17. Cache organization: Direct mapped & Set Associative

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Don Winsor & Todd Austin

EECS Department
University of Michigan in Ann Arbor, USA

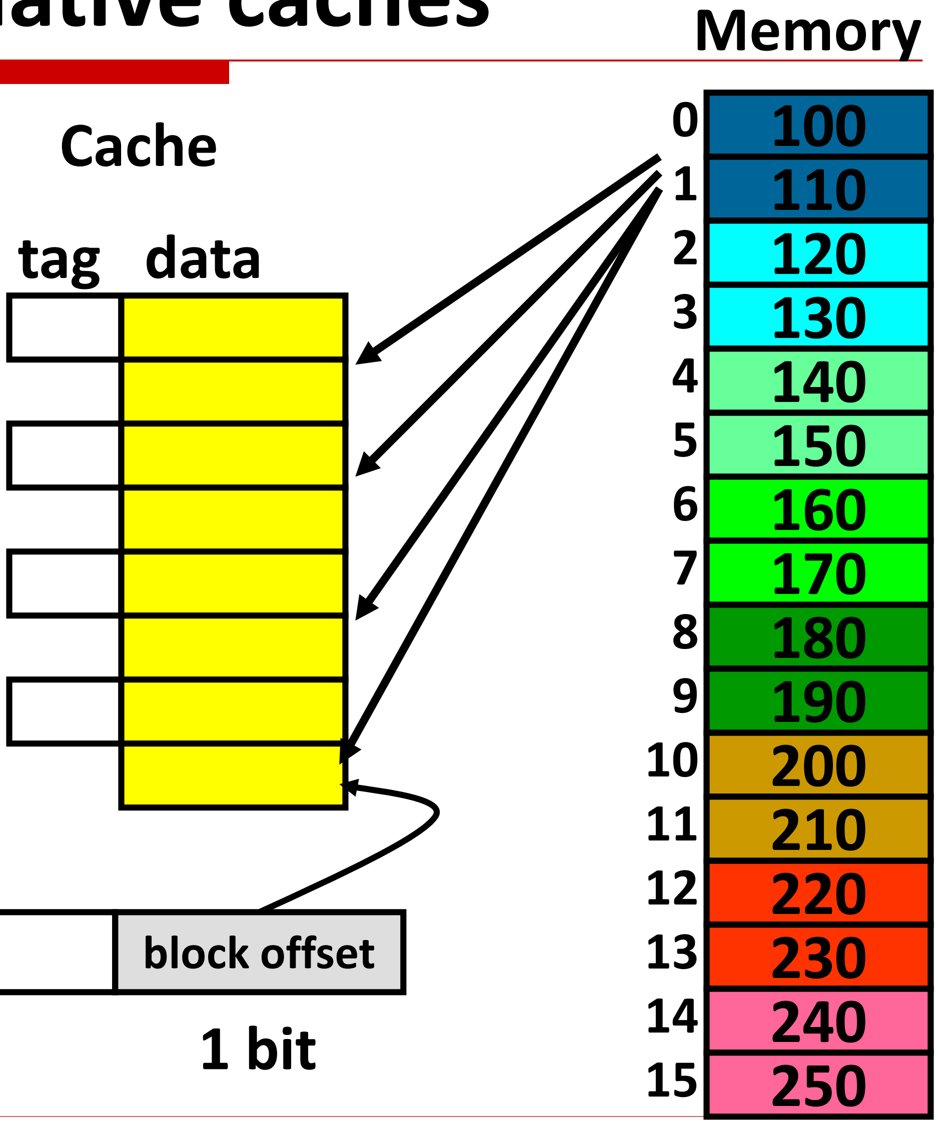
Recap: Fully-associative caches

- ❑ We designed a fully associative cache.
  - Any memory location can be copied to any cache line.
  - We check every cache tag to determine whether the data is in the cache.

- ❑ This approach can be too slow sometimes.
  - Parallel tag searches are expensive and can be slow. Why?

Recap: Fully-associative caches

A block can go to any location



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

Address:

tag	block offset
3 bits	1 bit

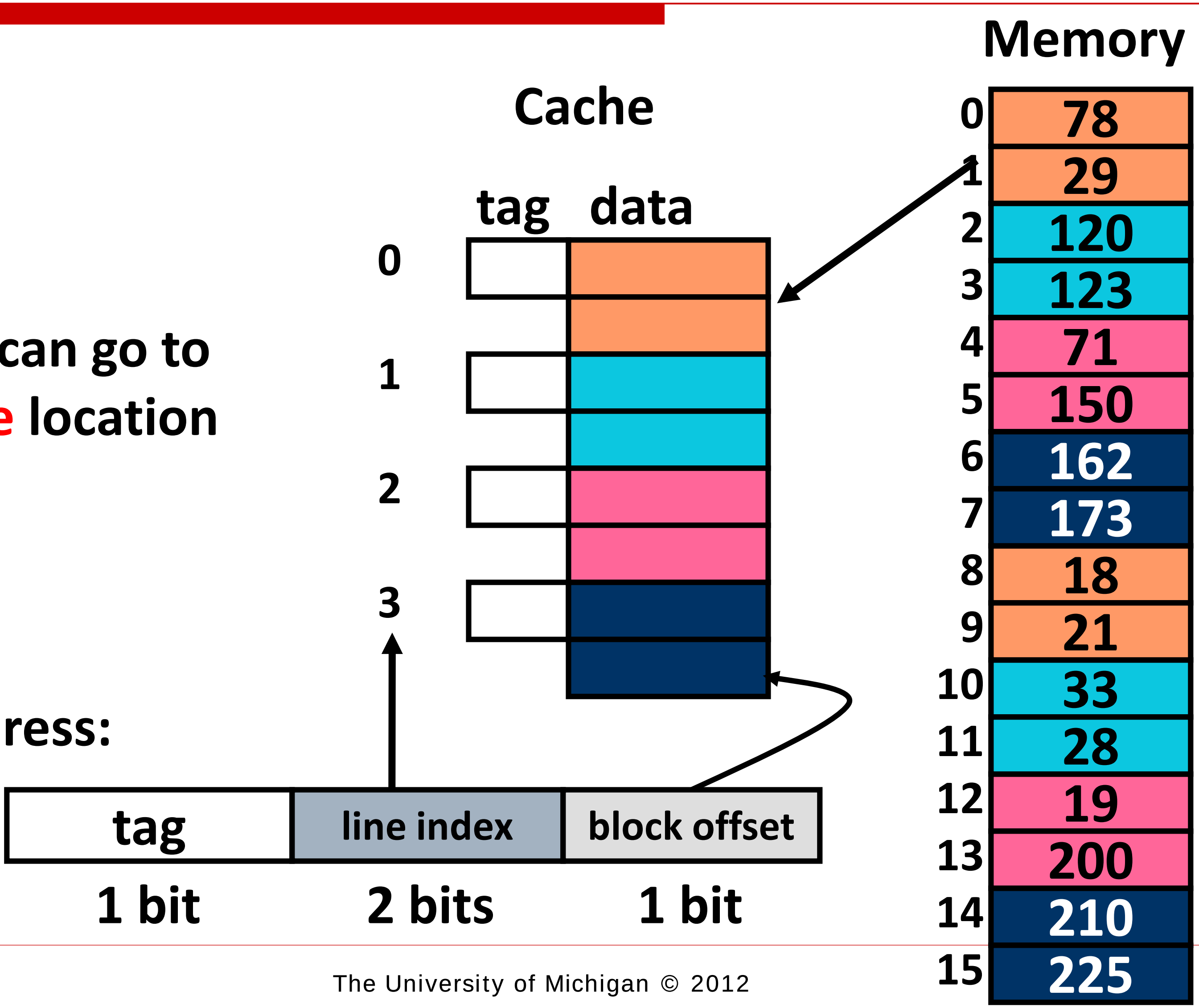
Direct mapped cache

- ❑ We can redesign the cache to eliminate the requirement for parallel tag lookups.
  - Direct mapped caches partition memory into as many regions as there are cache lines.
  - Each memory region maps to a single cache line in which data can be placed.
  - You then only need to check a single tag – the one associated with the region the reference is located in.

Mapping memory to cache

A block can go to only one location

Address:



	Memory
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

Direct mapped cache

- ❑ Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time

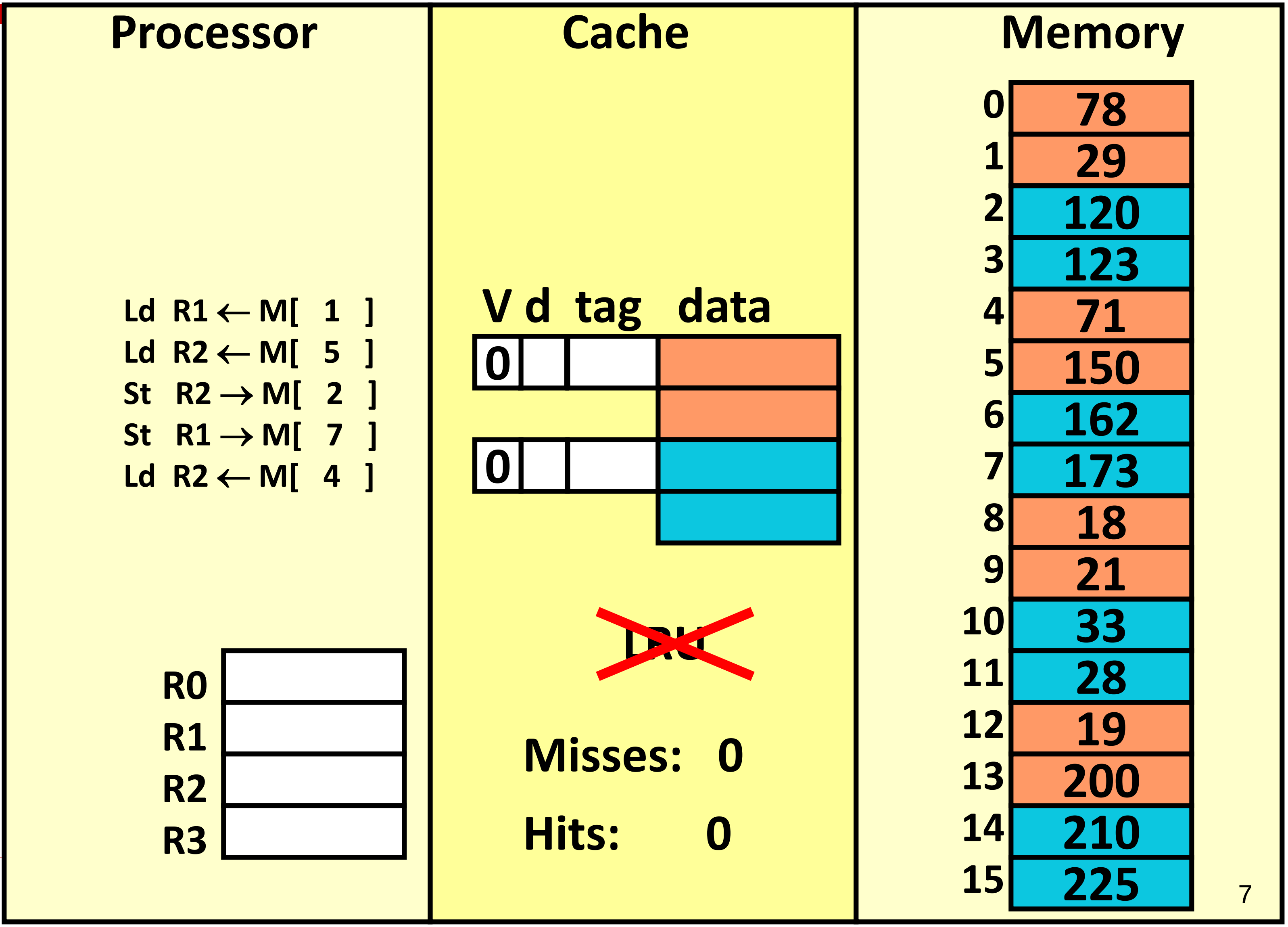
  One index → one entry

- ❑ Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits

    A, B, A, B, A, B, A, B, …

    All accesses are conflict misses

Direct-mapped cache



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[2]
St R1 → M[7]
Ld R2 ← M[4]

Cache

~~LRU~~

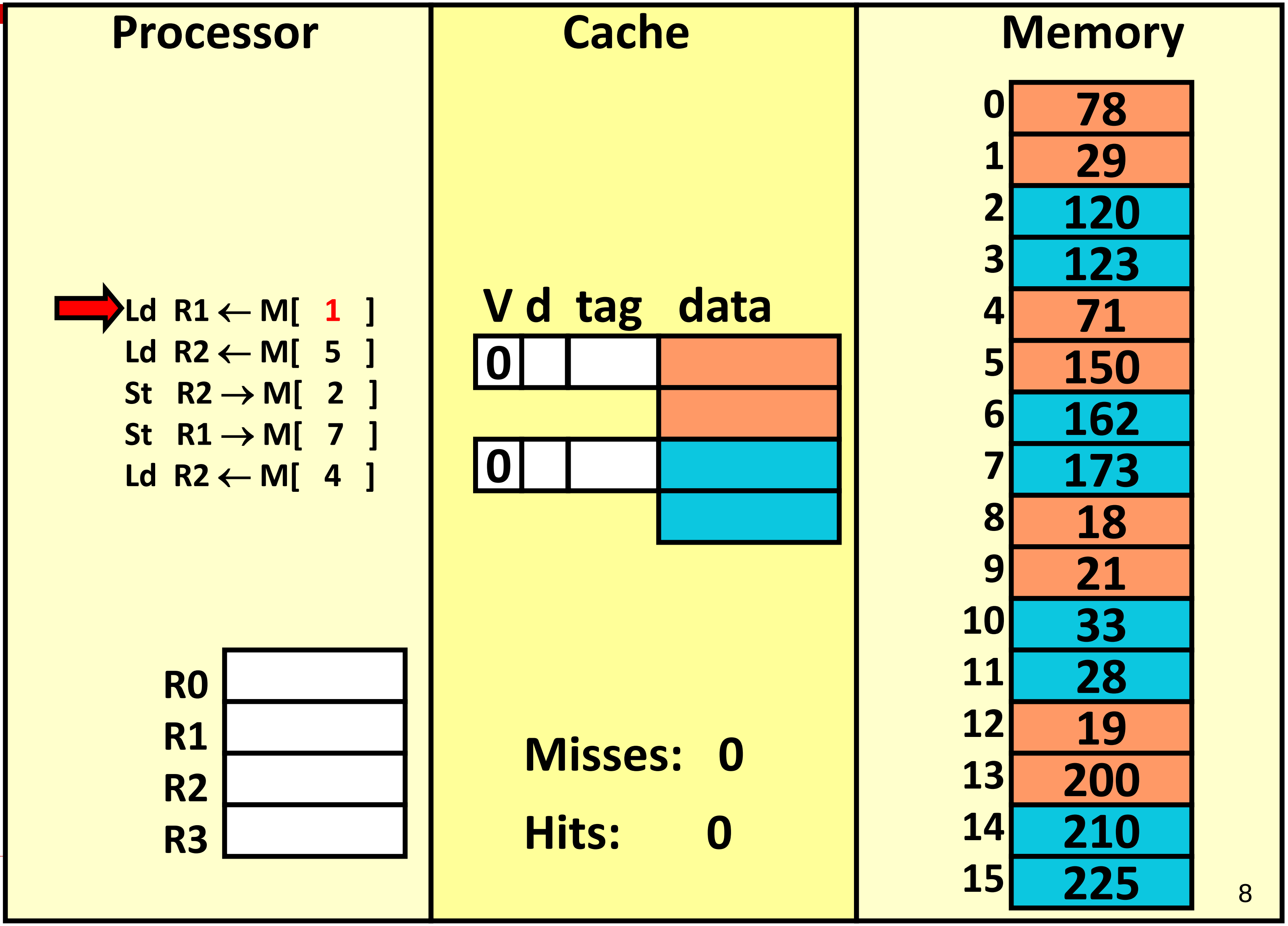
Misses: 0

Hits: 0

Memory

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

Direct-mapped (REF 1)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[2]
St R1 → M[7]
Ld R2 ← M[4]

R0	
R1	
R2	
R3	

Cache

V	d	tag	data
0			
0			

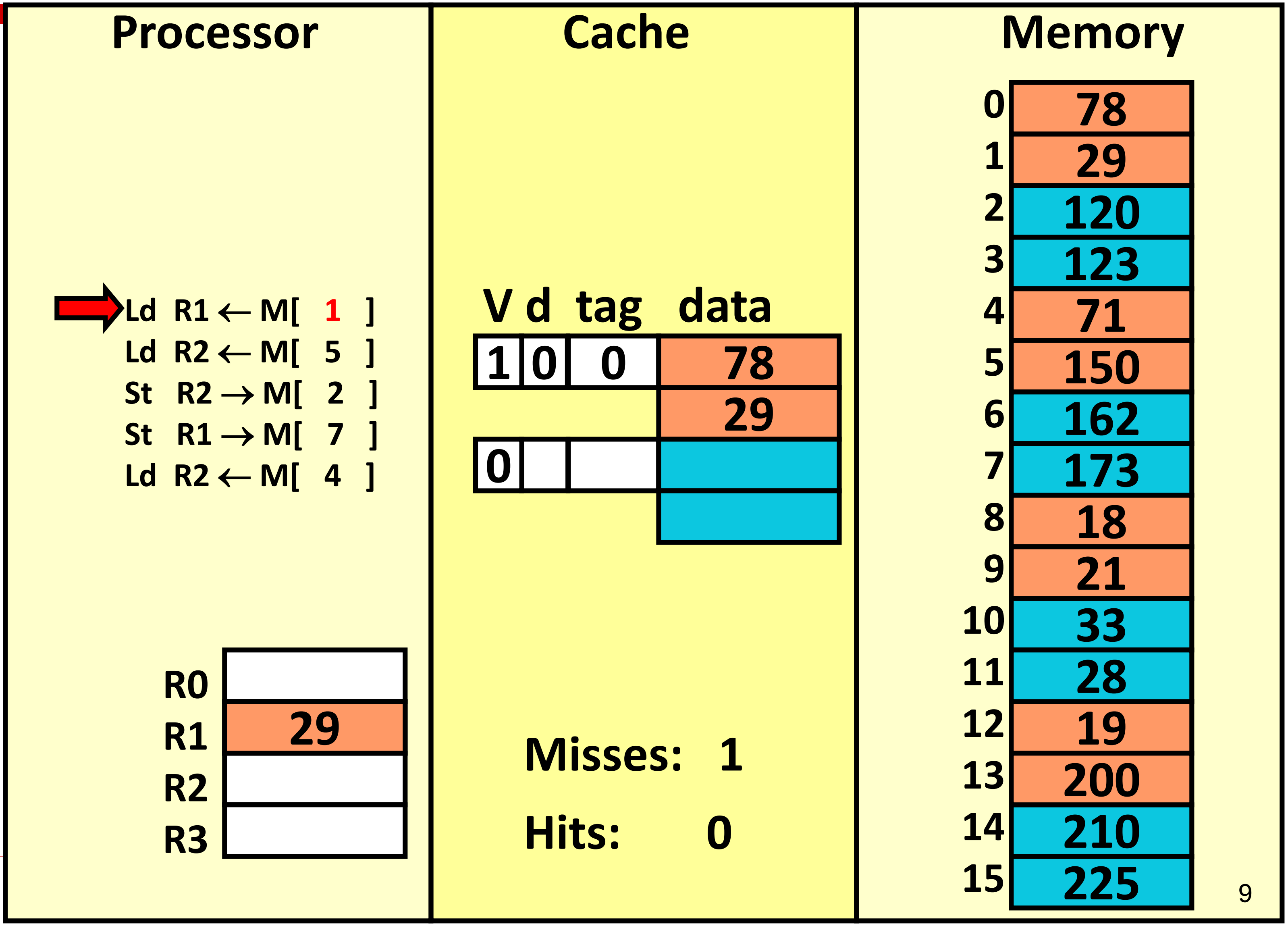
Misses: 0

Hits: 0

Memory

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

Direct-mapped (REF 1)



Processor

```
Ld  R1 ← M[ 1 ]
Ld  R2 ← M[ 5 ]
St  R2 → M[ 2 ]
St  R1 → M[ 7 ]
Ld  R2 ← M[ 4 ]
```

R0	
R1	29
R2	
R3	

Cache

V	d	tag	data
1	0	0	78
			29
0			

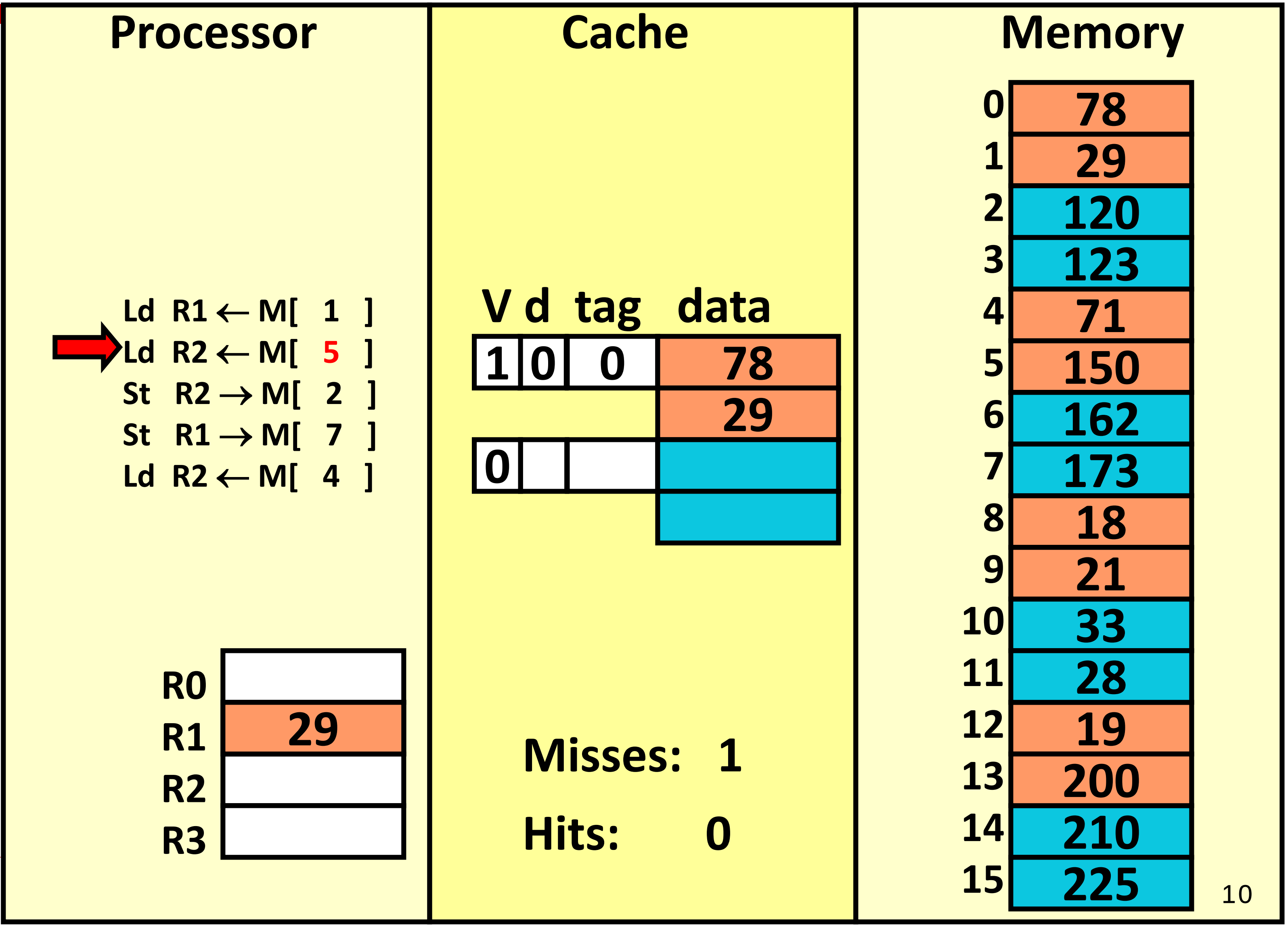
Misses: 1

Hits: 0

Memory

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

Direct-mapped (REF 2)



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  2  ]
St  R1 → M[  7  ]
Ld  R2 ← M[  4  ]
```

R0	
R1	29
R2	
R3	

Cache

V	d	tag	data
1	0	0	78
			29
0			

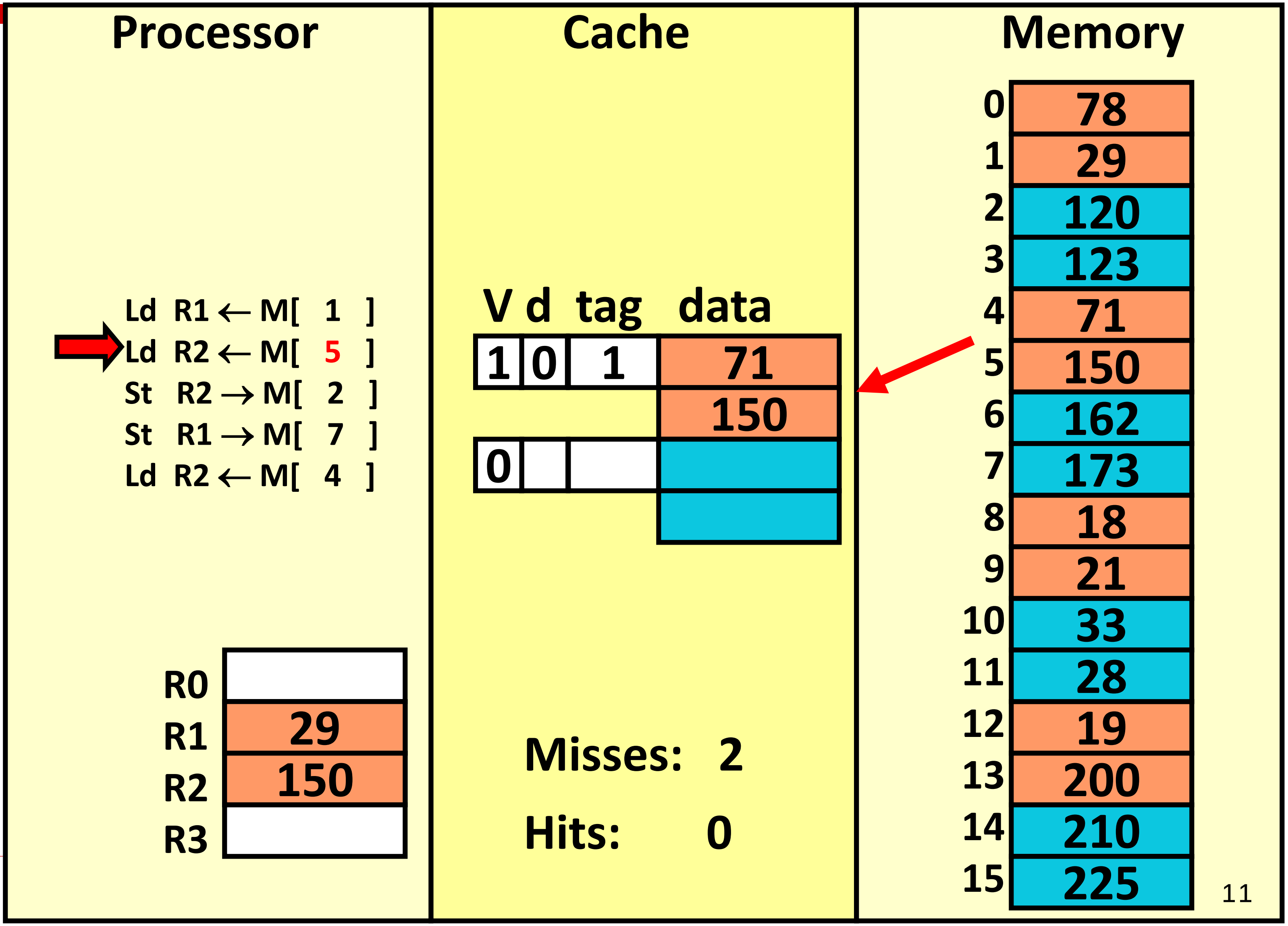
Misses: 1

Hits: 0

Memory

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

Direct-mapped (REF 2)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[2]
St R1 → M[7]
Ld R2 ← M[4]

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
0			

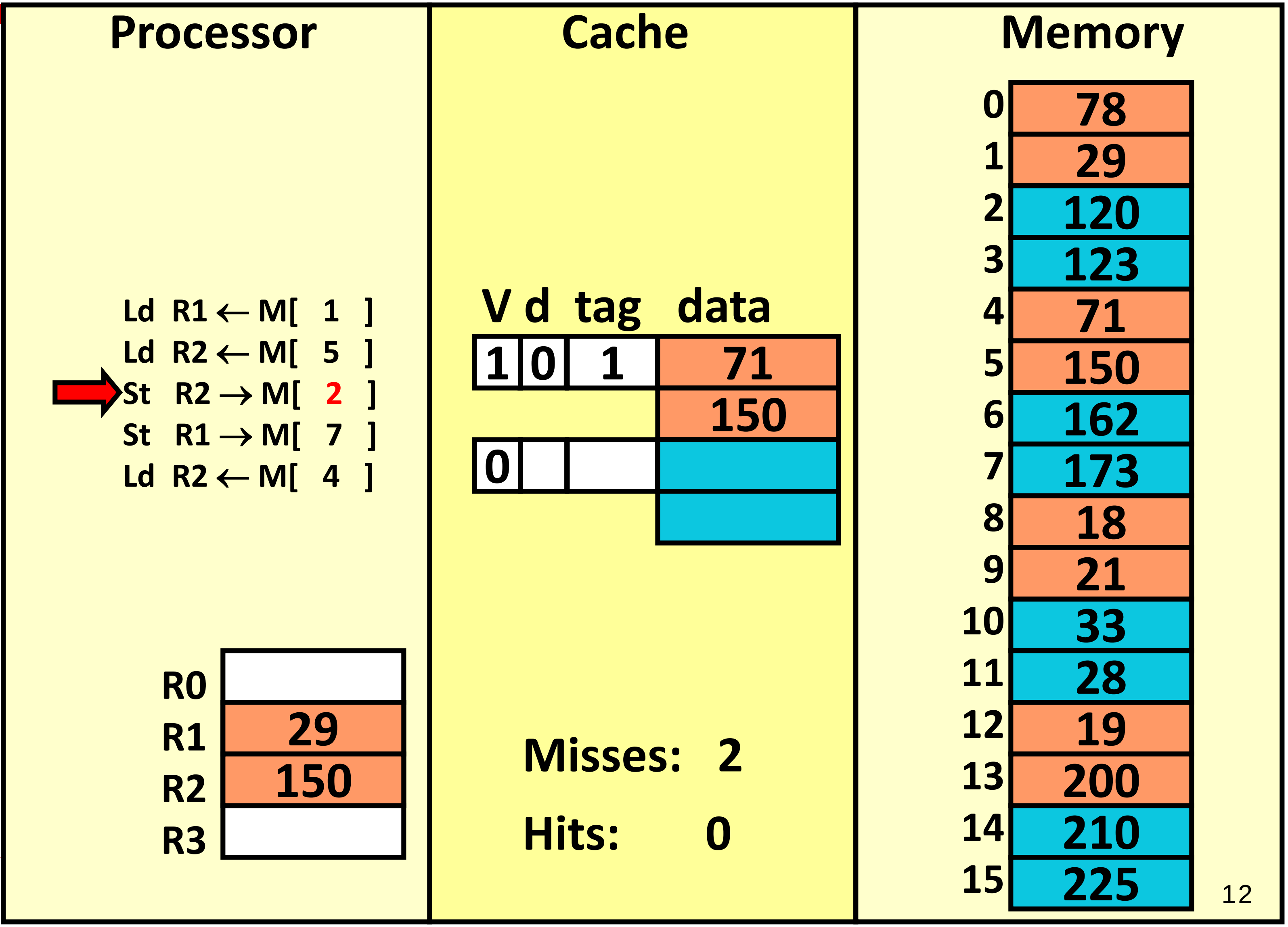
Misses: 2

Hits: 0

Memory

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

Direct-mapped (REF 3)



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  2  ]
St  R1 → M[  7  ]
Ld  R2 ← M[  4  ]
```

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
0			

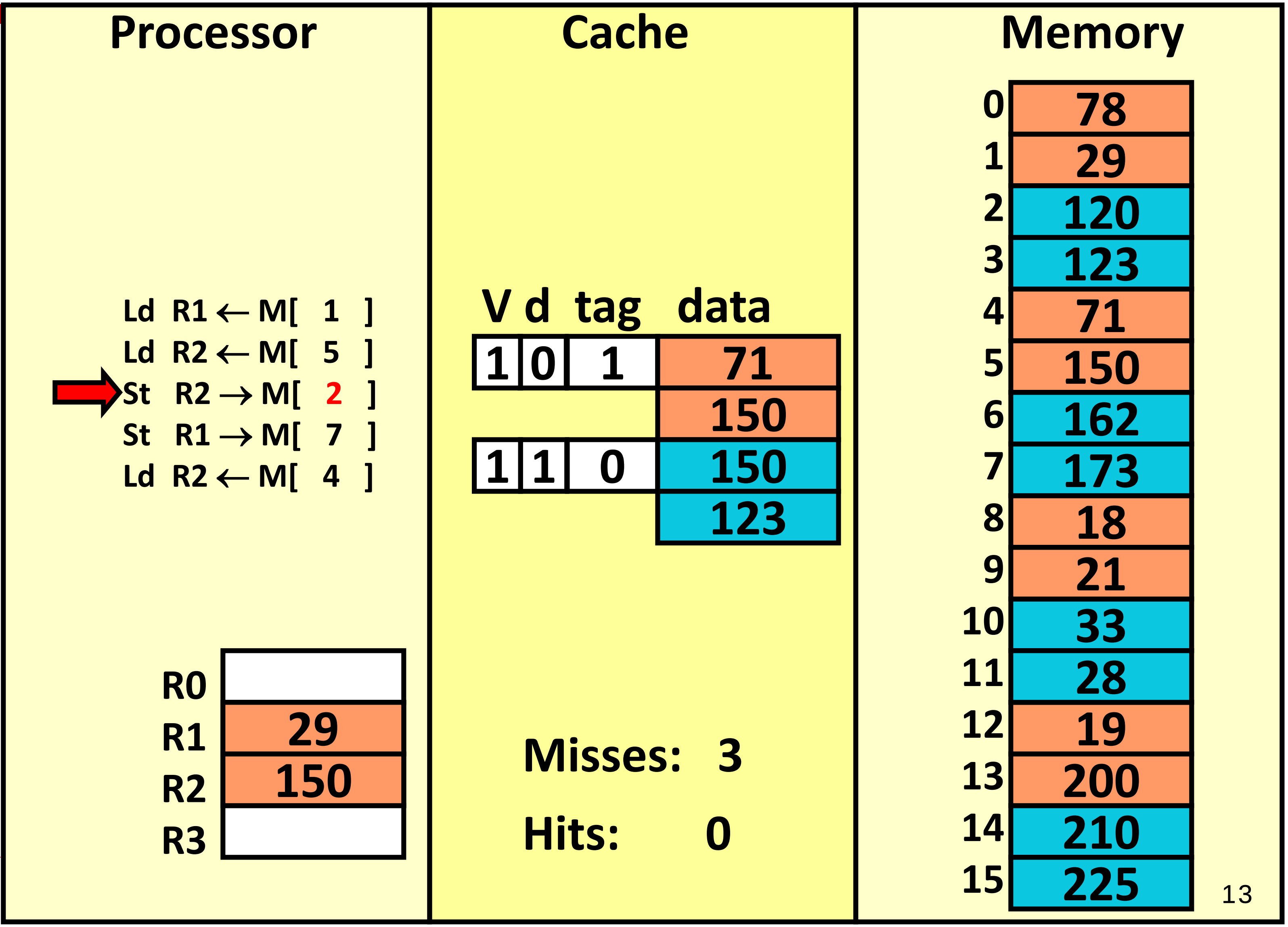
Misses: 2

Hits: 0

Memory

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

Direct-mapped (REF 3)



Processor

```
Ld  R1 ← M[ 1 ]
Ld  R2 ← M[ 5 ]
St  R2 → M[ 2 ]
St  R1 → M[ 7 ]
Ld  R2 ← M[ 4 ]
```

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
1	1	0	150
			123

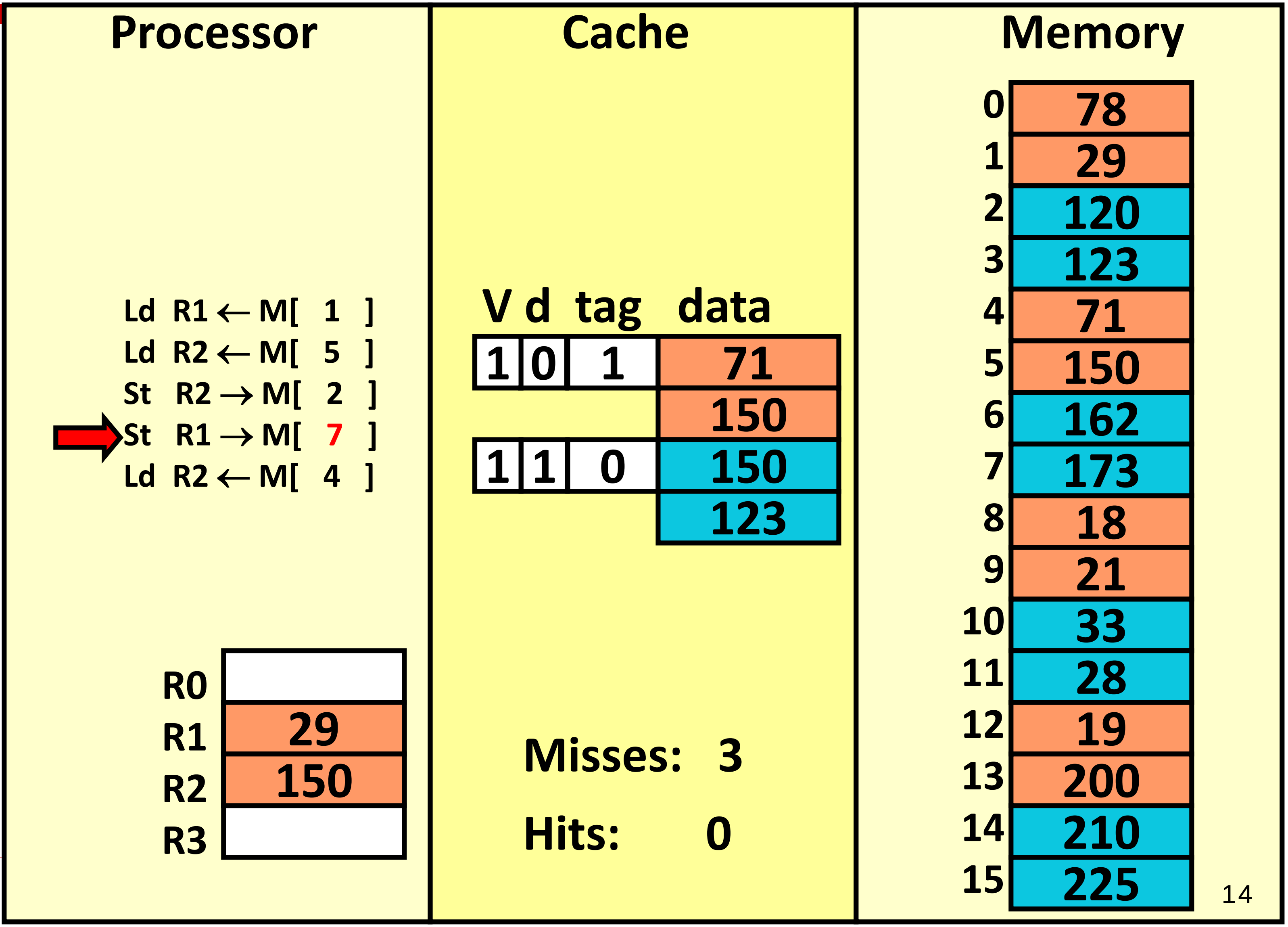
Misses: 3

Hits: 0

Memory

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

Direct-mapped (REF 4)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[2]
➡ St R1 → M[7]
Ld R2 ← M[4]

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
1	1	0	150
			123

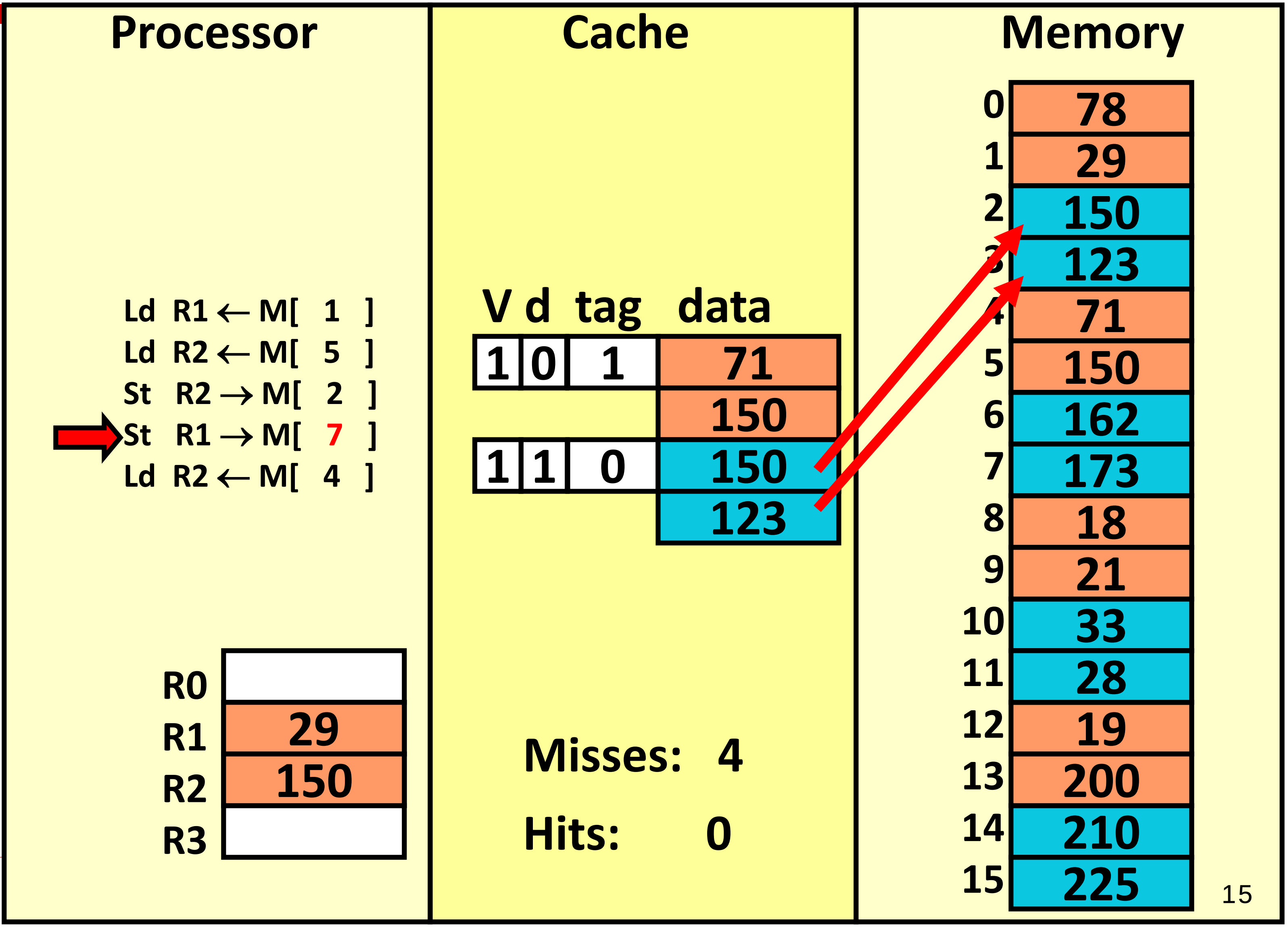
Misses: 3

Hits: 0

Memory

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

Direct-mapped (REF 4)



Processor

```
  Ld  R1 ← M[ 1 ]
  Ld  R2 ← M[ 5 ]
  St  R2 → M[ 2 ]
➡ St  R1 → M[ 7 ]
  Ld  R2 ← M[ 4 ]
```

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
1	1	0	150
			123

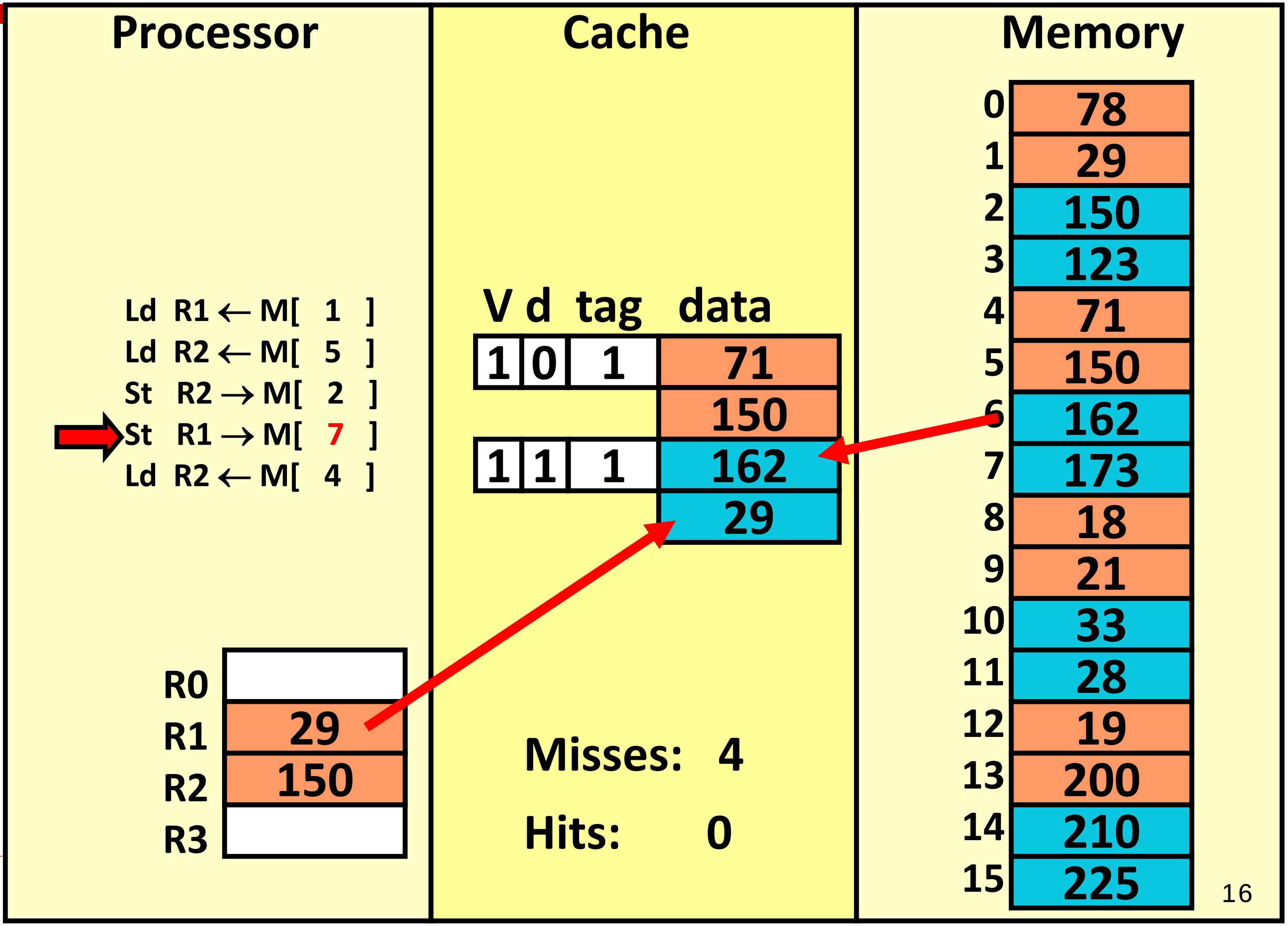
Misses: 4

Hits: 0

Memory

0	78
1	29
2	150
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

Direct-mapped (REF 4)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[2]
St R1 → M[7]
Ld R2 ← M[4]

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
1	1	1	162
			29

Misses: 4

Hits: 0

Memory

0	78
1	29
2	150
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

Direct-mapped (REF 5)

Processor

```
  Ld  R1 ← M[ 1 ]
  Ld  R2 ← M[ 5 ]
  St  R2 → M[ 2 ]
  St  R1 → M[ 7 ]
➡ Ld  R2 ← M[ 4 ]
```

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	1	71
			150
1	1	1	162
			29

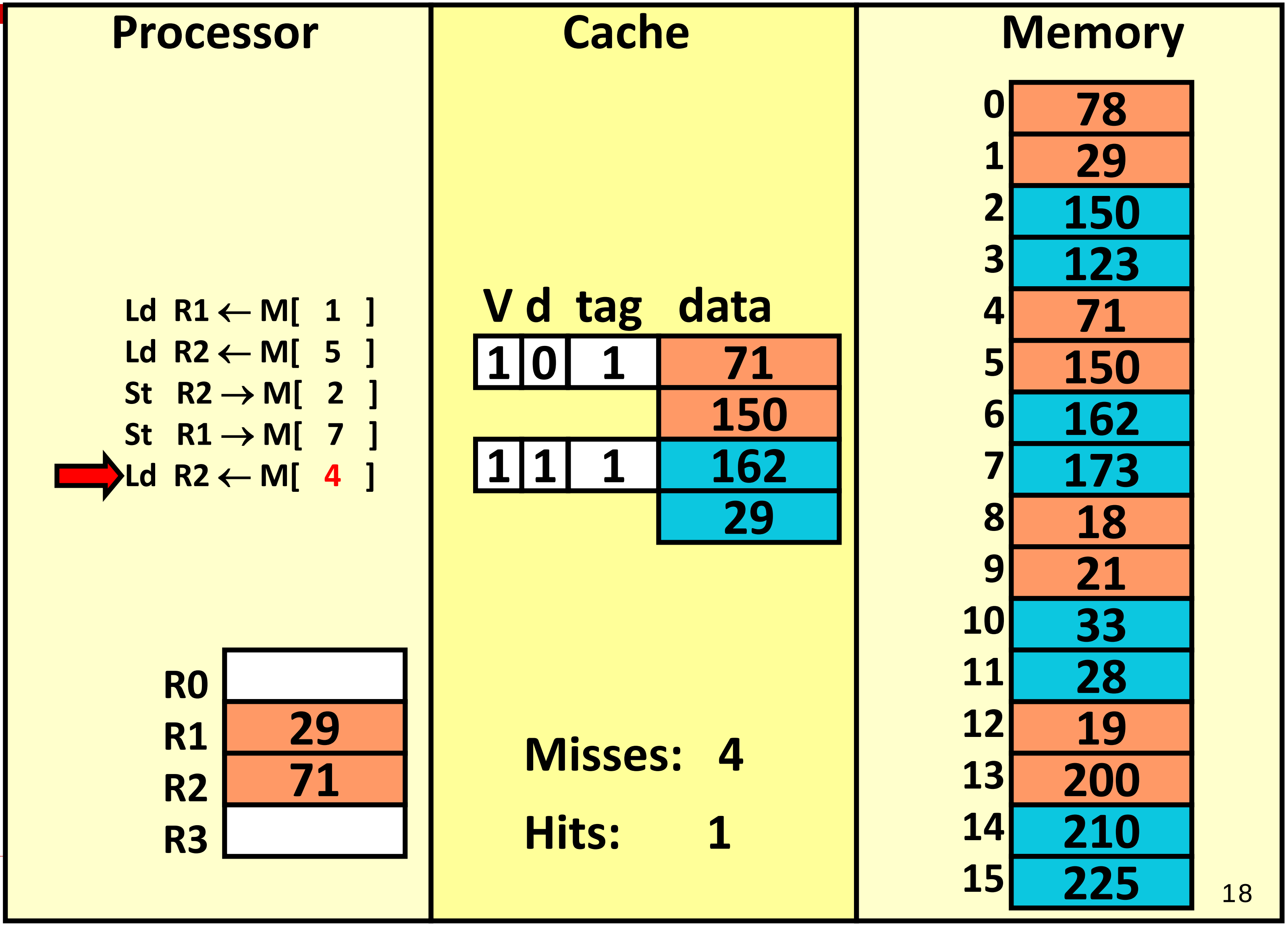
Misses: 4

Hits: 0

Memory

0	78
1	29
2	150
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

Direct-mapped (REF 5)



Processor

```
Ld  R1 ← M[ 1 ]
Ld  R2 ← M[ 5 ]
St  R2 → M[ 2 ]
St  R1 → M[ 7 ]
➡ Ld  R2 ← M[ 4 ]
```

R0	
R1	29
R2	71
R3	

Cache

V	d	tag	data
1	0	1	71
			150
1	1	1	162
			29

Misses: 4

Hits: 1

Memory

0	78
1	29
2	150
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

Class Problem 1

- How many tag bits are required for:
  - 32-bit address, byte addressed, direct-mapped 32k cache, 128 byte block size, write-back

- What are the overheads for this cache?

Class Problem 1

- How many tag bits are required for:
  - 32-bit address, byte addressed, direct-mapped 32k cache, 128 byte block size, write-back

    # Bytes in block = 128 ➜ Block offset = 7 bits (*byte addressable*)

    # Lines = 32k / 128 = 256 ➜ Line index = 8 bits

    Tag bits = 32 – 7 – 8 = 17 bits

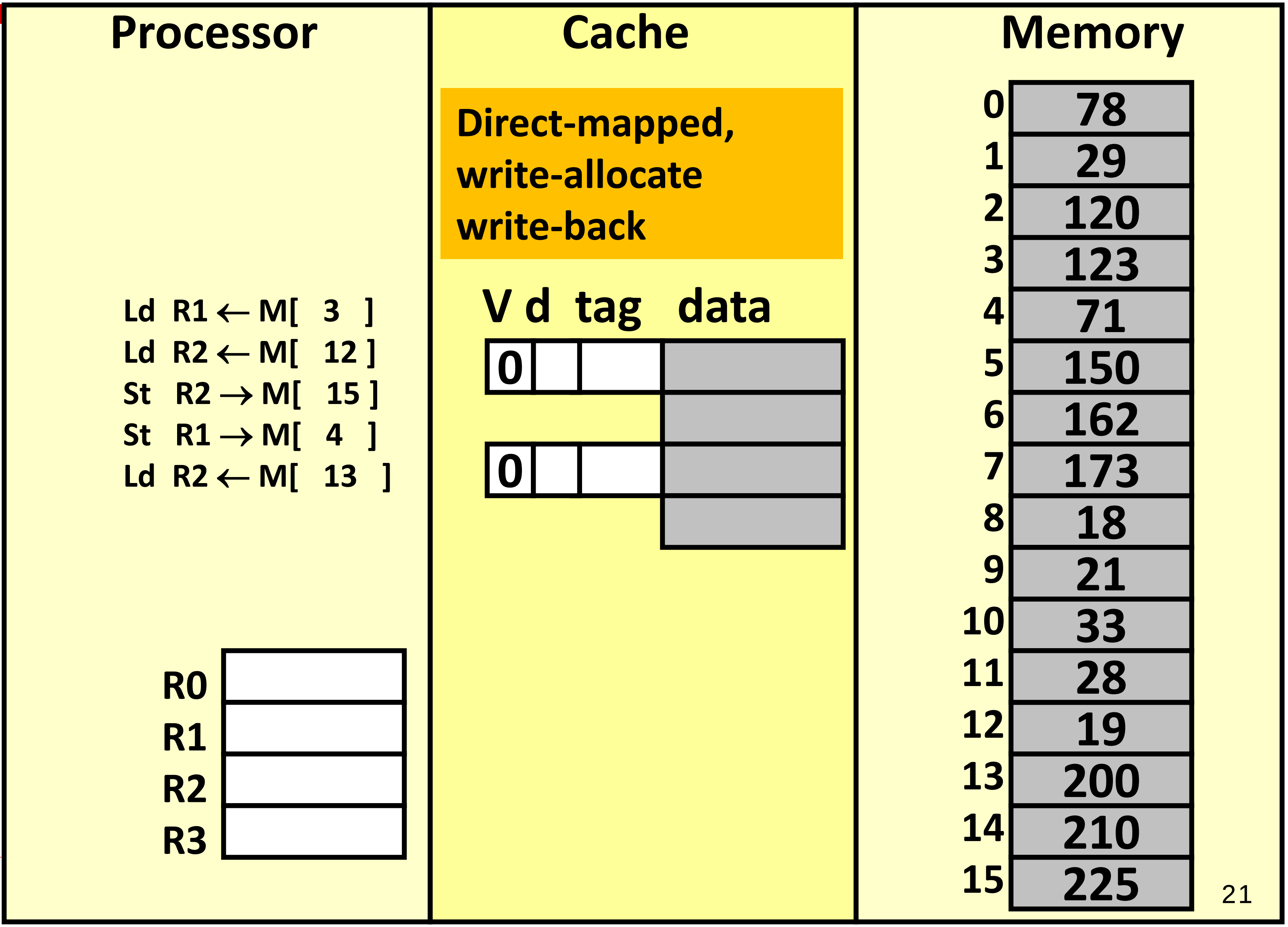
- What are the overheads for this cache?

  17 bits (Tag) + 1 bit (Valid) + 1 bit (Dirty) = 19 bits / line

  19 bits / line * 256 lines = 4864 bits

  4864 bits / 32KB = 1.9% overheads

Class Problem 2 – What is the state of the cache after executing the following instruction sequence?



Processor

```
Ld  R1 ← M[  3  ]
Ld  R2 ← M[  12 ]
St  R2 → M[  15 ]
St  R1 → M[  4  ]
Ld  R2 ← M[  13  ]
```

R0

R1

R2

R3

Cache

Direct-mapped,
write-allocate
write-back

V	d	tag	data
0			
0			

Memory

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

Class Problem 2 – What is the state of the cache after executing the following instruction sequence?

Processor

Ld R1 ← M[3]
Ld R2 ← M[12]
St R2 → M[15]
St R1 → M[4]
Ld R2 ← M[13]

R0	
R1	123
R2	19
R3	

Cache

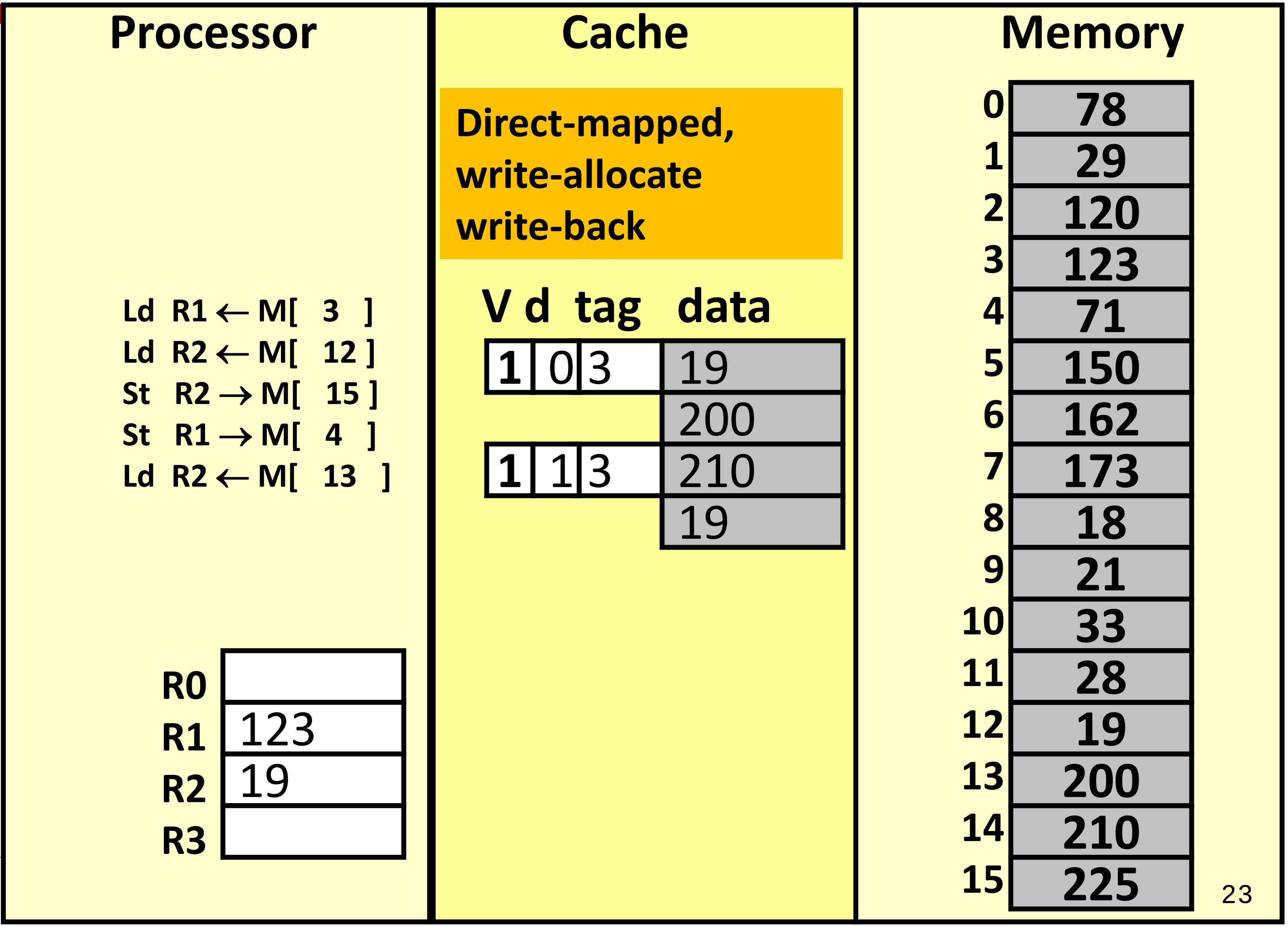
Direct-mapped,
write-allocate
write-back

V	d	tag	data
1	0	3	19
			200
1	0	0	120
			123

Memory

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

Class Problem 2 – What is the state of the cache after executing the following instruction sequence?



Processor

```
Ld  R1 ← M[  3  ]
Ld  R2 ← M[  12 ]
St  R2 → M[  15 ]
St  R1 → M[  4  ]
Ld  R2 ← M[  13  ]
```

R0	
R1	123
R2	19
R3	

Cache

Direct-mapped,
write-allocate
write-back

V	d	tag	data
1	0	3	19
			200
1	1	3	210
			19

Memory

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

Class Problem 2 – What is the state of the cache after executing the following instruction sequence?

Processor

```
Ld  R1 ← M[  3  ]
Ld  R2 ← M[  12 ]
St  R2 → M[  15 ]
St  R1 → M[  4  ]
Ld  R2 ← M[  13  ]
```

R0	
R1	123
R2	19
R3	

Cache

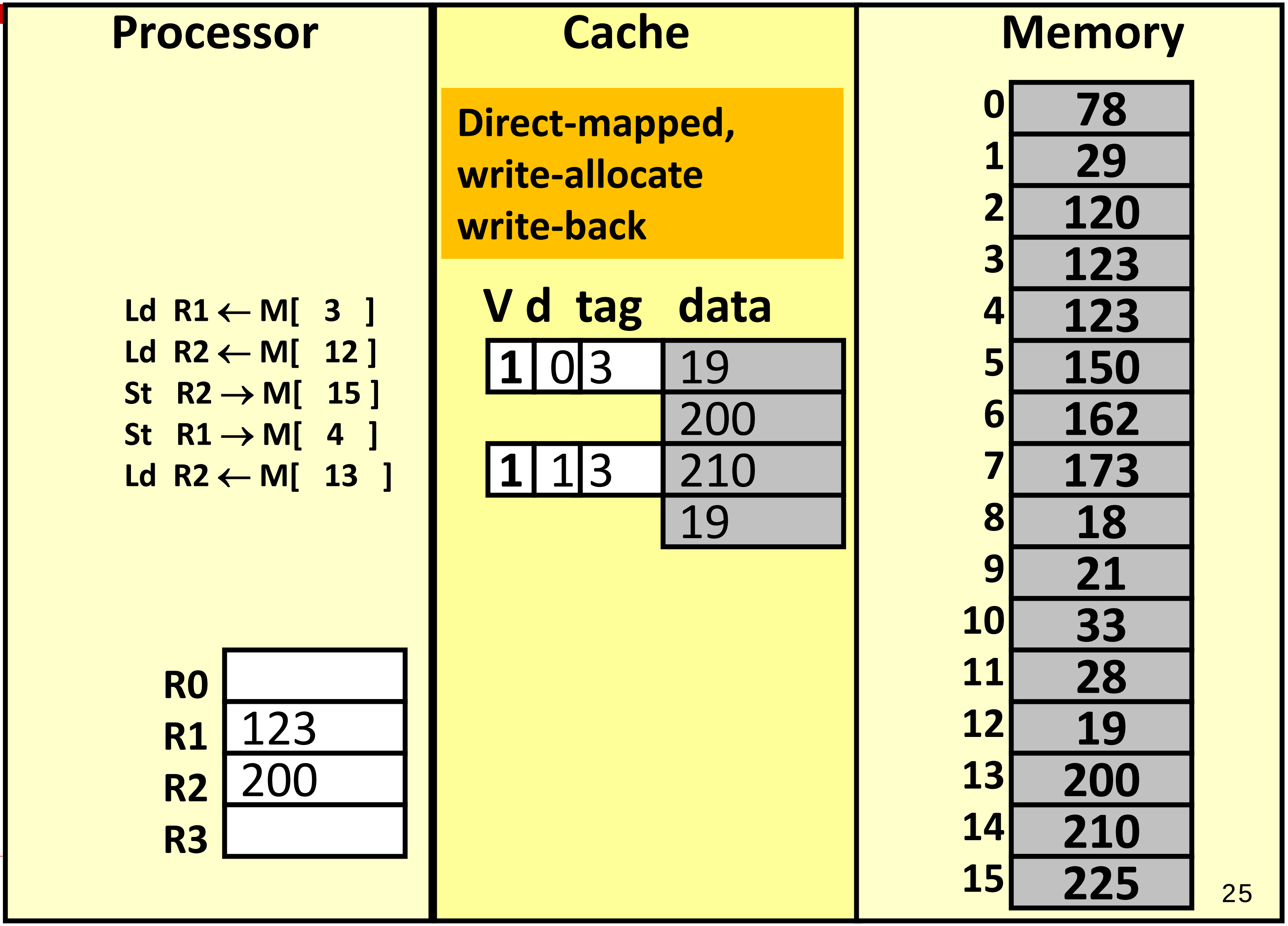
Direct-mapped,
write-allocate
write-back

V	d	tag	data
1	1	1	123
			150
1	1	3	210
			19

Memory

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

Class Problem 2 – What is the state of the cache after executing the following instruction sequence?



Processor

Ld R1 ← M[3]
Ld R2 ← M[12]
St R2 → M[15]
St R1 → M[4]
Ld R2 ← M[13]

R0	
R1	123
R2	200
R3	

Cache

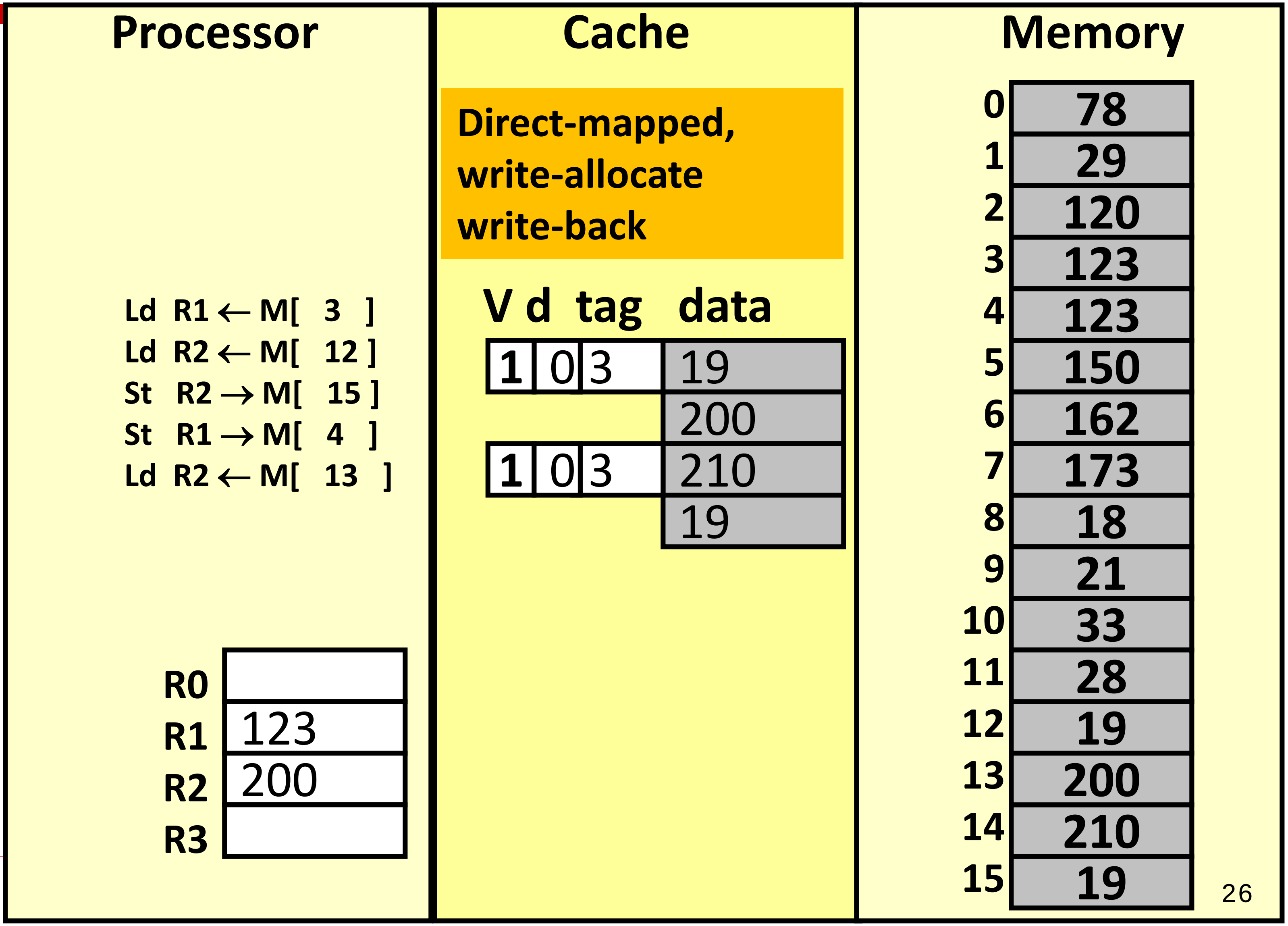
Direct-mapped,
write-allocate
write-back

V	d	tag	data
1	0	3	19
			200
1	1	3	210
			19

Memory

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

Class Problem 2 – What is the state of the cache after executing the following instruction sequence?



Processor

```
Ld  R1 ← M[  3  ]
Ld  R2 ← M[  12 ]
St  R2 → M[  15 ]
St  R1 → M[  4  ]
Ld  R2 ← M[  13  ]
```

R0	
R1	123
R2	200
R3	

Cache

Direct-mapped,
write-allocate
write-back

V	d	tag	data
1	0	3	19
			200
1	0	3	210
			19

Memory

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

What about Cache for Instructions?

- ❑ Instructions should be cached as well.
- ❑ We have two choices:

  1. Treat instruction fetches as normal data and allocate cache lines when fetched.
  2. Create a second cache (called the instruction cache or ICache) which caches instructions only.

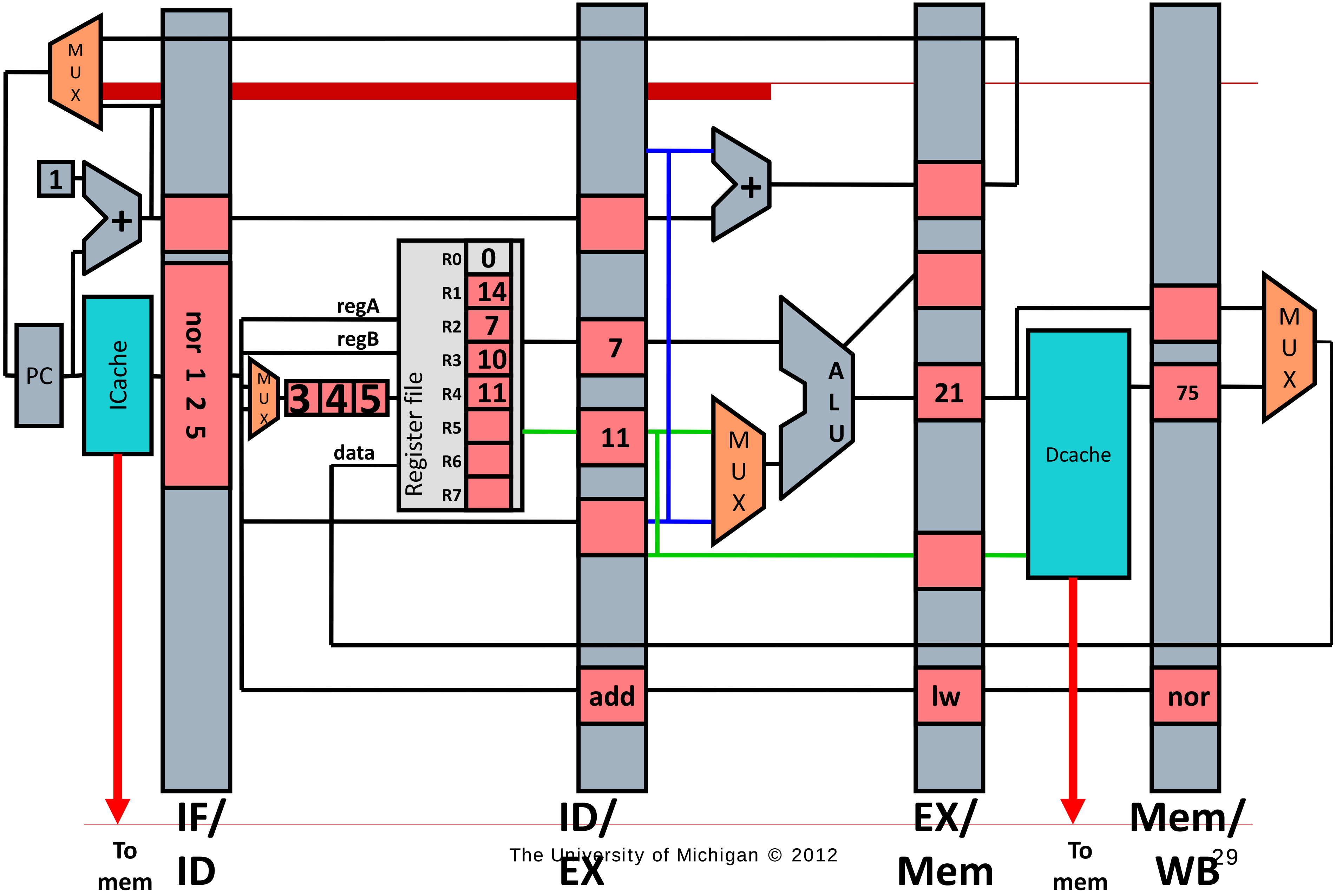
     How do you know which cache to use?

     What are advantages of a separate ICache?

Integrating Caches into a Pipeline

- How are caches integrated into a pipelined implementation?
  - Replace instruction memory with Icache
  - Replace data memory with Dcache
- Issues
  - Memory accesses now have variable latency
  - Both caches may miss at the same time

LC2K Pipeline with Caches



Class Problem 3

The *grinder* application run on the LC2k with full data forwarding and all branches predicted not-taken has the following instruction frequencies:

45% R-type   20% Branches   15% Loads   20% Stores

In *grinder*, 40% of branches are taken and 50% of LWs are followed by an immediate use.

The I-cache has a miss rate of 3% and the D-cache has a miss rate of 6% (no overlapping of misses). On a miss, the main memory is accessed and has a latency of 100 ns. The clock frequency is 500 MHz.

What is the CPI of *grinder* on the LC2k?

Class Problem 3

The *grinder* application run on the LC2k with full data forwarding and all branches predicted not-taken has the following instruction frequencies:

45% R-type 20% Branches 15% Loads 20% Stores

In *grinder*, 40% of branches are taken and 50% of LWs are followed by an immediate use.

The I-cache has a miss rate of 3% and the D-cache has a miss rate of 6% (no overlapping of misses). On a miss, the main memory is accessed and has a latency of 100 ns. The clock frequency is 500 MHz.

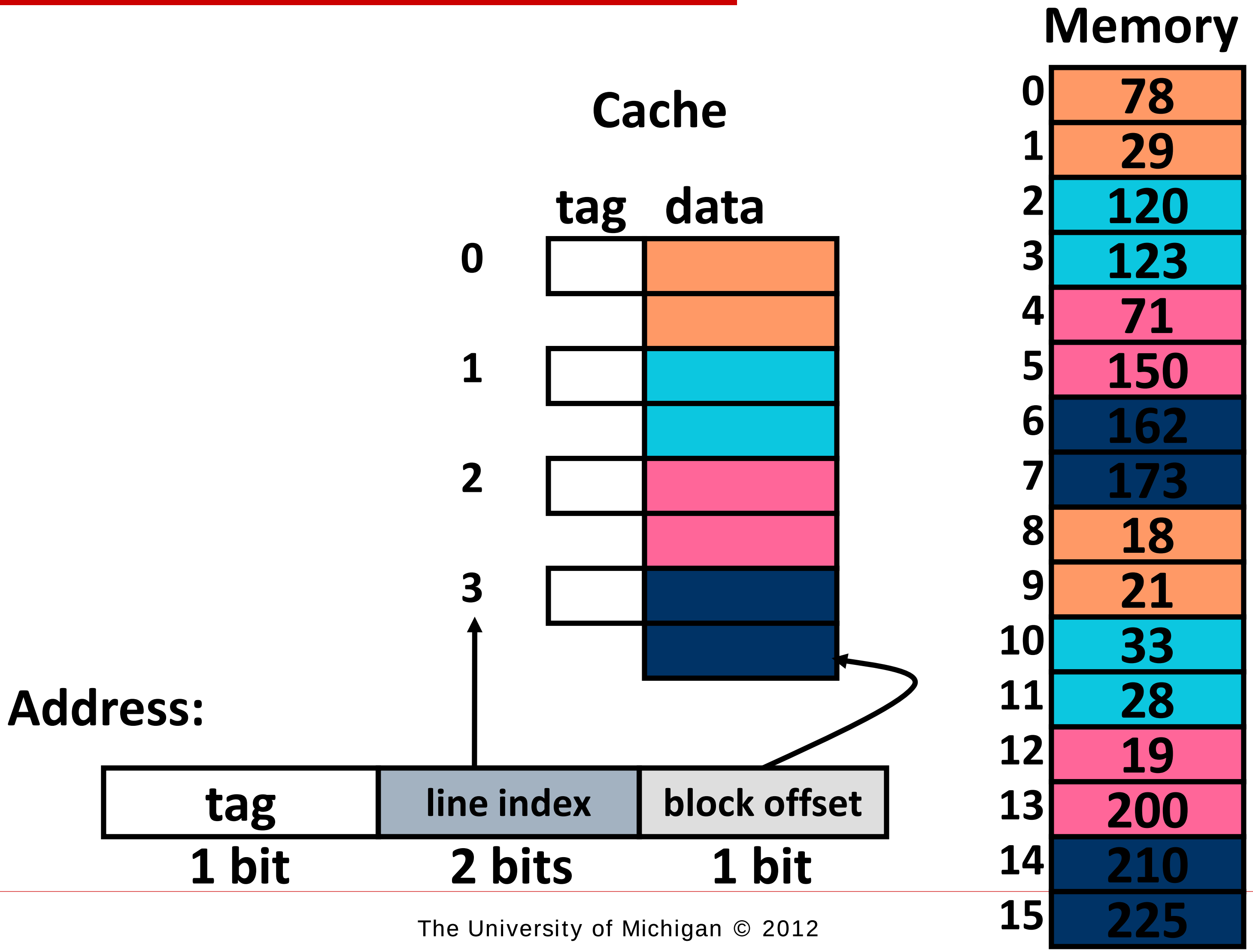
What is the CPI of *grinder* on the LC2k?

Stalls per cache miss = 100 ns / 2ns = 50 cycles (500 Mhz ➜ 2ns cycle time)

CPI = 1 + data hazard stalls + control hazard stalls + icache stalls + dcache stalls

CPI = 1 + 0.15*0.50*1 + 0.20*0.40*3 + 1*0.03*50 + 0.35*0.06*50

Summary: Direct-mapped caches



Cache

	tag	data
0		
1		
2		
3		

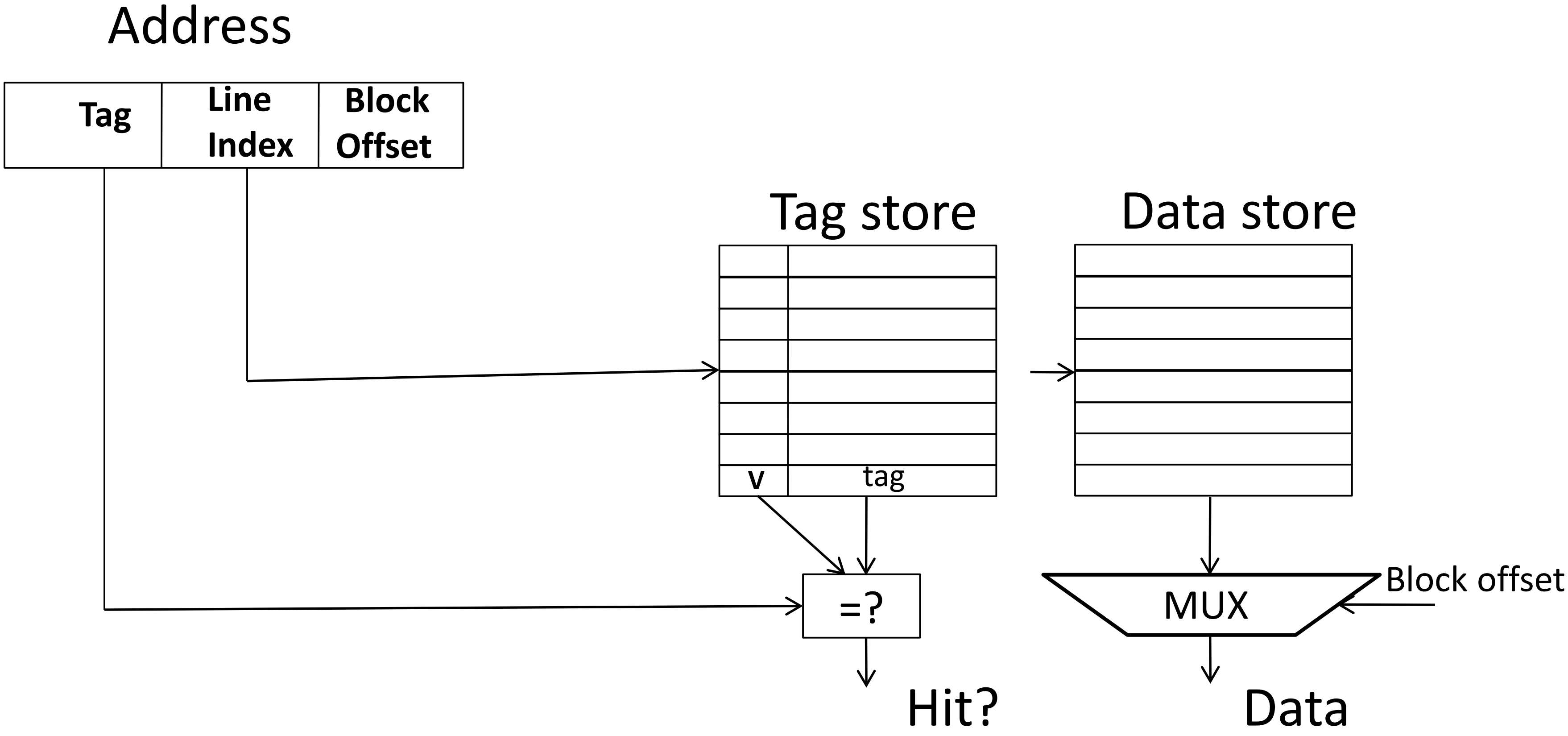
Address:

tag	line index	block offset
1 bit	2 bits	1 bit

Memory

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

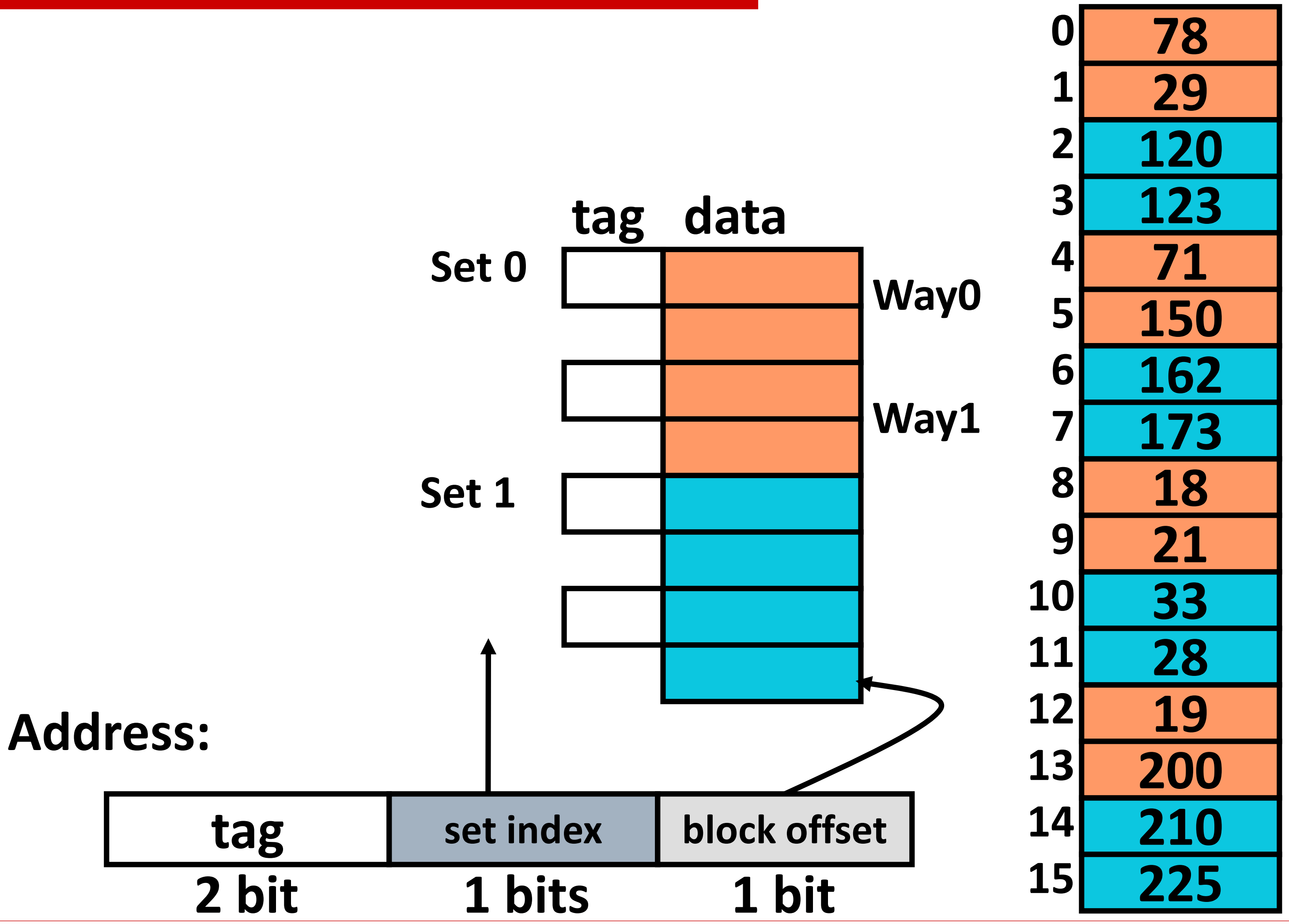
Direct-Mapped Cache: Placement and Access



Gets the advantages of both…

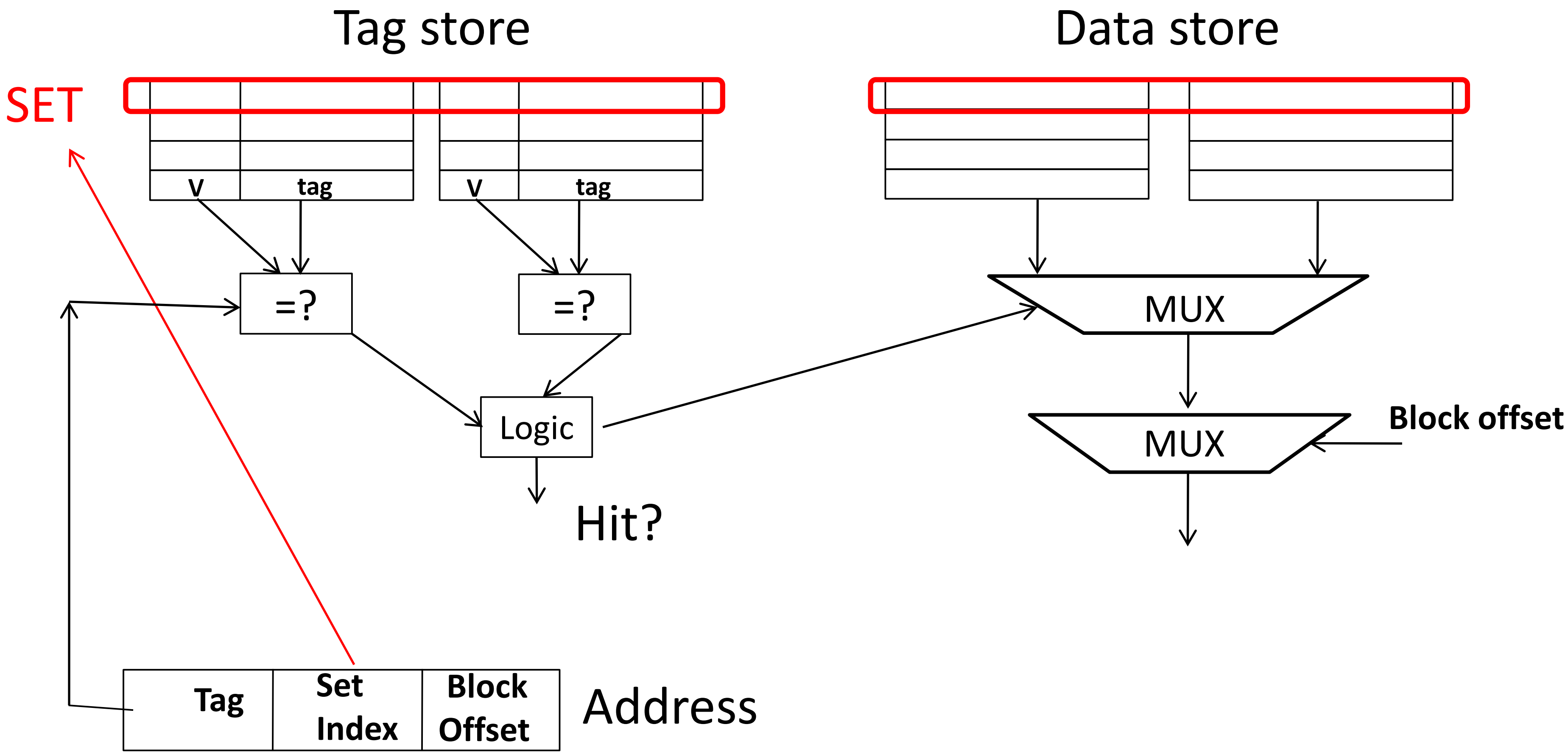
- Set associative caches:
  - Partition memory into regions
    - like direct mapped but fewer partitions
  - Associate a region to a set of cache lines
    - Check tags for all lines in a set to determine a HIT
- Treat each line in a set like a small fully associative cache.
  - LRU (or LRU-like) policy generally used.

Set-associative cache

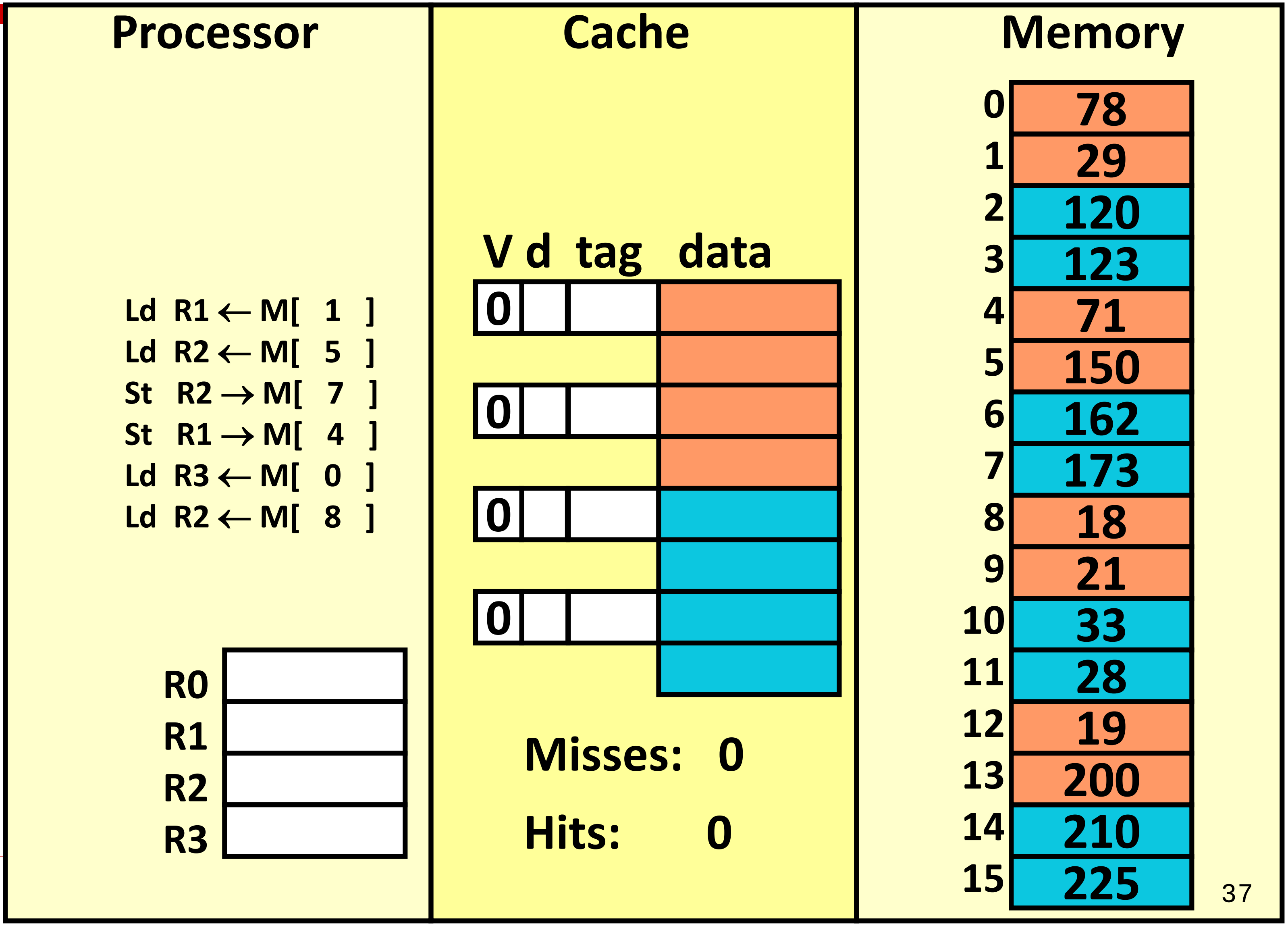


0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Set-Associative Cache: Placement and Access



Set-associative cache example



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  7  ]
St  R1 → M[  4  ]
Ld  R3 ← M[  0  ]
Ld  R2 ← M[  8  ]
```

R0	
R1	
R2	
R3	

Cache

V	d	tag	data
0			🟧
			🟧
0			🟧
			🟧
0			🟦
			🟦
0			🟦
			🟦

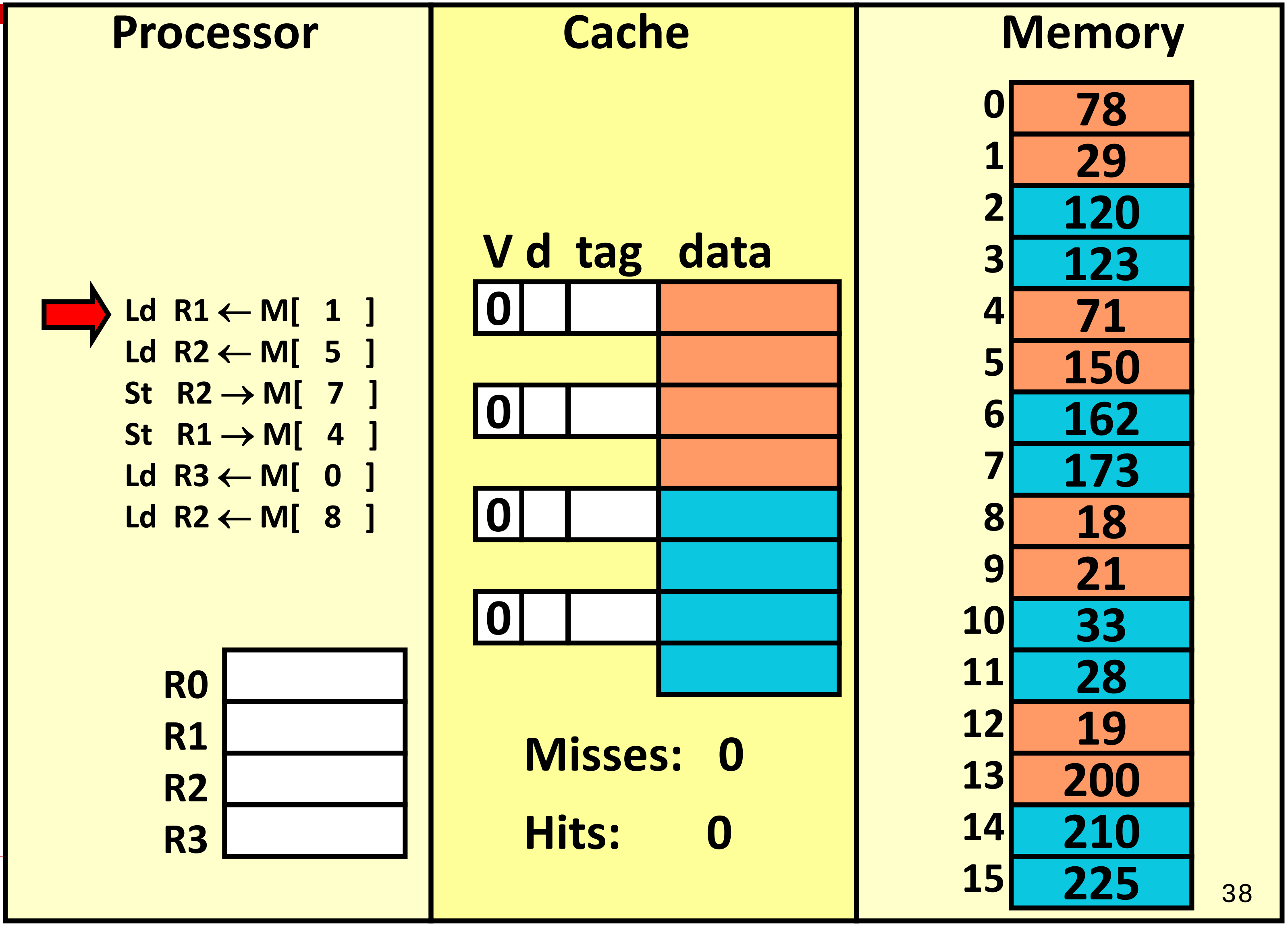
Misses: 0

Hits: 0

Memory

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

Set-associative cache (REF 1)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[7]
St R1 → M[4]
Ld R3 ← M[0]
Ld R2 ← M[8]

R0	
R1	
R2	
R3	

Cache

V	d	tag	data
0			
0			
0			
0			

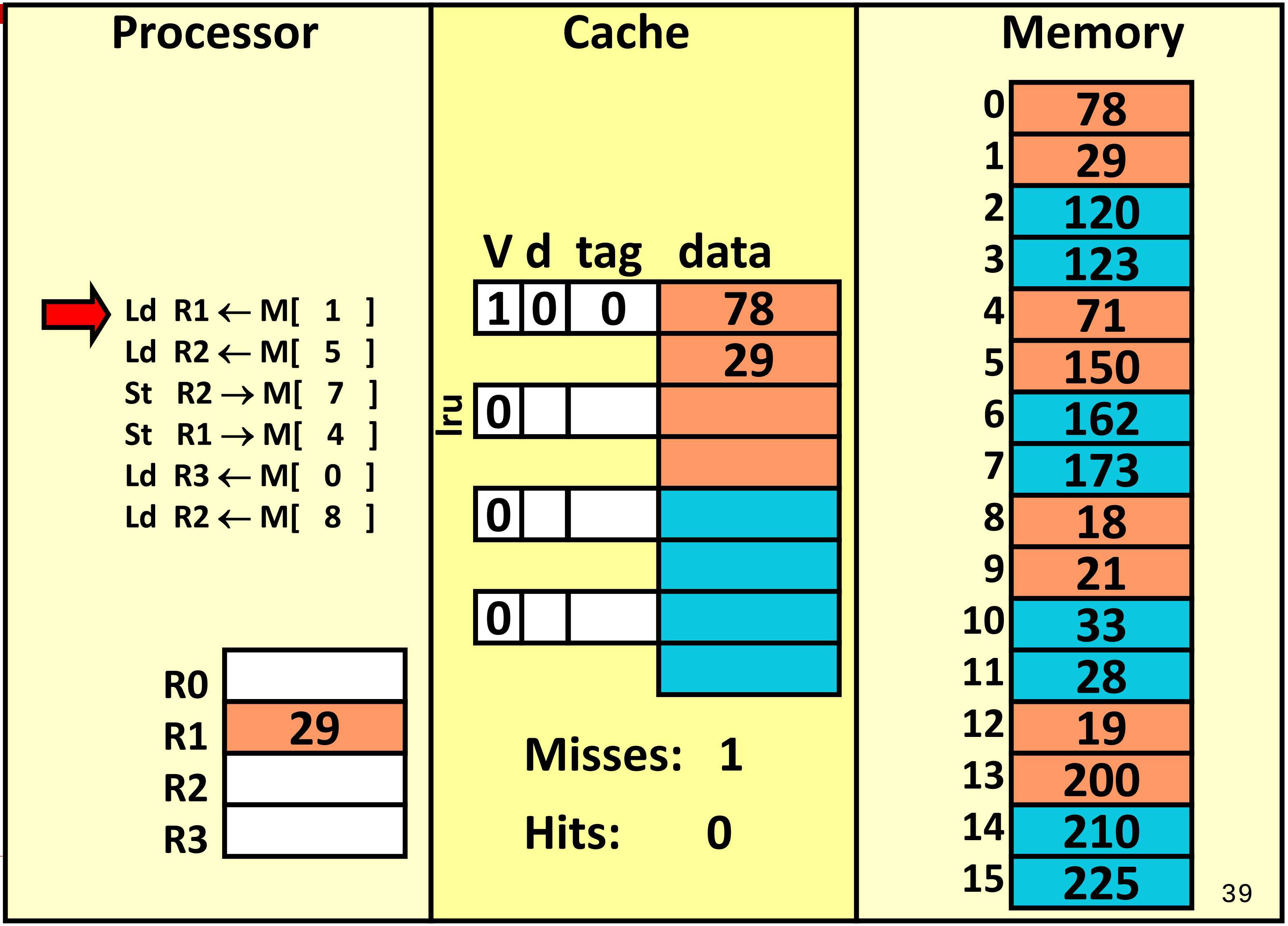
Misses: 0

Hits: 0

Memory

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

Set-associative cache (REF 1)



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  7  ]
St  R1 → M[  4  ]
Ld  R3 ← M[  0  ]
Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	
R3	

Cache

V	d	tag	data
1	0	0	78
			29
0			
0			
0			

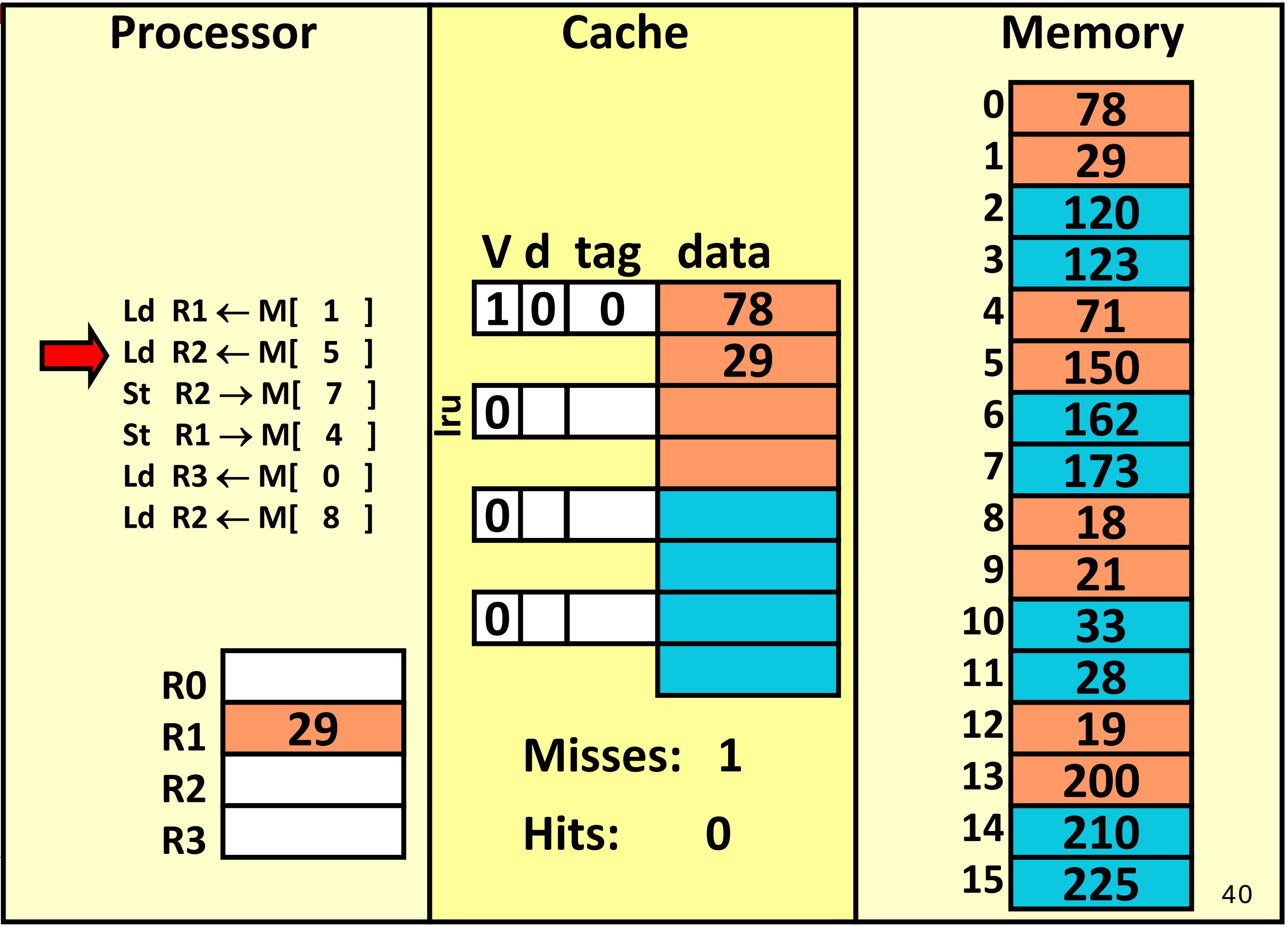
Misses: 1

Hits: 0

Memory

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

Set-associative cache (REF 2)



Processor

```
    Ld  R1 ← M[  1  ]
⇨   Ld  R2 ← M[  5  ]
    St  R2 → M[  7  ]
    St  R1 → M[  4  ]
    Ld  R3 ← M[  0  ]
    Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	
R3	

Cache

lru

V	d	tag	data
1	0	0	78
			29
0			
0			
0			

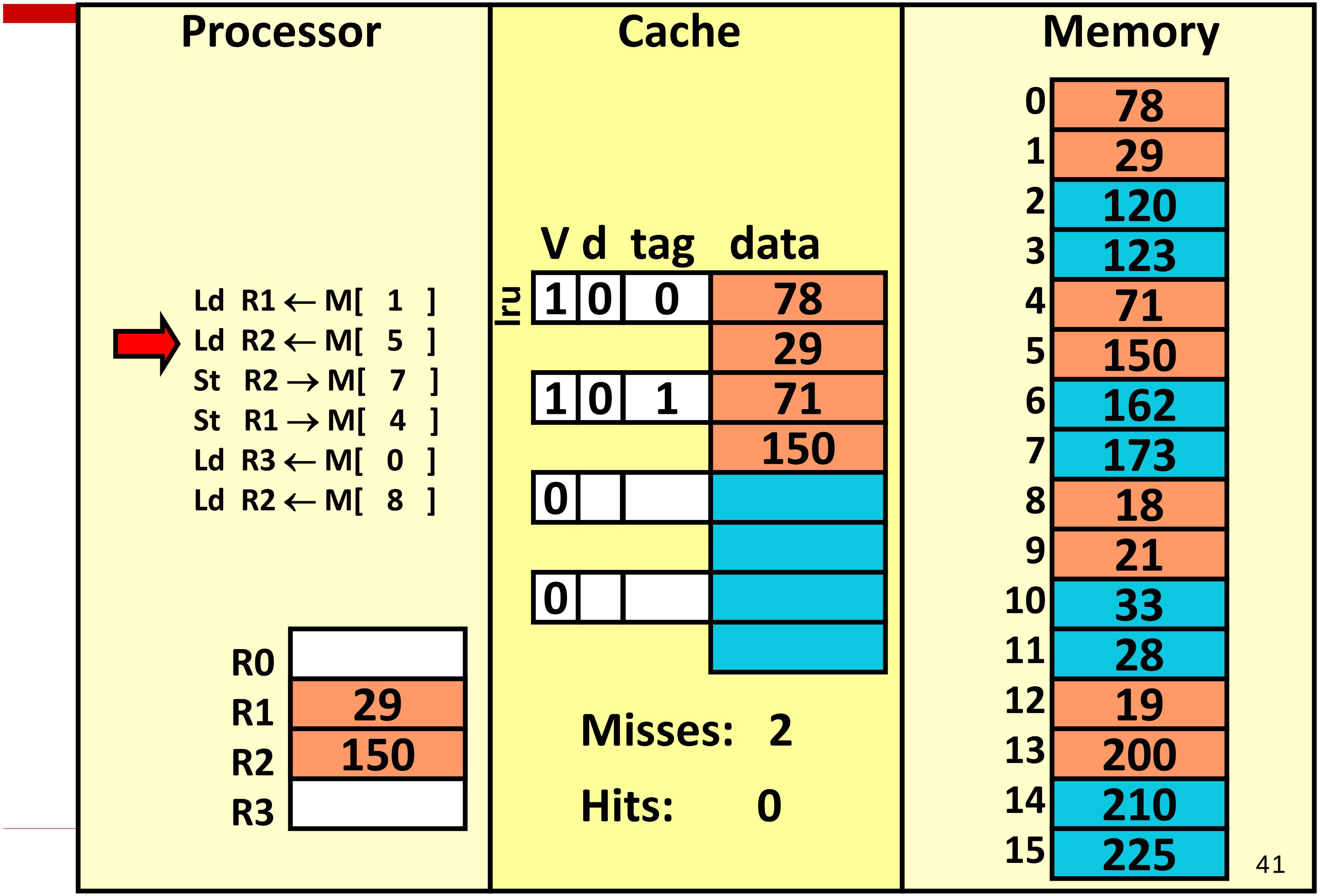
Misses: 1

Hits: 0

Memory

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

Set-associative cache (REF 2)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[7]
St R1 → M[4]
Ld R3 ← M[0]
Ld R2 ← M[8]

R0	
R1	29
R2	150
R3	

Cache

lru

V	d	tag	data
1	0	0	78
			29
1	0	1	71
			150
0			
0			

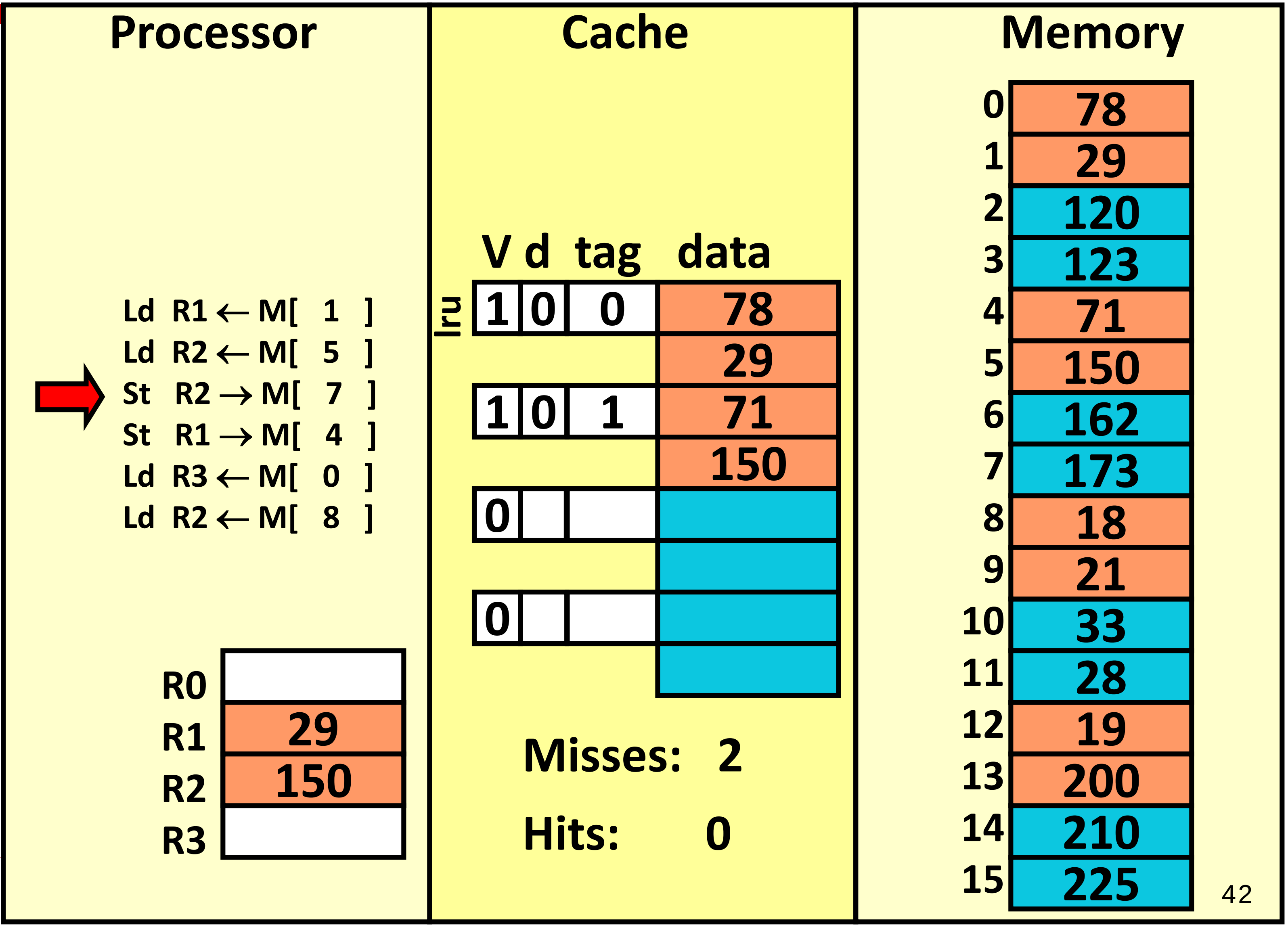
Misses: 2

Hits: 0

Memory

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

Set-associative cache (REF 3)



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  7  ]
St  R1 → M[  4  ]
Ld  R3 ← M[  0  ]
Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	150
R3	

Cache

V	d	tag	data
1	0	0	78
			29
1	0	1	71
			150
0			
0			

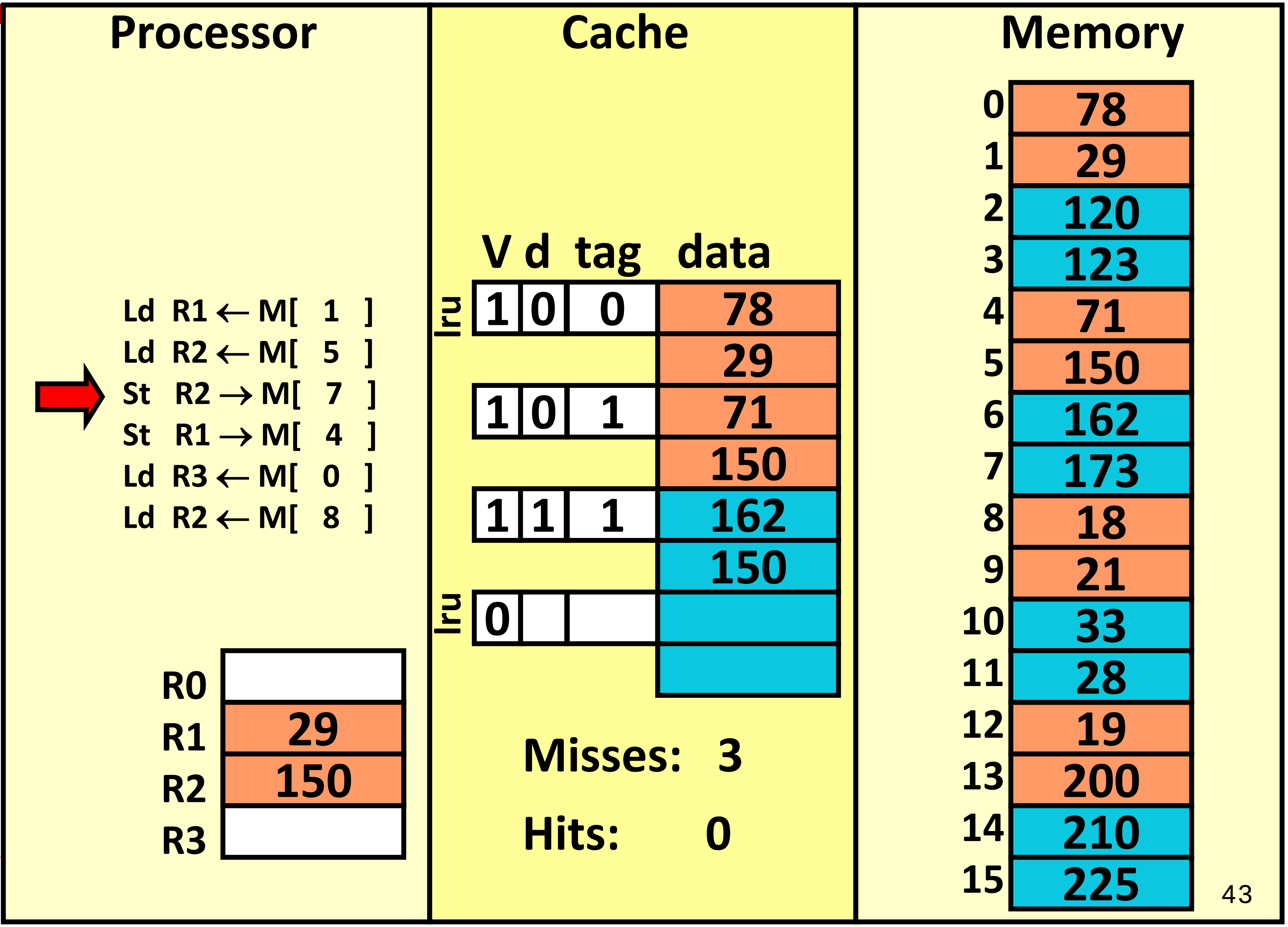
Misses: 2

Hits: 0

Memory

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

Set-associative cache (REF 3)



Processor

```
  Ld  R1 ← M[  1  ]
  Ld  R2 ← M[  5  ]
➡ St  R2 → M[  7  ]
  St  R1 → M[  4  ]
  Ld  R3 ← M[  0  ]
  Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	150
R3	

Cache

	V	d	tag	data
lru	1	0	0	78
				29
	1	0	1	71
				150
	1	1	1	162
				150
lru	0			

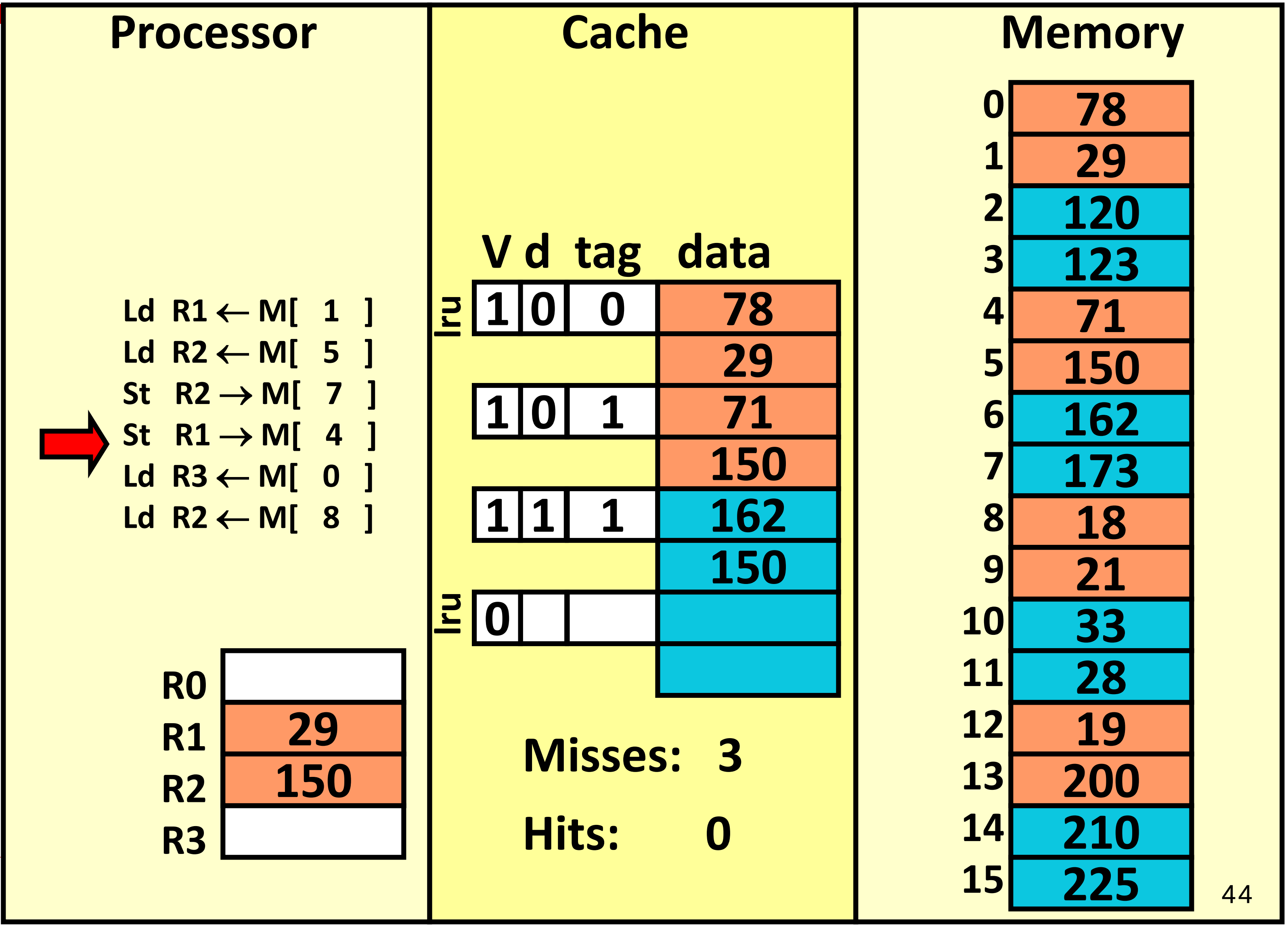
Misses: 3

Hits: 0

Memory

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

Set-associative cache (REF 4)



Processor

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
    St  R2 → M[  7  ]
➡   St  R1 → M[  4  ]
    Ld  R3 ← M[  0  ]
    Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	150
R3	

Cache

	V	d	tag	data
lru	1	0	0	78
				29
	1	0	1	71
				150
	1	1	1	162
				150
lru	0			

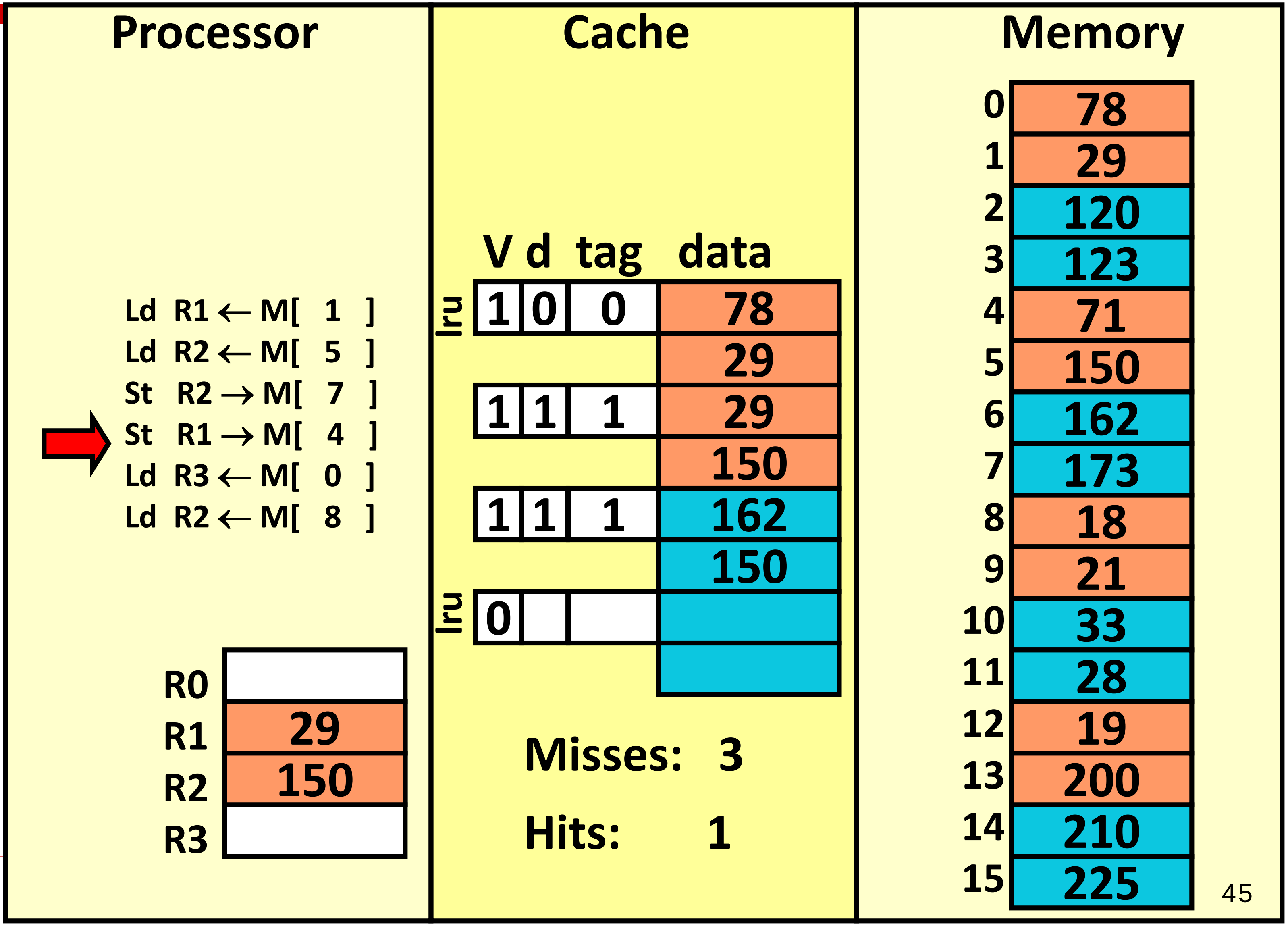
Misses: 3

Hits: 0

Memory

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

Set-associative cache (REF 4)



Processor

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
    St  R2 → M[  7  ]
➡   St  R1 → M[  4  ]
    Ld  R3 ← M[  0  ]
    Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	150
R3	

Cache

	V	d	tag	data
lru	1	0	0	78
				29
	1	1	1	29
				150
	1	1	1	162
				150
lru	0			

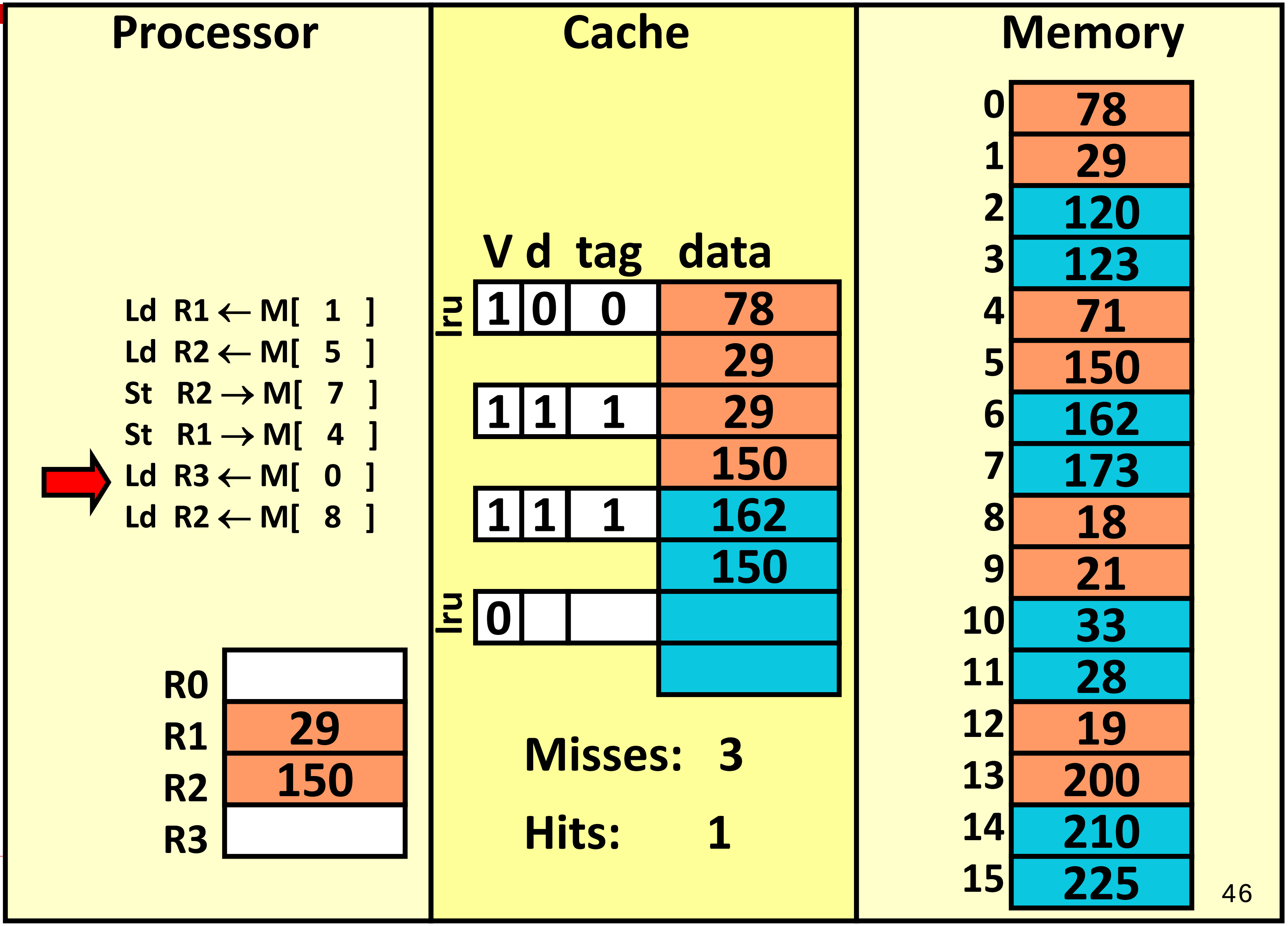
Misses: 3

Hits: 1

Memory

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

Set-associative cache (REF 5)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[7]
St R1 → M[4]
➡ Ld R3 ← M[0]
Ld R2 ← M[8]

R0	
R1	29
R2	150
R3	

Cache

	V	d	tag	data
lru	1	0	0	78
				29
	1	1	1	29
				150
	1	1	1	162
				150
lru	0			

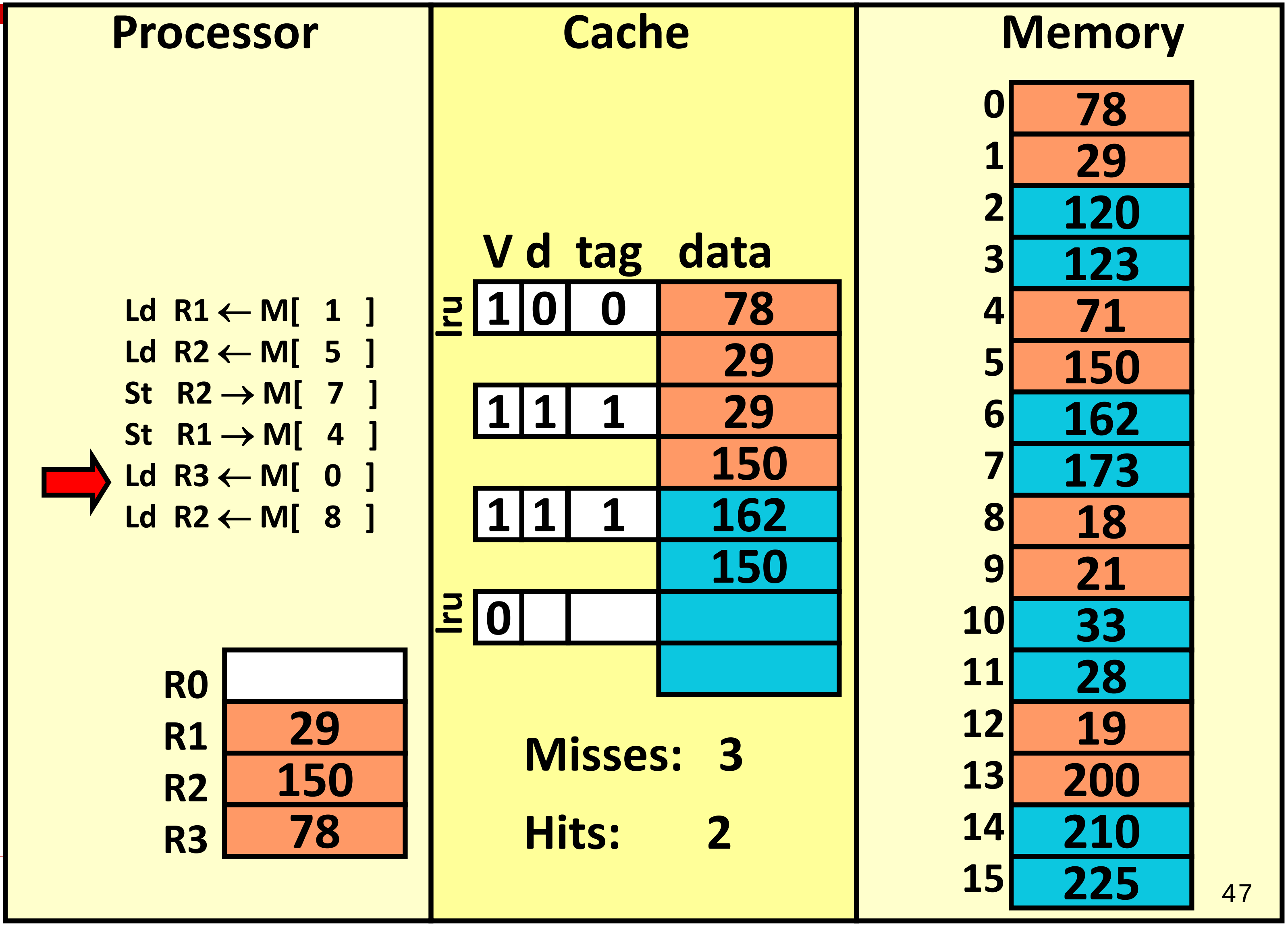
Misses: 3

Hits: 1

Memory

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

Set-associative cache (REF 5)



Processor

```
  Ld  R1 ← M[  1  ]
  Ld  R2 ← M[  5  ]
  St  R2 → M[  7  ]
  St  R1 → M[  4  ]
➡ Ld  R3 ← M[  0  ]
  Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	150
R3	78

Cache

	V	d	tag	data
lru	1	0	0	78
				29
	1	1	1	29
				150
	1	1	1	162
				150
lru	0			

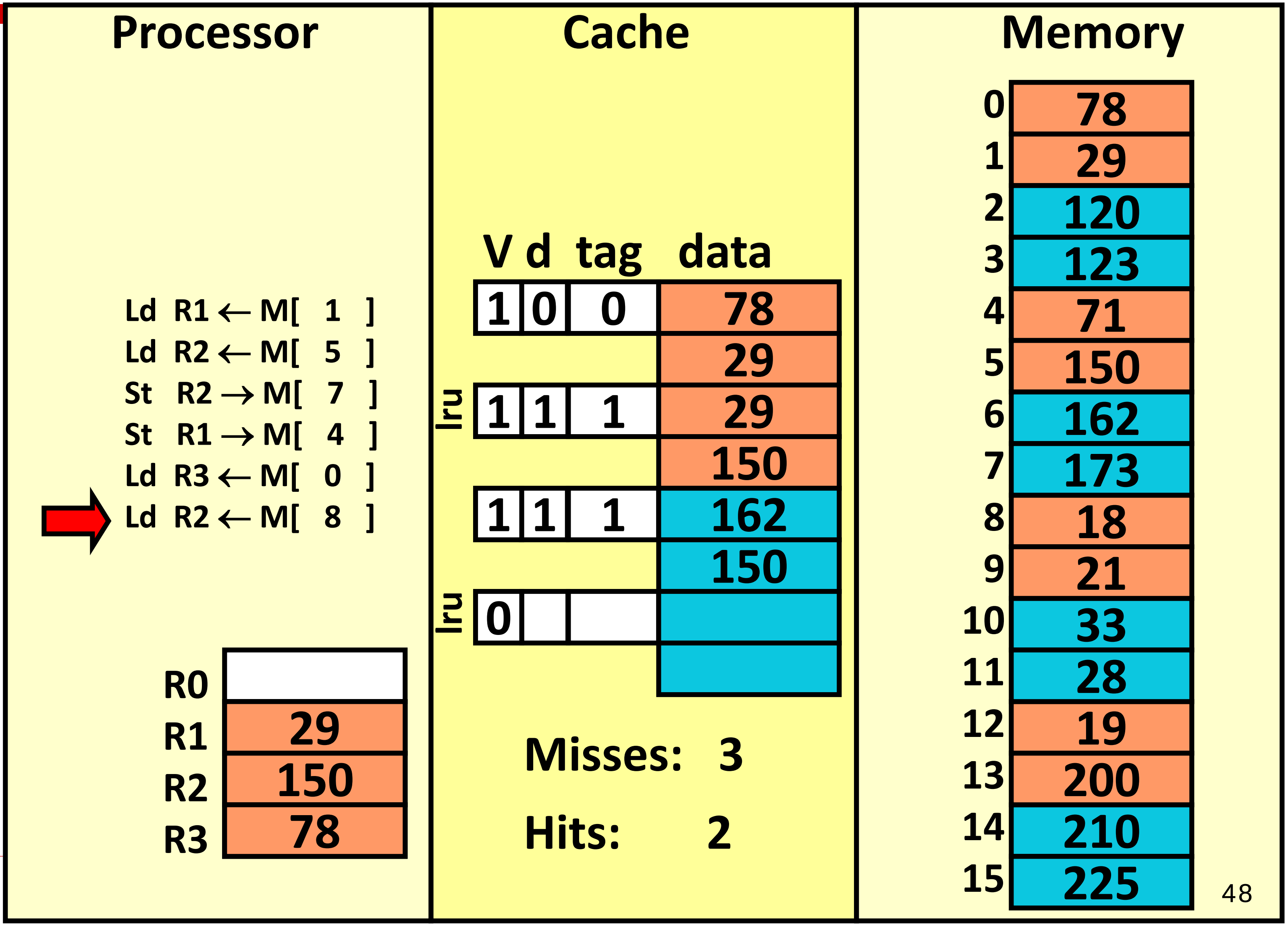
Misses: 3

Hits: 2

Memory

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

Set-associative cache (REF 6)



Processor

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
    St  R2 → M[  7  ]
    St  R1 → M[  4  ]
    Ld  R3 ← M[  0  ]
⇨   Ld  R2 ← M[  8  ]
```

R0	
R1	29
R2	150
R3	78

Cache

	V	d	tag	data
	1	0	0	78
				29
lru	1	1	1	29
				150
	1	1	1	162
				150
lru	0			

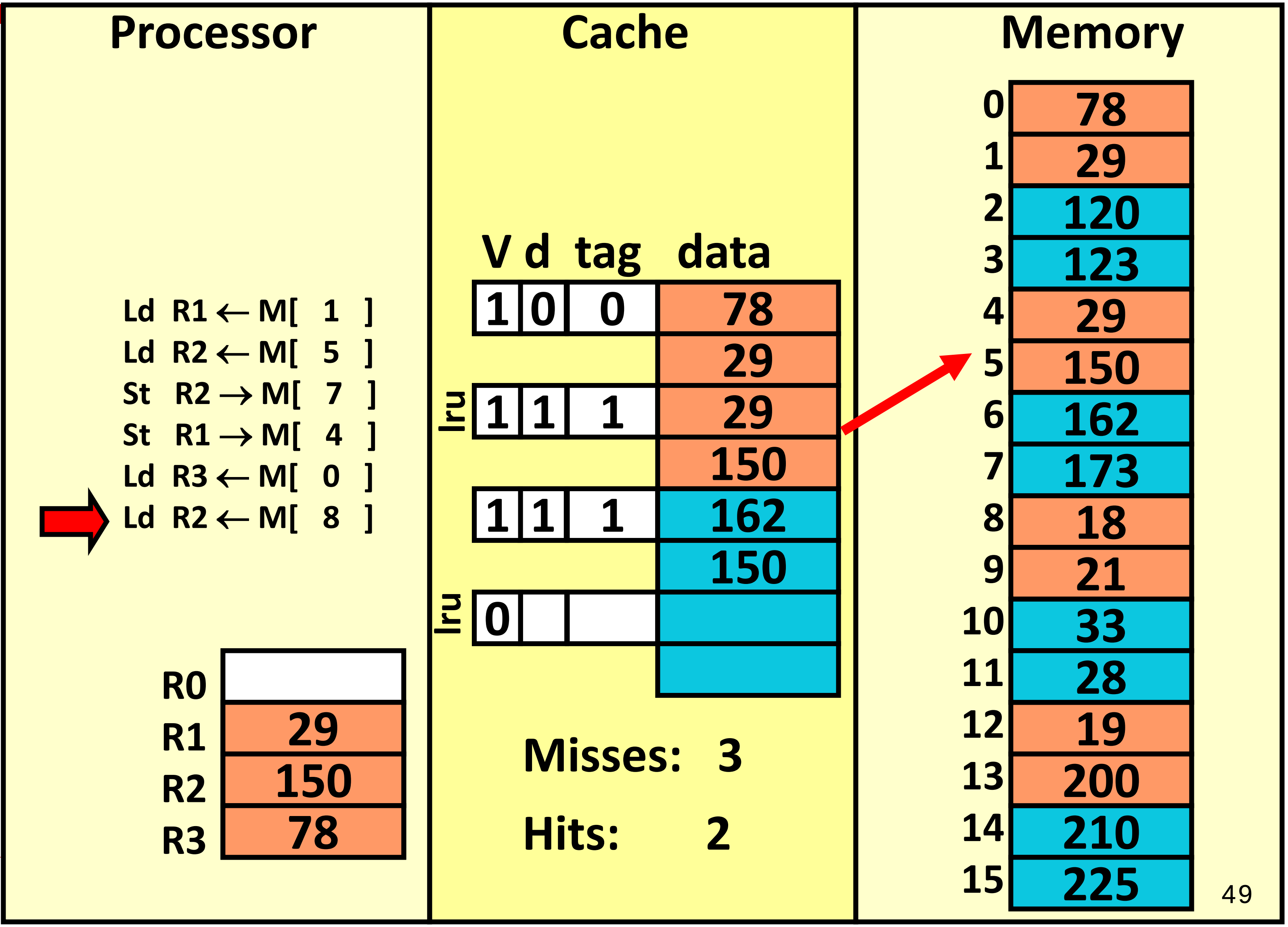
Misses: 3

Hits: 2

Memory

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

Set-associative cache (REF 6)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[7]
St R1 → M[4]
Ld R3 ← M[0]
Ld R2 ← M[8]

R0	
R1	29
R2	150
R3	78

Cache

	V	d	tag	data
	1	0	0	78
				29
lru	1	1	1	29
				150
	1	1	1	162
				150
lru	0			

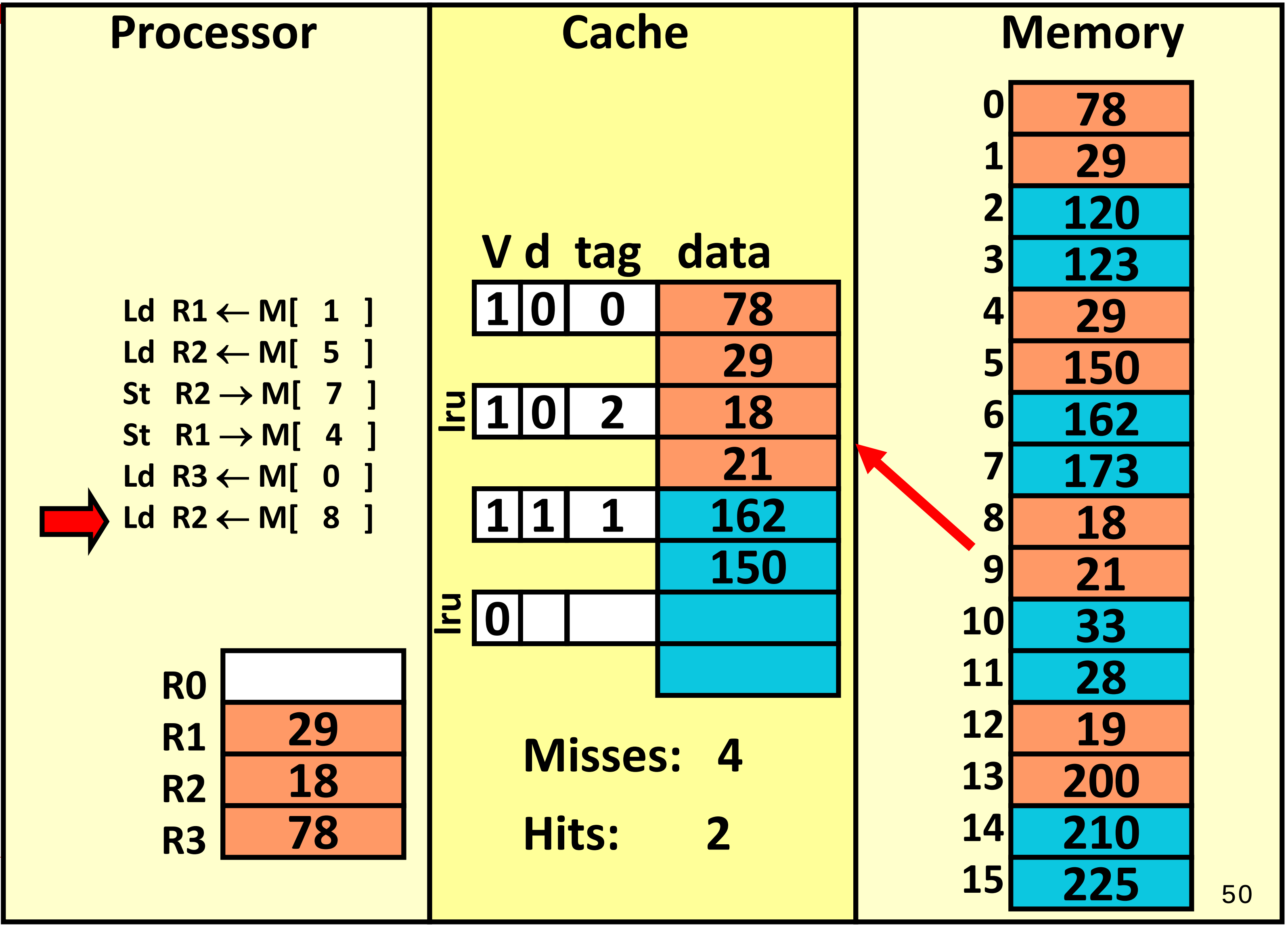
Misses: 3

Hits: 2

Memory

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

Set-associative cache (REF 6)



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
St R2 → M[7]
St R1 → M[4]
Ld R3 ← M[0]
Ld R2 ← M[8]

R0	
R1	29
R2	18
R3	78

Cache

	V	d	tag	data
	1	0	0	78
				29
lru	1	0	2	18
				21
	1	1	1	162
				150
lru	0			

Misses: 4

Hits: 2

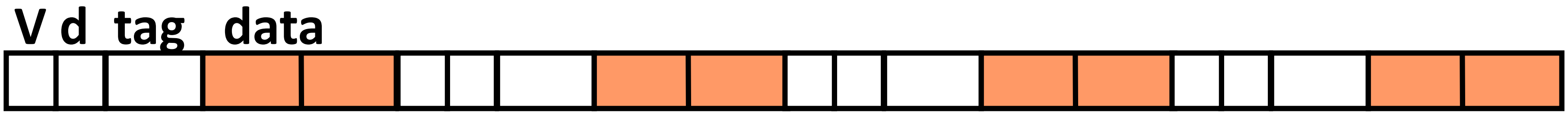
Memory

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

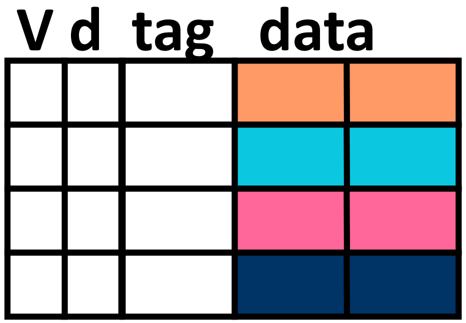
Cache Organization Comparison

Block size = 2 bytes, total cache size = 8 bytes for all caches

1. Fully associative (4-way associative)



2. Direct mapped



3. 2-way associative