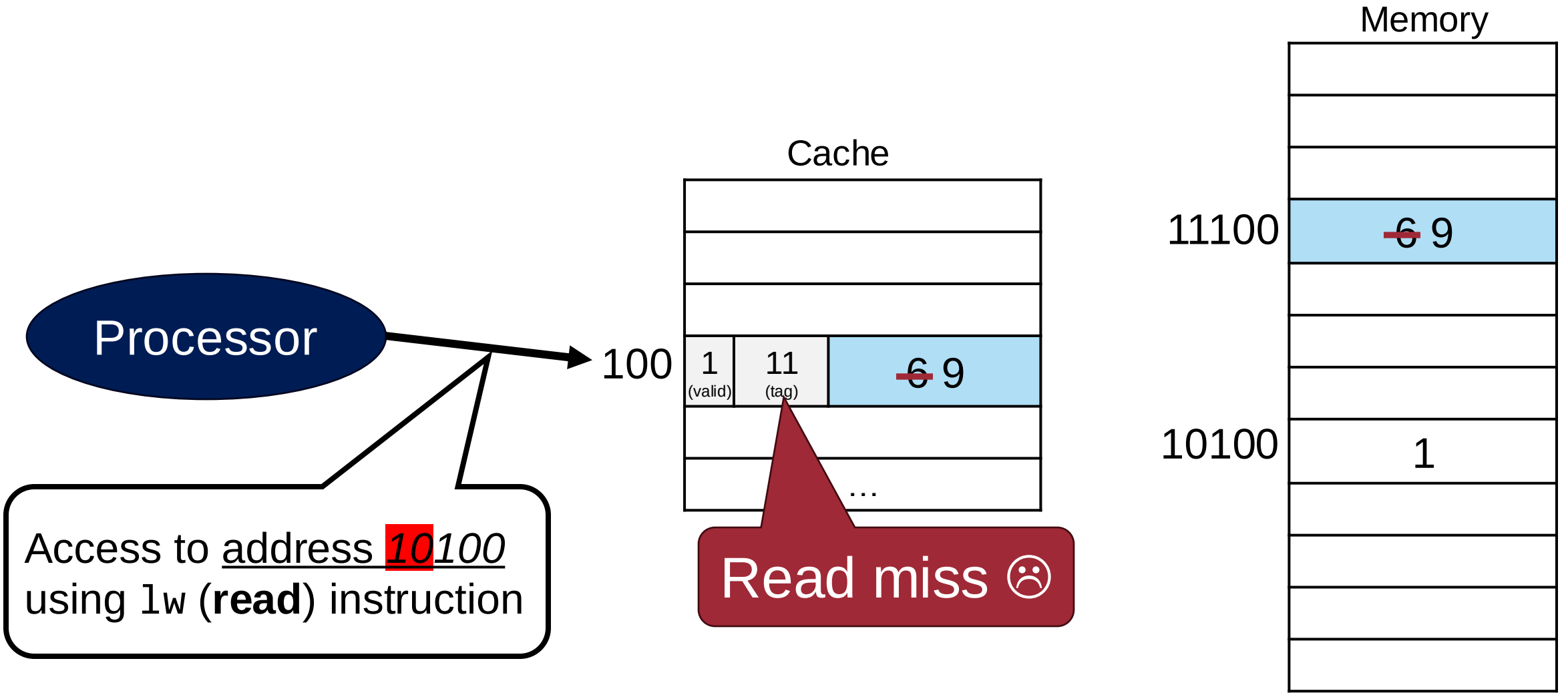
Write Through



Write Through

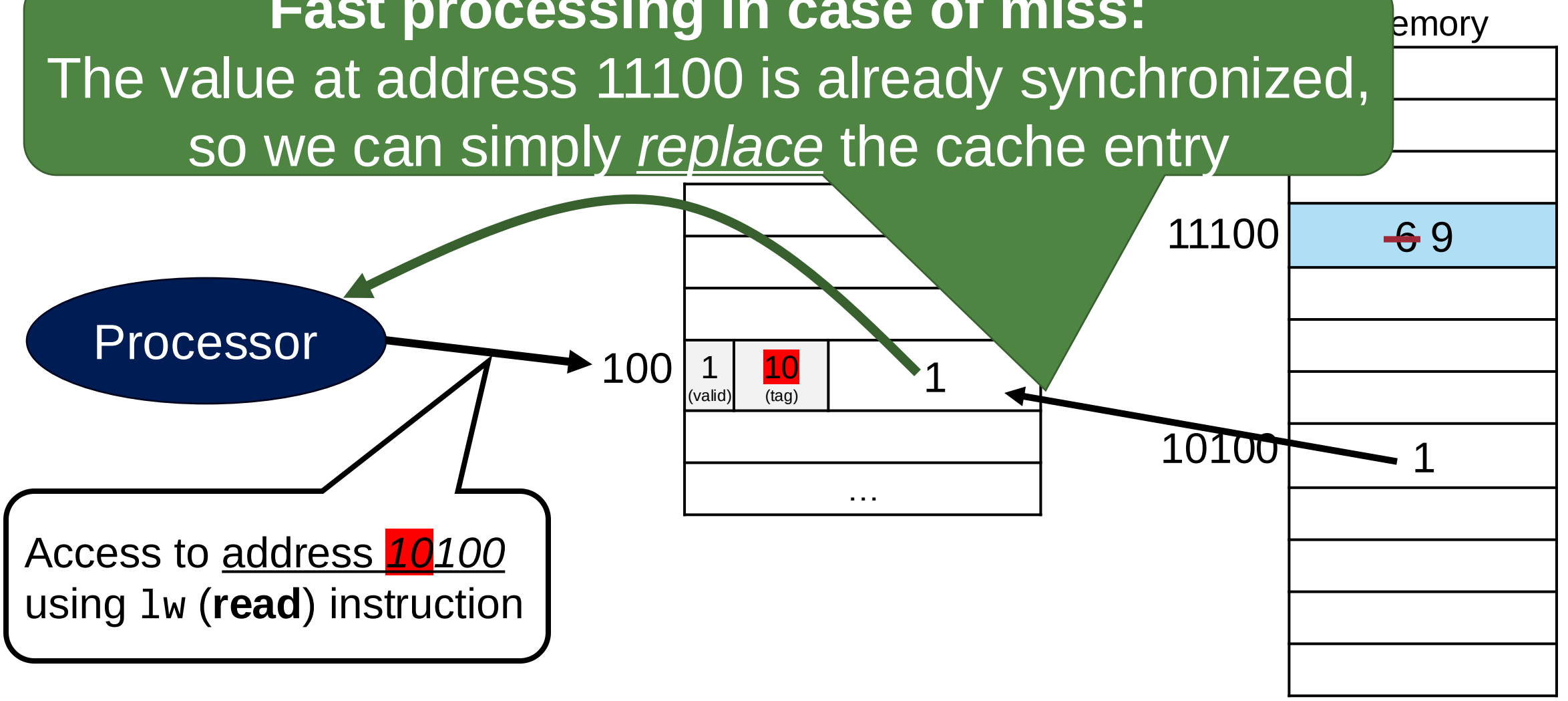


Write Through

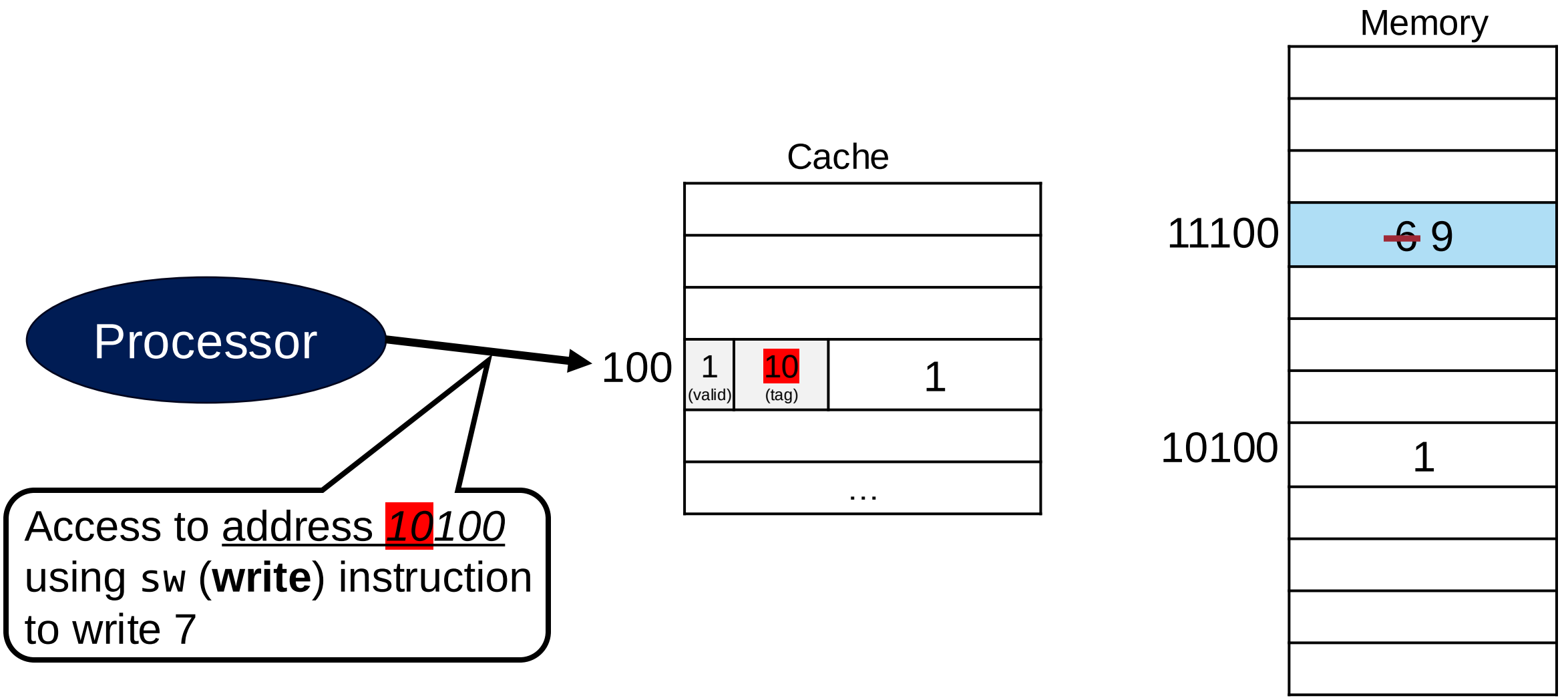


Write Through

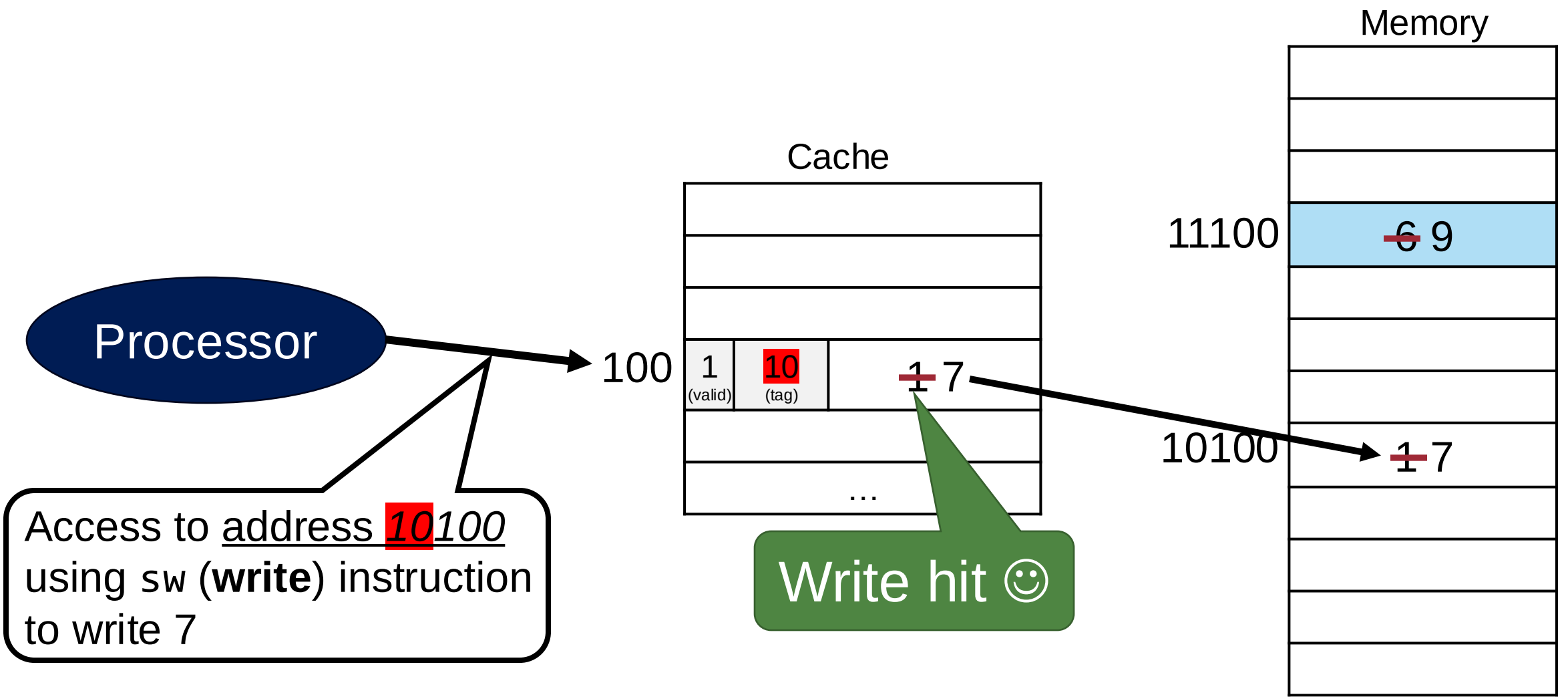
Fast processing in case of miss:
The value at address 11100 is already synchronized, so we can simply *replace* the cache entry



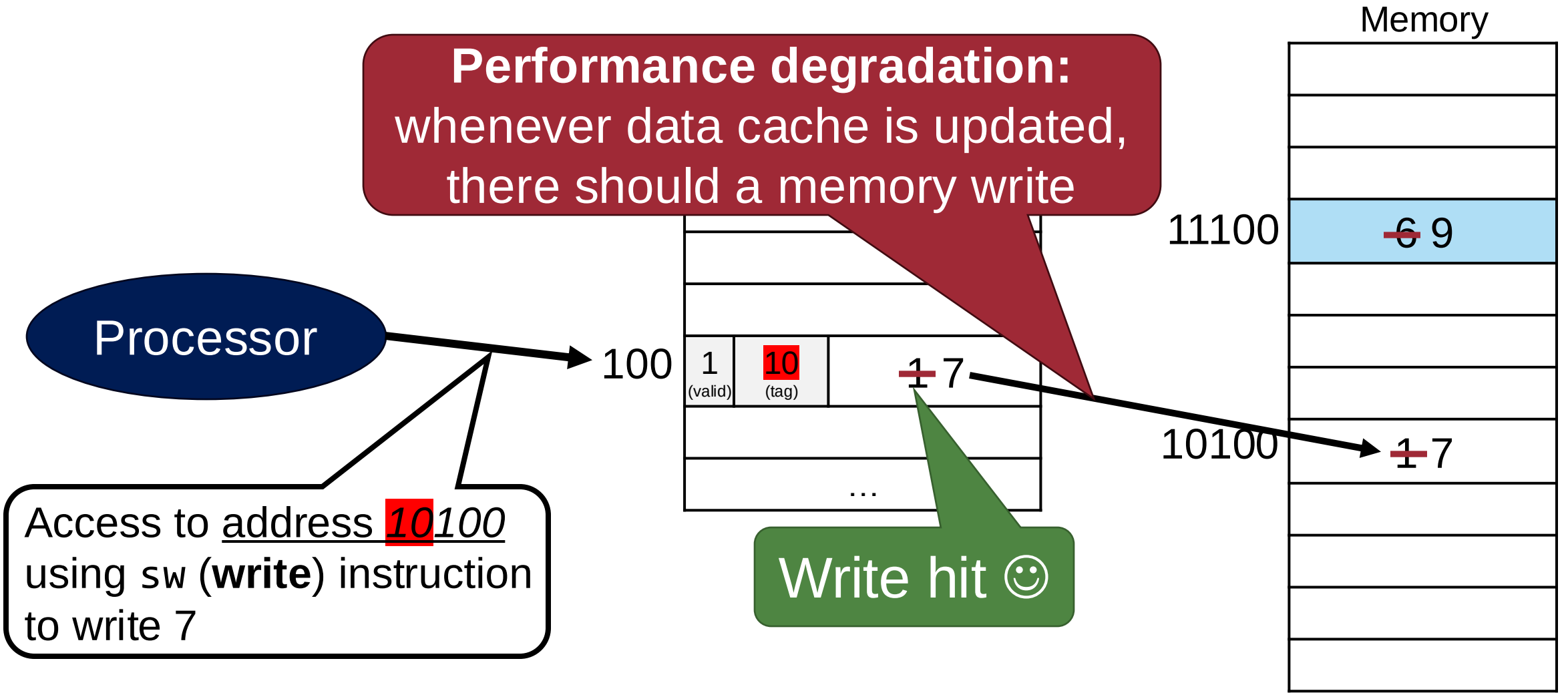
Write Through



Write Through



Main Disadvantage: Performance Degradation



Main Disadvantage: Performance Degradation

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

Memory

11100 ~~6~~ 9

Processor

100 1 (valid) 10 (tag) ~~1~~ 7

10100 ~~1~~ 7

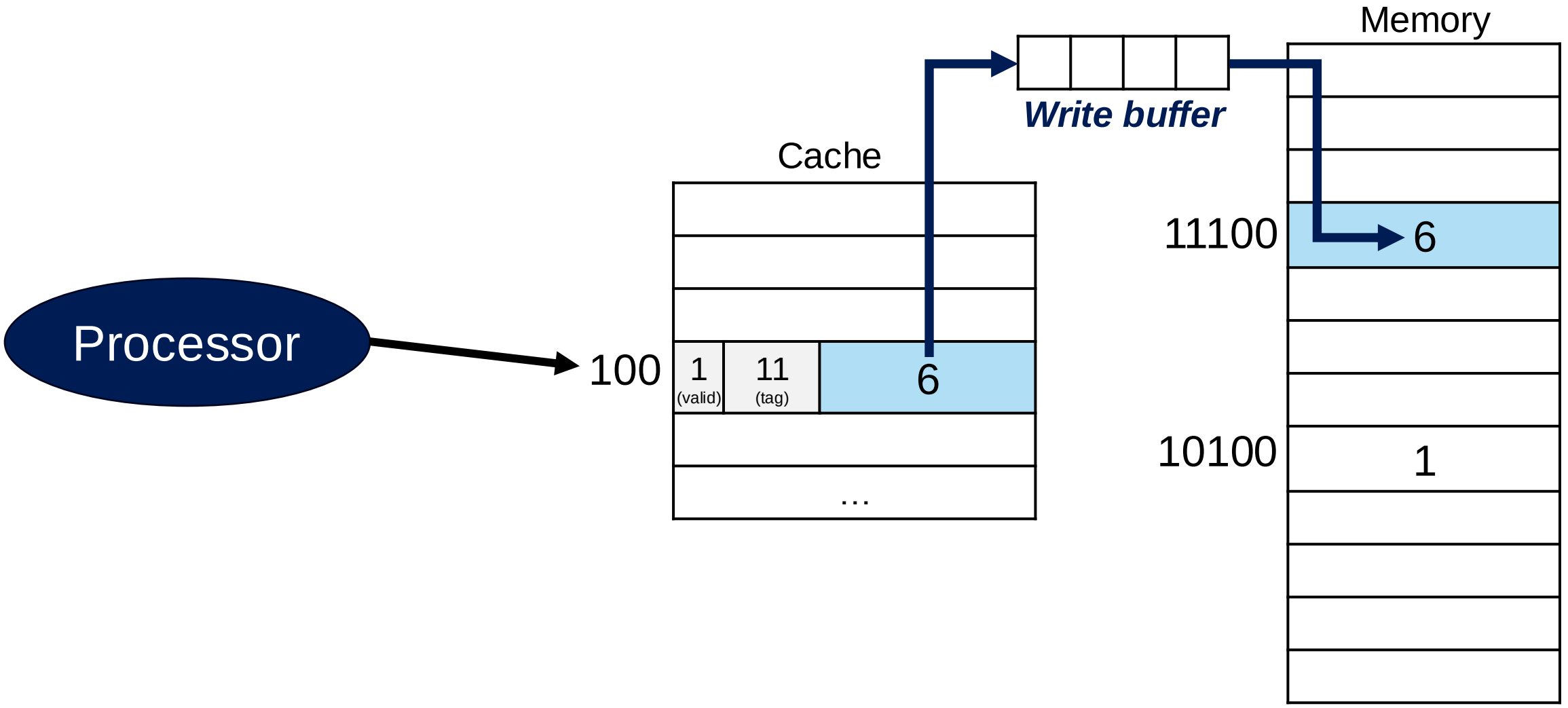
Access to address 10100

Solution? → Write Buffer

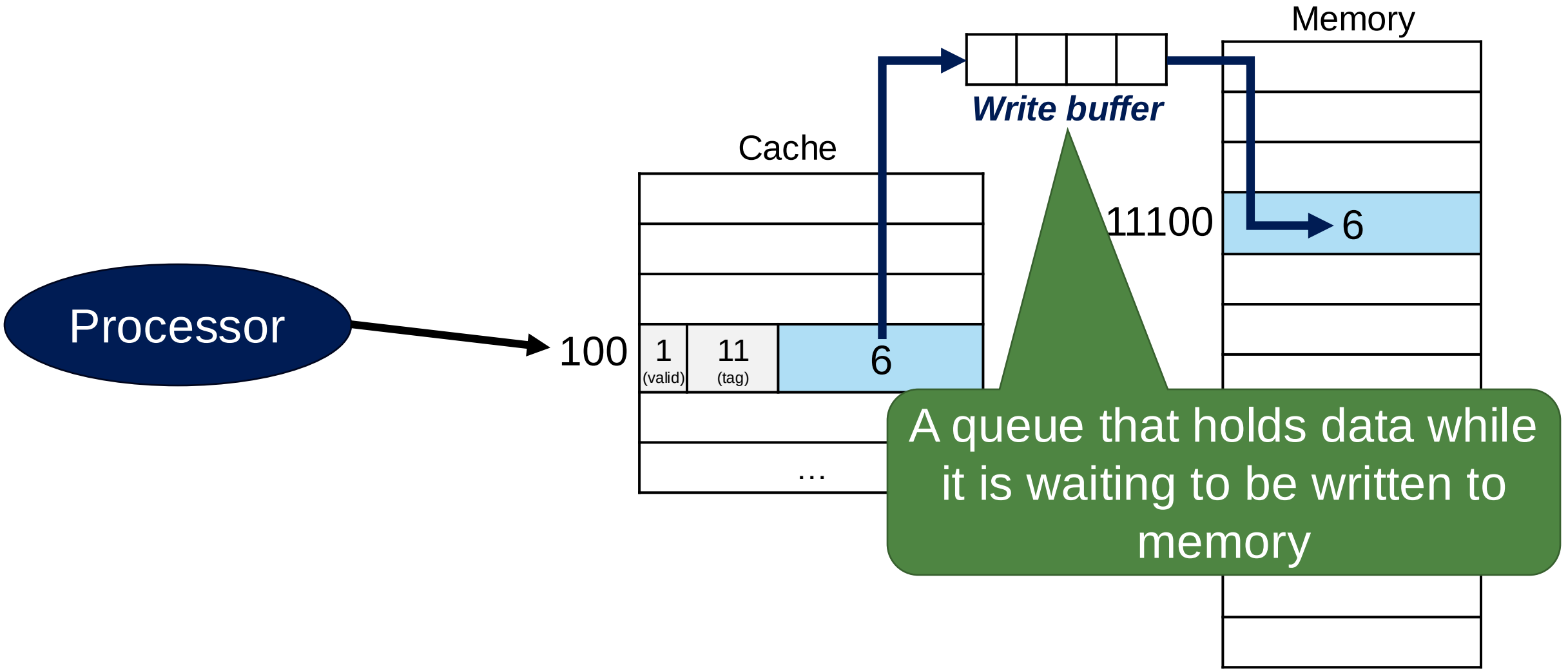
If base CPI = 1, 10% of instructions are stores,
write to memory takes 100 extra cycles

CPI = 1 + 0.1 x 100 = 11

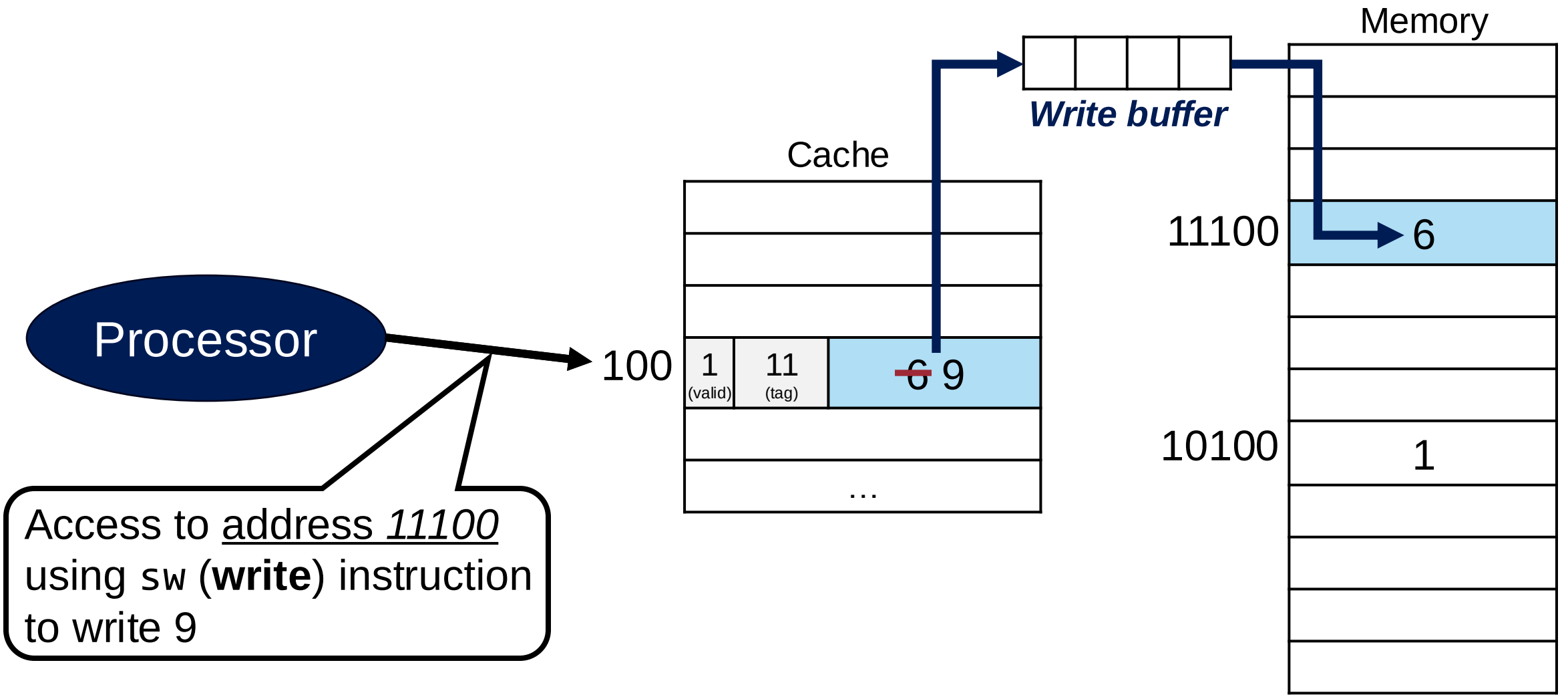
Simple Idea: Write Buffer + Write Through



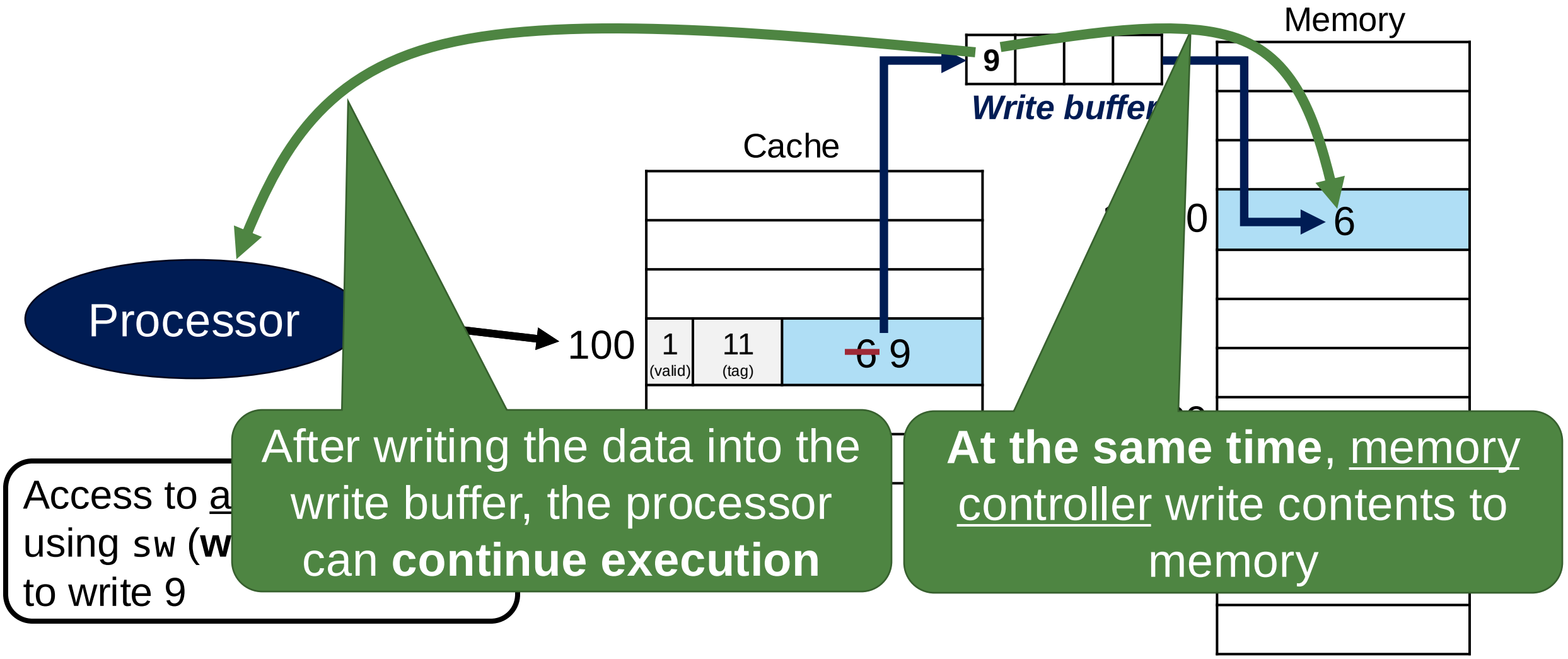
Simple Idea: Write Buffer + Write Through



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 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)
      - ✓Solution: write buffer

  (2) Write back

Write Policies

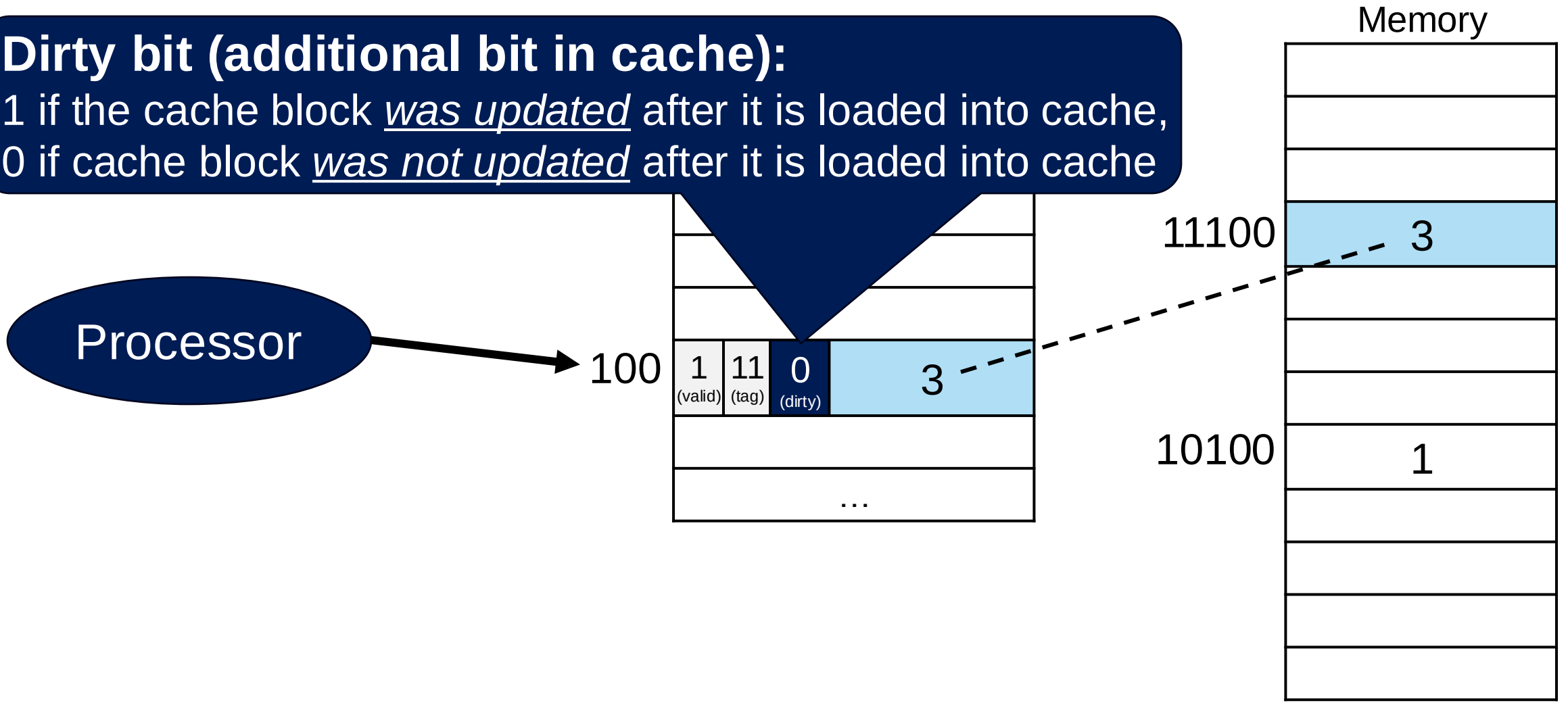
- Write hits
  - Cache and memory would be inconsistent! What can we do?

  (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)
      - ✓Solution: write buffer

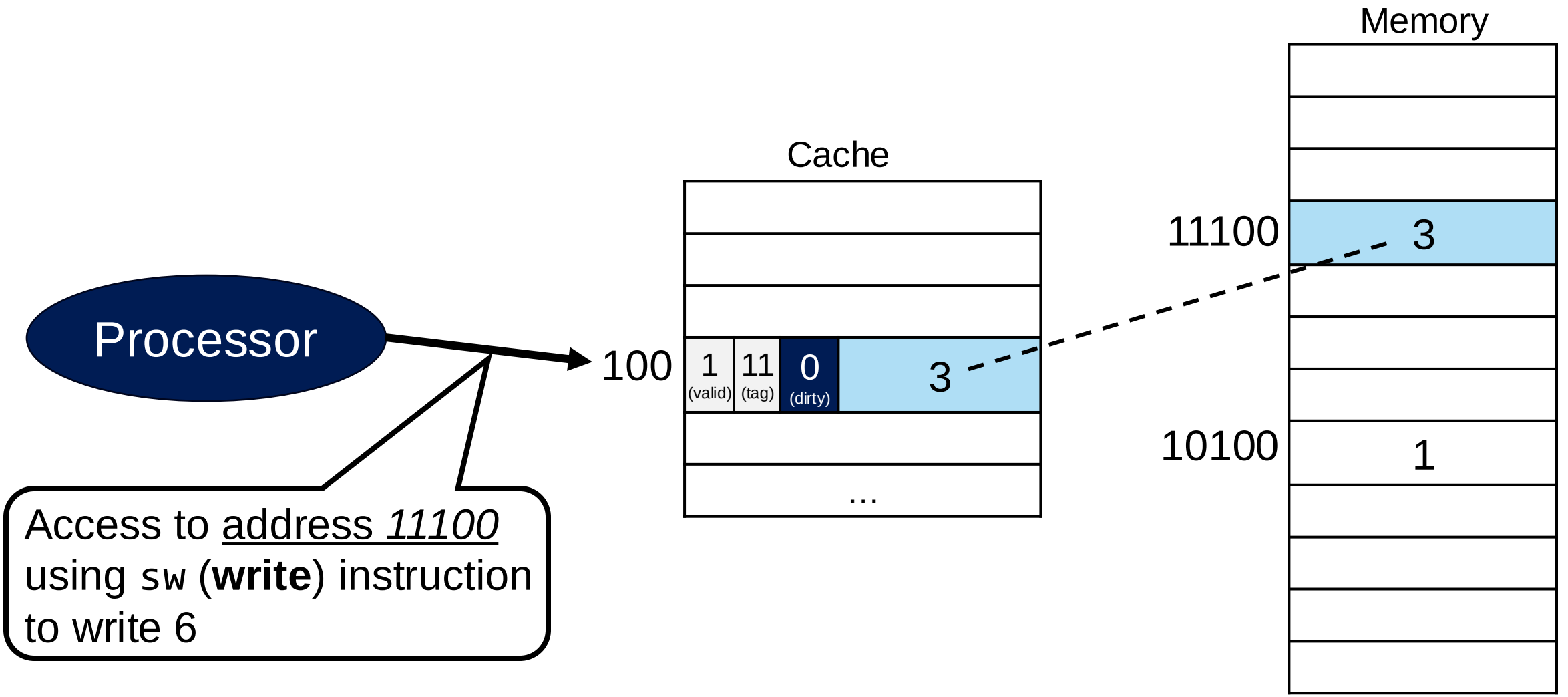
  (2) Write back: On each write hit, update only the block in cache. The modified cache block is written to memory ***only when it is replaced***
    - Keep track of whether each block is *dirty (additional bit)*.

Write Back – Dirty Bit

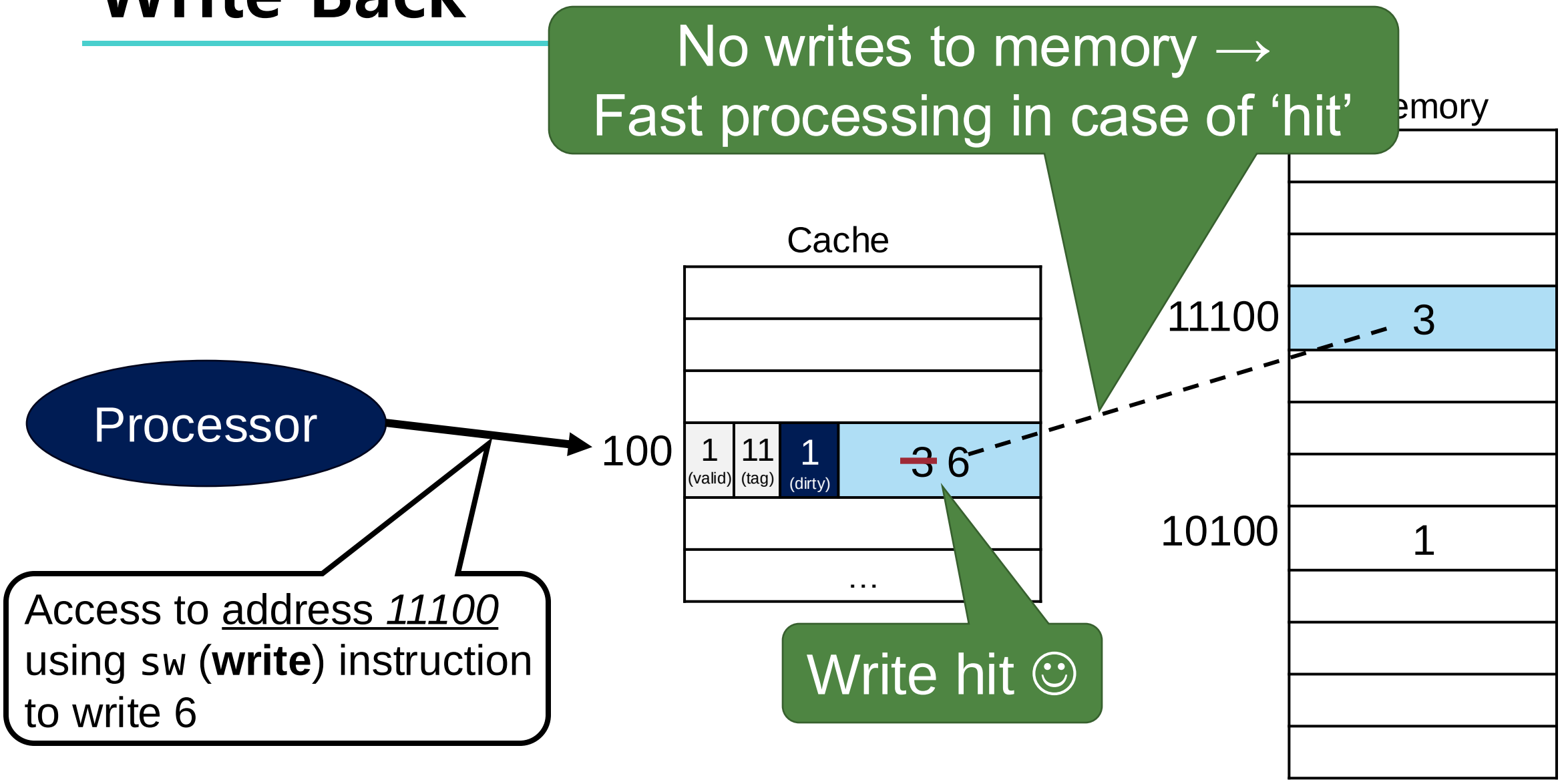
Dirty bit (additional bit in cache):
1 if the cache block *was updated* after it is loaded into cache,
0 if cache block *was not updated* after it is loaded into cache



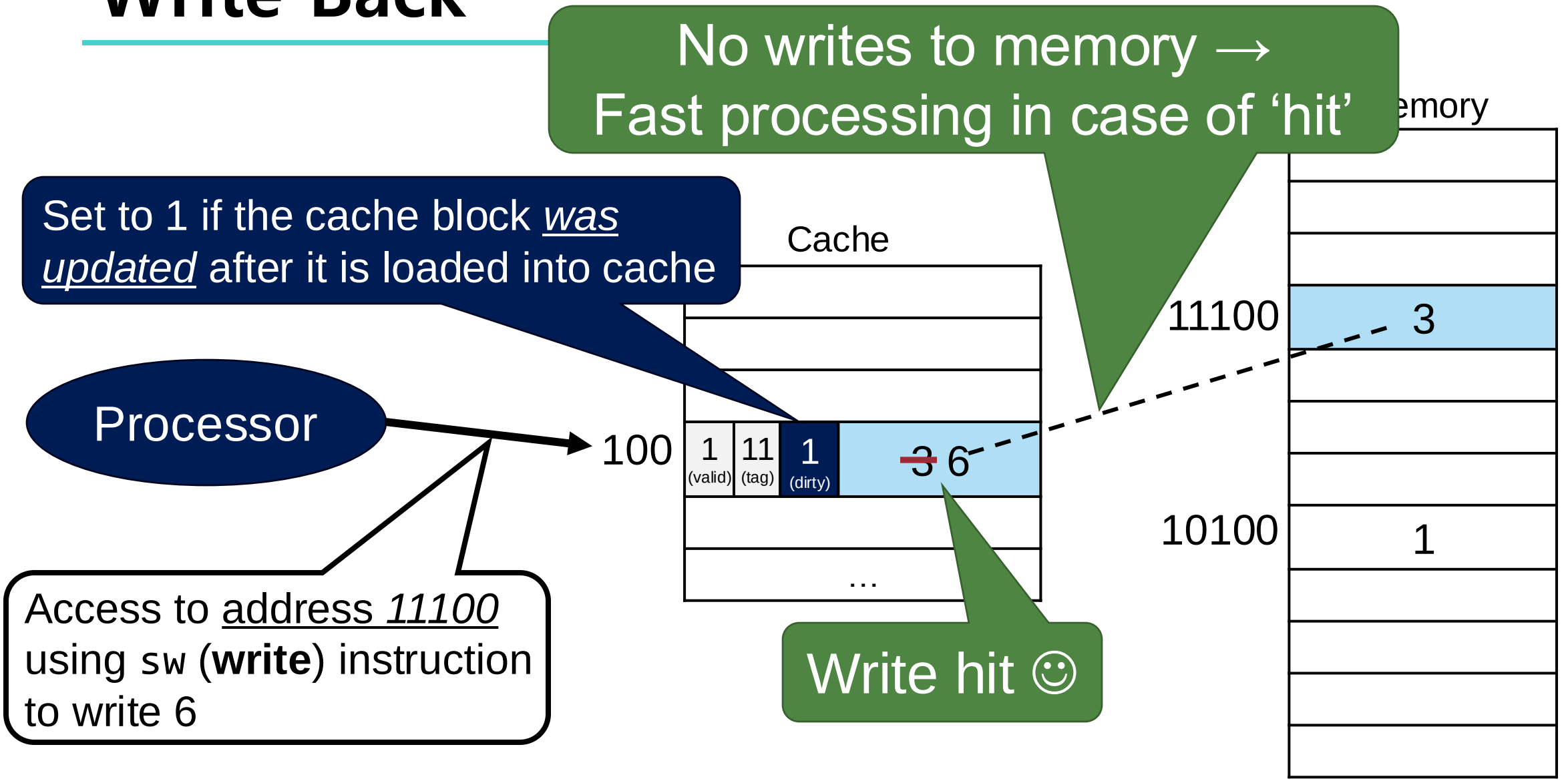
Write Back – Dirty Bit



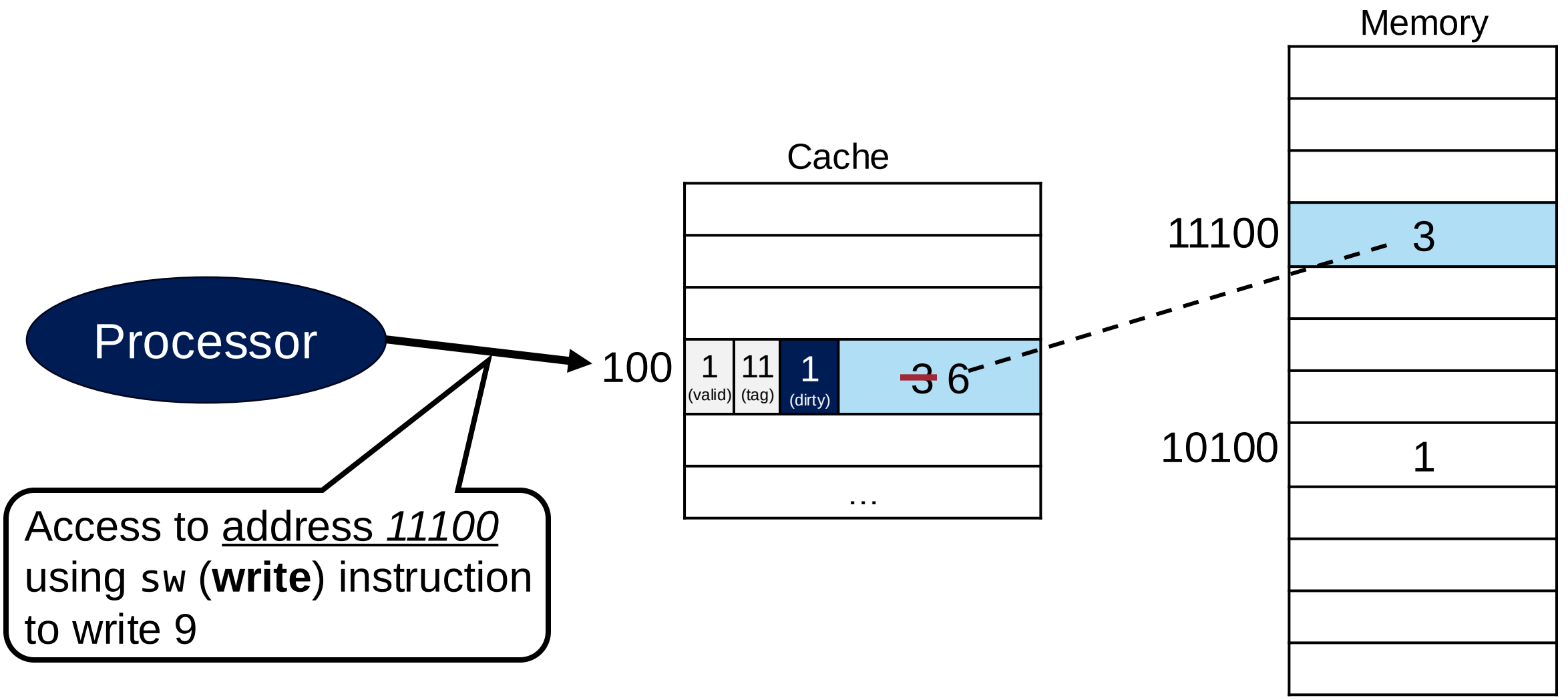
Write Back


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

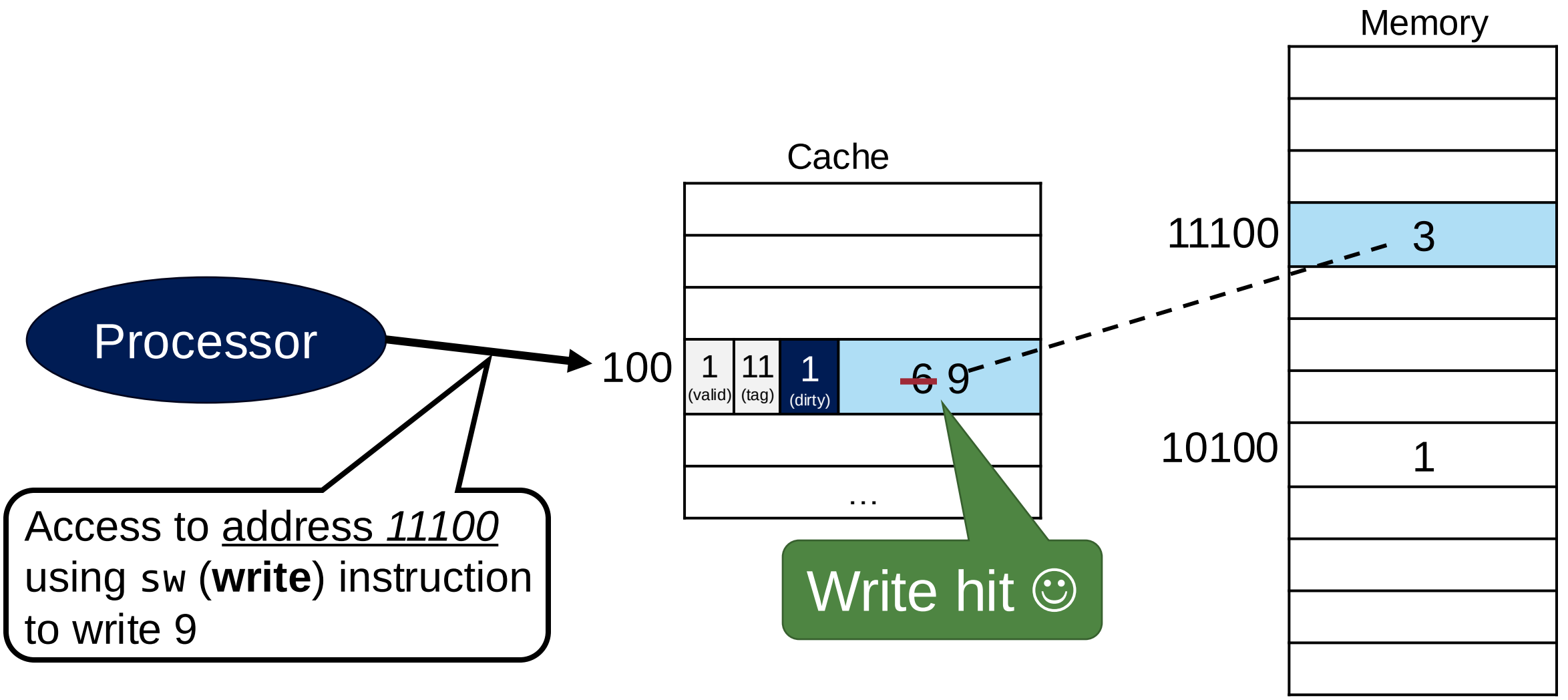
Write Back


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 sw (**write**) instruction to write 6
Cache
100
1 (valid)
11 (tag)
1 (dirty)
3 6
…
Write hit ☺
emory
11100
3
10100
1

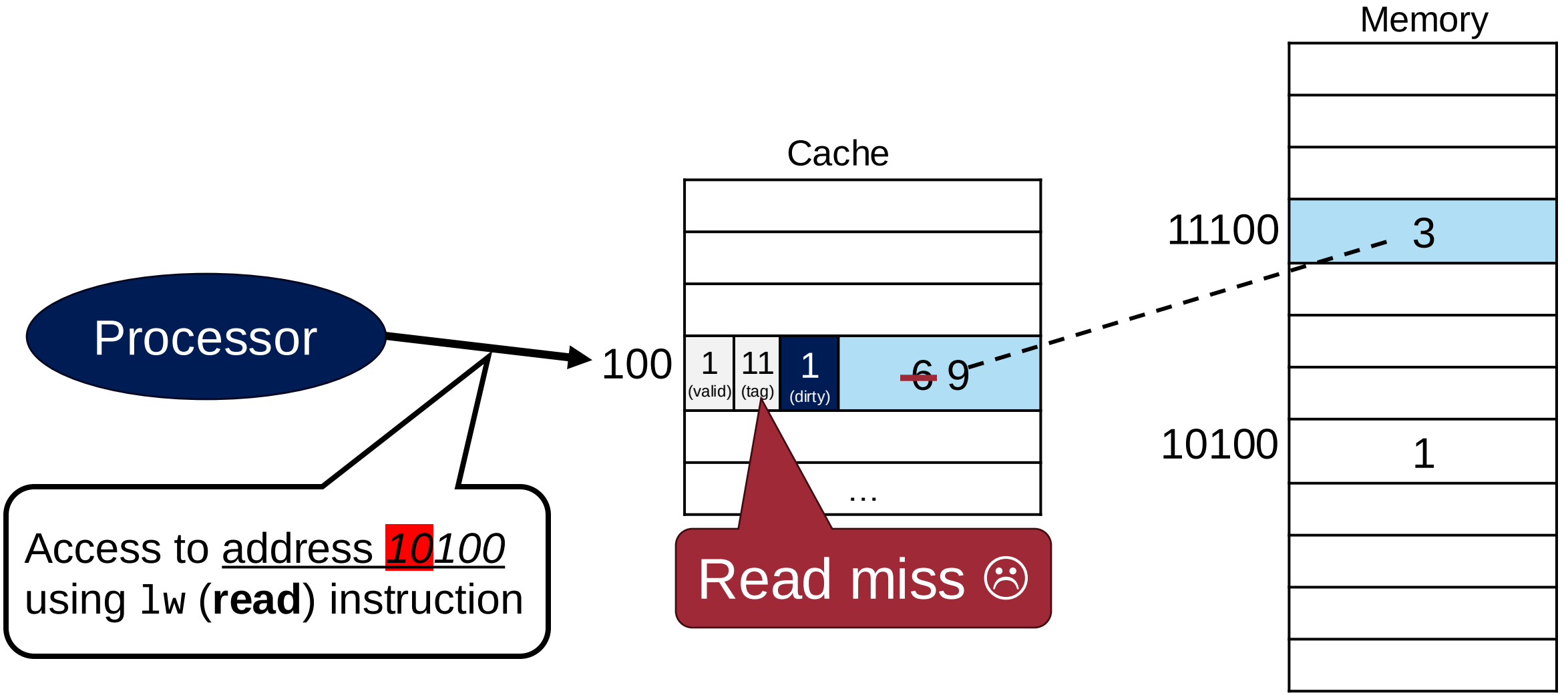
Write Back


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

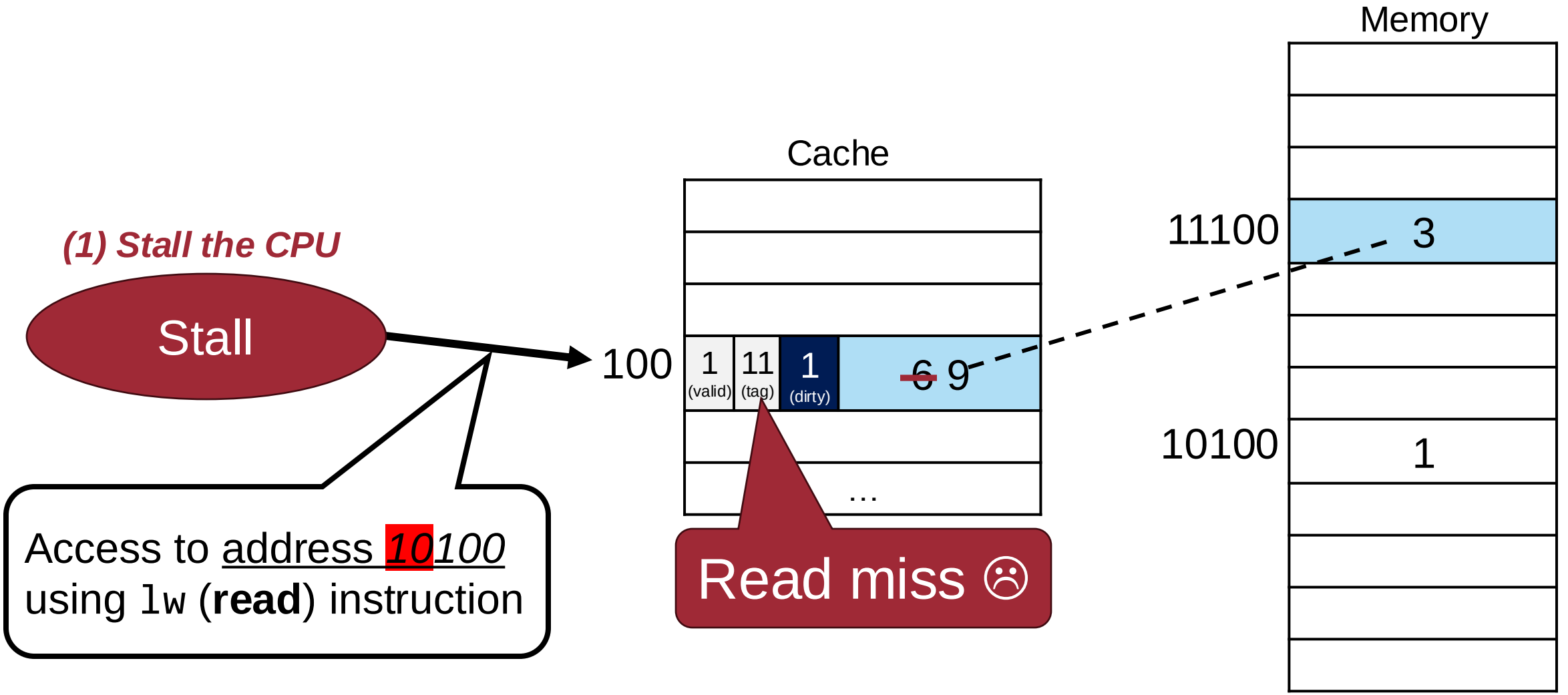
Write Back


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

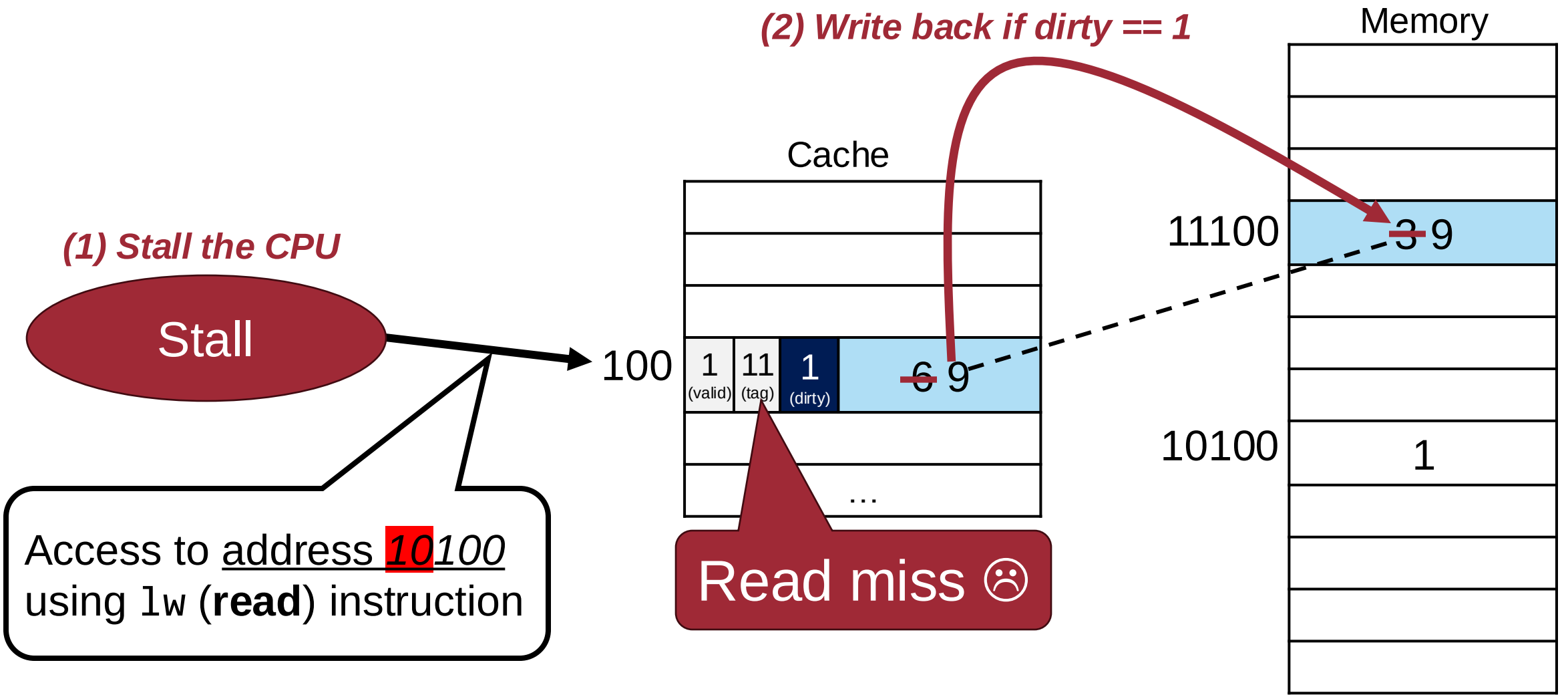
Write Back


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

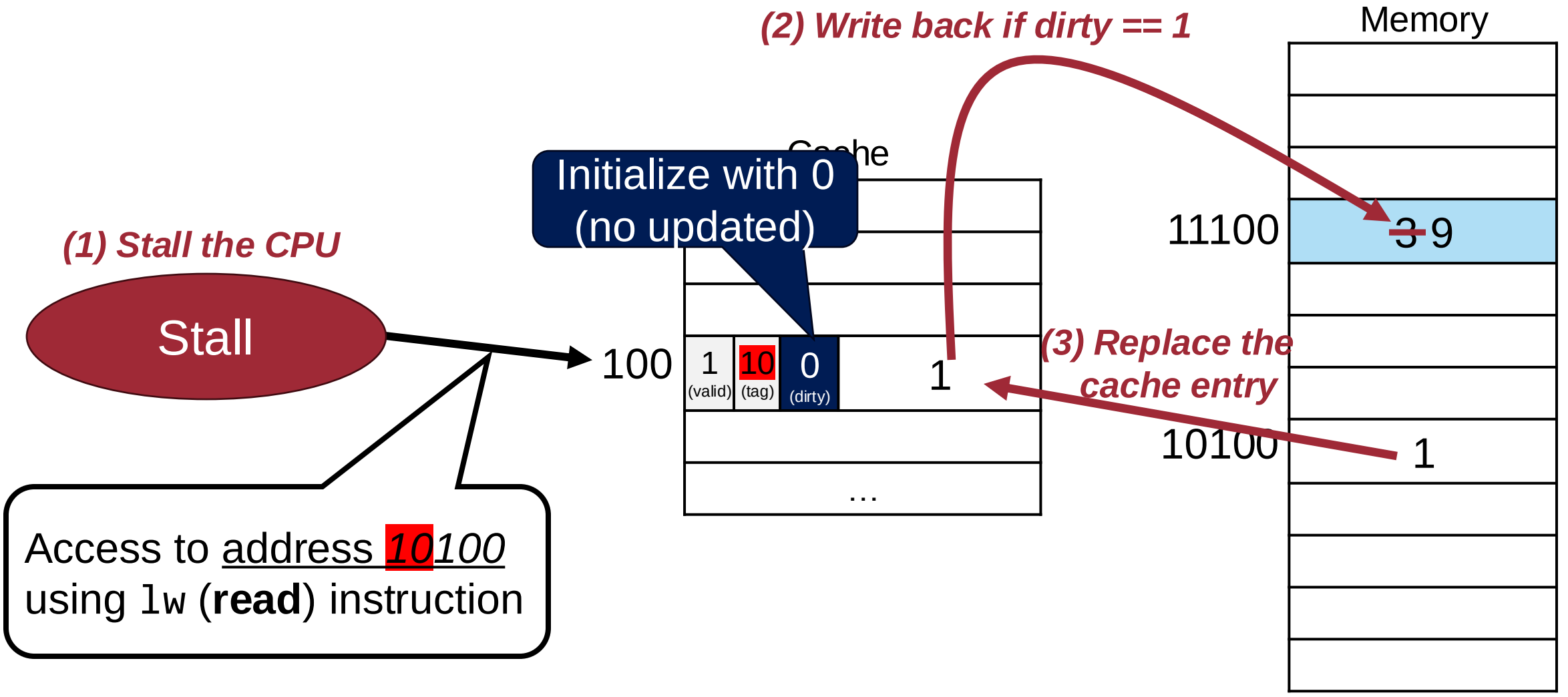
Write Back


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

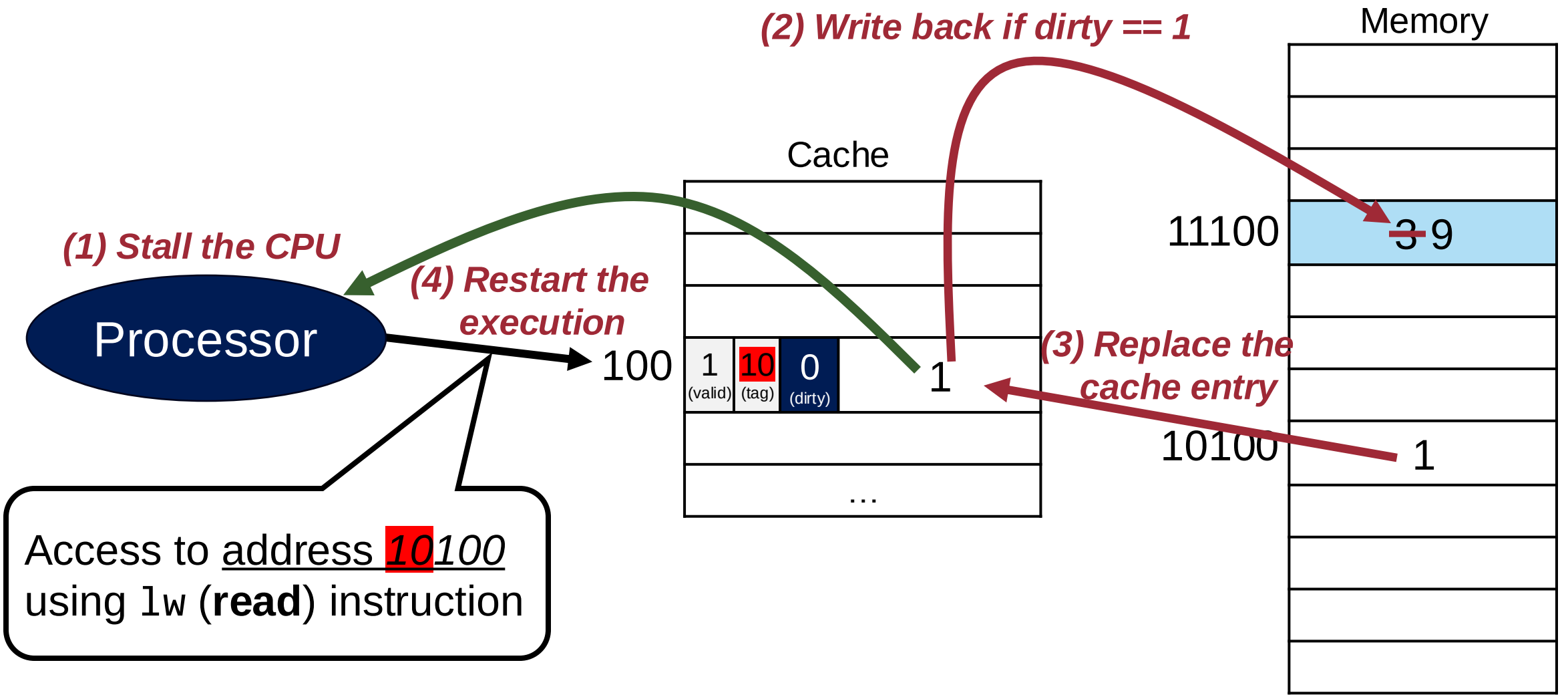
Write Back


(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 lw (read) instruction
Read miss ☹

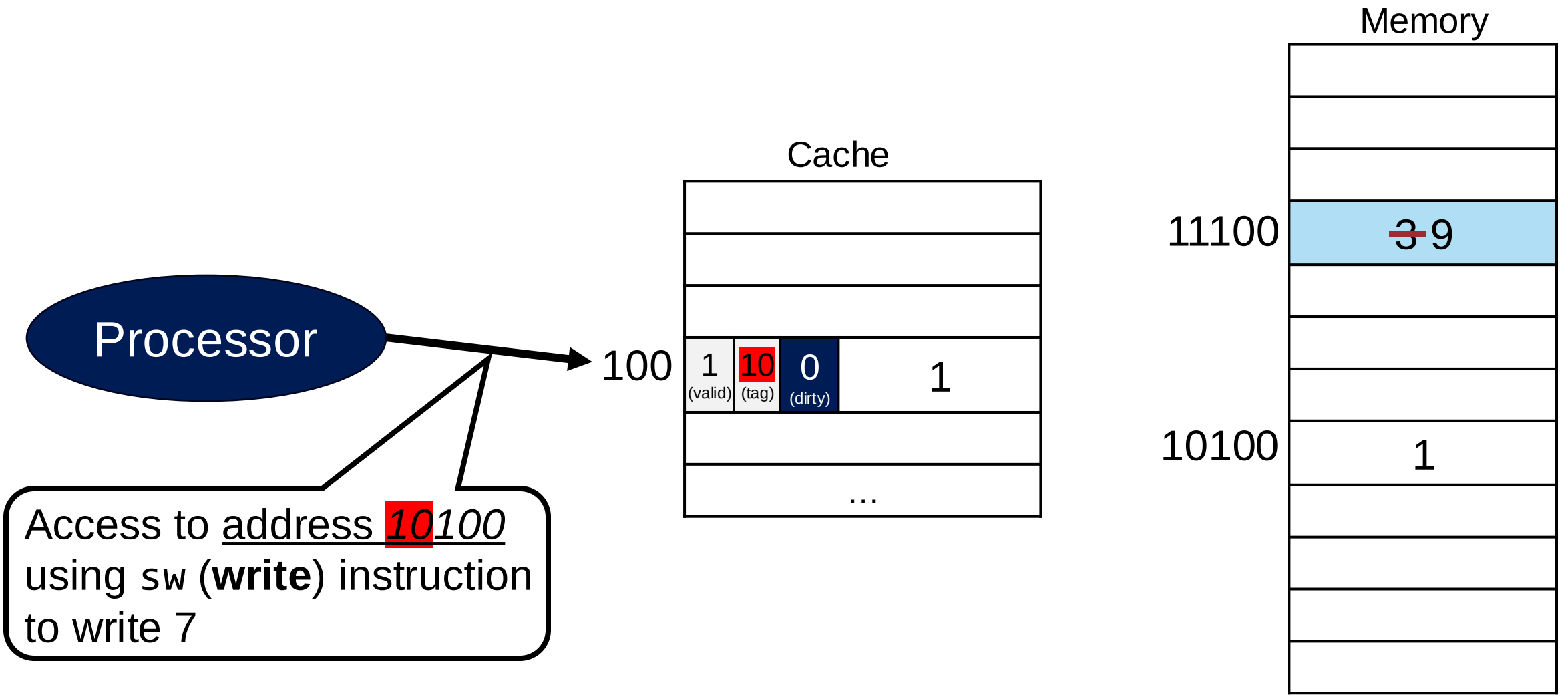
Write Back


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

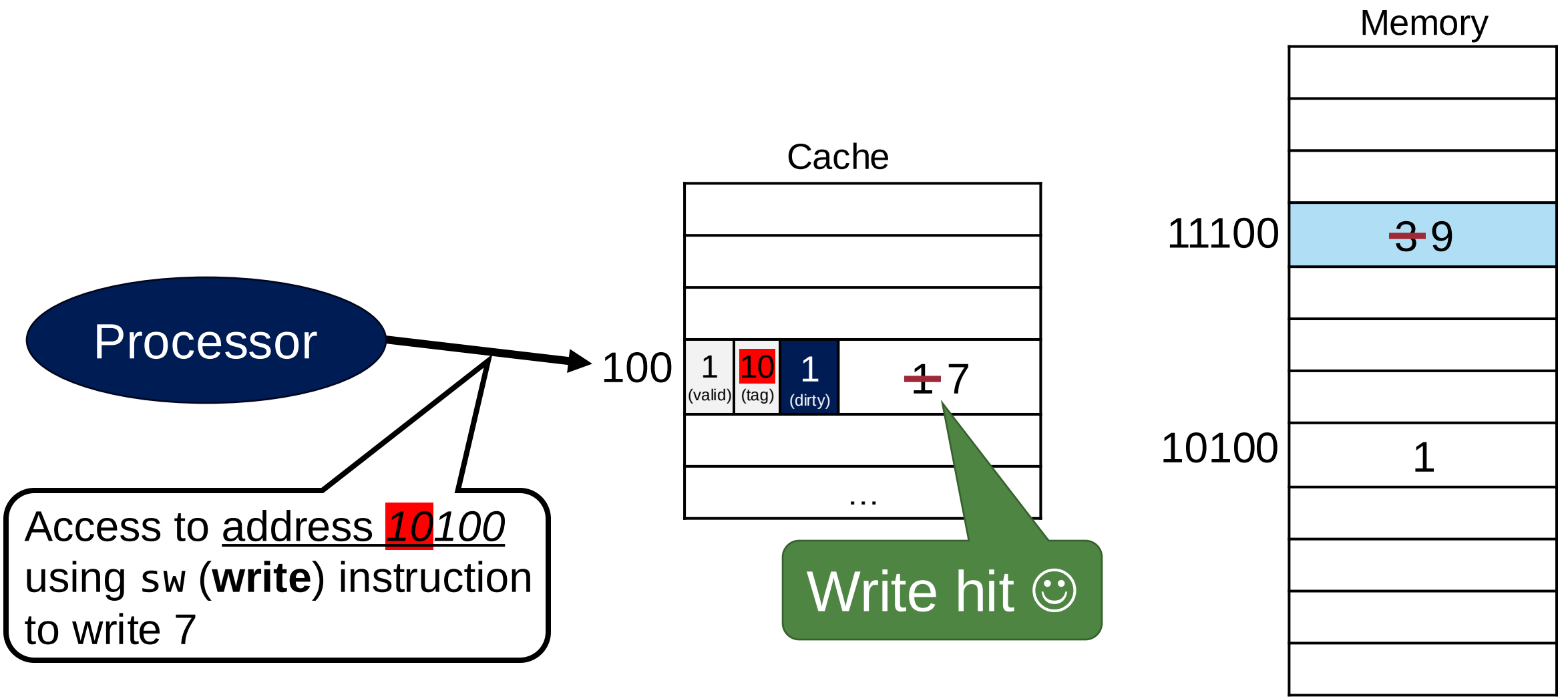
Write Back


(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 lw (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 sw (write) instruction to write 7

Write Back


Memory
11100
3 9
10100
1
Cache
100
1 (valid)
10 (tag)
1 (dirty)
1 7
...
Processor
Access to address 10100 using sw (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)
      - ✓Solution: write buffer

  (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

Cases to consider for the instruction cache

Cases to consider for the data cache

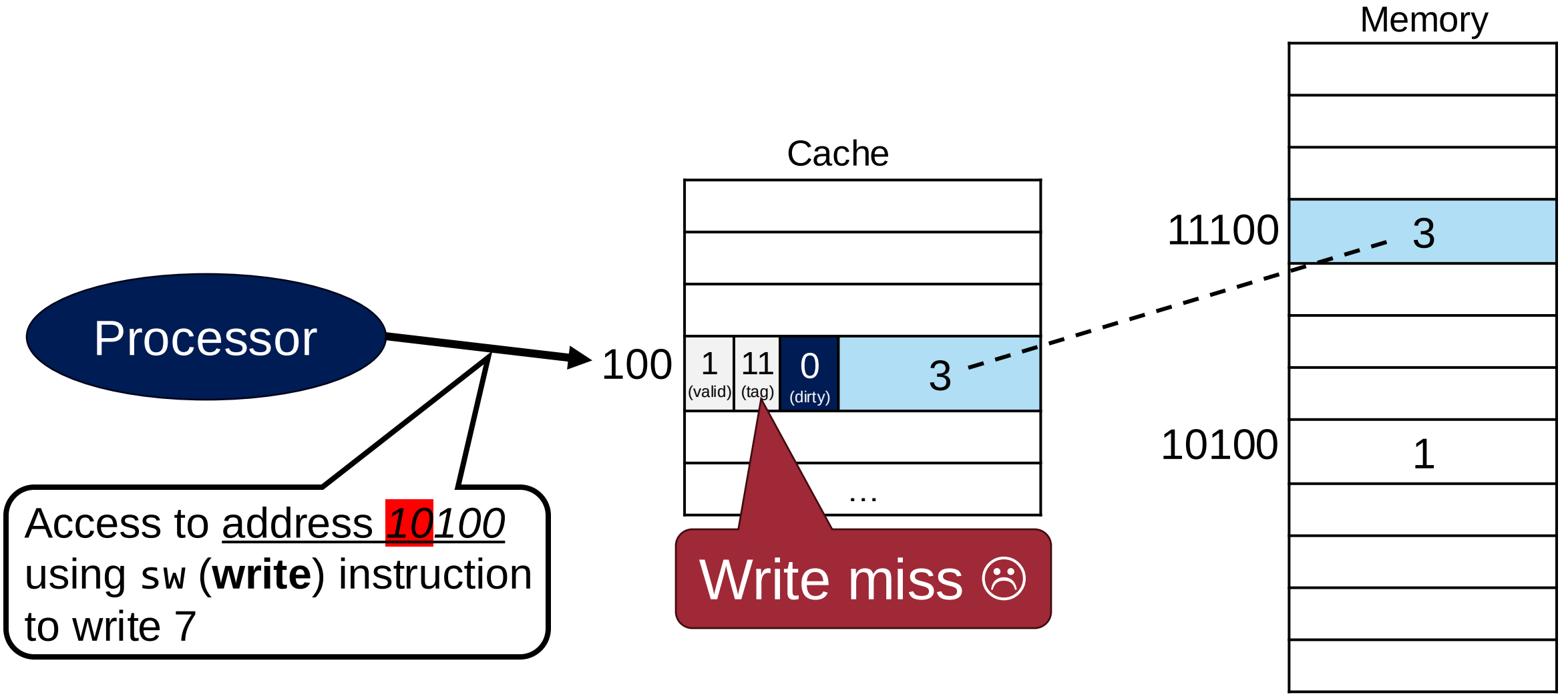
Write Misses

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

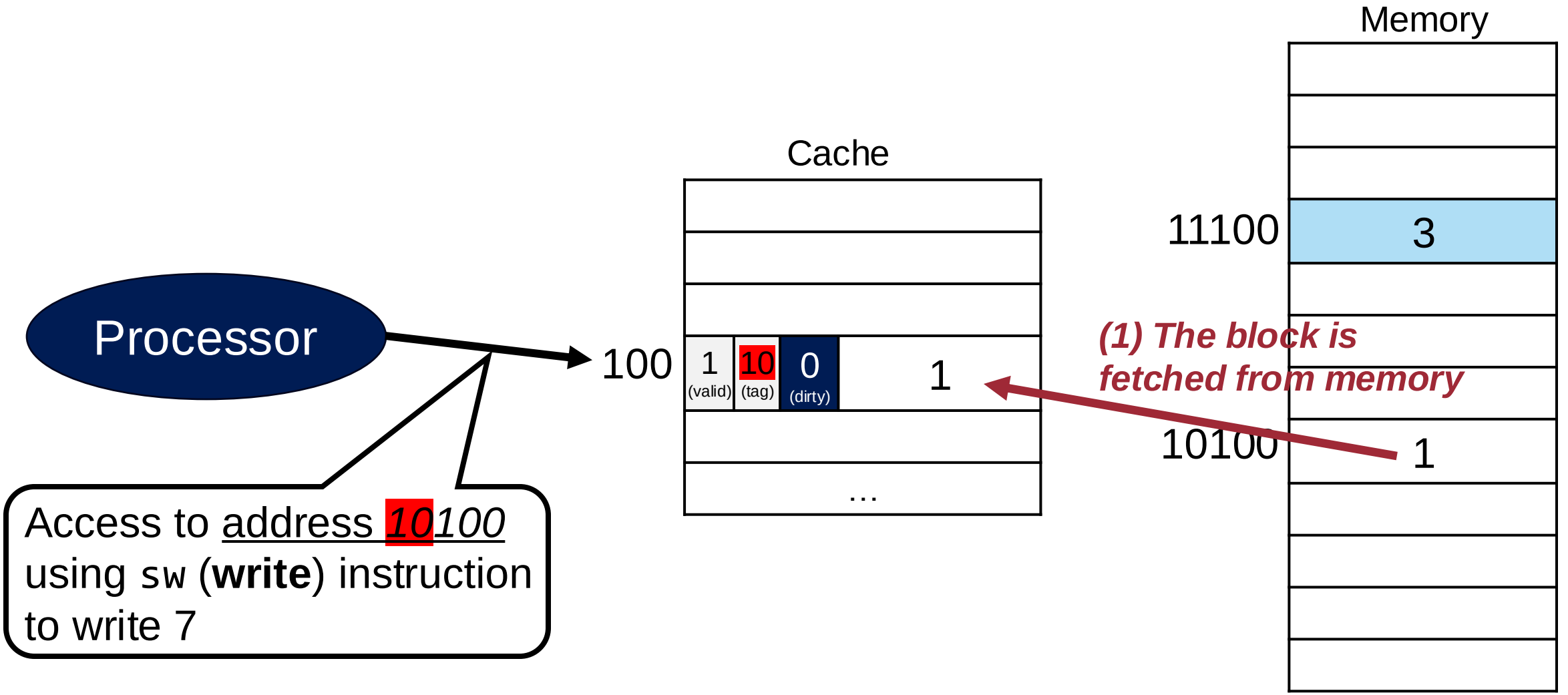
Write Misses

- 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**:

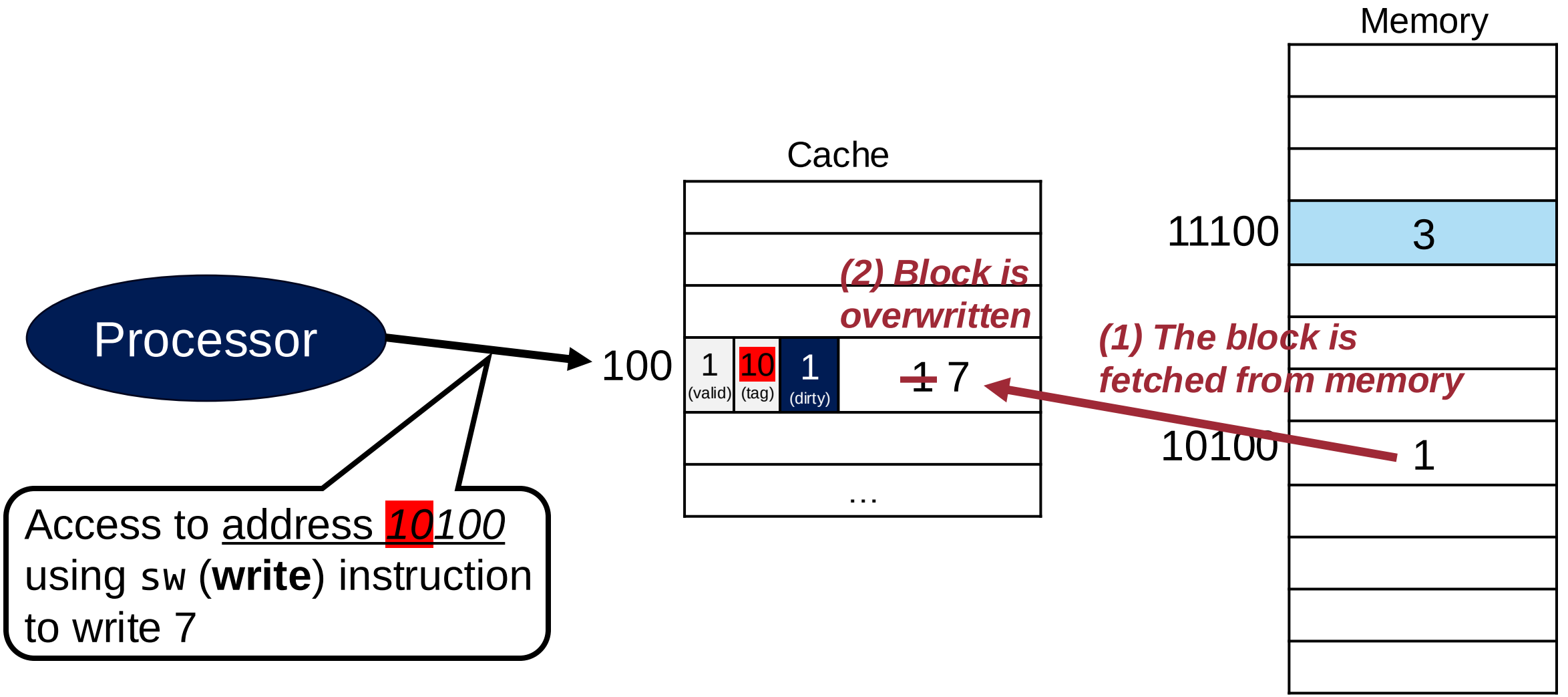
Write Misses: Write Allocate



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

Cases to consider for the instruction cache

Cases to consider for the data cache

Improving Cache Performance

How to Improve the Cache Performance?

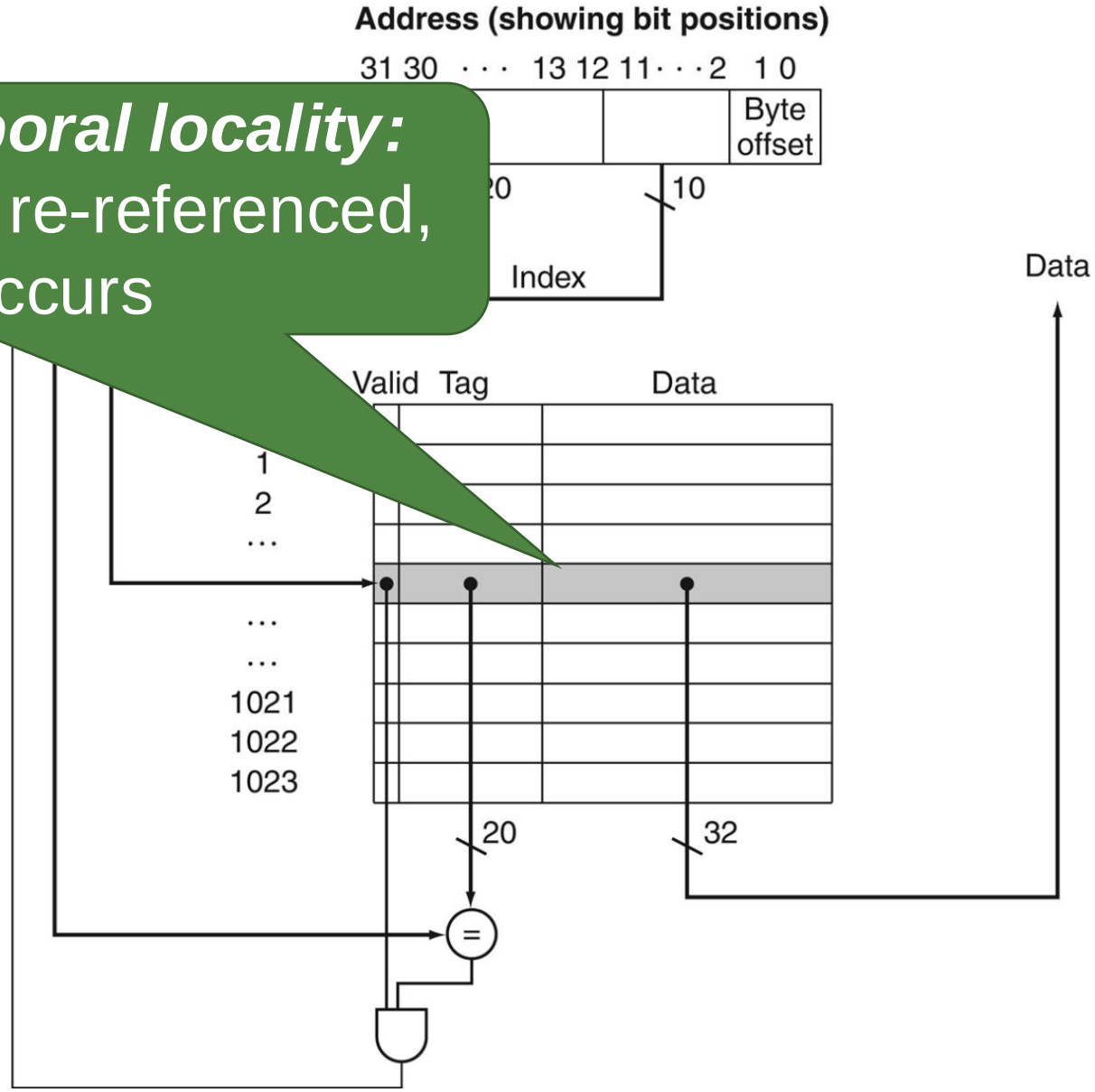
- **Adjust the block size**
- **Adjust the cache placement policy**
- **Adjust the cache replacement policy**
- **Multilevel caches**

How to Improve the Cache Performance?

- **Adjust the block size**
- Adjust the cache placement policy
- Adjust the cache replacement policy
- Multilevel caches

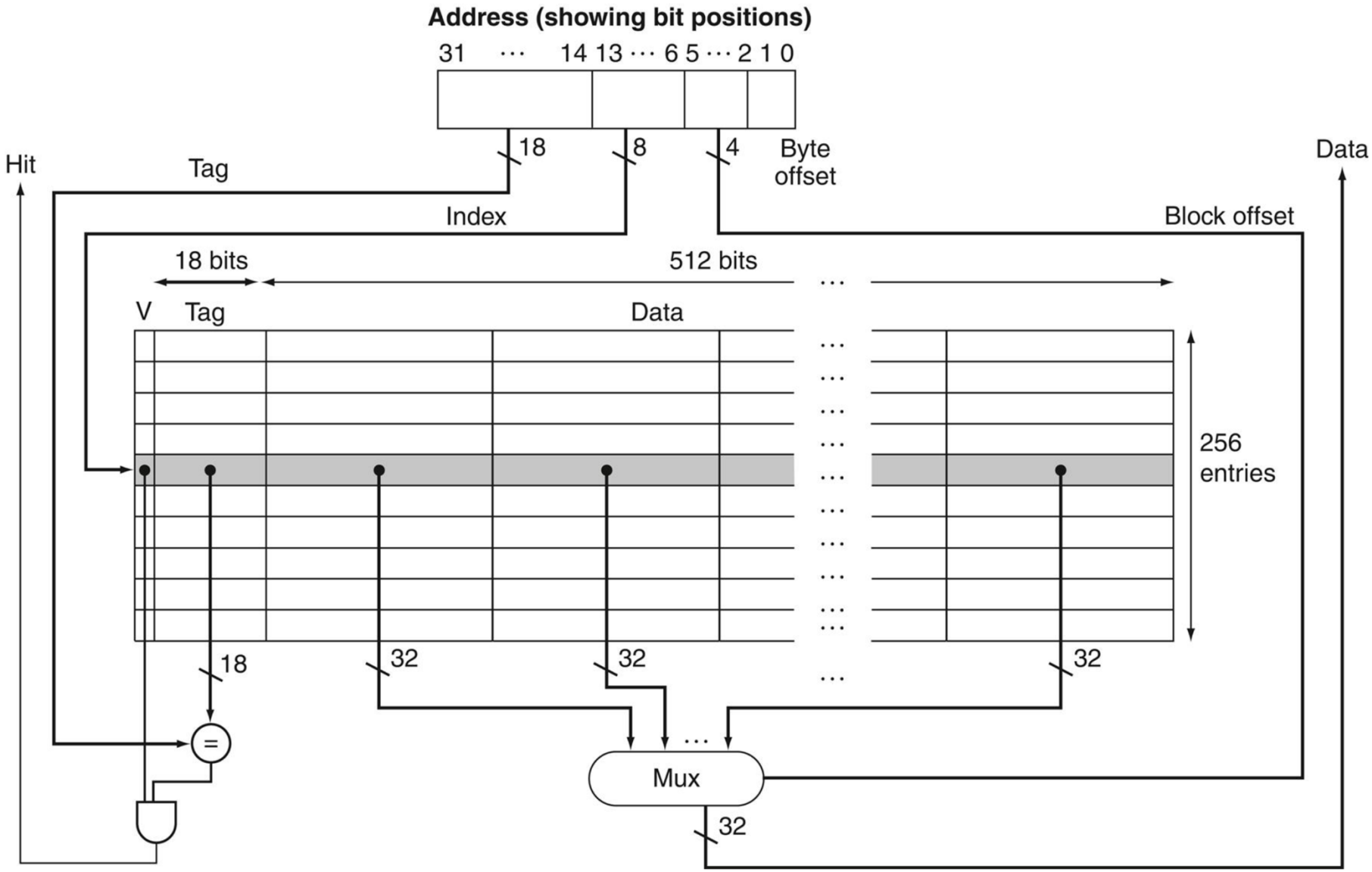
Problem with a One-word Block?

We can exploit temporal locality:
If the same address is re-referenced, a cache hit occurs

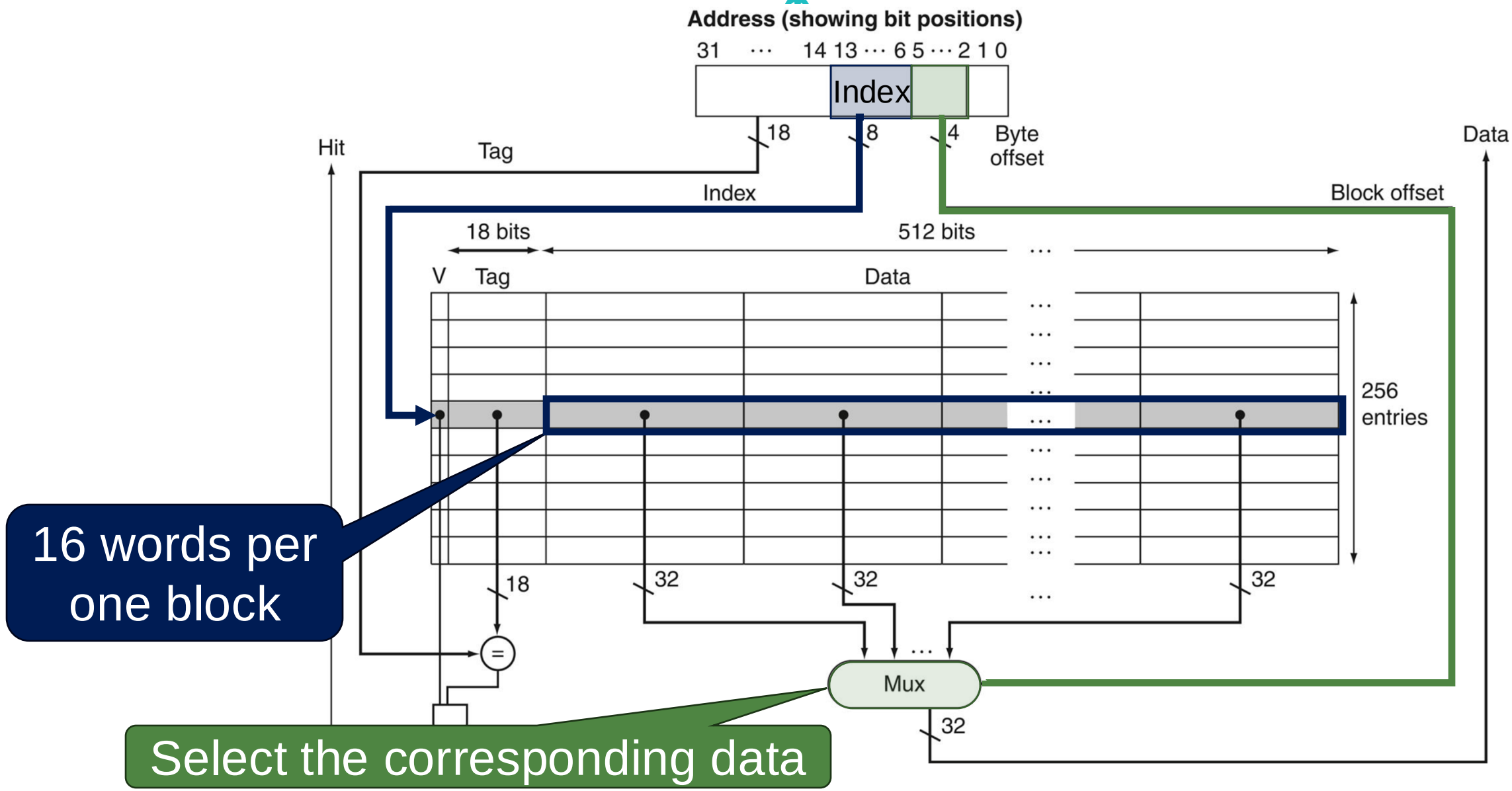


How about spatial locality?

Several Words as One Block

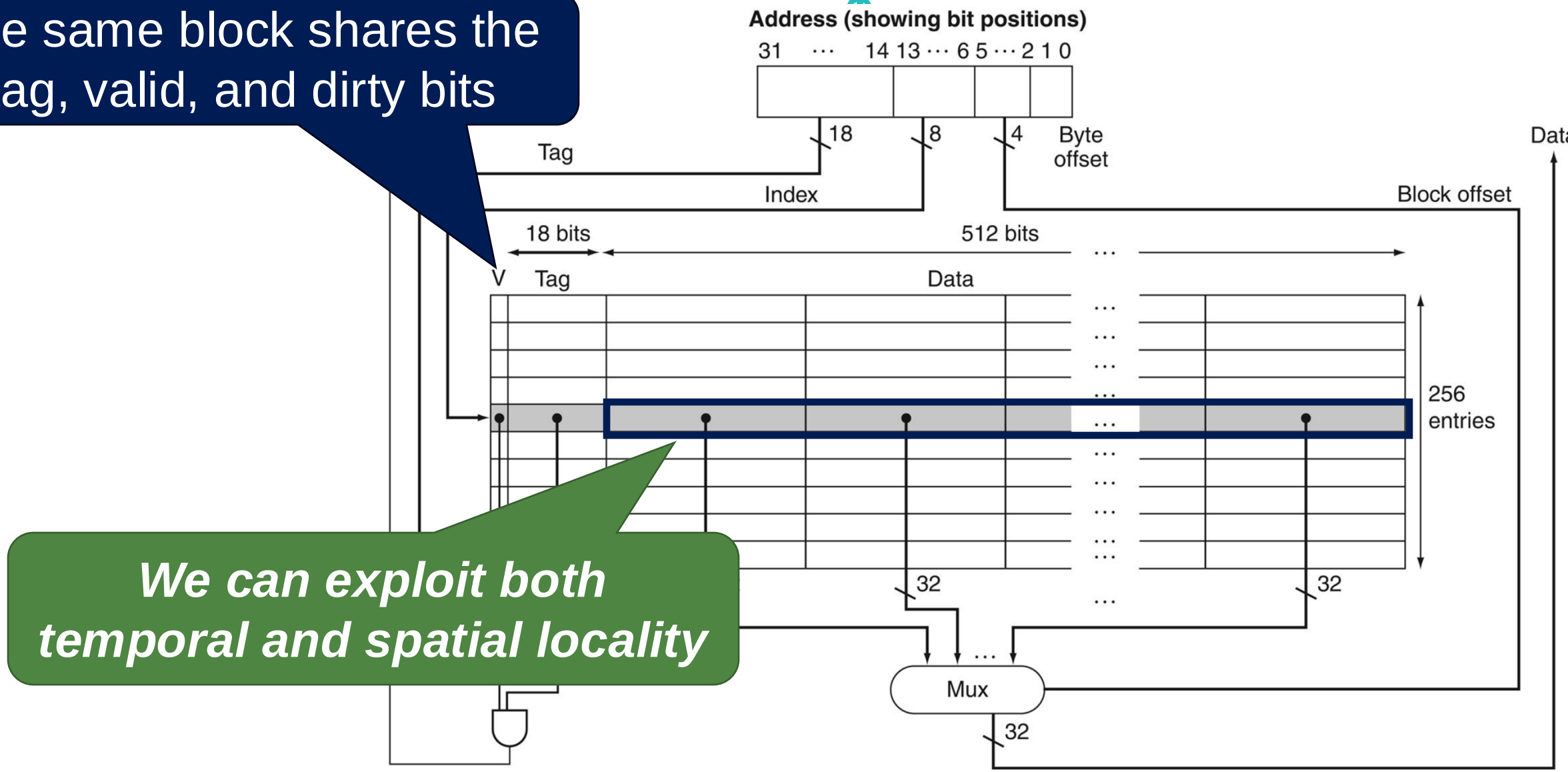


Several Words as One Block



Several Words as One Block

The same block shares the tag, valid, and dirty bits

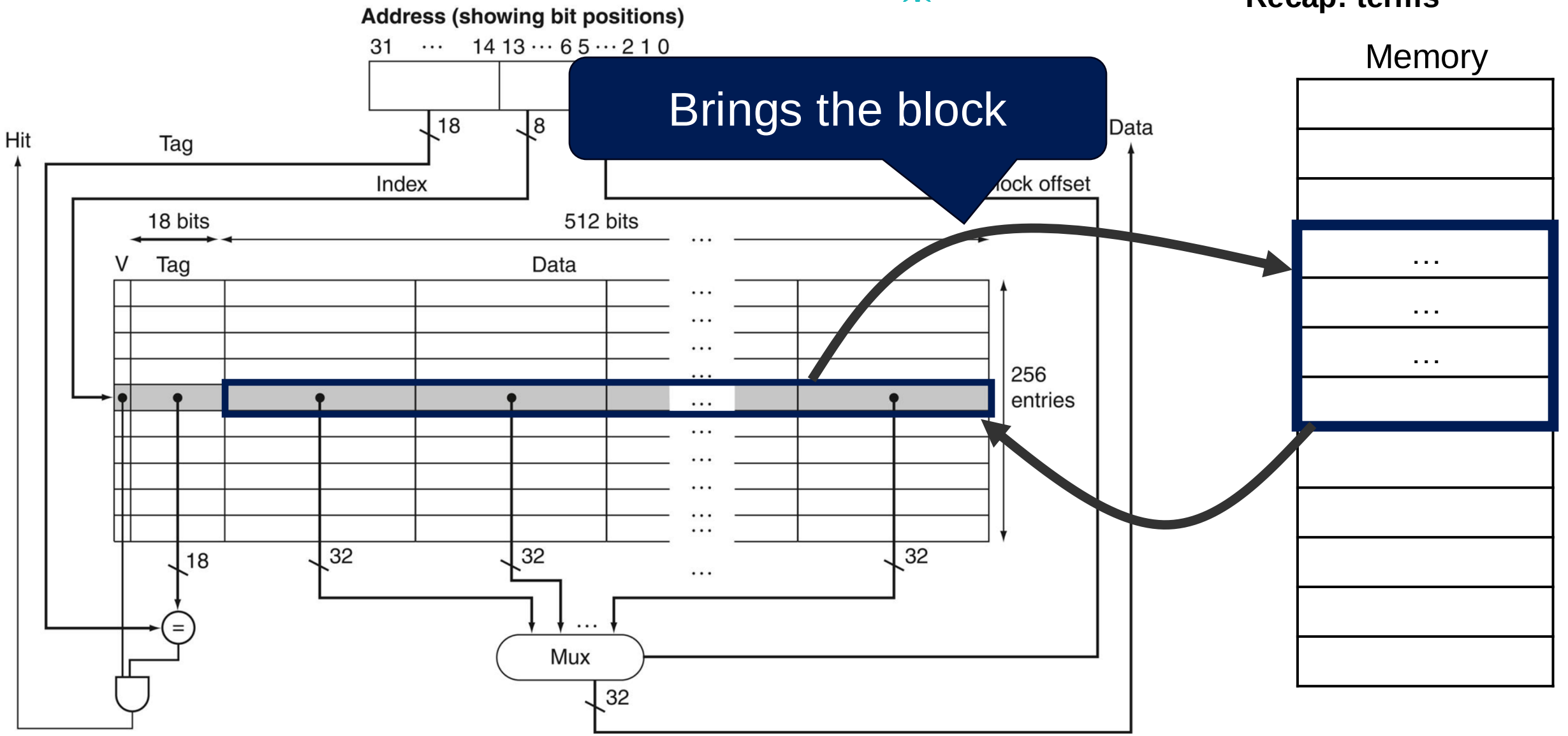


We can exploit both temporal and spatial locality

Several Words as One Block

- Block (a.k.a., line): unit of copying
  - Several words in cache memory

Recap: terms



Increasing the Block Size

- **Pros**: tends to decrease miss rate
  - Due to the spatial locality

- **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 level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

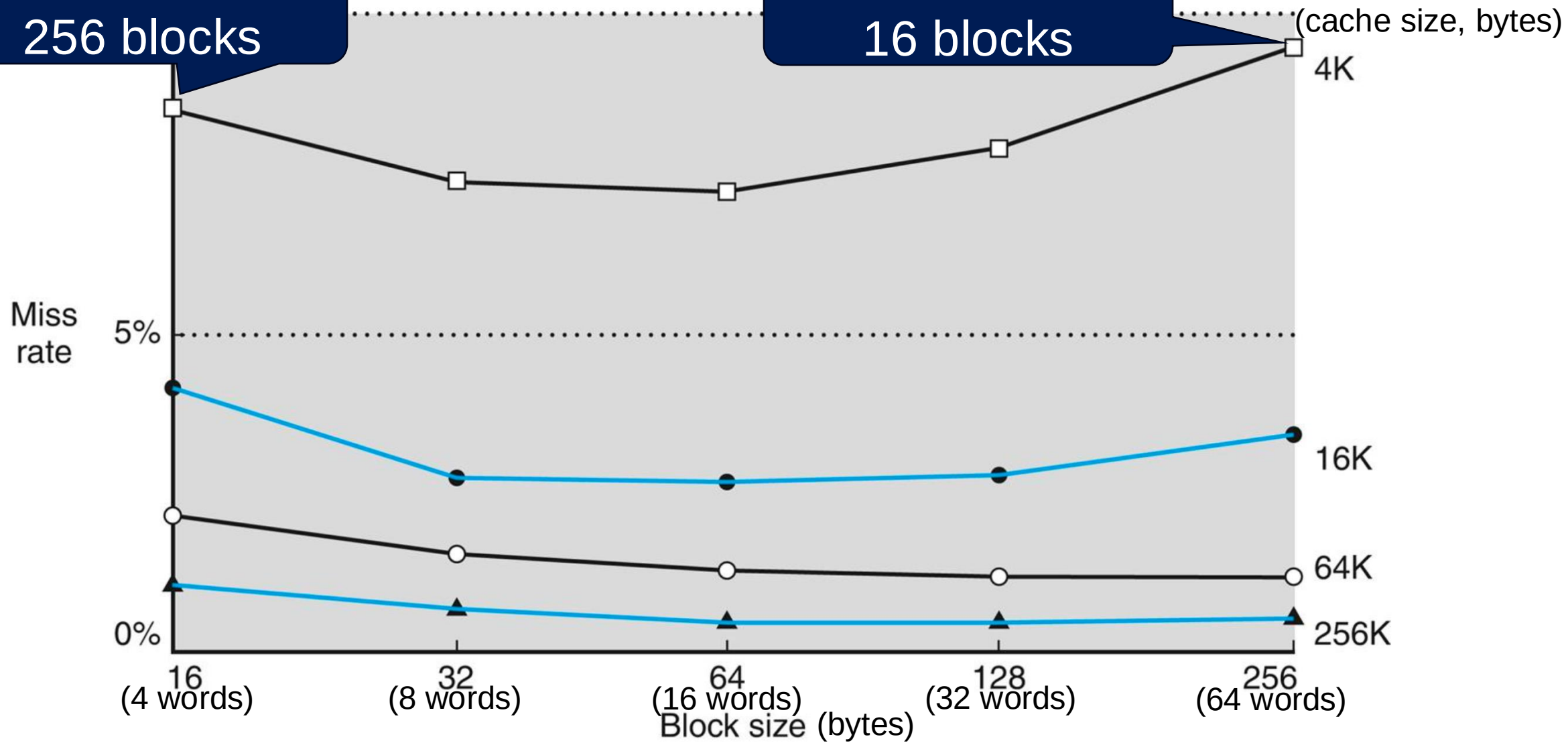
Recap: terms

- **Cons**:
  - However, if the block size is significantly increased (while keeping the total cache size fixed), the number of indexes (cache lines) decreases, which can actually **lead to an increase in the miss rate**
  - Also, **the miss penalty increases** because transferring such a large block takes more time

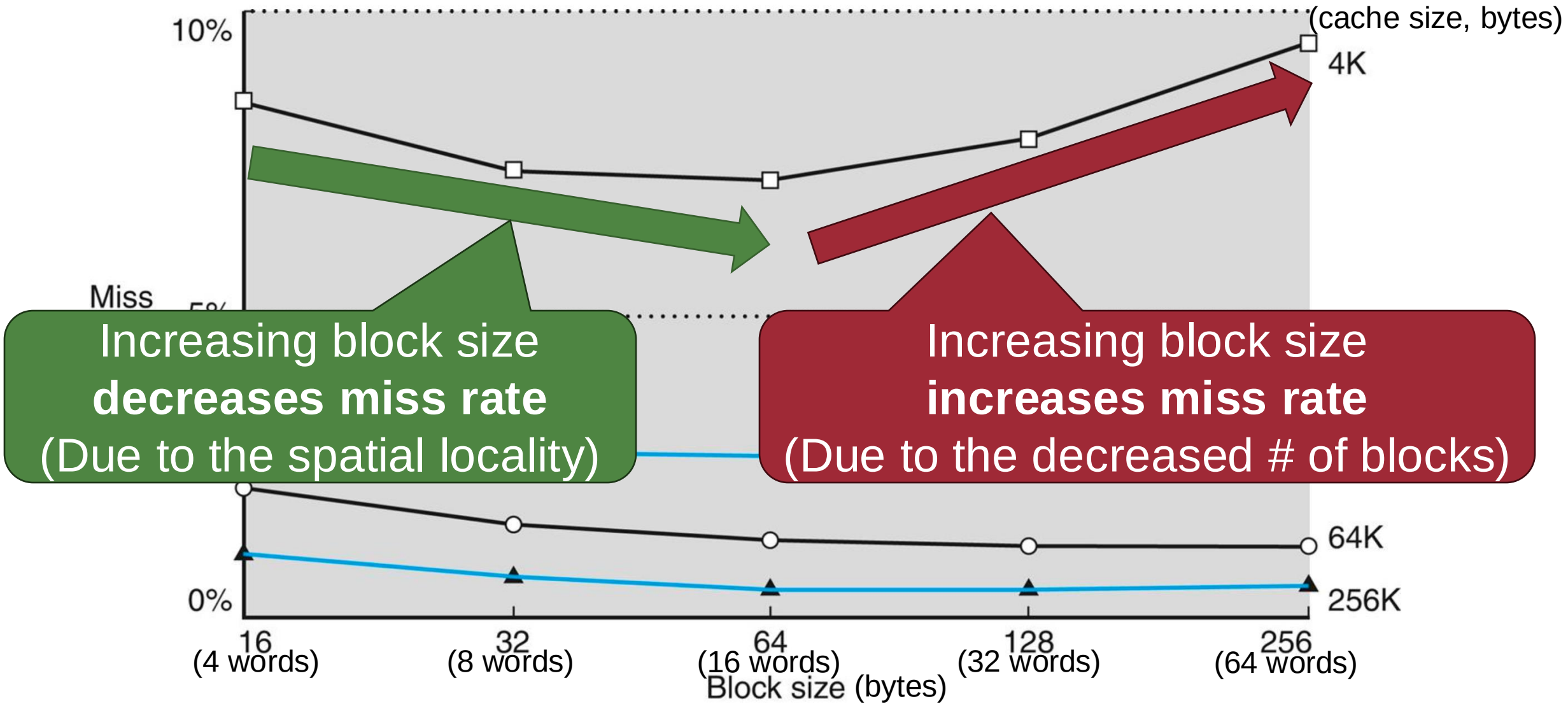
Block Size and Miss Rate

$4*2^{10}/2^4 = 2^8 =$ 256 blocks

$4*2^{10}/2^8 = 2^4 =$ 16 blocks



Block Size and Miss Rate



Cache Performance

- **Example:**
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- **Miss cycles (miss penalty) *per instruction***
  - I-cache: 0.02 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44

How to Improve the Cache Performance?

- **Adjust the block size**
- Adjust the cache placement policy
- Adjust the cache replacement policy
- Multilevel caches

How to Improve the Cache Performance?

- **Adjust the block size**
- **Adjust the cache placement policy**
- **Adjust the cache replacement policy**
- **Multilevel caches**

Cache Placement Policy

Cache placement policy: where can a block be placed?

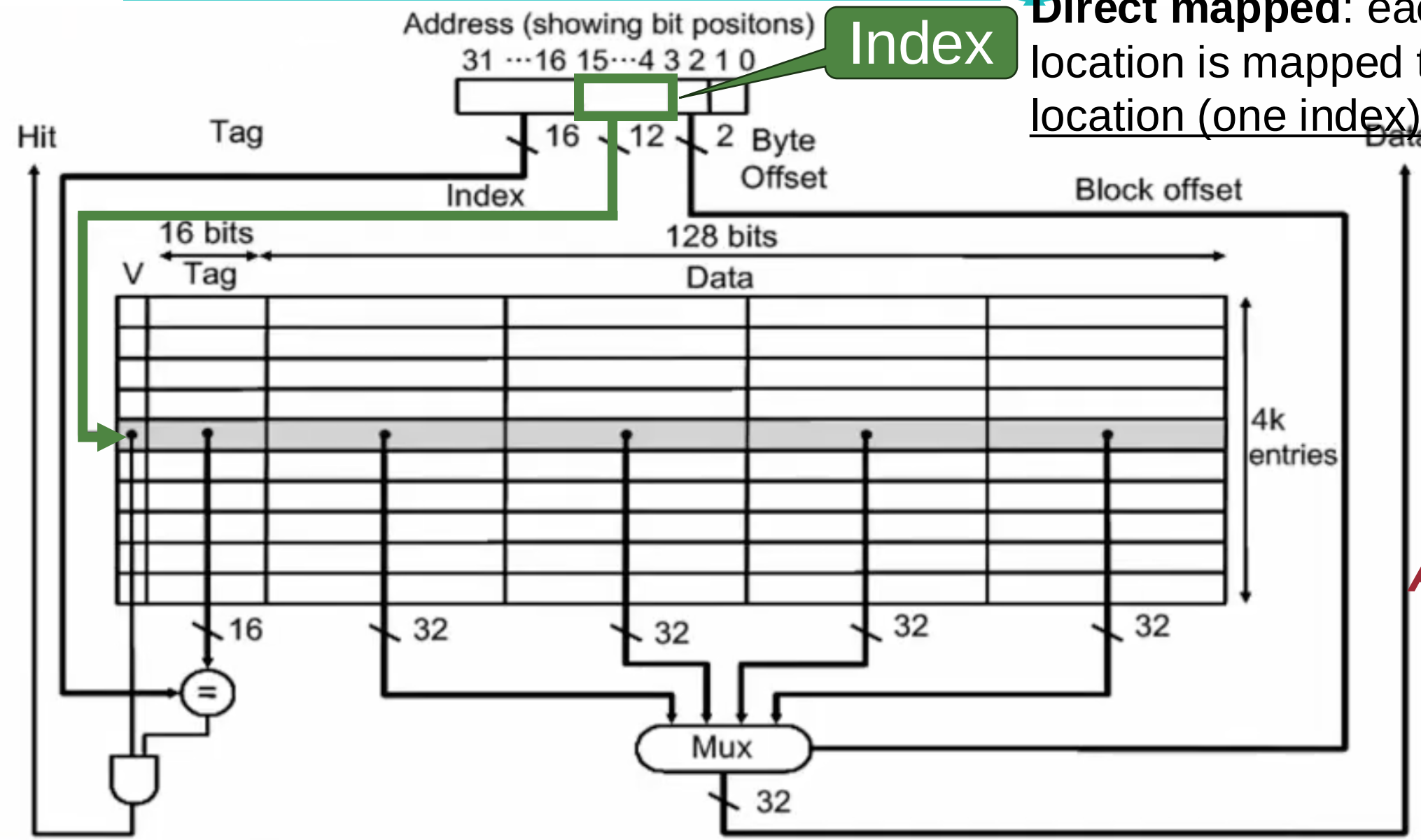
- Direct mapped
- Fully associative
- Set associative

Cache Placement Policy: Direct Mapped

- Address is modulo the number of blocks in the cache

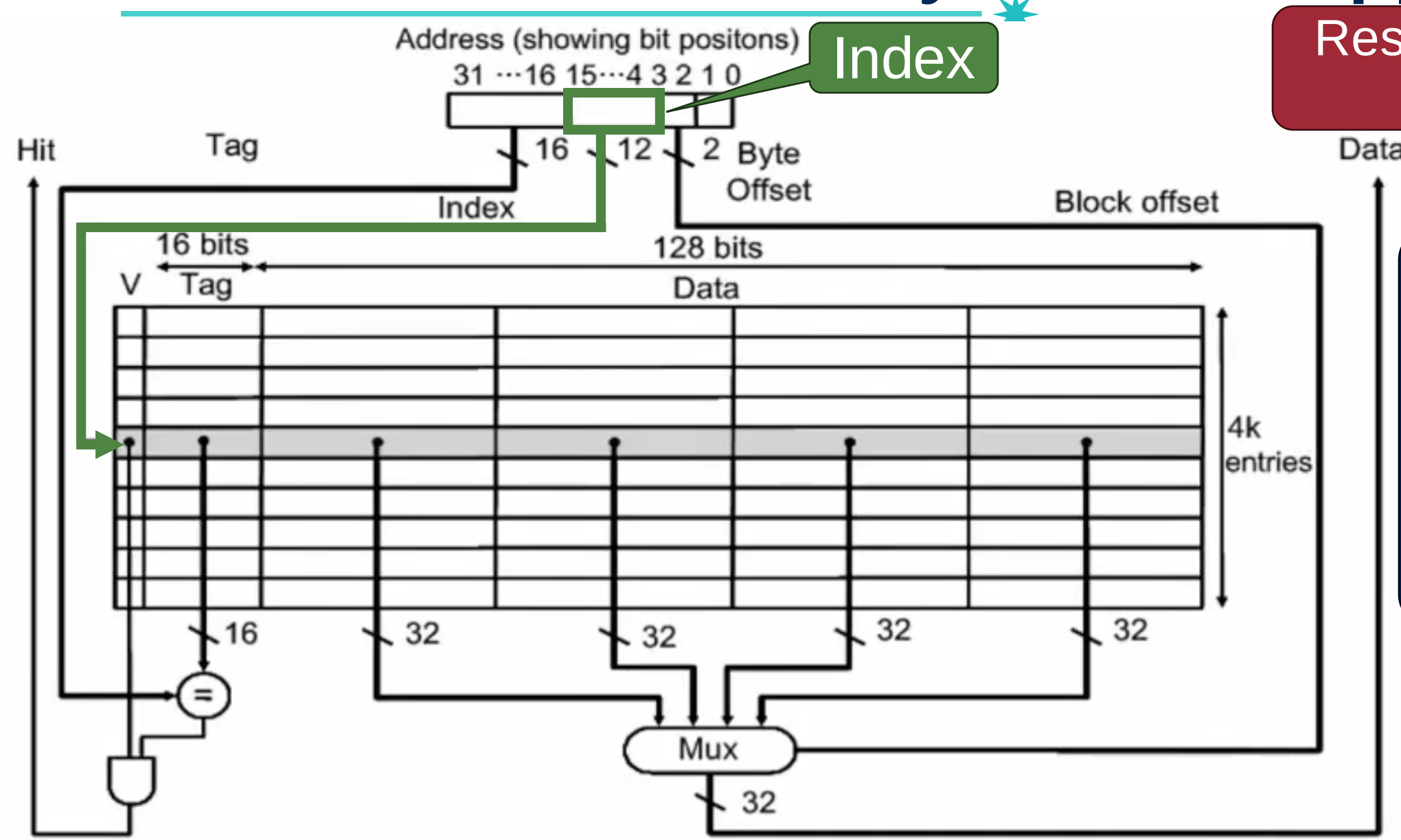
Cache Placement Policy: Direct Mapped

Direct mapped: each *memory* location is mapped to <u>exactly one location (one index)</u> in the *cache*



Any problems?

Cache Placement Policy: Direct Mapped



Results in lots of misses!

```
0x00011010
0x00101010
0x00011010
0x00101010
0x00011010
...
```

Memory access sequence

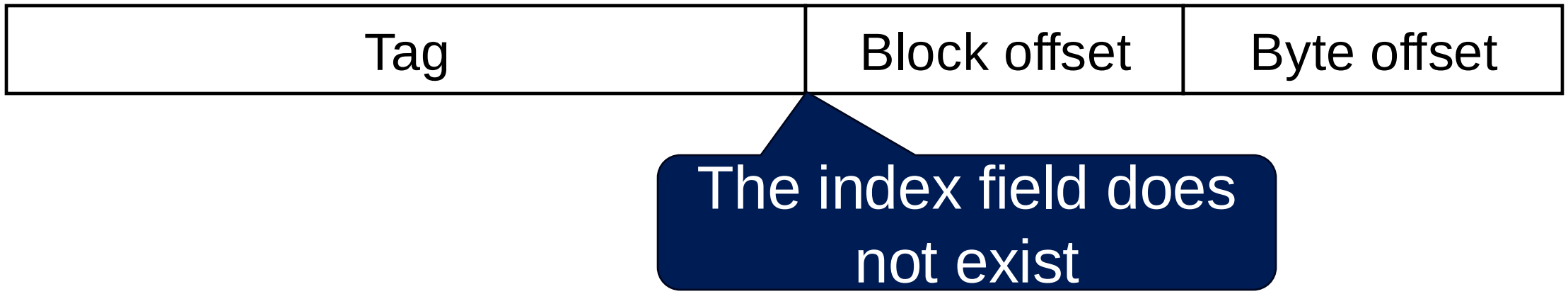
Direct Mapped: Summary

- Address is modulo the number of blocks in the cache

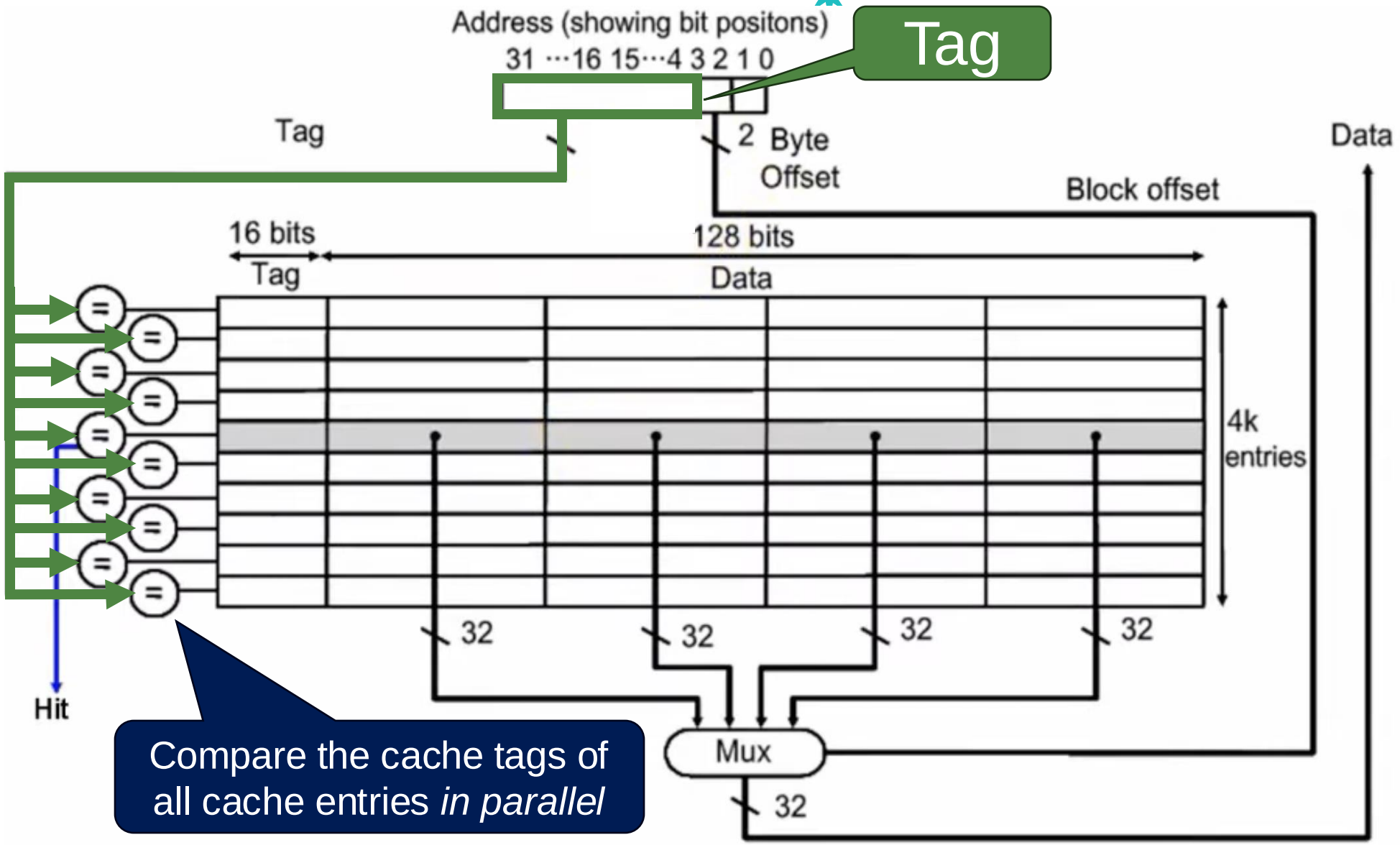
- **Disadvantage**: imagine two frequently-accessed variables. What if their addresses have the same index bits?
  - Such addresses are in conflict in the cache
  - Can't hold both in the cache at once
  - Result in lots of misses (bad!)

Cache Placement Policy: Fully Associative

- No cache index
- Block can be placed **in any location in the cache**
- Requires all entries to be searched at once (compare the cache tags of all cache entries in parallel)

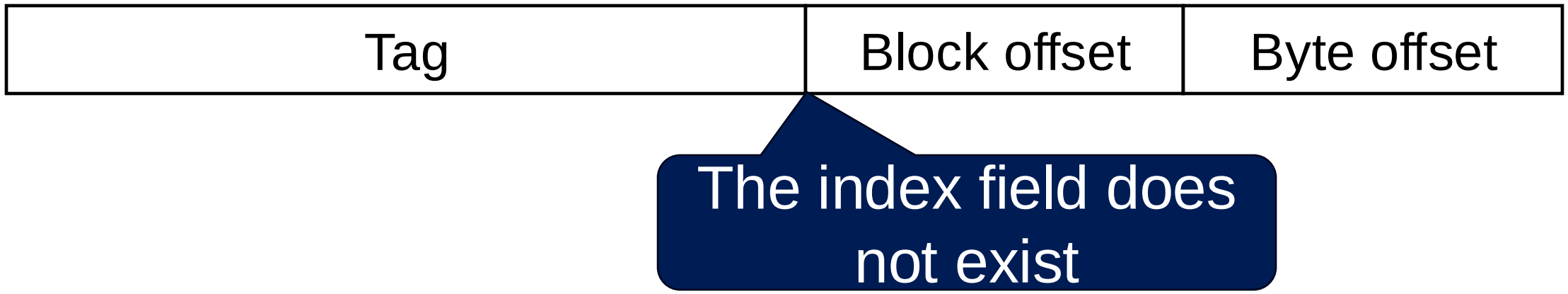


Cache Placement Policy: Fully Associative



Fully Associative: Summary

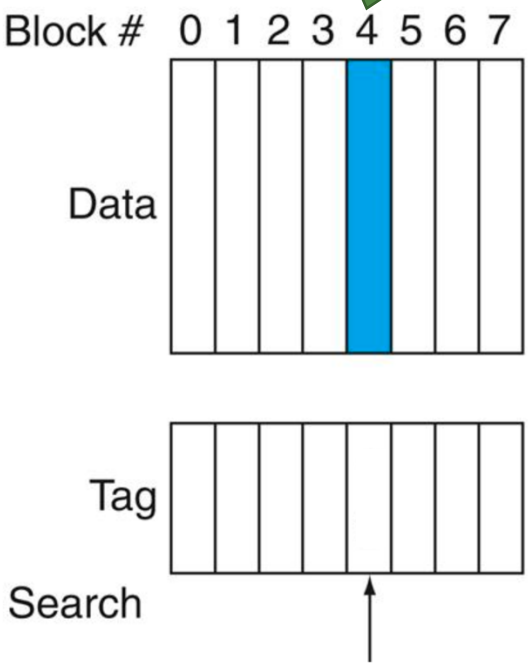
- No cache index
- Block can be placed **in any location in the cache**
- Requires all entries to be searched at once (compare the cache tags of all cache entries in parallel)



- **Disadvantage**: Comparator per entry (expensive hardware)

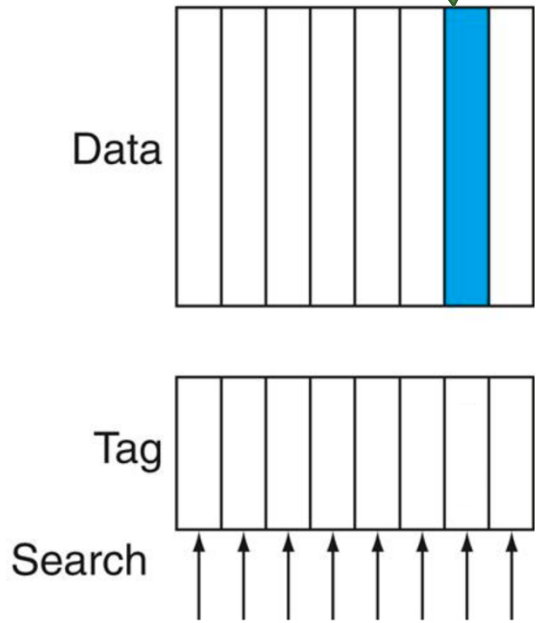
Associativity – Access to Address 12

12 mod 8 = 4



Direct mapped

Any location



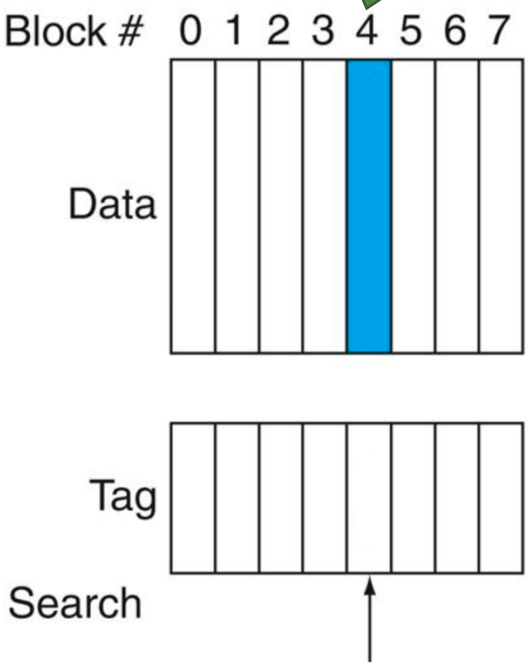
Fully associative

Increasing associativity (shrink index & expand tag)

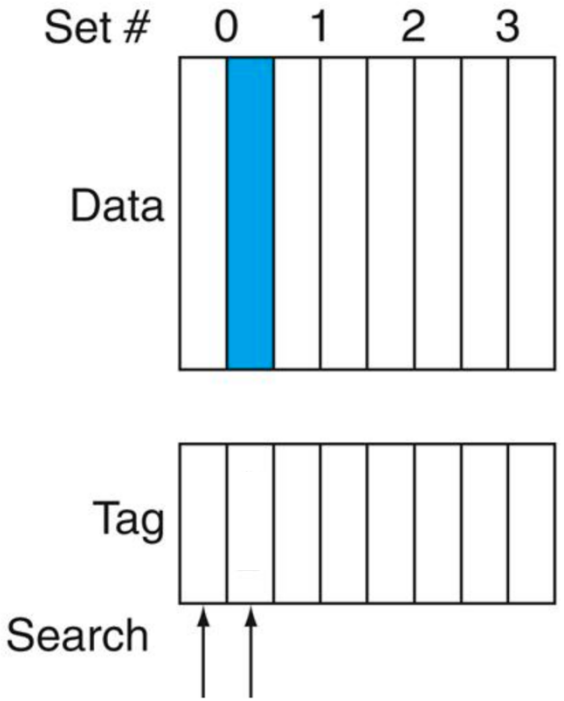
Reducing miss rate by more flexible placement

Introduction to Set-associative Cache

12 mod 8 = 4

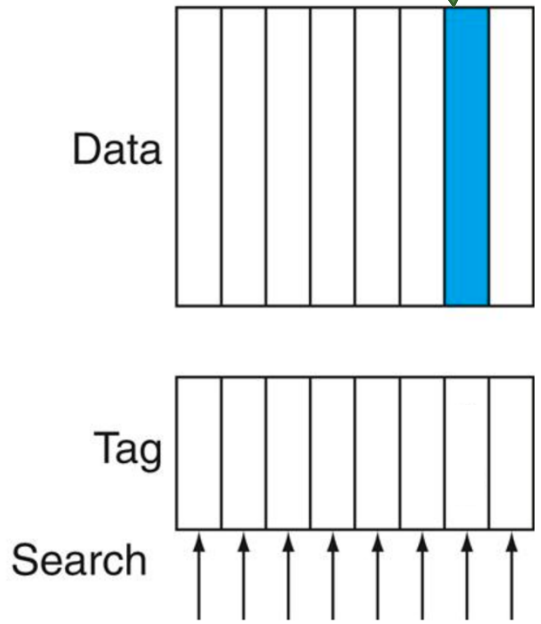


Direct mapped



Set-associative

Any location



Fully associative

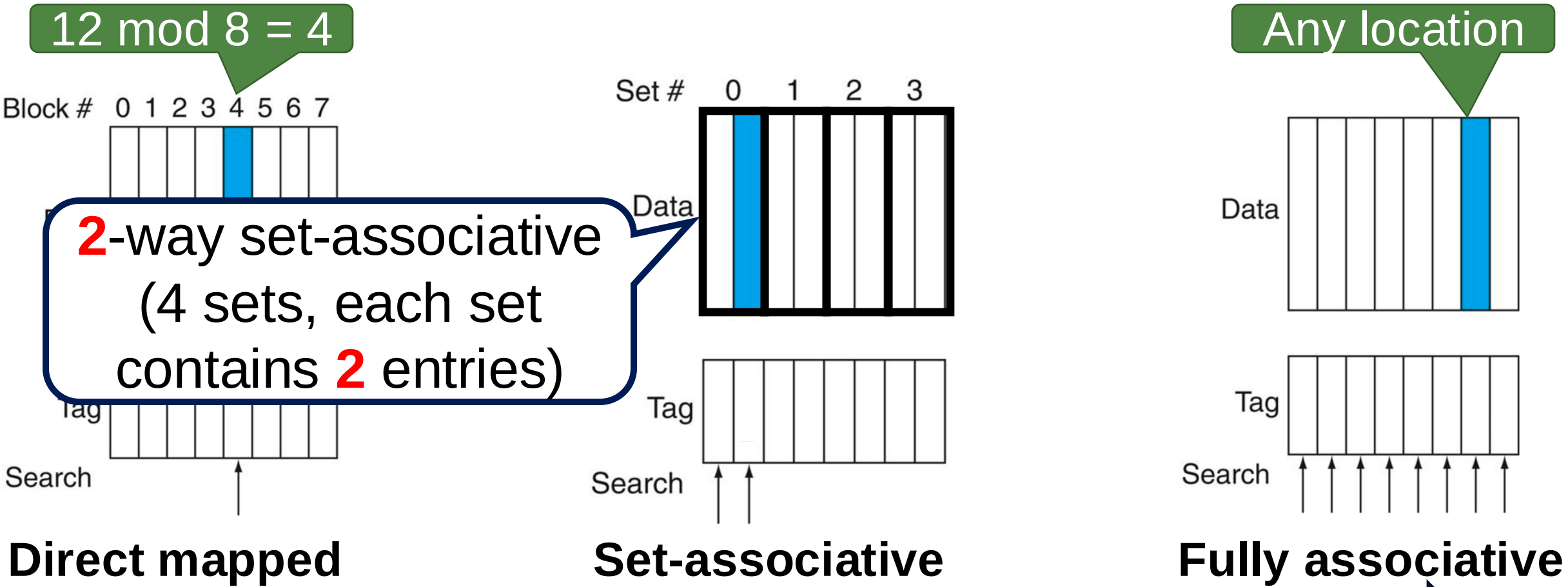
Miss rate ↑

easing associativity (shrink index & expand tag

Reducing miss rate by more flexible placement

HW complexity ↑

Introduction to Set-associative Cache



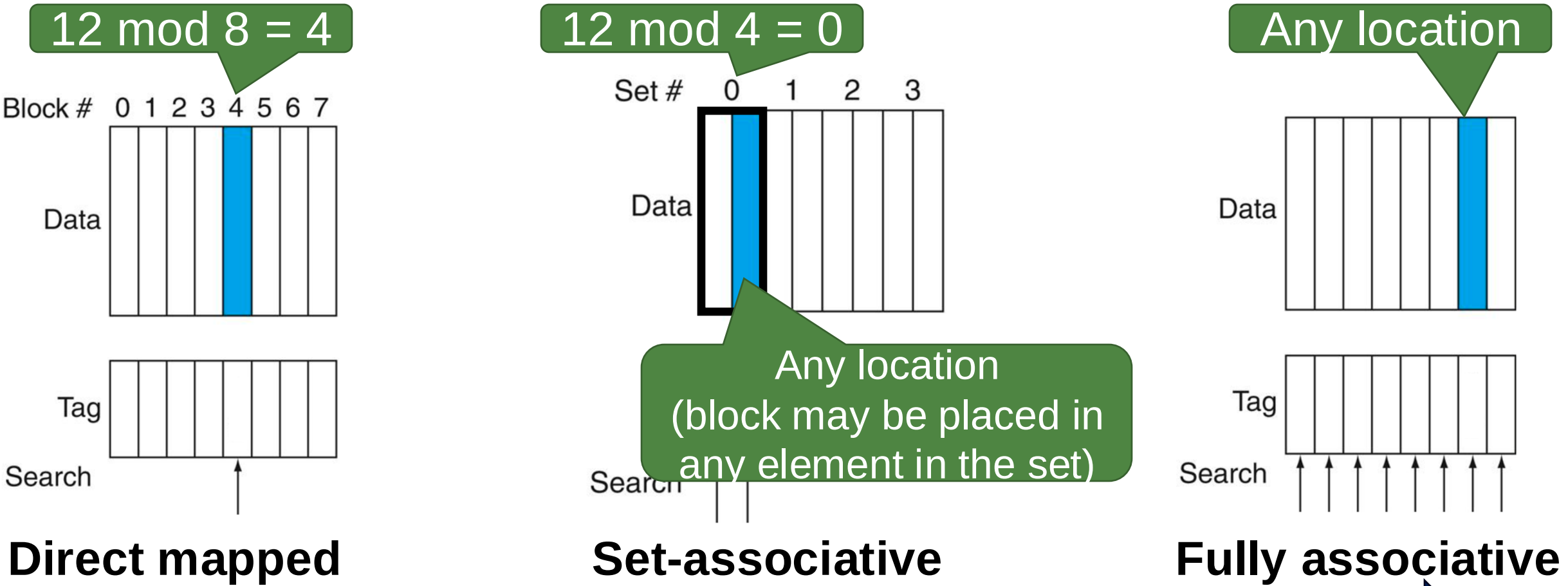
Miss rate ↑

easing associativity (shrink index & expand tag

Reducing miss rate by more flexible placement

HW complexity ↑

Introduction to Set-associative Cache



Miss rate ↑

easing associativity (shrink index & expand tag

Reducing miss rate by more flexible placement

HW complexity ↑

Cache Placement Policy: *N*-way Set Associative

- Allow a given block to go in a target set
  - Position of memory block = (block #) mod (# of sets in the cache)
- Each set contains *N* entries
- Search all entries *in a given set* at once
  - *N* comparators (less expensive)

Spectrum of Associative

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

2 sets

4 way

Four-way set associative

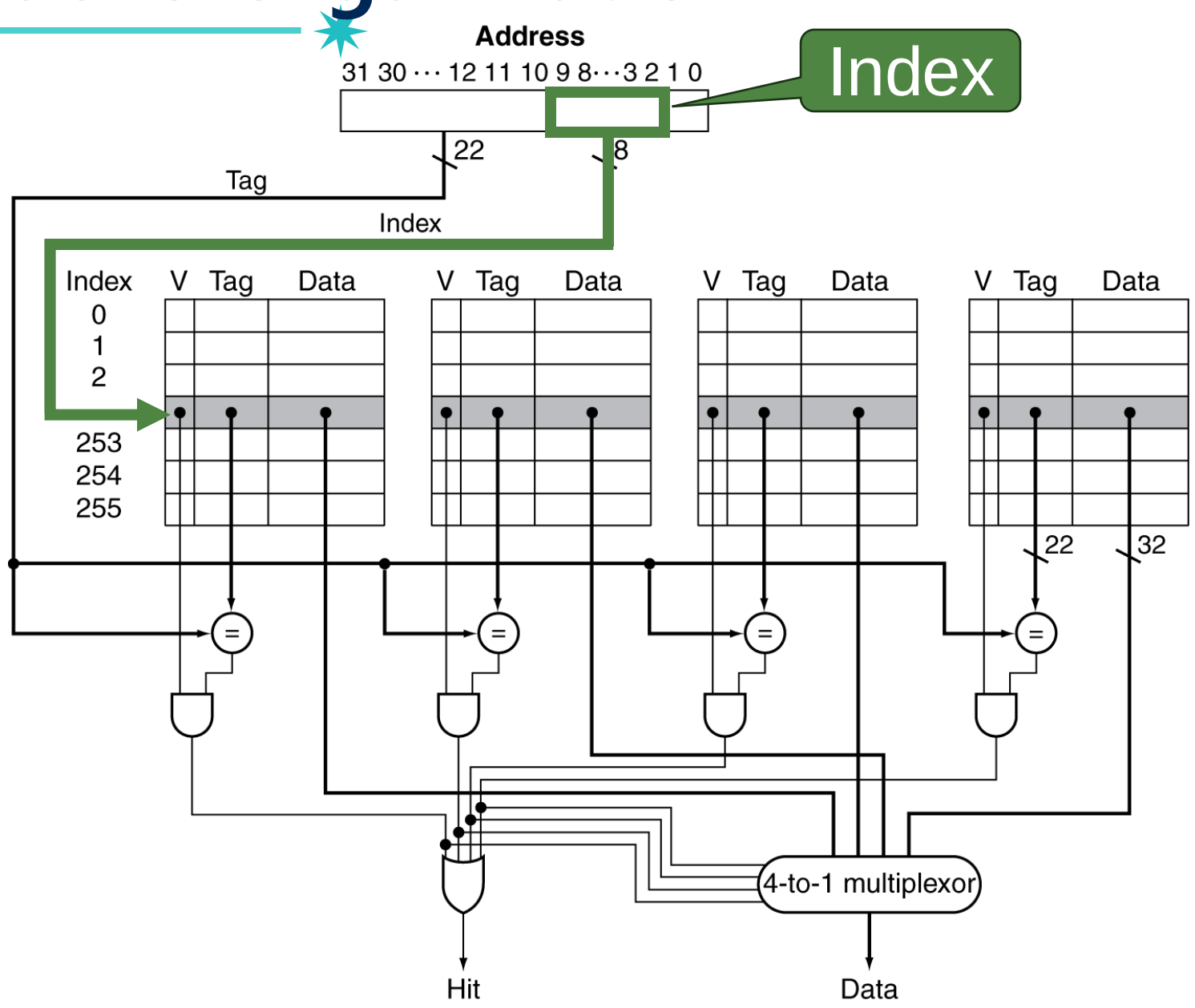
Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

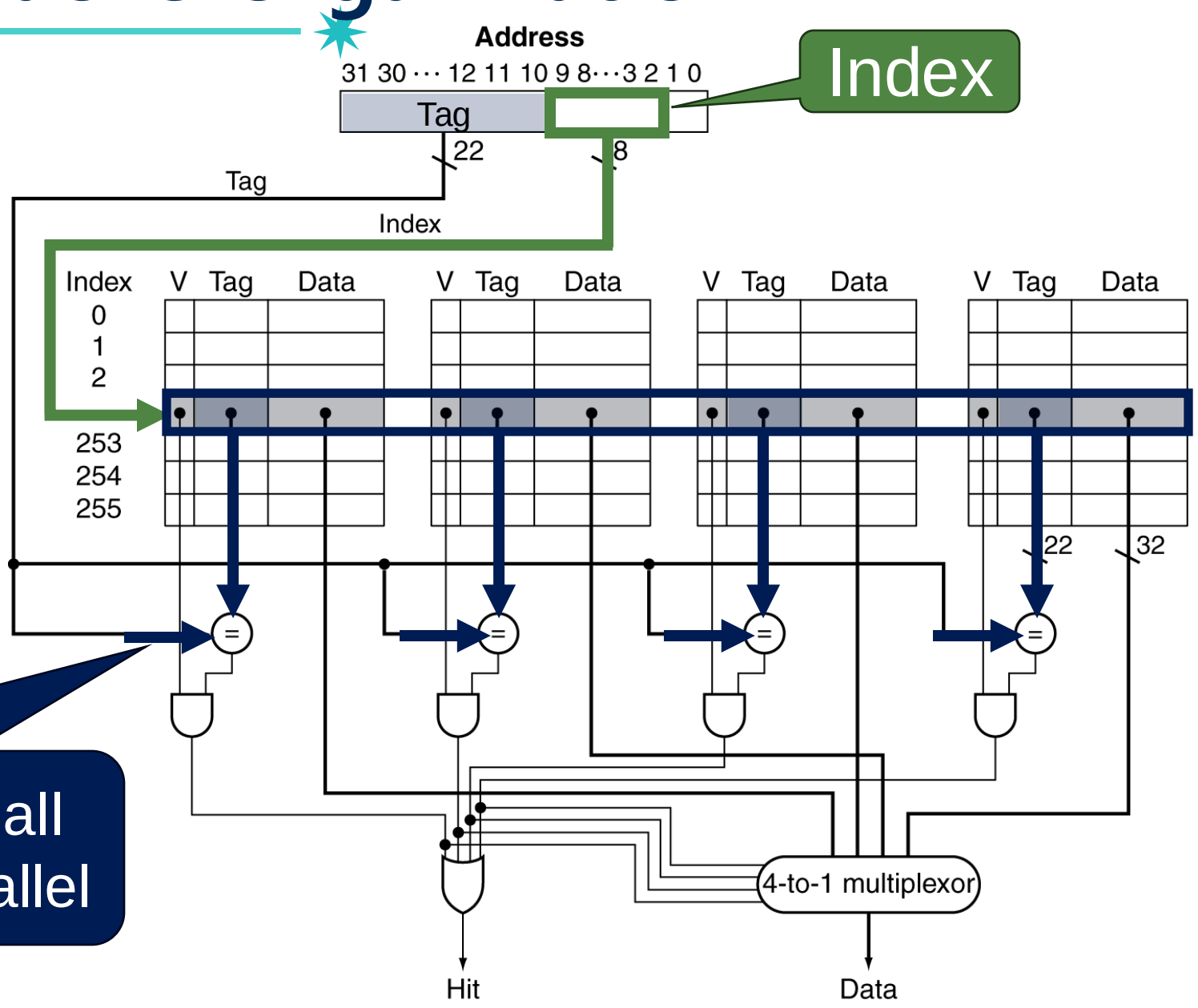
Set Associative Cache Organization

- 1024 blocks
- 4-way set associativity
- 1 block = 1 word



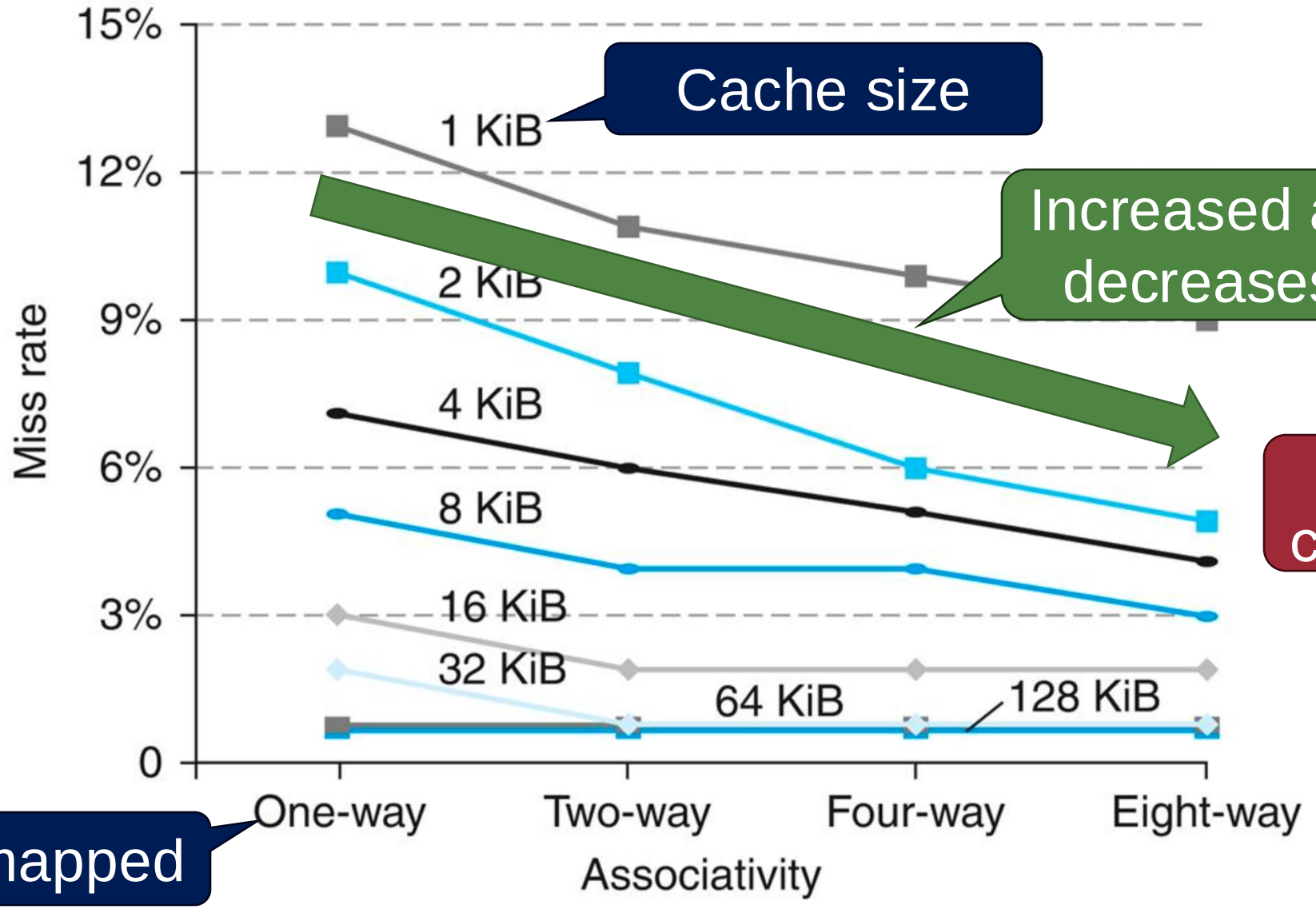
Set Associative Cache Organization

- 1024 blocks
- 4-way set associativity
- 1 block = 1 word



Compare the cache tags of all entries **in a given set** in parallel

Associativity and Size



Cache size

Increased associativity decreases miss rate

HW complexity ↑

Direct mapped

Summary: Cache Placement Policy

Cache placement policy: where can a block be placed?

- Direct mapped
- Fully associative
- Set associative

How to Improve the Cache Performance?

- **Adjust the block size**
- **Adjust the cache placement policy**
- **Adjust the cache replacement policy**
- **Multilevel caches**

How to Improve the Cache Performance?

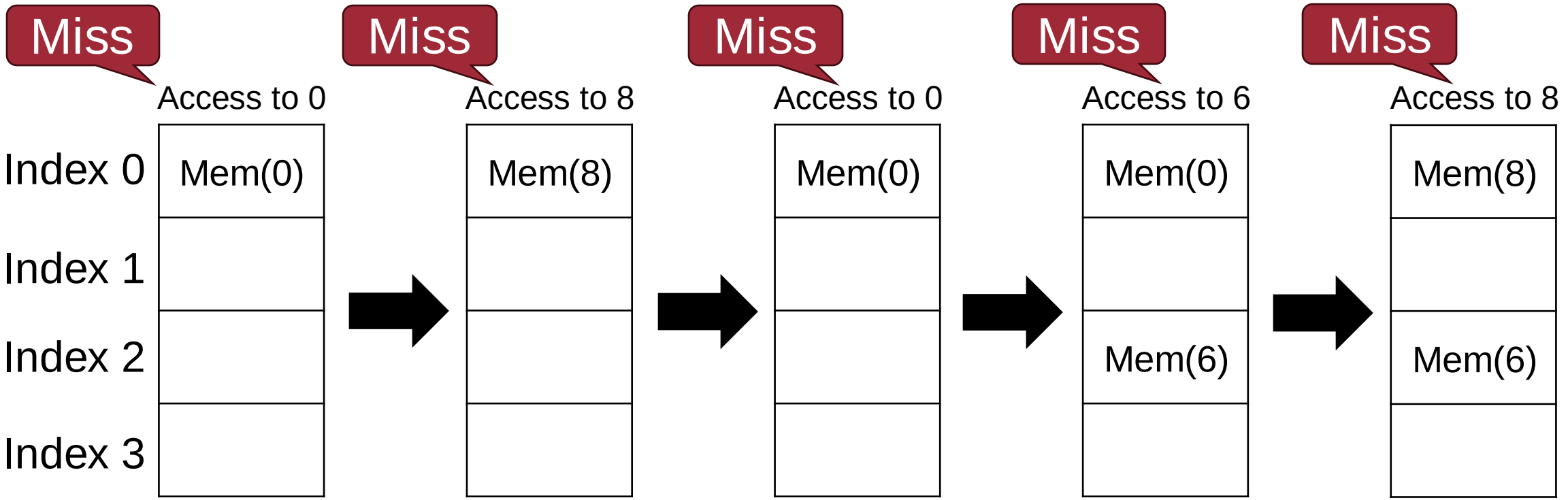
- **Adjust the block size**
- **Adjust the cache placement policy**
- **Adjust the cache replacement policy**
- **Multilevel caches**

Motivation Example: Cache Replacement Policy

- Block access sequence: 0, 8, 0, 6, 8
- 4 Blocks
- Block size: 4 bytes (1 word)

Motivation Example: Cache Replacement Policy

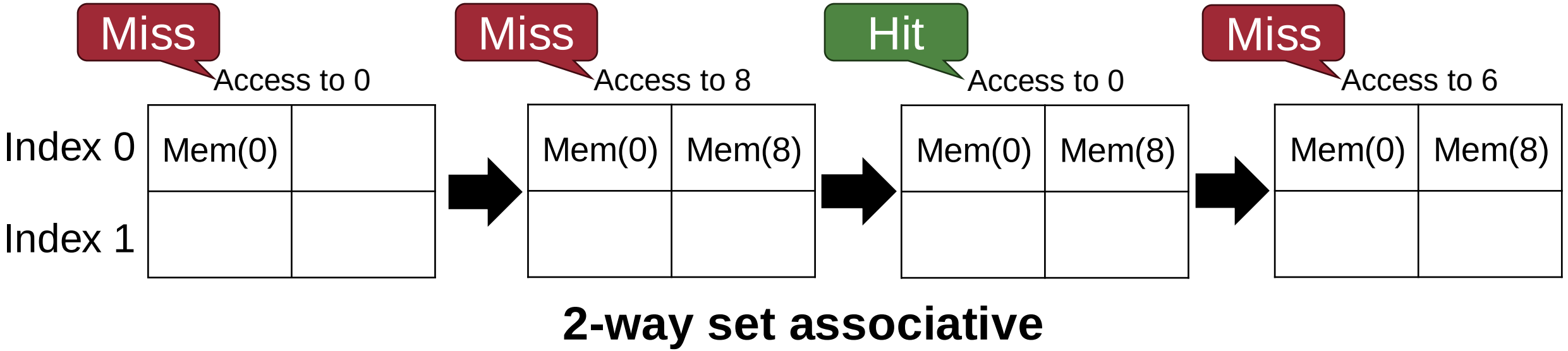
- Block access sequence: 0, 8, 0, 6, 8
- 4 Blocks
- Block size: 4 bytes (1 word)



Direct mapped

Motivation Example: Cache Replacement Policy

- Block access sequence: 0, 8, 0, 6, 8
- 4 Blocks
- Block size: 4 bytes (1 word)

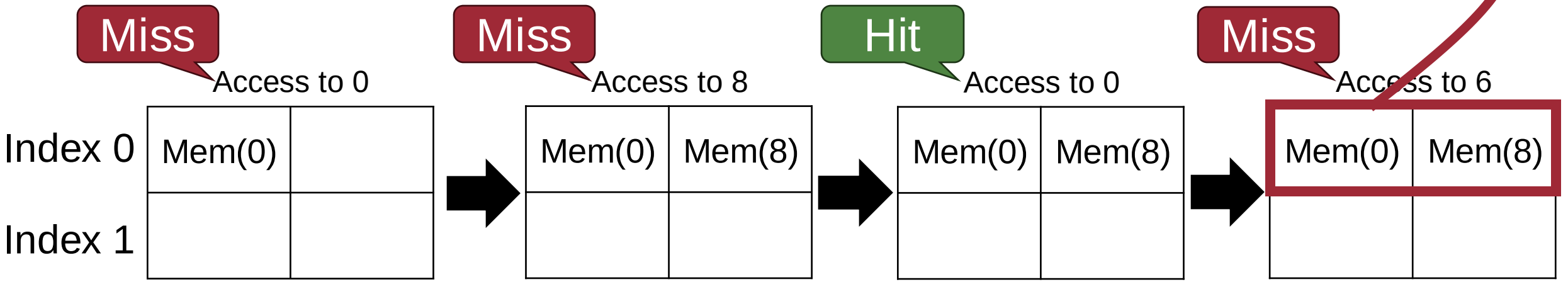


2-way set associative

Motivation Example: Cache Replacement Policy

- Block access sequence: 0, 8, 0, 6, 8
- 4 Blocks
- Block size: 4 bytes (1 word)

Which block should be replaced?



2-way set associative

Which Block Should be Replaced on a Cache Miss?

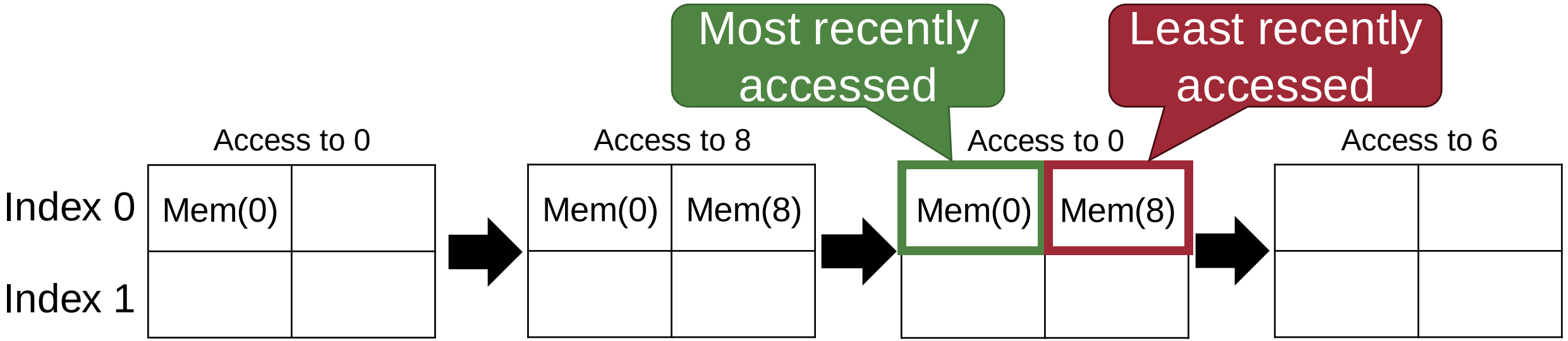
- Direct mapped: no choice
- Set associative & fully associative: follow the **replacement policy**

Replacement Policy

- **Least Recently Used (LRU)**: replace the one NOT used (accessed) for the longest time

Replacement Policy

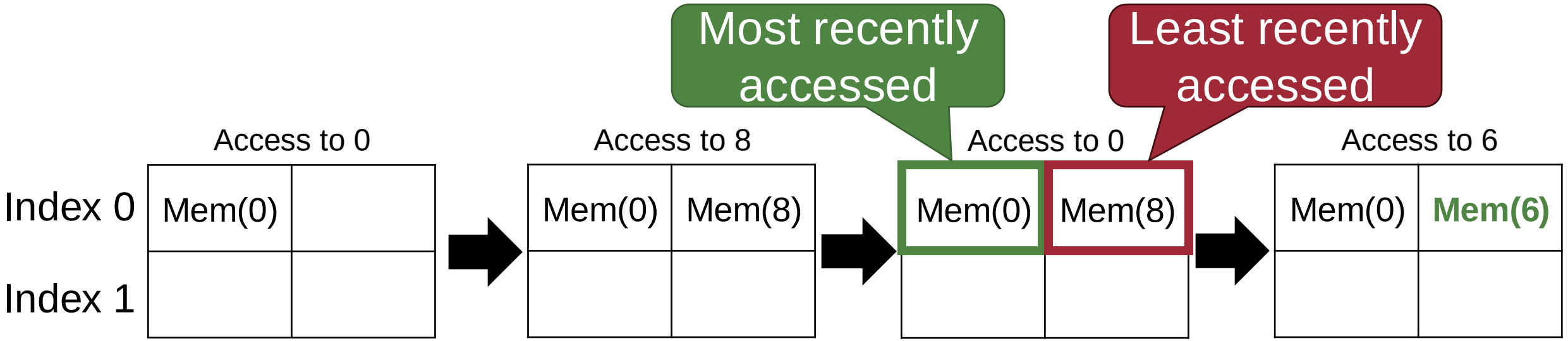
- **Least Recently Used (LRU)**: replace the one NOT used (accessed) for the longest time
  - Temporal locality of access is considered
  - Need a reference history information



2-way set associative

Replacement Policy

- **Least Recently Used (LRU)**: replace the one NOT used (accessed) for the longest time
  - Temporal locality of access is considered
  - Need a reference history information



2-way set associative

Replacement Policy

- **Least Recently Used (LRU)**: replace the one NOT used (accessed) for the longest time
  - Temporal locality of access is considered
  - Need a reference history information

- **Random**
  - Gives approximately the same performance as LRU for high associativity

+ **First In First Out (FIFO)**: replace the block with long-lived in the cache

+ **Least Frequently Used (LFU)**: replace the block with fewest reference

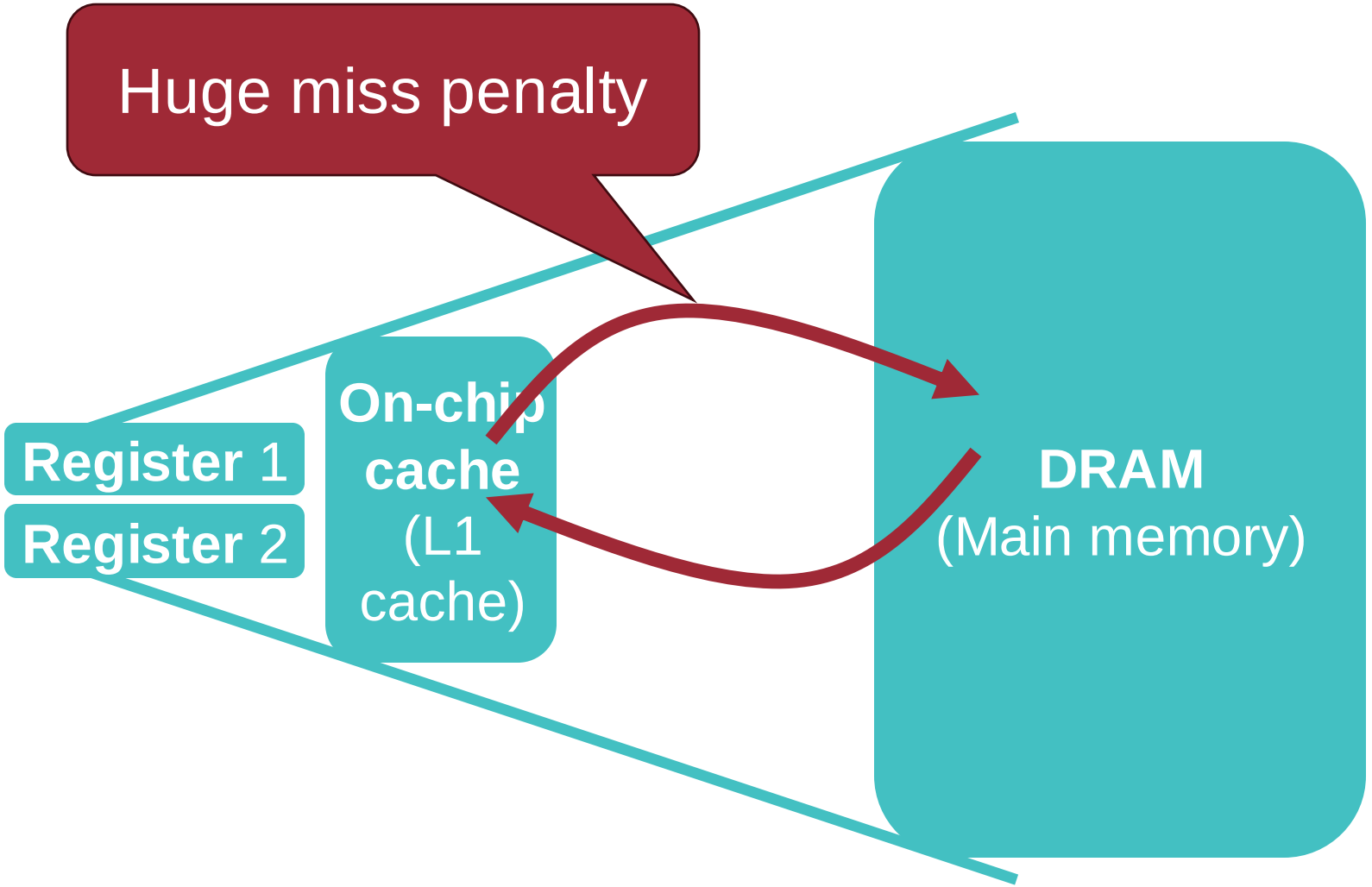
How to Improve the Cache Performance?

- **Adjust the block size**
- **Adjust the cache placement policy**
- **Adjust the cache replacement policy**
- **Multilevel caches**

How to Improve the Cache Performance?

- Adjust the block size
- Adjust the cache placement policy
- Adjust the cache replacement policy
- Multilevel caches

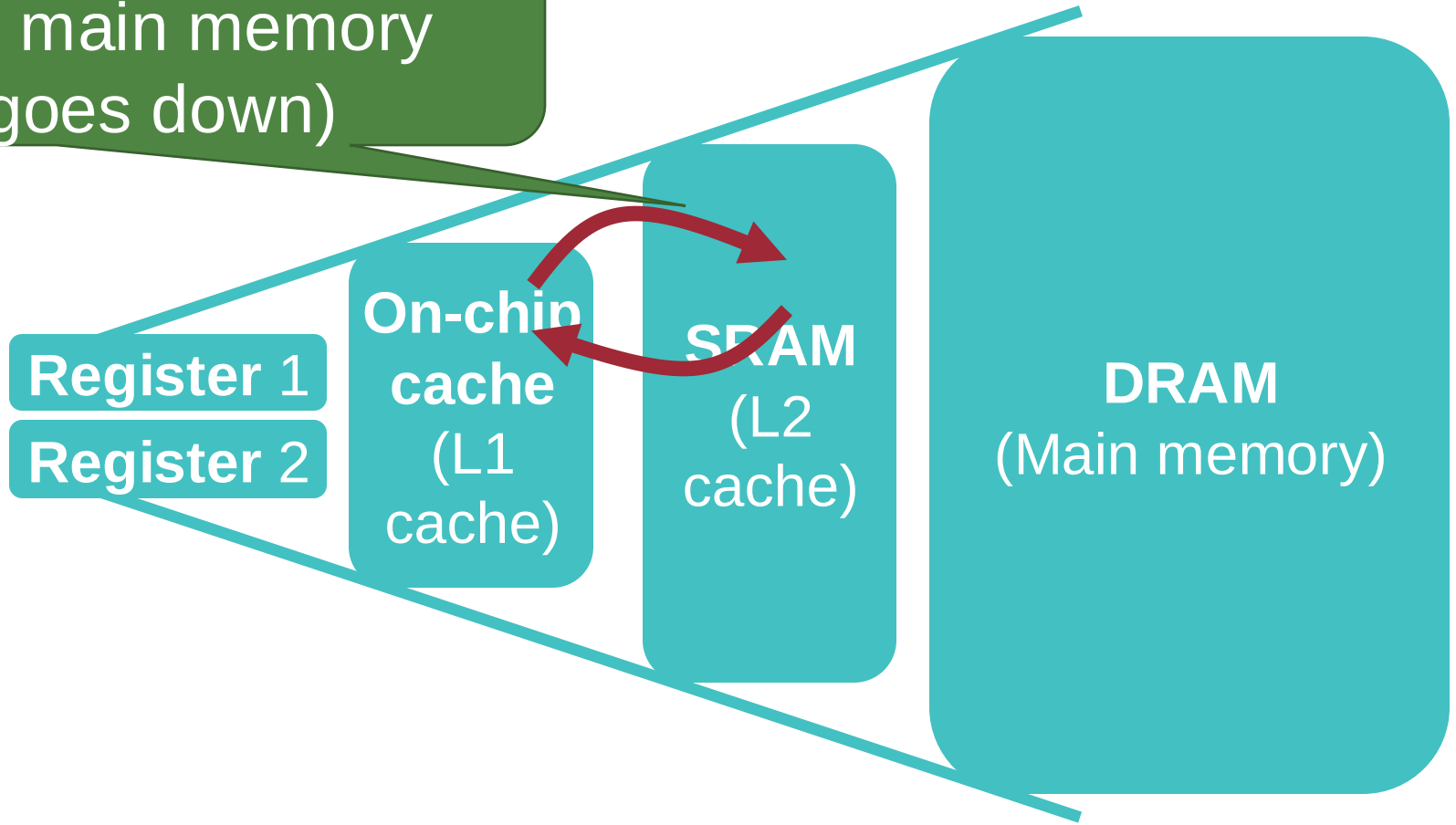
Multilevel Caches



Multilevel Caches

- Add a second level (L2) cache

Larger and slower than the L1 cache, but sill faster than main memory (Miss penalty goes down)



Summary: How to Improve the Cache Performance?

- **Adjust the block size**
  - When the cache block size is too small or too large, cache shows poor performance
- **Adjust the cache placement policy**
  - Direct mapped, fully associative, set associative
  - A fully associative cache theoretically shows best performance, but it is complex
- **Adjust the cache replacement policy**
  - Generally, LRU usually shows best performance
- **Multilevel caches**
  - Nowadays, there are more than one level of cache

Question?