

Caches (Writing)

Hakim Weatherspoon

CS 3410, Spring 2013

Computer Science

Cornell University

P & H Chapter 5.2-3, 5.5

Goals for Today: caches

Writing to the Cache

- Write-through vs Write-back

Cache Parameter Tradeoffs

Cache Conscious Programming

Writing with Caches

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

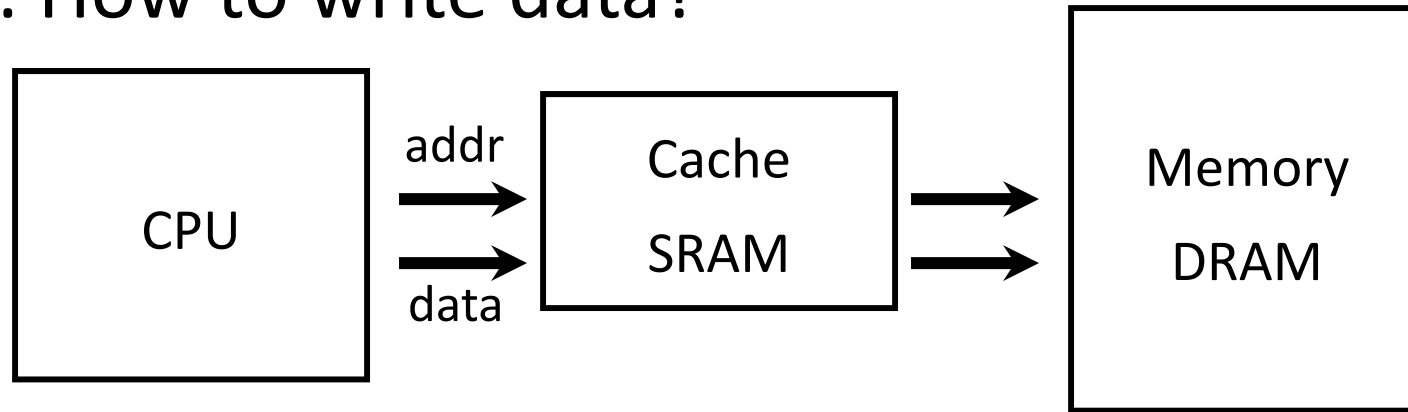
Next Goal

What about writes?

What happens when the CPU writes to a register and calls a store instruction?!

Cached Write Policies

Q: How to write data?



If data is already in the cache...

No-Write

- writes invalidate the cache and go directly to memory

Write-Through

- writes go to main memory and cache

Write-Back

- CPU writes only to cache
- cache writes to main memory later (when block is evicted)

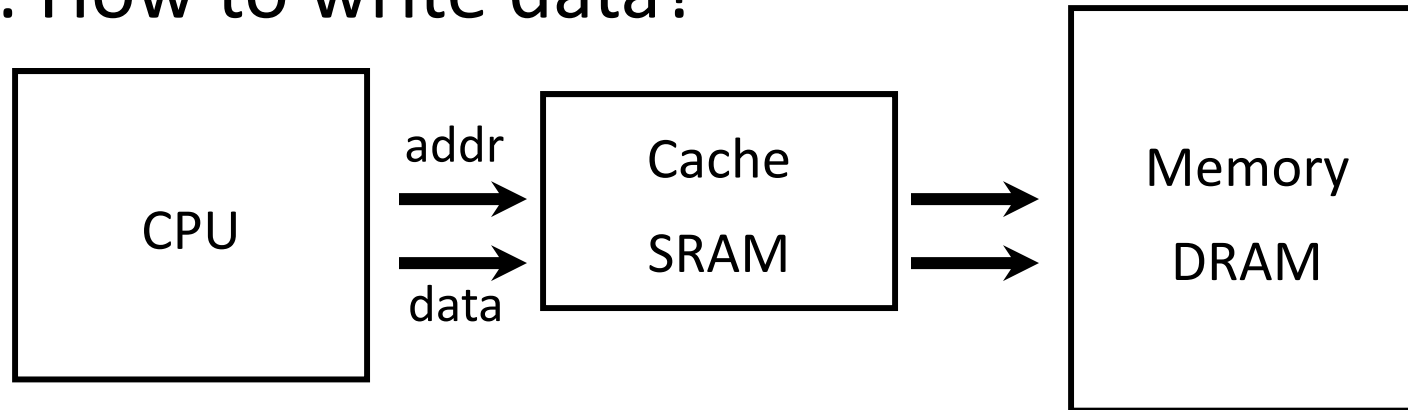
What about Stores?

Where should you write the result of a store?

- If that memory location is in the cache?
 - Send it to the cache
 - Should we also send it to memory right away?
(write-through policy)
 - Wait until we kick the block out (write-back policy)
- If it is not in the cache?
 - Allocate the line (put it in the cache)?
(write allocate policy)
 - Write it directly to memory without allocation?
(no write allocate policy)

Write Allocation Policies

Q: How to write data?



If data is not in the cache...

Write-Allocate

- allocate a cache line for new data (and maybe write-through)

No-Write-Allocate

- ignore cache, just go to main memory

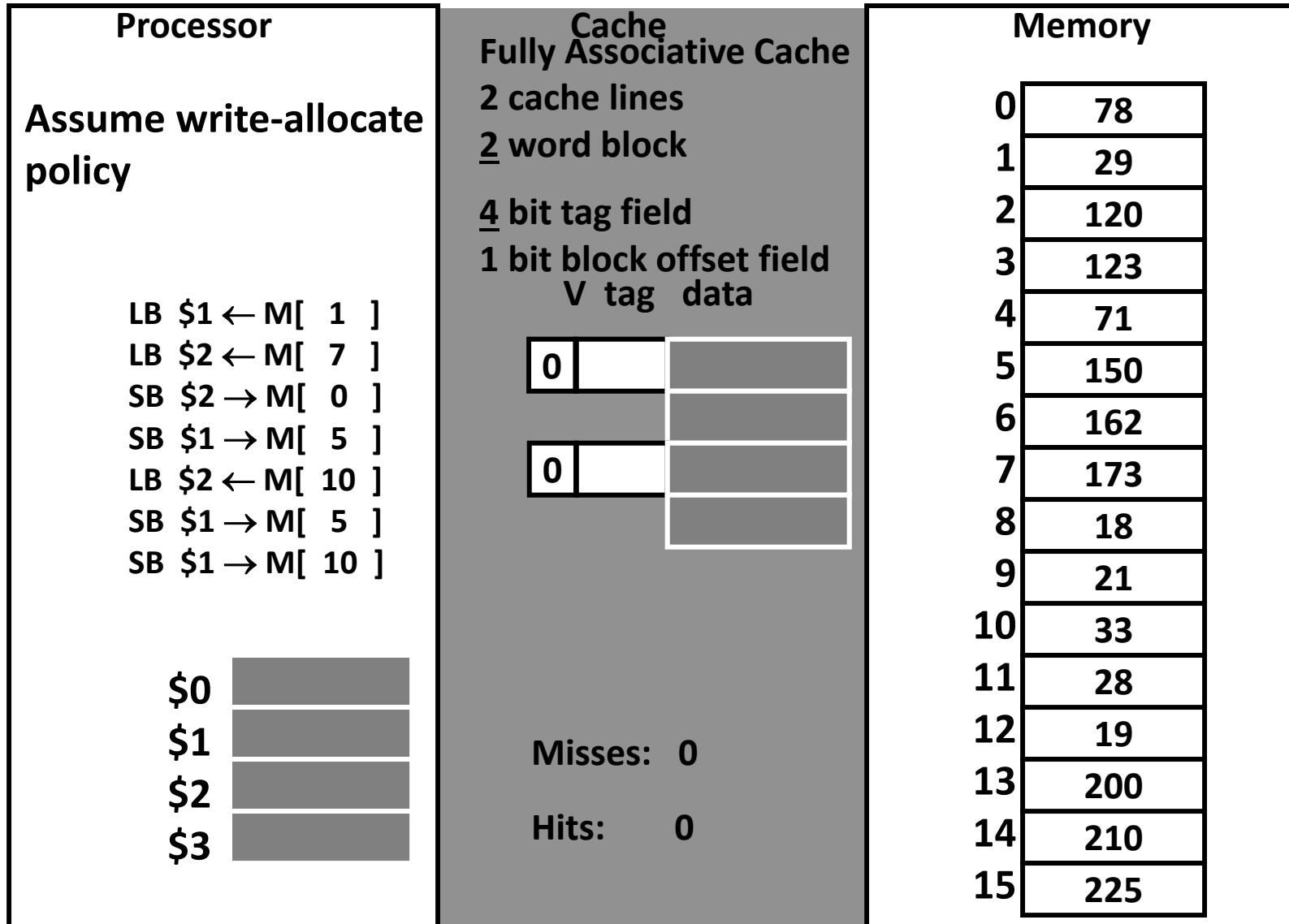
Next Goal

Example: How does a write-through cache work?

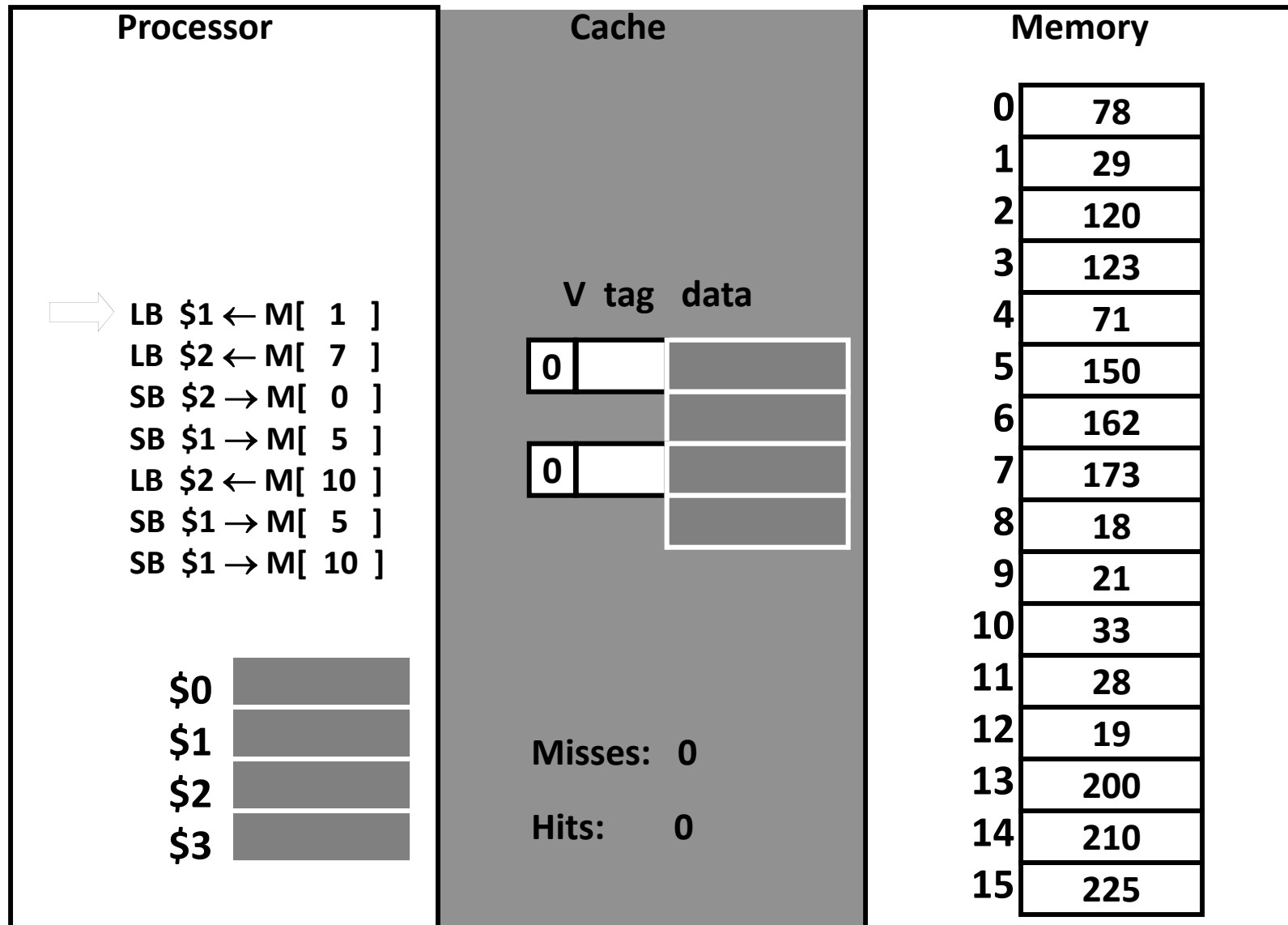
Assume write-allocate.

Handling Stores (Write-Through)

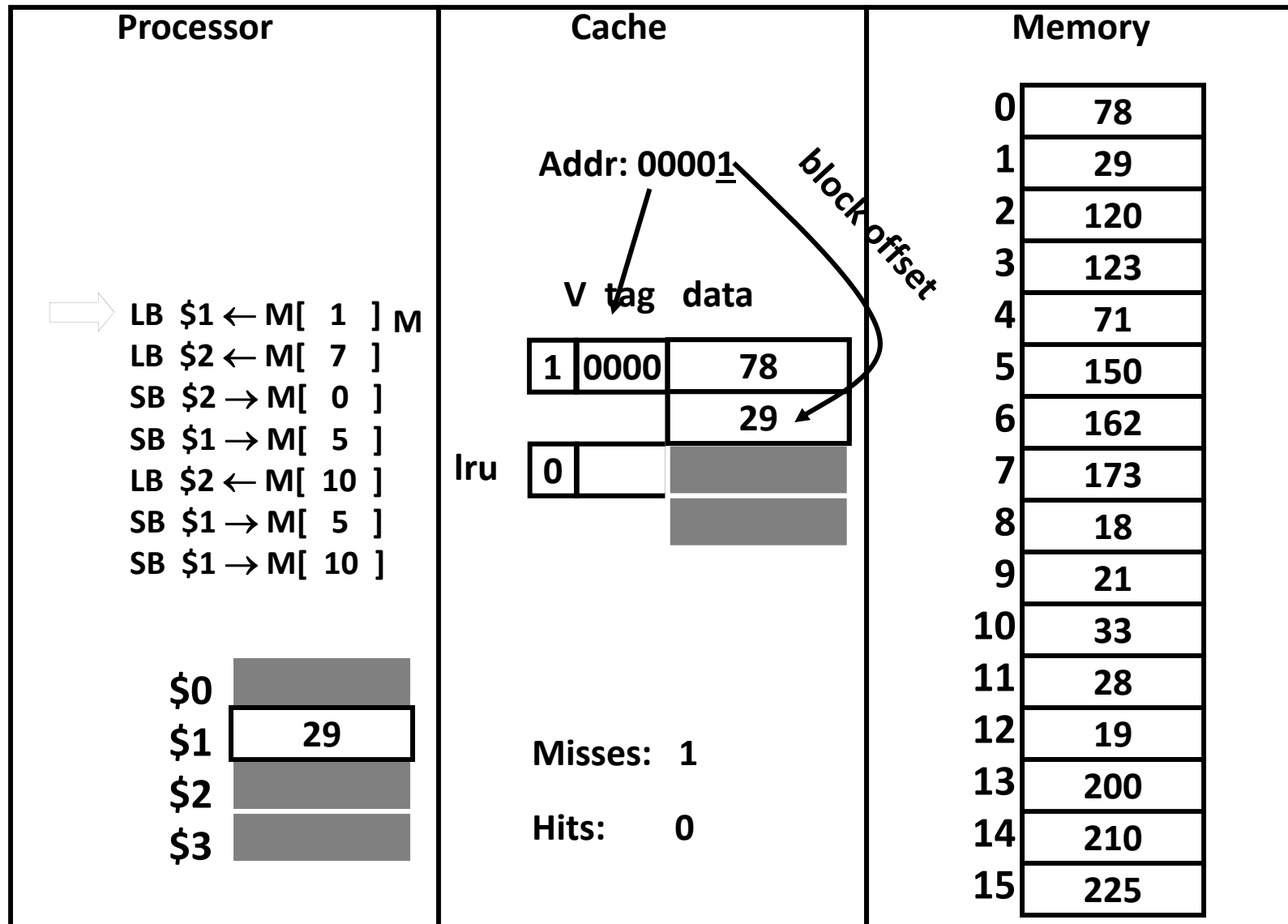
Using **byte addresses** in this example! Addr Bus = 5 bits



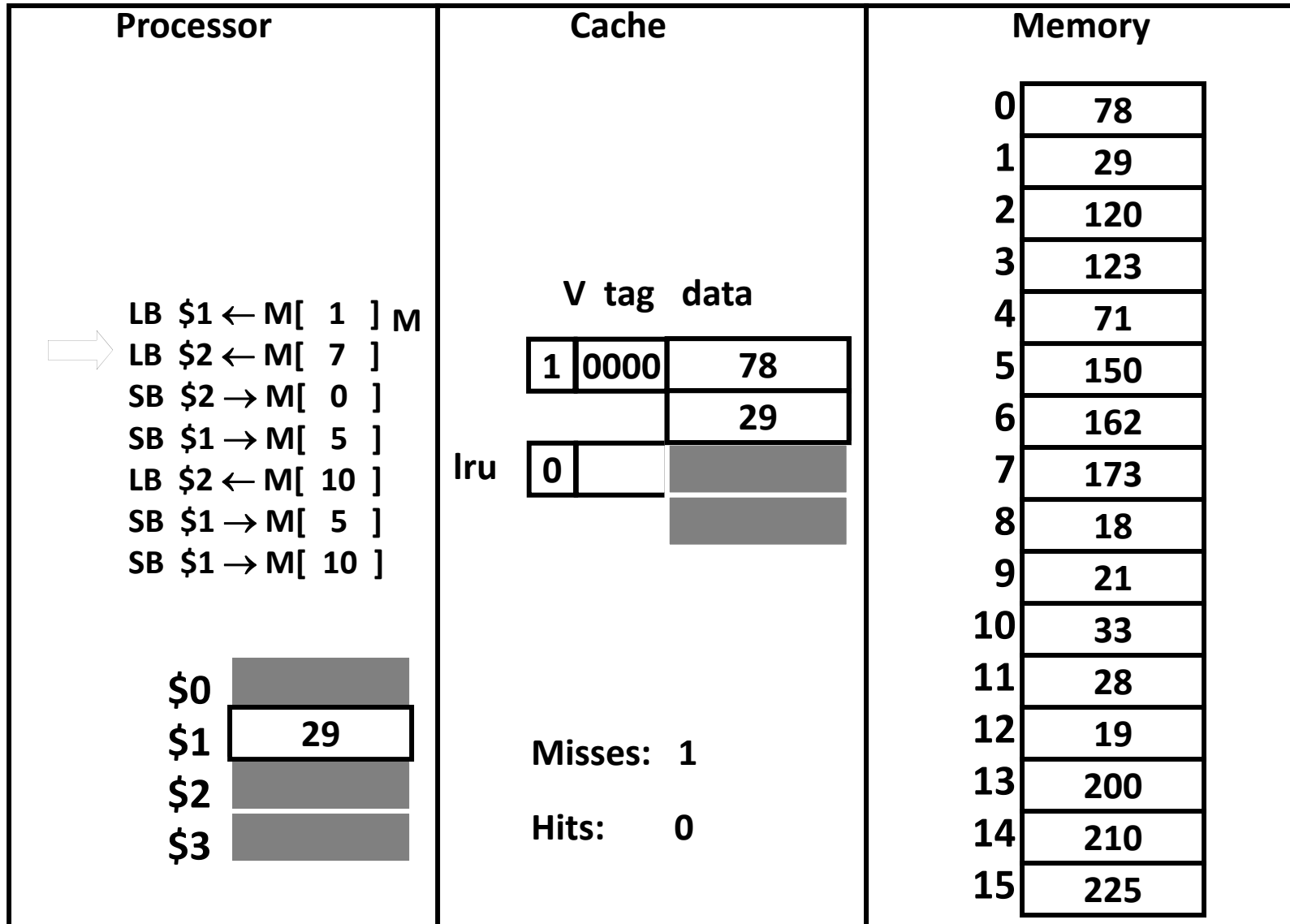
Write-Through (REF 1)



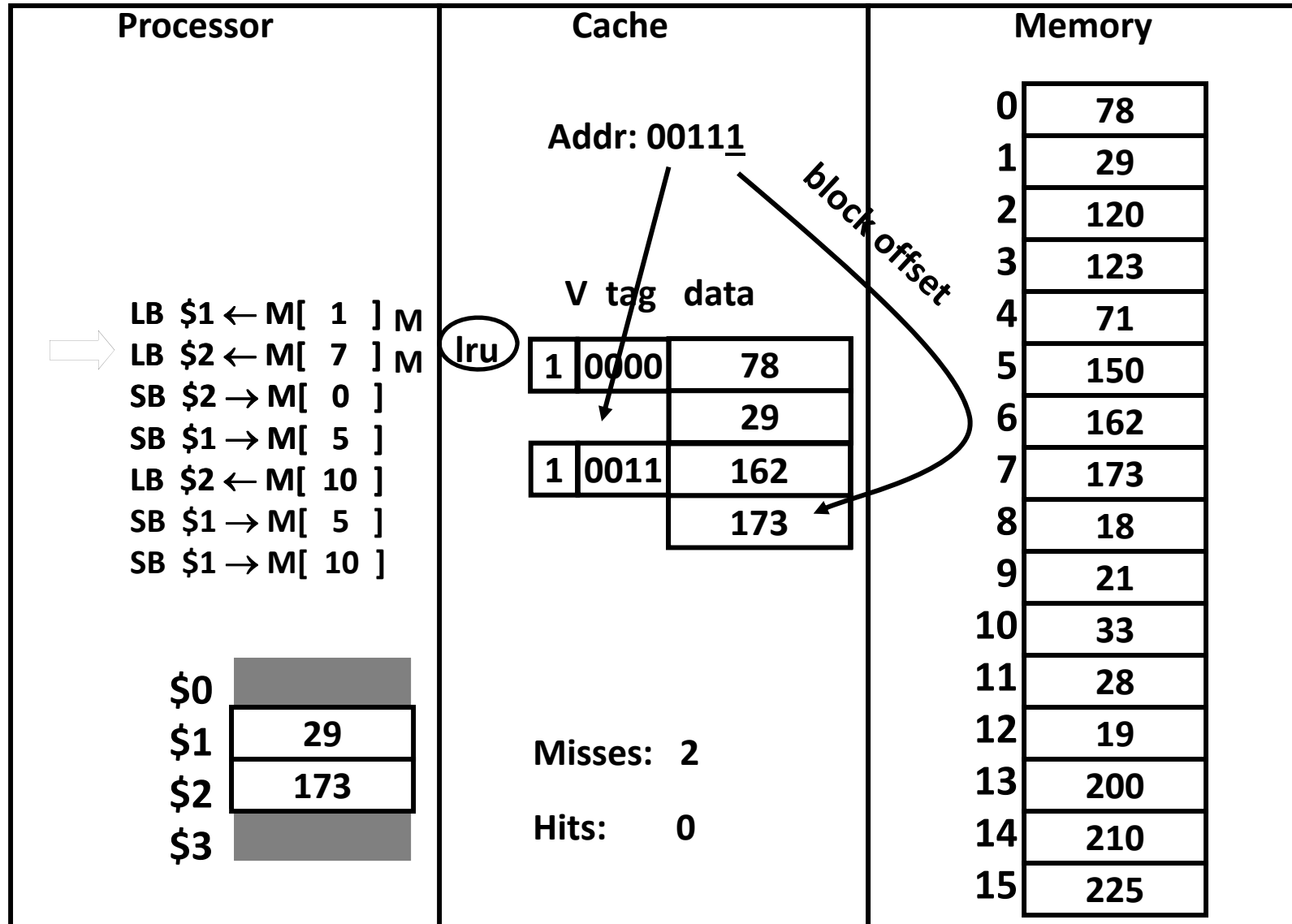
Write-Through (REF 1)



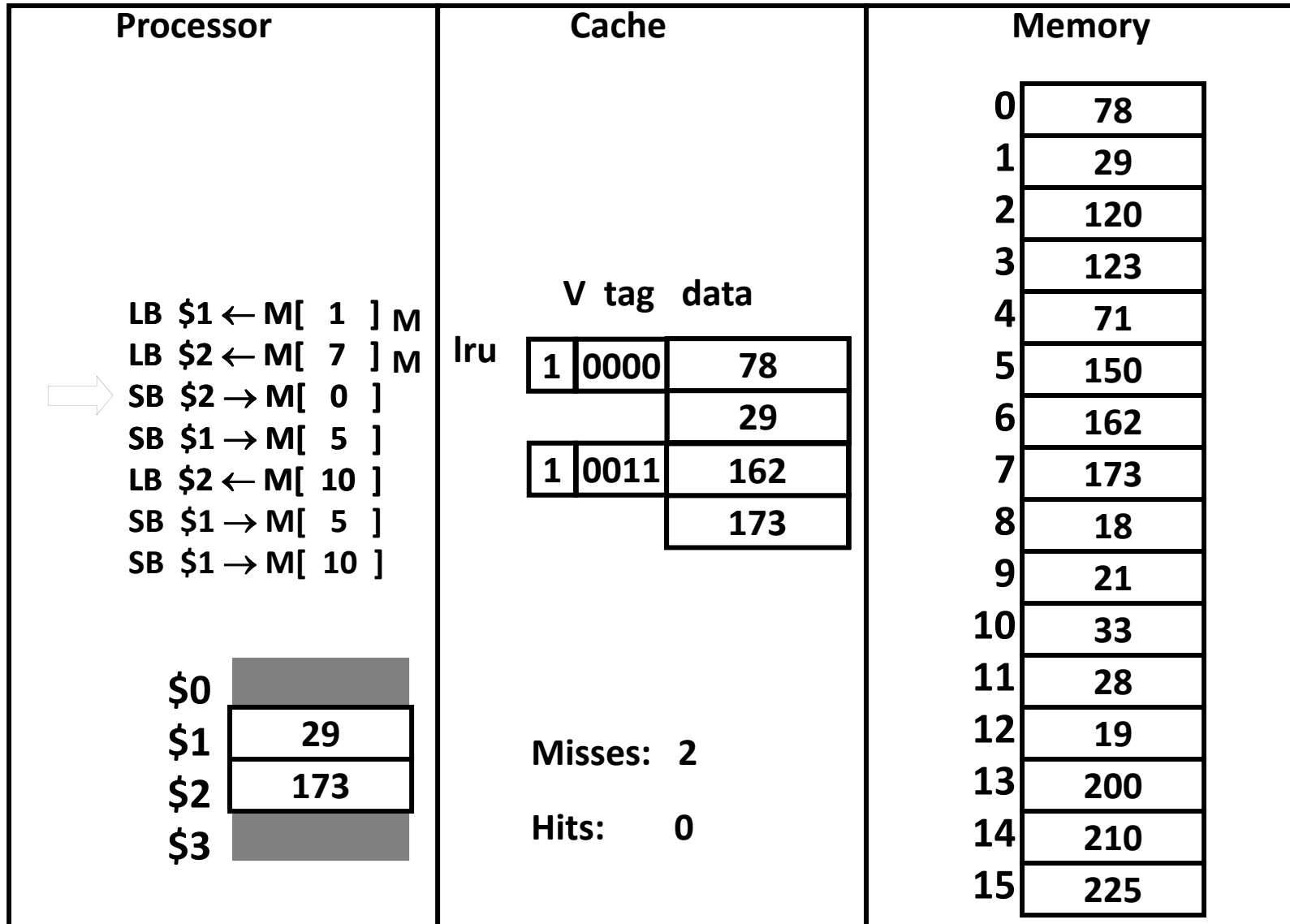
Write-Through (REF 2)



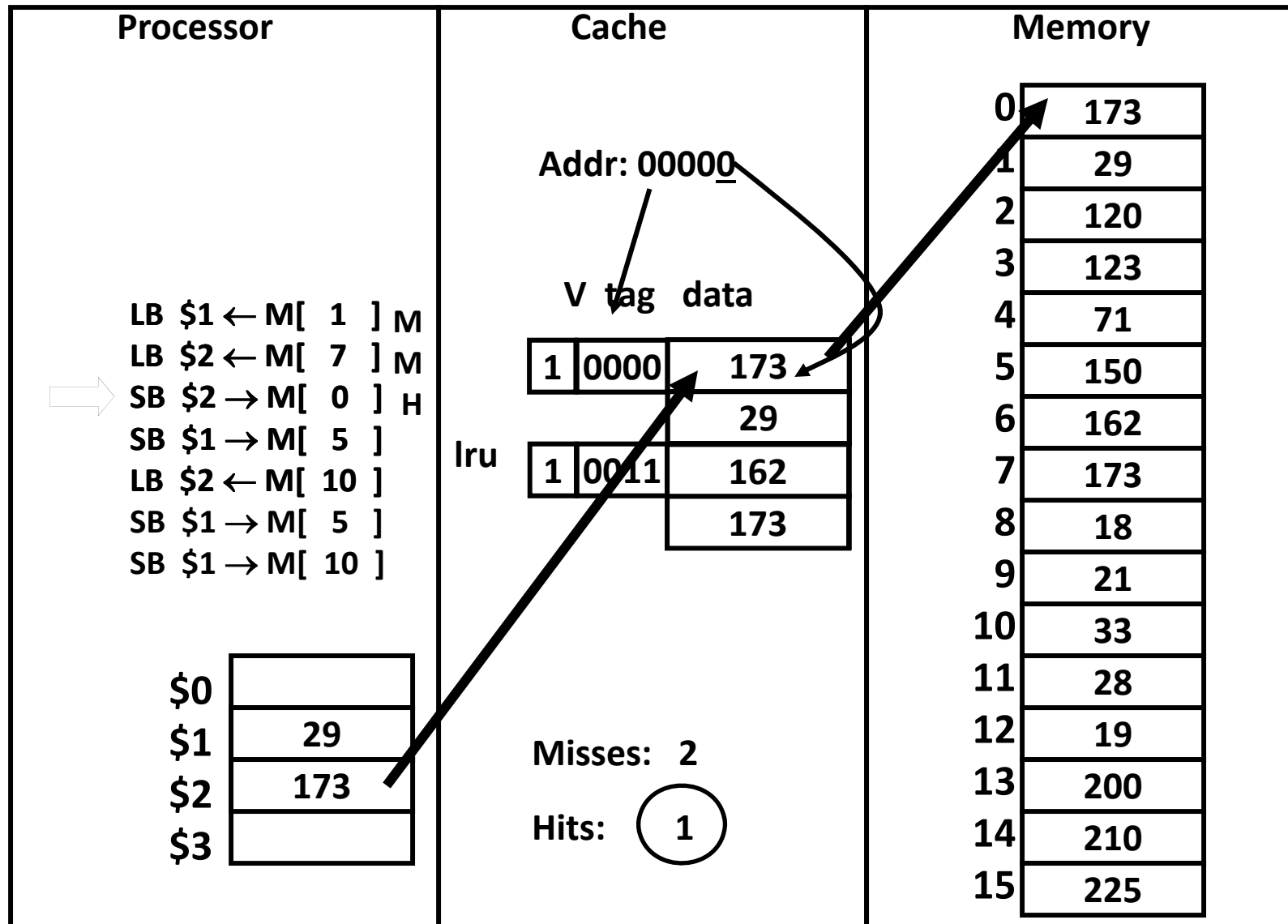
Write-Through (REF 2)



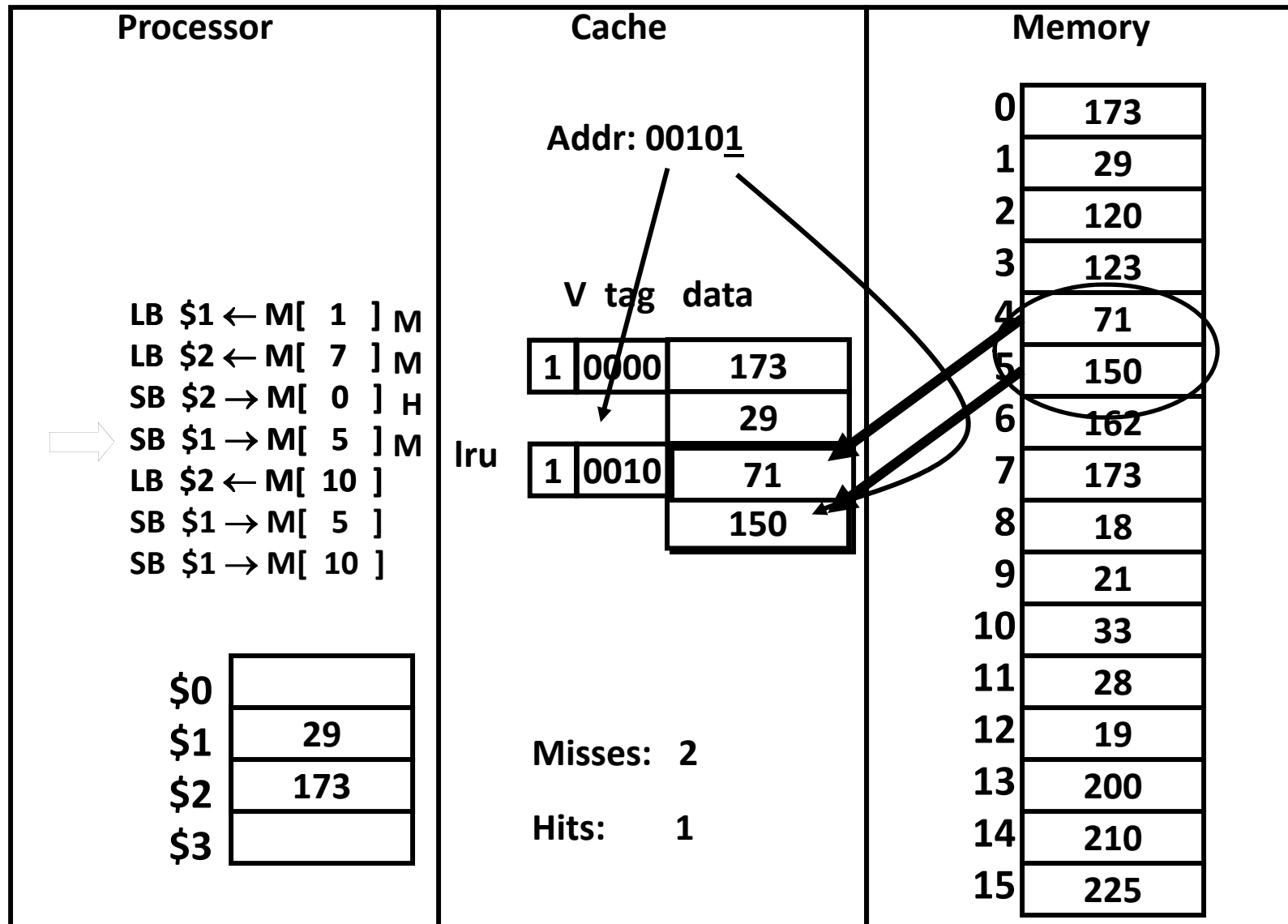
Write-Through (REF 3)



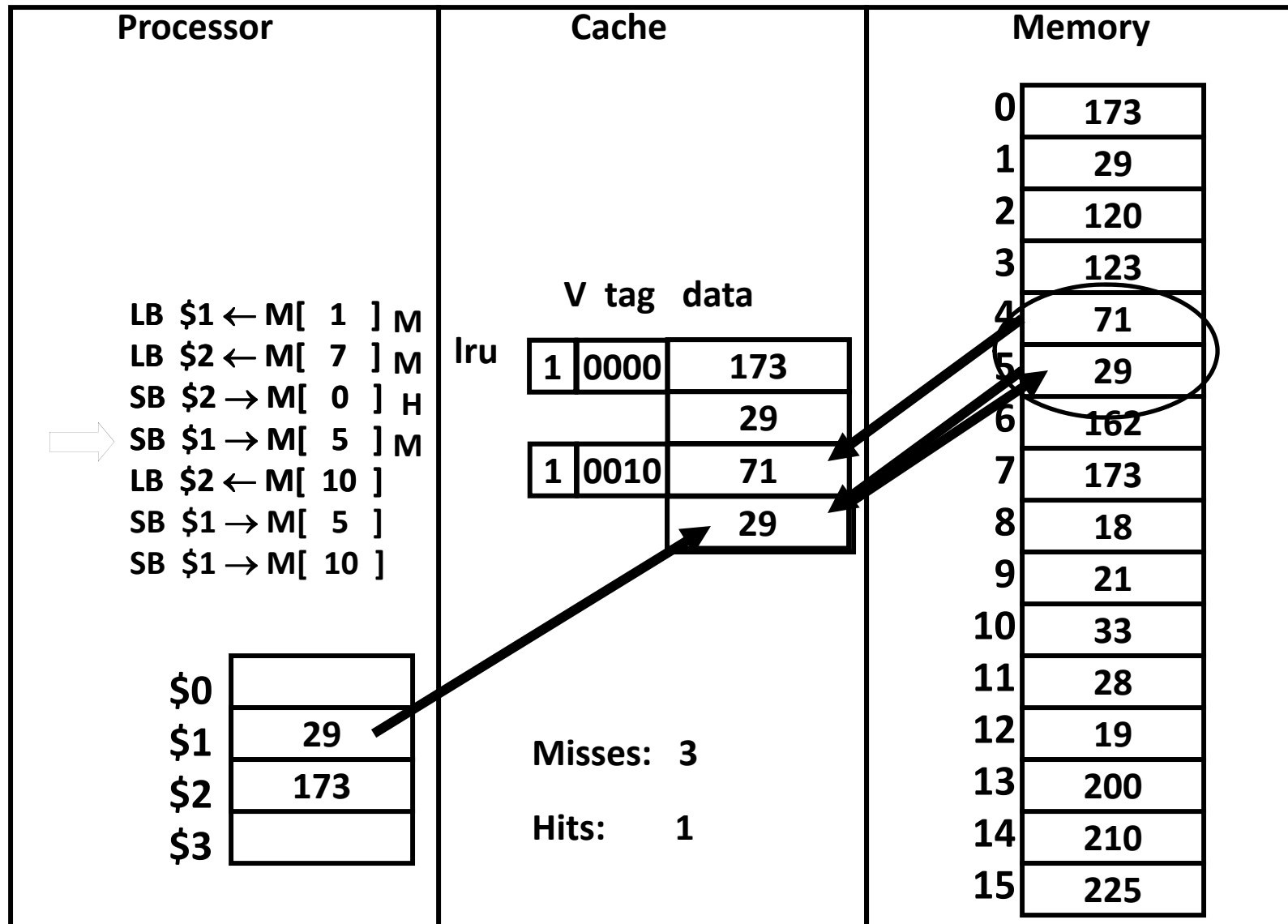
Write-Through (REF 3)



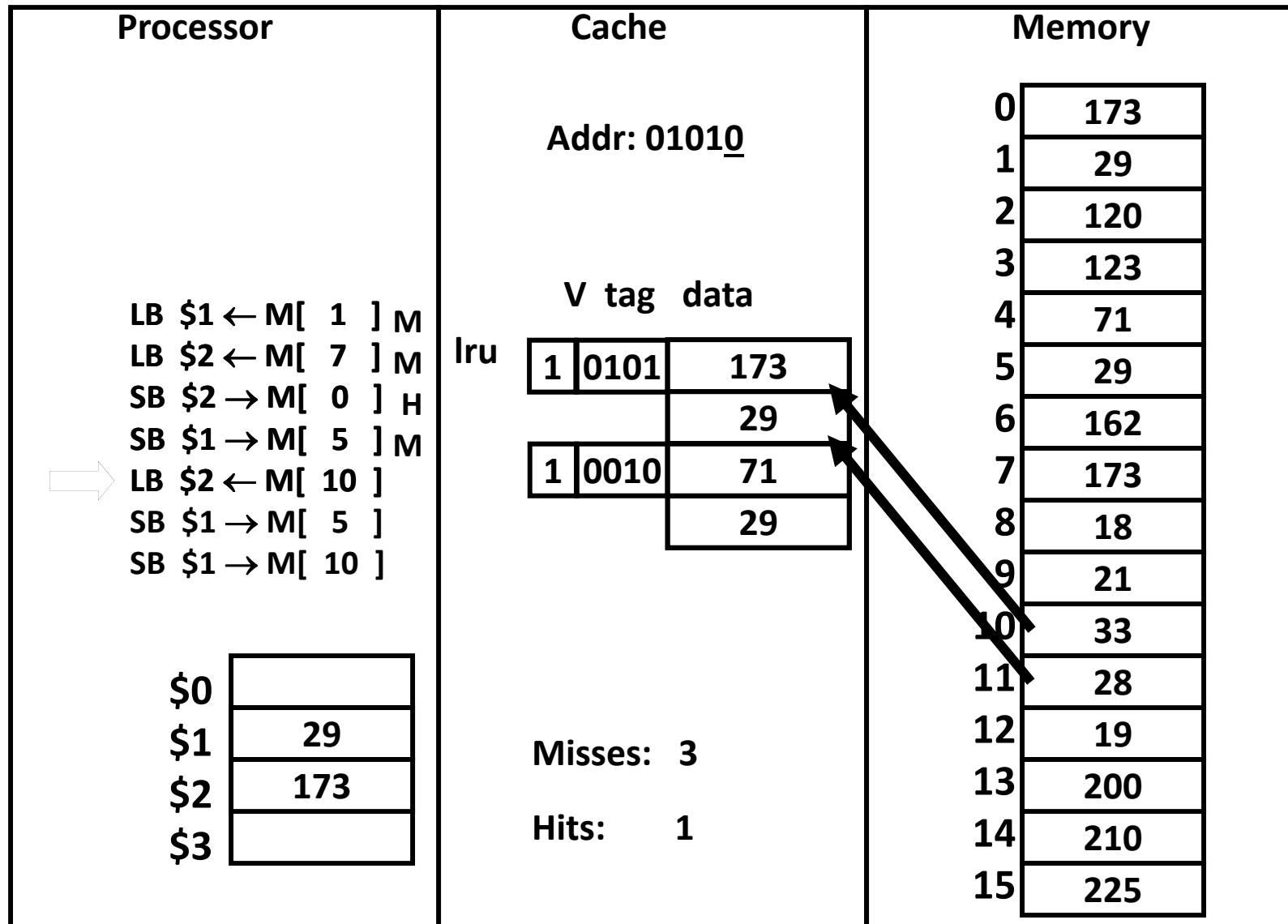
Write-Through (REF 4)



Write-Through (REF 4)



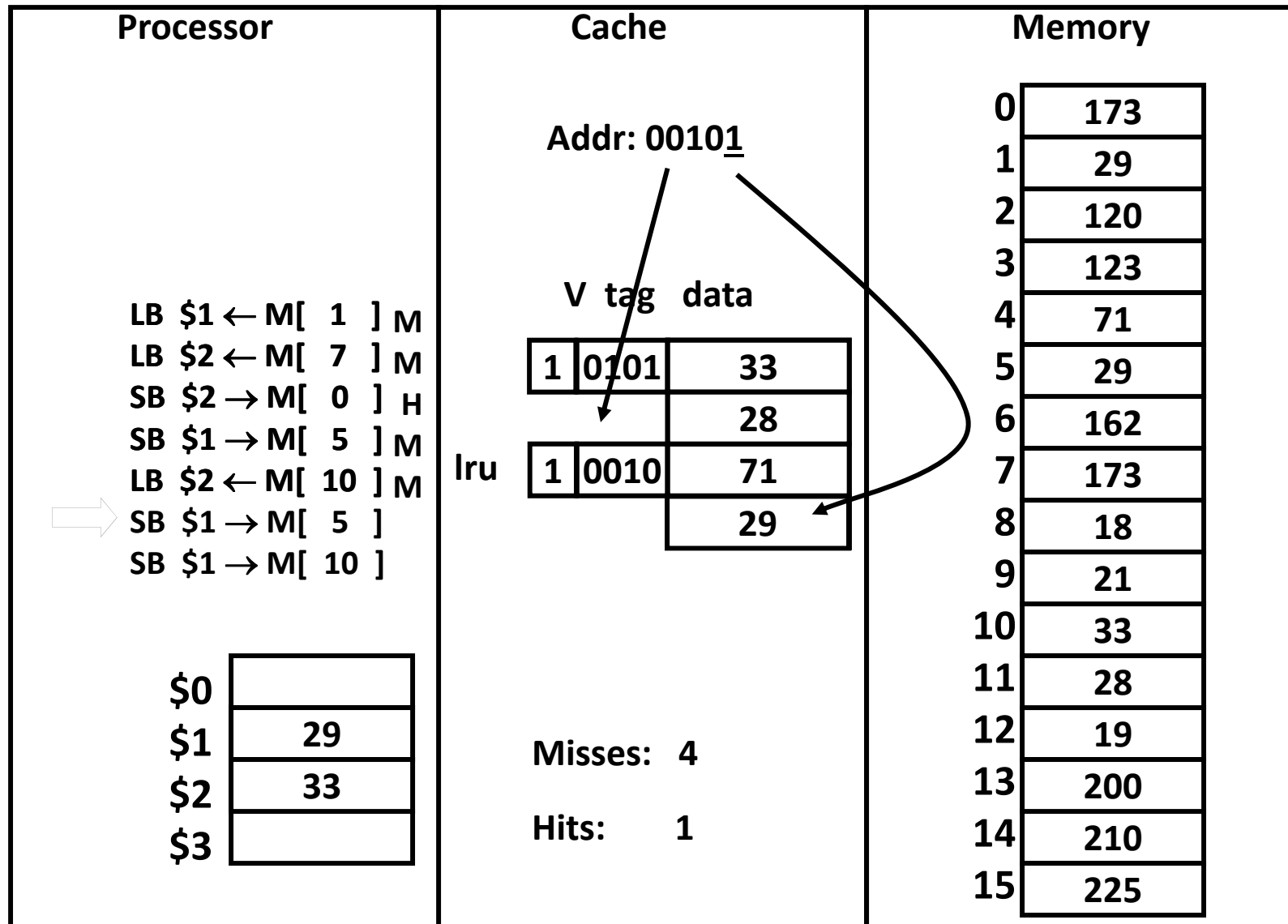
Write-Through (REF 5)



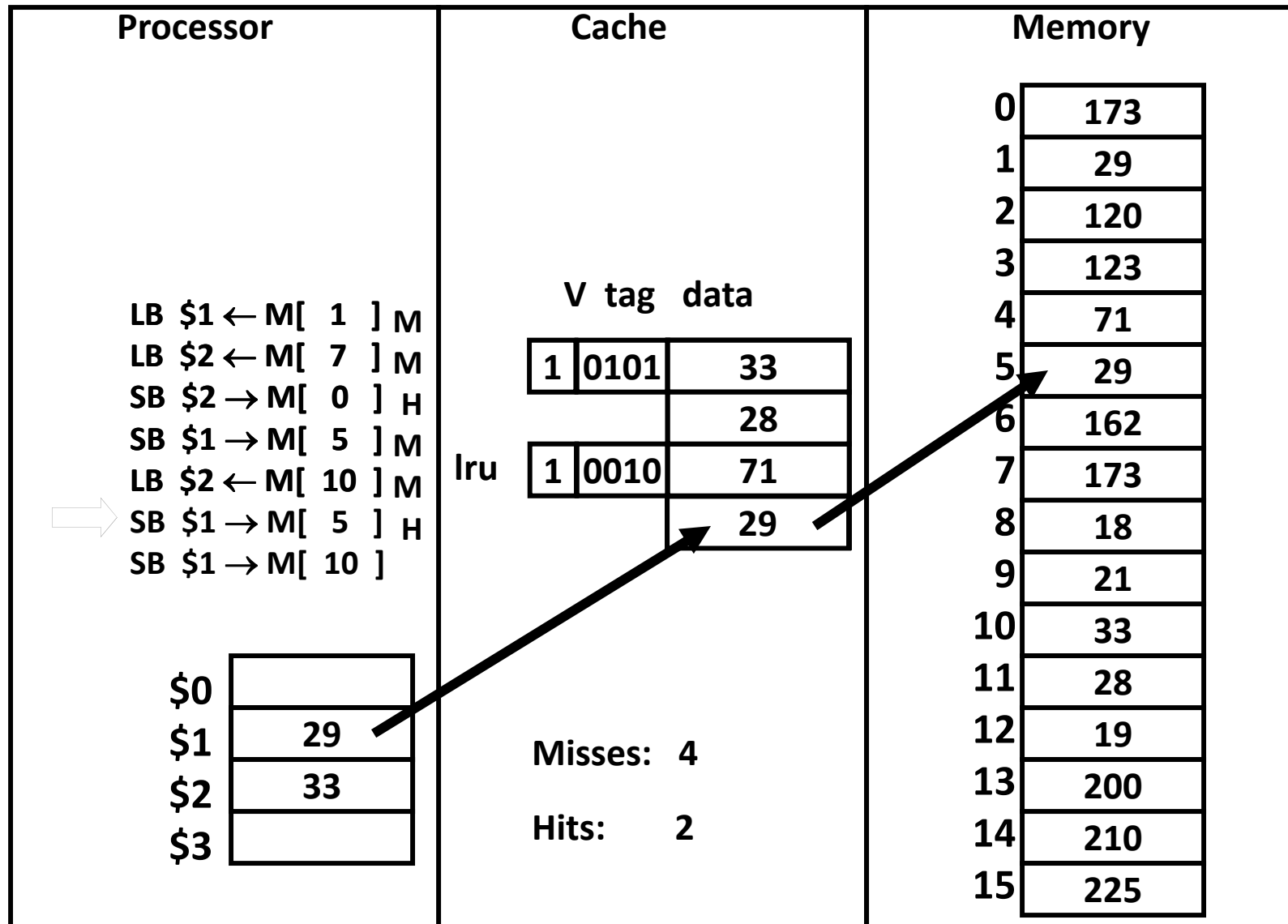
Write-Through (REF 5)

Processor	Cache	Memory
LB \$1 ← M[1] M LB \$2 ← M[7] M SB \$2 → M[0] H SB \$1 → M[5] M LB \$2 ← M[10] M SB \$1 → M[5] SB \$1 → M[10] \$0 \$129 \$233 \$3	Vtagdata 1010133 28 1001071 29 lru Misses: 4 Hits: 1	0173 129 2120 3123 471 529 6162 7173 818 921 1033 1128 1219 13200 14210 15225

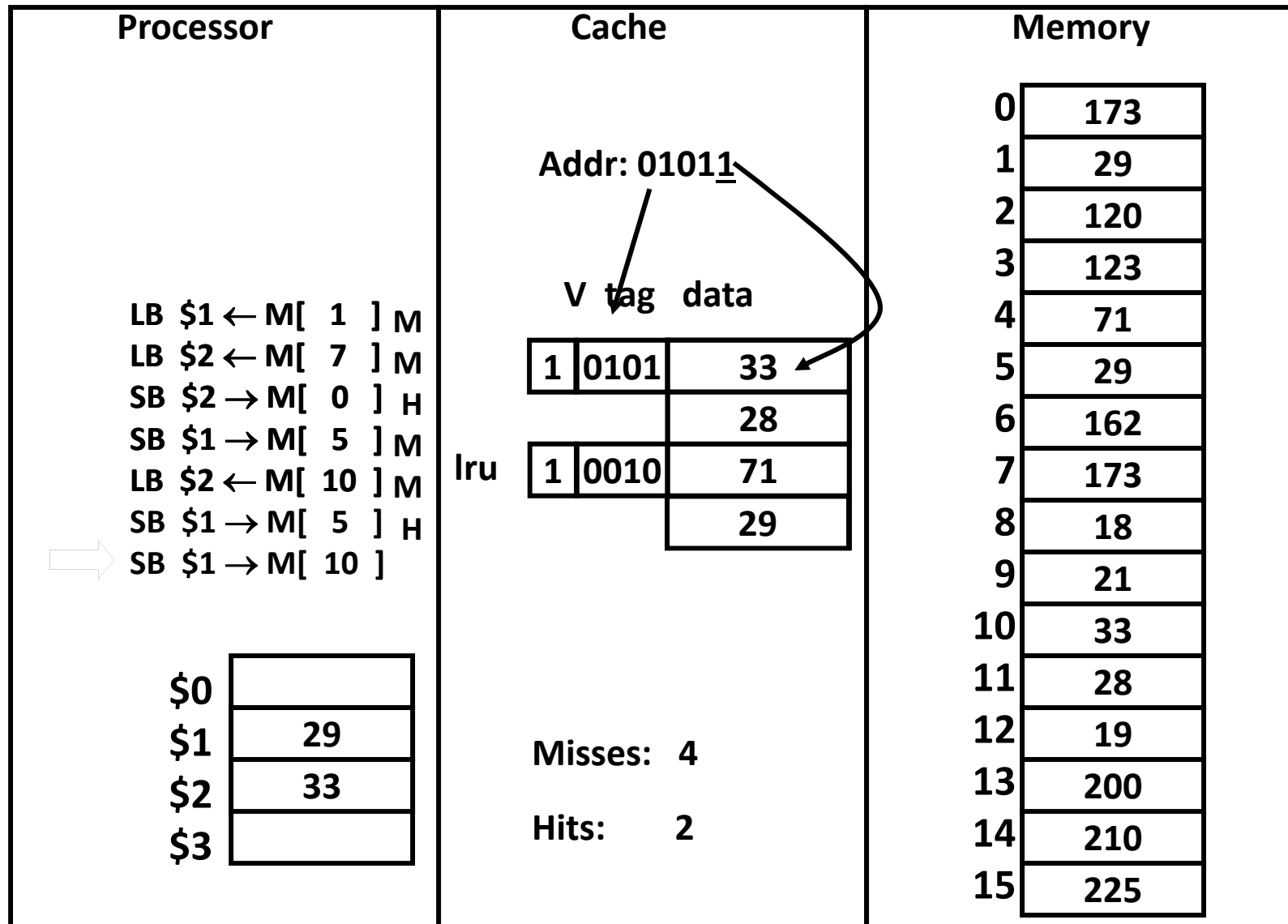
Write-Through (REF 6)



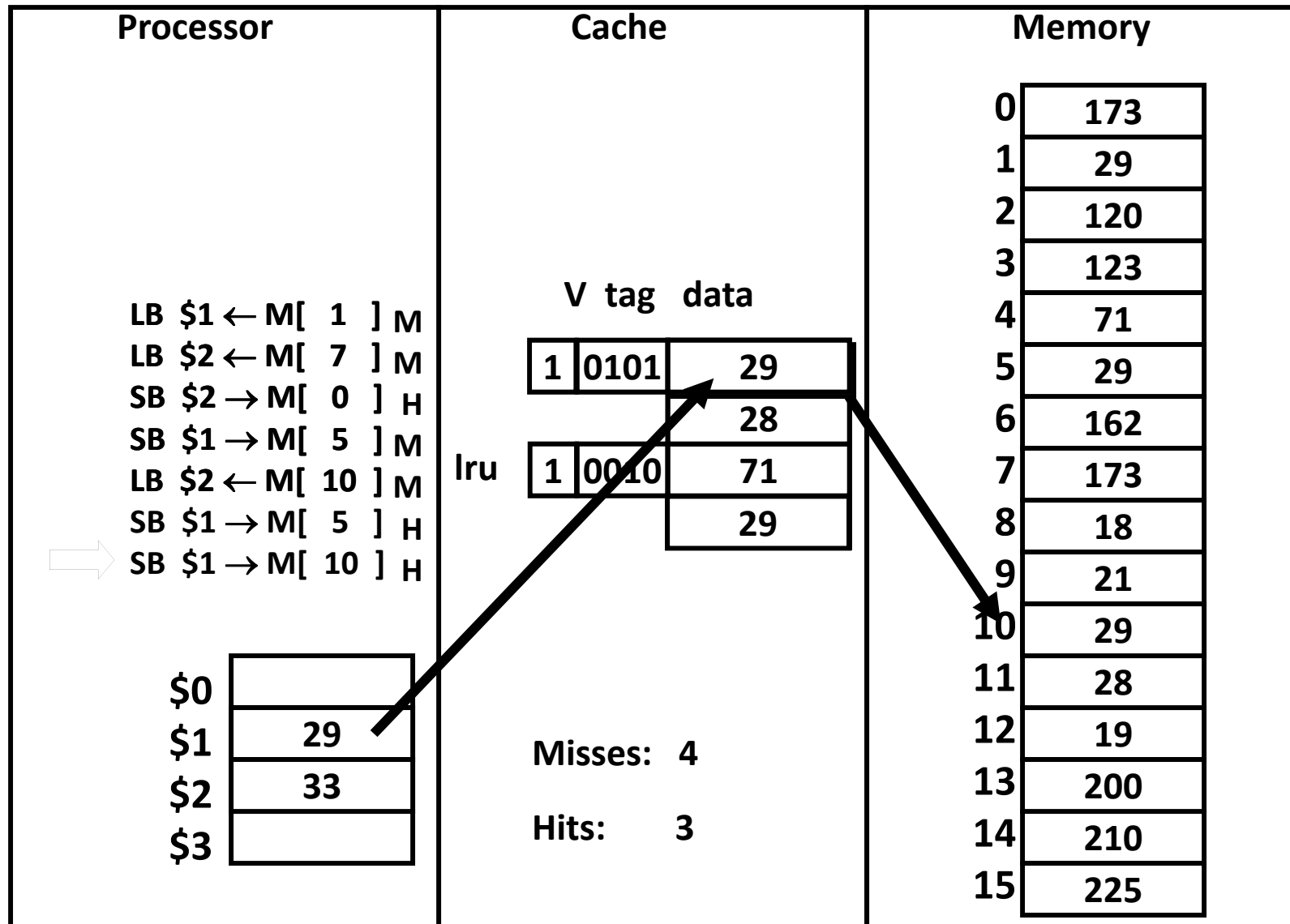
Write-Through (REF 6)



Write-Through (REF 7)



Write-Through (REF 7)



How Many Memory References?

Write-through performance

Each miss (read or write) reads a block from mem

- 4 misses → 8 mem reads

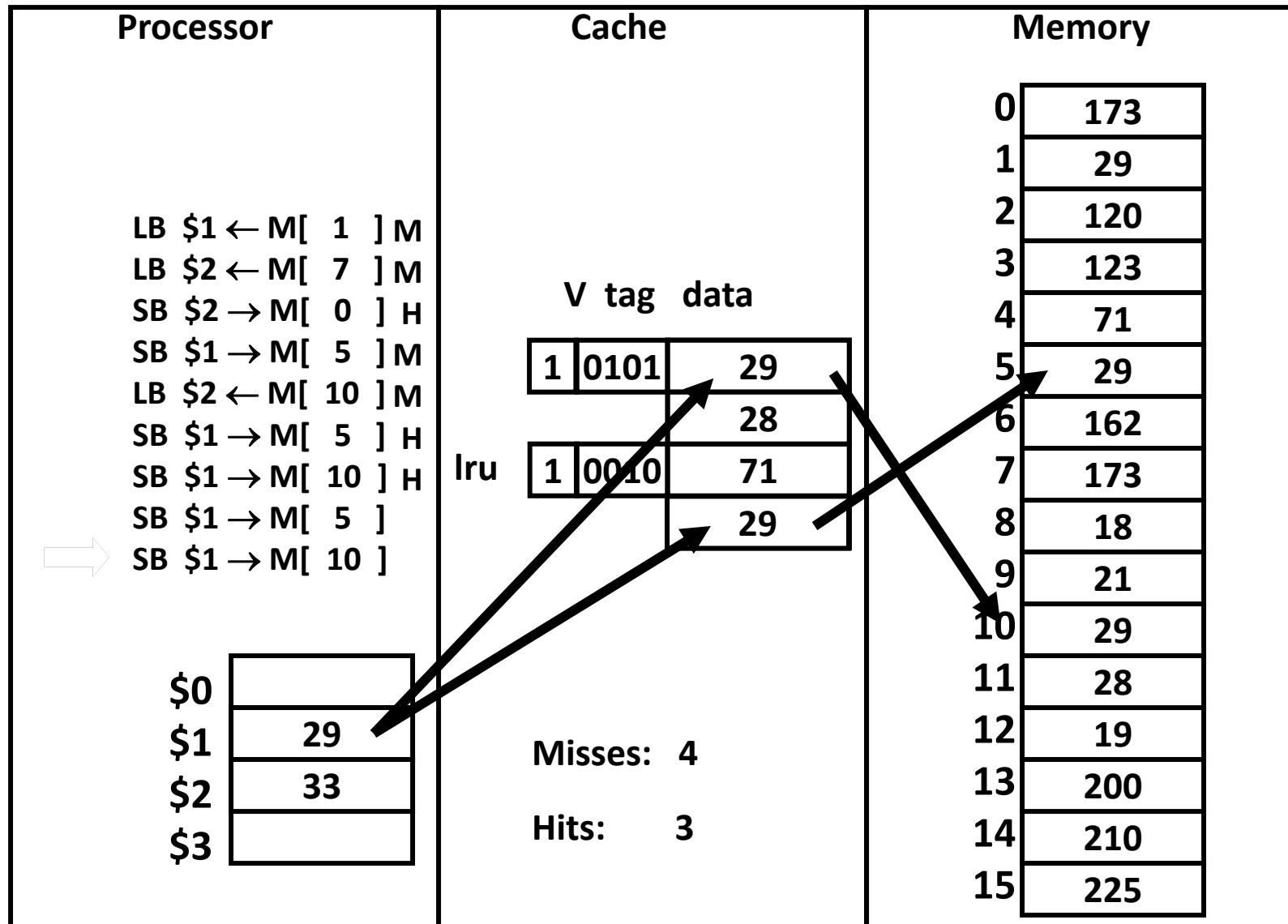
Each store writes an item to mem

- 4 mem writes

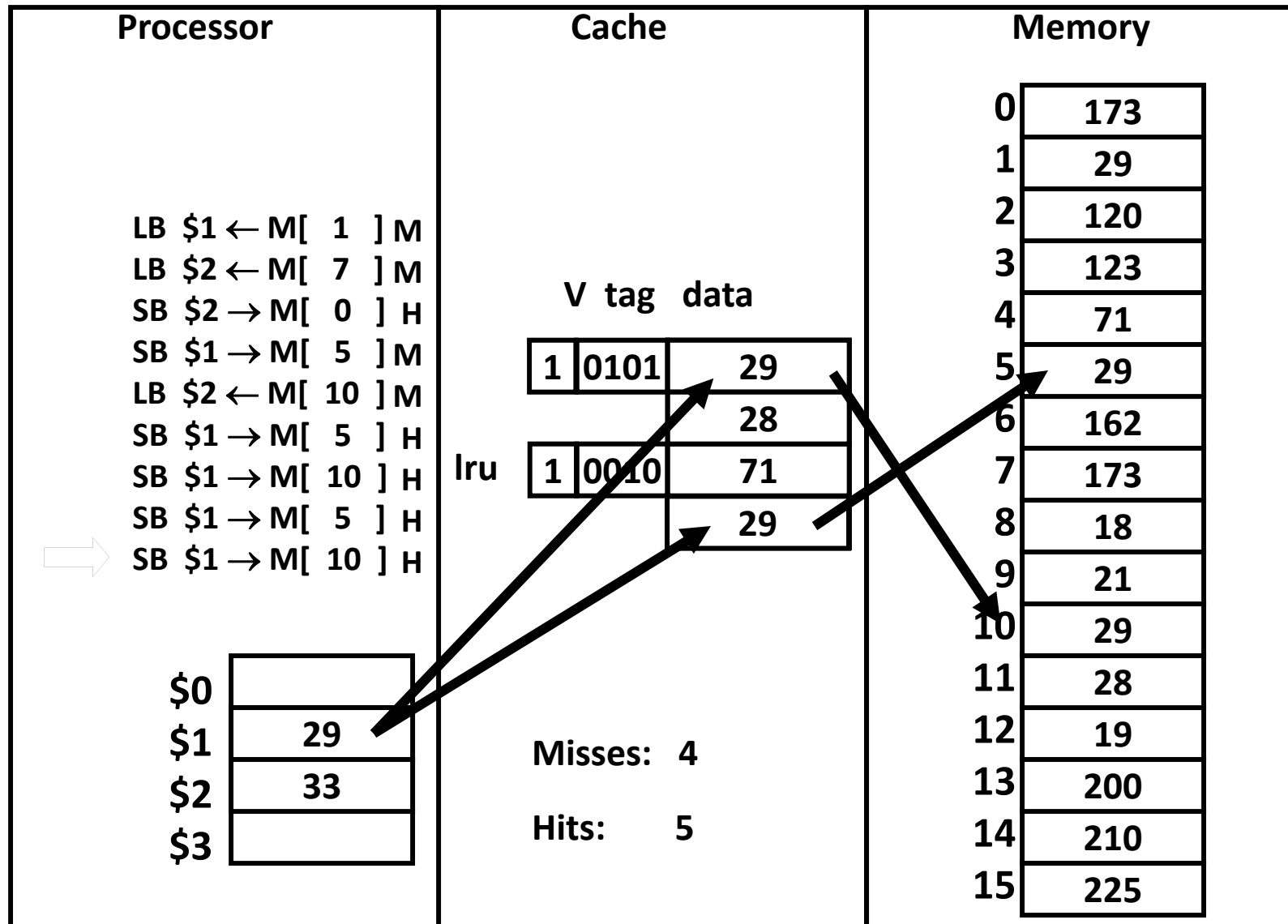
Evictions don't need to write to mem

- no need for dirty bit

Write-Through (REF 8,9)



Write-Through (REF 8,9)



Takeaway

A cache with a write-through policy (and write-allocate) reads an entire block (cacheline) from memory on a cache miss and writes only the updated item to memory for a store. Evictions do not need to write to memory.

Next Goal

Can we also design the cache NOT write all stores immediately to memory?

- Keep the most current copy in cache, and update memory when that data is ***evicted*** (write-back policy)

Write-Back Meta-Data

V	D	Tag	Byte 1	Byte 2	... Byte N

V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

- Set D = 1

When evicting line:

- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0

Write-back Example

Example: How does a write-back cache work?

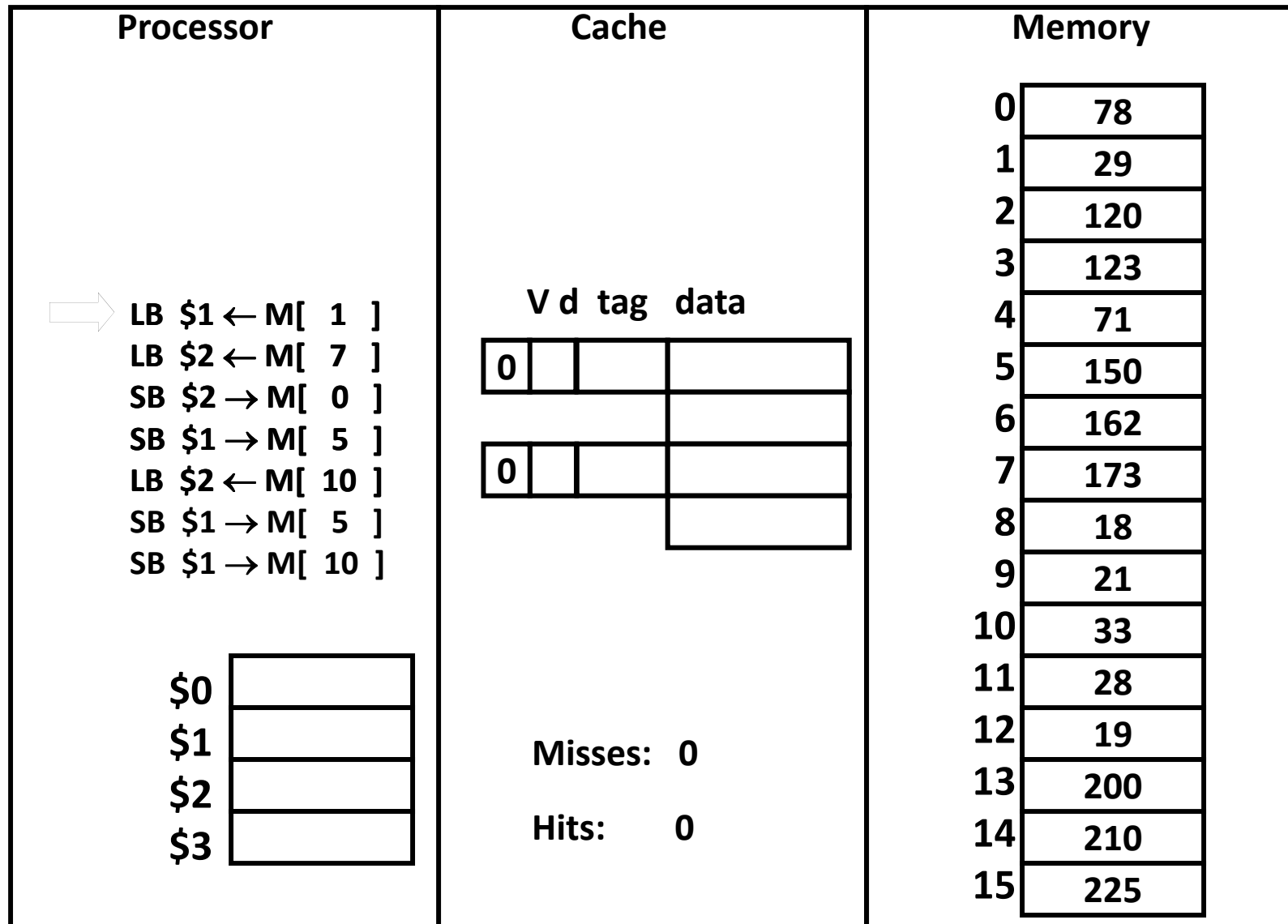
Assume write-allocate.

Handling Stores (Write-Back)

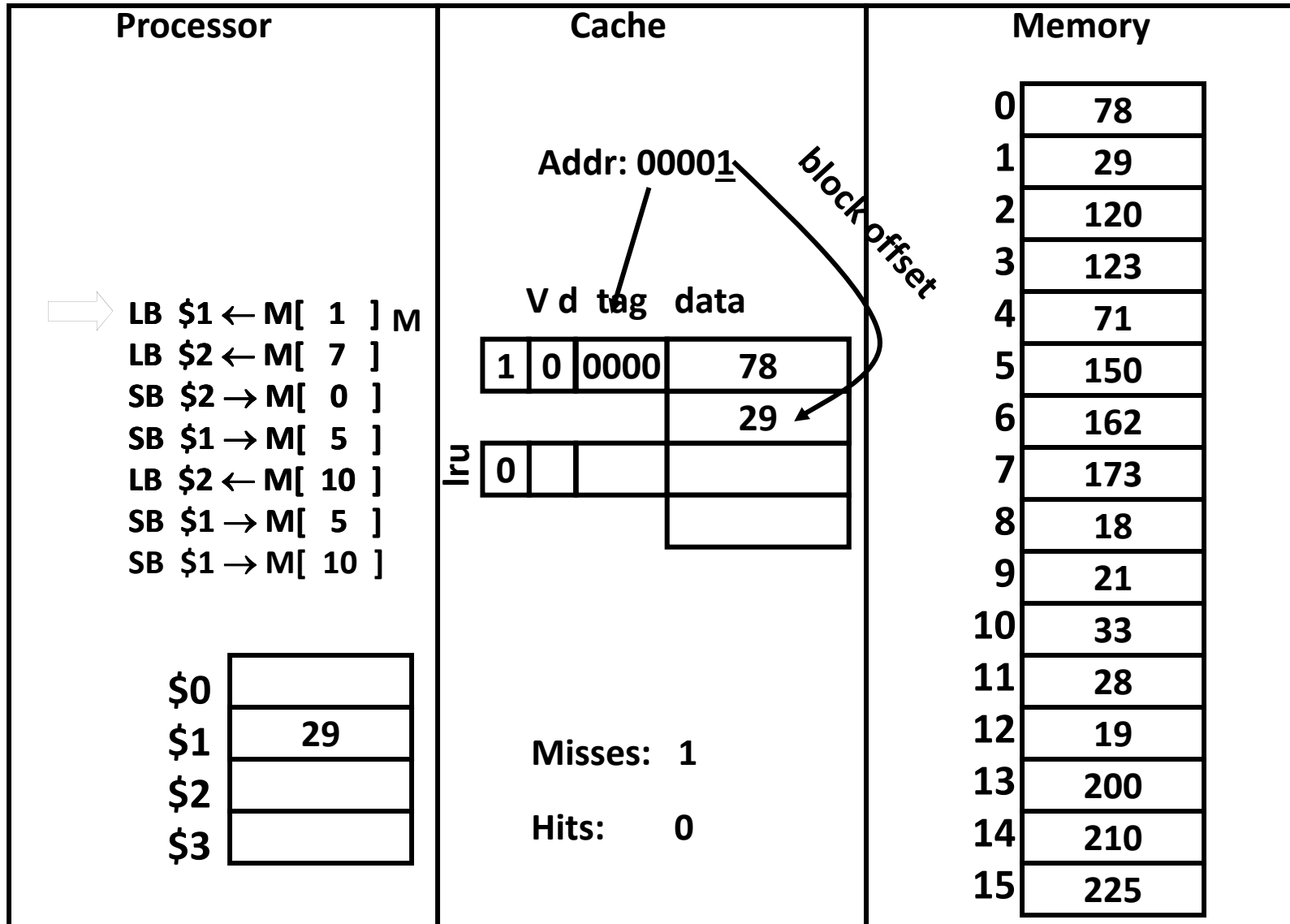
Using **byte addresses** in this example! Addr Bus = 5 bits

Processor	Cache	Memory																																																								
Assume write-allocate policy LB \$1 ← M[1] LB \$2 ← M[7] SB \$2 → M[0] SB \$1 → M[5] LB \$2 ← M[10] SB \$1 → M[5] SB \$1 → M[10] <table><tr><td>\$0</td><td></td></tr><tr><td>\$1</td><td></td></tr><tr><td>\$2</td><td></td></tr><tr><td>\$3</td><td></td></tr></table>	\$0		\$1		\$2		\$3		Fully Associative Cache 2 cache lines <u>2</u> word block <u>3</u> bit tag field 1 bit block offset field V d tag data <table><tr><td>0</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td>0</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table> Misses: 0 Hits: 0	0								0								<table><tr><td>0</td><td>78</td></tr><tr><td>1</td><td>29</td></tr><tr><td>2</td><td>120</td></tr><tr><td>3</td><td>123</td></tr><tr><td>4</td><td>71</td></tr><tr><td>5</td><td>150</td></tr><tr><td>6</td><td>162</td></tr><tr><td>7</td><td>173</td></tr><tr><td>8</td><td>18</td></tr><tr><td>9</td><td>21</td></tr><tr><td>10</td><td>33</td></tr><tr><td>11</td><td>28</td></tr><tr><td>12</td><td>19</td></tr><tr><td>13</td><td>200</td></tr><tr><td>14</td><td>210</td></tr><tr><td>15</td><td>225</td></tr></table>	0	78	1	29	2	120	3	123	4	71	5	150	6	162	7	173	8	18	9	21	10	33	11	28	12	19	13	200	14	210	15	225
\$0																																																										
\$1																																																										
\$2																																																										
\$3																																																										
0																																																										
0																																																										
0	78																																																									
1	29																																																									
2	120																																																									
3	123																																																									
4	71																																																									
5	150																																																									
6	162																																																									
7	173																																																									
8	18																																																									
9	21																																																									
10	33																																																									
11	28																																																									
12	19																																																									
13	200																																																									
14	210																																																									
15	225																																																									

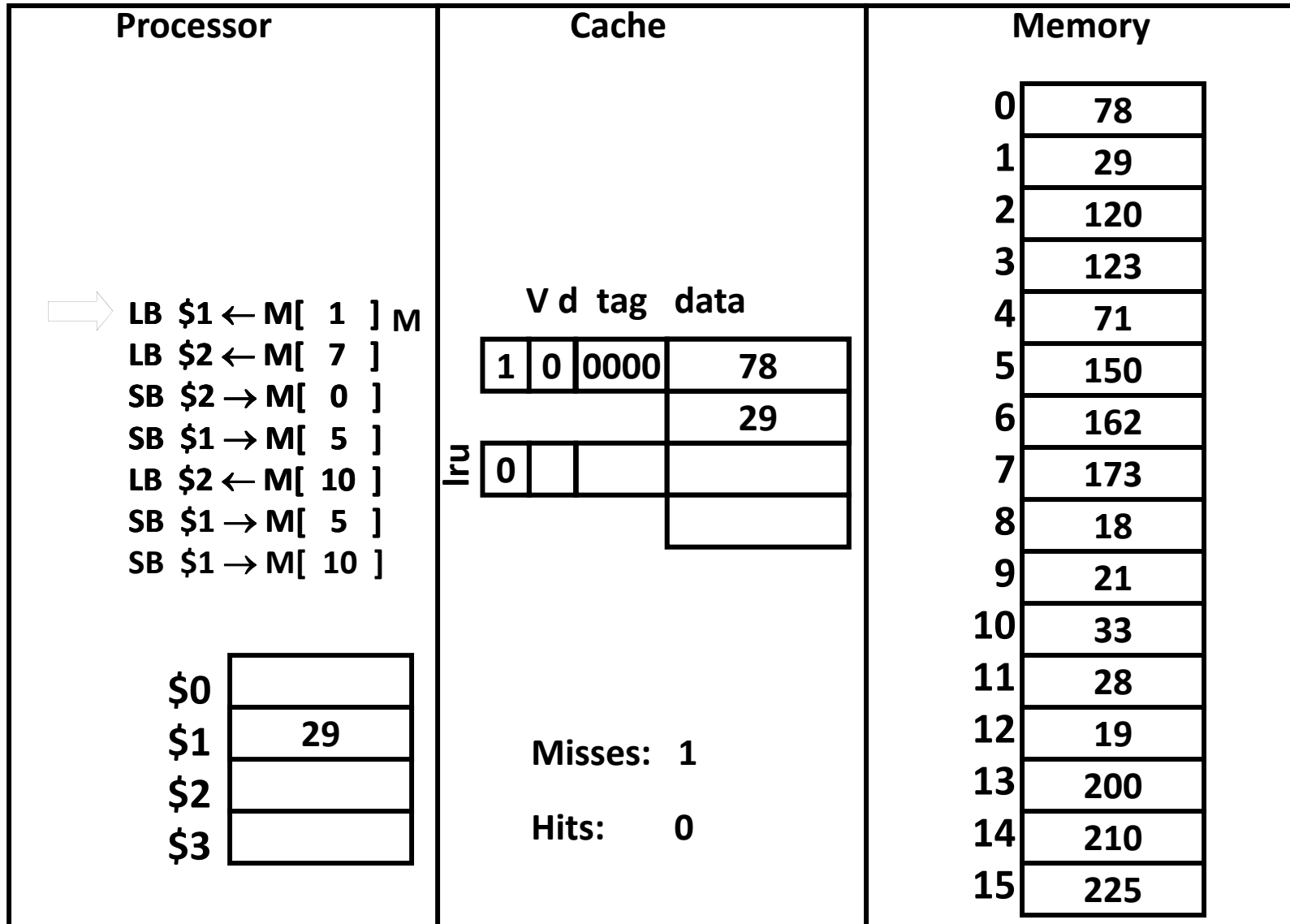
Write-Back (REF 1)



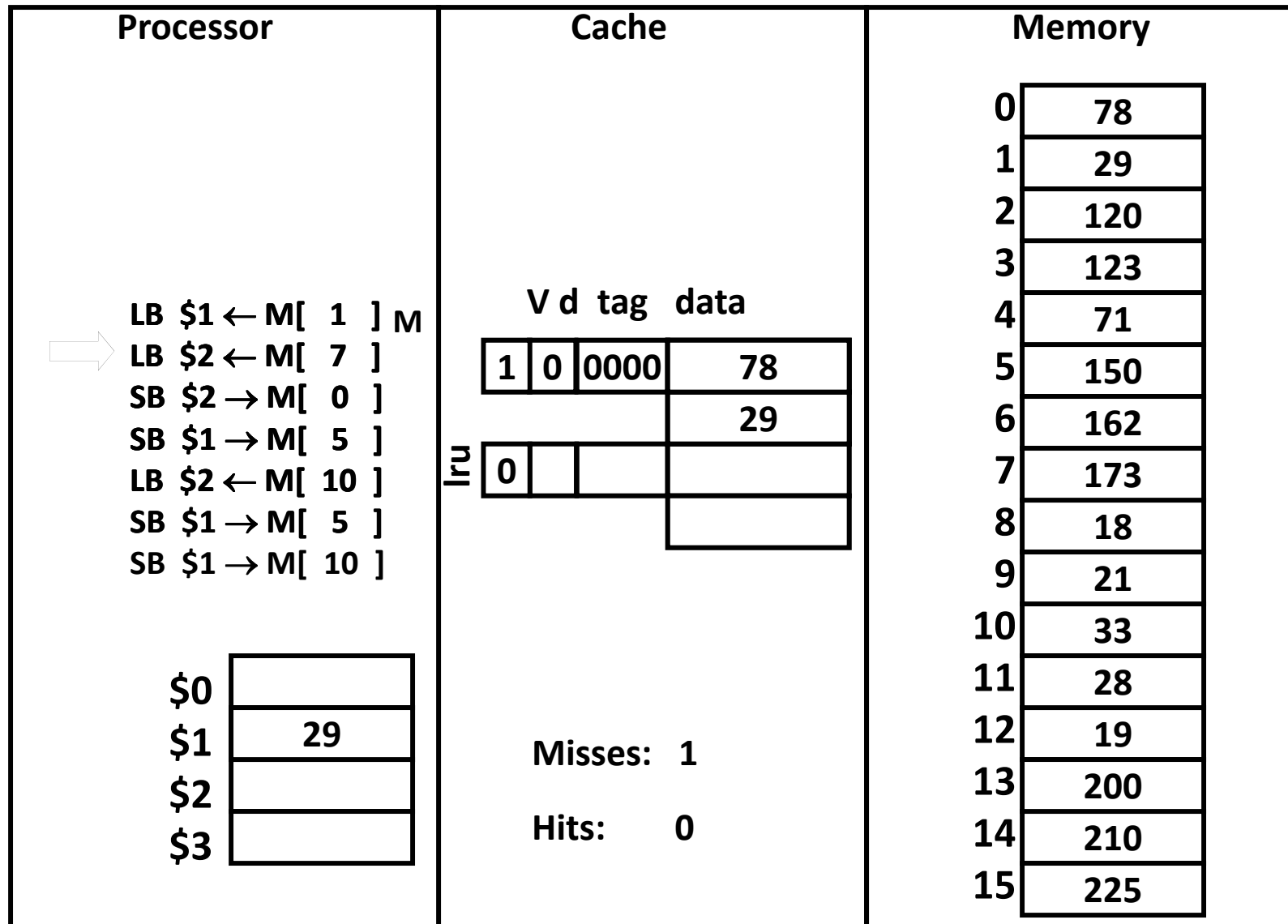
Write-Back (REF 1)



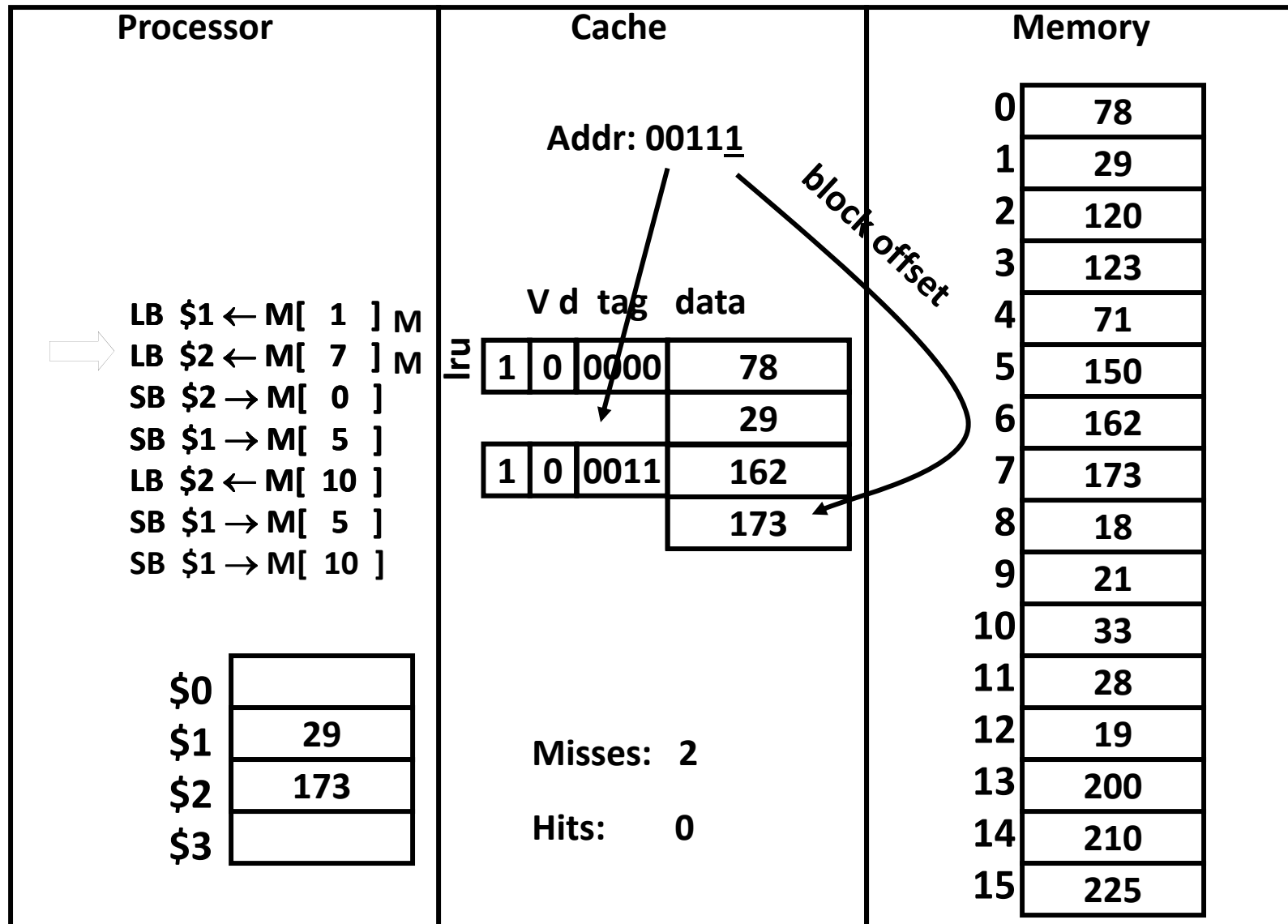
Write-Back (REF 1)



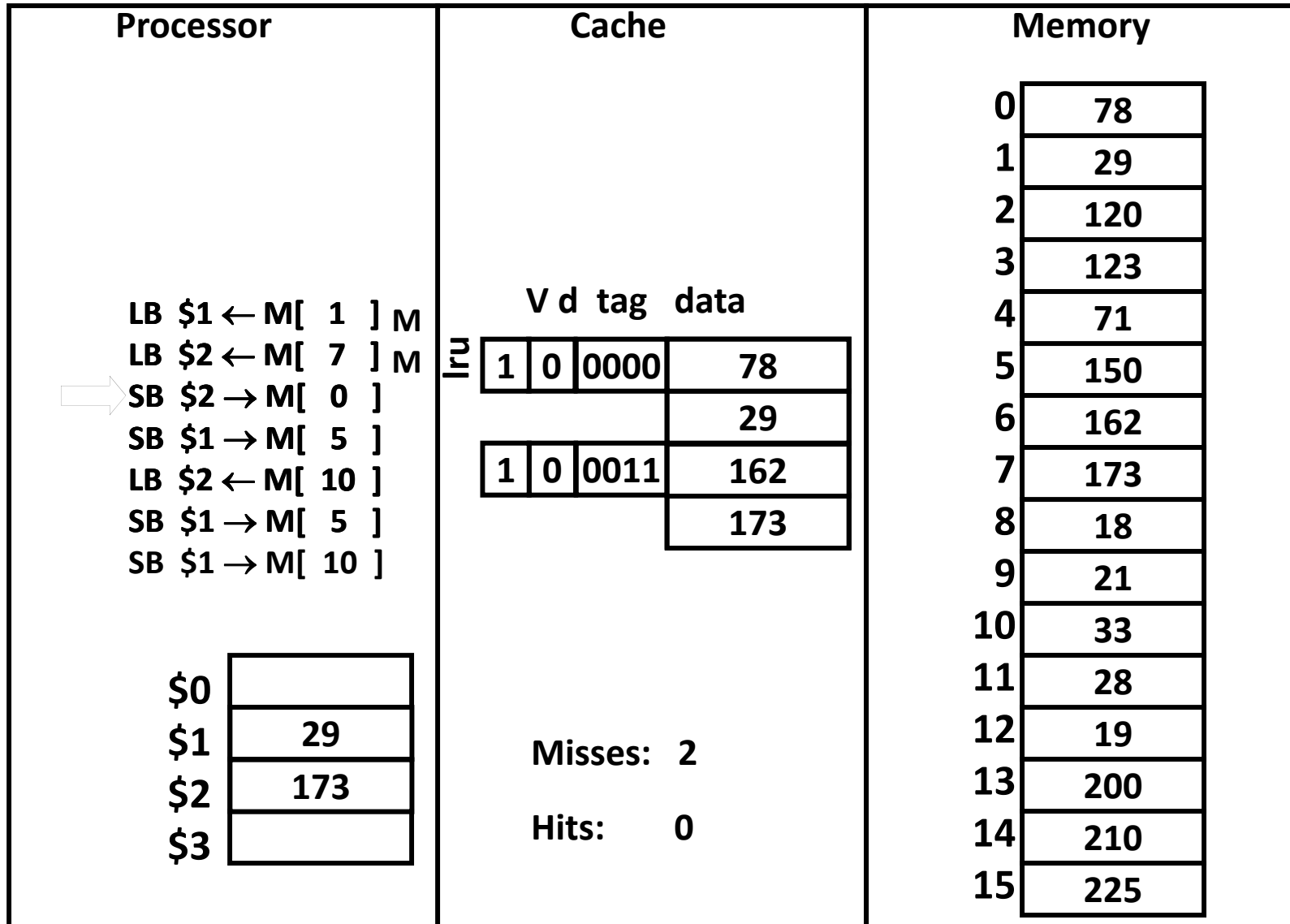
Write-Back (REF 2)



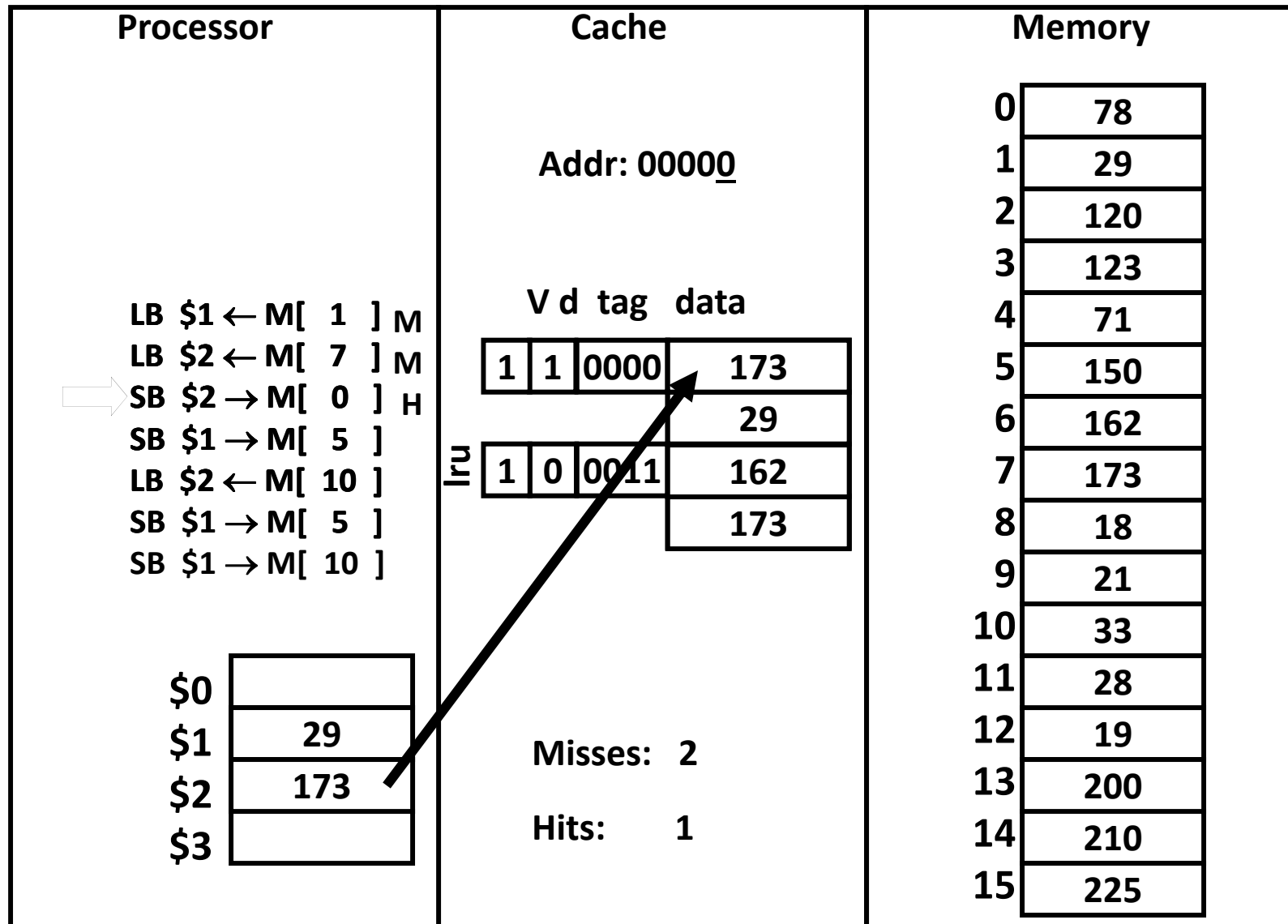
Write-Back (REF 2)



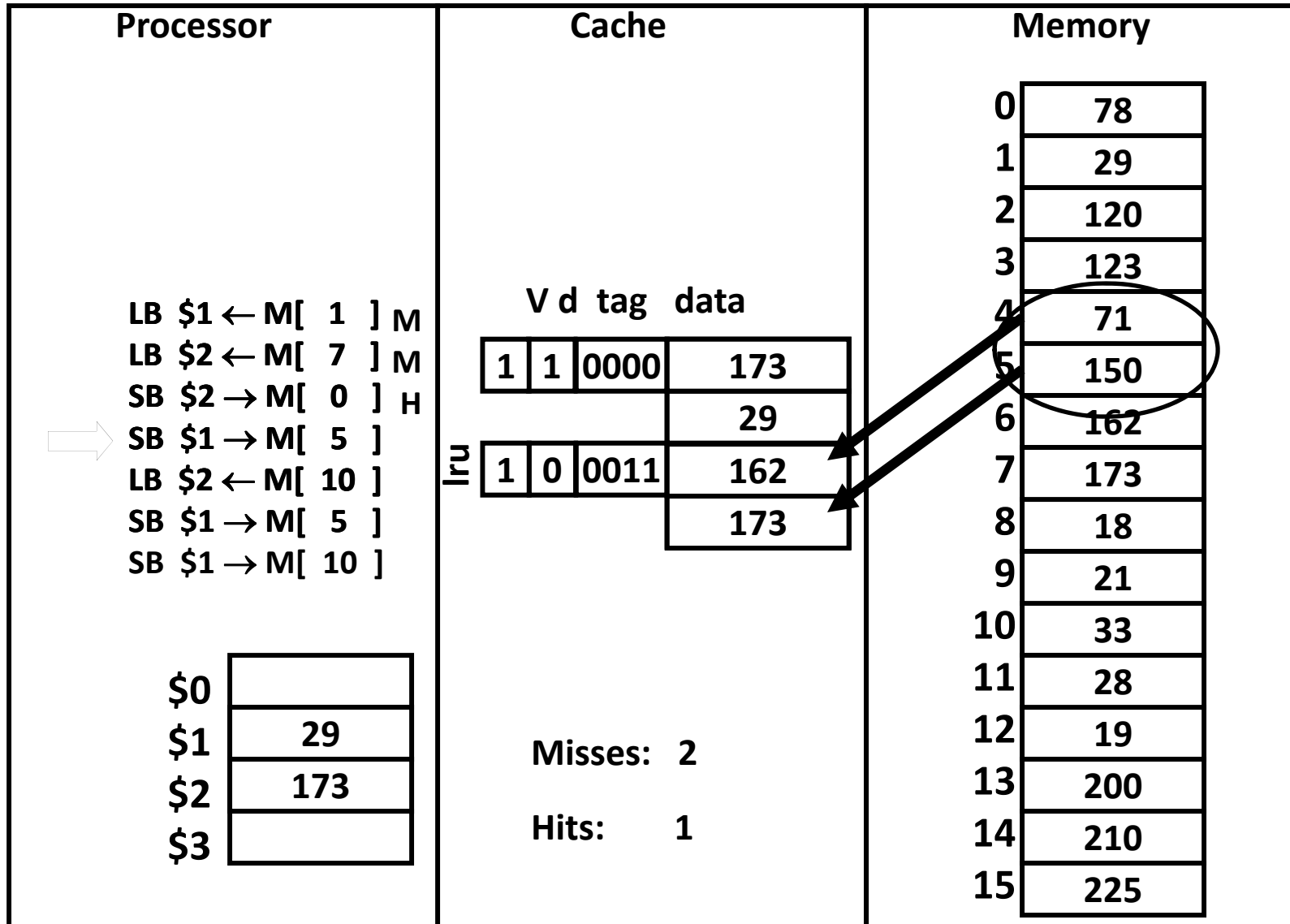
Write-Back (REF 3)



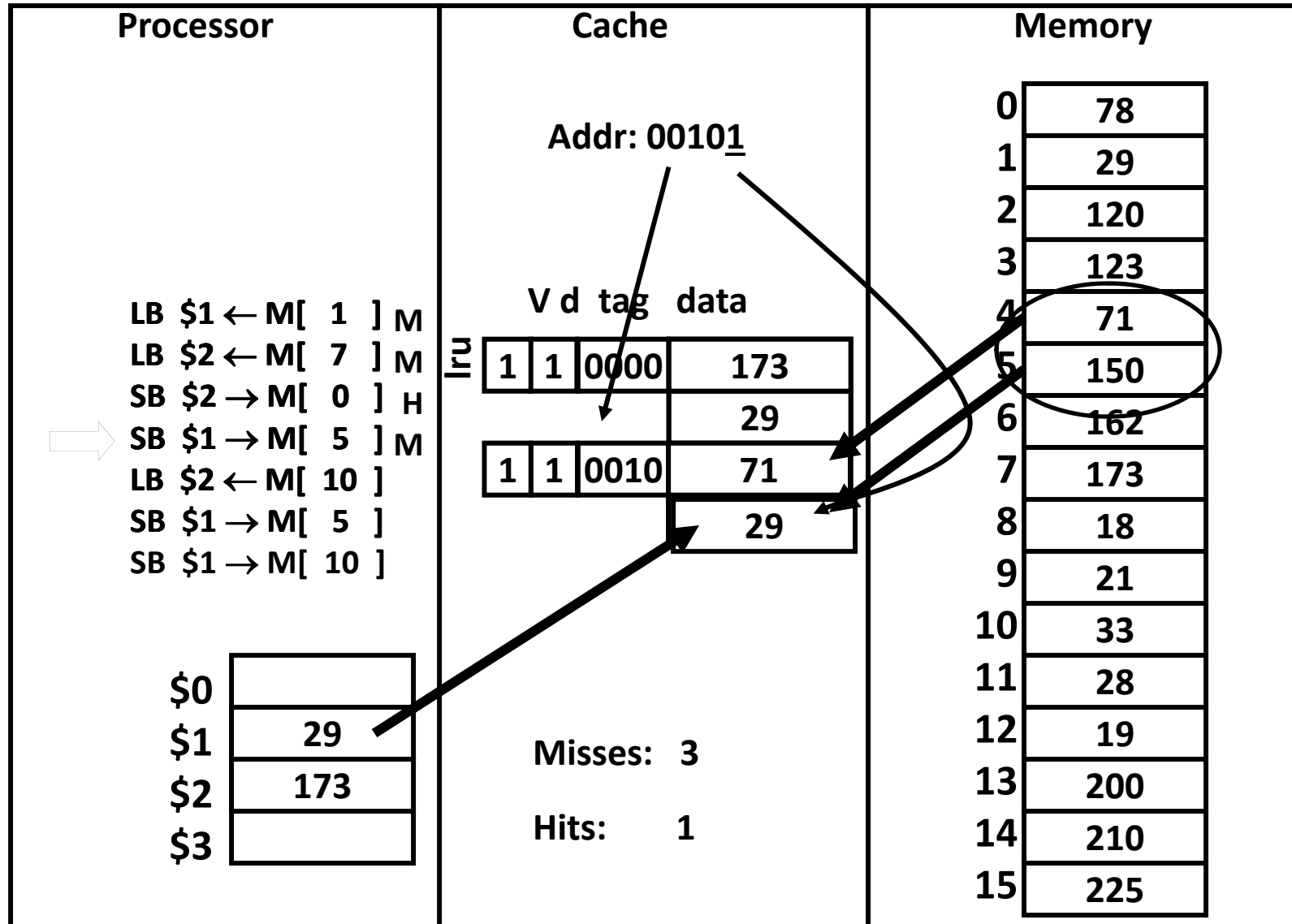
Write-Back (REF 3)



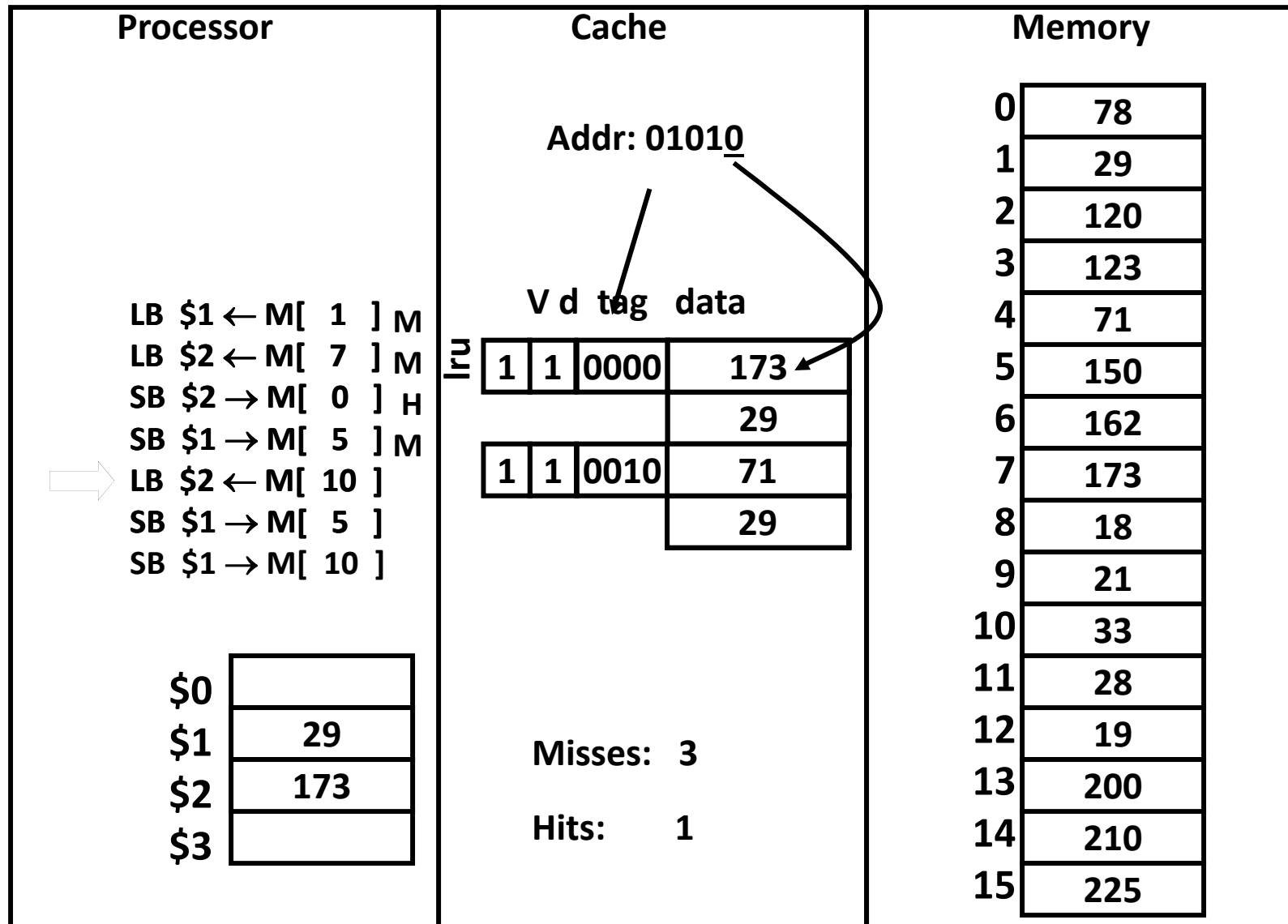
Write-Back (REF 4)



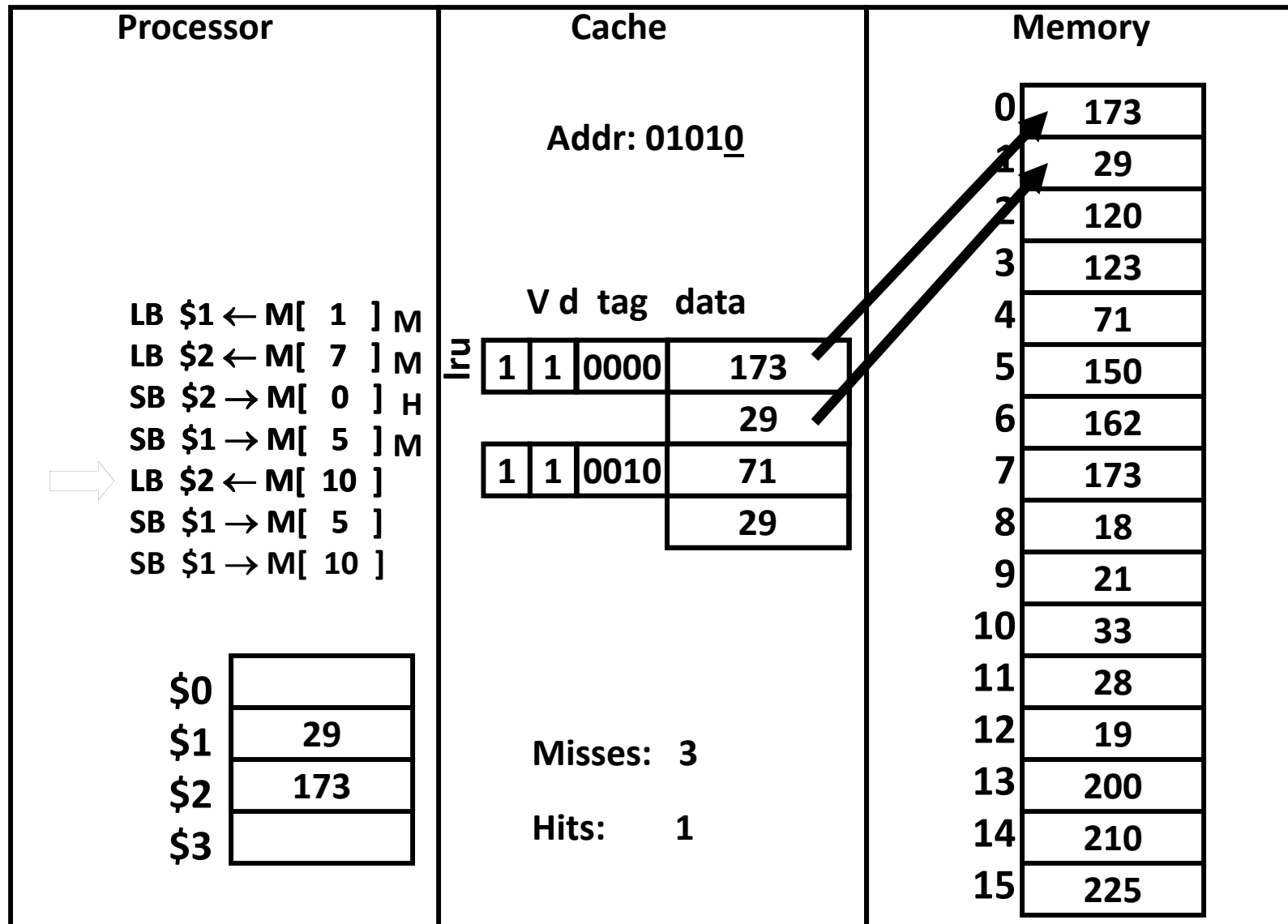
Write-Back (REF 4)



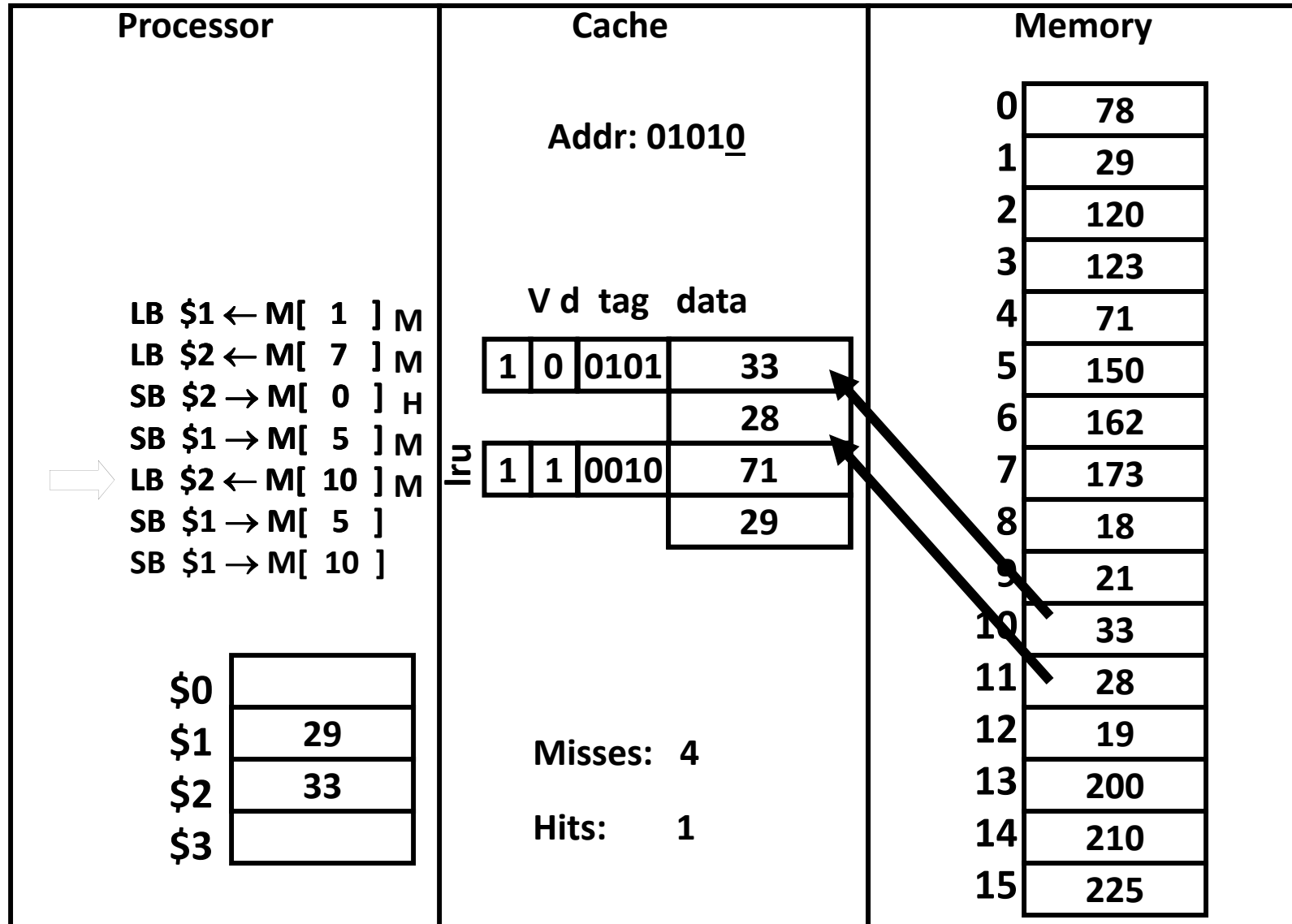
Write-Back (REF 5)



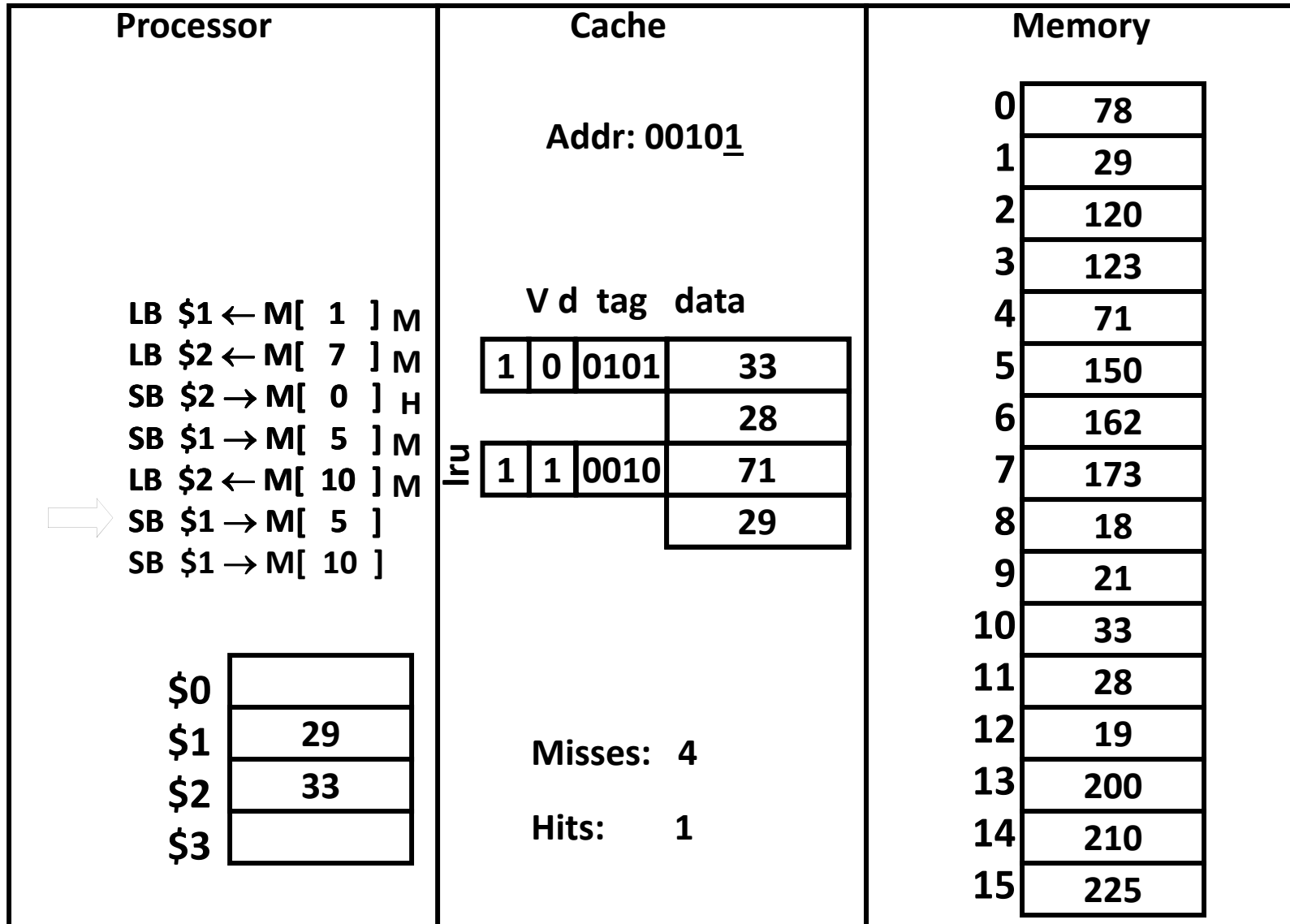
Write-Back (REF 5)



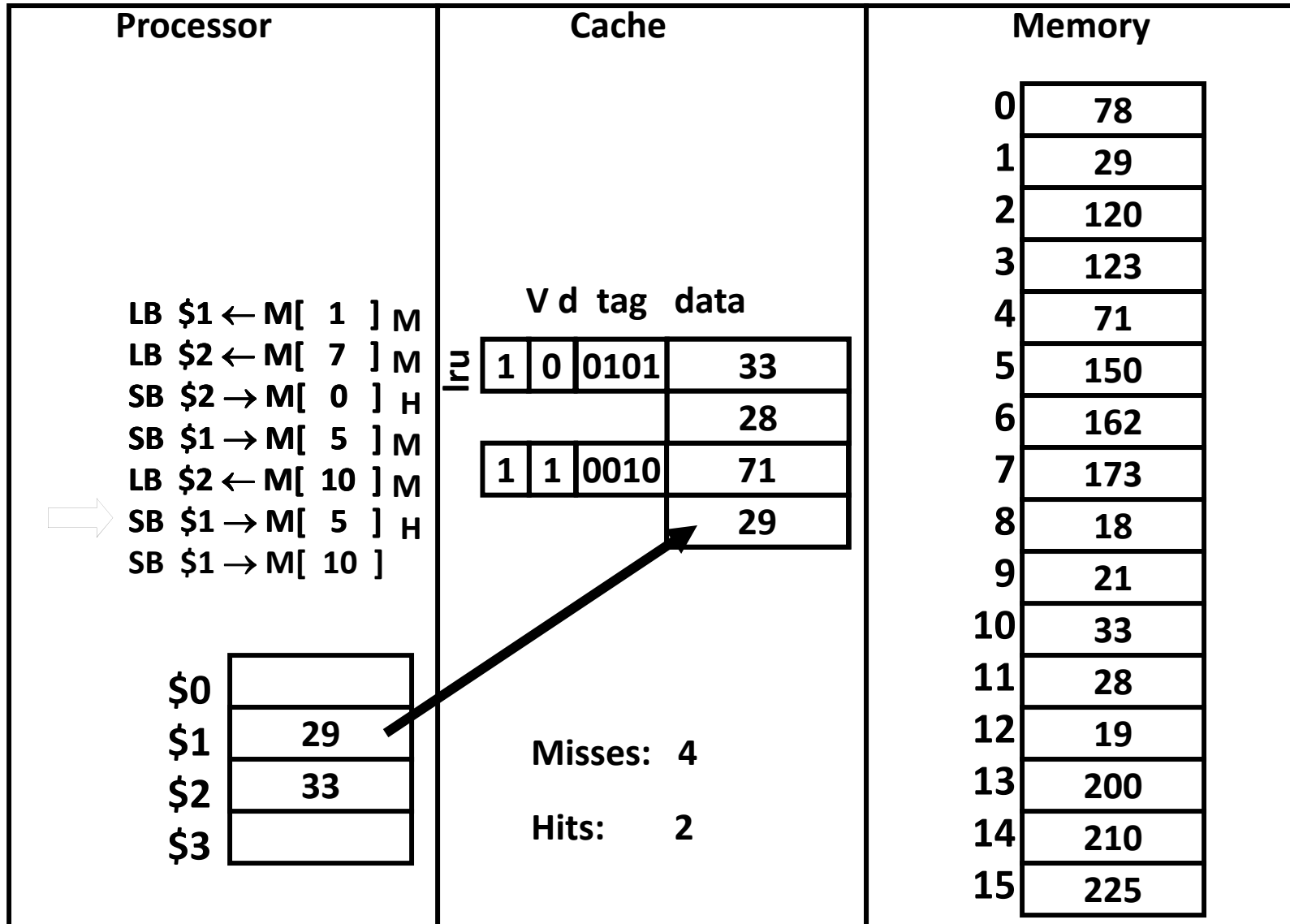
Write-Back (REF 5)



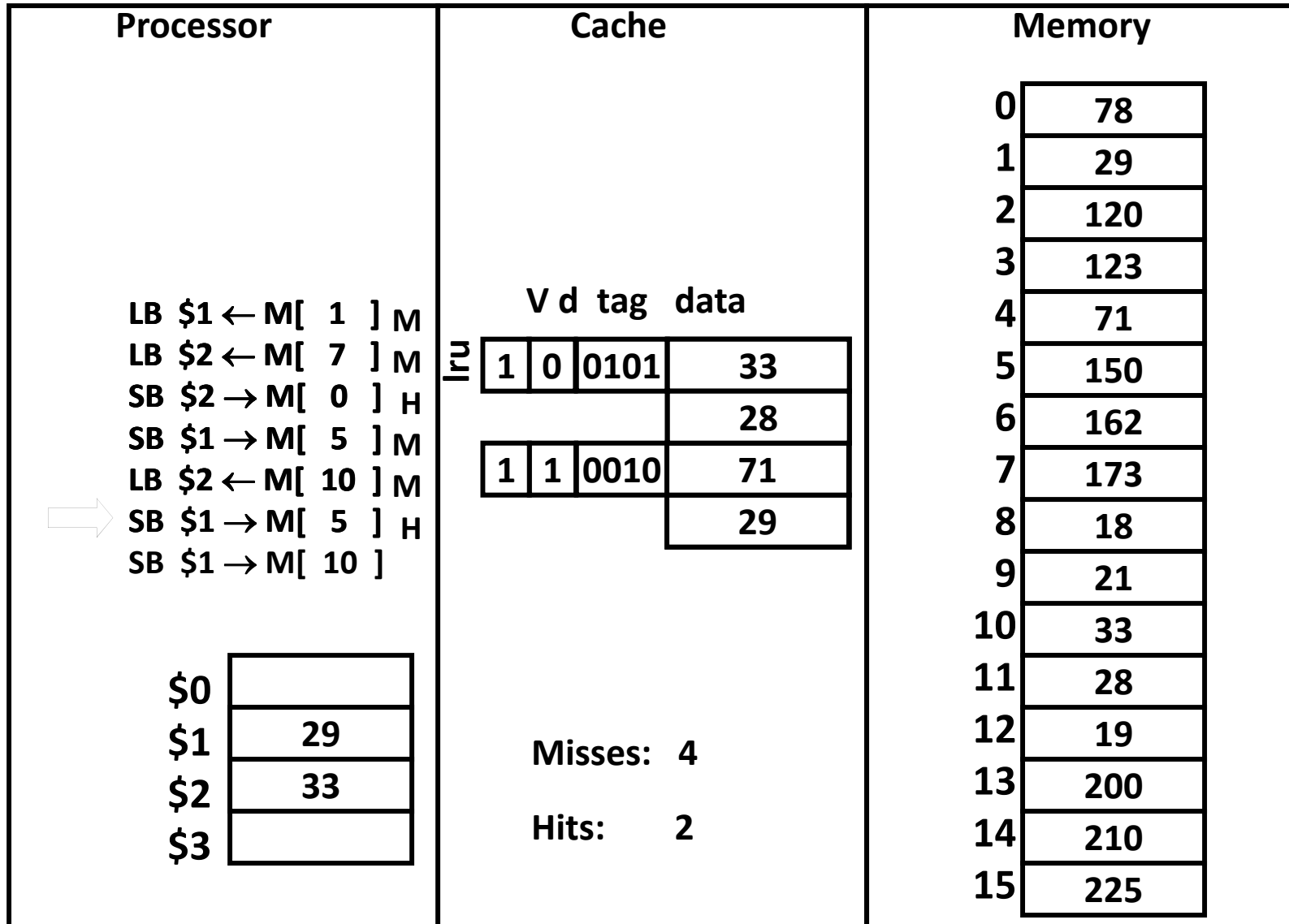
Write-Back (REF 6)



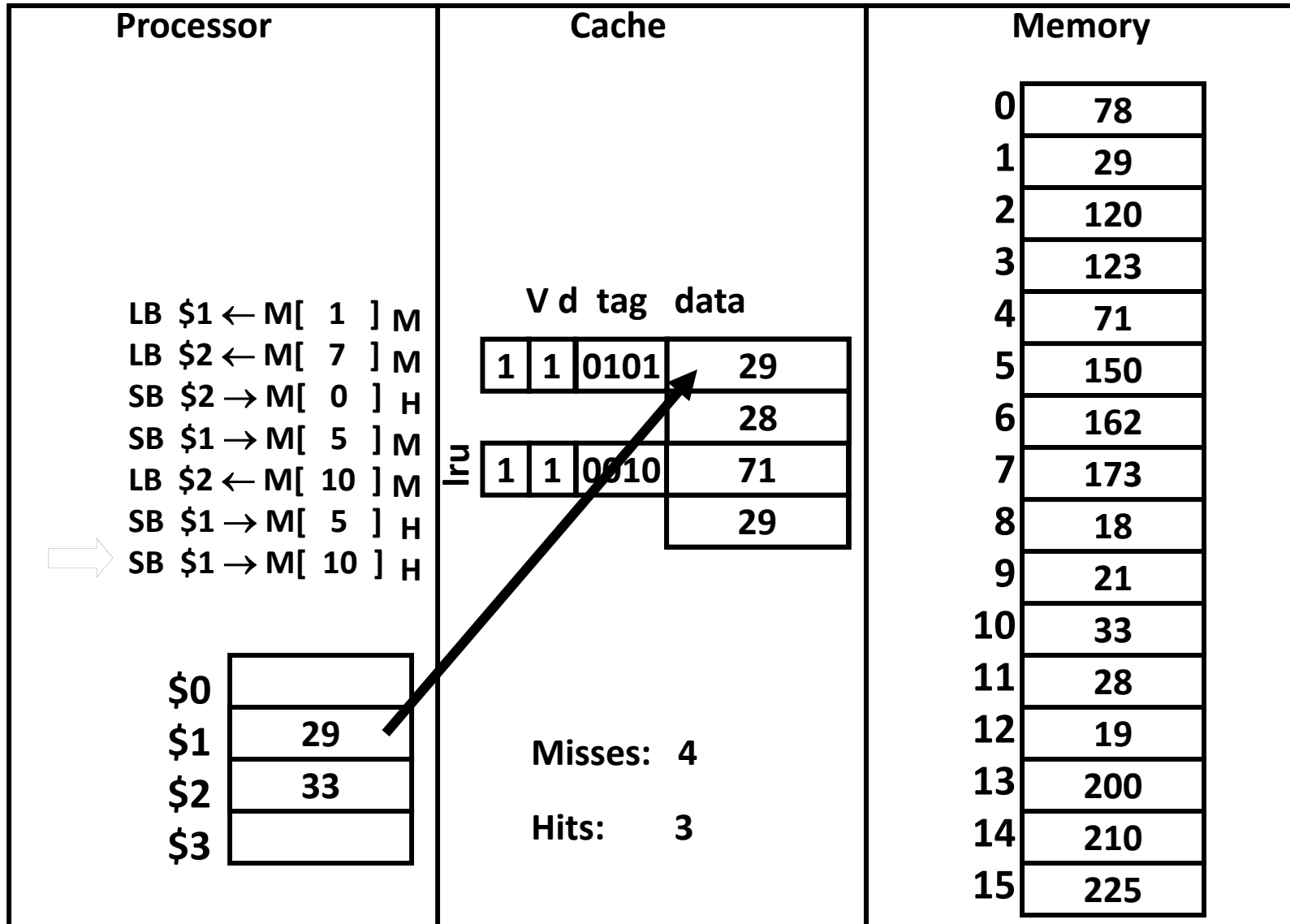
Write-Back (REF 6)



Write-Back (REF 7)



Write-Back (REF 7)



How Many Memory References?

Write-back performance

Each miss (read or write) reads a block from mem

- 4 misses $\rightarrow$ 8 mem reads

Some evictions write a block to mem

- 1 dirty eviction $\rightarrow$ 2 mem writes
- (+ 2 dirty evictions later $\rightarrow$ +4 mem writes)

How many memory references?

Each miss reads a block

Two words in this cache

Each evicted dirty cache line writes a block

Write-Back (REF 8,9)

Processor	Cache	Memory
LB \$1 ← M[1] M LB \$2 ← M[7] M SB \$2 → M[0] H SB \$1 → M[5] M LB \$2 ← M[10] M SB \$1 → M[5] H SB \$1 → M[10] H SB \$1 → M[5] SB \$1 → M[10] \$0 \$129 \$233 \$3	V d tag data 11010129 11001071 Misses: 4 Hits: 3	078 129 2120 3123 471 5150 6162 7173 818 921 1033 1128 1219 13200 14210 15225

Write-Back (REF 8,9)

Processor	Cache	Memory
LB \$1 ← M[1] M LB \$2 ← M[7] M SB \$2 → M[0] H SB \$1 → M[5] M LB \$2 ← M[10] M SB \$1 → M[5] H SB \$1 → M[10] H SB \$1 → M[5] H SB \$1 → M[10] H \$0 \$1 \$2 \$3 29 33	V d tag data 1 1 0101 29 28 1 1 0010 71 29 Misses: 4 Hits: 5	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 78 29 120 123 71 150 162 173 18 21 33 28 19 200 210 225

How Many Memory References?

Write-back performance

Each miss (read or write) reads a block from mem

- 4 misses $\rightarrow$ 8 mem reads

Some evictions write a block to mem

- 1 dirty eviction $\rightarrow$ 2 mem writes
- (+ 2 dirty evictions later $\rightarrow$ +4 mem writes)

By comparison write-through was

- Reads: eight words
- Writes: 4/6/8/10/12/... etc words
- Write-through or Write-back?

Write-through vs. Write-back

Write-through is slower

- But cleaner (memory always consistent)

Write-back is faster

- But complicated when multi cores sharing memory

Takeaway

A cache with a write-through policy (and write-allocate) reads an entire block (cacheline) from memory on a cache miss and writes only the updated item to memory for a store. Evictions do not need to write to memory.

A cache with a **write-back** policy (and write-allocate) reads an entire block (cacheline) from memory on a cache miss, **may need to write dirty cacheline first**. Any writes to memory need to be the entire cacheline since no way to distinguish which word was dirty with only a single dirty bit. Evictions of a dirty cacheline cause a write to memory.

Next Goal

What are other performance tradeoffs between write-through and write-back?

How can we further reduce penalty for cost of writes to memory?

Performance: An Example

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```
for (i=1; i<n; i++)  
    A[0] += A[i];
```

```
for (i=0; i<n; i++)  
    B[i] = A[i]
```

Performance: An Example

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```
for (i=1; i<n; i++)
```

```
    A[0] += A[i];
```

N words

Write-through: $n/16$ reads

n writes

Write-back: $n/16$ reads

1 write

```
for (i=0; i<n; i++)
```

```
    B[i] = A[i]
```

Write-through: $2 \times n/16$ reads

n writes

Write-back: $2 \times n/16$ reads

n write

Performance Tradeoffs

Q: Hit time: write-through vs. write-back?

A: Write-through slower on writes.

Q: Miss penalty: write-through vs. write-back?

A: Write-back slower on evictions.

Write Buffering

Q: Writes to main memory are **slow!**

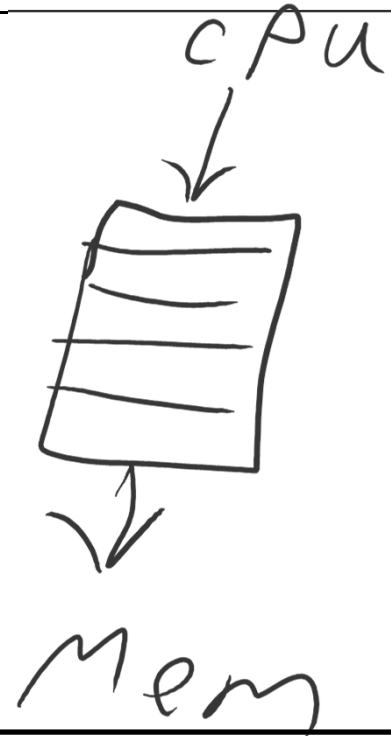
A: Use a write-back buffer

- A small queue holding dirty lines
- Add to end upon eviction
- Remove from front upon completion

Q: What does it help?

A: short bursts of writes (but not sustained writes)

A: fast eviction reduces miss penalty



Write-through vs. Write-back

Write-through is slower

- But simpler (memory always consistent)

Write-back is almost always faster

- write-back buffer hides large eviction cost
- But what about multiple cores with separate caches but sharing memory?

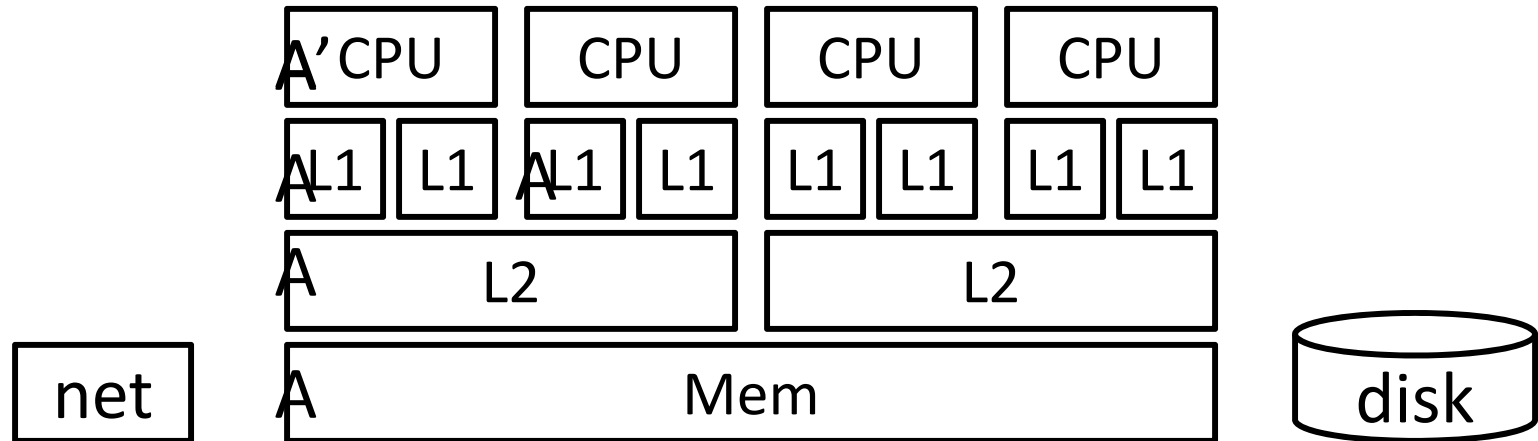
Write-back requires a cache coherency protocol

- Inconsistent views of memory
- Need to “snoop” in each other’s caches
- Extremely complex protocols, very hard to get right

Cache-coherency

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory



Cache coherency protocol

- May need to snoop on other CPU's cache activity
- Invalidate cache line when other CPU writes
- Flush write-back caches before other CPU reads
- Or the reverse: Before writing/reading...
- Extremely complex protocols, very hard to get right

Takeaway

A cache with a write-through policy (and write-allocate) reads an entire block (cacheline) from memory on a cache miss and writes only the updated item to memory for a store. Evictions do not need to write to memory.

A cache with a **write-back** policy (and write-allocate) reads an entire block (cacheline) from memory on a cache miss, **may need to write dirty cacheline first**. Any writes to memory need to be the entire cacheline since no way to distinguish which word was dirty with only a single dirty bit. Evictions of a dirty cacheline cause a write to memory.

Write-through is slower, but simpler (memory always consistent)/
Write-back is almost always faster (a write-back buffer can hidee large eviction cost), but will need a coherency protocol to maintain consistency will all levels of cache and memory.

Cache Design Tradeoffs

Cache Design

Need to determine parameters:

- Cache size
- Block size (aka line size)
- Number of ways of set-associativity (1, N, ∞)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy

A Real Example

```
> dmidecode -t cache
```

Cache Information

Configuration: Enabled, Not Socketed, Level 1
Operational Mode: Write Back
Installed Size: 128 KB
Error Correction Type: None

Cache Information

Configuration: Enabled, Not Socketed, Level 2
Operational Mode: Varies With Memory Address
Installed Size: 6144 KB
Error Correction Type: Single-bit ECC

```
> cd /sys/devices/system/cpu/cpu0; grep cache/*/*
```

```
cache/index0/level:1  
cache/index0/type:Data  
cache/index0/ways_of_associativity:8  
cache/index0/number_of_sets:64  
cache/index0/coherency_line_size:64  
cache/index0/size:32K  
cache/index1/level:1  
cache/index1/type:Instruction  
cache/index1/ways_of_associativity:8  
cache/index1/number_of_sets:64  
cache/index1/coherency_line_size:64  
cache/index1/size:32K  
cache/index2/level:2  
cache/index2/type:Unified  
cache/index2/shared_cpu_list:0-1  
cache/index2/ways_of_associativity:24  
cache/index2/number_of_sets:4096  
cache/index2/coherency_line_size:64  
cache/index2/size:6144K
```

Dual-core 3.16GHz Intel
(purchased in 2011)

A Real Example

Dual-core 3.16GHz Intel

Dual 32K L1 Instruction caches

(purchased in 2009)

- 8-way set associative
- 64 sets
- 64 byte line size

Dual 32K L1 Data caches

- Same as above

Single 6M L2 Unified cache

- 24-way set associative (!!!)
- 4096 sets
- 64 byte line size

4GB Main memory

1TB Disk

Basic Cache Organization

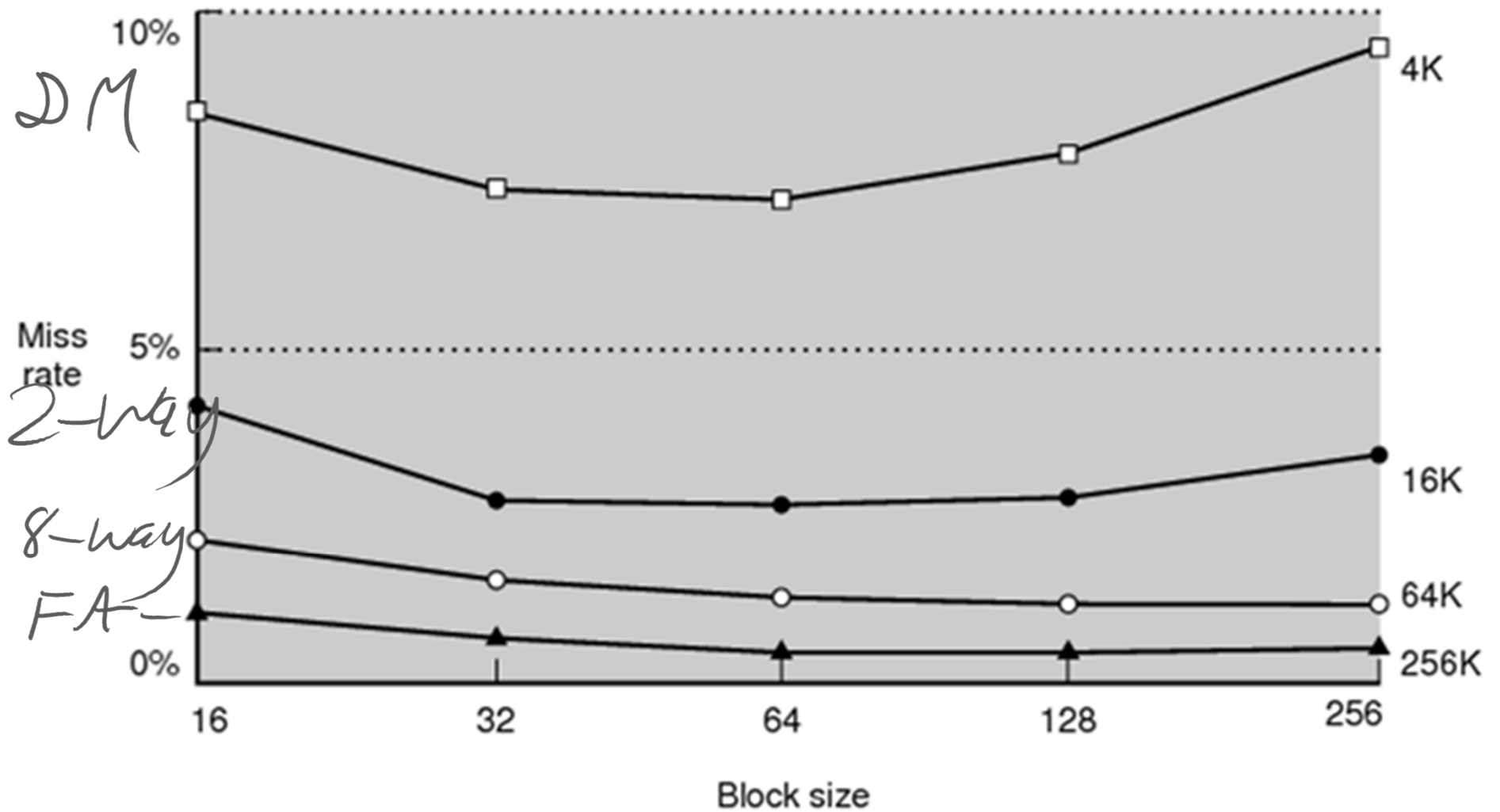
Q: How to decide block size?

A: Try it and see

But: depends on cache size, workload, associativity, ...

Experimental approach!

Experimental Results



Tradeoffs

For a given total cache size,
larger block sizes mean....

- fewer lines
- so fewer tags (and smaller tags for associative caches)
- so 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)

Cache Conscious Programming

Cache Conscious Programming

```
// H = 12, W = 10
```

```
int A[H][W];
```

```
for(x=0; x < W; x++)
```

```
    for(y=0; y < H; y++)
```

```
        sum += A[y][x];
```

row col

1	11	21							
		2	12	22					
				3	13	23			
						4	14	24	
								5	15
25									
6	16	26							
		7	17	...					
				8	18				
						9	19		
								10	20

Every access is a cache miss!

(unless *entire* matrix can fit in cache)

Cache Conscious Programming

```
// H = 12, W = 10
```

```
int A[H][W];
```

```
for(y=0; y < H; y++)
```

```
for(x=0; x < W; x++)
```

```
sum += A[y][x];
```

Block size = 4 $\rightarrow$ 75% hit rate

Block size = 8 $\rightarrow$ 87.5% hit rate

Block size = 16 $\rightarrow$ 93.75% hit rate

And you can easily prefetch to warm the cache.

[illegible]

Summary

Caching assumptions

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

Benefits

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

Tradeoffs:

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

Summary

Memory performance matters!

- often more than CPU performance
- ... because it is the bottleneck, and not improving much
- ... because most programs move a LOT of data

Design space is huge

- Gambling against program behavior
- Cuts across all layers:
users → programs → os → hardware

Multi-core / Multi-Processor is complicated

- Inconsistent views of memory
- Extremely complex protocols, very hard to get right

Administrivia

Prelim1: ***TODAY, Thursday***, March 28th in evening

- Time: We will start at ***7:30pm sharp***, so come early
- **Two Location: PHL101 and UPSB17**
 - If NetID ends with even number, then go to PHL101 (Phillips Hall rm 101)
 - If NetID ends with odd number, then go to UPSB17 (Upson Hall rm B17)
- Closed Book: ***NO NOTES, BOOK, ELECTRONICS, CALCULATOR, CELL PHONE***
- Practice prelims are online in CMS
- Material covered everything up to end of ***week before spring break***
 - Lecture: Lectures 9 to 16 (new since last prelim)
 - Chapter 4: Chapters 4.7 (Data Hazards) and 4.8 (Control Hazards)
 - Chapter 2: Chapter 2.8 and 2.12 (Calling Convention and Linkers), 2.16 and 2.17 (RISC and CISC)
 - Appendix B: B.1 and B.2 (Assemblers), B.3 and B.4 (linkers and loaders), and B.5 and B.6 (Calling Convention and process memory layout)
 - Chapter 5: 5.1 and 5.2 (Caches)
 - HW3, Project1 and Project2

Administrivia

Next six weeks

- Week 9 (May 25): Prelim2
- Week 10 (Apr 1): Project2 due and Lab3 handout
- Week 11 (Apr 8): Lab3 due and Project3/HW4 handout
- Week 12 (Apr 15): Project3 design doc due and HW4 due
- Week 13 (Apr 22): Project3 due and Prelim3
- Week 14 (Apr 29): Project4 handout

Final Project for class

- Week 15 (May 6): Project4 design doc due
- Week 16 (May 13): Project4 due