EECS 370 - Lecture 18

Cache Blocks, Writeback, and Associativity

Announcements

- P3 published
  - Checkpoint due **today**
    - 5% of project – have pipeline working without data hazards or branches
- Midterm scores published **soon**
- HW 4 dueMonday

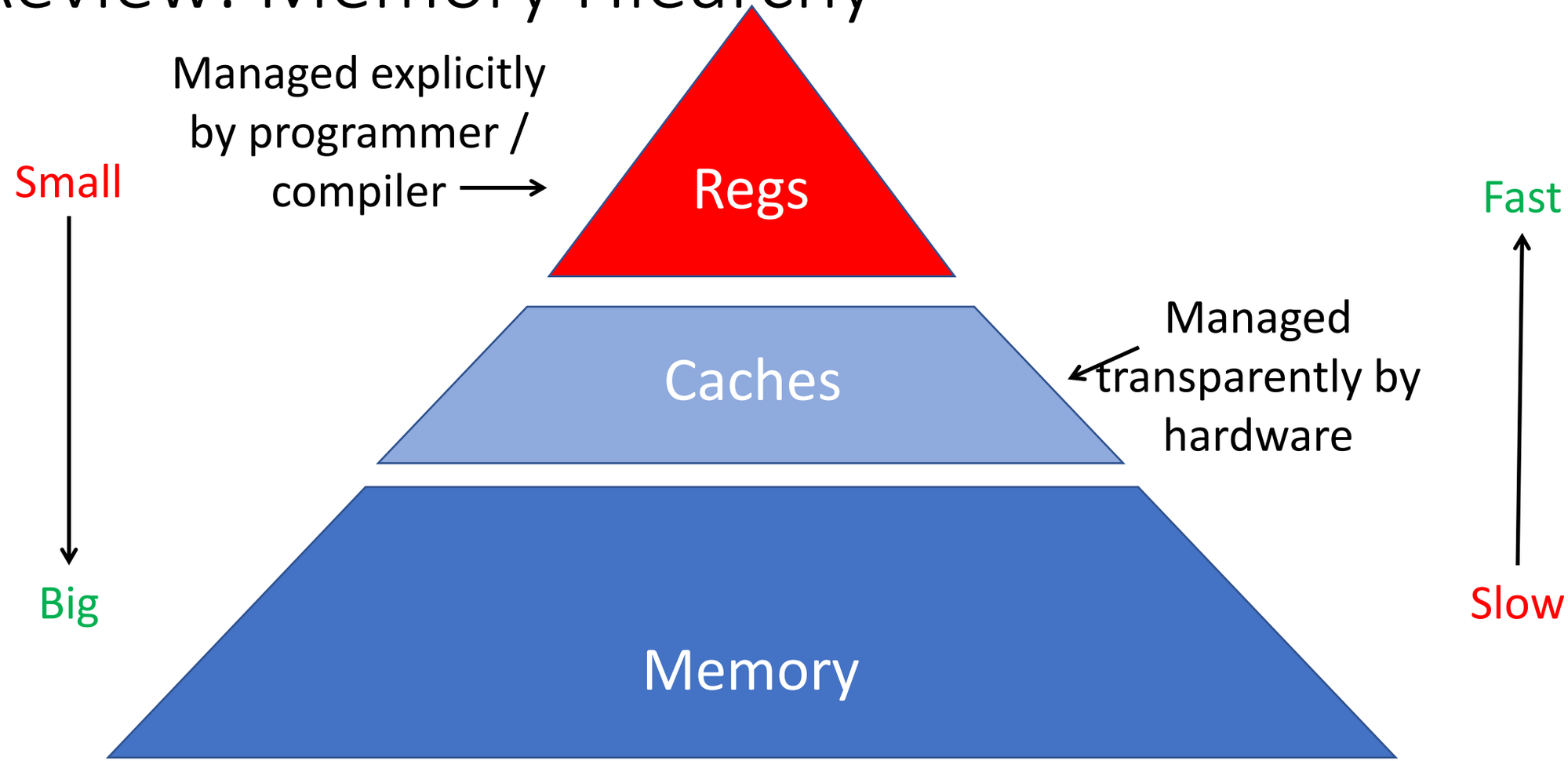
Resources

- Want extra examples with pipelining? Try playing with the "Pipeline Simulator" under "Resources" on the website
  - https://vhosts.eecs.umich.edu/370simulators/pipeline/simulator.html
  - Several pre-written programs you can step through to understand what's going on
  - Note that the project pipeline is slightly different

Review: Memory Hierarchy

- Problem: Large memory is much slower than processor
  - We'd have to wait ~100s of cycles for every load
- Observation:
  1. Small memory can be made as fast as processor
  2. We only need a small amount of memory at a time
- Idea: let's include a small amount of memory – a **cache** – that holds data hardware thinks is likely to be needed in the near future
  - Check the cache before going to main memory

Review: Memory Hiearchy



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

Poll: What will the overhead of this cache be?

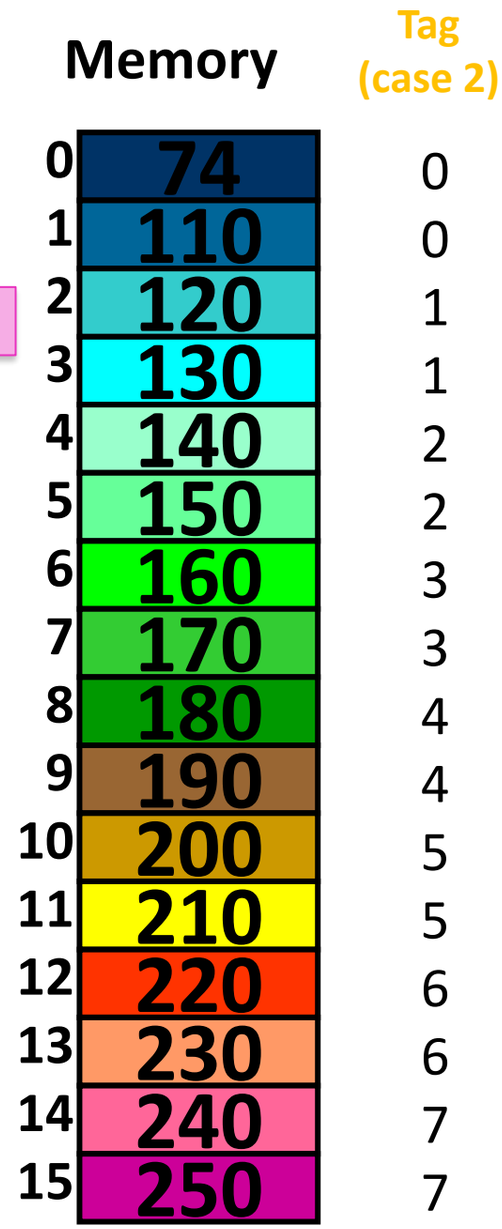
Figuring out the tag

Poll: What's the pattern?

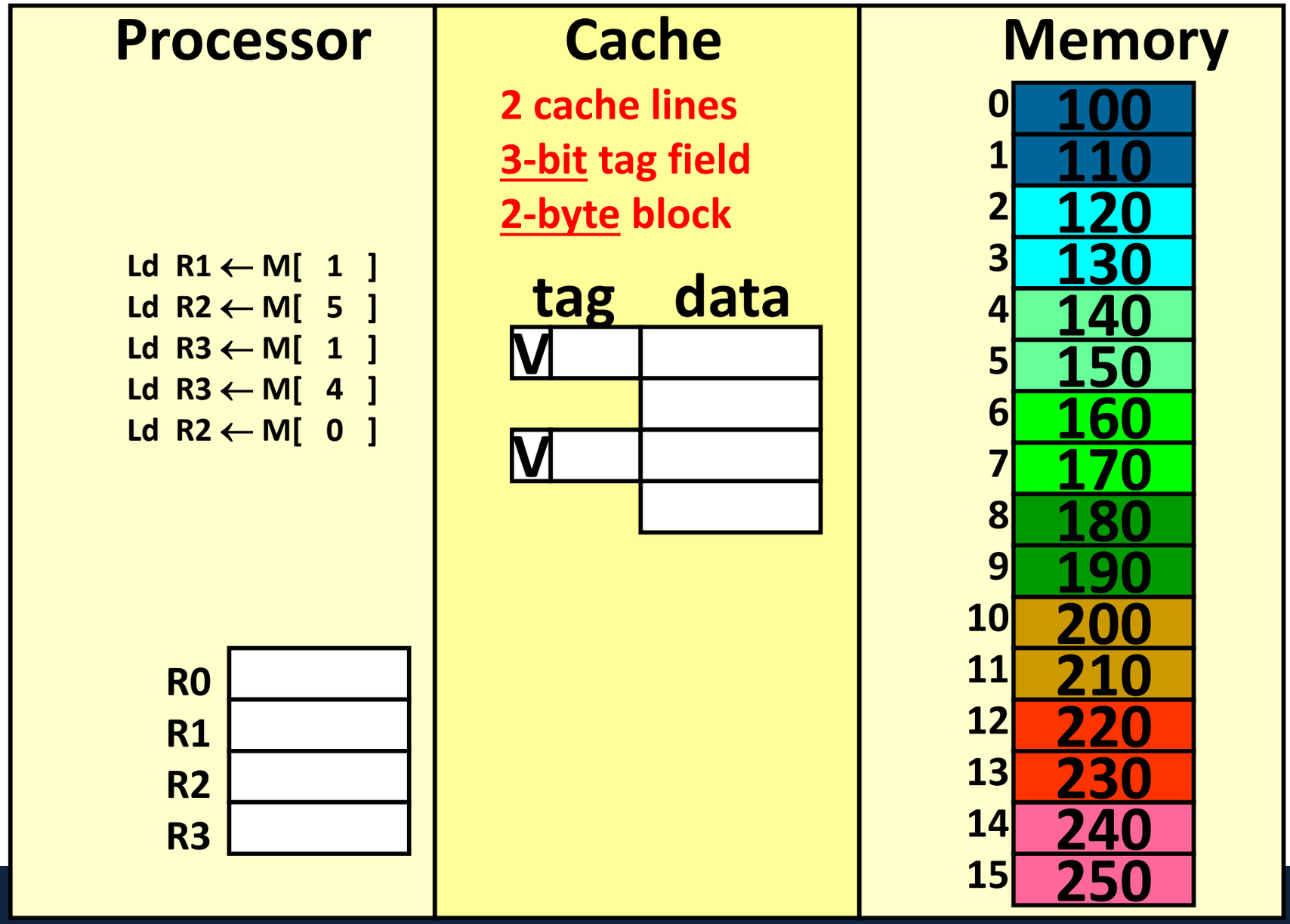
- If block size is N, what's the pattern for figuring out the tag from the address?
  - $tag = \left\lfloor \frac{addr}{block\ size} \right\rfloor$
- If block size is power of 2, then this is just everything except the $\log_2(block\ size)$ bits of the address in binary!
- E.g.

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

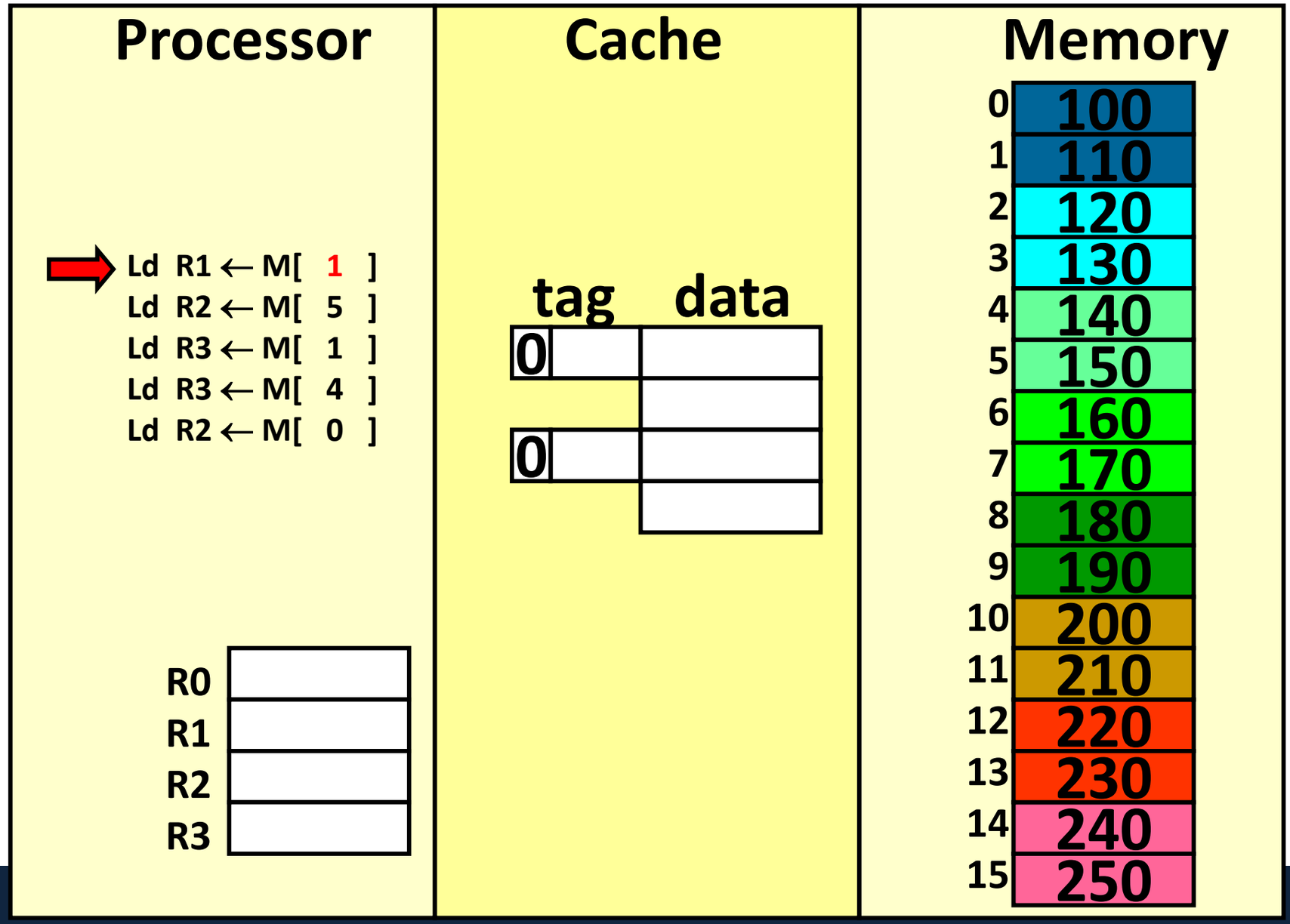
- Remaining bits (block offset) tells us how far into the block the data is



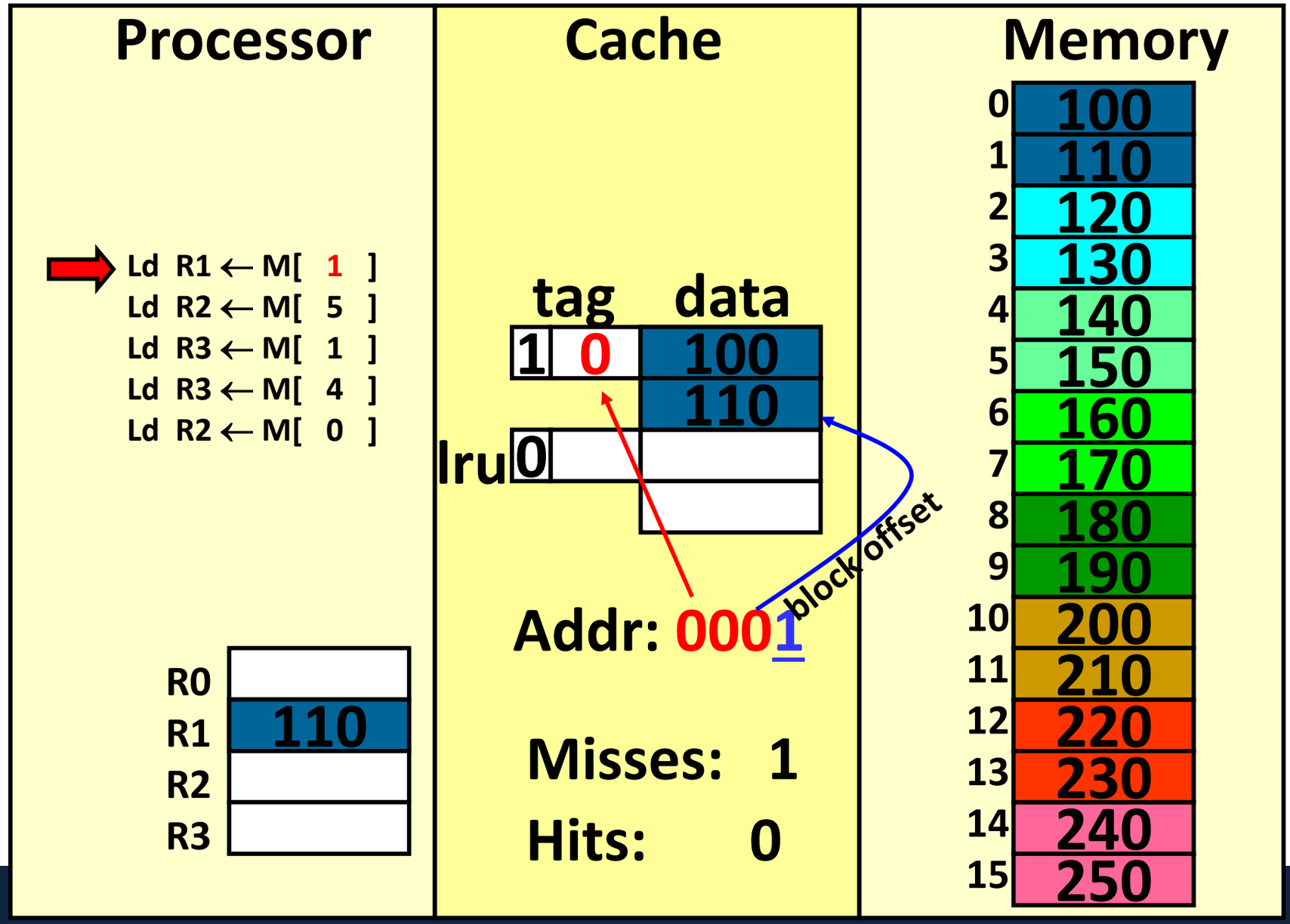
Block size for caches



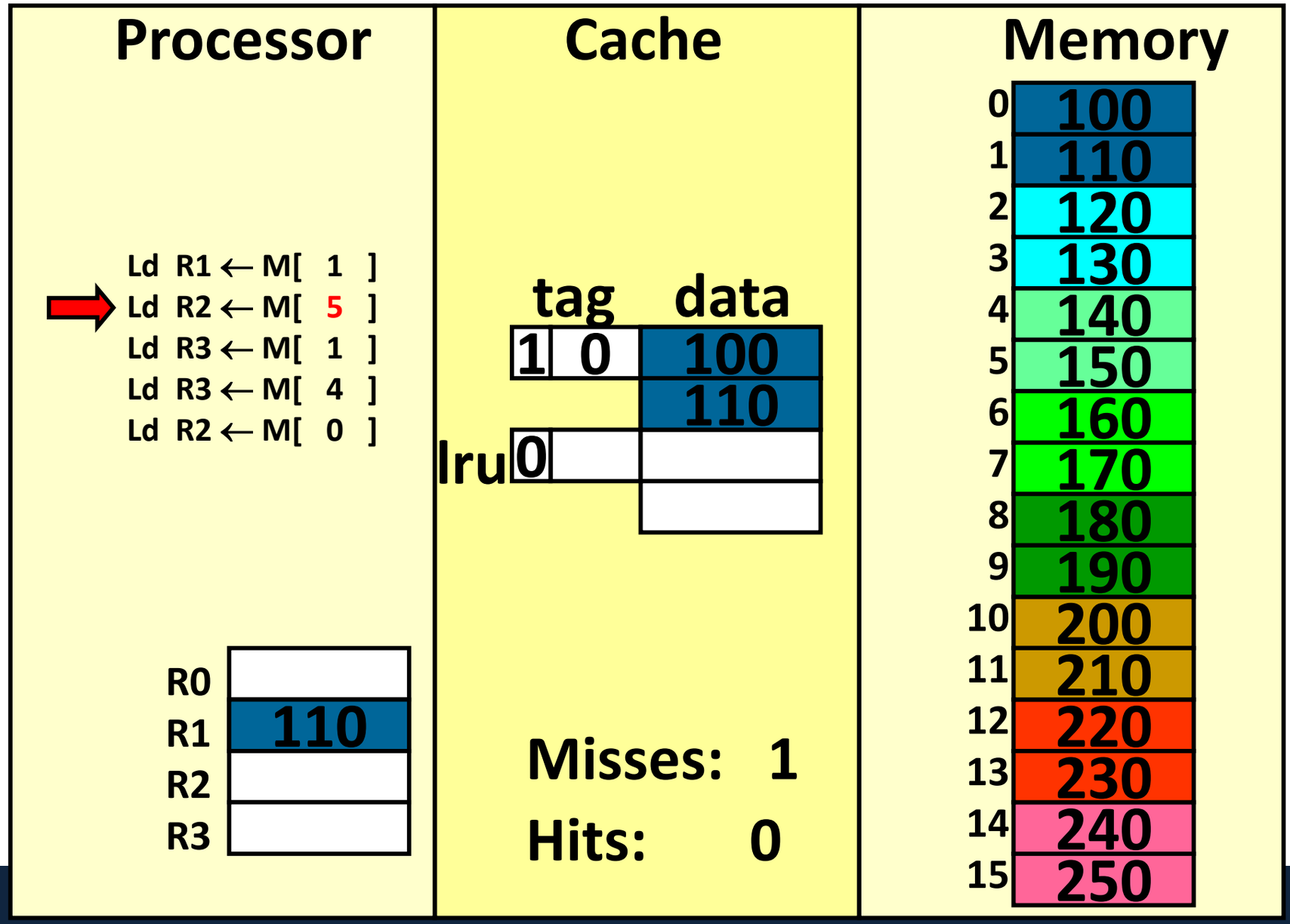
Block size for caches



Block size for caches

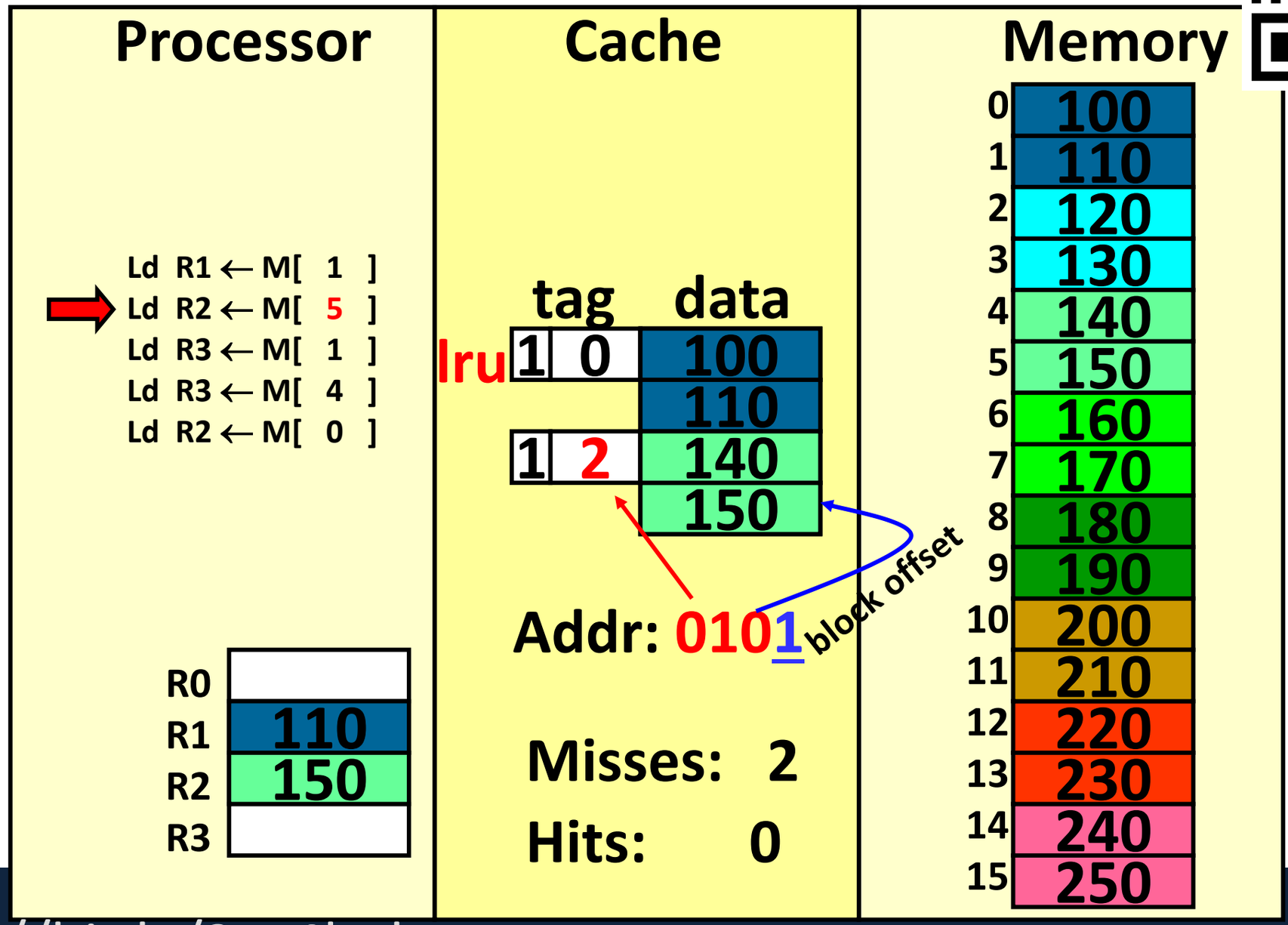


Block size for caches

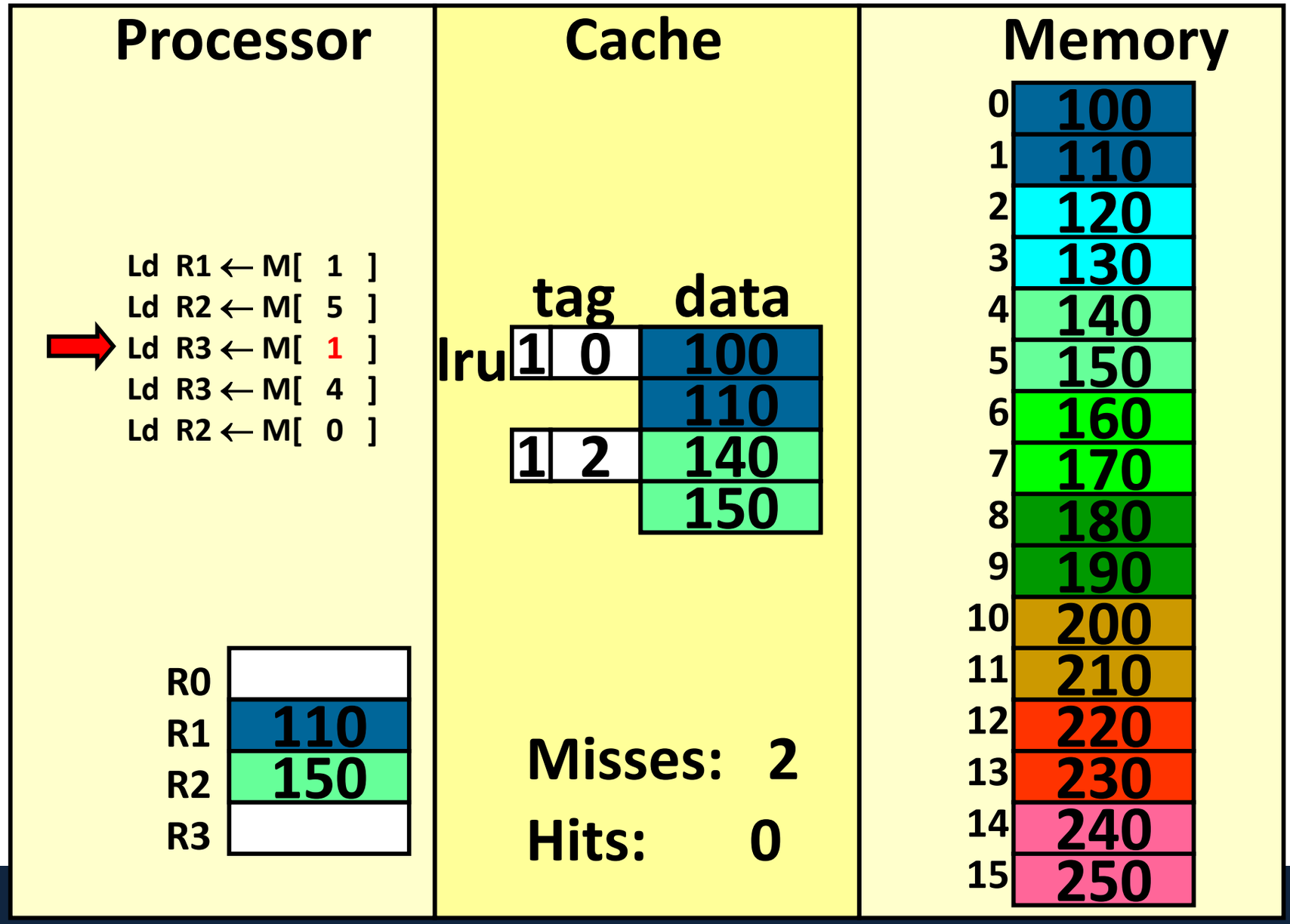


Block size for caches

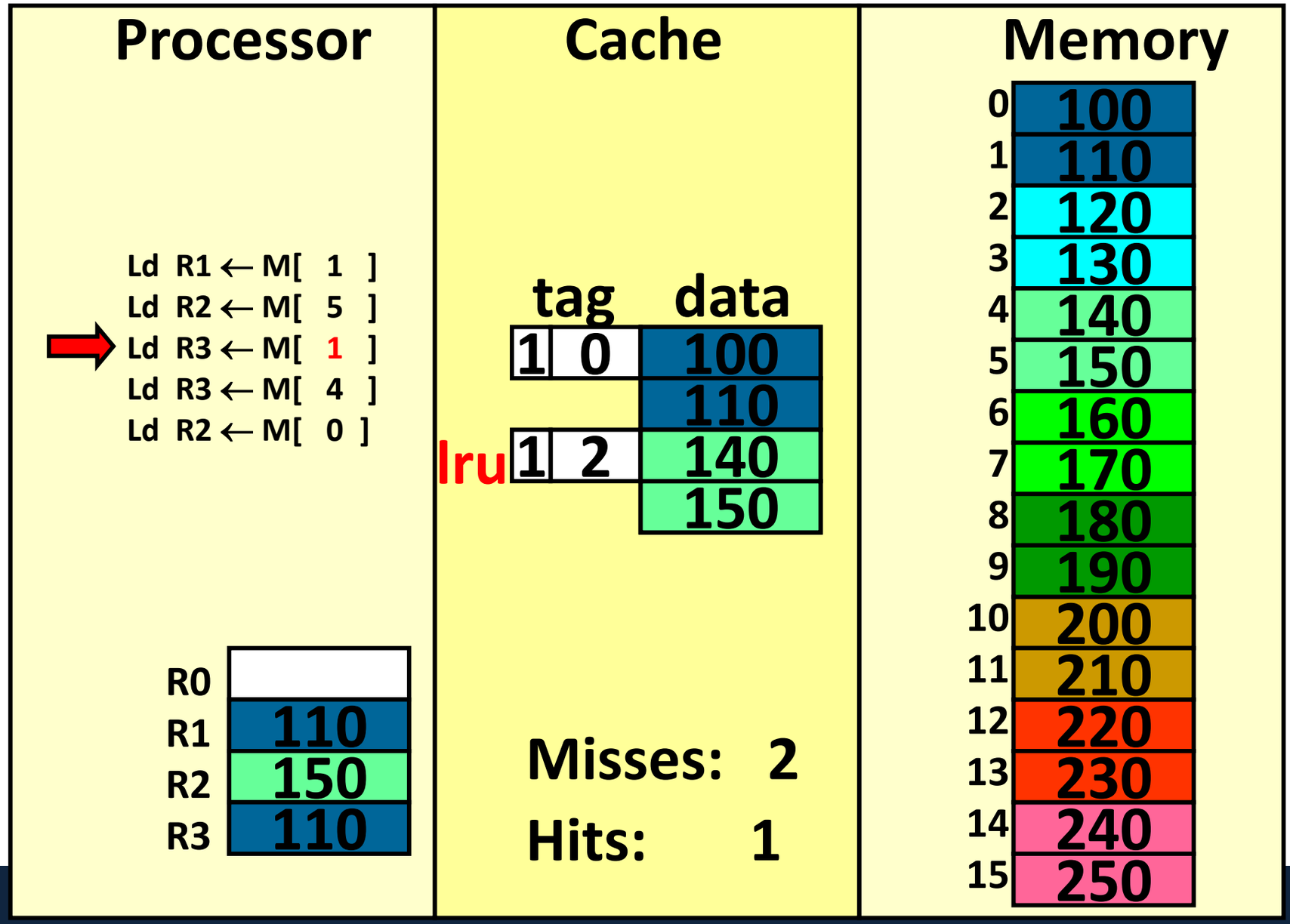
Poll: Complete the last 3 instructions yourself



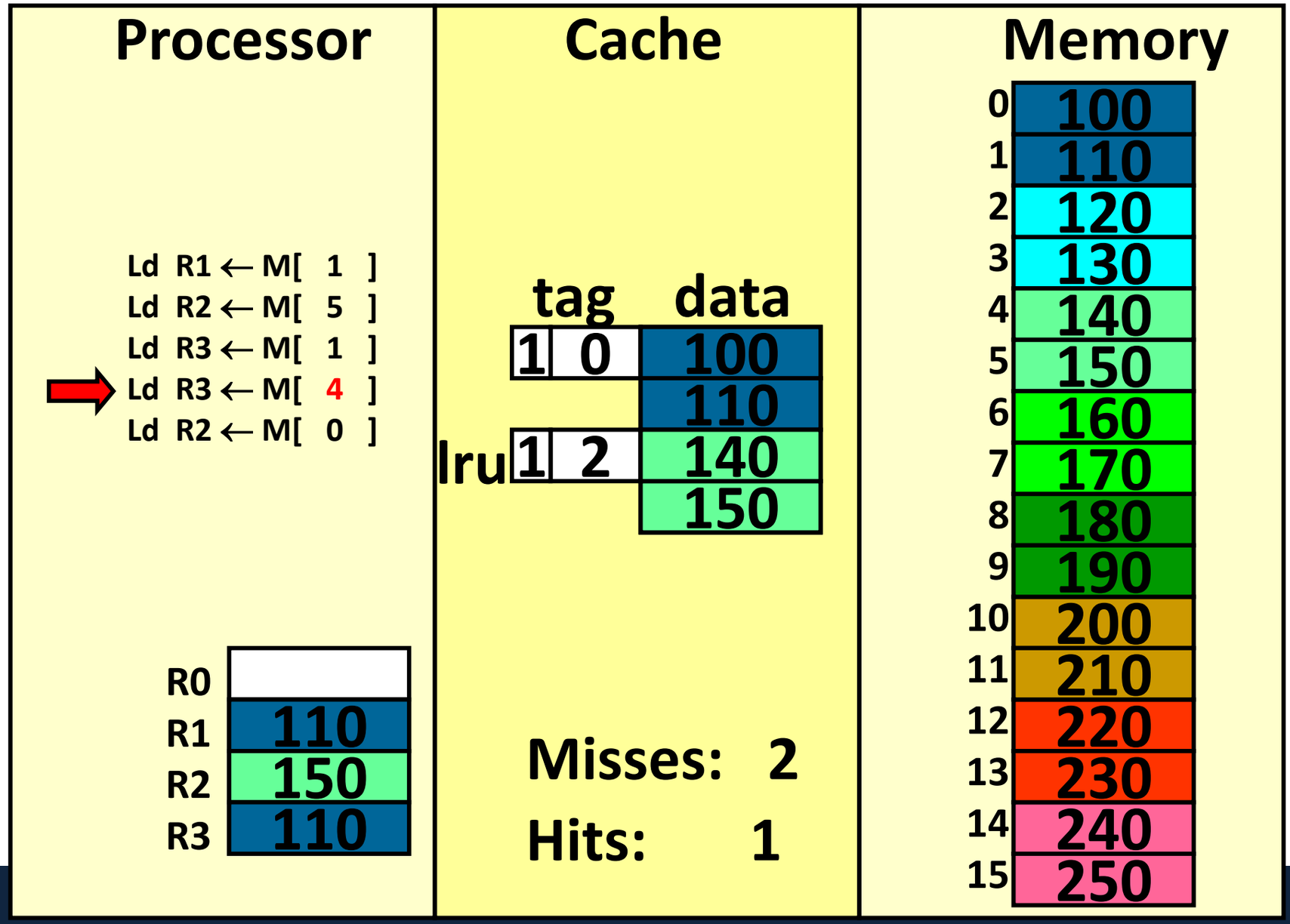
Block size for caches



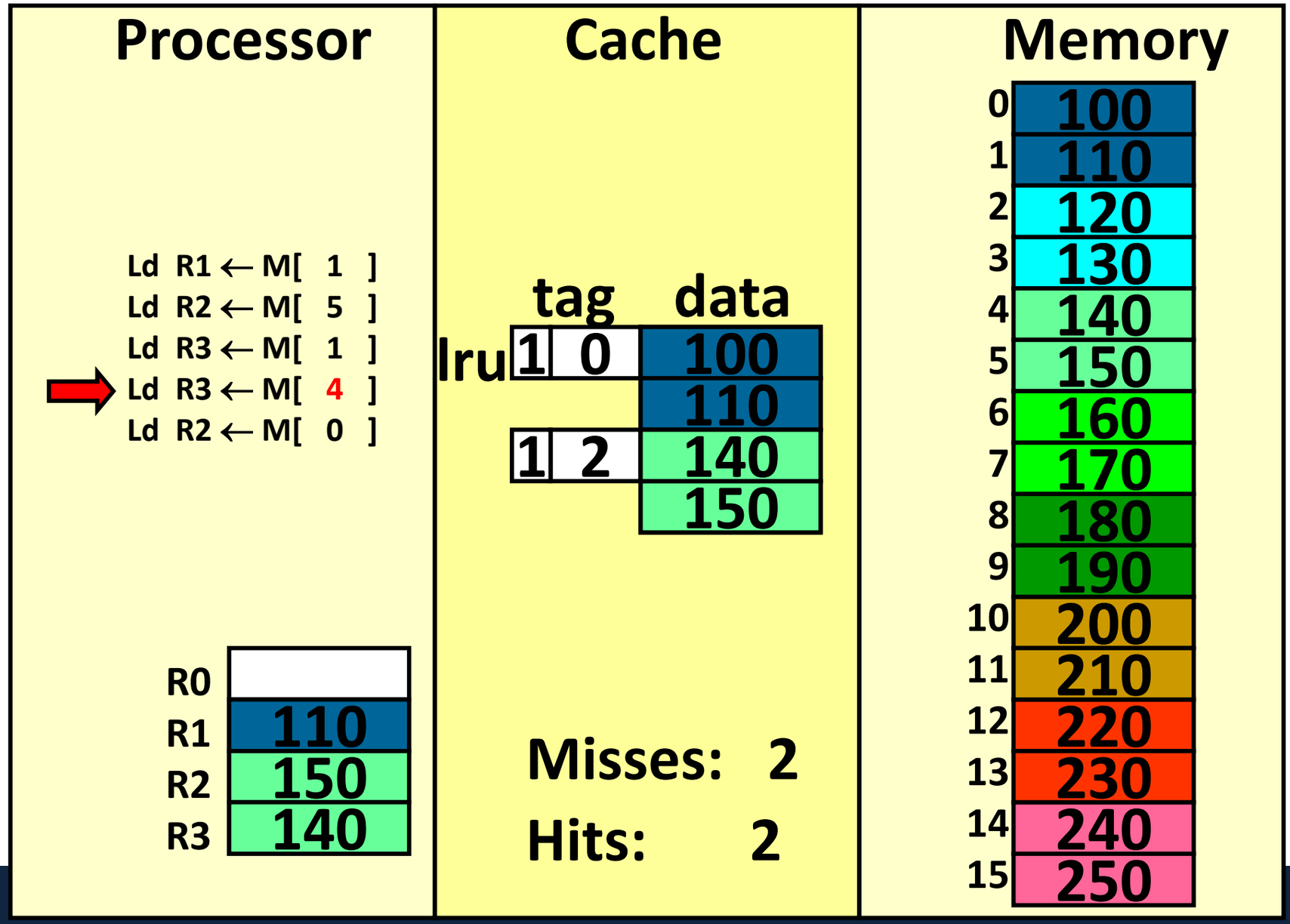
Block size for caches



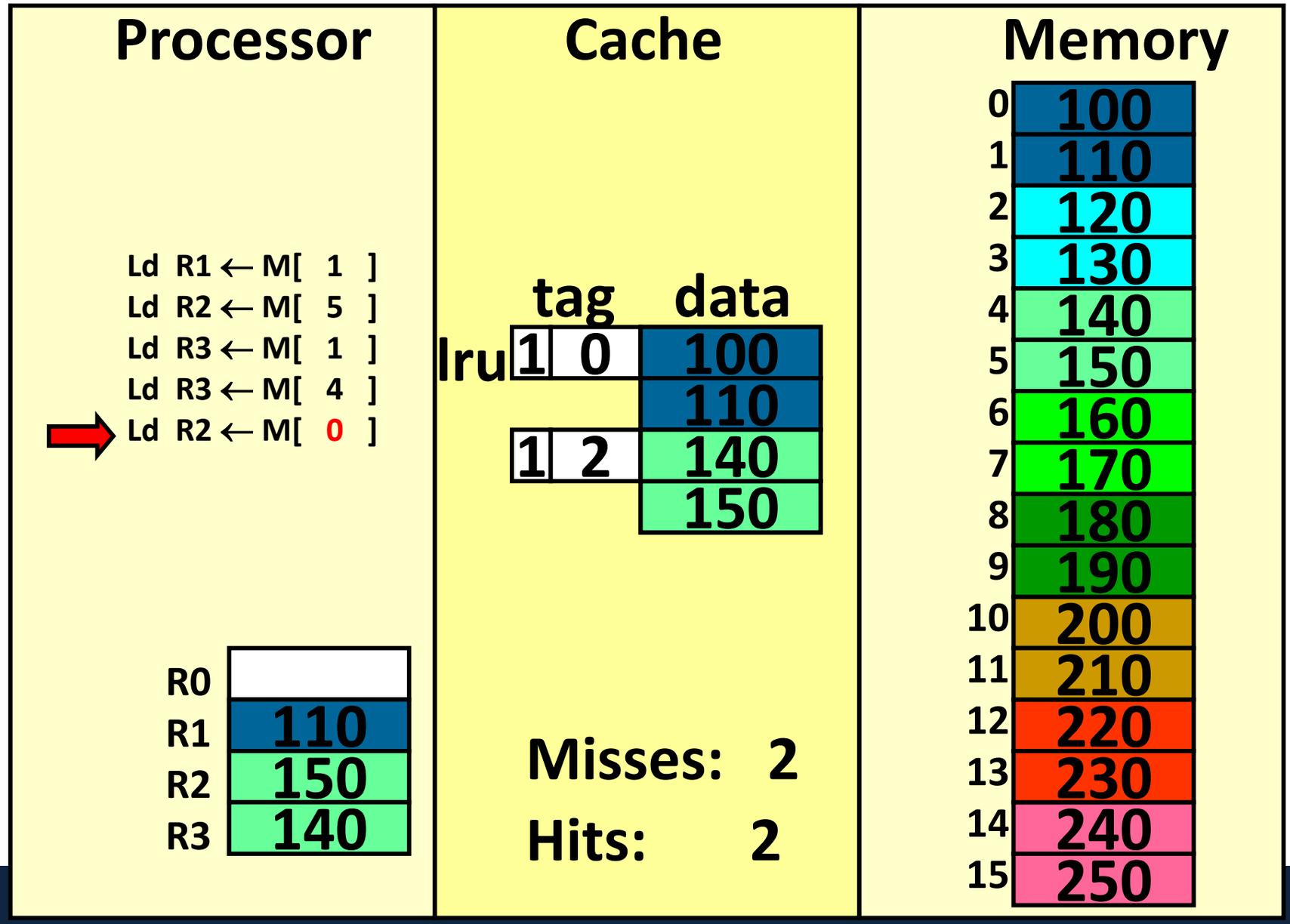
Block size for caches



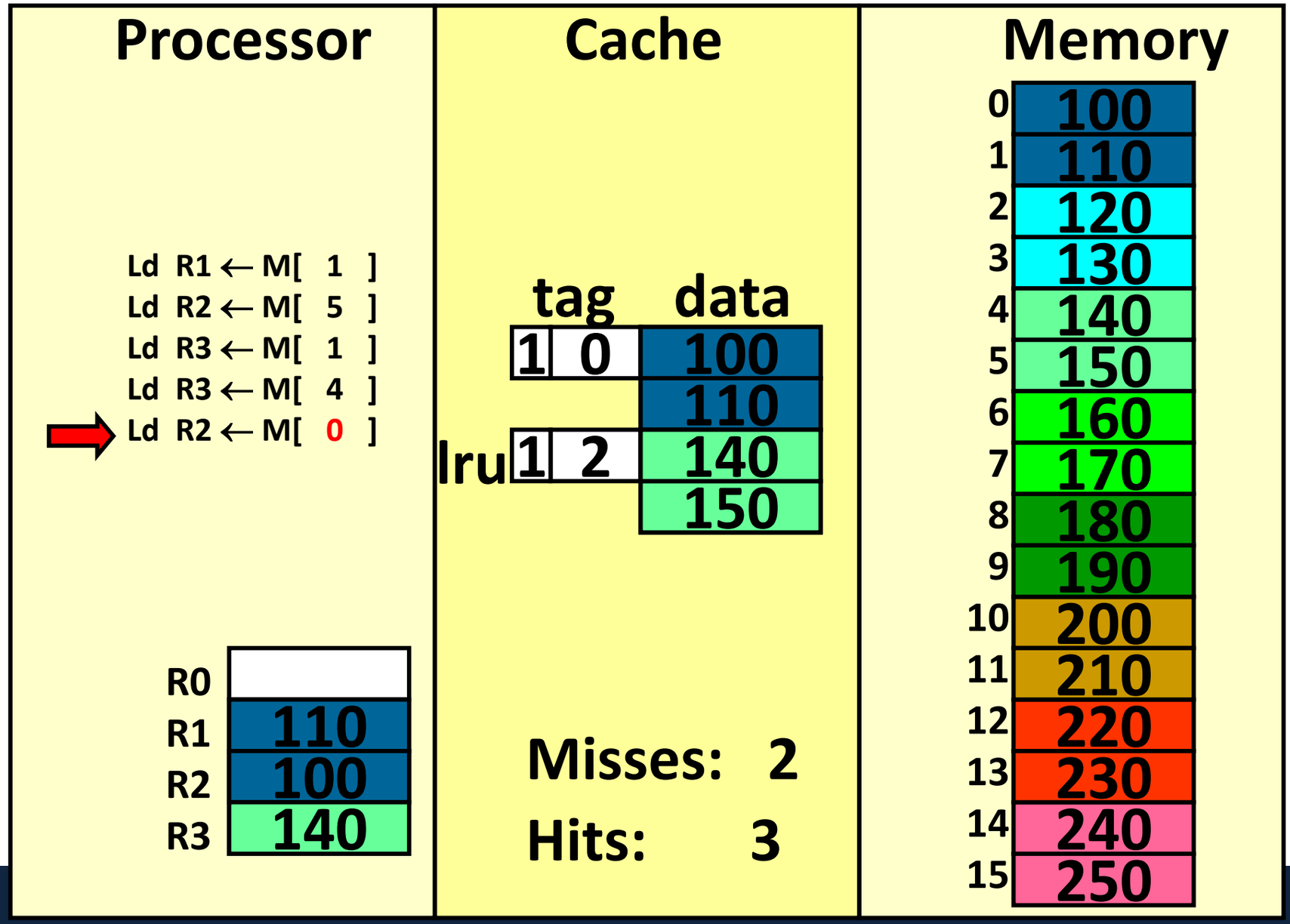
Block size for caches



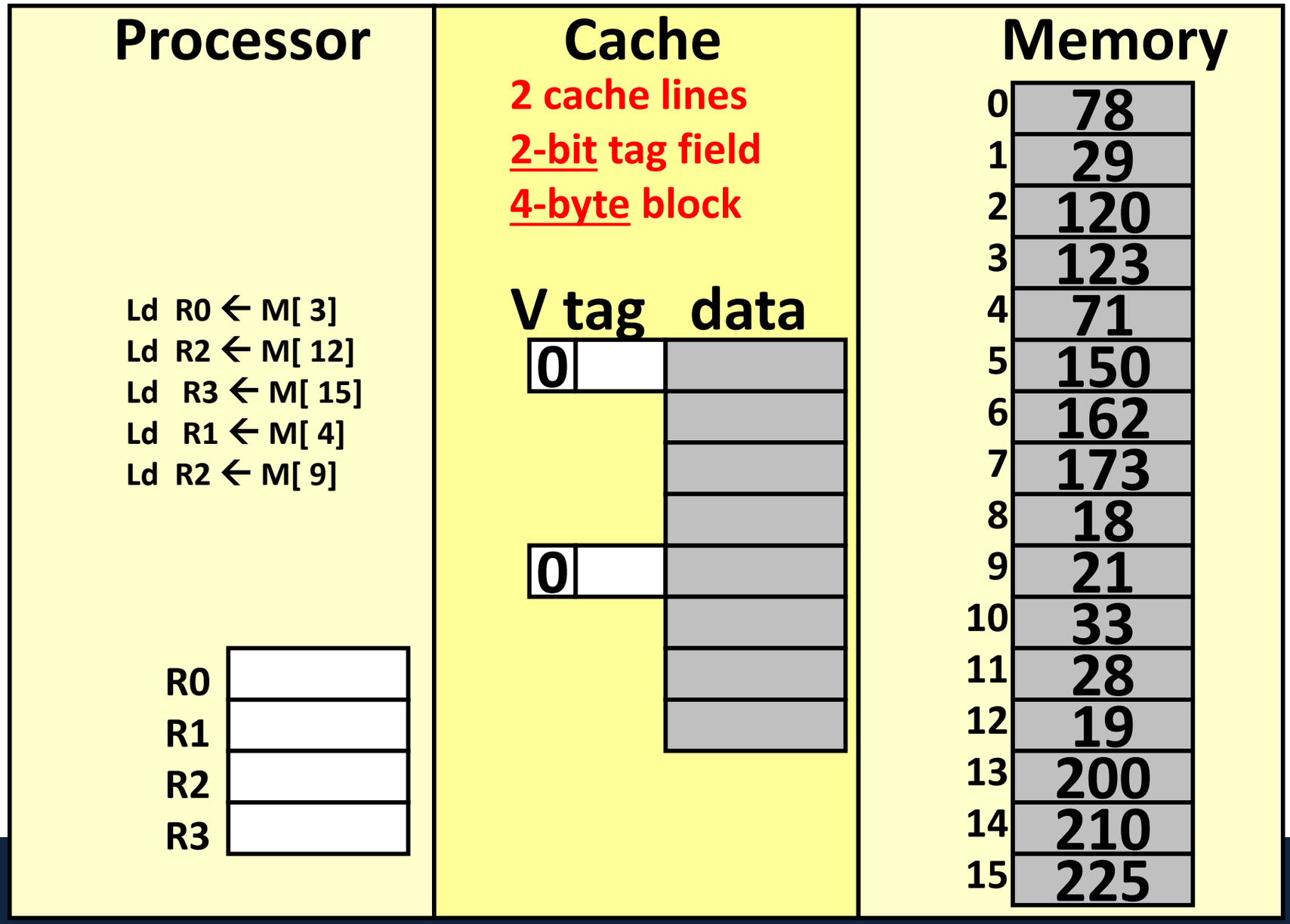
Block size for caches



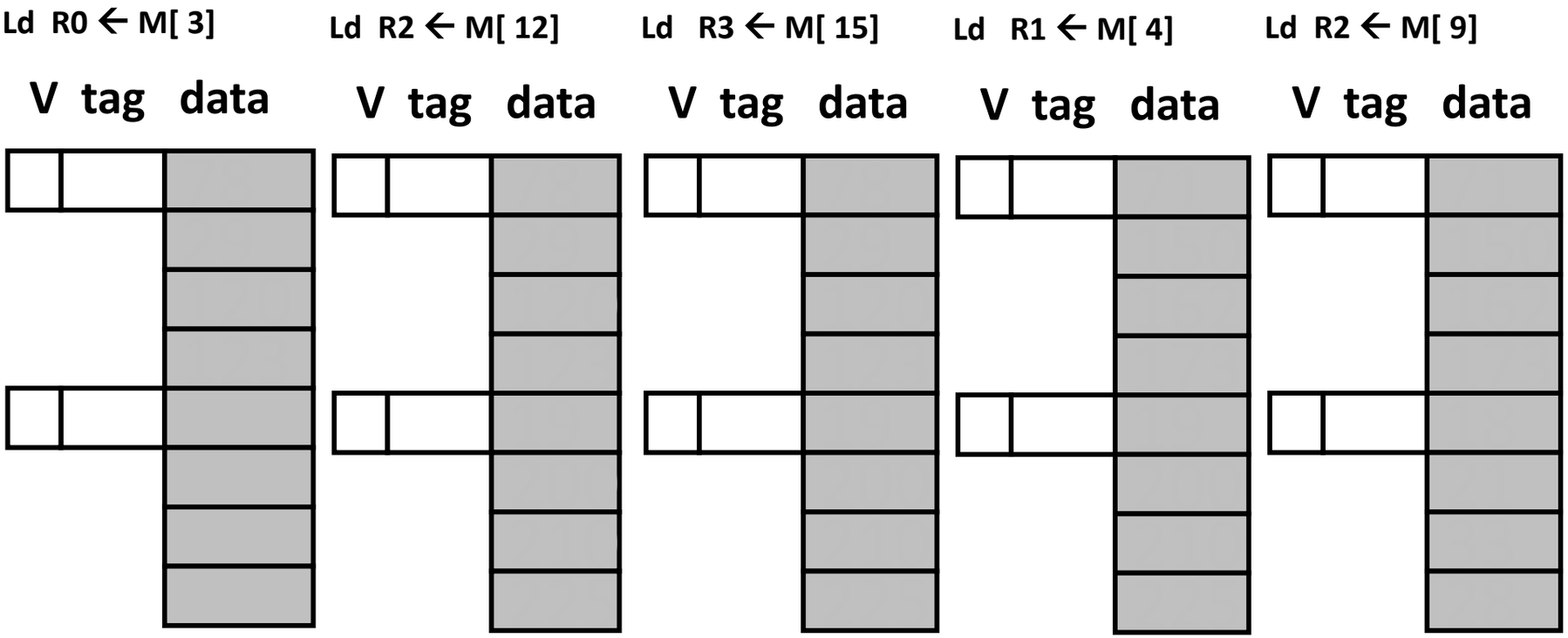
Block size for caches



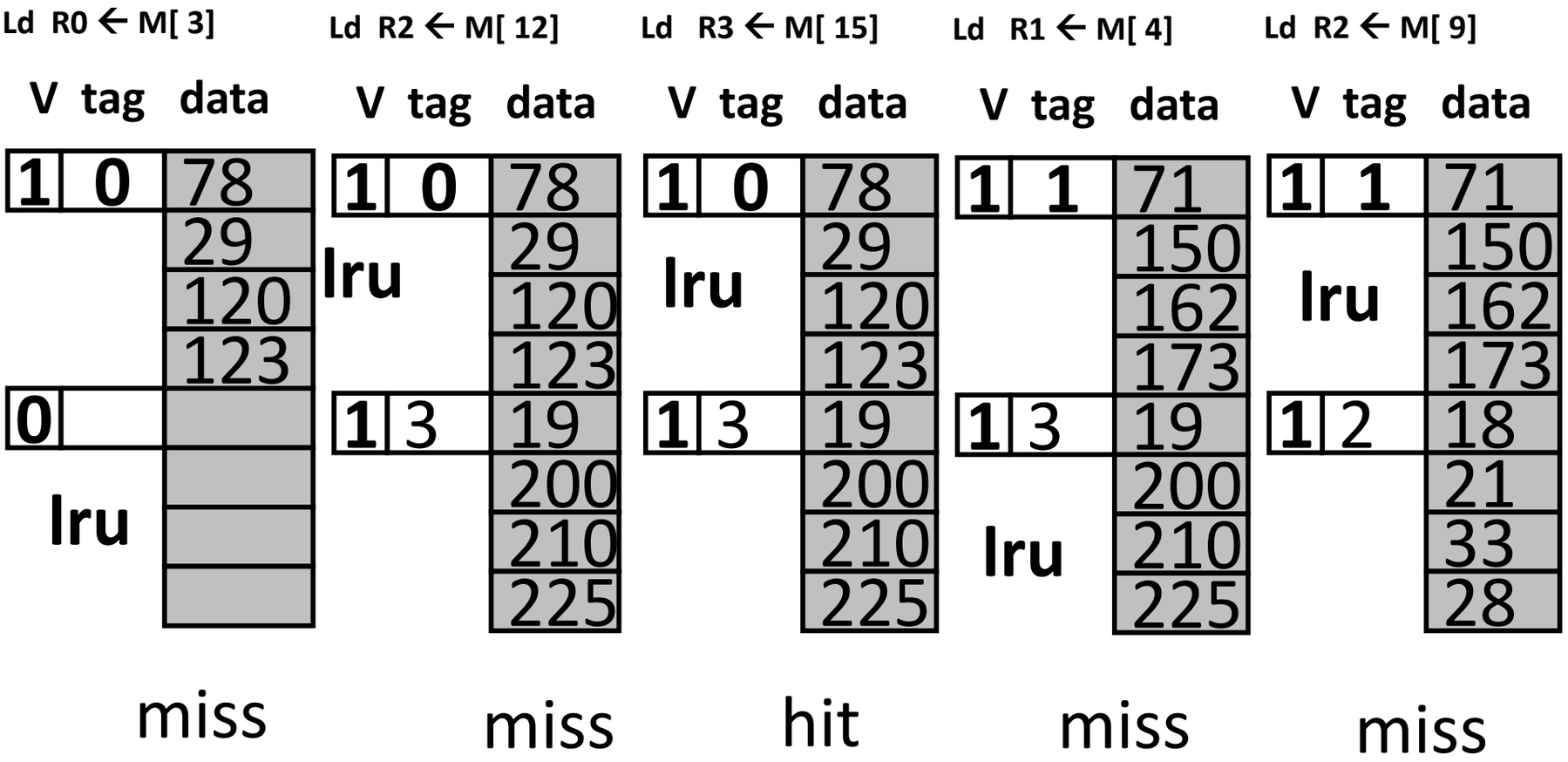
Extra Practice Problem



Solution to Practice Problem



Solution to Practice Problem



Extra Class Problem

Poll: Answer the below questions

*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?

     Tag = 16 – 1 (block offset) = 15 bits;
     2 byte blocks, 8 bytes total = 4 blocks.

  2. For the following reference stream, indicate whether each reference is a hit or miss: 0, 1, 3, 5, 12, 1, 2, 9, 4

     M,H,M,M,M,H,H,M,M

  3. What is the hit rate?

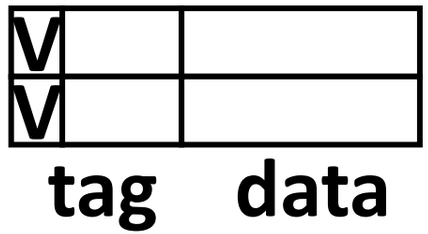
     3/9 = 33 %

  4. How many bits are needed for storage overhead for each block?

     Overhead = 15 (Tag) + 1 (V) + 2 (LRU) = 18 bits

"Way" explanation

- A "way" in a cache is a particular location that a piece of memory **could** exist in
  - E.g. our last cache example was a "two-way" cache
  - Is it the same as the number of entries in the cache?
    - In this example, yes
    - But generally no. Hold on until we talk about set-associate caches



LRU Implementation with Counters

- **2 ways**
  - 1 bit per set to mark latest way accessed in set
  - Evict way not pointed by bit
- **k-way set associative LRU**
  - Requires full ordering of way accesses
  - Hold a $\log_2 k$ bit counter per line
  - When a way *i* is accessed

```
X = Counter[i]
Counter[i] = k-1
for (j = 0 to k-1)
    if ((j≠i) AND (Counter[j]>X)) Counter[j]--;
```

  - When replacement is needed
    - evict way with counter = 0
  - Expensive for even small k's

Initial State

Way	0	1	2	3
Count	0	1	2	3

Access way 2

Way	0	1	2	3
Count	0	1	3	2

Access way 0

Way	0	1	2	3
Count	3	1	2	1

Spatial Locality

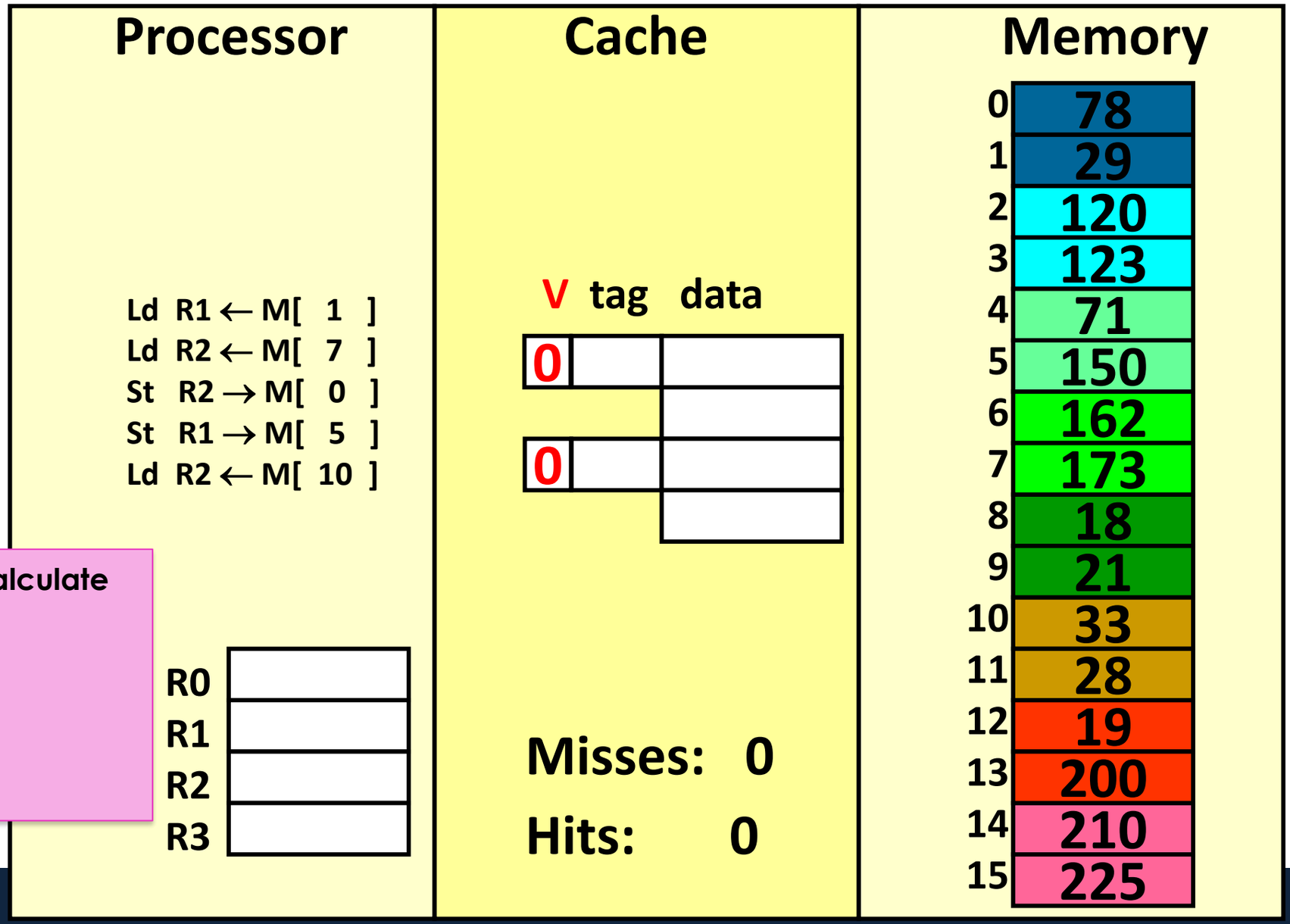
- Notice that when we accessed address 1, we also brought in address 0
  - This turned out to be a good thing, since we later referenced address 0 and found it in the cache
- This is taking advantage of **spatial locality:**
  - If we access a memory location (e.g. 1000), we are more likely to access 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;
  }
```

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?
      (write-through policy)
  - If it is not in the cache:
    - Allocate the line (put it in the cache)?
      (allocate-on-write policy)
    - Write it directly to memory without allocation?
      (no allocate-on-write policy)

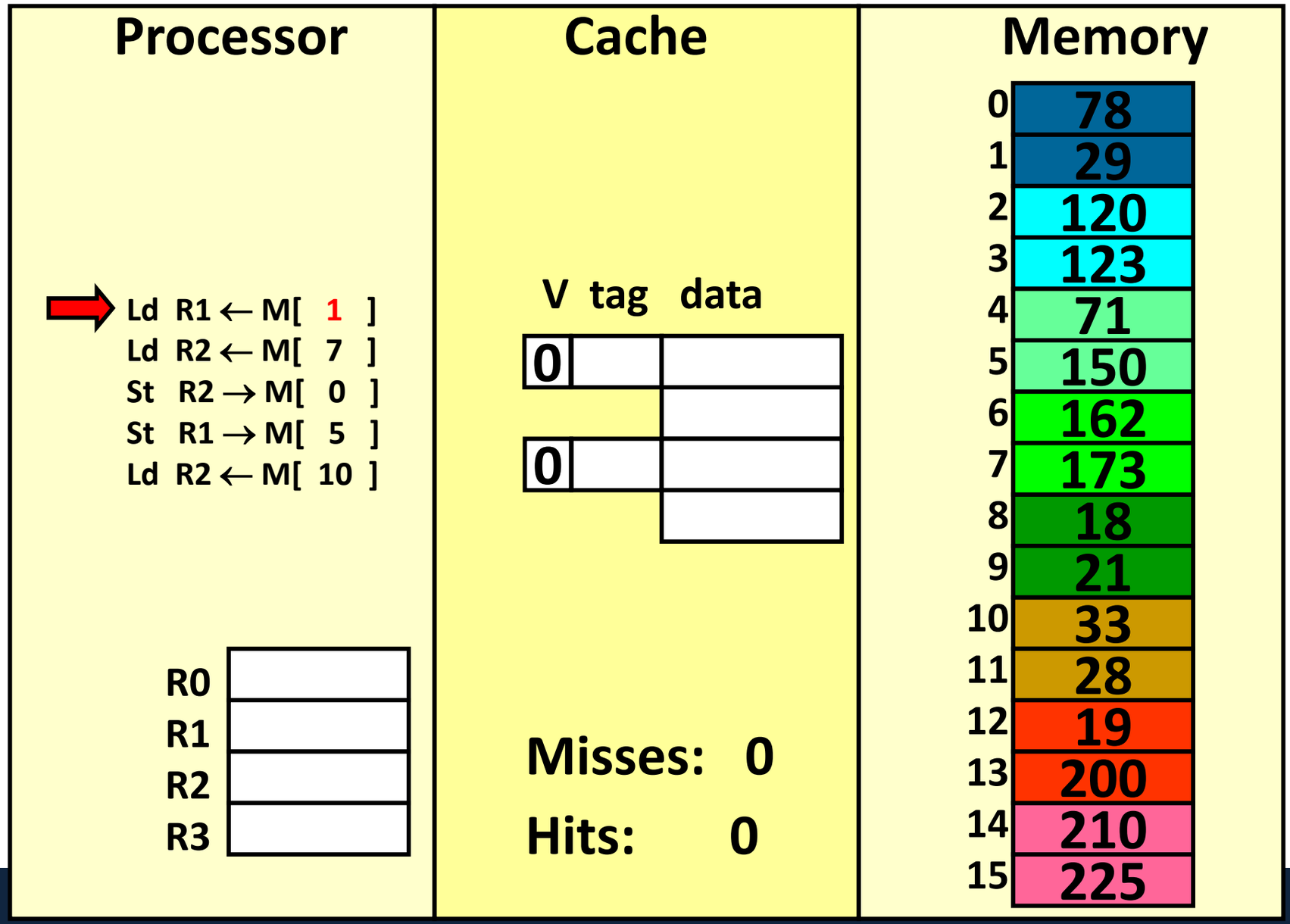
Handling stores (write-through, allocate on write)



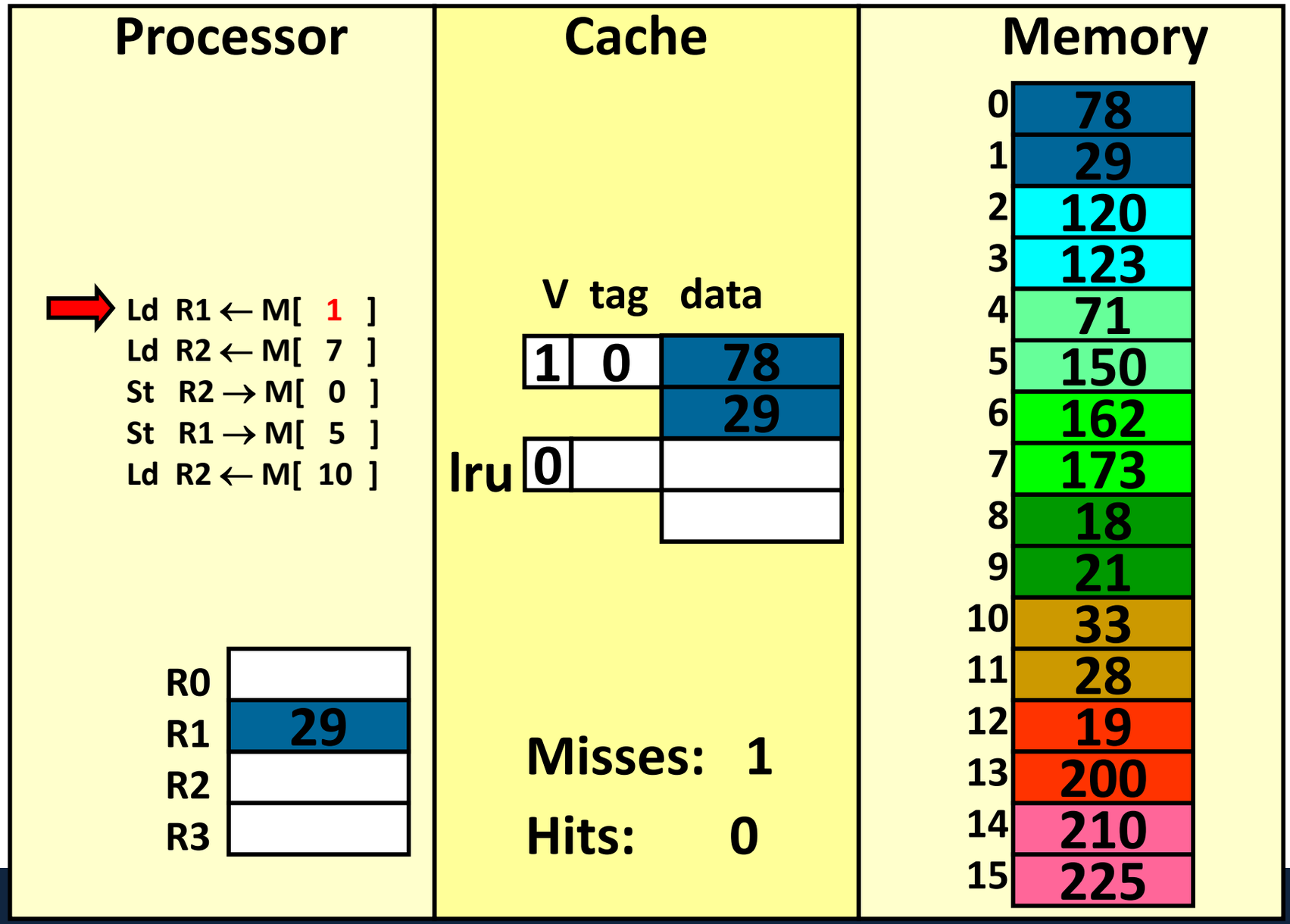
Poll: How can we calculate the tag?

a) addr
b) floor(addr/2)
c) ceil(addr/2)
d) addr*2

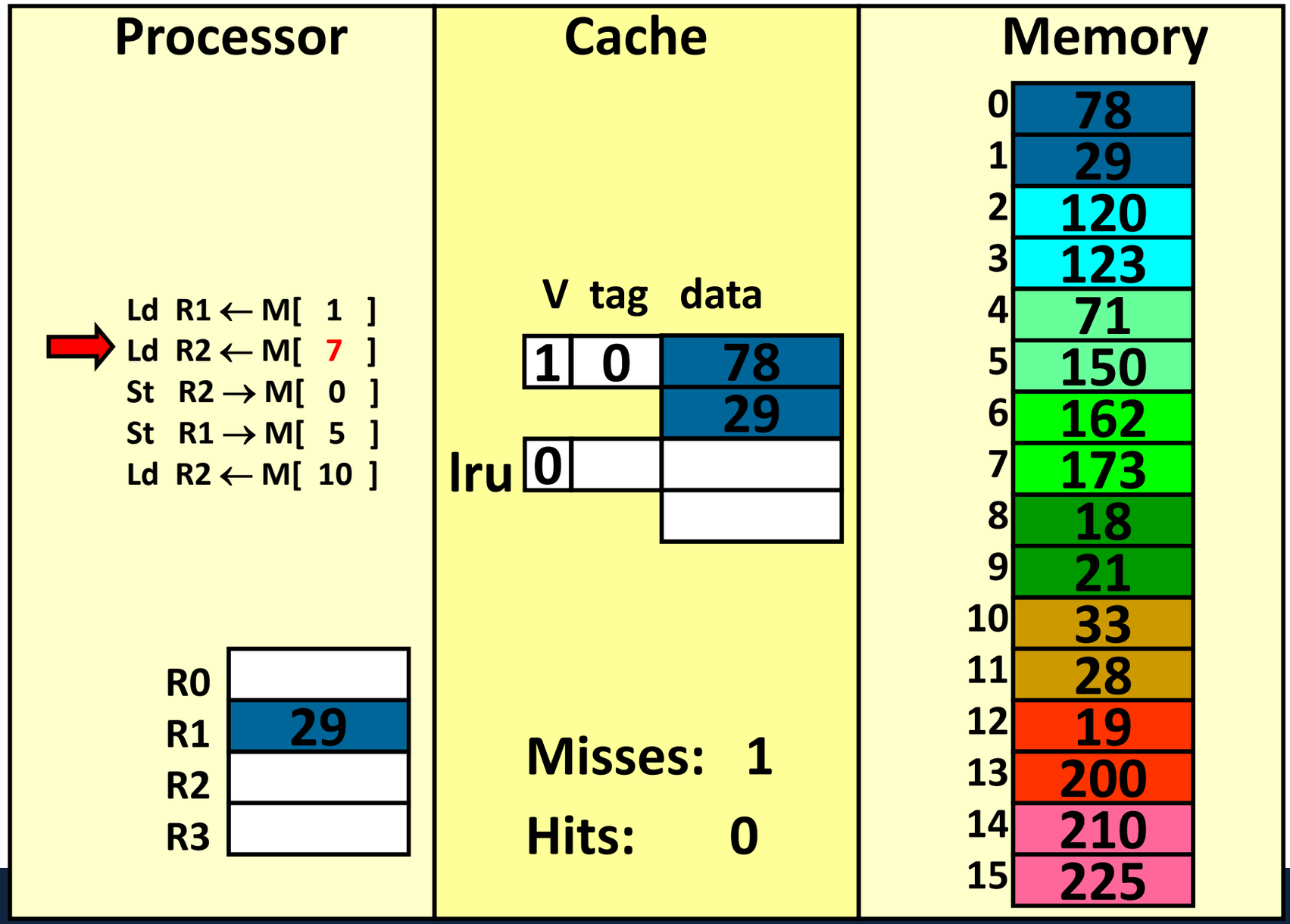
write-through, allocate on write (REF 1)



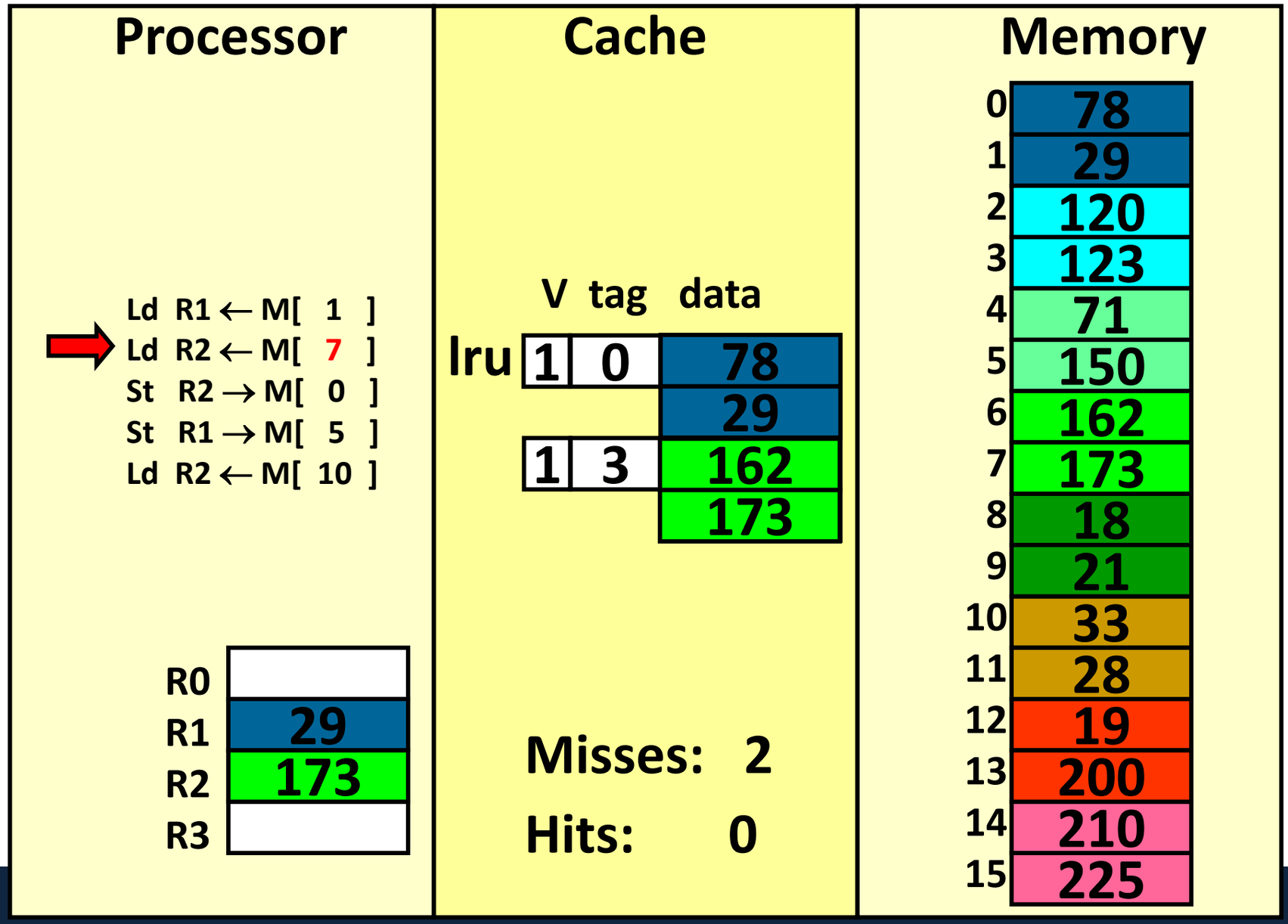
write-through, allocate on write (REF 1)



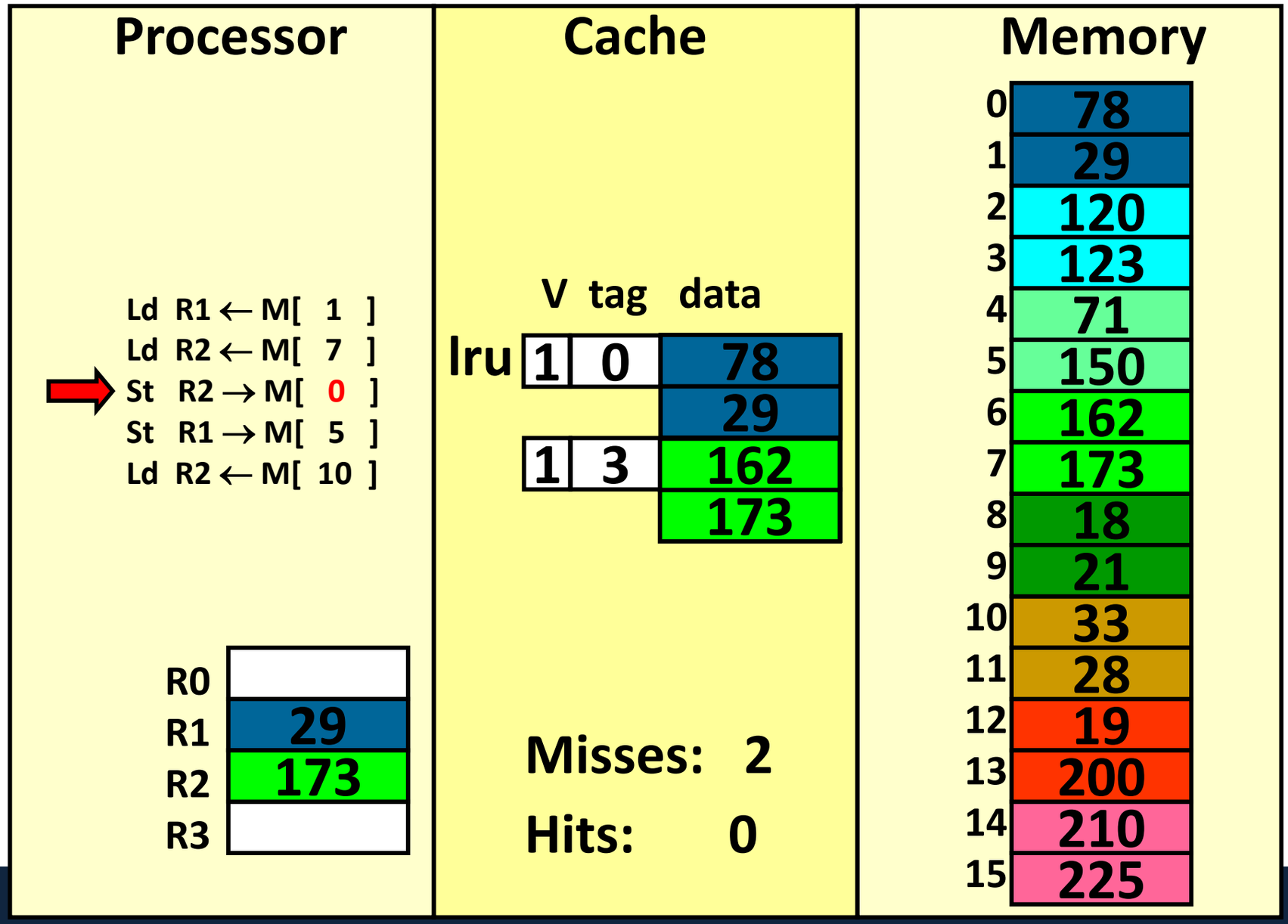
write-through, allocate on write (REF 2)



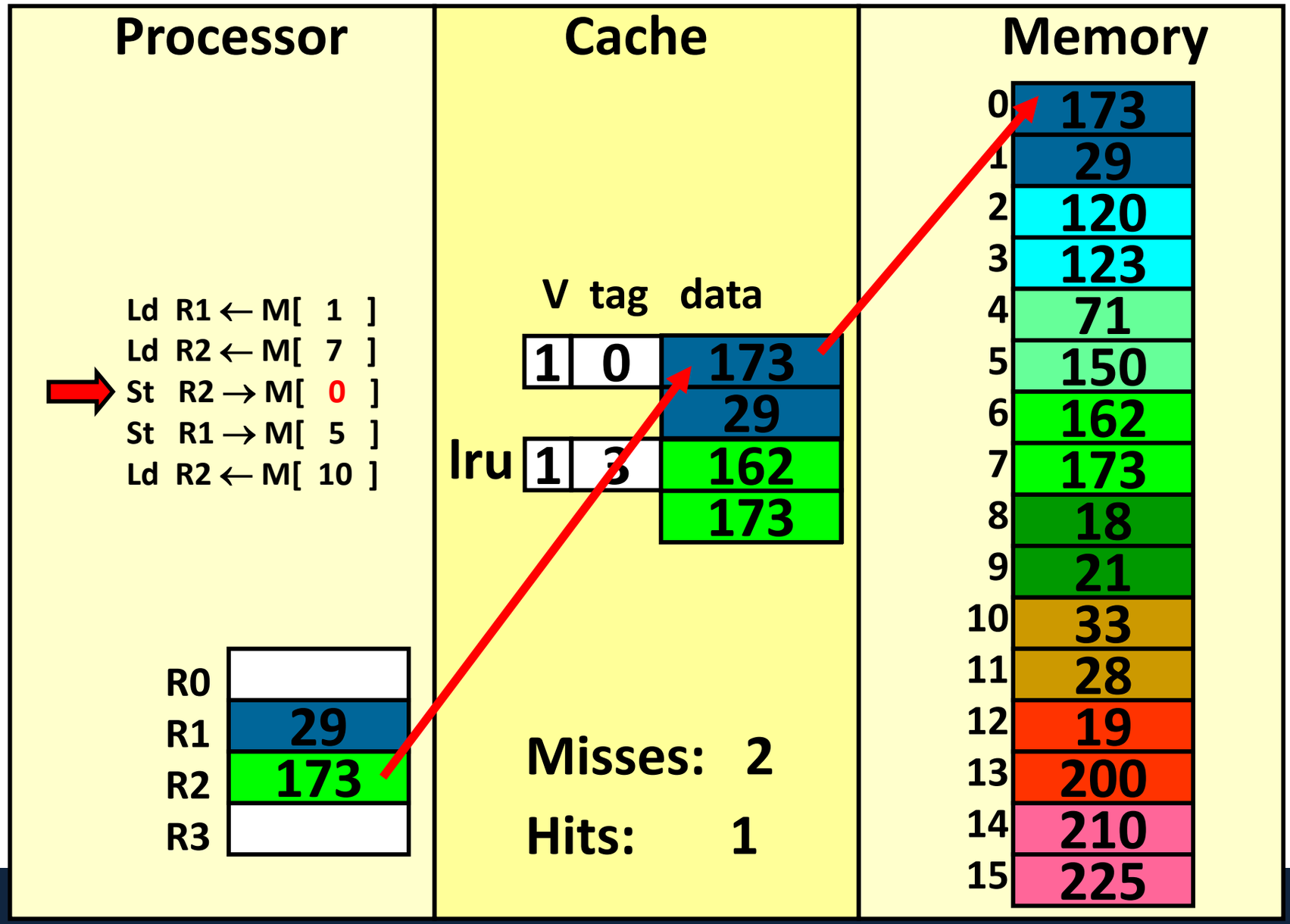
write-through, allocate on write (REF 2)



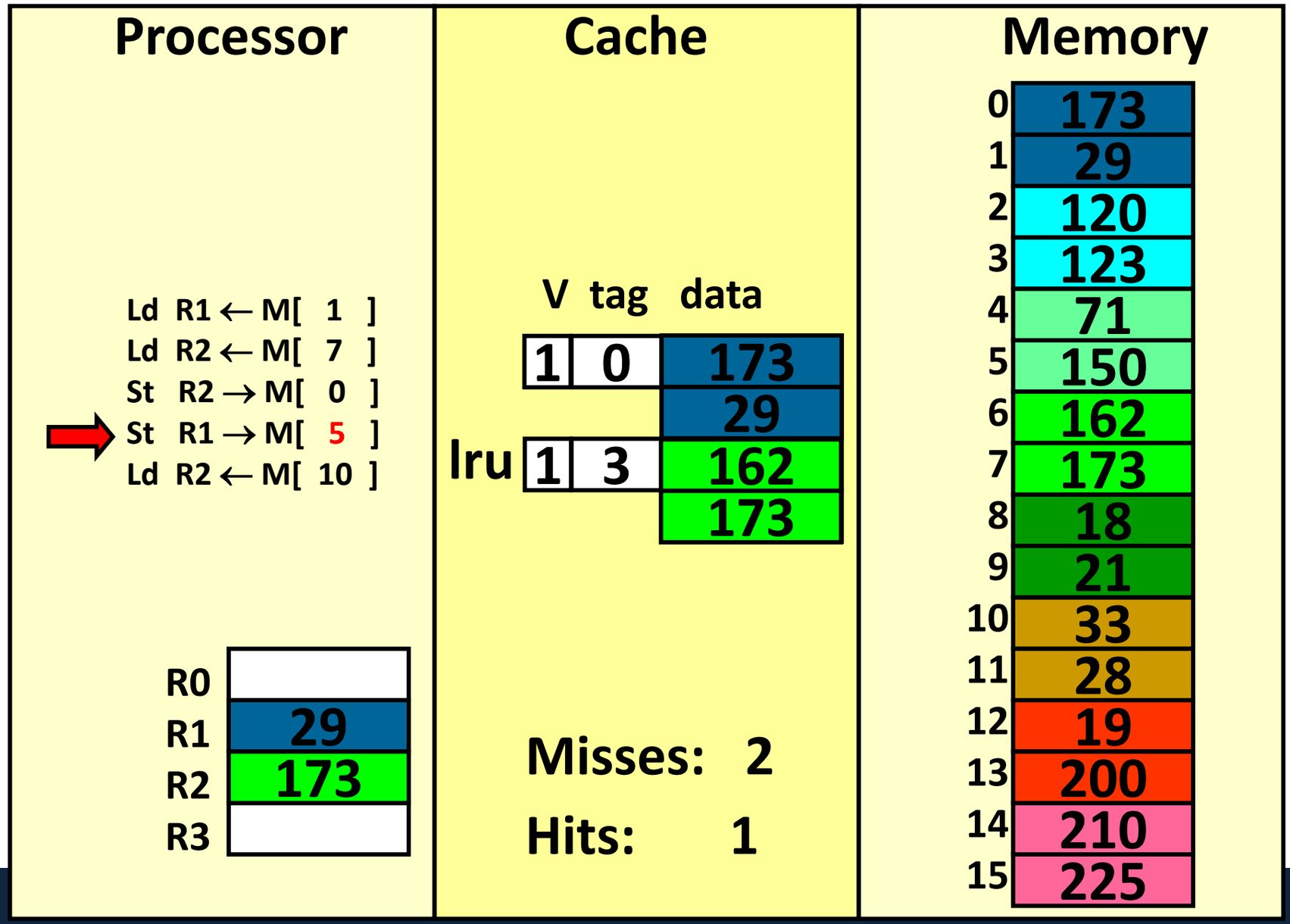
write-through, allocate on write (REF 3)



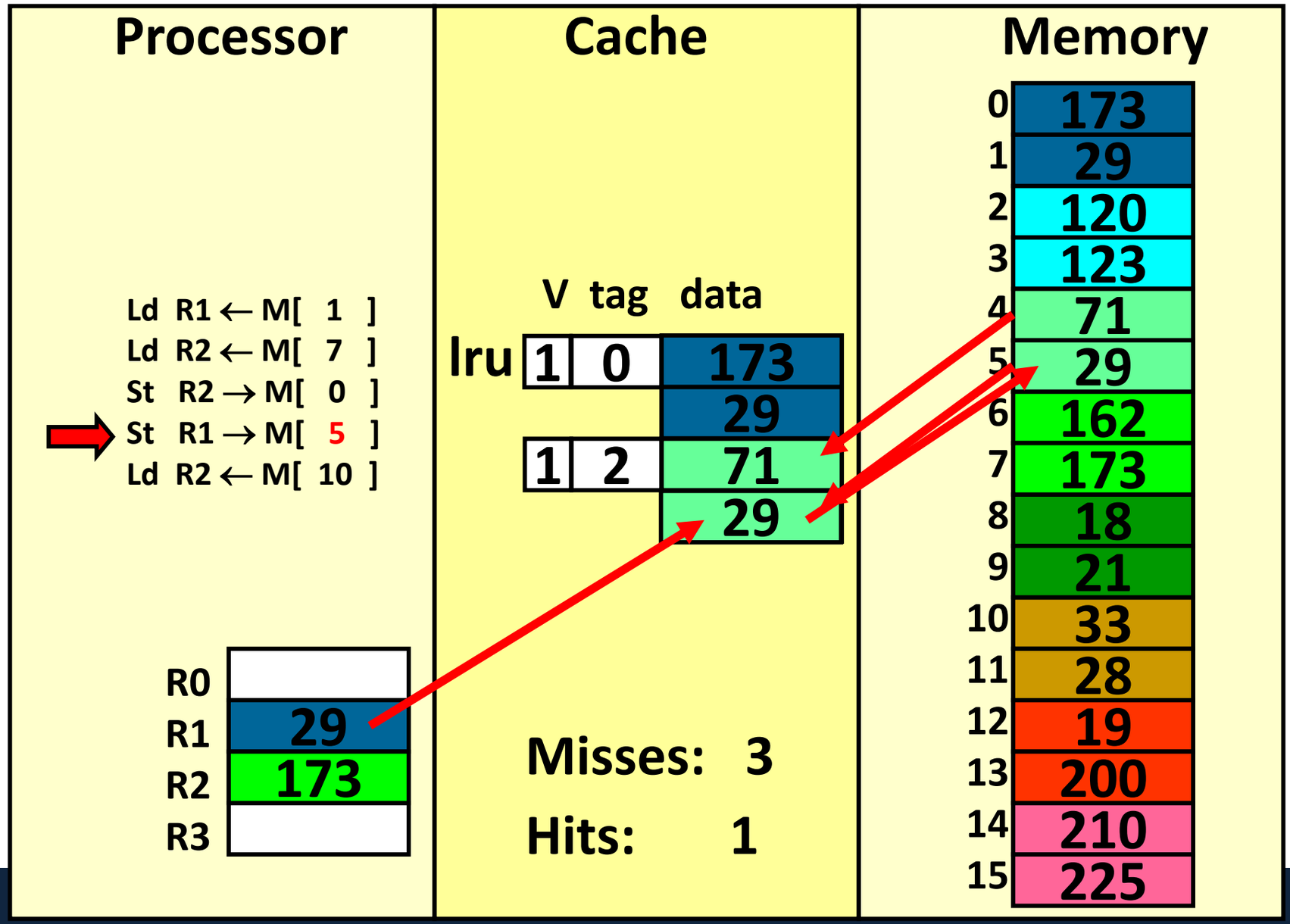
write-through, allocate on write (REF 3)



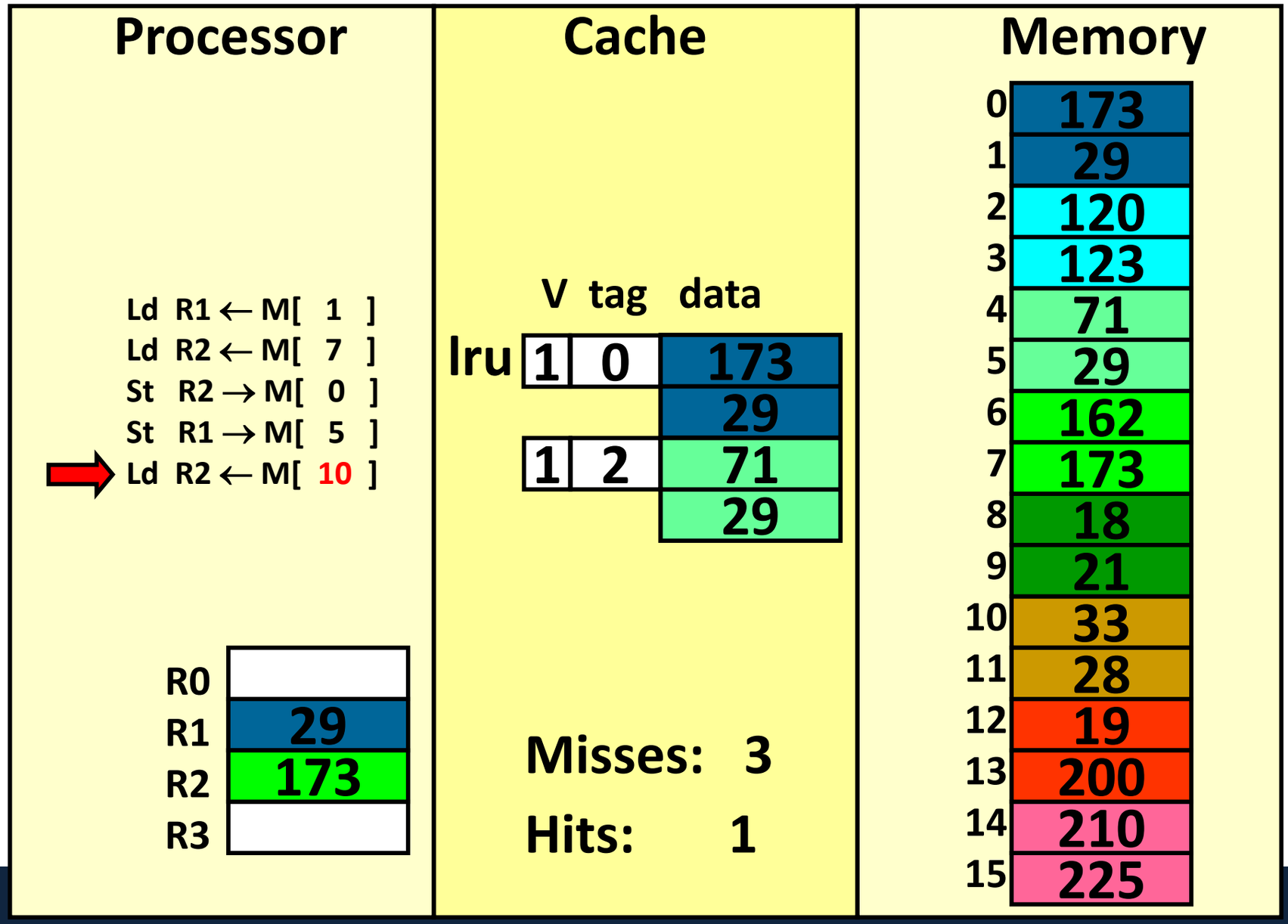
write-through, allocate on write (REF 4)



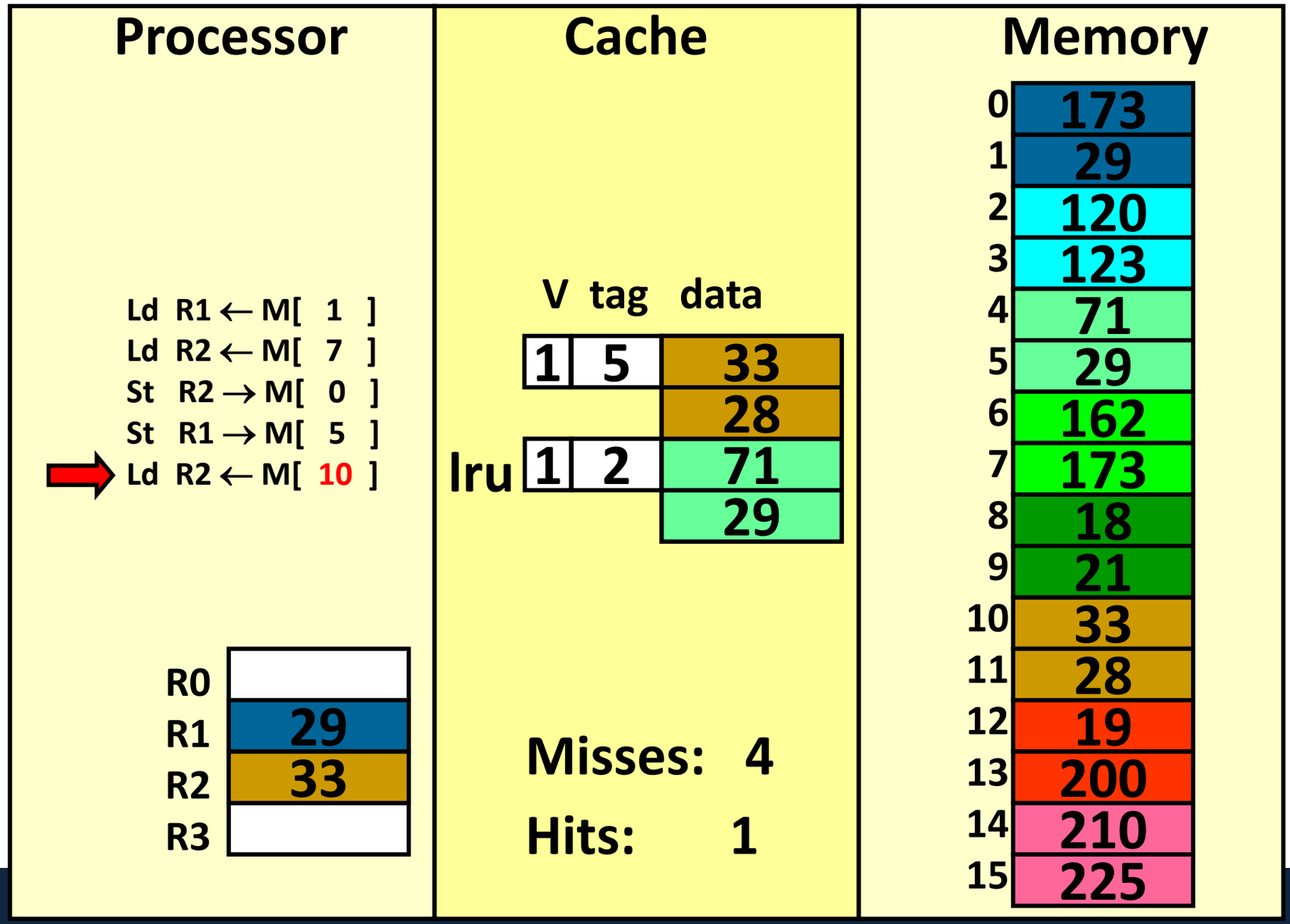
write-through, allocate on write (REF 4)



write-through, allocate on write (REF 6)



write-through, allocate on write (REF 6)



How many memory references?

- Each miss reads a block
  - 2 bytes in this cache
- Each store writes a byte
- Total reads: 8 bytes
- Total writes: 2 bytes

- but caches generally miss < 20%
  - Can we take advantage of that?
  - Multi-core processors have limited bandwidth between caches and memory
  - Extra stores also cost power

Next time

- Write-back caches
- Direct-mapped vs associative caches.
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah