



Cache Memory

Outline

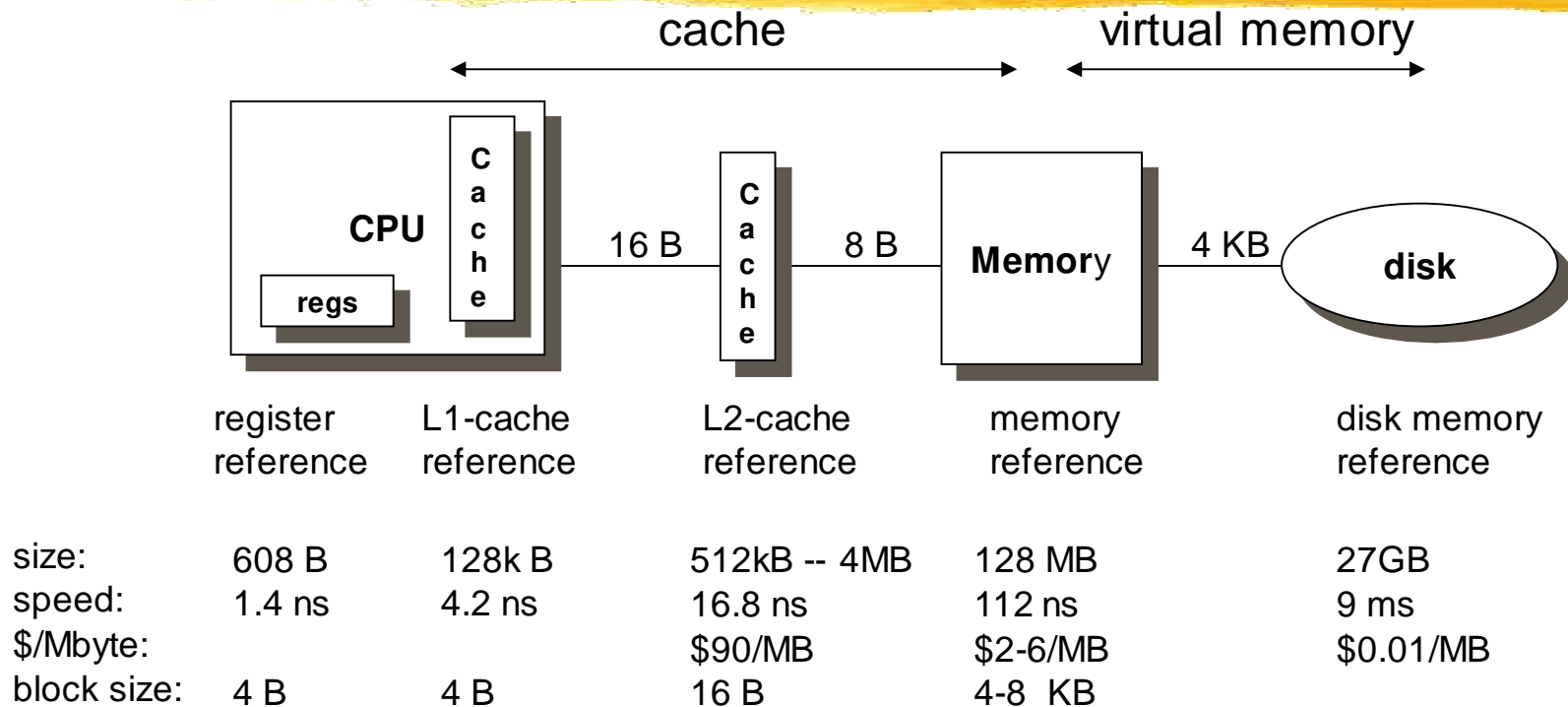


- Memory Hierarchy
- Direct-Mapped Cache
- Write-Through, Write-Back
- Cache replacement policy
- Examples

Principal of Locality

- Cache work on a principal known as **locality of reference**. This principal states asserts that programs continually use and re-use the same locations.
 - Instructions: Loops, common subroutines
 - Data: Look-up Tables, data sets(arrays)
- **Temporal Locality (locality in time)**: Same location will be referenced again soon
- **Spatial Locality (locality in space)**: Nearby locations will be referenced soon

The Tradeoff

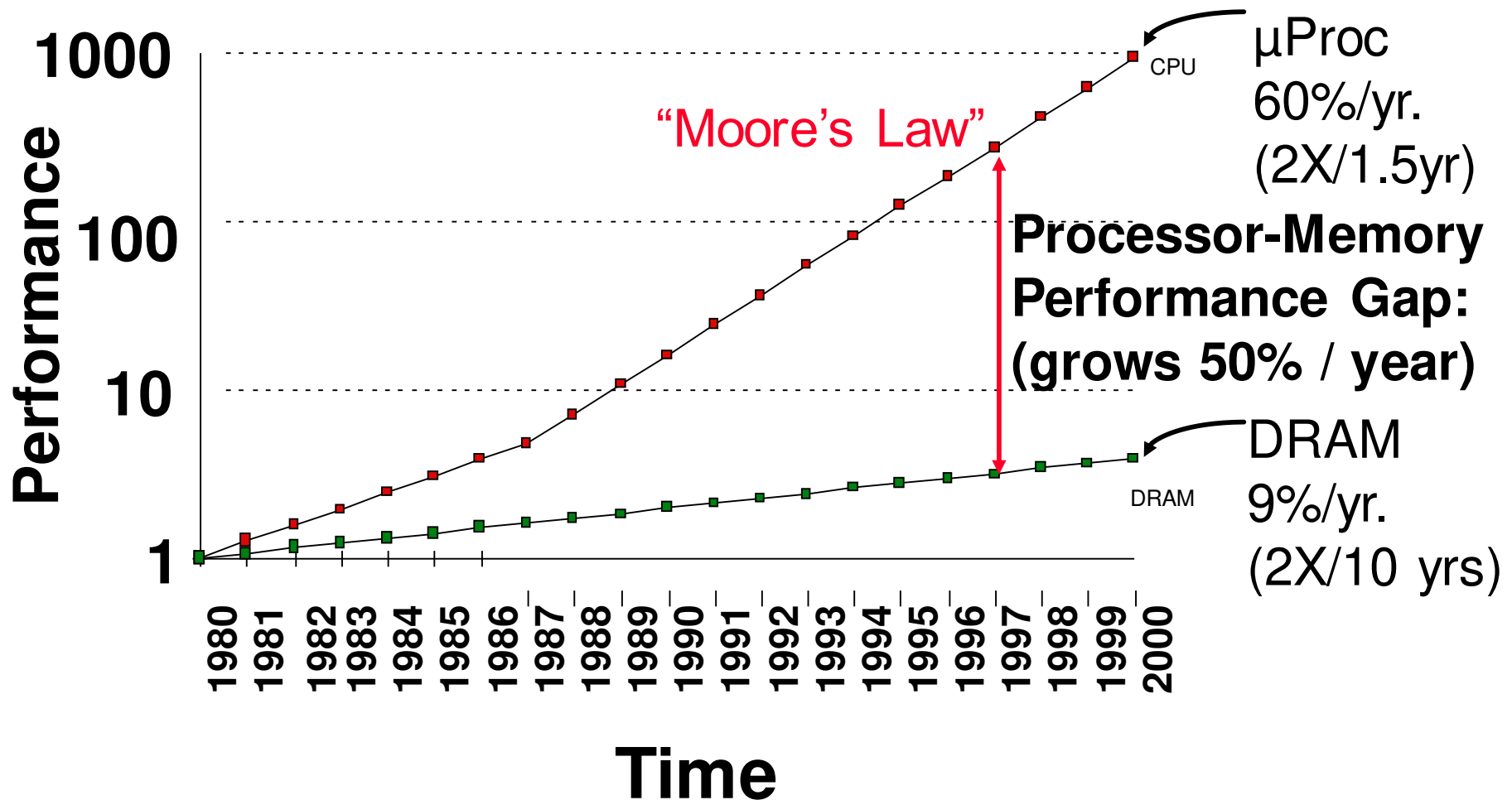


larger, slower, cheaper

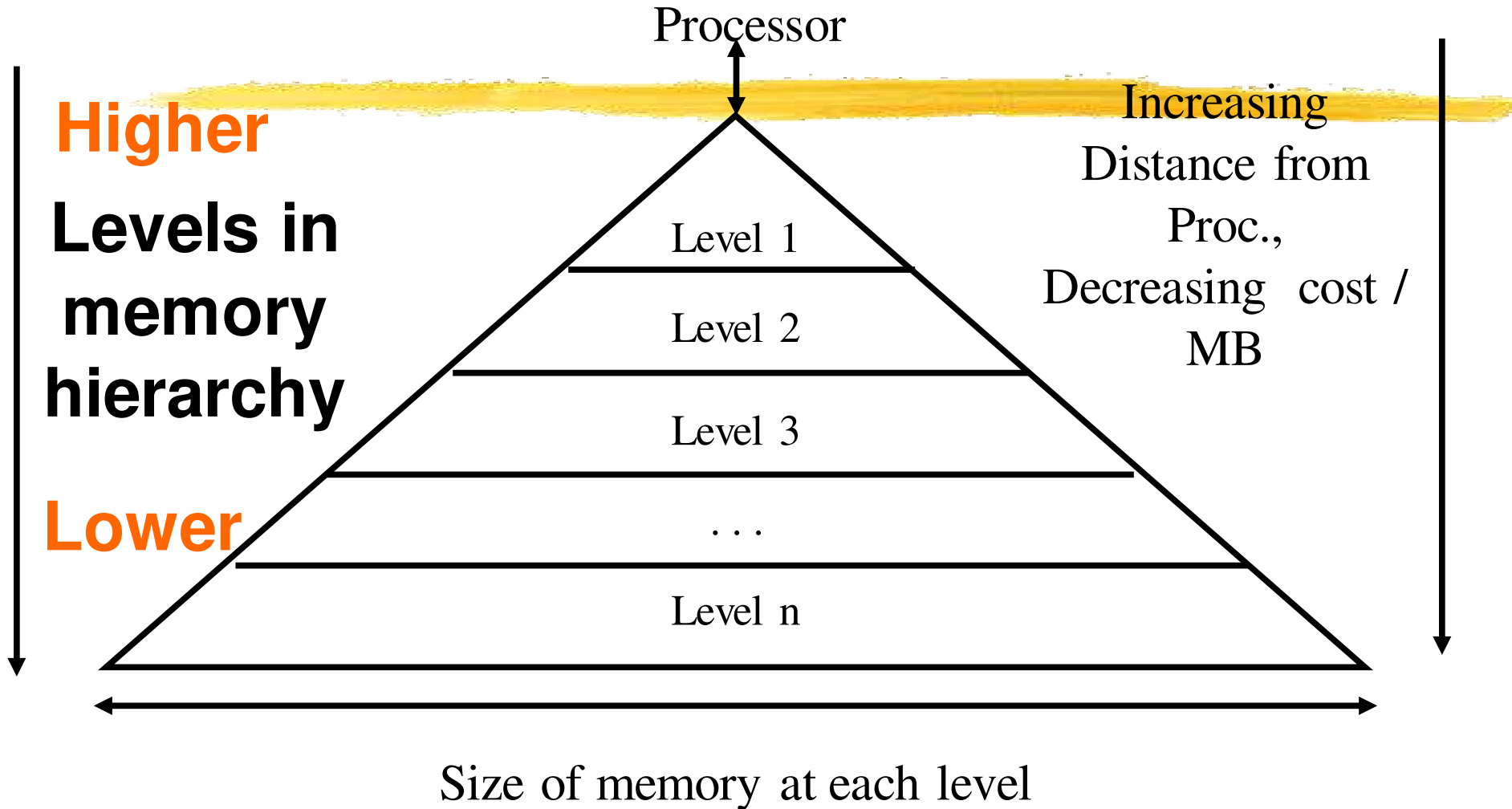
(Numbers are for a DEC Alpha 21264 at 700MHz)

MOORE's LAW

Processor-DRAM Memory Gap (latency)



Memory Hierarchy



Cache Design



- How do we organize cache?
- Where does each memory address map to? (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
- How do we know which elements are in cache?
- How do we quickly locate them?

Cache mappings



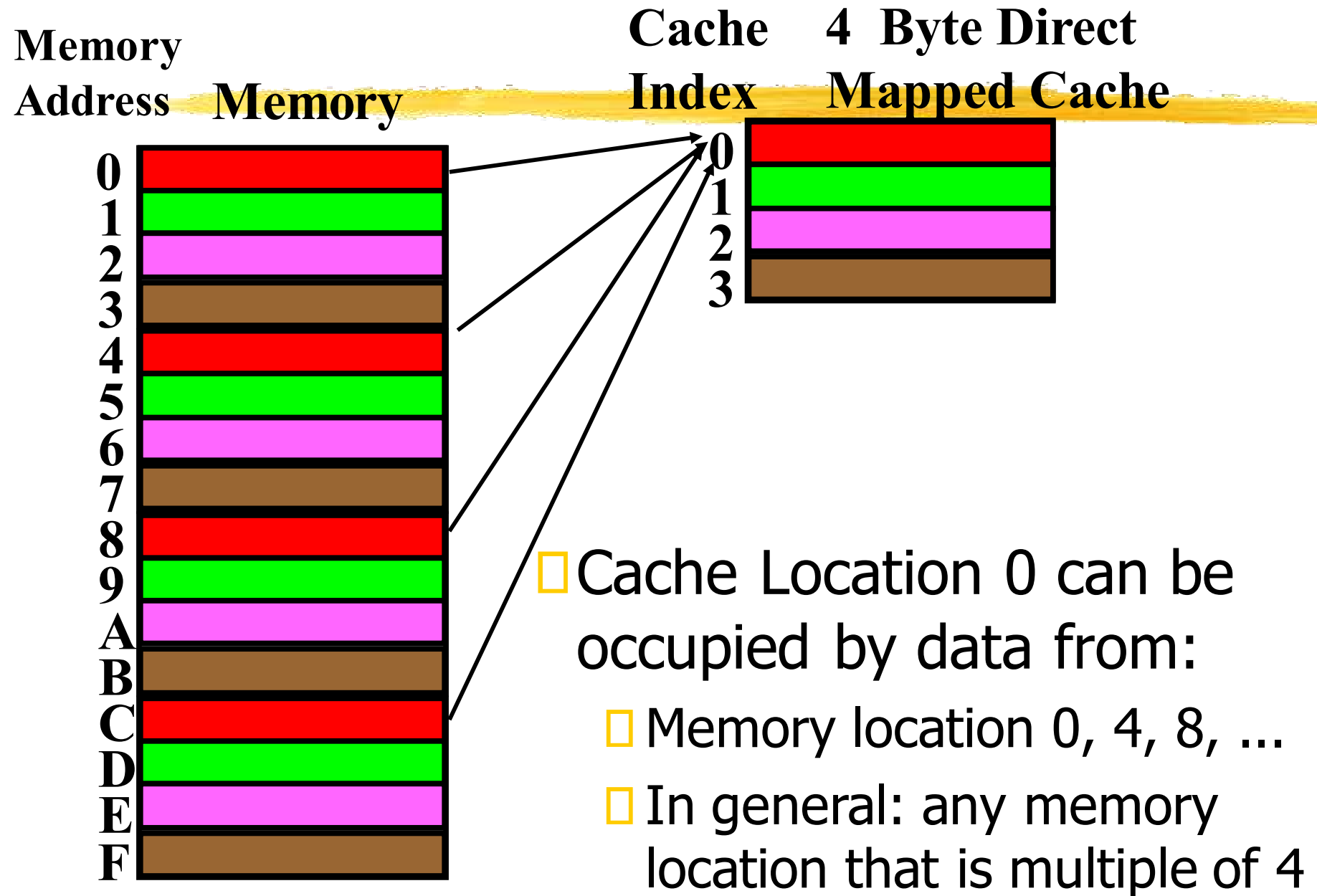
- There are three types of methods for mapping between the main memory addresses and the cache addresses.
- Fully associative cache
- Direct mapped cache
- Set associative cache

Direct-Mapped Cache



- In a direct-mapped cache, each memory address is associated with one possible block within the cache
 - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
 - Block is the unit of transfer between cache and memory

Direct-Mapped Cache



Issues with Direct-Mapped

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Result: divide memory address into three fields



tag
to check
if have
correct block

index
to
select
block

byte
offset
within
block

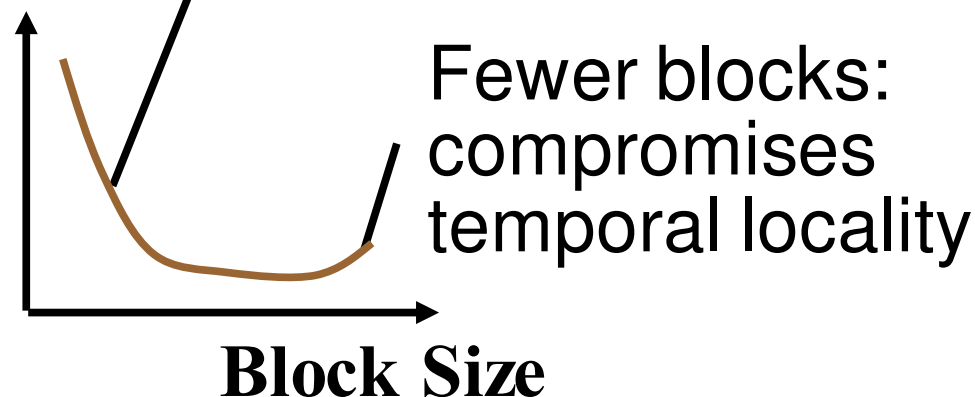
Direct-Mapped Cache Terminology



- All fields are read as unsigned integers.
- Index: specifies the cache index (which “row” of the cache we should look in)
- Offset: once we’ve found correct block, specifies which byte within the block we want
- Tag: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

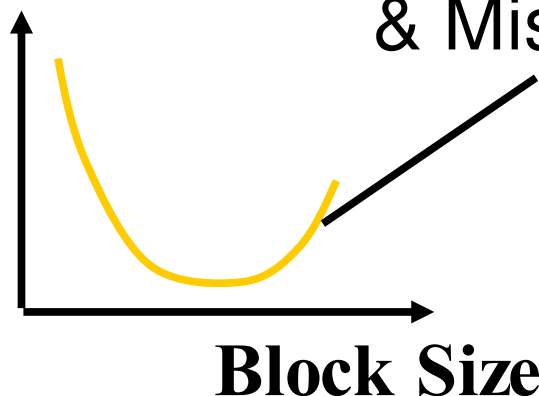
Block Size Tradeoff Conclusions

Miss Rate Exploits Spatial Locality



Average Access Time

Increased Miss Penalty & Miss Rate



Direct-Mapped Cache Example

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
 - need to specify correct byte within a block
 - block contains 4 words = 16 bytes = 2^4 bytes
 - need 4 bits to specify correct byte

Direct-Mapped Cache Example

□ Index: (~index into an "array of blocks")

□ need to specify correct row in cache

□ cache contains 16 KB = 2^{14} bytes

□ block contains 2^4 bytes (4 words)

rows/cache = # blocks/cache (since
there's one block/row)

$$= \frac{\text{bytes/cache}}{\text{bytes/row}}$$

$$= \frac{2^{14} \text{ bytes/cache}}{2^4 \text{ bytes/row}}$$

$$= 2^{10} \text{ rows/cache}$$

need 10 bits to specify this many rows

Direct-Mapped Cache Example

- Tag: use remaining bits as tag
 - tag length = mem addr length - offset - index
$$= 32 - 4 - 10 \text{ bits}$$
$$= 18 \text{ bits}$$
 - so tag is leftmost 18 bits of memory address
- Why not full 32 bit address as tag?
 - All bytes within block need same address (-4b)
 - Index must be same for every address within a block, so its redundant in tag check, thus can leave off to save memory (- 10 bits in this example)

Accessing data in a direct mapped cache

Memory

- Ex.: 16KB of data, direct-mapped, 4 word blocks
- Read 4 addresses
 - 0x00000014,
 - 0x0000001C,
 - 0x00000034, 0x0000003C
- Memory values on right:
 - only cache/memory level of hierarchy

Address (hex)	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

Accessing data in a direct mapped cache

□ 4 Addresses:

□ 0x00000014, 0x0000001C, 0x00000034, 0x00008014

□ 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

00000000000000000000	0000000001	0100
00000000000000000000	0000000001	1100
00000000000000000000	0000000011	0100
00000000000000000010	0000000001	0100
Tag	Index	Offset

Accessing data in a direct mapped cache

- So let's go through accessing some data in this cache
 - 16KB data, direct-mapped, 4 word blocks
- Will see 4 types of events:
- Cache loading: Before any words and tags have been loaded into the cache, all locations contain invalid information. As lines are fetched from main memory into the cache, cache entries become valid. To represent this structure a **valid bit** added to each cache entry.
- cache miss: nothing in cache in appropriate block, so fetch from memory
- cache hit: cache block is valid and contains proper address, so read desired word
- cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory

Cache Loading



- Before any words and tags have been loaded into the cache, all locations contains invalid information. As lines are fetched from main memory into the cache, cache entries become valid. To represent this structure a **valid bit** added to each cache entry. This valid bit indicates that the associated cache line is valid(1) or invalid(0).
- If the valid bit is 0, then a cache miss occurs even if the tag matches the address from CPU, required addressed word to be taken from main memory.

16 KB Direct Mapped Cache, 16B blocks

- Valid bit: determines whether anything is stored in that row (when computer initially turned on, all entries are invalid)

Example Block

Valid

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				

Read 0x00000014 = 0...00 0..001 0100

000000000000000000000000 000000000001 0100

Tag field

Index field

Offset

Valid

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...		...				
1022	0					
1023	0					

So we read block 1 (0000000001)

[illegible]

~~Valid~~

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				

• • •

● ● ●

1022	0					
1023	0					

No valid data

00000000000000000000000000000000 000000000001 0100
Tag field **Index field** **Offset**

Valid

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

So load that data into cache, setting tag, valid

[illegible]

Valid

~~Tag field~~

Index field

Offset

0x0-3

0x4-7

0x8-b

0xc-f

Index

Tag

0

0

1

1

0

a

b

C

d

2

0

3

0

4

0

45

0

56

0	0
---	---

67

0
0

/

0

...

...

1022

0

1023

0

0 00000000000000000000000000000000 0000000000001 0100
Tag field **Index field** **Offset**

Index	Tag	0x0-3	<u>0x4-7</u>	0x8-b	0xc-f
0	0				
<u>1</u>	0	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				

● ● ●

1022	0					
1023	0					

Read 0x0000001C = 0...00 0..001 1100

000000000000000000000000 000000000001 1100
Tag field Index field Offset

Valid					
Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	1	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...		...			
1022	0				
1023	0				

Data valid, tag OK, so read offset return word d

□ 00000000000000000000000000000000 000000000001 1100

Valid

Index	Tag	0x0-3	0x4-7	0x8-b	<u>0xc-f</u>
0	0				
<u>1</u>	<u>1</u> 0	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				

Read 0x00000034 = 0...00 0..011 0100

00000000000000000000000000000000 000000000011 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

So read block 3

00000000000000000000000000000000 0000000011 0100

Valid **Tag field** **Index field** **Offset**

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

No valid data

00000000000000000000000000000000 000000000011 0100
Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

Load that cache block, return word f

000000000000000000000000 000000000011 0100

Valid Tag field Index field Offset

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	0	a	b	c	d
2	0				
3	0	e	f	g	h
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				

Read 0x00008014 = 0...10 0..001 0100

0000000000000000000010 00000000001 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

00000000000000000000000010 000000000001 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

Cache Block 1 Tag does not match (0 != 2)

0000000000000000000010 000000000001 0100
Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

□ 0000000000000000000010 00000000001 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

And return word j

0000000000000000000010 000000000001 0100

Valid Tag field index field Offset

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	1	2	i	k	l
2	0				
3	1	0	e	f	g
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				

Vector Product Example

```
float dot_prod(float x[1024], y[1024])
{
    float sum = 0.0;
    int i;
    for (i = 0; i < 1024; i++)
        sum += x[i]*y[i];
    return sum;
}
```

□ Machine

- DEC Station 5000

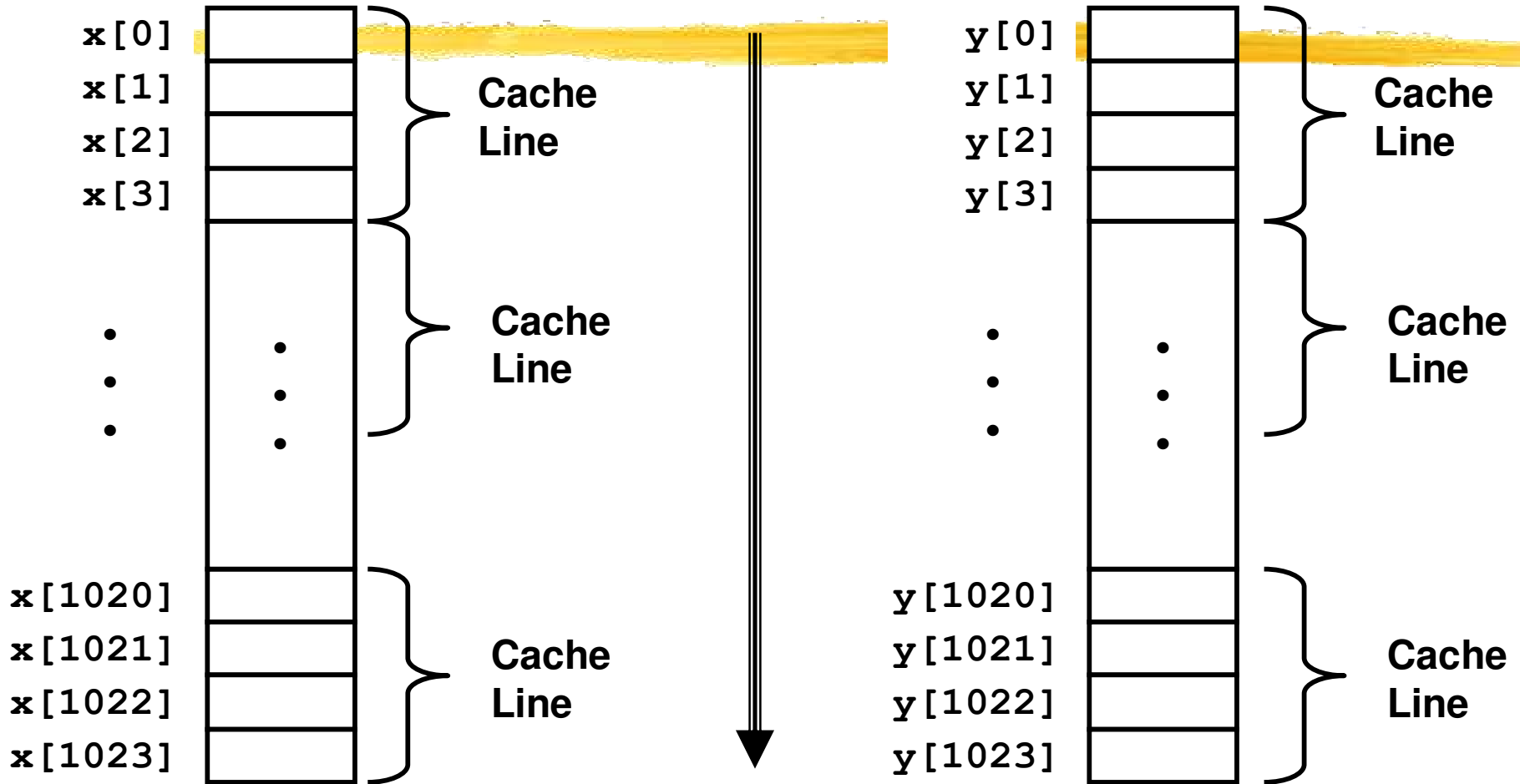
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

□ Performance

- Good case: 24 cycles / element

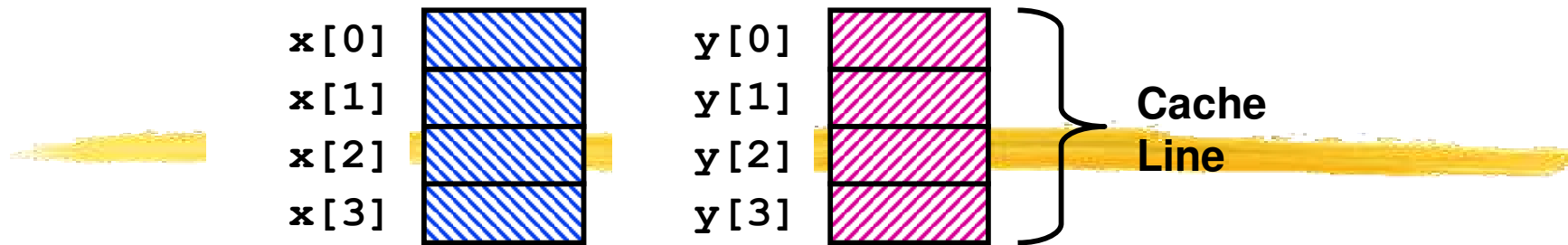
- Bad case: 66 cycles / element

Thrashing Example



□ Access one element from each array per iteration

Thrashing Example: Good Case



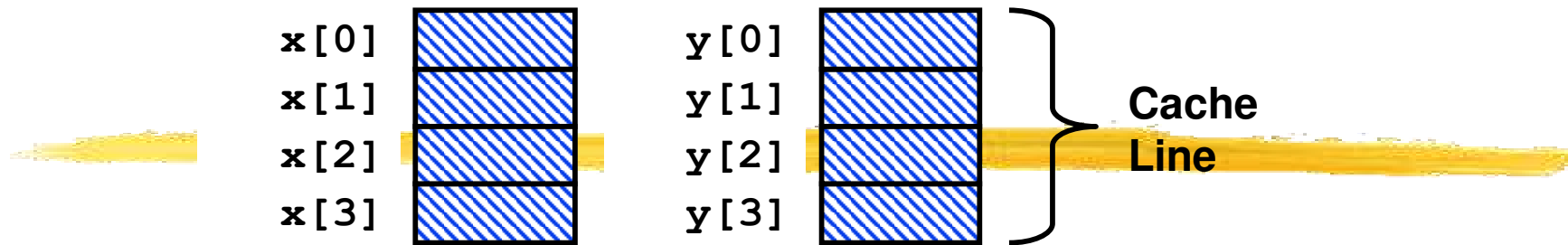
Access Sequence

- Read $x[0]$
 - $x[0], x[1], x[2], x[3]$ loaded
- Read $y[0]$
 - $y[0], y[1], y[2], y[3]$ loaded
- Read $x[1]$
 - Hit
- Read $y[1]$
 - Hit
- ...
- 2 misses / 8 reads

Analysis

- $x[i]$ and $y[i]$ map to different cache lines
- Miss rate = 25%
 - Two memory accesses / iteration
 - On every 4th iteration have two misses

Thrashing Example: Bad Case



Access Pattern

- Read $x[0]$
 - $x[0], x[1], x[2], x[3]$ loaded
- Read $y[0]$
 - $y[0], y[1], y[2], y[3]$ loaded
- Read $x[1]$
 - $x[0], x[1], x[2], x[3]$ loaded
- Read $y[1]$
 - $y[0], y[1], y[2], y[3]$ loaded
- 8 misses / 8 reads

Analysis

- $x[i]$ and $y[i]$ map to same cache lines
- Miss rate = 100%
 - Two memory accesses / iteration
 - On *every* iteration have two misses

□ **Average access time**

□ Average access time, t_a , given by:

□ $t_a = t_c + (1 - h)t_m$

□ assuming again that the first access must be to the cache before an access is made to the main memory. Only read requests are consider so far.

□ **Example**

□ If hit ratio is 0.85 (a typical value), main memory access time is 50 ns and cache access time is 5 ns, then average access time is $5 + 0.15 \times 50 = 12.5$ ns.

□ **THROUGHOUT** t_c is the time to access the cache, get (or write) the data if a hit or recognize a miss. In practice, these times could be different.

Example-1

□ Assume

- Hit Time = 1 cycle
- Miss rate = 5%
- Miss penalty = 20 cycles

□ Average memory access time = $t_c + m * t_m$

(m: miss rate, h: hit rate,

t_m : main memory access time, t_c : Cache access time)

Average memory access time = $1 + 0.05 \times 20 = 2$ cycle

or

$$\square = t_c * h + (t_m + t_c) * m$$

$$1 \times 0.95 + 21 \times 0.05 = 2 \text{ cycle}$$

Example-2

□ Assume

□ L1 Hit Time = 1 cycle

□ L1 Miss rate = 5%

□ L2 Hit Time = 5 cycles

□ L2 Miss rate = 15% (% L1 misses that miss)

□ L2 Miss Penalty = 100 cycles

□ L1 miss penalty = $5 + 0.15 * 100 = 20$

□ Average memory access time = $1 + 0.05 \times 20$
= 2 cycle

Replacement Algorithms

- *When a block is fetched, which block in the target set should be replaced?*
- Optimal algorithm:
 - replace the block that will not be used for the longest time (must know the future)
- Usage based algorithms:
 - Least recently used (LRU)
 - replace the block that has been referenced least recently
 - hard to implement
- Non-usage based algorithms:
 - First-in First-out (FIFO)
 - treat the set as a circular queue, replace head of queue.
 - easy to implement
 - Random (RAND)
 - replace a random block in the set
 - even easier to implement

Implementing RAND and FIFO

□ FIFO:

- maintain a modulo E counter for each set.
- counter in each set points to next block for replacement.
- increment counter with each replacement.

□ RAND:

- maintain a single modulo E counter.
- counter points to next block for replacement in any set.
- increment counter according to some schedule:
 - each clock cycle, each mem reference, or each replacement.

□ LRU

- Need state machine for each set
- Encodes usage ordering of each element in set
- $E!$ possibilities $\Rightarrow \sim E \log E$ bits of state

What Happens on a Write?



- Need to keep cache consistent with the main memory
 - Reads are easy - require no modification
 - Writes- when does the update occur
- Write-Through
 - On cache write- always update main memory as well
- Write-Back
 - On cache write- remember that block is modified (dirty)
 - Update main memory when dirty block is replaced
 - Sometimes need to flush cache (I/O, multiprocessing)

What to do on a write hit?

☐ Write-through

- ☐ update the word in cache block and corresponding word in memory

☐ Write-back

- ☐ update word in cache block
- ☐ allow memory word to be "stale"

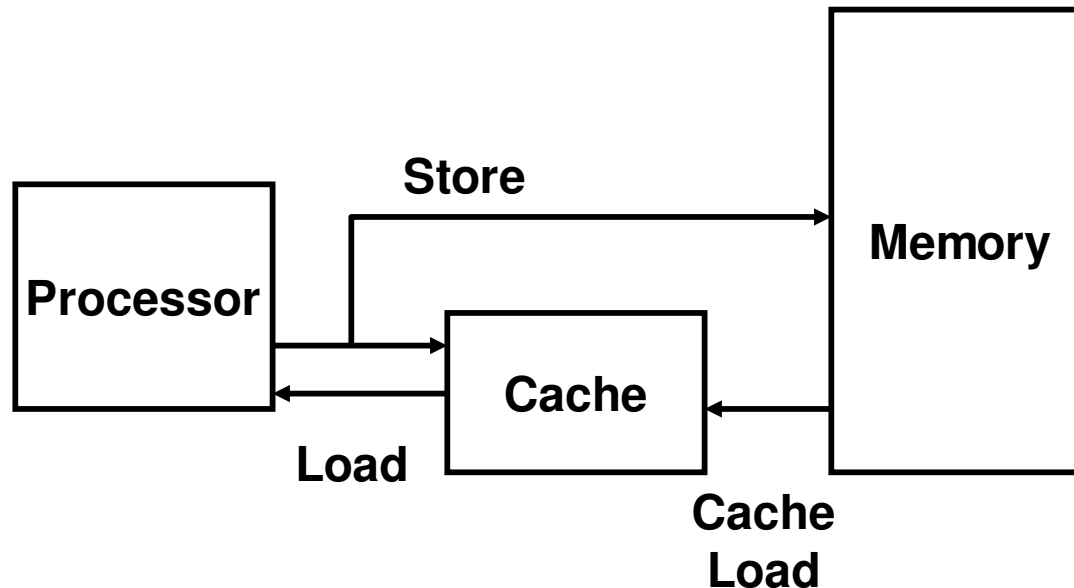
=> add 'dirty' bit to each line indicating that memory needs to be updated when block is replaced

=> OS flushes cache before I/O !!!

☐ Performance trade-offs?

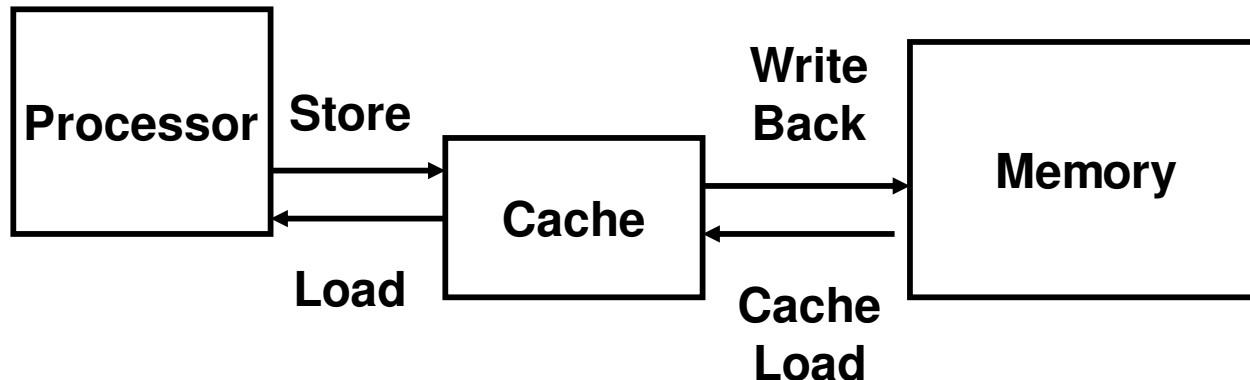
Write Through

- Store by processor updates cache *and* memory
- Memory always consistent with cache
- ~2X more loads than stores
- WT always combined with write buffers so that don't wait for lower level memory



Write Back

- ❑ Store by processor only updates cache line
- ❑ Modified line written to memory only when it is evicted
 - ❑ Requires “dirty bit” for each line
 - ❑ Set when line in cache is modified
 - ❑ Indicates that line in memory is stale
- ❑ Memory not always consistent with cache
- ❑ No writes of repeated writes



Store Miss?



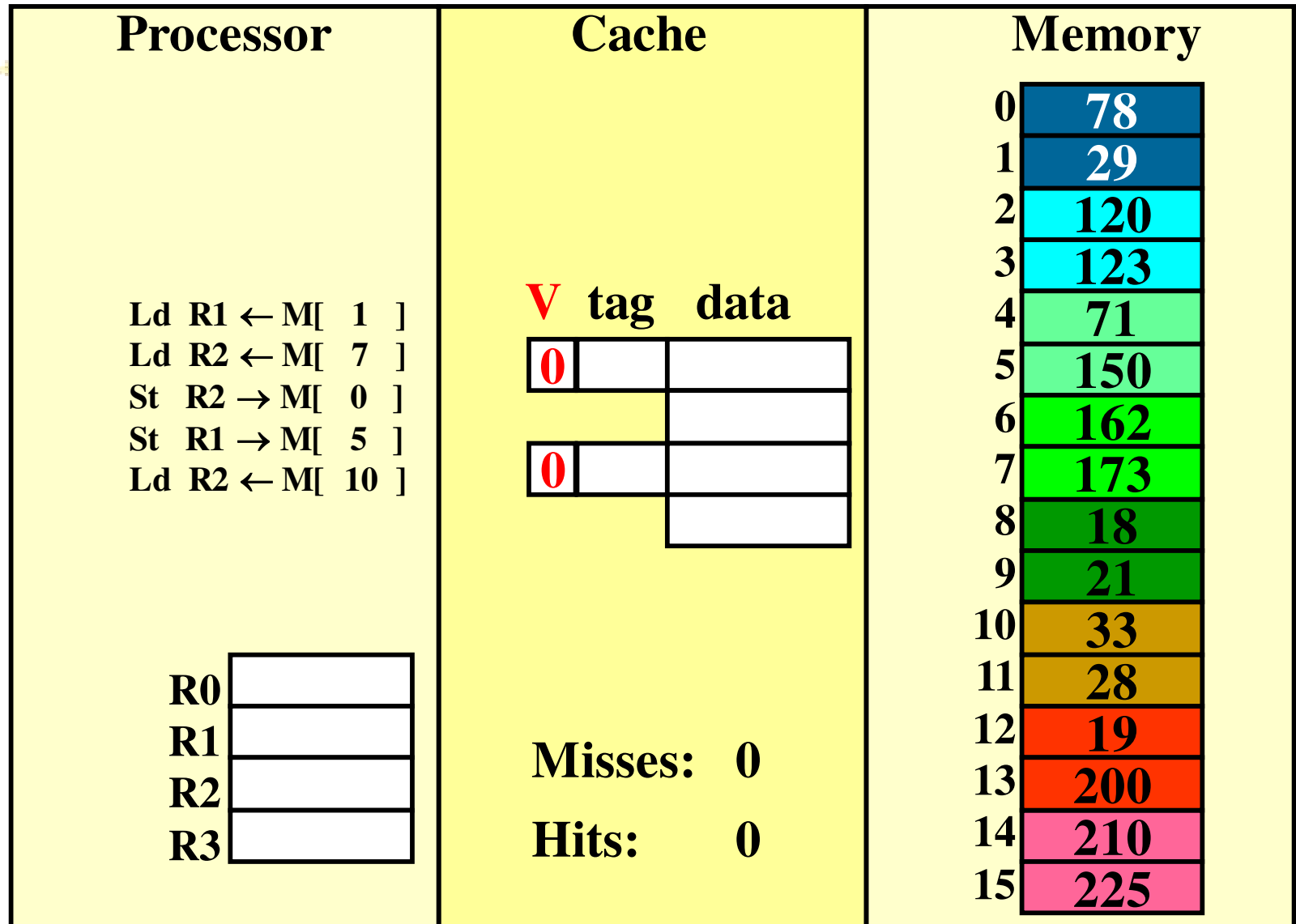
□ Write-Allocate

- Bring written block into cache
- Update word in block
- Anticipate further use of block

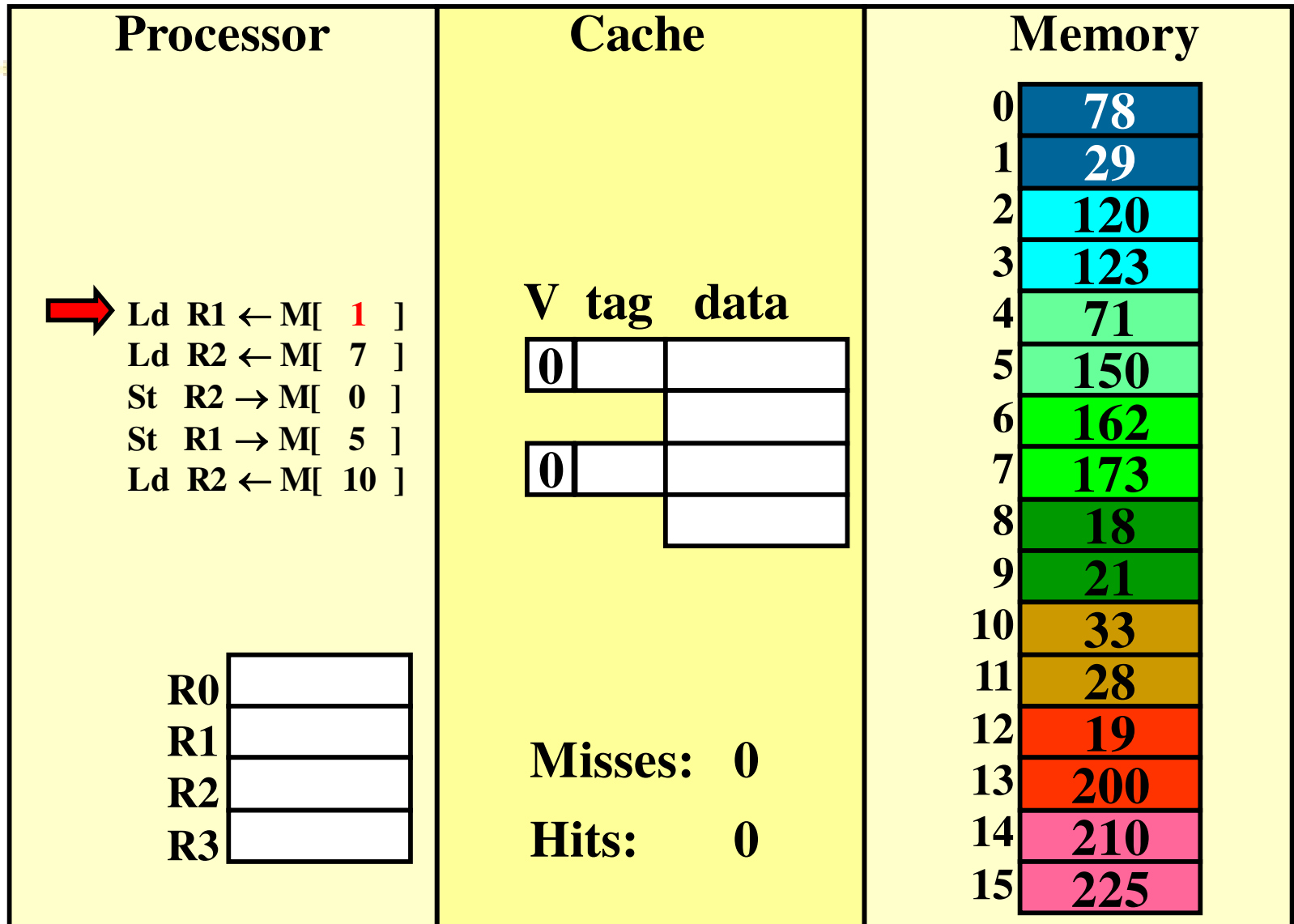
□ No-write Allocate

- Main memory is updated
- Cache contents unmodified

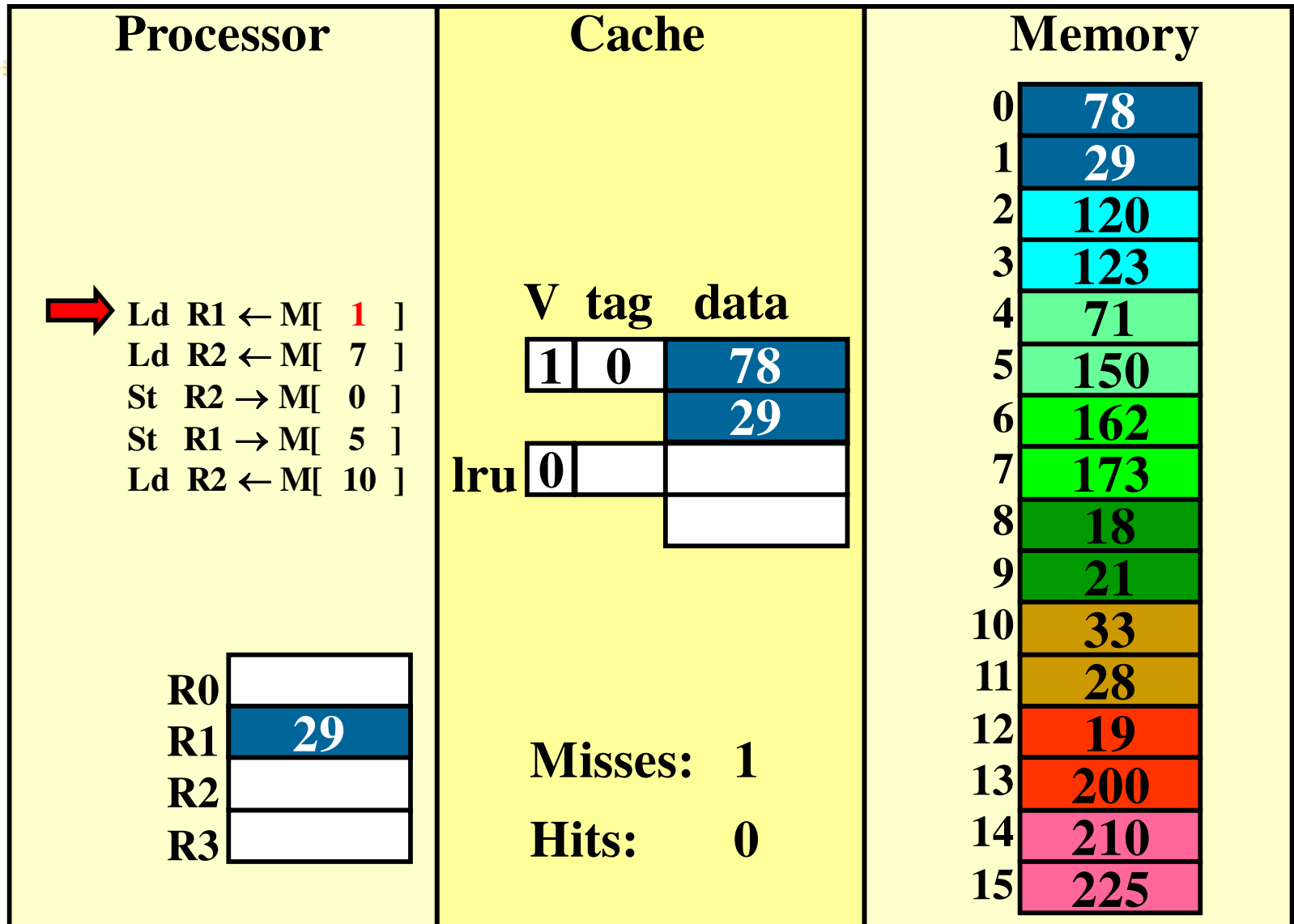
Handling stores (write-through)



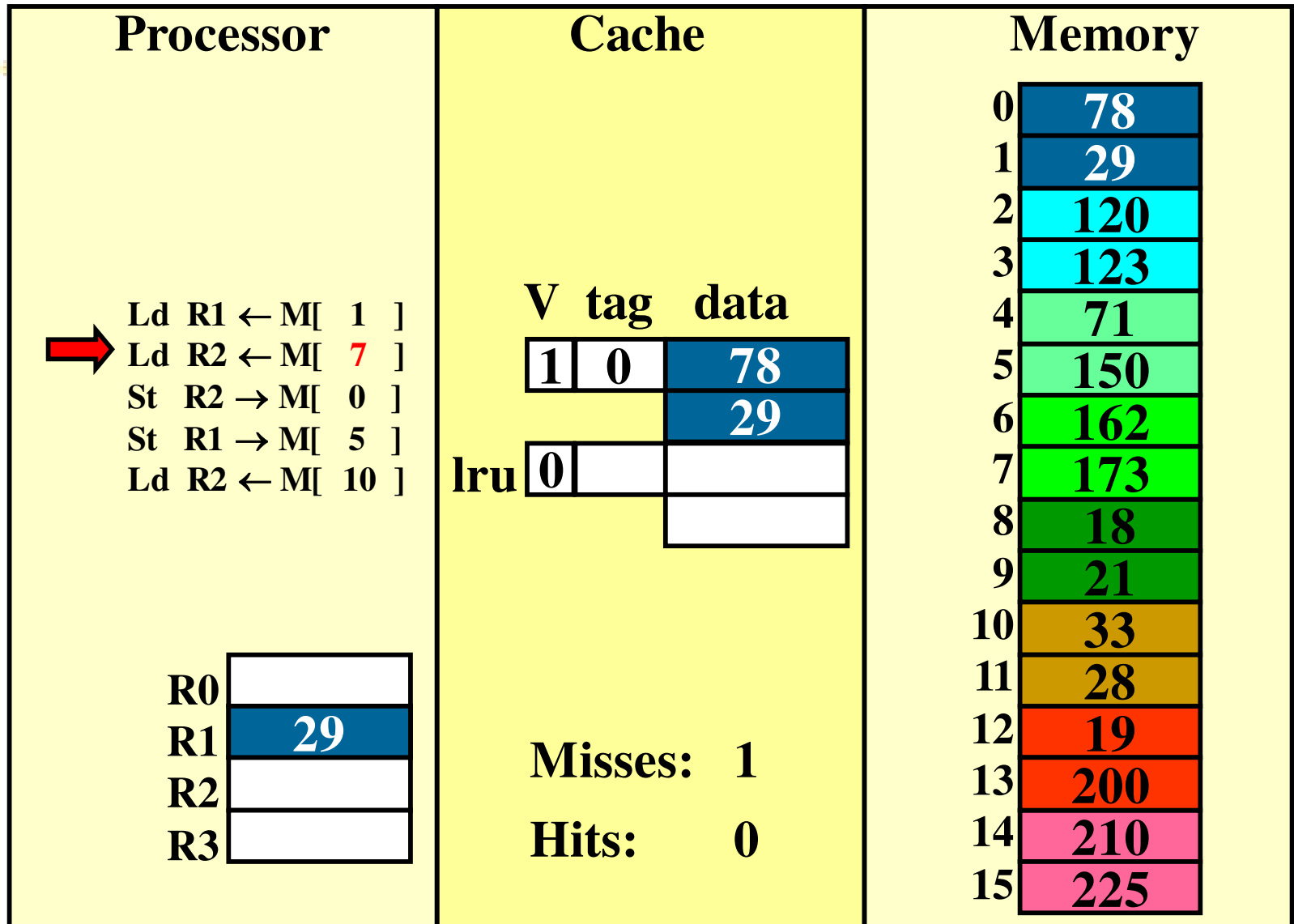
write-through (REF 1)



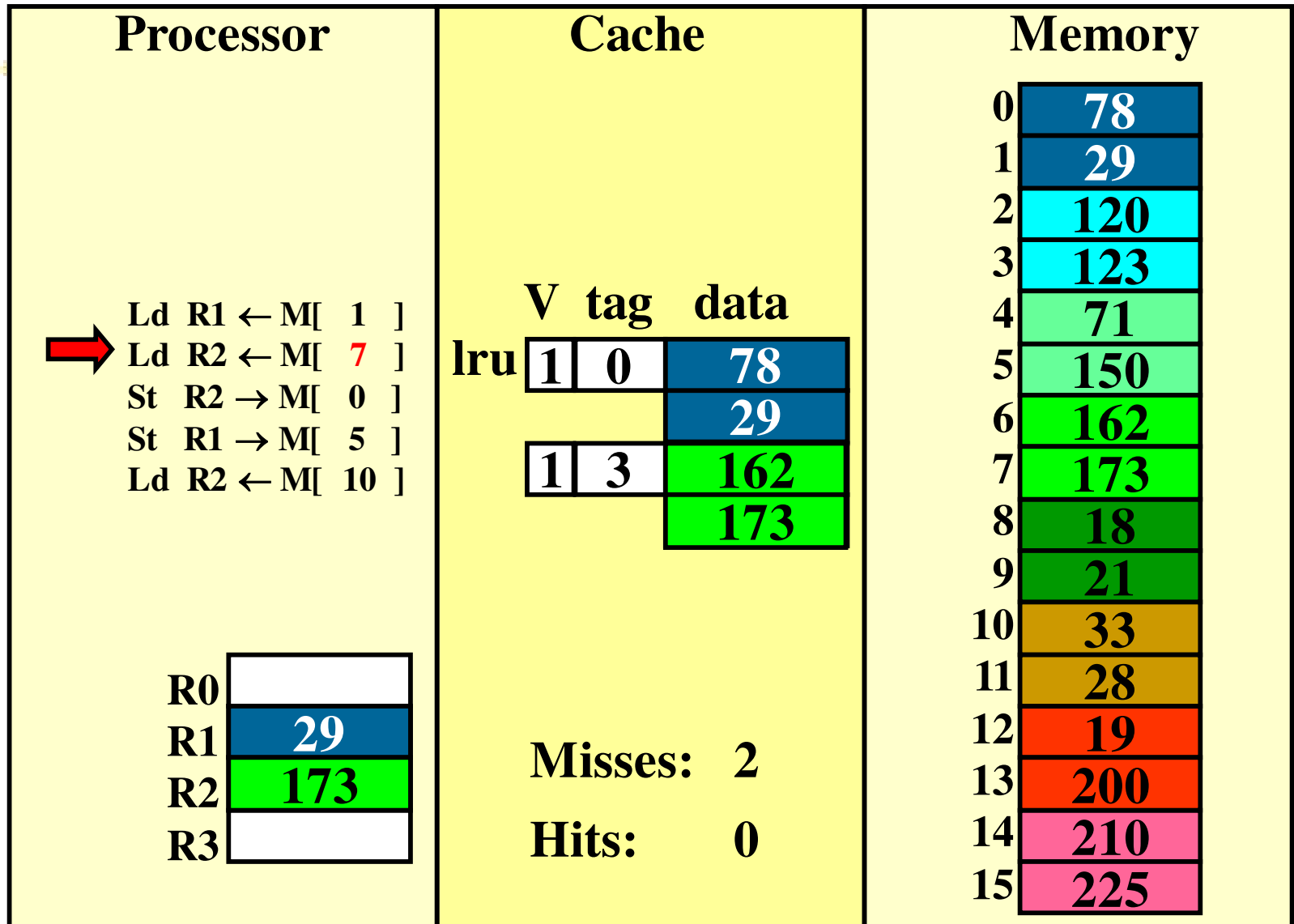
write-through (REF 1)



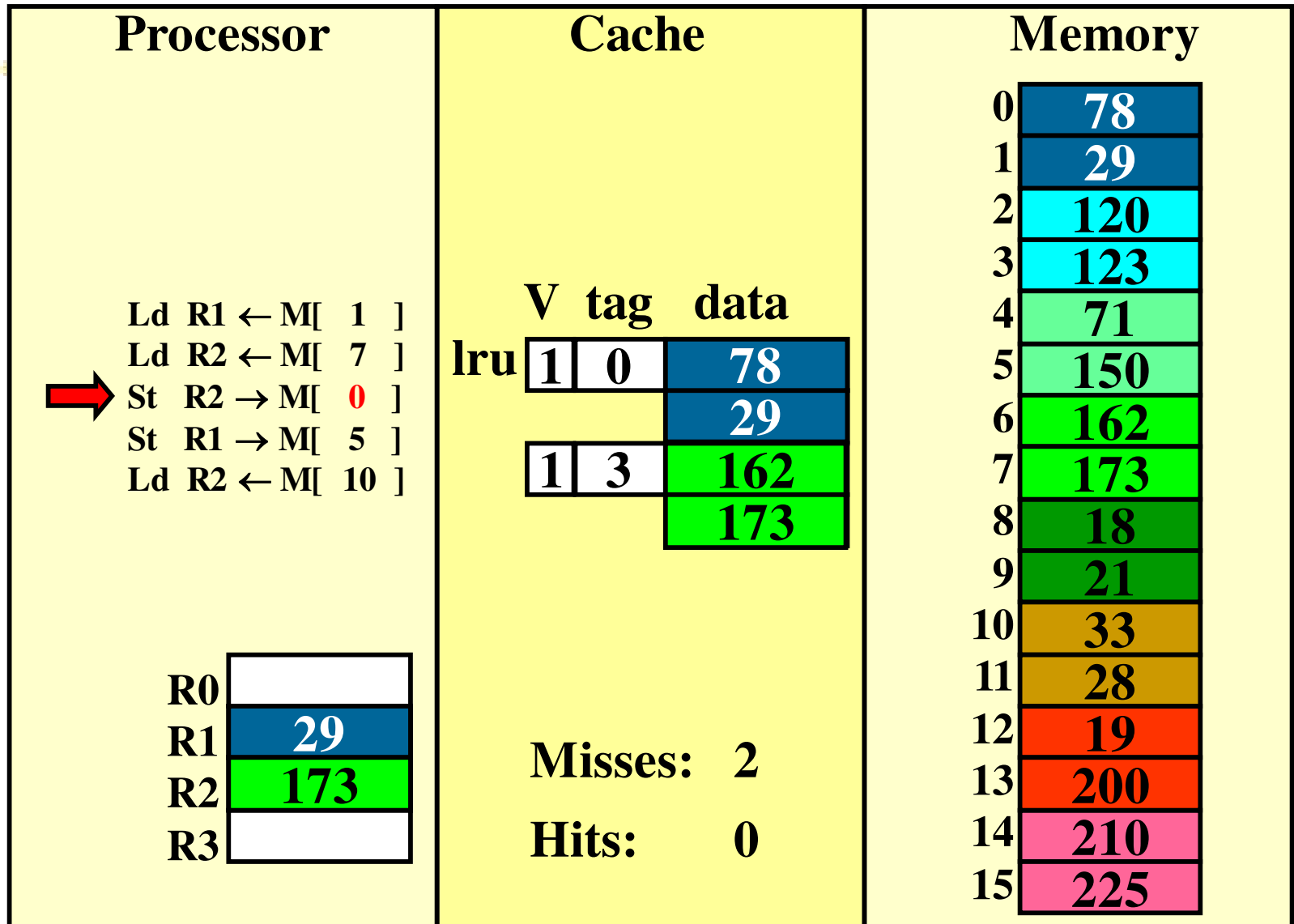
write-through (REF 2)



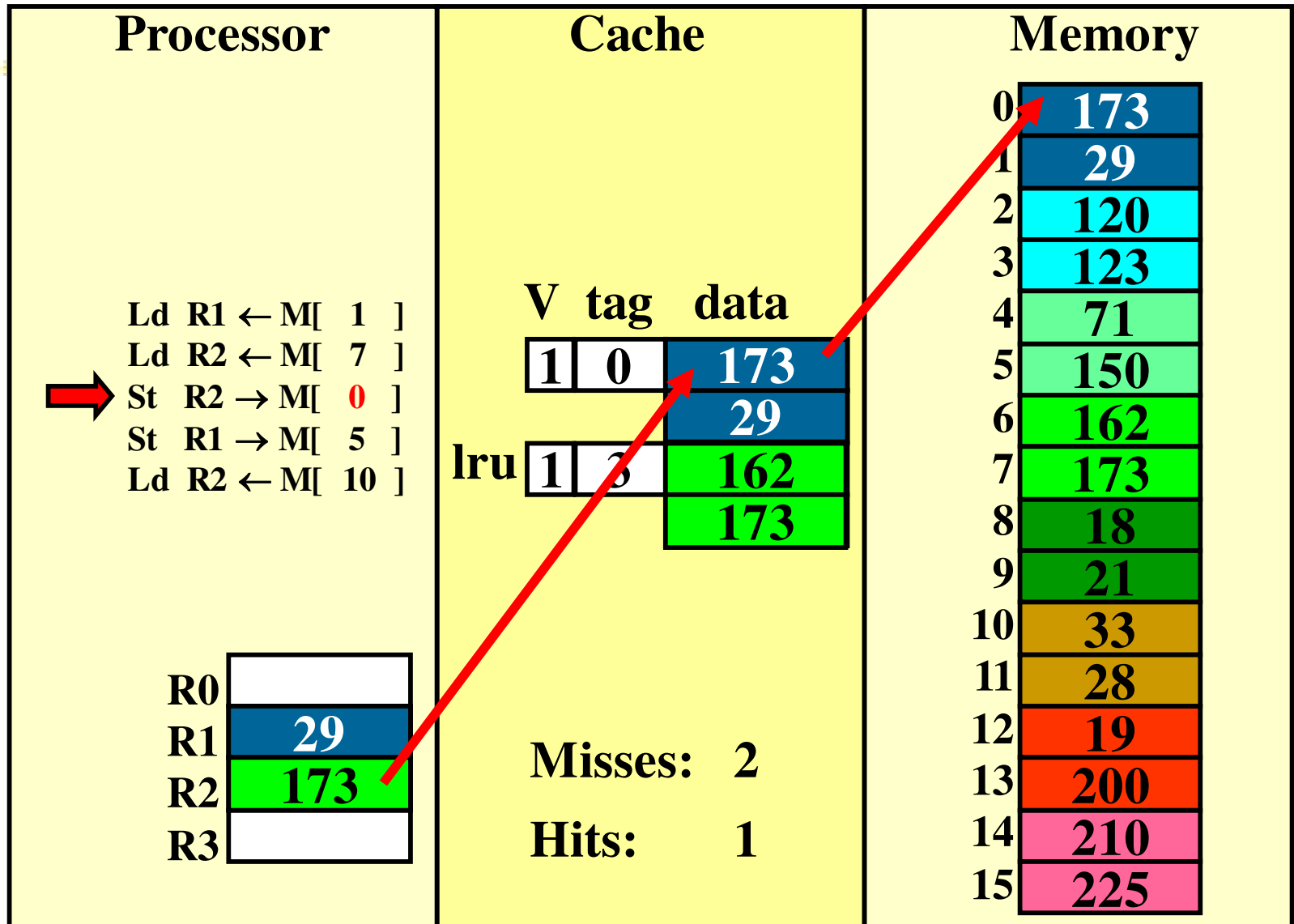
write-through (REF 2)



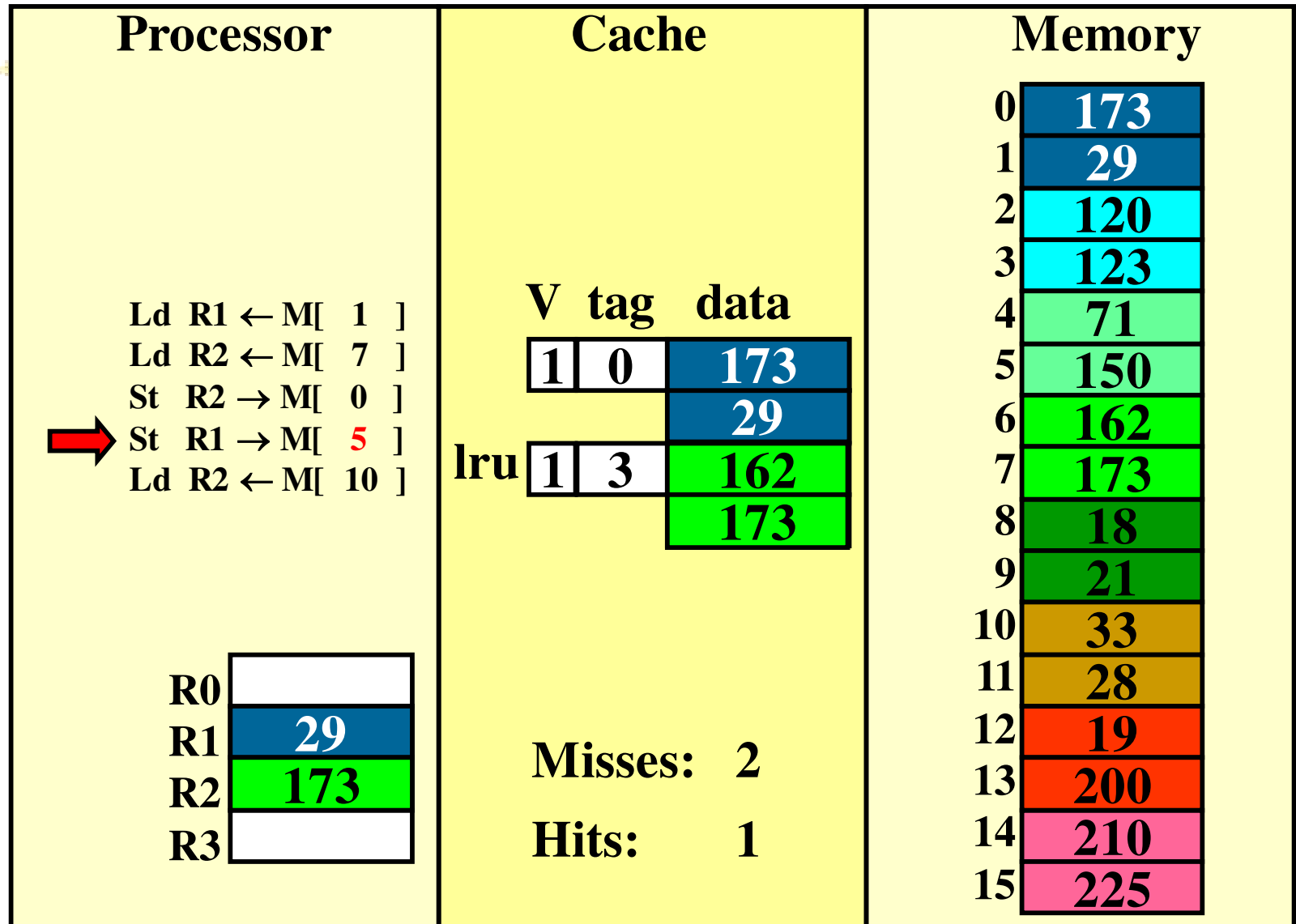
write-through (REF 3)



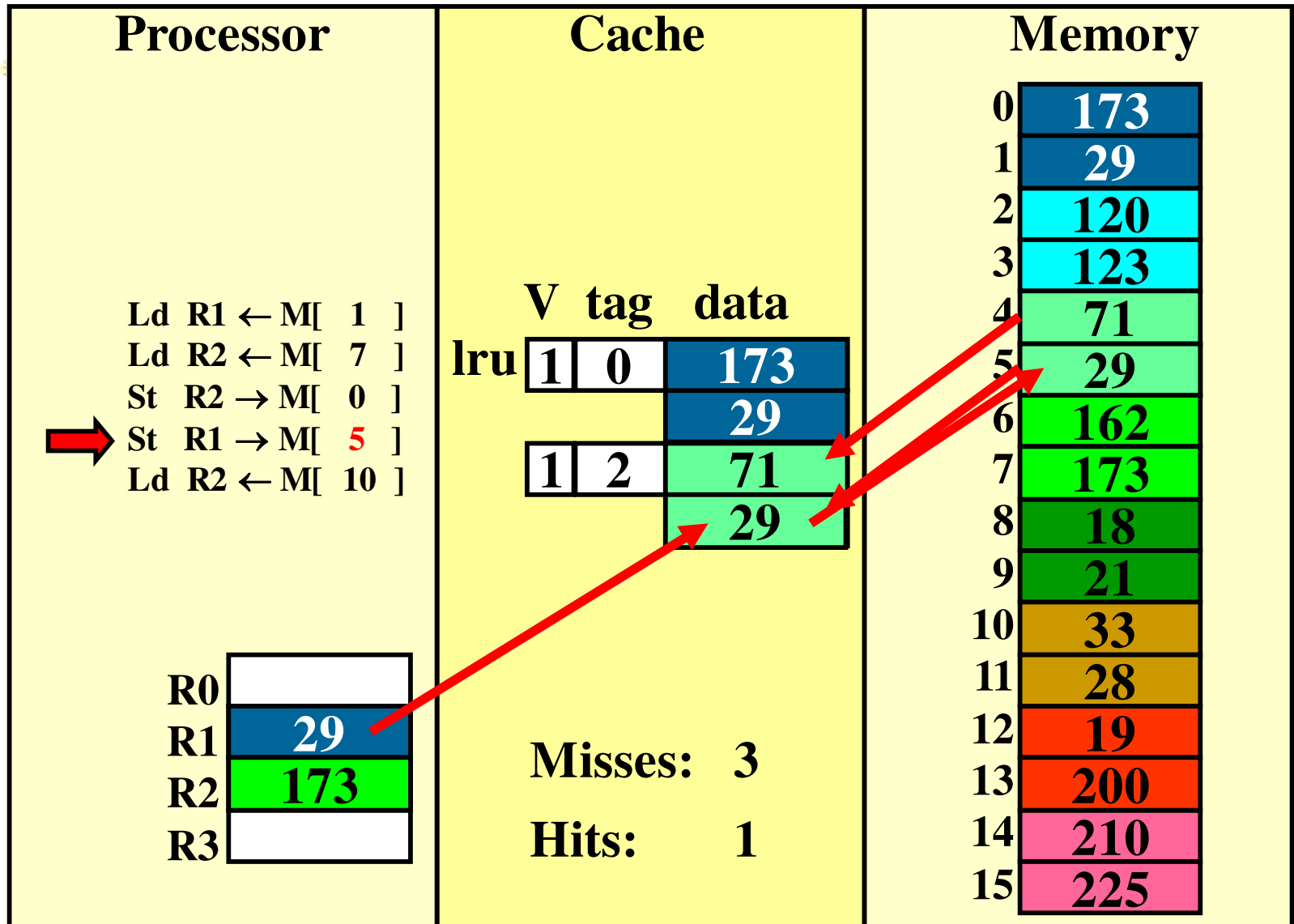
write-through (REF 3)



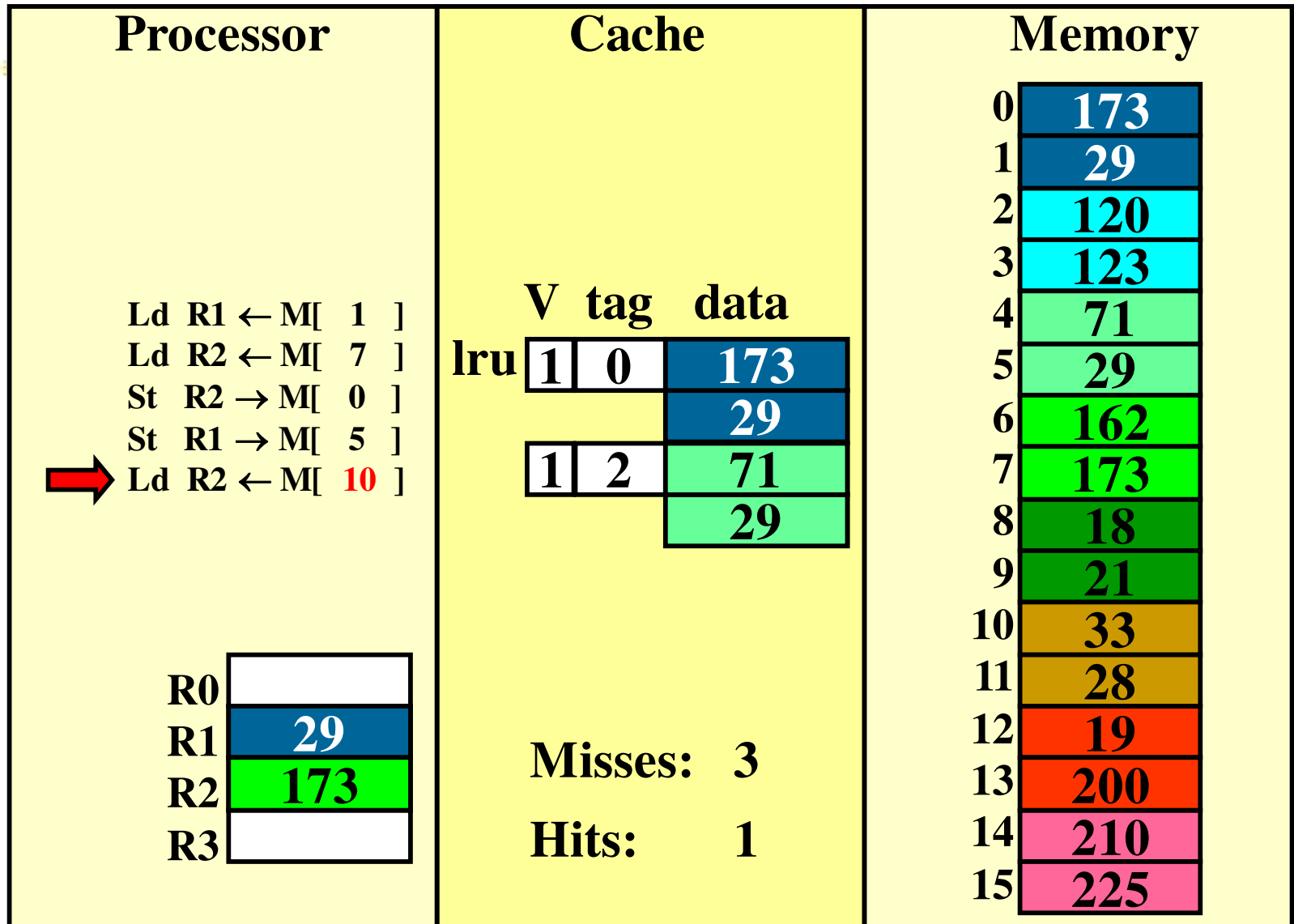
write-through (REF 4)



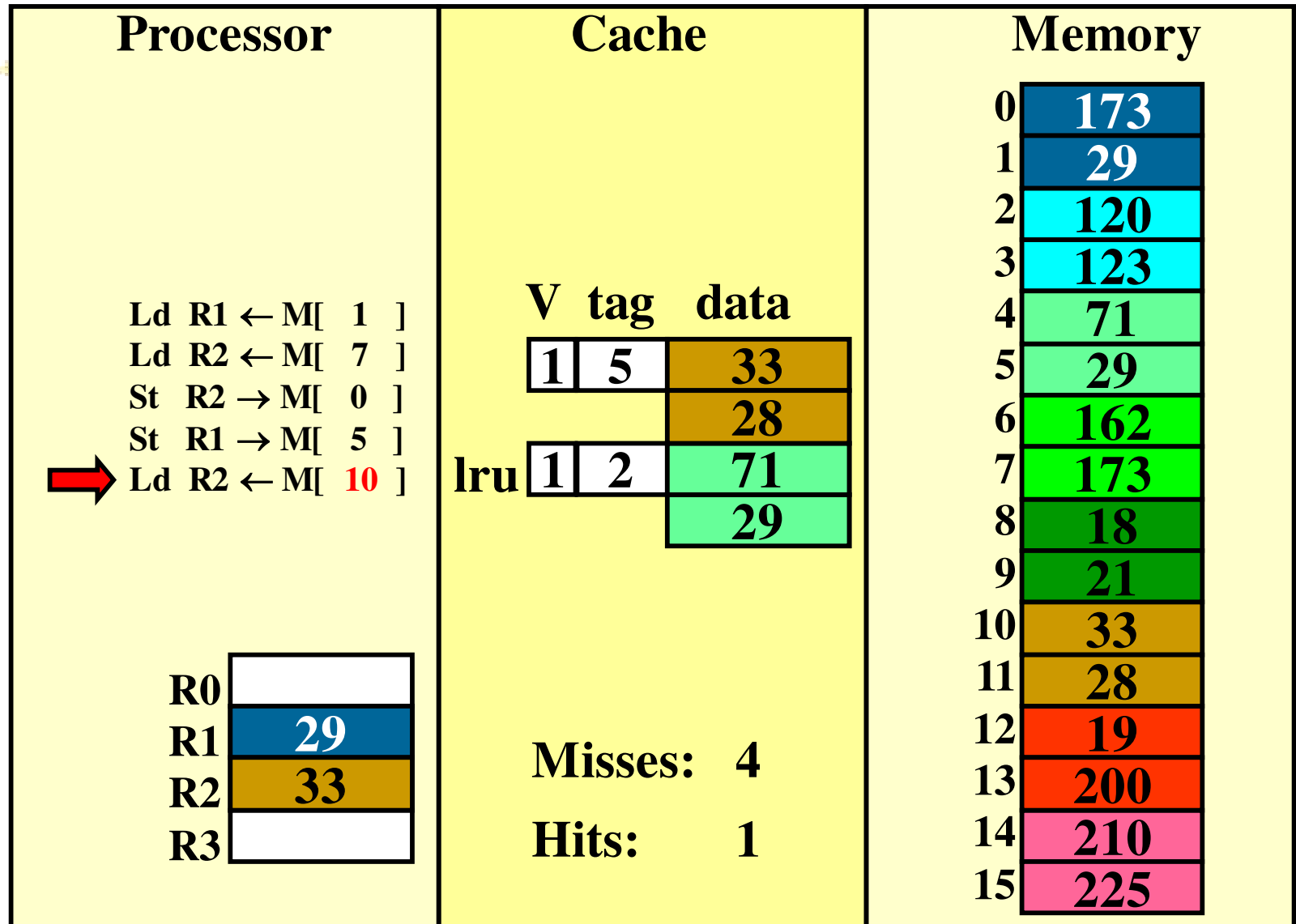
write-through (REF 4)



write-through (REF 5)



write-through (REF 5)



How many memory references?



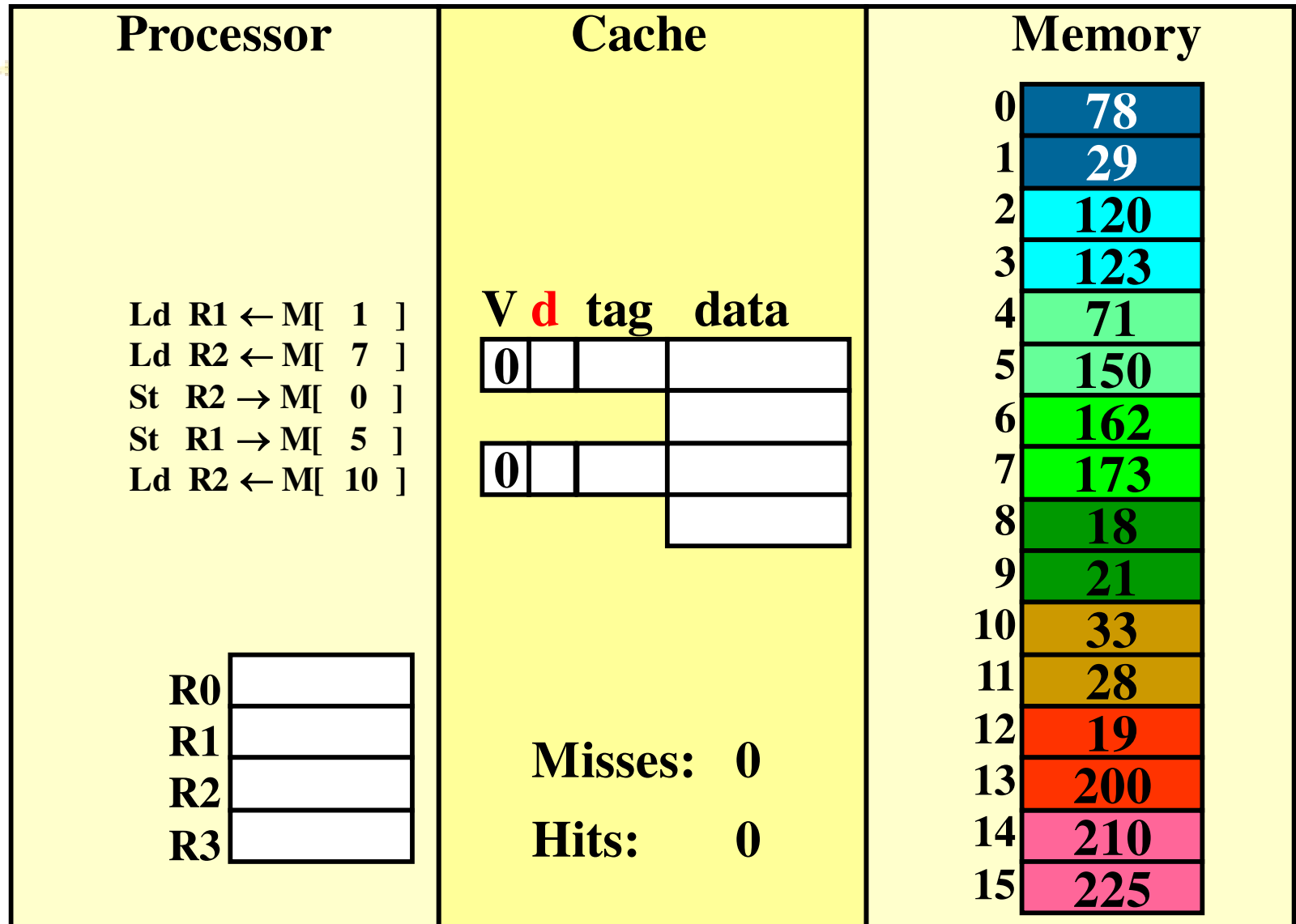
- Each miss reads a block
 - 2 bytes in this cache
- Each store writes a byte
- Total reads: 8 bytes
- Total writes: 2 bytes

but caches generally miss $< 20\%$

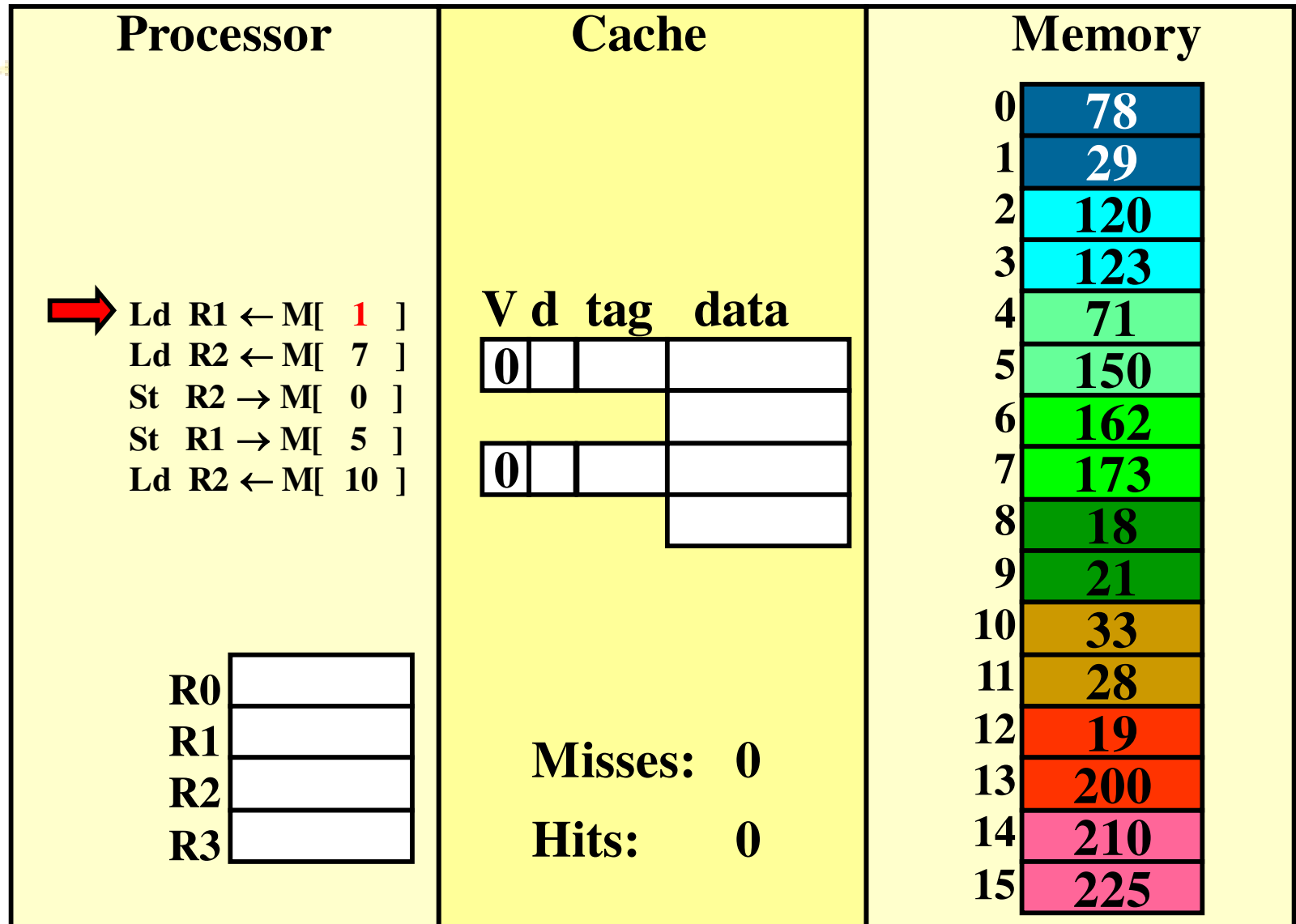
Write-through vs. write-back

- Does the cache **NOT** write all stores to memory immediately?
 - Keep the most current copy in the cache and update the memory when that data is evicted from the cache (a **write-back** policy).
 - write-back all evicted lines?
 - No, only blocks that have been stored into
 - Keep a “**dirty bit**”, reset when the line is allocated, set when the block is stored into. If a block is “dirty” when evicted, write its data back into memory.

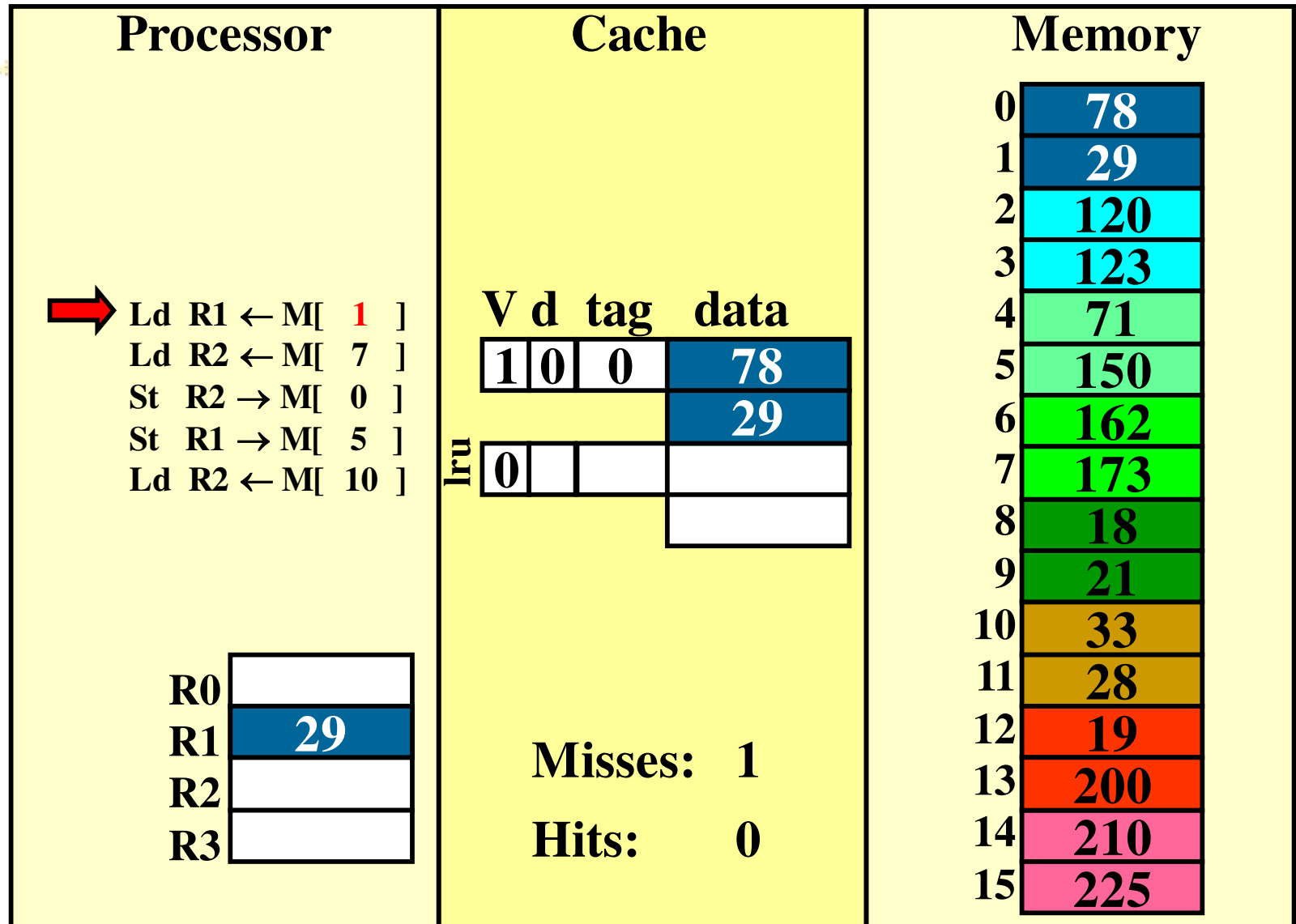
Handling stores (write-back)



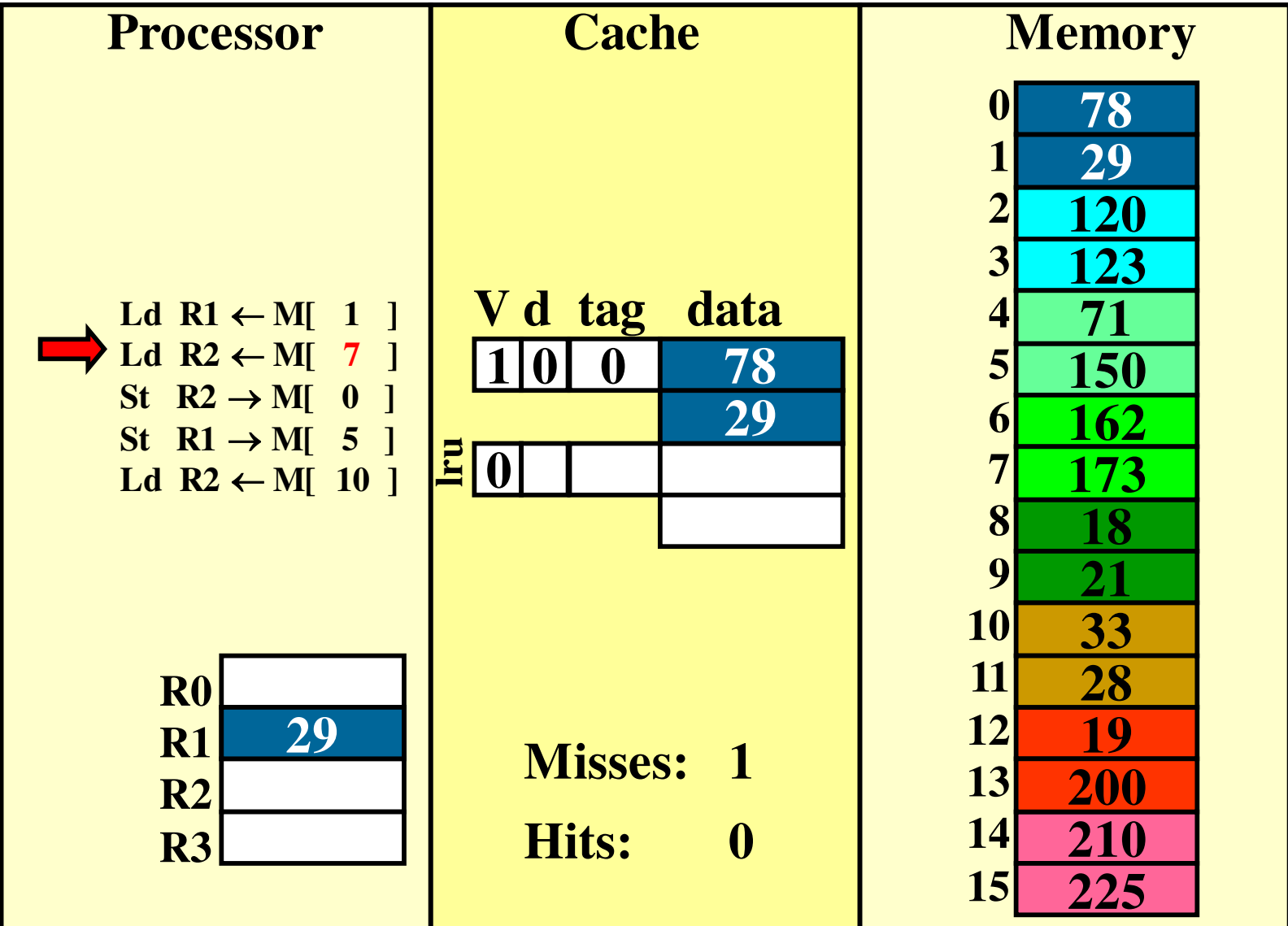
write-back (REF 1)



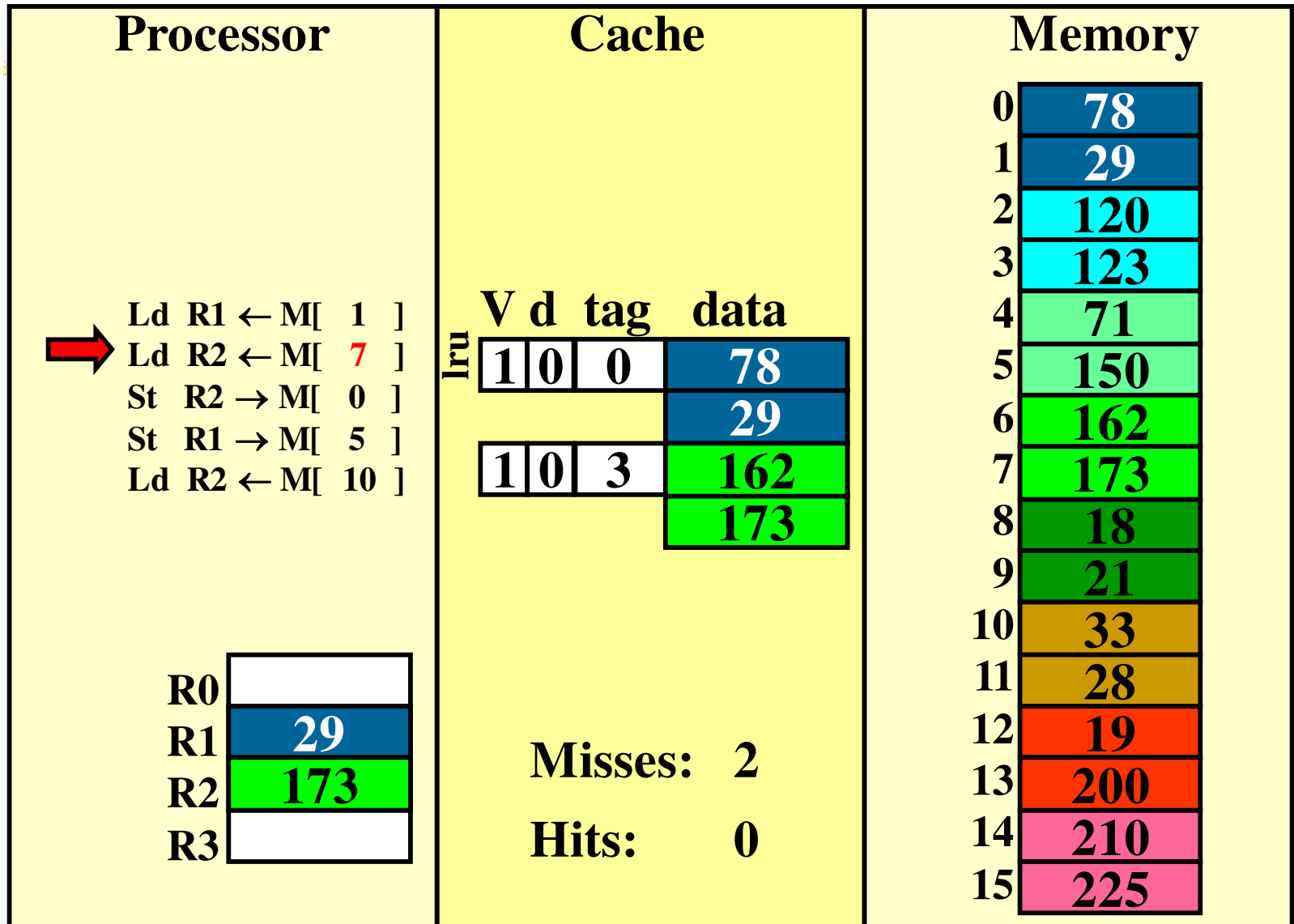
write-back (REF 1)



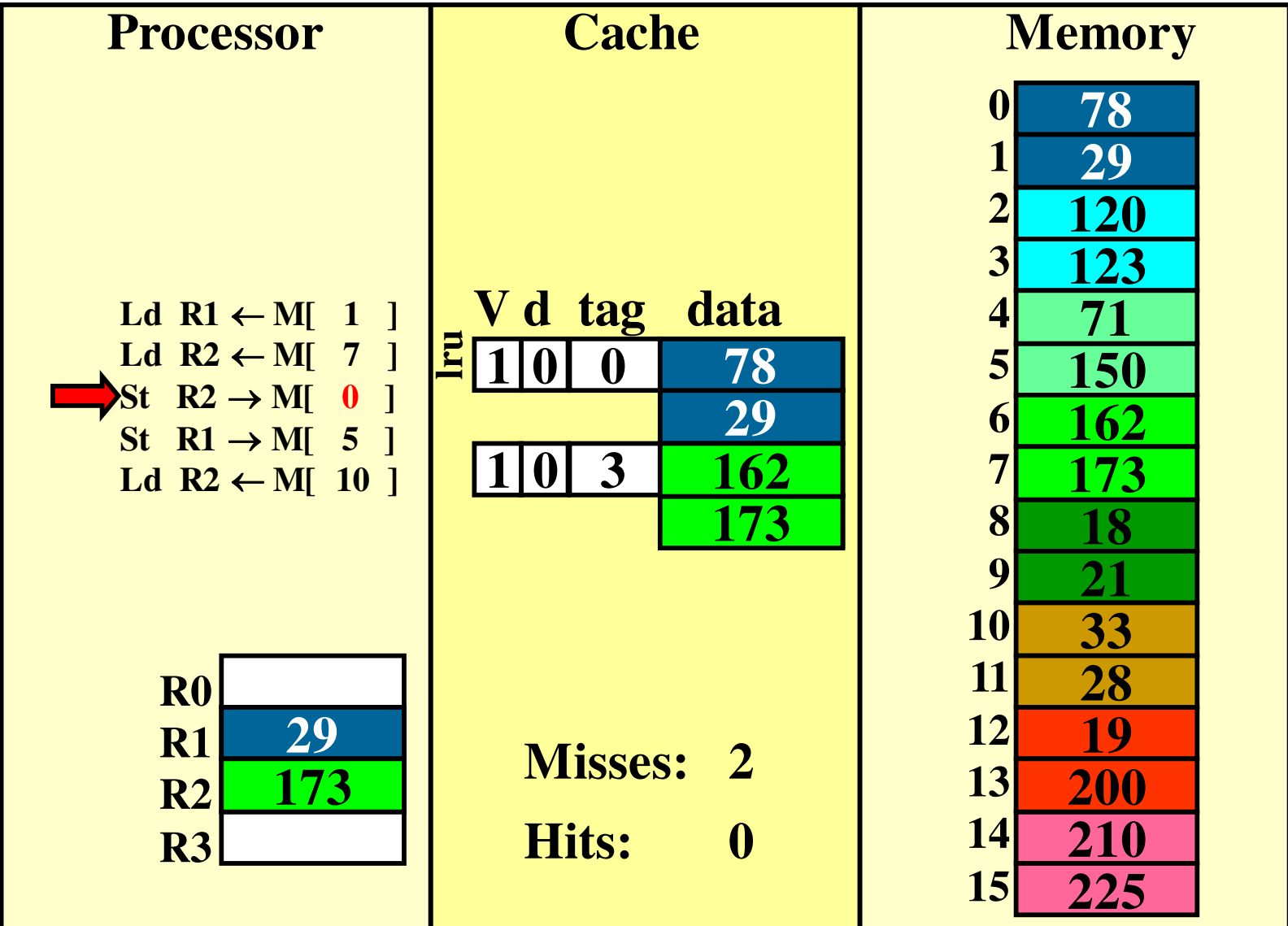
write-back (REF 2)



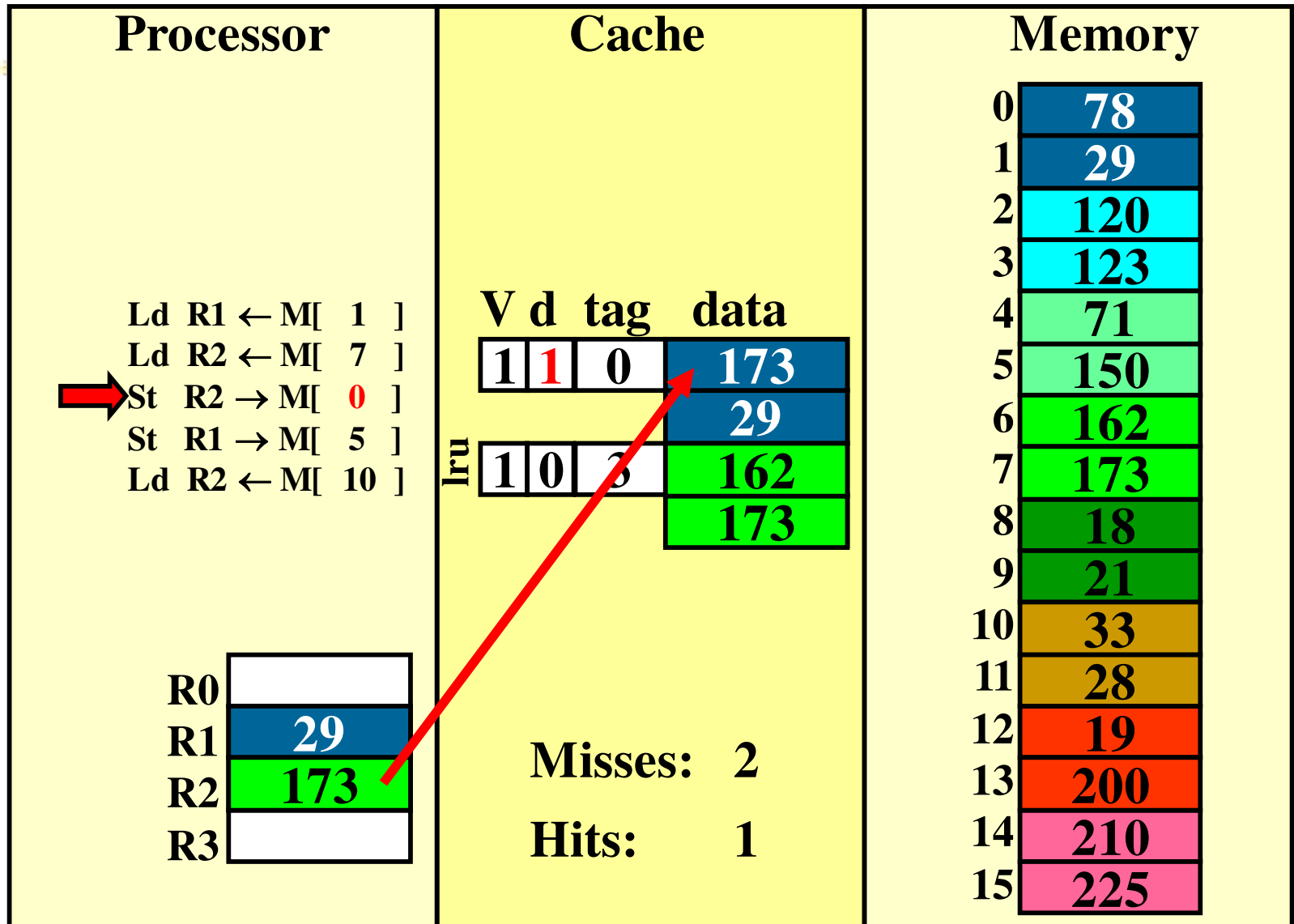
write-back (REF 2)



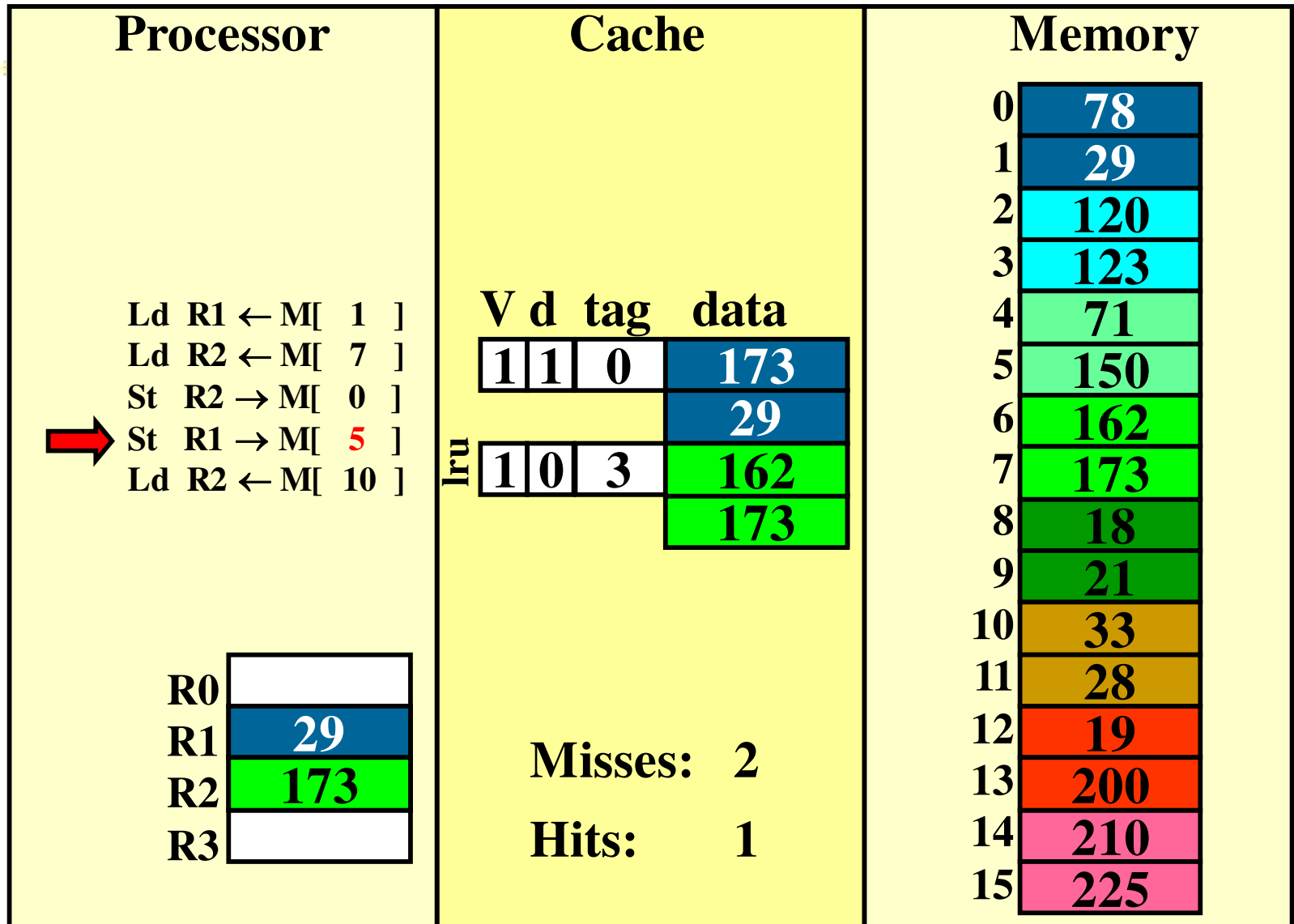
write-back (REF 3)



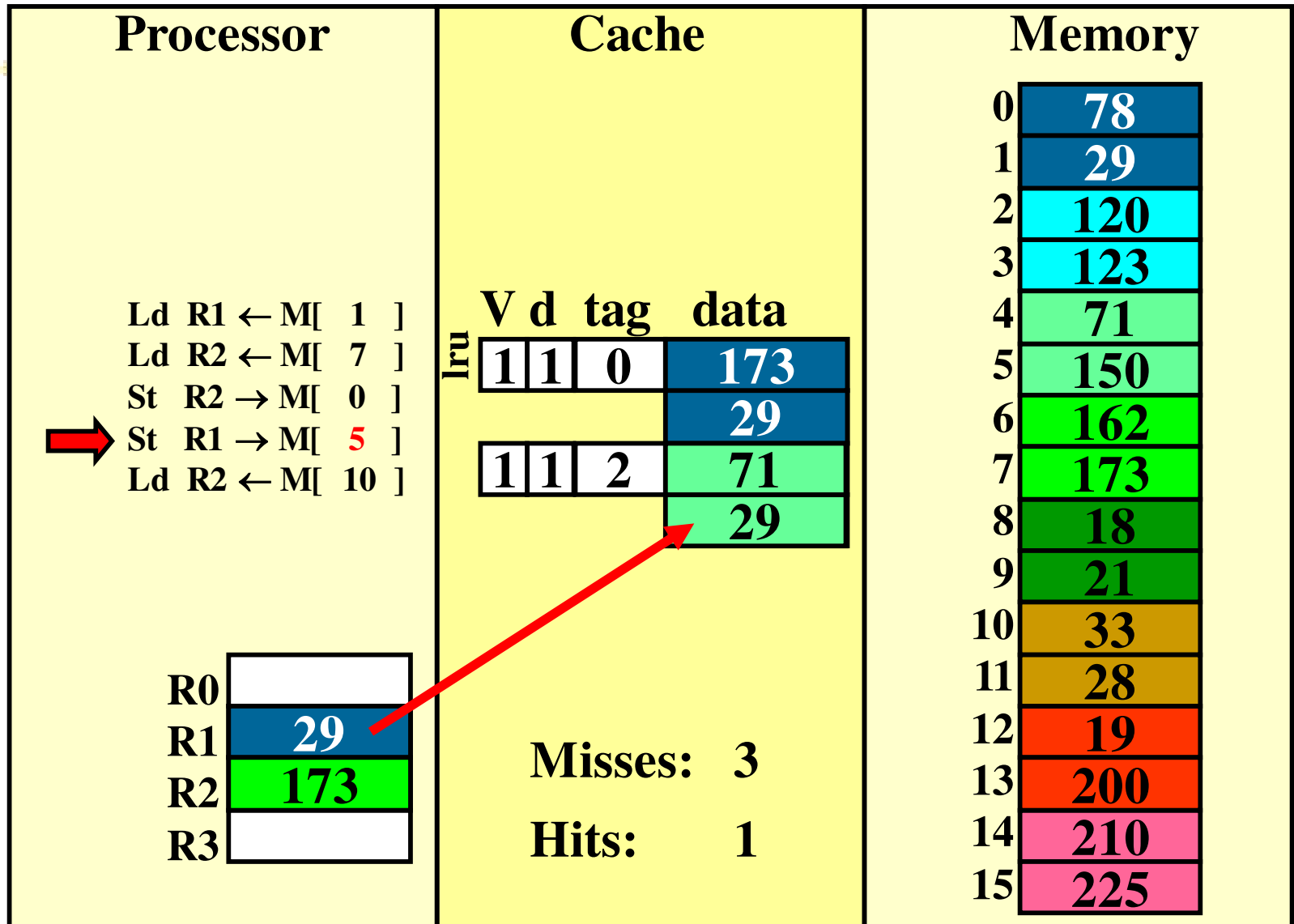
write-back (REF 3)



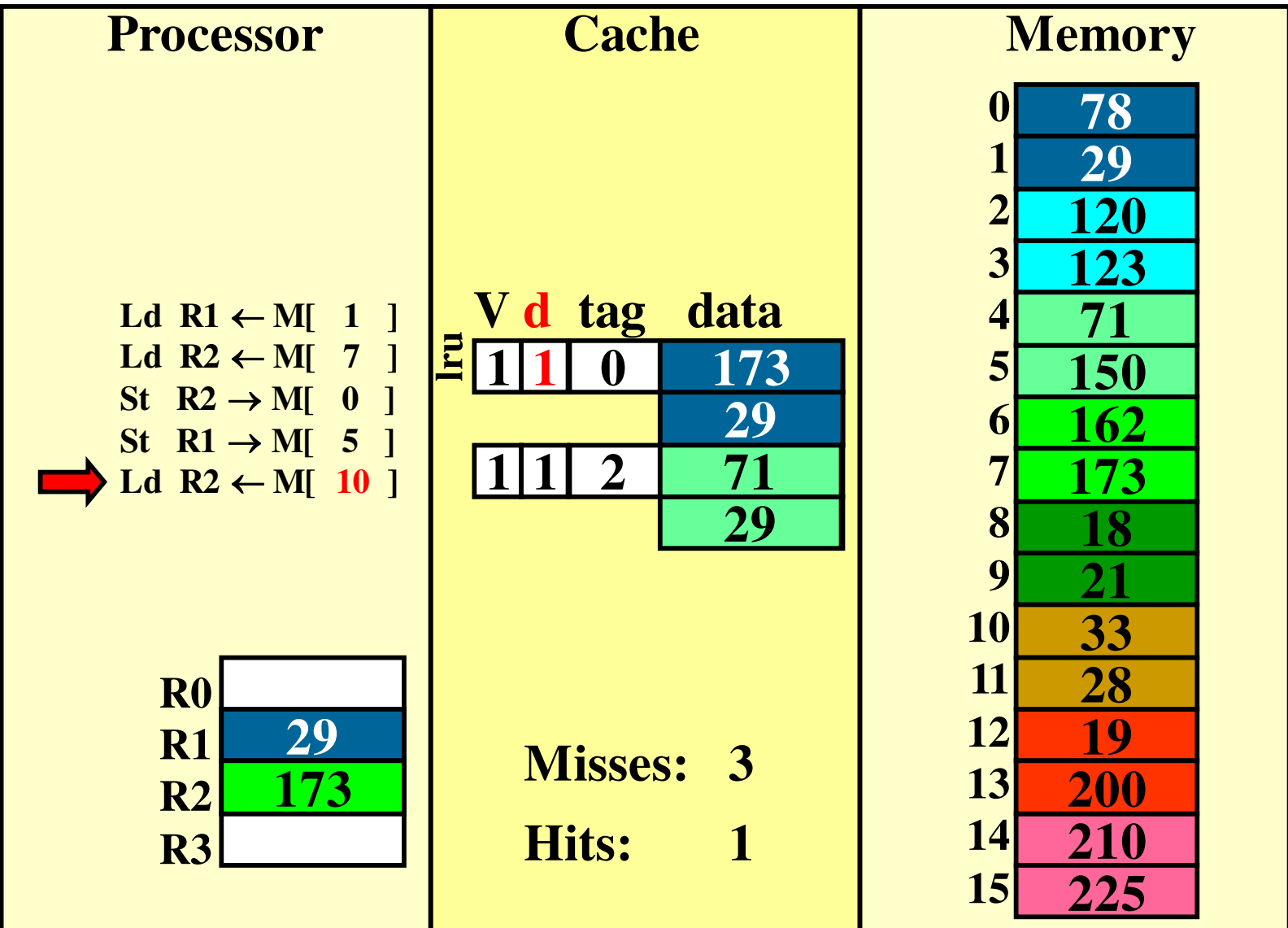
write-back (REF 4)



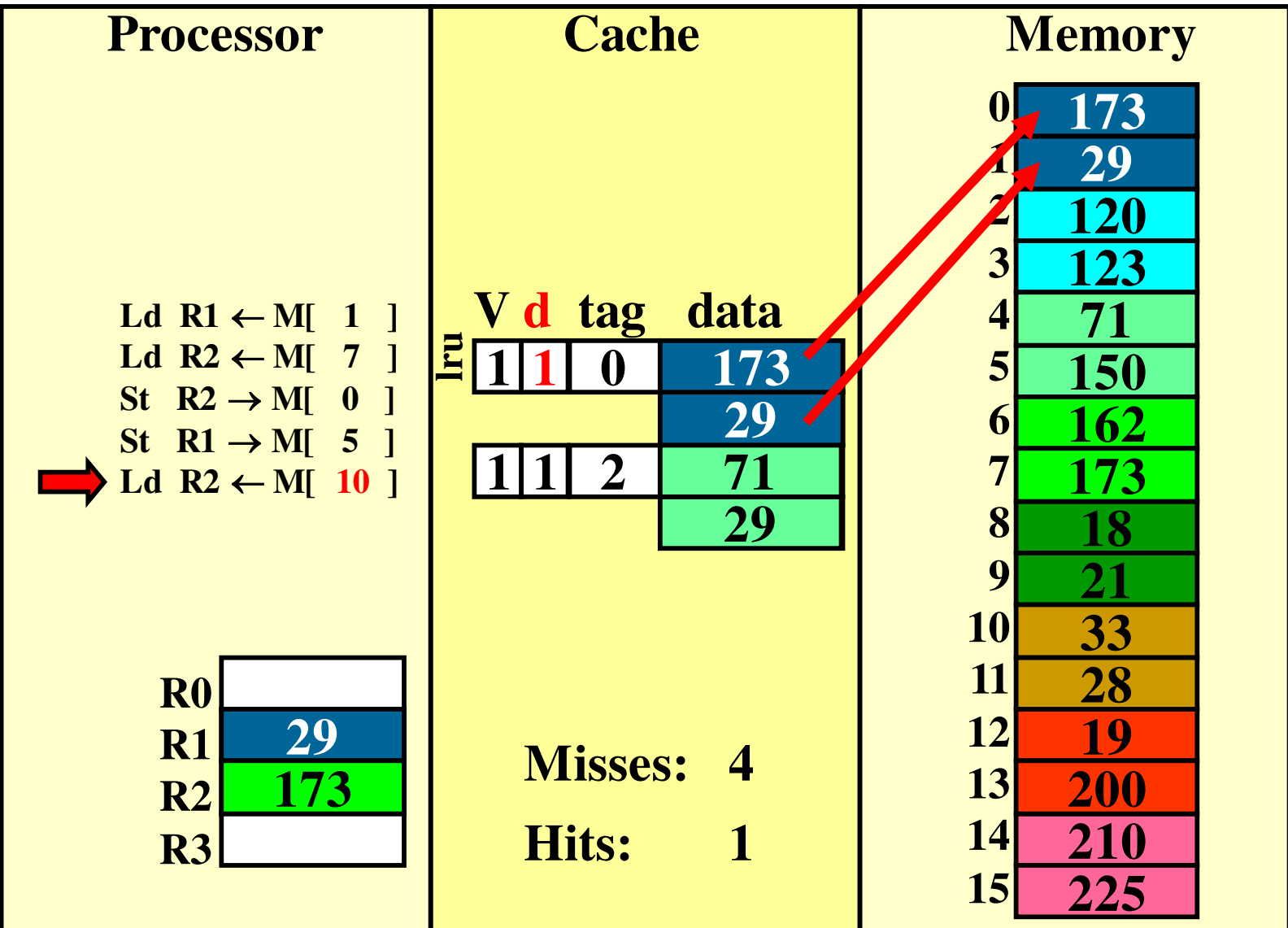
write-back (REF 4)



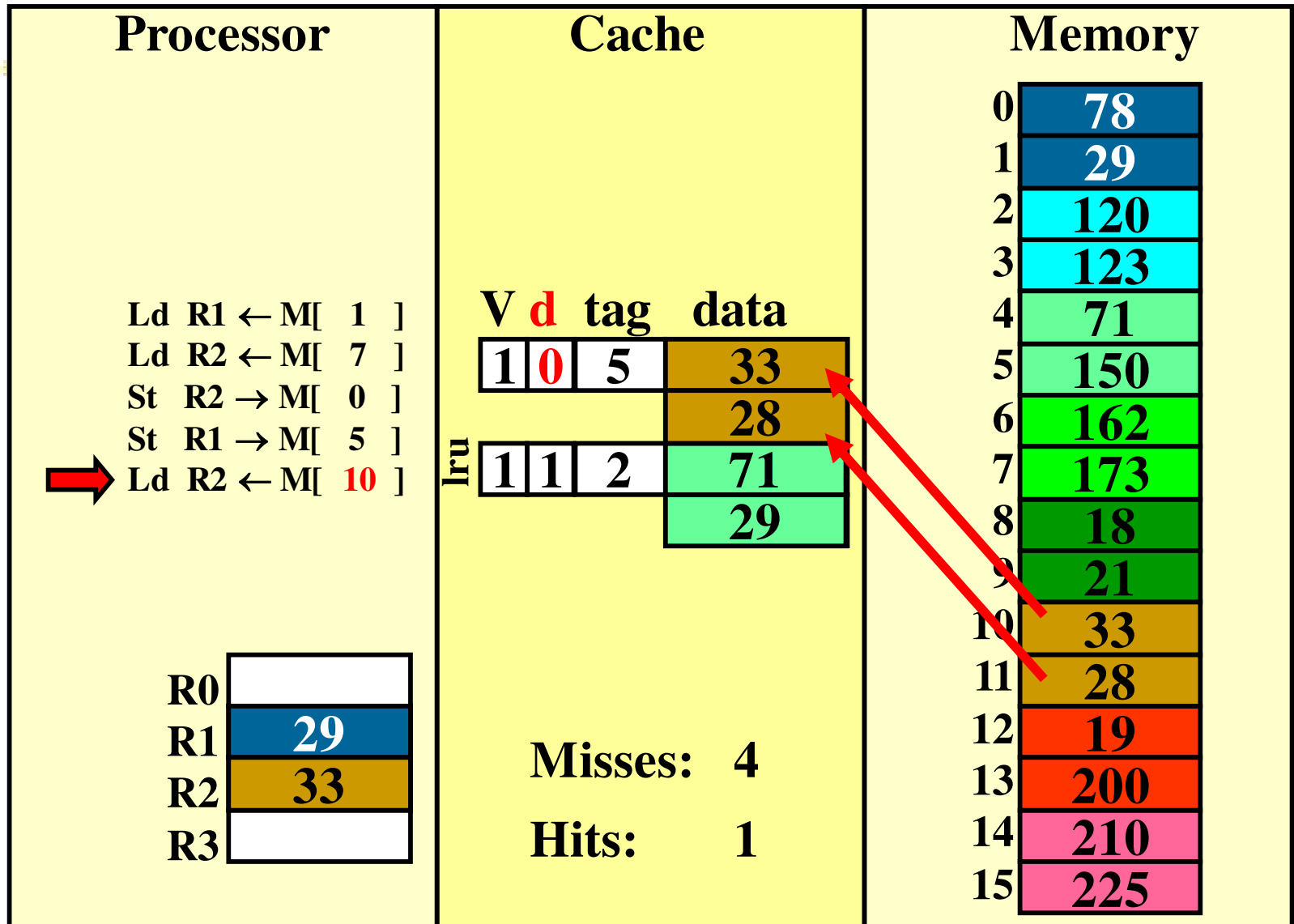
write-back (REF 5)



write-back (REF 5)



write-back (REF 5)



How many memory references?



- Each miss reads a block
 - 2 bytes in this cache
- Each evicted dirty cache line writes a block
- Total reads: 8 bytes
- Total writes: 4 bytes (after final eviction)

Choose write-back or write-through?