Caches (Writing)

Hakim Weatherspoon

CS 3410, Spring 2013

Computer Science

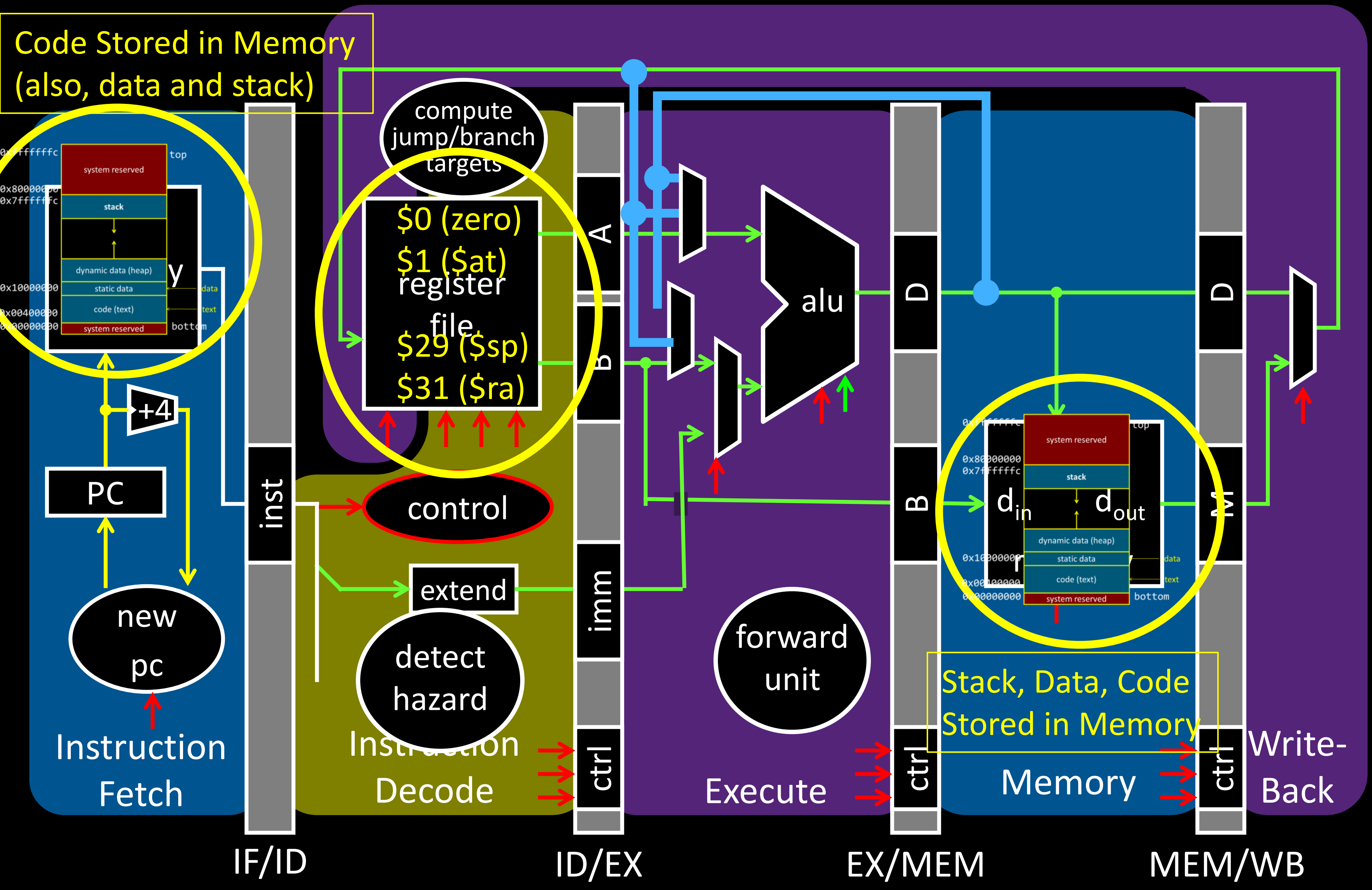
Cornell University

P & H Chapter 5.2-3, 5.5

Welcome back from Spring Break!

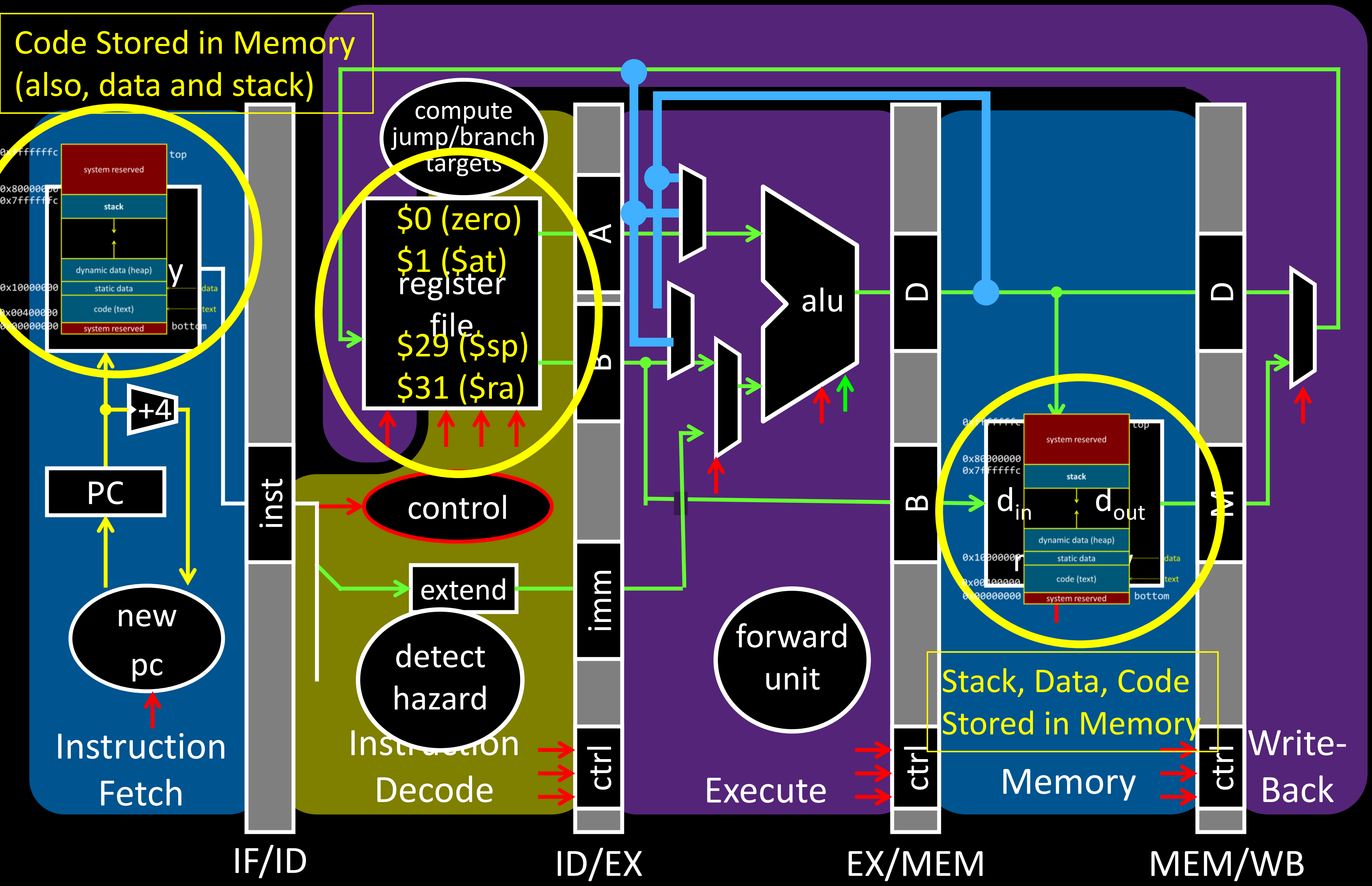
Welcome back from Spring Break!

Big Picture: Memory



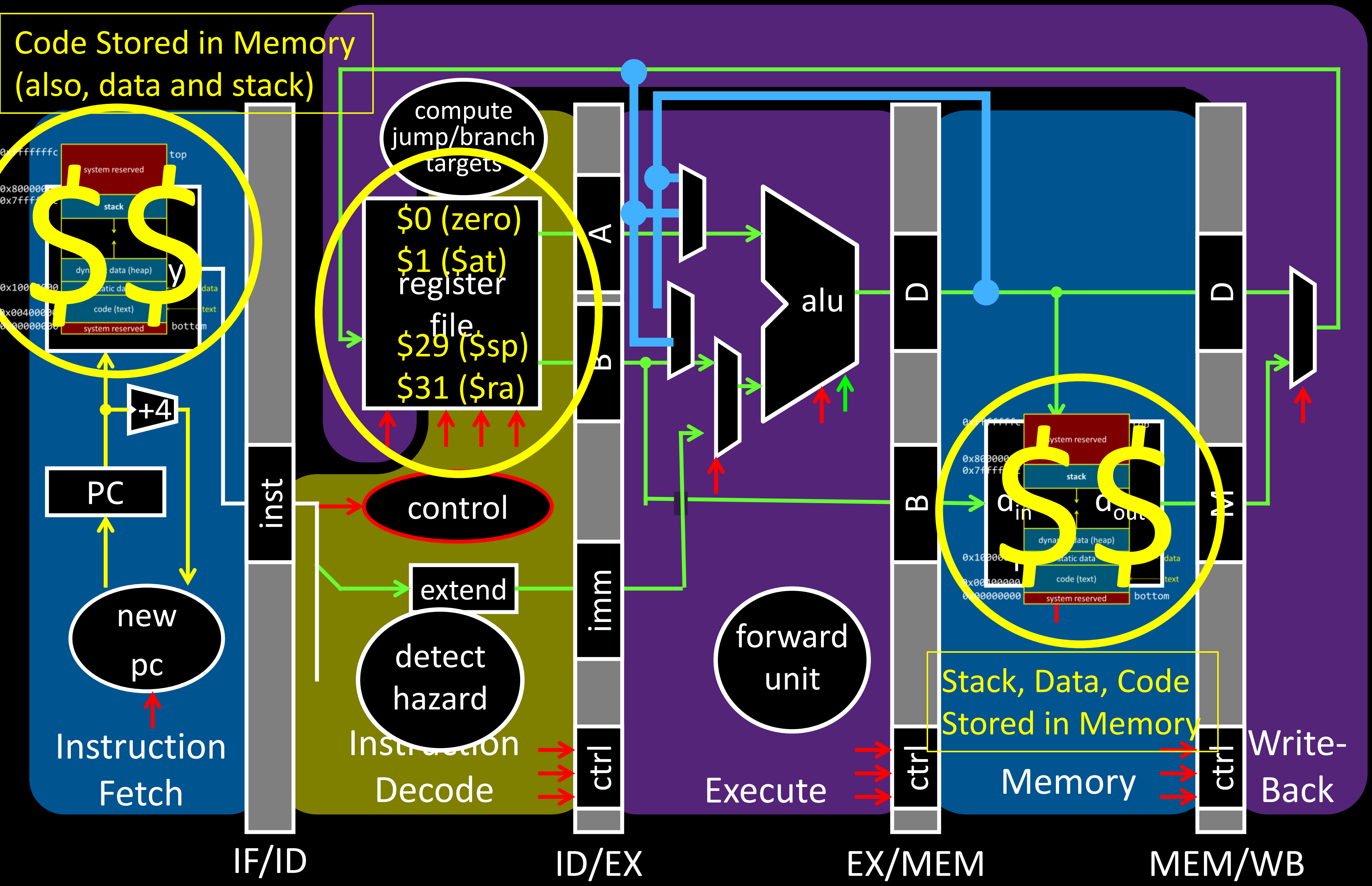
Big Picture: Memory

Memory: big & slow vs Caches: small & fast



Big Picture: Memory

Memory: big & slow vs Caches: small & fast



Big Picture

How do we make the processor fast,

Given that memory is VEEERRRYYYY SLLOOOWWW!!

Big Picture

How do we make the processor fast,
Given that memory is VEEERRRYYYY SLLOOOWWW!!

But, 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] <u>higher</u> in memory hierarchy since it will likely be accessed again soon

… then Mem[x ± *ε*] 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

Goals for Today: caches

Comparison of cache architectures:

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

Writing to the Cache

- Write-through vs Write-back

Caching Questions

- How does a cache work?
- How effective is the cache (hit rate/miss rate)?
- How large is the cache?
- How fast is the cache (AMAT=average memory access time)

Next Goal

How do the different cache architectures compare?

- Cache Architecture Tradeoffs?
- Cache Size?
- Cache Hit rate/Performance?

Cache Tradeoffs

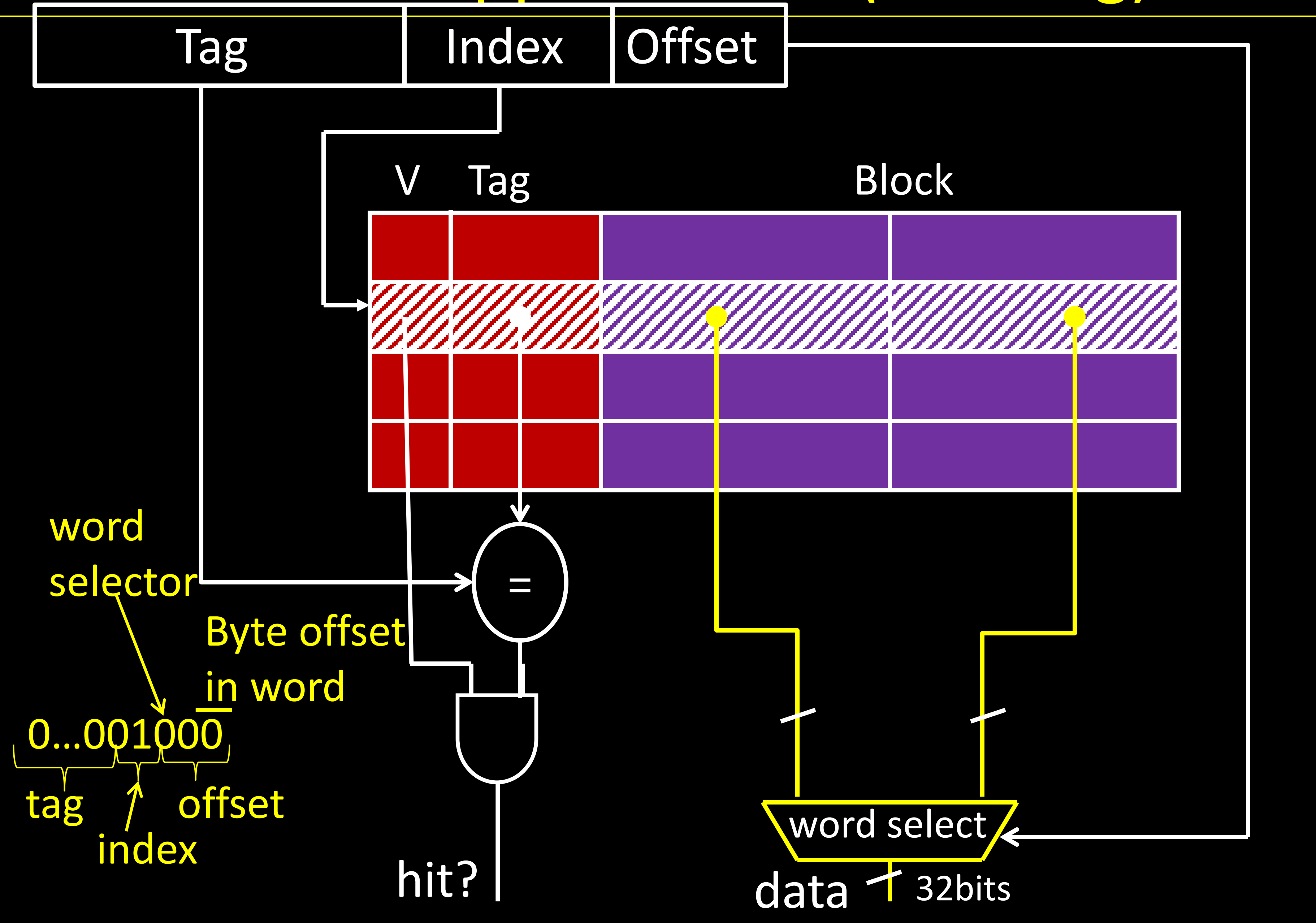
A given data block can be placed…

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

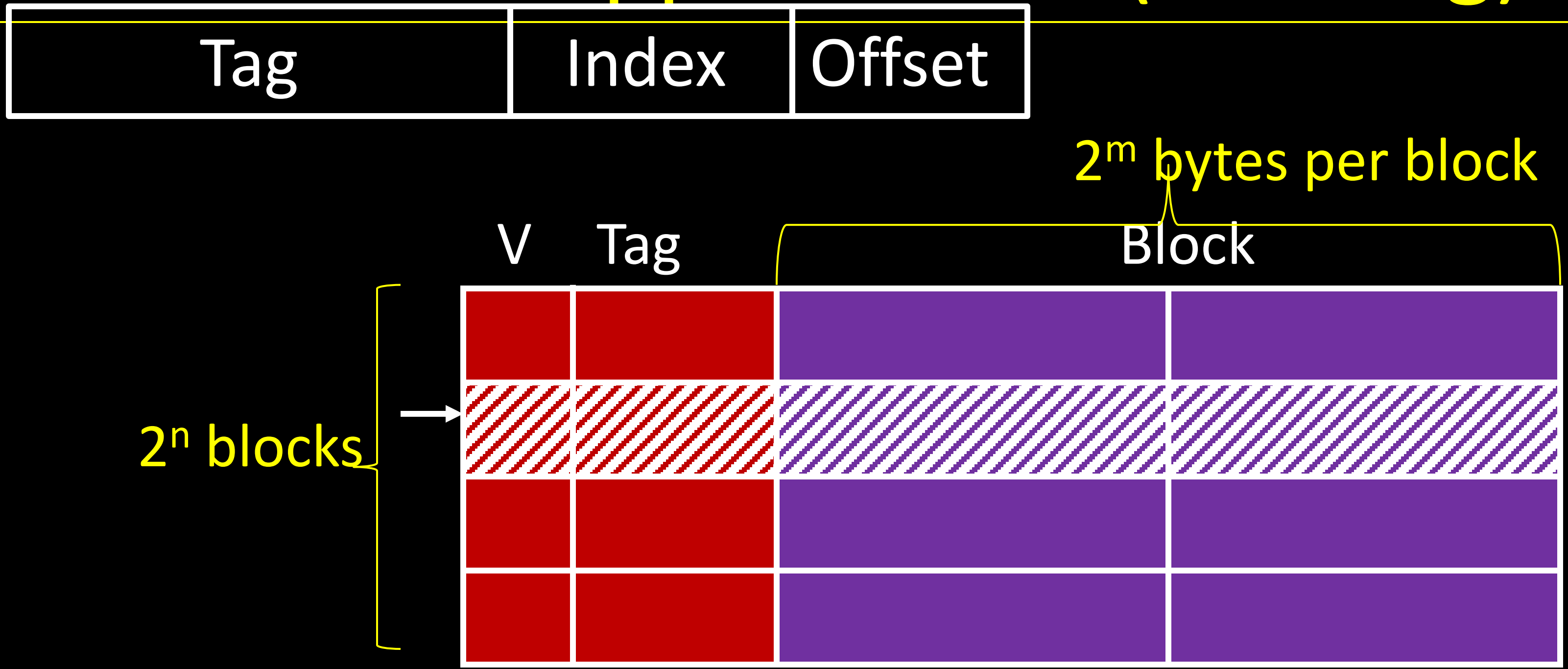
Cache Tradeoffs

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?	?

Direct Mapped Cache (Reading)



Direct Mapped Cache (Reading)



n bit index, *m* bit offset

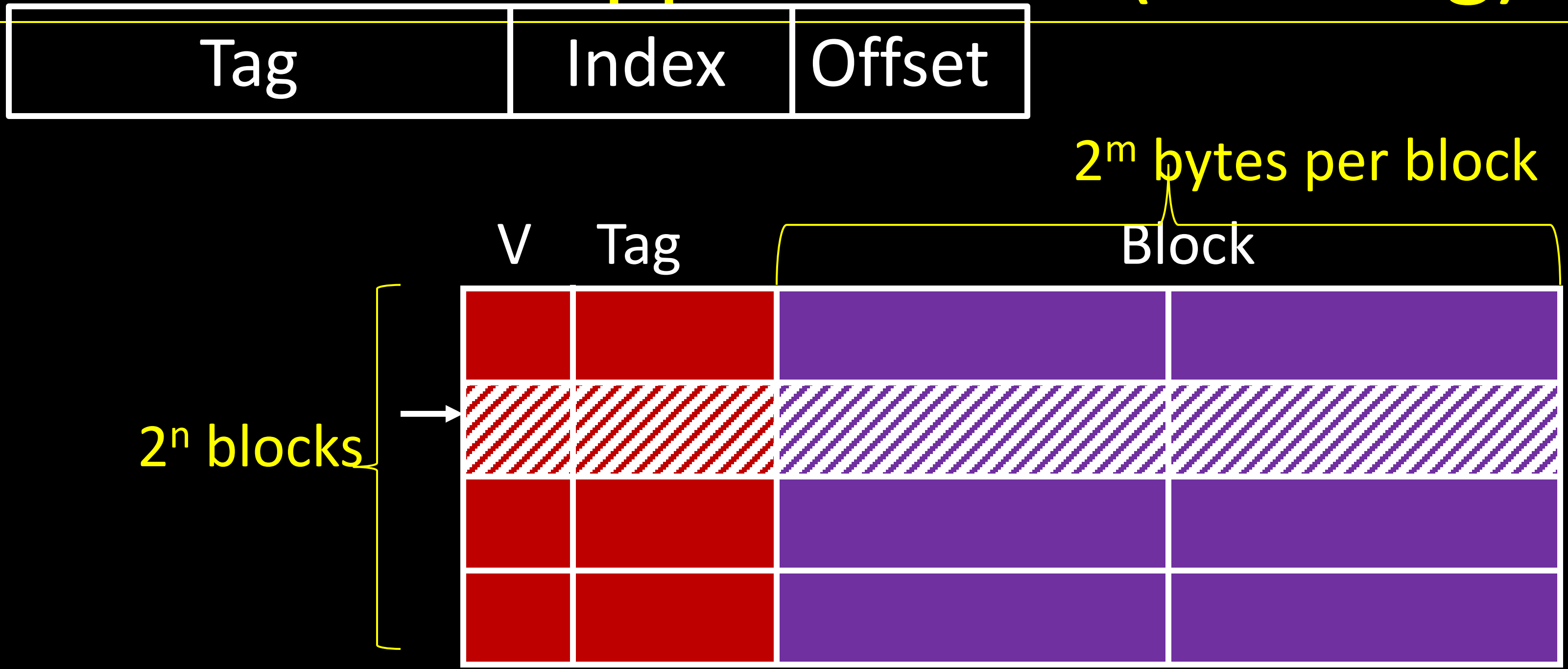
Q: How big is cache (***data only***)?

Cache of size 2^n blocks

Block size of 2^m bytes

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

Direct Mapped Cache (Reading)



n bit index, *m* bit offset

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

Cache of size 2^n blocks

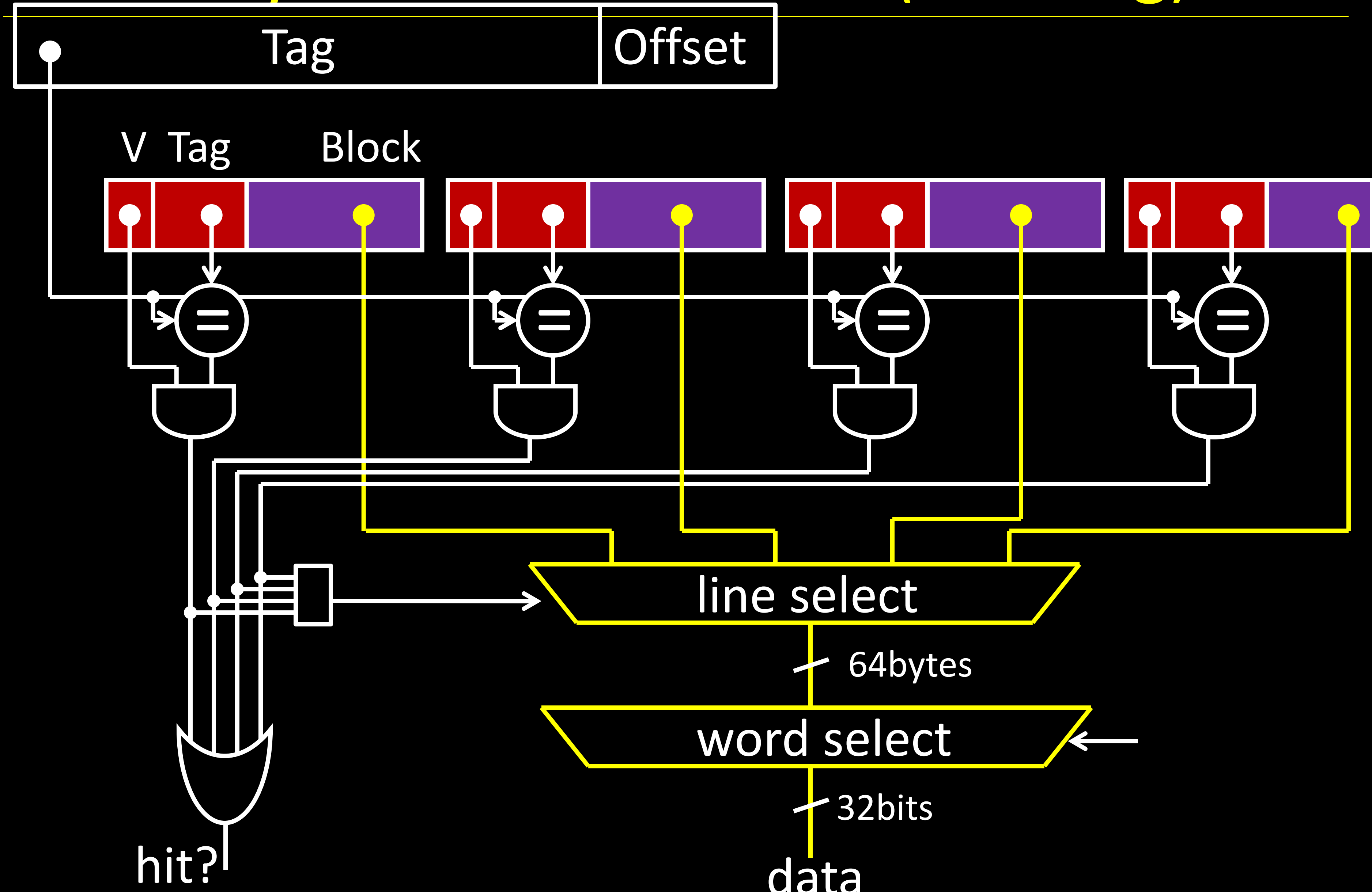
Block size of 2^m bytes

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

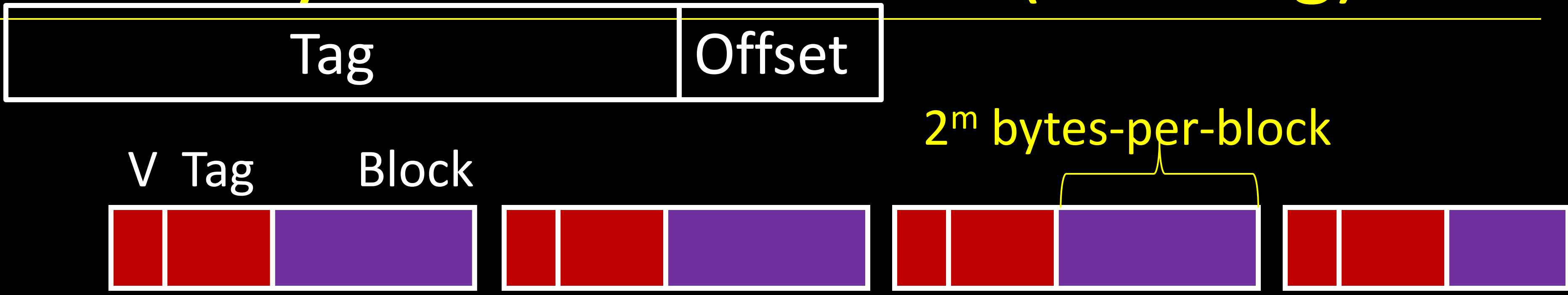
SRAM Size: 2^n x (block size + tag size + valid bit size)

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

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

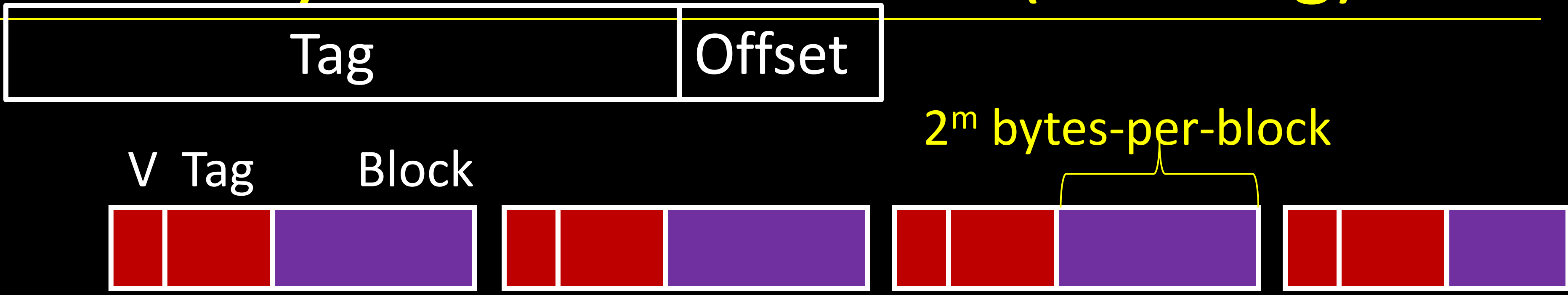
Block size of 2^m bytes

Cache Size: number-of-blocks x block size

$= 2^n \times 2^m$ 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)

Cache Tradeoffs

Compromise: Set-associative cache

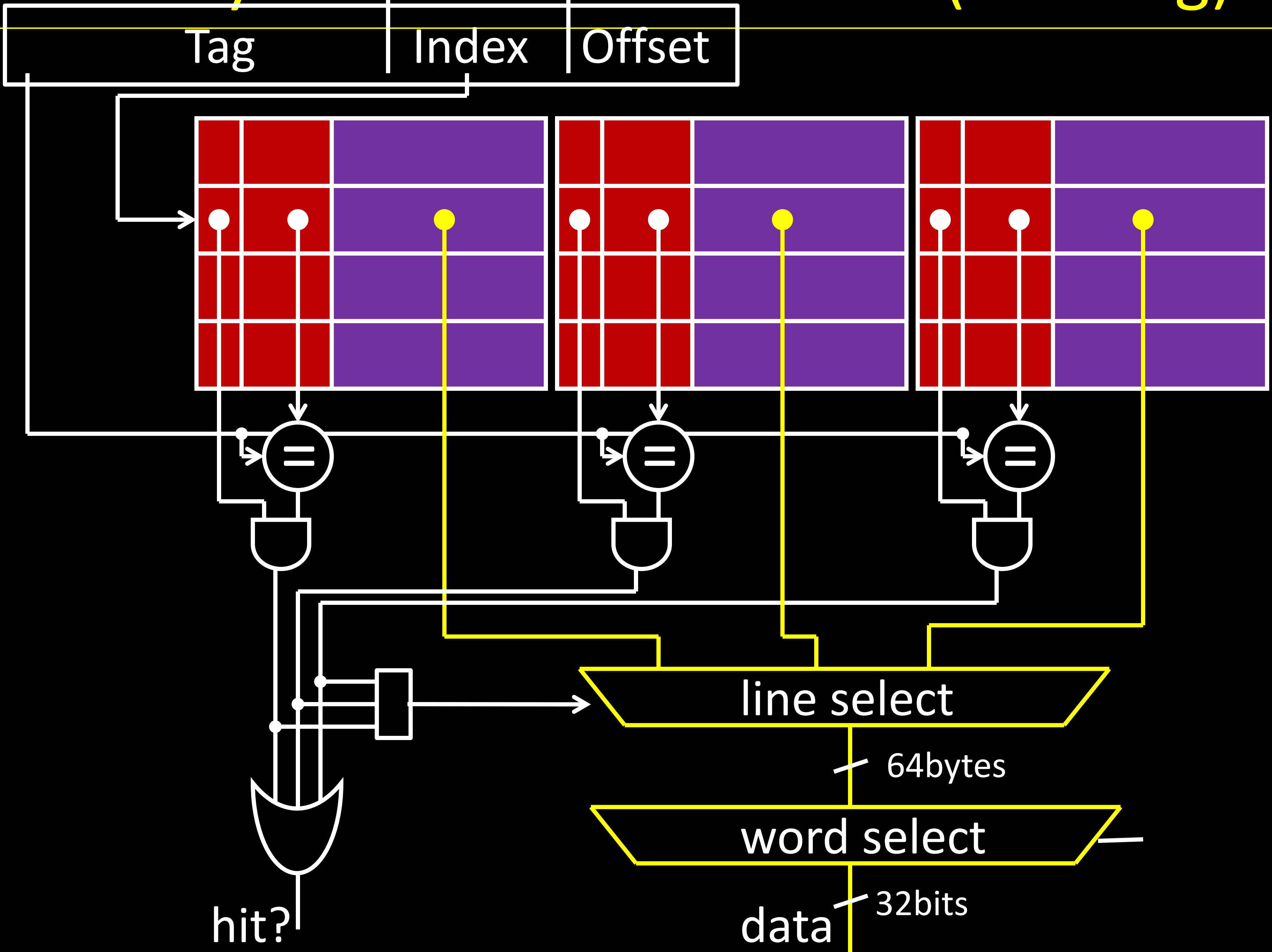
Like a direct-mapped cache

- Index into a location
- Fast

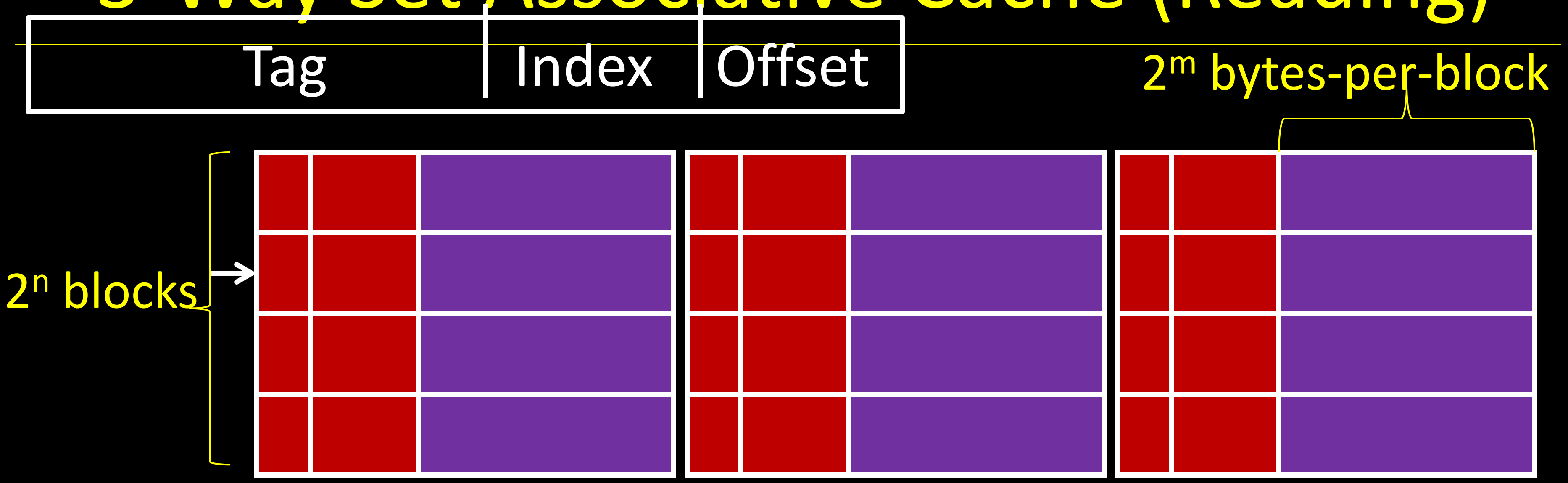
Like a fully-associative cache

- Can store multiple entries
  - decreases thrashing in cache
- Search in each element

3-Way Set Associative Cache (Reading)



3-Way Set Associative Cache (Reading)



n bit index, *m* bit offset, **N-way Set Associative**

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

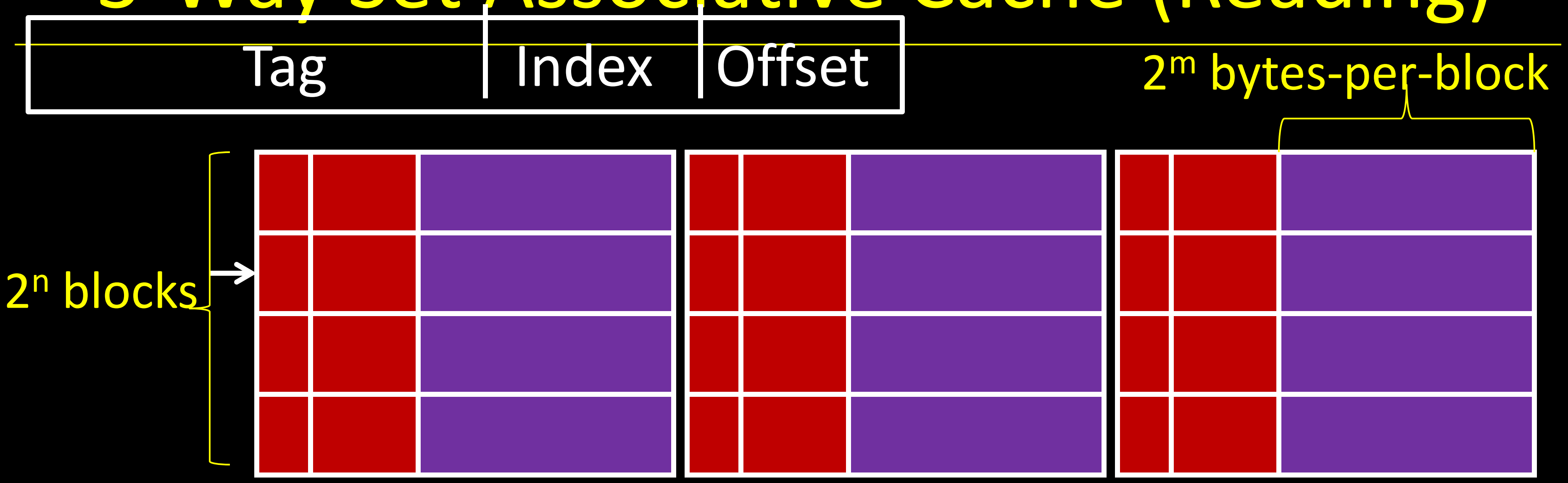
Cache of size 2^n sets

Block size of 2^m bytes, N-way set associative

Cache Size: 2^m bytes-per-block x (2^n sets x N-way-per-set)

= N x 2^{n+m} bytes

3-Way Set Associative Cache (Reading)



n bit index, *m* bit offset, **N-way Set Associative**

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

Cache of size 2^n sets

Block size of 2^m bytes, N-way set associative

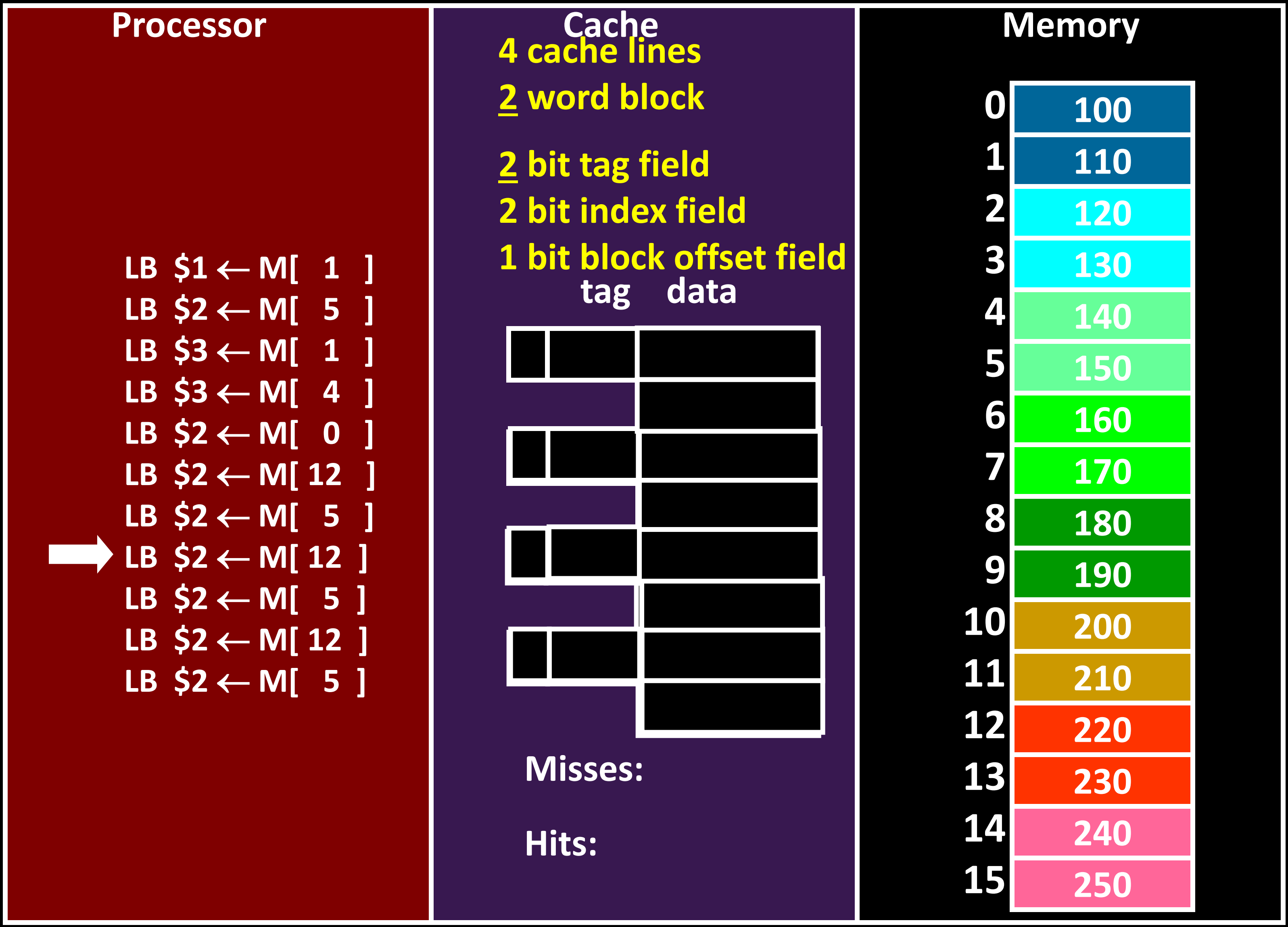
Tag field: 32 – (n + m), Valid bit: 1

SRAM Size: 2^n sets x N-way-per-set x (block size + tag size + valid bit size)

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

Comparison: Direct Mapped

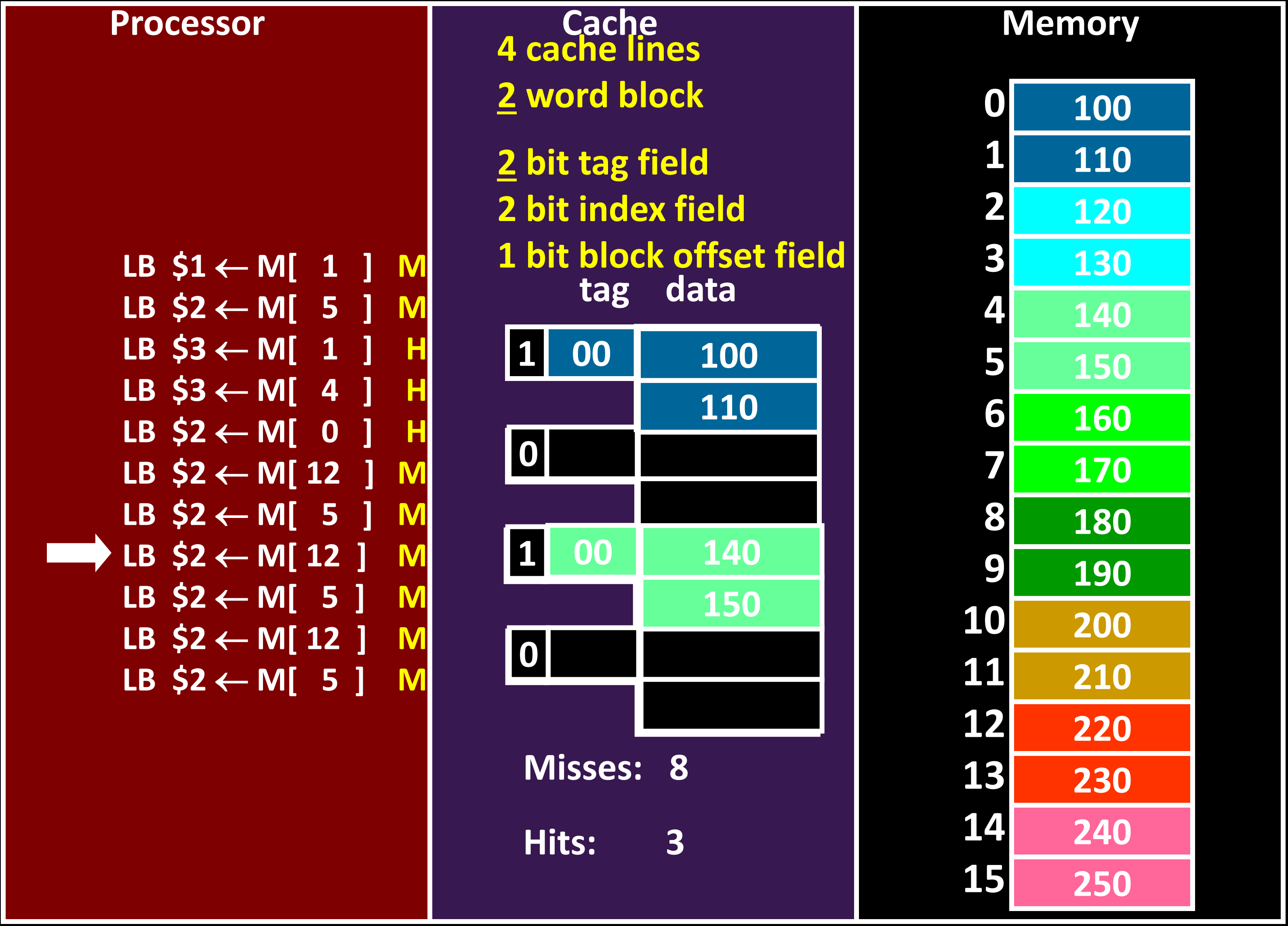
Using **byte addresses** in this example! Addr Bus = 5 bits



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: Direct Mapped

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



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]	M
	LB $2 ← M[5]	M
	LB $2 ← M[12]	M
	LB $2 ← M[5]	M

Cache

4 cache lines
<u>2</u> word block

<u>2</u> bit tag field
2 bit index field
1 bit block offset field

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

Misses: 8

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

Comparison: Fully Associative

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

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 ]
```

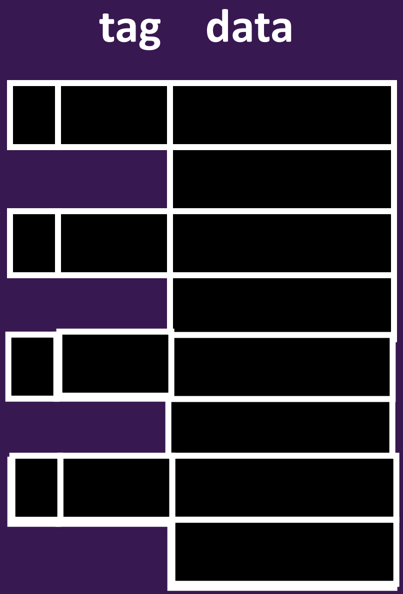
Cache

4 cache lines

2 word block

4 bit tag field

1 bit block offset field



Misses:

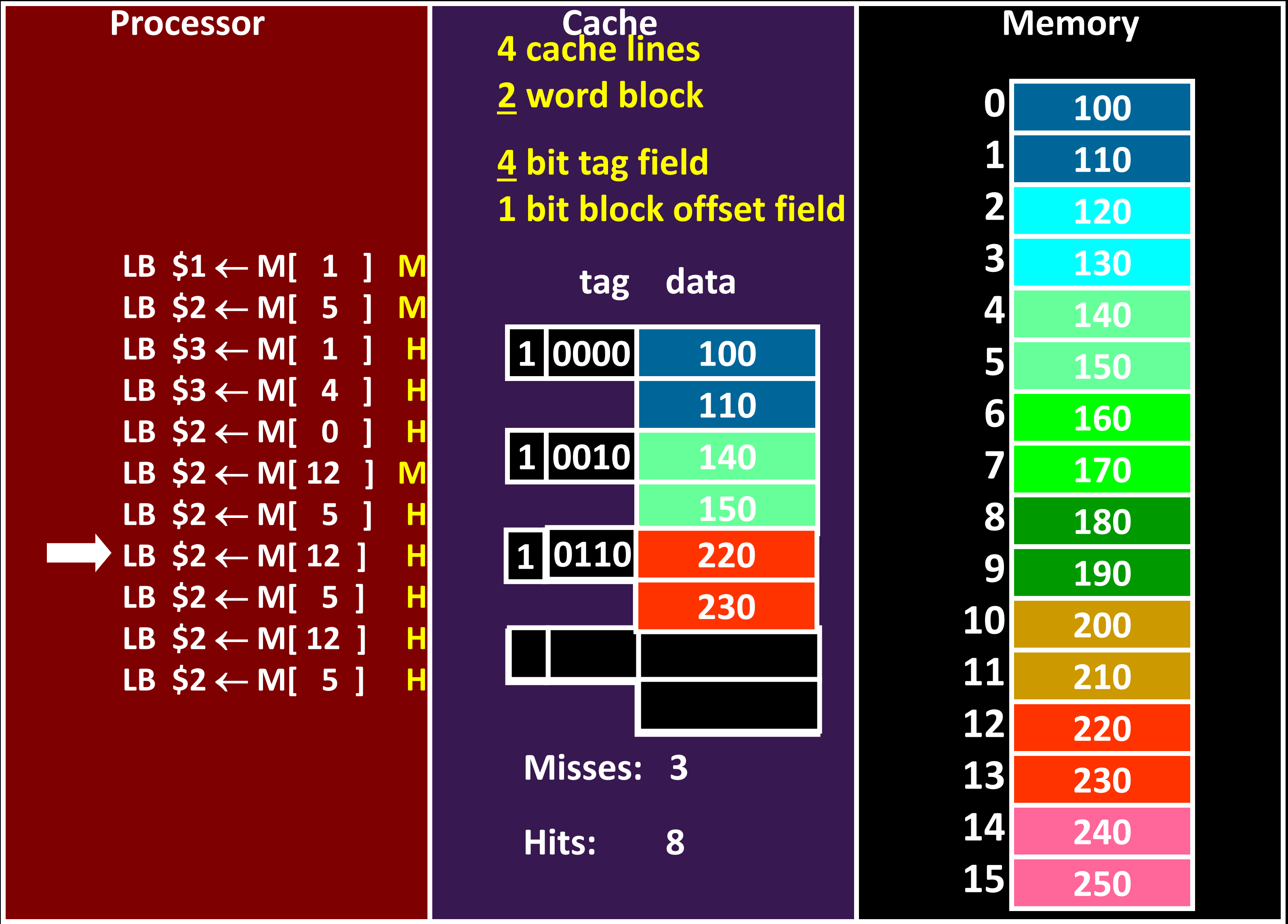
Hits:

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: Fully Associative

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



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
LB  $2 ← M[ 12 ]  H
LB  $2 ← M[ 5 ]   H
LB  $2 ← M[ 12 ]  H
LB  $2 ← M[ 5 ]   H
```

Cache

4 cache lines

2 word block

4 bit tag field

1 bit block offset field

	tag	data
1	0000	100
		110
1	0010	140
		150
1	0110	220
		230

Misses: 3

Hits: 8

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

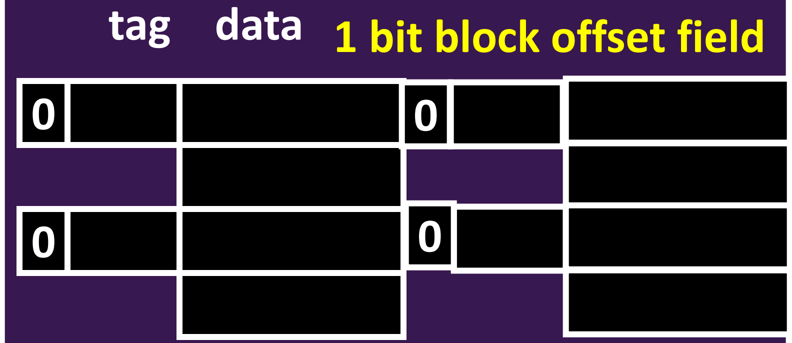
Using **byte addresses** in this example! Addr Bus = 5 bits

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

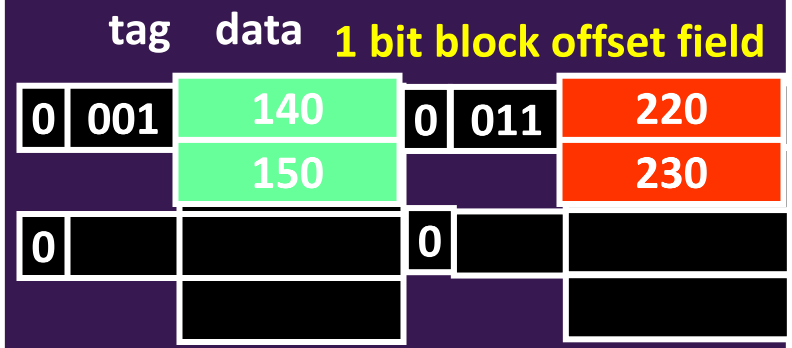
Using **byte addresses** in this example! Addr Bus = 5 bits

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

Misses

Cache misses: classification

The line is being referenced for the first time

- Cold (aka Compulsory) Miss

The line was in the cache, but has been evicted…

… because some other access with the same index

- Conflict Miss

… because the cache is too small

- i.e. the *working set* of program is larger than the cache
- Capacity Miss

Takeway

Direct Mapped → fast but low hit rate

Fully Associative → higher hit cost, but higher hit rate

N-way Set Associative → middleground

Next Goal

Do larger caches and larger cachelines (aka cache blocks) increase hit rate?

Larger Cachelines

Bigger does not always help

Mem access trace: 0, 16, 1, 17, 2, 18, 3, 19, 4, 20, …

Hit rate with four direct mapped 2-byte cache lines?

Hit rate = 0%

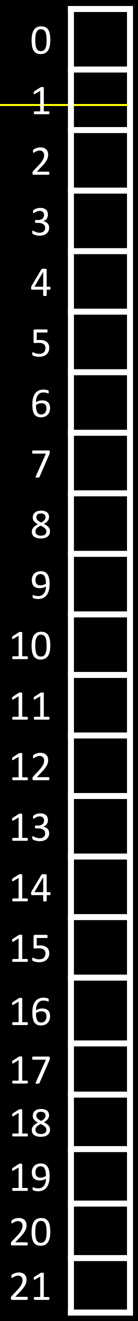
line 0	M[0] M[16]	M[1] M[17]
line 1	M[2] M[18]	M[3] M[19]
line 2		
line 3		

With eight 2-byte cache lines? Hit rate = 0%

With two 4-byte cache lines?

Hit rate = 0%

line 0	M[0] M[16]	M[1] M[17]	M[2] M[18]	M[3] M[19]
line 1				



Larger Cachelines

Fully-associative reduces conflict misses...

... assuming good eviction strategy

Mem access trace: 0, 16, 1, 17, 2, 18, 3, 19, 4, 20, ...

Hit rate with four fully-associative 2-byte cache lines?

Hit rate = 50%

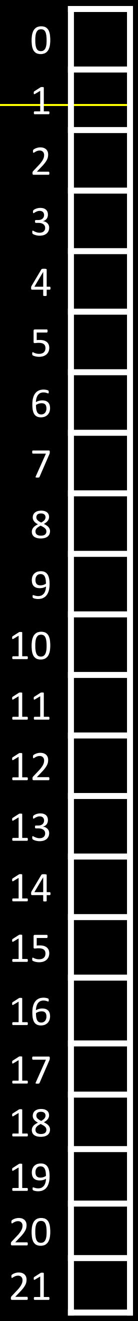
line 0	M[0] M[4]	M[1] M[20]
line 1	M[16]	M[17]
line 2	M[2]	M[3]
line 3	M[18]	M[19]

... A larger block size can often increase hit rate

With two fully-associative 4-byte cache lines?

Hit rate = 75%

line 0	M[0] M[4]	M[1] M[5]	M[2] M[6]	M[3] M[7]
line 1	M[16] M[20]	M[17] M[21]	M[18] M[22]	M[19] M[23]



Larger Cachelines

… but large block size can still reduce hit rate

Mem access trace: 0, 100, 200, 1, 101, 201, 2, 102, 202,…

Hit rate with four fully-associative 2-byte cache lines?

Hit rate = 50%

line 0	M[0] M[102]	M[1] M[103]
line 1	M[100] M[202]	M[101] M[203]
line 2	M[200]	M[201]
line 3	M[2]	M[3]

With two fully-associative 4-byte cache lines?

Hit rate = 0%

line 0	M[0] M[200]	M[1] M[201]	M[2] M[202]	M[3] M[203]
line 1	M[0] M[100]	M[1] M[101]	M[2] M[102]	M[3] M[103]

Misses

Cache misses: classification

Cold (aka Compulsory)

- The line is being referenced for the first time

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

Takeway

Direct Mapped → fast but low hit rate

Fully Associative → higher hit cost, but higher hit rate

N-way Set Associative → middleground

Cacheline size matters. Larger cache lines can increase performance due to prefetching. BUT, can also decrease performance is **working set** size cannot fit in cache.

Next Goal

Performance: What is the average memory access time (AMAT) for a cache?

AMAT = %hit x hit time + % miss x miss time

Cache Performance Example

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

16 words (i.e. 64 / 4 = 16)

Mem (DRAM): 4GB

Data cost: 50 cycle per word, plus 3 cycle per consecutive word

AMAT = %hit x hit time + % miss + miss time

Hit time = 5 cycles

Miss time = 2 (L1 lookup) + 3 (L1 access) + 50 (first word) + 15 x 3 (following word

= 100 cycles

If %hit = 90%, then

AMAT = .9 x 5 + .1 x 100 = 14.5 cycles

Performance depends on:

Access time for hit, miss penalty, hit rate

Takeway

Direct Mapped → fast but low hit rate

Fully Associative → higher hit cost, but higher hit rate

N-way Set Associative → middleground

Cacheline size matters. Larger cache lines can increase performance due to prefetching. BUT, can also decrease performance is **working set** size cannot fit in cache.

Ultimately, cache performance is measured by the average memory access time (AMAT), which depends cache architecture and size, but also the Access time for hit, miss penalty, hit rate

Goals for Today: caches

Comparison of cache architectures:

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

Writing to the Cache

- Write-through vs Write-back

Caching Questions

- How does a cache work?
- How effective is the cache (hit rate/miss rate)?
- How large is the cache?
- How fast is the cache (AMAT=average memory access time)

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

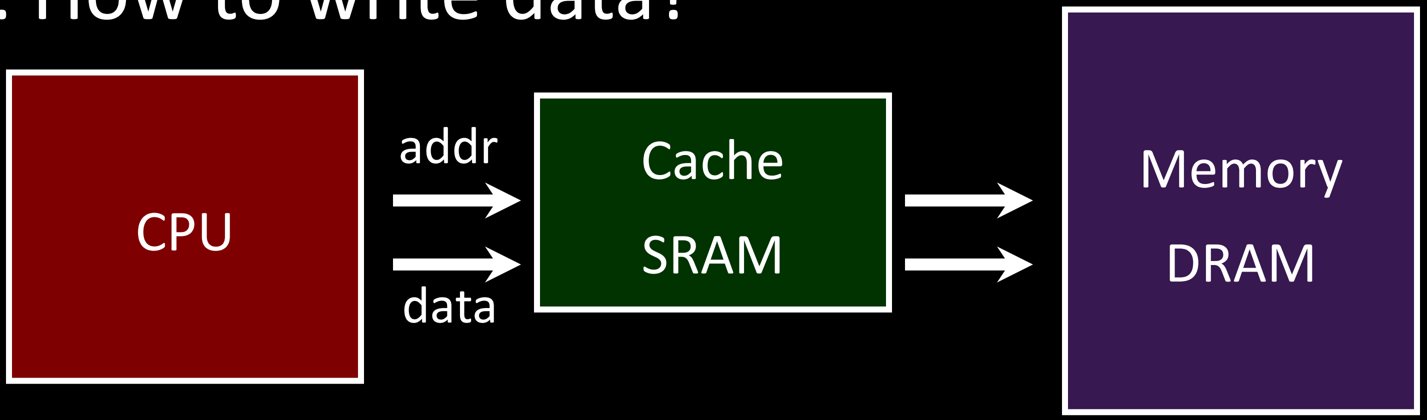
Next Goal

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)

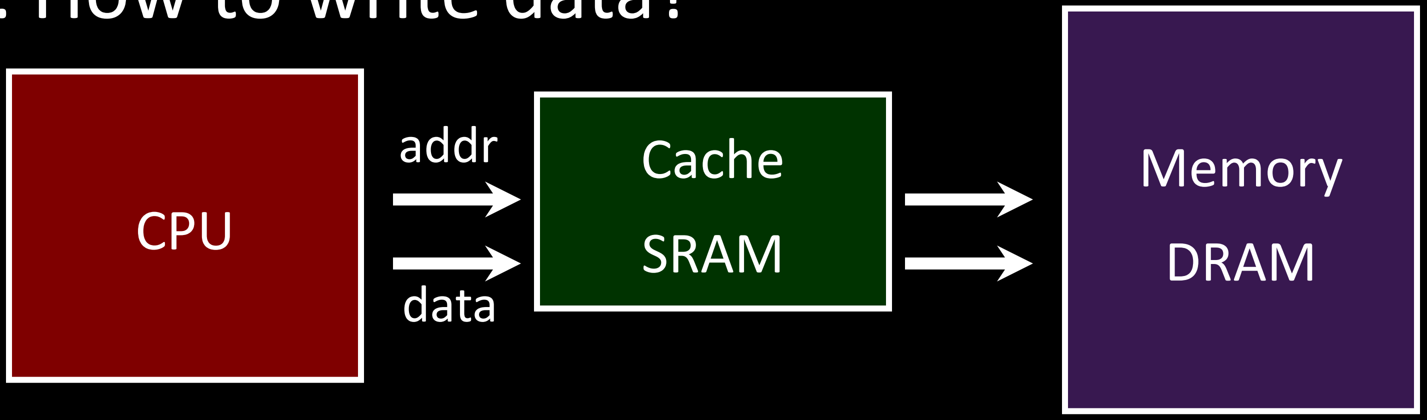
What about Stores?

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

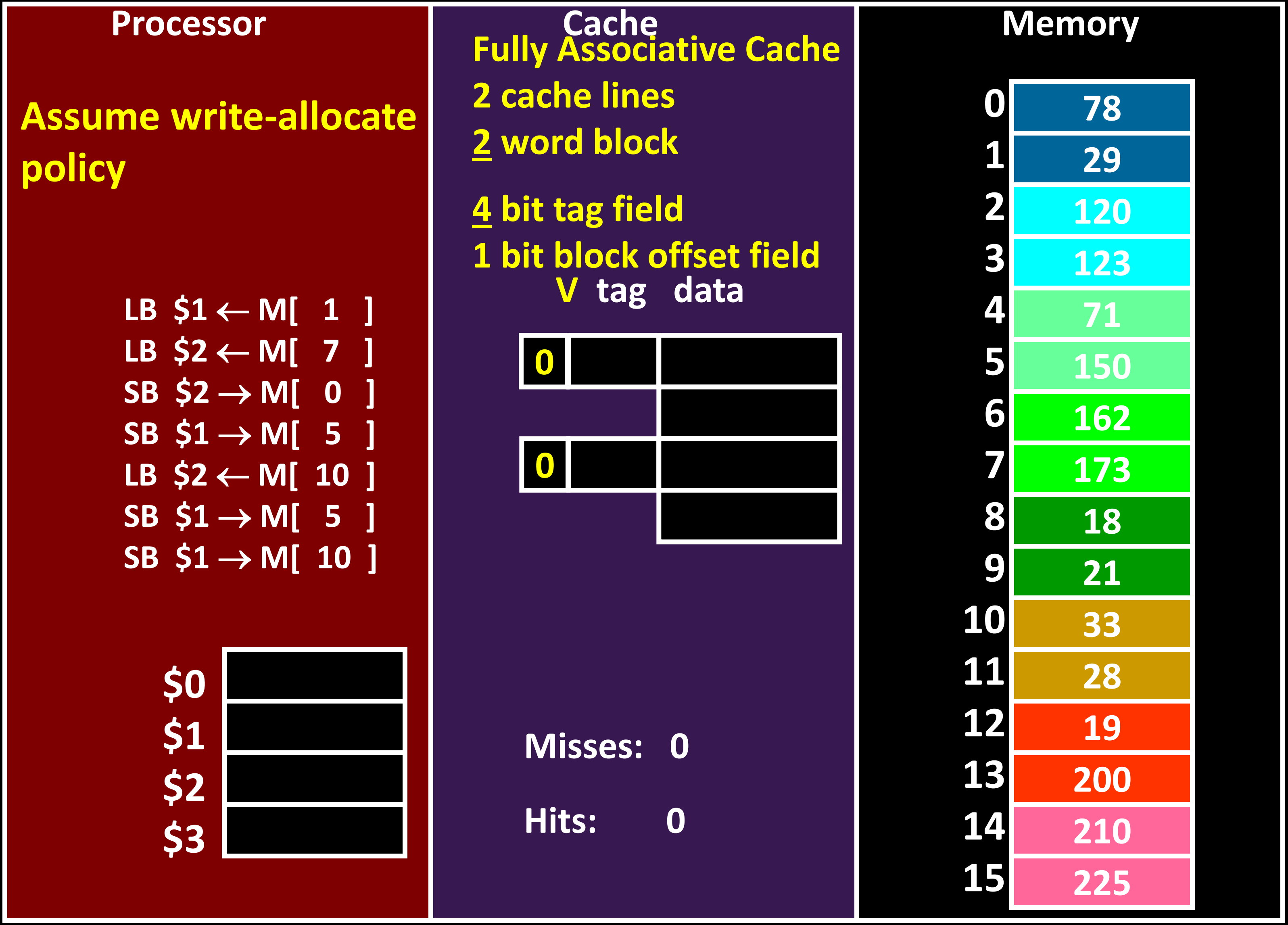
Next Goal

Example: How does a write-through cache work?

Assume write-allocate.

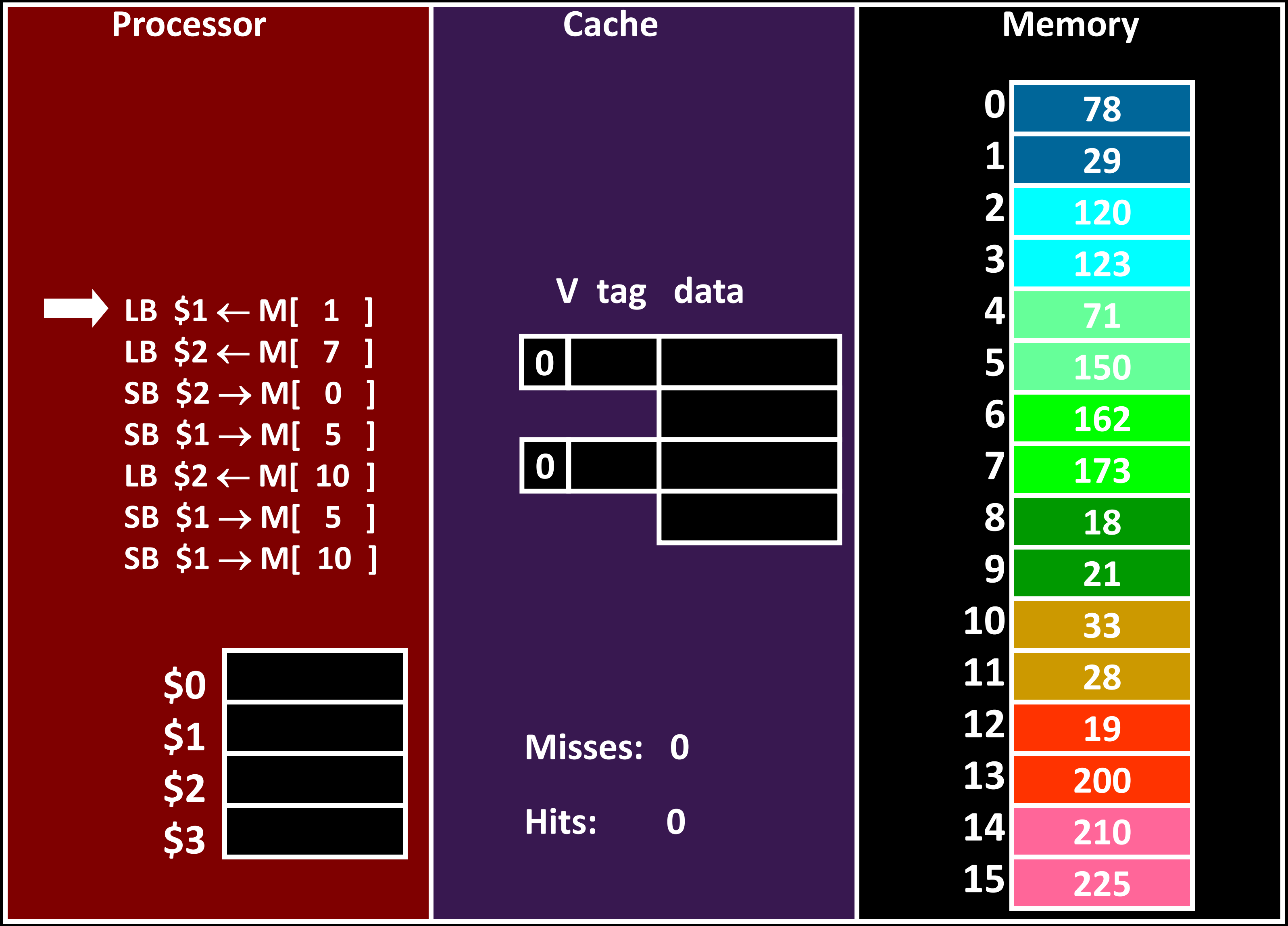
Handling Stores (Write-Through)

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

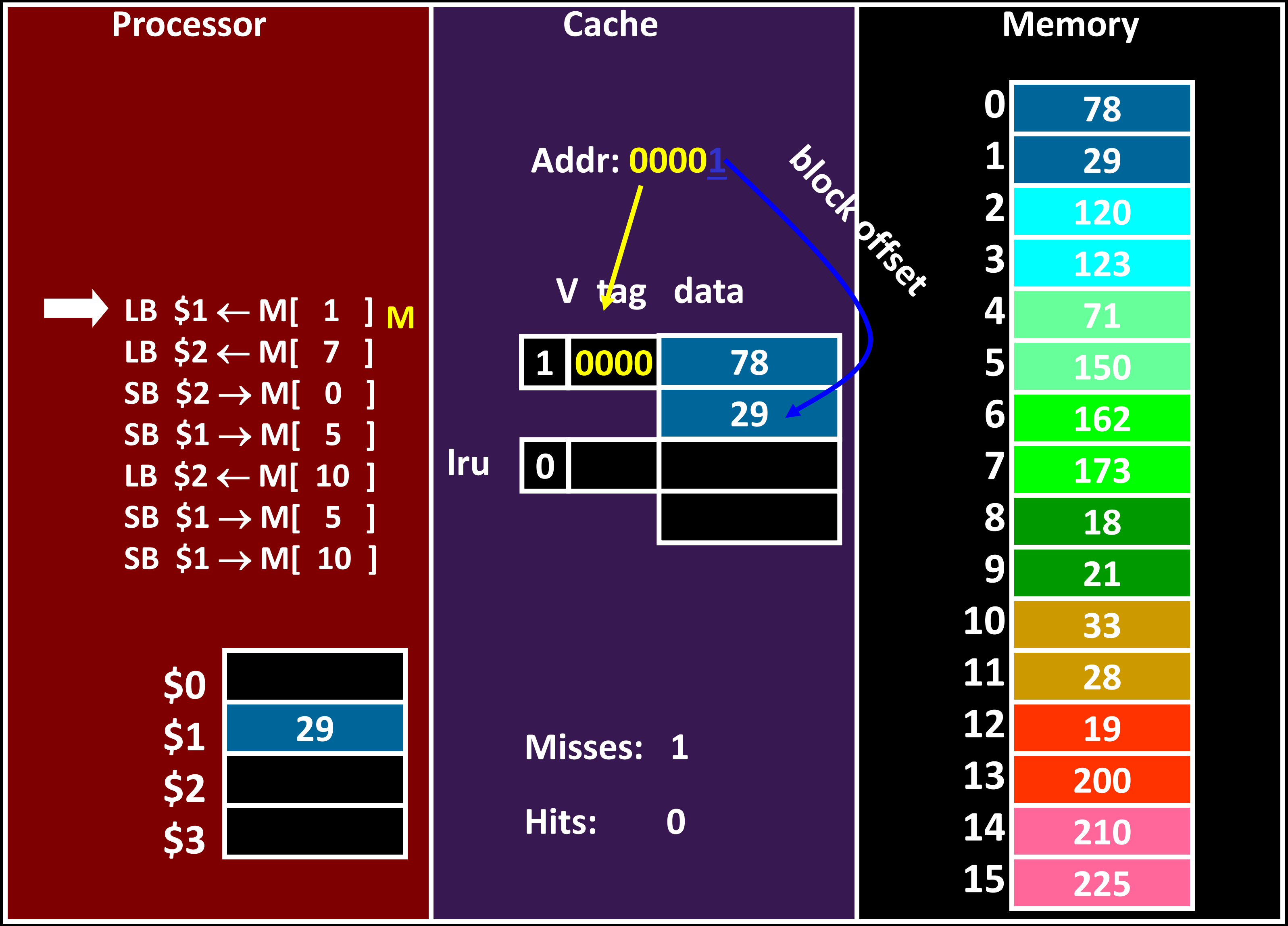


Memory address	Value
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

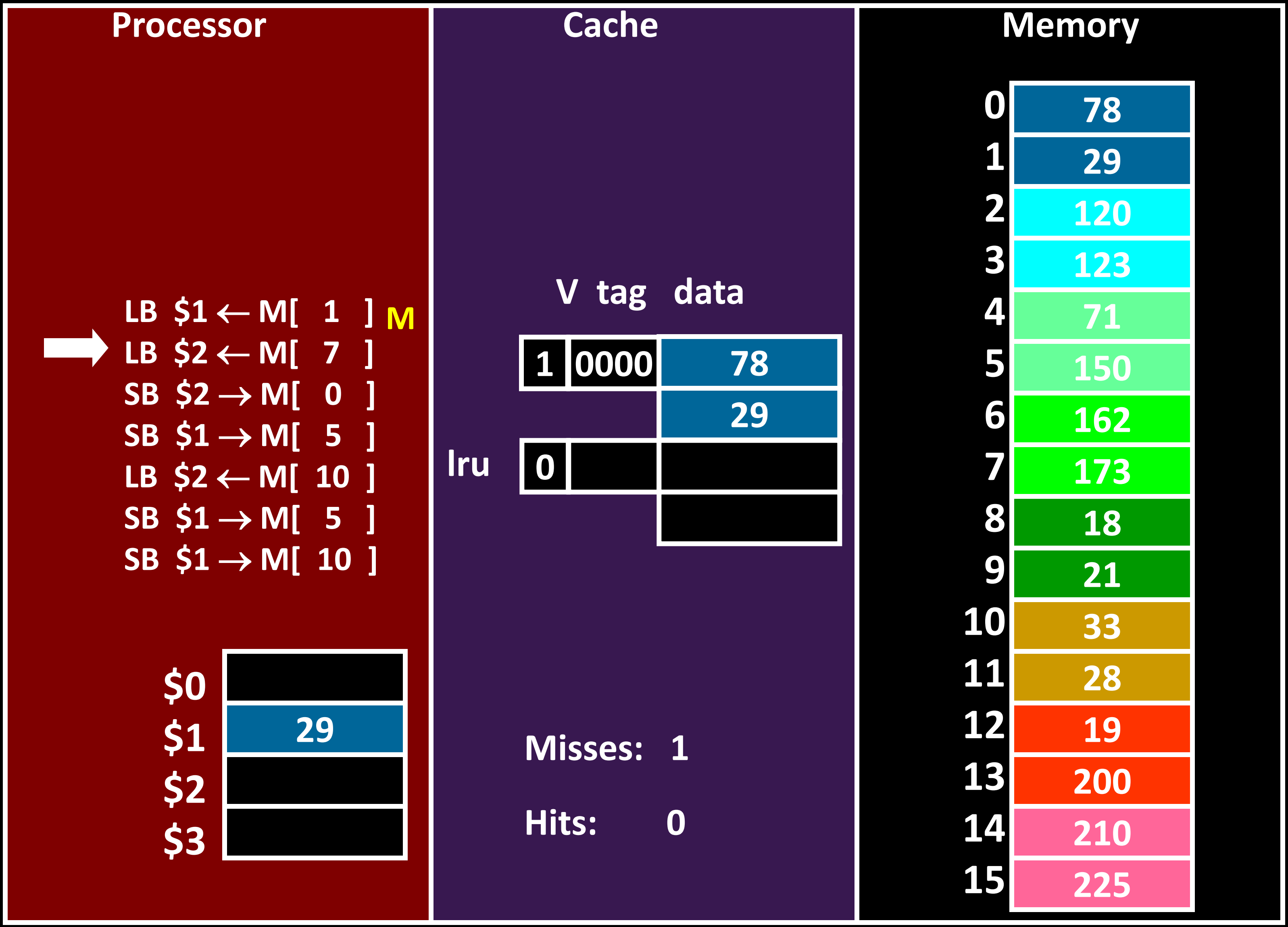
Write-Through (REF 1)



Write-Through (REF 1)



Write-Through (REF 2)



```
  LB  $1 ← M[  1  ]  M
→ LB  $2 ← M[  7  ]
  SB  $2 → M[  0  ]
  SB  $1 → M[  5  ]
  LB  $2 ← M[ 10  ]
  SB  $1 → M[  5  ]
  SB  $1 → M[ 10  ]
```

Register	Value
$0	
$1	29
$2	
$3	

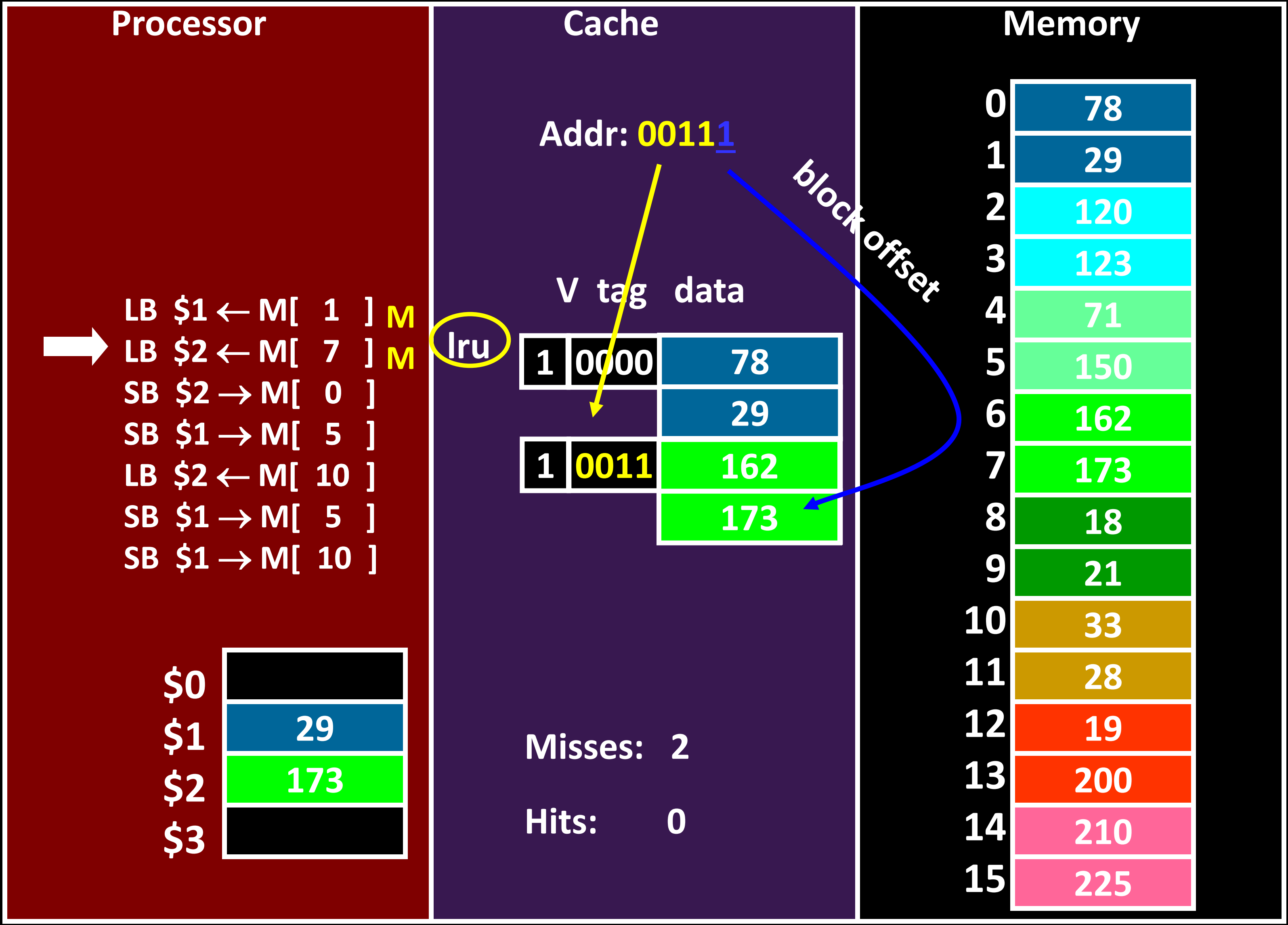
	V	tag	data
	1	0000	78
			29
lru	0		

Misses: 1

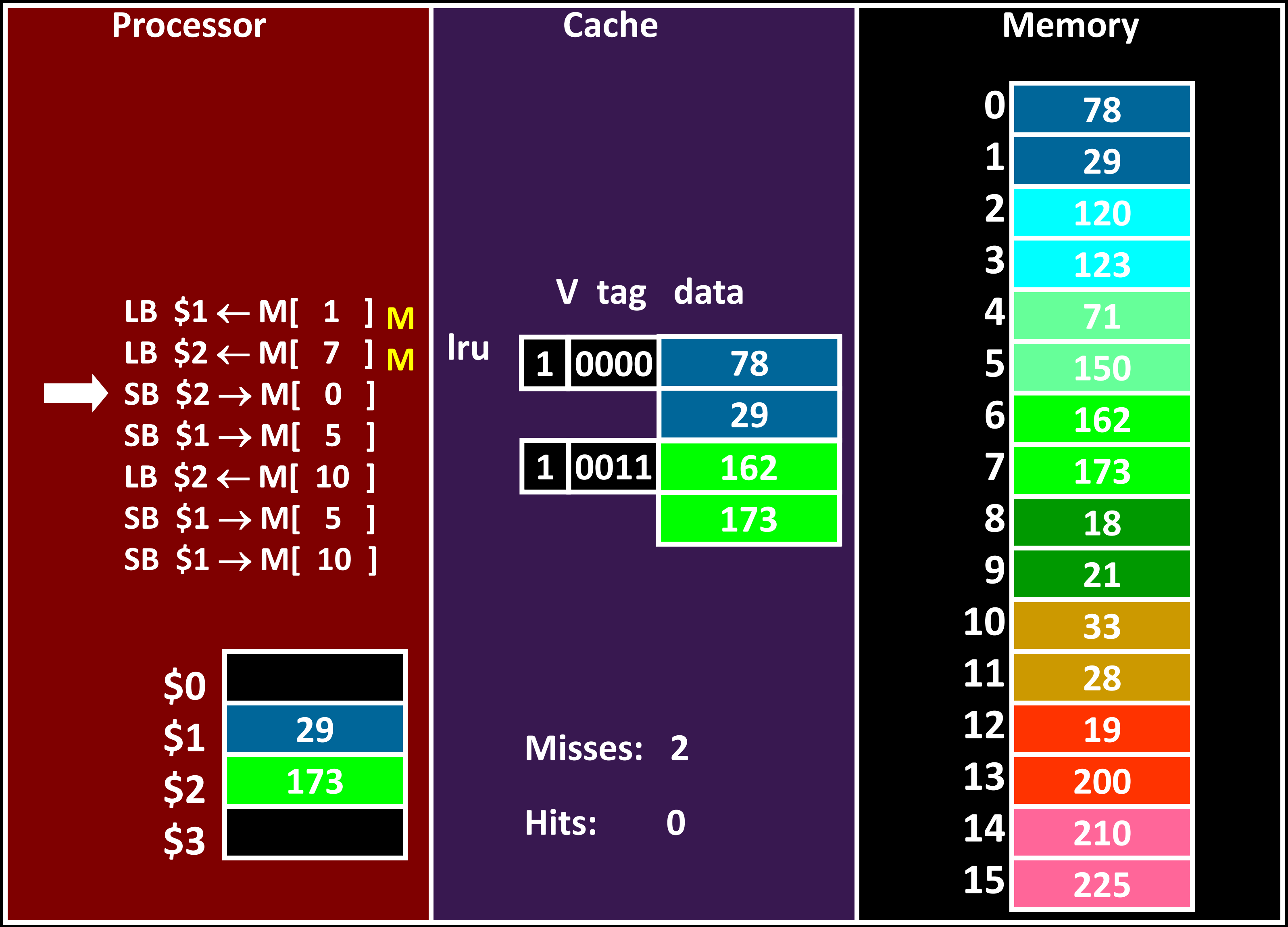
Hits: 0

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

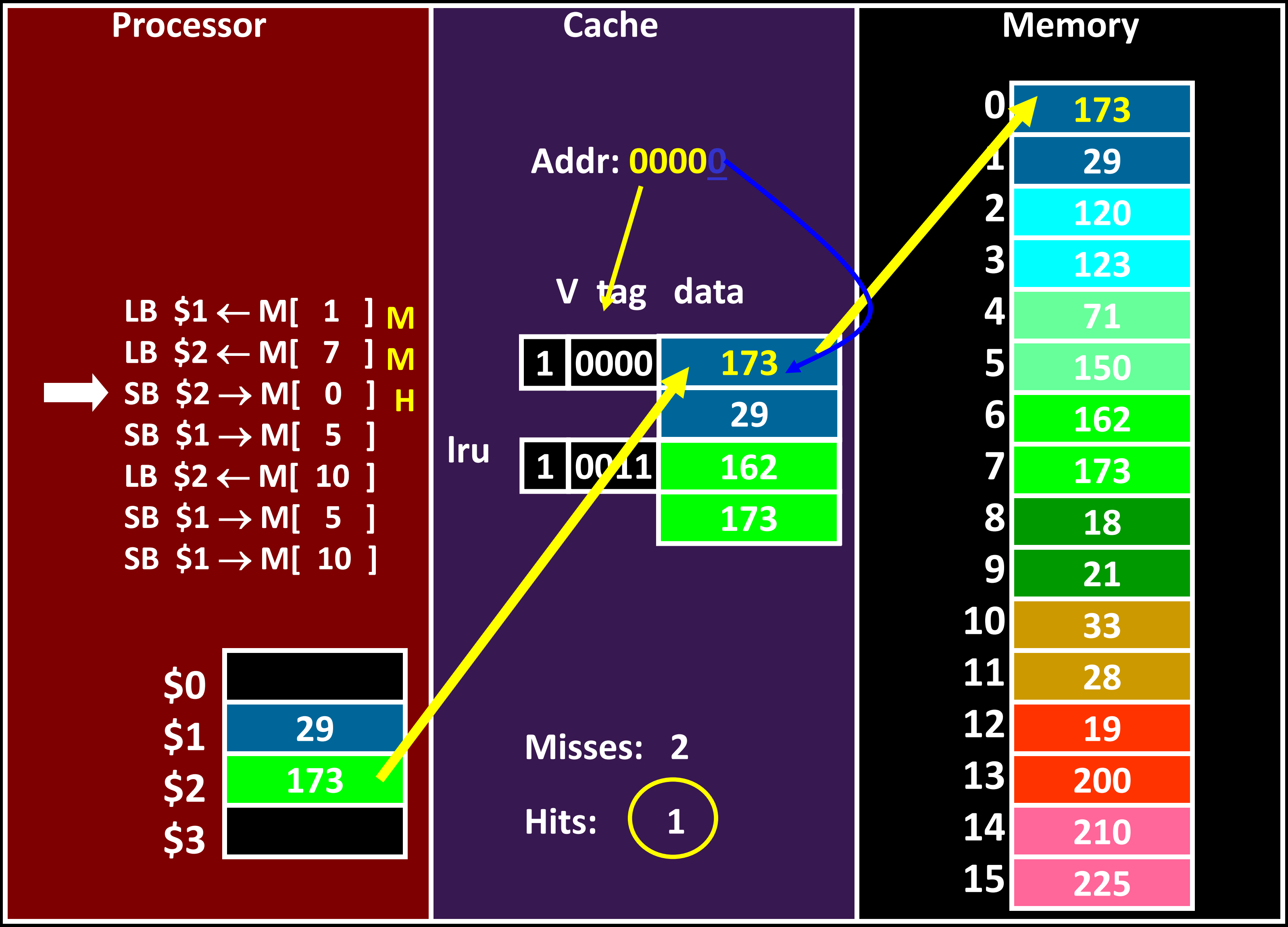
Write-Through (REF 2)



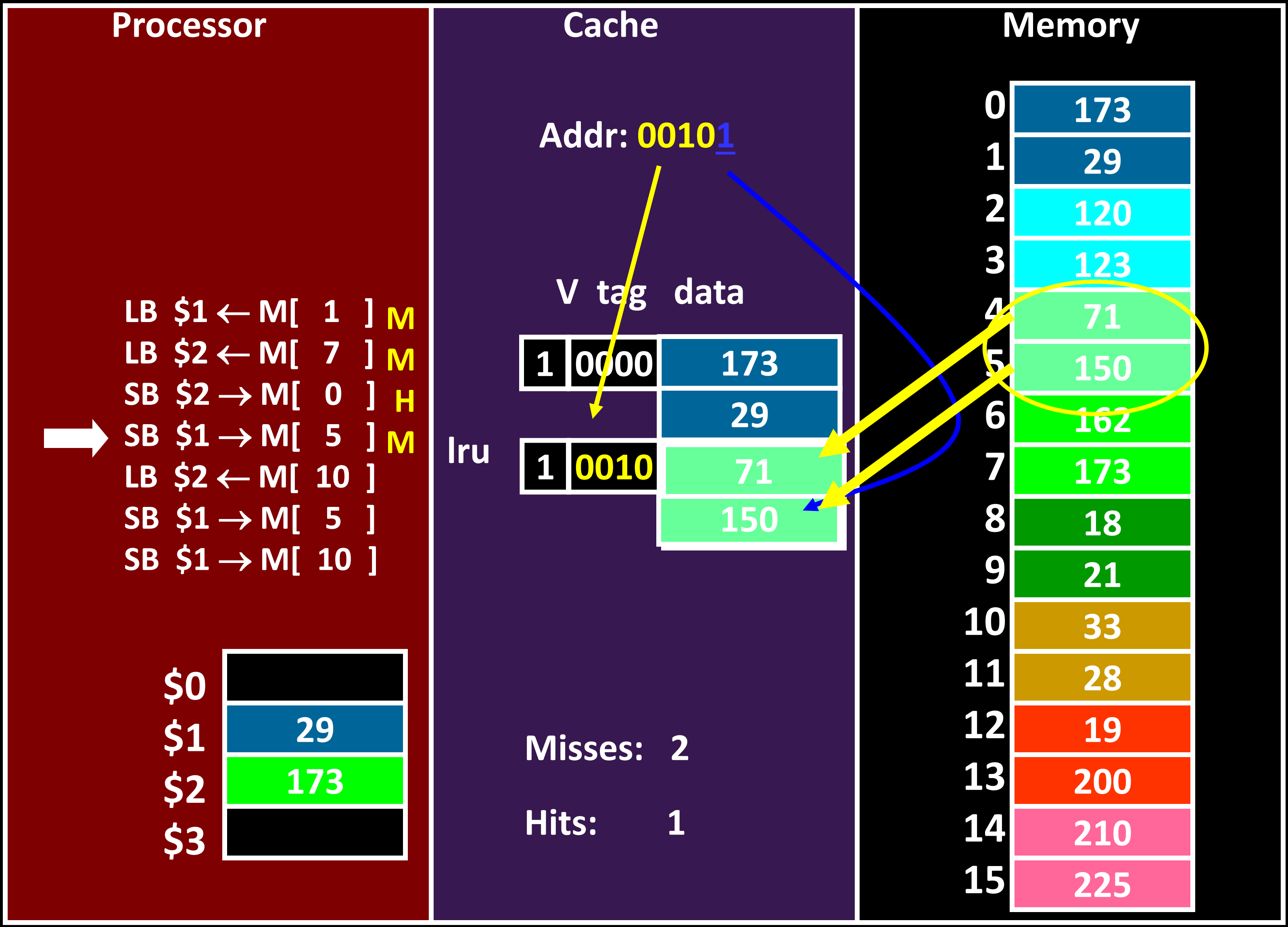
Write-Through (REF 3)



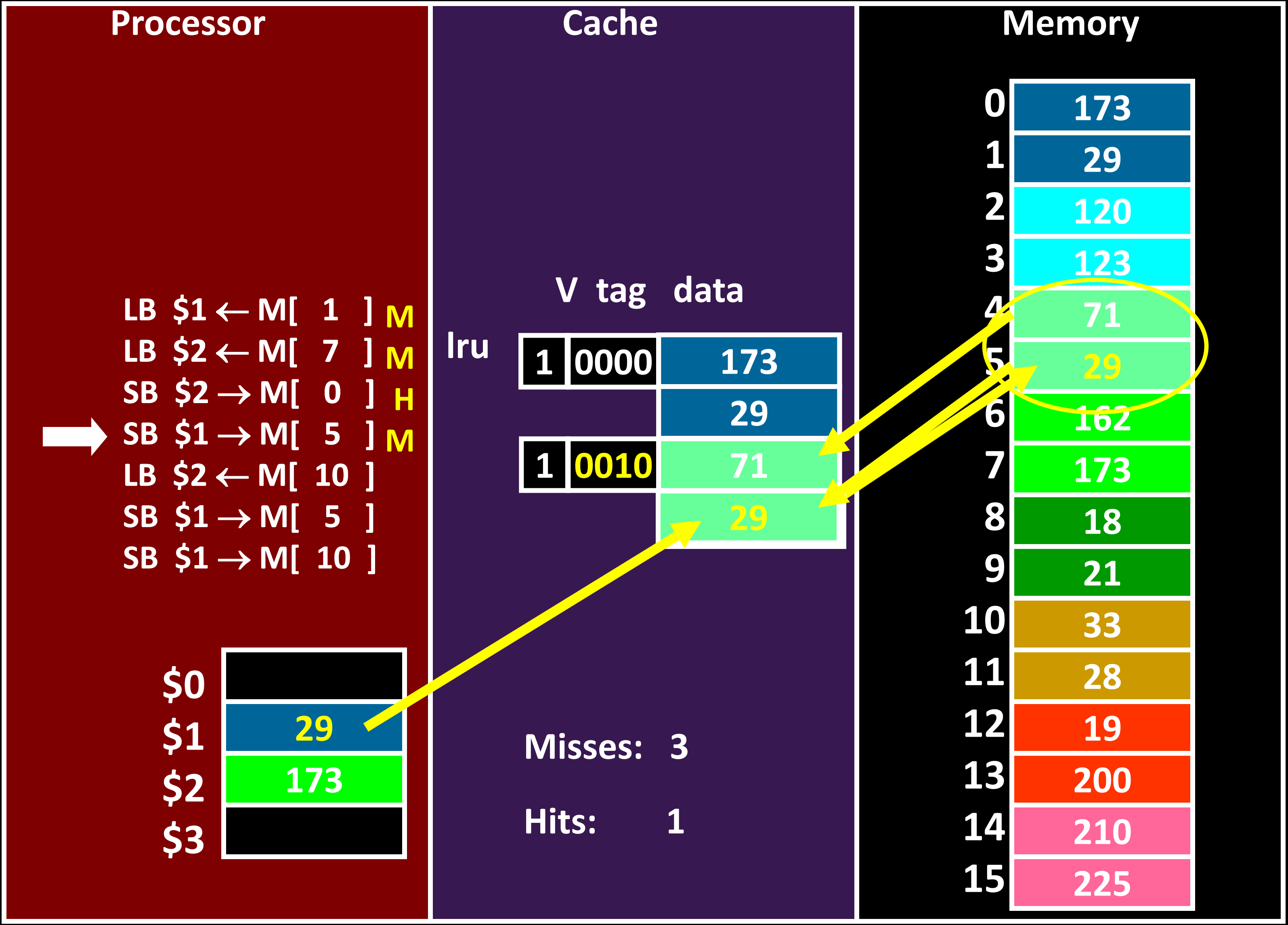
Write-Through (REF 3)



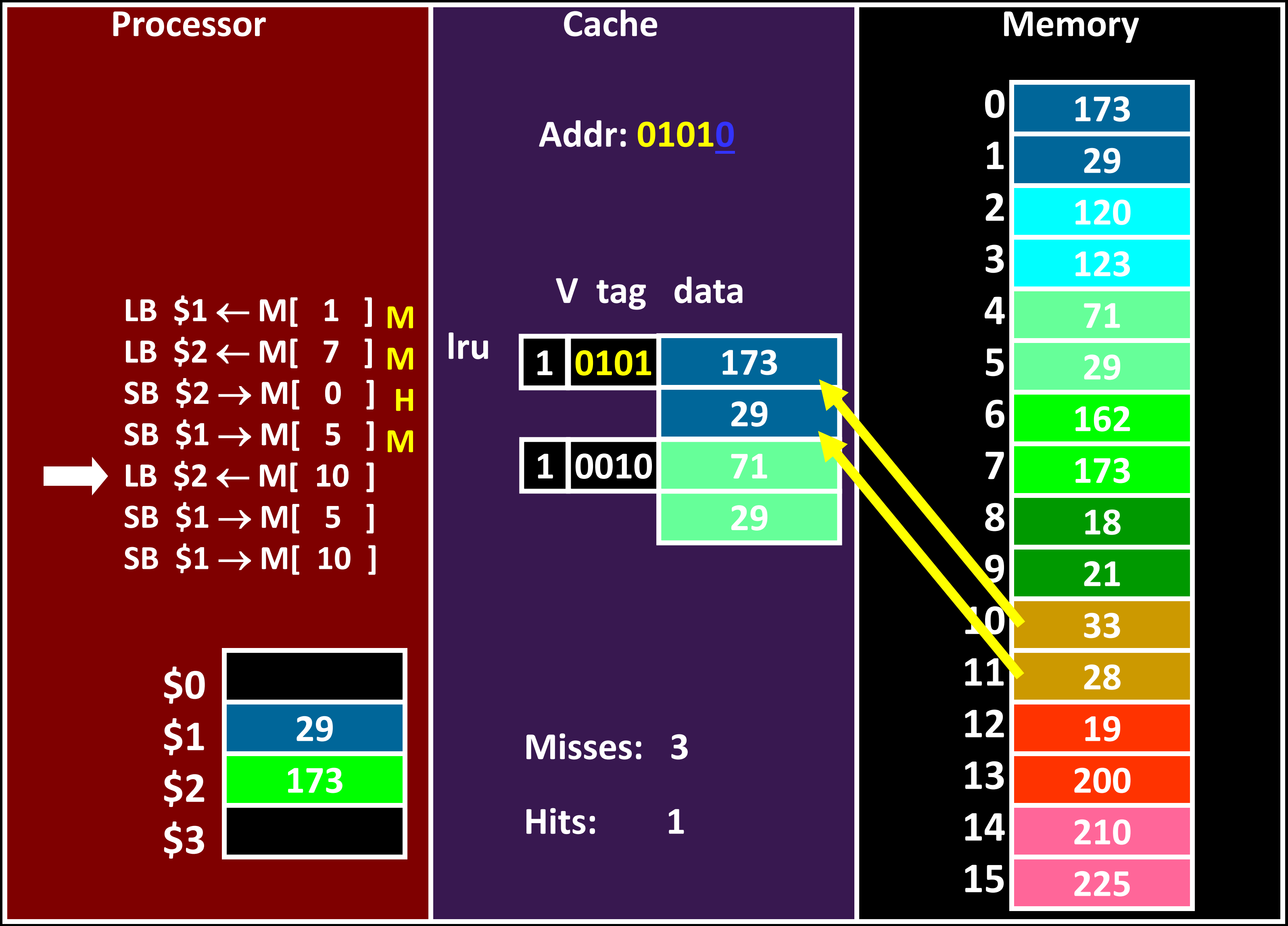
Write-Through (REF 4)



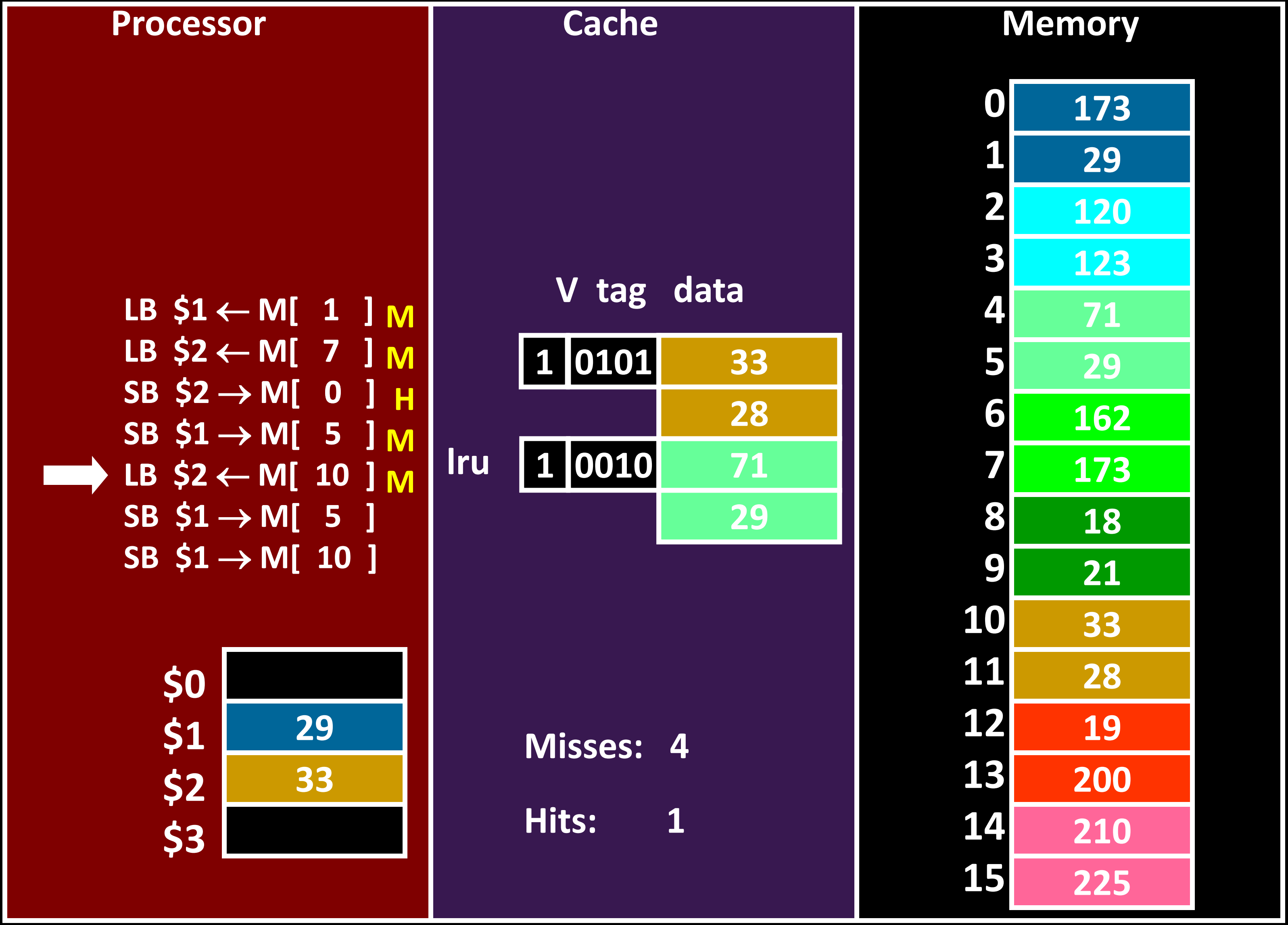
Write-Through (REF 4)



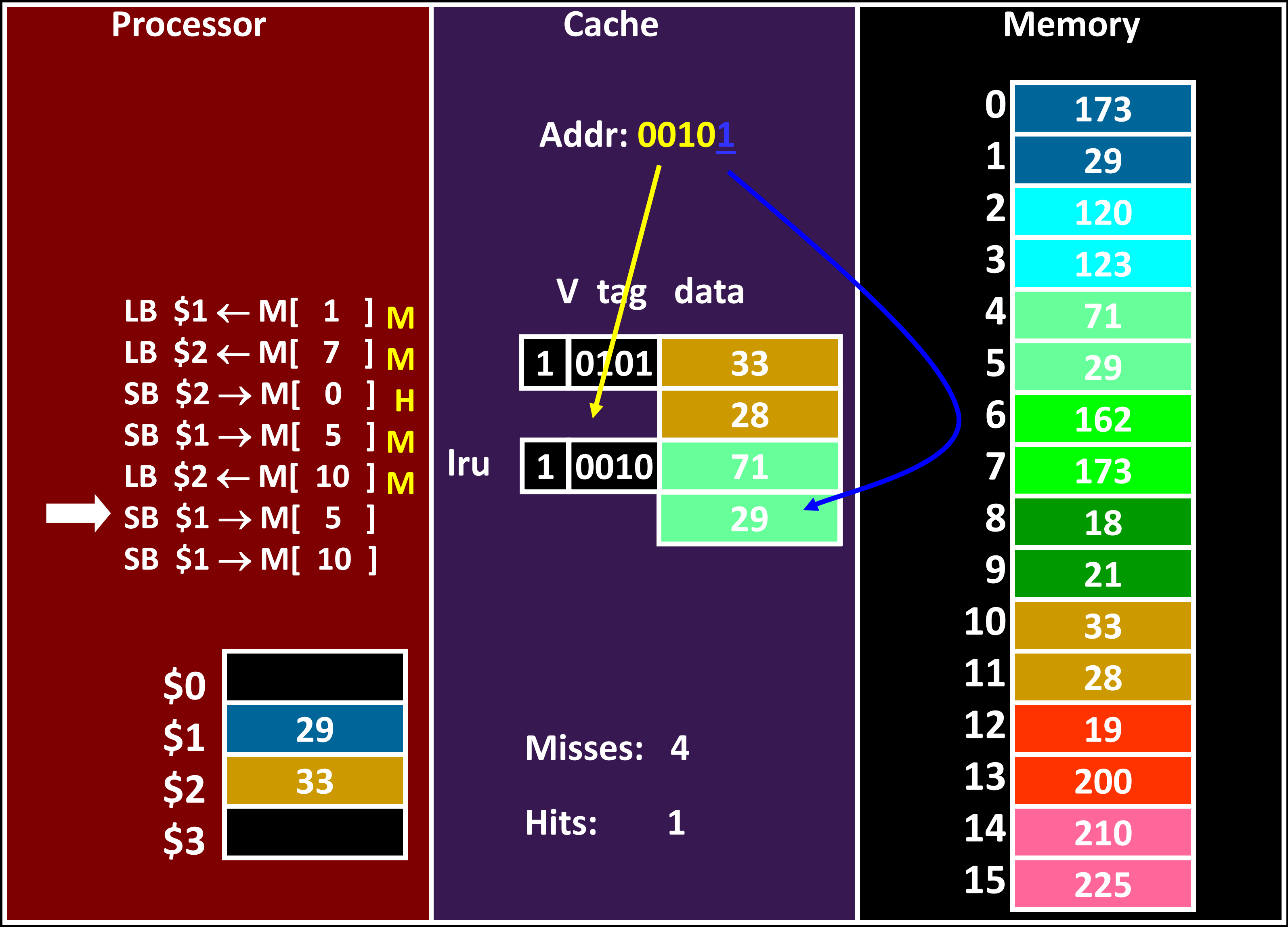
Write-Through (REF 5)



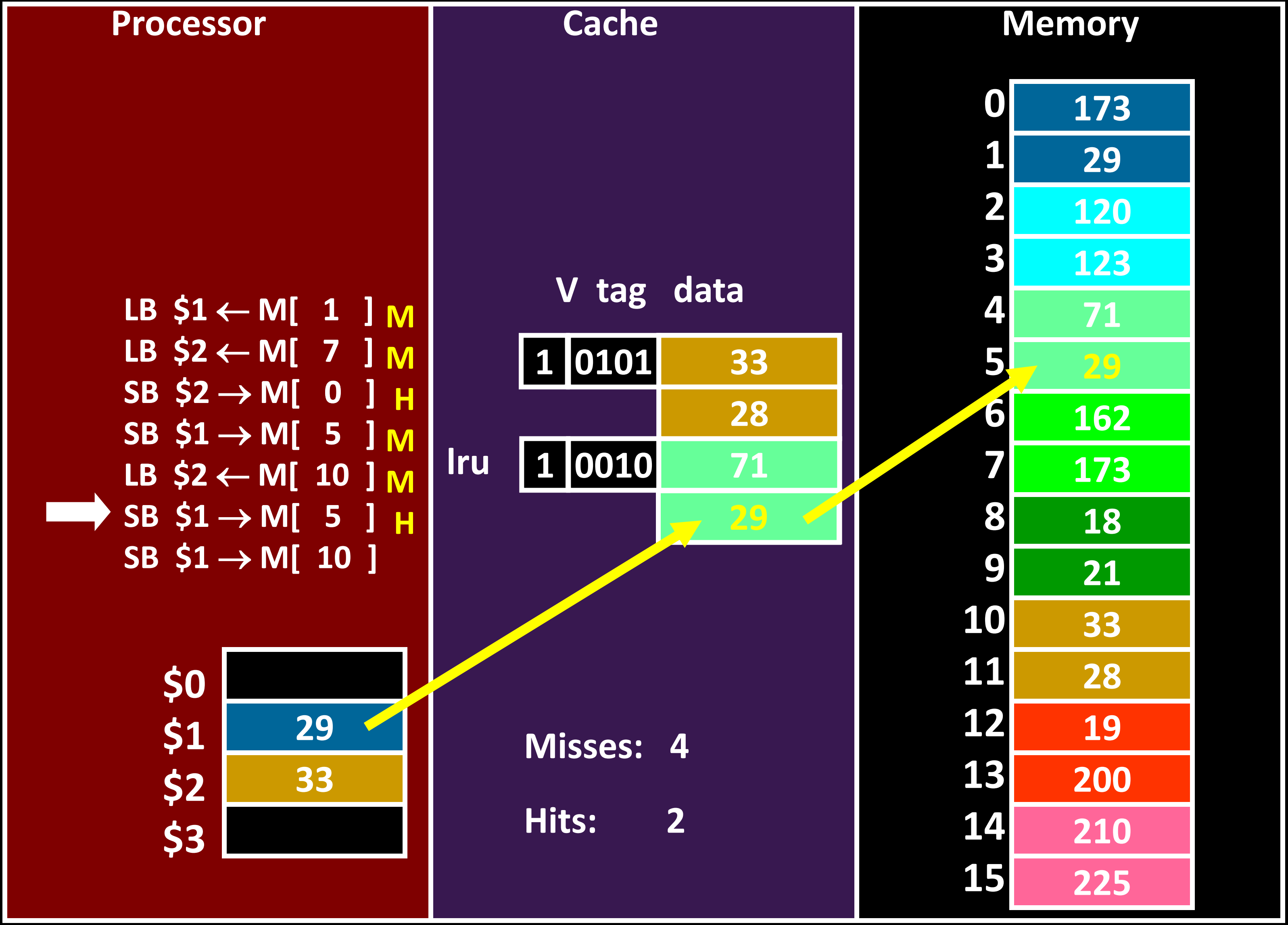
Write-Through (REF 5)



Write-Through (REF 6)



Write-Through (REF 6)



Write-Through (REF 7)

Processor

```
   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  ]
```

$0	
$1	29
$2	33
$3	

Cache

Addr: 0101<u>1</u>

	V	tag	data
	1	0101	33
			28
lru	1	0010	71
			29

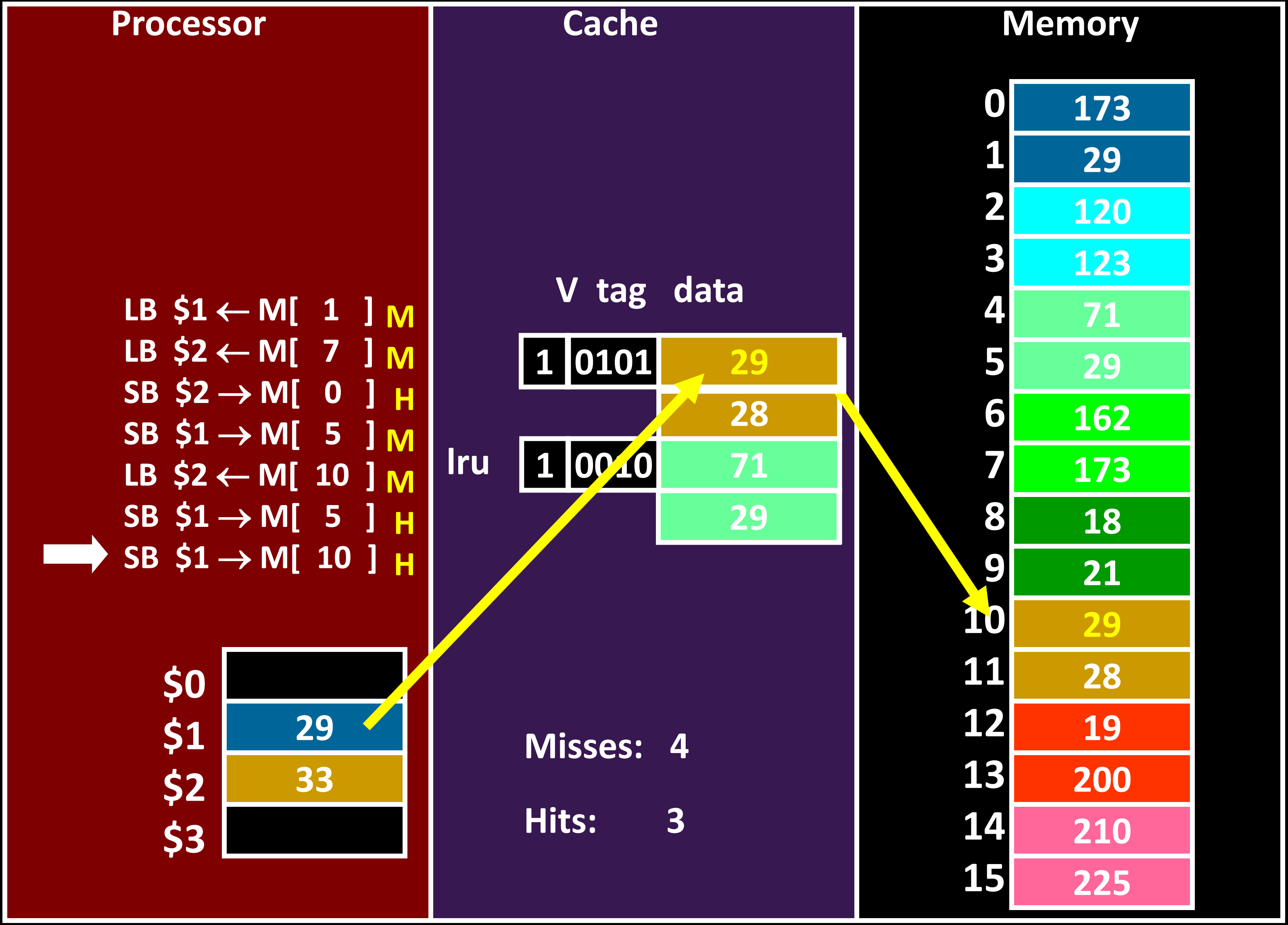
Misses: 4

Hits: 2

Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

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

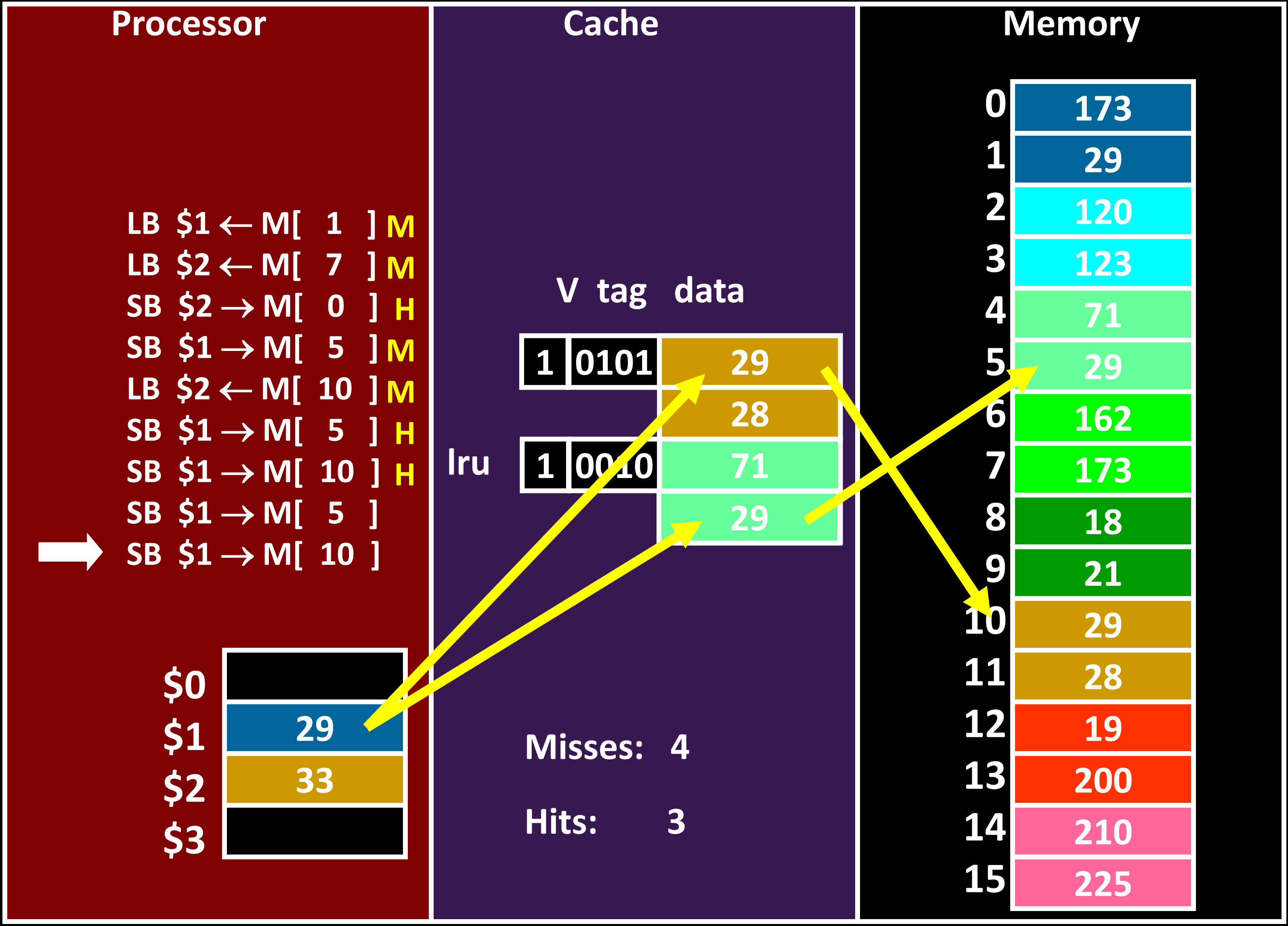
Each store writes an item to mem

- 4 mem writes

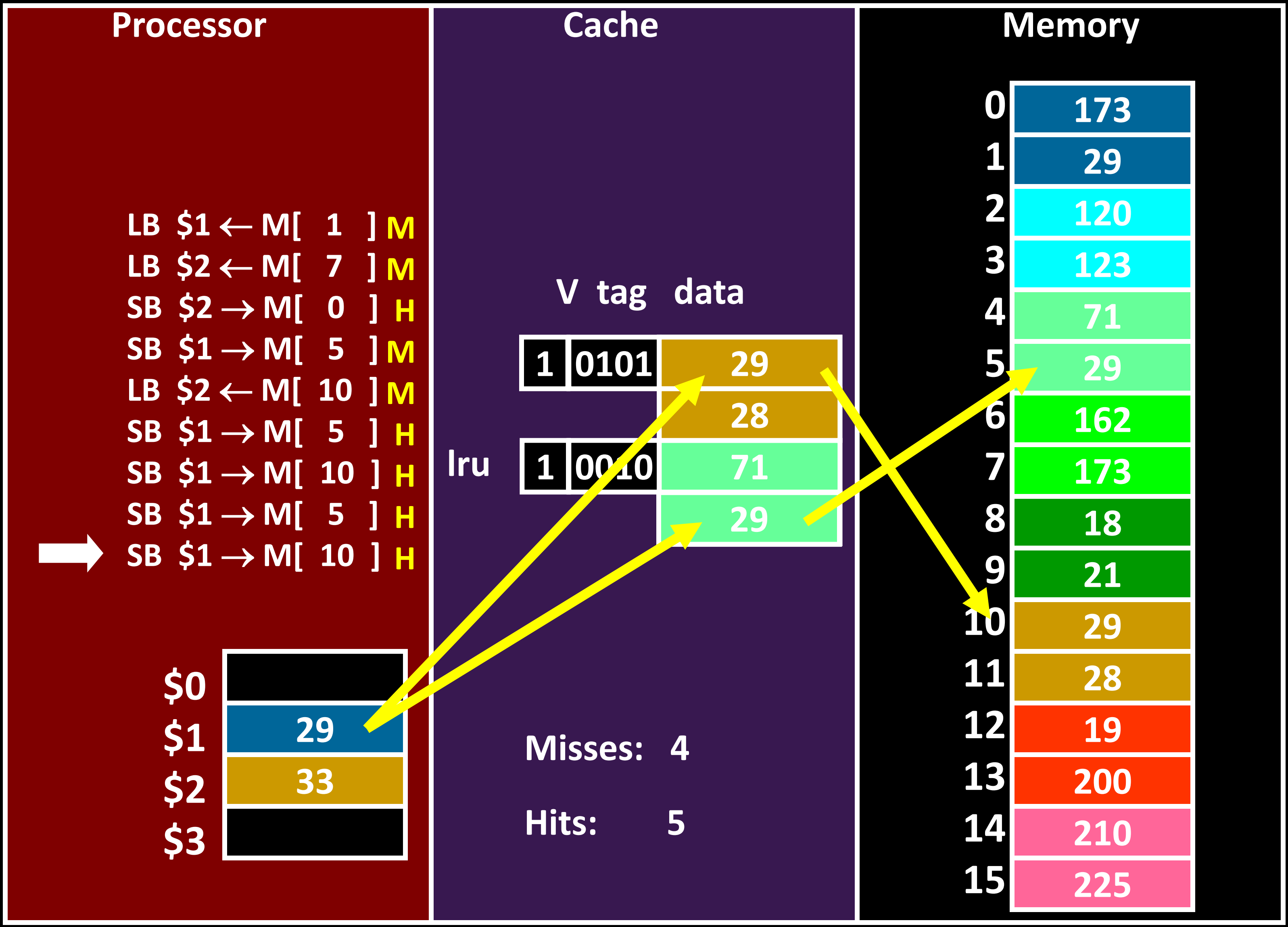
Evictions don’t need to write to mem

- no need for dirty bit

Write-Through (REF 8,9)



Write-Through (REF 8,9)



Takeway

Direct Mapped → fast but low hit rate
Fully Associative → higher hit cost, but higher hit rate
N-way Set Associative → middleground

Cacheline size matters. Larger cache lines can increase performance due to prefetching. BUT, can also decrease performance if **working set** size cannot fit in cache.

Ultimately, cache performance is measured by the avg memory access time (AMAT), which depends on app, cache architecture and size, but also the Access time for hit, miss penalty, hit rate.

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.

Administrivia

Prelim1: ***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)**
- Prelim Review: Yesterday, Mon, at 7pm and today, Tue, at 5:00pm.
  Both in 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