CDA 3101

Memory System Introduction

Memory So Far

- We have discussed two structures that hold data:
  - Register file (little array of storage)
  - Memory (bigger array of storage)
- We have discussed several methods of implementing storage devices:
  - Static memory (made with logic gates)
  - ROM, and other ROM-like storage (diodes)

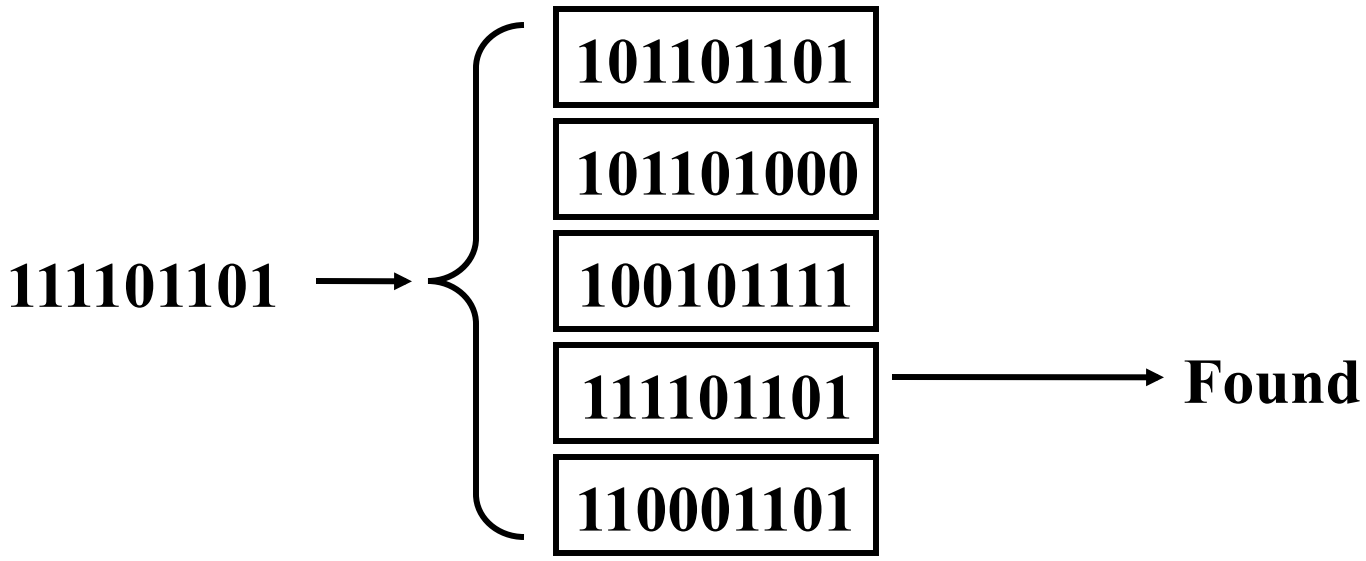
Content Addressable Memories

- Instead of thinking of memory as an array of data indexed by a memory address.
- Think of memory as a set of data matching a query.
  - Instead of an address, we send data to the memory, asking if any location contains that data.

Operations on CAMs

- Search: the primary way to access a CAM
  - Send data to CAM memory
  - Return “found” or “not found”
  - Alternately, return the address of where it was found
- Write:
  - Send data for CAM to remember
    - Where should it be stored if CAM is full?
      - Replacement policy
        - » Replace oldest data in the CAM
        - » Replace least recently searched data

CAM Array



5 storage element CAM array of 9 bits each

Previous Use of CAMs

- You have seen a simple CAM used before, when?

Memory Hierarchy

- We want to have lots of memory for our processor:
  - LC3101 needs 2^{16} words of memory
  - MIPS needs 2^{32} bytes of memory
  - x86-64 needs 2^{64} bytes of memory
- What are our choices?
  - SRAM, DRAM, Disk, paper?

Option 1: Build It Out of Fast SRAM

- About 5-10 ns access
  - Decoders are big
  - Array are big
- It will cost LOTS of money
  - SRAM costs $7 per megabyte
    - $0.43 for LC3101
    - $28,672 for MIPS
    - $123 trillion for x86-64

Option 2: Build It Out of DRAM

- DRAM: save bit in a capacitor
- About 100 ns access
  - Why build a fast processor that stalls for dozens of cycles on each memory load?
- Still costs lots of money for new machines
  - DRAM costs $68 per gigabyte
    - <$0.01 for LC3101
    - $272 for MIPS
    - $1 trillion for x86-64

Option 3: Build It Using Disks

- About 10,000,000 ns access (snore!)
  - We could have stopped with the Intel 4004
- Costs are pretty reasonable
  - Disk storage costs $0.25 per gigabyte
    - Basically free for LC3101
    - $1 for MIPS
    - $4 billion for x86-64 (ouch!)

Our Requirements

- We want a memory system that runs a processor clock speed (about 1 ns access)
- We want a memory system that we can afford (maybe 25% to 33% of the total system costs).
- Options 1-3 are too slow
- Options 1 (or 1-3) are too expensive

Time for option 4!

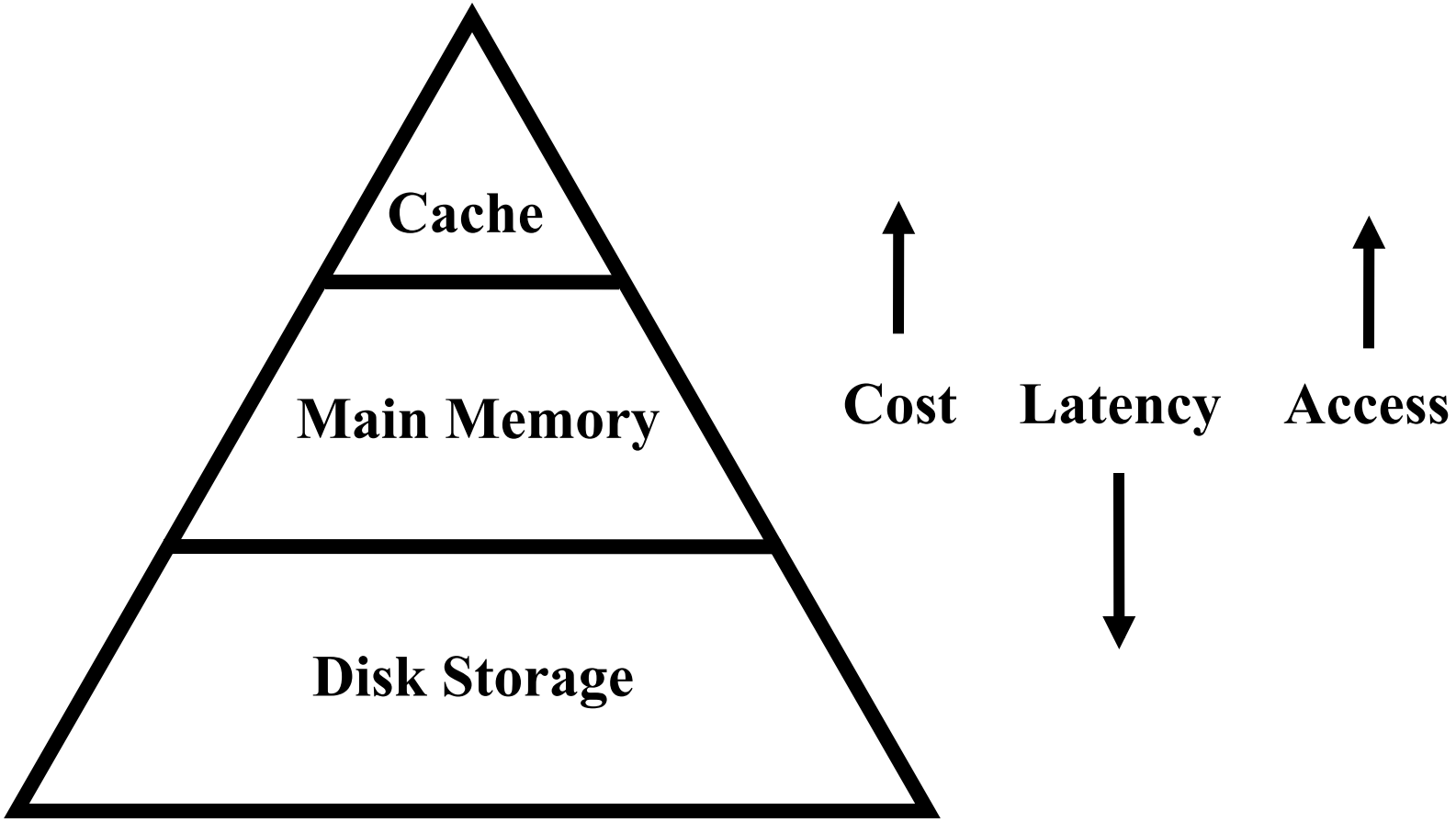
Option 4: Use a Little of Everything (Wisely)

- Use a small array of SRAM
  - Small means fast!
  - Small means cheap!
- Use a larger amount of DRAM
  - And hope that you rarely have to use it
- Use a really big amount of Disk storage
  - Disks are getting cheaper at a faster rate than we fill them up with data (for most people).
- Don't try to buy 2^{64} bytes of anything
  - It would take decades to format it anyway!

Option 4: The Memory Hierarchy

- Use a small array of SRAM
  - For the **CACHE** (hopefully for most accesses)
- Use a larger amount of DRAM
  - For the **Main memory**
- Use a really large amount of Disk storage
  - For the **Virtual memory**
- Don't try to buy 2^{64} bytes of anything
  - At least for a while!

Famous Picture of ~~Food~~ Memory Hierarchy



Rehashing our Terms

- The **Architectural** view of memory is:
  - What the machine language sees
  - Memory is just a big array of storage
- Breaking up the memory system into different pieces – cache, main memory (made up of DRAM) and Disk storage – is **not architectural**.
  - The machine language doesn't know about it
  - An new implementation may not break it up into the same pieces (or break it up at all).

Function of the Cache

- The cache will hold the data that we think is most likely to be referenced.
  - Because we want to maximize the number of references that are serviced by the cache to minimize the **average memory access latency**
  - How do we decide what the most likely accessed memory location are??

Cache Analogy

- Hungry! must eat!
  - Option 1: go to refrigerator
    - Found τηεν eat!
    - Latency = 1 minute
  - Option 2: go to store
    - Found τηεν purchase, take home, eat!
    - Latency = 20-30 minutes
  - Option 3: grow food!
    - Plant, wait … wait … wait … , harvest, eat!
    - Latency = ~250,000 minutes (or ~ 6 months)

Class Problem 1

Given the following:

Cache: 1 cycle access time

Main memory: 100 cycle access time

Disk: 10000 cycles access time

What is the average access time for 100 memory references if you measure that 90% of the cache accesses are hits and 80% of the accesses to main memory are hits.

Basic Cache Design

- Cache memory can copy data from any part of main memory
  - It has 2 parts:
    - The TAG (CAM) holds the memory address
    - The BLOCK (SRAM) holds the memory data
- Accessing the cache:
  - Compare the reference address with the tag
    - If they match, get the data from the cache block
    - If the don't match, get the data from main memory

Cache Organization

- A cache memory consists of multiple tag/block pairs (called cache lines)
  - Searches can be done in parallel (within reason)
  - At most one tag will match
- If there is a tag match, it is a cache HIT
- If there is no tag match, it is a cache MISS

Our goal is to keep the data we think will be accessed in the near future in the cache

Cache Operation

- Every cache miss will get the data from memory and ALLOCATE a cache line to put the data in.
  - Just like any CAM write
- Which line should be allocated?
  - Random? OK, but hard to grade test questions
  - Better than random? How?

Picking the Most Likely Addresses

- What is the probability of accessing a random memory location?
  - With no information, it is just as likely as any other address
- But programs are not random
  - They tend to use the same memory locations over and over.
  - We can use this to pick the most referenced locations to put into the cache

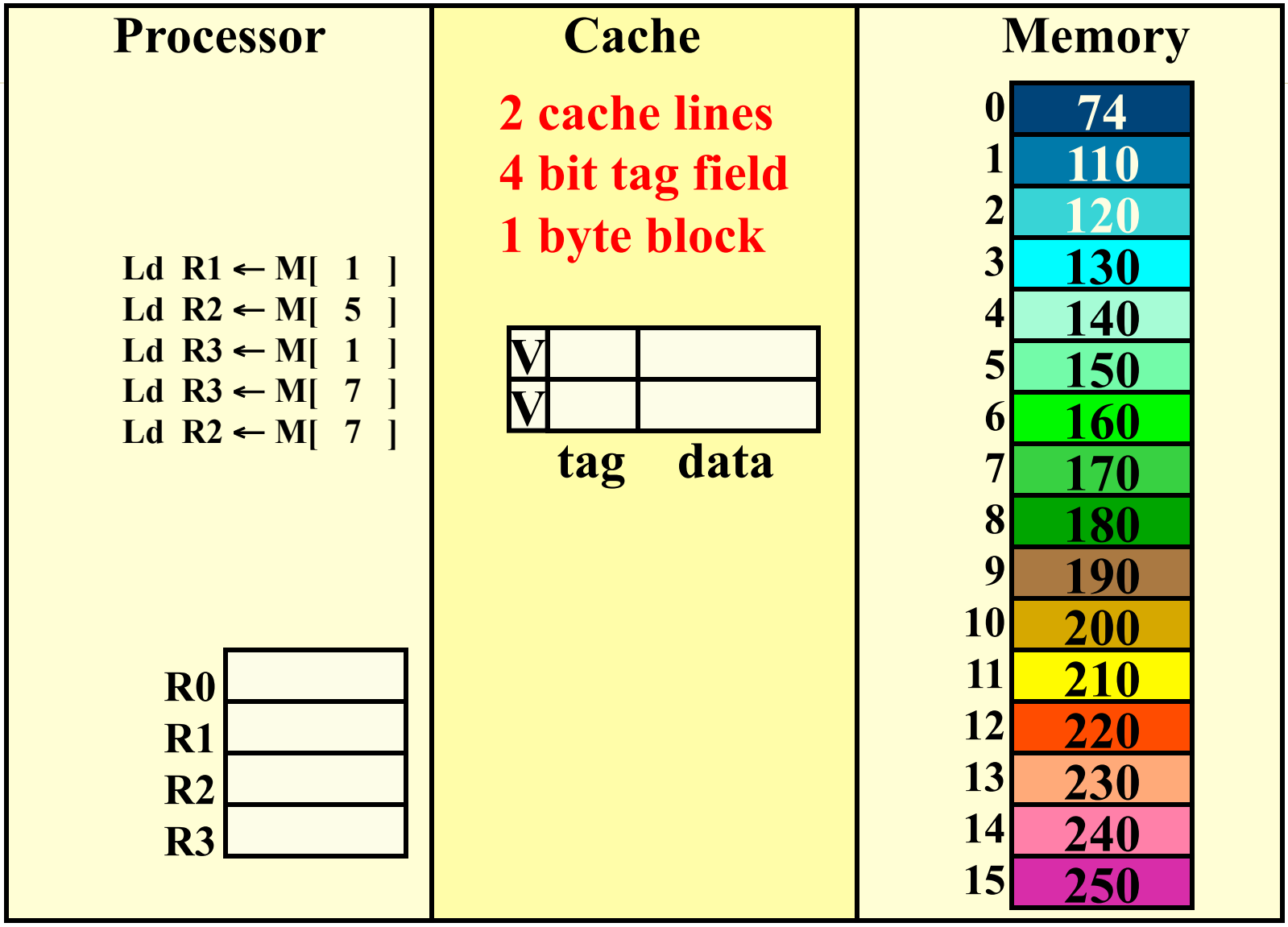
Temporal Locality

- The principle of temporal locality in program references says that if you access a memory location (e.g., 1000) you will be more likely to re-access that location than you will be to reference some other random location.

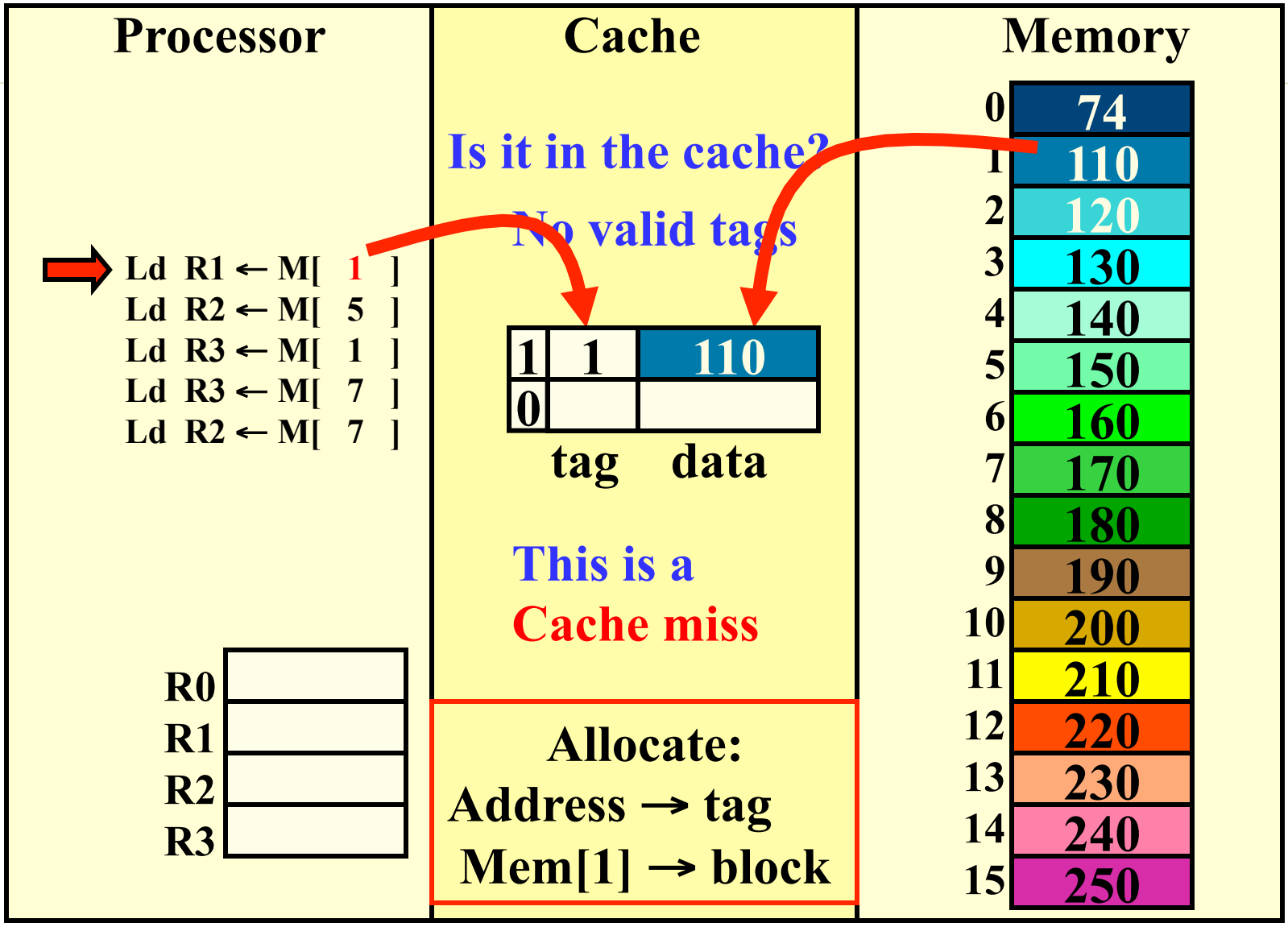
Using Locality in the Cache

- Temporal locality says any miss data should be placed into the cache
  - It is the most recent reference location
- Temporal locality says that the least recently referenced (or least recently used – LRU) cache line should be evicted to make room for the new line.
  - Because the re-access probability falls over time as a cache line isn't referenced, the LRU line is least likely to be re-referenced.

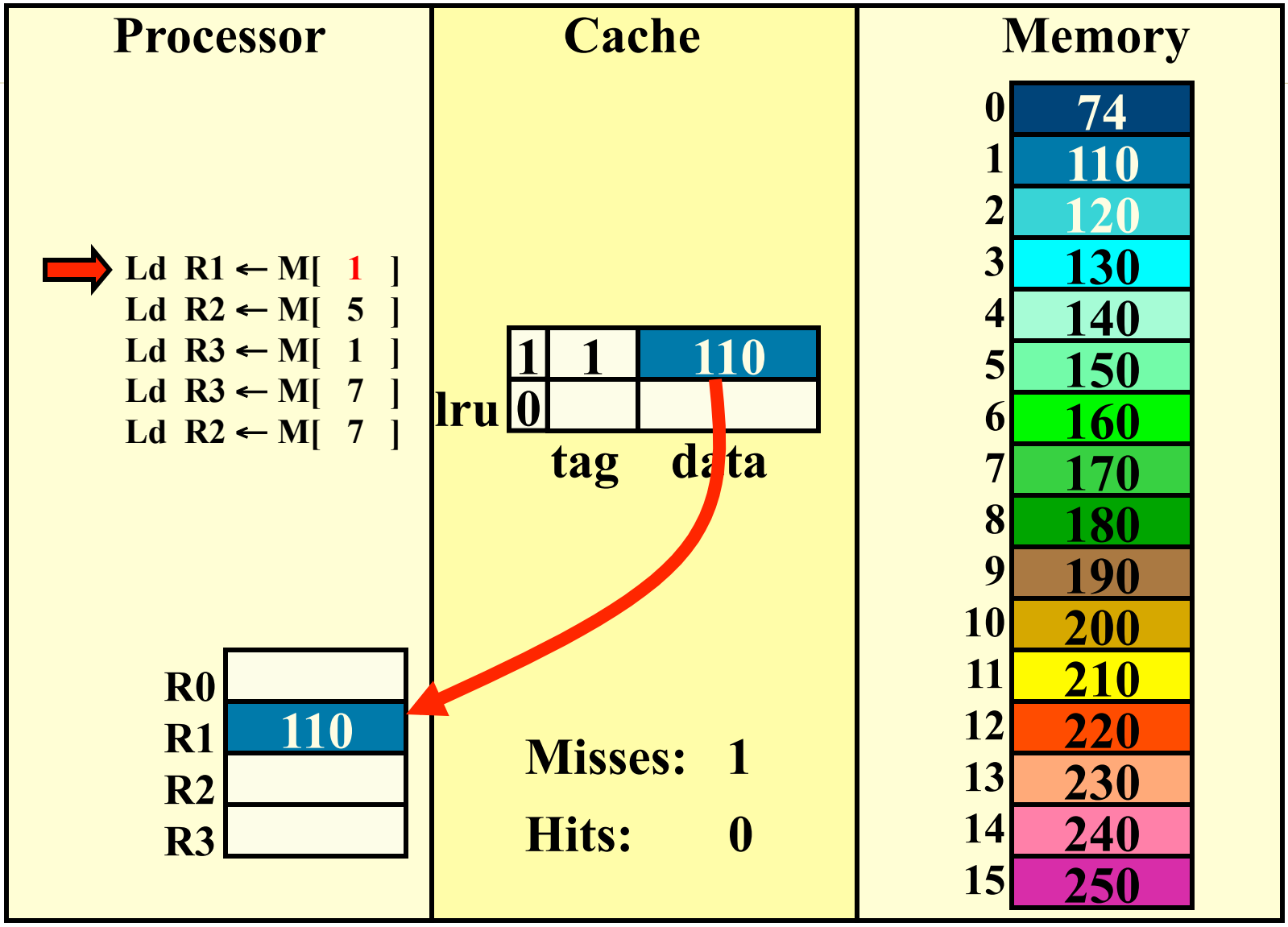
A Very Simple Memory System



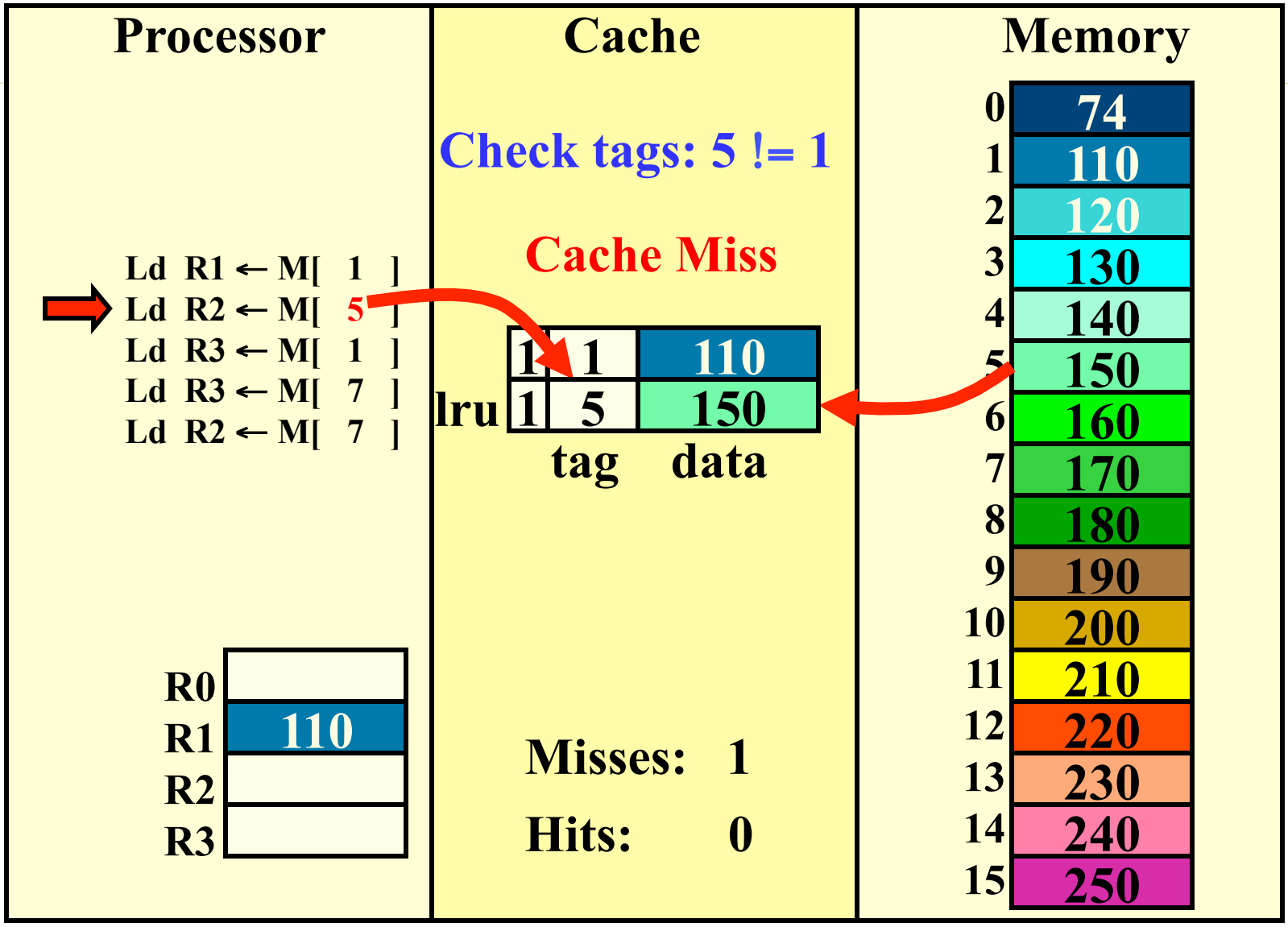
A Very Simple Memory System



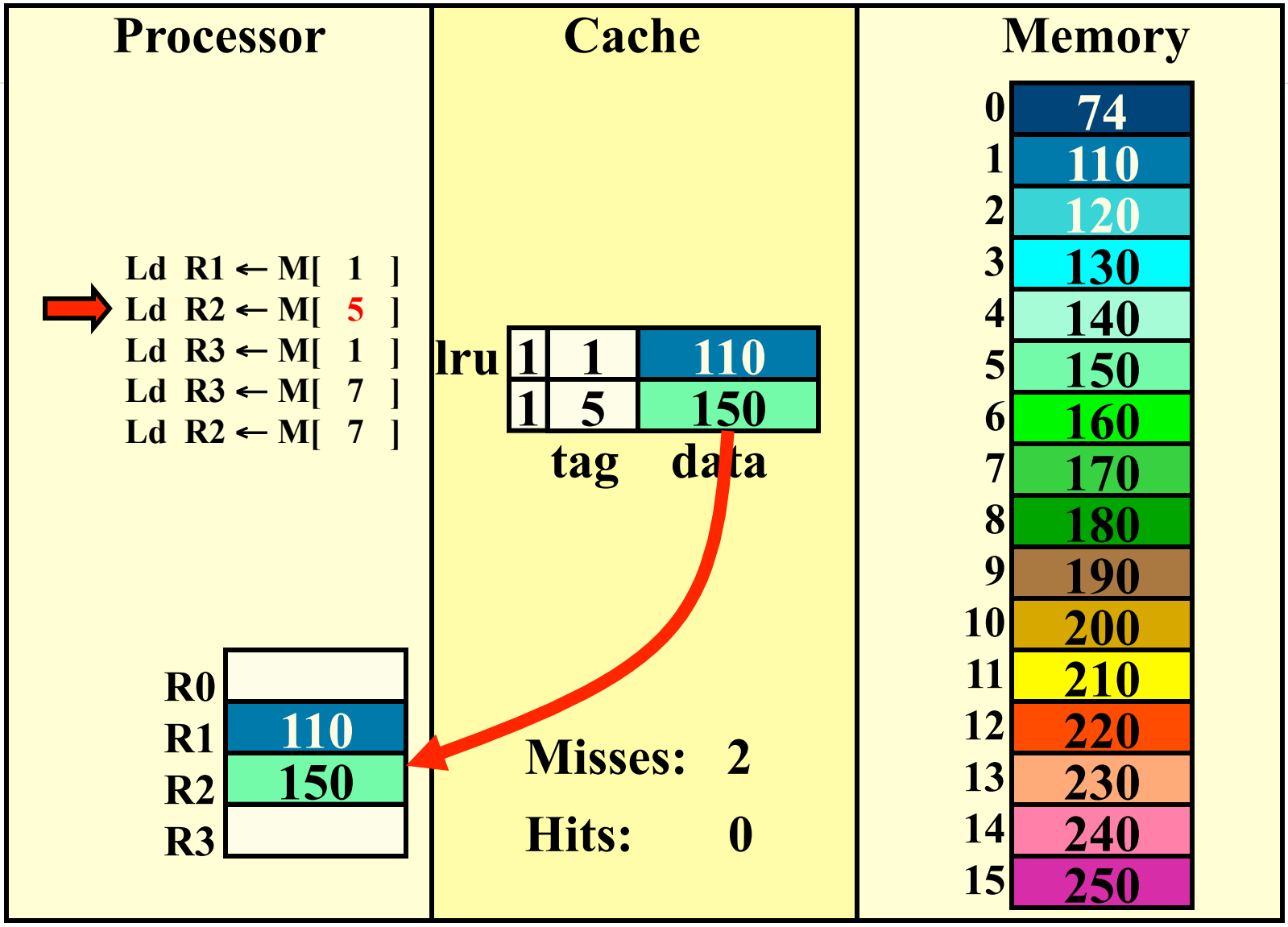
A Very Simple Memory System



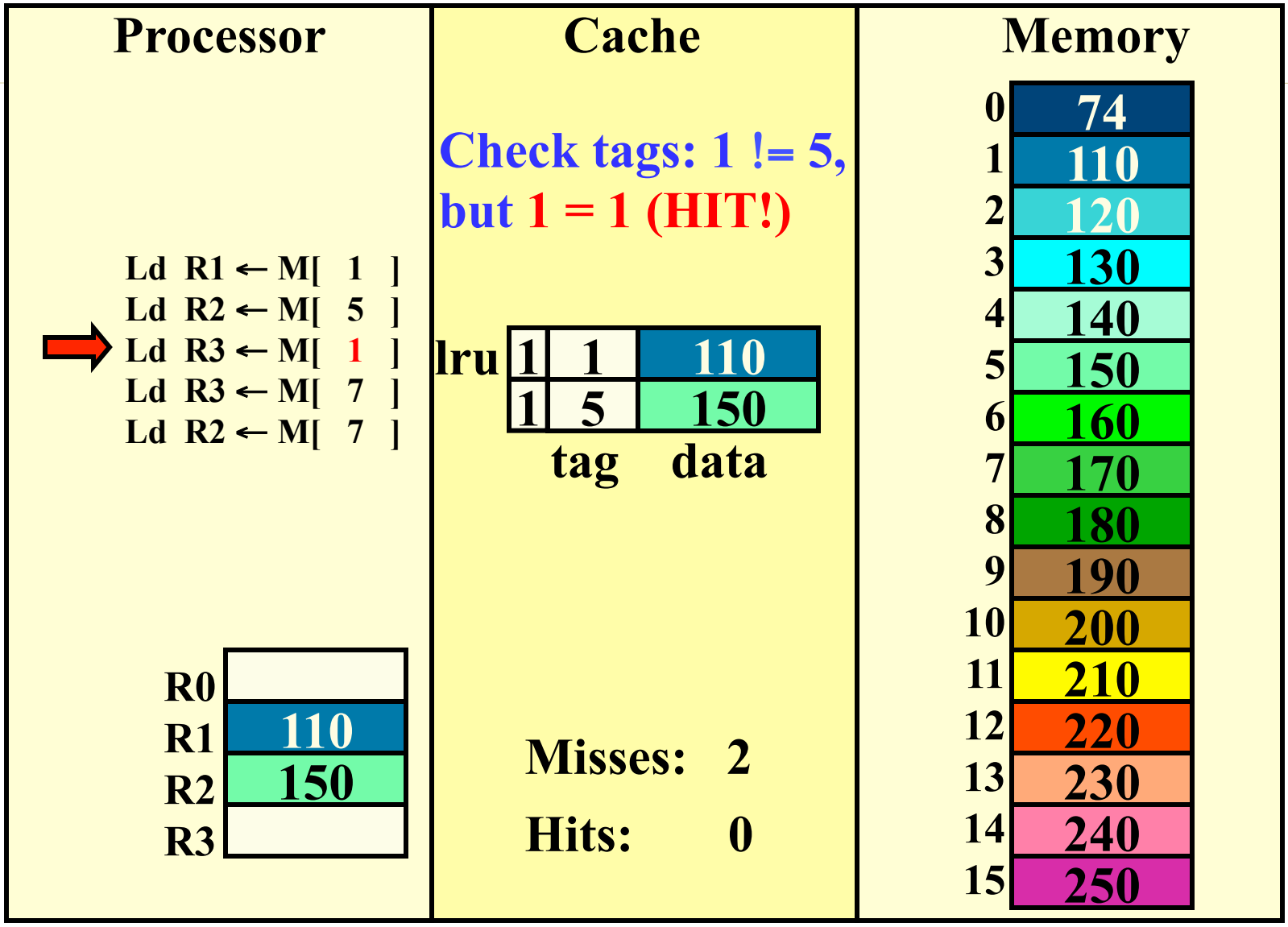
A Very Simple Memory System



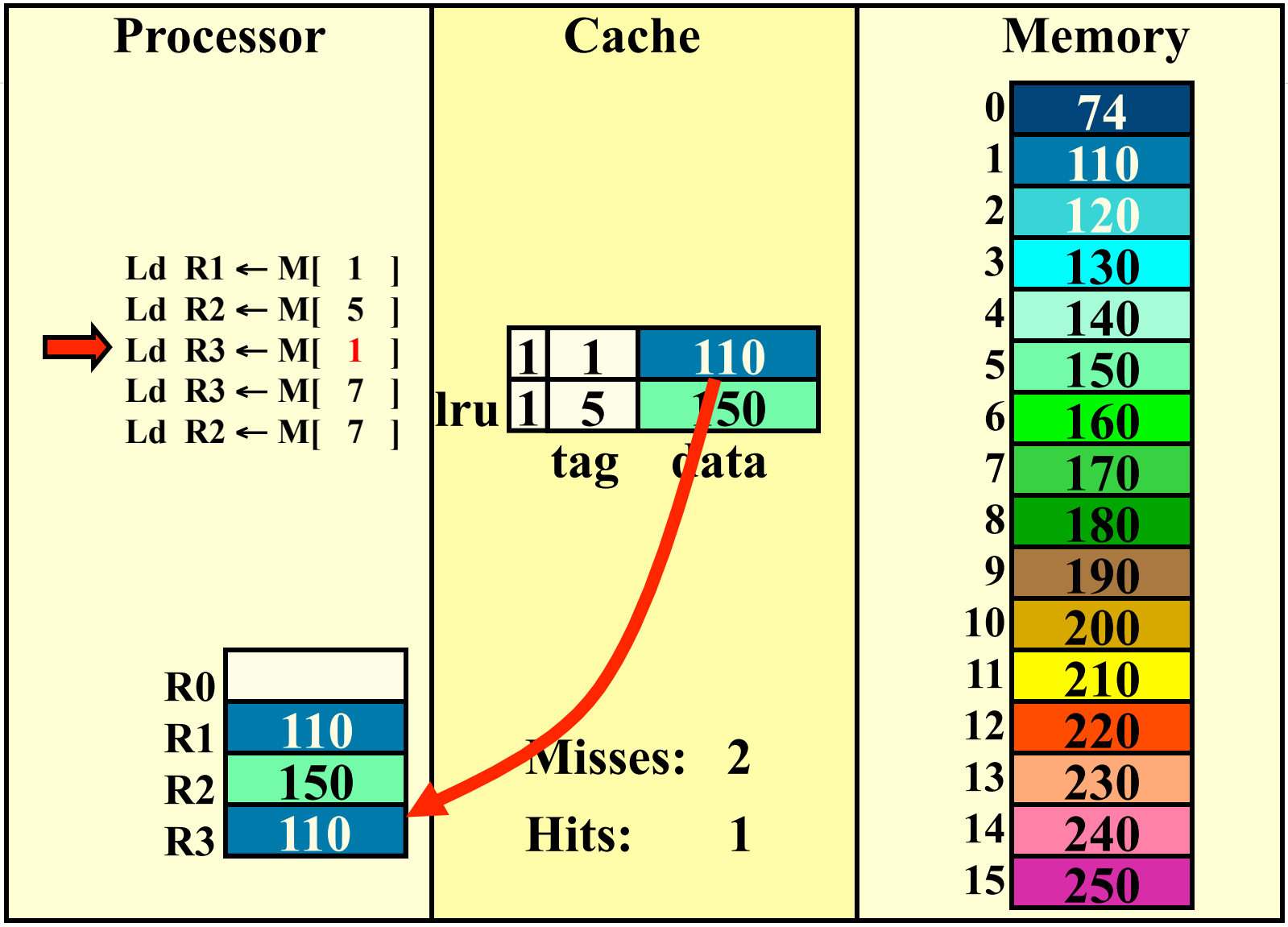
A Very Simple Memory System



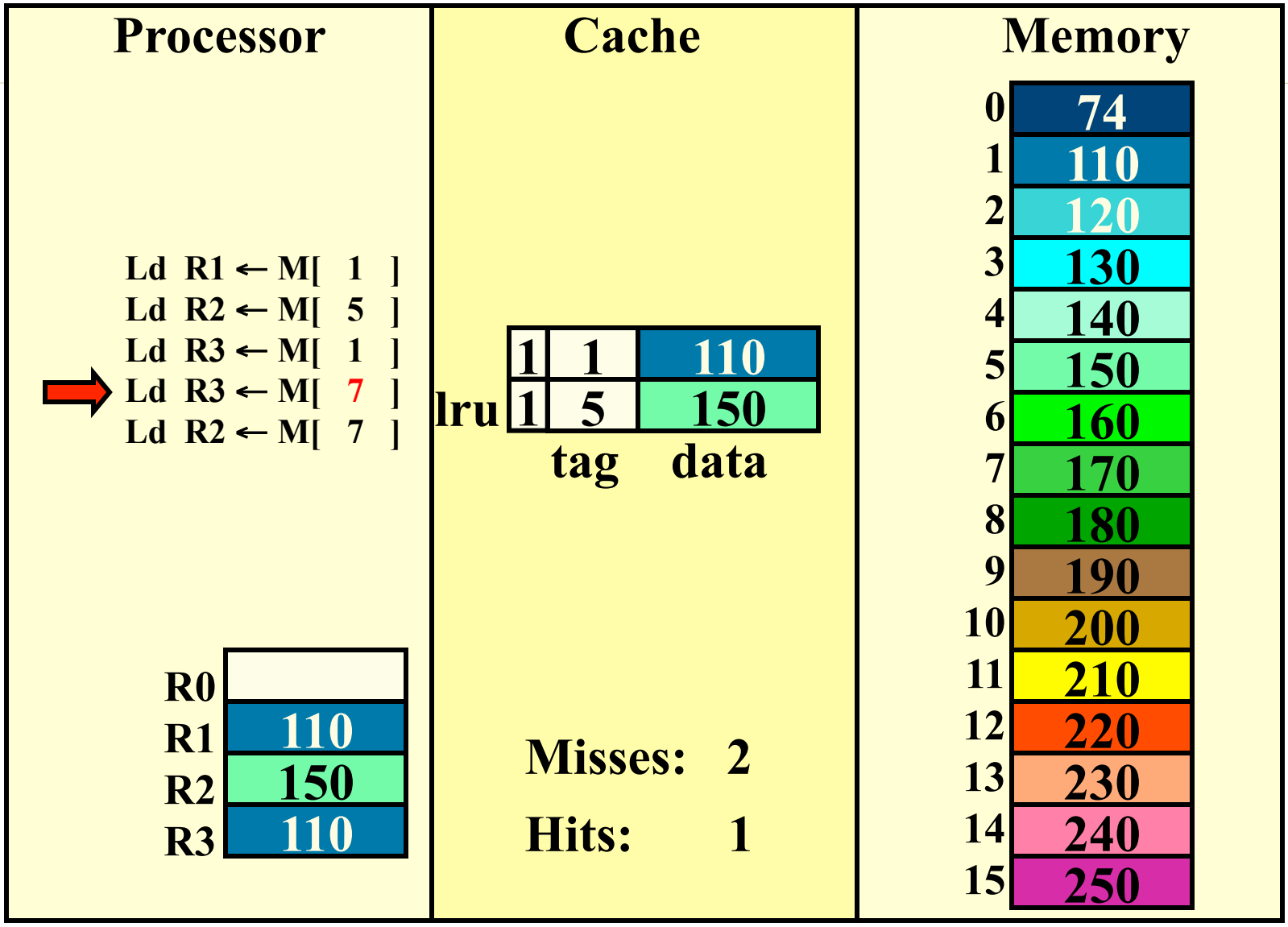
A Very Simple Memory System



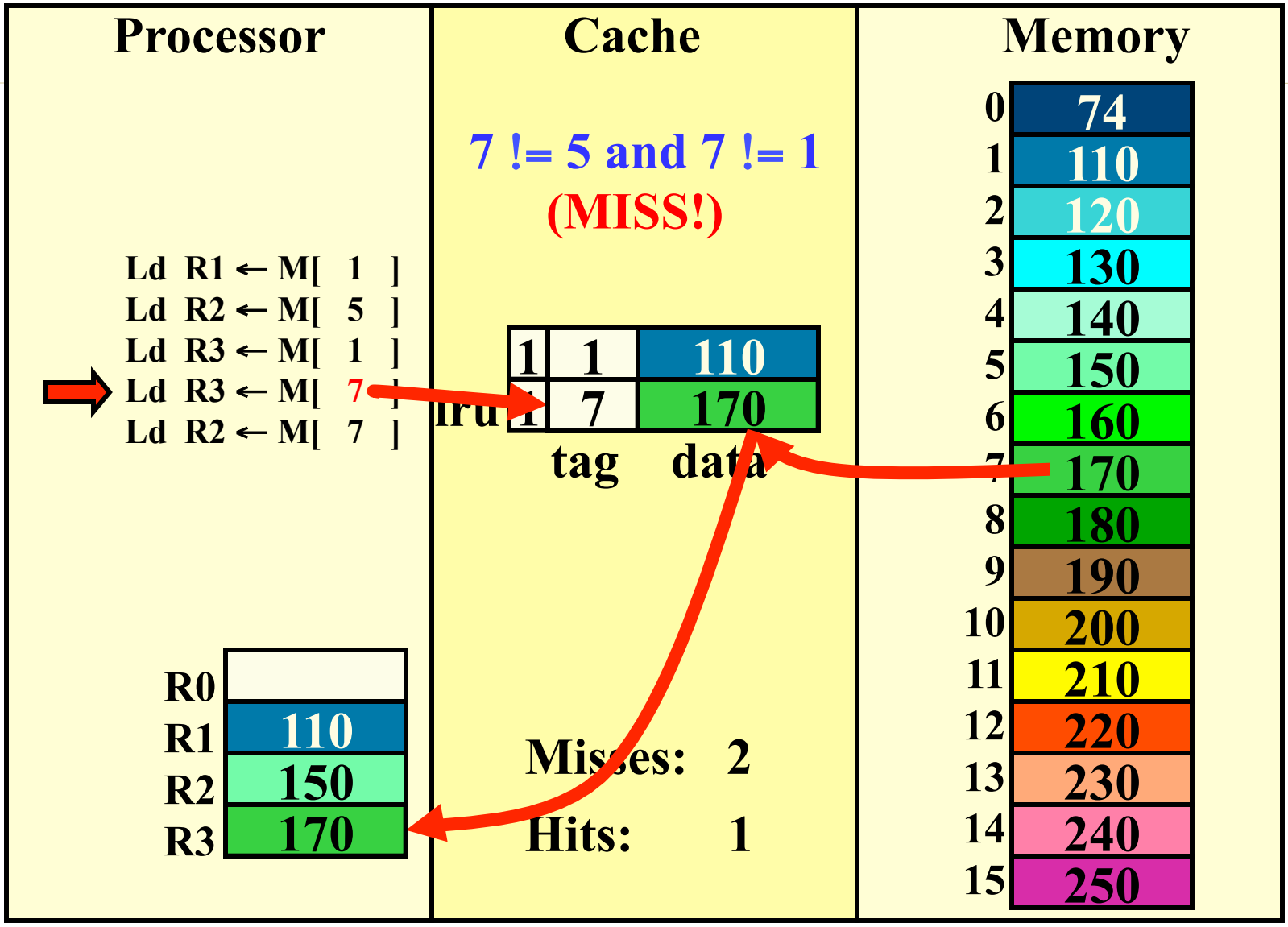
A Very Simple Memory System



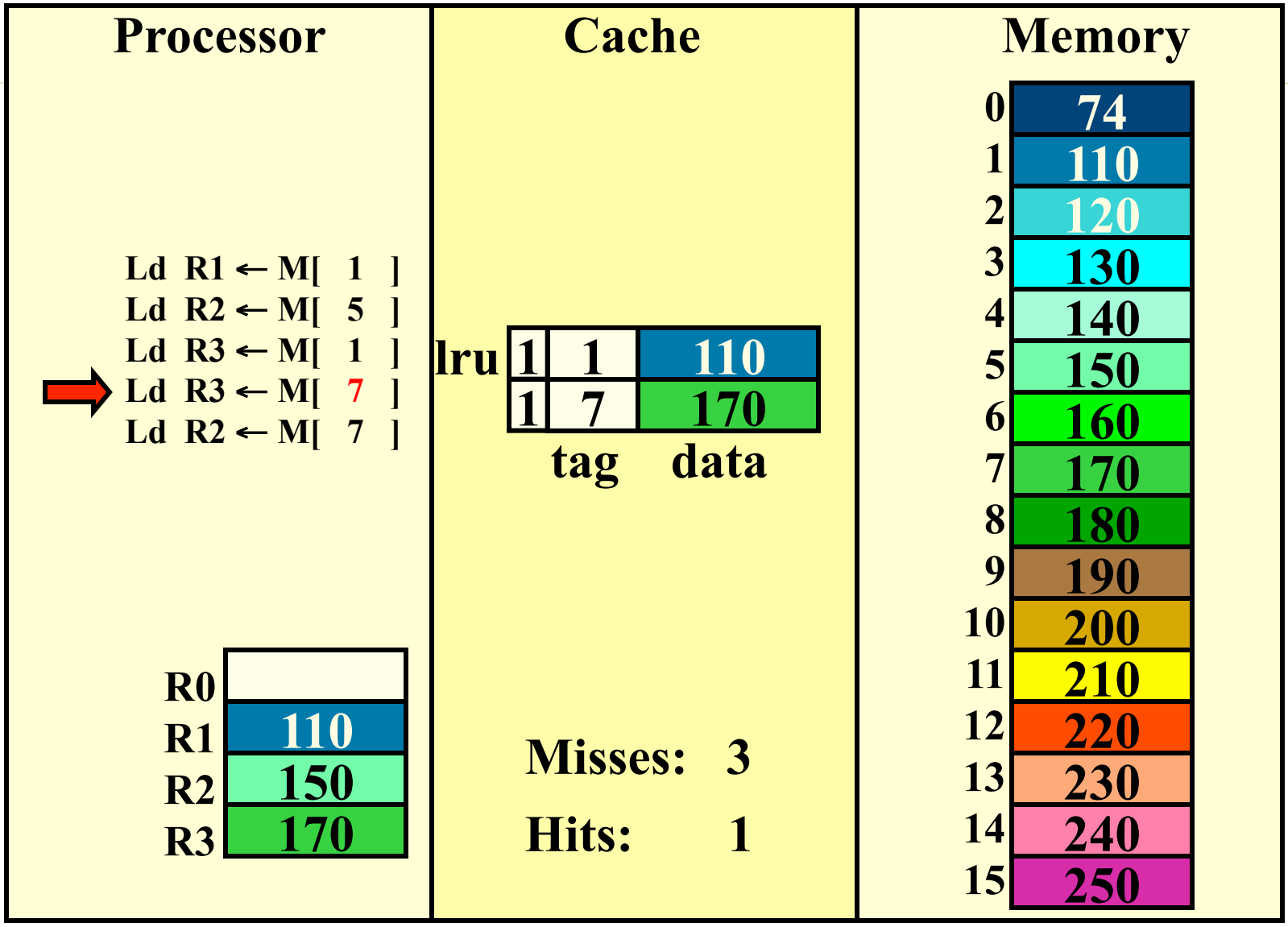
A Very Simple Memory System



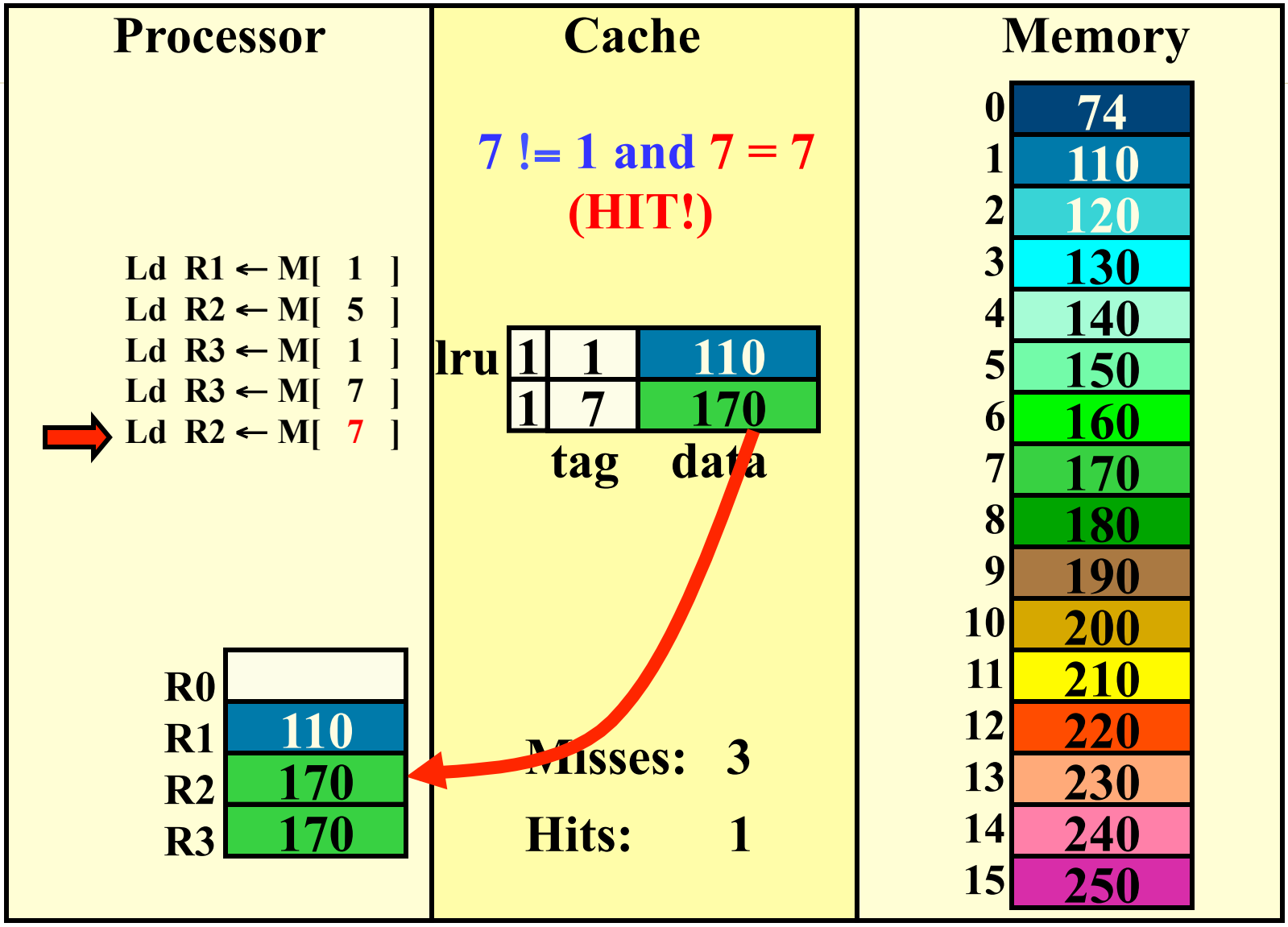
A Very Simple Memory System



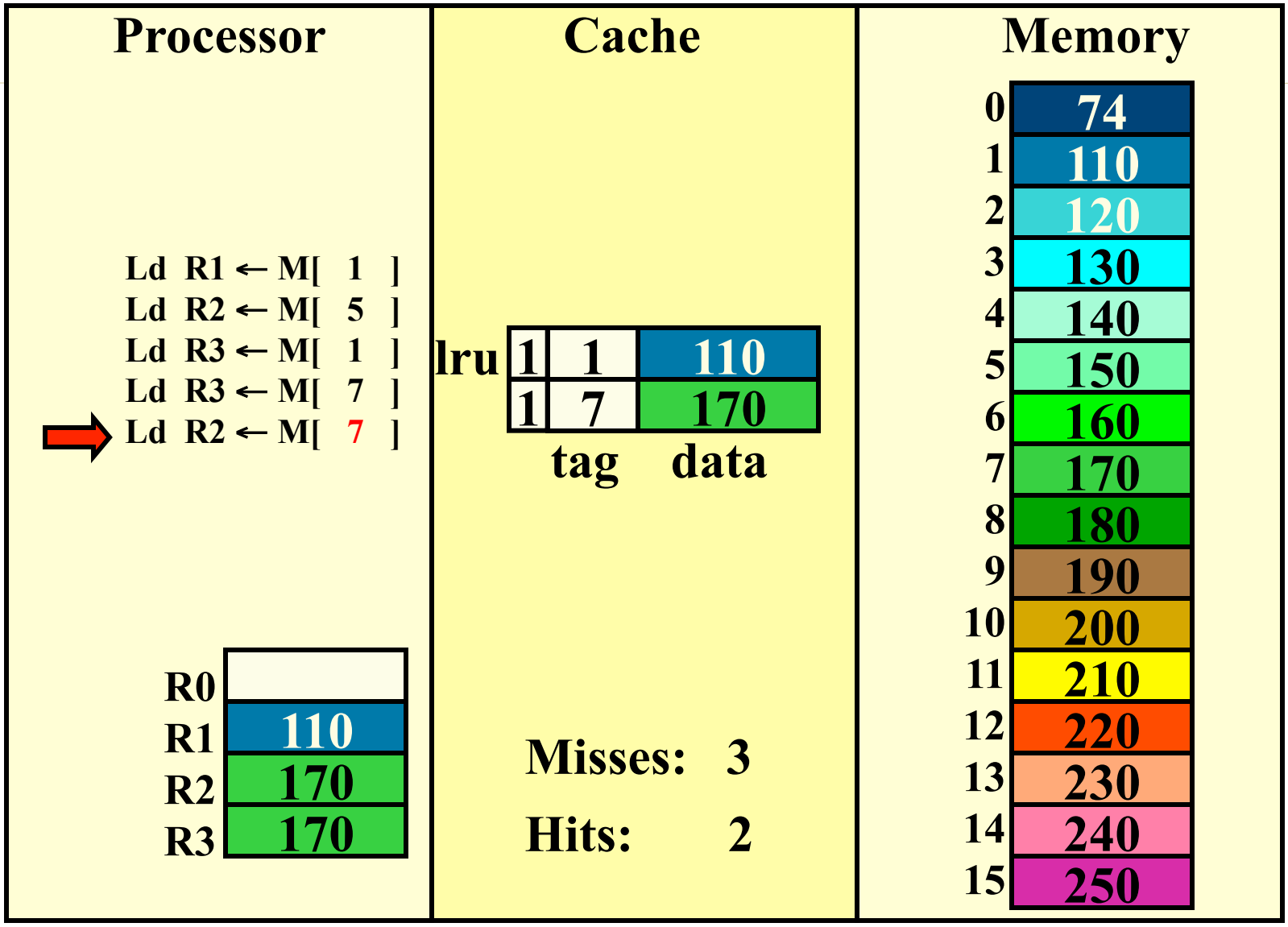
A Very Simple Memory System



A Very Simple Memory System



A Very Simple Memory System



Calculating Average Access Latency

Avg latency = cache latency · hit rate + memory latency · miss rate

Avg latency = 1 cycle · (2/5) + 15 · (3/5)
= 9.4 cycles per reference

- To improve average latency:
  - Improve memory access latency, or
  - Improve cache access latency, or
  - Improve cache hit rate

Visualization tools

http://cs.lafayette.edu/~ilbeyib/vcache/applet/

Class Problem 2

For application X running on the LC3101 single cycle datapath, you measure that 40% of the operations access memory. You also measure that 90% of the memory references hit in the cache. Whenever a cache miss occurs, the processor is stalled for 20 cycles to transfer the block from memory into the cache (so that instruction takes 21 cycles).

What is the CPI for application X on this processor? What would the CPI be if there were no cache?

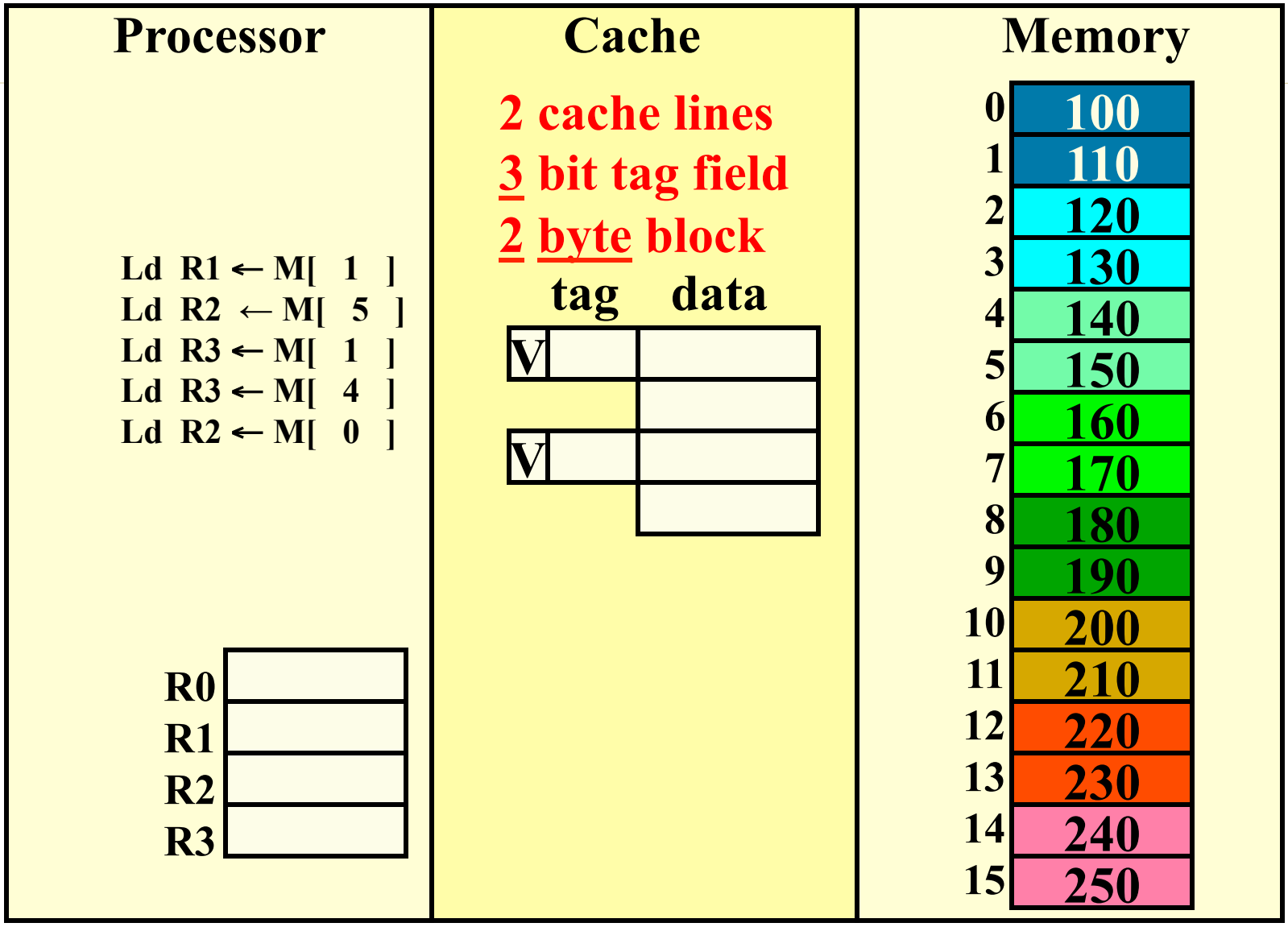
Calculating Cost

- How much does a cache cost?
  - Calculate storage requirements
    - 2 bytes of SRAM
  - Calculate overhead to support access (tags)
    - 2 4-bit tags
    - The cost of the tags is often forgotten for caches, but this cost drives the design of real caches
- What is the cost if a 32 bit address is used?

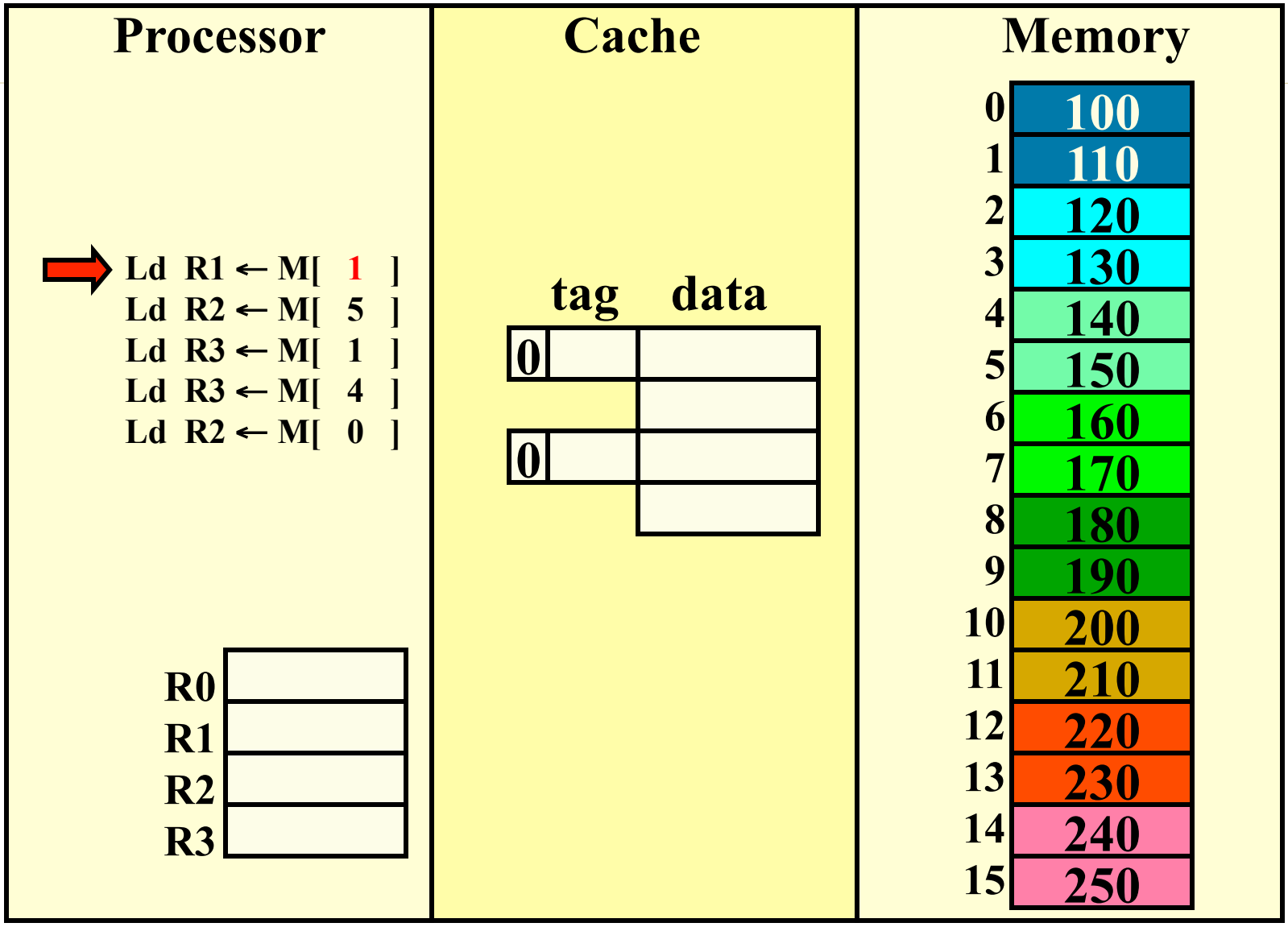
How do we reduce the overhead?

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

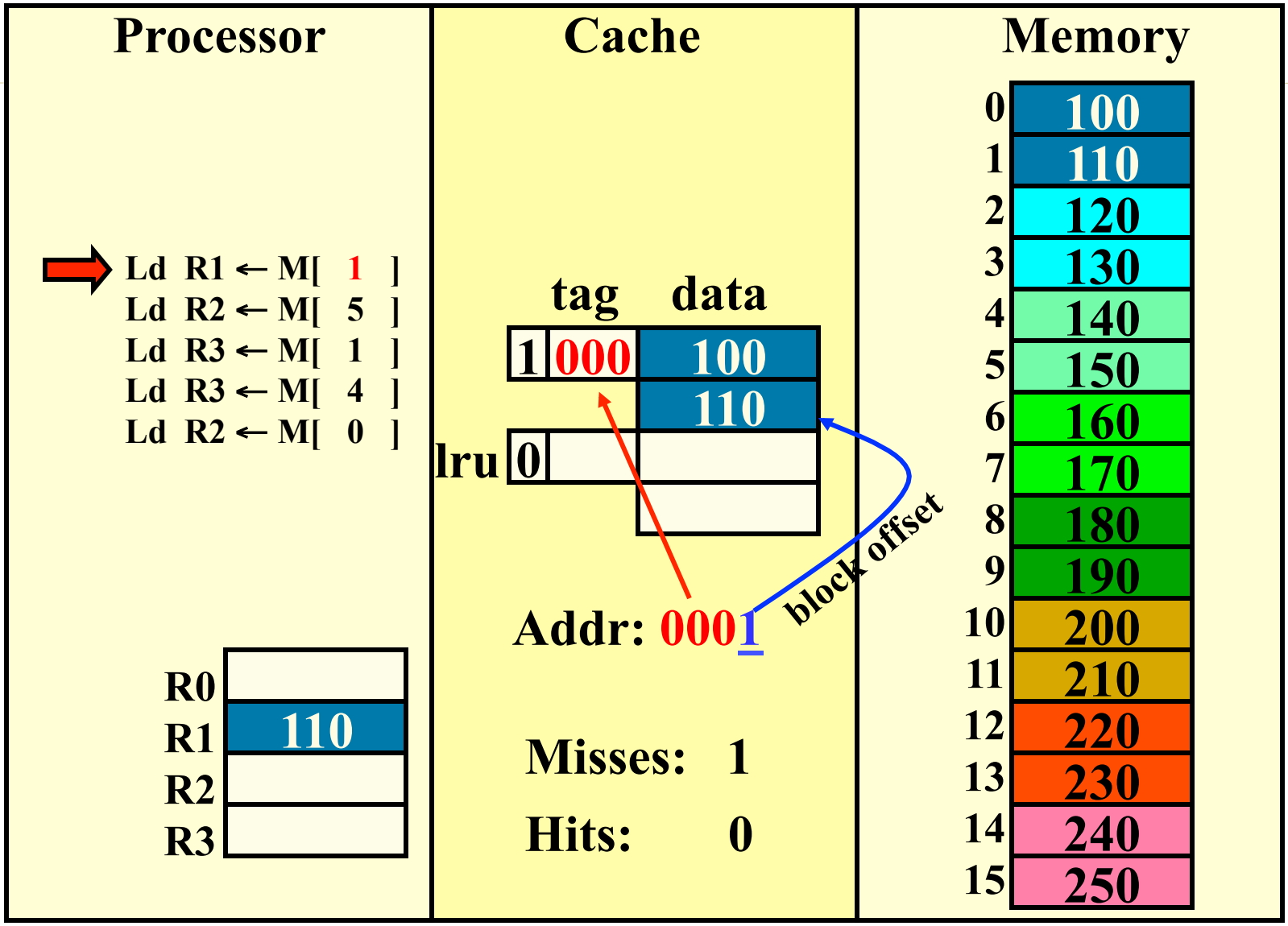
Block size for caches



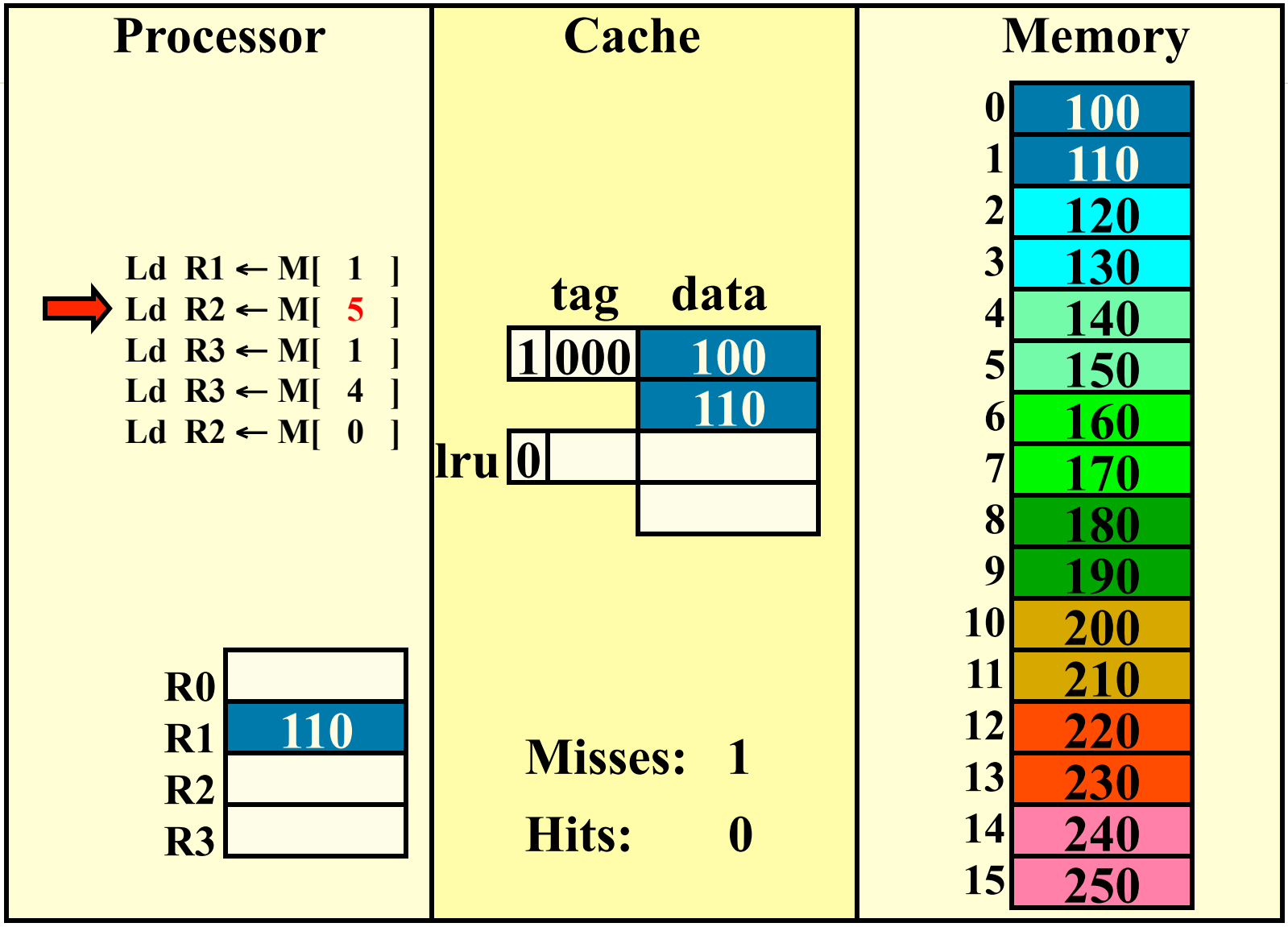
Block size for caches



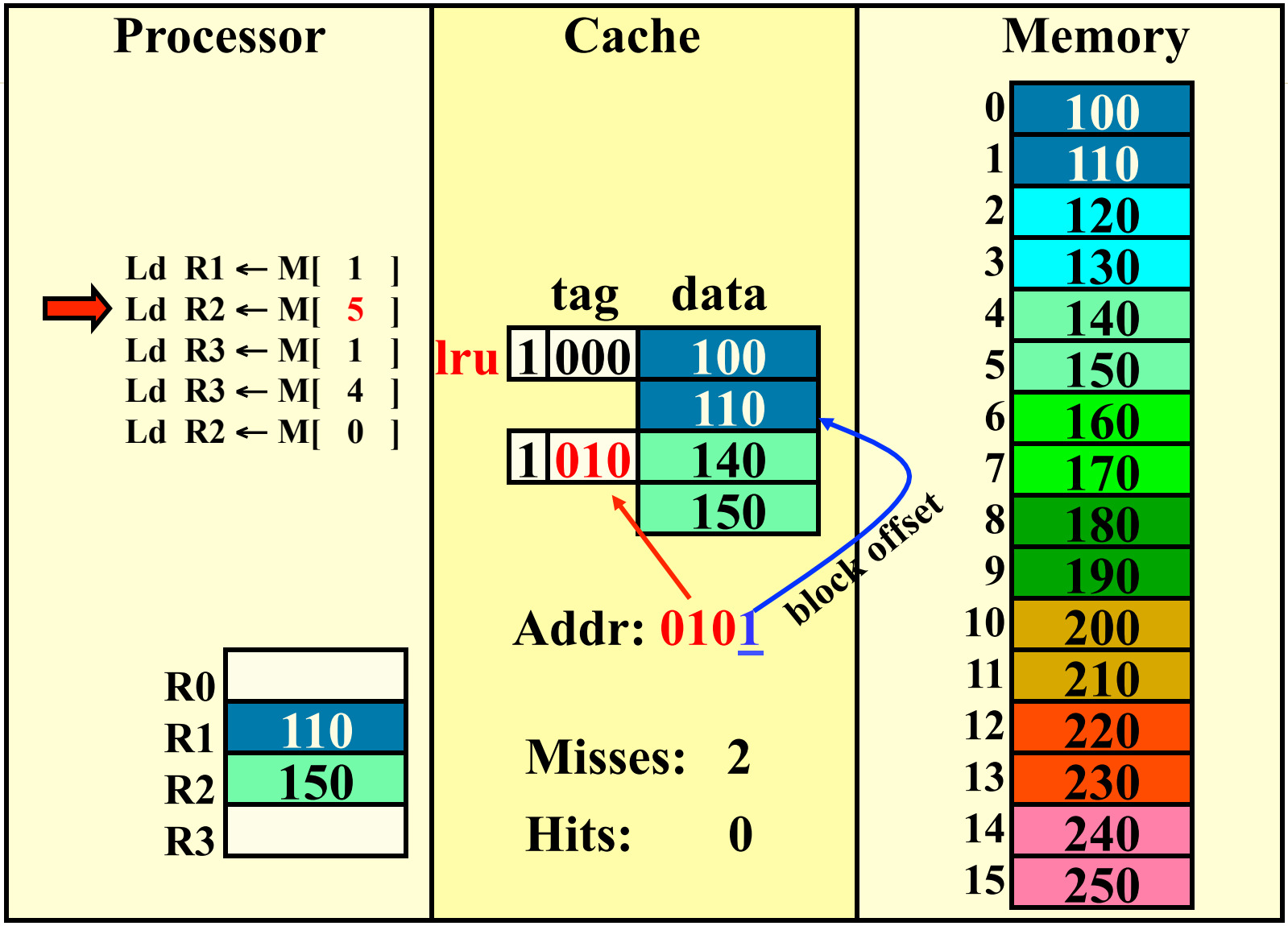
Block size for caches



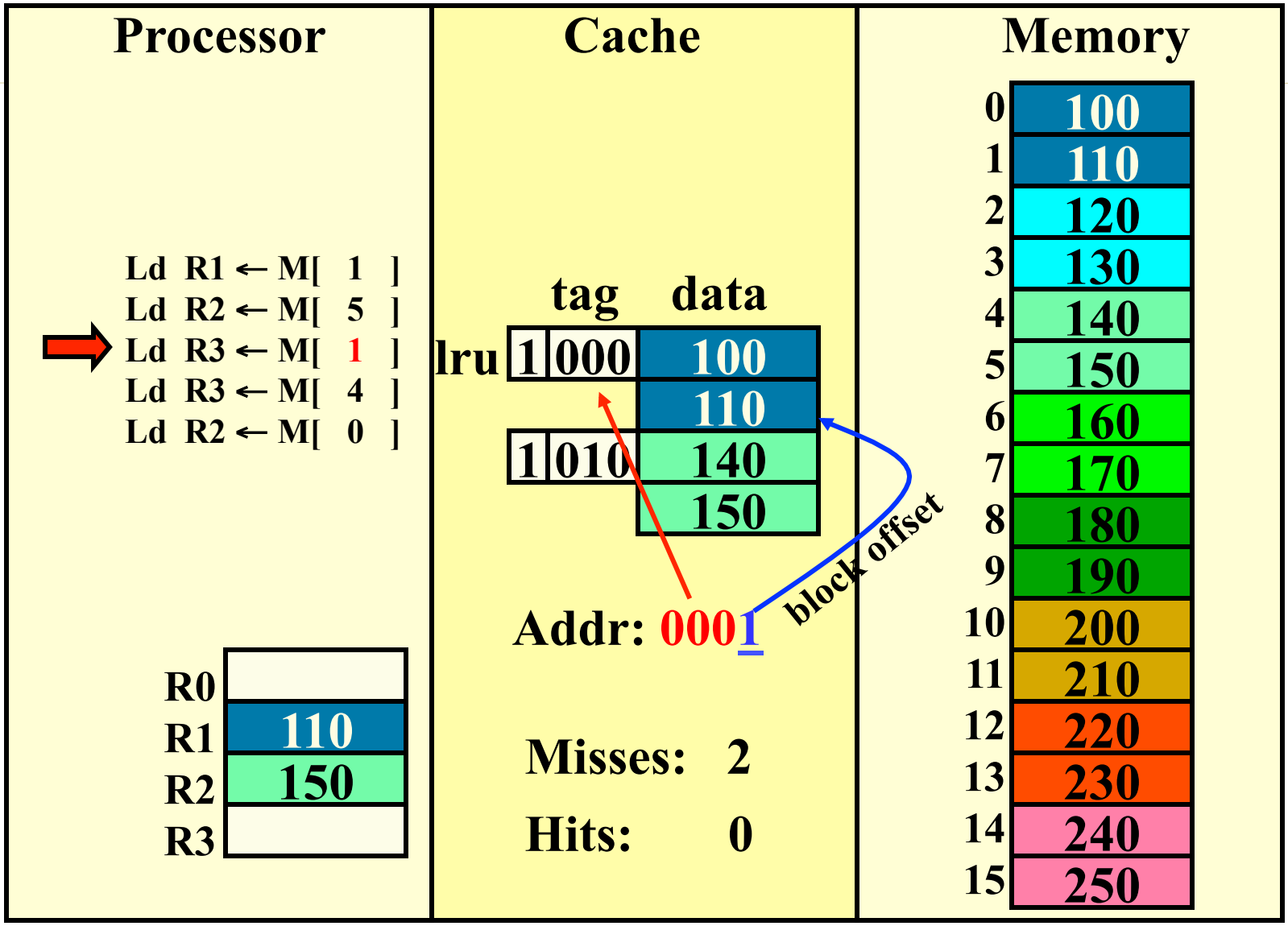
Block size for caches



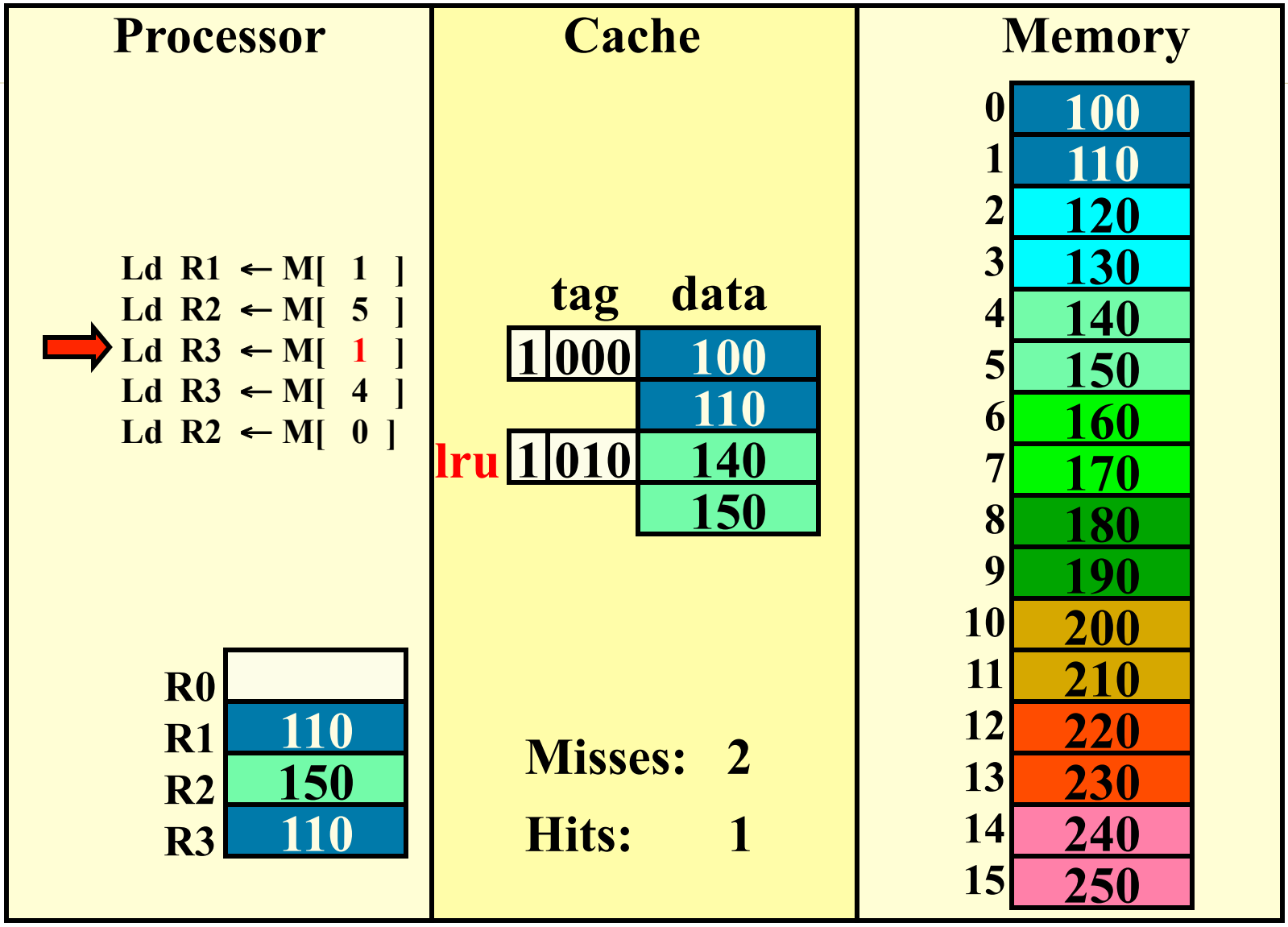
Block size for caches



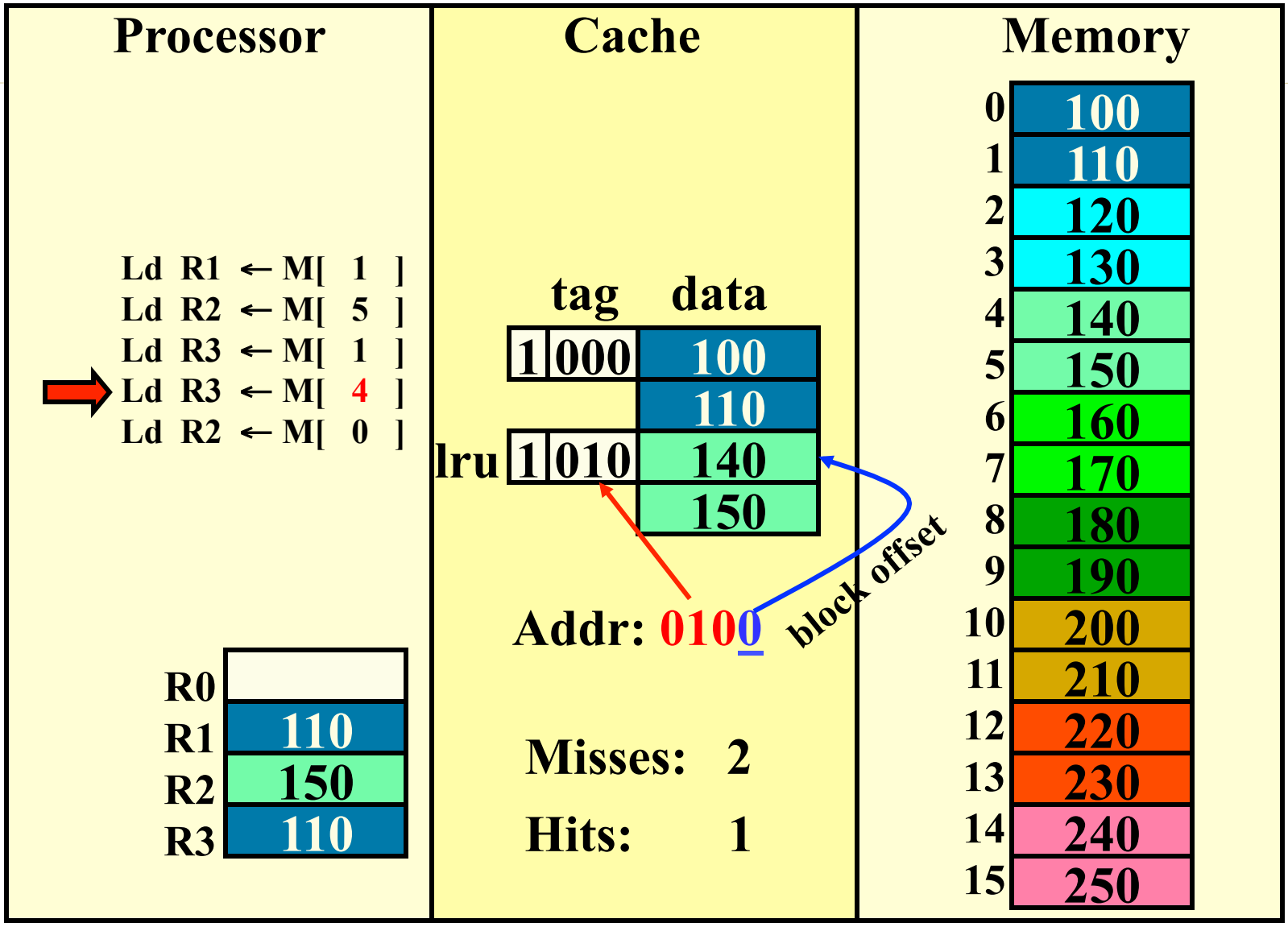
Block size for caches



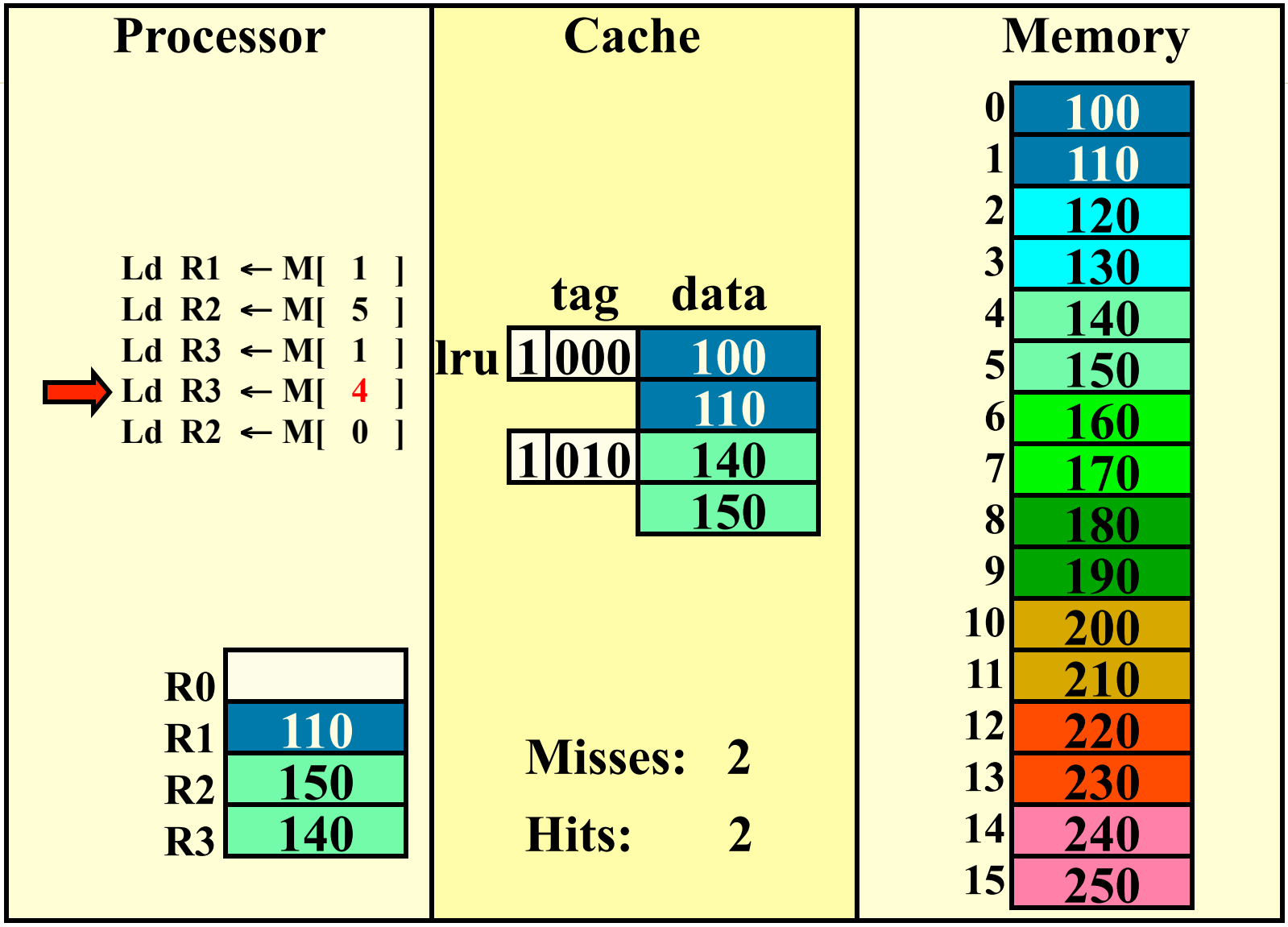
Block size for caches



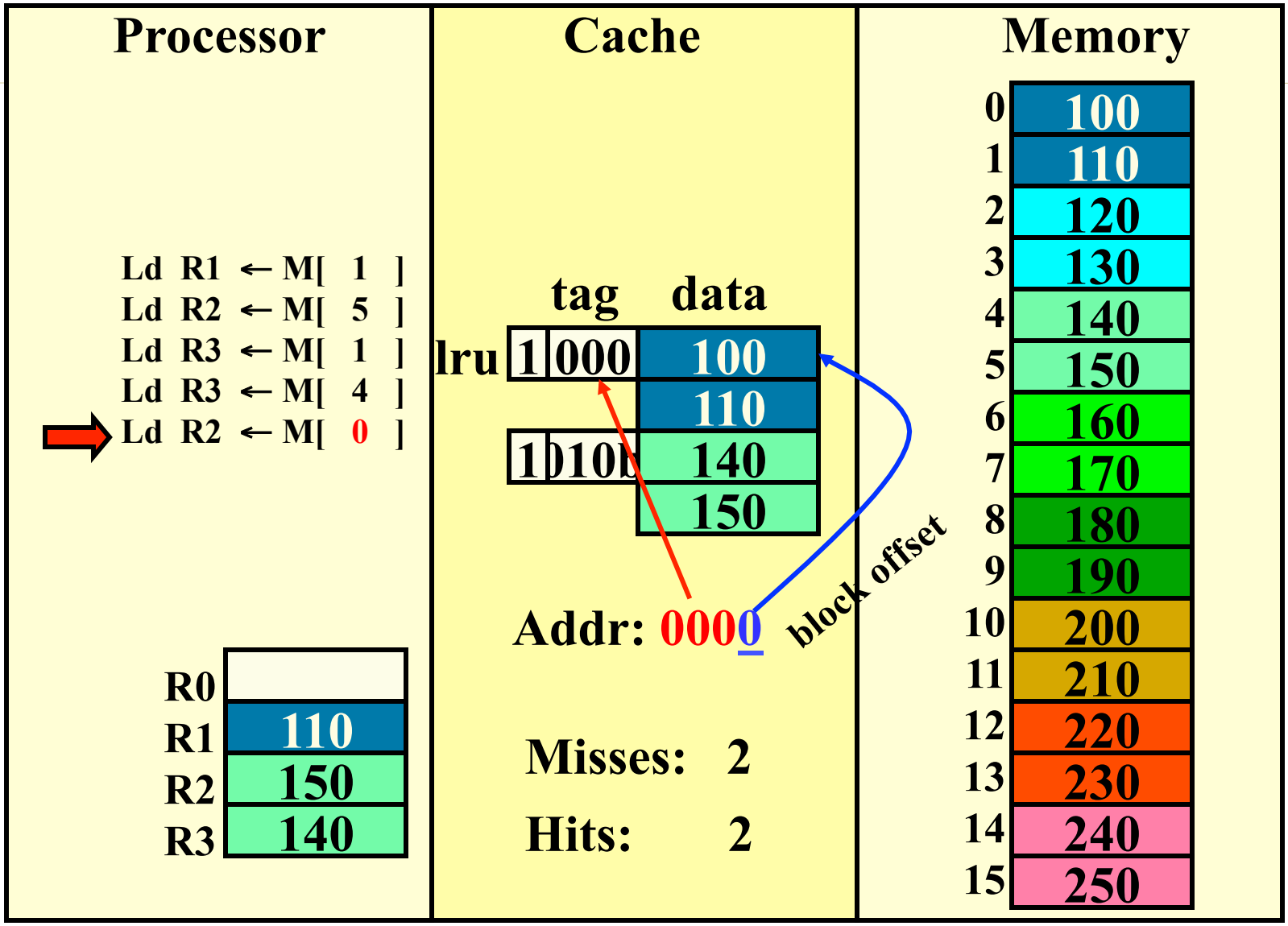
Block size for caches



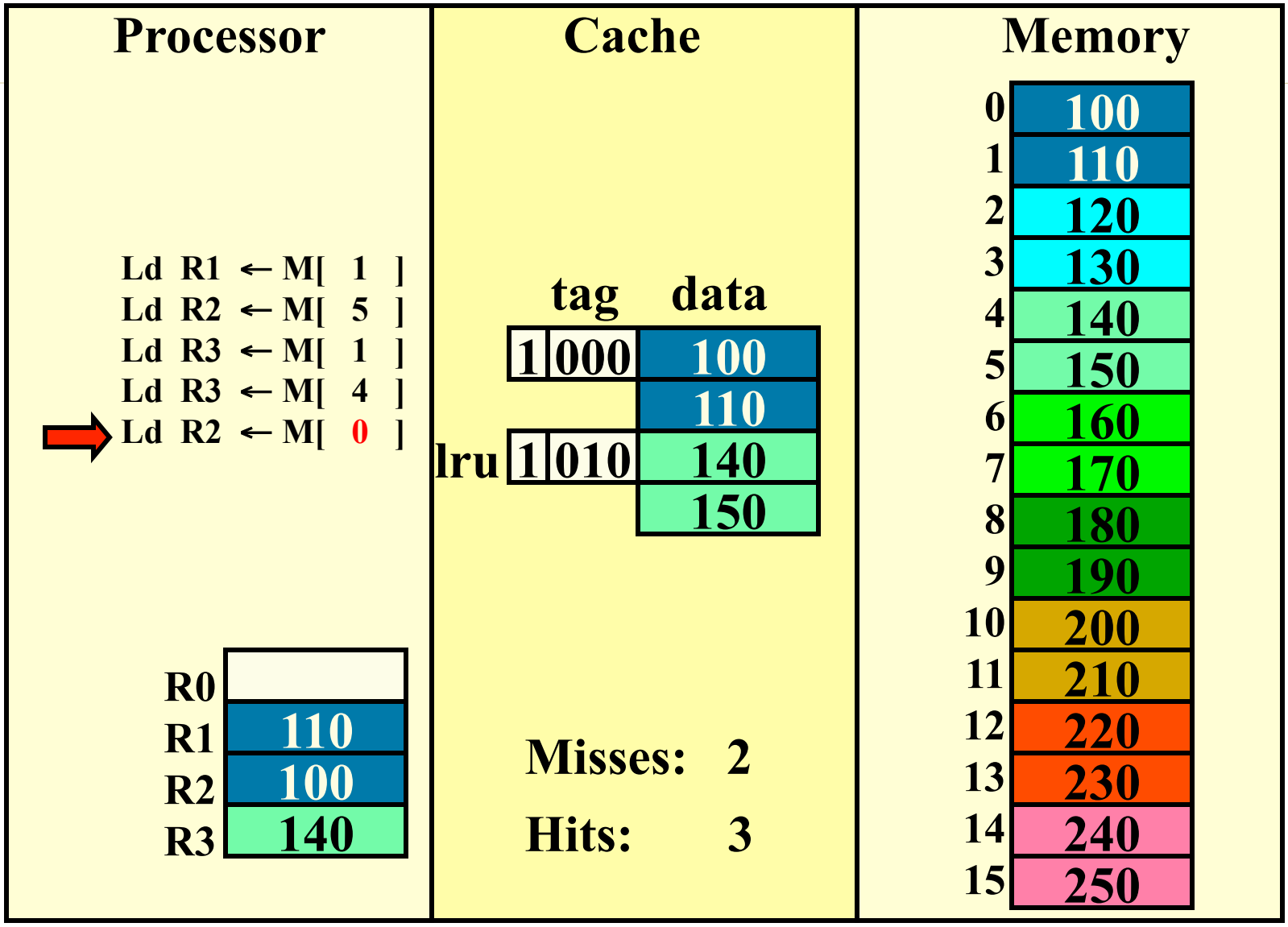
Block size for caches



Block size for caches



Block size for caches



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.
- Spatial locality in a program says that if we reference a memory location (e.g., 1000), we are more likely to reference a location near it (e.g. 1001) than some random location.

Basic Cache Organization

- Decide on the block size
  - How? Simulate lots of different block sizes and see which one gives the best performance
  - Most systems use a block size between 32 bytes and 128 bytes
  - Longer sizes reduce the overhead by:
    - Reducing the number of CAMs
    - Reducing the size of each CAM

Class Problem 3

Processor

Whats the cache hit rate when the following code is executed?

```
Ld  R1 ← M[  7  ]
Ld  R2 ← M[  6  ]
Ld  R3 ← M[  9  ]
Ld  R3 ← M[  10 ]
Ld  R2 ← M[  11 ]
```

R0	
R1	
R2	
R3	

Cache

2 cache lines
3 bit tag field
2 byte block

	tag	data
V		
V		

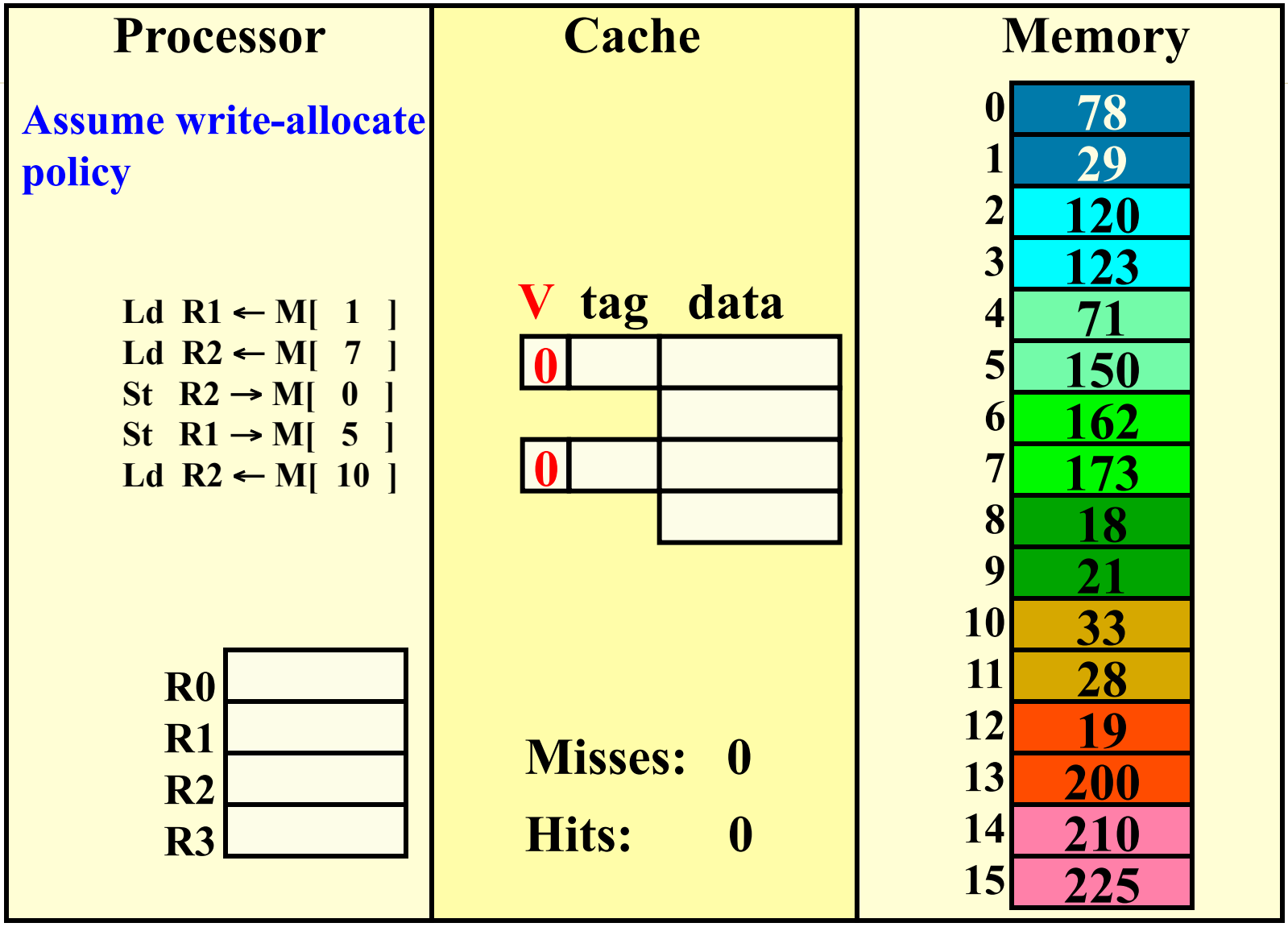
Memory

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

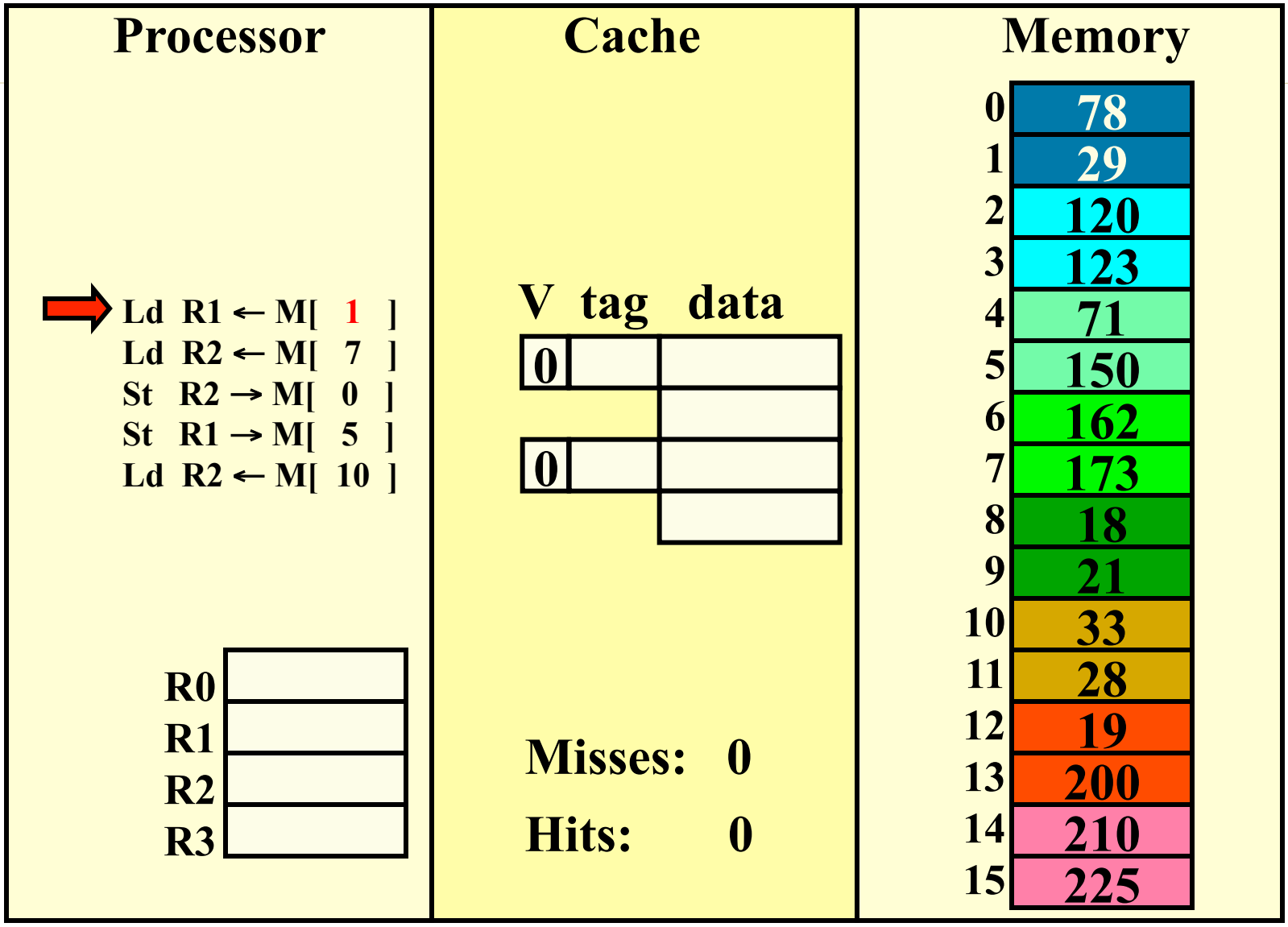
What About Stores?

- 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)

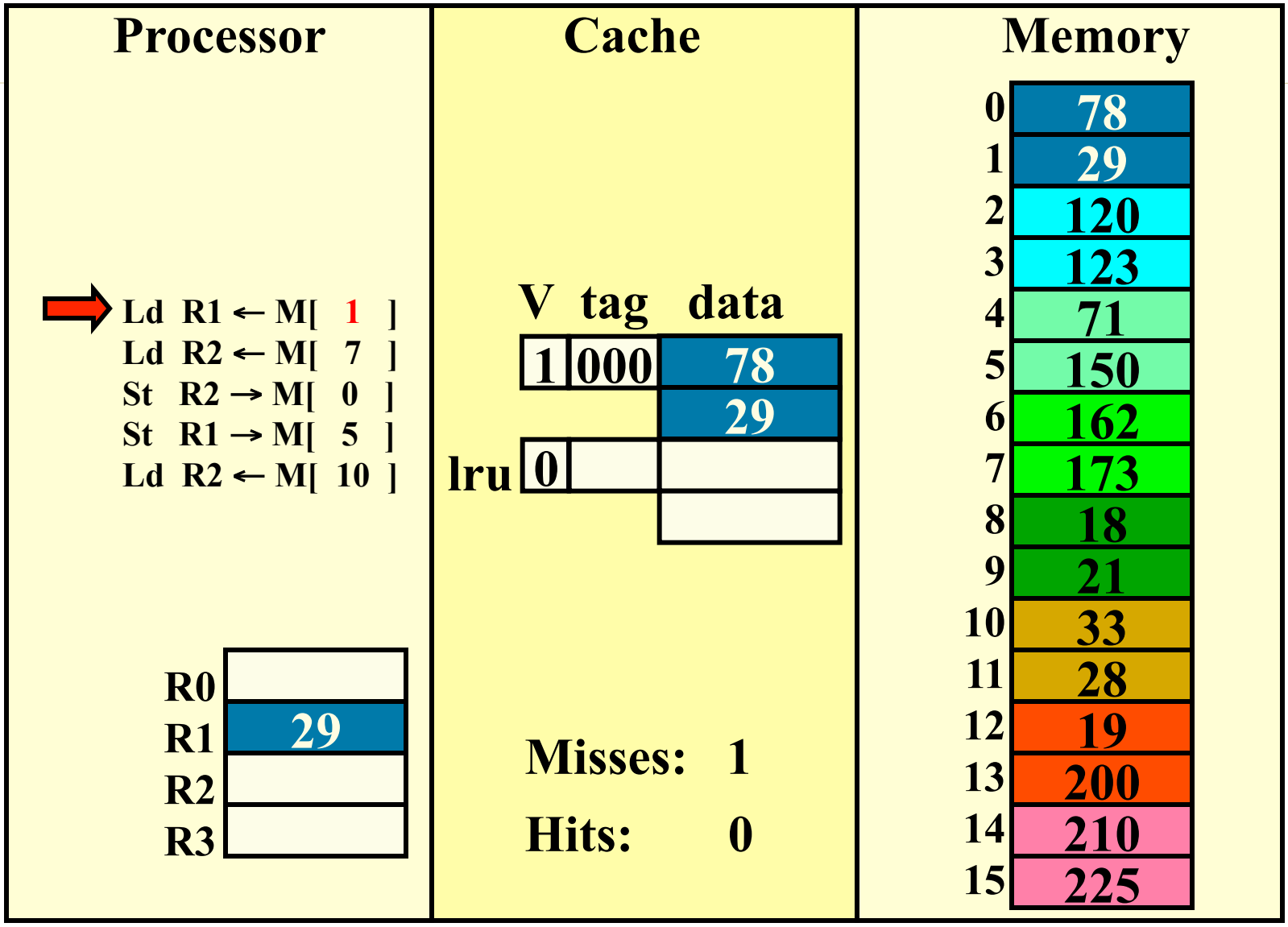
Handling Stores (write-through)



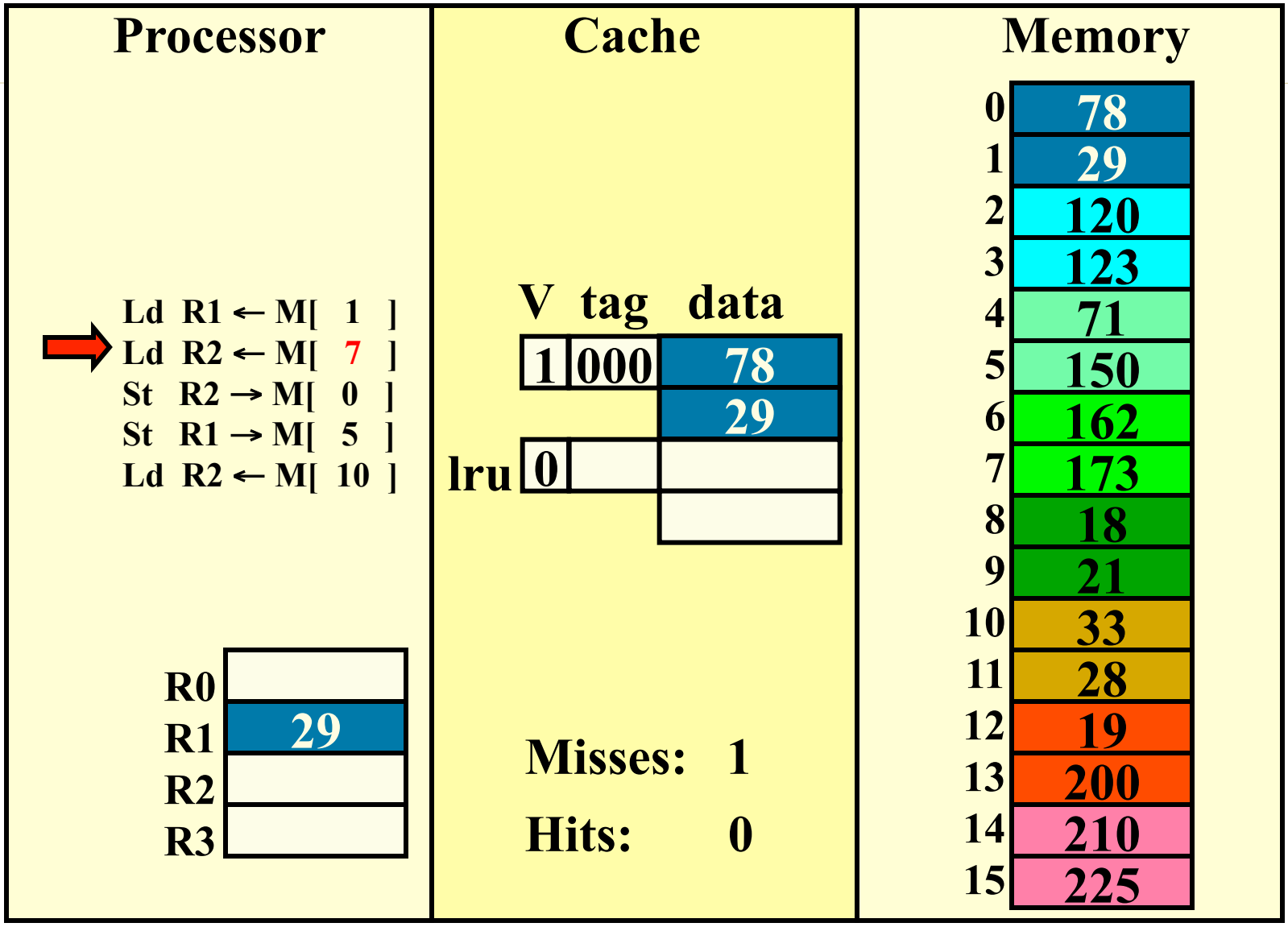
write-through (REF 1)



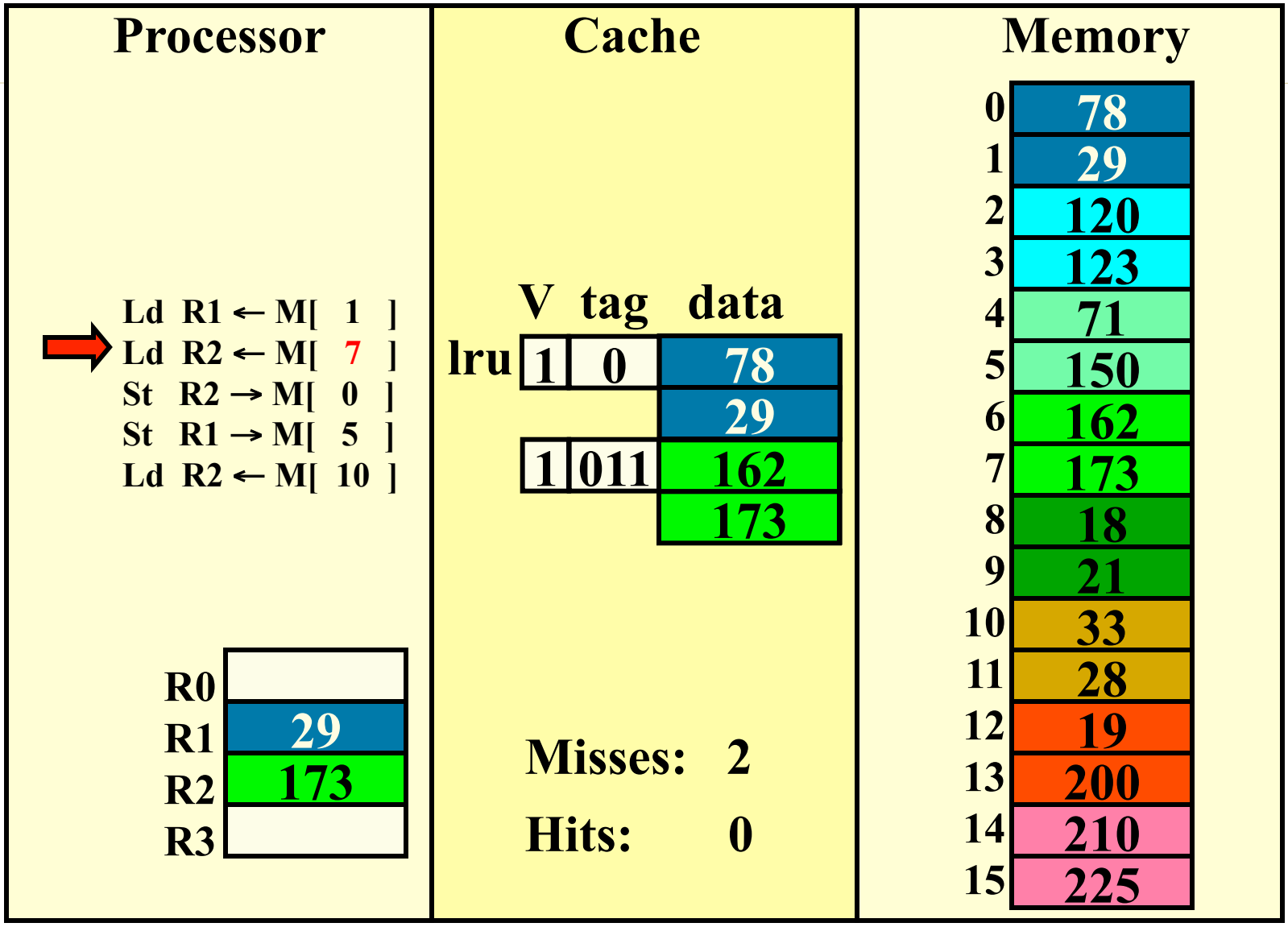
write-through (REF 1)



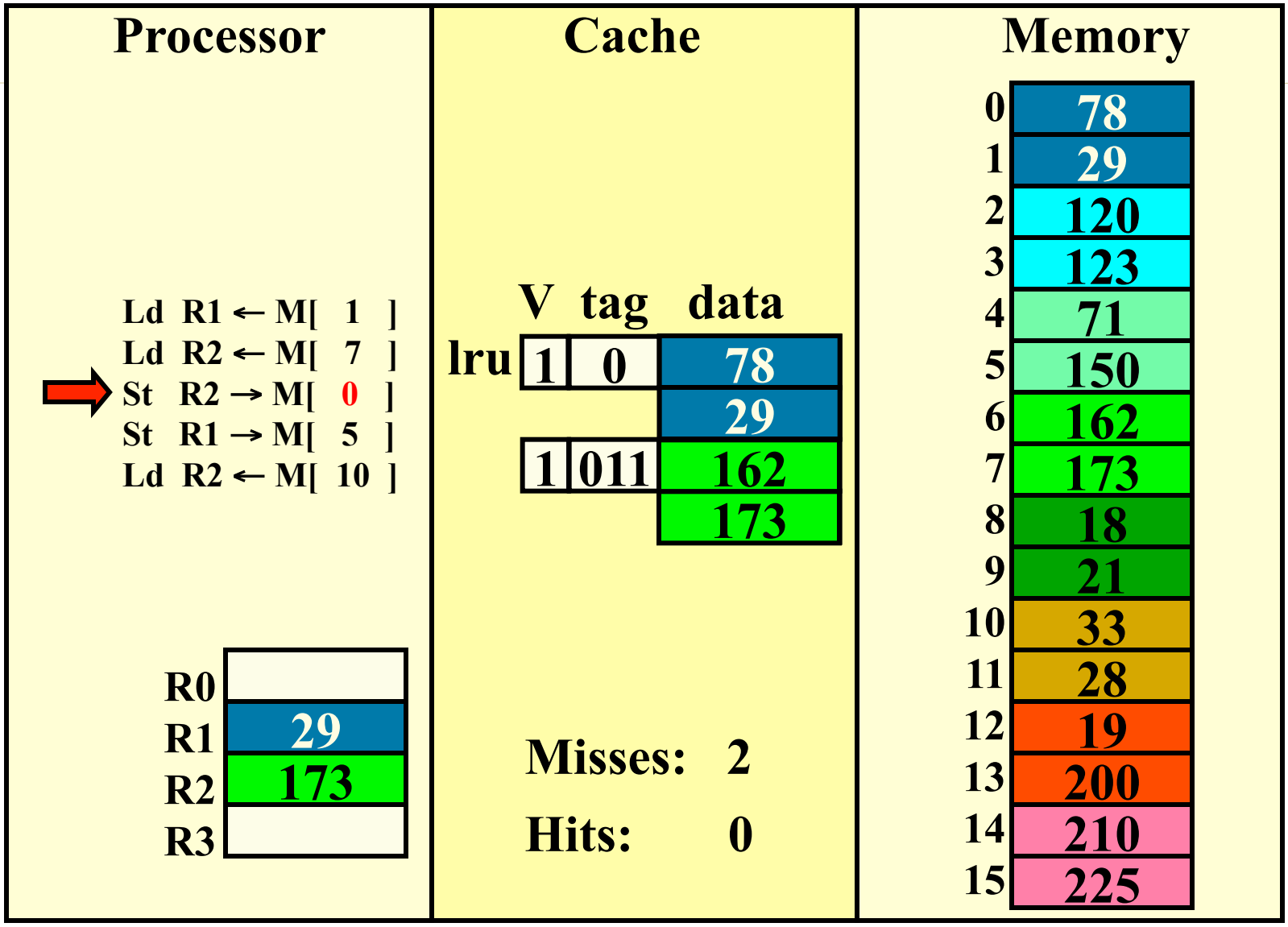
write-through (REF 2)



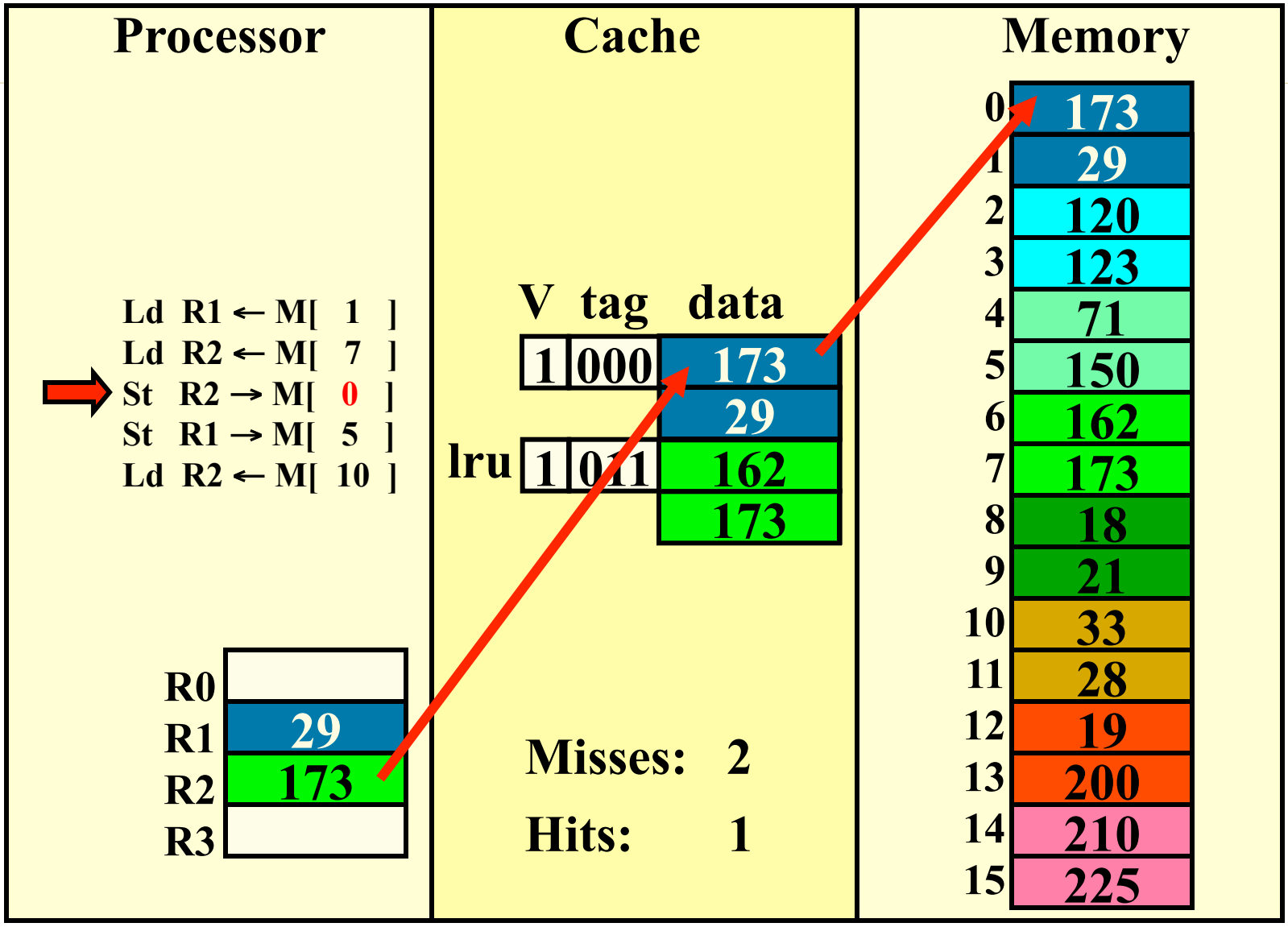
write-through (REF 2)



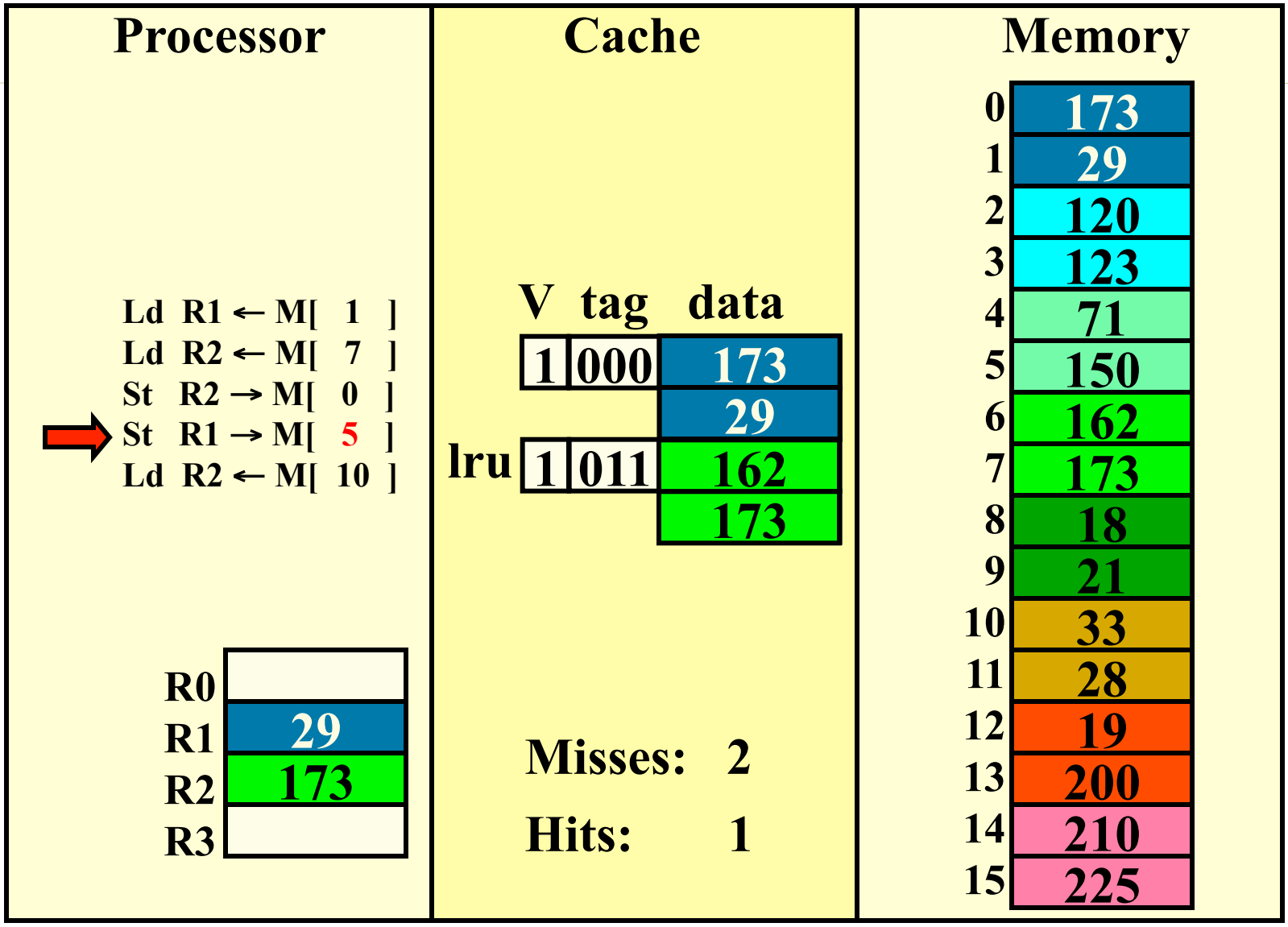
write-through (REF 3)



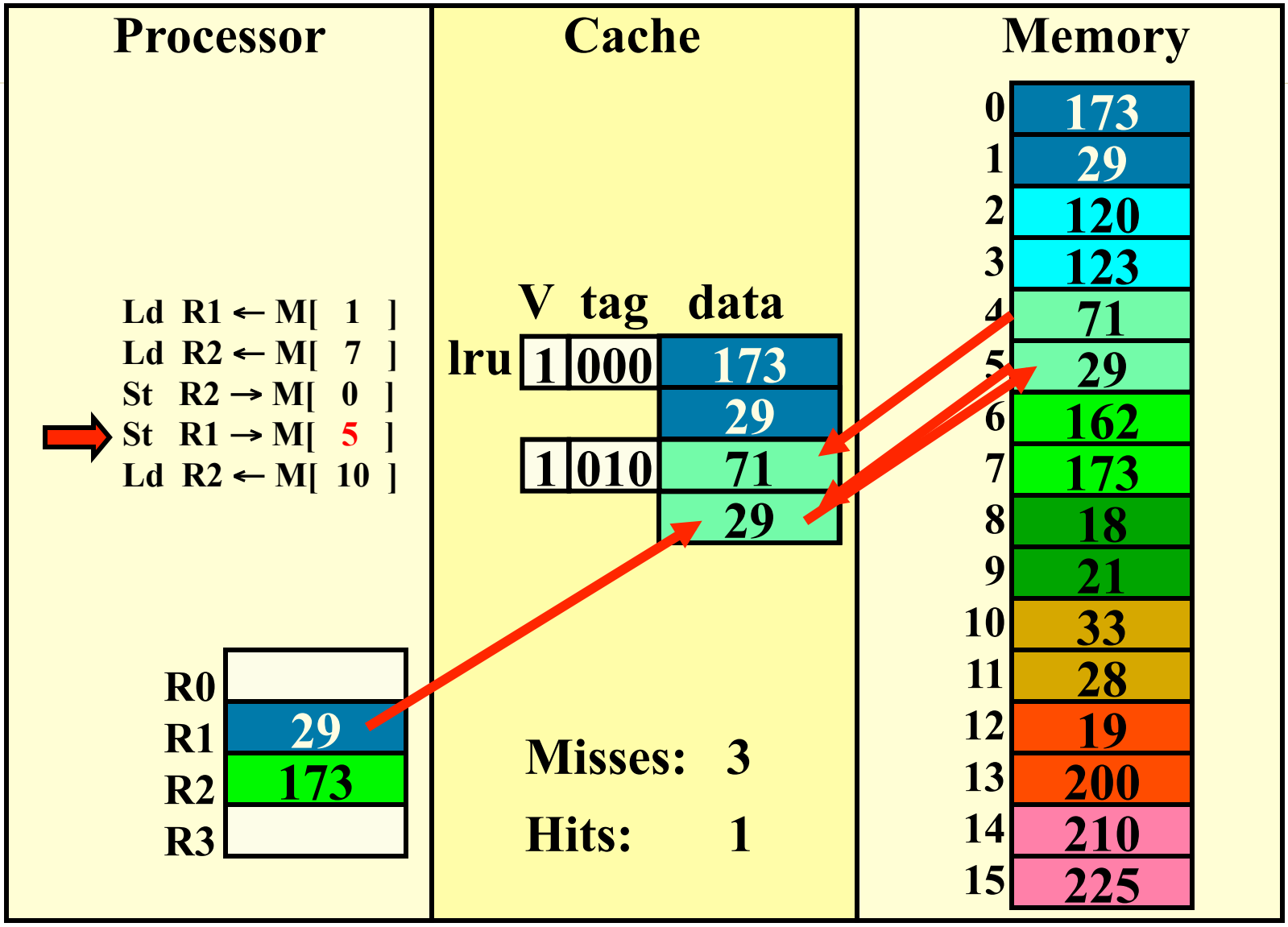
write-through (REF 3)



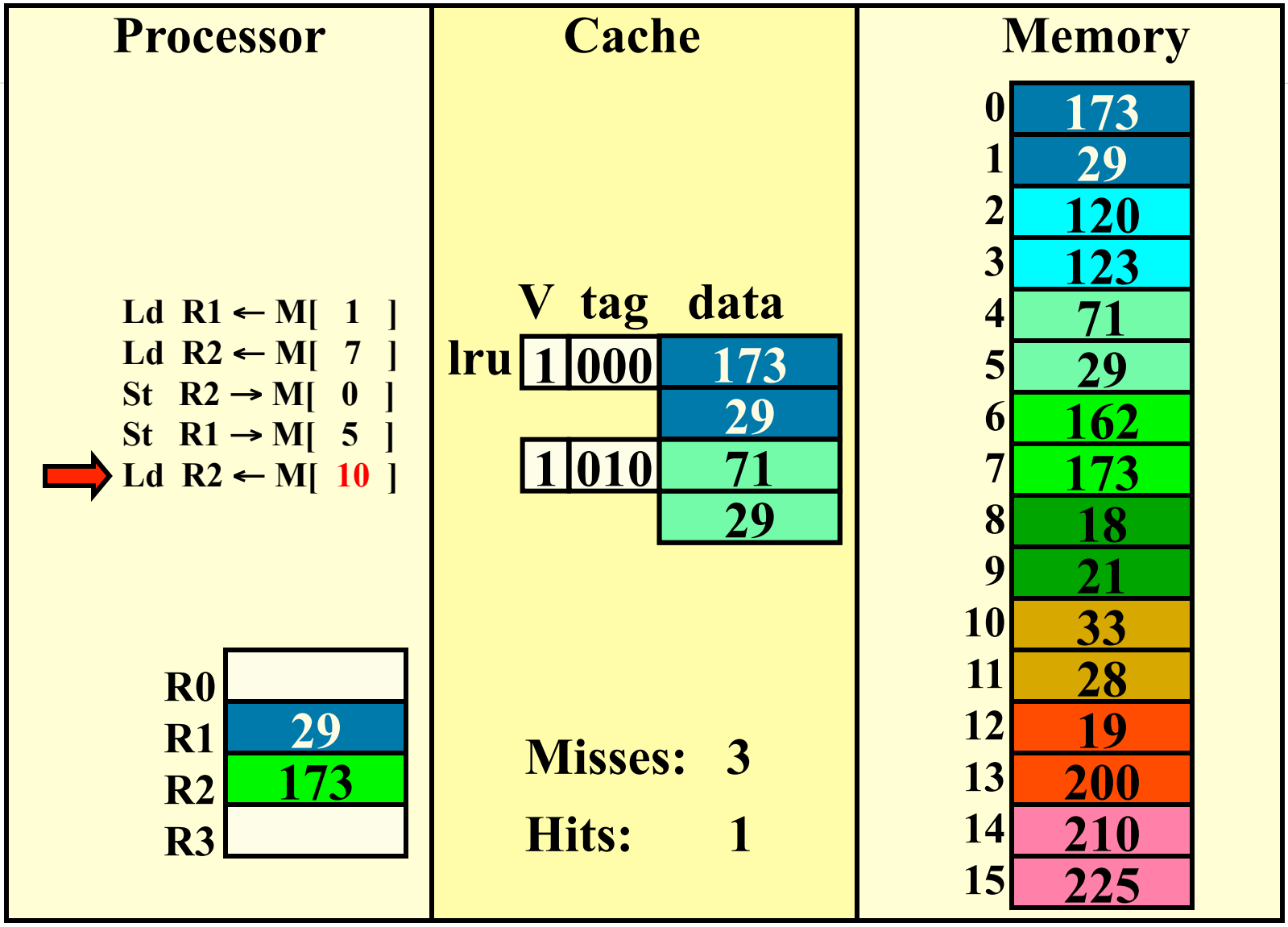
write-through (REF 4)



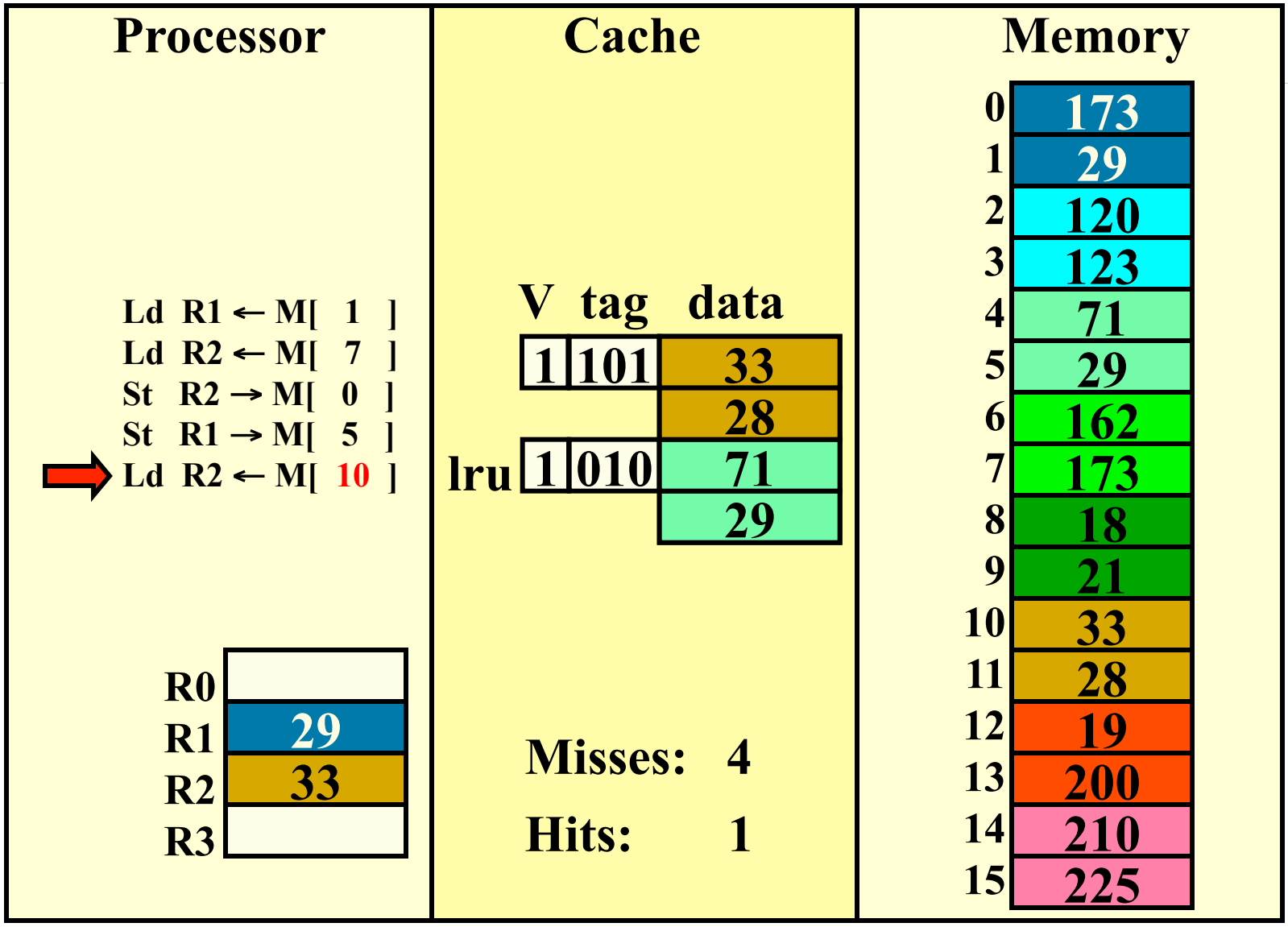
write-through (REF 4)



write-through (REF 6)



write-through (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%

Class Problem 3

How many hits/misses result when the following code is executed. Assume a no-write allocate policy

Processor	Cache	Memory

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

	V	tag	data
	0		
	0		

R0	
R1	
R2	
R3	

Misses: 0

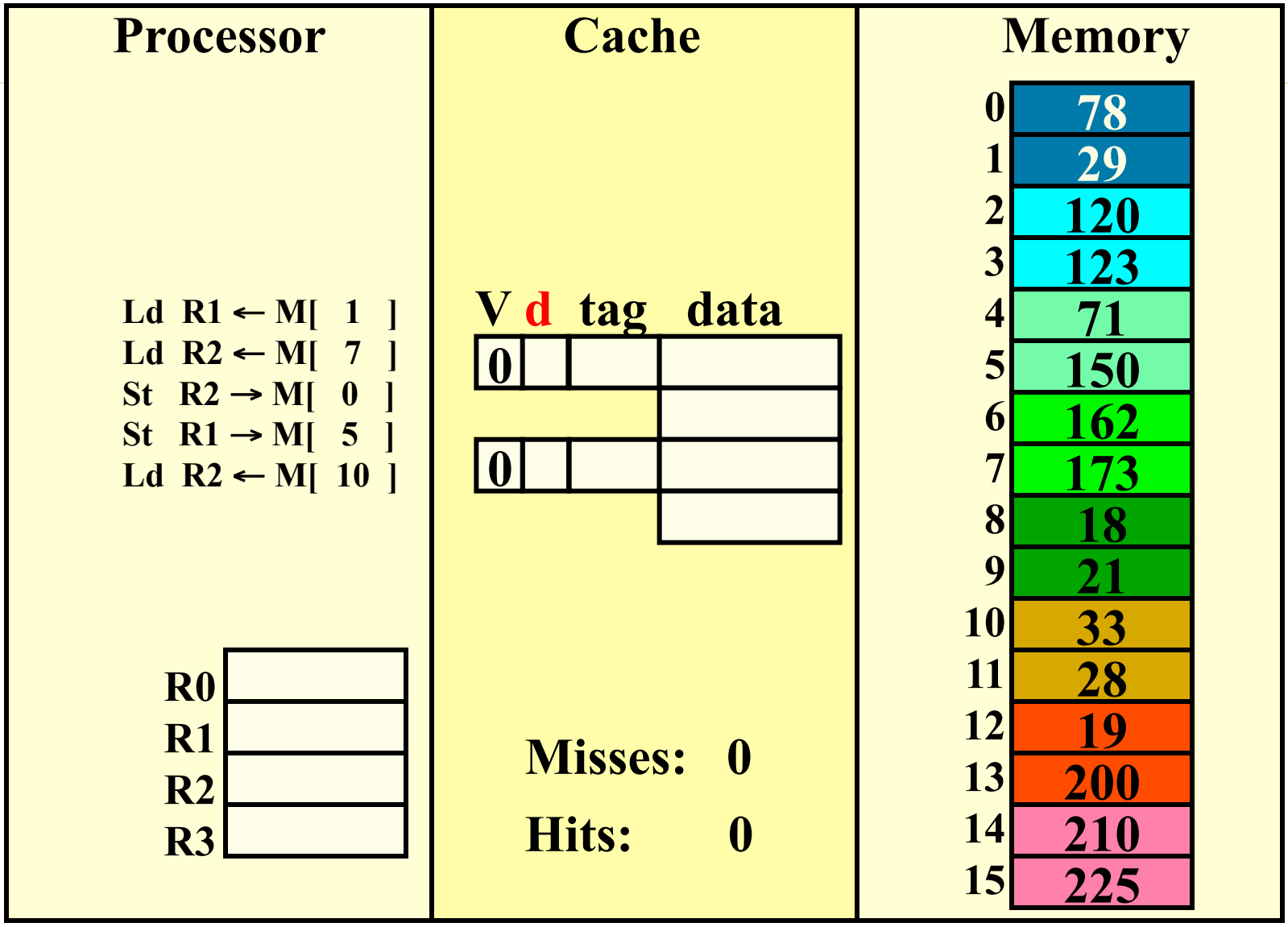
Hits: 0

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

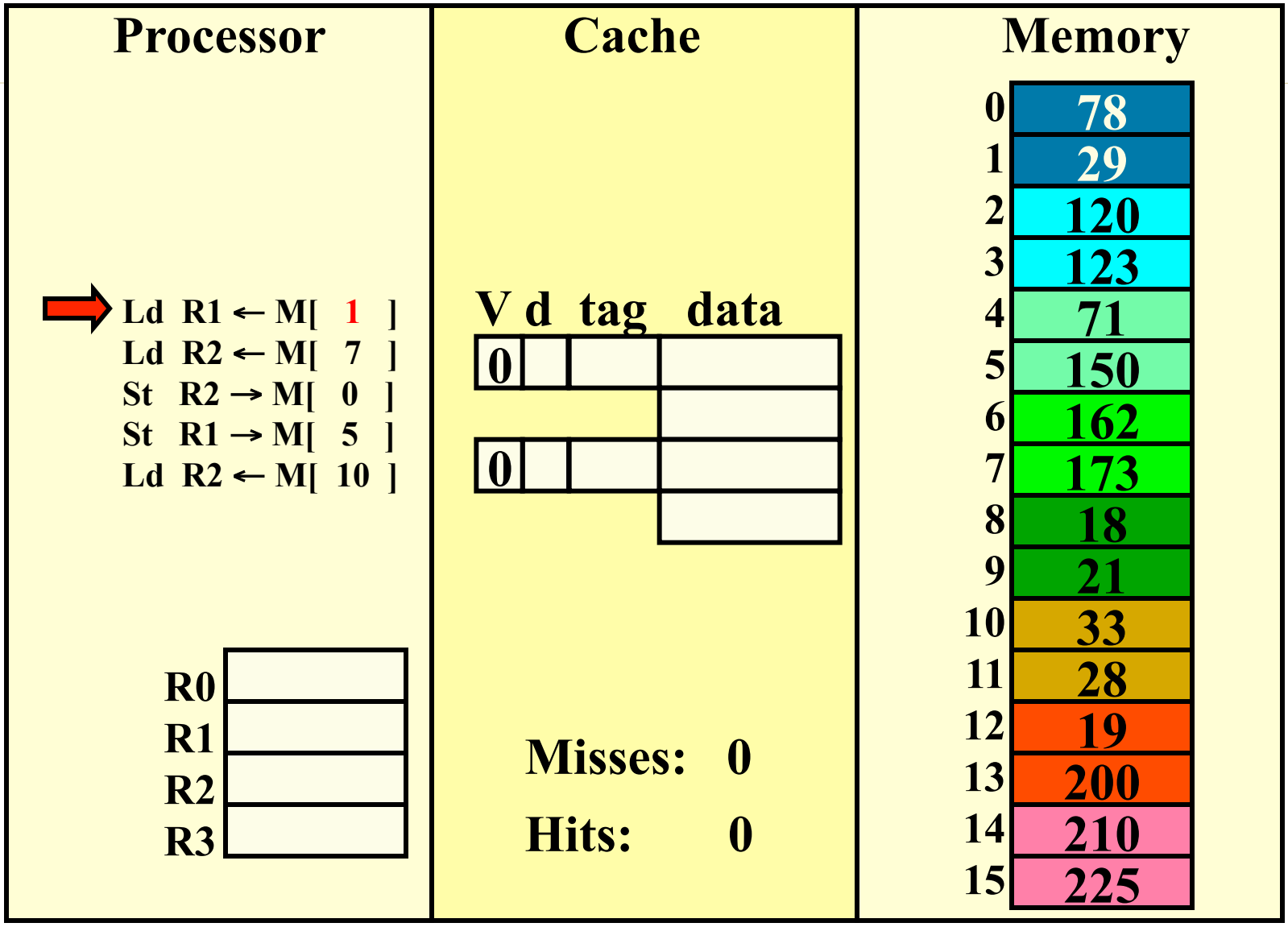
Write-through vs. write-back

- We can also design the cache to NOT write all stores to memory immediately?
  - We can keep the most current copy in the cache and update the memory when that data is evicted from the cache (a write-back policy).
    - Do we need to 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.

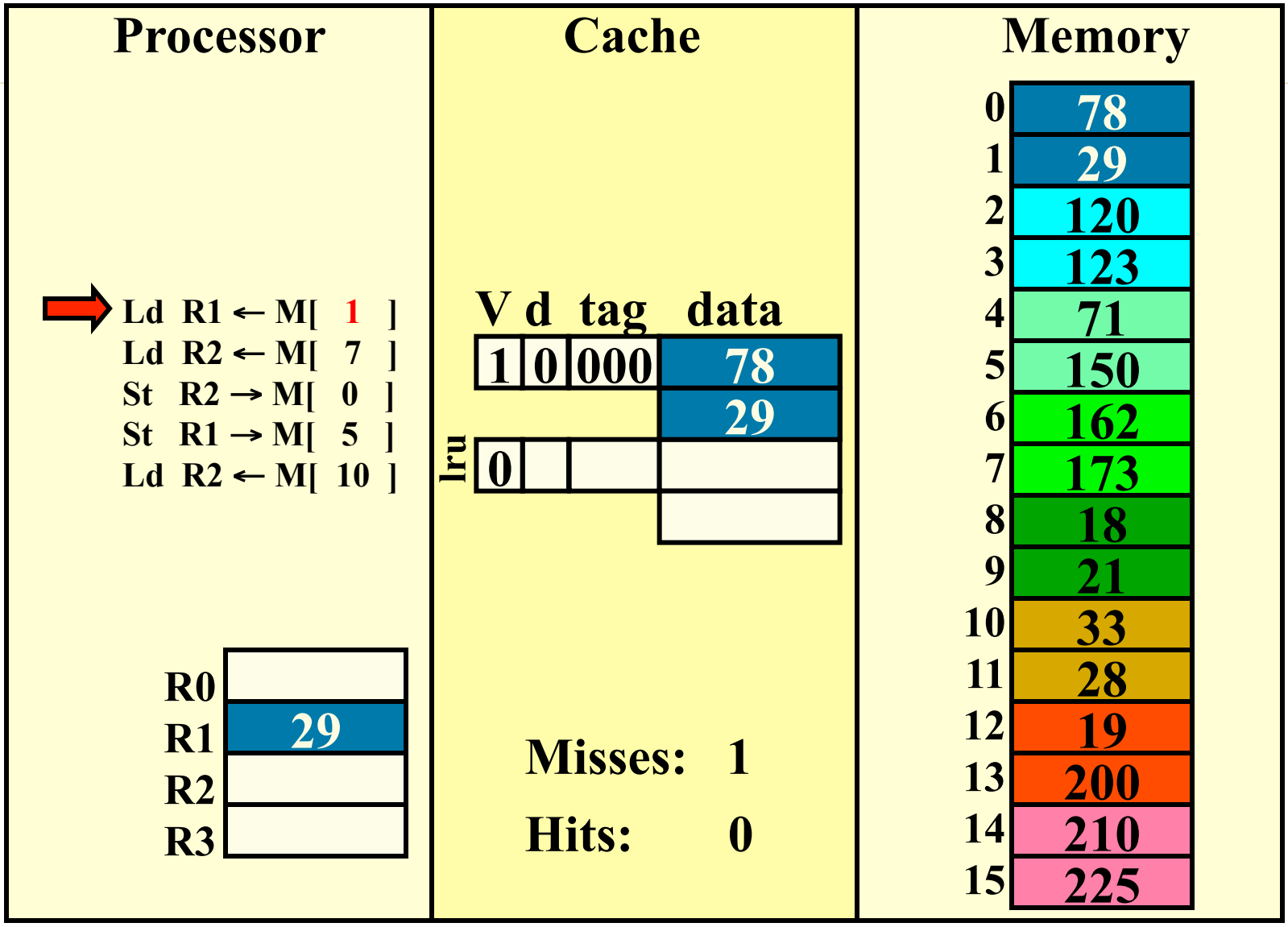
Handling Stores (write-back)



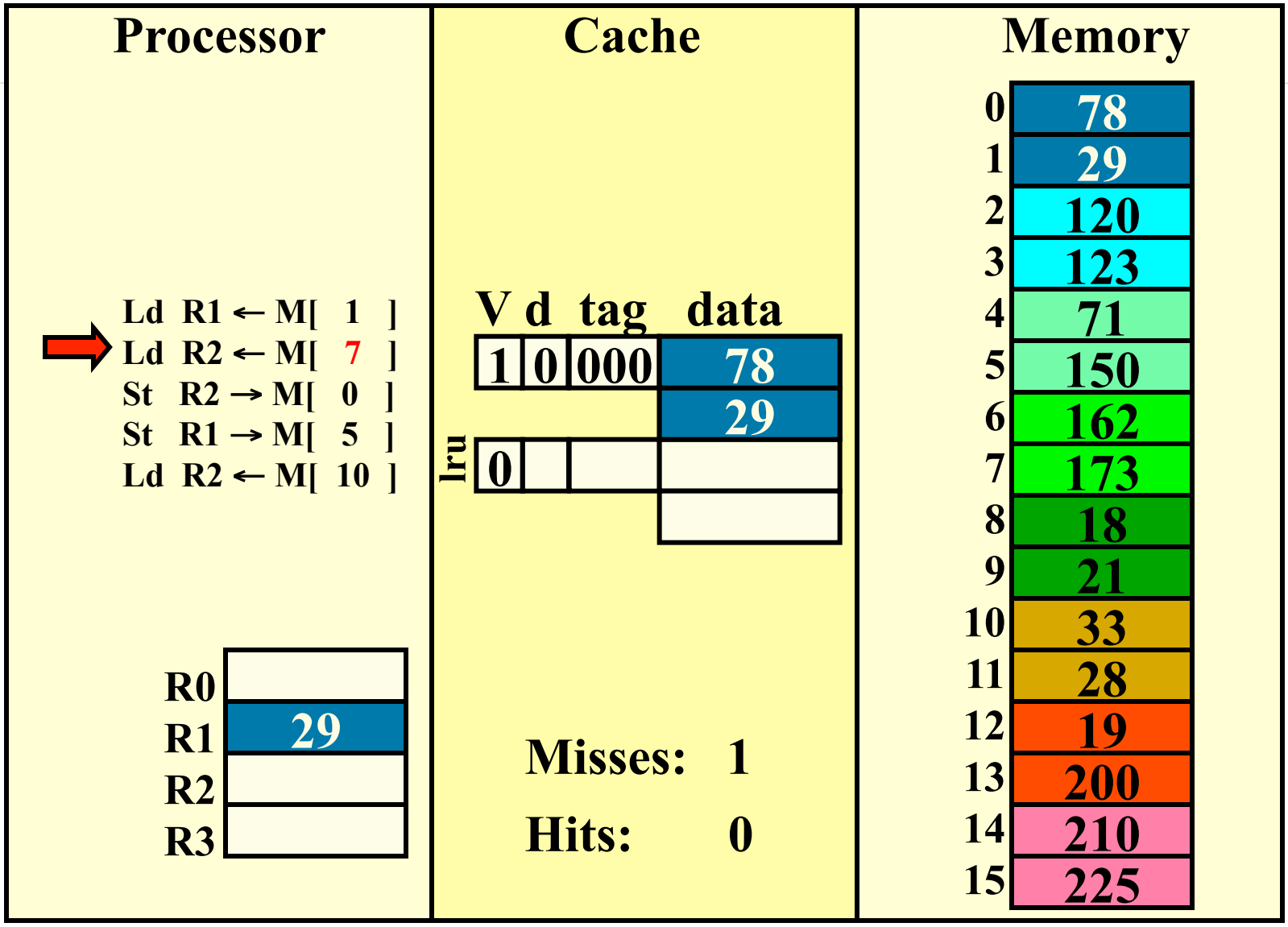
write-back (REF 1)



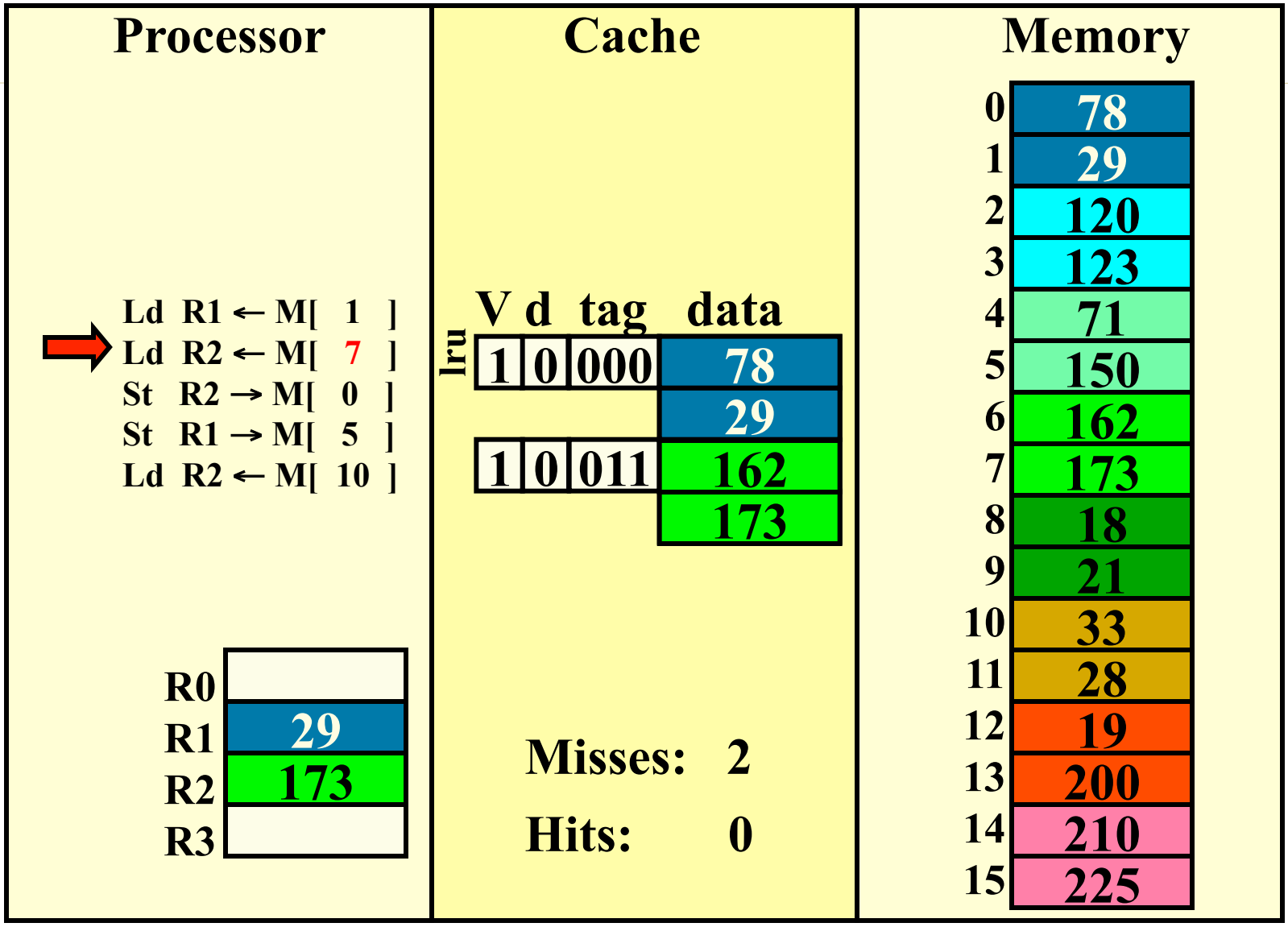
write-back (REF 1)



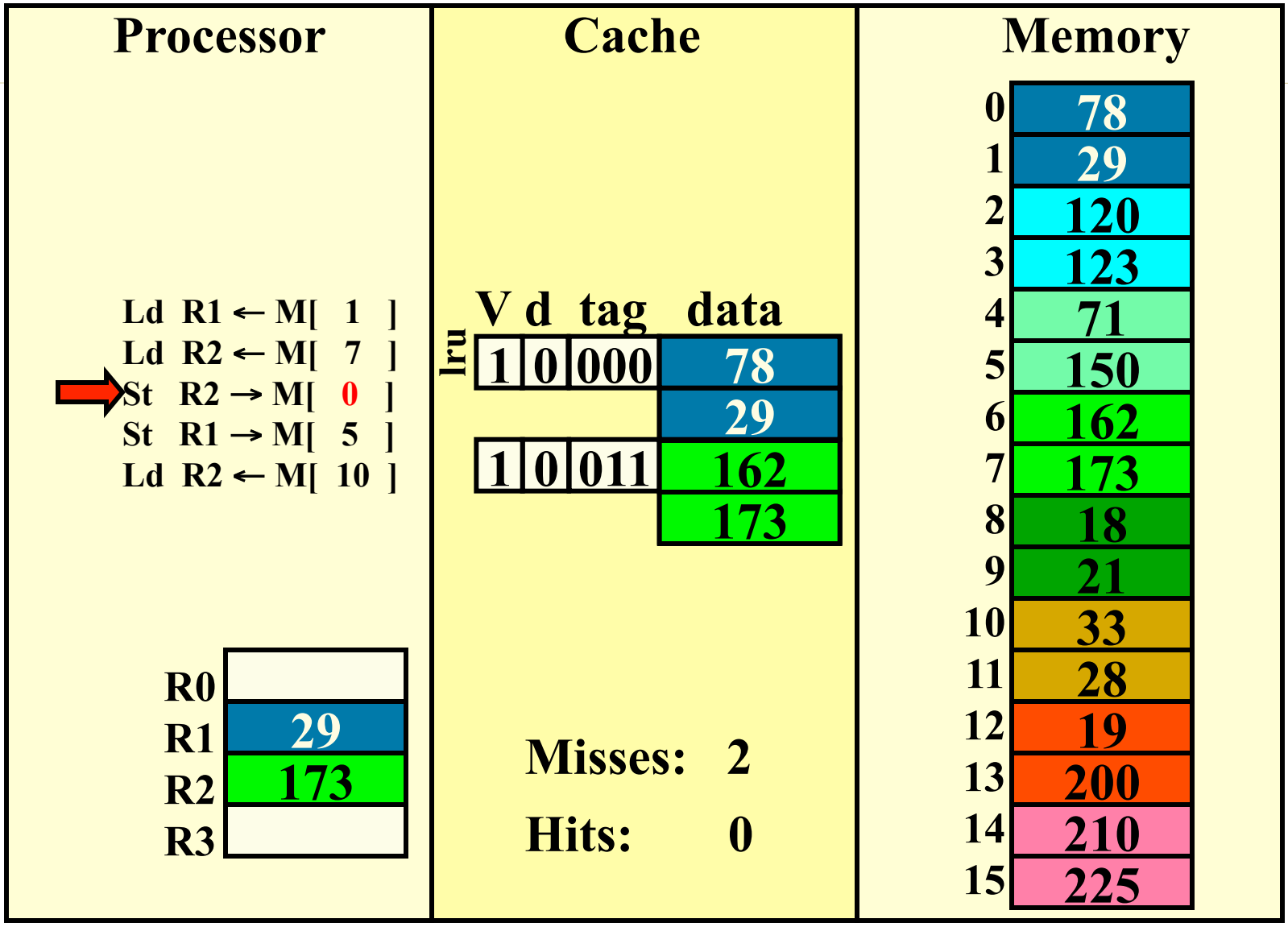
write-back (REF 2)



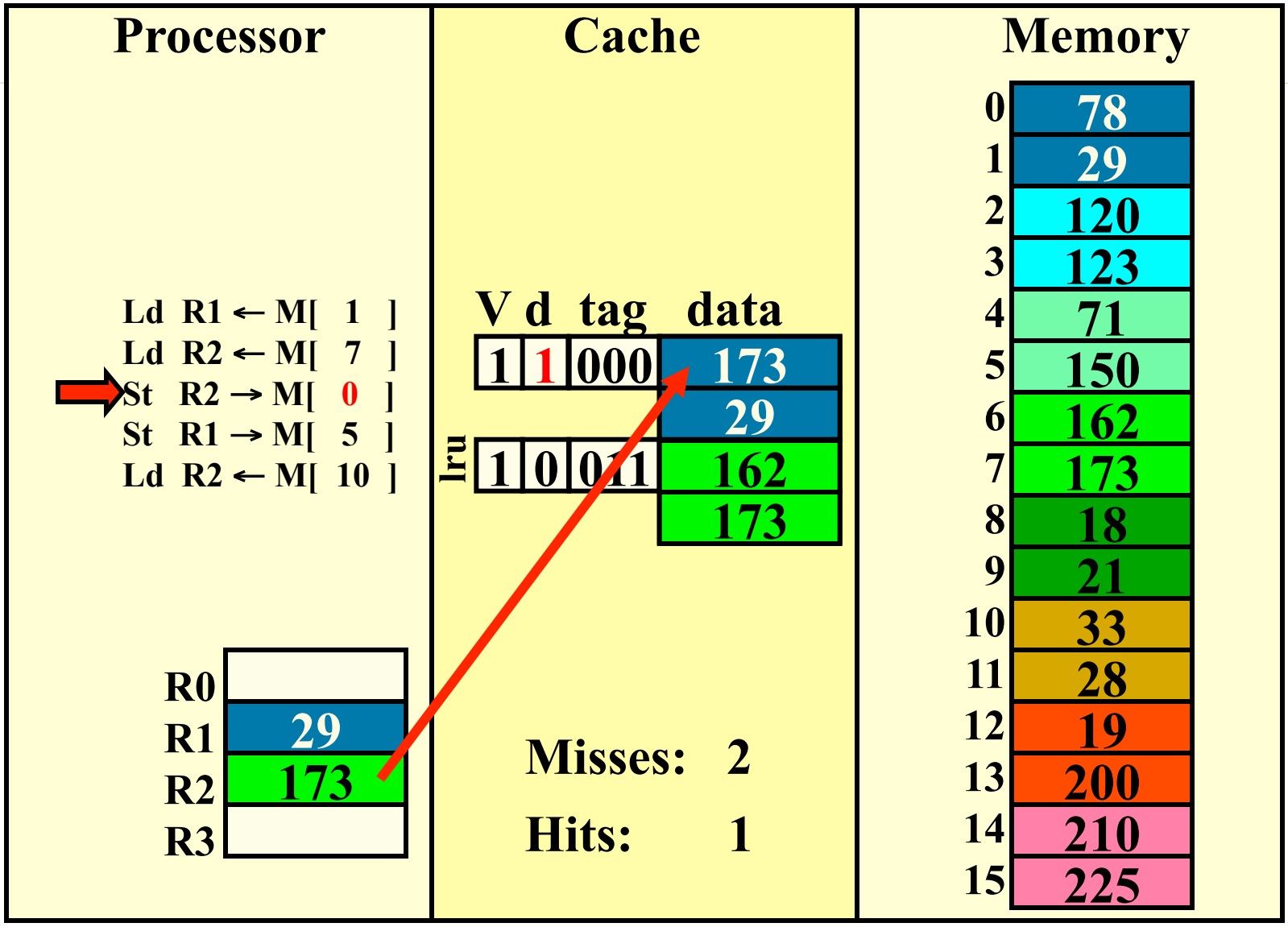
write-back (REF 2)



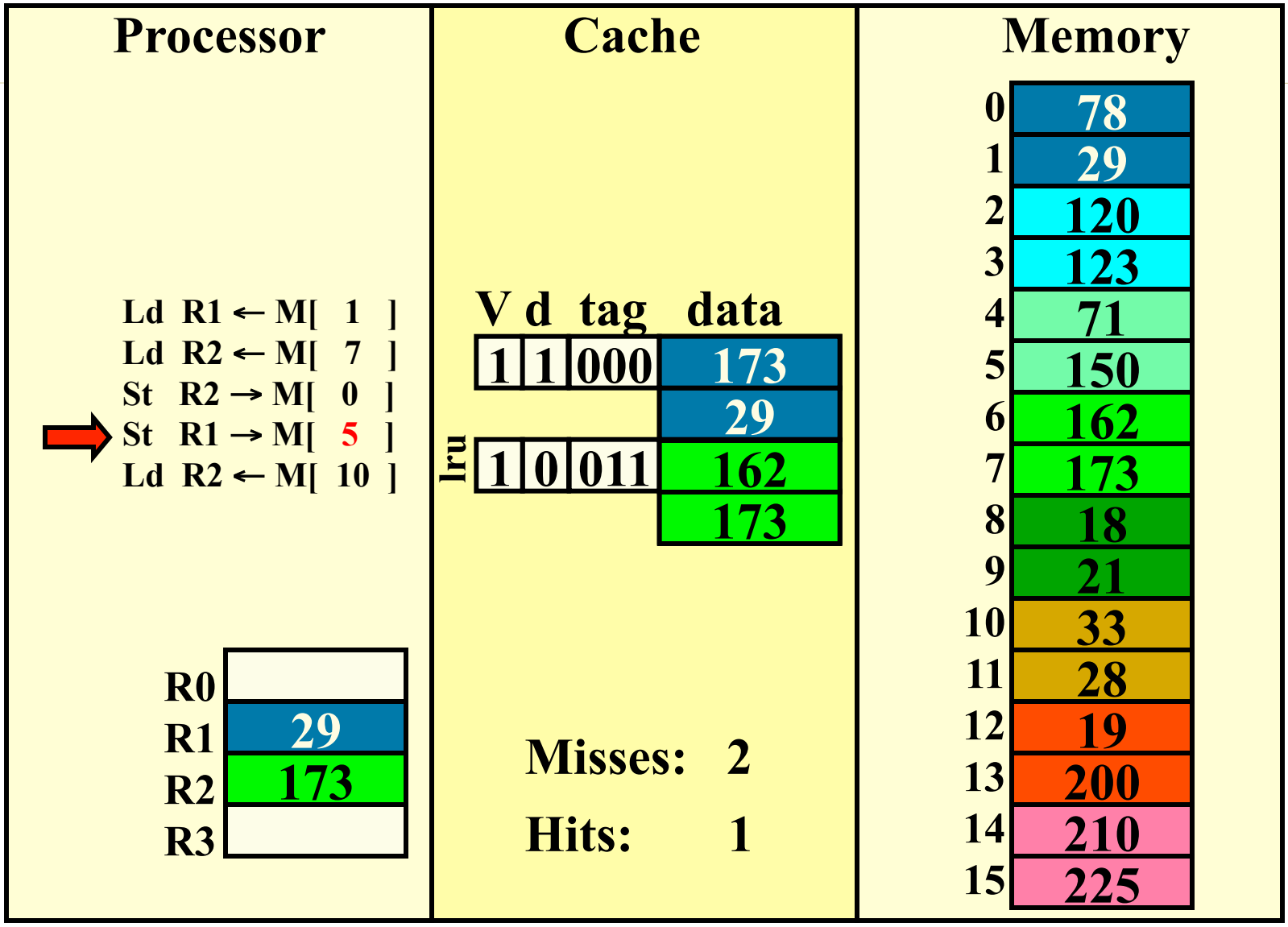
write-back (REF 3)



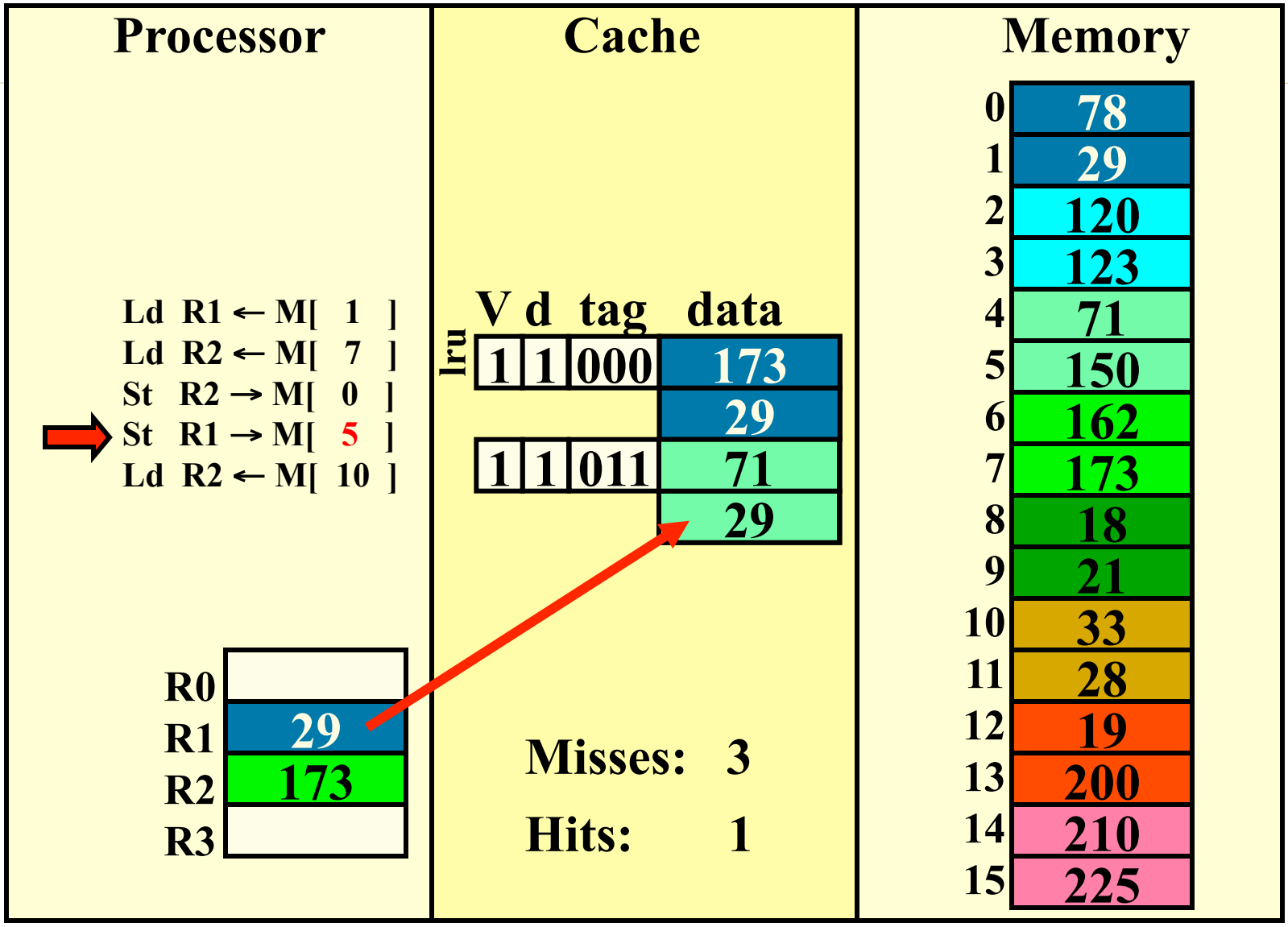
write-back (REF 3)



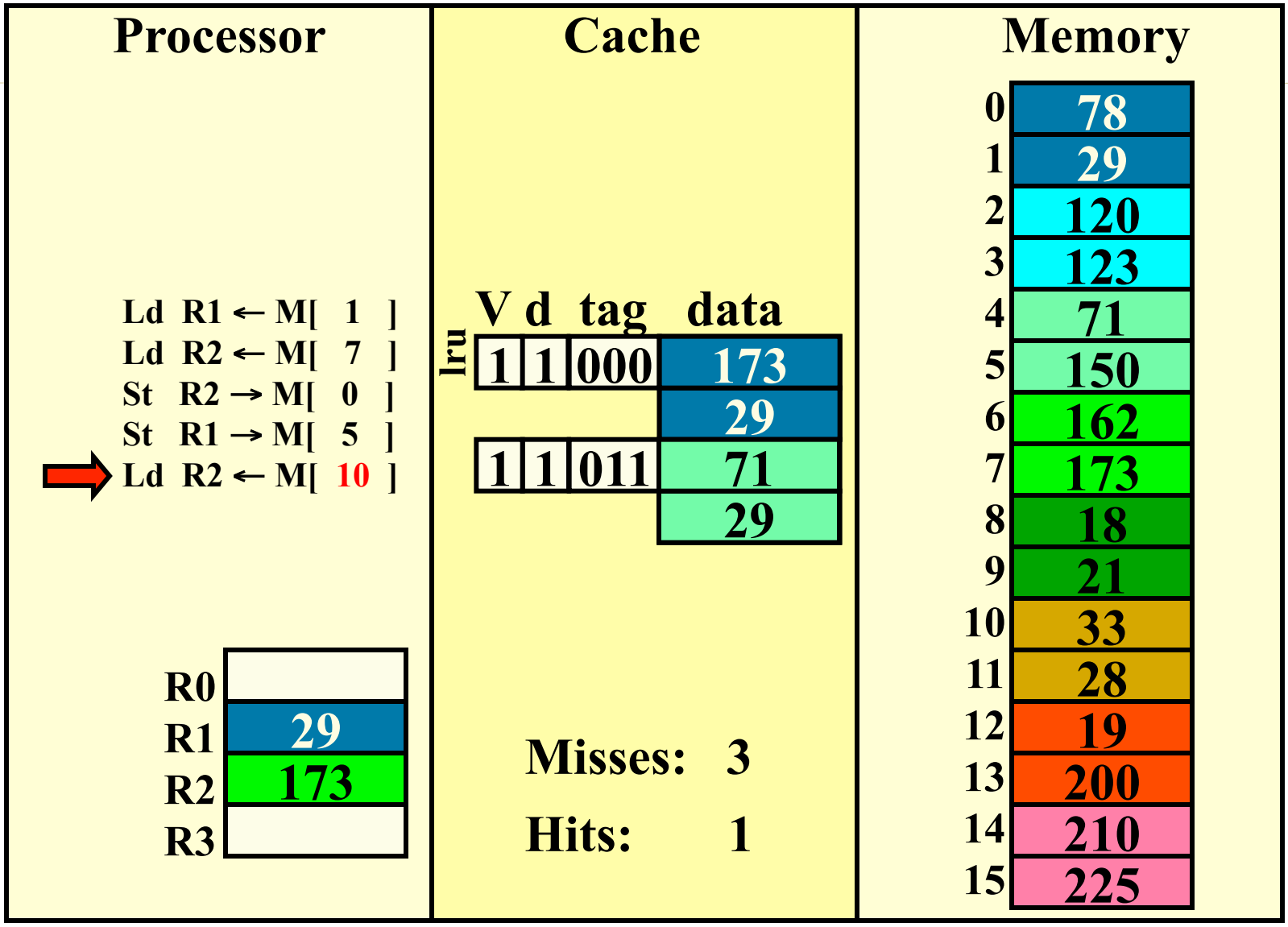
write-back (REF 4)



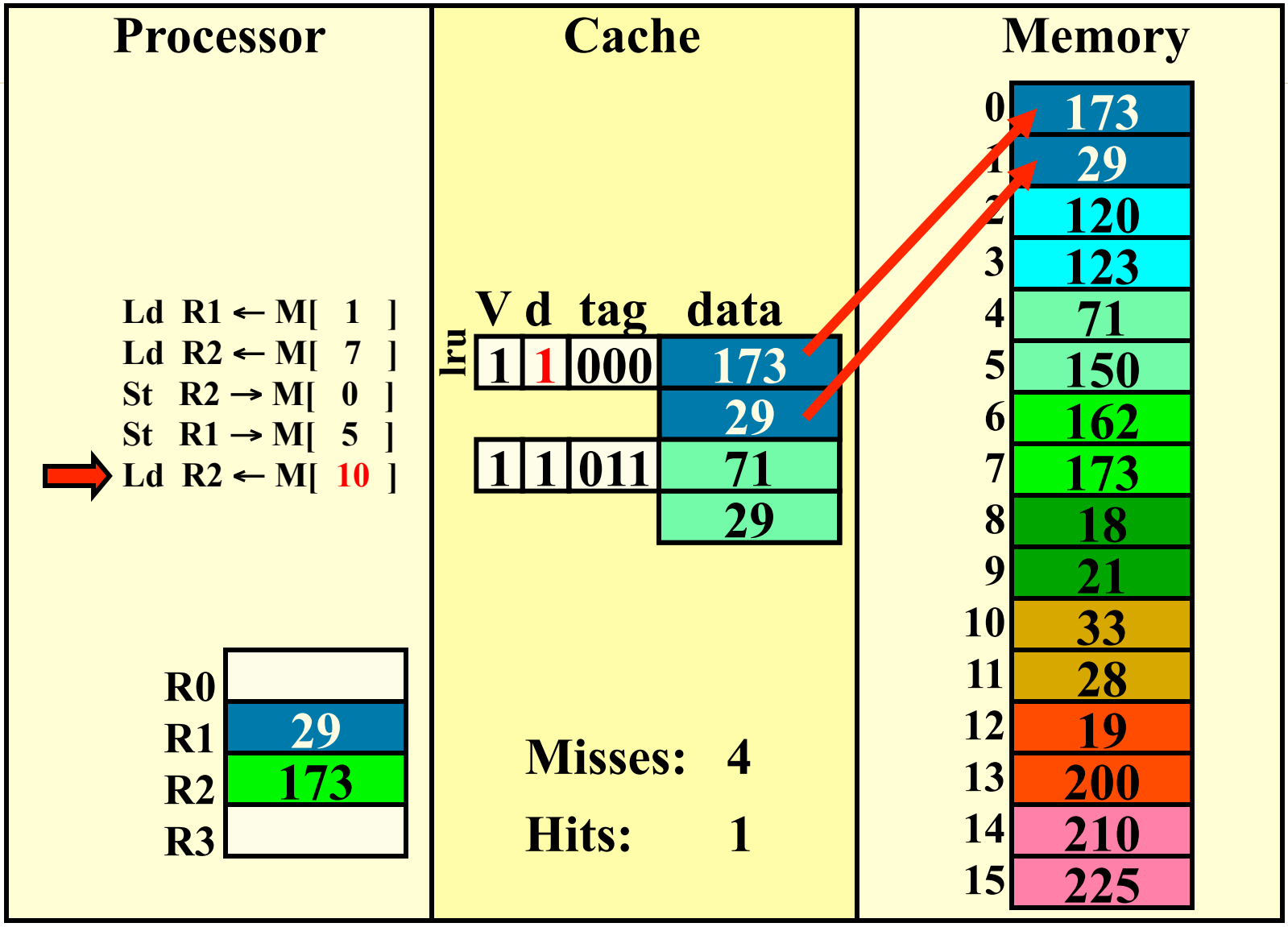
write-back (REF 4)



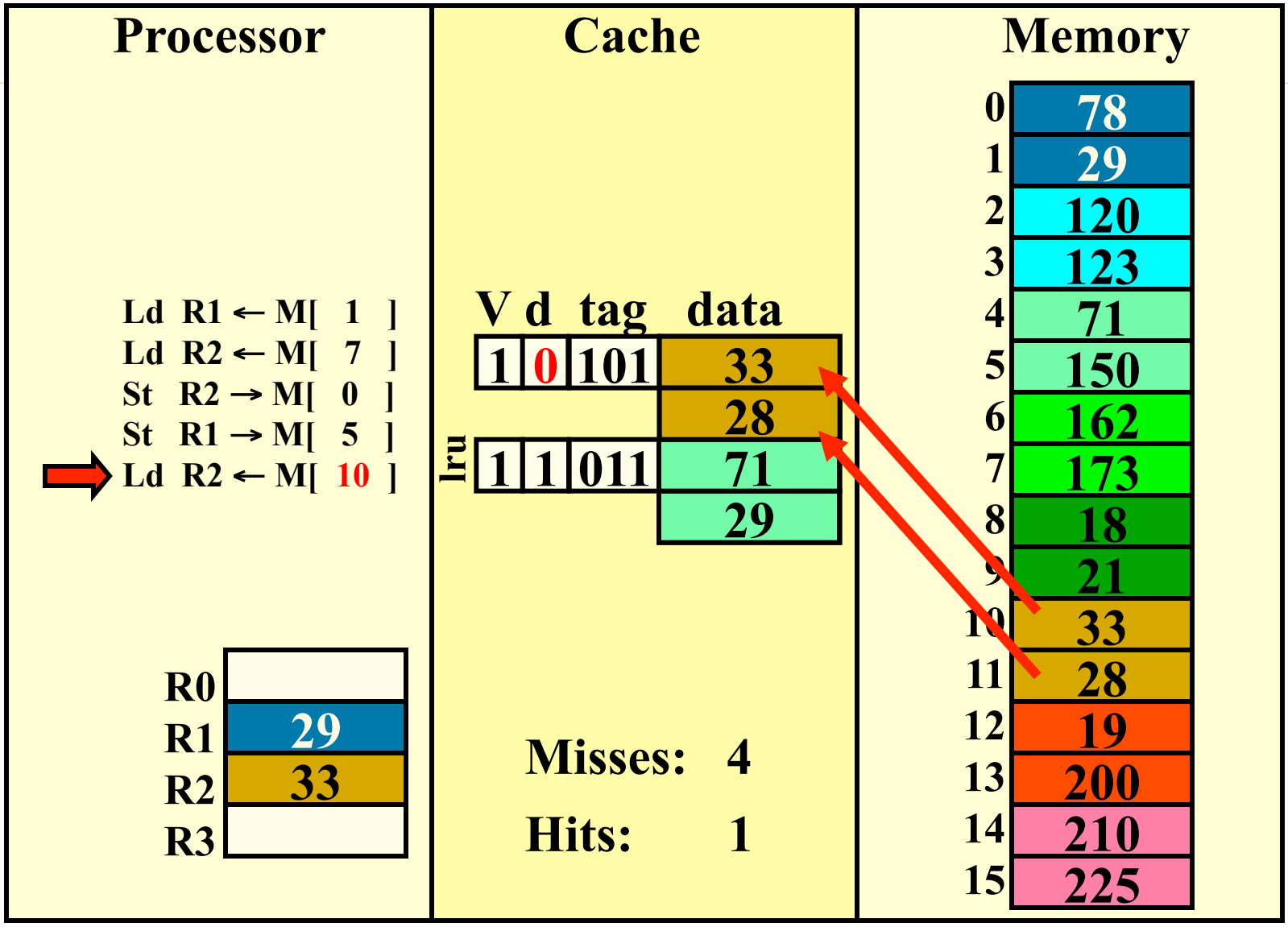
write-back (REF 5)



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?

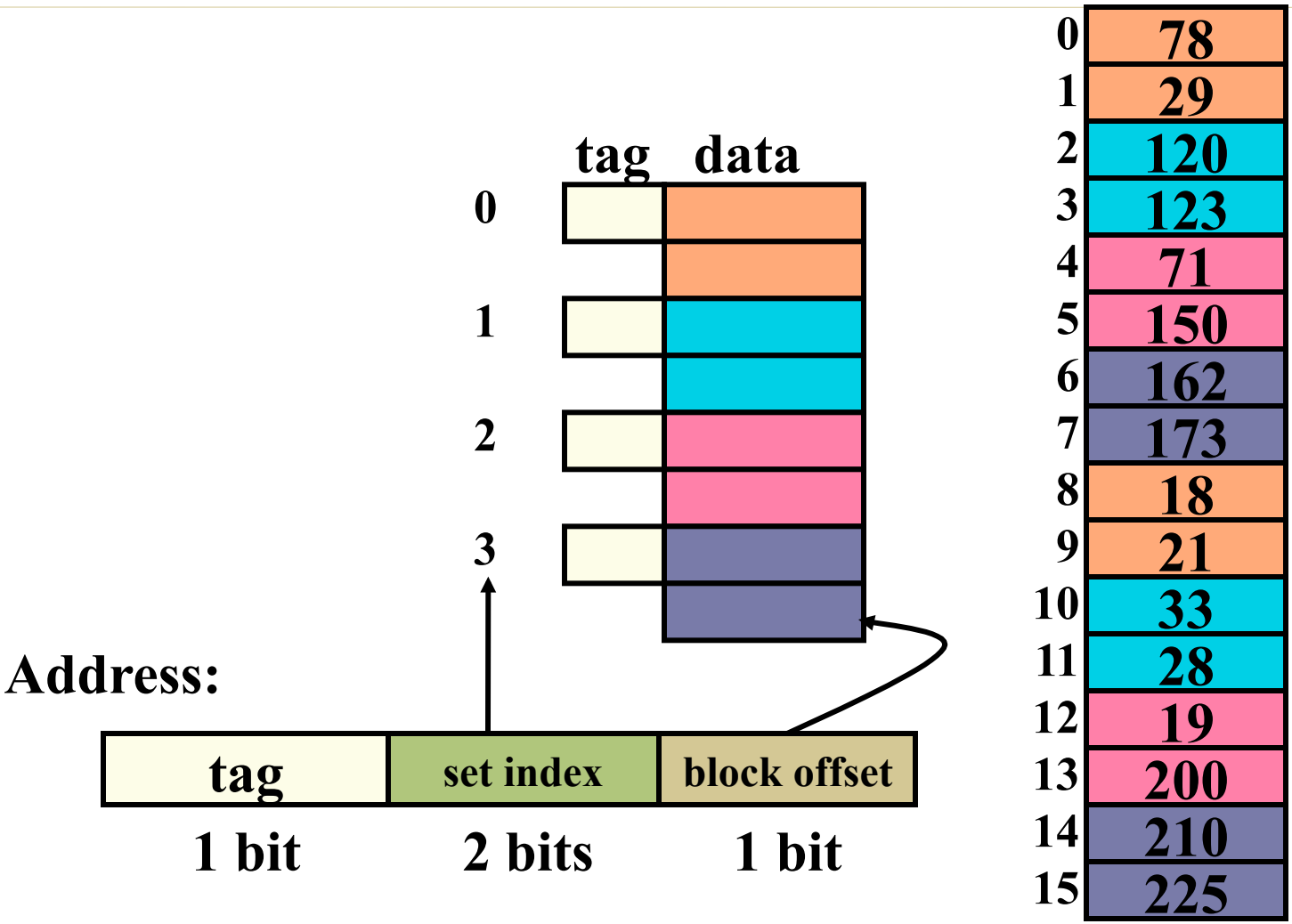
Associativity

- 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 slower. Why?

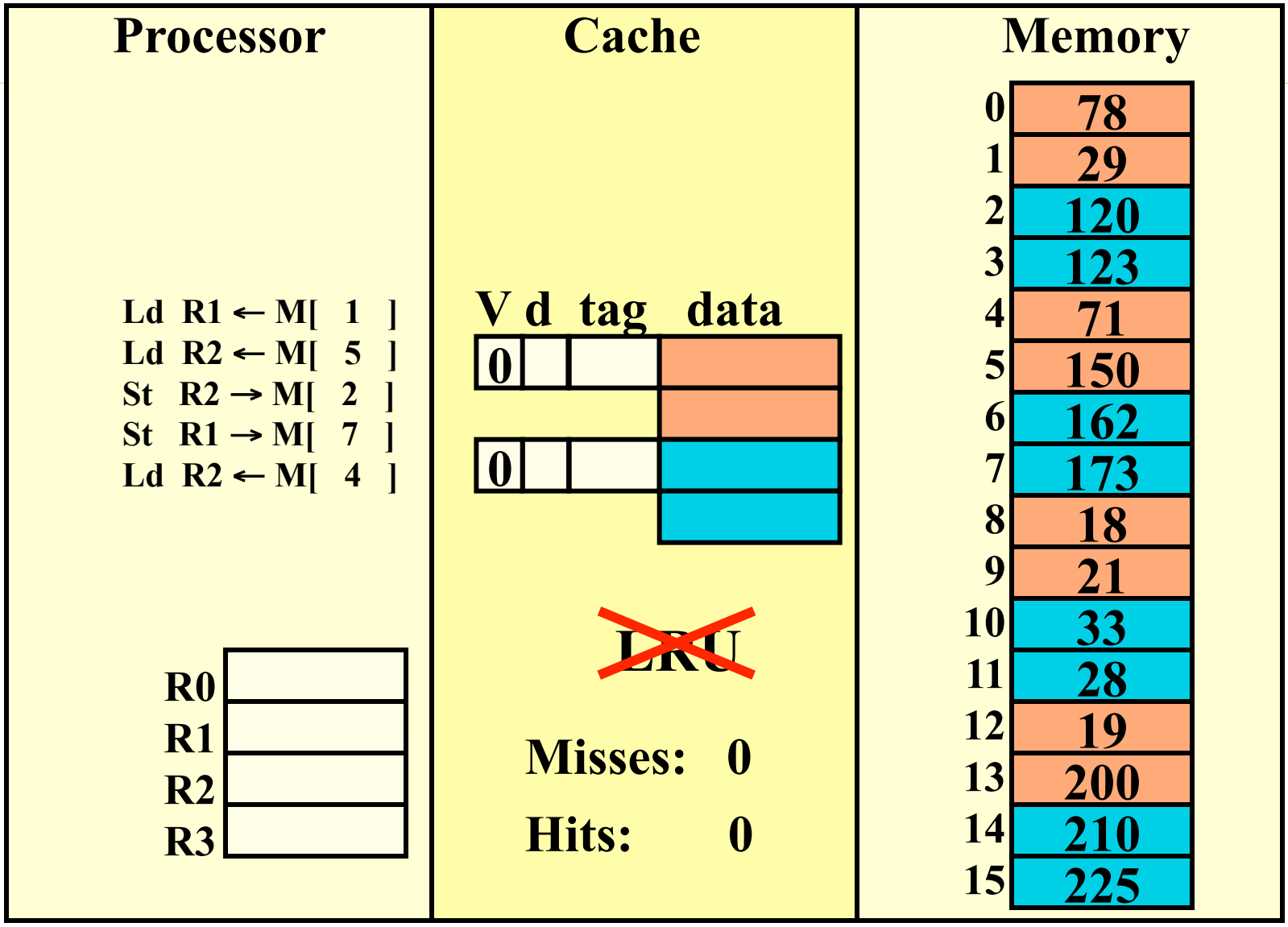
Direct mapped cache

- 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 has 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.

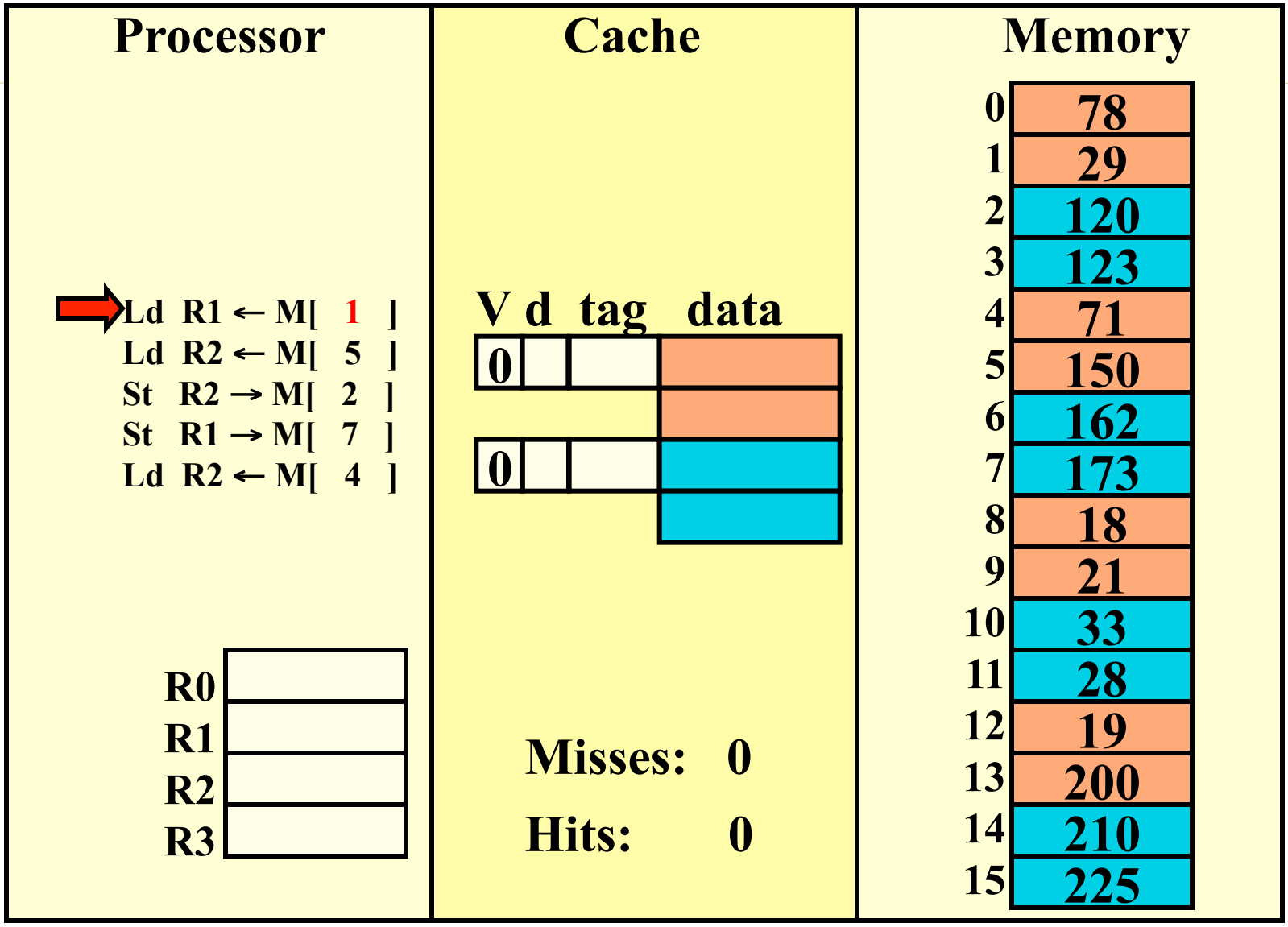
Mapping Memory to Cache



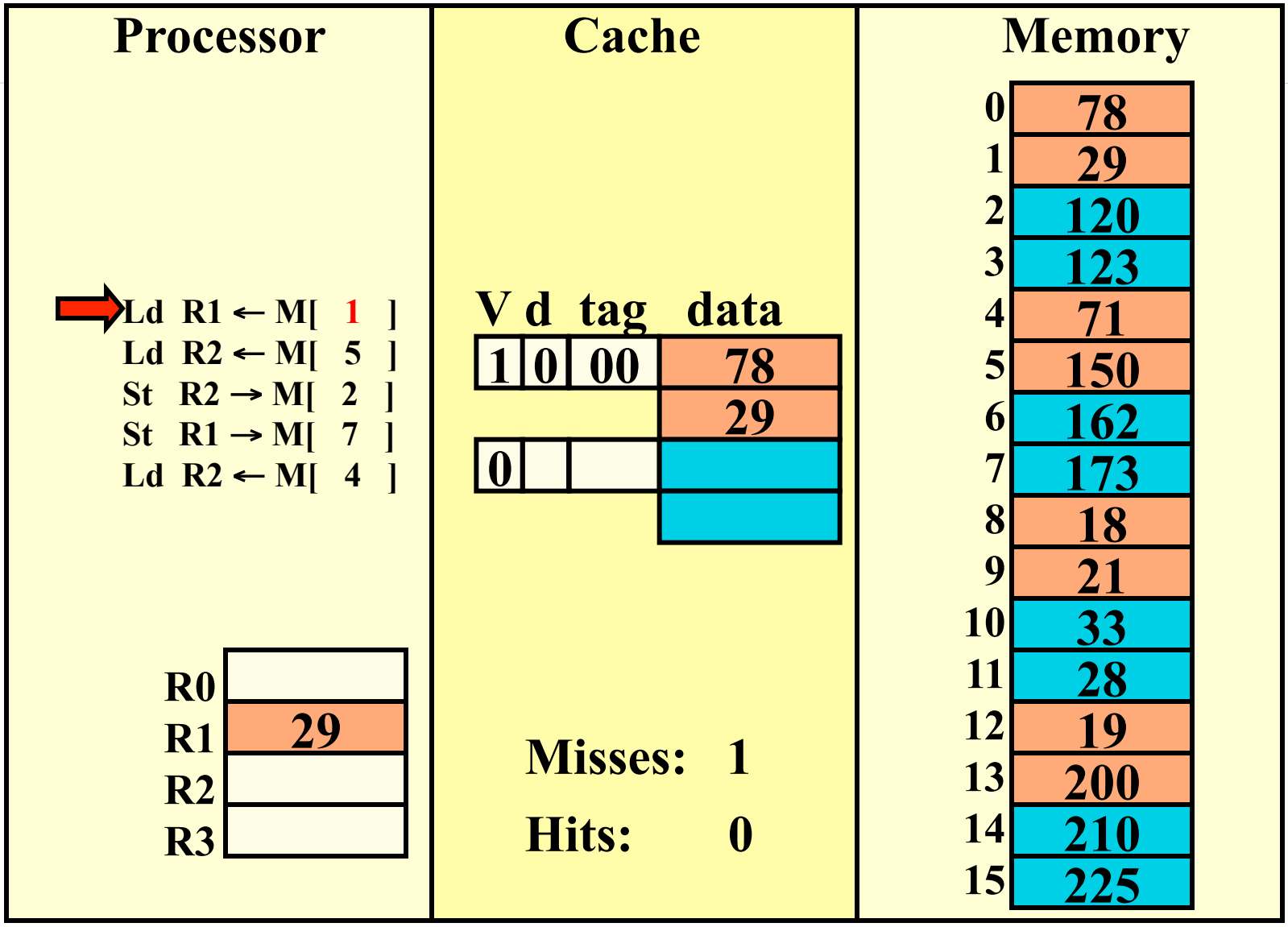
Direct-mapped Cache



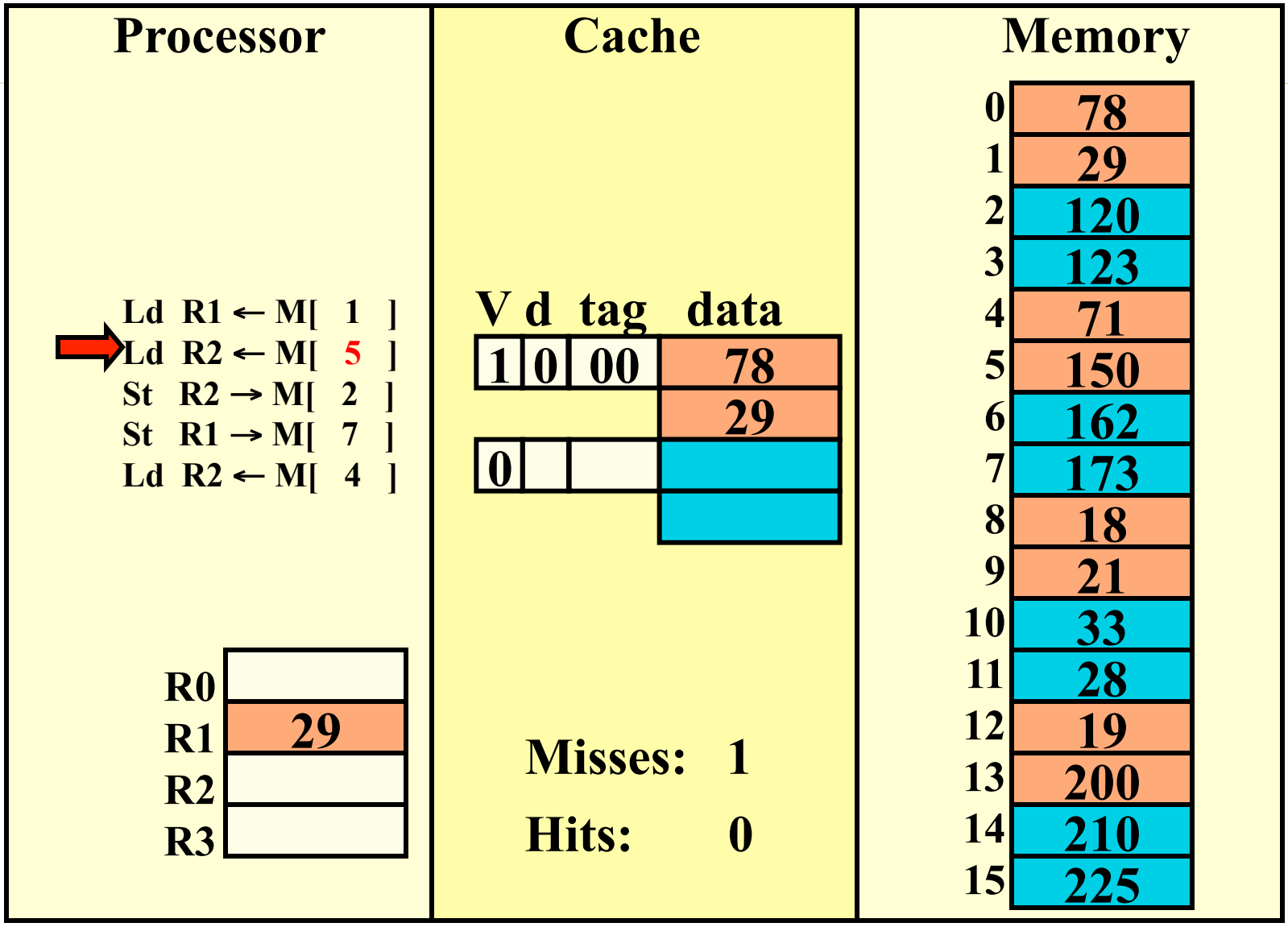
Direct-mapped (REF 1)



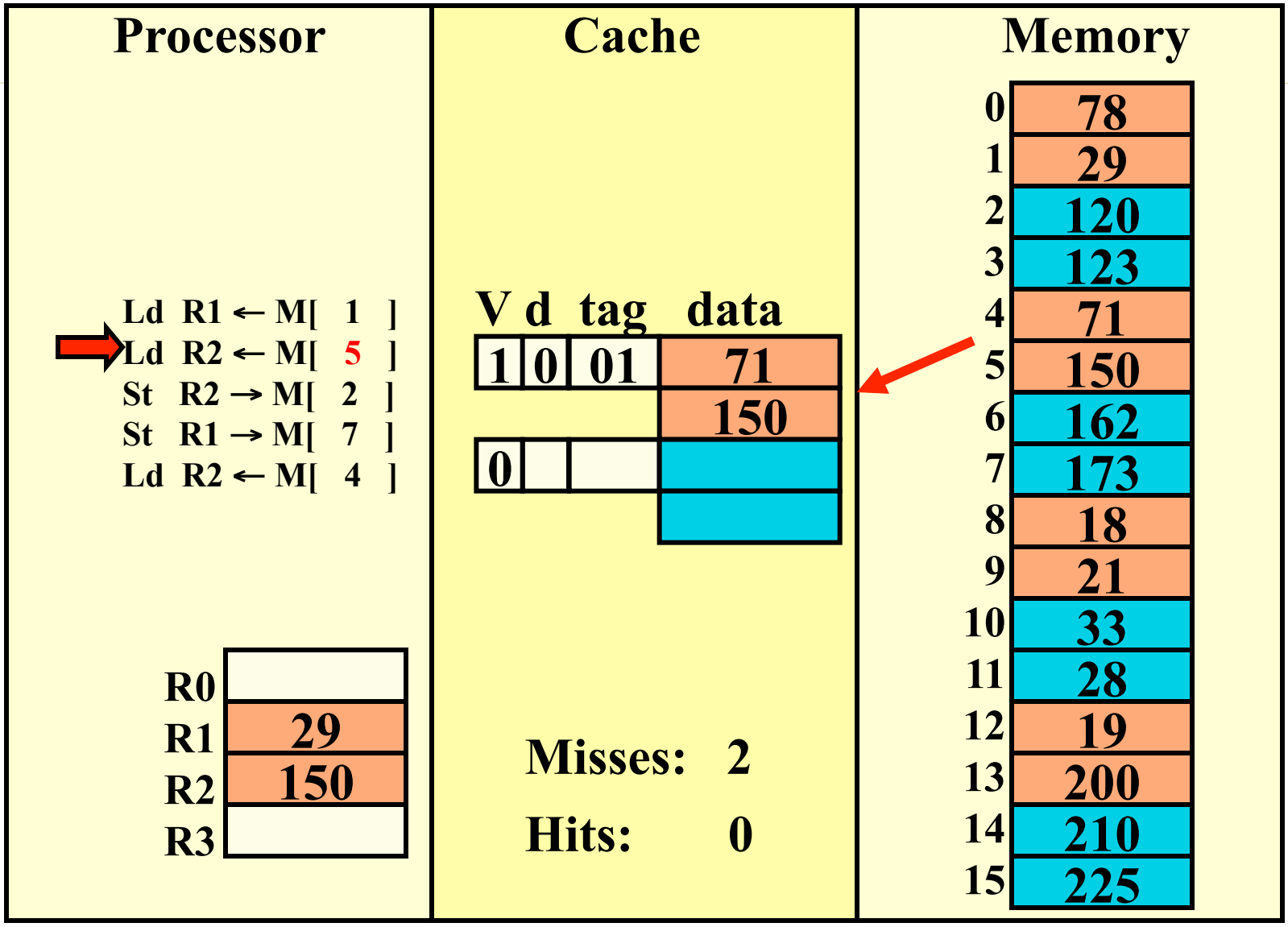
Direct-mapped (REF 1)



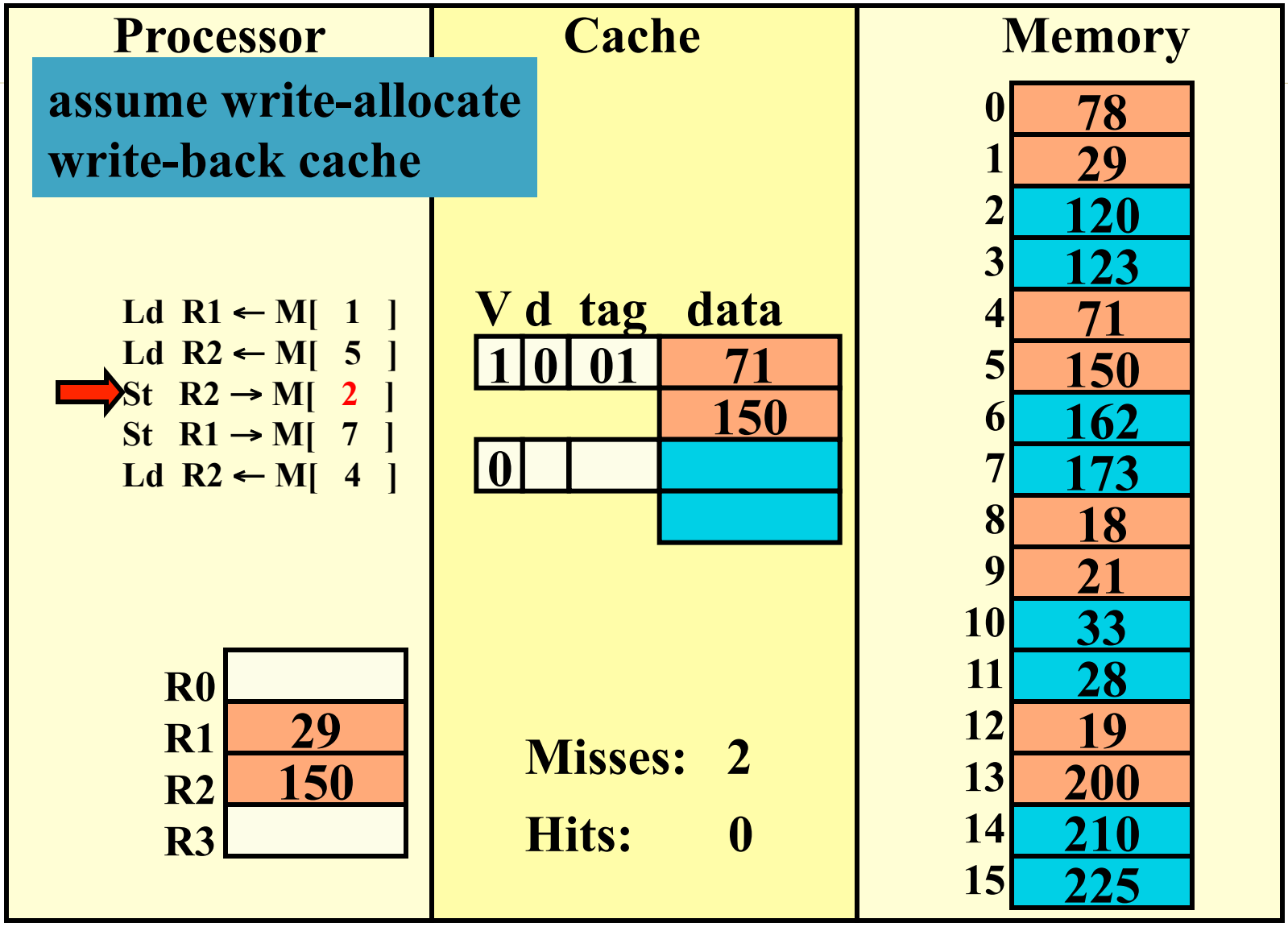
Direct-mapped (REF 2)



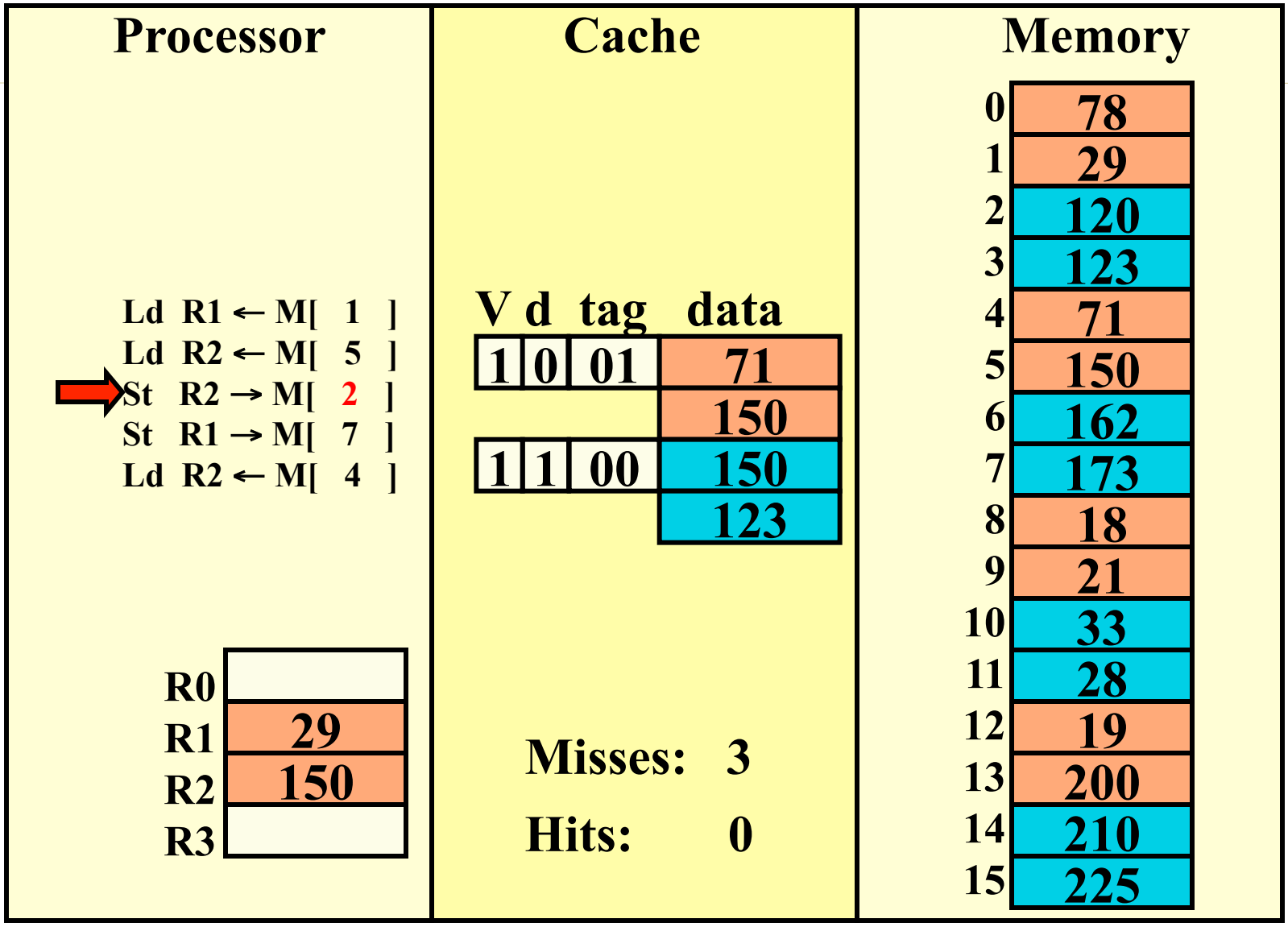
Direct-mapped (REF 2)



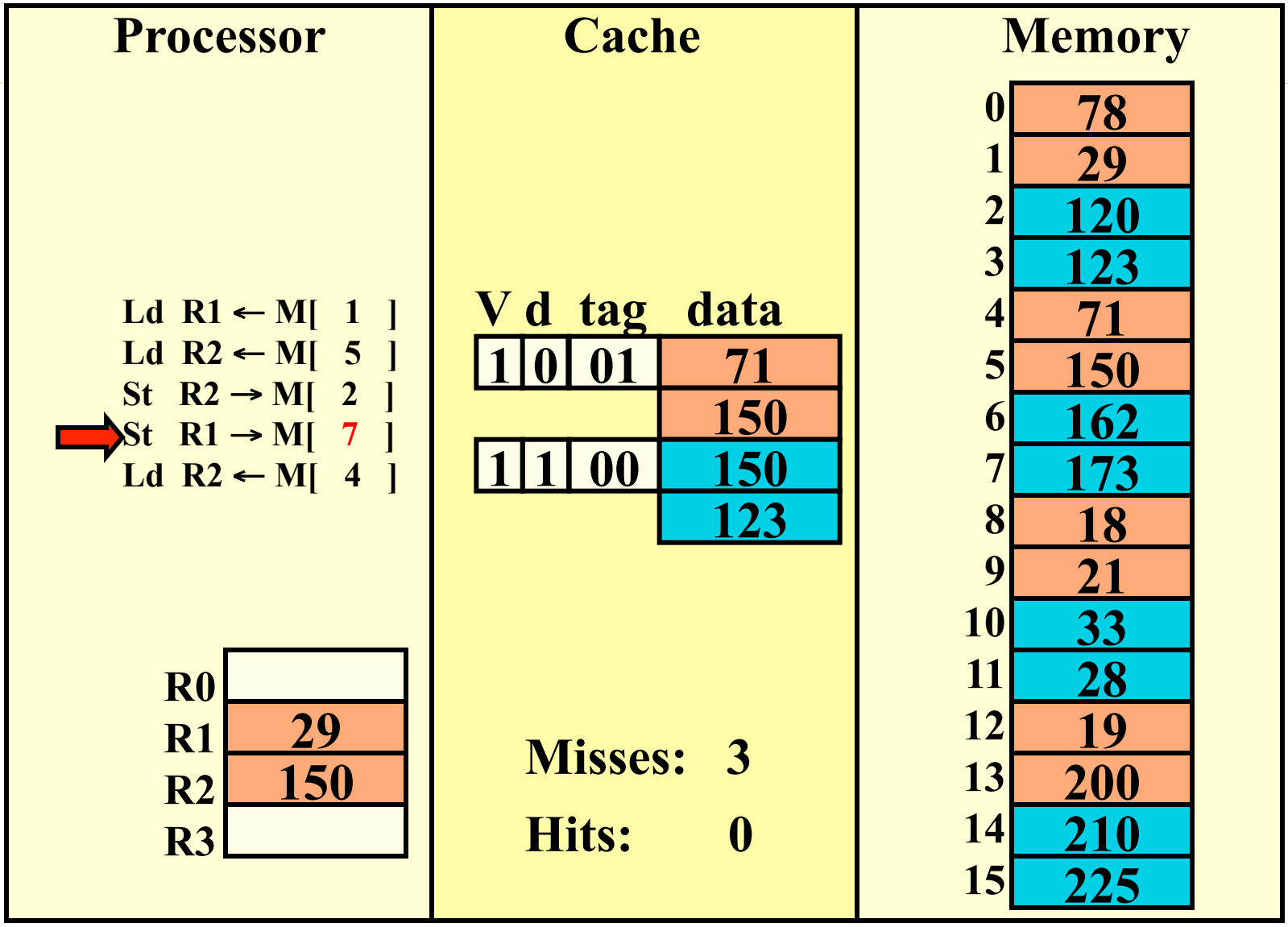
Direct-mapped (REF 3)



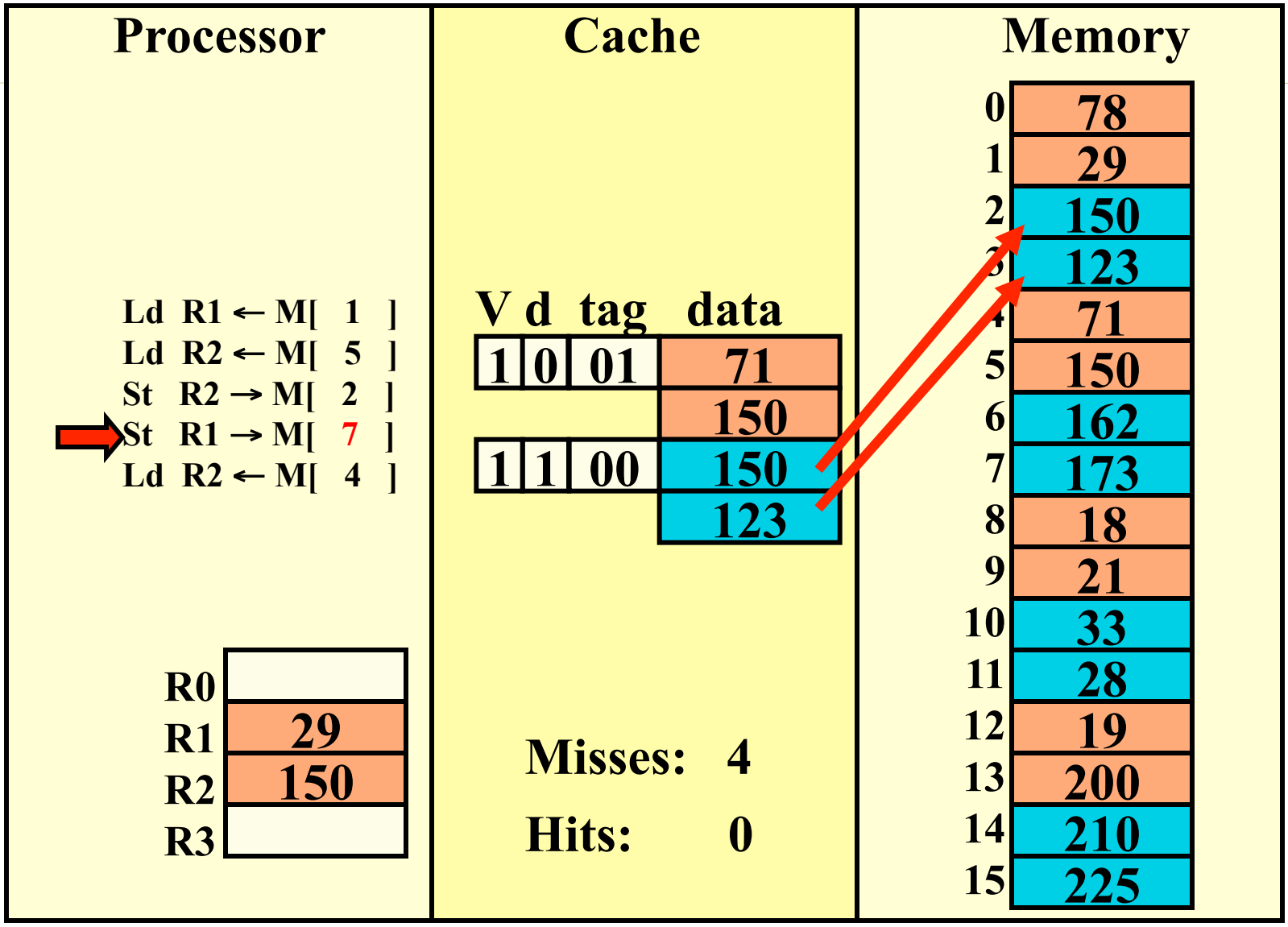
Direct-mapped (REF 3)



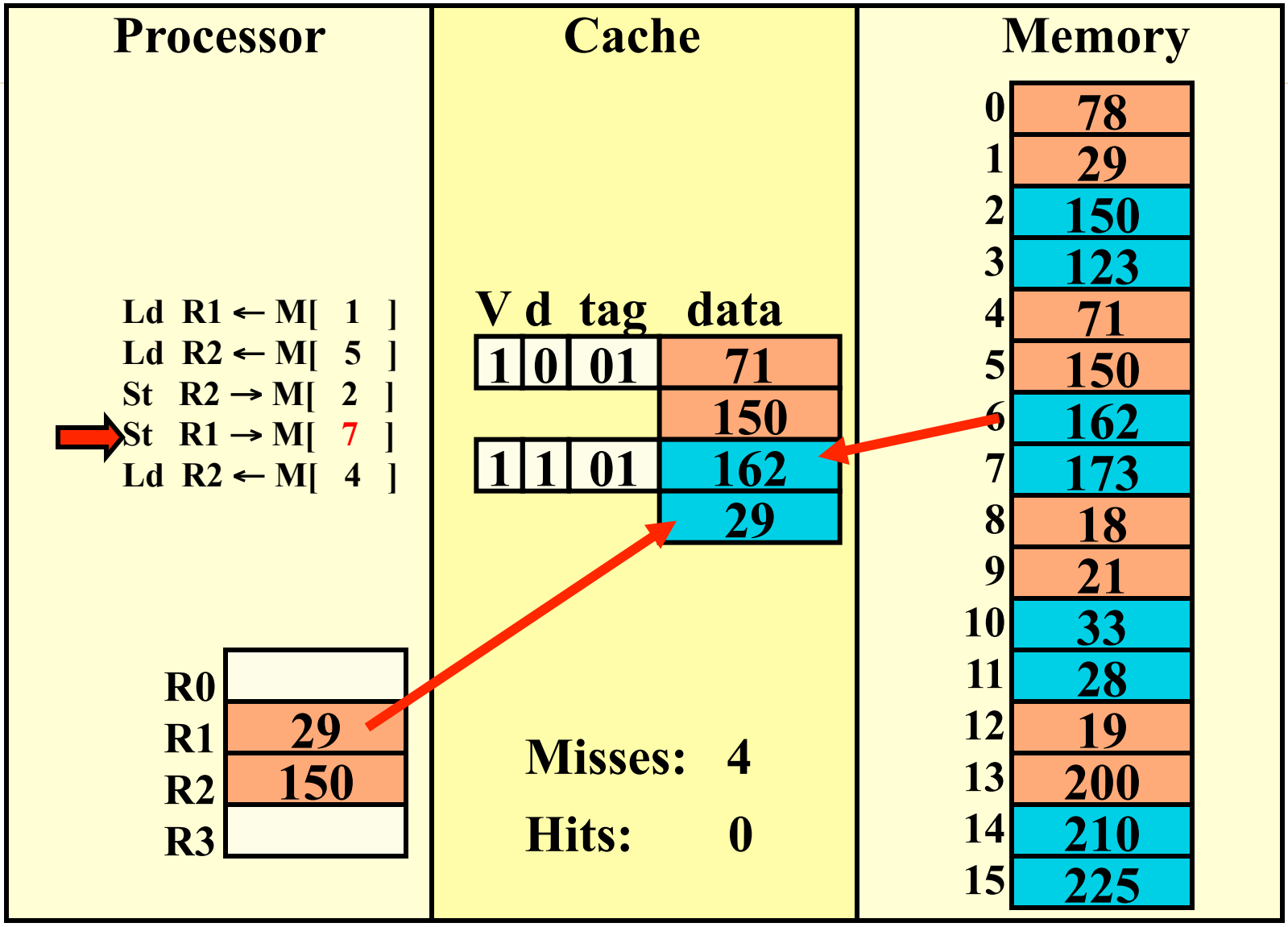
Direct-mapped (REF 4)



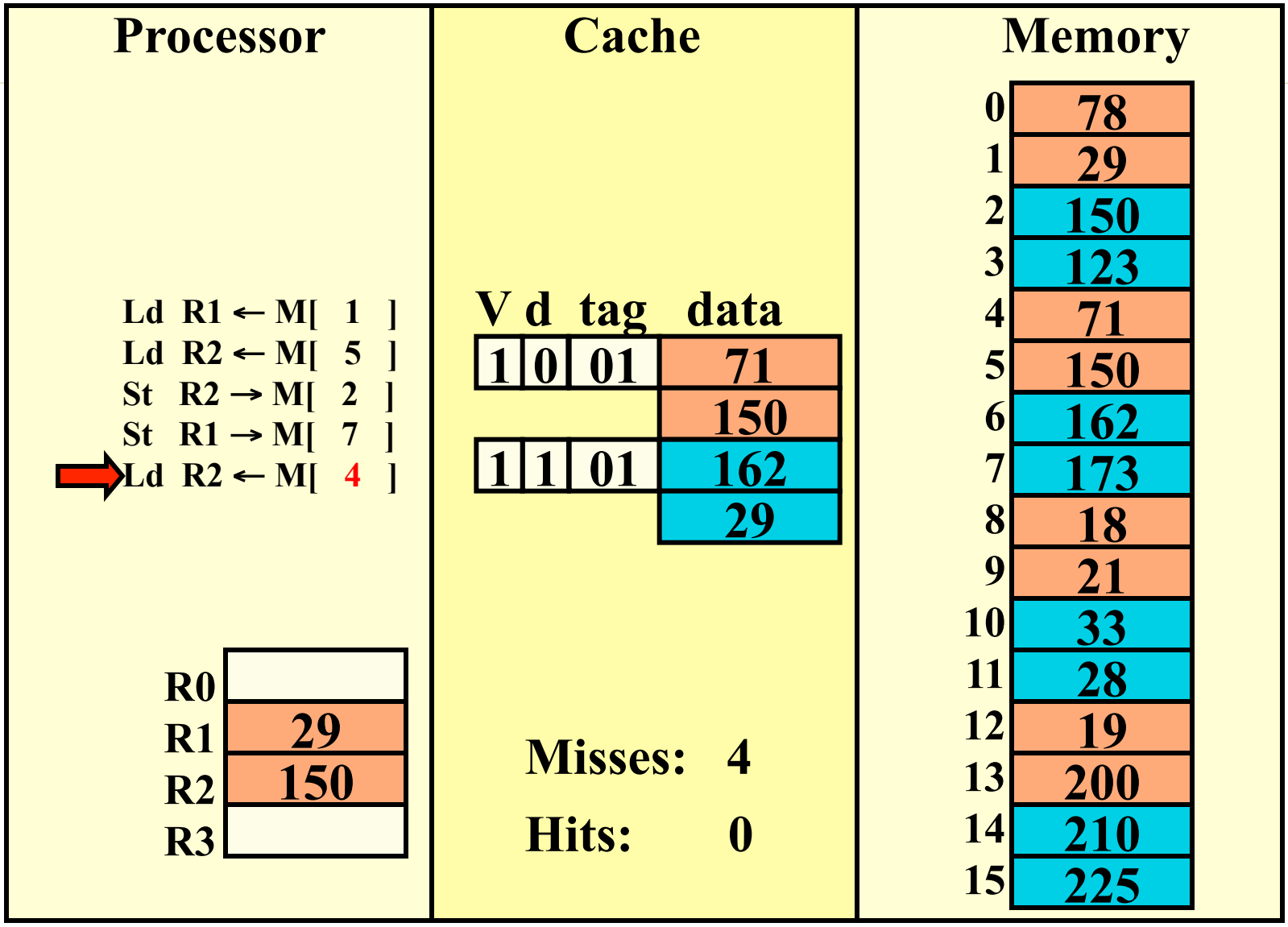
Direct-mapped (REF 4)



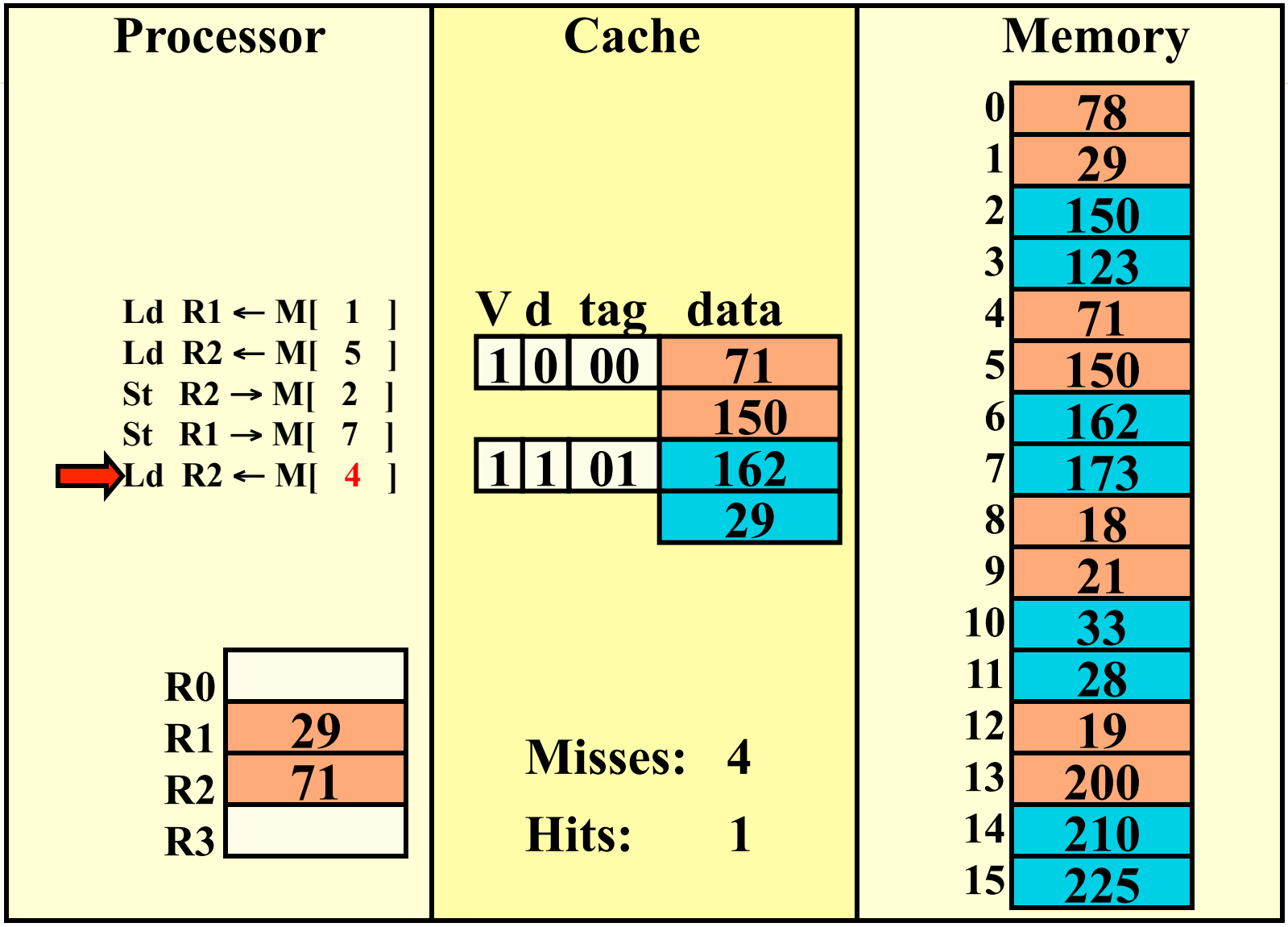
Direct-mapped (REF 4)



Direct-mapped (REF 5)



Direct-mapped (REF 5)

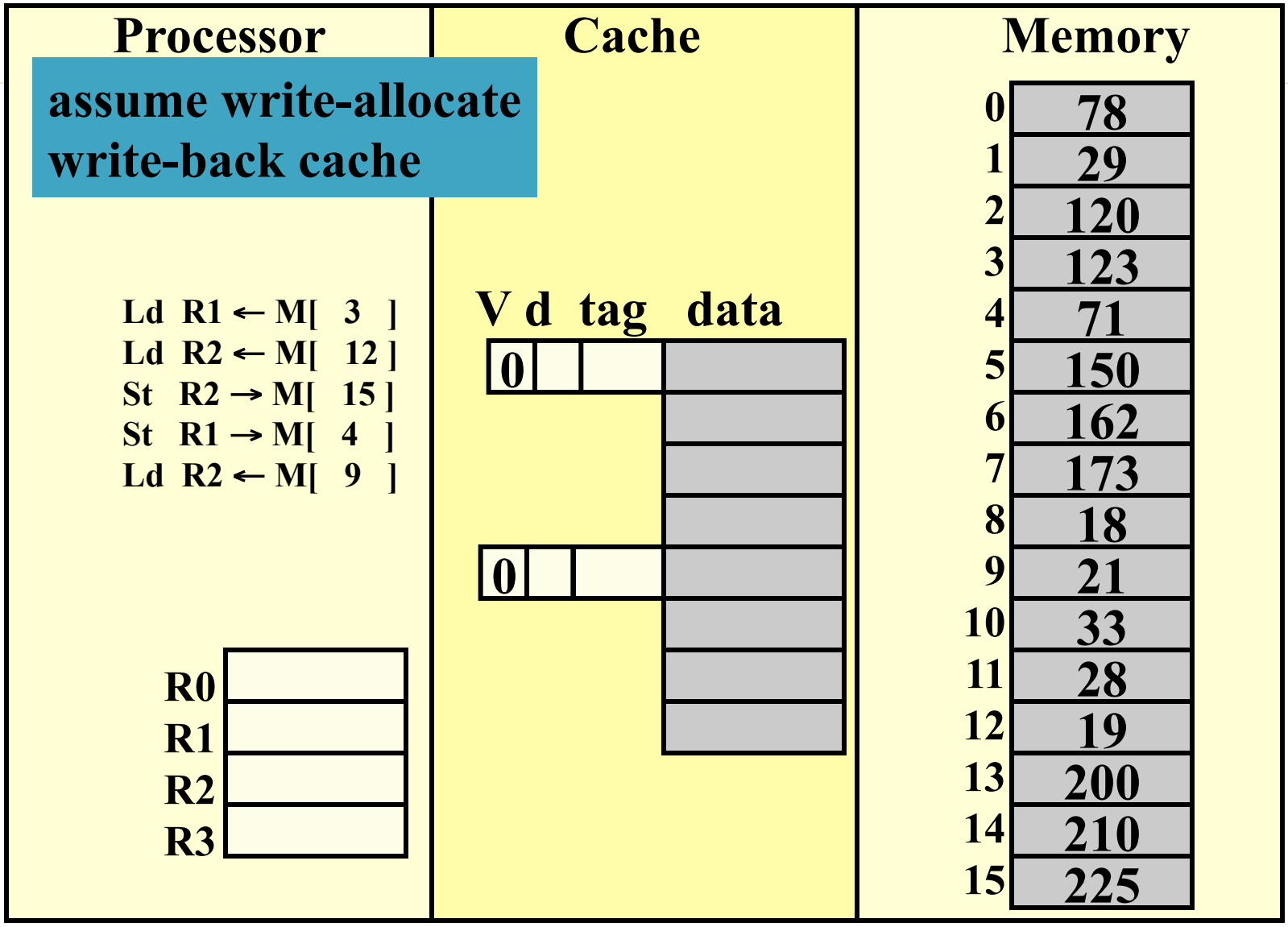


Class Problem 4

Show the breakdown of the address for the following cache configuration:

32 bit address
16K cache
Direct-mapped cache
32-byte blocks

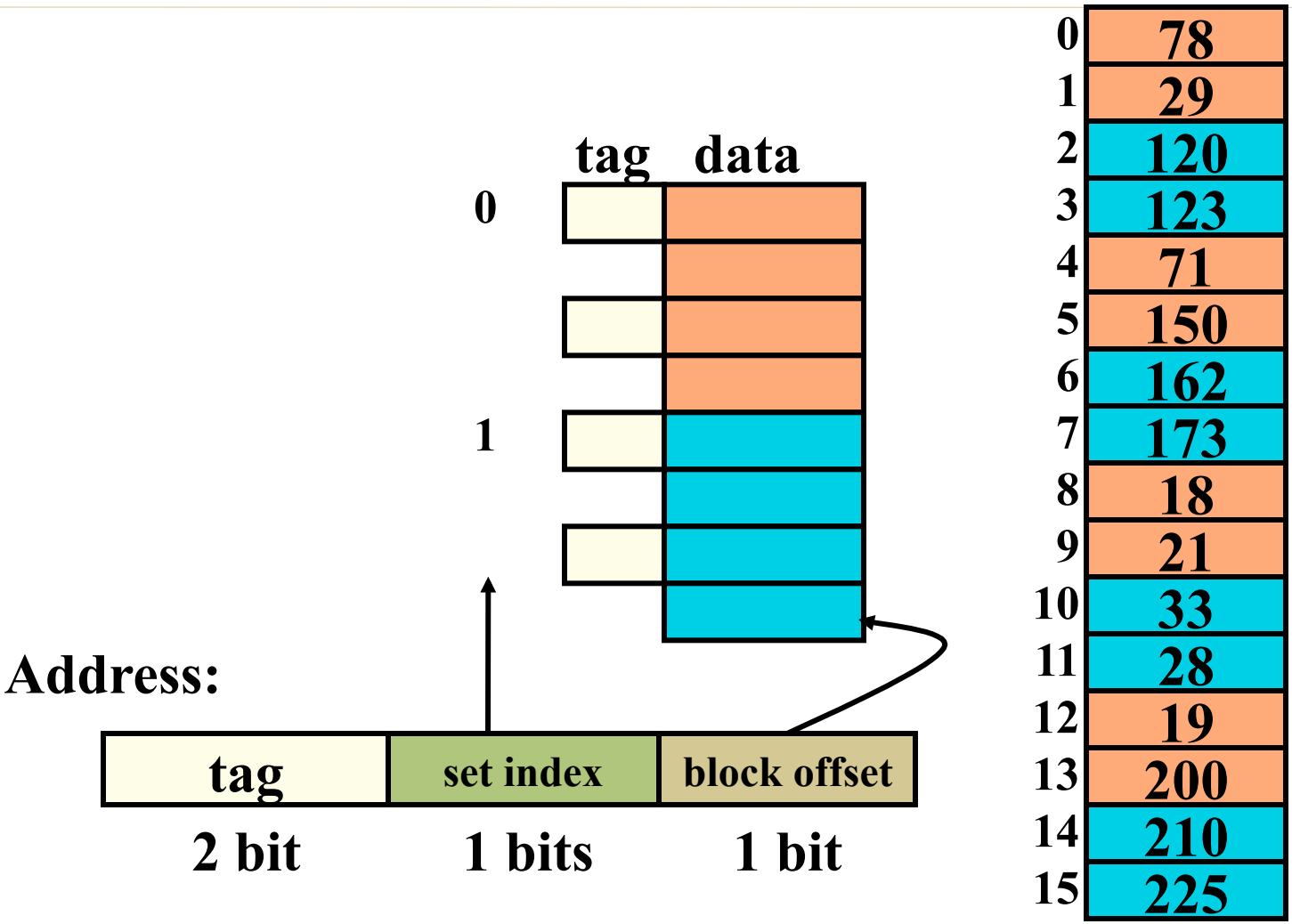
Class Problem 5



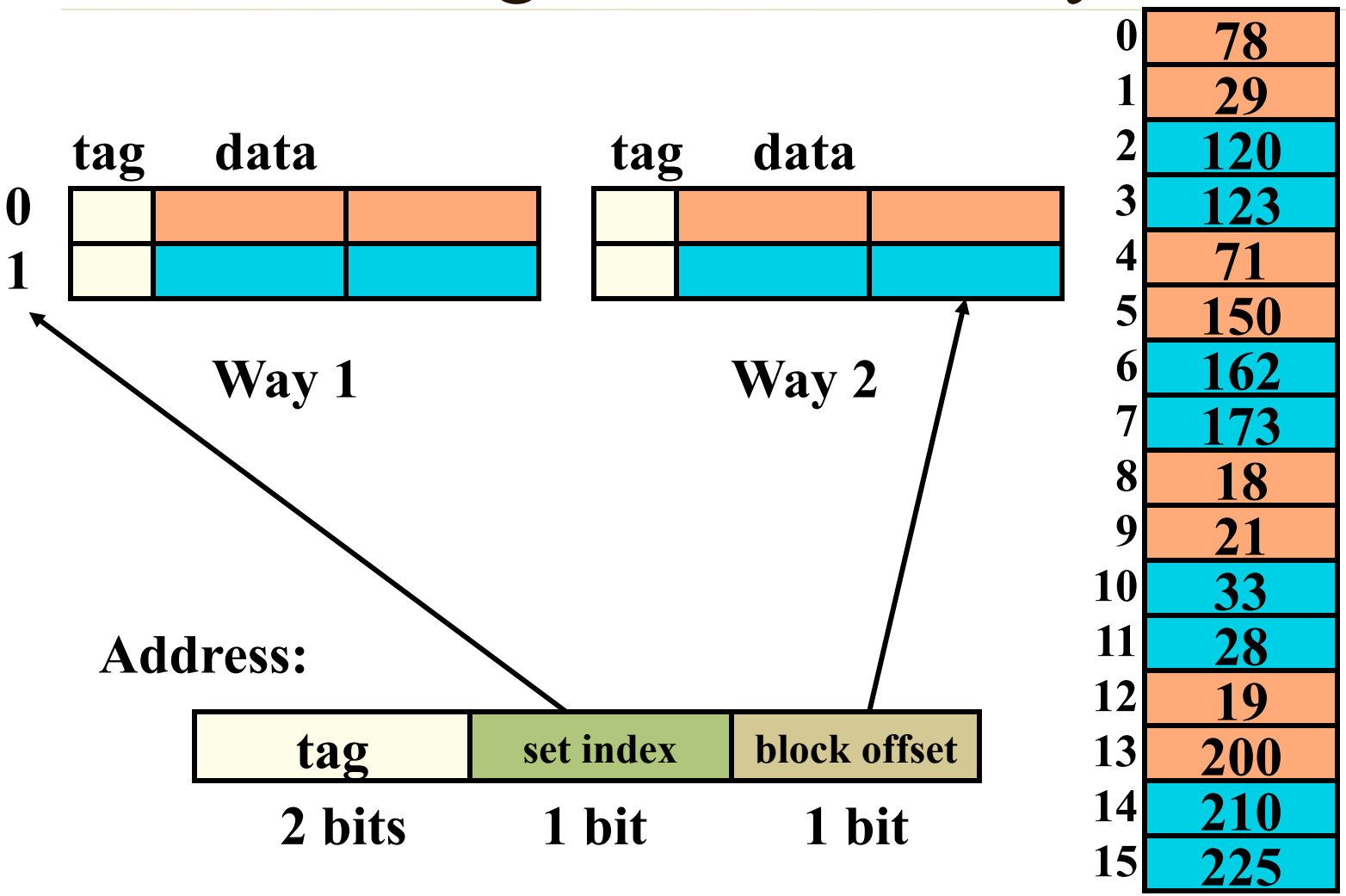
Split the Difference

- 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.

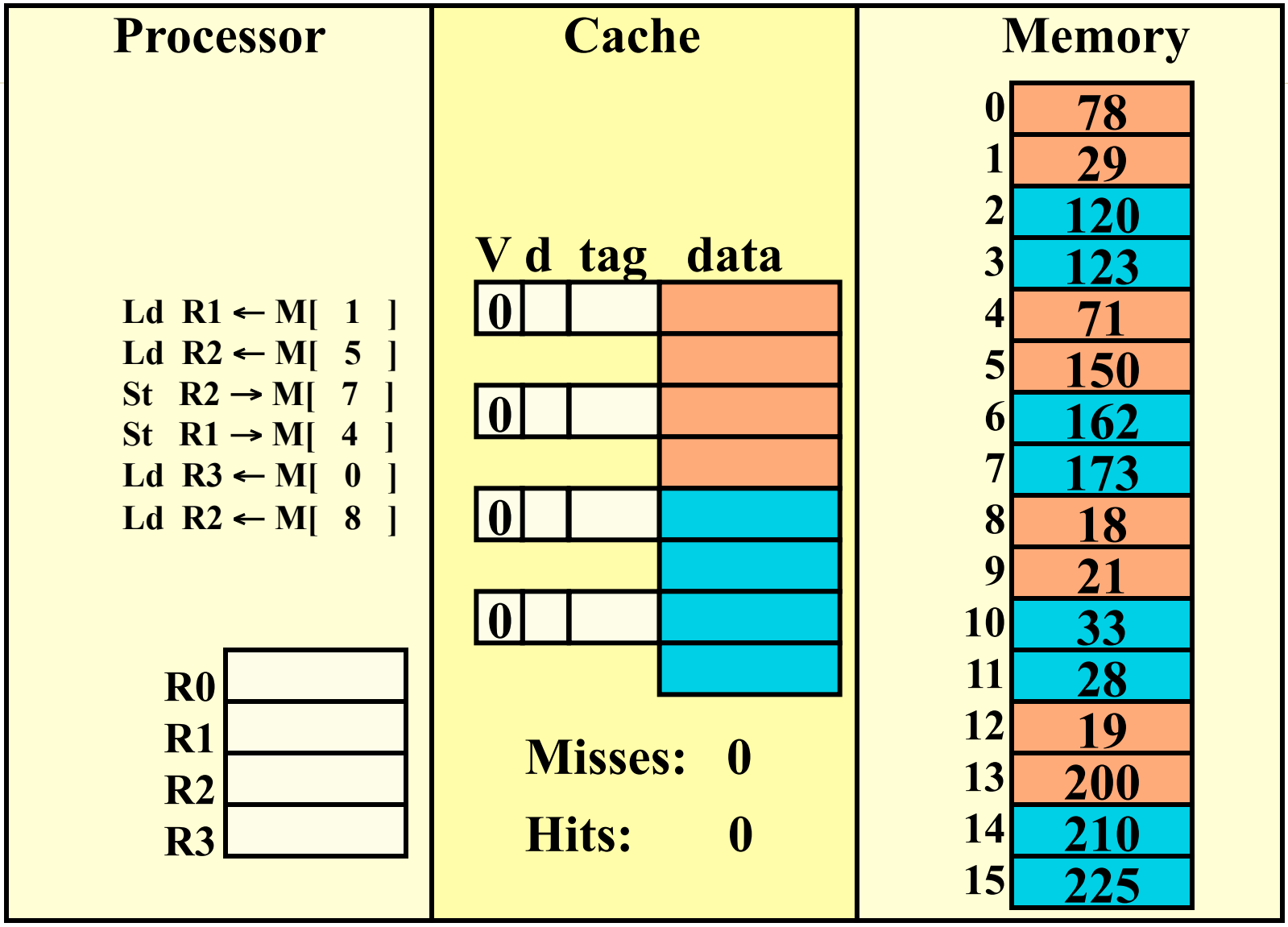
Set Associative Cache



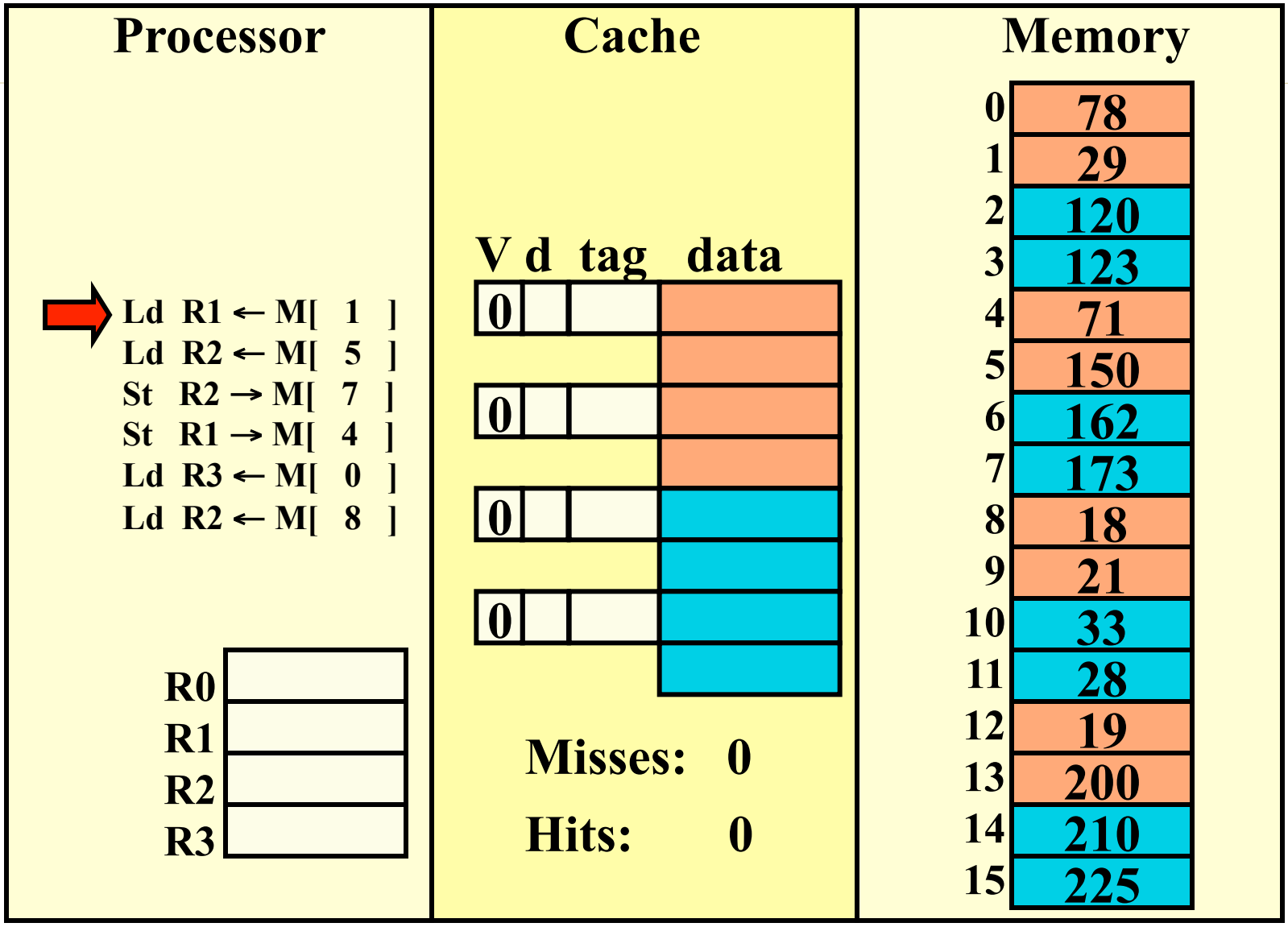
Set Associative Cache using the Book's Style



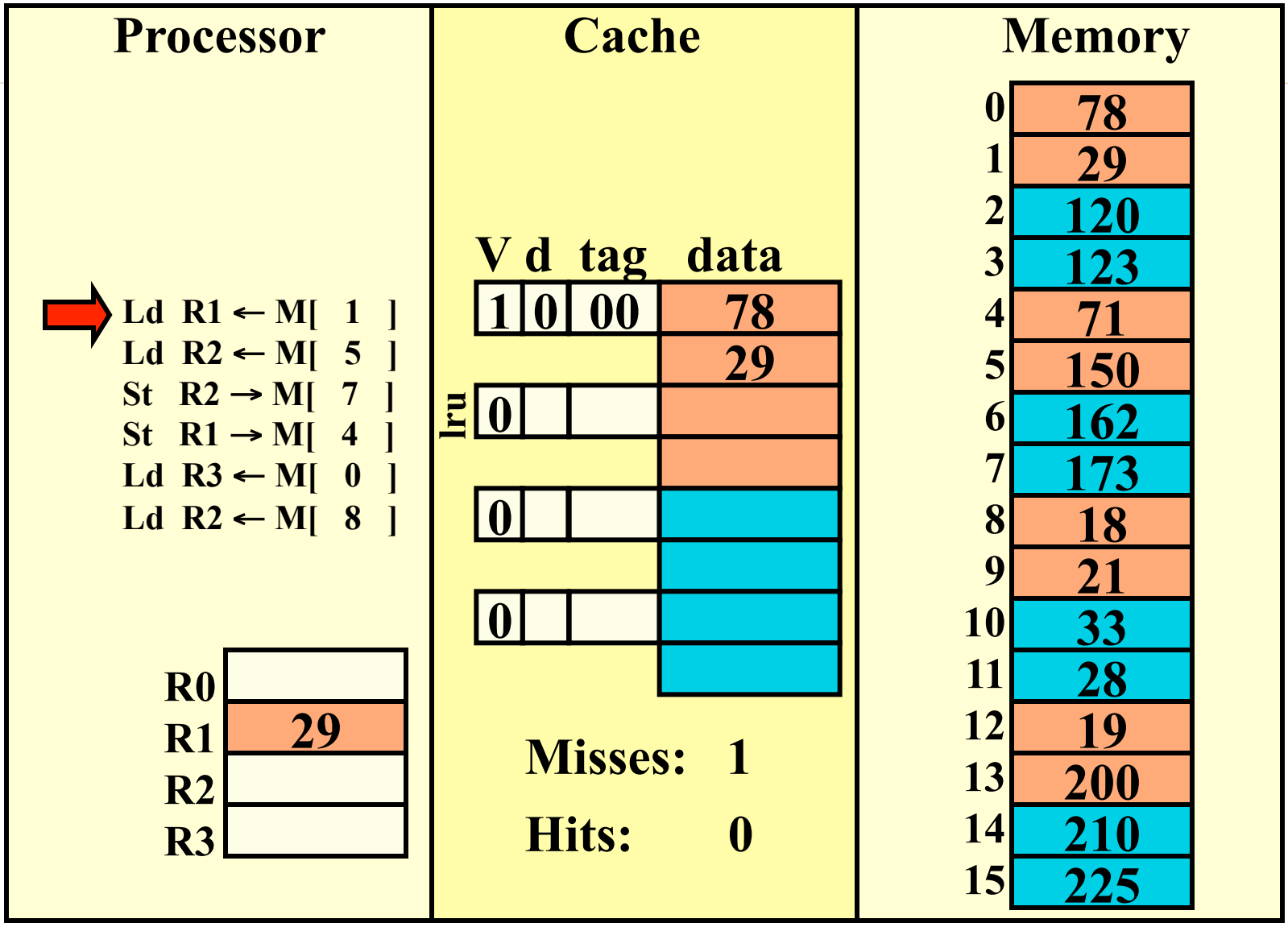
Set-associative Cache Example



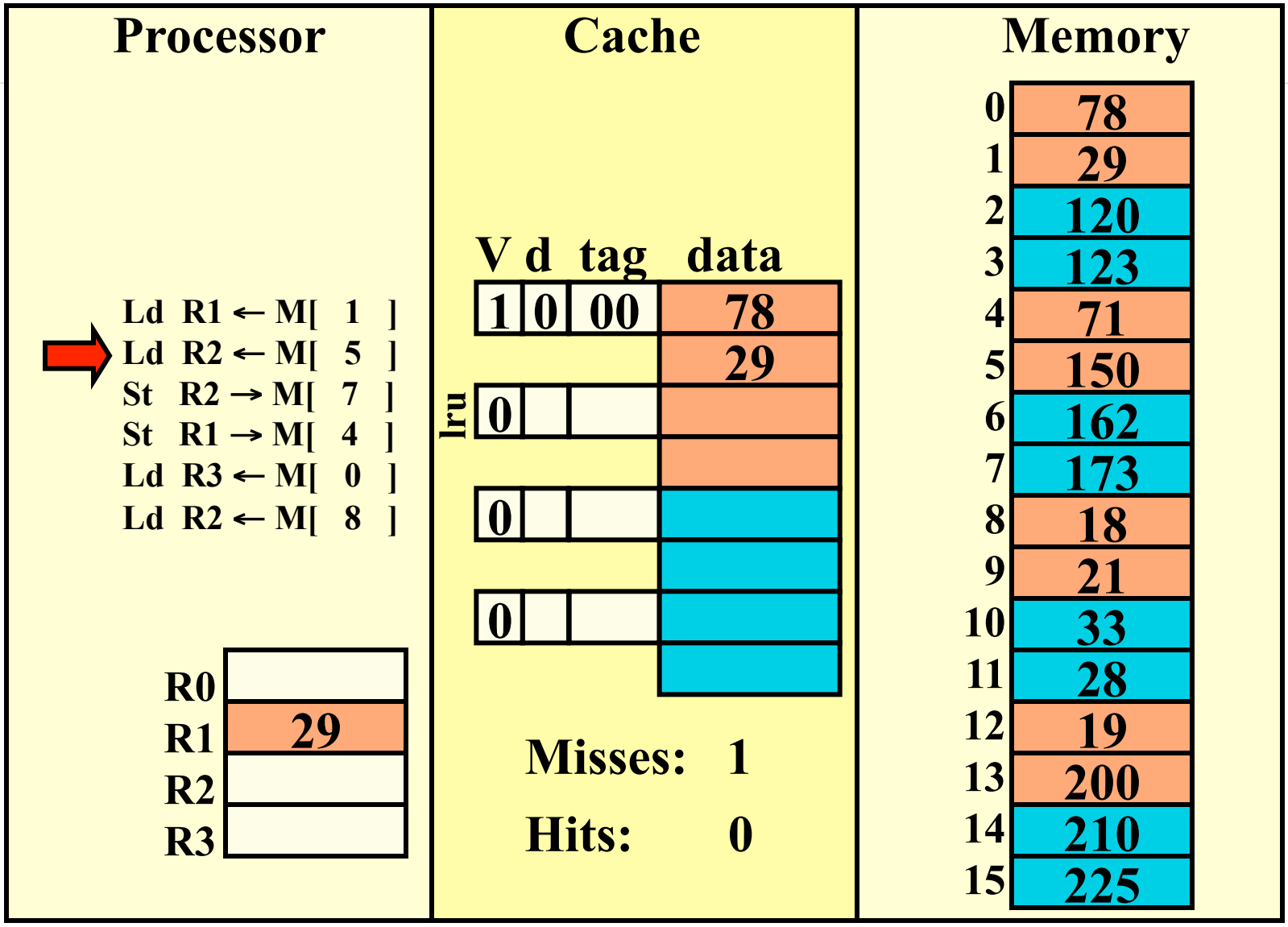
Set-associative cache (REF 1)



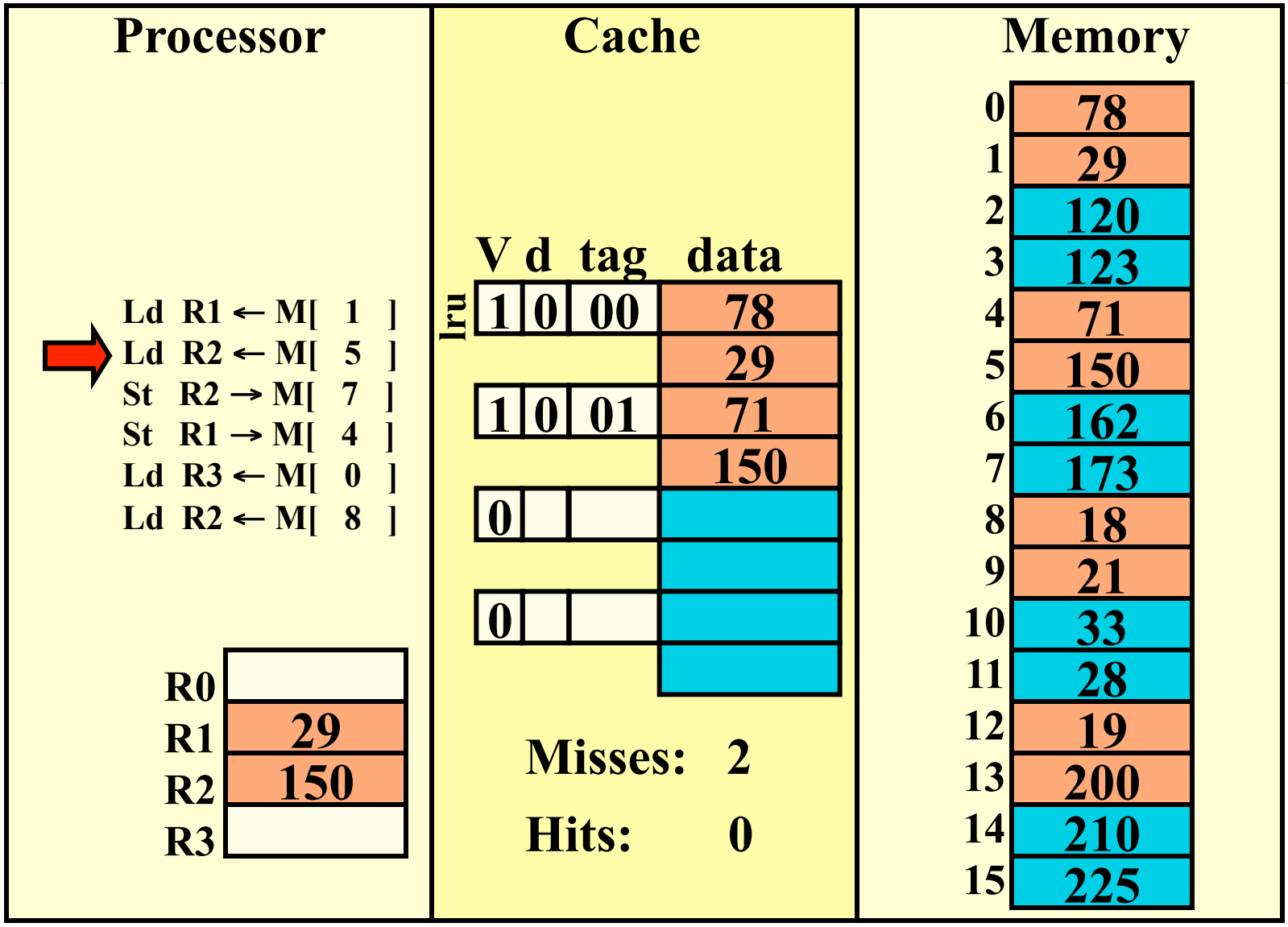
Set-associative cache (REF 1)



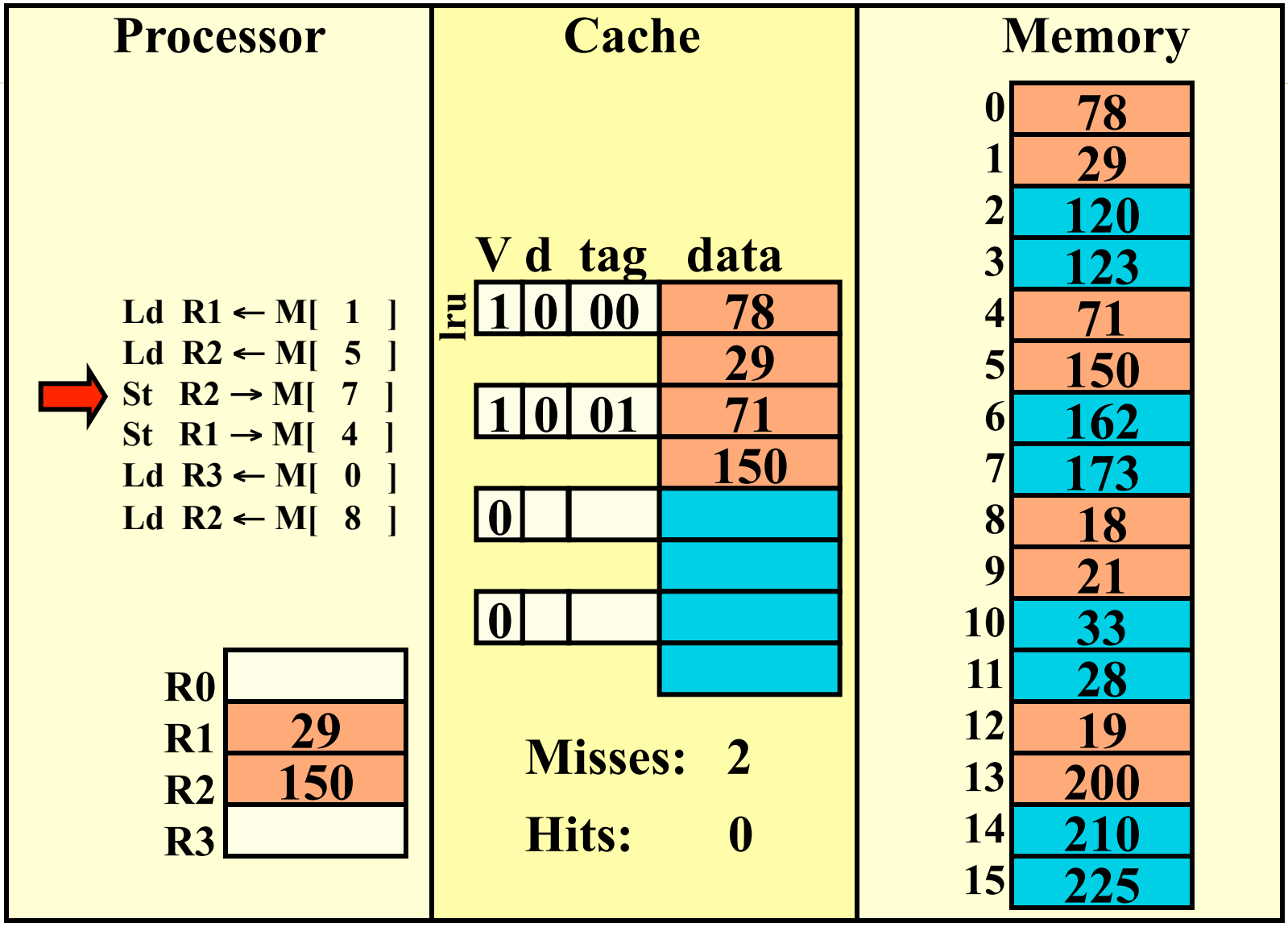
Set-associative cache (REF 2)



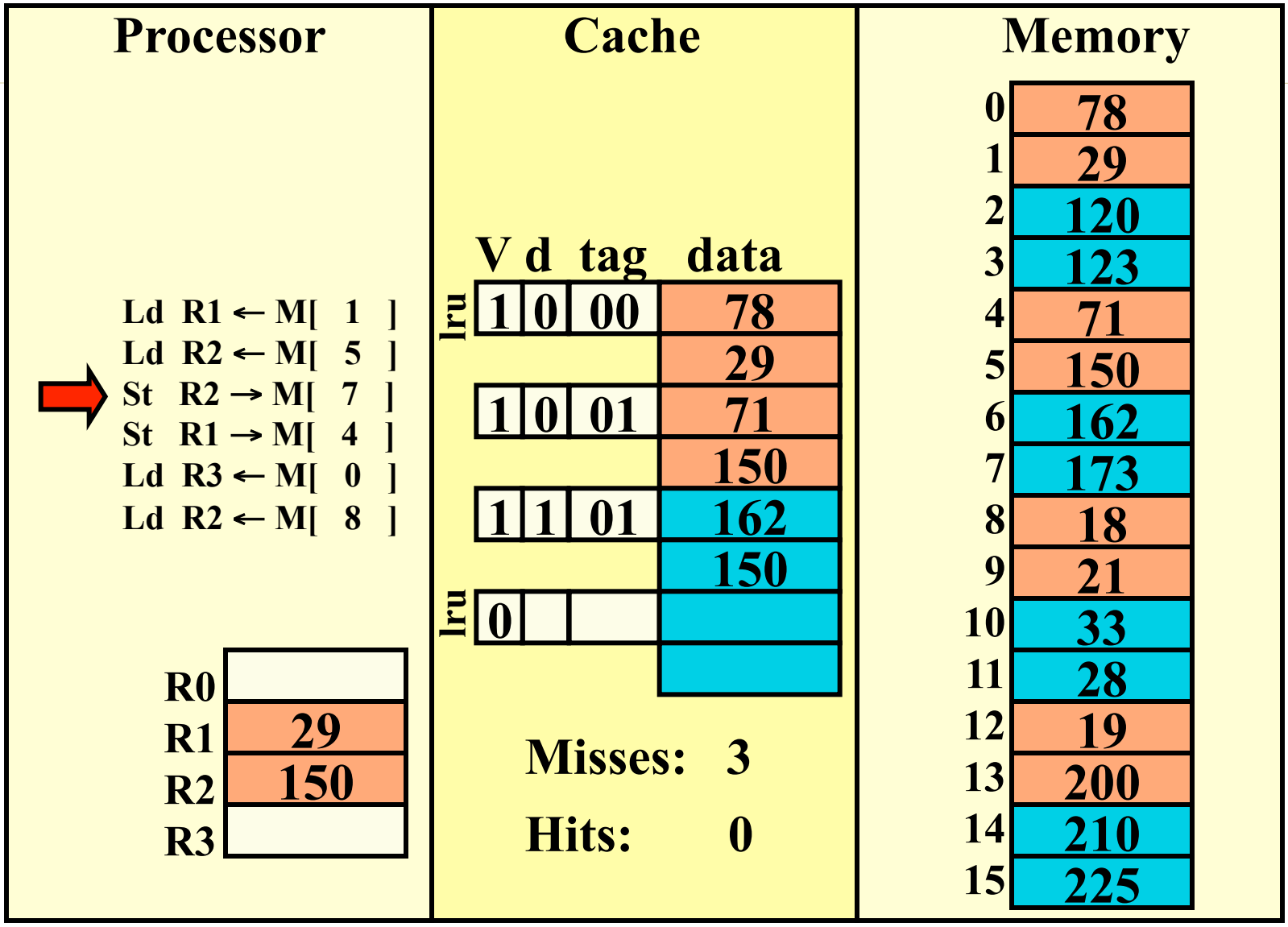
Set-associative cache (REF 2)



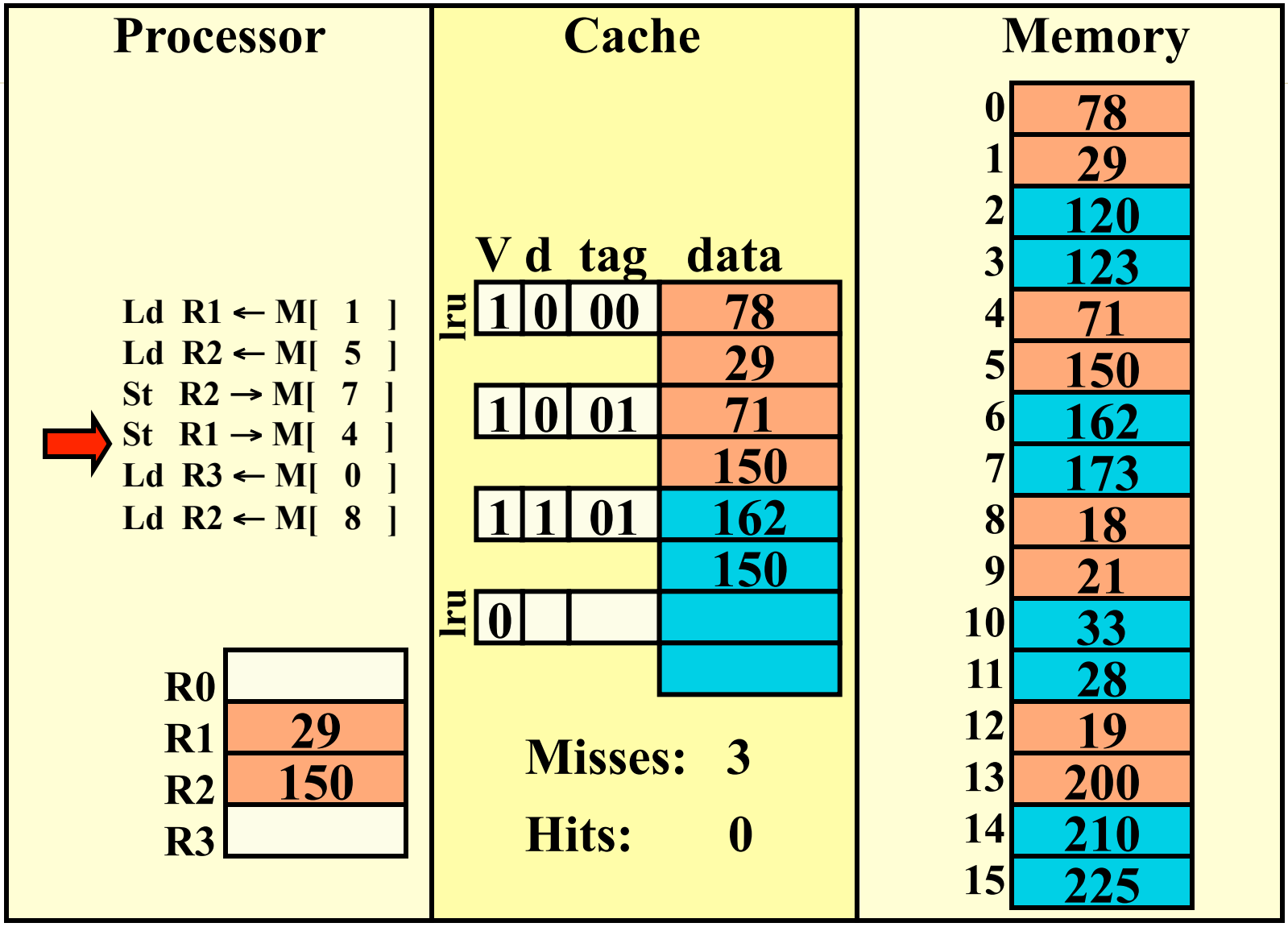
Set-associative cache (REF 3)



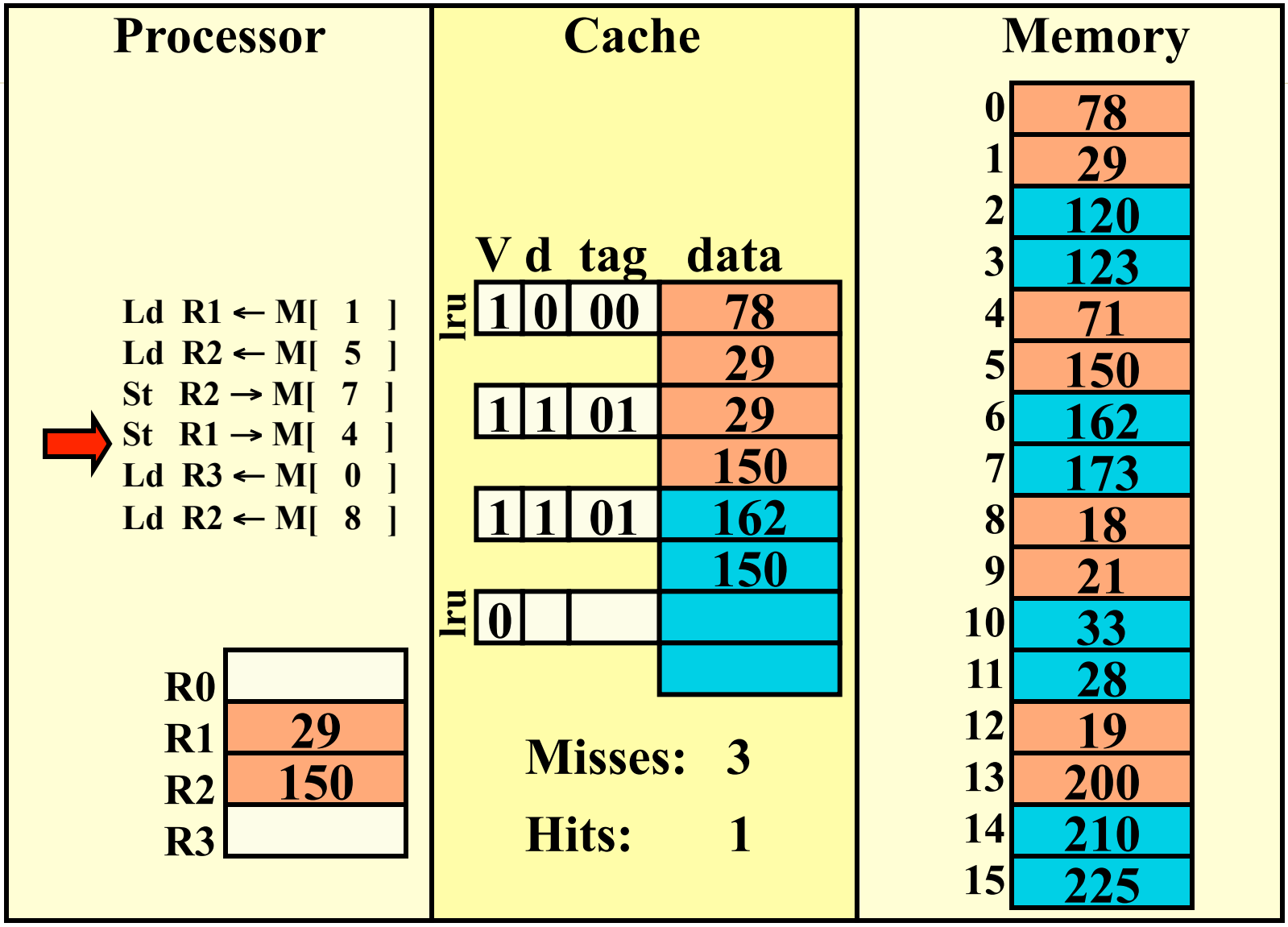
Set-associative cache (REF 3)



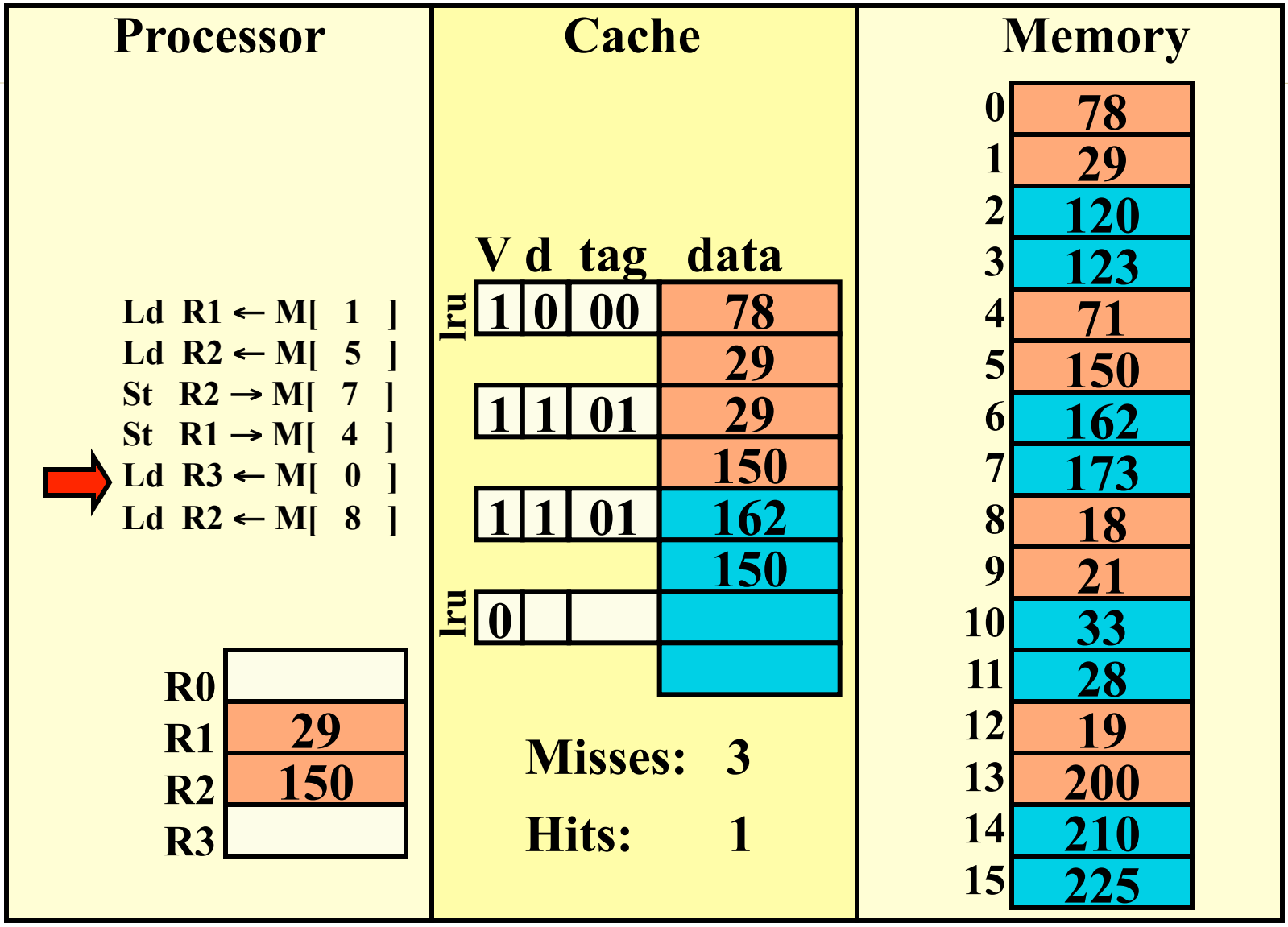
Set-associative cache (REF 4)



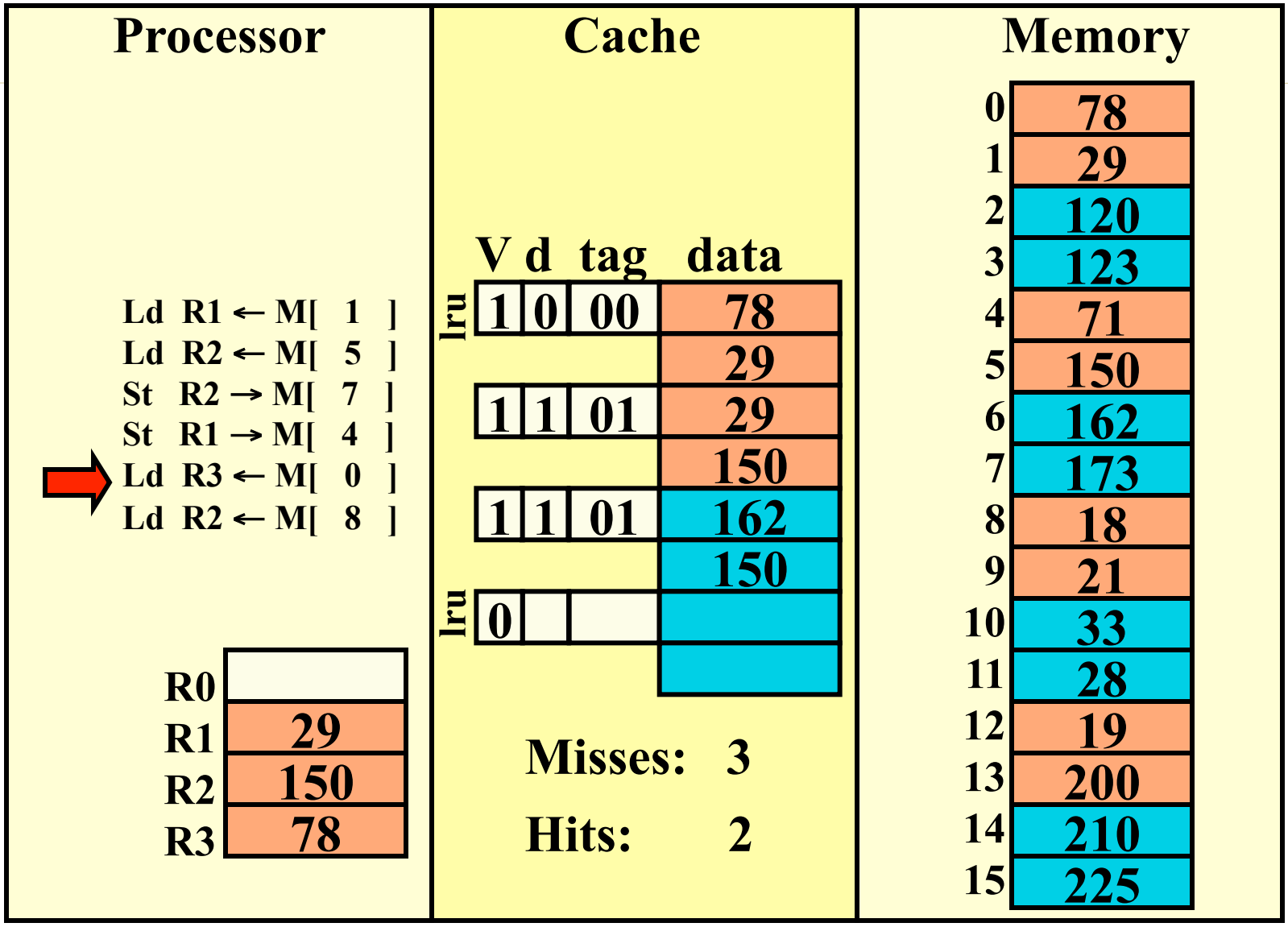
Set-associative cache (REF 4)



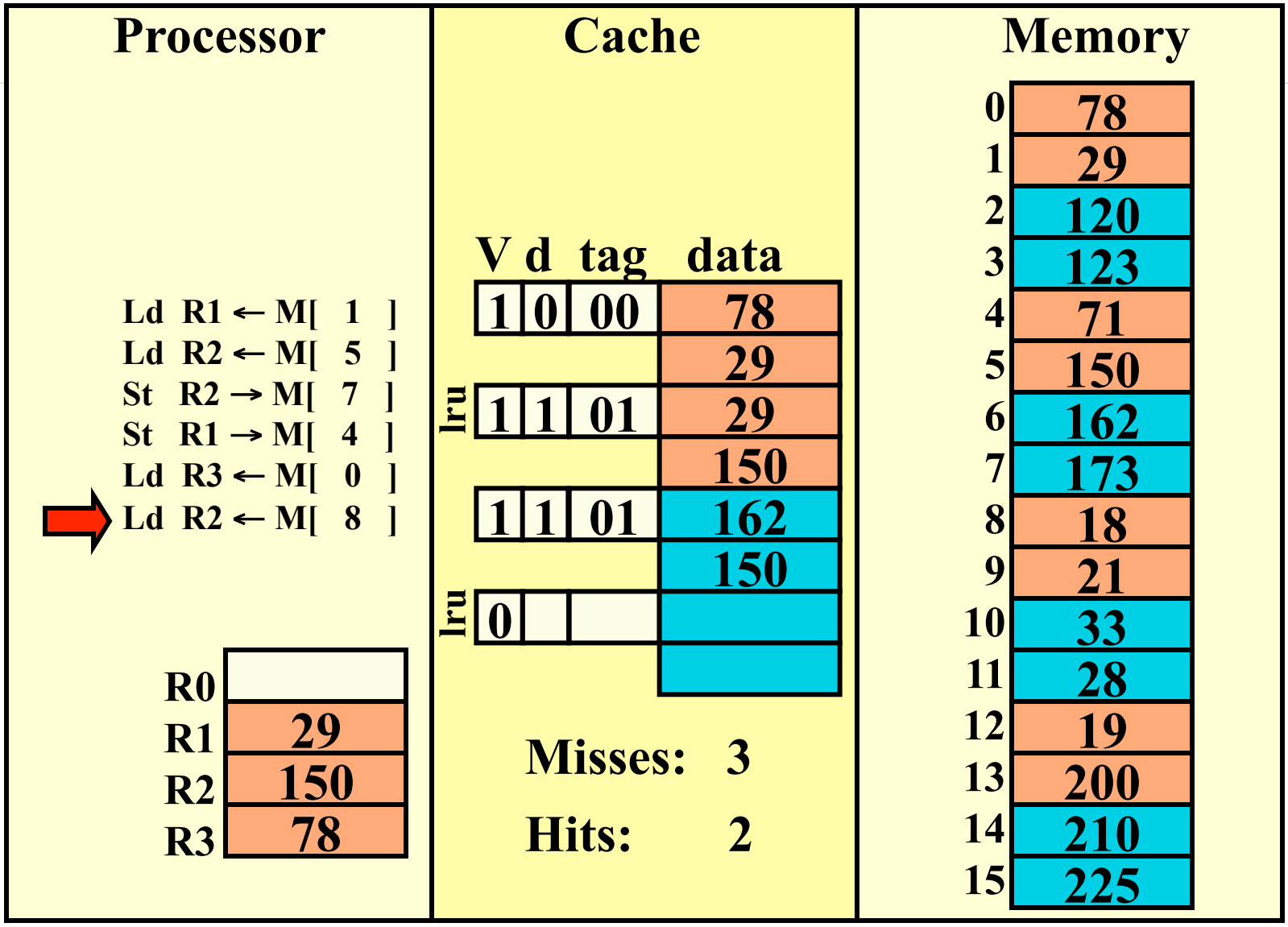
Set-associative cache (REF 5)



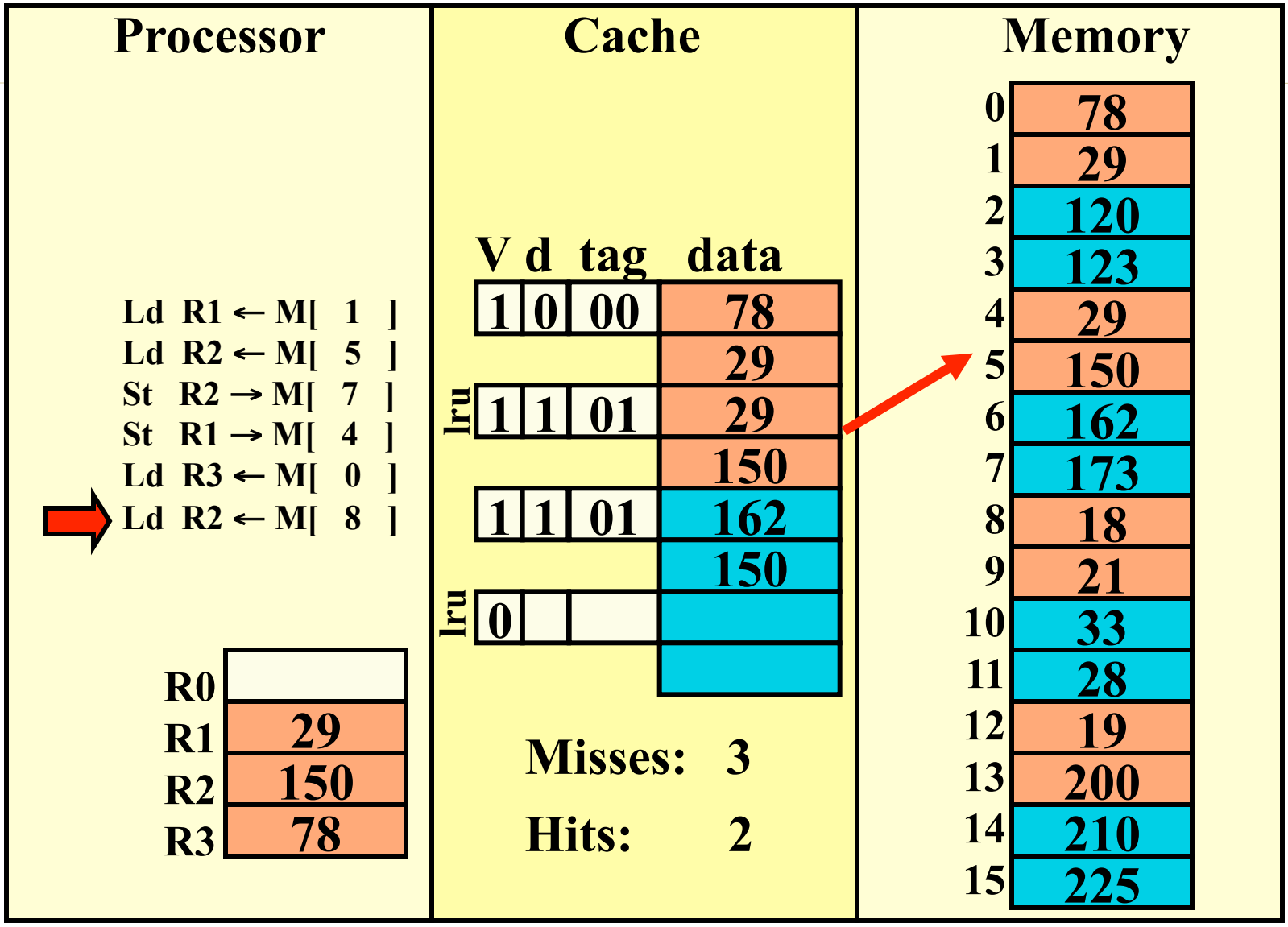
Set-associative cache (REF 5)



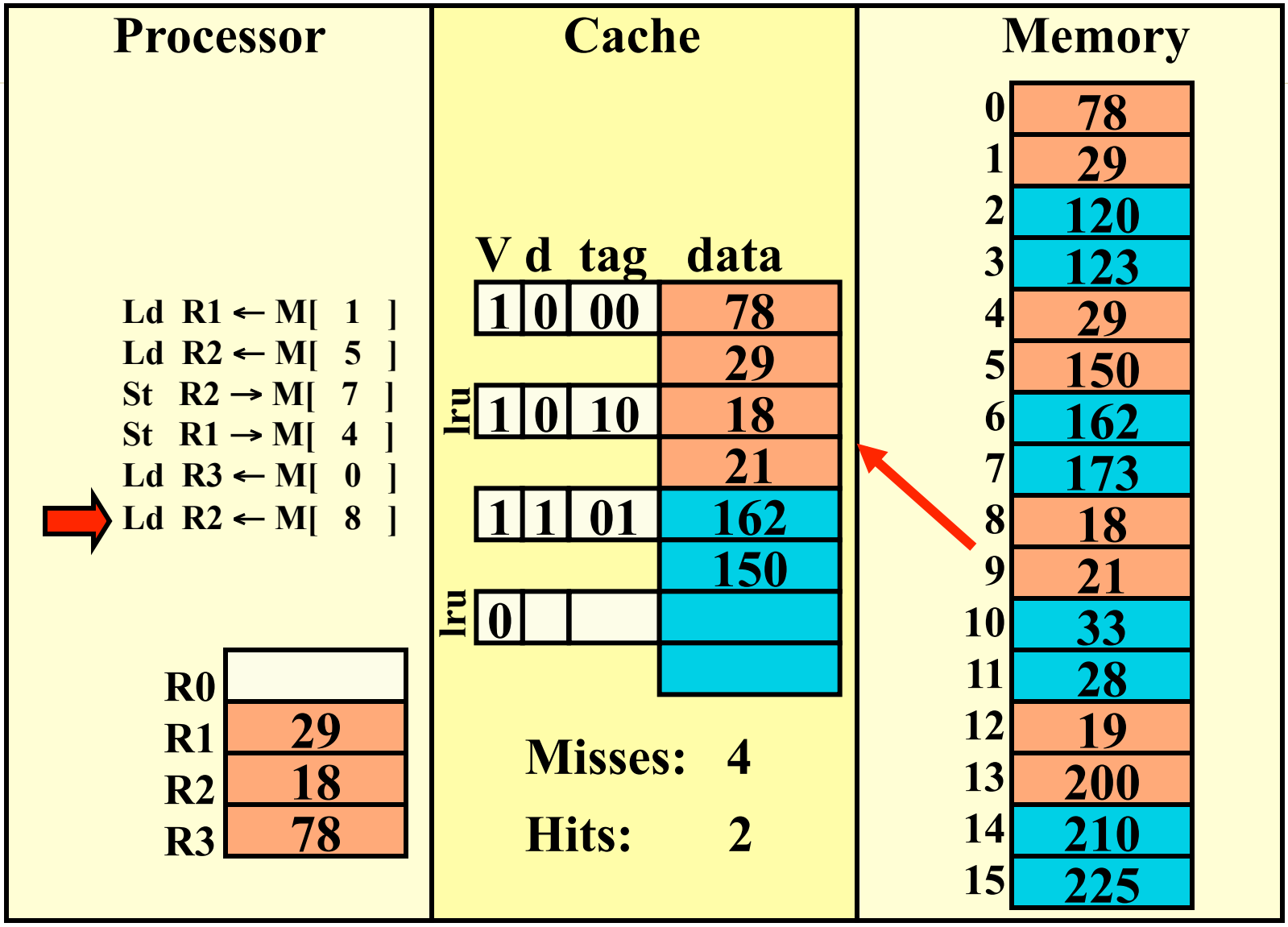
Set-associative cache (REF 6)



Set-associative cache (REF 6)



Set-associative cache (REF 6)



Class Problem 6

Show the breakdown of the address for the following cache configuration:

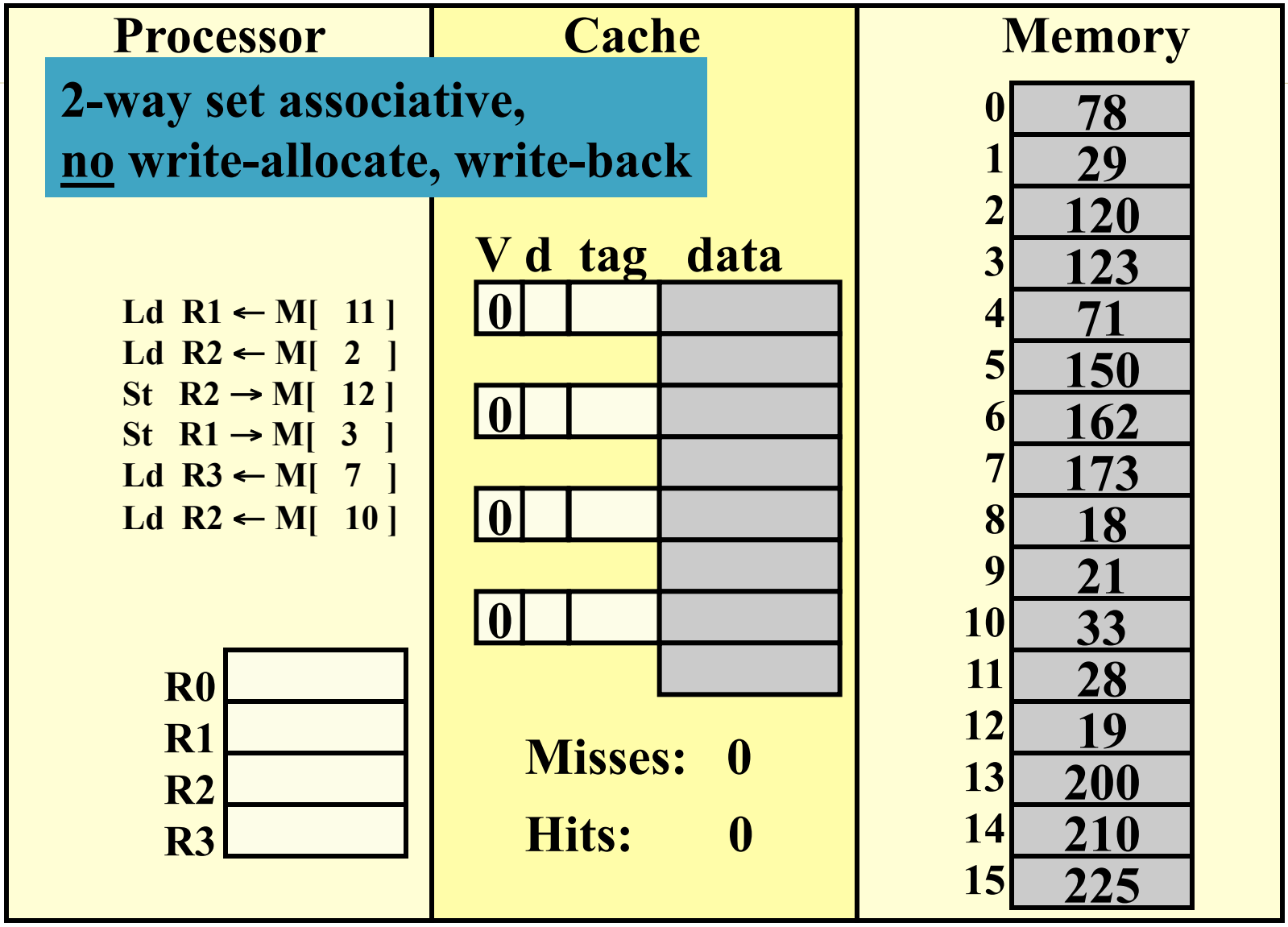
32 bit address
16K cache
4-way set associative cache
64-byte blocks

Class Problem 7

Show the breakdown of the address for the following cache configuration:

32 bit address
16K cache
fully associative cache
64-byte blocks

Class Problem 8



Reasons for Cache Misses
Aka the 3 C's of Cache Misses

- Cache is too small to hold all the data
  - **Capacity** miss
    - Reduce misses by building a bigger cache
- First reference to an address
  - **Compulsory** miss
    - Reduce by increasing block size
- Replaced it from a busy set
  - **Conflict** miss
    - Reduce by increasing associativity
    - Would have had hit with a fully associative cache

What about 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?

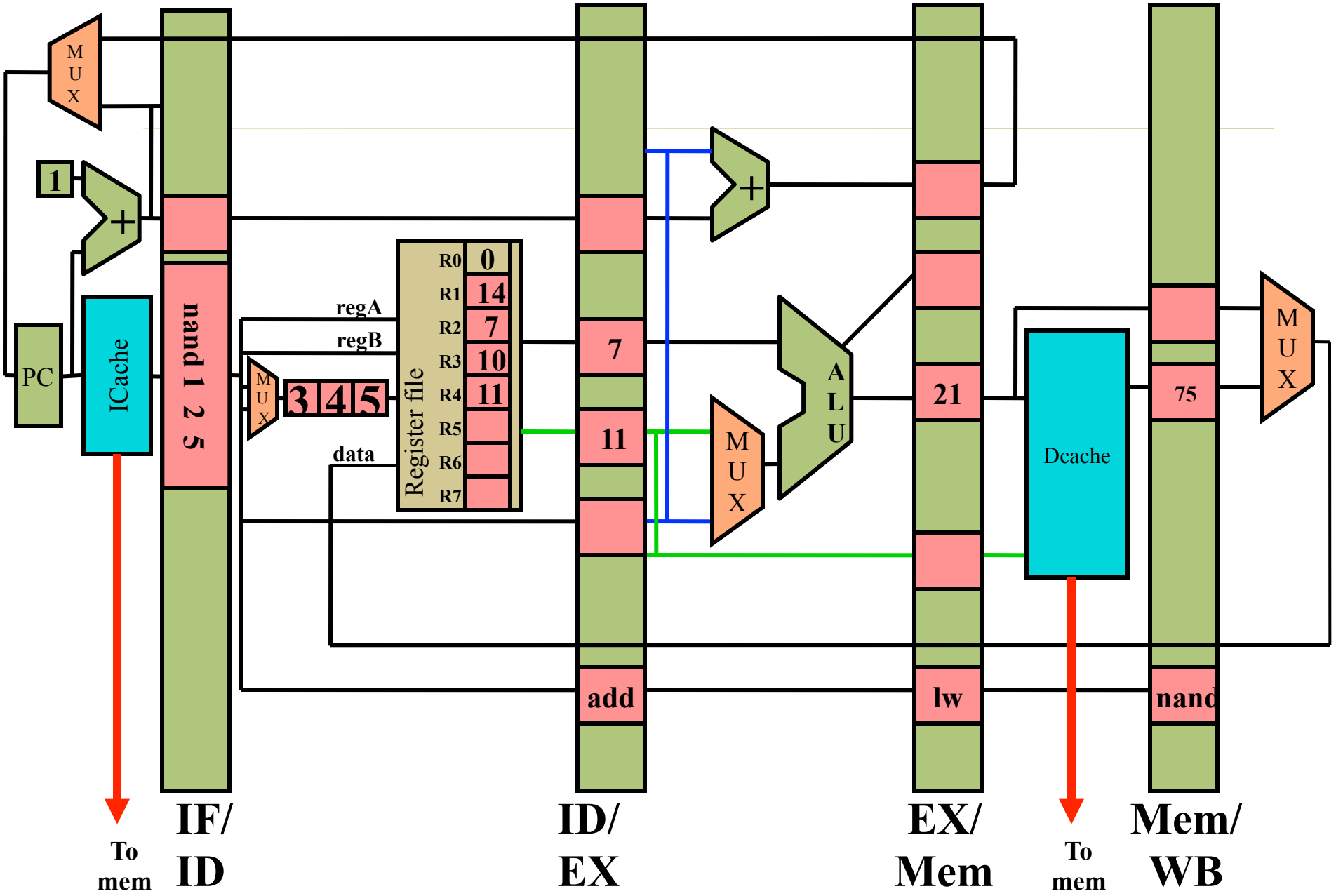
     What are advantages of a separate ICache?

     Can anything go wrong with this?

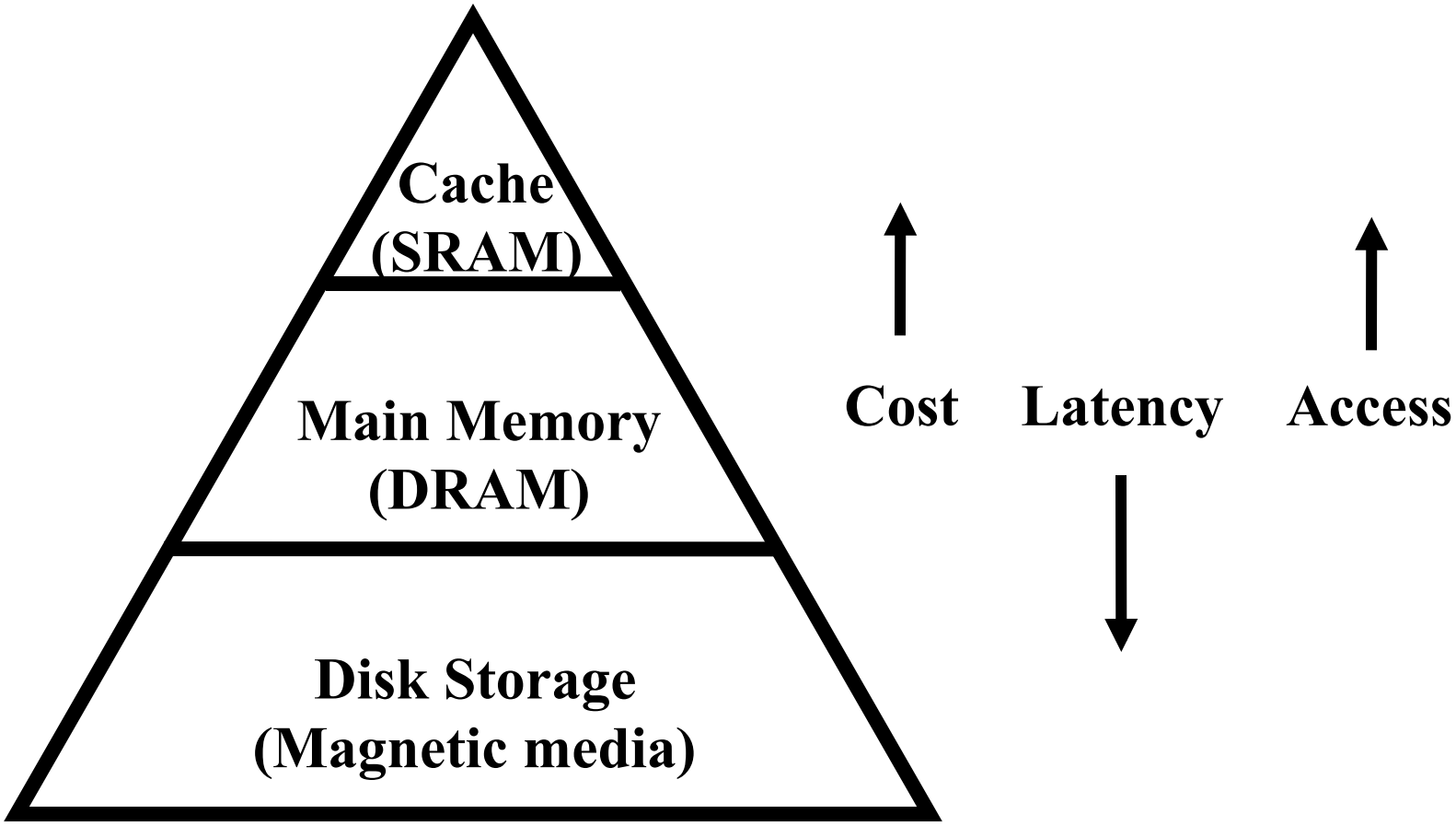
Integrating Caches into a Pipeline

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

LCS4 Pipeline with Caches



Memory Hierarchy



The problem(s)

- DRAM is too expensive to buy many gigabytes
  - Yet we want our programs to work even if they require more storage than we bought.
  - We also don't want a program that works on a machine with 512 megabytes to stop working if we try to run it on a machine with only 256 megabytes of memory.
- We run more than one program on the machine.

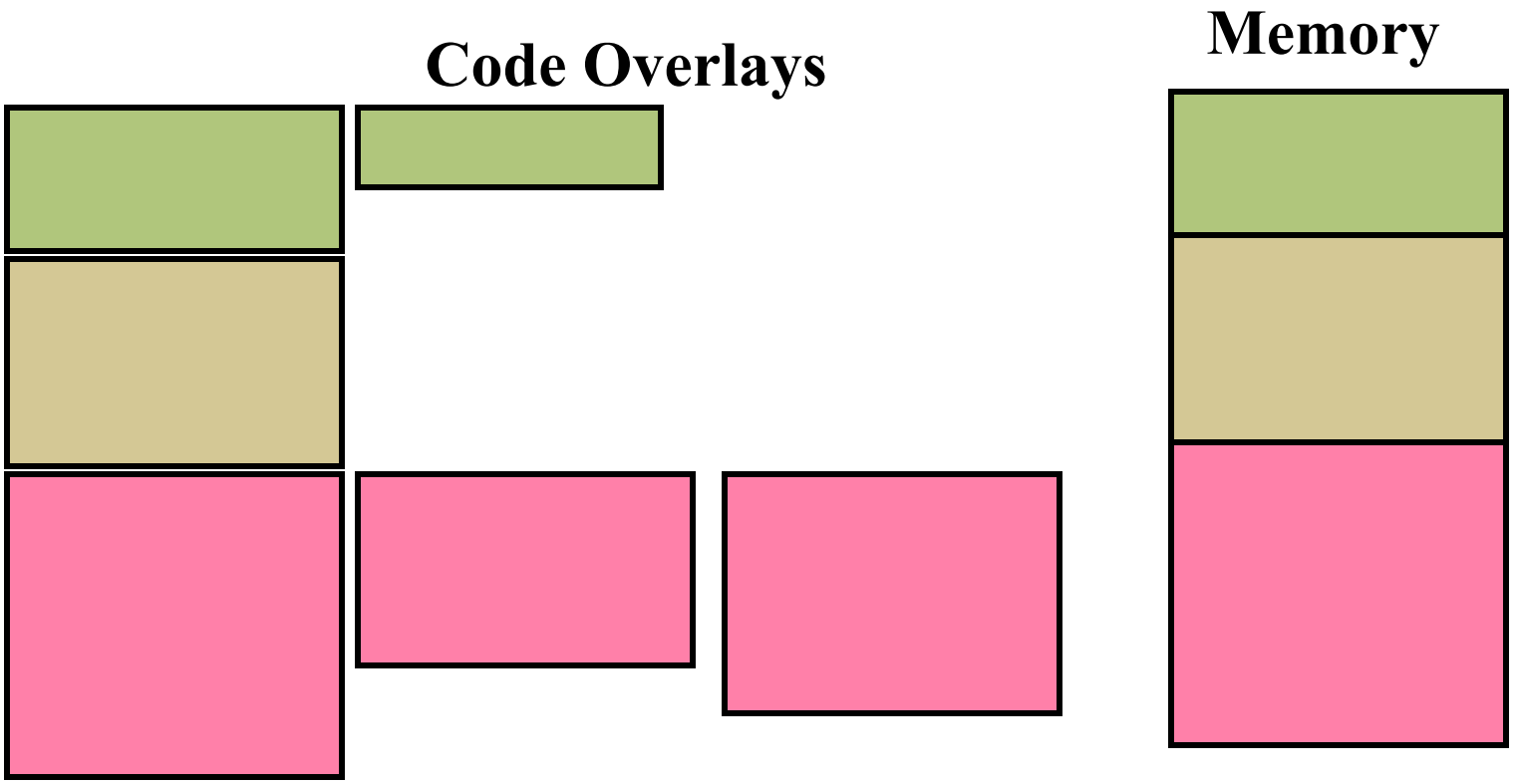
Solution 1: User control

- Leave the problem to the programmer
  - Assume the programmer knows the exact configuration of the machine.
    - Programmer must either make sure the program fits in memory, or break the program up into pieces that do fit and load each other off the disk when necessary
- Not a bad solution in some domains
  - Playstation 2, cell phones, etc.

Solution 2: Overlays

- A little automation to help the programmer
  - build the application in overlays
    - Two pieces of code/data may be overlayed iff
      - They are not active at the same time
      - They are placed in the same memory region
- Managing overlays is performed by the compiler
  - Good compilers may determine overlay regions
  - Compiler adds code to read the required overlay memory off the disk when necessary

Overlay example

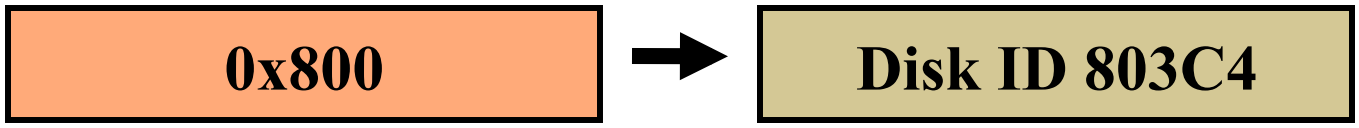


Solution 3: Virtual memory

- Build new hardware and software that automatically translates each memory reference from a virtual address (that the programmer sees as an array of bytes) to a physical address (that the hardware uses to either index DRAM or identify where the storage resides on disk)

Basics of Virtual Memory

- Any time you see the word virtual in computer science and architecture it means "using a level of indirection"
- Virtual memory hardware changes the virtual address the programmer sees into the physical ones the memory chips see.



Virtual address **Physical address**

Virtual Memory View

- Virtual memory lets the programmer “see” a memory array larger than the DRAM available on a particular computer system.
- Virtual memory enables multiple programs to share the physical memory without:
  - Knowing other programs exist (transparency).
  - Worrying about one program modifying the data contents of another (protection).

Managing virtual memory

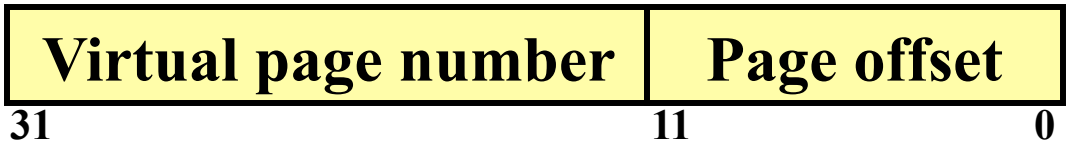
- Managed by hardware logic ***and*** operating system software.
  - Hardware for speed
  - Software for flexibility and because disk storage is controlled by the operating system.

Virtual Memory

- Treat main memory like a cache
  - Misses go to the disk
- How do we minimize disk accesses?
  - Buy lots of memory.
  - Exploit temporal locality
    - Fully associative? Set associative? Direct mapped?
  - Exploit spatial locality
    - How big should a block be?
  - Write-back or write-through?

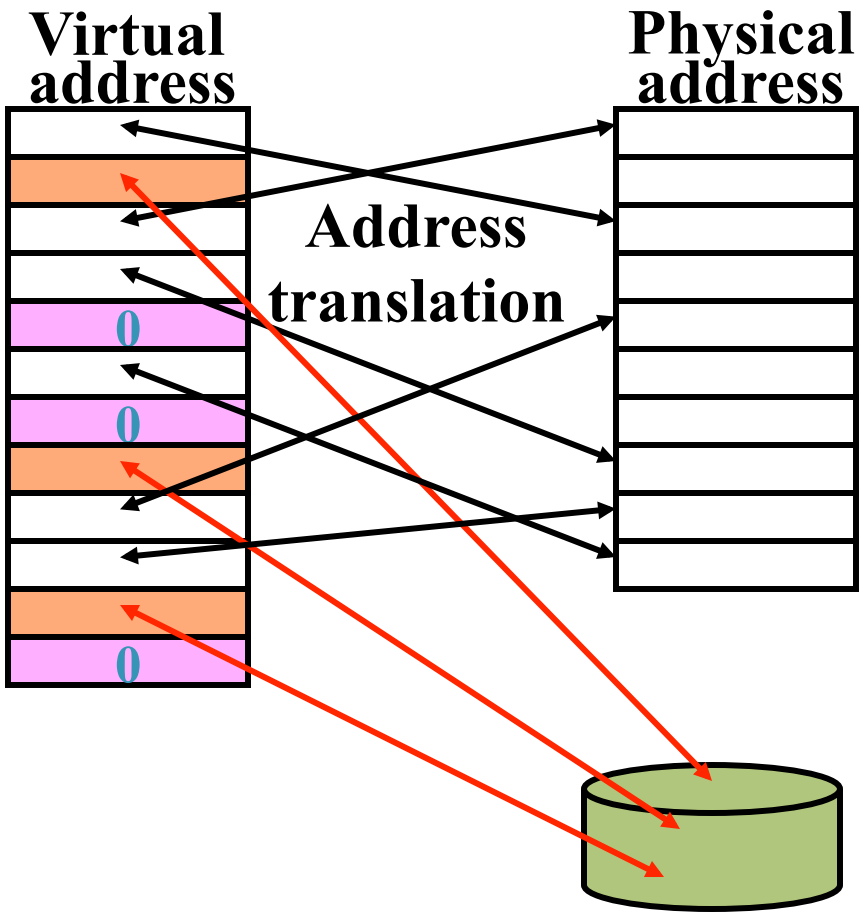
Virtual memory terminology

- Blocks are called Pages
  - A virtual address consists of
    - A virtual page number
    - A page offset field (low order bits of the address)



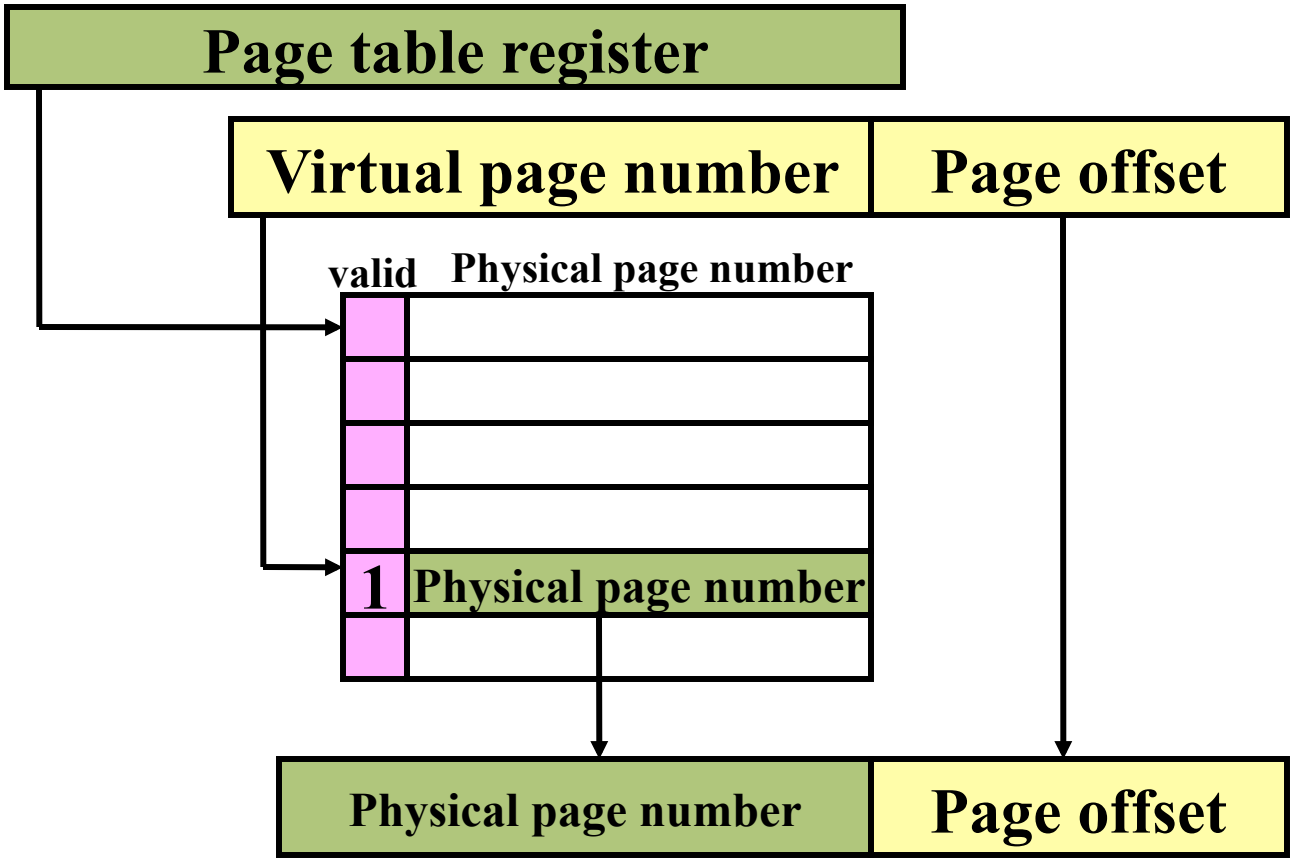
- Misses are call Page faults
  - and they are generally handled as an exception

Address Translation

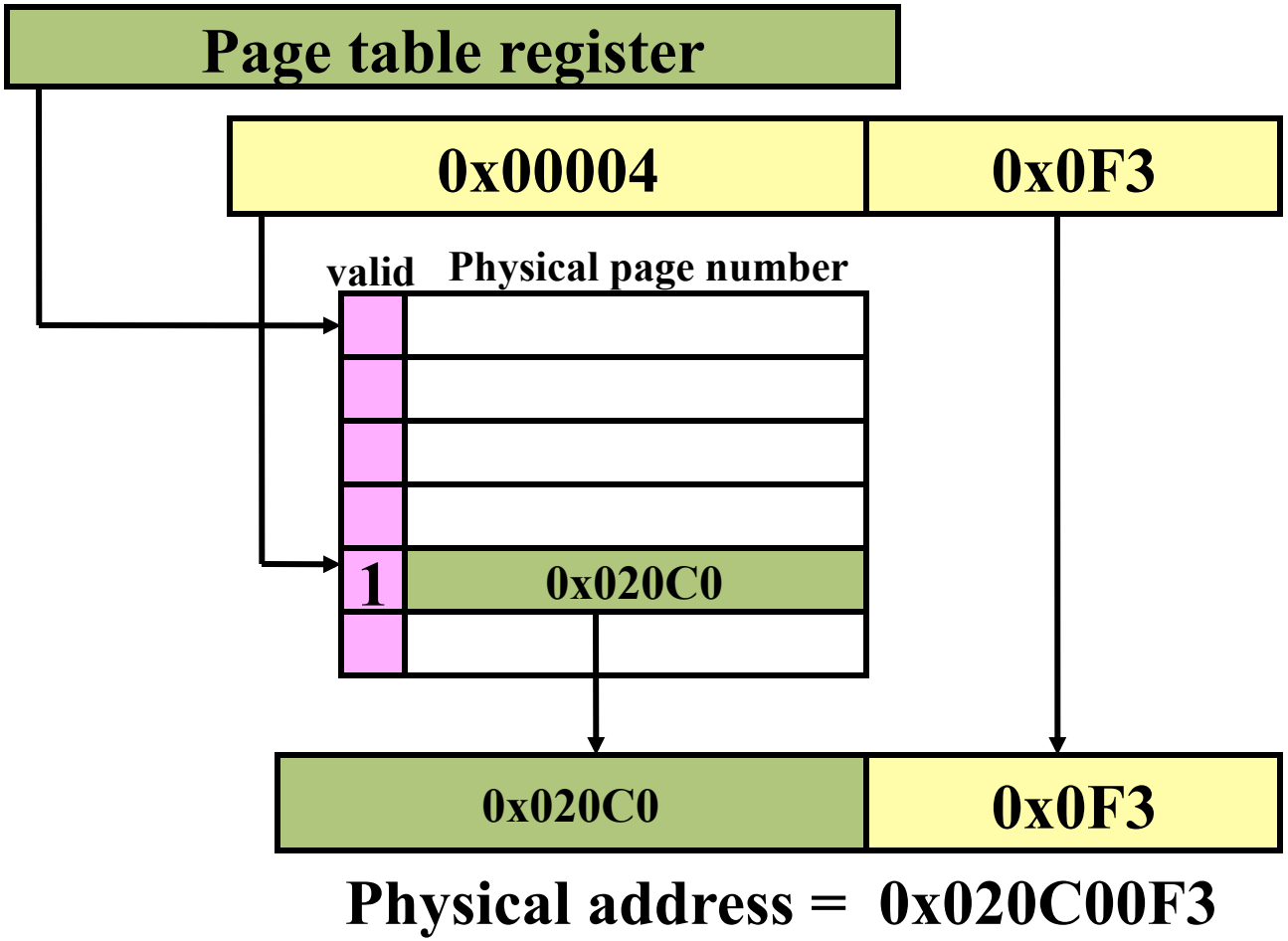


page table: **contains address translation info of the program.**

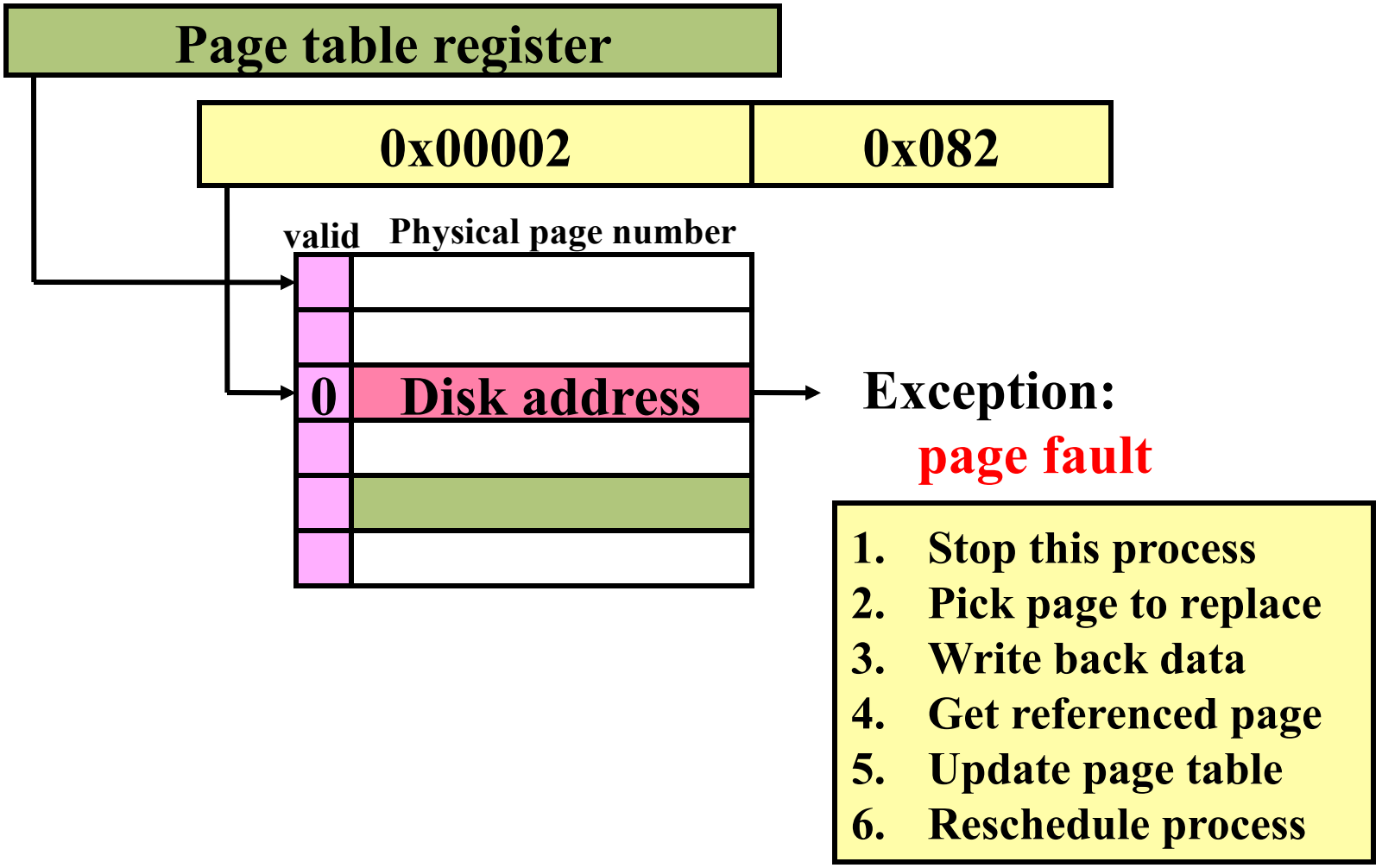
Page table components



Page table components



Page table components



How do we find it on disk?

- That is not a hardware problem! ☺
- Most operating systems partition the disk into logical devices (C: , D: , /home, etc.)
- They also have a hidden partition to support the disk portion of virtual memory
  - Swap partition on UNIX machines
  - You then index into the correct page in the swap partition.

Size of page table

- How big is a page table entry?
  - For MIPS the virtual address is 32 bits
    - If the machine can support 1GB = 2^{30} of physical memory and we use pages of size 4KB = 2^{12}, then the physical page number is 30-12 = 18 bits. Plus another valid bit + other useful stuff (read only, dirty, etc.)
    - Let say about 3 bytes.
- How many entries in the page table?
  - MIPS virtual address is 32 bits – 12 bit page offset = 2^{20} or ~1,000,000 entries
- Total size of page table: 2^{20} x 18 bits ~ 3 MB

How can you organize the page table?

1. Continuous 3MB region of physical memory
2. Use a hash function instead of an array
   - Slower, but less memory
3. Page the page table! (Build a hierarchical page table)
   - Super page table in physical memory
   - Second (and maybe third) level page tables in virtual address space.

Putting it all together

- Loading your program in memory
  - Ask operating system to create a new process
  - Construct a page table for this process
  - Mark all page table entries as invalid with a pointer to the disk image of the program
    - That is, point to the executable file containing the binary.
  - Run the program and get an immediate page fault on the first instruction.

Page replacement strategies

- Page table indirection enables a fully associative mapping between virtual and physical pages.
- How do we implement LRU?
  - True LRU is expensive, but LRU is a heuristic anyway, so approximating LRU is fine
  - Reference bit on page, cleared occasionally by operating system. Then pick any "unreferenced" page to evict.

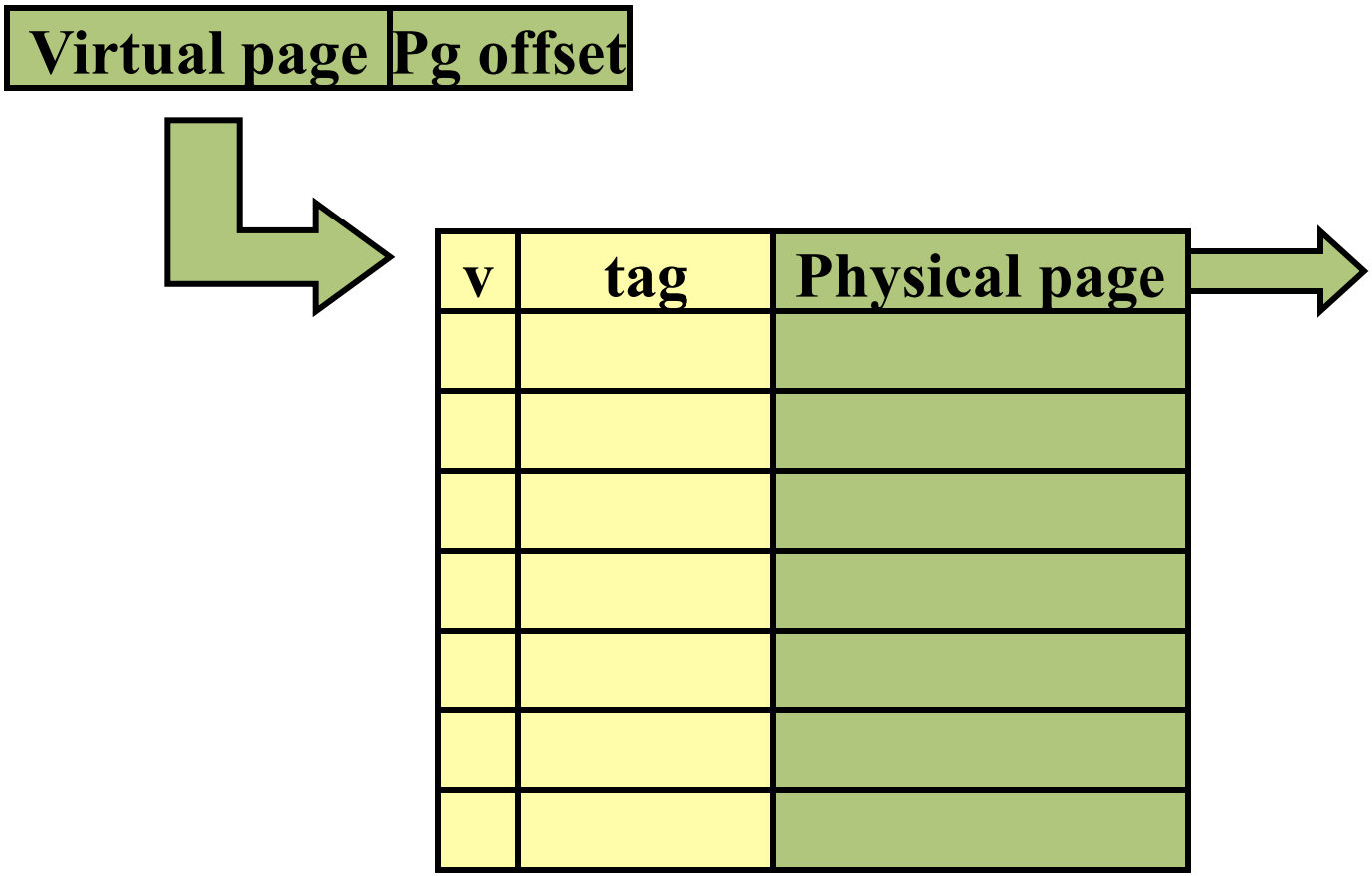
Performance of virtual memory

- To translate a virtual address into a physical address, we must first access the page table in physical memory.
- Then we access physical memory again to get (or store) the data
  - A load instruction performs at least 2 memory reads
  - A store instruction performs at least 1 read and then a write.
- Every memory access performs at least one slow access to main memory!

Translation lookaside buffer

- We fix this performance problem by avoiding main memory in the translation from virtual to physical pages.
- We buffer the common translations in a **Translation lookaside buffer (TLB),** a fast cache memory dedicated to storing a small subset of valid VtoP translations.

TLB



Where is the TLB lookup?

- We put the TLB lookup in the pipeline after the virtual address is calculated and before the memory reference is performed.
  - This may be before or during the data cache access.
  - Without a TLB we need to perform the translation during the memory stage of the pipeline.

Other VM translation functions

- Page data location
  - Physical memory, disk, uninitialized data
- Access permissions
  - Read only pages for instructions
- Gathering access information
  - Identifying dirty pages by tracking stores
  - Identifying accesses to help determine LRU candidate

Placing caches in a VM system

- VM systems give us two different addresses: virtual and physical
- Which address should we use to access the data cache?
  - Virtual address (before VM translation)
    - Faster access? More complex?
  - Physical address (after VM translations)
    - Delayed access?

Other VM functions

- Page data location
  - Physical memory, disk, uninitialized data
- Access permissions
  - Read only pages for instructions
- Gathering access information
  - Identifying dirty pages by tracking stores
  - Identifying accesses to help determine LRU candidate

Page replacement strategies

- Page table indirection enables a fully associative mapping between virtual and physical pages.
- How do we implement LRU?
  - True LRU is expensive, but LRU is a heuristic anyway, so approximating LRU is fine
  - Reference bit on page, cleared occasionally by operating system. Then pick any "unreferenced" page to evict.

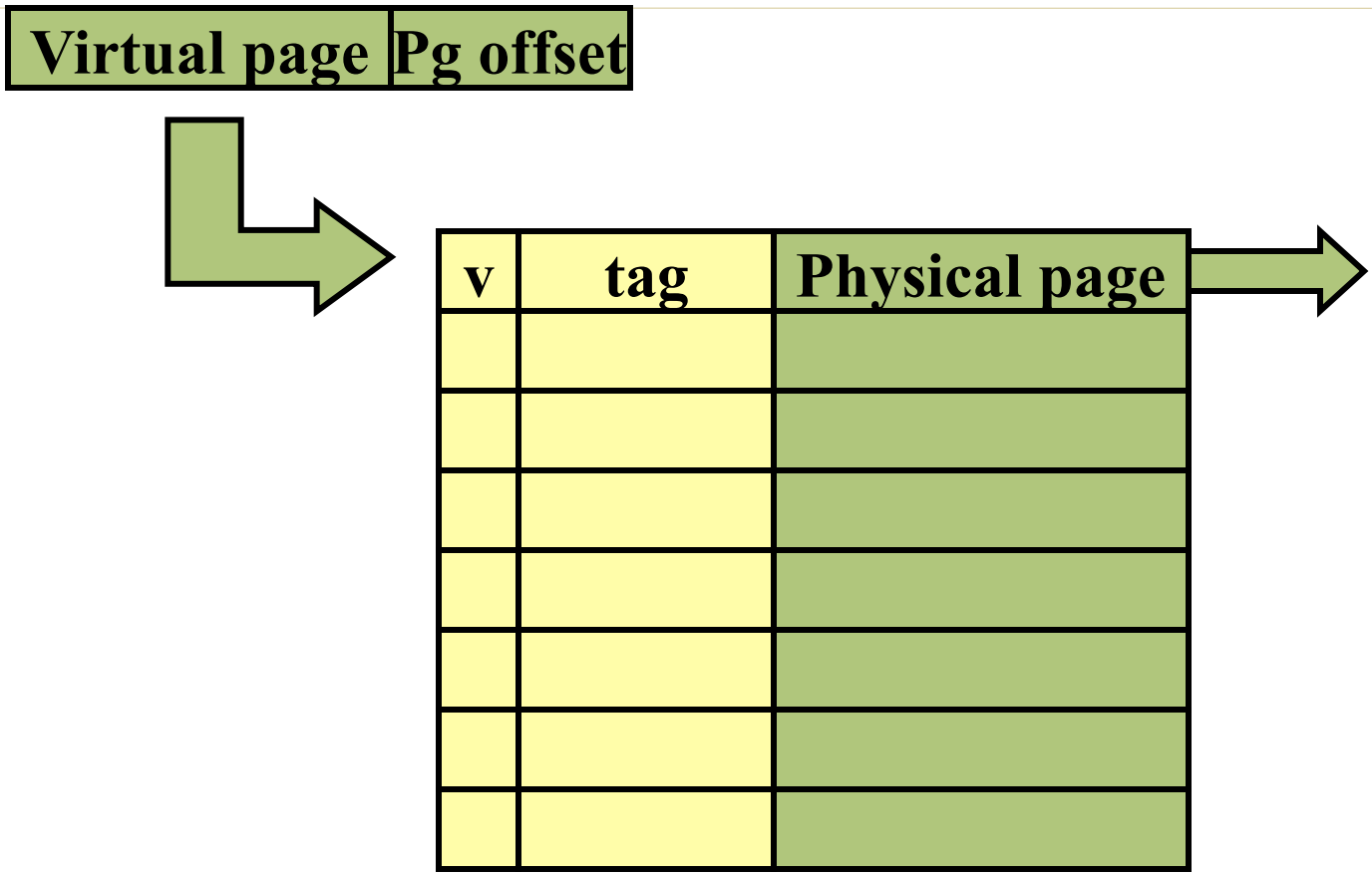
Performance of virtual memory

- We must access physical memory to access the page table to make the translation from a virtual address to a physical one
- Then we access physical memory again to get (or store) the data
  - A load instruction performs at least 2 memory reads
  - A store instruction performs at least 1 read and then a write.

Translation lookaside buffer

- We fix this performance problem by avoiding memory in the translation from virtual to physical pages.
- We buffer the common translations in a **translation lookaside buffer (TLB)**

TLB



Where is the TLB lookup?

- We put the TLB lookup in the pipeline after the virtual address is calculated and before the memory reference is performed.
  - This may be before or during the data cache access.
  - Without a TLB we need to perform the translation during the memory stage of the pipeline.

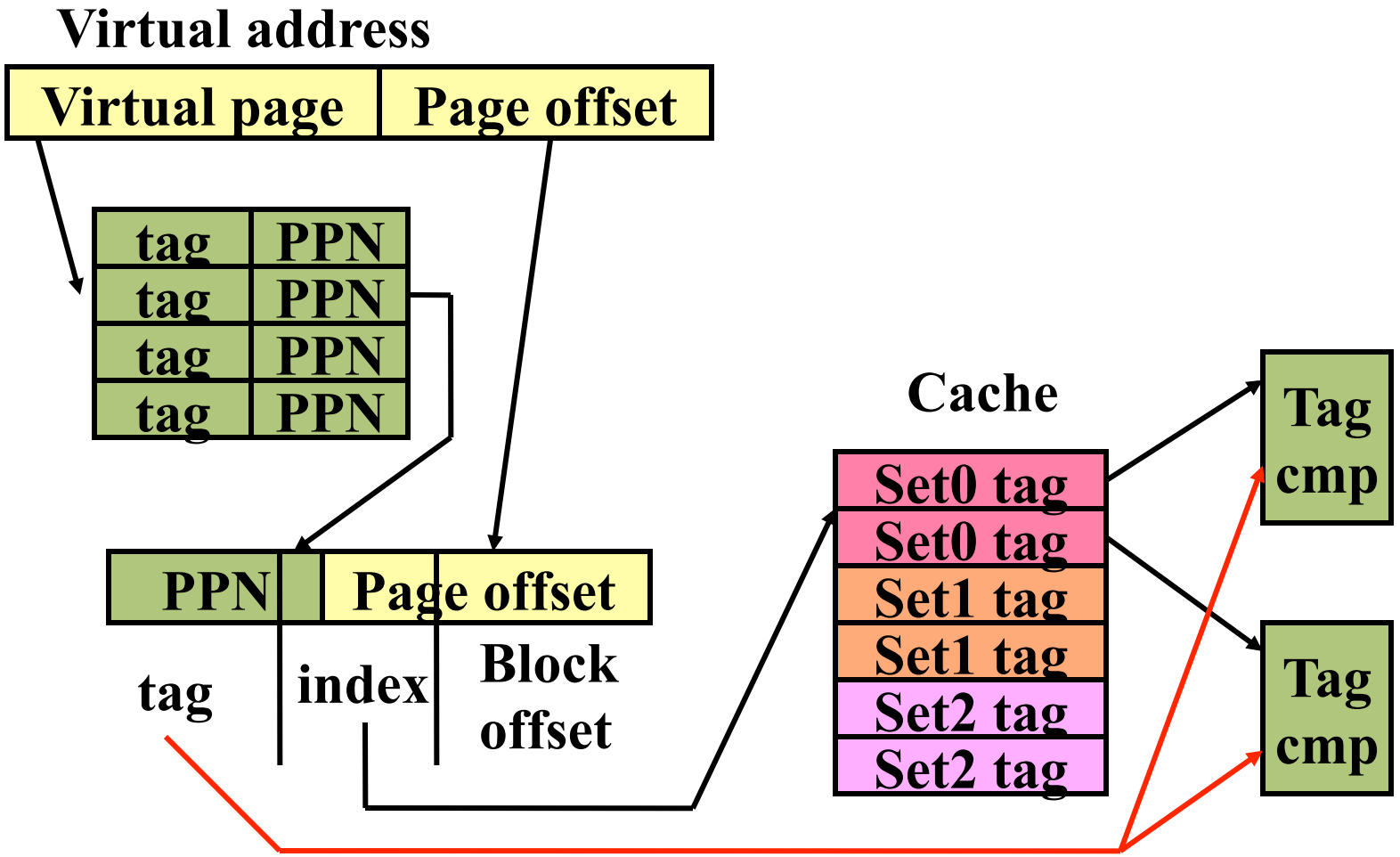
Placing caches in a VM system

- VM systems give us two different addresses: virtual and physical
- Which address should we use to access the data cache?
  - Virtual address (before VM translation)
    - Faster access? More complex?
  - Physical address (after VM translations)
    - Delayed access?

Physically addressed caches

- Perform TLB lookup ***before*** cache tag comparison.
  - Use bits from physical address to index set
  - Use bits from physical address to compare tag
- Slower access?
  - Tag lookup takes place ***after*** the TLB lookup.
- Simplifies some VM management
  - When switching processes, TLB must be invalidated, but cache OK to stay as is.

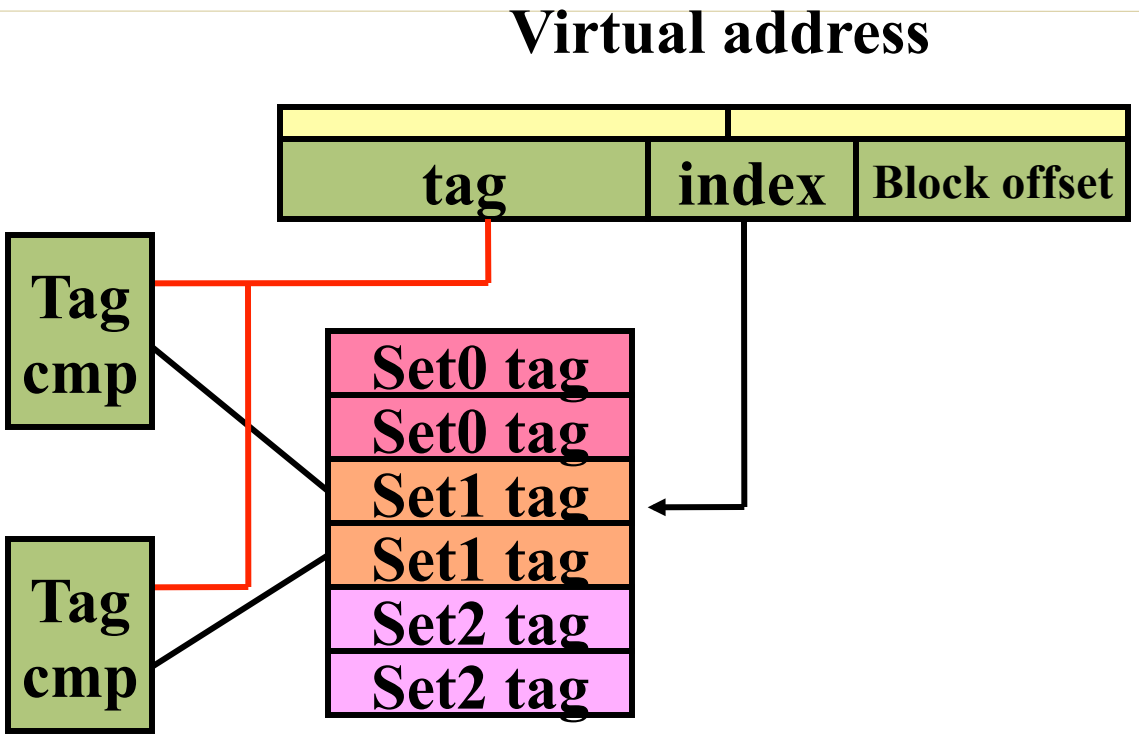
Picture of Physical caches



Virtually addressed caches

- Perform the TLB lookup at the same time as the cache tag compare.
  - Uses bits from the virtual address to index the cache set
  - Uses bits from the virtual address for tag match.
- Problems:
  - Aliasing: Two processes may refer to the same physical location with different virtual addresses.
  - When switching processes, TLB must be invalidated, and dirty cache blocks must be written back to memory.

Picture of Virtual Caches

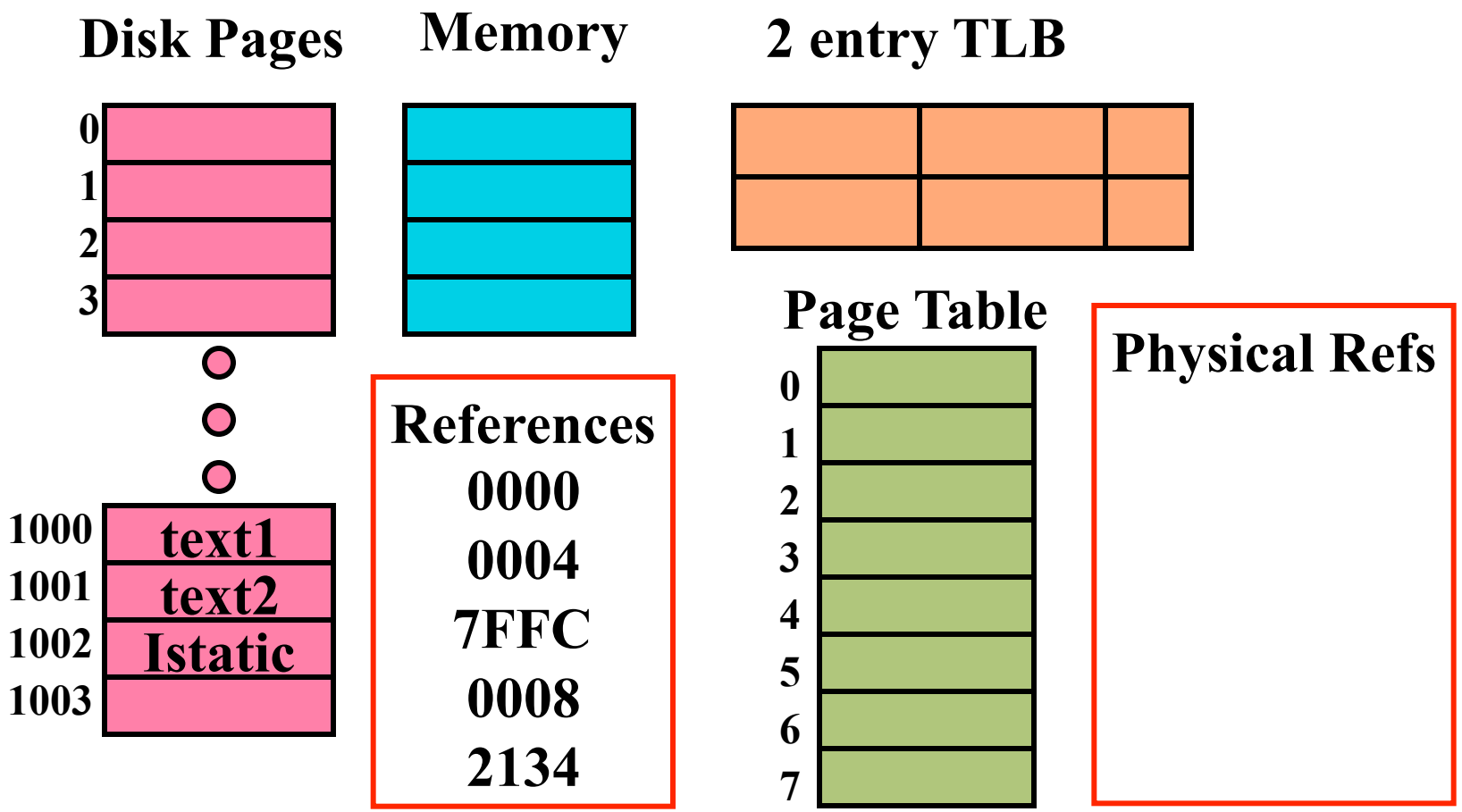


- **TLB is accessed in parallel with cache lookup**
- **Physical address is used to access main memory in case of a cache miss.**

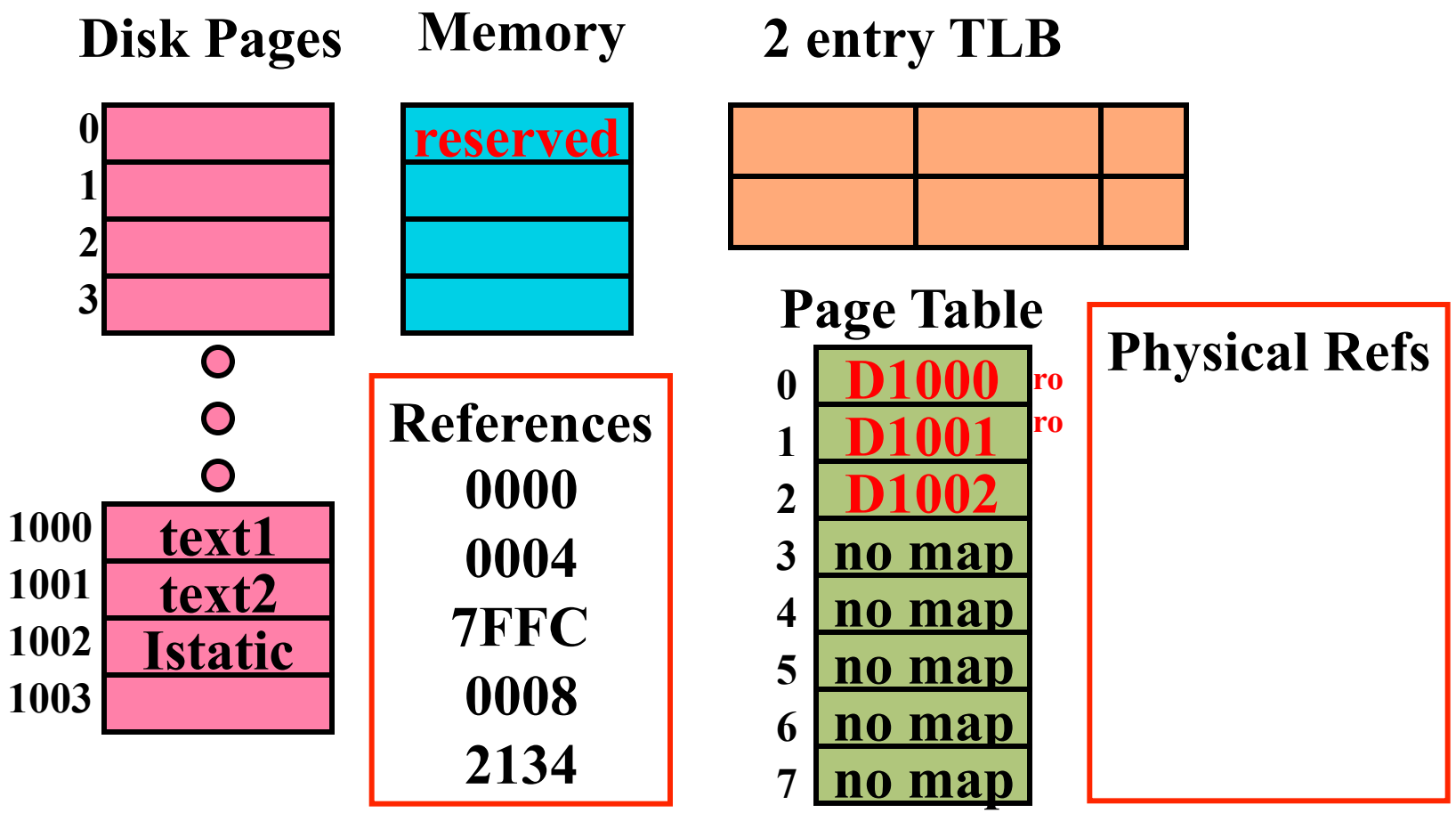
OS support for Virtual Memory

- It must be able to modify the page table register, update page table values, etc.
  - To enable the OS to do this, AND not the user program, we have different execution modes for a process – one which has executive (or supervisor or kernel level) permissions and one that has user level permissions.

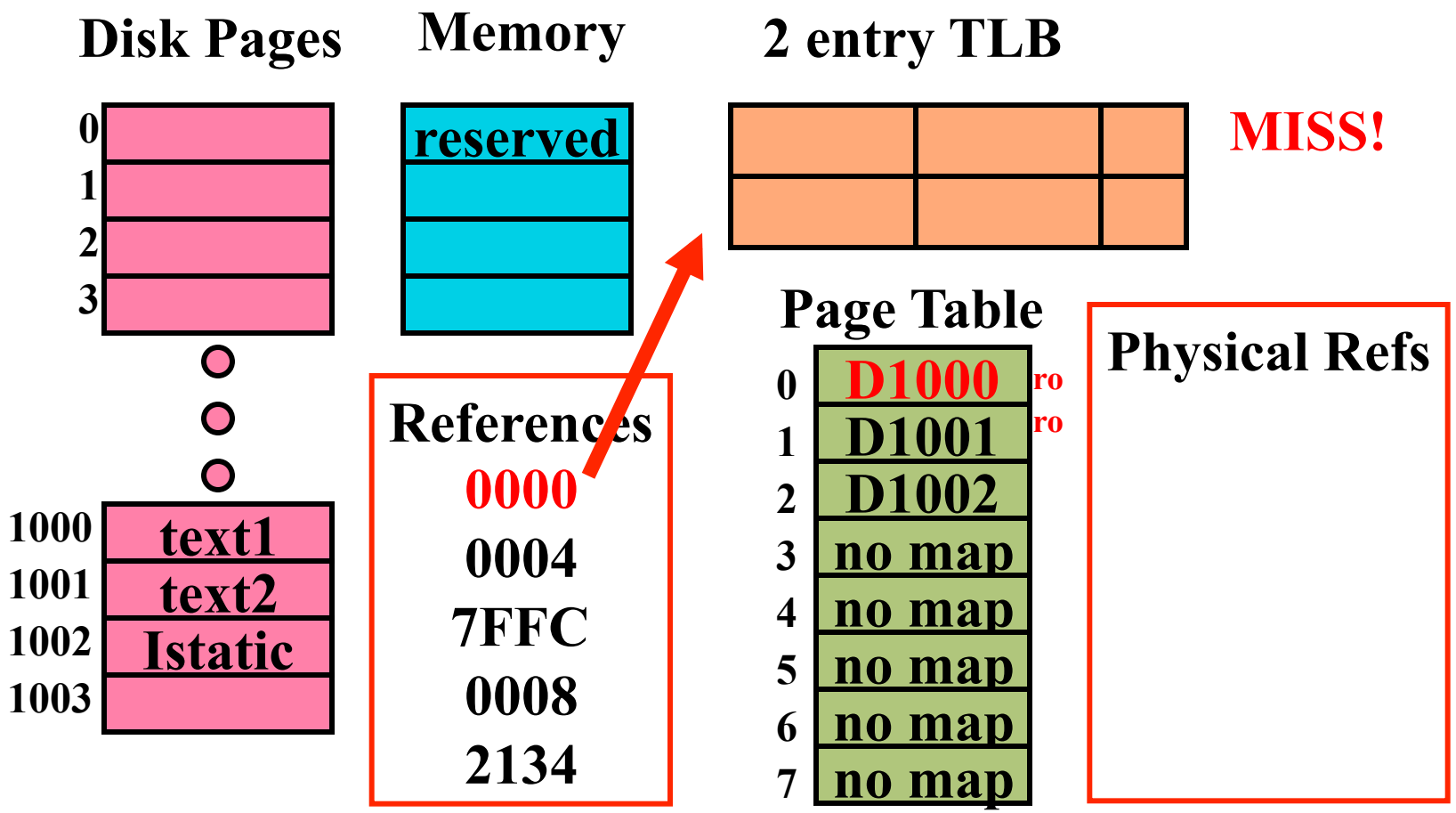
Extended Example: Loading a program into memory



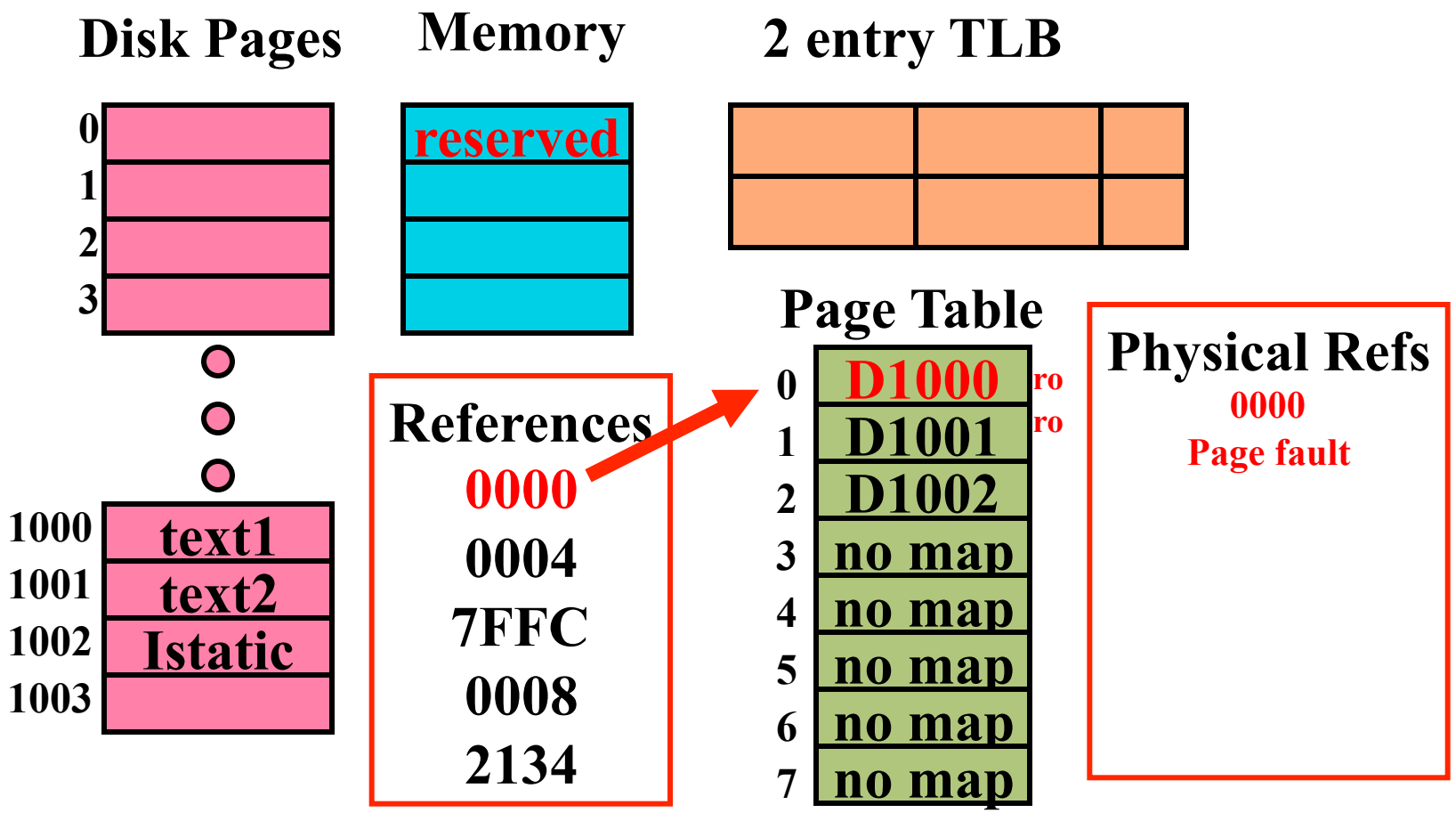
Step 1: read executable header and initialize page table



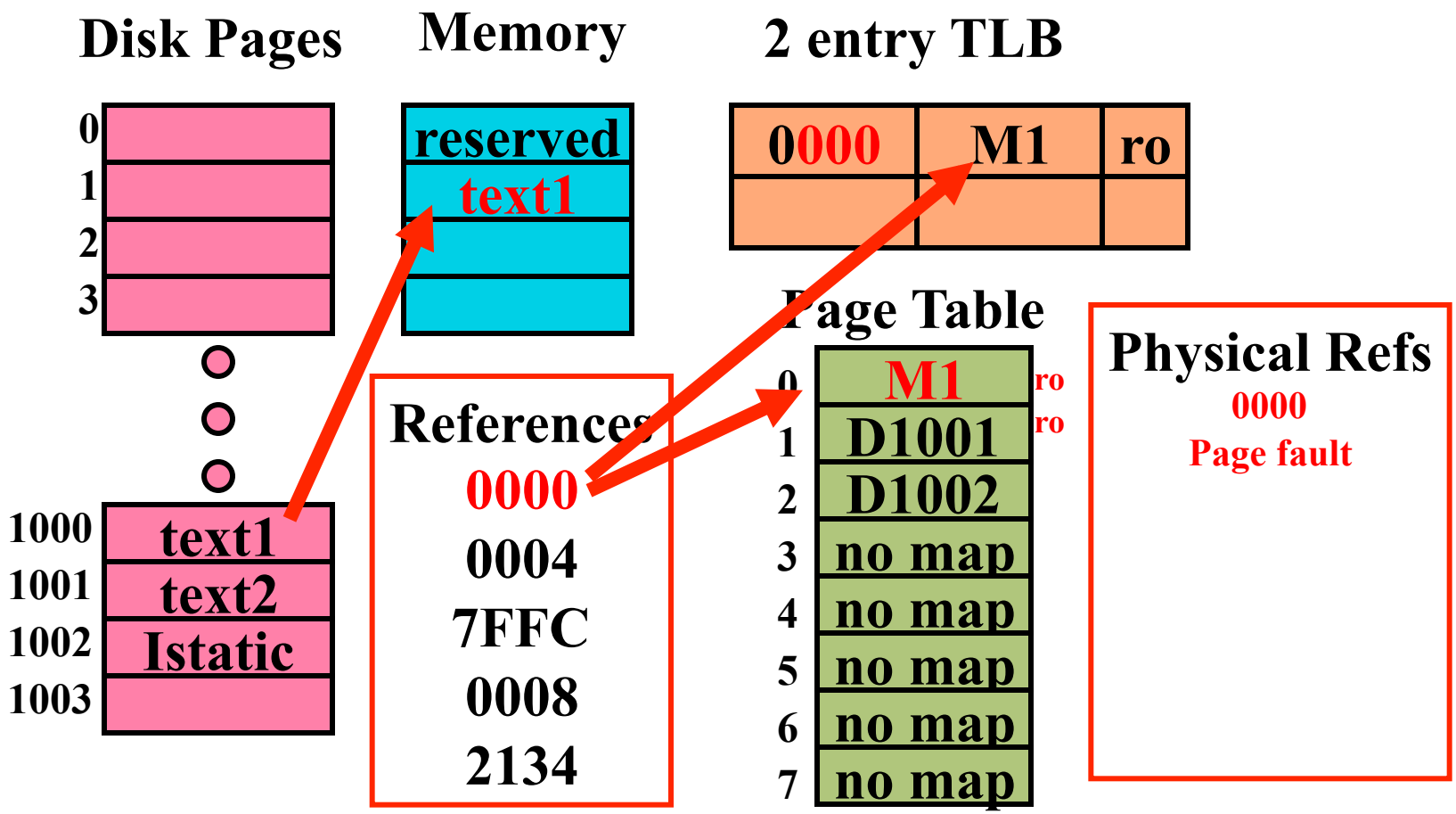
Step 2: load PC from header and start execution



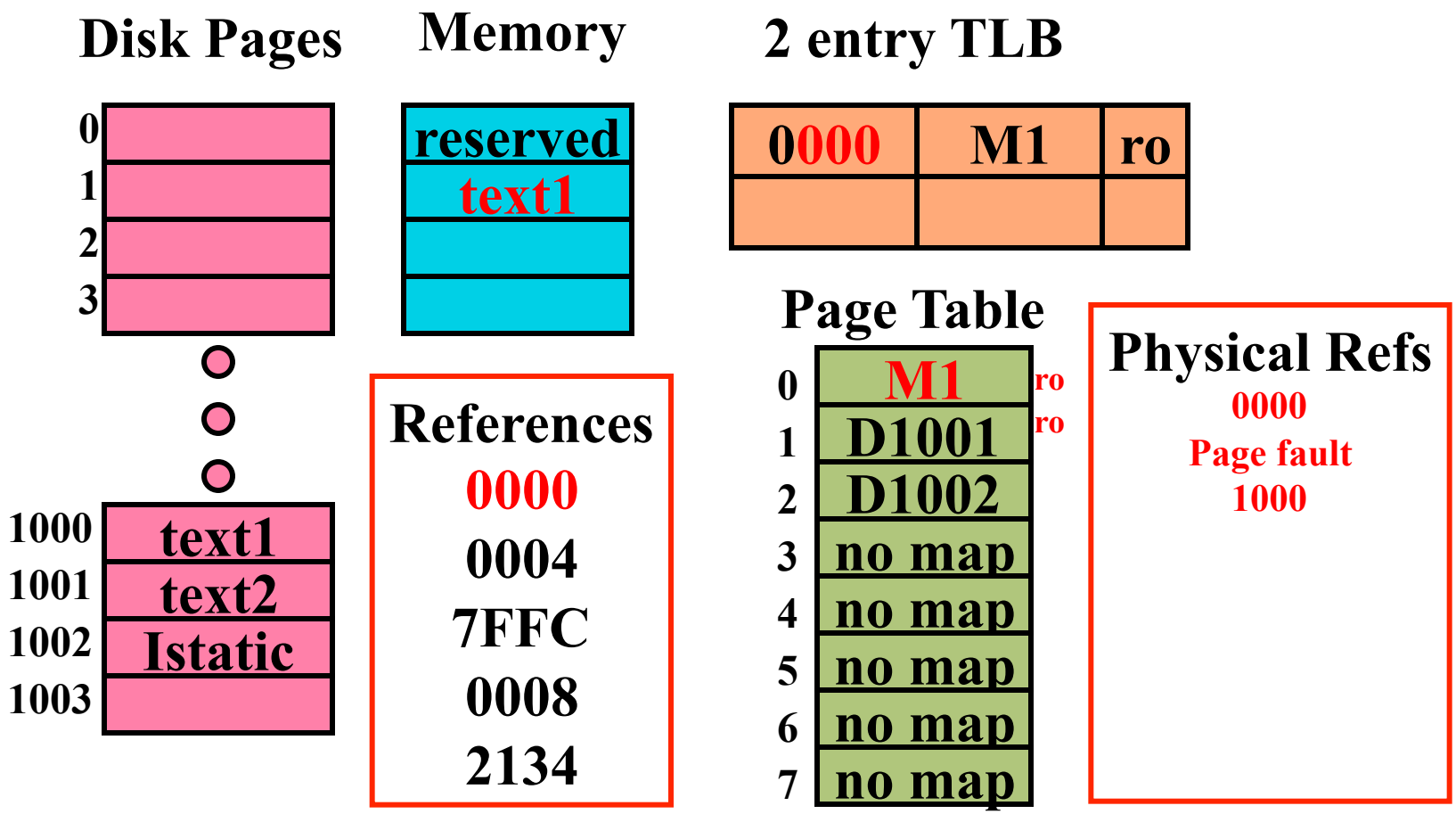
Fetching instr 0000



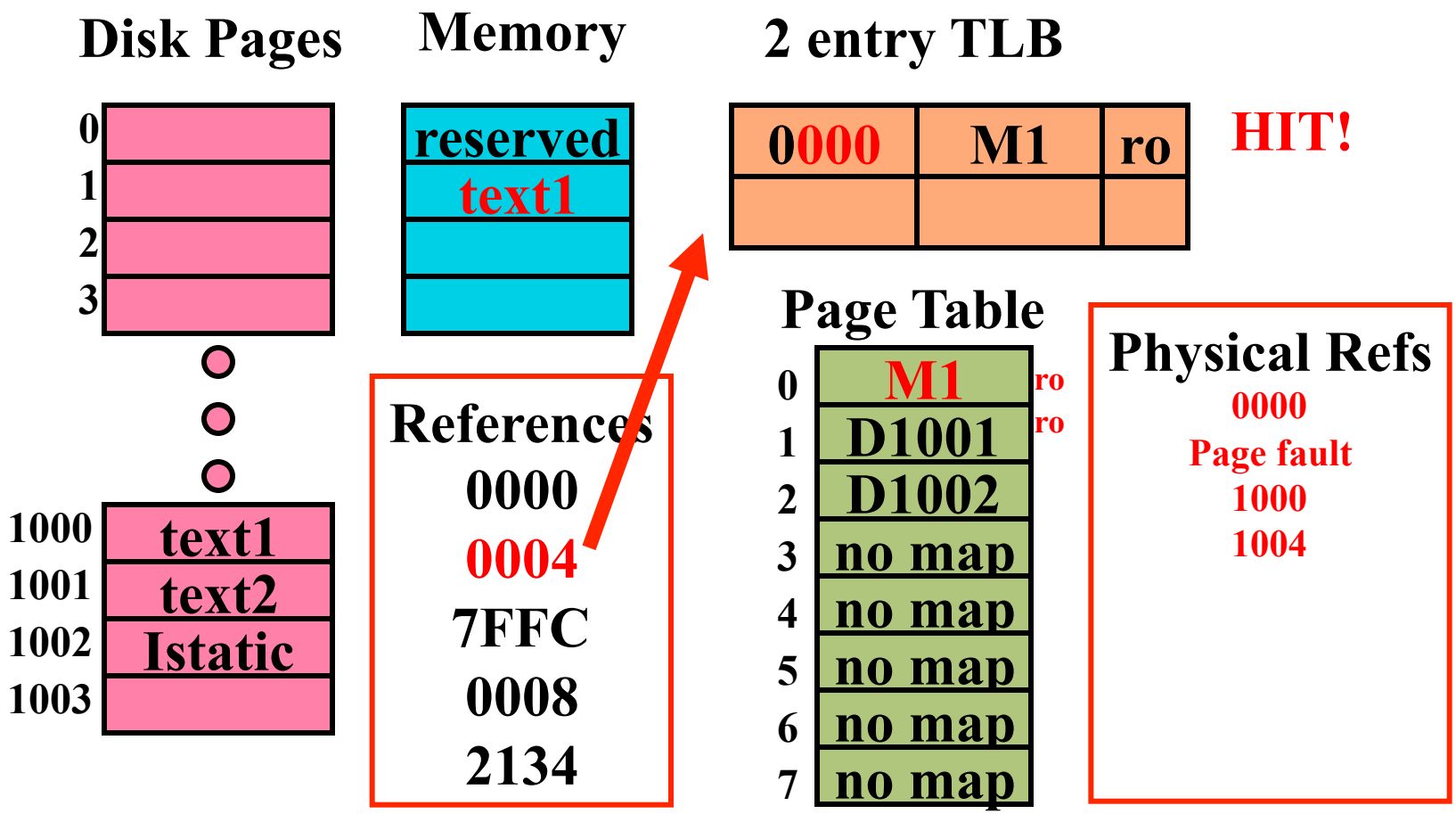
Fetching instr 0000



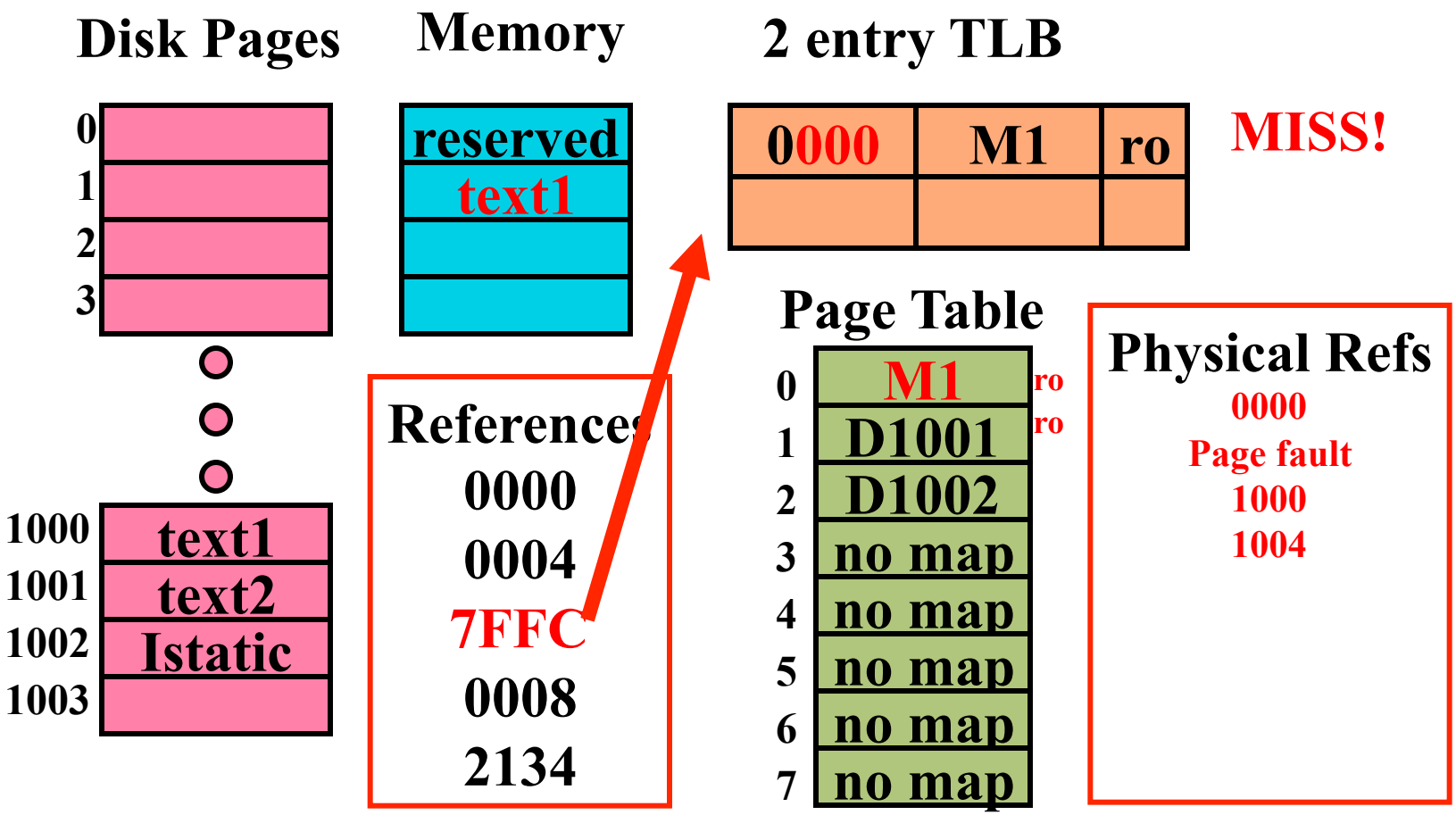
Fetching instr 0000



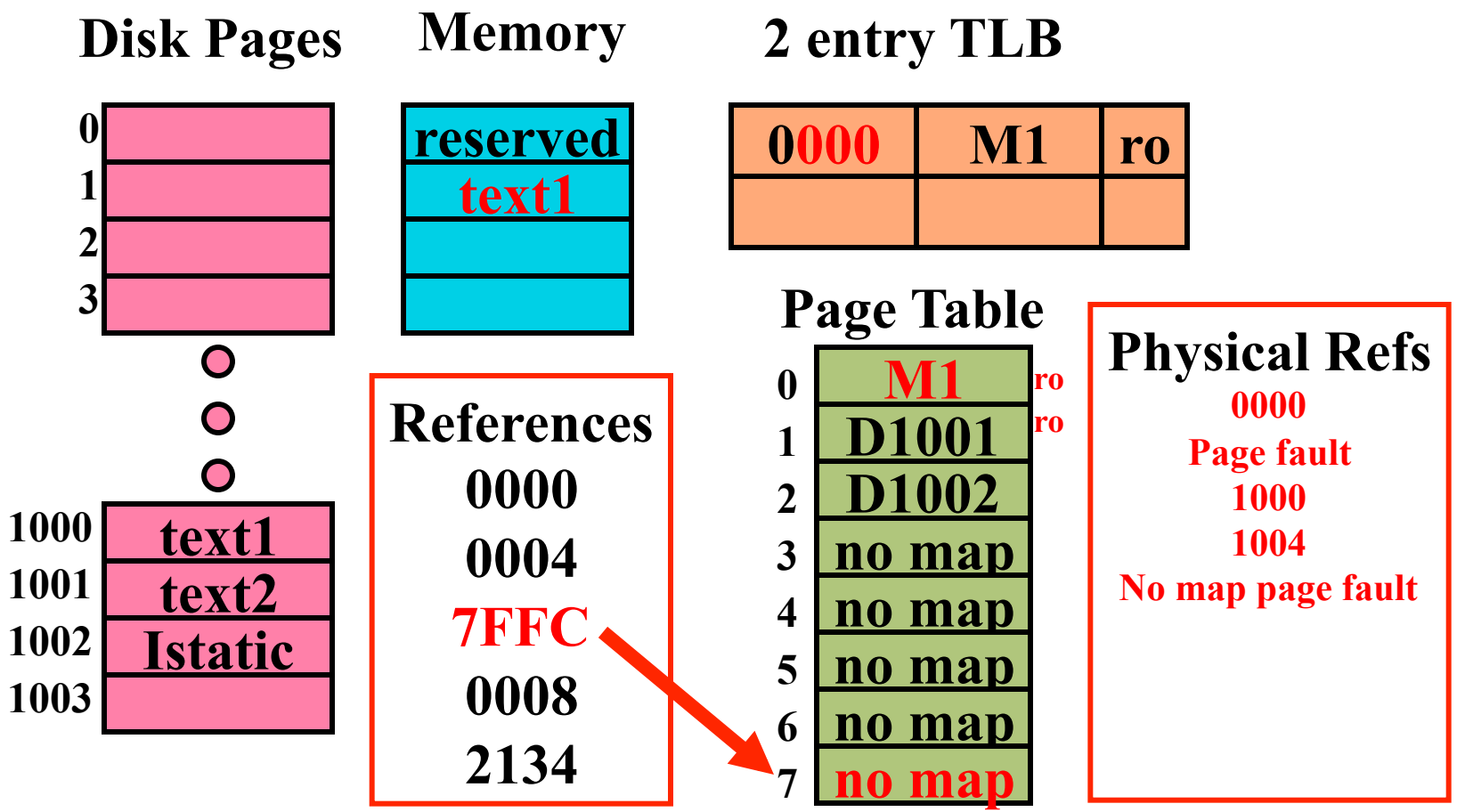
Fetching instr 0004



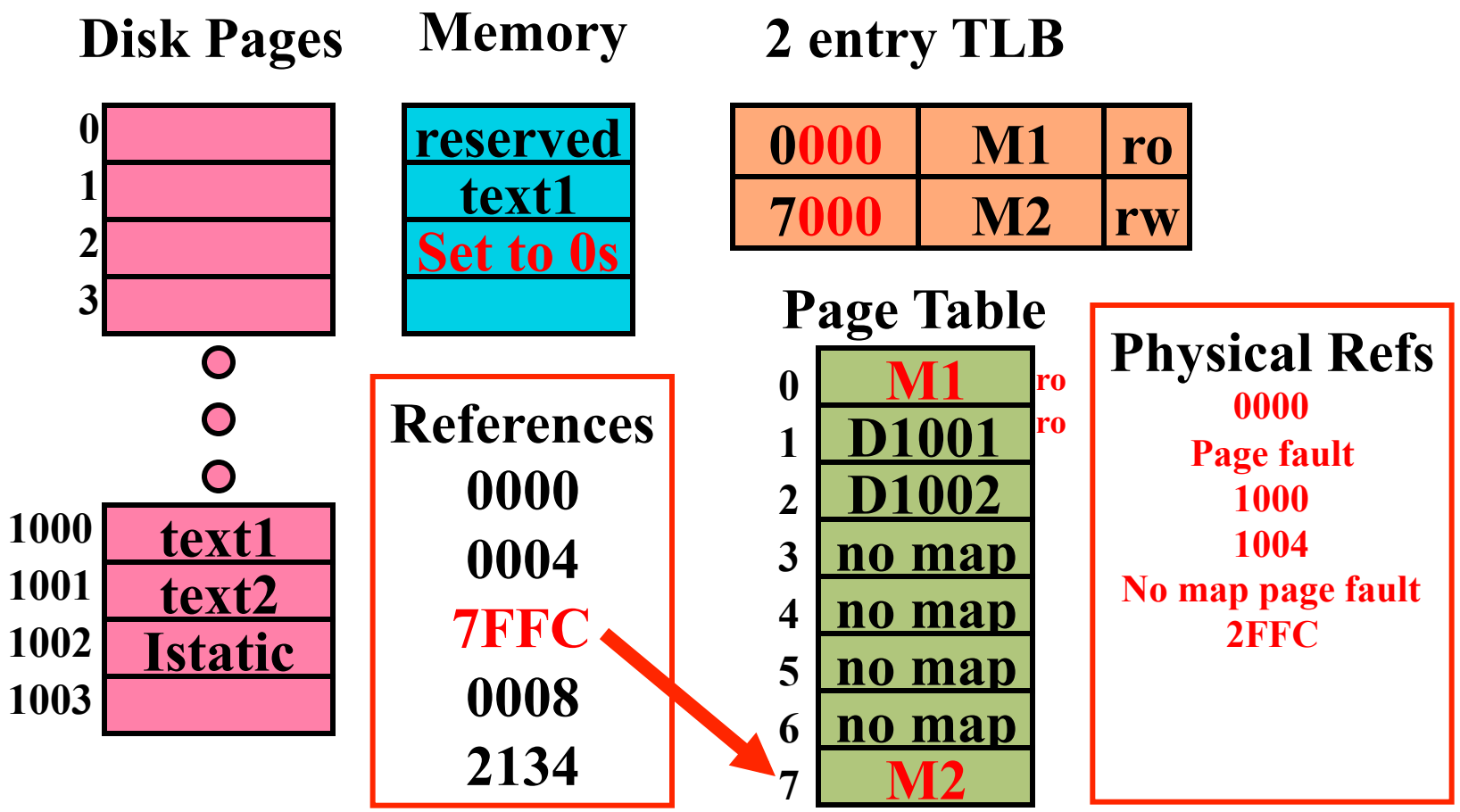
reference 7FFC



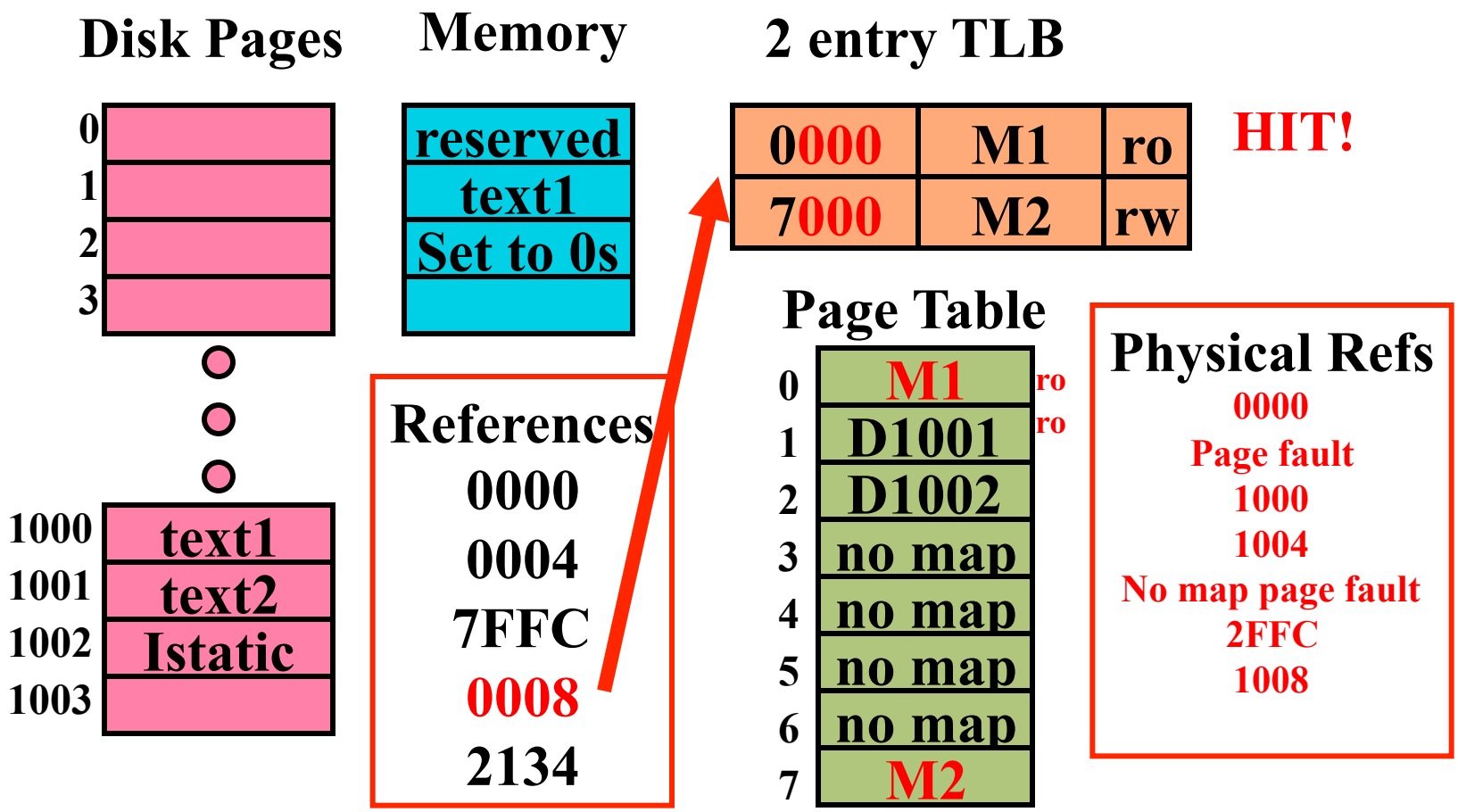
reference 7FFC



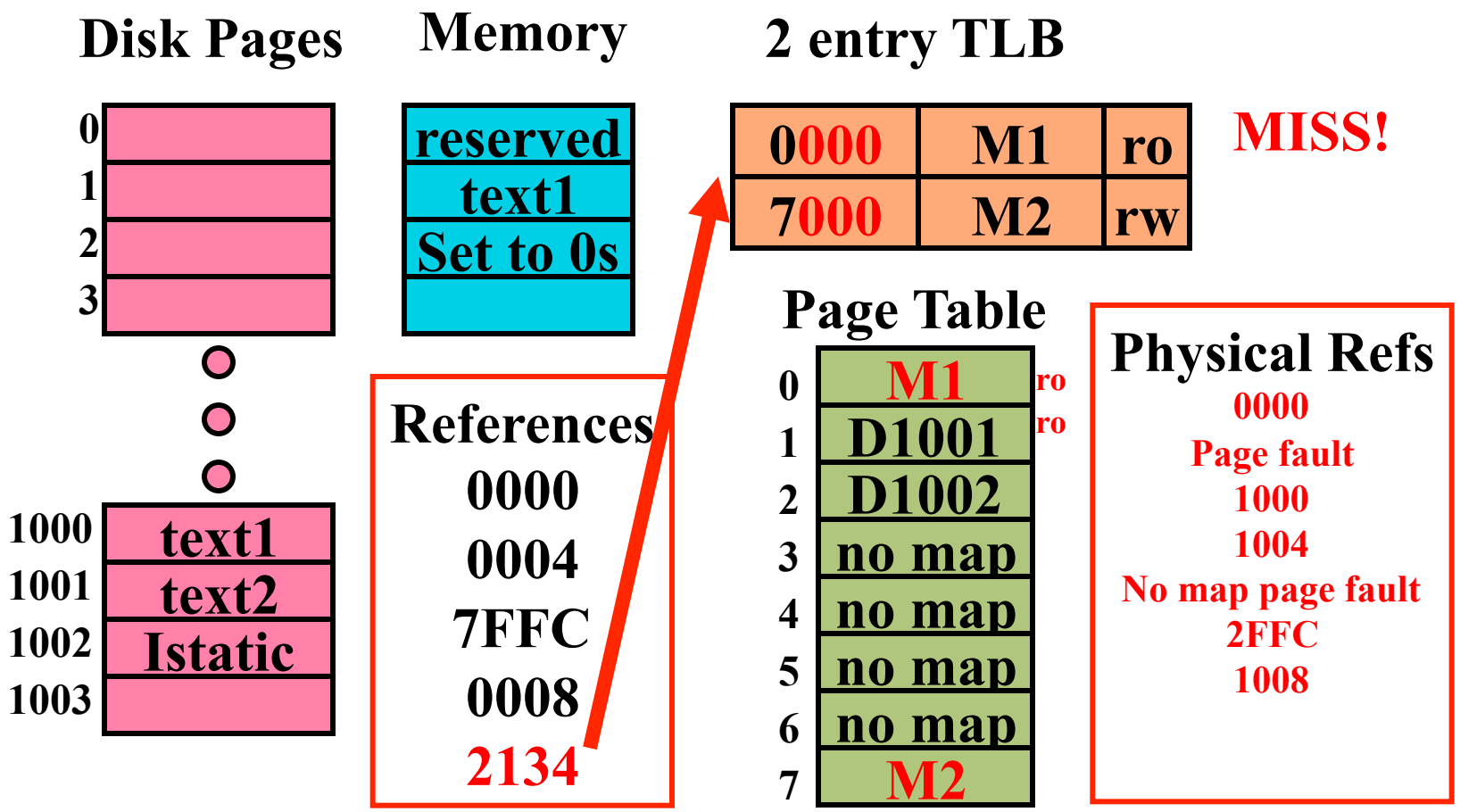
reference 7FFC



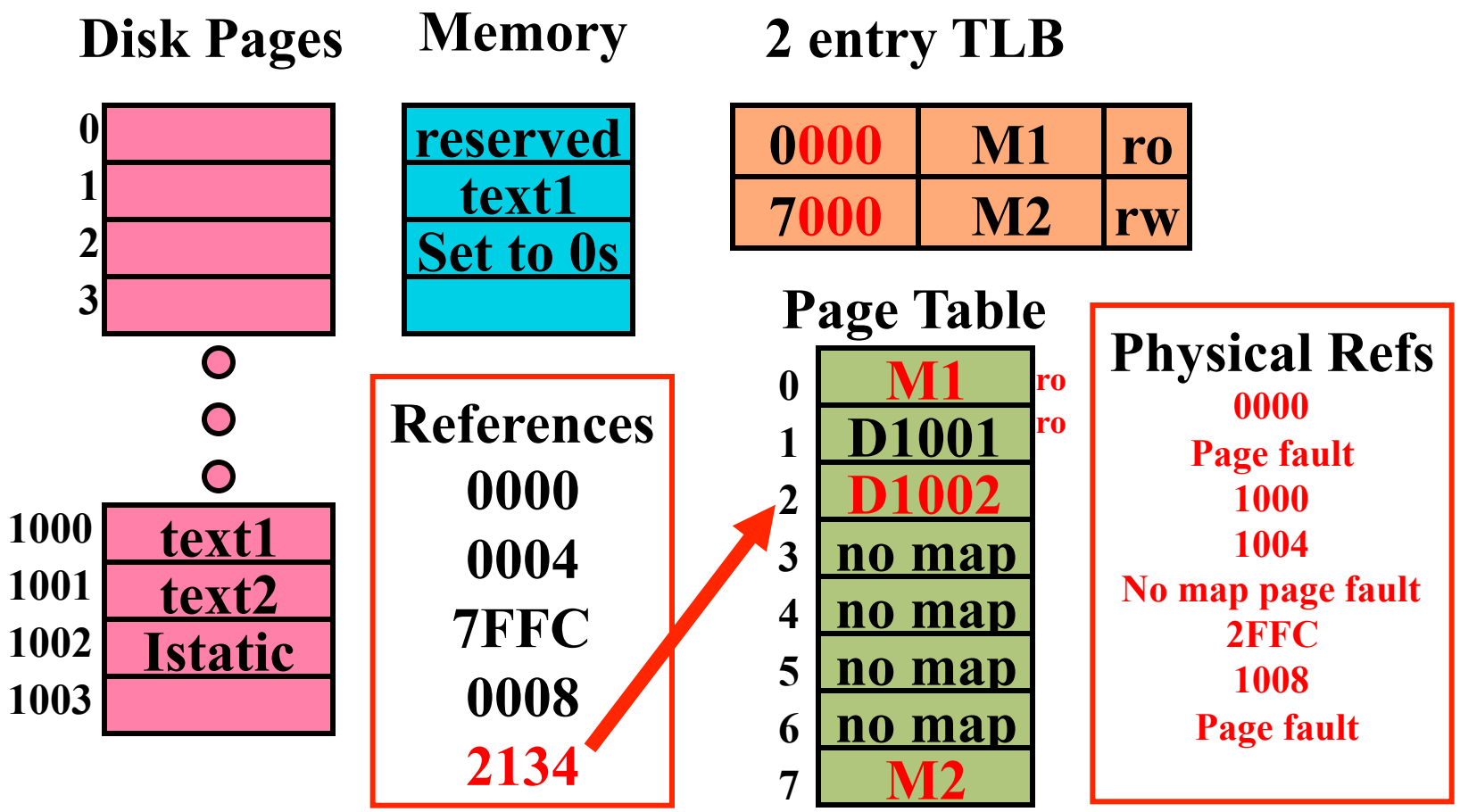
Fetching instr 0008



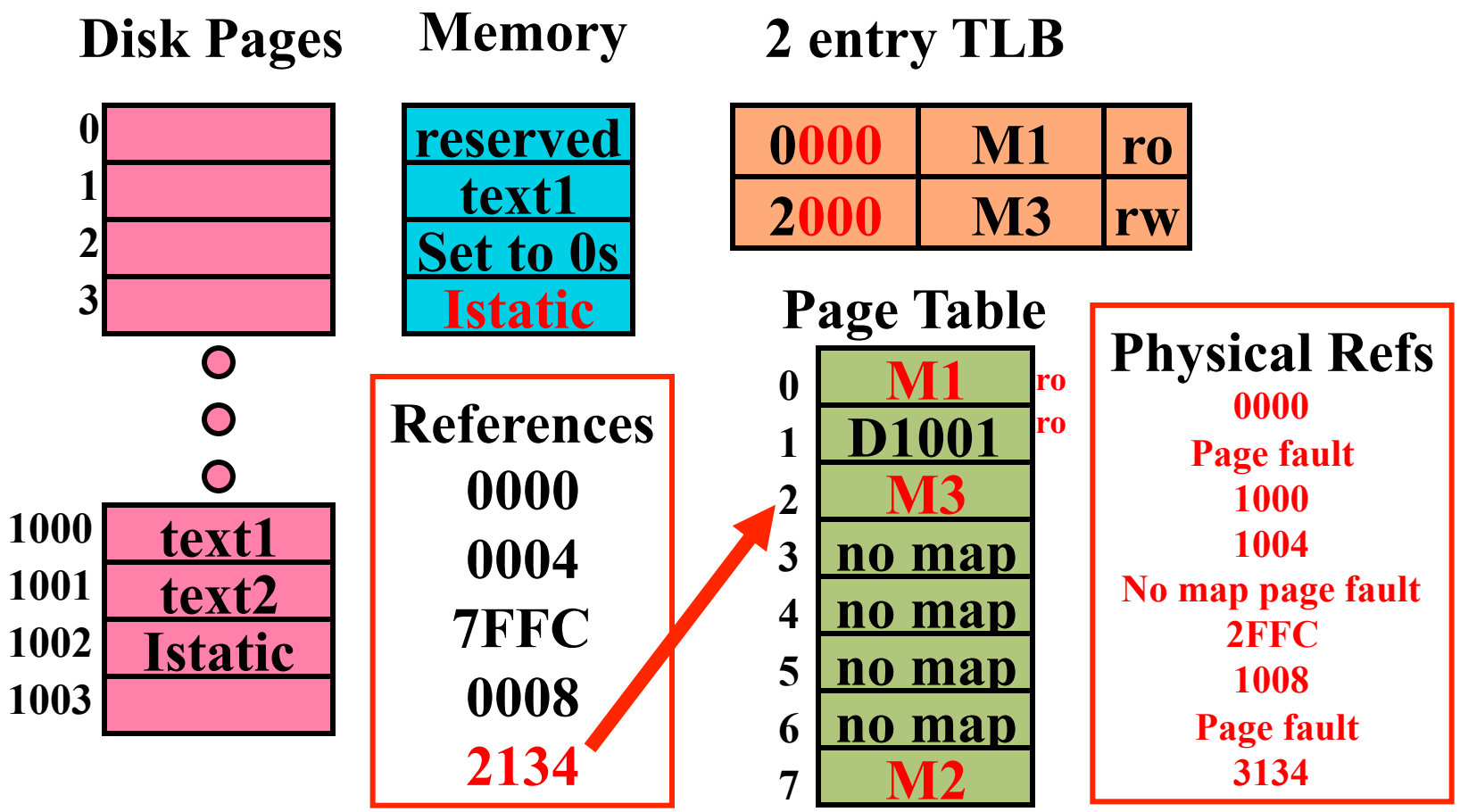
reference 2134



reference 2134



reference 2134



Multiple processes

- Virtual cache support for multiple processes:
  - Flush the cache between each context switch.
  - Use processID (a unique number for each processes given by the operating system) as part of the tag

Multiple processors

- Can run two programs at the same time
  - Each processor has its own cache. Why?

- May or may not share data
  - Sharing code is not a problem (read only)
    - Example: shared libraries, DDLs
  - Sharing data (read/write) is a problem
    - What if it is in one processors cache?
      - Solution: Snoopy caches