Caches 2

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 5.1-5.4, 5.8, 5.15

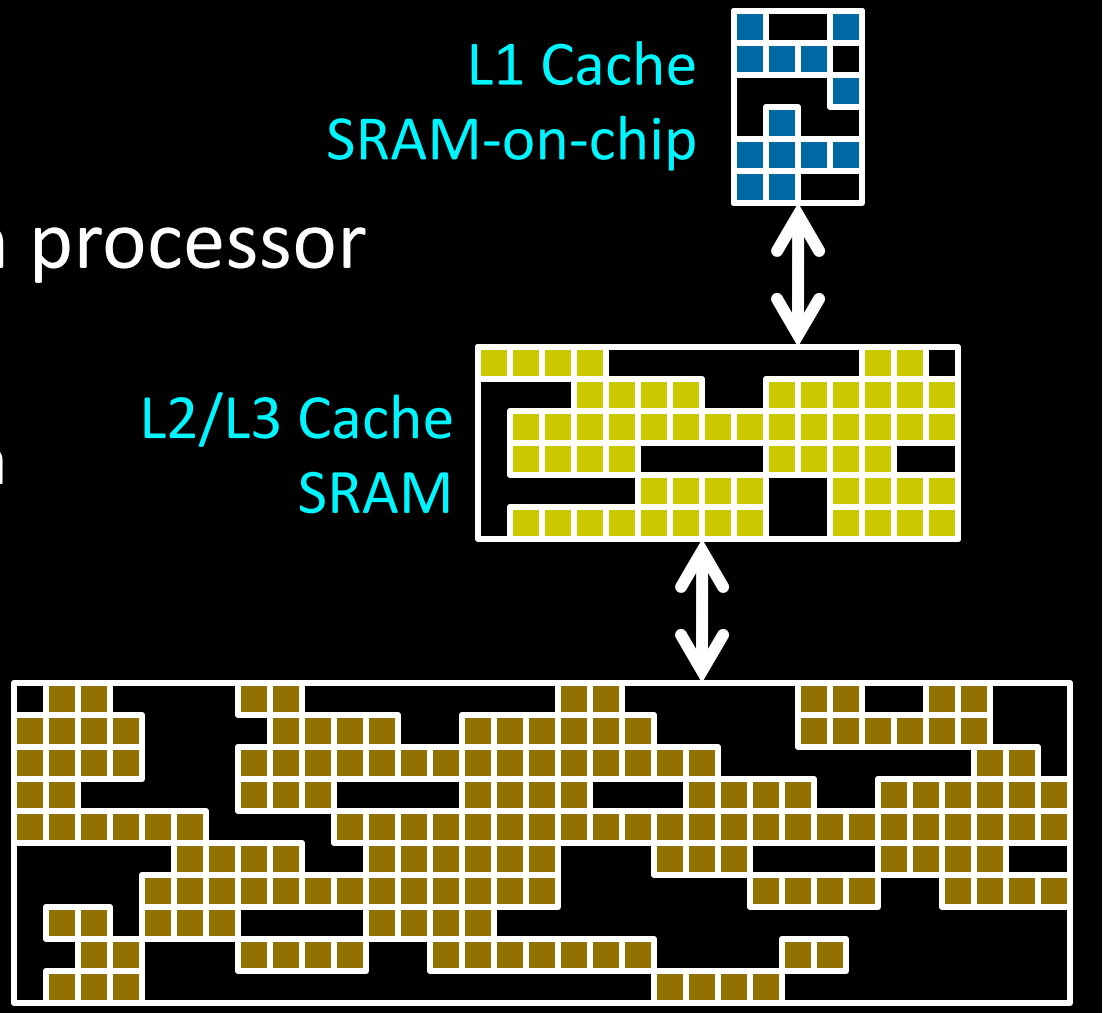
Memory Hierarchy

Memory closer to processor

- small & fast
- stores active data

Memory farther from processor

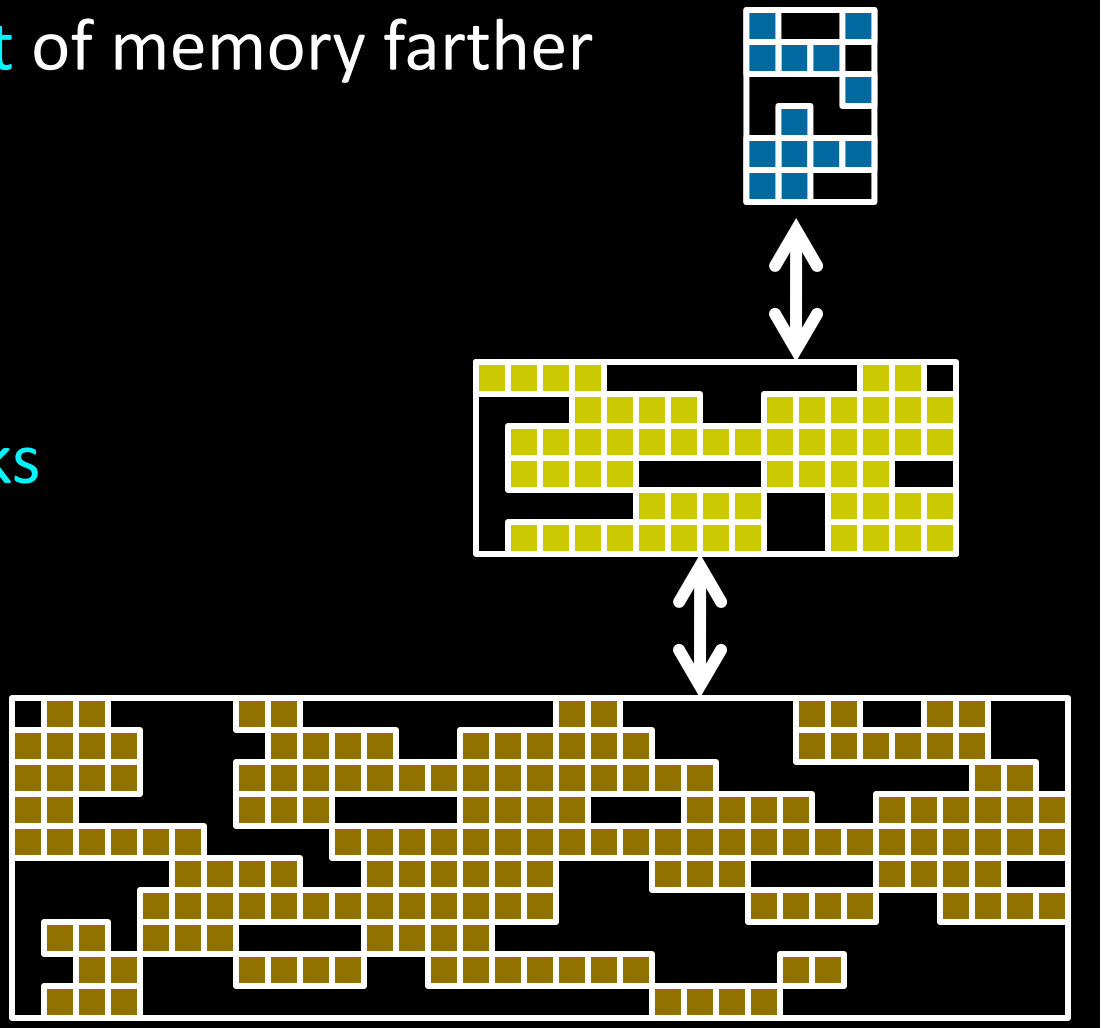
- big & slow
- stores inactive data



Memory Hierarchy

Memory closer to processor is fast but small

- usually stores subset of memory farther
  - "strictly inclusive"



- Transfer whole blocks (cache lines):

  4kb: disk ↔ RAM

  256b: RAM ↔ L2

  64b: L2 ↔ L1

Cache Questions

- What structure to use?
  - Where to place a block (book)?
  - How to find a block (book)?
- When miss, which block to replace?
- What happens on write?

Today

Cache organization

- Direct Mapped
- Fully Associative
- N-way set associative

Cache Tradeoffs

Next time: cache writing

Cache Lookups (Read)

Processor tries to access Mem[x]

Check: is block containing Mem[x] in the cache?

- Yes: cache hit
  - return requested data from cache line
- No: cache miss
  - read block from memory (or lower level cache)
  - (evict an existing cache line to make room)
  - place new block in cache
  - return requested data

  → and stall the pipeline while all of this happens

Questions

How to organize cache

What are tradeoffs in performance and cost?

Three common designs

A given data block can be placed…

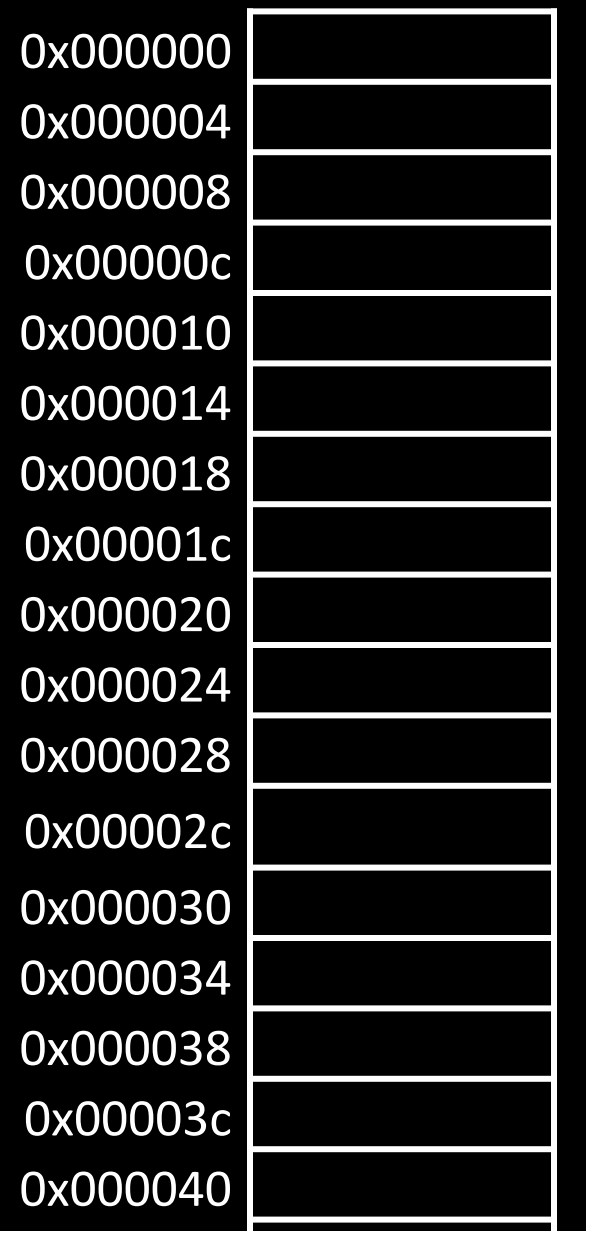
- … in exactly one cache line → Direct Mapped
- … in any cache line → Fully Associative
  - This is most like my desk with books
- … in a small set of cache lines → Set Associative

Direct Mapped Cache

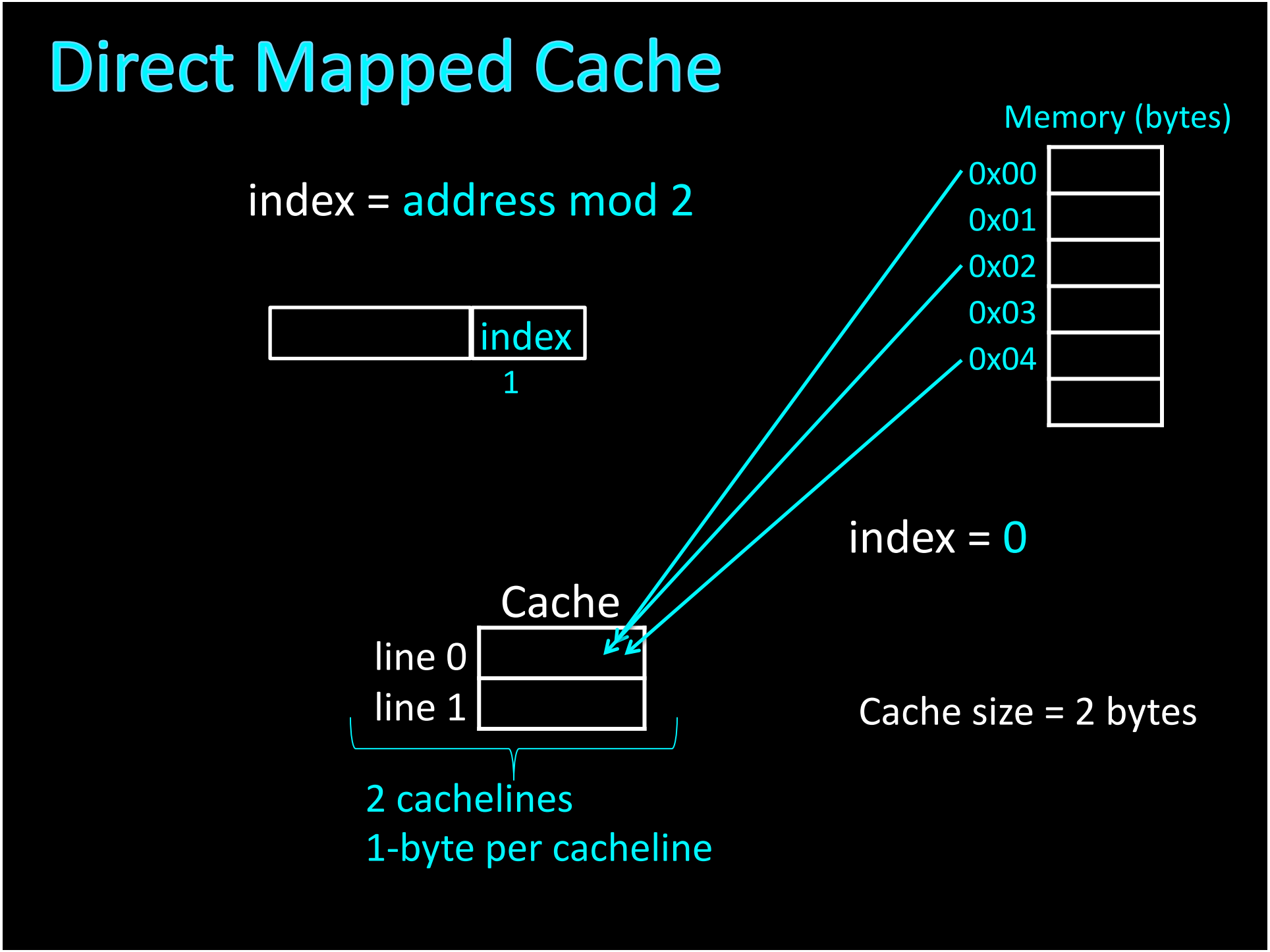
- Each block number maps to a single cache line index

- Where?

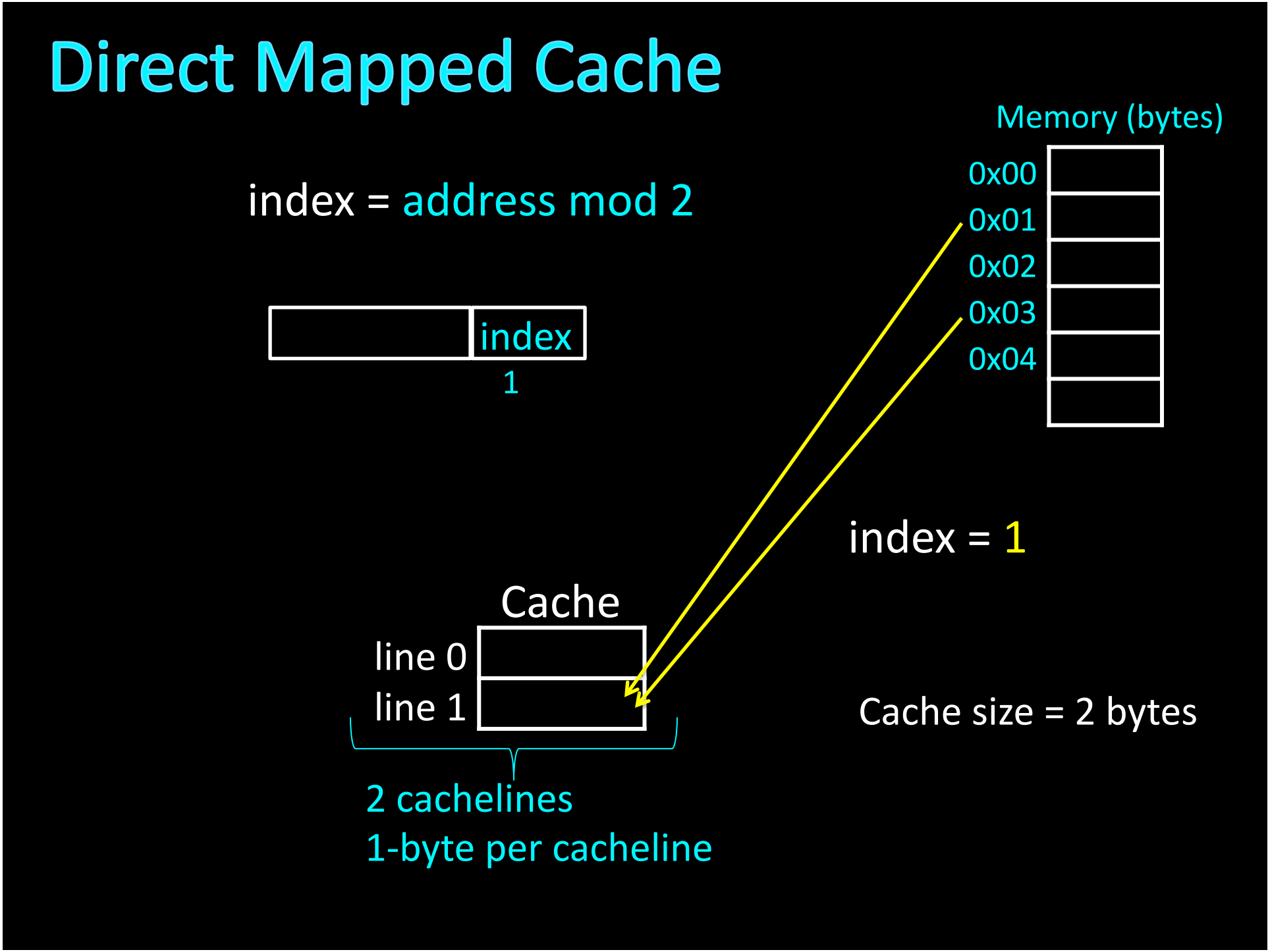
  address mod #blocks in cache



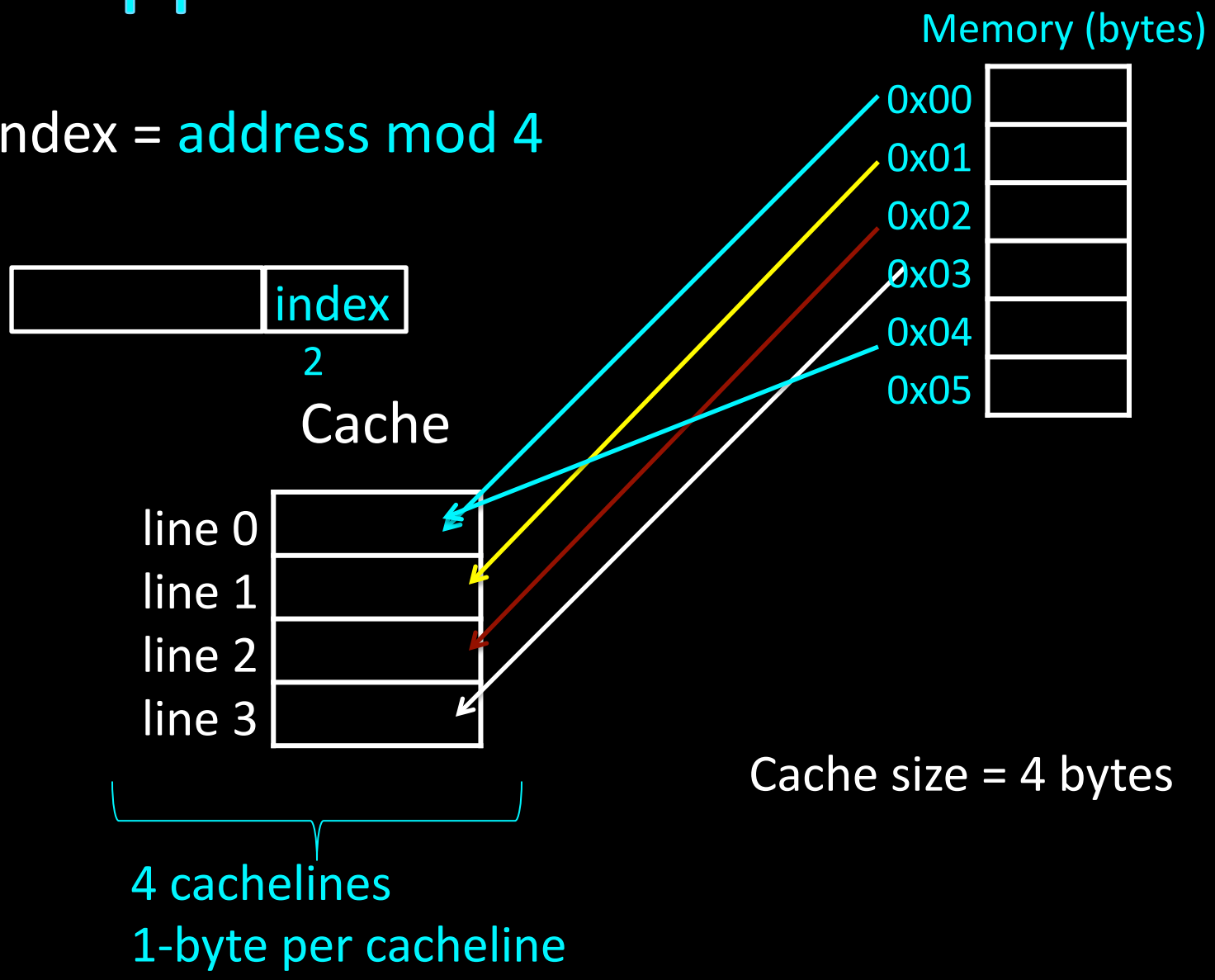
Direct Mapped Cache



Direct Mapped Cache



Direct Mapped Cache



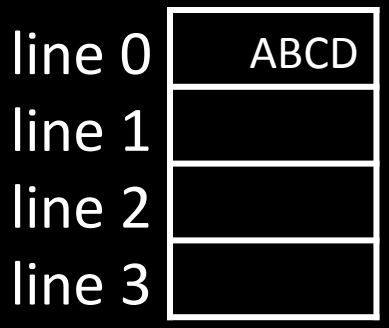
Direct Mapped Cache

index = address mod 4

offset = which byte in each line

Memory (word)

Cache



4 cachelines

1-**word** per cacheline

Cache size = 16 bytes

Direct Mapped Cache: 2

index = address mod 4

offset = which byte in each line

offset 3 bits: A, B, C, D, E, F, G, H

Cache

line 0	ABCD	EFGH
line 1	IJKL	MNOP
line 2	QRST	UVWX
line 3	YZ12	3456

4 cachelines

2-words (8 bytes) per cacheline

Memory

line 0	0x000000	ABCD
	0x000004	EFGH
line 1	0x000008	IJKL
	0x00000c	MNOP
line 2	0x000010	QRST
	0x000014	UVWX
line 3	0x000018	YZ12
	0x00001c	3456
line 0	0x000020	abcd
	0x000024	efgh
line 1	0x000028	
	0x00002c	
line 2	0x000030	
	0x000034	
line 3	0x000038	
	0x00003c	
	0x000040	
	0x000044	

Direct Mapped Cache: 2

tag = which memory element is it?

0x00, 0x20, 0x40?

Cache

	Tag & valid bits		
line 0		ABCD	EFGH
line 1		IJKL	MNOP
line 2		QRST	UVWX
line 3		YZ12	3456

4 cachelines

2-words (8 bytes) per cacheline

Memory

line 0	0x000000	ABCD
	0x000004	EFGH
line 1	0x000008	IJKL
	0x00000c	MNOP
line 2	0x000010	QRST
	0x000014	UVWX
line 3	0x000018	YZ12
	0x00001c	3456
line 0	0x000020	abcd
	0x000024	efgh
line 1	0x000028	
	0x00002c	
line 2	0x000030	
	0x000034	
line 3	0x000038	
	0x00003c	
	0x000040	
	0x000044	

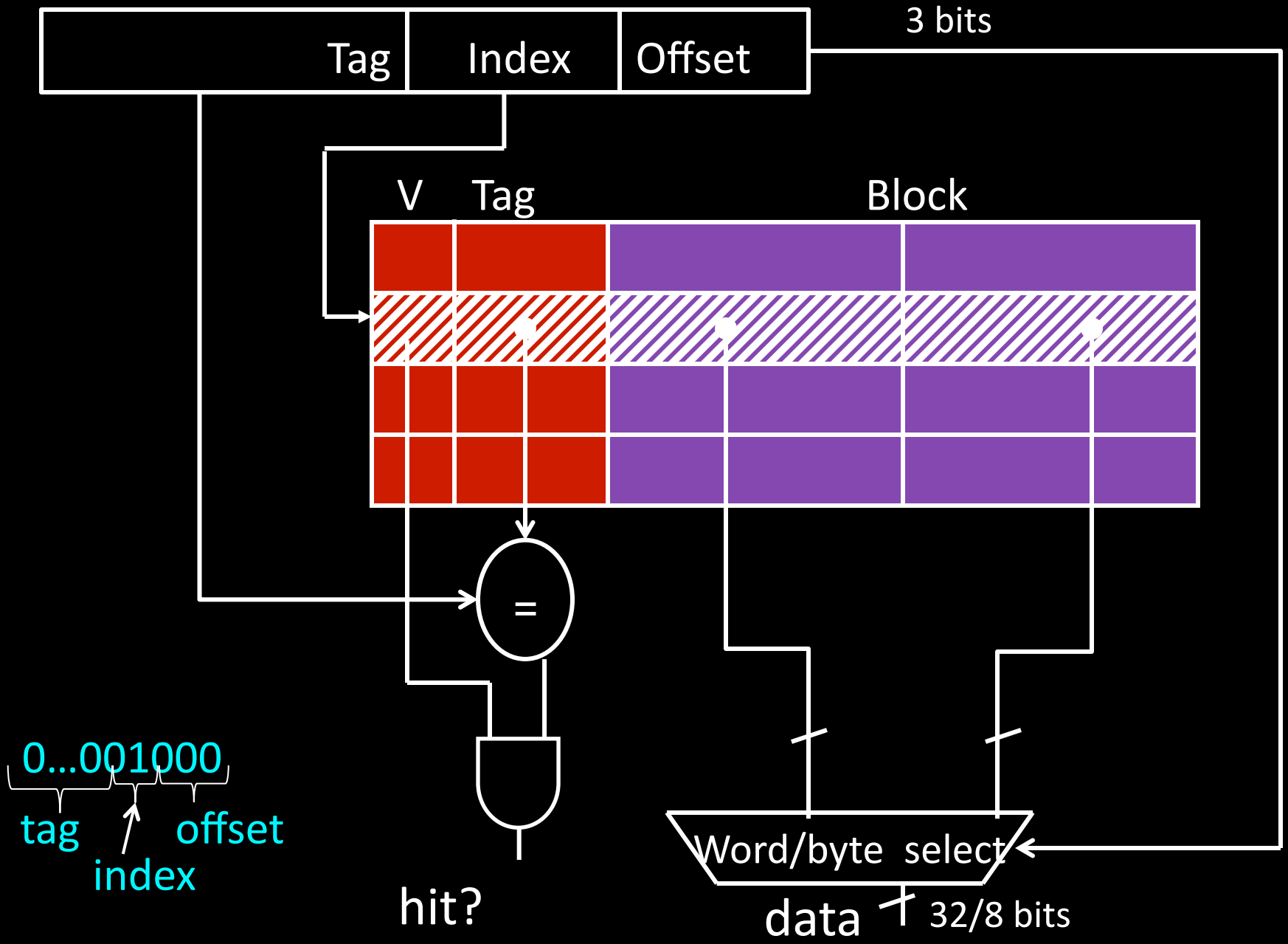
Direct Mapped Cache

Every address maps to one location

Pros: Very simple hardware

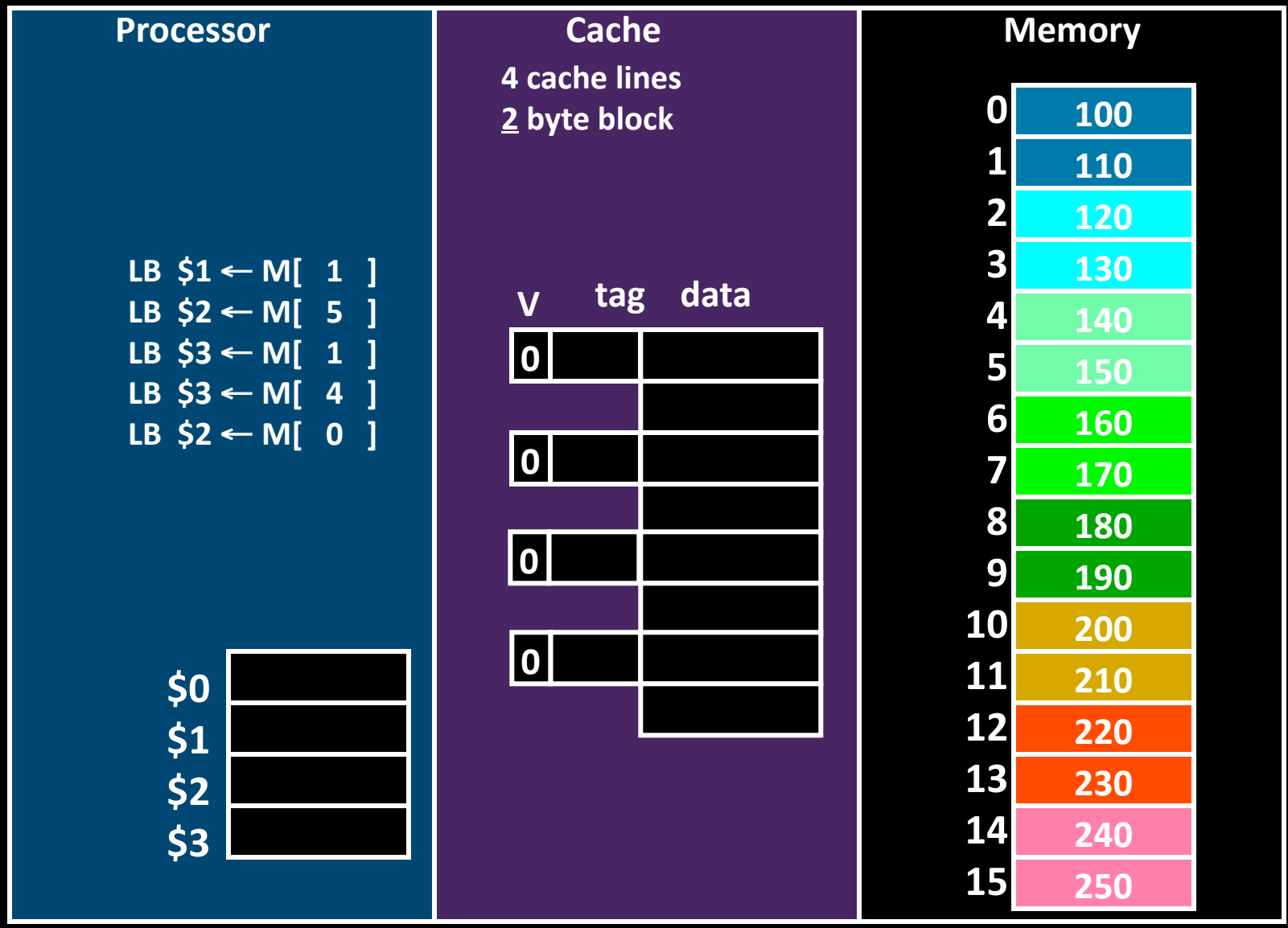
Cons: many different addresses land on same location and may compete with each other

Direct Mapped Cache (Hardware)



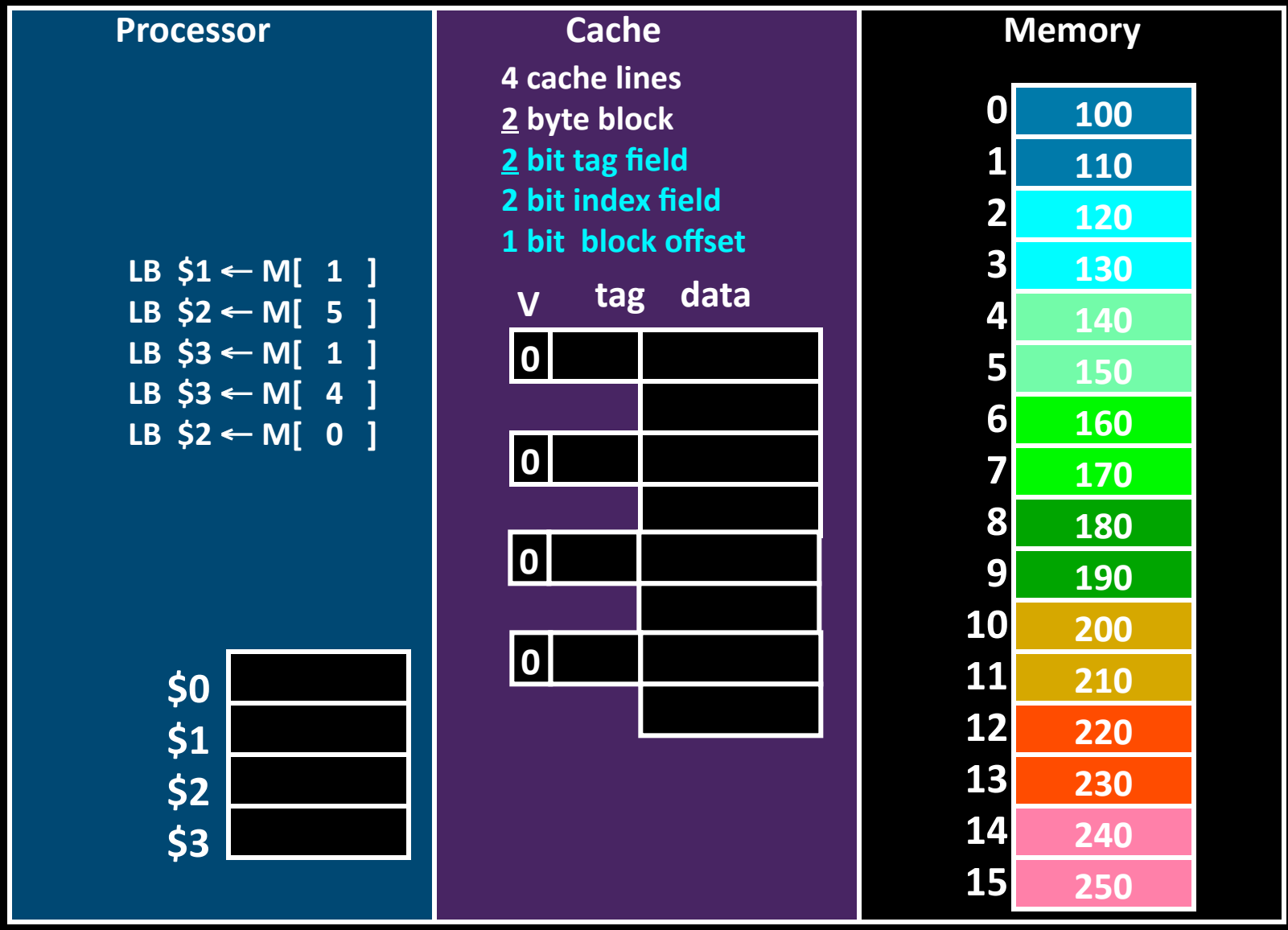
Example: Direct Mapped Cache

Using **byte addresses** in this example. Addr Bus = 5 bits



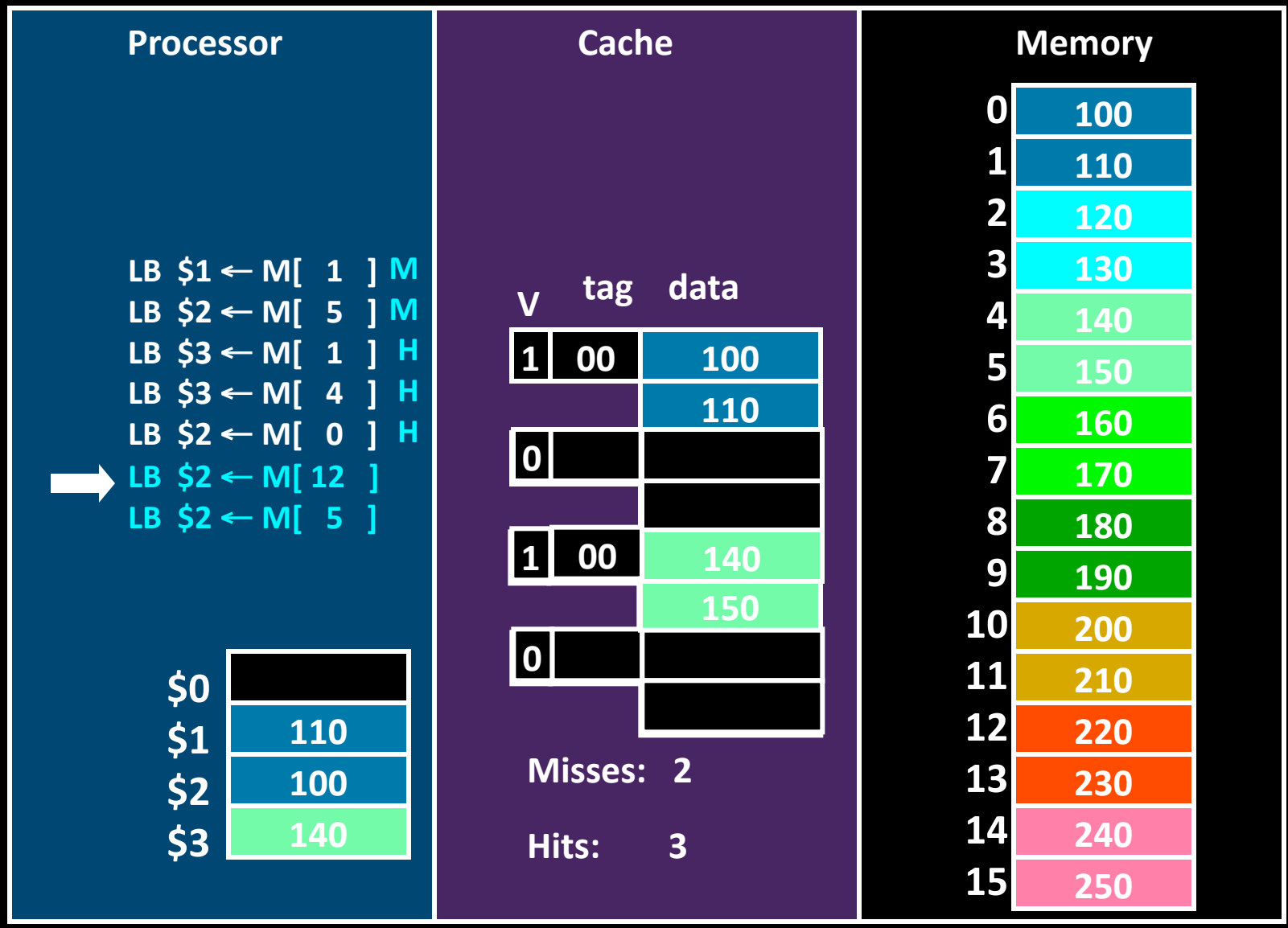
Example: Direct Mapped Cache

Using **byte addresses** in this example. Addr Bus = 5 bits

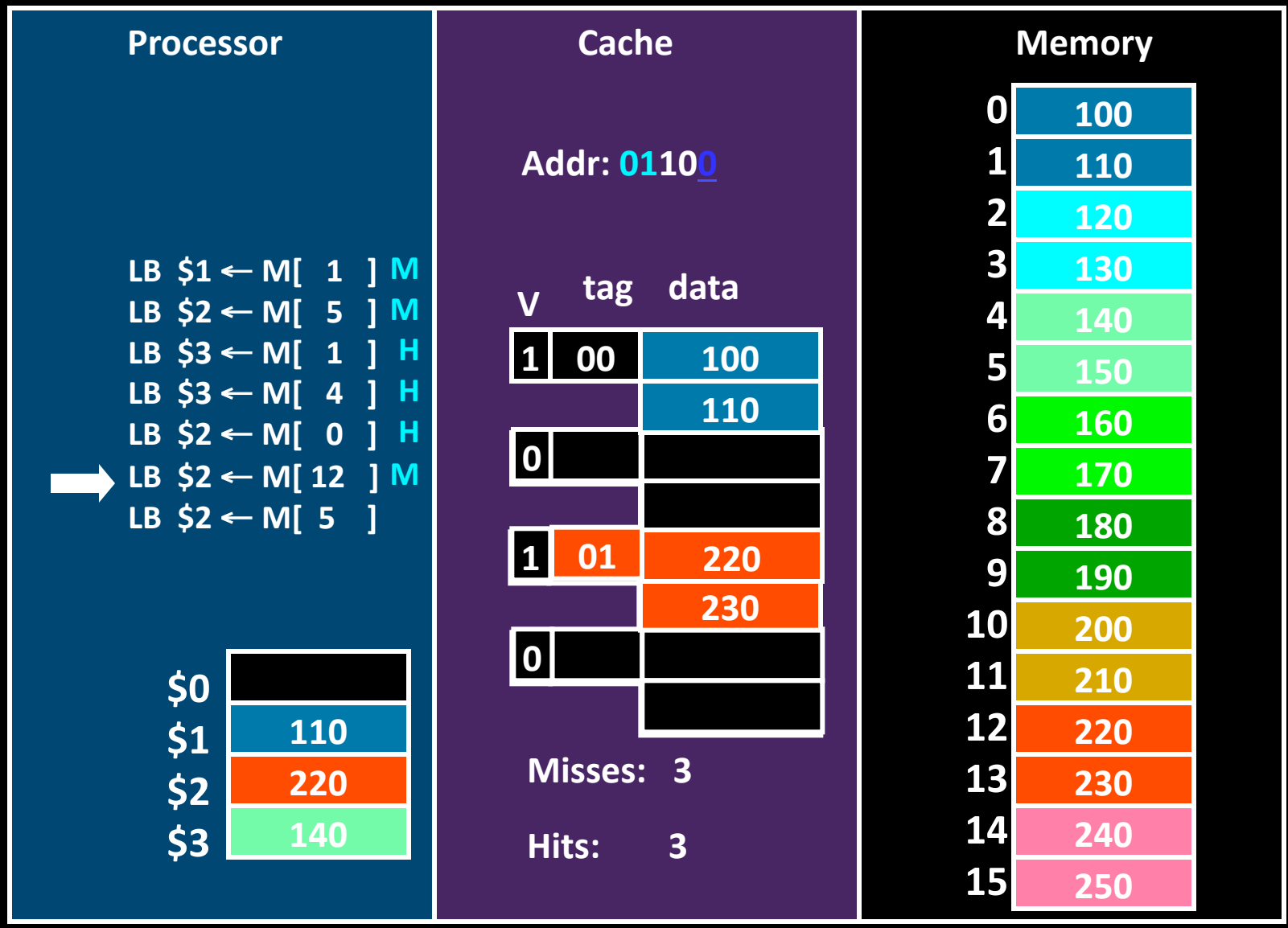


Direct Mapped Example: 6th Access

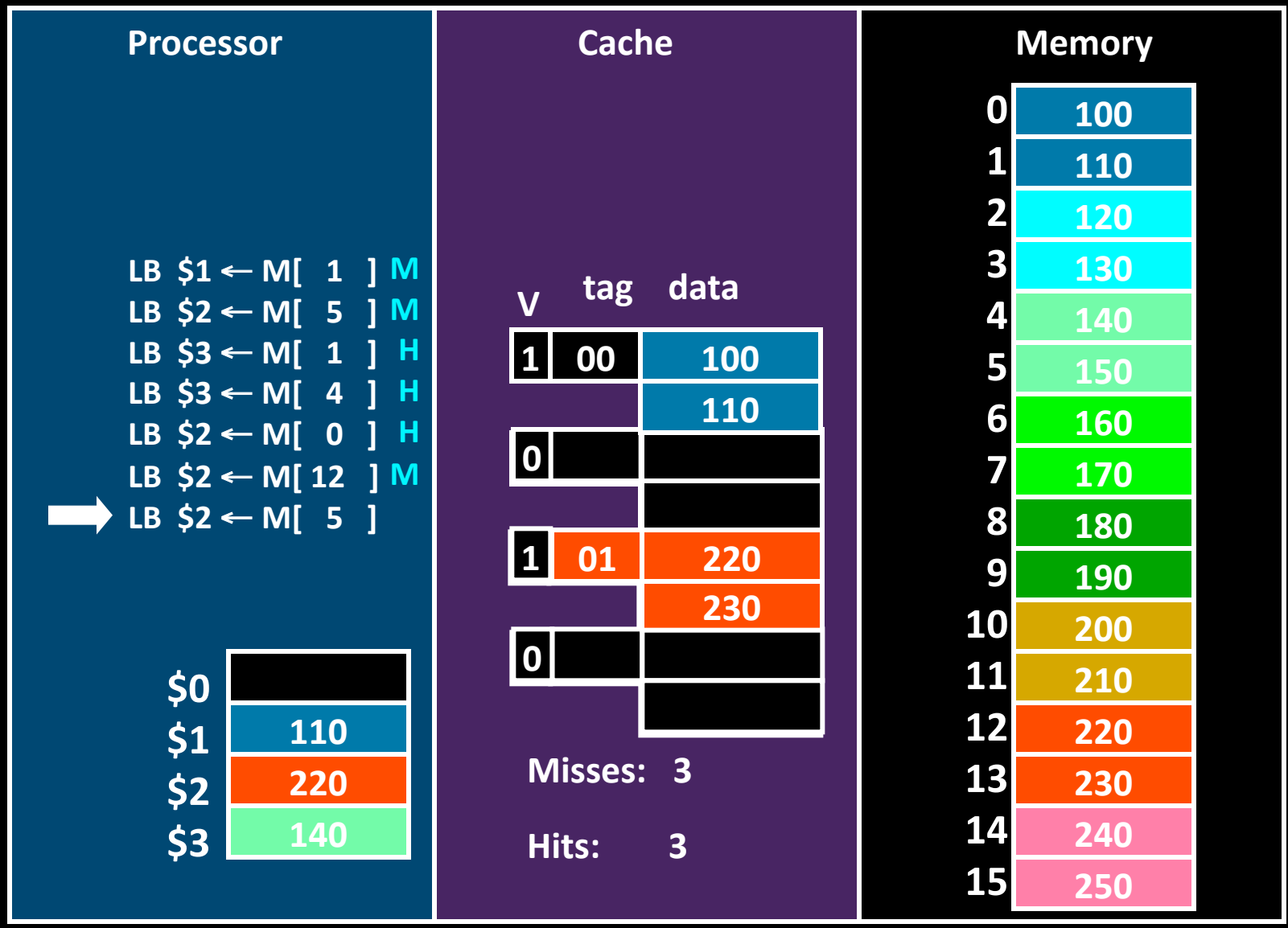
Pathological example



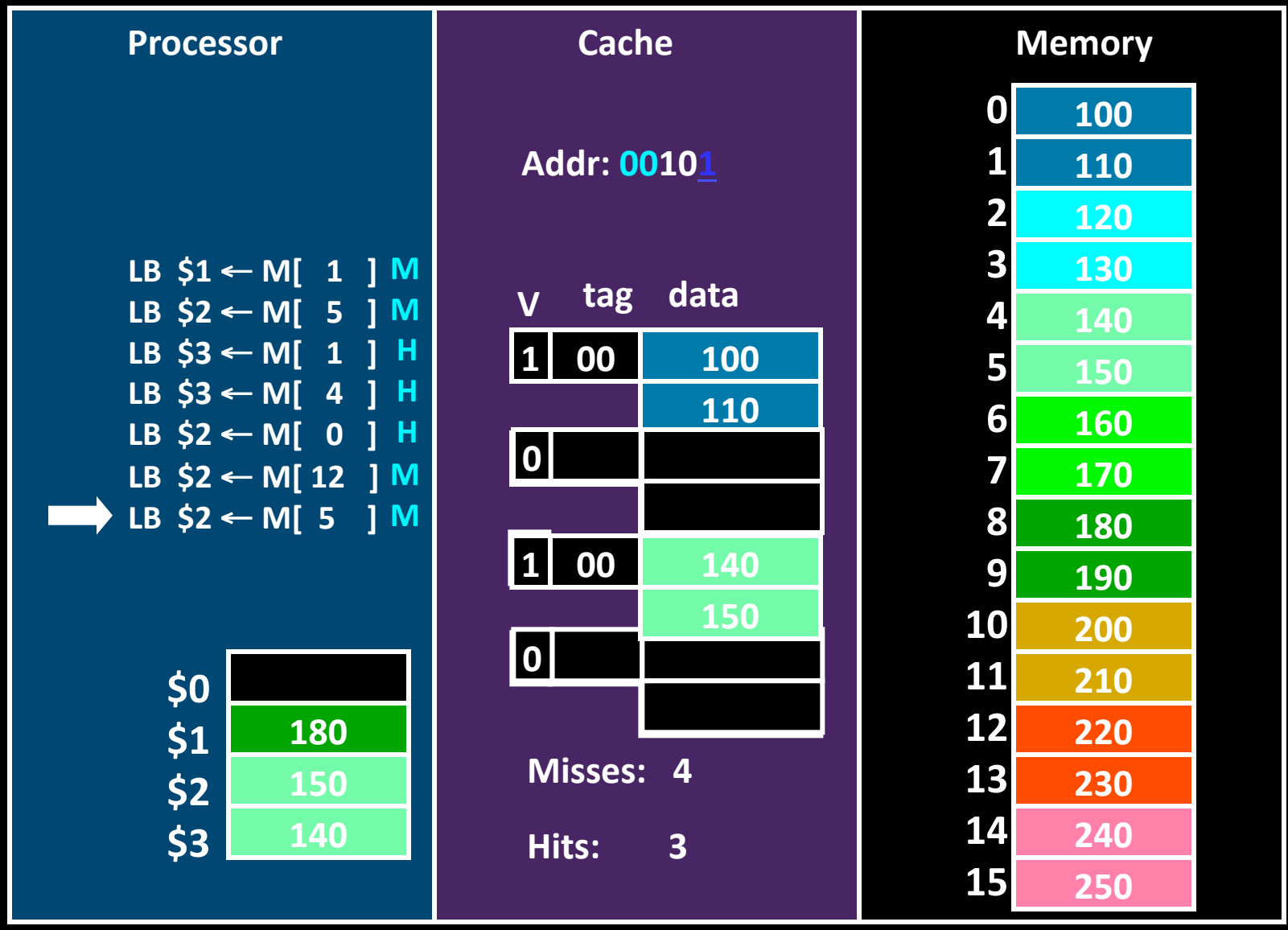
6th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[12] M
LB $2 ← M[5]
$0
$1 110
$2 220
$3 140
Cache
Addr: 01100
V tag data
1 00 100
110
0
1 01 220
230
0
Misses: 3
Hits: 3
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

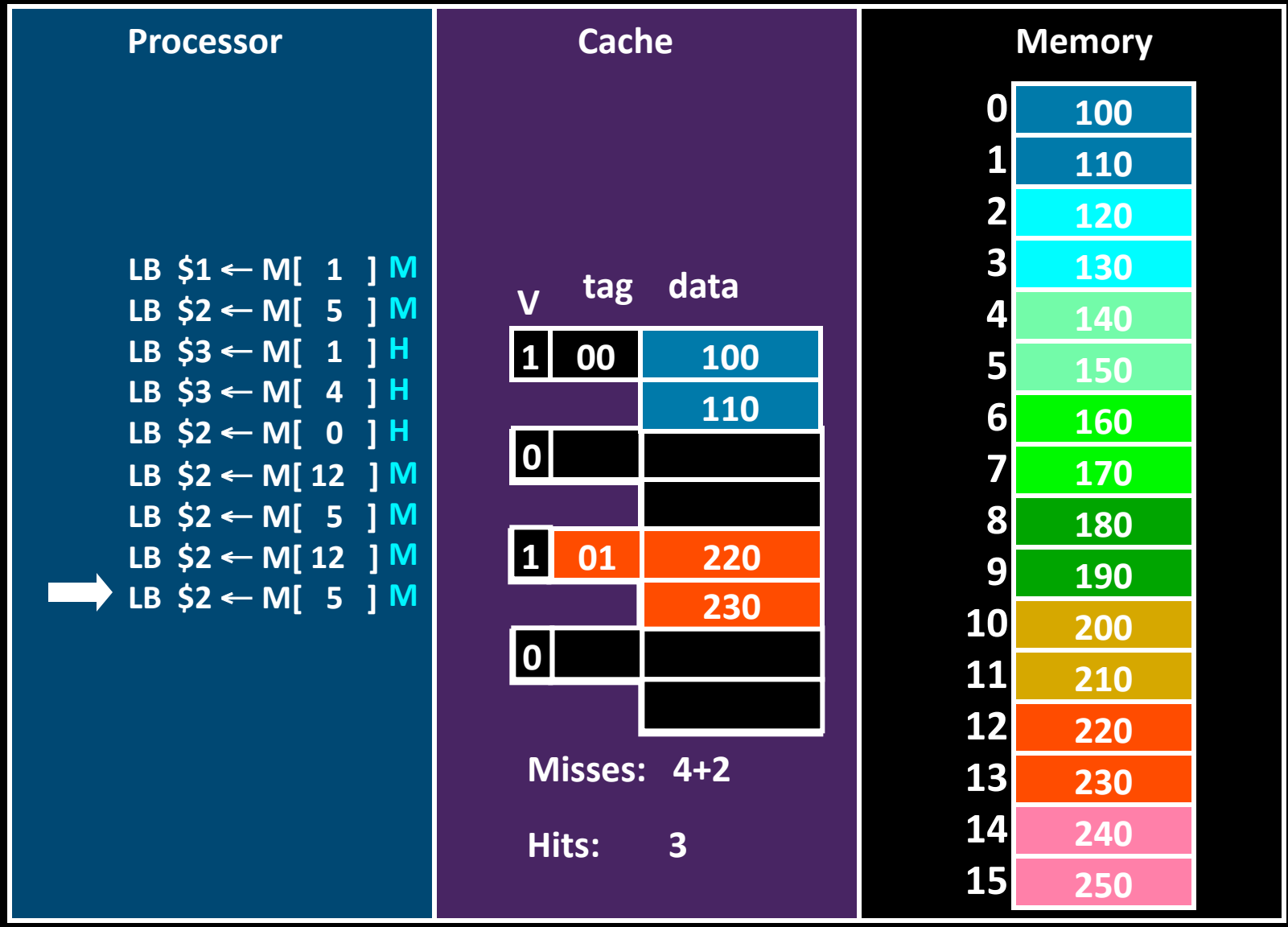
7th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[12] M
LB $2 ← M[5]
$0
$1 110
$2 220
$3 140
Cache
V tag data
1 00 100
110
0
1 01 220
230
0
Misses: 3
Hits: 3
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

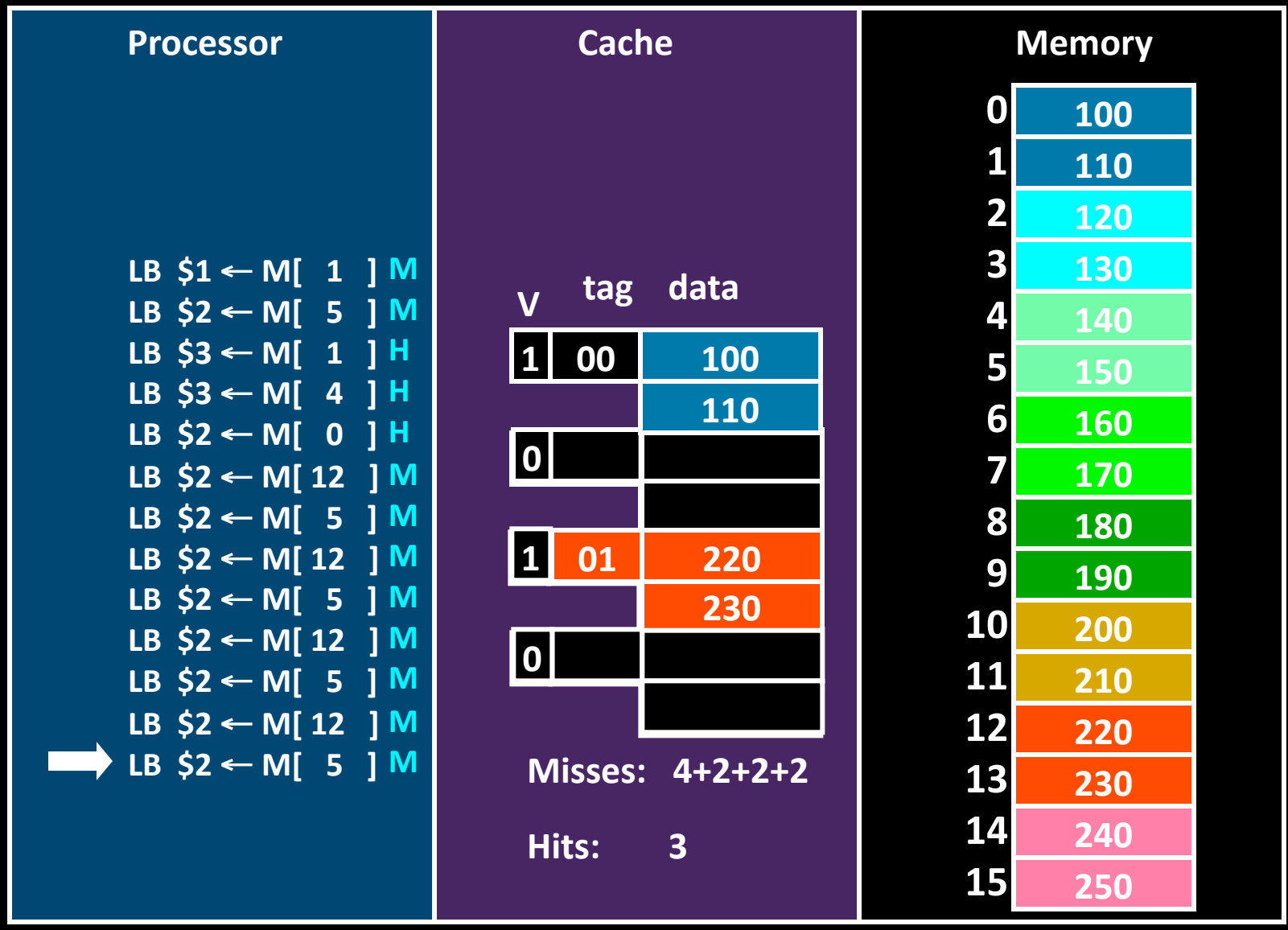
7th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[12] M
LB $2 ← M[5] M
$0
$1 180
$2 150
$3 140
Cache
Addr: 00101
V tag data
1 00 100
110
0
1 00 140
150
0
Misses: 4
Hits: 3
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

8th and 9th Access



10th and 11th, 12th and 13th Access



Problem

Working set is not too big for cache

Yet, we are not getting any hits?!

Misses

Three types of misses

- Cold (aka Compulsory)
  - The line is being referenced for the first time
- Capacity
  - The line was evicted because the cache was not large enough
- Conflict
  - The line was evicted because of another access whose index conflicted

Misses

Q: How to avoid…

Cold Misses

- Unavoidable? The data was never in the cache…
- Prefetching!

Capacity Misses

- Buy more cache

Conflict Misses

- Use a more flexible cache design

Cache Organization

How to avoid Conflict Misses

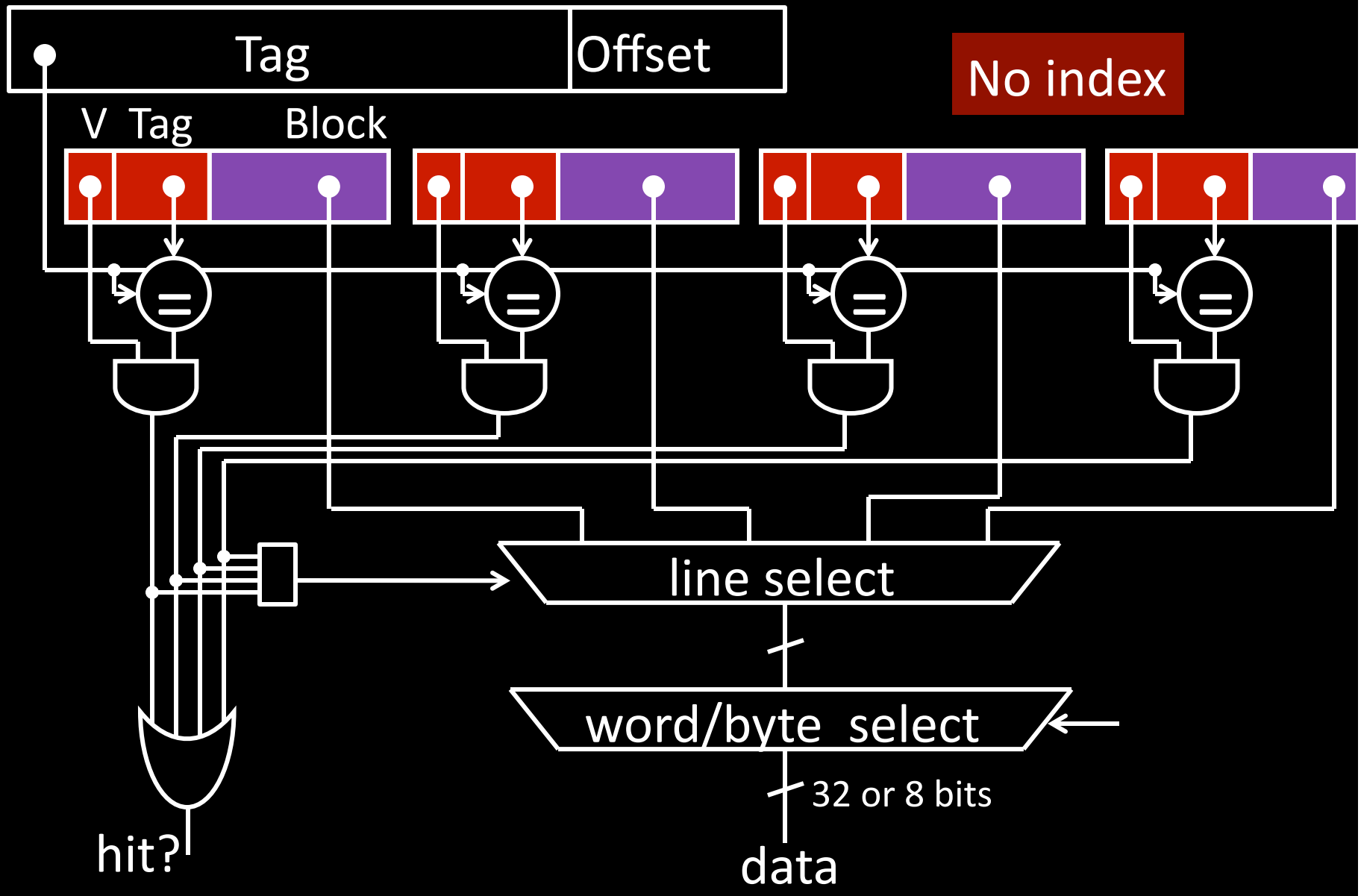
Three common designs

- Direct mapped: Block can only be in one line in the cache
- Fully associative: Block can be anywhere in the cache
- Set-associative: Block can be in a few (2 to 8) places in the cache

Fully Associative Cache

- Block can be anywhere in the cache
  - Most like our desk with library books
- Have to search in all entries to check for match
  - More expensive to implement in hardware
- But as long as there is capacity, can store in cache
  - So least misses

Fully Associative Cache (Reading)



Fully Associative Cache (Reading)

m bit offset , 2^n blocks (cache lines)

Q: How big is cache (**data only**)?

Cache of size 2^n blocks

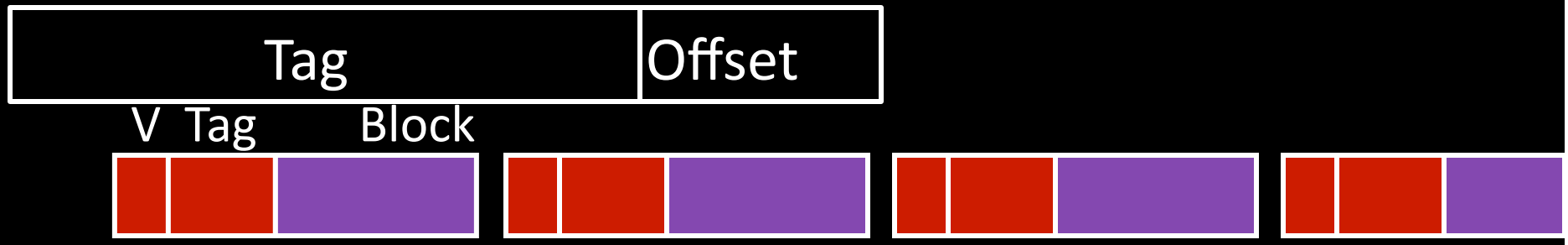
Block size of 2^m bytes

Cache Size: number-of-blocks x block size

= 2^n x 2^m bytes

= 2^{n+m} bytes

Fully Associative Cache (Reading)



m bit offset , 2^n blocks (cache lines)

Q: How much SRAM needed (data + overhead)?

Cache of size 2^n blocks

Block size of 2^m bytes

Tag field: 32 – m

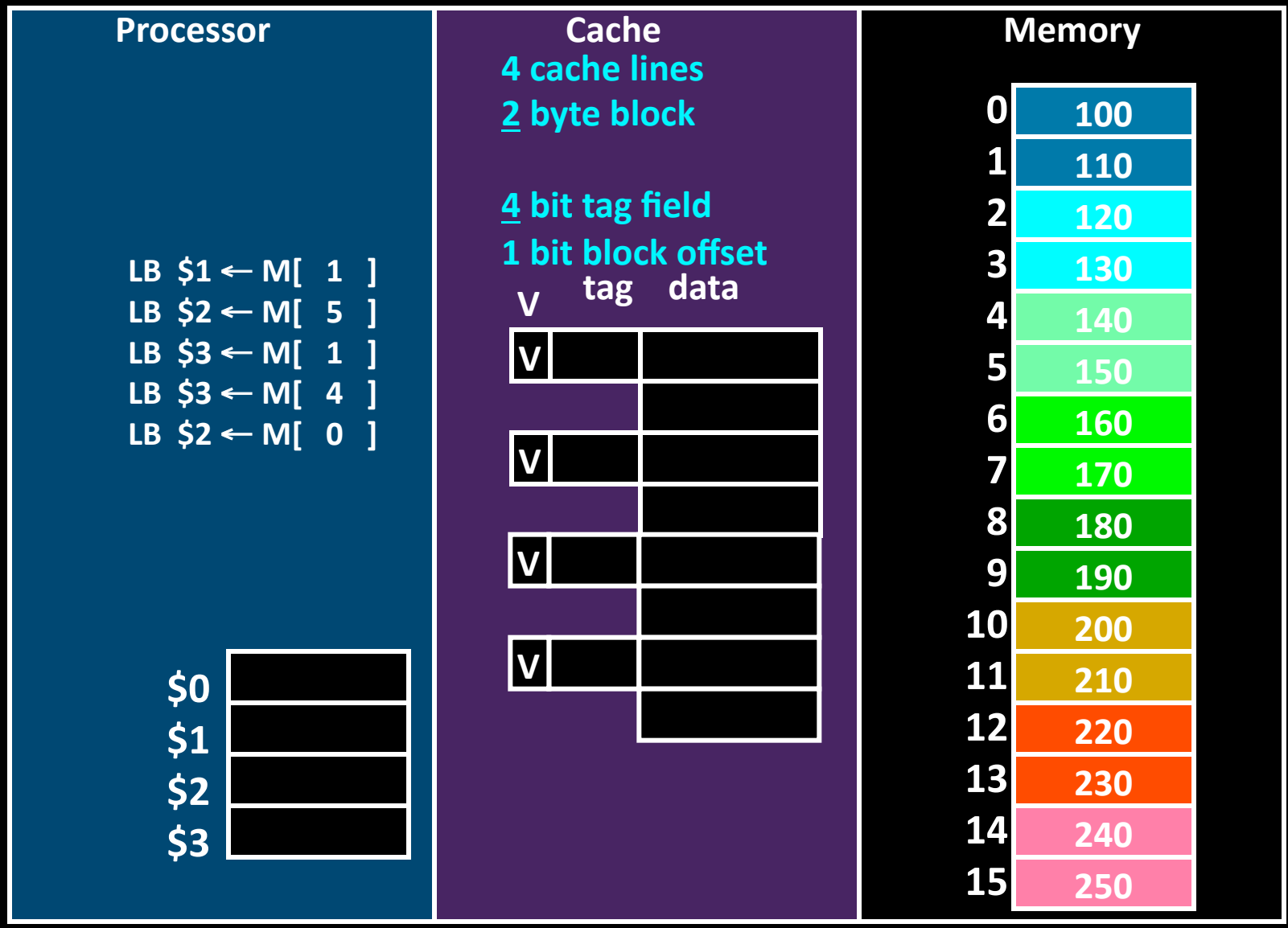
Valid bit: 1

SRAM size: 2^n x (block size + tag size + valid bit size)

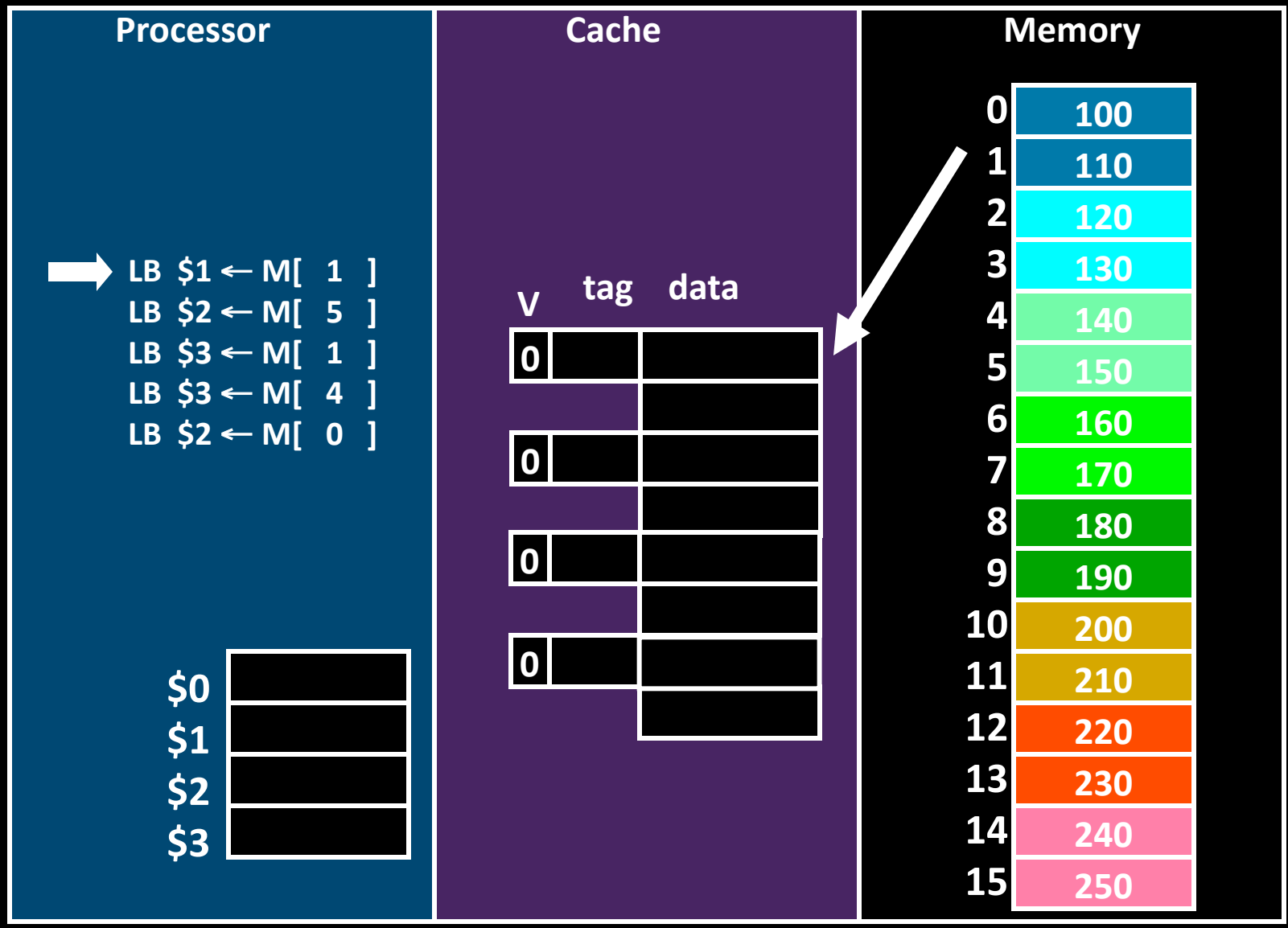
= 2^nx (2^m bytes x 8 bits-per-byte + (32-m) + 1)

Example: Simple Fully Associative Cache

Using **byte addresses** in this example! Addr Bus = 5 bits



1st Access

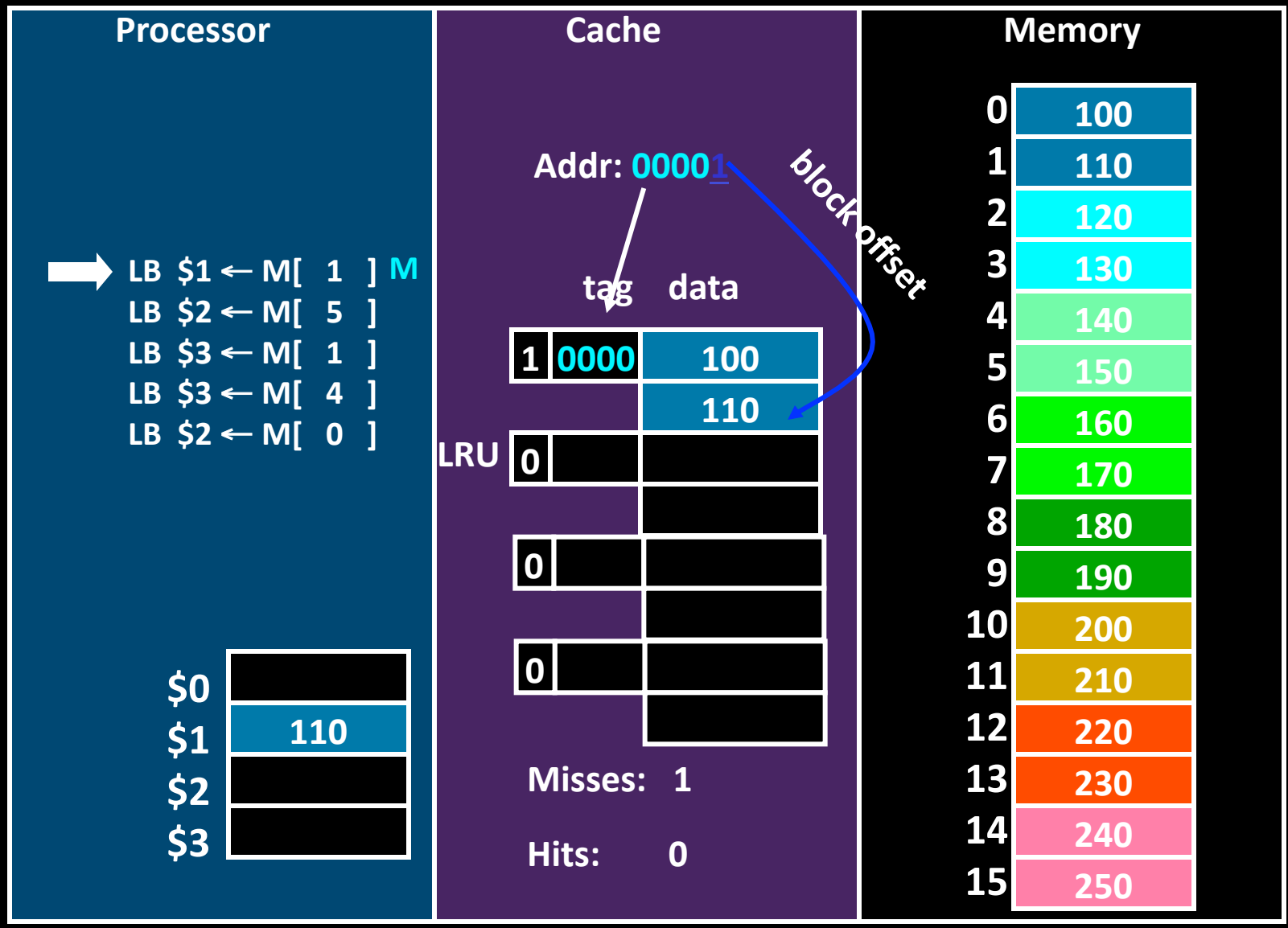

Processor
Cache
Memory
LB $1 ← M[1]
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
V
tag
data
0
0
0
0
$0
$1
$2
$3
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

Eviction

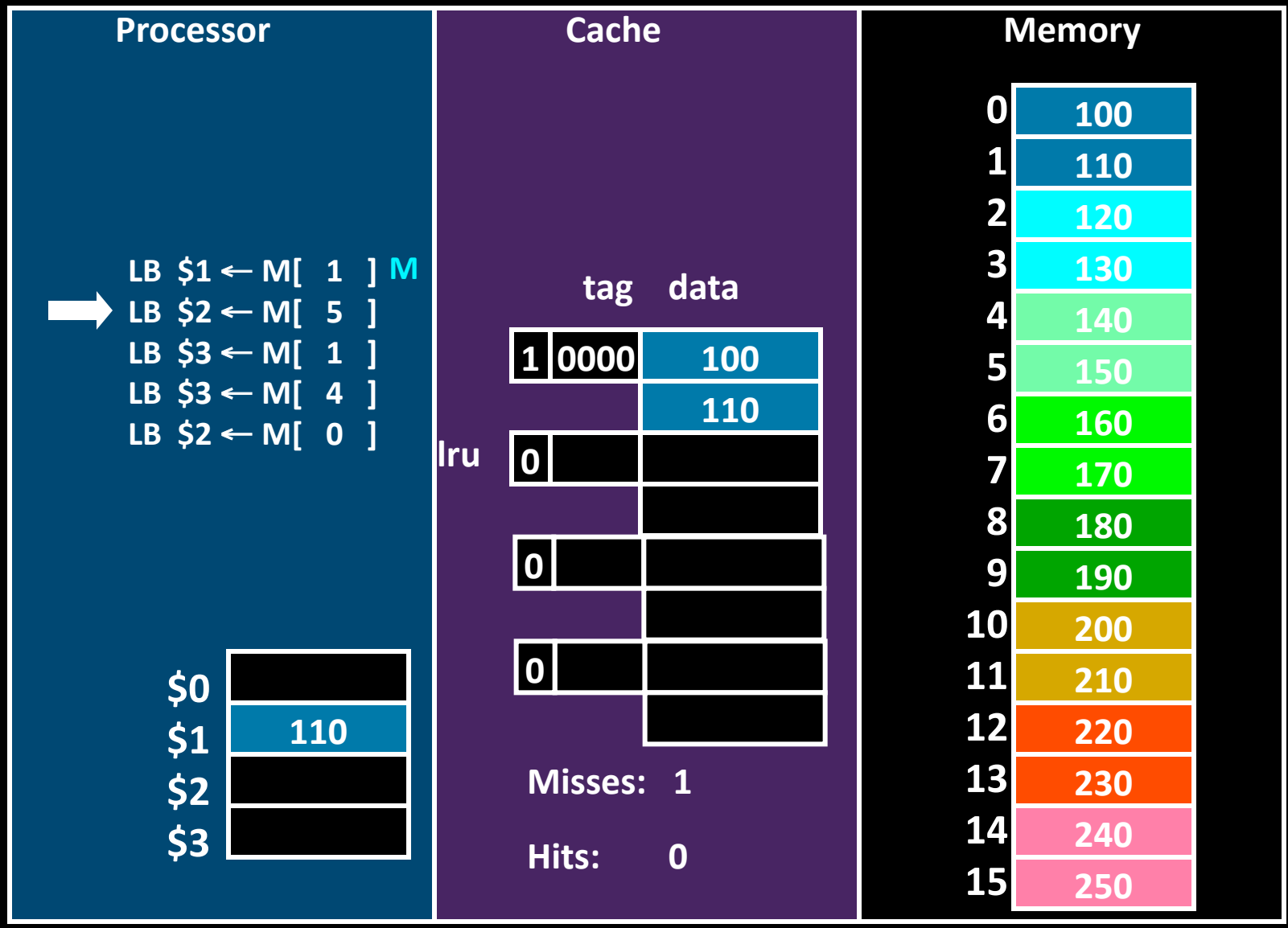
Which cache line should be evicted from the cache to make room for a new line?

- Direct-mapped
  - no choice, must evict line selected by index
- Associative caches
  - random: select one of the lines at random
  - round-robin: similar to random
  - FIFO: replace oldest line
  - LRU: replace line that has not been used in the longest time

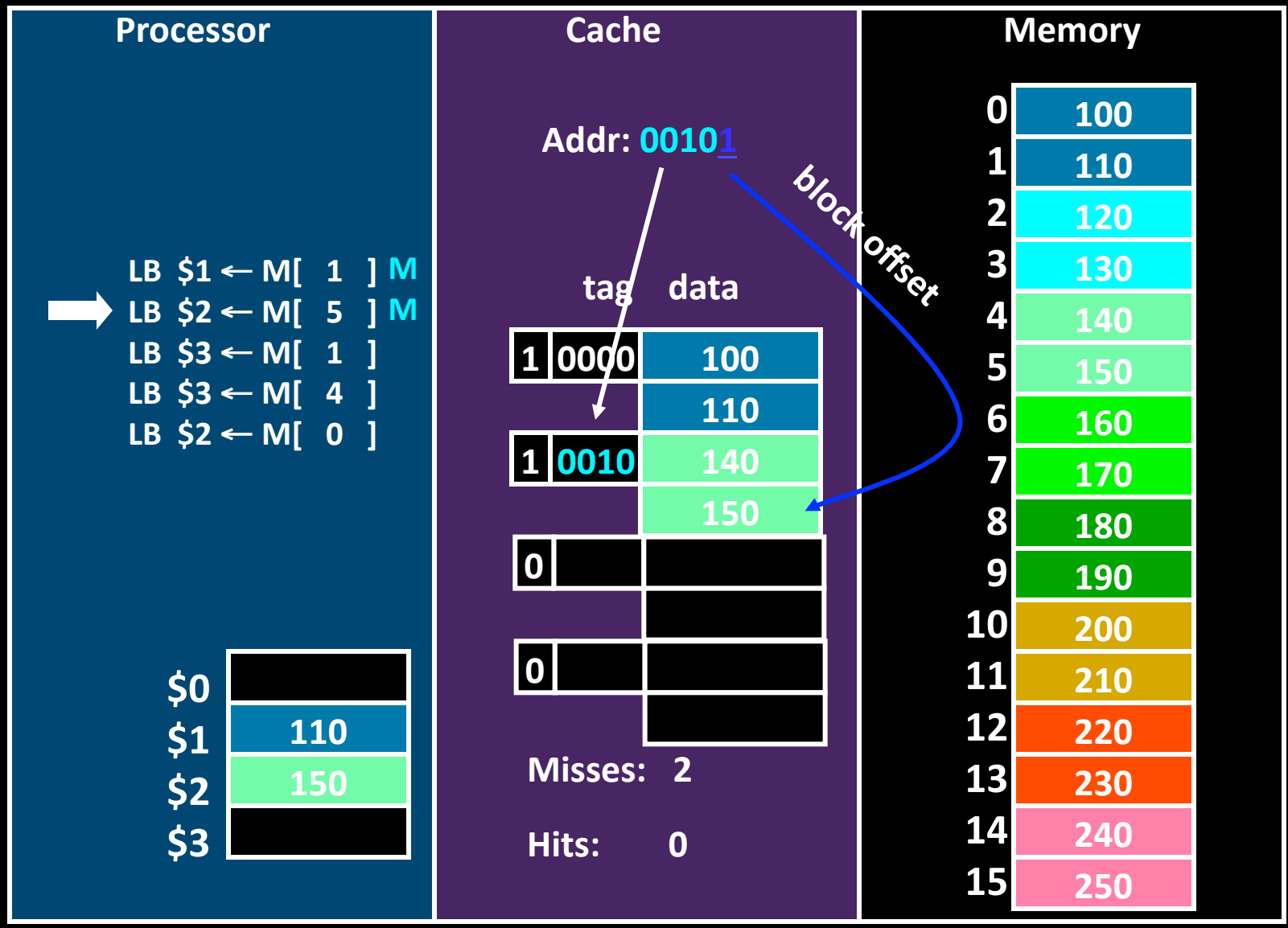
1st Access


Processor
Cache
Memory
LB $1 ← M[1] M
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1
110
$2
$3
Addr: 00001
block offset
tag
data
1
0000
100
110
LRU
0
0
0
Misses: 1
Hits: 0
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

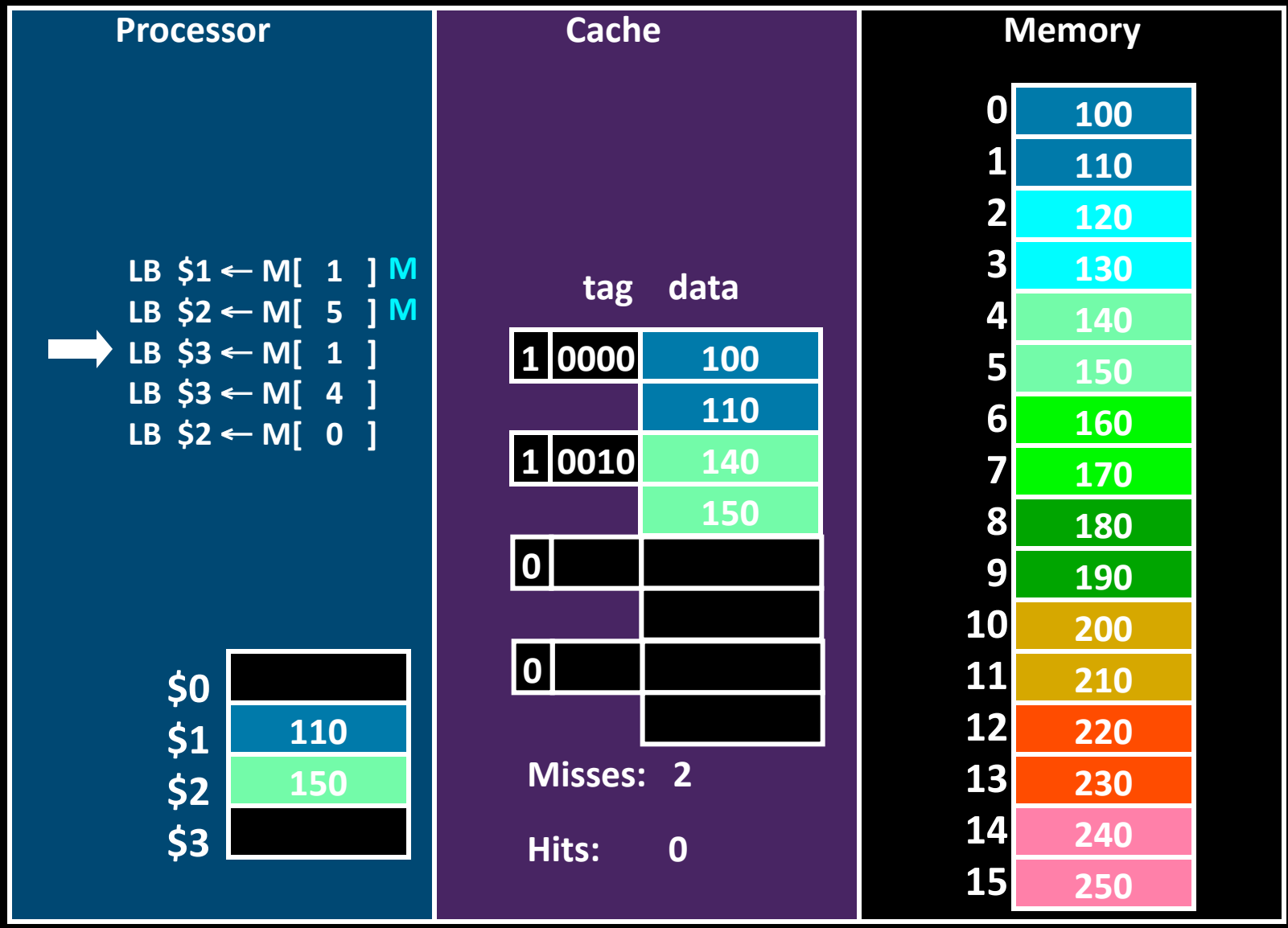
2nd Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1
110
$2
$3
Cache
tag data
1 0000 100
110
lru
0
0
0
Misses: 1
Hits: 0
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

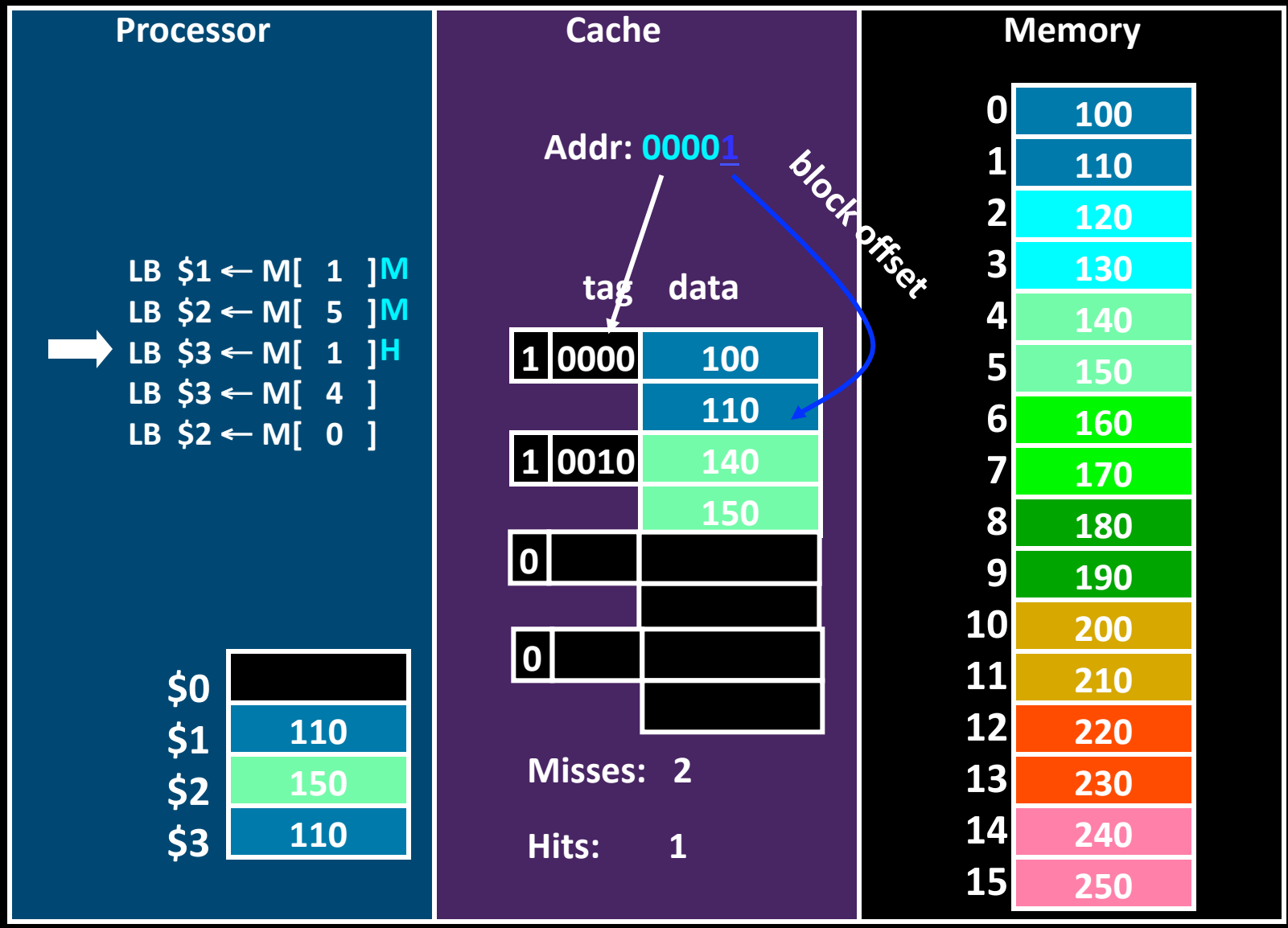
2nd Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1
110
$2
150
$3
Cache
Addr: 00101
block offset
tag
data
1 0000
100
110
1 0010
140
150
0
0
Misses: 2
Hits: 0
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

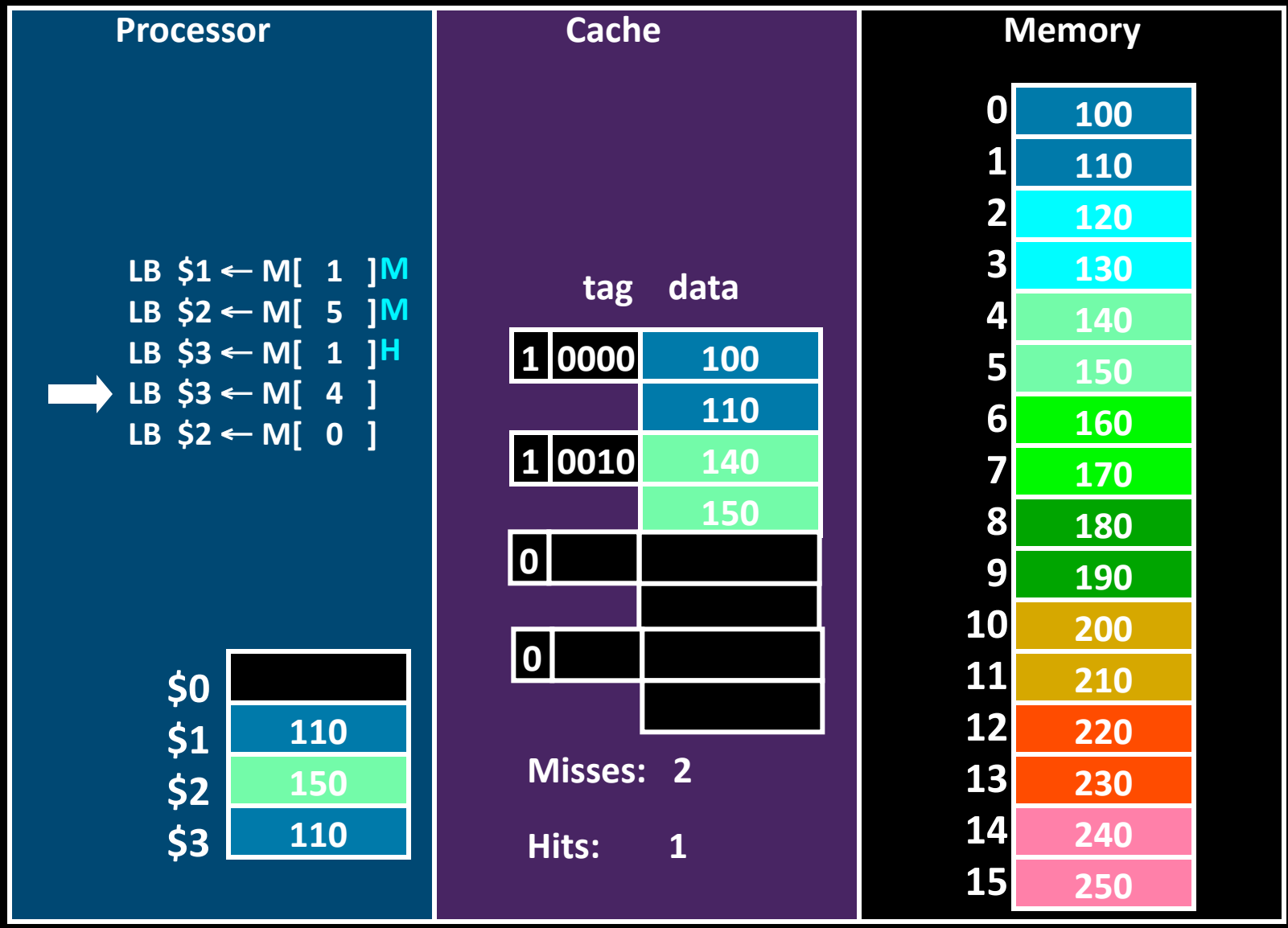
3rd Access


Processor
Cache
Memory
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
tag data
1 0000 100
110
1 0010 140
150
0
0
$0
$1 110
$2 150
$3
Misses: 2
Hits: 0
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

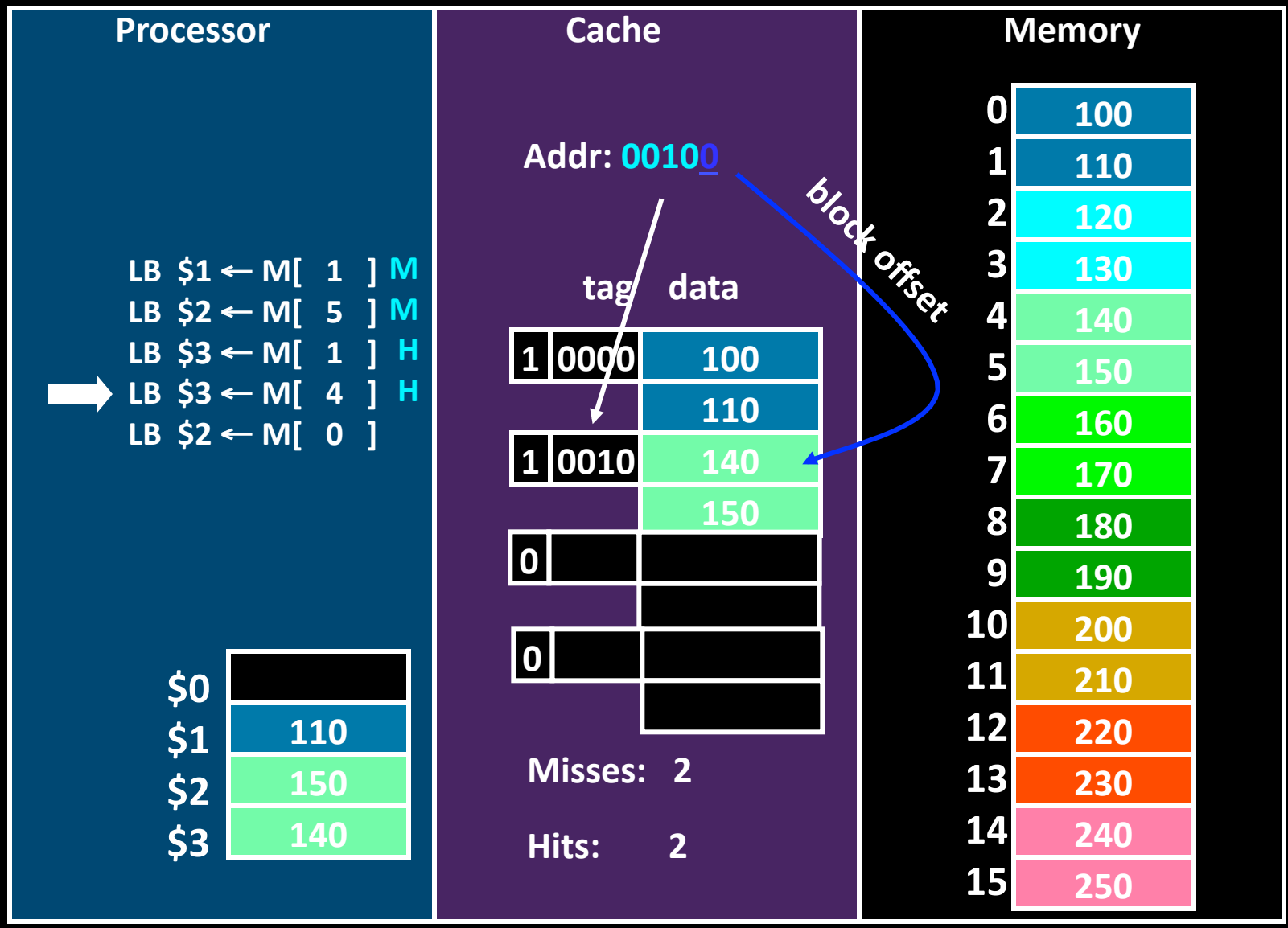
3rd Access


Processor
Cache
Memory
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1 110
$2 150
$3 110
Addr: 00001
block offset
tag data
1 0000 100
110
1 0010 140
150
0
0
Misses: 2
Hits: 1
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

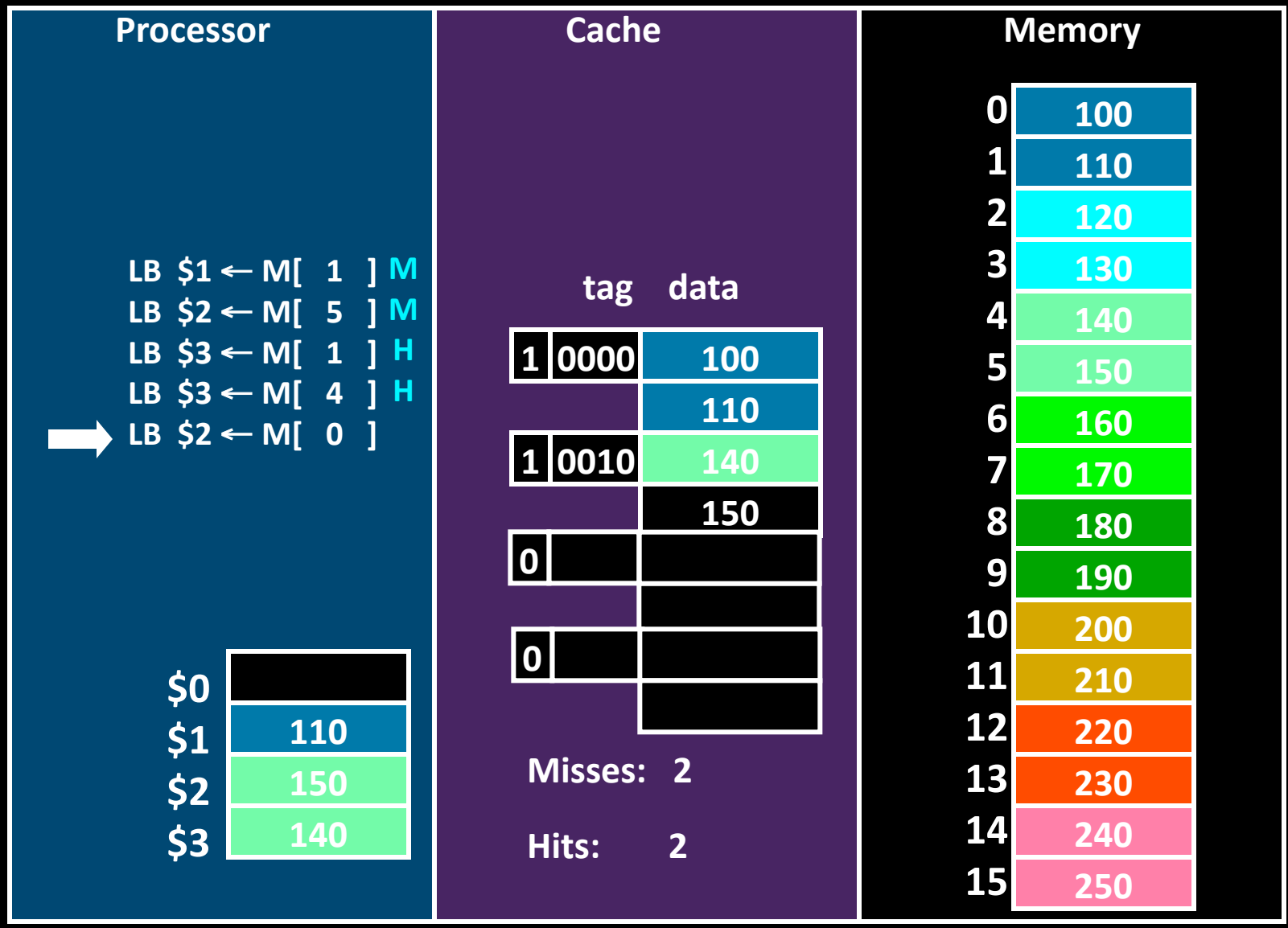
4th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1 110
$2 150
$3 110
Cache
tag data
1 0000 100
110
1 0010 140
150
0
0
Misses: 2
Hits: 1
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

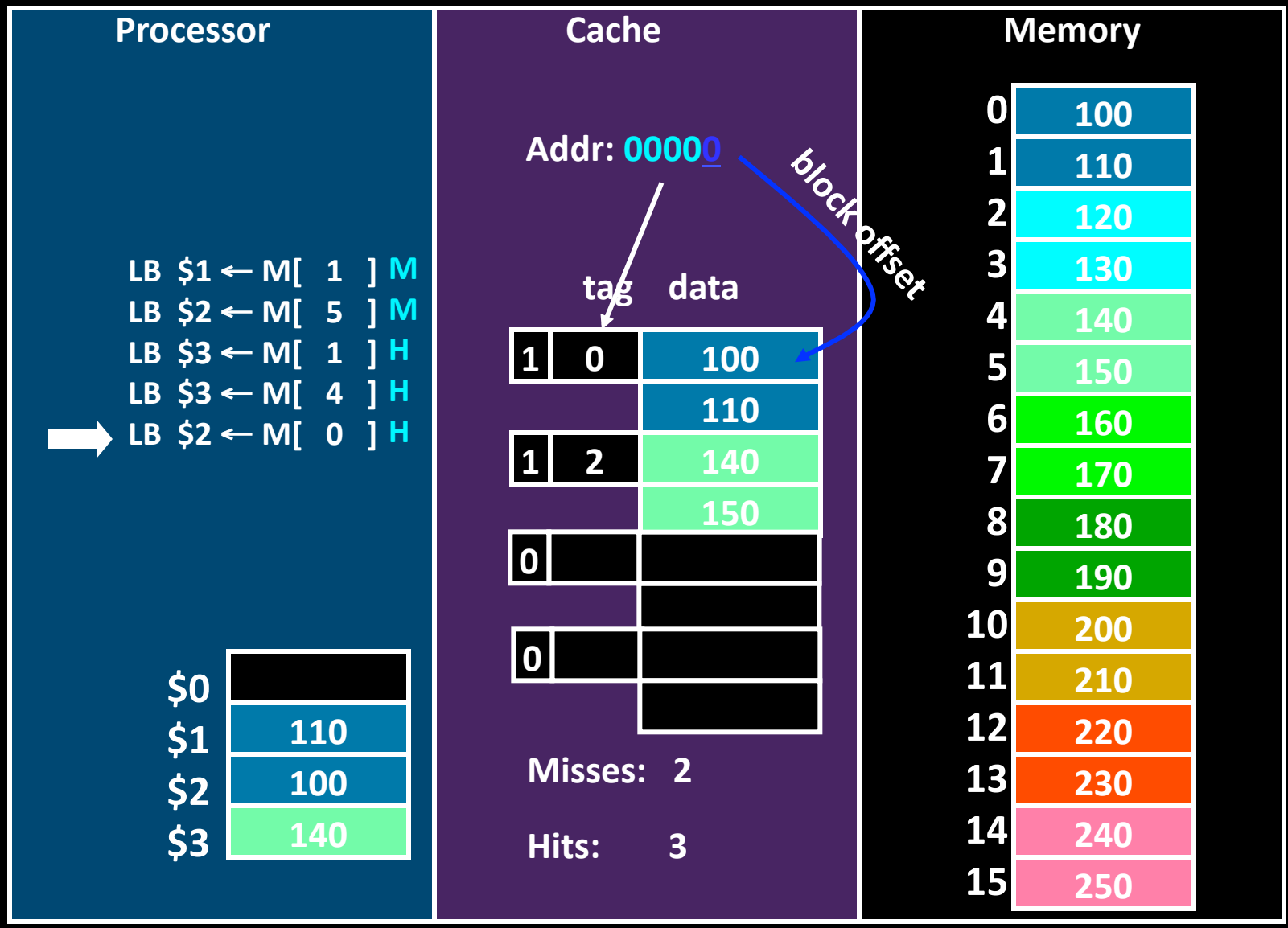
4th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0]
$0
$1 110
$2 150
$3 140
Cache
Addr: 00100
tag
data
block offset
1 0000 100
110
1 0010 140
150
0
0
Misses: 2
Hits: 2
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

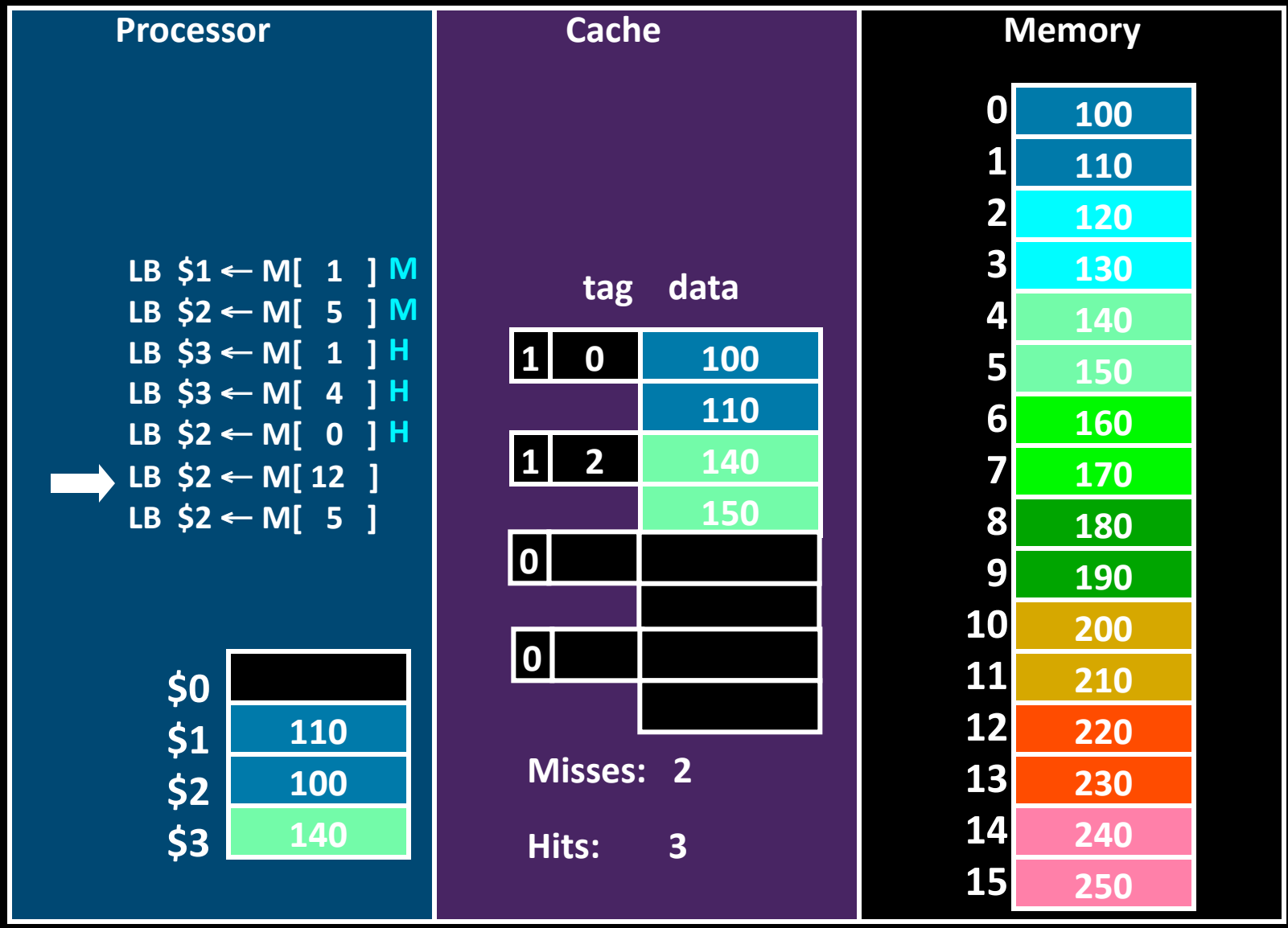
5th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0]
$0
$1 110
$2 150
$3 140
Cache
tag data
1 0000 100
110
1 0010 140
150
0
0
Misses: 2
Hits: 2
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

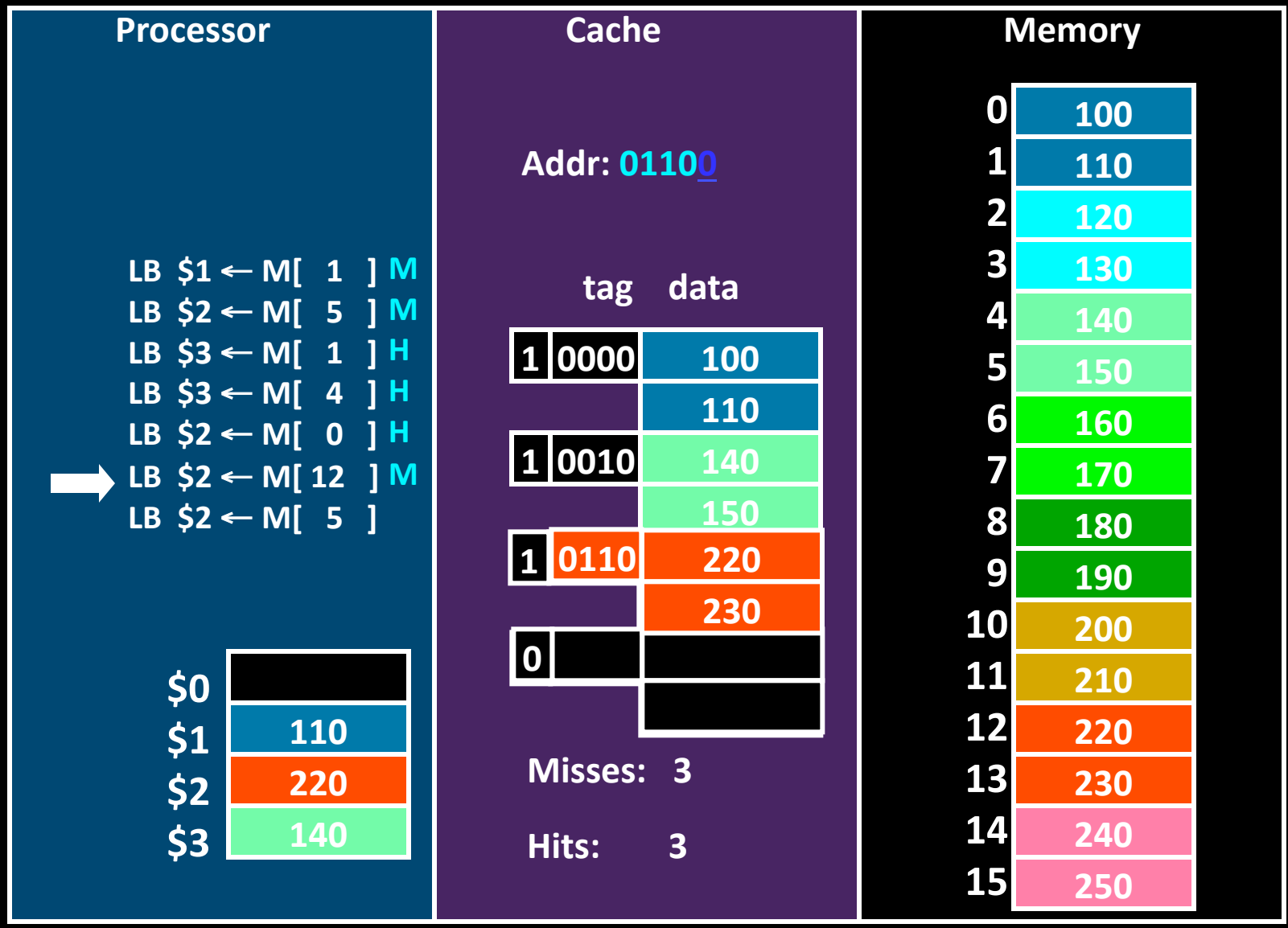
5th Access


Processor
Cache
Memory
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
$0
$1
$2
$3
110
100
140
Addr: 00000
block offset
tag
data
1 0 100
110
1 2 140
150
0
0
Misses: 2
Hits: 3
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

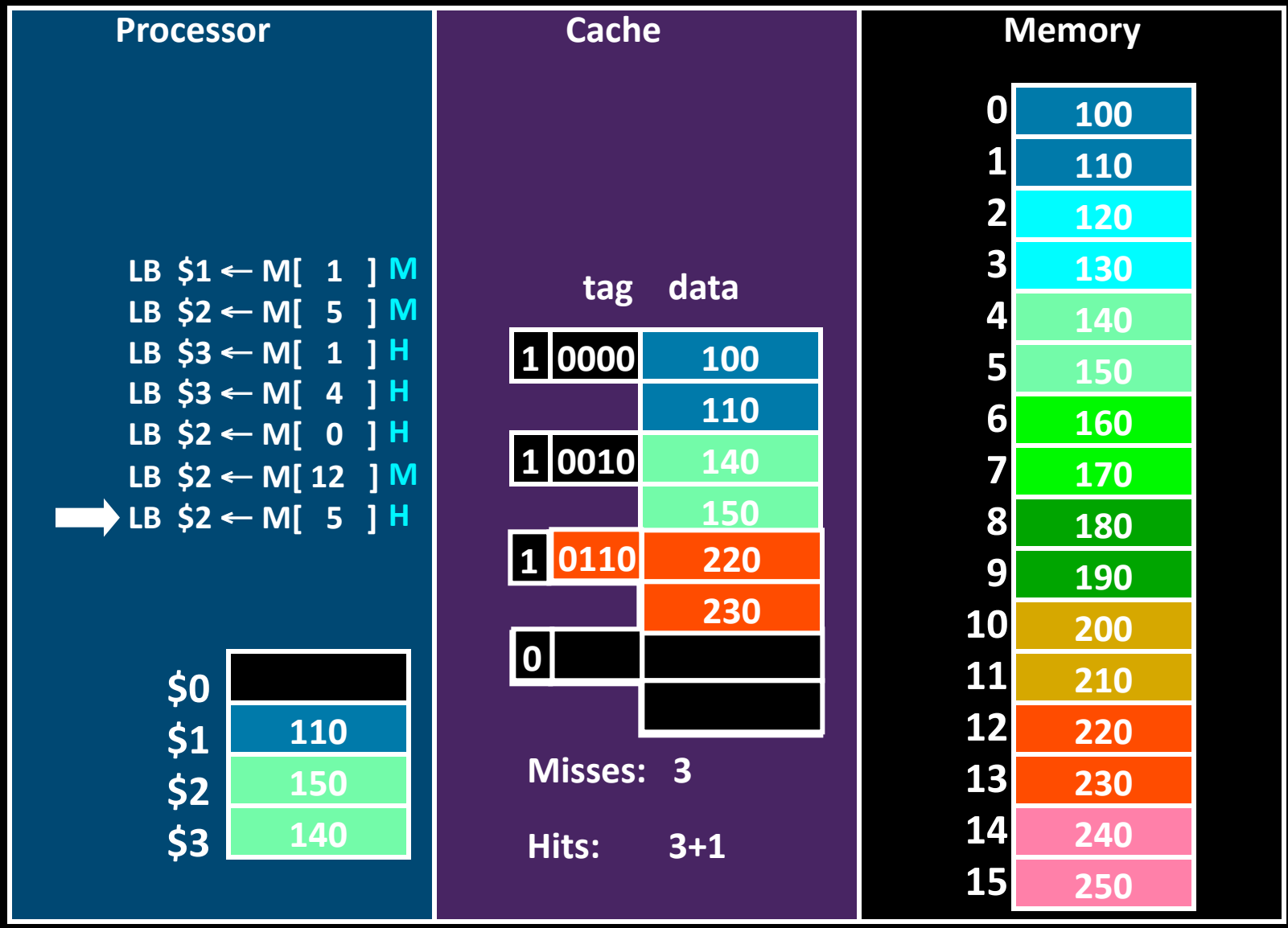
6th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[12]
LB $2 ← M[5]
$0
$1 110
$2 100
$3 140
Cache
tag data
1 0 100
110
1 2 140
150
0
0
Misses: 2
Hits: 3
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

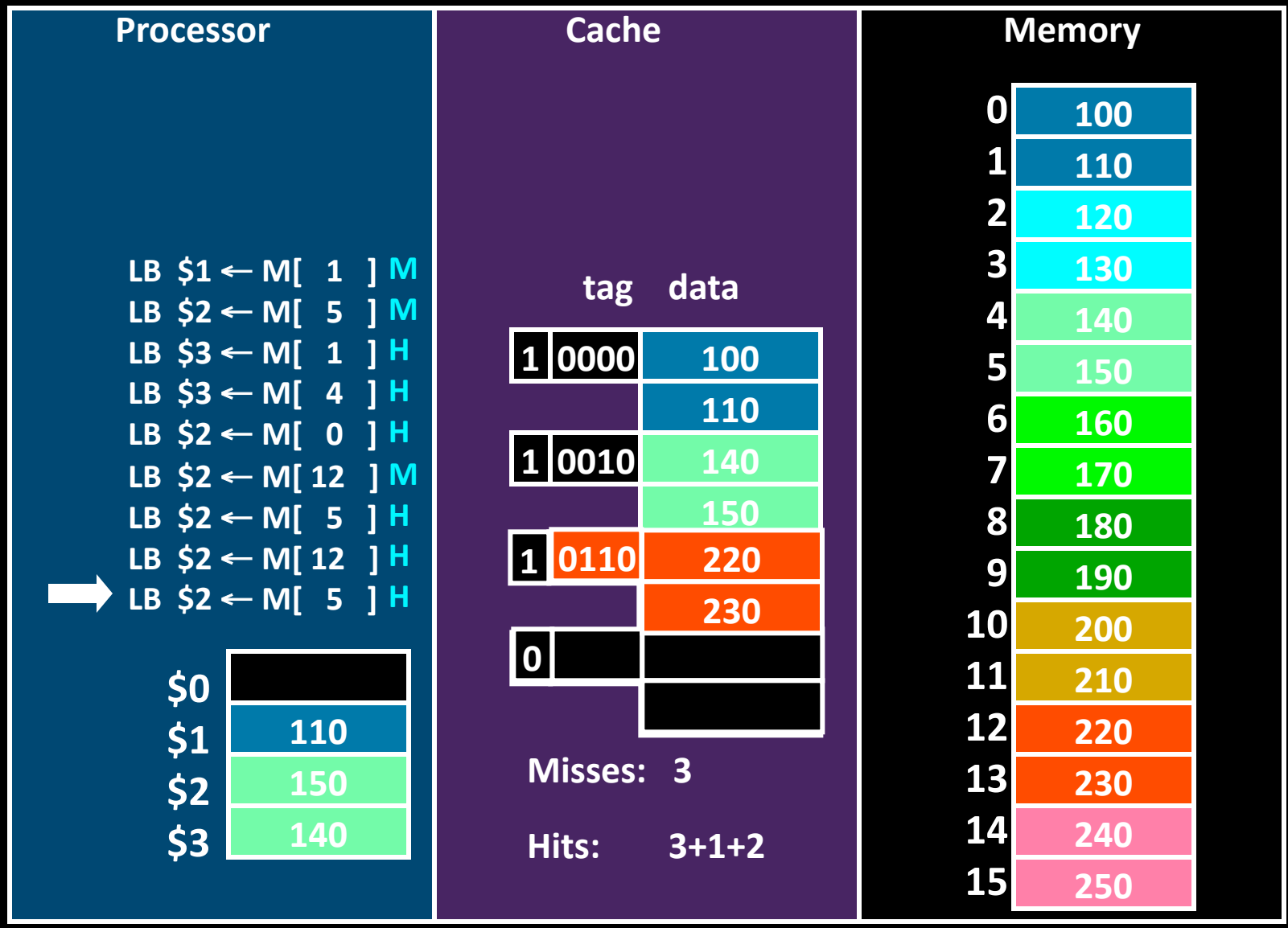
6th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[12] M
LB $2 ← M[5]
$0
$1 110
$2 220
$3 140
Cache
Addr: 01100
tag data
1 0000 100
110
1 0010 140
150
1 0110 220
230
0
Misses: 3
Hits: 3
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

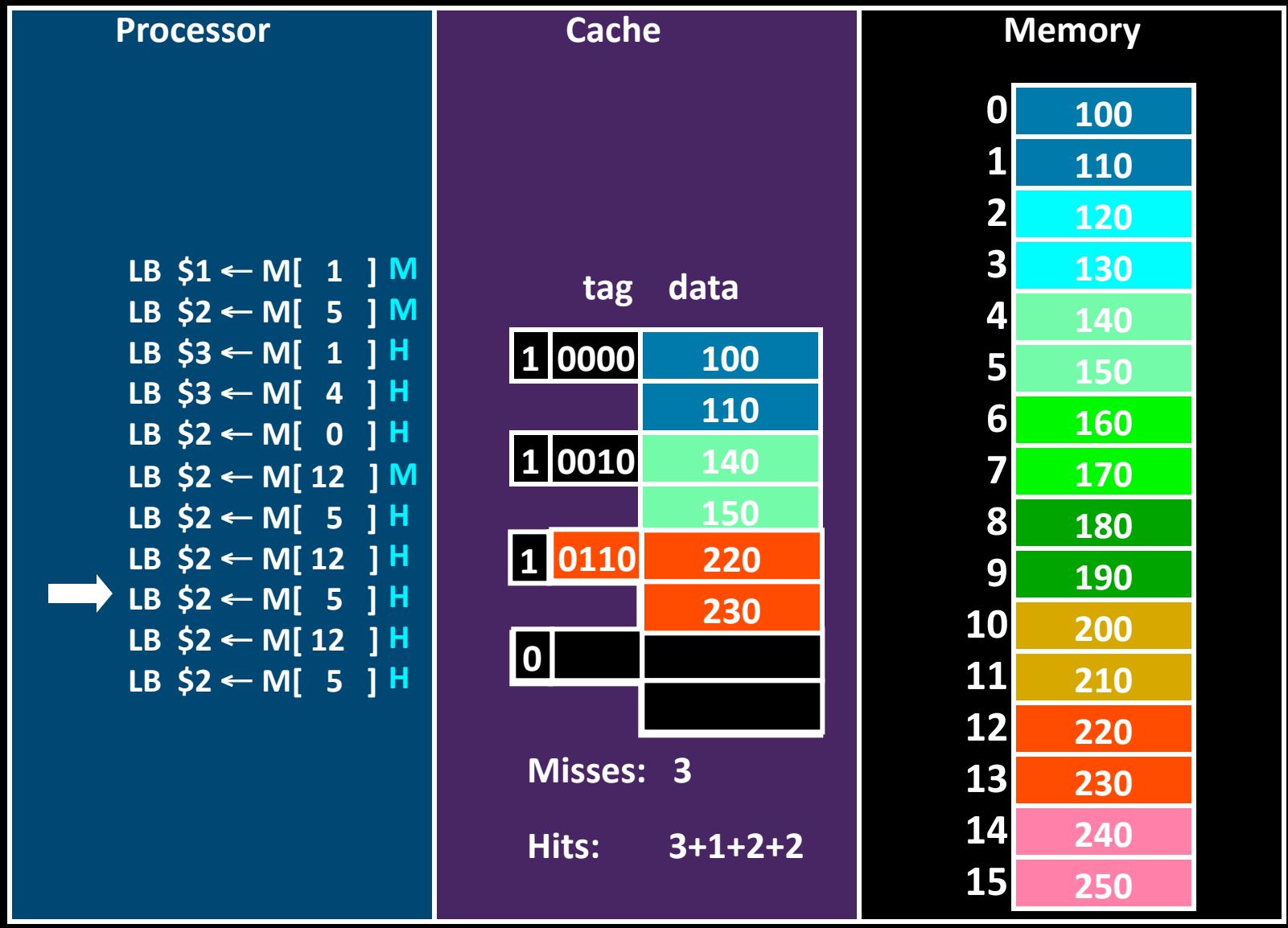
7th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[12] M
LB $2 ← M[5] H
$0
$1 110
$2 150
$3 140
Cache
tag data
1 0000 100
110
1 0010 140
150
1 0110 220
230
0
Misses: 3
Hits: 3+1
Memory
0 100
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250

8th and 9th Access



10th and 11th Access



Cache Tradeoffs

Direct Mapped		Fully Associative
+ Smaller	Tag Size	Larger –
+ Less	SRAM Overhead	More –
+ Less	Controller Logic	More –
+ Faster	Speed	Slower –
+ Less	Price	More –
+ Very	Scalability	Not Very –
– Lots	# of conflict misses	Zero +
– Low	Hit rate	High +
– Common	Pathological Cases?	?

Compromise

Set-associative cache

Like a direct-mapped cache

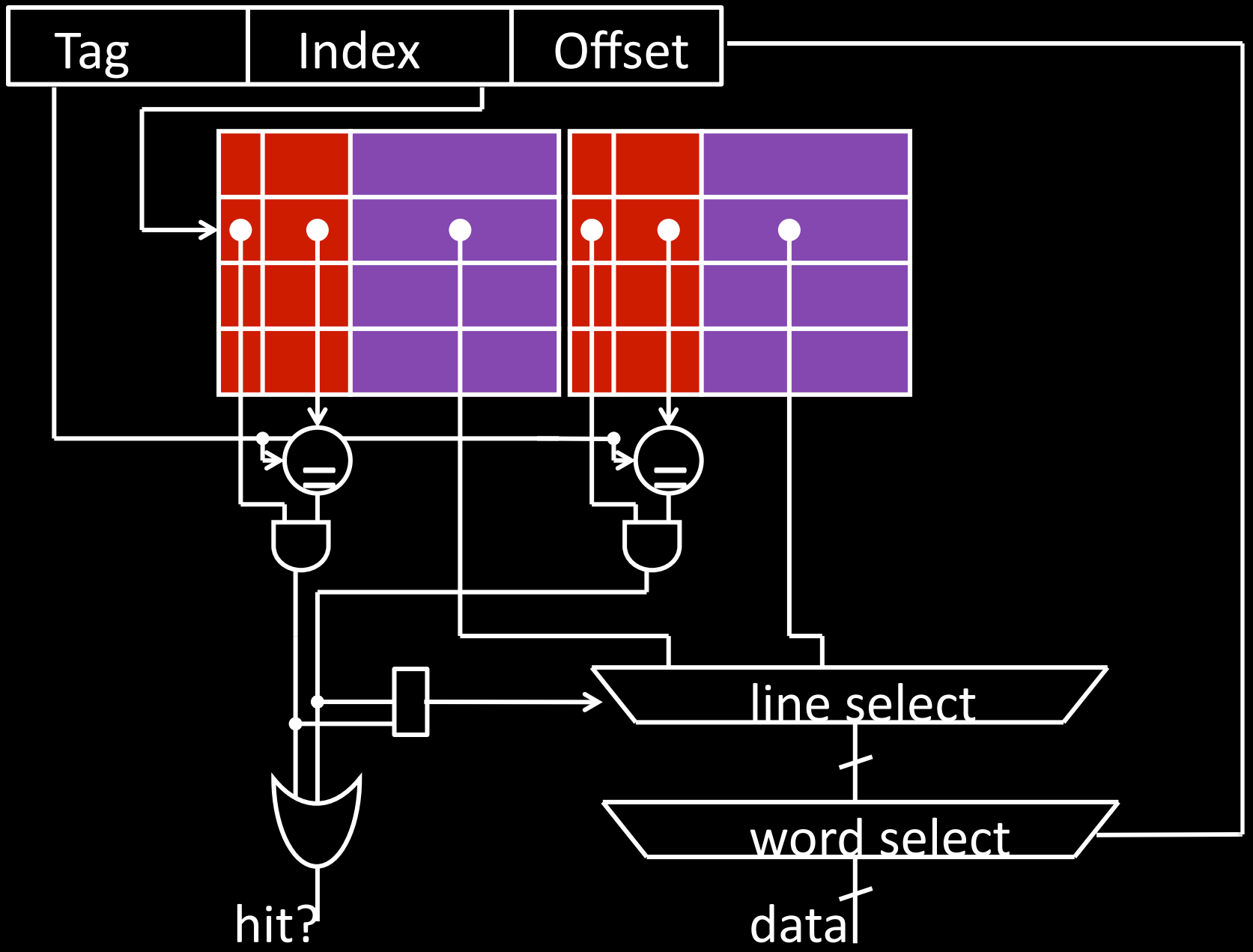
- Index into a location
- Fast

Like a fully-associative cache

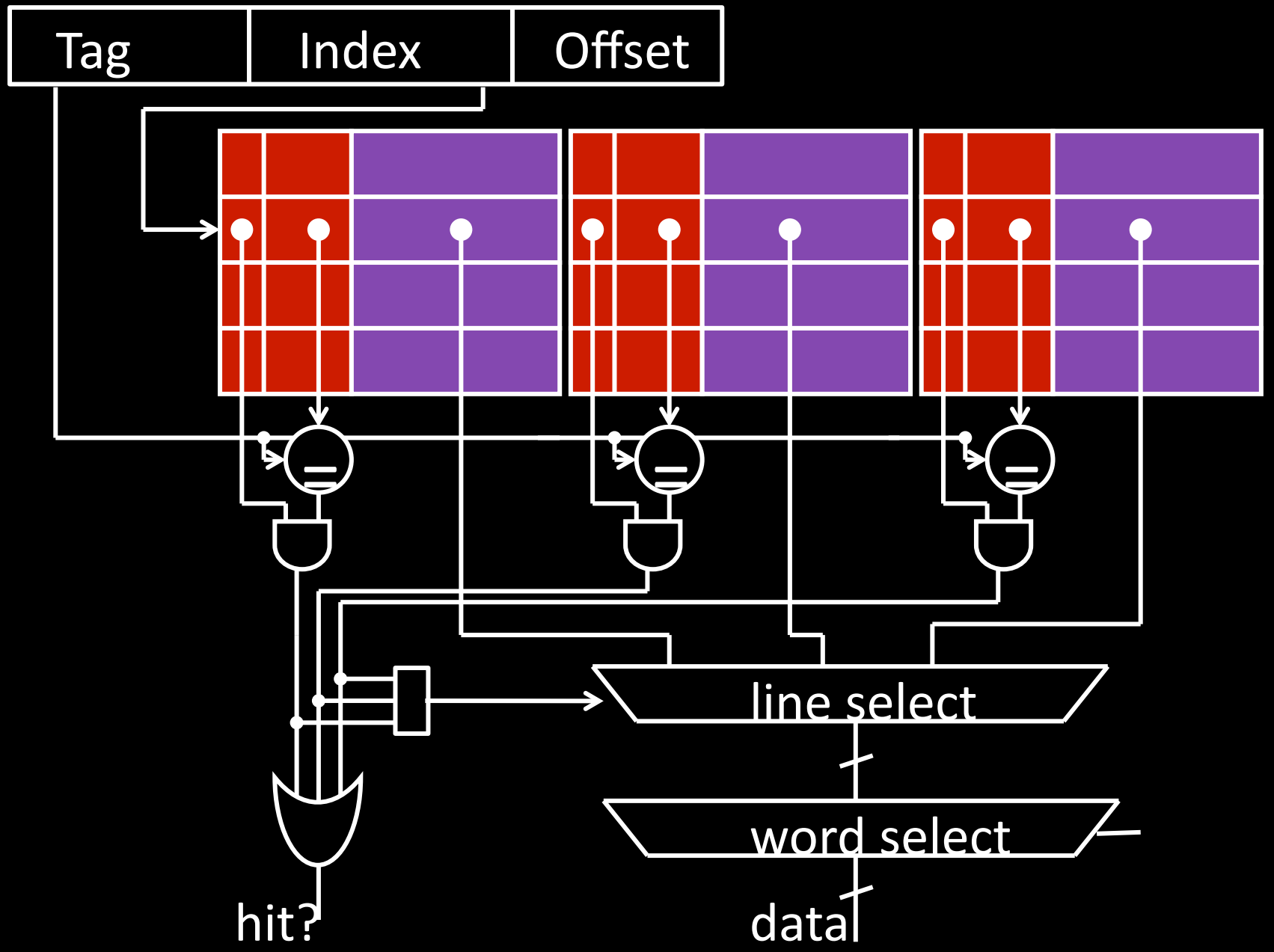
- Can store multiple entries
  - decreases conflicts
- Search in each element

n-way set assoc means n possible locations

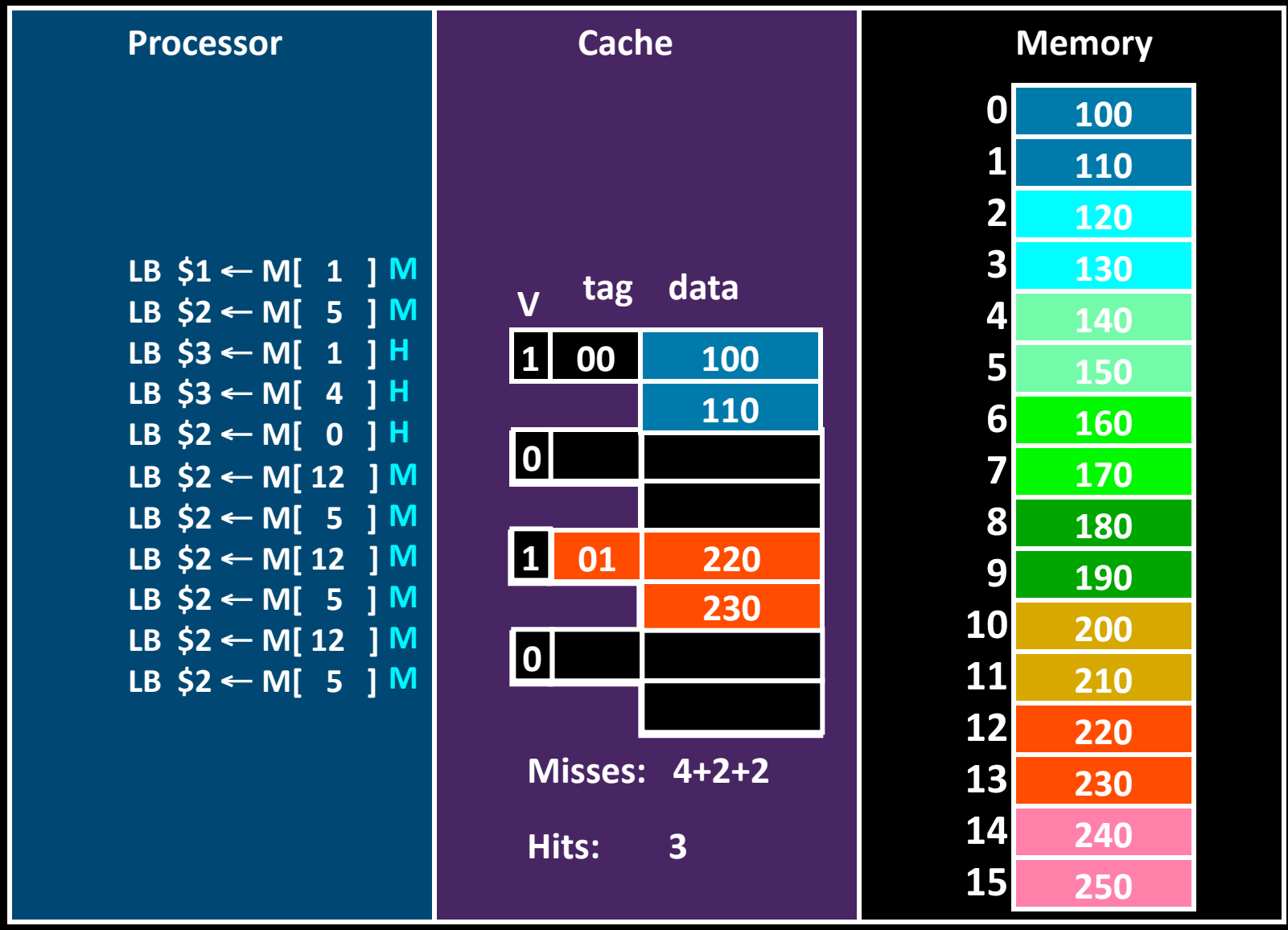
2-Way Set Associative Cache (Reading)



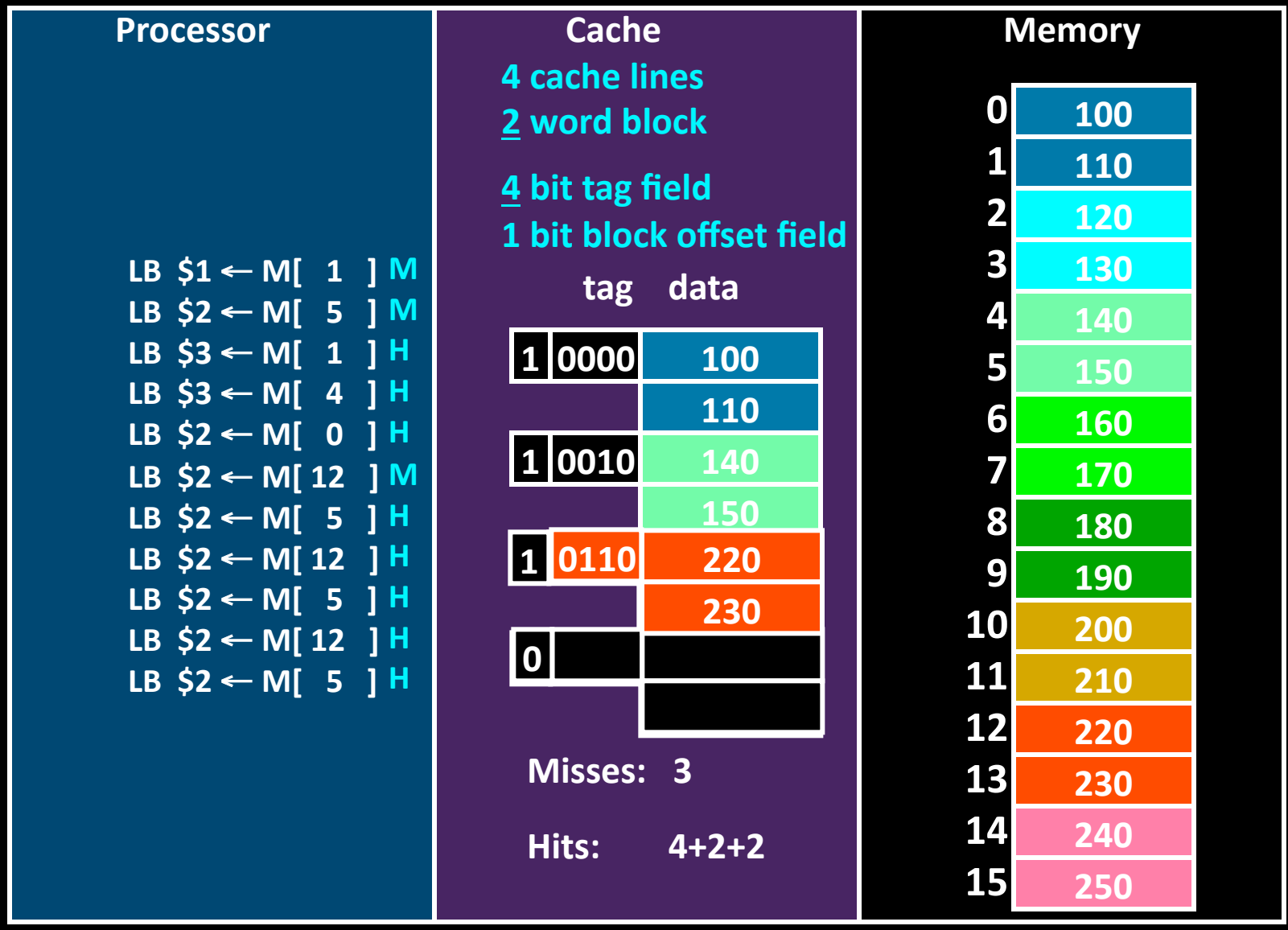
3-Way Set Associative Cache (Reading)



Comparison: Direct Mapped



Comparison: Fully Associative



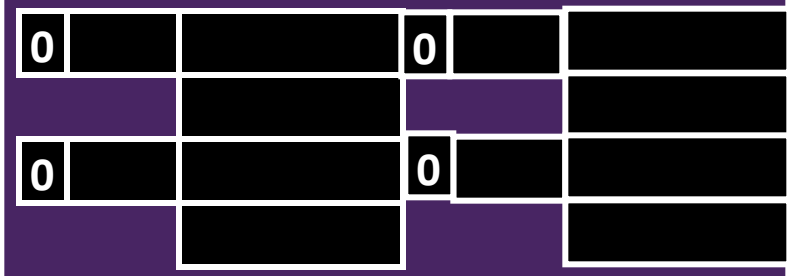
Comparison: 2 Way Set Assoc

Processor

LB $1 ← M[1]
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
LB $2 ← M[12]
LB $2 ← M[5]
LB $2 ← M[12]
LB $2 ← M[5]
LB $2 ← M[12]
LB $2 ← M[5]

Cache

2 sets
2 word block
3 bit tag field
1 bit set index field
1 bit block offset field



Misses:

Hits:

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

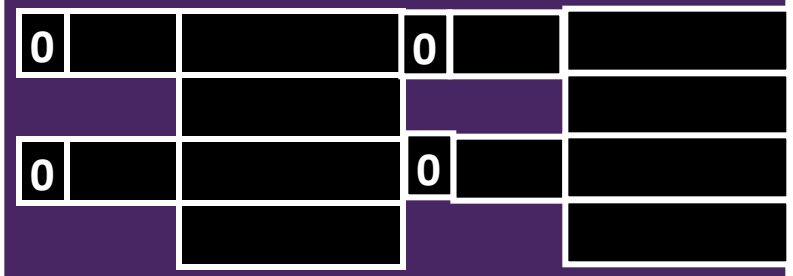
Comparison: 2 Way Set Assoc

Processor

LB	$1 ← M[1]	M
LB	$2 ← M[5]	M
LB	$3 ← M[1]	H
LB	$3 ← M[4]	H
LB	$2 ← M[0]	H
LB	$2 ← M[12]	M
LB	$2 ← M[5]	M
LB	$2 ← M[12]	H
LB	$2 ← M[5]	H
LB	$2 ← M[12]	H
LB	$2 ← M[5]	H

Cache

2 sets
2 word block
3 bit tag field
1 bit set index field
1 bit block offset field



Misses: 4

Hits: 7

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Summary on Cache Organization

Direct Mapped → simpler, low hit rate

Fully Associative → higher hit cost, higher hit rate

N-way Set Associative → middleground

Misses

Cache misses: classification

Cold (aka Compulsory)

- The line is being referenced for the first time
  - Block size can help

Capacity

- The line was evicted because the cache was too small
- i.e. the *working set* of program is larger than the cache

Conflict

- The line was evicted because of another access whose index conflicted
  - Not an issue with fully associative

Cache Performance

Average Memory Access Time (AMAT)

Cache Performance (very simplified):

L1 (SRAM): 512 x 64 byte cache lines, direct mapped

Data cost: 3 cycle per word access

Lookup cost: 2 cycle

Mem (DRAM): 4GB

Data cost: 50 cycle plus 3 cycle per word

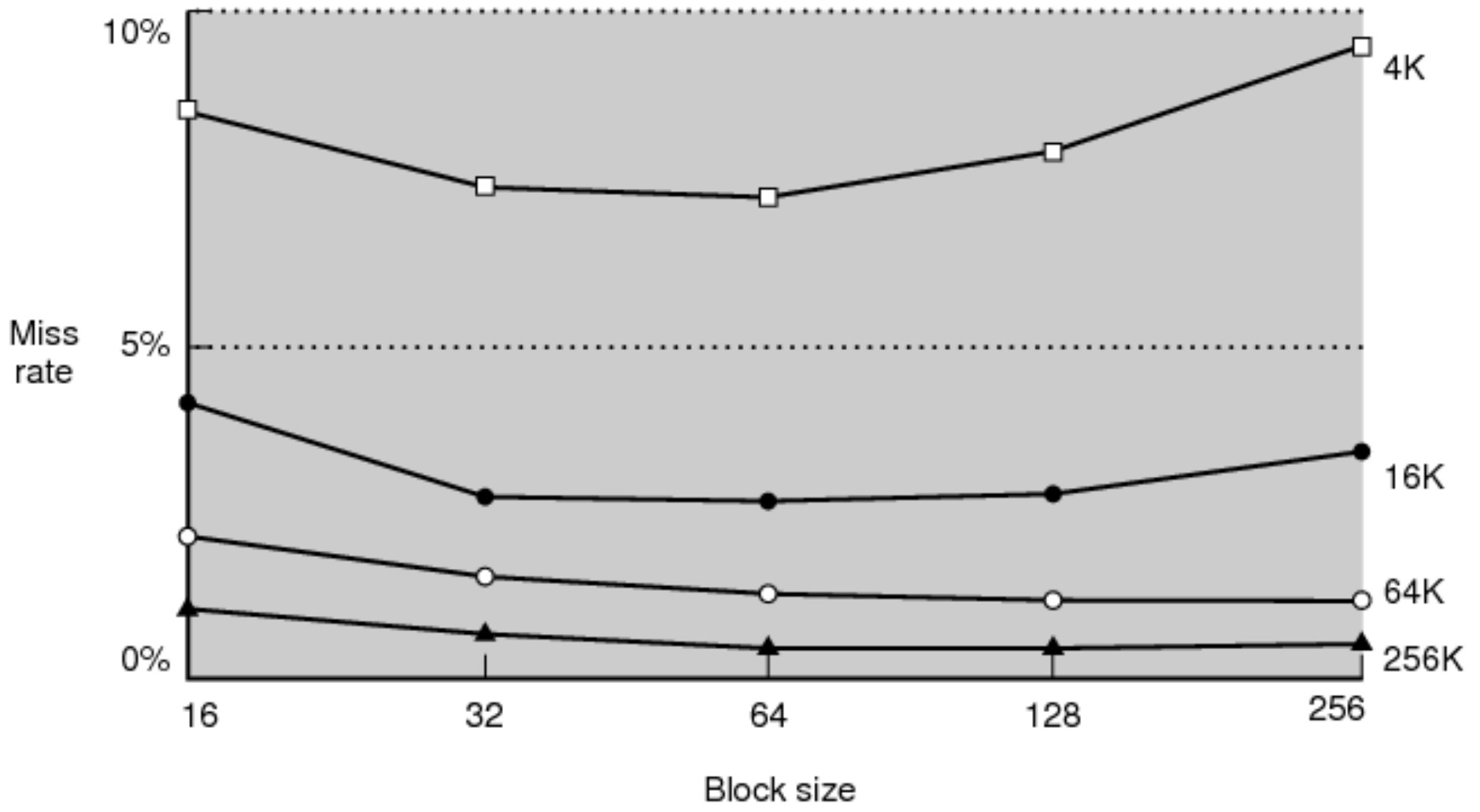
Performance depends on:

Access time for hit, hit rate, miss penalty

Basic Cache Organization

Q: How to decide block size?

Experimental Results



Tradeoffs

For a given total cache size,

larger block sizes mean….

- fewer lines
- so fewer tags, less overhead
- and fewer cold misses (within-block “prefetching”)

But also…

- fewer blocks available (for scattered accesses!)
- so more conflicts
- and larger miss penalty (time to fetch block)

Summary

Caching assumptions

- small working set: 90/10 rule
- can predict future: spatial & temporal locality

Benefits

- big & fast memory built from (big & slow) + (small & fast)

Tradeoffs:

associativity, line size, hit cost, miss penalty, hit rate

- Fully Associative → higher hit cost, higher hit rate
- Larger block size → lower hit cost, higher miss penalty