Caches and Memory

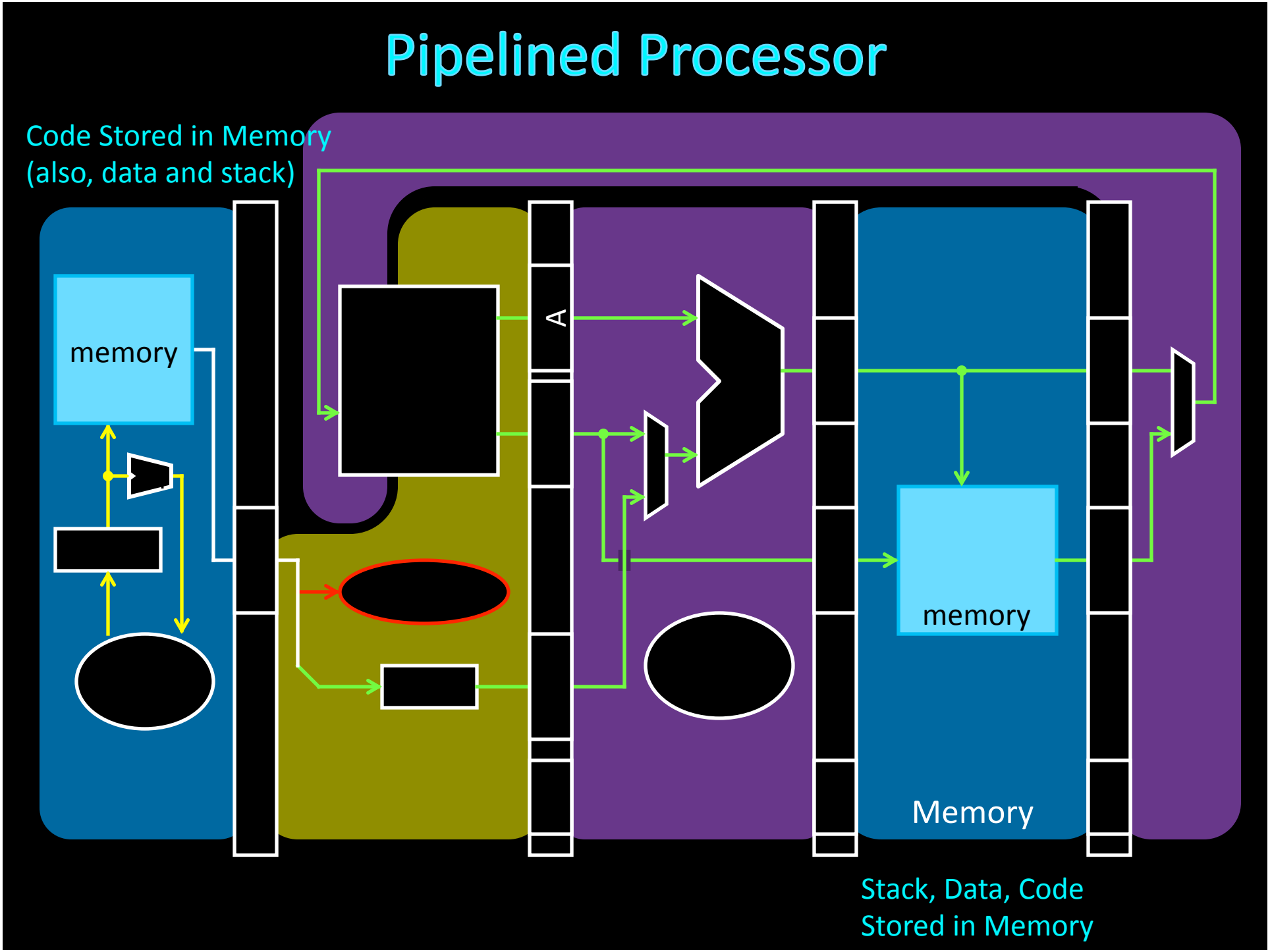
CS 3410, Spring 2014

Computer Science

Cornell University

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

Pipelined Processor



Problem

Main memory is very very slow

Remember:

SRAM

- 6-8 transistors, no refresh, fast

DRAM

- 1 transistor, denser, cheaper/bit, needs refresh

Problem

Main memory is very very slow

CPU clock rates ~0.33ns – 2ns (3GHz-500MHz)

Memory technology	Access time in nanosecs (ns)	Access time in cycles
SRAM (on chip)	0.5-2.5 ns	1-3 cycles
SRAM (off chip)	1.5-30 ns	5-15 cycles
DRAM	50-70 ns	150-200 cycles
SSD (Flash)	5k-50k ns	Tens of thousands
Disk	5M-20M ns	Millions

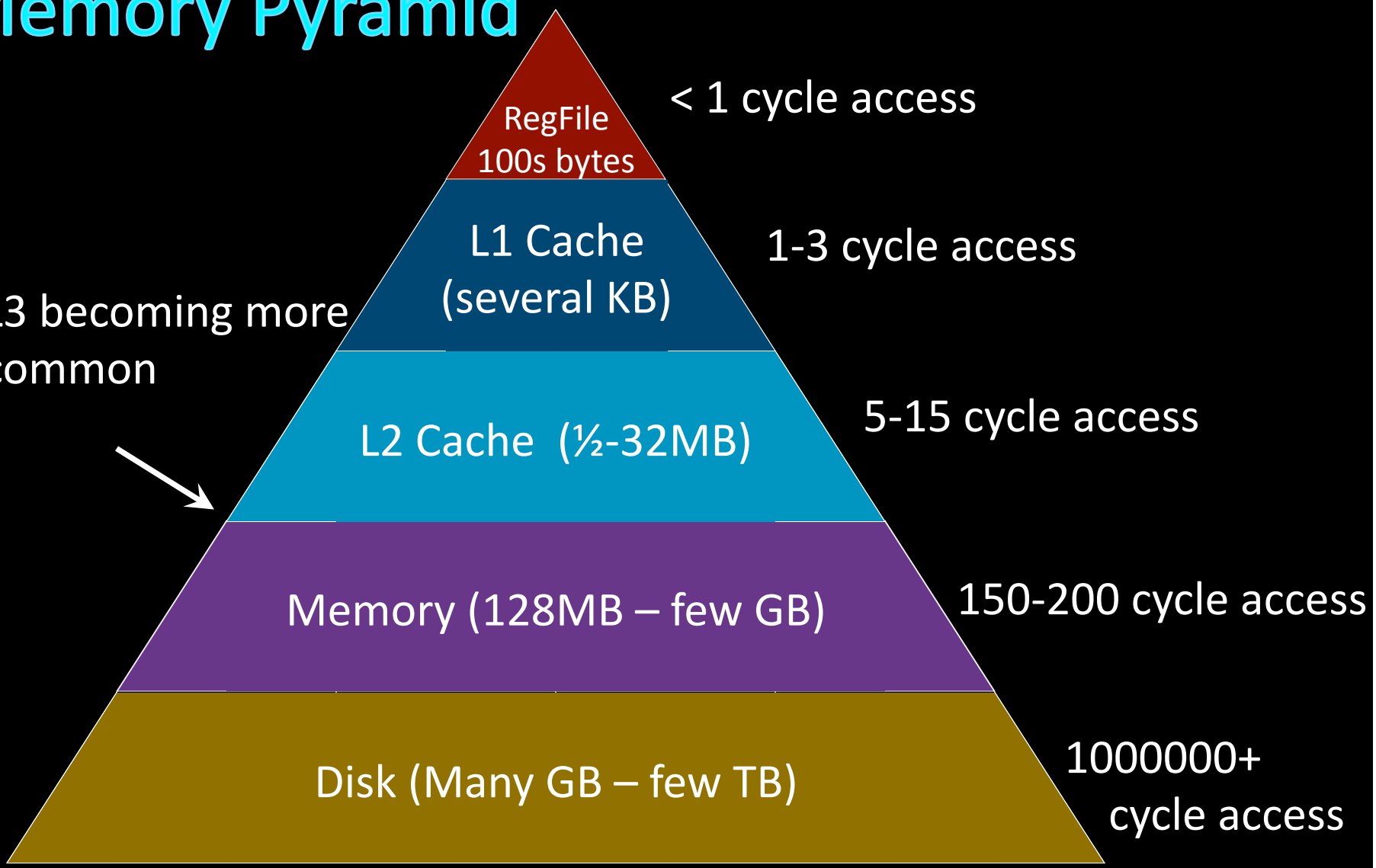
Problem

Main memory is very very slow

CPU clock rates ~0.33ns – 2ns (3GHz-500MHz)

Memory technology	Access time in nanosecs (ns)	Access time in cycles	$ per GIB in 2012	Capacity
SRAM (on chip)	0.5-2.5 ns	1-3 cycles		256 KB
SRAM (off chip)	1.5-30 ns	5-15 cycles	$4k	32 MB
DRAM	50-70 ns	150-200 cycles	$10-$20	8 GB
SSD (Flash)	5k-50k ns	Tens of thousands	$0.75-$1	512 GB
Disk	5M-20M ns	Millions	$0.05-$0.1	4 TB

Memory Pyramid



These are rough numbers: mileage may vary for latest/greatest
Caches usually made of SRAM

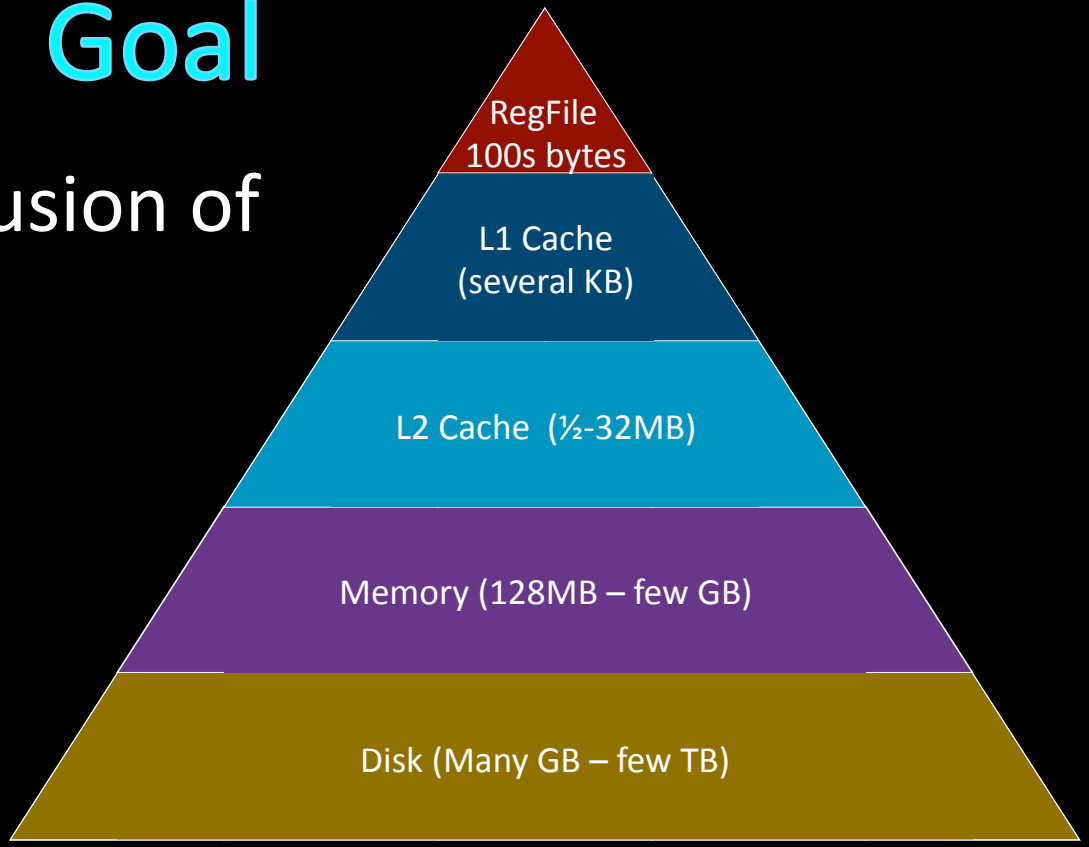
Goal

Can we create an illusion of

cheap,

large and

fast memory?



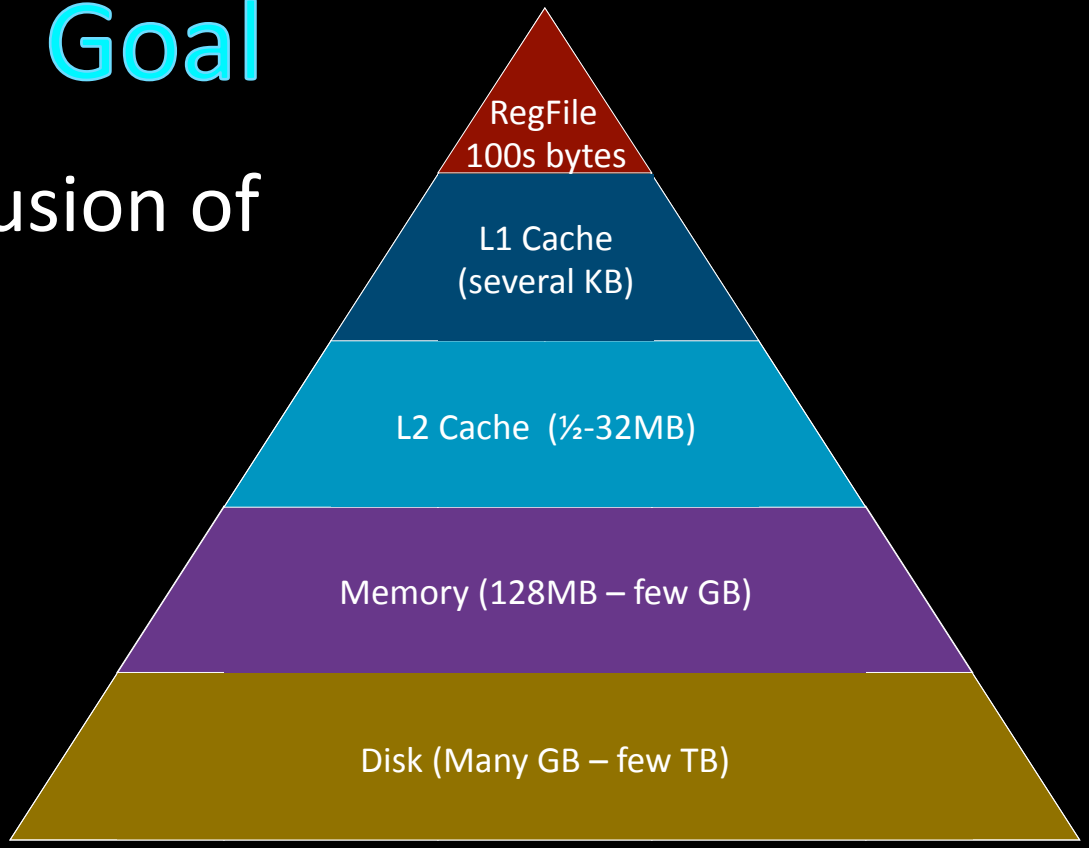
Goal

Can we create an illusion of

cheap,

large and

fast memory?



Yes, using caches

and assuming temporal and spatial locality

By the end of the cache lectures…

MacBook Pro

Retina, Mid 2012

Processor 2.7 GHz Intel Core i7

Memory 16 GB 1600 MHz DDR3

Graphics NVIDIA GeForce GT 650M 1024 MB

Serial Number C02J70TTDKQ5

Software OS X 10.9.2 (13C64)

Model Name:	MacBook Pro
Model Identifier:	MacBookPro10,1
Processor Name:	Intel Core i7
Processor Speed:	2.7 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	8 MB
Memory:	16 GB
Boot ROM Version:	MBP101.00EE.B02
SMC Version (system):	2.3f36
Serial Number (system):	C02J70TTDKQ5
Hardware UUID:	F588E08C-60BF-5B35-A087-07714C2B2D11

- 32 KB data + 32 KB instruction L1 cache (3 clocks) and 256 KB L2 cache (8 clocks) per core.
- Shared L3 cache includes the processor graphics (LGA 1155).
- 64-byte cache line size.

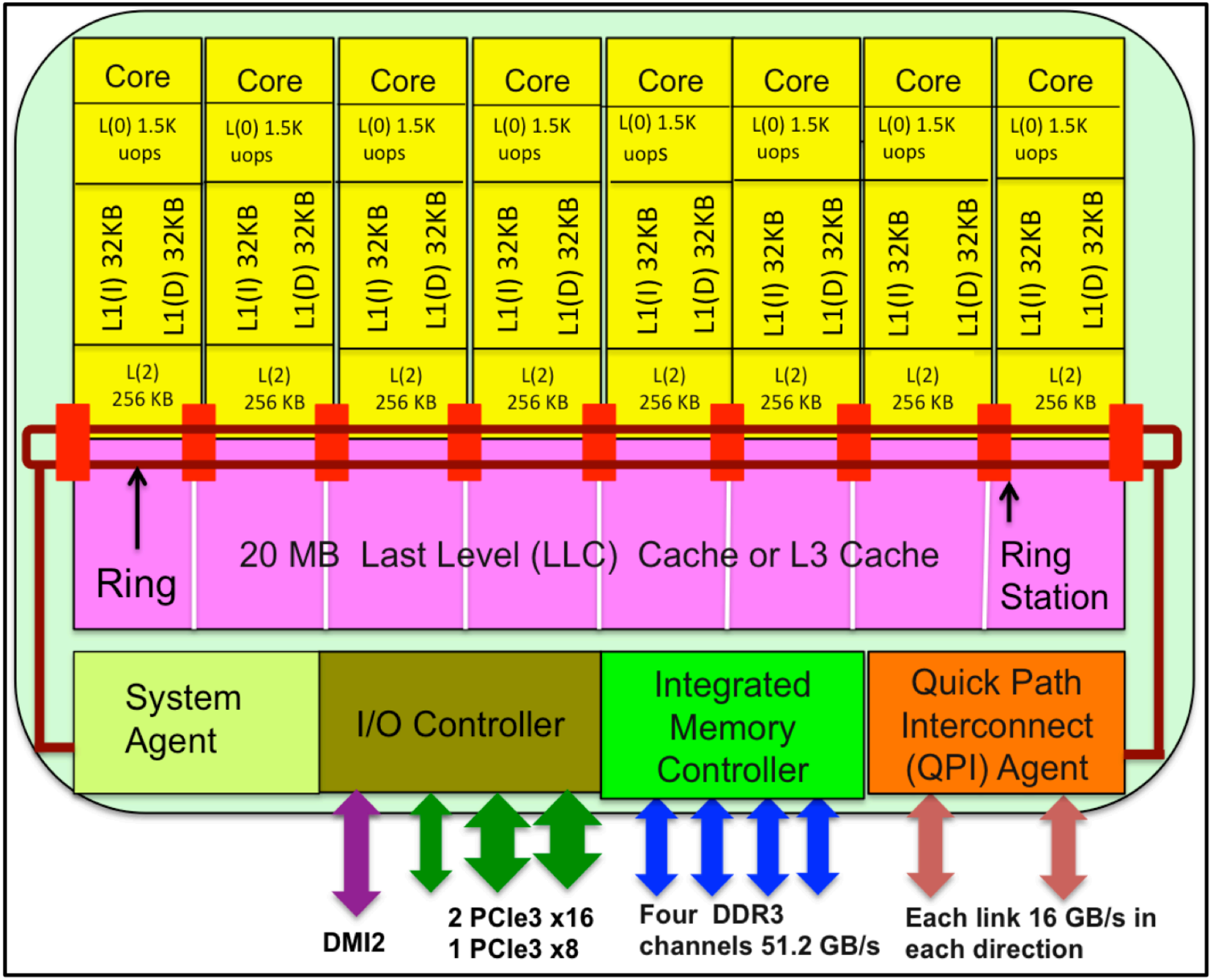
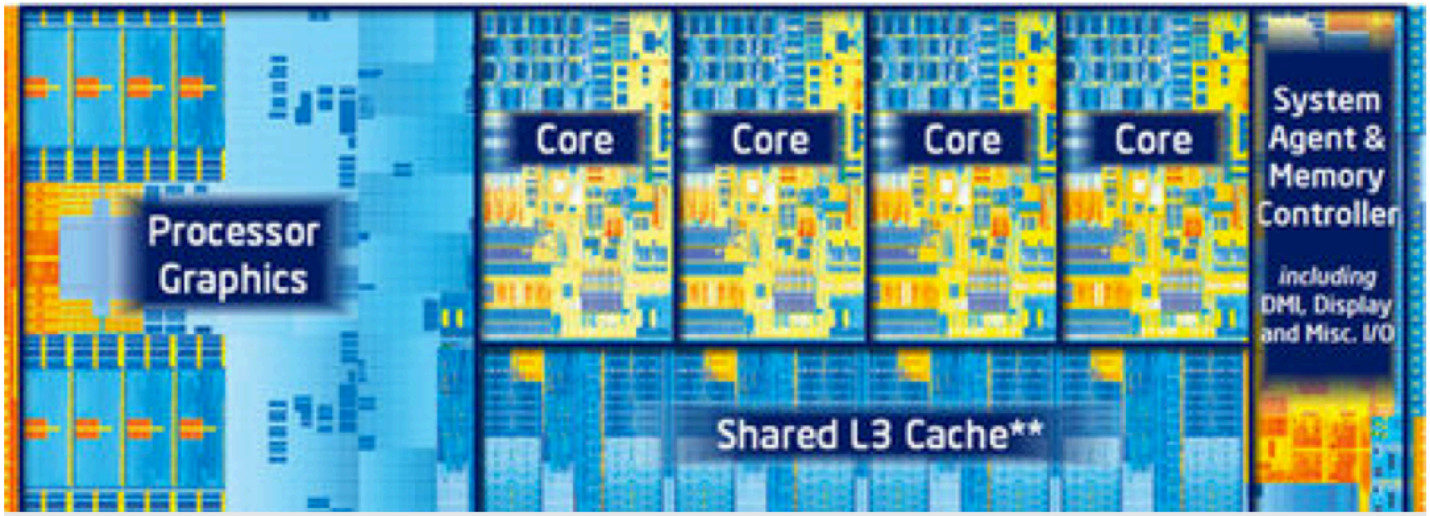
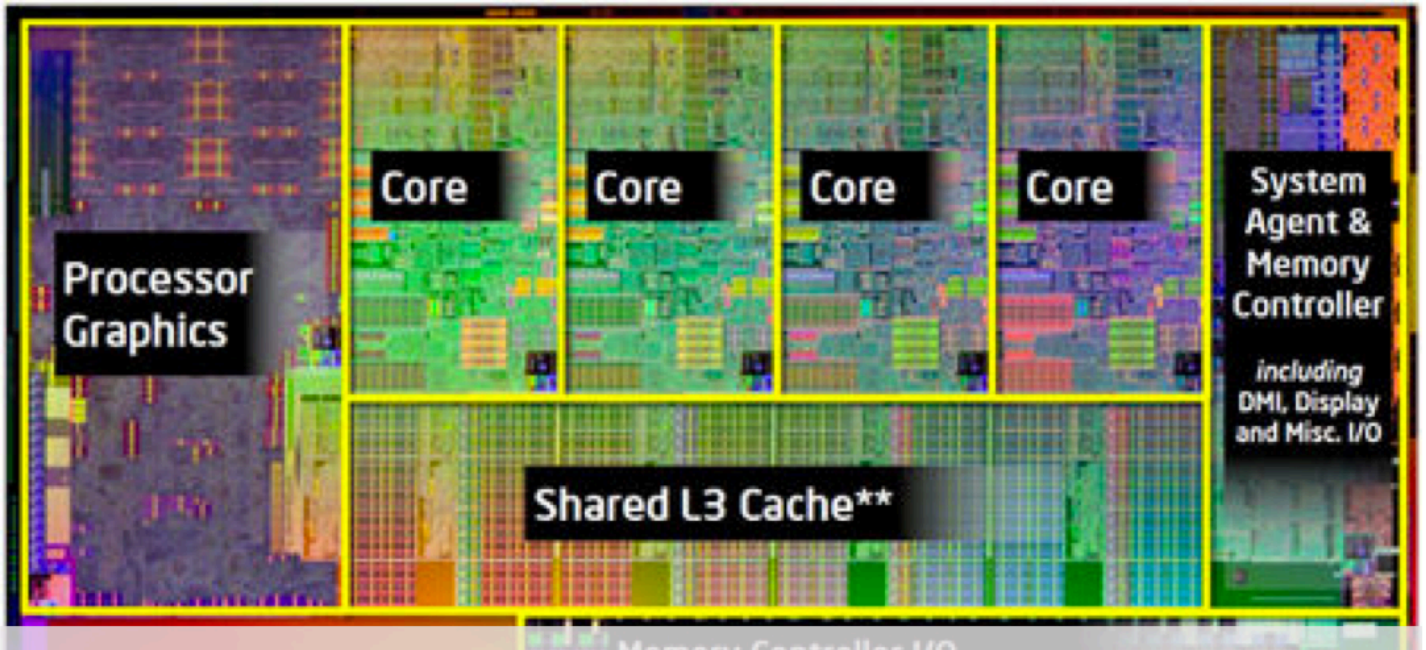


Figure 1. Schematic diagram of a Sandy Bridge processor.



Ivy Bridge: 1.4 billion transistors; 160 square millimeters



Sandy Bridge: 995 million transistors; 216 square millimeters

Today

Caches vs memory vs tertiary storage

- Tradeoffs

Cache organization

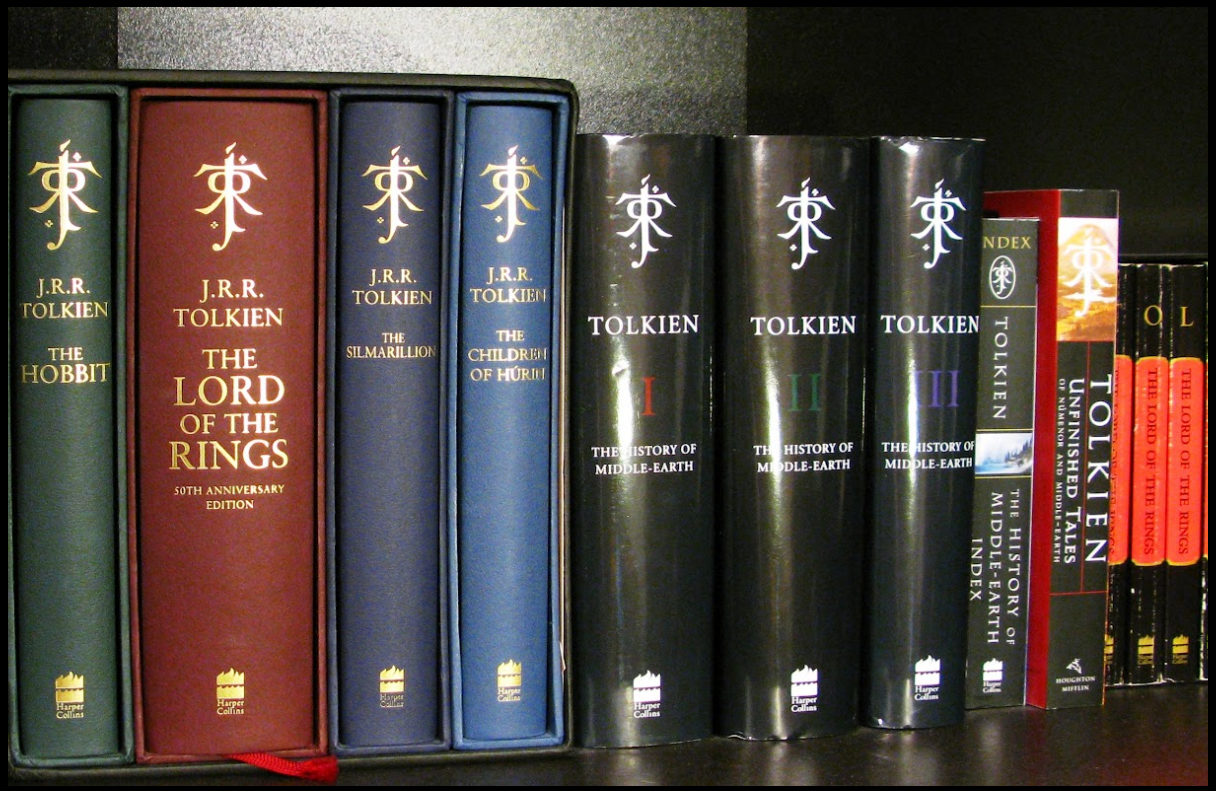
- Direct Mapped
- Fully Associative
- N-way set associative

Caching Questions

- How does a cache work? How fast? How big?

Library analogy

Writing a paper on Beren and Lúthien



Book Analogy

- Pick a small set of books; not entire shelf
- Spend time on small set of chapters

Book Analogy

- Pick a small set of books; not entire shelf
- Spend time on small set of chapters
- Sometimes get other books as well
  - Norse mythology, Tolkien biography
- Your desk: out of space
  - Replace less useful books with new ones

Book Analogy

- Pick a small set of books; not entire shelf
  - Cache vs. main memory
  - Working set (the subset in use)
- Spend time on small set of chapters
  - Cache hit
  - Locality of access: temporal and spatial
- Sometimes go to other books
  - Cache may not have data (cache miss)
- Shelf out of space
  - Cache eviction policy

Forms of locality

```
int n = 4;
int k[] = { 3, 14, 0, 10 };

int fib(int i) {
   if (i <= 2) return i;
   else return fib(i-1)+fib(i-2);
}

int main(int ac, char **av) {
   for (int i = 0; i < n; i++) {
      printi(fib(k[i]));
      prints("\n");
   }
}
```

Temporal Locality

Spatial Locality

Insight for Caches

If Mem[x] was accessed *recently*...

... then Mem[x] is likely to be accessed *soon*

- Exploit temporal locality:
  - Put recently accessed Mem[x] higher in memory hierarchy since it will likely be accessed again soon

... then Mem[$x \pm \varepsilon$] is likely to be accessed *soon*

- Exploit spatial locality:
  - Put **entire block** containing Mem[x] and surrounding addresses higher in memory hierarchy since nearby address will likely be accessed

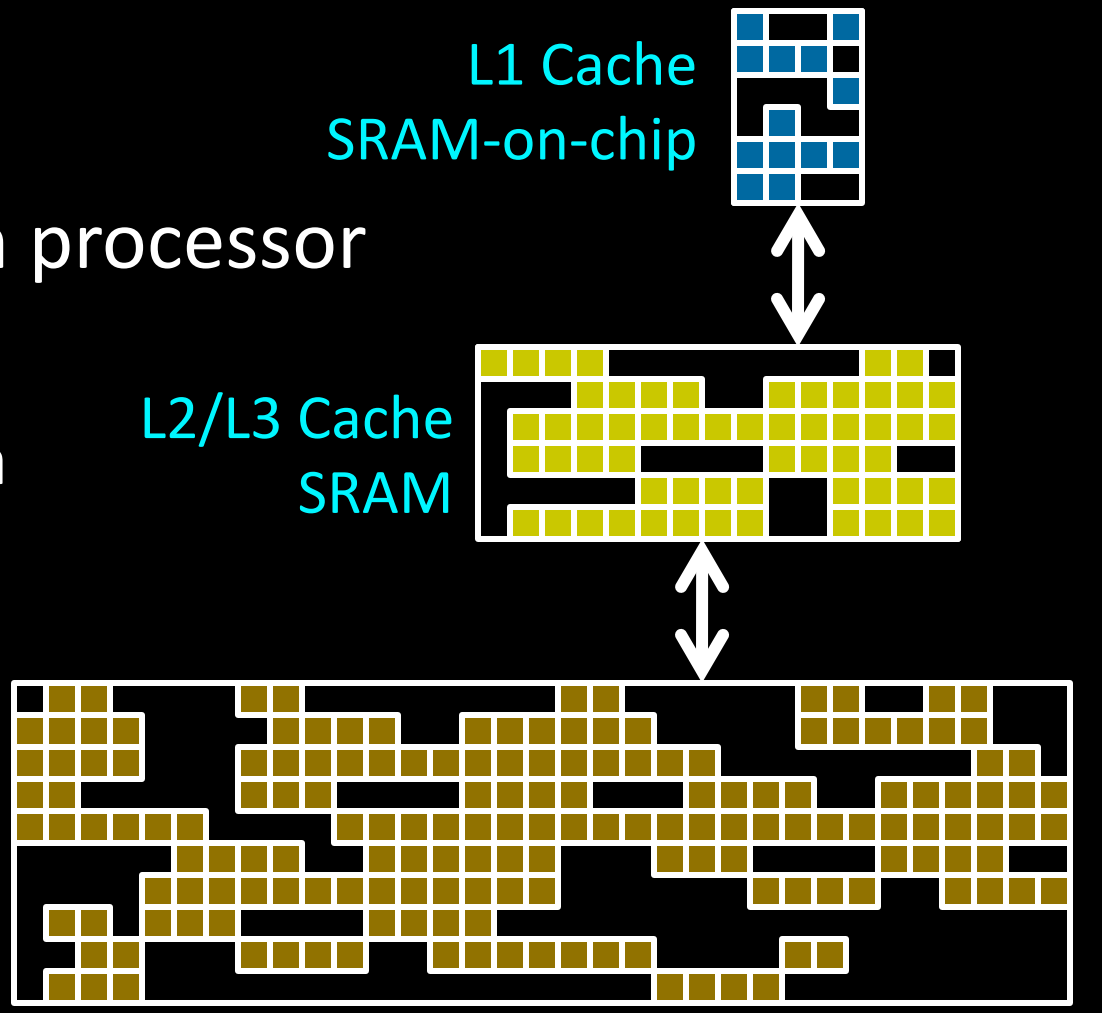
Memory Hierarchy

Memory closer to processor

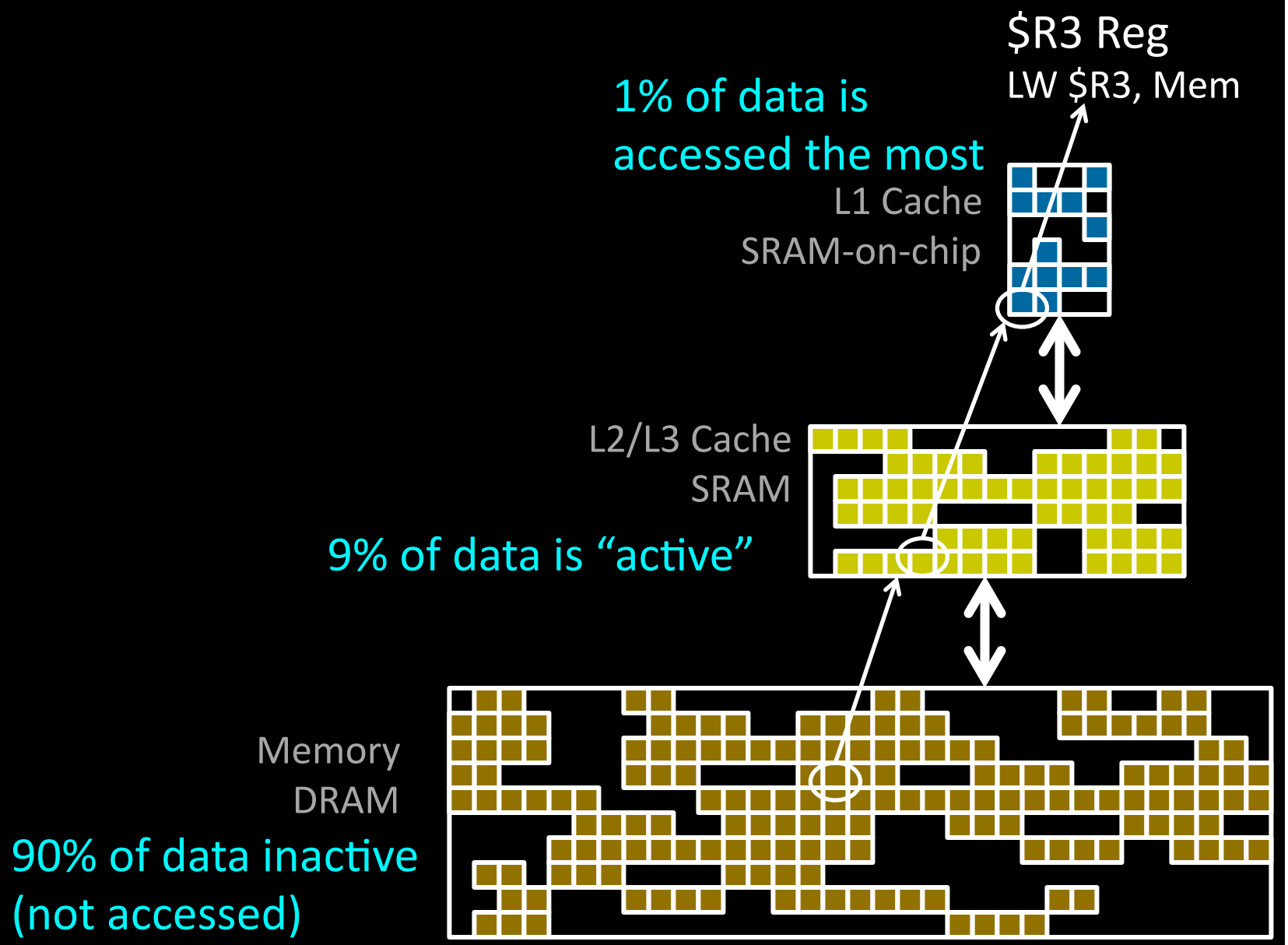
- small & fast
- stores active data

Memory farther from processor

- big & slow
- stores inactive data



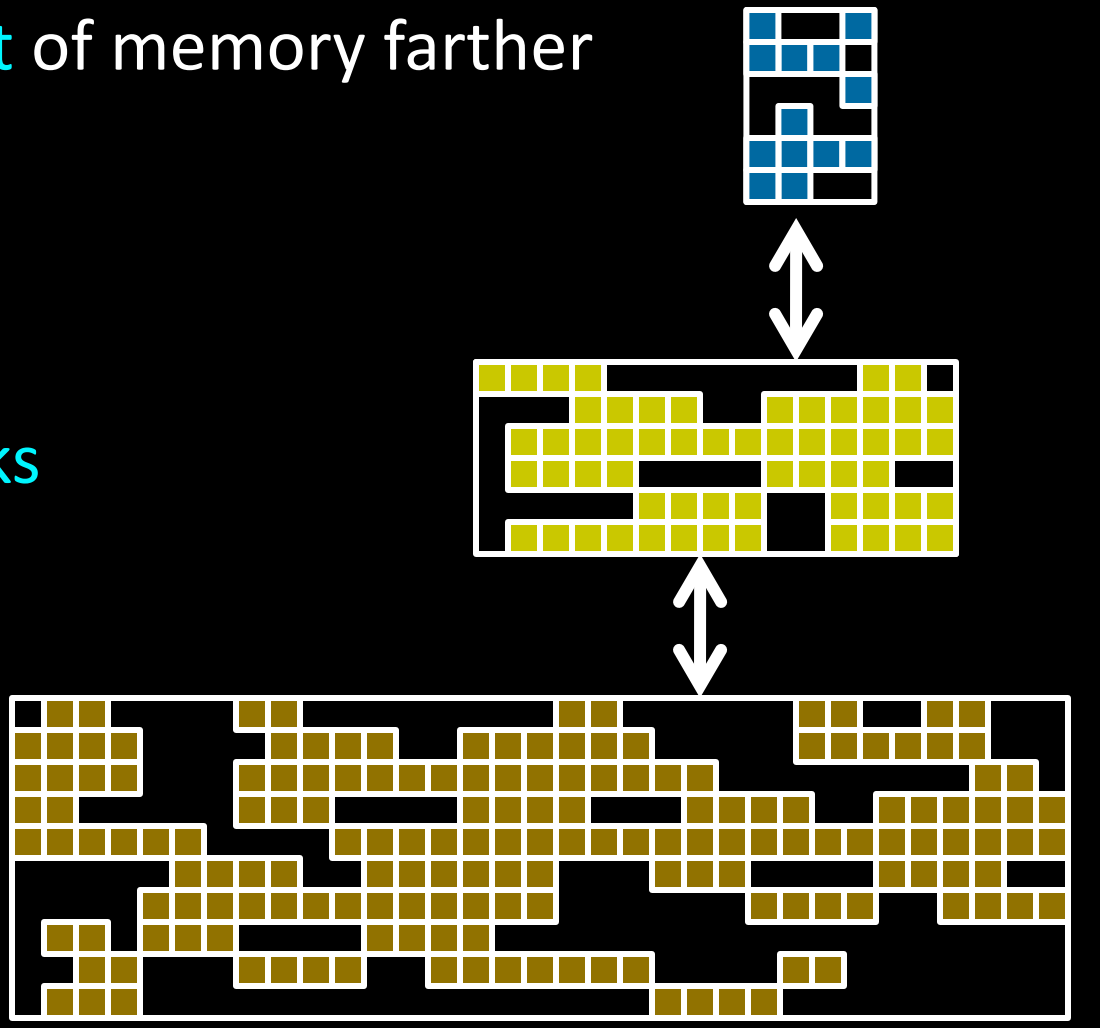
90-10 Rule


$R3 Reg
LW $R3, Mem
1% of data is
accessed the most
L1 Cache
SRAM-on-chip
L2/L3 Cache
SRAM
9% of data is “active”
Memory
DRAM
90% of data inactive
(not accessed)

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 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

Definitions and Terms

- Block (or line)
  - Minimum unit of information that is present/or not in the cache
- Cache hit, miss
- Hit rate
  - The fraction of memory accesses found in a level of the memory hierarchy
- Miss rate
  - The converse

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?

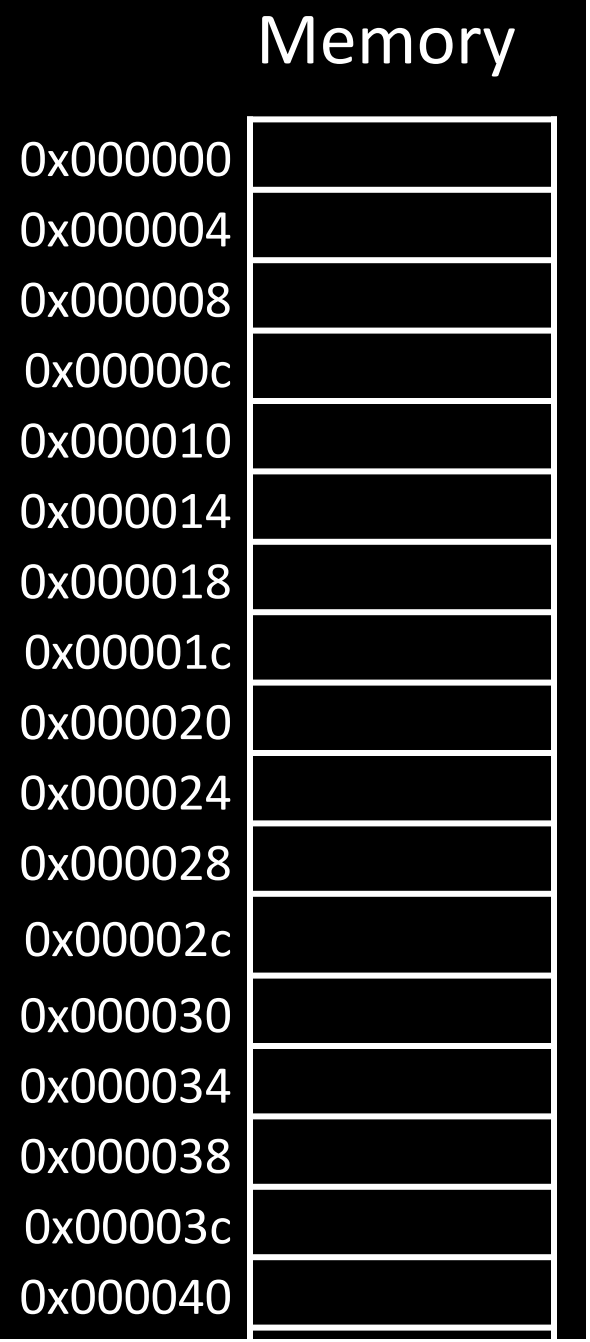
Three common designs

A given data block can be placed…

- … in exactly one cache line → Direct Mapped
- … in any cache line → Fully Associative
- … in a small set of cache lines → Set Associative

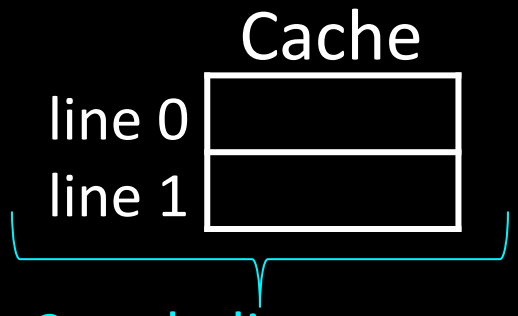
Direct Mapped Cache

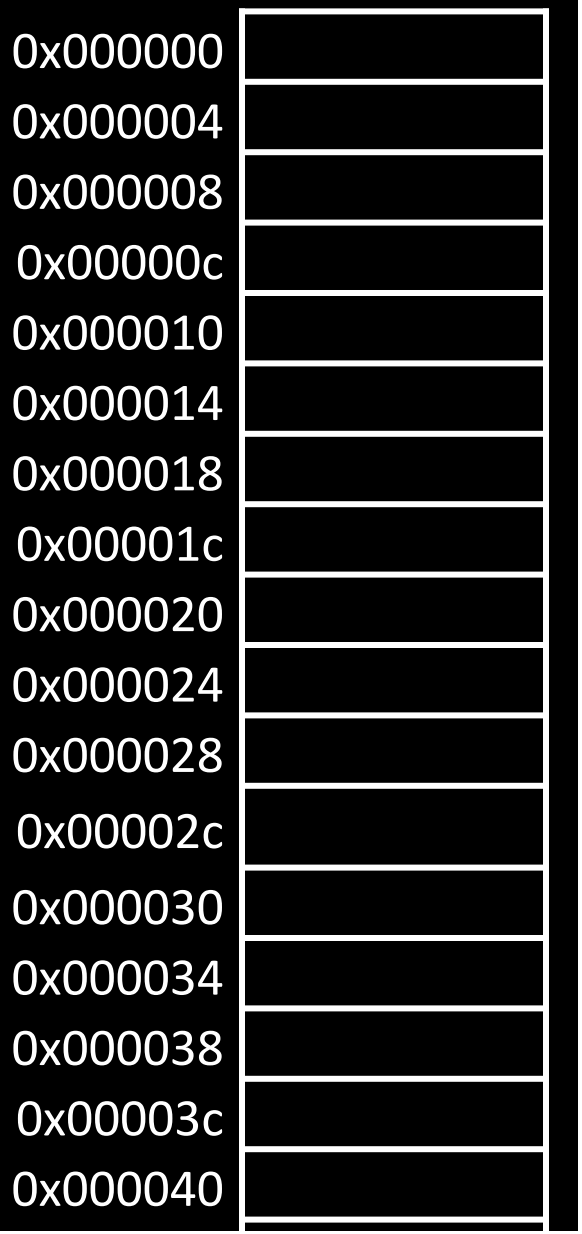
- Each block number maps to a single cache line index
- Simplest hardware
- Questions
  - How to index into cache
  - How to find correct word/byte
  - How to match it



Direct Mapped Cache

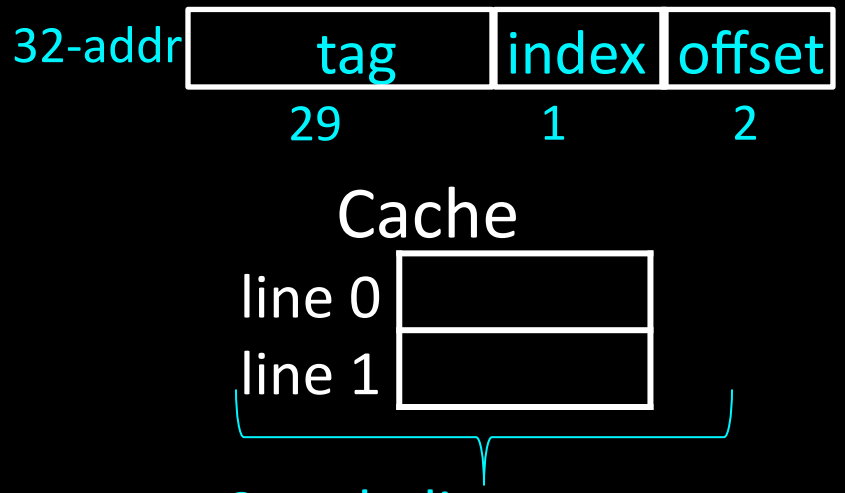
- Each block number maps to a single cache line index
- Simplest hardware
- Questions
  - How to index into cache
  - How to find correct word/byte
  - How to match it

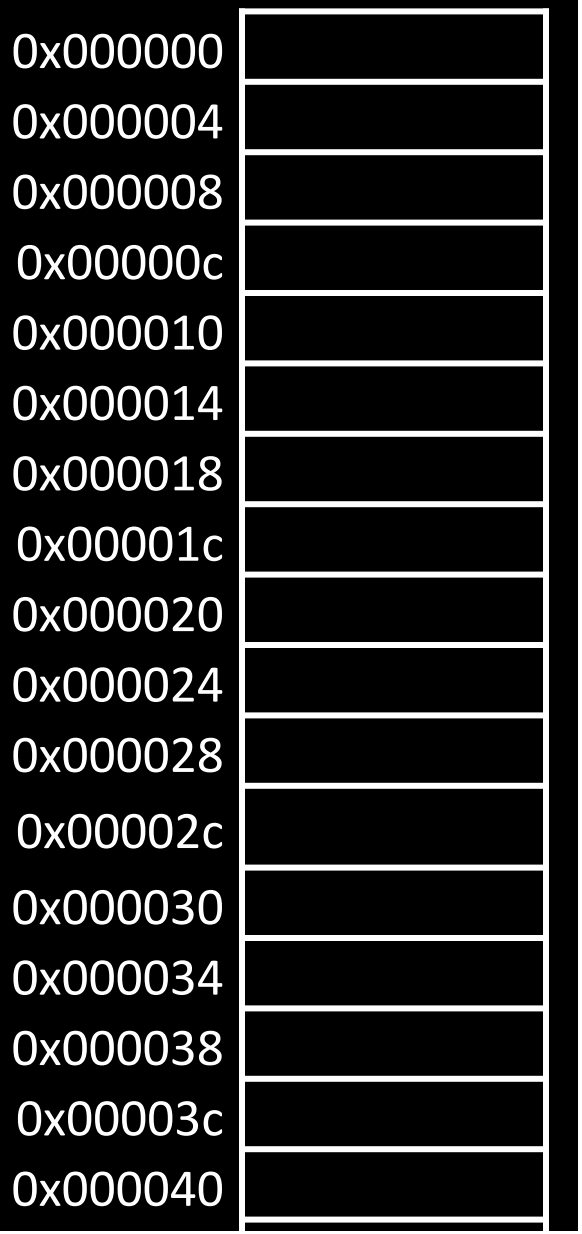




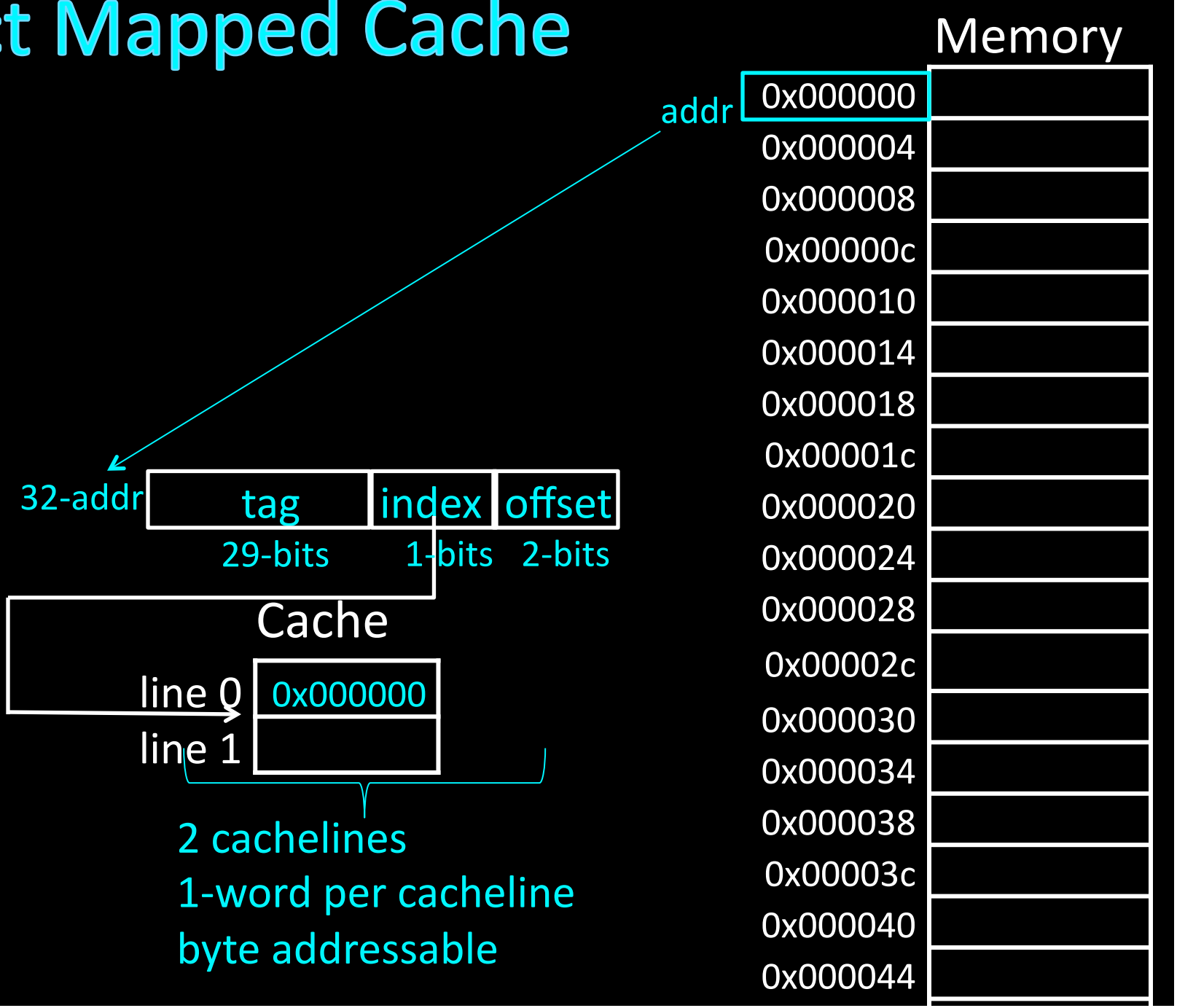
Direct Mapped Cache

- Questions
  - How to index into cache
  - How to find correct word/byte
  - How to match it



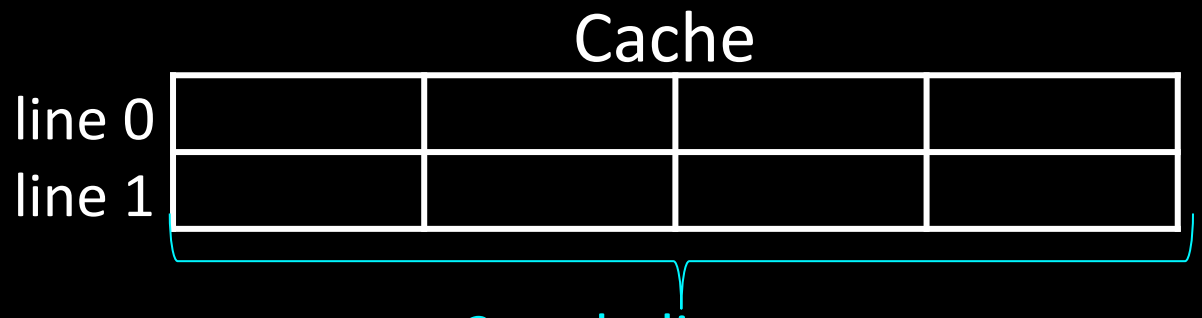


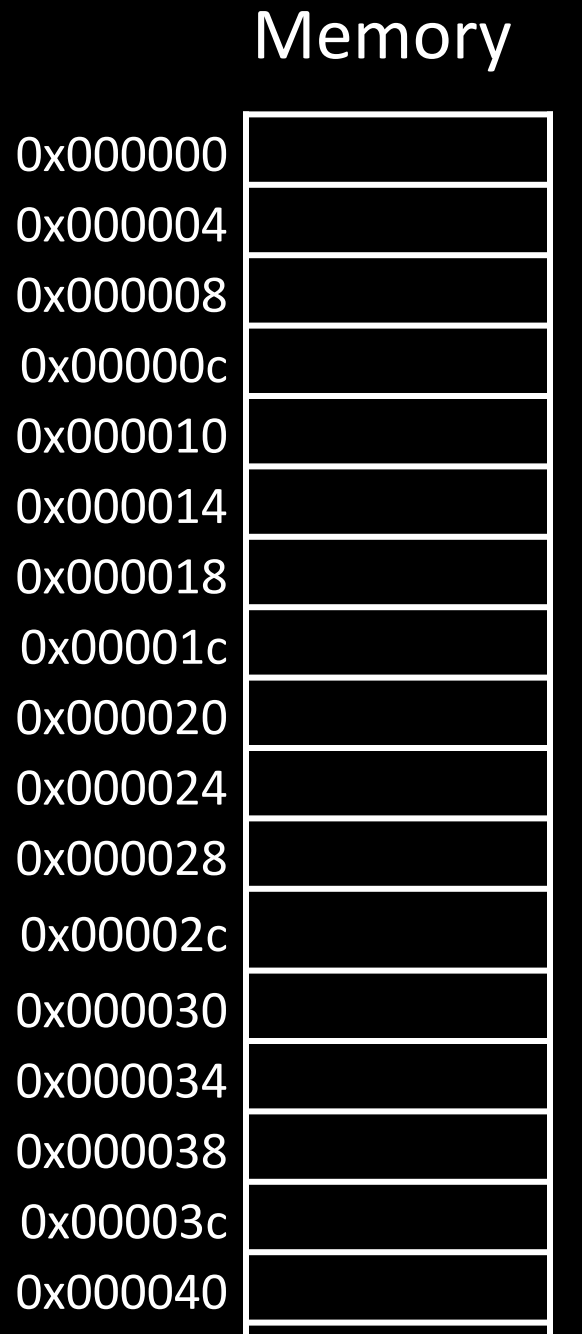
Direct Mapped Cache



Direct Mapped Cache

- Each block number maps to a single cache line index
- Simplest hardware





iClicker

Size of offset?

A) 1

B) 2

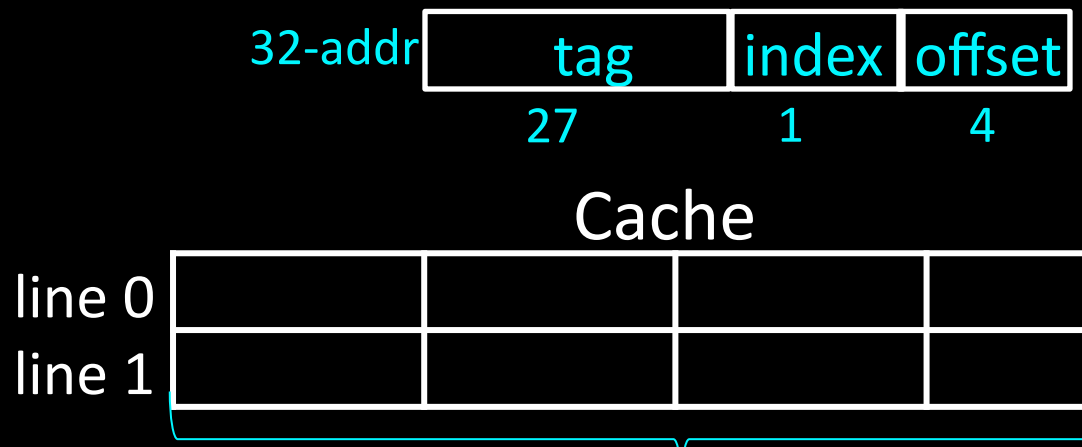
C) 3

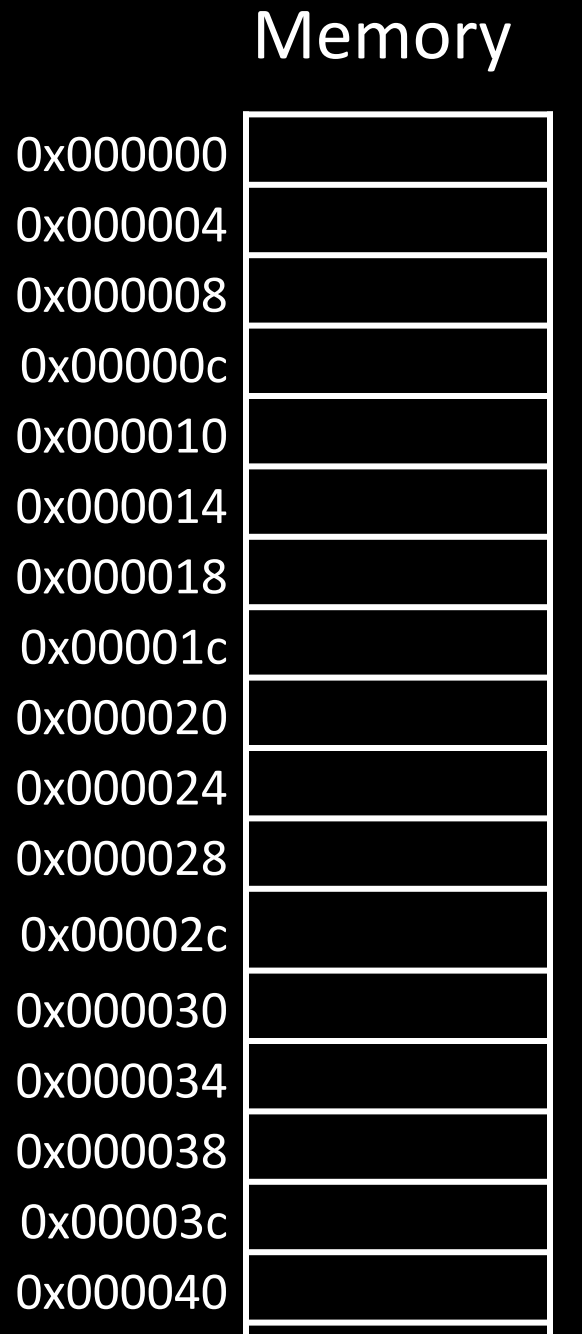
D) 4

E) 5

Size of tag?

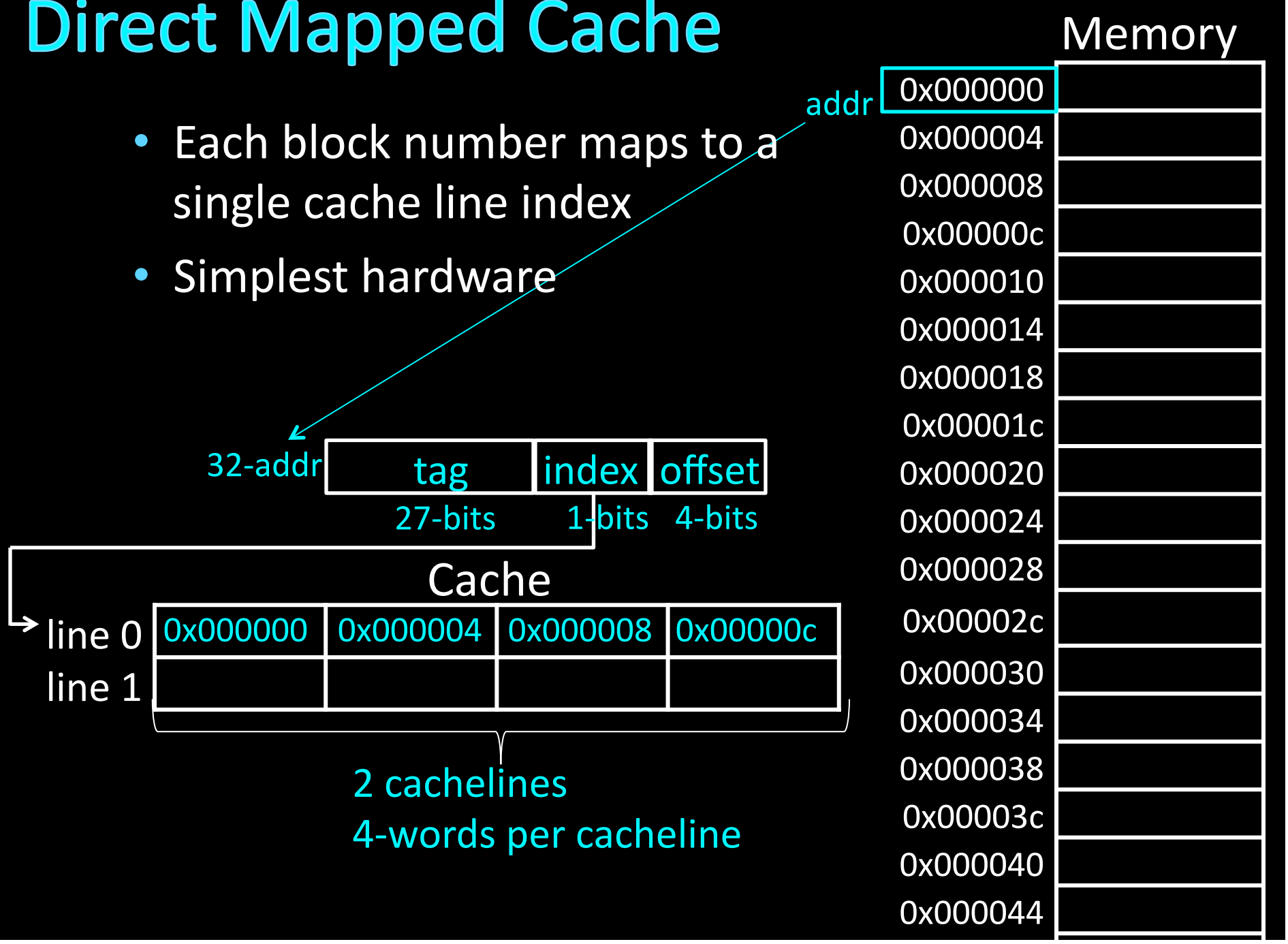
Direct Mapped Cache



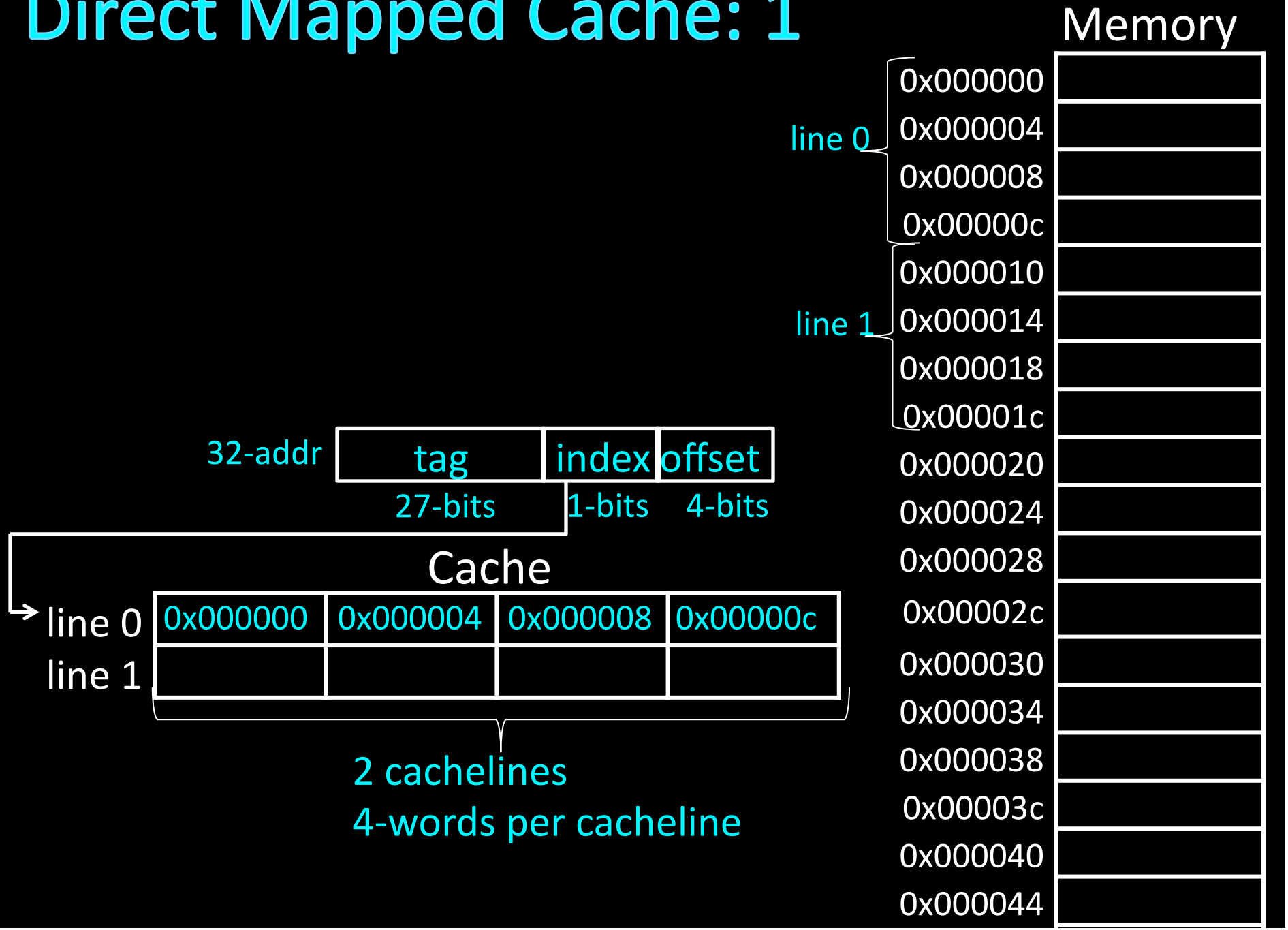


Direct Mapped Cache

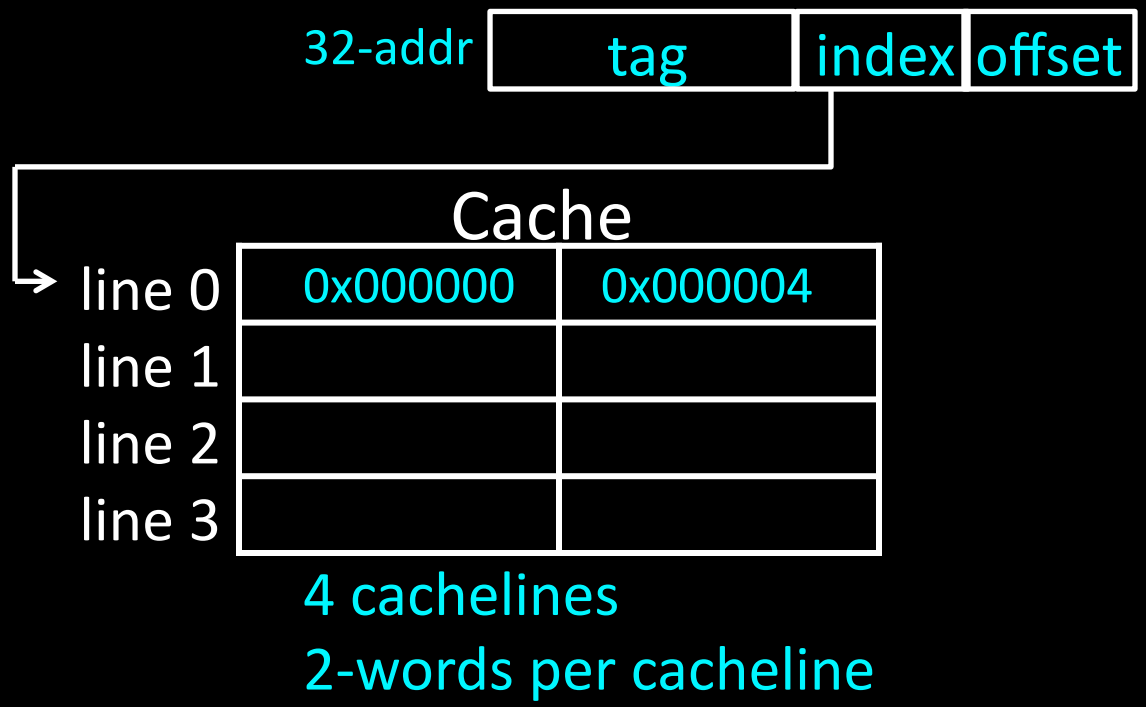
- Each block number maps to a single cache line index
- Simplest hardware

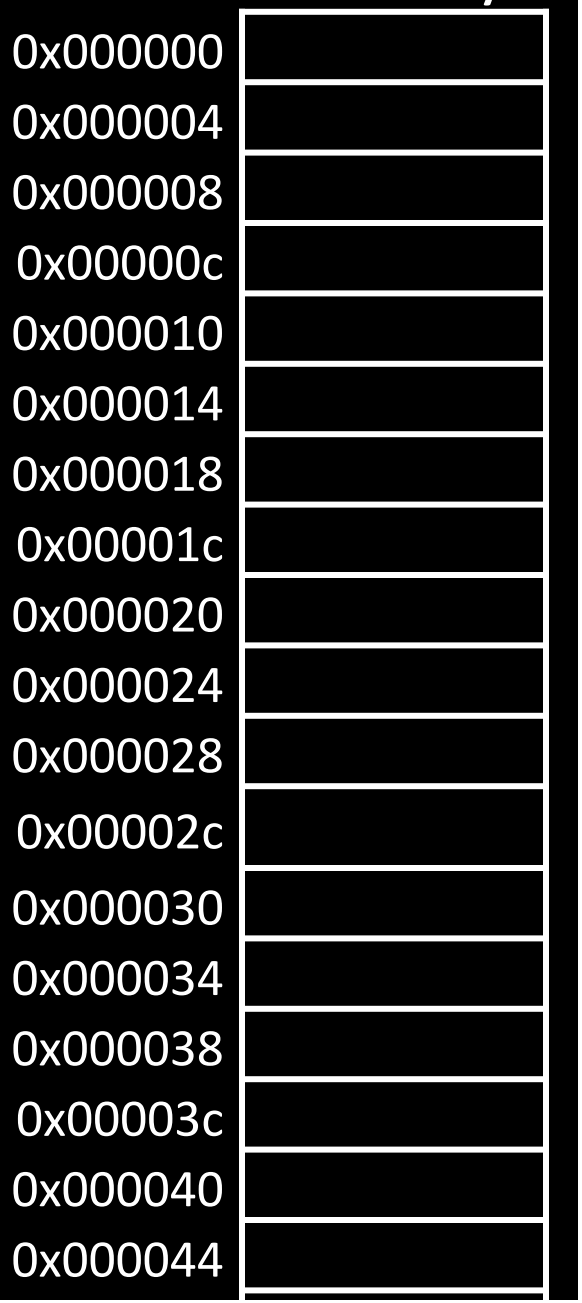


Direct Mapped Cache: 1

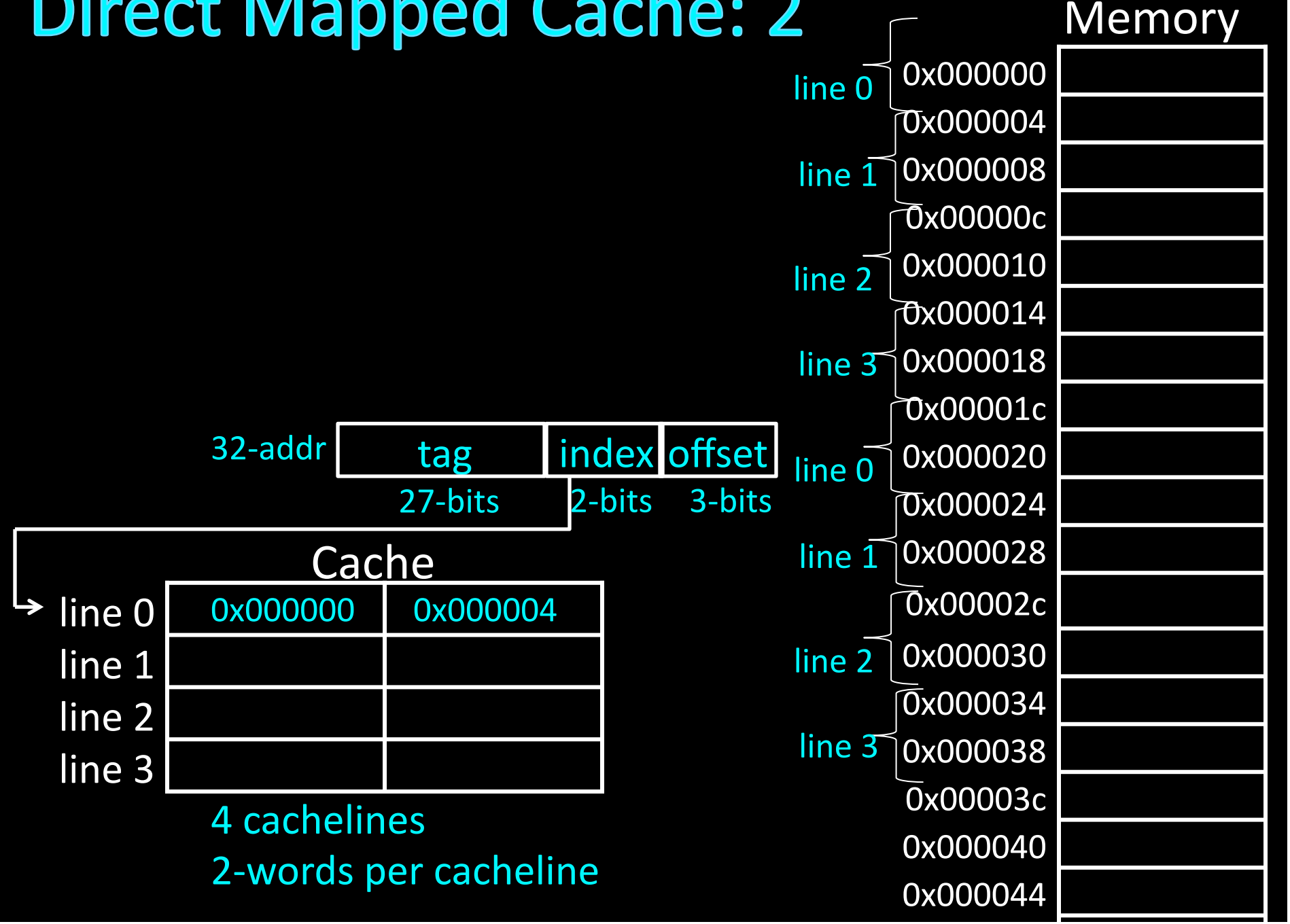


Direct Mapped Cache: 2





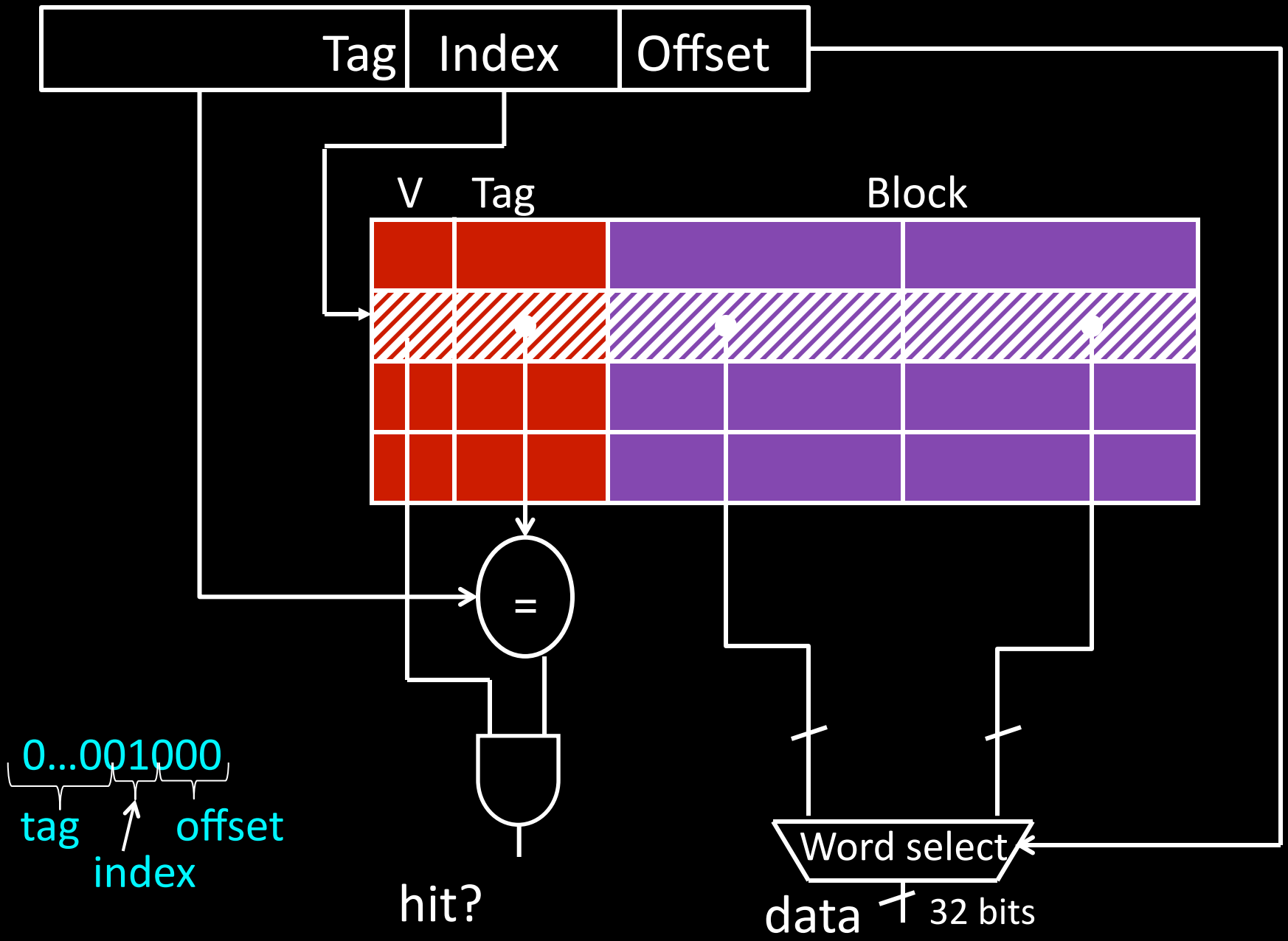
Direct Mapped Cache: 2



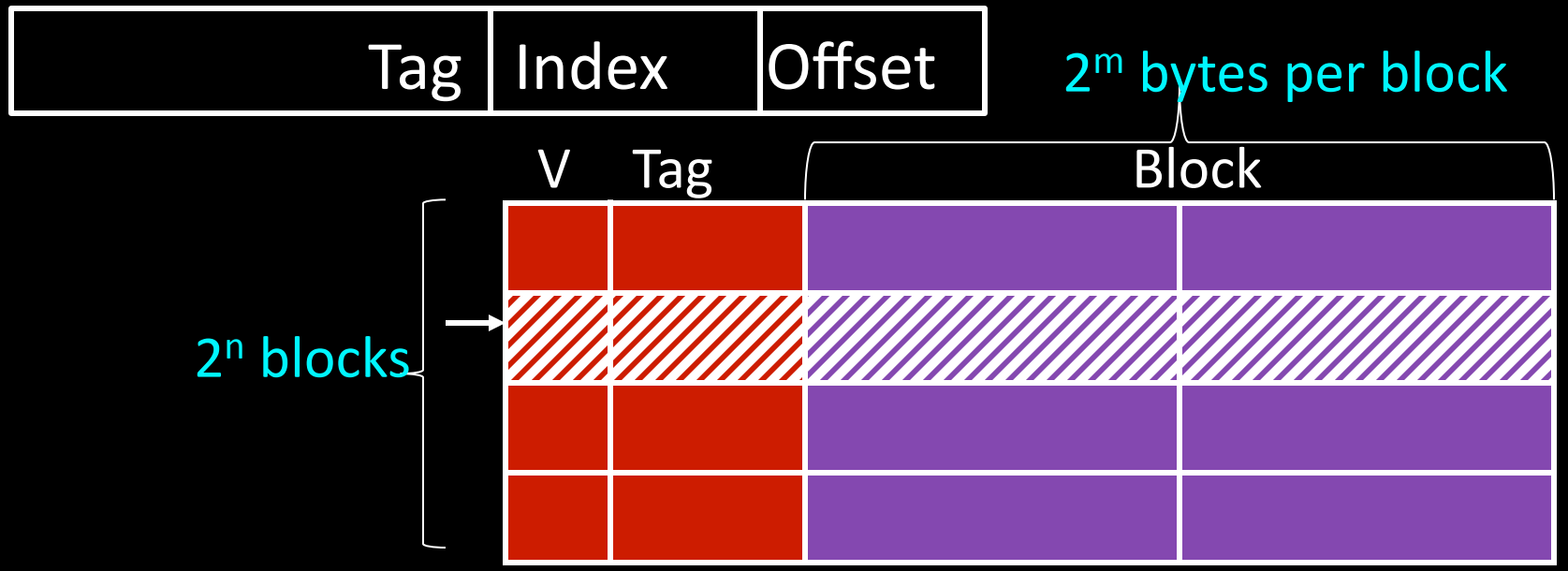
Direct Mapped Cache

Pros: Very simple hardware

Direct Mapped Cache (Hardware)



Direct Mapped Cache Size



n bit index, *m* bit offset

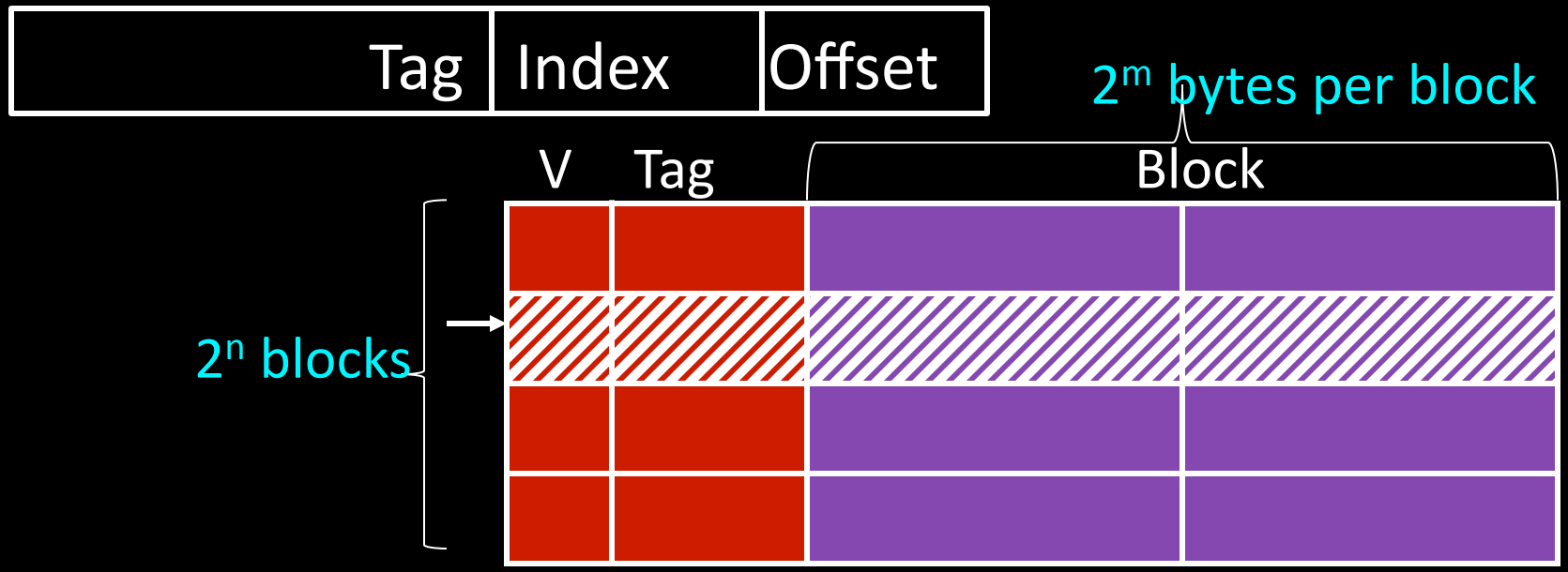
Q: How big is cache (***data only***)?

Cache of size 2^n blocks

Block size of 2^m bytes

Cache Size: 2^n bytes per block x 2^n blocks = 2^{n+m} bytes

Direct Mapped Cache Size



n bit index, *m* bit offset

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

Cache of size 2^n blocks

Block size of 2^m bytes

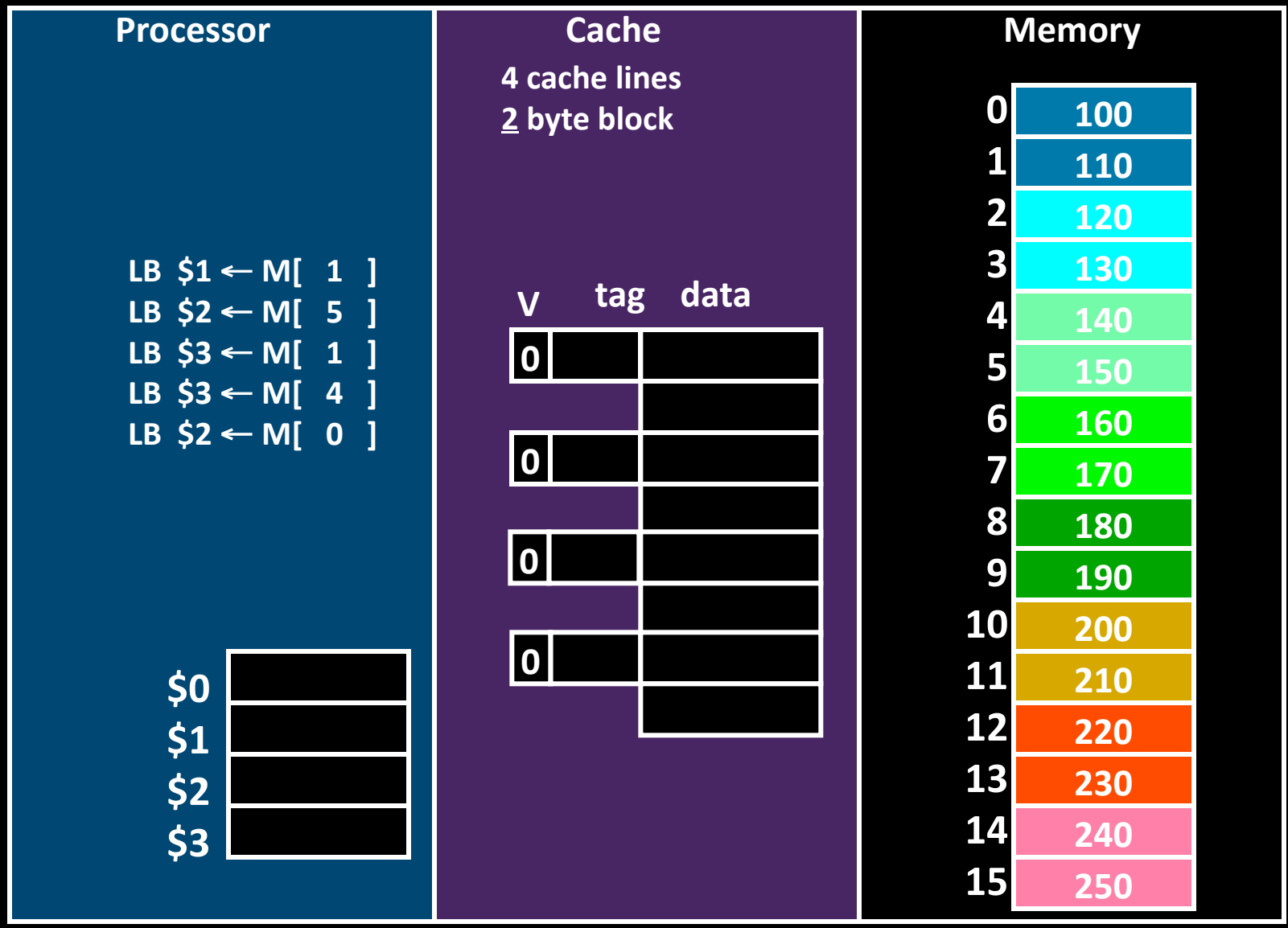
Tag field: 32 – (n + m), Valid bit: 1

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

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

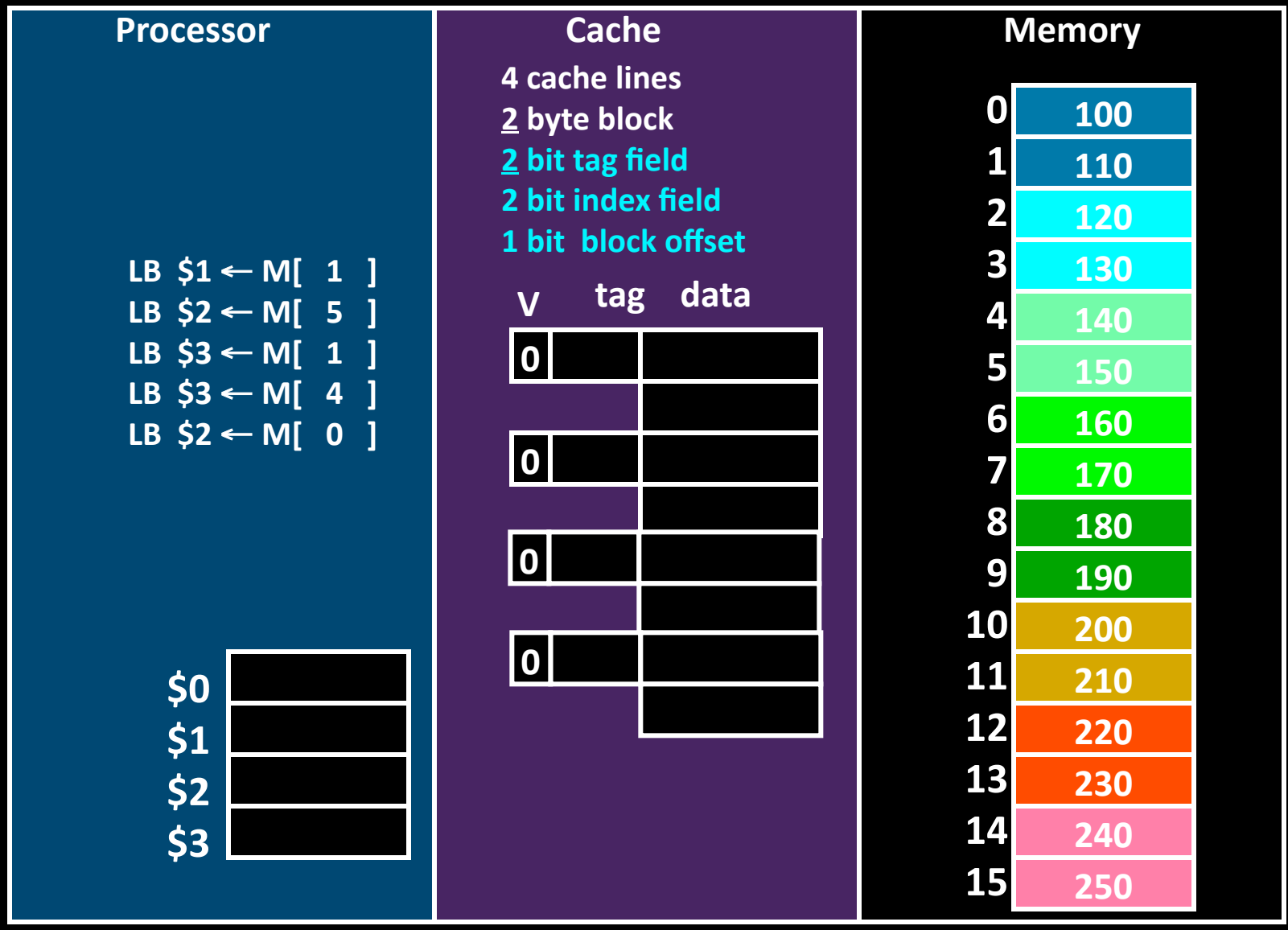
Example: Direct Mapped Cache

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

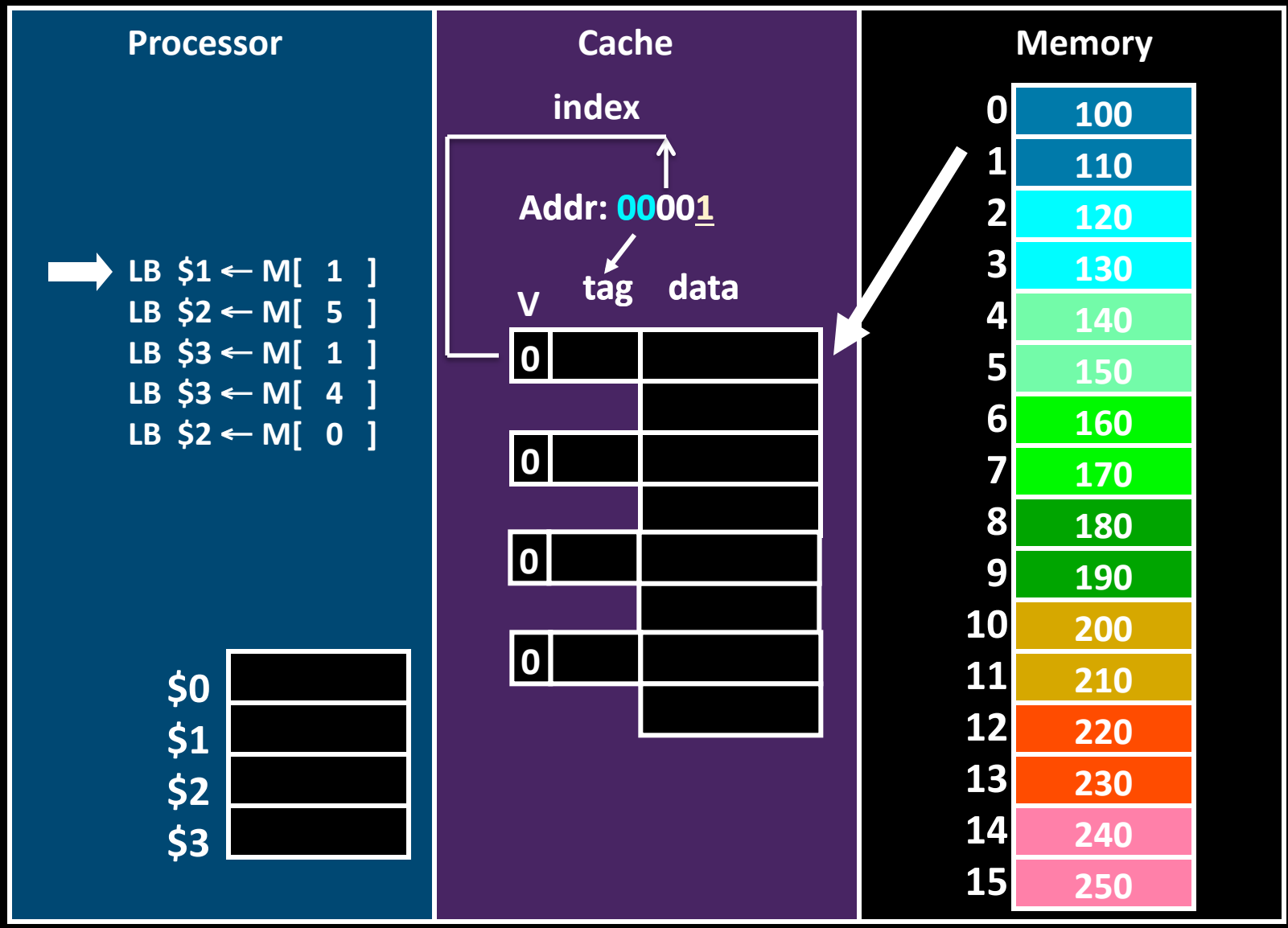


Example: Direct Mapped Cache

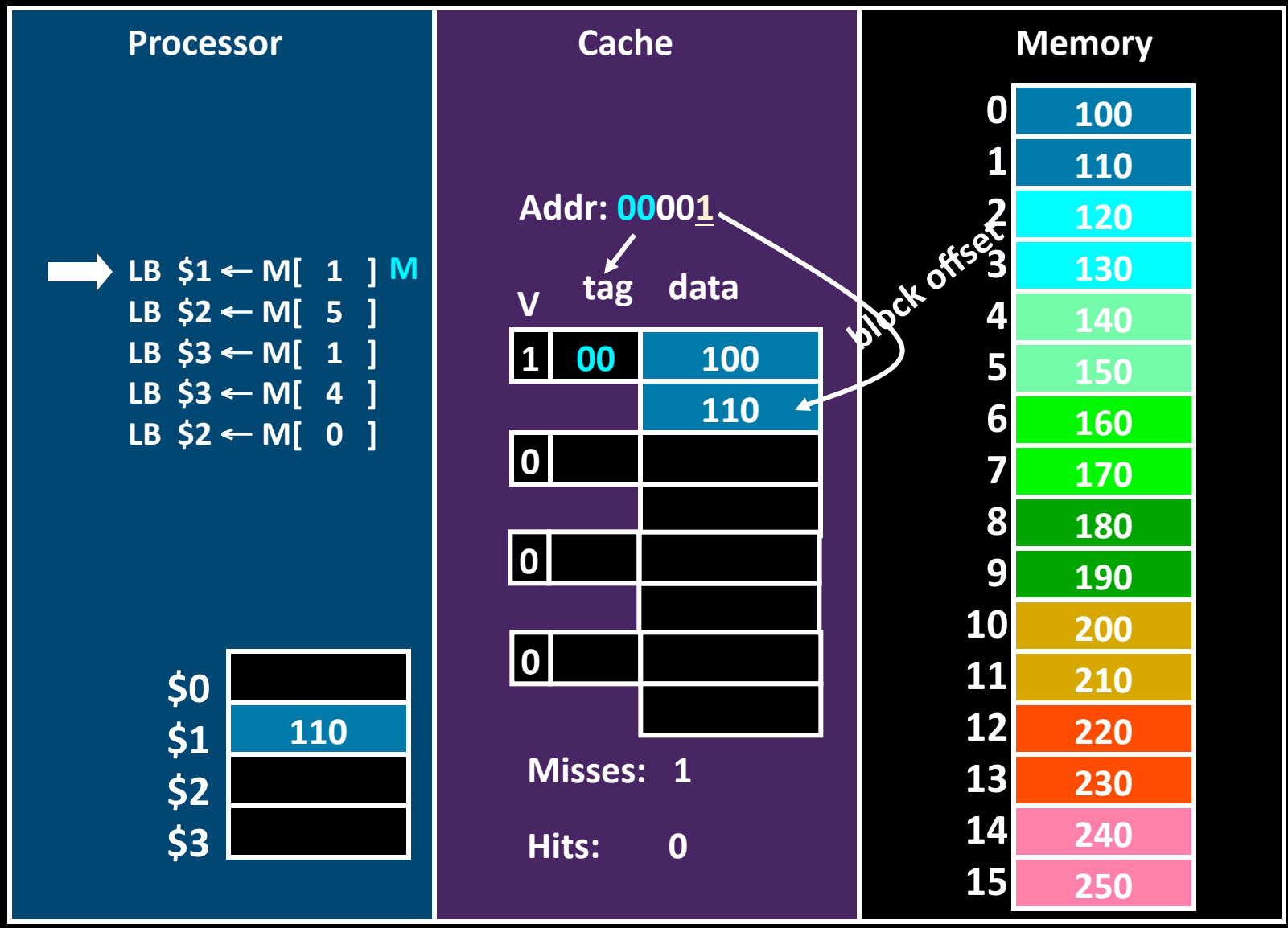
Using **byte addresses** in this example. Addr Bus = 5 bits



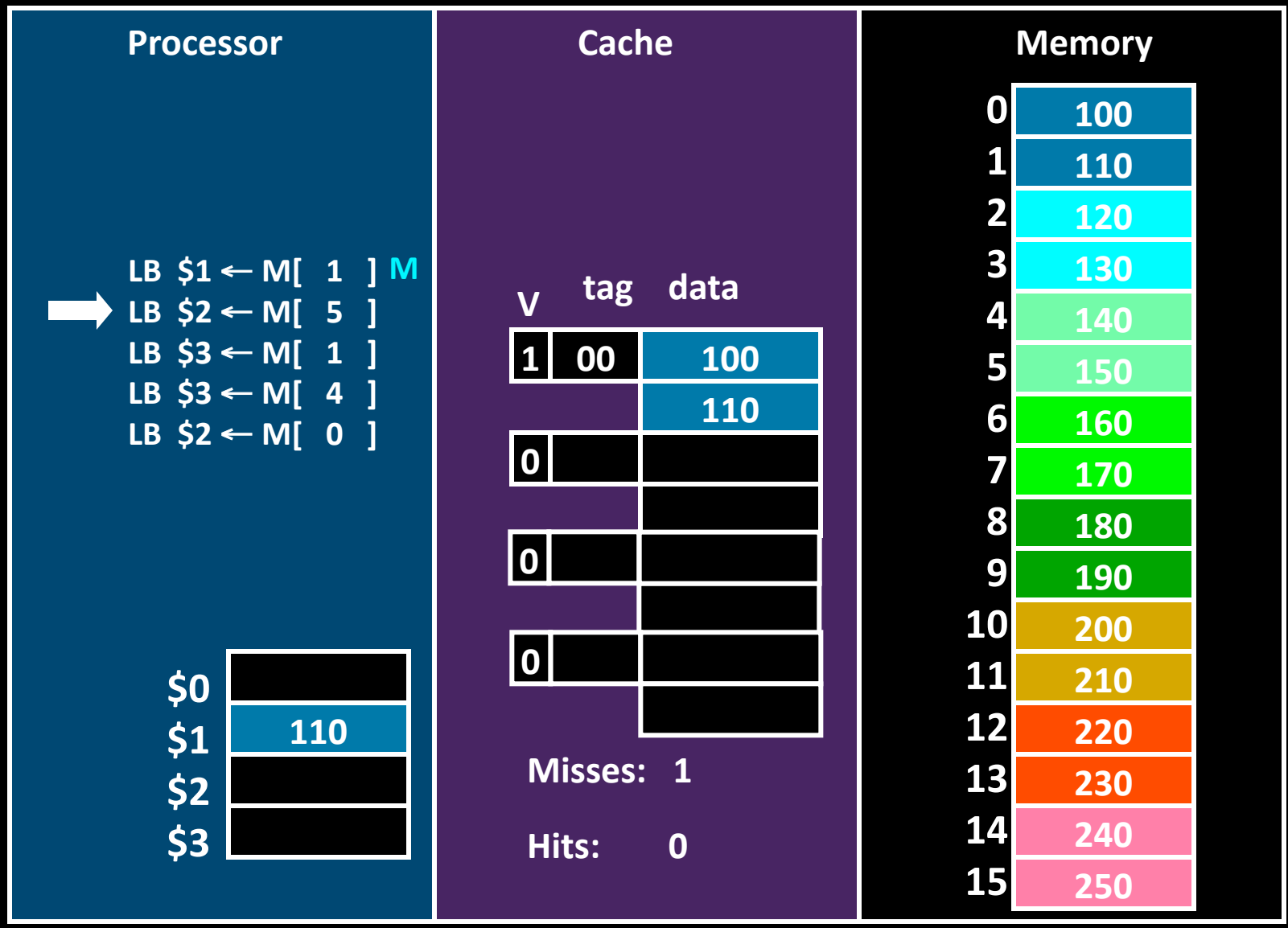
1st Access


Processor
Cache
Memory
index
Addr: 00001
tag
data
V
LB $1 ← M[1]
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1
$2
$3
0
0
0
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

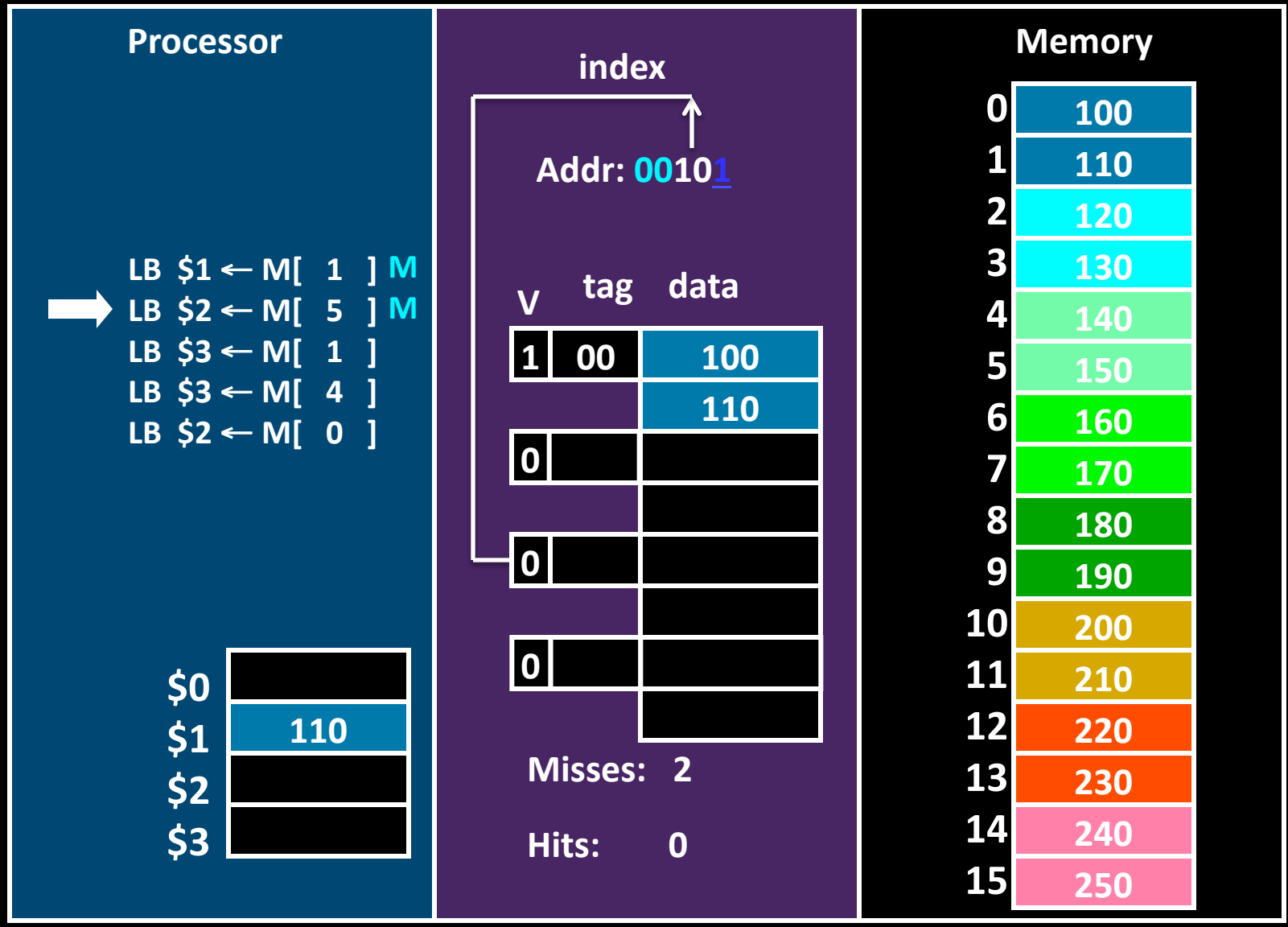
1st Access


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
V
tag
data
1
00
100
110
0
0
0
Misses: 1
Hits: 0
block offset
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

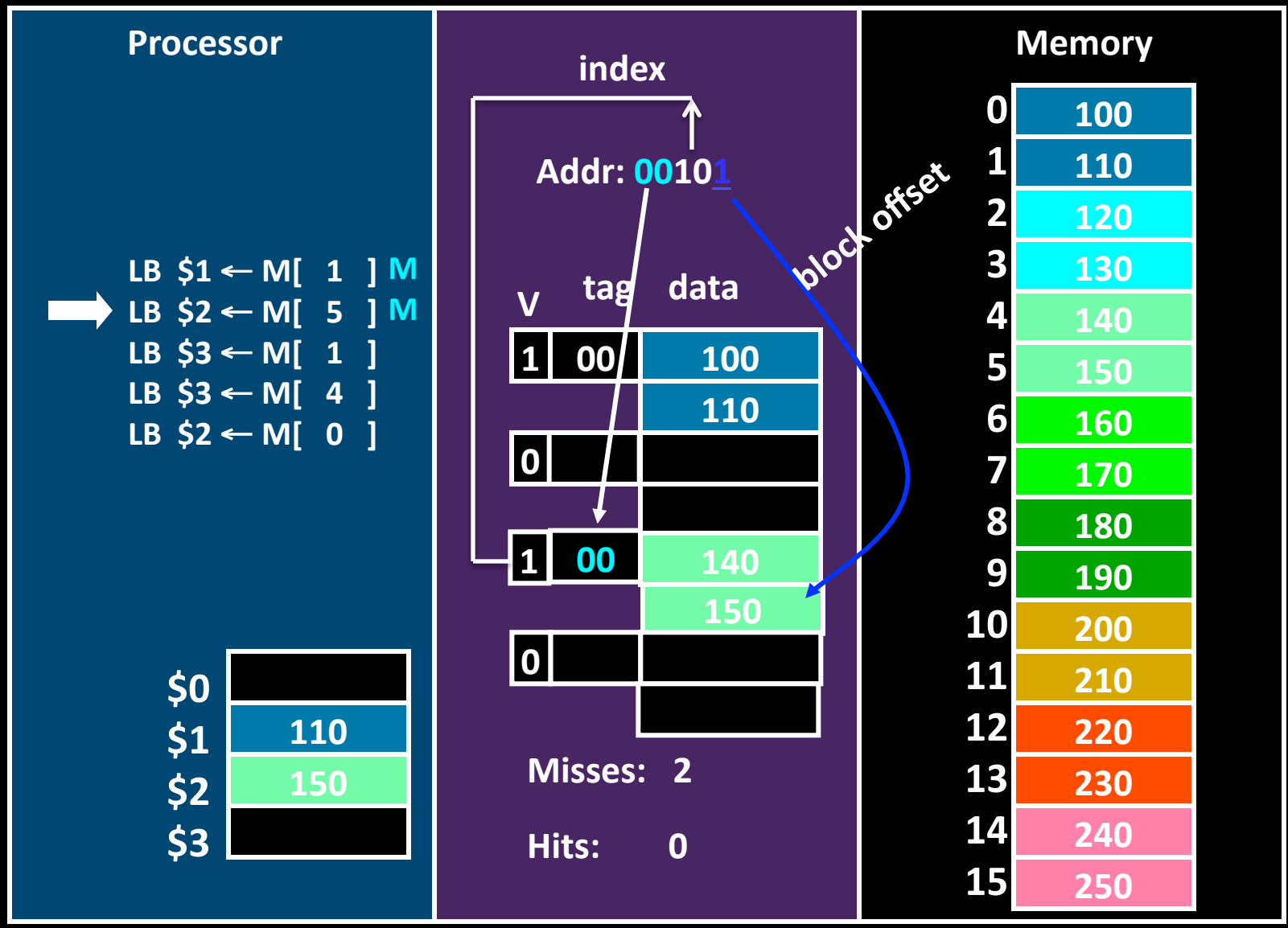
2nd Access


Processor
LB $1 ← M[1]
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]
M
$0
$1
110
$2
$3
Cache
V
tag
data
1
00
100
110
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

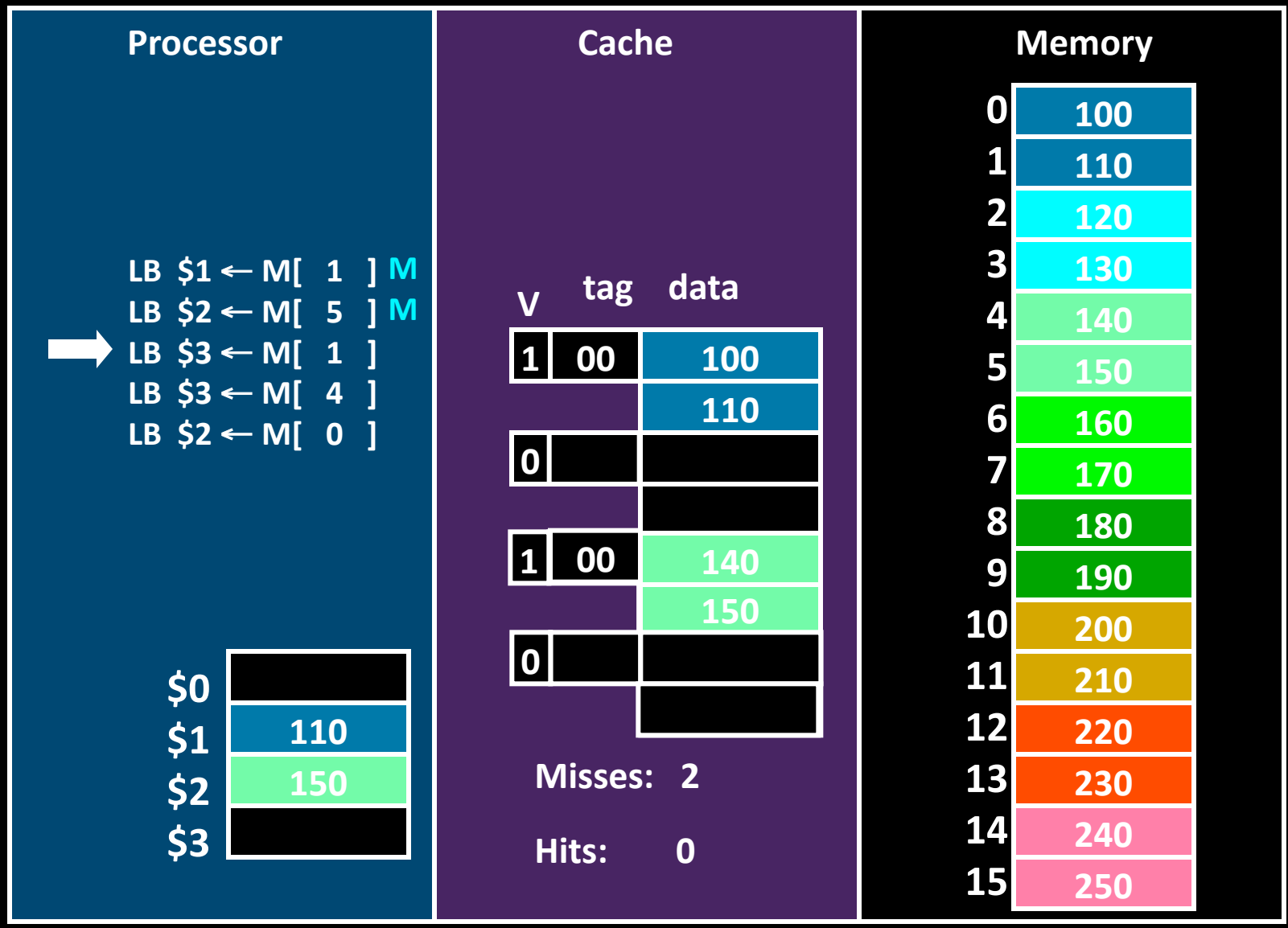
2nd Access


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
$3
index
Addr: 00101
V
tag
data
1
00
100
110
0
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

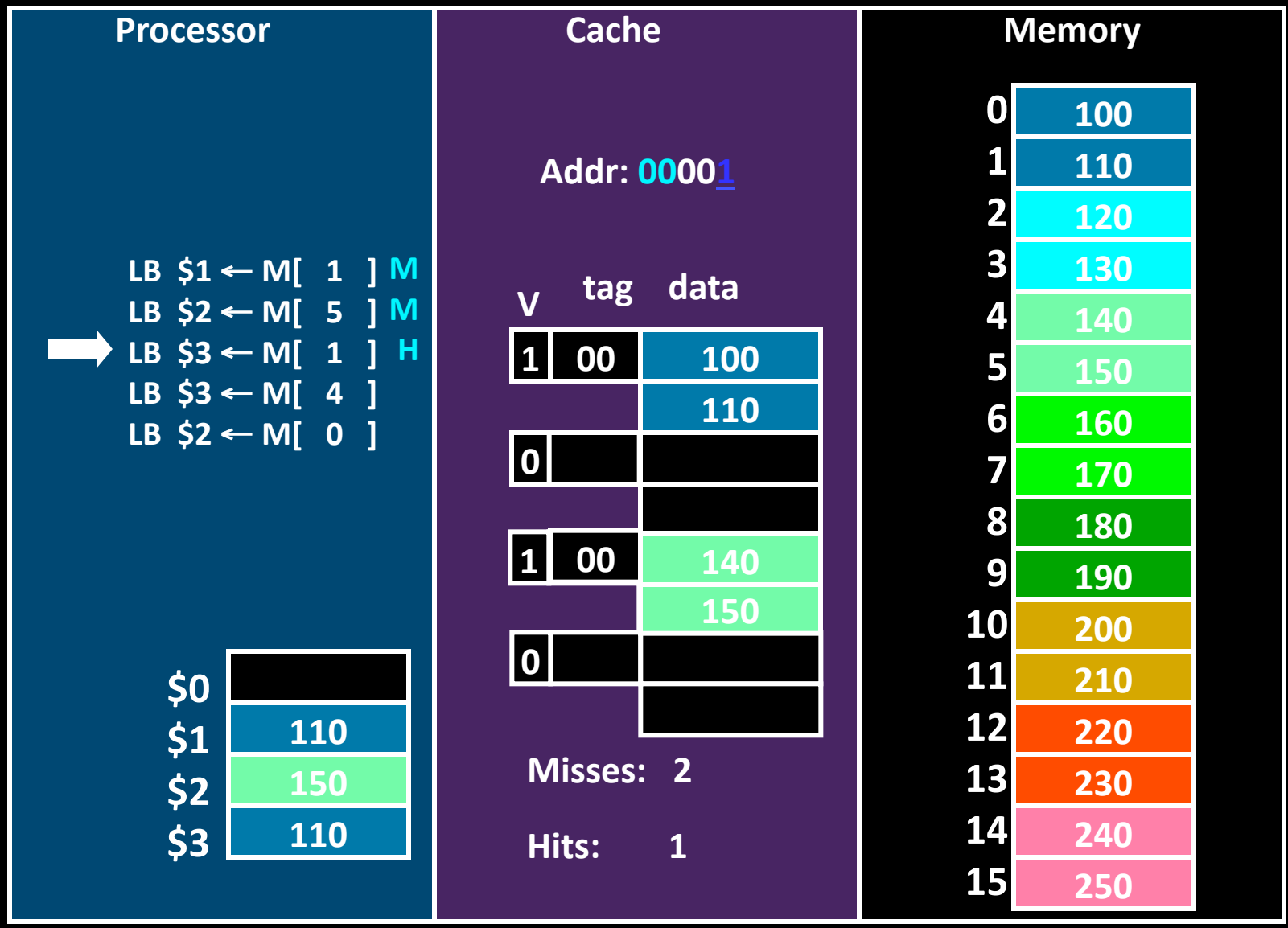
2nd Access


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
index
Addr: 00101
block offset
V
tag
data
1
00
100
110
0
1
00
140
150
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

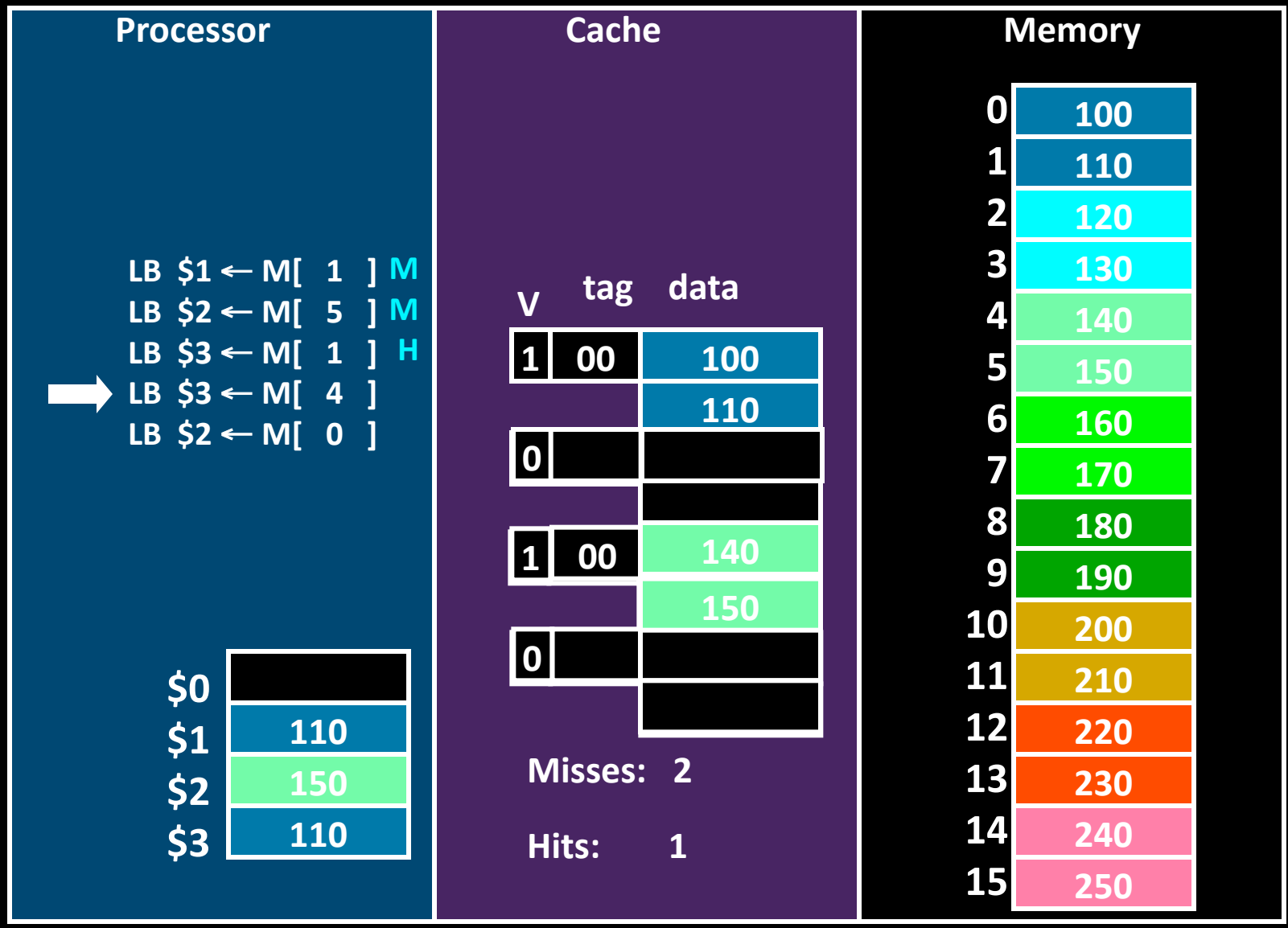
3rd Access


Processor
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
V
tag
data
1
00
100
110
0
1
00
140
150
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

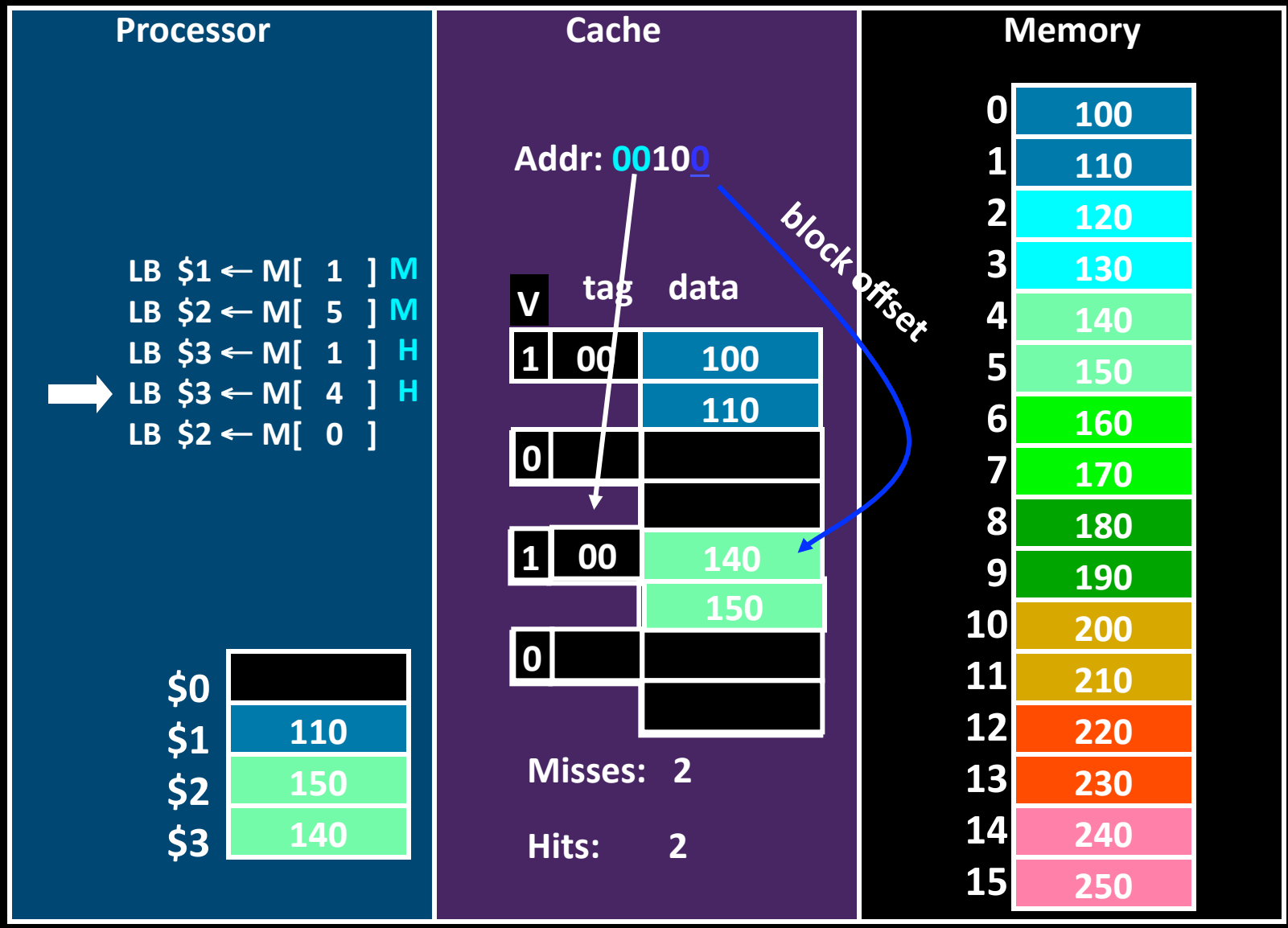
3rd Access


Processor
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
Addr: 00001
V tag data
1 00 100
110
0
1 00 140
150
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

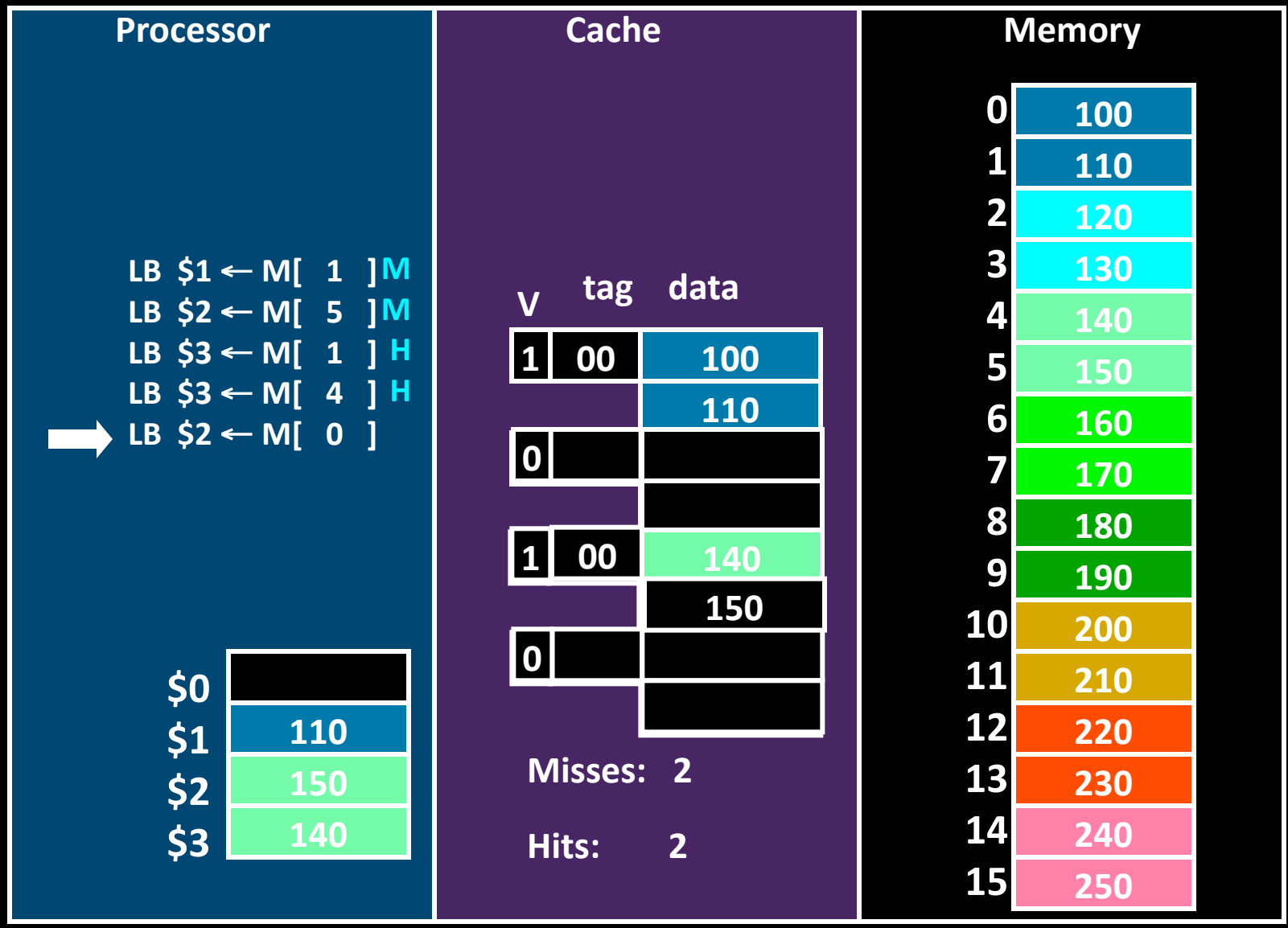
4th Access


Processor
LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4]
LB $2 ← M[0]
$0
$1 110
$2 150
$3 110
Cache
V tag data
1 00 100
110
0
1 00 140
150
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

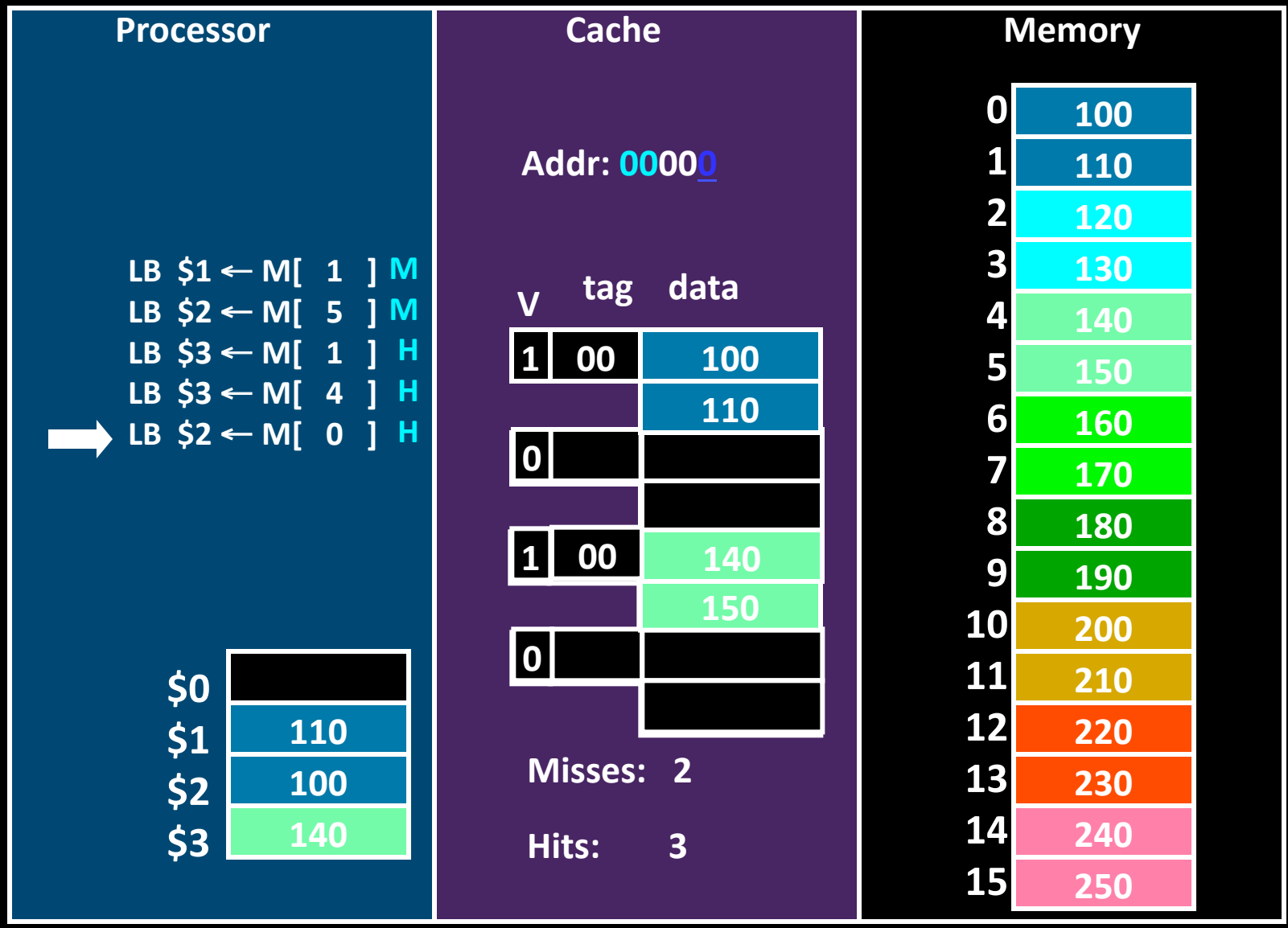
4th Access


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
block offset
V tag data
1 00 100
110
0
1 00 140
150
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

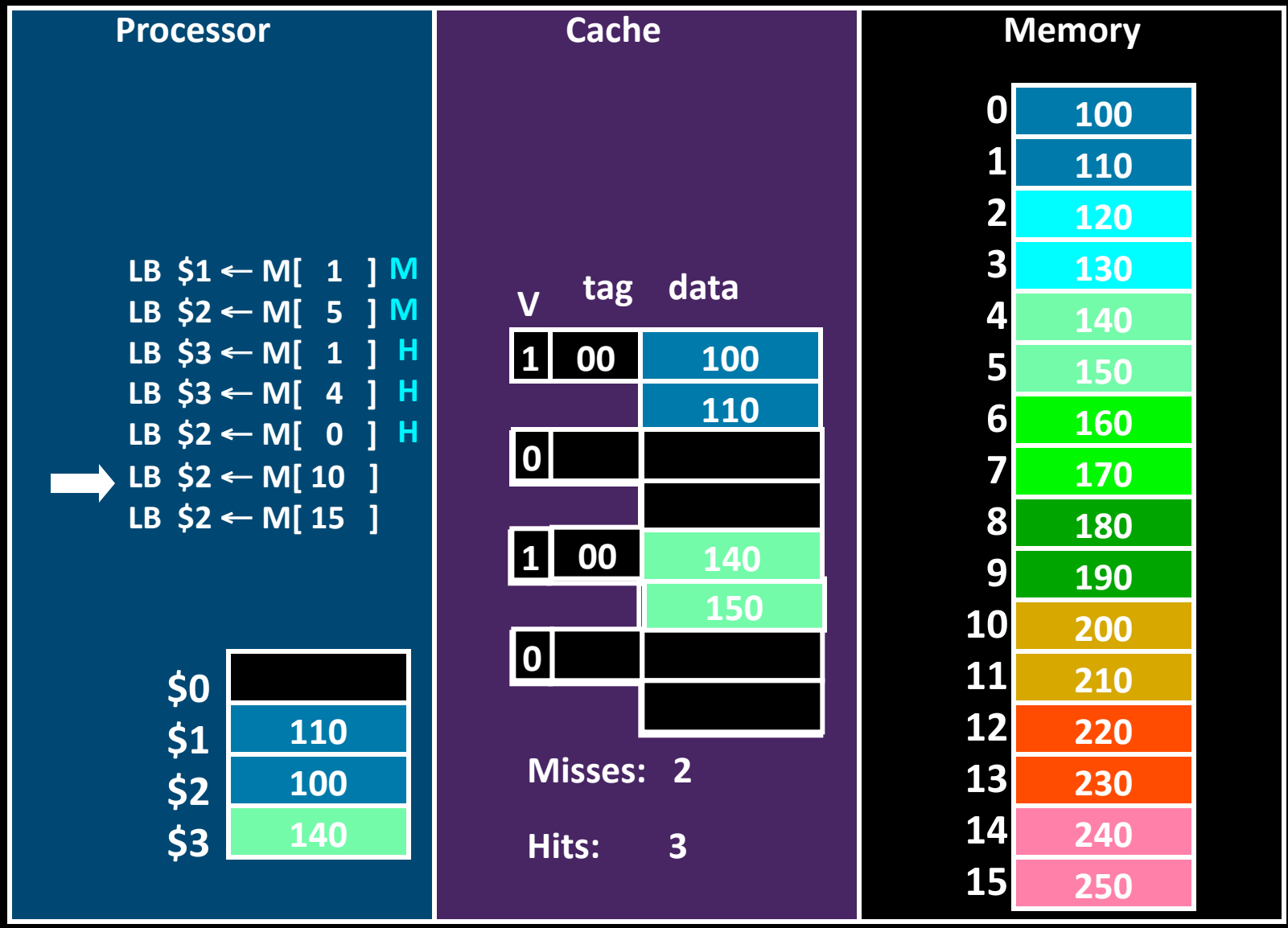
5th Access


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]
V
tag
data
1 00 100
110
0
1 00 140
150
0
Misses: 2
Hits: 2
$0
$1 110
$2 150
$3 140
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

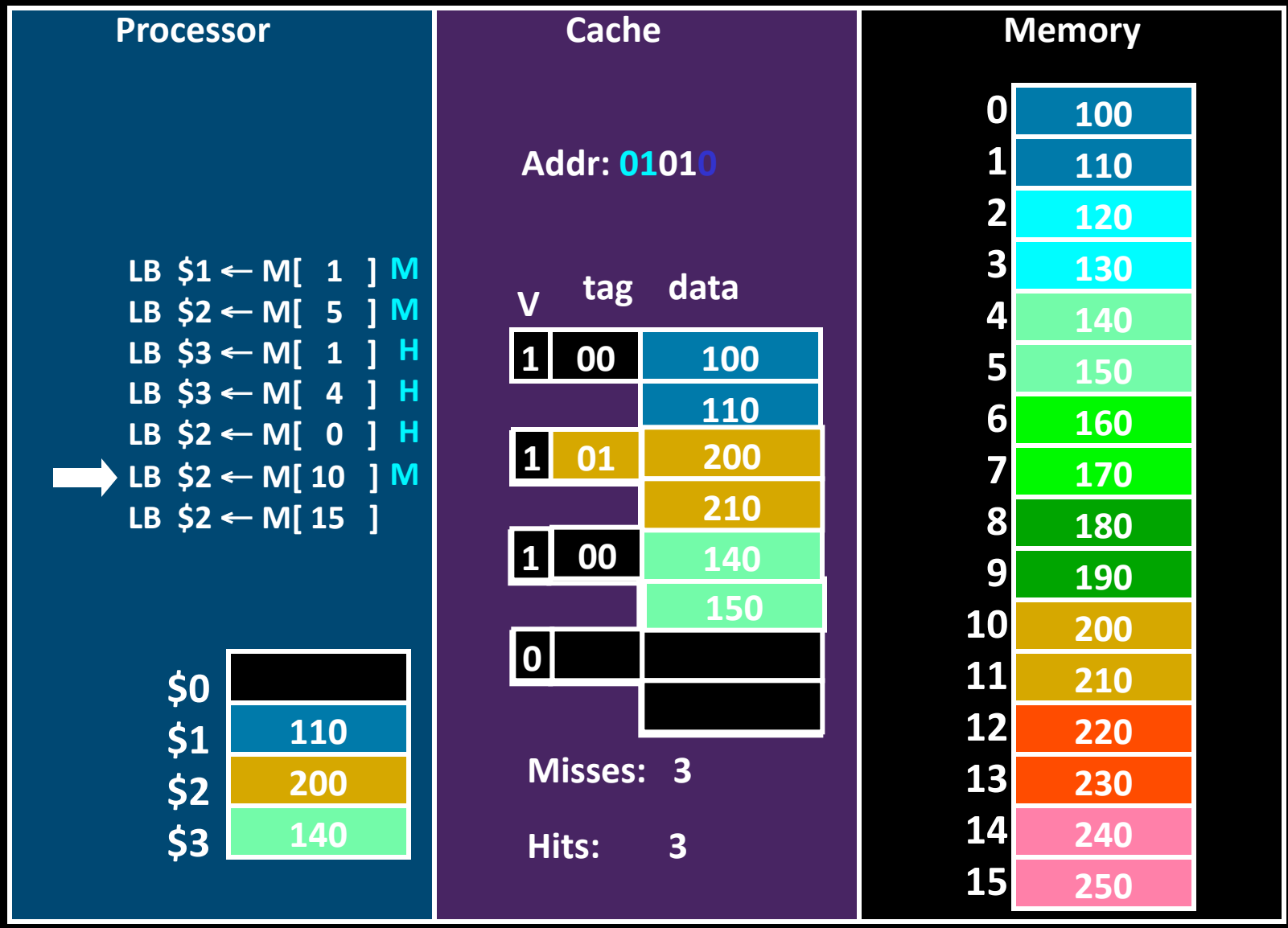
5th Access


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] H
$0
$1 110
$2 100
$3 140
Cache
Addr: 00000
V tag data
1 00 100
110
0
1 00 140
150
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

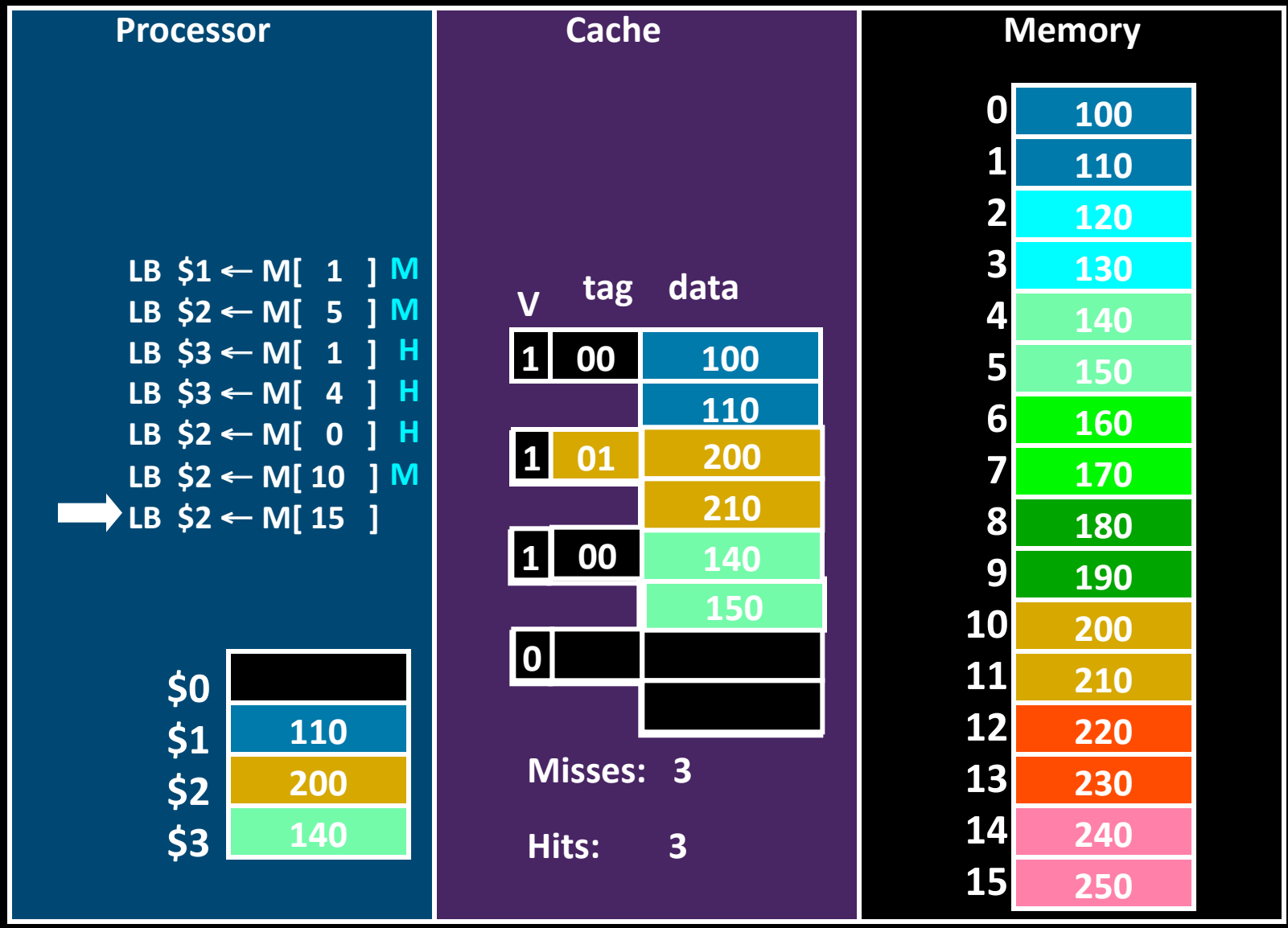
6th Access


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[10]
LB $2 ← M[15]
$0
$1 110
$2 100
$3 140
Cache
V tag data
1 00 100
110
0
1 00 140
150
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

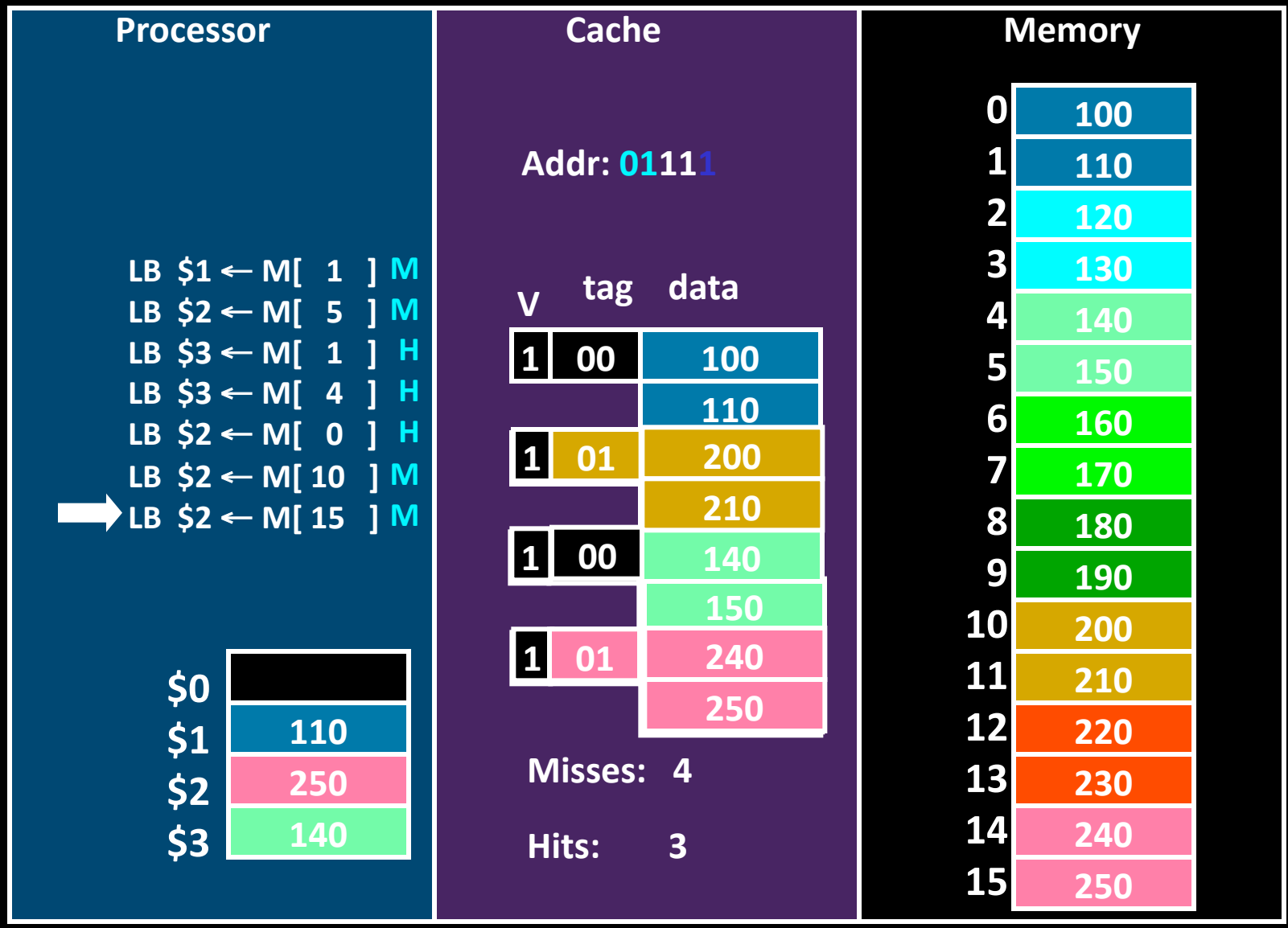
6th Access


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[10] M
LB $2 ← M[15]
$0
$1 110
$2 200
$3 140
Cache
Addr: 01010
V tag data
1 00 100
110
1 01 200
210
1 00 140
150
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

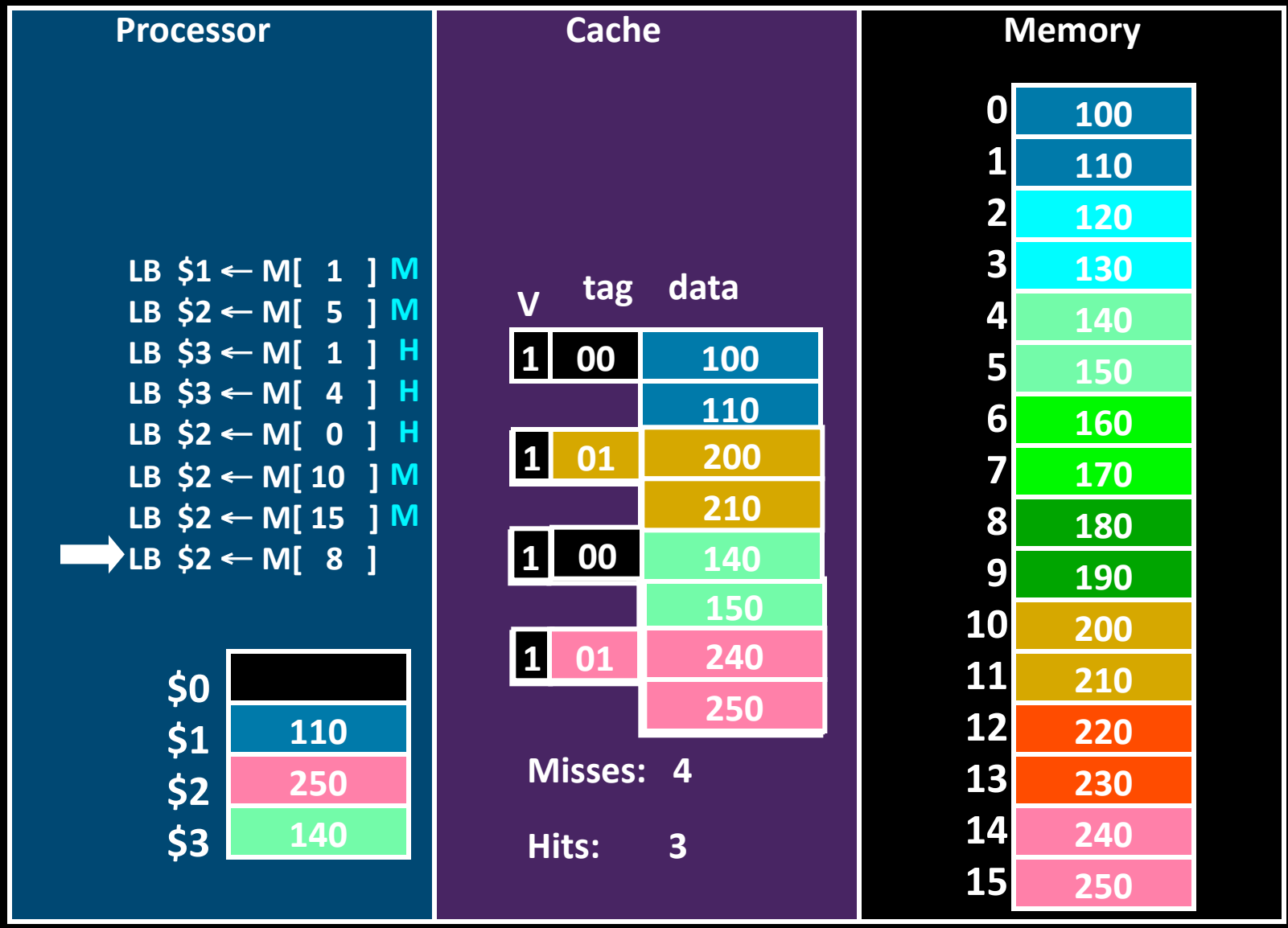
7th Access


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[10] M
LB $2 ← M[15]
$0
$1 110
$2 200
$3 140
Cache
V tag data
1 00 100
110
1 01 200
210
1 00 140
150
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

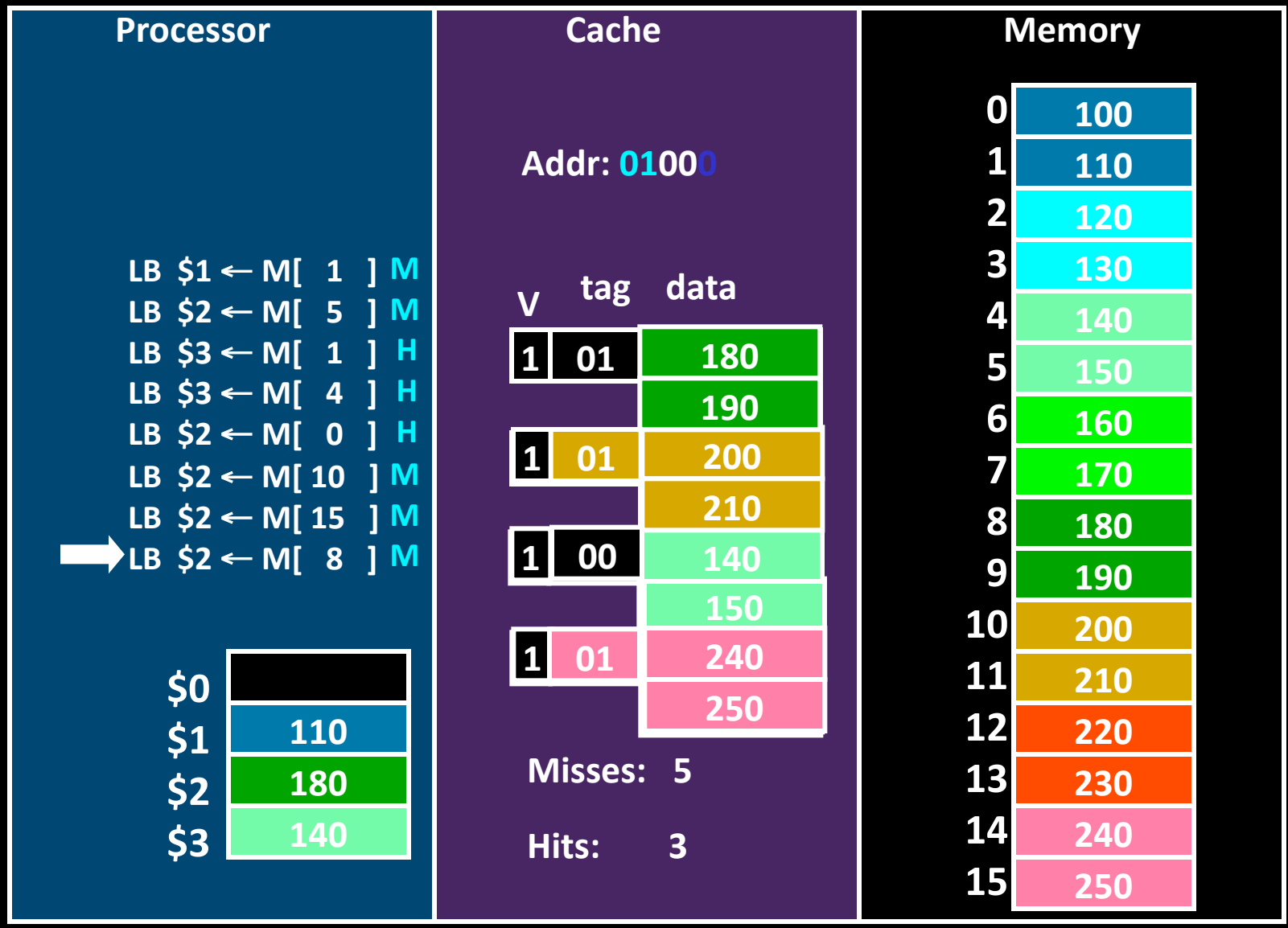
7th Access


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[10] M
LB $2 ← M[15] M
$0
$1 110
$2 250
$3 140
Cache
Addr: 01111
V tag data
1 00 100
110
1 01 200
210
1 00 140
150
1 01 240
250
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

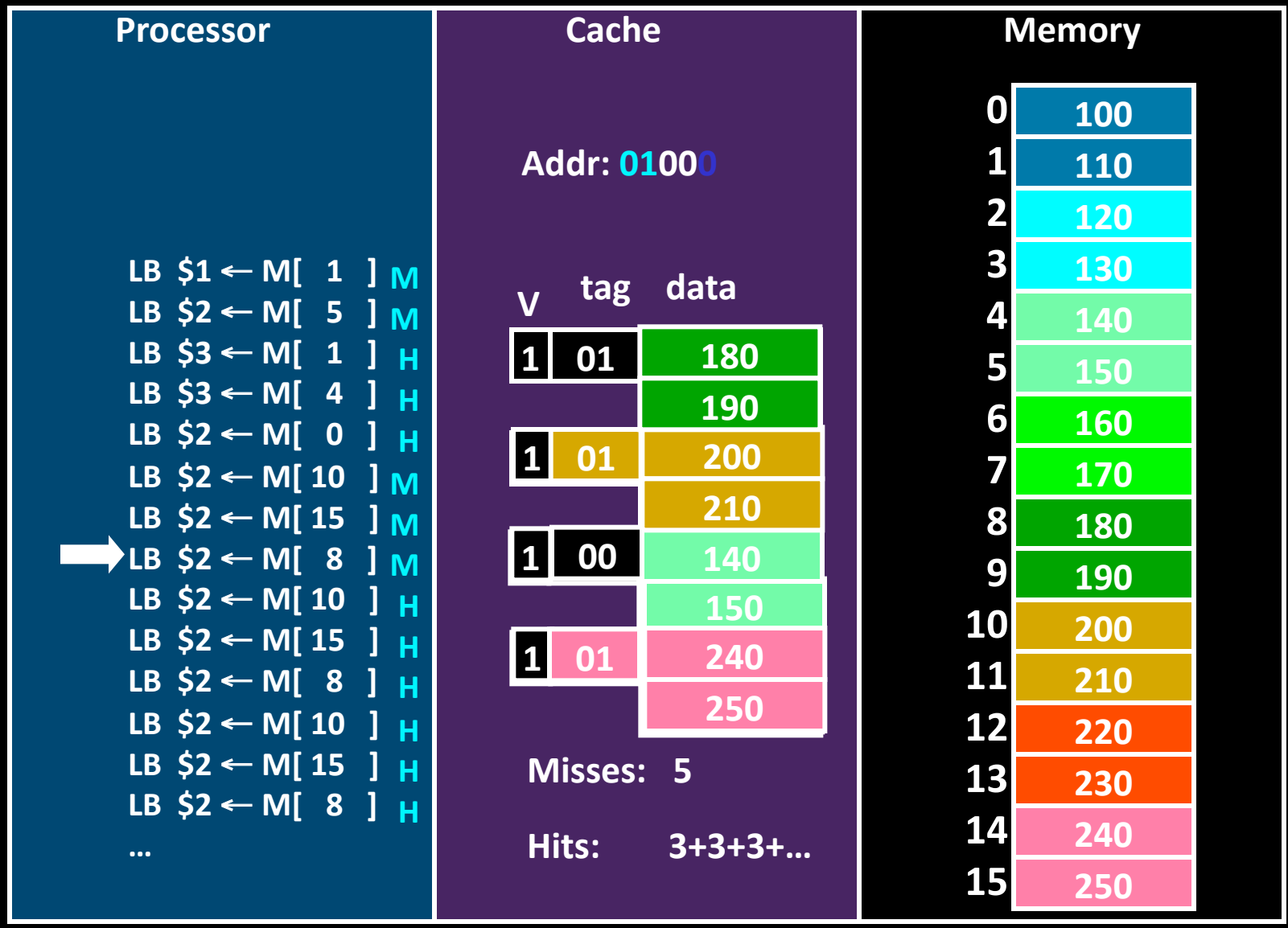
8th Access


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[10] M
LB $2 ← M[15] M
LB $2 ← M[8]
$0
$1 110
$2 250
$3 140
Cache
V tag data
1 00 100
110
1 01 200
210
1 00 140
150
1 01 240
250
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 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[10] M
LB $2 ← M[15] M
LB $2 ← M[8] M
$0
$1 110
$2 180
$3 140
Cache
Addr: 01000
V tag data
1 01 180
190
1 01 200
210
1 00 140
150
1 01 240
250
Misses: 5
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

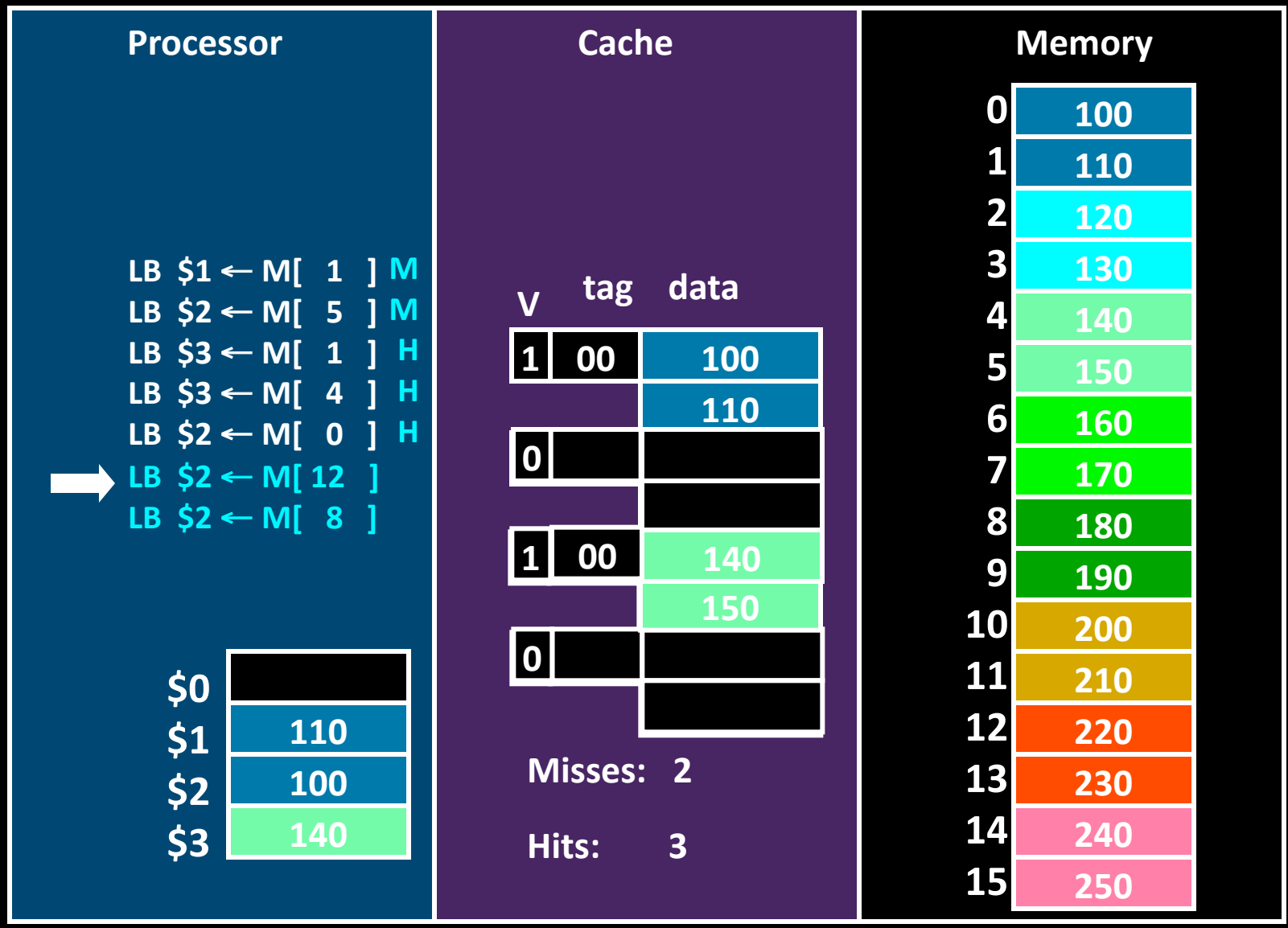
8th Access


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[10] M
LB $2 ← M[15] M
LB $2 ← M[8] M
LB $2 ← M[10] H
LB $2 ← M[15] H
LB $2 ← M[8] H
LB $2 ← M[10] H
LB $2 ← M[15] H
LB $2 ← M[8] H
…
Cache
Addr: 01000
V tag data
1 01 180
190
1 01 200
210
1 00 140
150
1 01 240
250
Misses: 5
Hits: 3+3+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

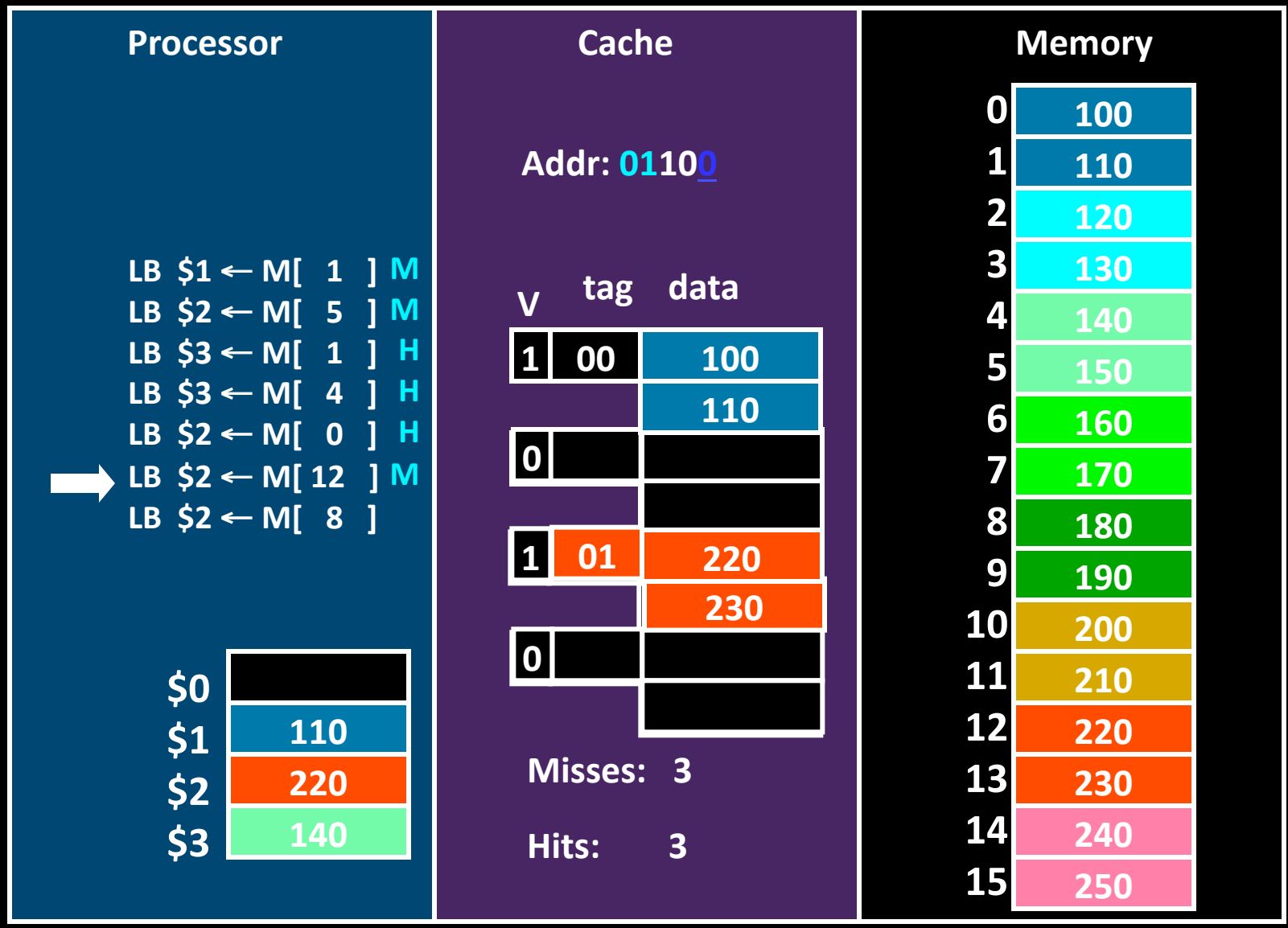
Pathological Case

Direct Mapped Example: 6th Access

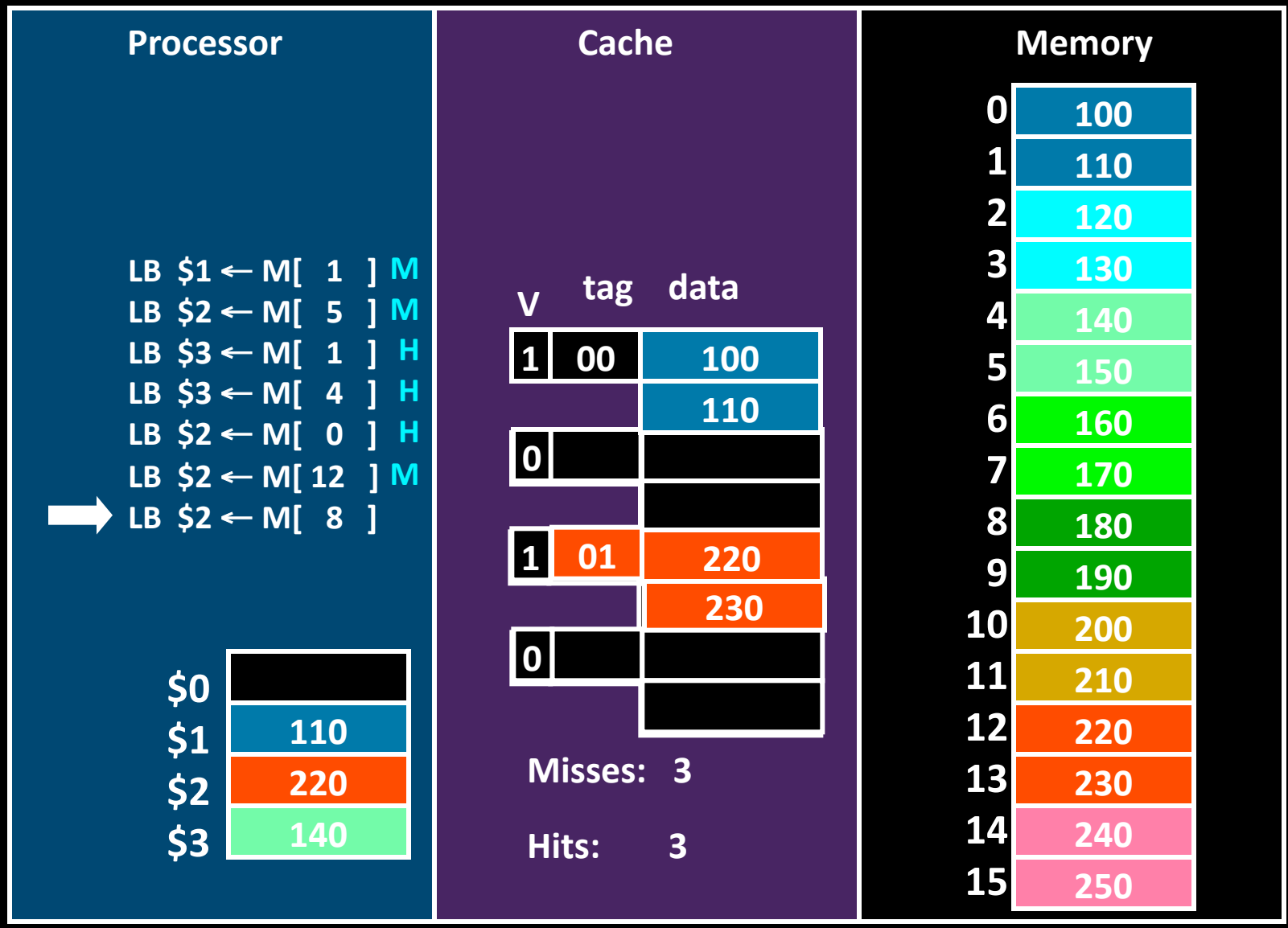
Pathological example



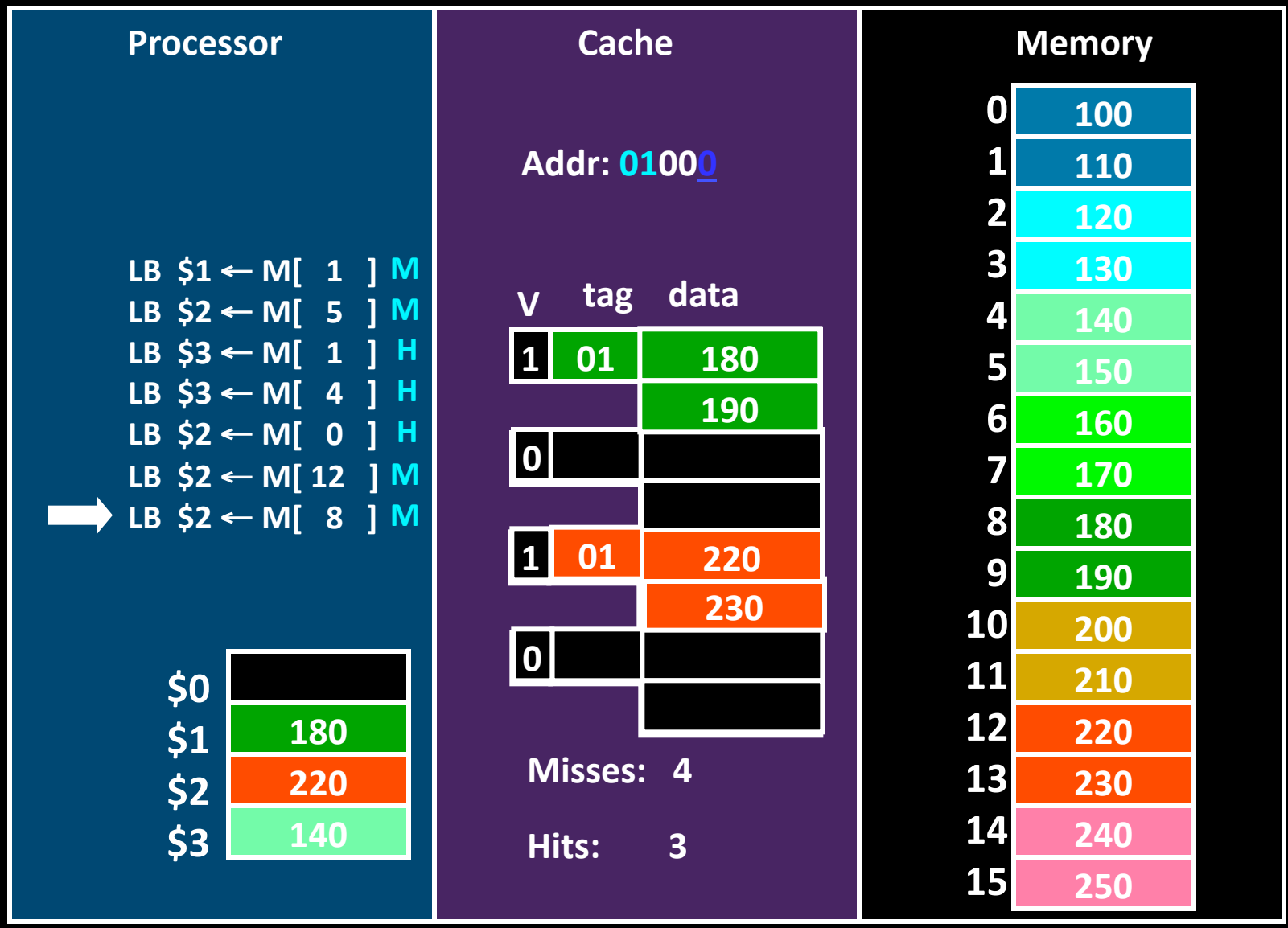
6th Access


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[8]
$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

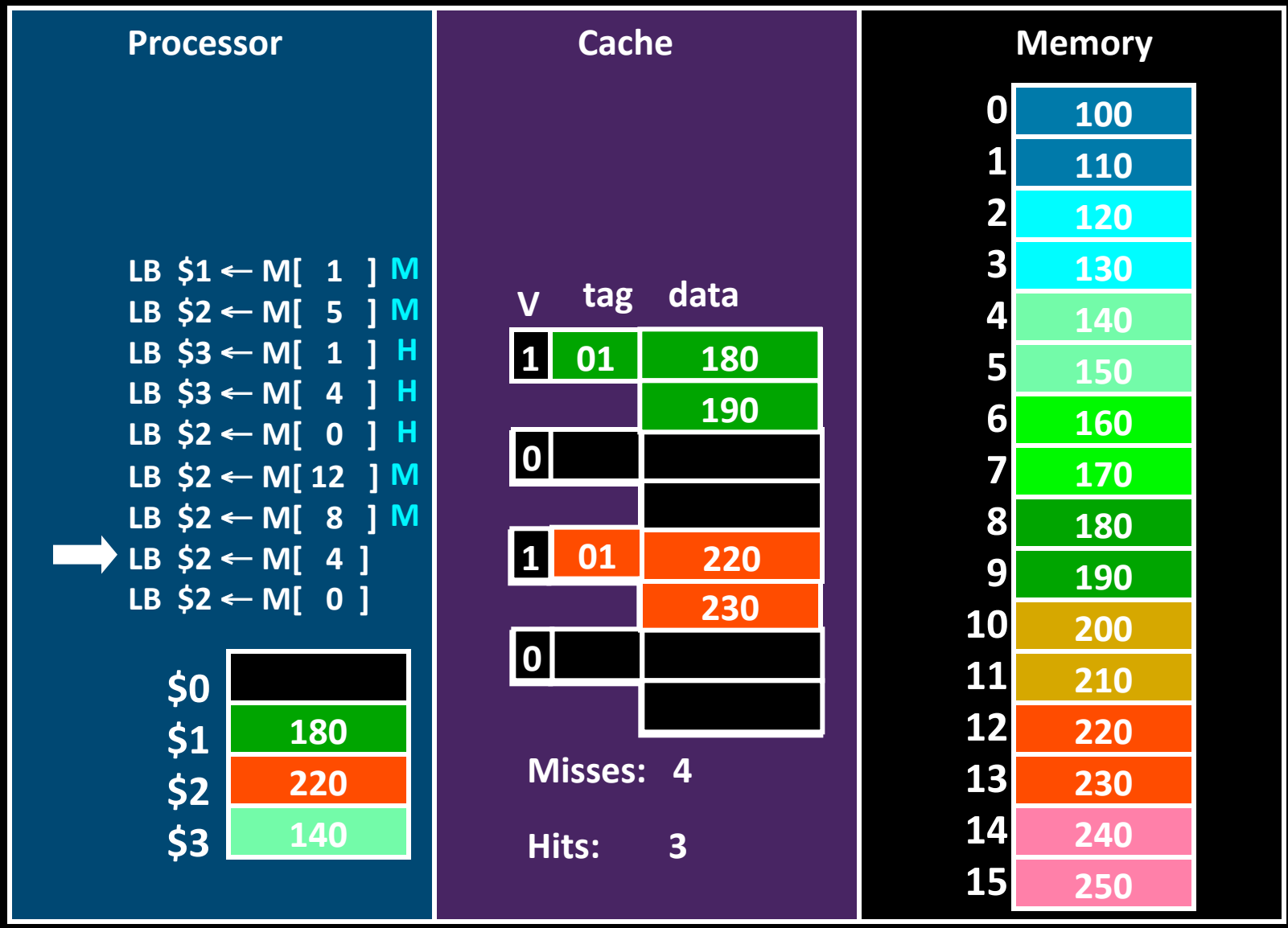
7th Access


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[8]
$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

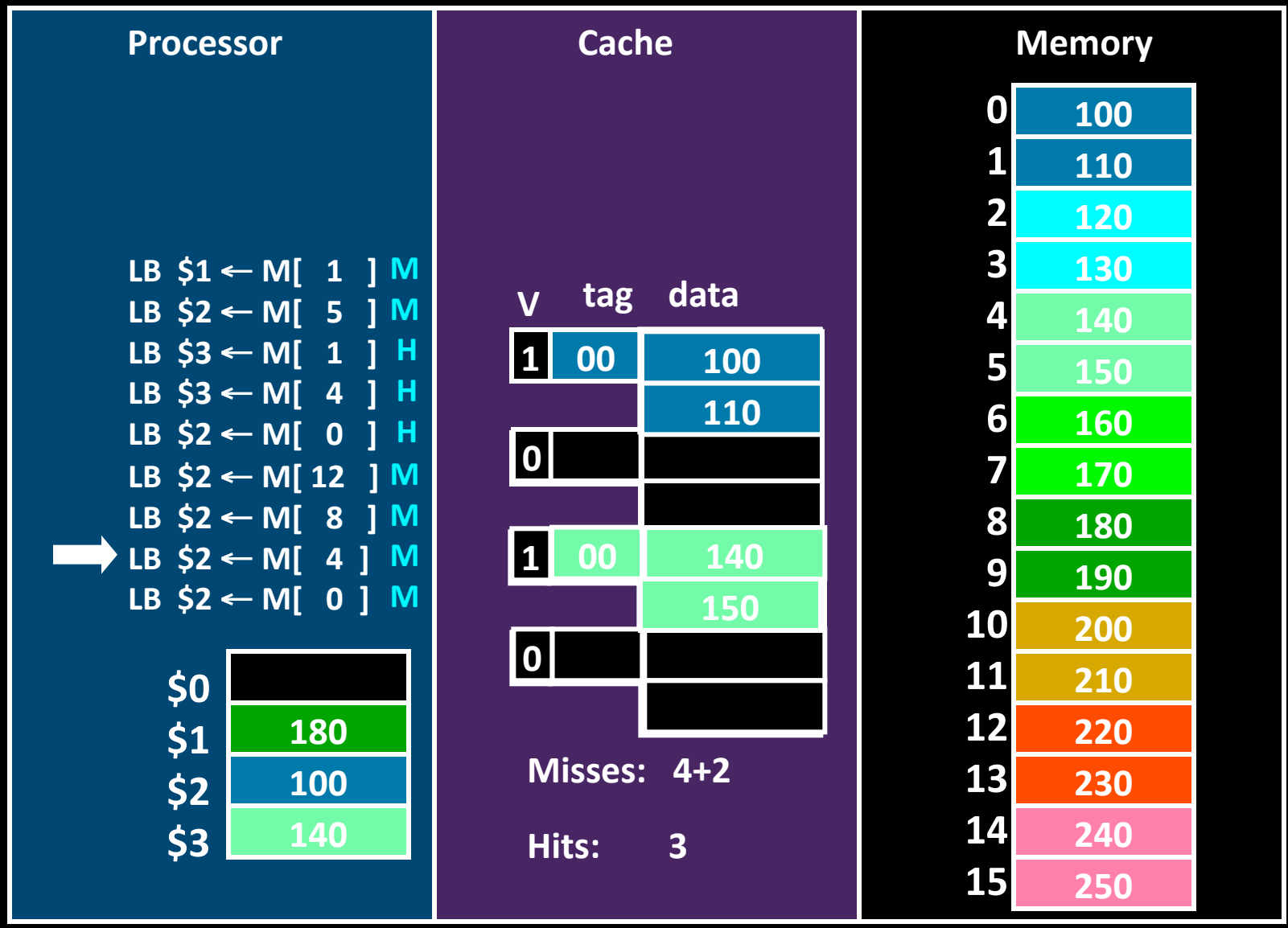
7th Access


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[8] M
$0
$1 180
$2 220
$3 140
Cache
Addr: 01000
V tag data
1 01 180
190
0
1 01 220
230
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

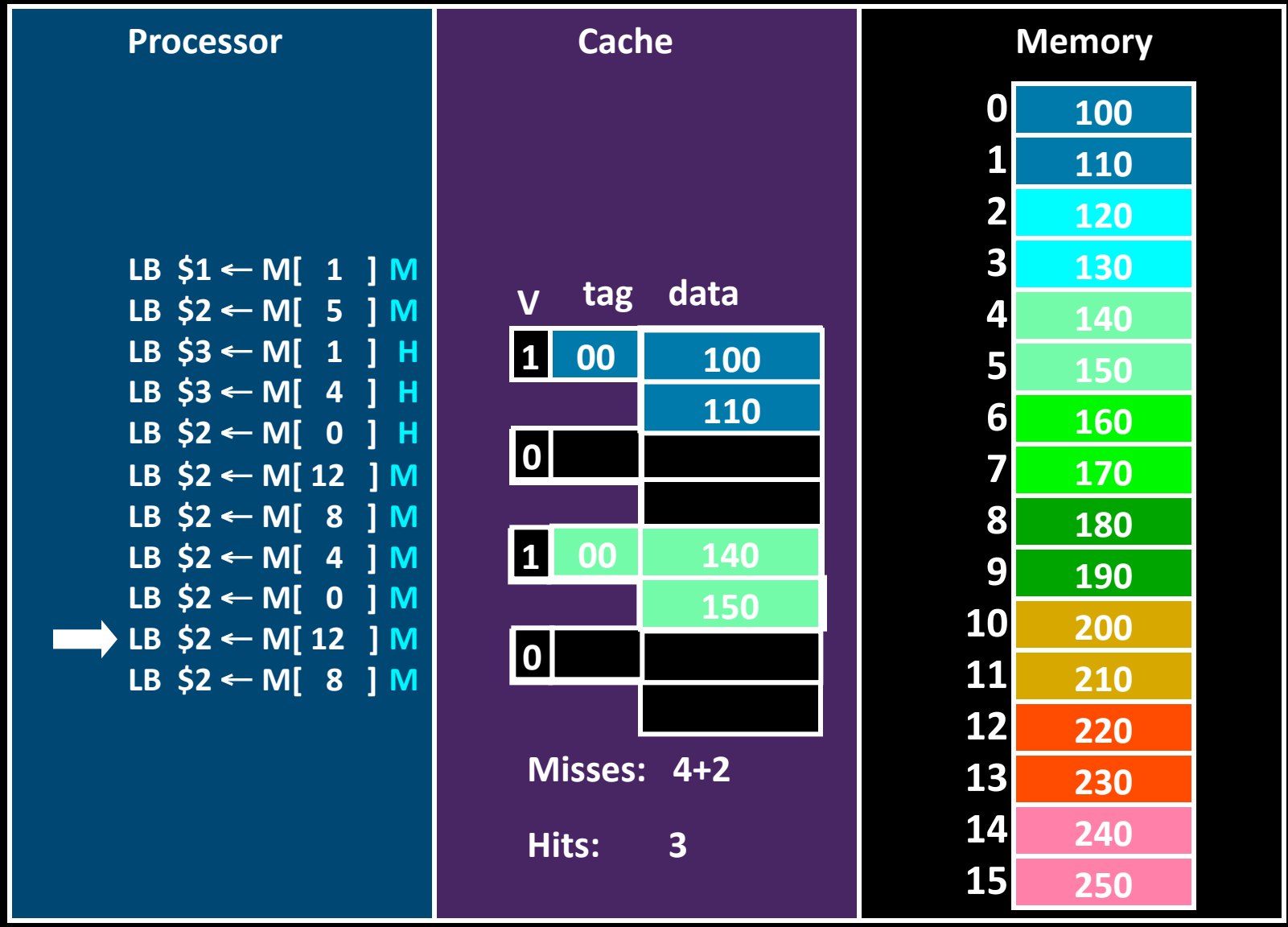
8th and 9th Access



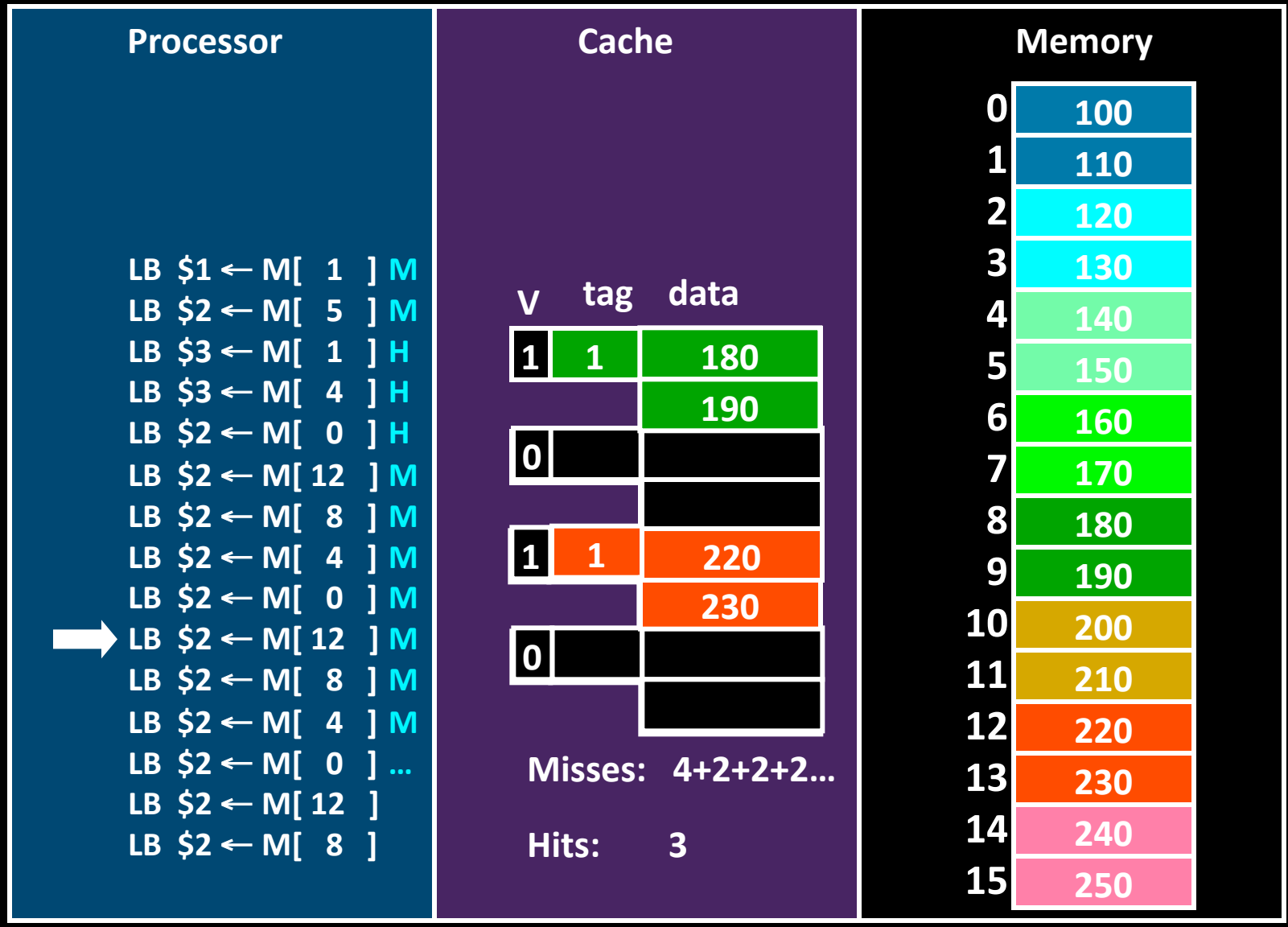
8th and 9th Access



10th and 11th Access



10th and 11th Access



Problem

Working set is not too big for cache

Yet, we can’t make it work