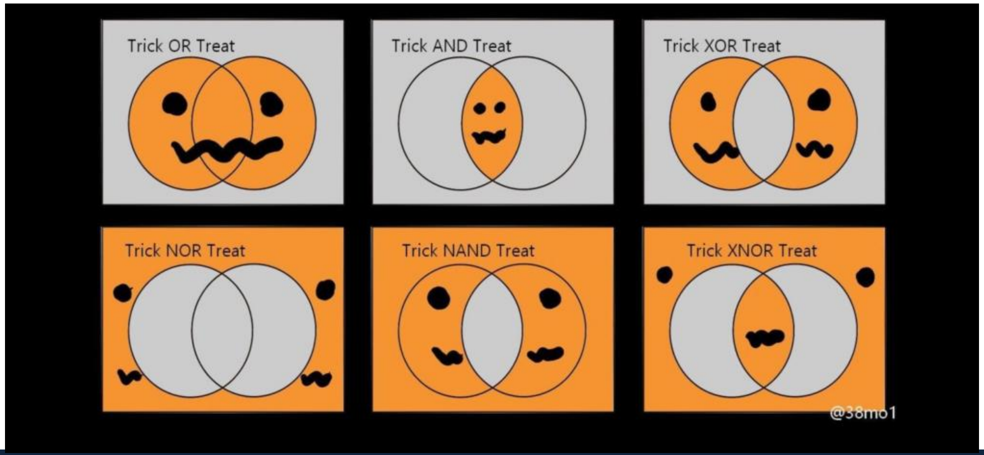
EECS 370

Improving Caches



Announcements

- Lab
  - Gradescope quiz due tonight
- P3 checkpoint due next Thursday
  - Get pipeline simulator working without hazards
- My Monday OH next week are cancelled
  - I’ll add extra later in the week

Memory Hierarchy

- Key observation: we only need to access a small amount of data at a time

- Let's use a small array of SRAM to hold data we need now
  - Call this the **cache**
  - Able to keep up with processor
  - Small (~Kilobytes), so it should be relatively cheap
- Use a large amount of DRAM for **main memory**
  - Can scale up to ~Gigabytes in size

Scaling Up

- What if we access many memory locations, and cache isn't large enough to hold all of them?
- How do we choose what to keep in the cache?
  - How does hardware predict what's most likely to be needed soon?
- Answer: **locality**
  - Temporal locality
  - Spatial locality

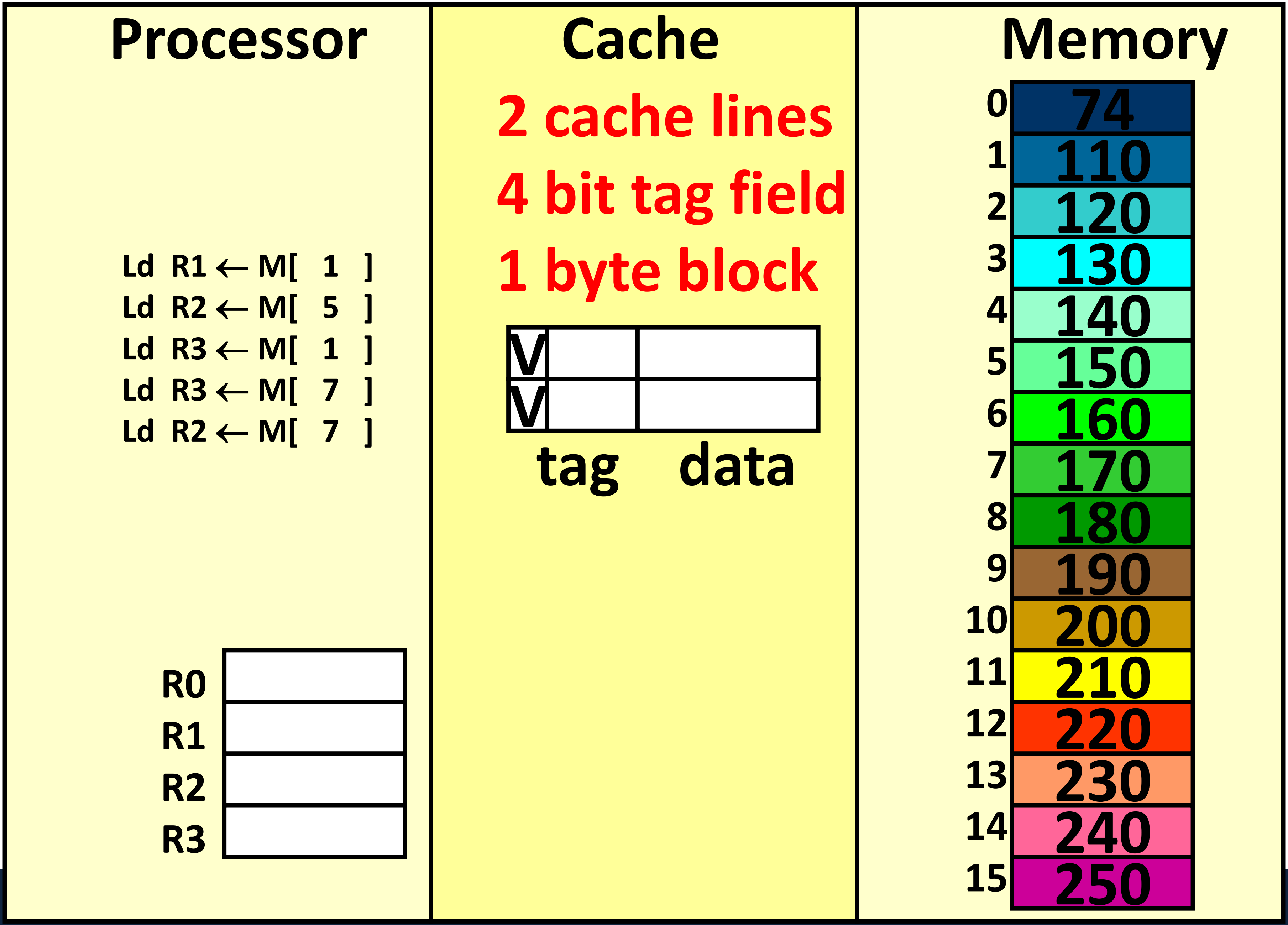
Temporal Locality

- Temporal locality: if a given memory location is referenced now, it will probably be used again in the near future
  - Why? Take a look at code you've written. You tend to use a variable multiple times
  - Corollary: if you haven't used a variable in a while, you probably won't need it very soon either
- Hardware should take advantage of this by:
  - Placing items we just accessed in the cache
  - When we need to evict something, evict whatever data was **least recently used (LRU)**

Definitions

- Hit: when data for a memory access is found in the cache
- Miss: when data for a memory access is not found in the cache
- Hit/Miss rate: percentage of memory accesses that hit/miss in the cache

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	
R2	
R3	

Cache

2 cache lines

4 bit tag field

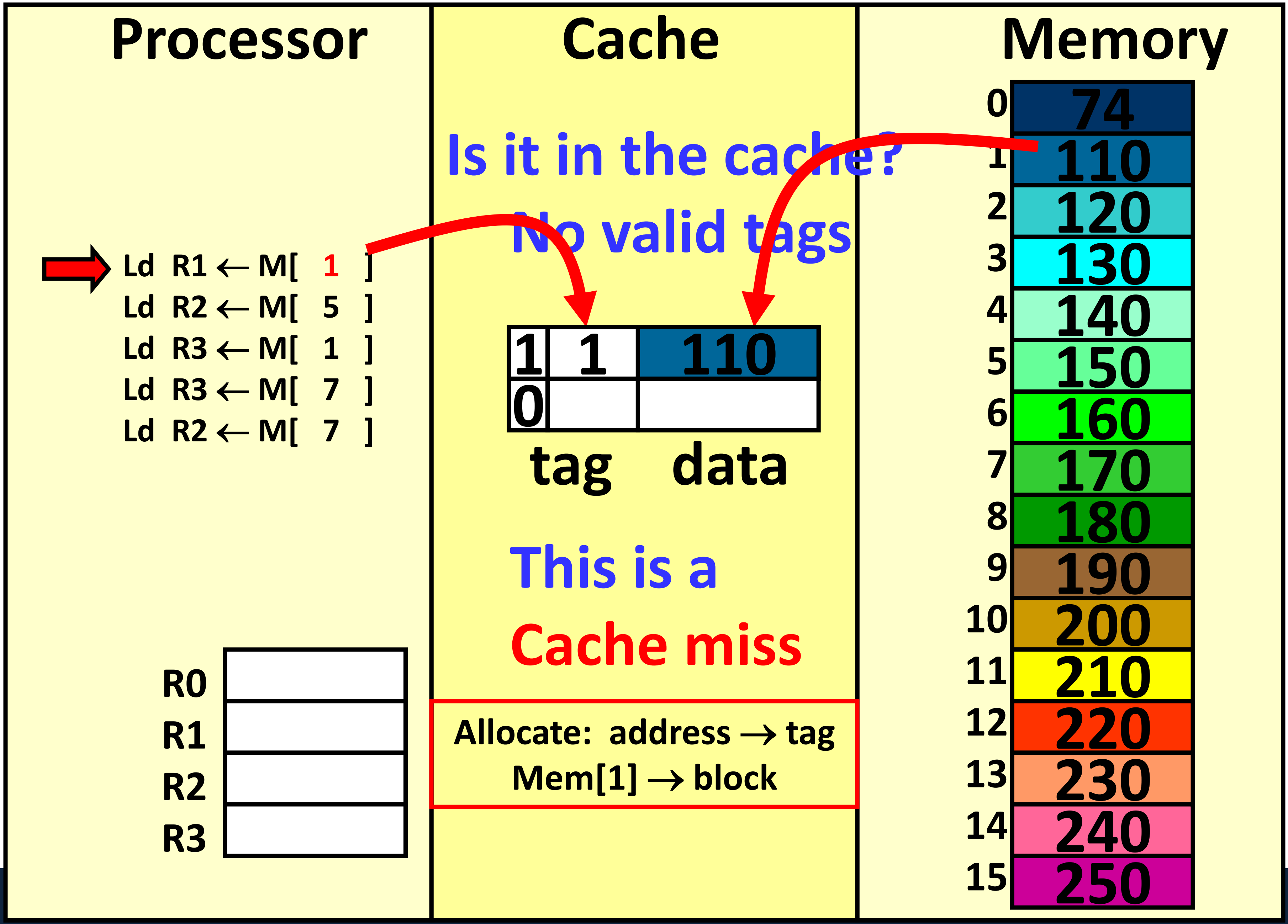
1 byte block

	tag	data
V		
V		

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0
R1
R2
R3

Cache

Is it in the cache?

No valid tags

	tag	data
1	1	110
0		

This is a

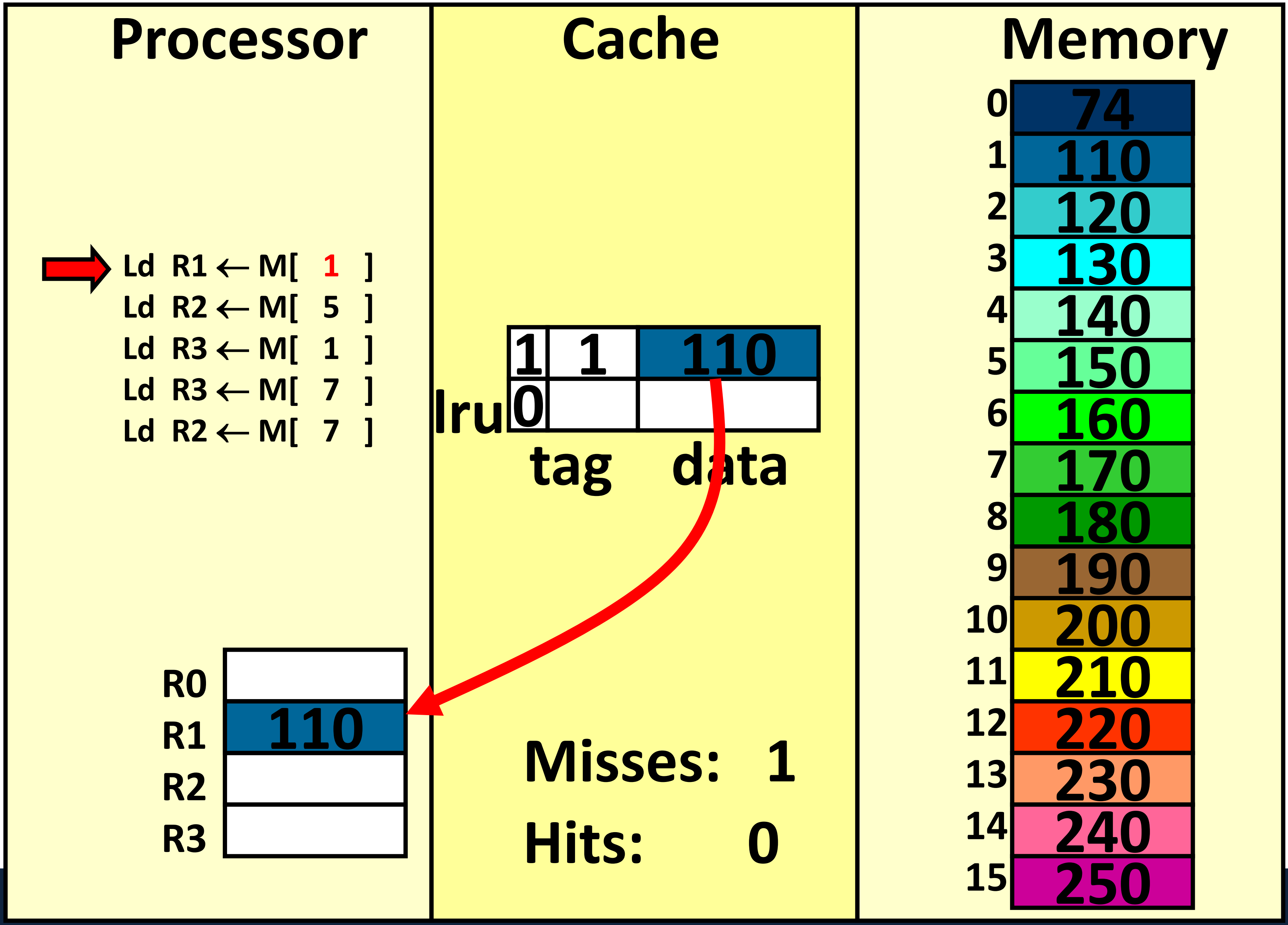
Cache miss

Allocate: address → tag
Mem[1] → block

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	
R3	

Cache

lru	tag	data
1	1	110
0		

Misses: 1

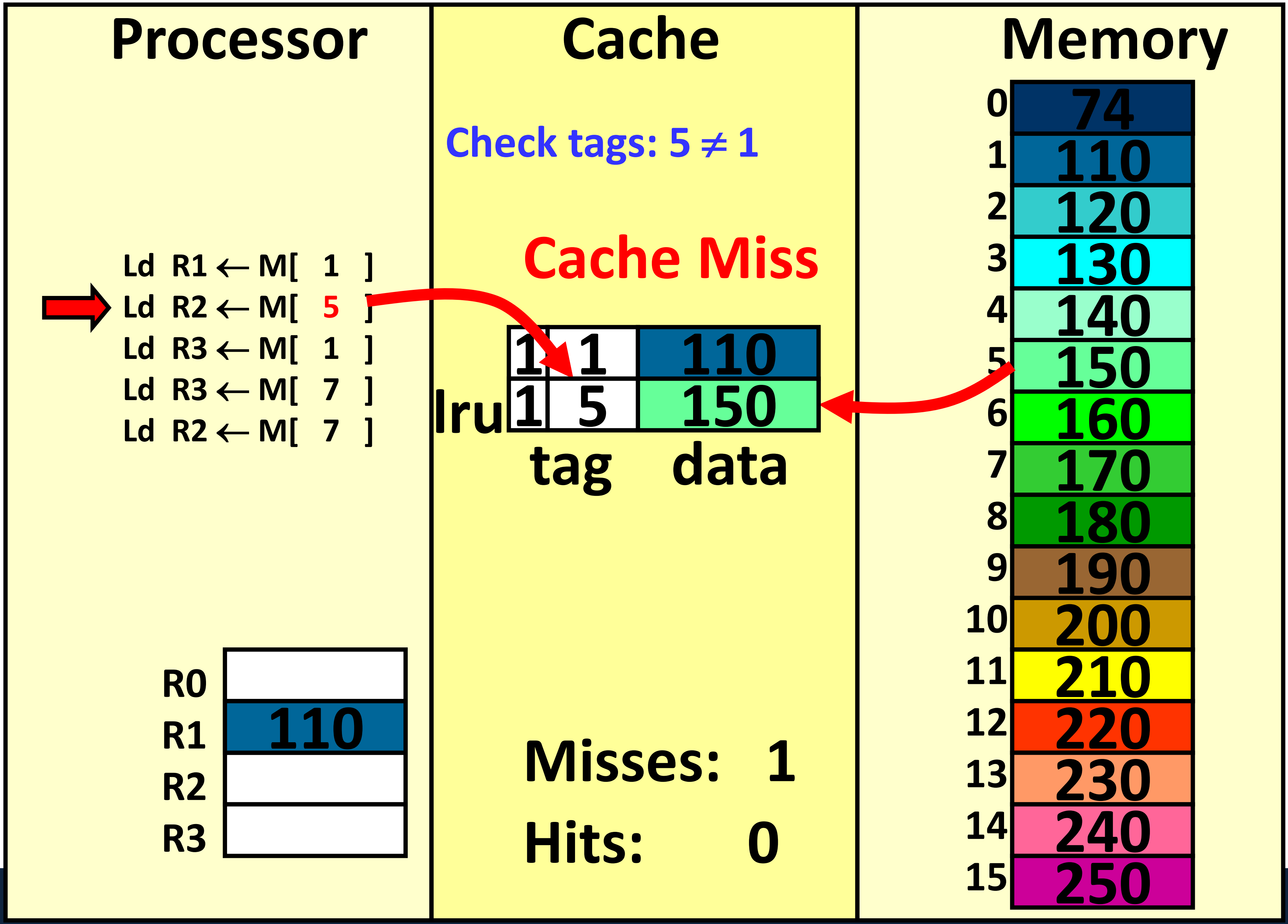
Hits: 0

Memory

0	74
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

A Very Simple Memory System

Tag comparison uses hardware called "content-addressable memory (CAM)"



Processor

```
   Ld R1 ← M[ 1 ]
⇒ Ld R2 ← M[ 5 ]
   Ld R3 ← M[ 1 ]
   Ld R3 ← M[ 7 ]
   Ld R2 ← M[ 7 ]
```

R0	
R1	110
R2	
R3	

Cache

Check tags: 5 ≠ 1

Cache Miss

	1	1	110
lru	1	5	150
		tag	data

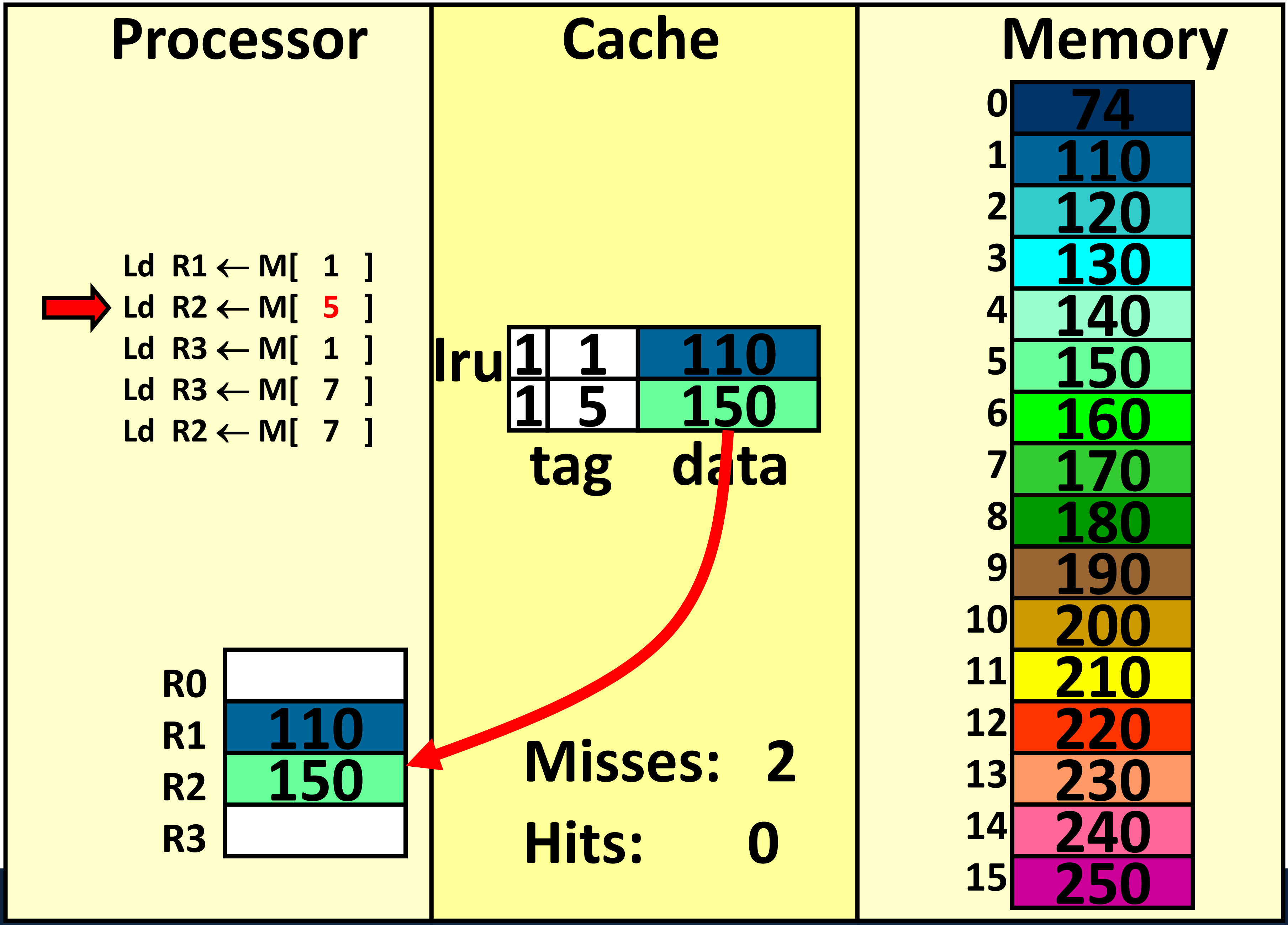
Misses: 1

Hits: 0

Memory

0	74
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

A Very Simple Memory System



Processor

```
  Ld  R1 ← M[  1  ]
⇒ Ld  R2 ← M[  5  ]
  Ld  R3 ← M[  1  ]
  Ld  R3 ← M[  7  ]
  Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	

Cache

lru	tag	data
1	1	110
1	5	150

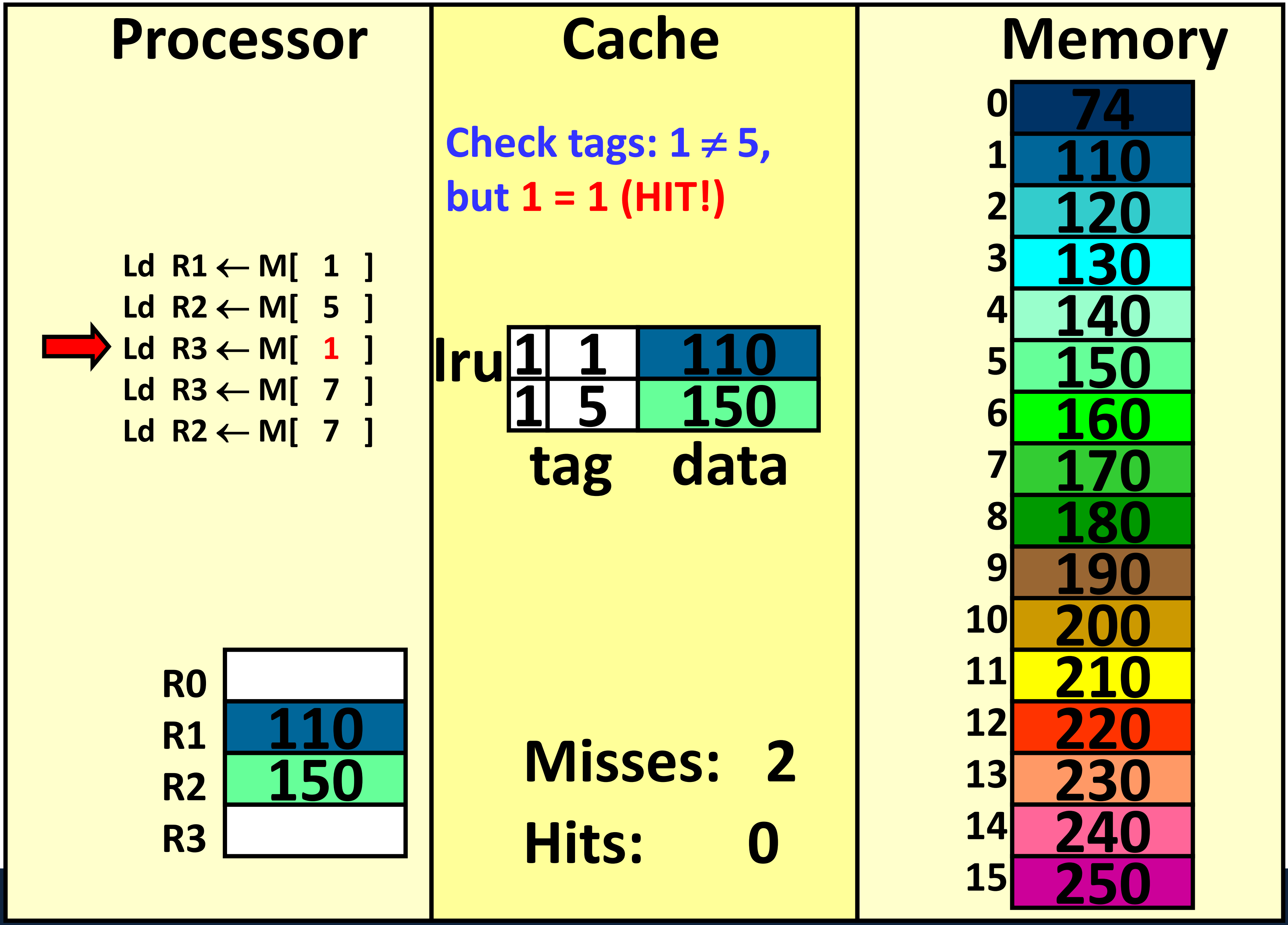
Misses: 2

Hits: 0

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	

Cache

Check tags: 1 ≠ 5, but 1 = 1 (HIT!)

lru	tag	data
1	1	110
1	5	150

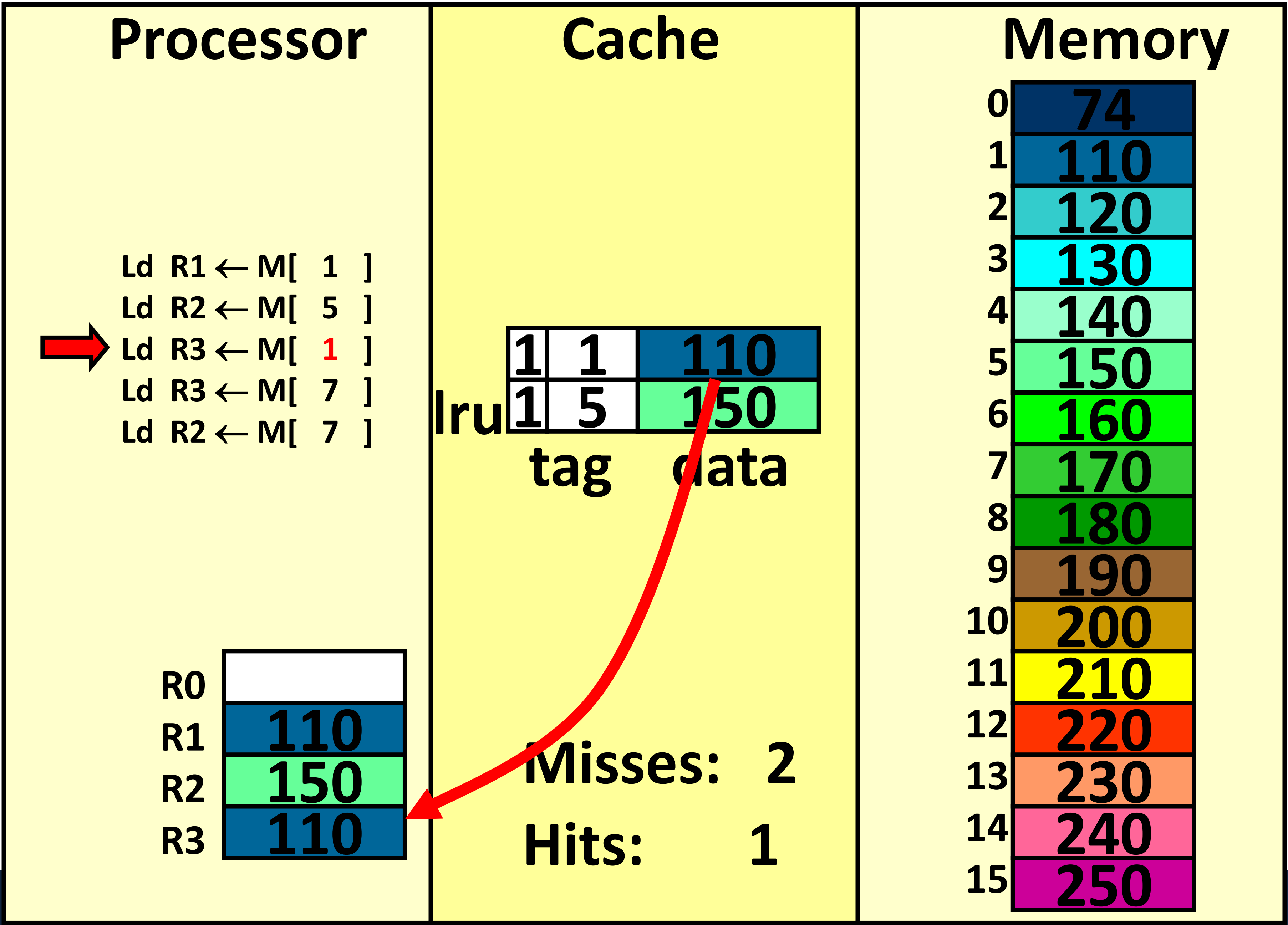
Misses: 2

Hits: 0

Memory

0	74
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

A Very Simple Memory System



Processor

```
   Ld  R1 ← M[  1  ]
   Ld  R2 ← M[  5  ]
➡  Ld  R3 ← M[  1  ]
   Ld  R3 ← M[  7  ]
   Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	110

Cache

lru	tag	data
1	1	110
1	5	150

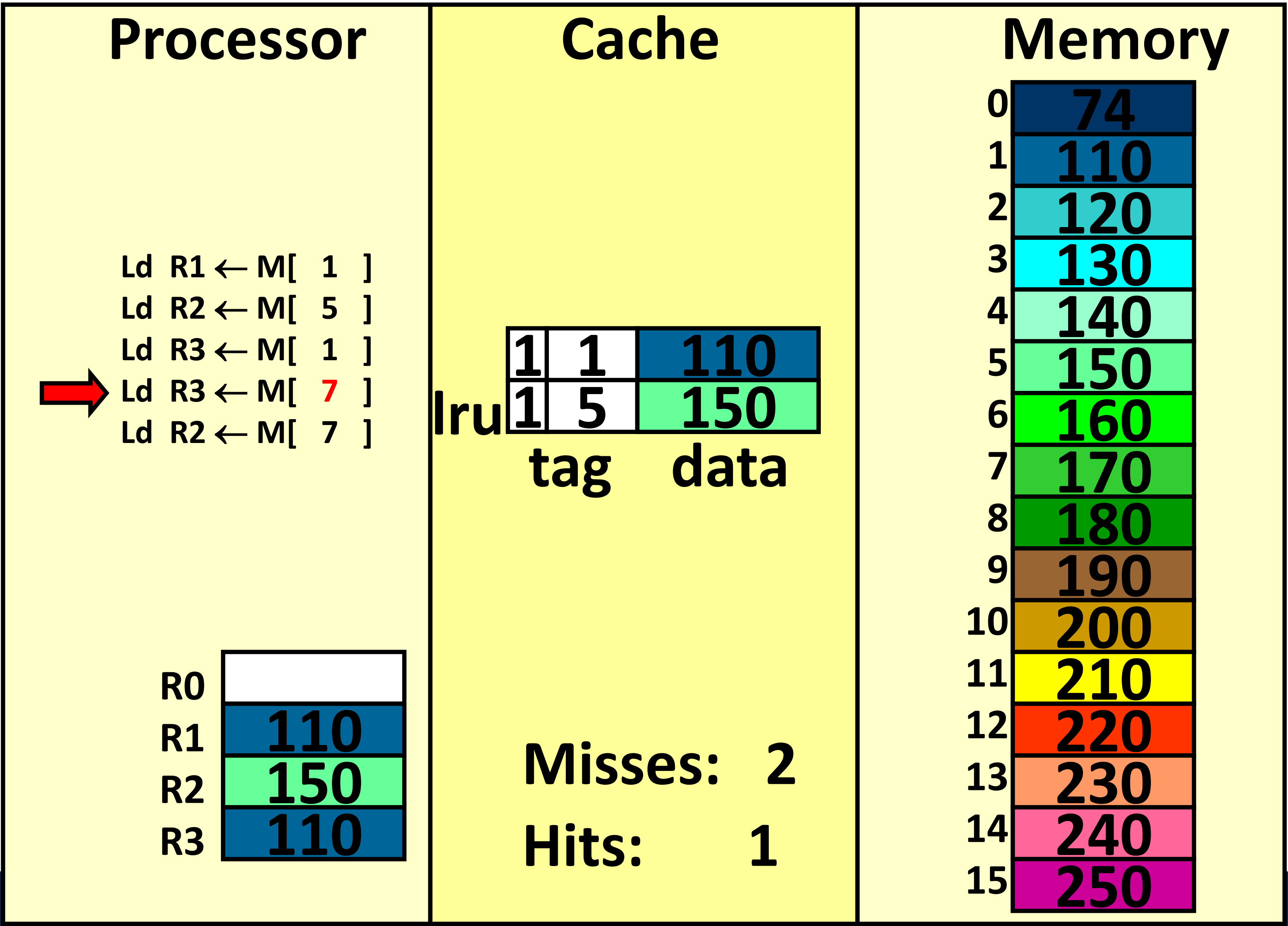
Misses: 2

Hits: 1

Memory

0	74
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

A Very Simple Memory System



Processor

```
  Ld  R1 ← M[  1  ]
  Ld  R2 ← M[  5  ]
  Ld  R3 ← M[  1  ]
➡ Ld  R3 ← M[  7  ]
  Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	110

Cache

		tag	data
	1	1	110
lru	1	5	150

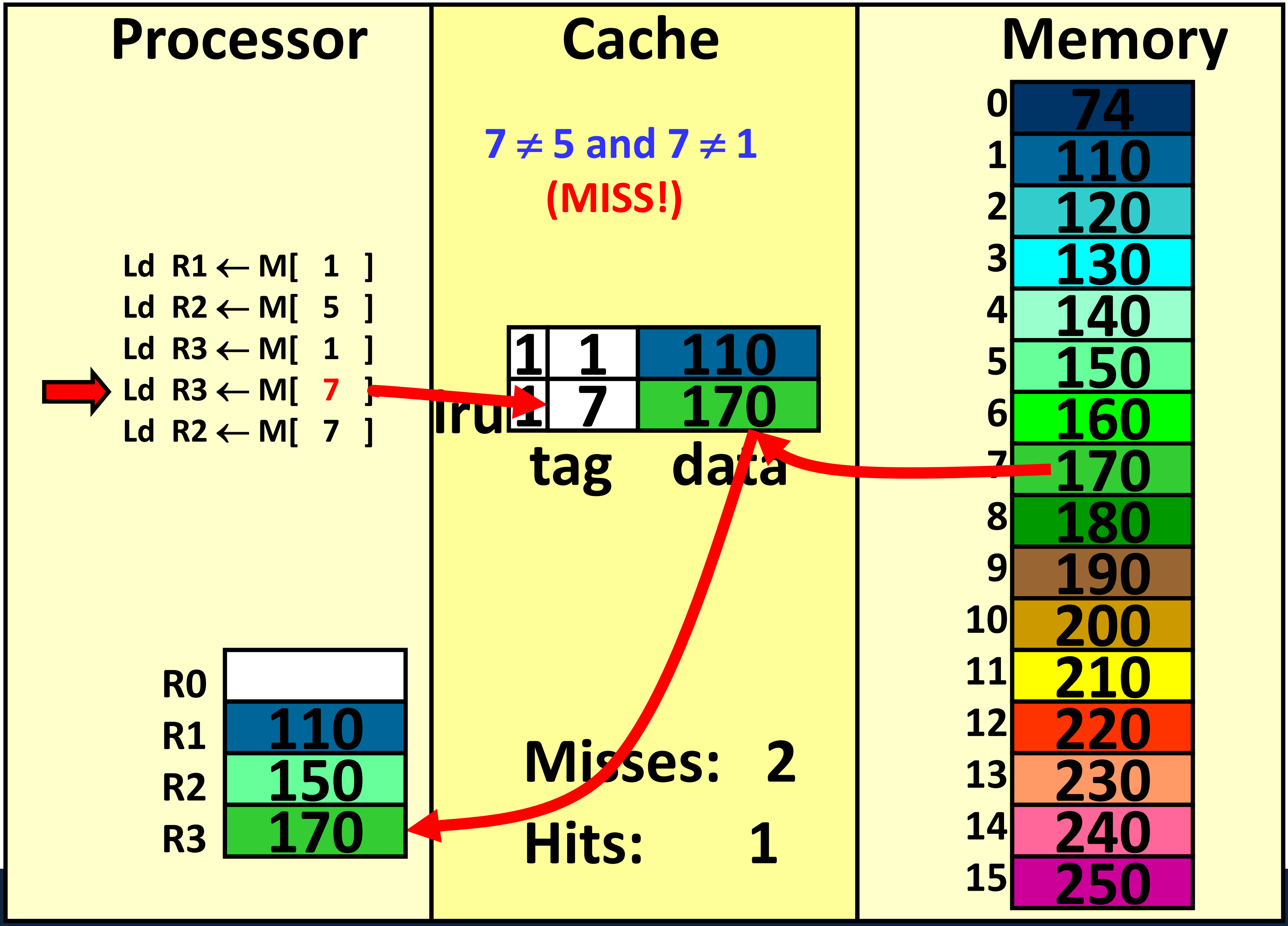
Misses: 2

Hits: 1

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	170

Cache

7 ≠ 5 and 7 ≠ 1

(MISS!)

lru	tag	data
1	1	110
1	7	170

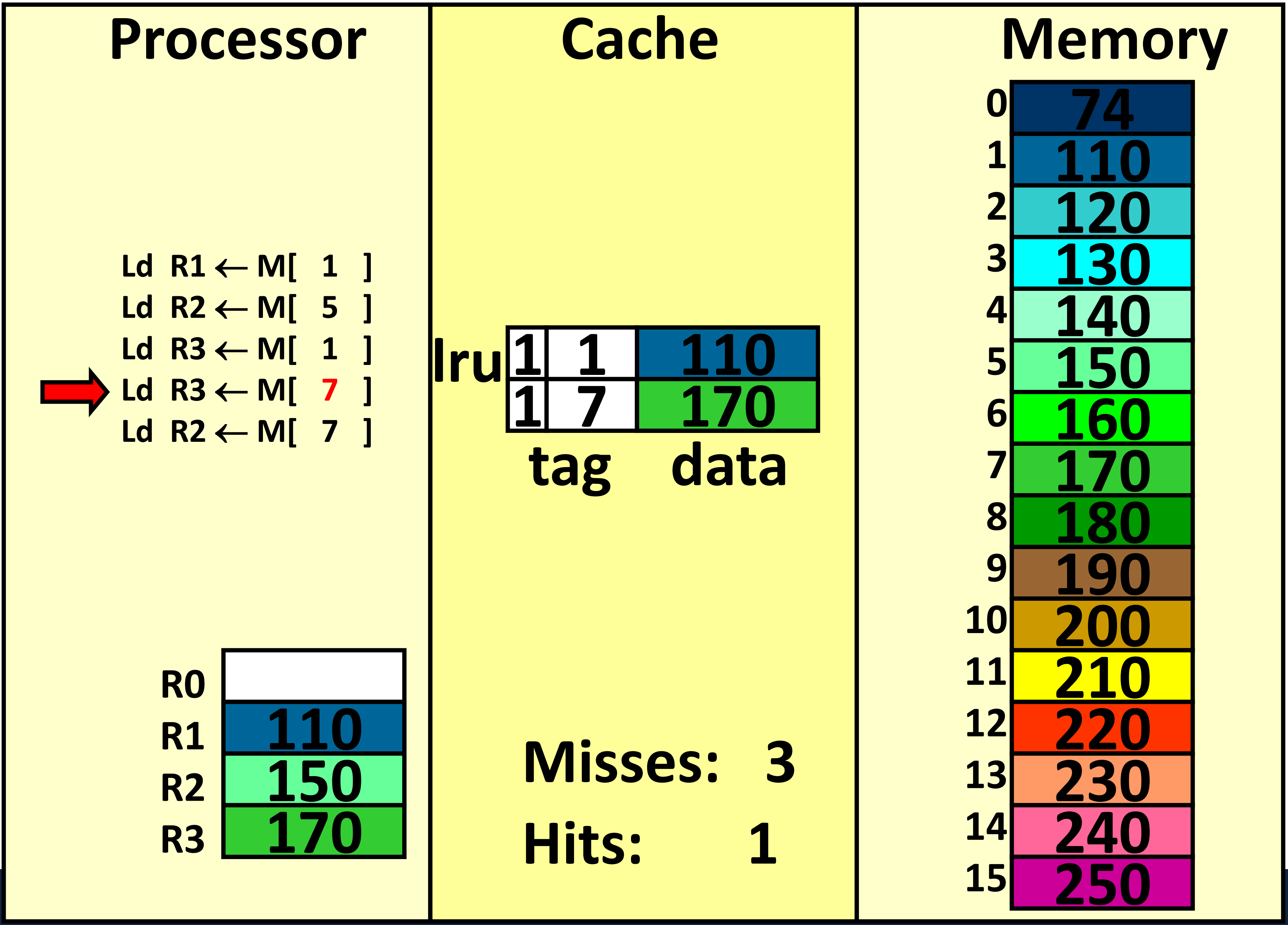
Misses: 2

Hits: 1

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

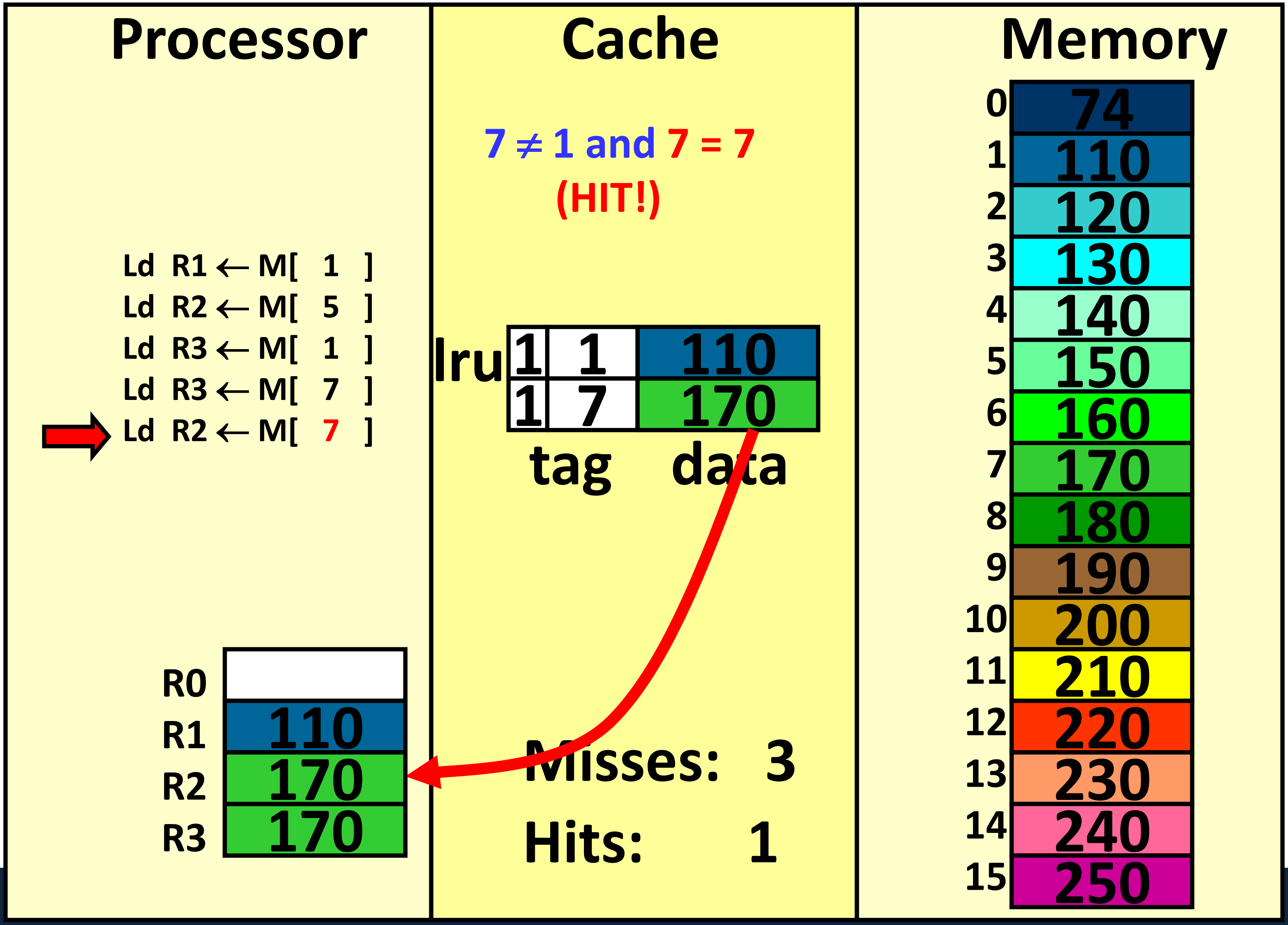
Misses: 3

Hits: 1

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	170
R3	170

Cache

7 ≠ 1 and 7 = 7
(HIT!)

lru	tag	data
1	1	110
1	7	170

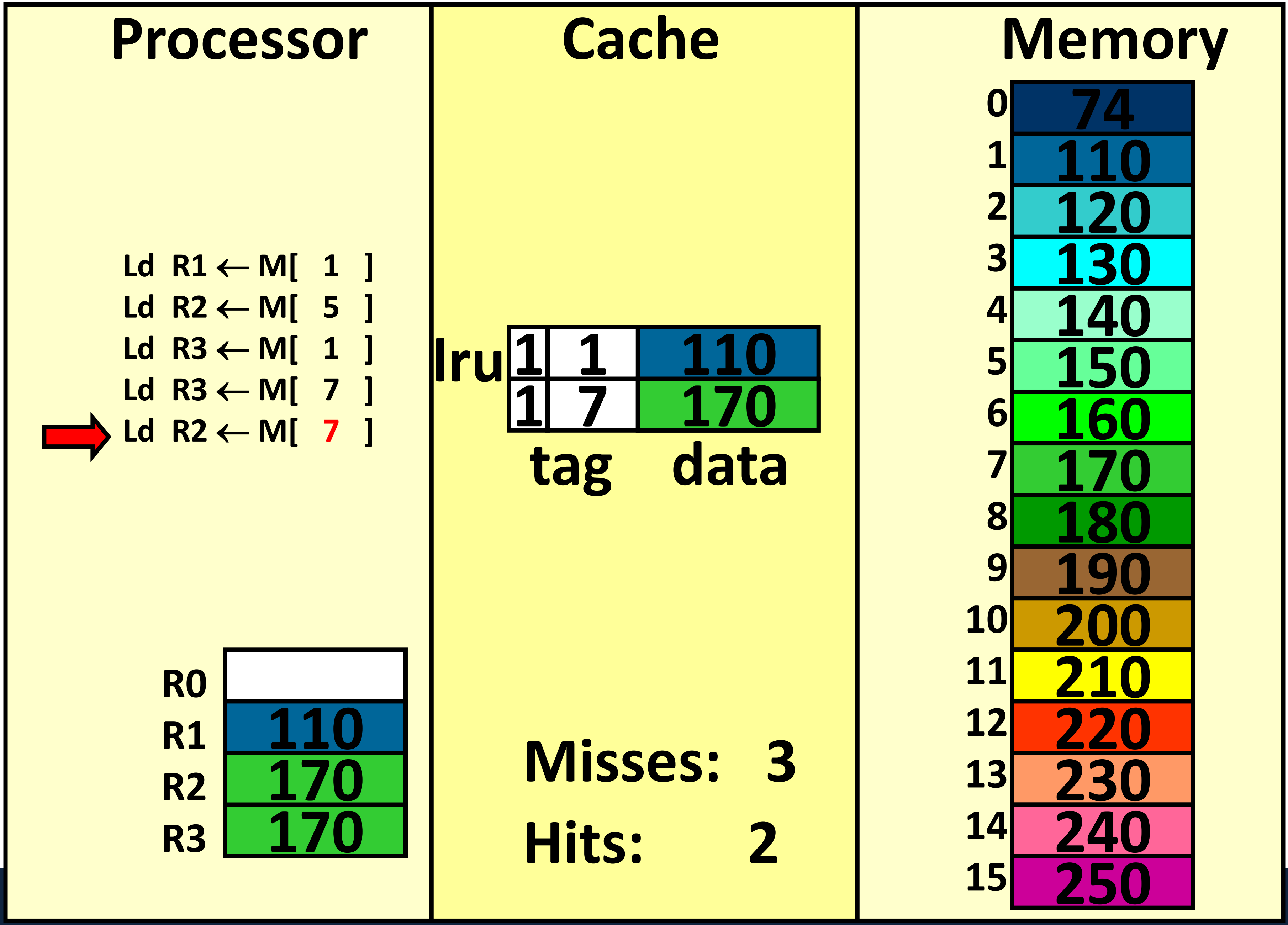
Misses: 3

Hits: 1

Memory

0	74
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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
➡ Ld R2 ← M[7]

R0	
R1	110
R2	170
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

Misses: 3
Hits: 2

Memory

0	74
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

<u>Poll:</u> Average access time?

Example Problem

- Assume the following:
  - Cache has 1 cycle access time
  - If data is not found in cache, main memory is then accessed instead
  - Main memory has 100 cycle access time
- If we have a 90% hit rate in the cache, what is the average memory latency?

$1 + 0.1 * (100) = 11$

Calculating Average Access Latency

- Average Latency:
  - $cache\ latency + (memory\ latency \cdot miss\ rate)$
- To improve latency, either:
  - Improve memory access latency, or
  - Improve cache access latency, or
  - Improve cache hit rate

Agenda

- **Larger Cache Blocks**
- Extra Problems
- LRU with More than Two Blocks
- Write-Through Cache
- Write-Back Cache

Calculating Size

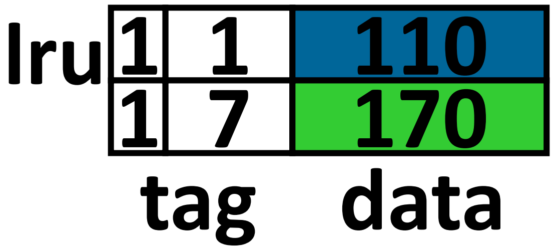
- How many bits is used in cache?
  - Storing data
    - 2 bytes of SRAM
  - Calculate overhead (non-data)
    - This cost is often forgotten for caches, but it drives up the cost of real designs!
    - 2 4-bit tags
    - 2 valid bits
- What is the storage requirement

__Poll:__ Which of the following would reduce tag overhead (as an overall percentage)? (select all that apply)

a) Increase number of cache entries
b) Decrease number of cache entries
c) Use smaller addresses
d) Store more data in each cache entry

How can we reduce overhead?

- Have a smaller address
  - Impractical, and caches are supposed to be micro-architectural
- Cache bigger units than bytes
  - Each block has a single tag, and blocks can be whatever size we choose.



lru	tag	data
1	1	110
1	7	170

Increasing Block Size

Case 1:

Block size: **1** bytes

V	tag	data (block)
1	0	74
1	6	160

How many bits needed per tag?

= $\log_2$(number of blocks in memory) = $\log_2(16)$

= 4 bits

Overhead = (4+1) / 8 = 62.5%

Case 2:

Block size: **2** bytes

V	tag	data (block)	
1	0	74	110
1	3	160	170

How many bits needed per tag?

= $\log_2$(number of blocks in memory) = $\log_2(8)$

= 3 bits

Overhead = (3+1) / 16 = 25%

	Memory	Tag (case 1)	Tag (case 2)
0	74	0	0
1	110	1	0
2	120	2	1
3	130	3	1
4	140	4	2
5	150	5	2
6	160	6	3
7	170	7	3
8	180	8	4
9	190	9	4
10	200	10	5
11	210	11	5
12	220	12	6
13	230	13	6
14	240	14	7
15	250	15	7

Figuring out the tag

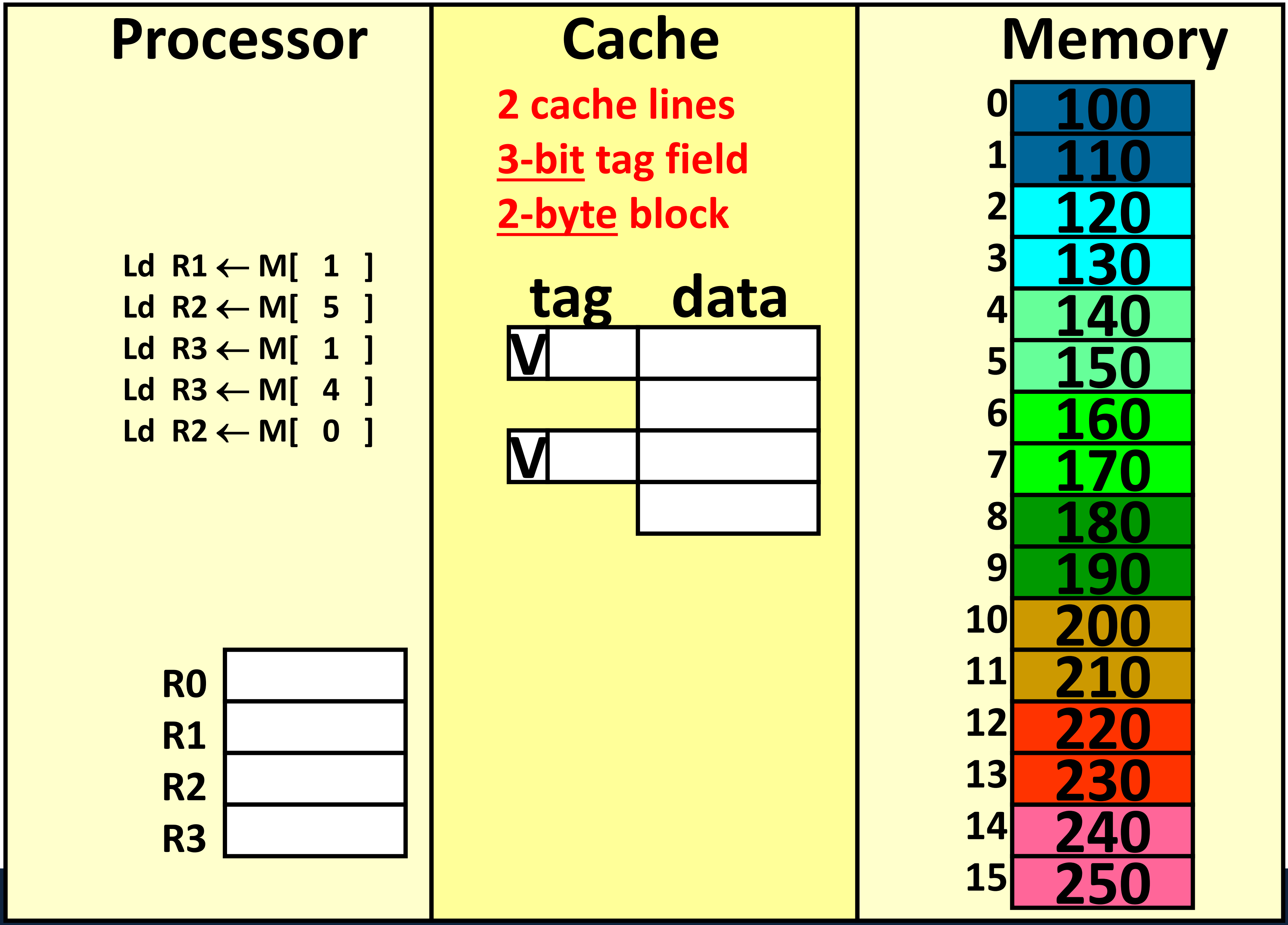
- If block size is N, what's the pattern for figuring out the tag from the address?
  - $tag = \left\lfloor \frac{addr}{block\ size} \right\rfloor$
- If block size is power of 2, then this is just everything except the $\log_2(block\ size)$ bits of the address in binary!
- E.g.

$$0d11 = 0b1011$$
$$Tag = 0b101 = 0d5$$
$$Block\ Offset = 1$$

- Remaining bits (block offset) tells us how far into the block the data is

	Memory	Tag (case 2)
0	74	0
1	110	0
2	120	1
3	130	1
4	140	2
5	150	2
6	160	3
7	170	3
8	180	4
9	190	4
10	200	5
11	210	5
12	220	6
13	230	6
14	240	7
15	250	7

Block size for caches



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  4  ]
Ld  R2 ← M[  0  ]
```

Cache

2 cache lines

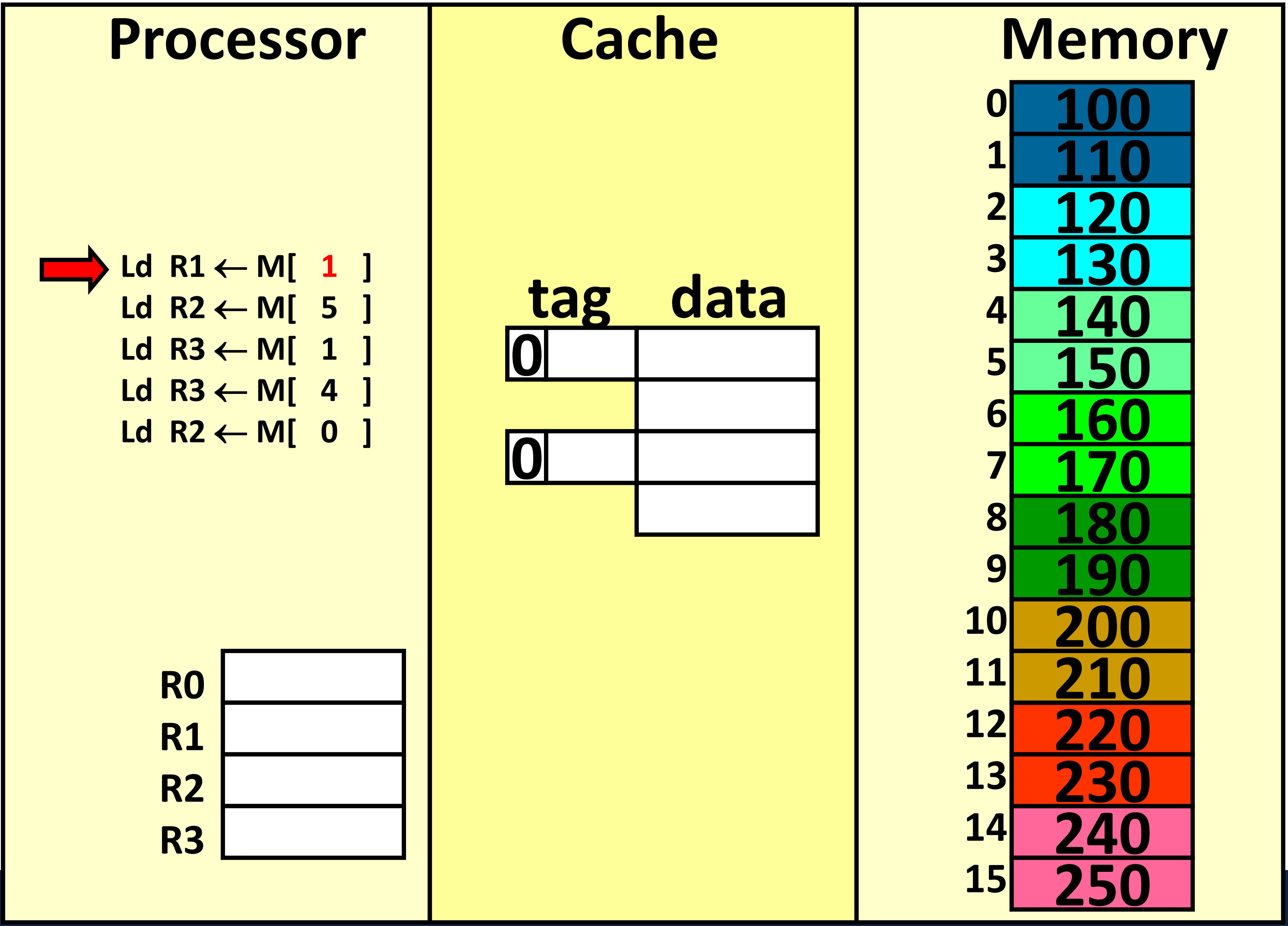
3-bit tag field

2-byte block

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

Block size for caches



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[4]
Ld R2 ← M[0]

R0	
R1	
R2	
R3	

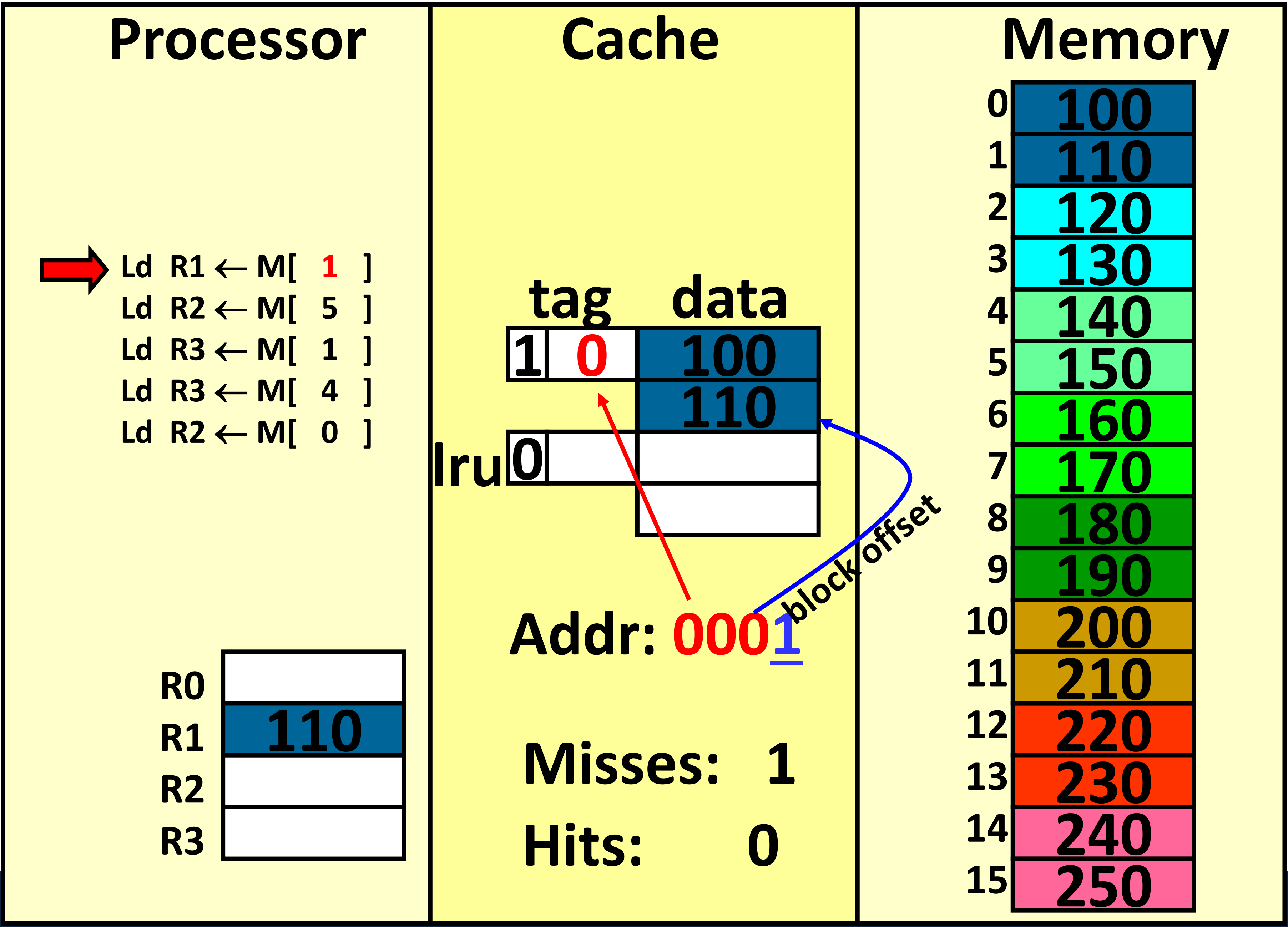
Cache

	tag	data
0		
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

Block size for caches



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[4]
Ld R2 ← M[0]

R0	
R1	110
R2	
R3	

Cache

	tag	data
1	0	100
		110
0		

Addr: 0001

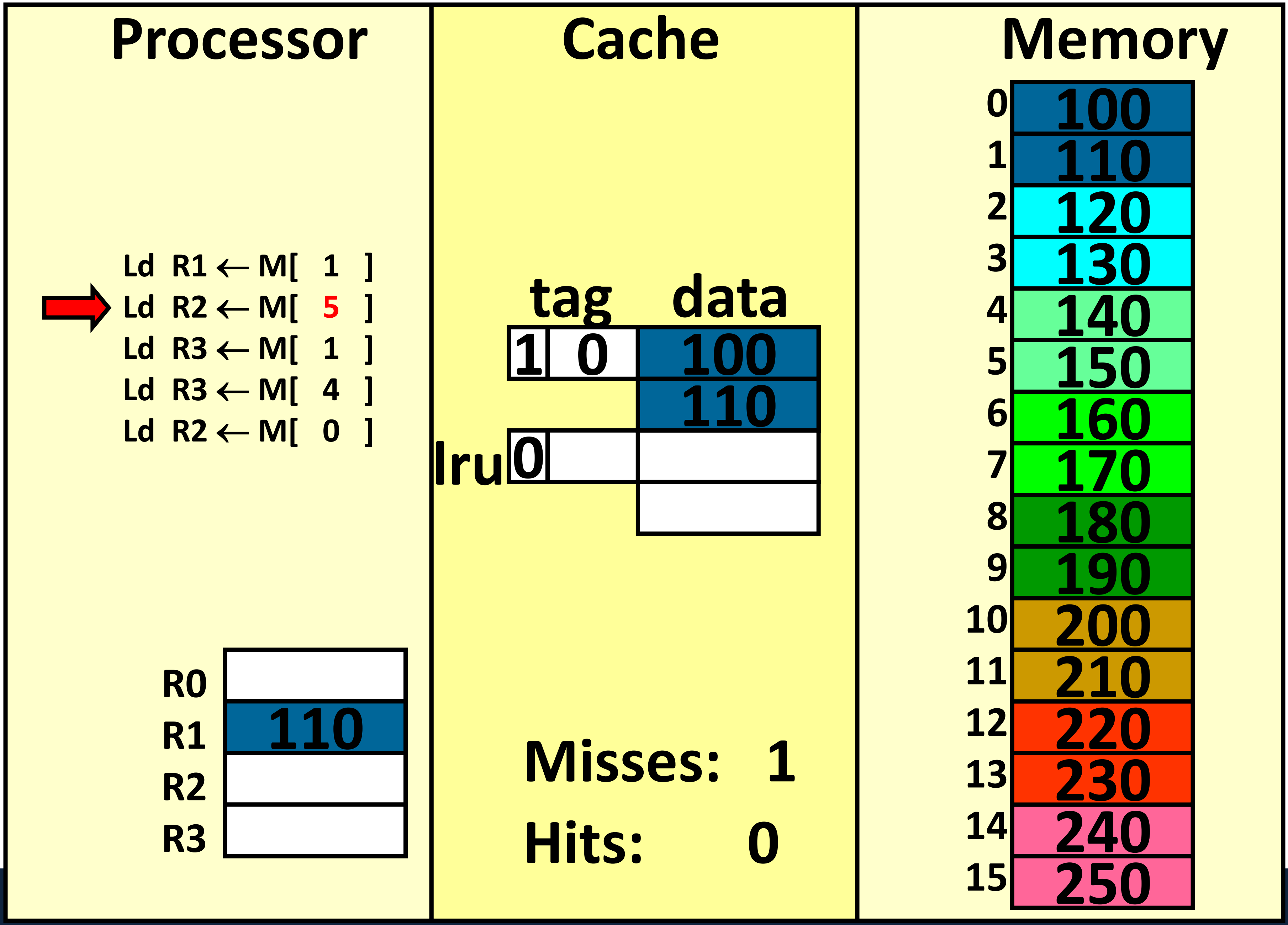
Misses: 1

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

Block size for caches



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[4]
Ld R2 ← M[0]

R0	
R1	110
R2	
R3	

Cache

		tag	data
	1	0	100
			110
lru	0		

Misses: 1

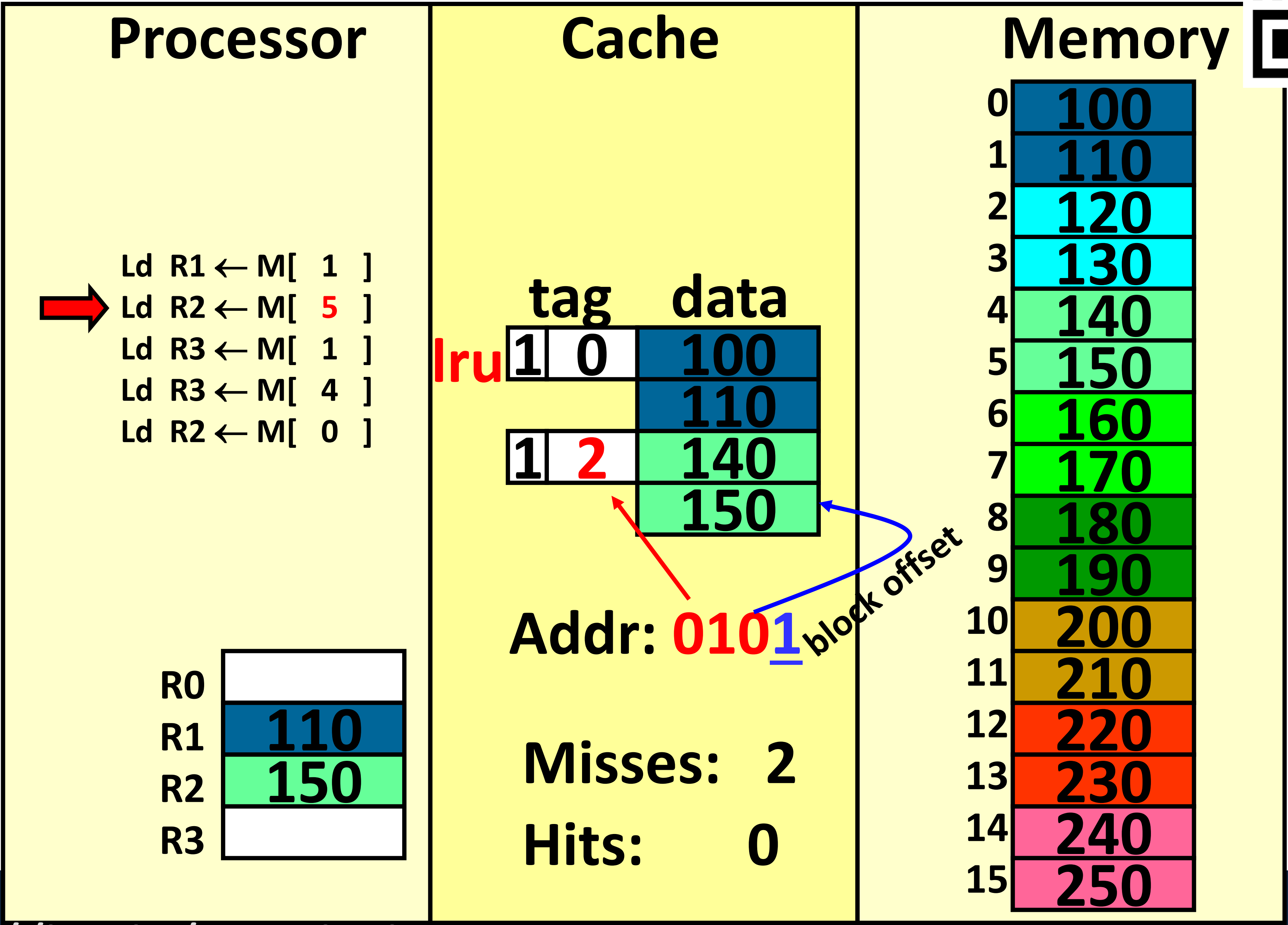
Hits: 0

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

Block size for caches

Poll: Complete the last 3 instructions yourself



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[4]
Ld R2 ← M[0]

R0	
R1	110
R2	150
R3	

Cache

		tag	data
lru	1	0	100
			110
	1	2	140
			150

Addr: 0101 block offset

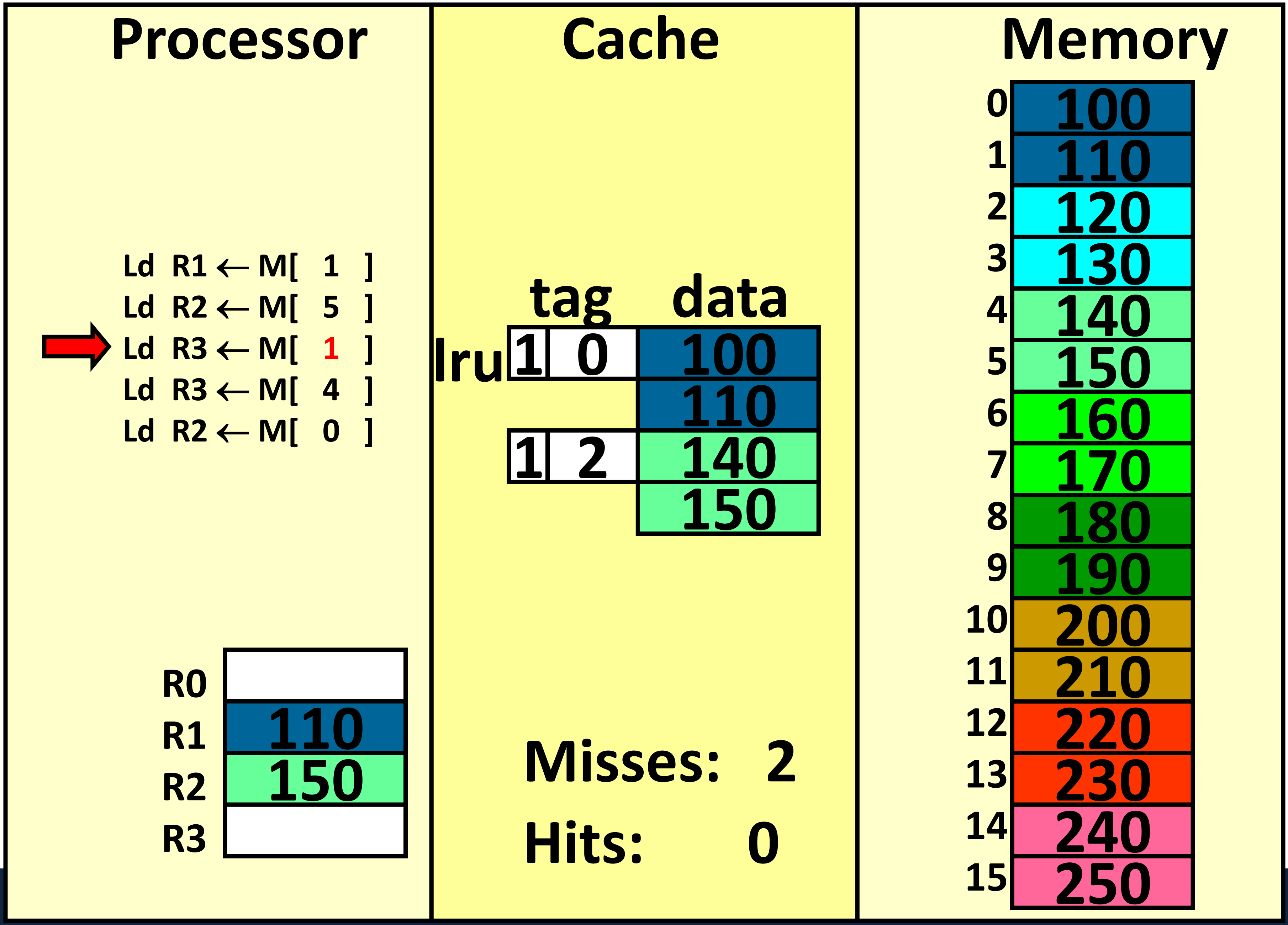
Misses: 2

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

Block size for caches



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  4  ]
Ld  R2 ← M[  0  ]
```

R0	
R1	110
R2	150
R3	

Cache

lru	tag	data
1	0	100
		110
1	2	140
		150

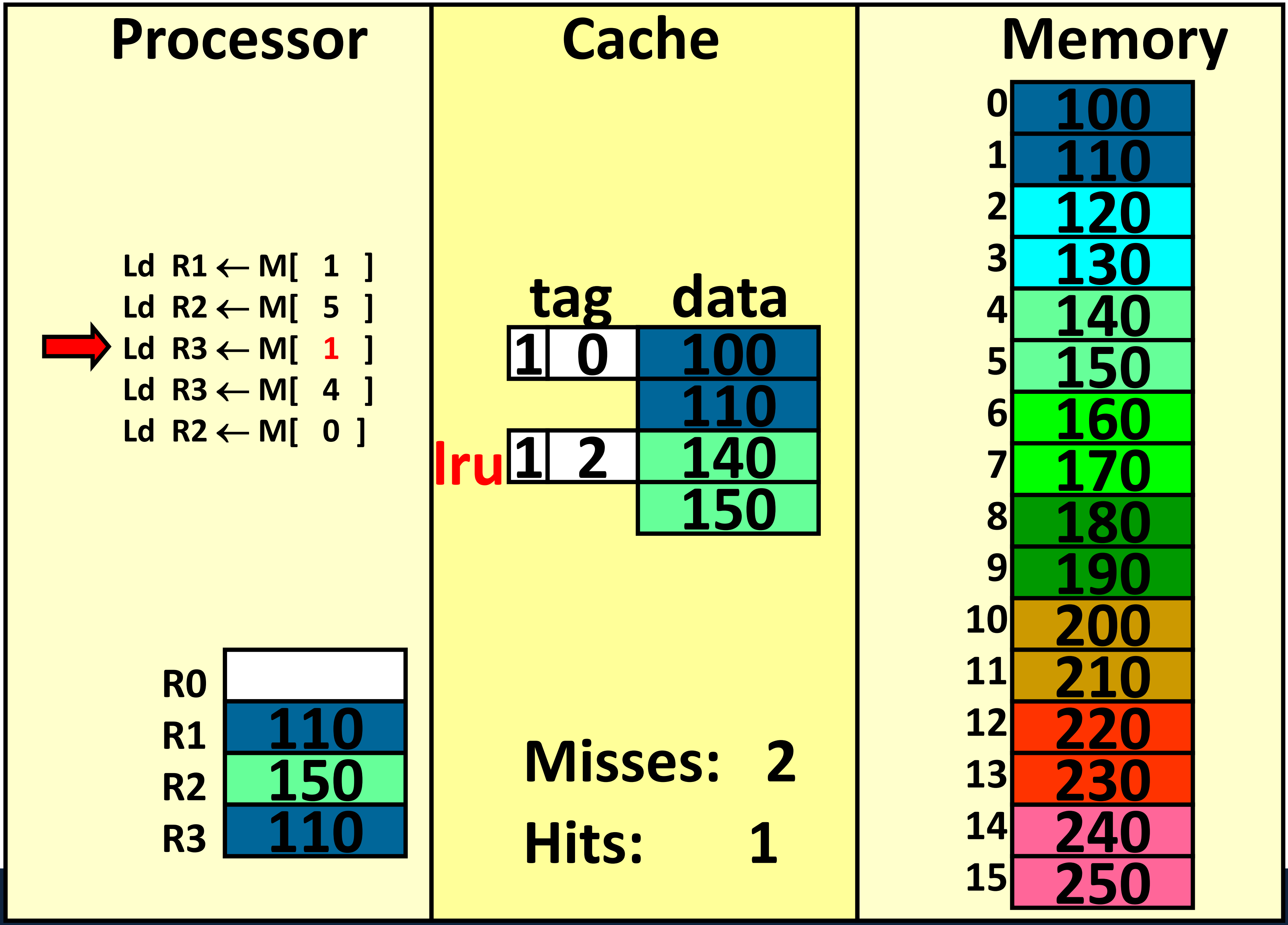
Misses: 2

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

Block size for caches



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  4  ]
Ld  R2 ← M[  0  ]
```

R0	
R1	110
R2	150
R3	110

Cache

		tag	data
	1	0	100
			110
lru	1	2	140
			150

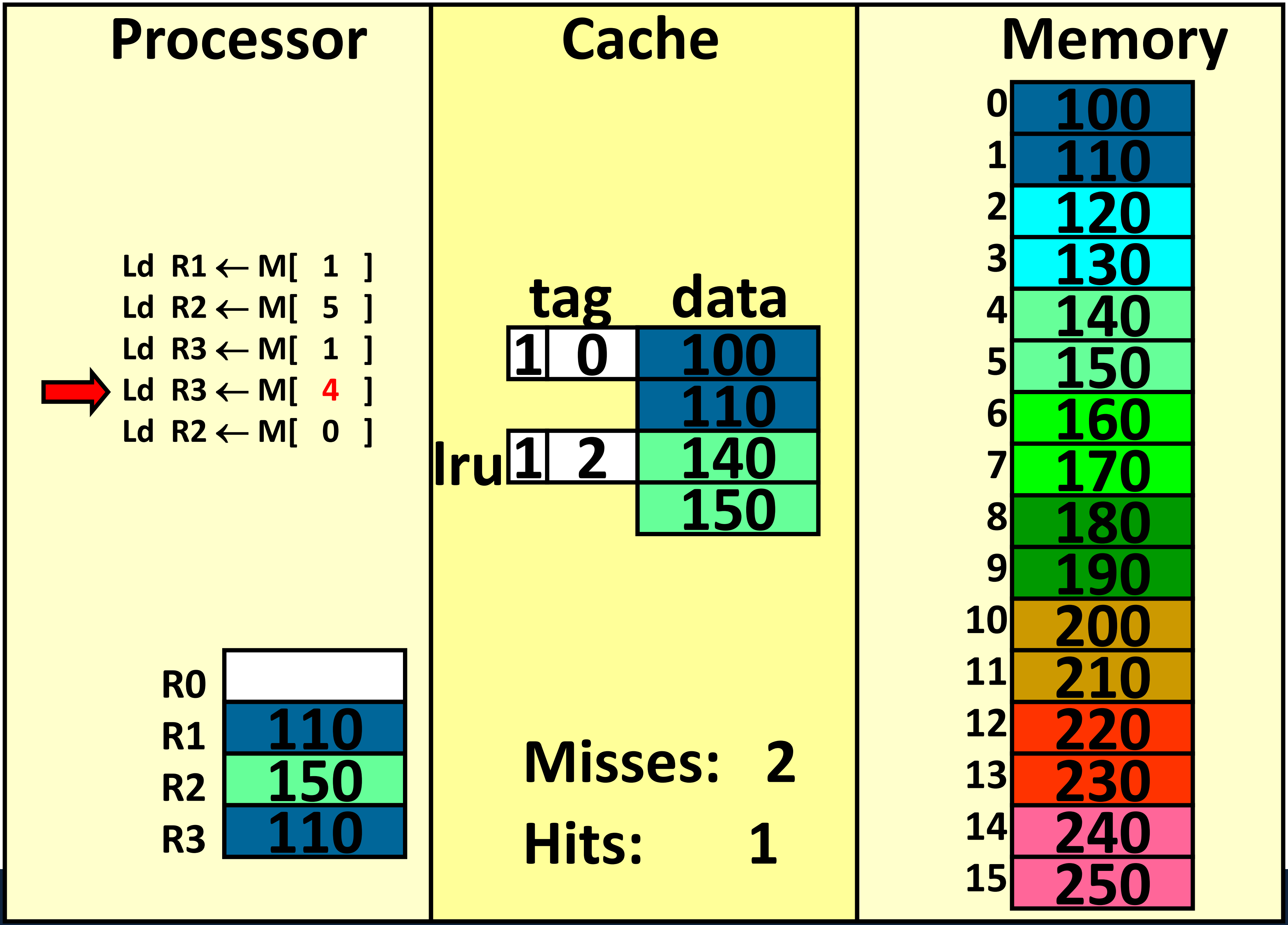
Misses: 2

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

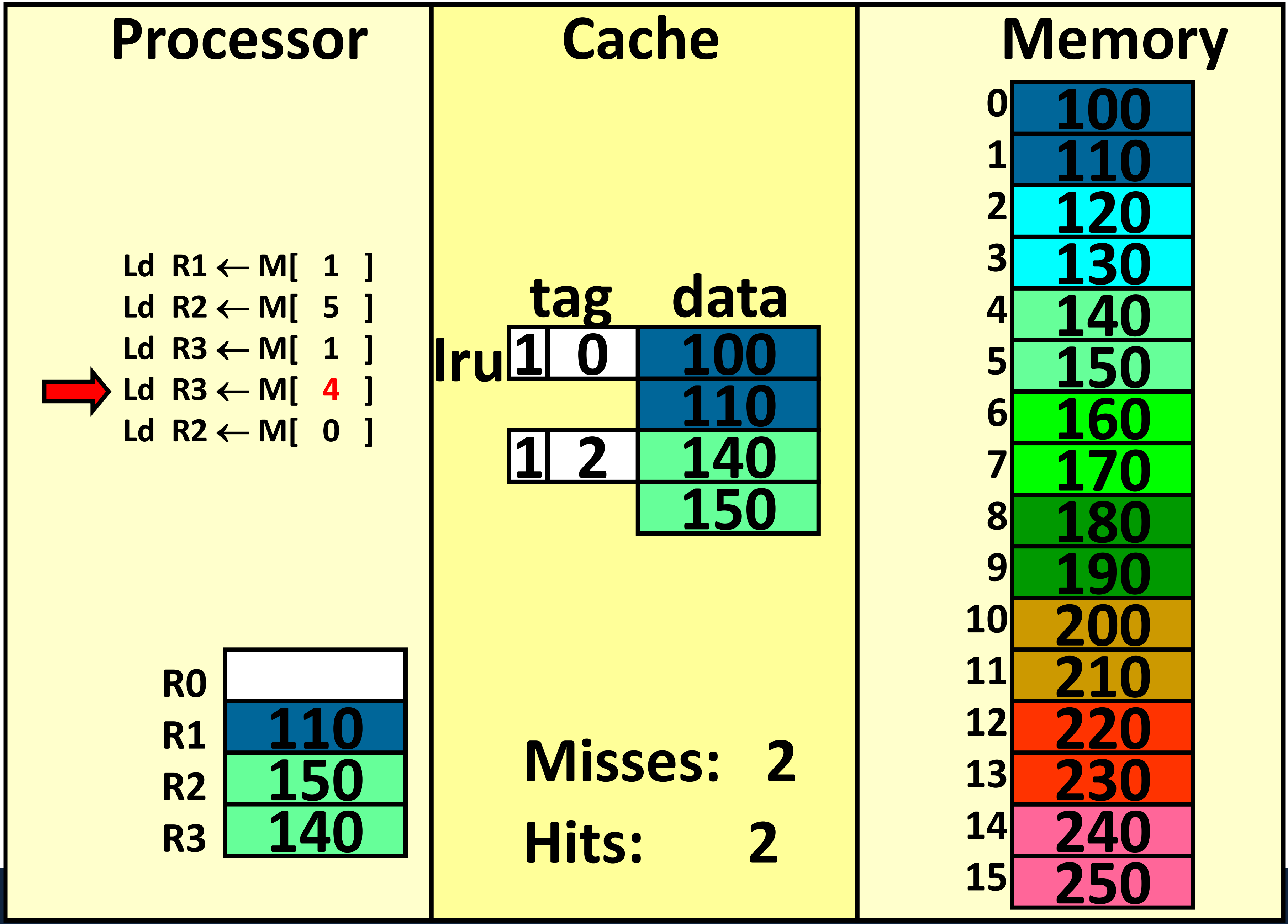
Block size for caches



Misses: 2

Hits: 1

Block size for caches



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  4  ]
Ld  R2 ← M[  0  ]
```

R0	
R1	110
R2	150
R3	140

Cache

	lru	tag	data
lru	1	0	100
			110
	1	2	140
			150

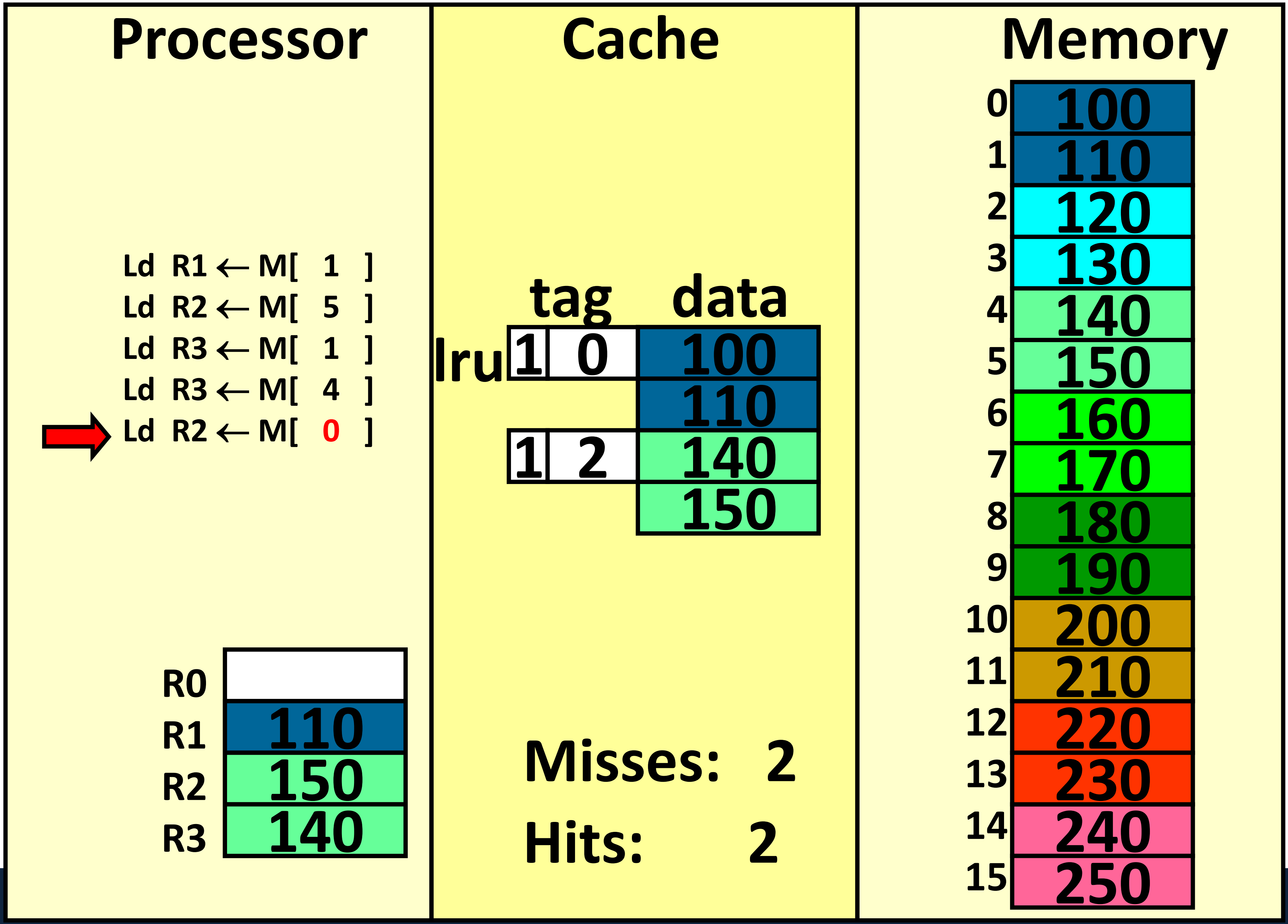
Misses: 2

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

Block size for caches



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[4]
Ld R2 ← M[0]

R0	
R1	110
R2	150
R3	140

Cache

lru	tag	data
1	0	100
		110
1	2	140
		150

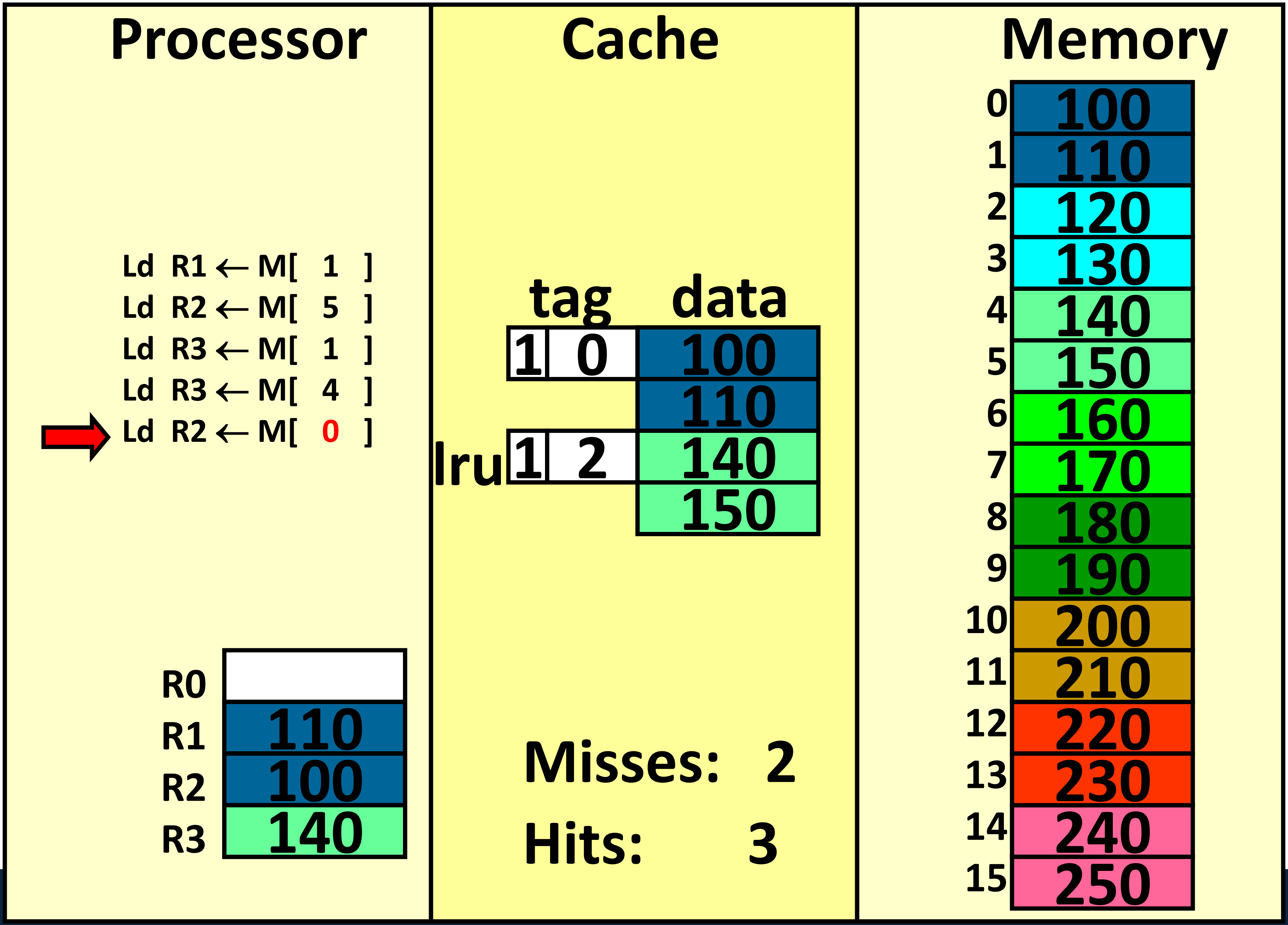
Misses: 2

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

Block size for caches



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  4  ]
Ld  R2 ← M[  0  ]
```

R0	
R1	110
R2	100
R3	140

Cache

		tag	data
	1	0	100
			110
lru	1	2	140
			150

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

Spatial Locality

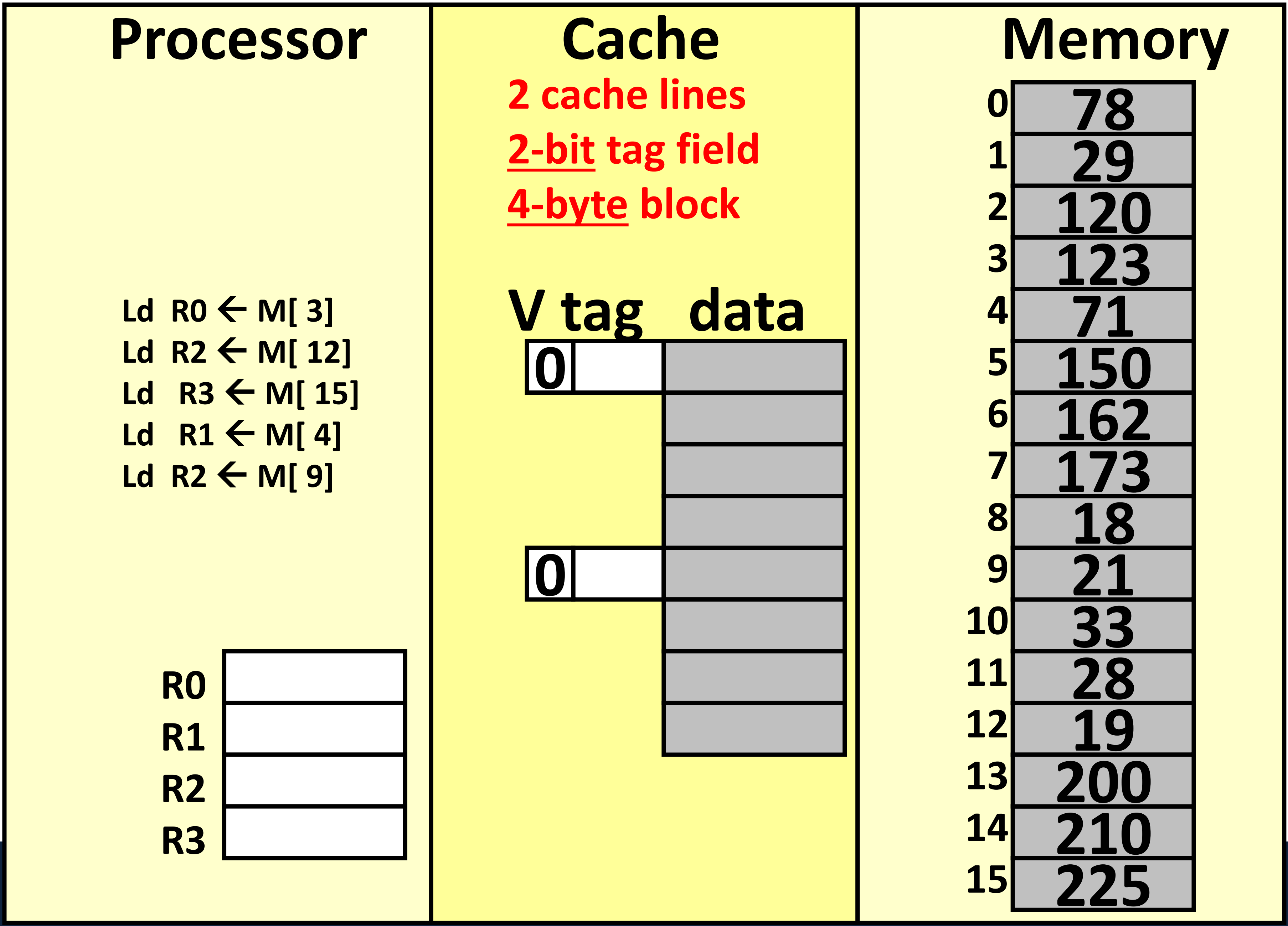
- Notice that when we accessed address 1, we also brought in address 0
    - This turned out to be a good thing, since we later referenced address 0 and found it in the cache
- This is taking advantage of **spatial locality:**
    - If we access a memory location (e.g. 1000), we are more likely to access a location near it (e.g. 1001) than some random location
    - Arrays and structs are a big reason for this

```
for(i=0; i< N; i++)
  for(j = 0; j < N; j++ )
  {
      count++;
      arrayInt[i][j] = 10;
  }
```

Agenda

- Larger Cache Blocks
- **Extra Problems**
- LRU with More than Two Blocks
- Write-Through Cache
- Write-Back Cache

Extra Practice Problem



Processor

Ld R0 ← M[3]
Ld R2 ← M[12]
Ld R3 ← M[15]
Ld R1 ← M[4]
Ld R2 ← M[9]

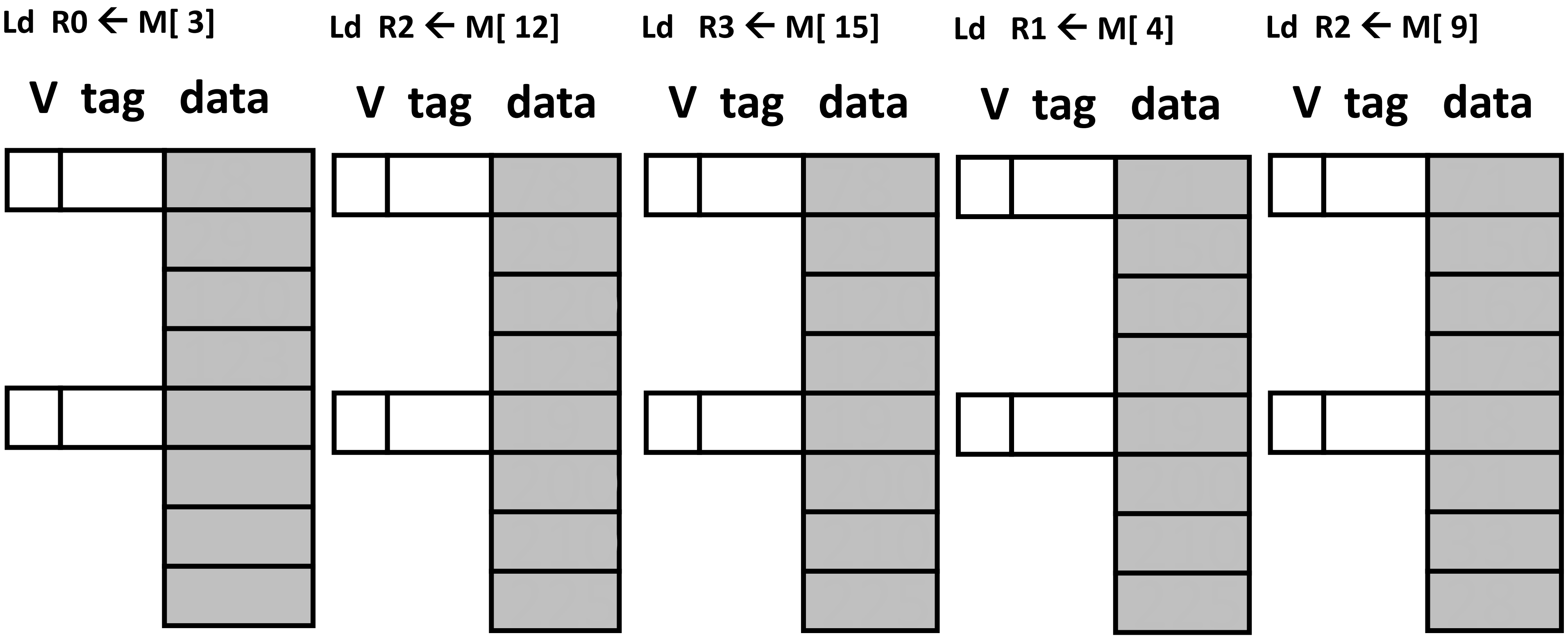
Cache

2 cache lines
2-bit tag field
4-byte block

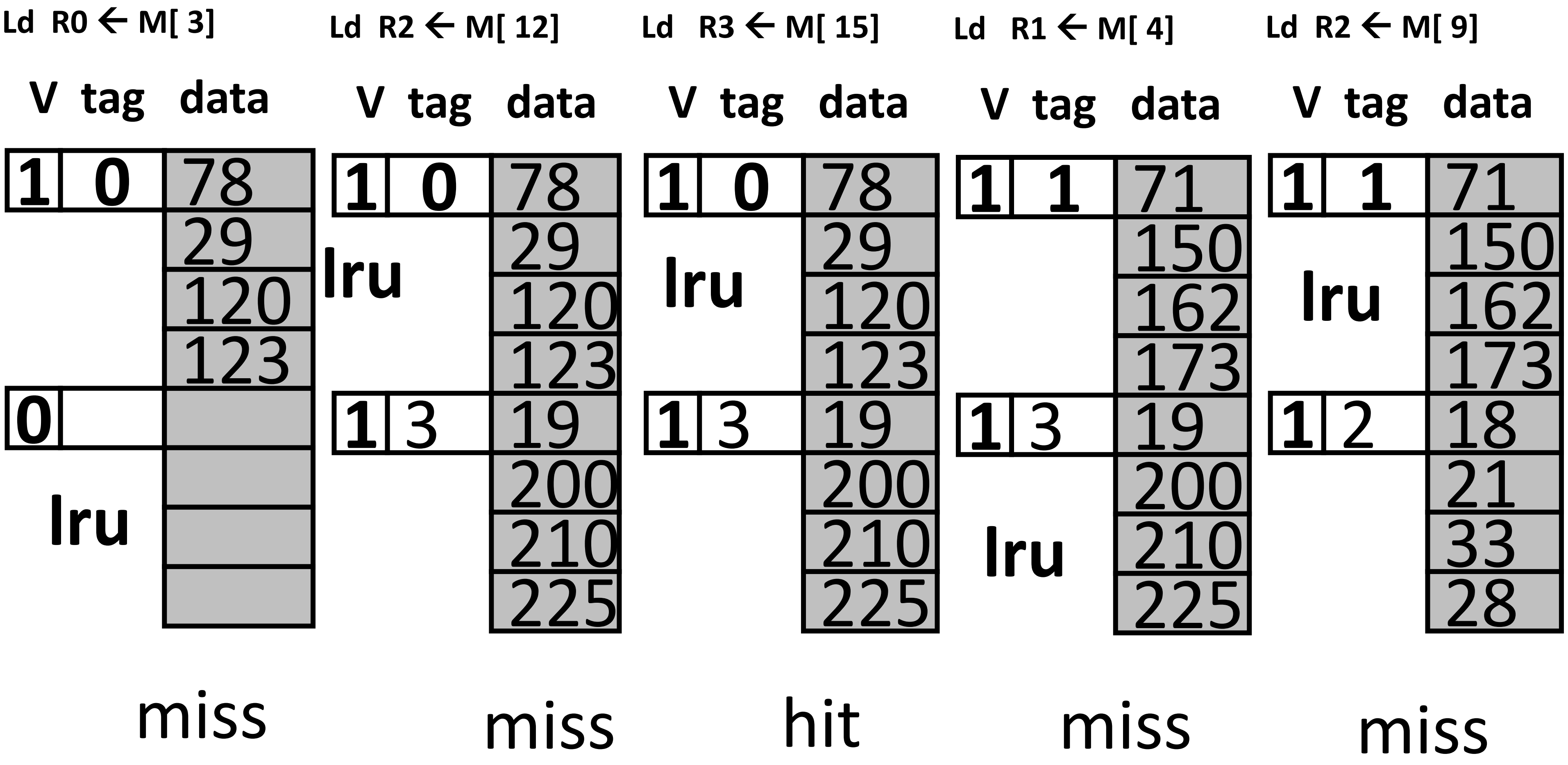
Memory

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

Solution to Practice Problem



Solution to Practice Problem



Ld R0 ← M[3]

V	tag	data
1	0	78
		29
		120
		123
0		
lru		

miss

Ld R2 ← M[12]

V	tag	data
1	0	78
lru		29
		120
		123
1	3	19
		200
		210
		225

miss

Ld R3 ← M[15]

V	tag	data
1	0	78
lru		29
		120
		123
1	3	19
		200
		210
		225

hit

Ld R1 ← M[4]

V	tag	data
1	1	71
		150
		162
		173
1	3	19
		200
lru		210
		225

miss

Ld R2 ← M[9]

V	tag	data
1	1	71
		150
lru		162
		173
1	2	18
		21
		33
		28

miss

Extra Class Problem

*We'll see later that this is called a "fully-associative cache"

- Given a cache that works as we've described* with the following configuration: total size is 8 bytes, block size is 2 bytes, LRU replacement. The memory address size is 16 bits and is byte addressable.
  1. How many bits are for each tag? How many blocks in the cache?
  2. For the following reference stream, indicate whether each reference is a hit or miss: 0, 1, 3, 5, 12, 1, 2, 9, 4
  3. What is the hit rate?
  4. How many bits are needed for storage overhead for each block?

Agenda

- Larger Cache Blocks
- Extra Problems
- **LRU with More than Two Blocks**
- Write-Through Cache
- Write-Back Cache

LRU with more than 2 entries

- If we have more than 2 things we're keeping track of…
  - Can't **just** track LRU
  - Once we access that element, how do we know which of the other elements are LRU?
  - Must track the *full ordering* of when elements were accessed*
- Each element must store a number [0-(N-1)] -> $\log_2(N)$ bits
- 0 is LRU, 1 is 2nd LRU… N-1 is most recently used

LRU with more than 2 entries

- If we have more than 2 things we're keeping track of…
  - Can't **just** track LRU
  - Once we access that element, how do we know which of the other elements are LRU?
  - Must track the *full ordering* of when elements were accessed*
- Each element must store a number [0-(N-1)] -> $\log_2(N)$ bits
- 0 is LRU, 1 is 2nd LRU… N-1 is most recently used
- When element i is used:

```
X = counter[i]
counter[i] = N-1
for (j=0  to N-1)
    if ((j != i) AND (counter[j]>X)) counter[j]—
```

Initial State				
Element	0	1	2	3
Count	0	1	2	3

LRU with more than 2 entries

- If we have more than 2 things we're keeping track of…
    - Can't **just** track LRU
    - Once we access that element, how do we know which of the other elements are LRU?
    - Must track the *full ordering* of when elements were accessed*
- Each element must store a number [0-(N-1)] -> $\log_2(N)$ bits
- 0 is LRU, 1 is 2nd LRU… N-1 is most recently used
- When element i is used:

```
X = counter[i]
counter[i] = N-1
for (j=0  to N-1)
        if ((j != i) AND (counter[j]>X)) counter[j]—
```

Initial State				
Element	0	1	2	3
Count	0	1	2	3

Access Element 2				
Element	0	1	2	3
Count	0	1	3	2

LRU with more than 2 entries

- If we have more than 2 things we're keeping track of…
  - Can't **just** track LRU
  - Once we access that element, how do we know which of the other elements are LRU?
  - Must track the *full ordering* of when elements were accessed*
- Each element must store a number [0-(N-1)] -> $\log_2(N)$ bits
- 0 is LRU, 1 is 2nd LRU… N-1 is most recently used
- When element i is used:

```
X = counter[i]
counter[i] = N-1
for (j=0  to N-1)
        if ((j != i) AND (counter[j]>X)) counter[j]—
```

- Evict element with counter = 0 when needed
- Get's expensive for moderate to large N

Initial State				
Element	0	1	2	3
Count	0	1	2	3

Access Element 2				
Element	0	1	2	3
Count	0	1	3	2

Access Element 0				
Element	0	1	2	3
Count	3	0	2	1