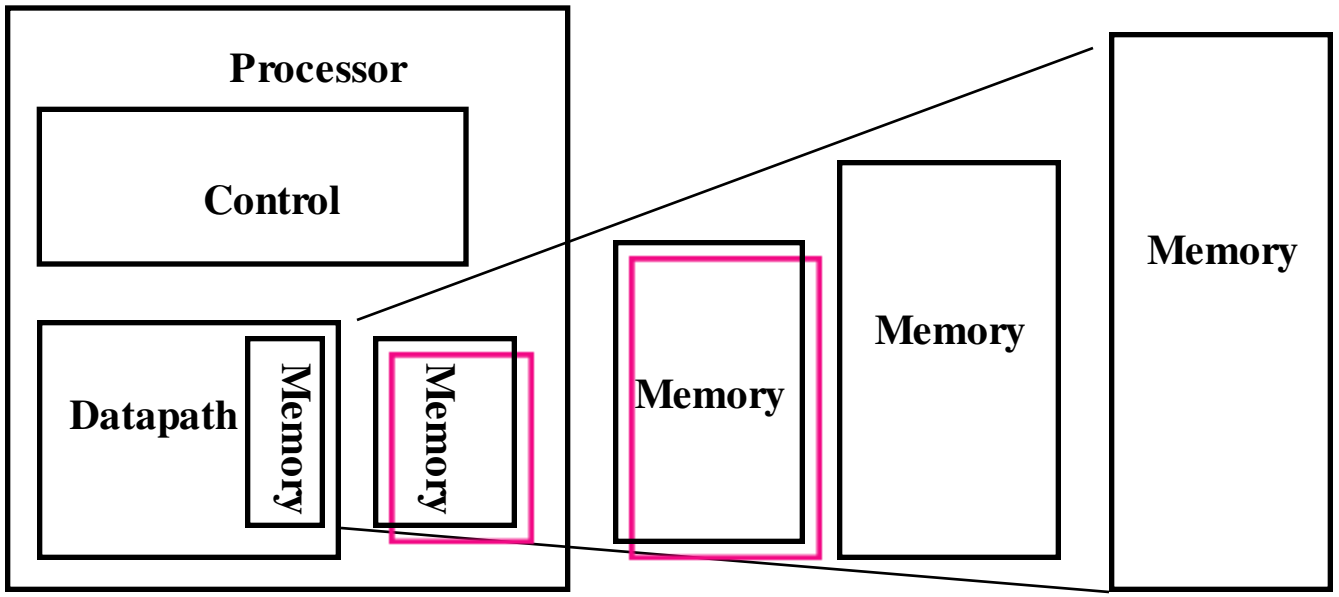
# BİLGİSAYAR MİMARİSİ

## Bellek Yönetim Sistemi

## Bölüm 7

- Memory Hierarchy
- Associative Memory
- Cache Memory
- Virtual Memory
- Memory Management Hardware

# An Expanded View of the Memory System




**Speed:** Fastest
**Size:** Smallest
**Cost:** Highest

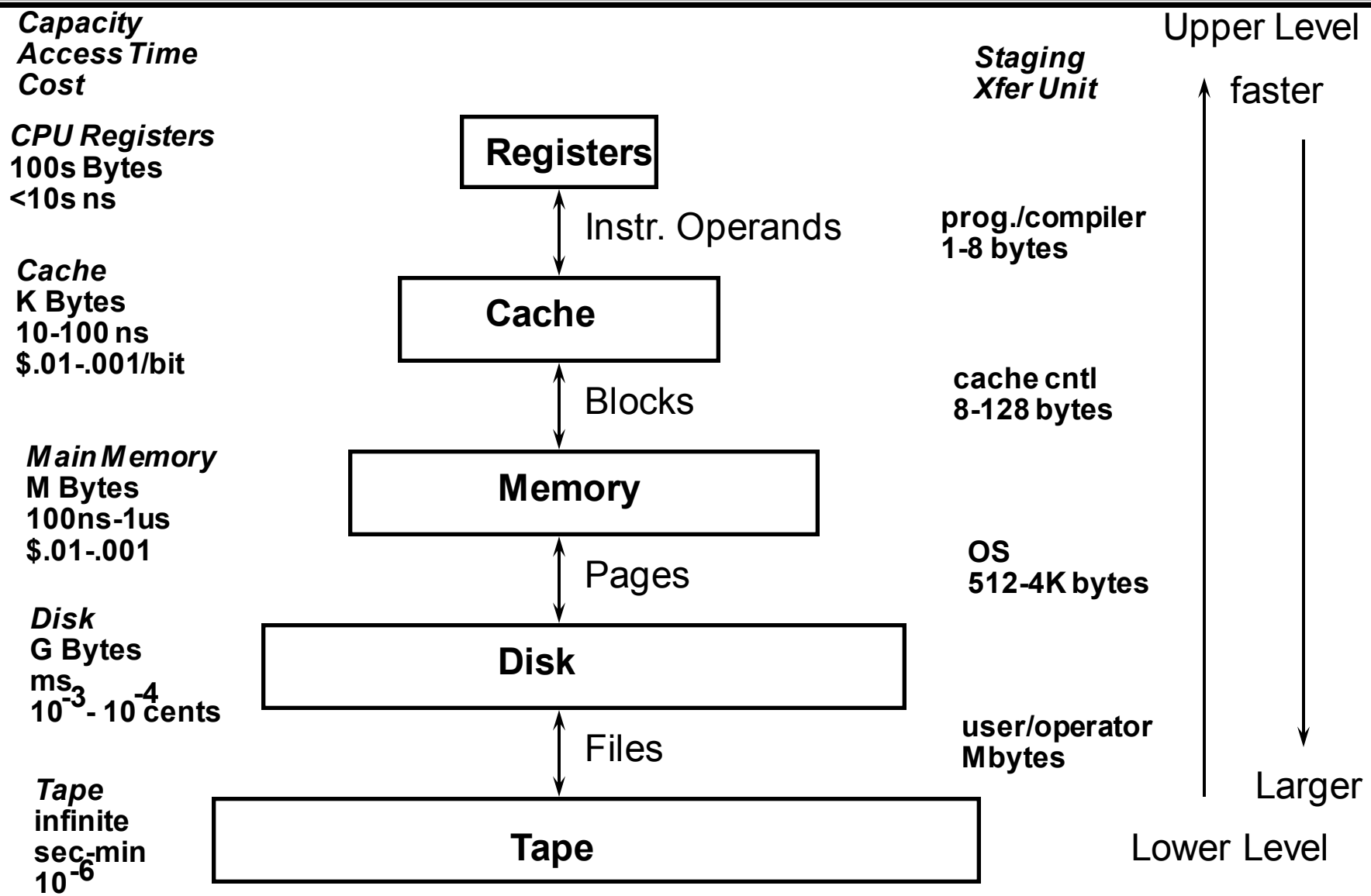
Slowest
Biggest
Lowest

# The important characteristics of a device:

- **Access Time**
- **average memory access time (AMAT)** is a common metric to analyze memory system performance. AMAT uses hit time, miss penalty, and miss rate to measure memory performance.
- **Transfer Rate**
- The memory transfer rate is determined by three factors; the memory bus clock rate, the type of transfer process and the number of bits transferred.
- **Step 1**
- Determine the bus clock rate. For example, the memory might operate at 300 MHz.
- **Step 2**
- Multiply the bus clock rate by 1, 2 or 3 depending on how many streams of information are flowing at once. DDR2 RAM, for example, has a multiplication factor of 2, and DDR3 RAM has a factor of 3.
- **Step 3**
- Multiply the result from the previous step by 64, which is the number of bits transferred.
- Divide the result from the previous step by 8 to get the transfer rate in bytes instead of bits, as there are 8 bits in a byte. You now know the memory transfer rate in both bits and bytes
- **Capacity**
- The total memory (RAM) that can be added to a computer depends on the address registers built into the CPU. For example, most 32-bit CPUs can address only up to 4 gigabytes (GB) of memory.

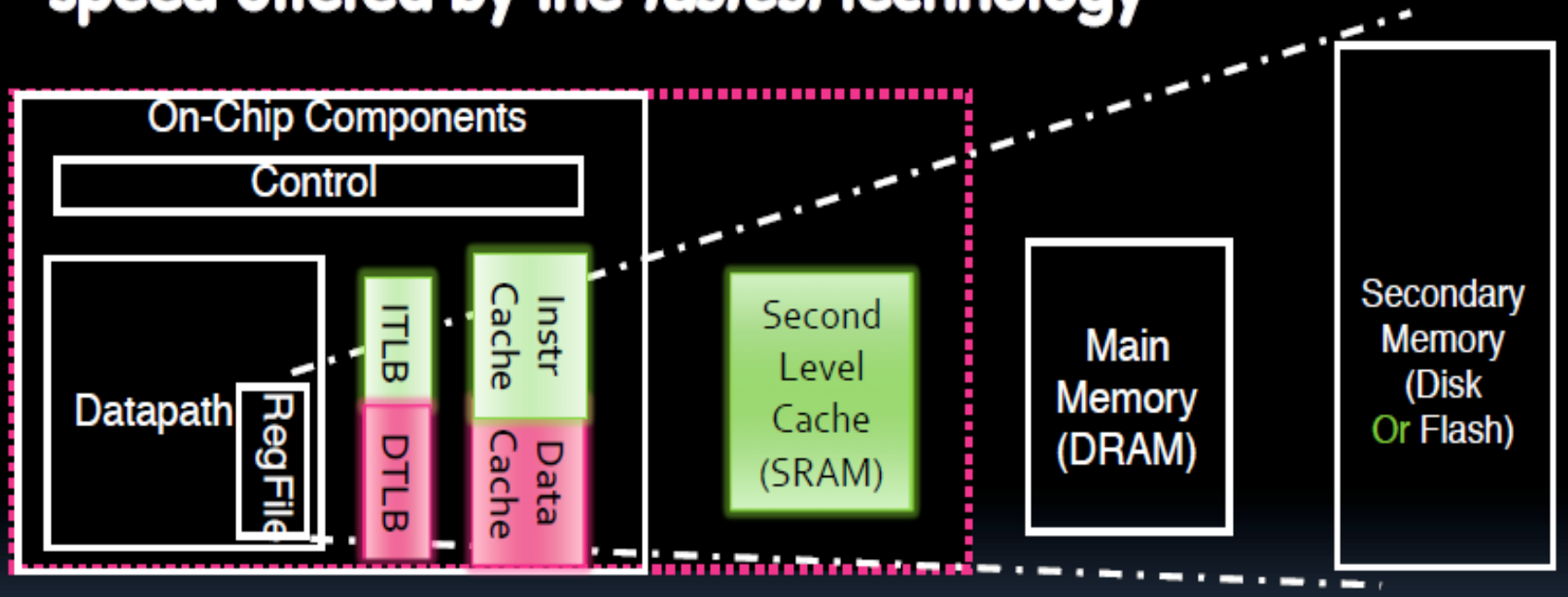
**Cost**

# Levels of the Memory Hierarchy



# Typical Memory Hierarchy

- **The Trick:** present processor with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology




| | | | | | |
|---|---|---|---|---|---|
| **Speed (#cycles):** | ½'s | 1's | 10's | 100's | 10,000's |
| **Size (bytes):** | 100's | 10K's | M's | G's | T's |
| **Cost:** | highest | | | | lowest |

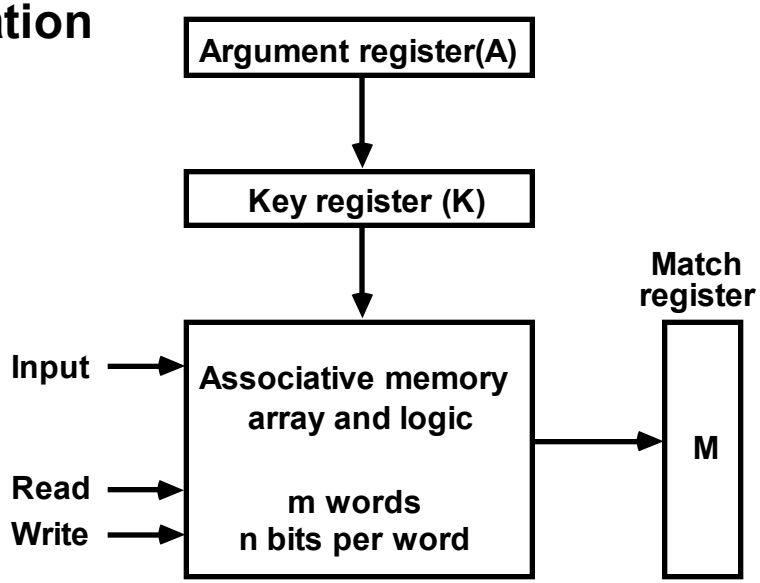
# Memory Hierarchy Analogy: Library

- You're writing a term paper (Processor) at a table in Doe
- Doe Library is equivalent to disk
  - essentially limitless capacity, very slow to retrieve a book
- Table is main memory
  - smaller capacity: means you must return book when table fills up
  - easier and faster to find a book there once you've already retrieved it
- Open books on table are cache
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library

# ASSOCIATIVE MEMORY

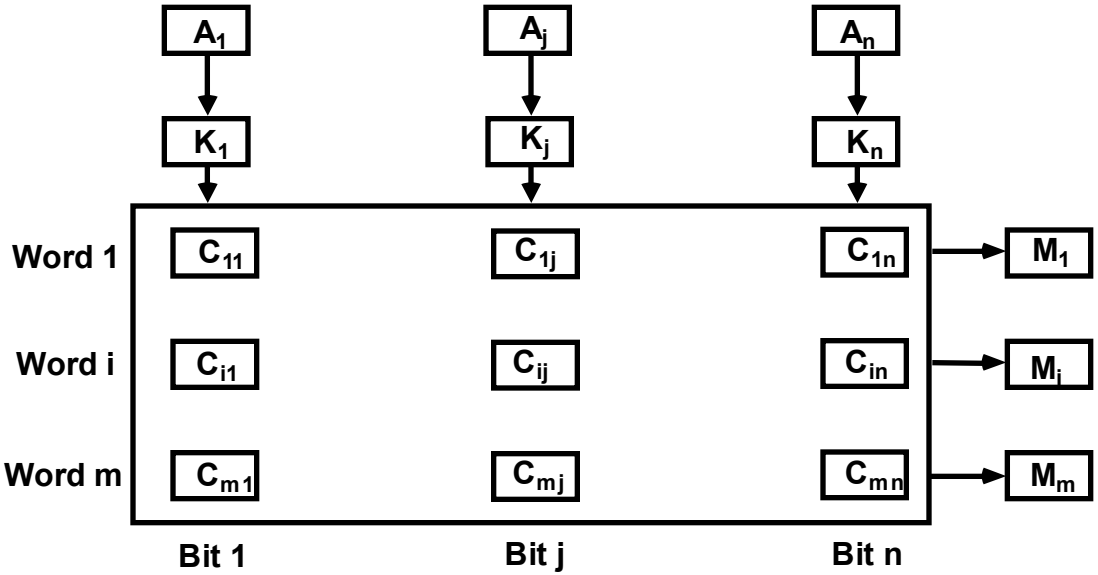
- Accessed by the content of the data rather than by an address
- Also called Content Addressable Memory (CAM)

Hardware Organization

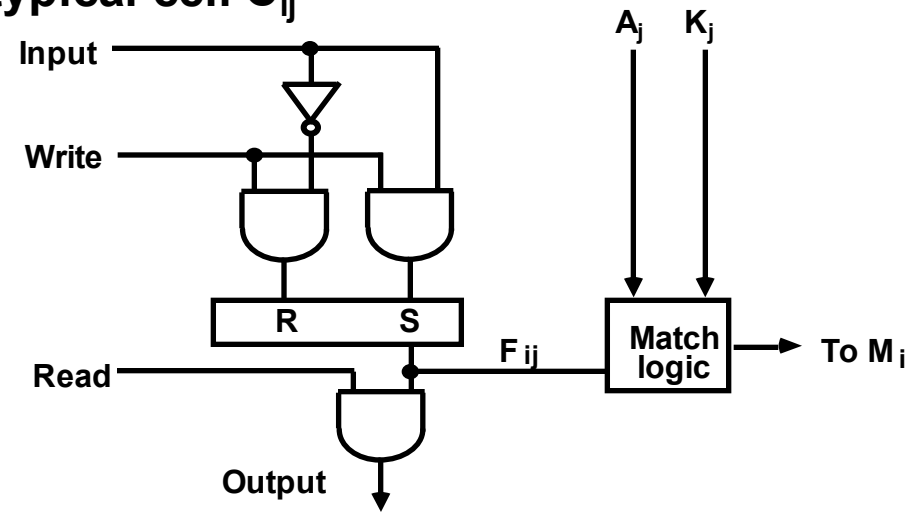



- Compare each word in CAM in parallel with the content of A(Argument Register)
- If CAM Word[i] = A, M(i) = 1
- Read sequentially accessing CAM for CAM Word(i) for M(i) = 1
- K(Key Register) provides a mask for choosing a particular field or key in the argument in A (only those bits in the argument that have 1's in their corresponding position of K are compared)

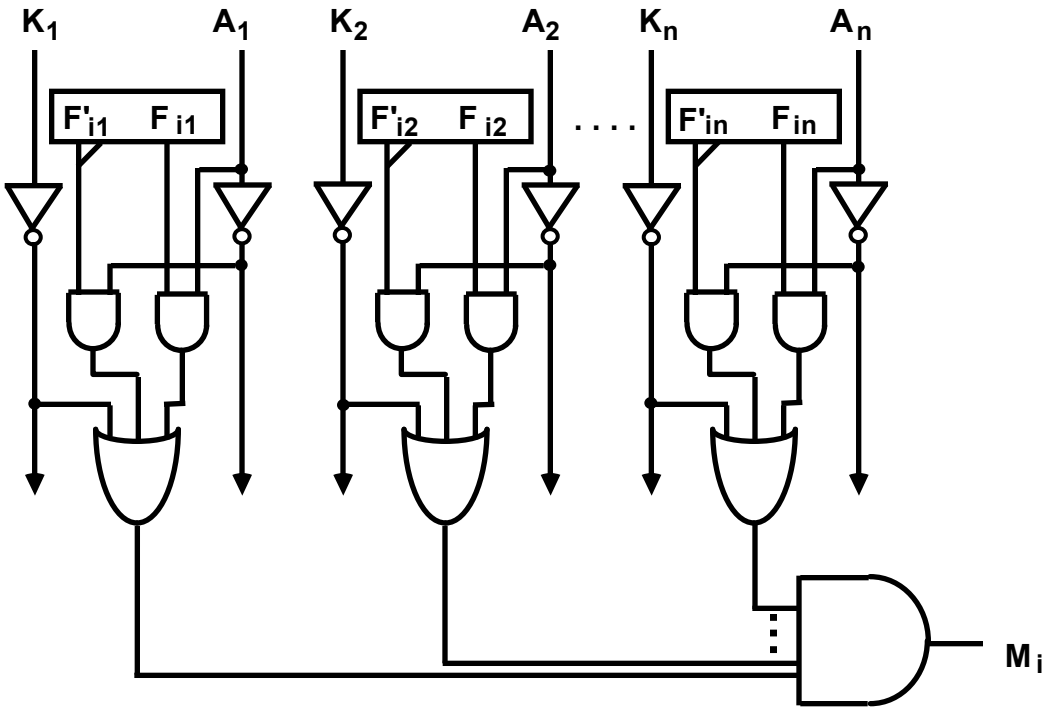
# ORGANIZATION OF CAM




## Internal organization of a typical cell $C_{ij}$



# MATCH LOGIC



# CACHE MEMORY

**Locality of Reference**

- The references to memory at any given time interval tend to be confined within a localized areas
- This area contains a set of information and the membership changes gradually as time goes by

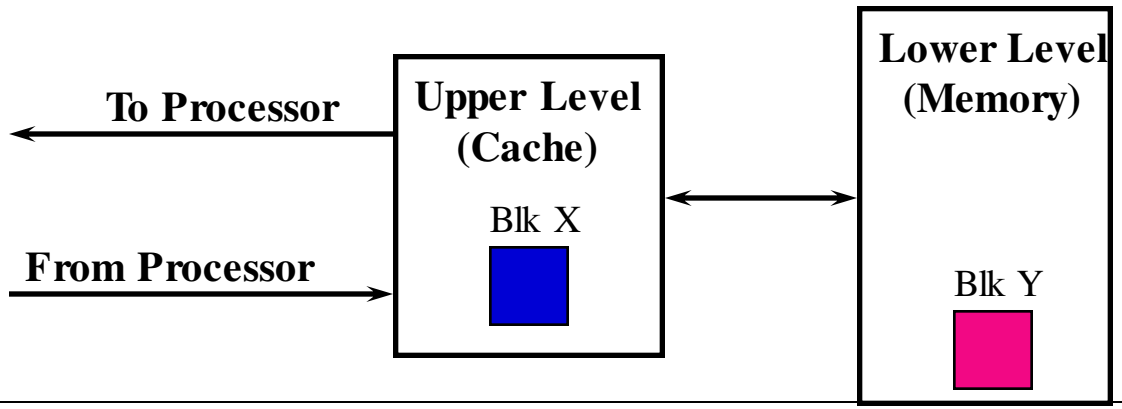
- *Temporal Locality*
  The information which will be used in near future is likely to be in use already
  ( e.g. Reuse of information in loops)

- *Spatial Locality*
  If a word is accessed,
  adjacent(near) words are likely accessed soon

  (e.g. Related data items (arrays) are usually stored together; instructions are executed sequentially)

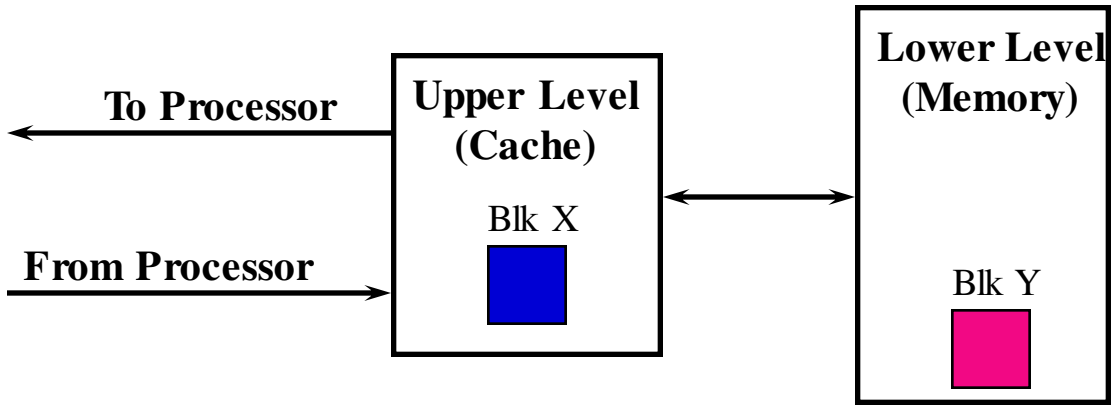
# Terminology

- **Hit:** data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of
    RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty = Time to replace a block in the upper level +
    Time to deliver the block the processor
- Hit Time << Miss Penalty



# Principles of Operation

- At any given time, data is copied between only 2 adjacent levels:
  - Upper Level (Cache) : the one closer to the processor
    - » Smaller, faster, and uses more expensive technology
  - Lower Level (Memory): the one further away from the processor
    - » Bigger, slower, and uses less expensive technology
- Block:
  - The minimum unit of information that can either be present or not present in the two level hierarchy



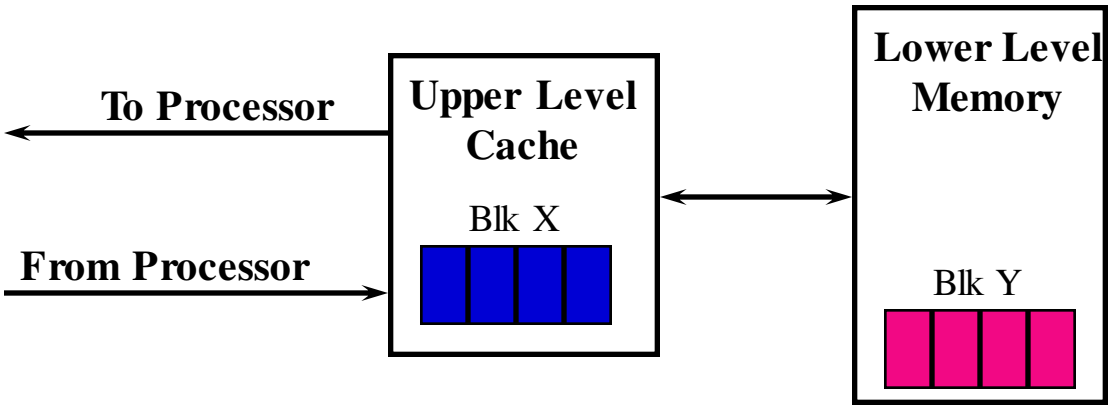
# Typical Values

| | |
|---|---|
| Block (line) size | 4 - 128 bytes |
| Hit time | 1 - 4 cycles |
| Miss penalty | 8 - 32 cycles (and increasing) |
| (access time) | (6-10 cycles) |
| (transfer time) | (2 - 22 cycles) |
| Miss rate | 1% - 20% |
| Cache Size | 1 KB - 256 KB |

# How Does Cache Work?

- **Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.**
  - **Keep more recently accessed data items closer to the processor**

- **Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.**
  - **Move blocks consists of contiguous words to the cache**



# Memory Access with Cache

- Load word instruction: lw $t0, 0($t1)
- $t1 contains $1022_{ten}$, Memory[1022] = 99
- With cache (similar to a hash)
  1. Processor issues address $1022_{ten}$ to Cache
  2. Cache checks to see if has copy of data at address $1022_{ten}$
     - 2a. If finds a match (Hit): cache reads 99, sends to processor
     - 2b. No match (Miss): cache sends address 1022 to Memory
       - I. Memory reads 99 at address $1022_{ten}$
       - II. Memory sends 99 to Cache
       - III. Cache replaces word with new 99
       - IV. Cache sends 99 to processor
  3. Processor loads 99 into register $t1

# PERFORMANCE OF CACHE

## Performance of Cache Memory System

Hit Ratio - % of memory accesses satisfied by Cache memory system

Te: Effective memory access time in Cache memory system

Tc: Cache access time

Tm: Main memory access time

$$Te = Tc*h + (1 - h)\ Tm$$

Example: Tc = 0.4 μs, Tm = 1.2μs, h = 0.85%

$$Te = 0.4*0.85 + (1 - 0.85) * 1.2 = 0.52\mu s$$

# Mapping Function

**Specification of correspondence between main memory blocks and cache blocks**

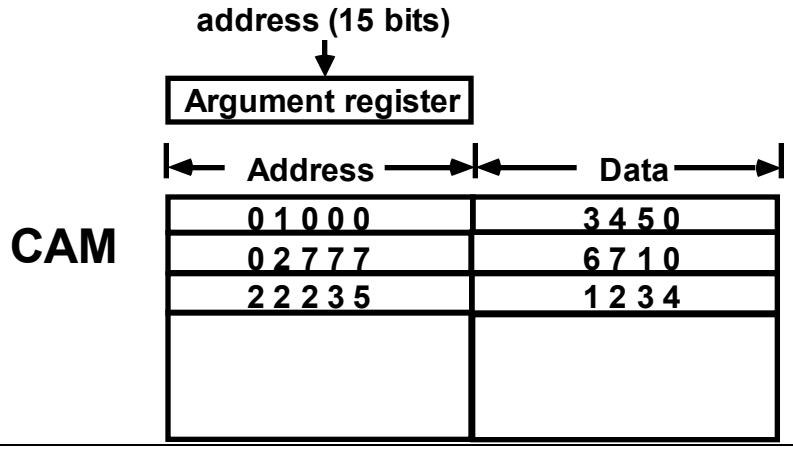
- **Associative mapping**
- **Direct mapping**
- **Set-associative mapping**

# ASSOCIATIVE MAPPING -

- Any block location in Cache can store any block in memory
  -> Most flexible

- Mapping Table is implemented in an associative memory
  -> Fast, very Expensive

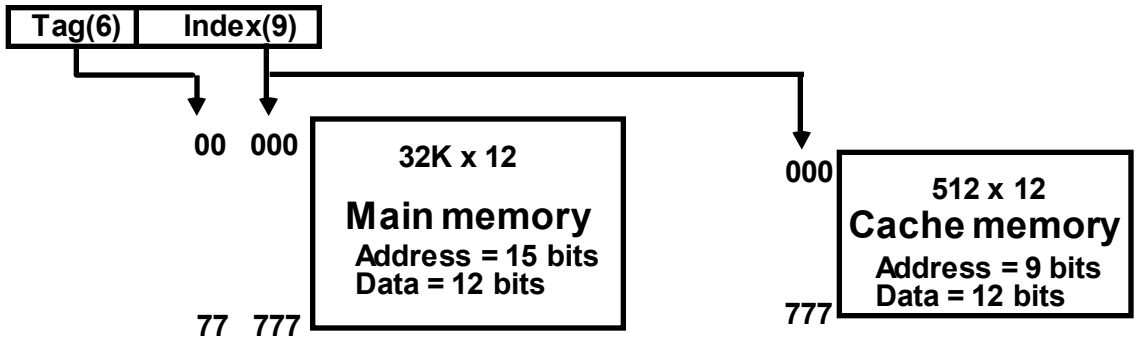
- Mapping Table
  Stores both address and the content of the memory word



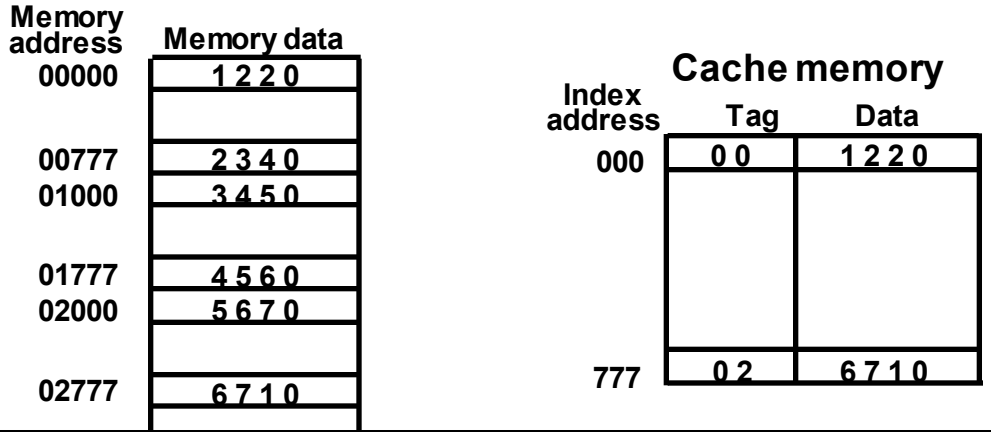
# DIRECT MAPPING -

- Each memory block has only one place to load in Cache
- Mapping Table is made of RAM instead of CAM
- n-bit memory address consists of 2 parts; k bits of Index field and n-k bits of Tag field
- n-bit addresses are used to access main memory and k-bit Index is used to access the Cache

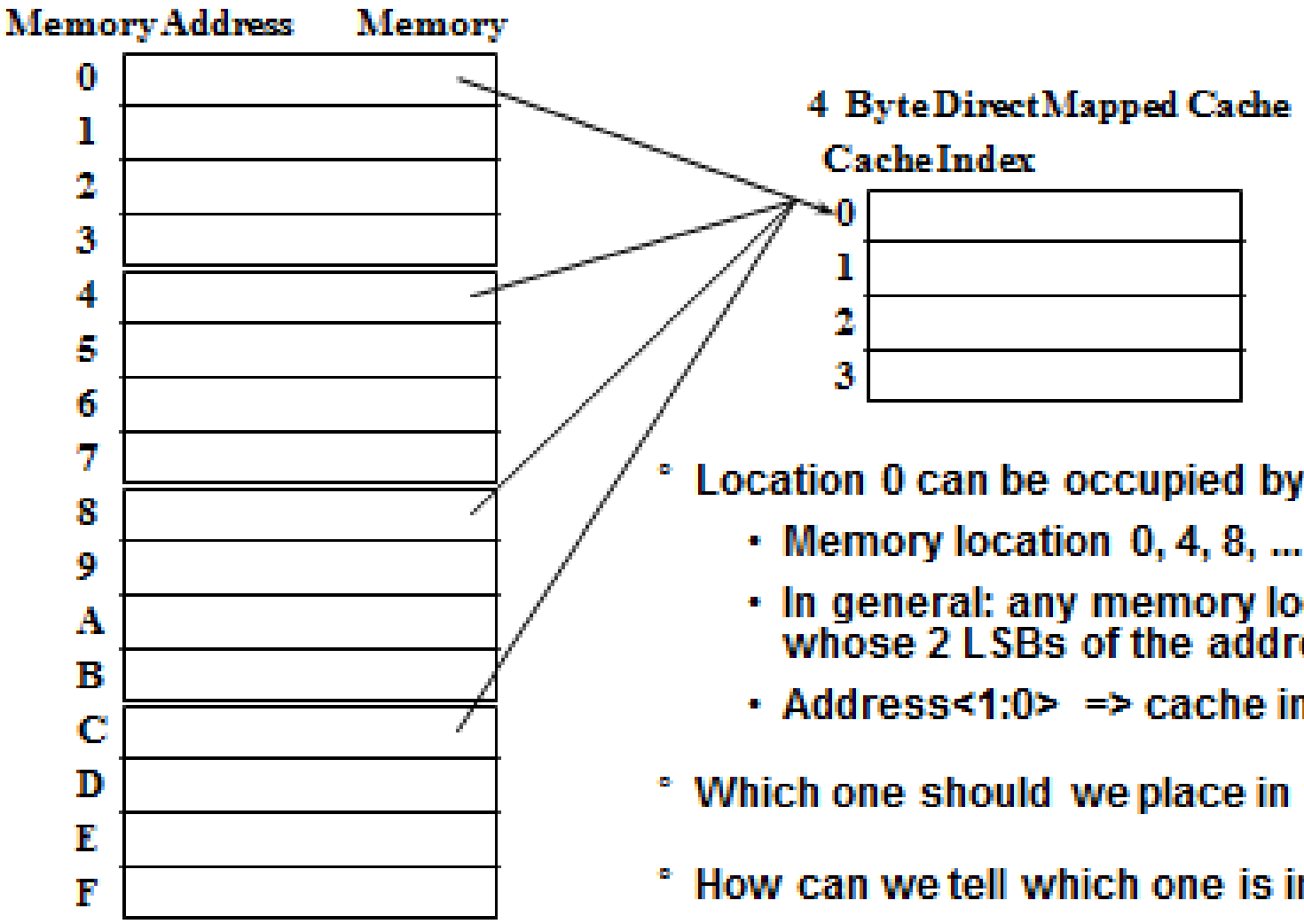
## Addressing Relationships




## Direct Mapping Cache Organization



# The Simplest Cache: Direct Mapped Cache




- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index
- Which one should we place in the cache?
- How can we tell which one is in the cache?

oache.12

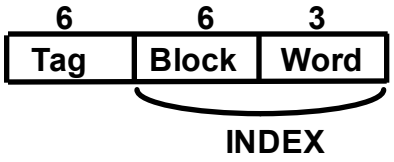
# DIRECT MAPPING

## Operation

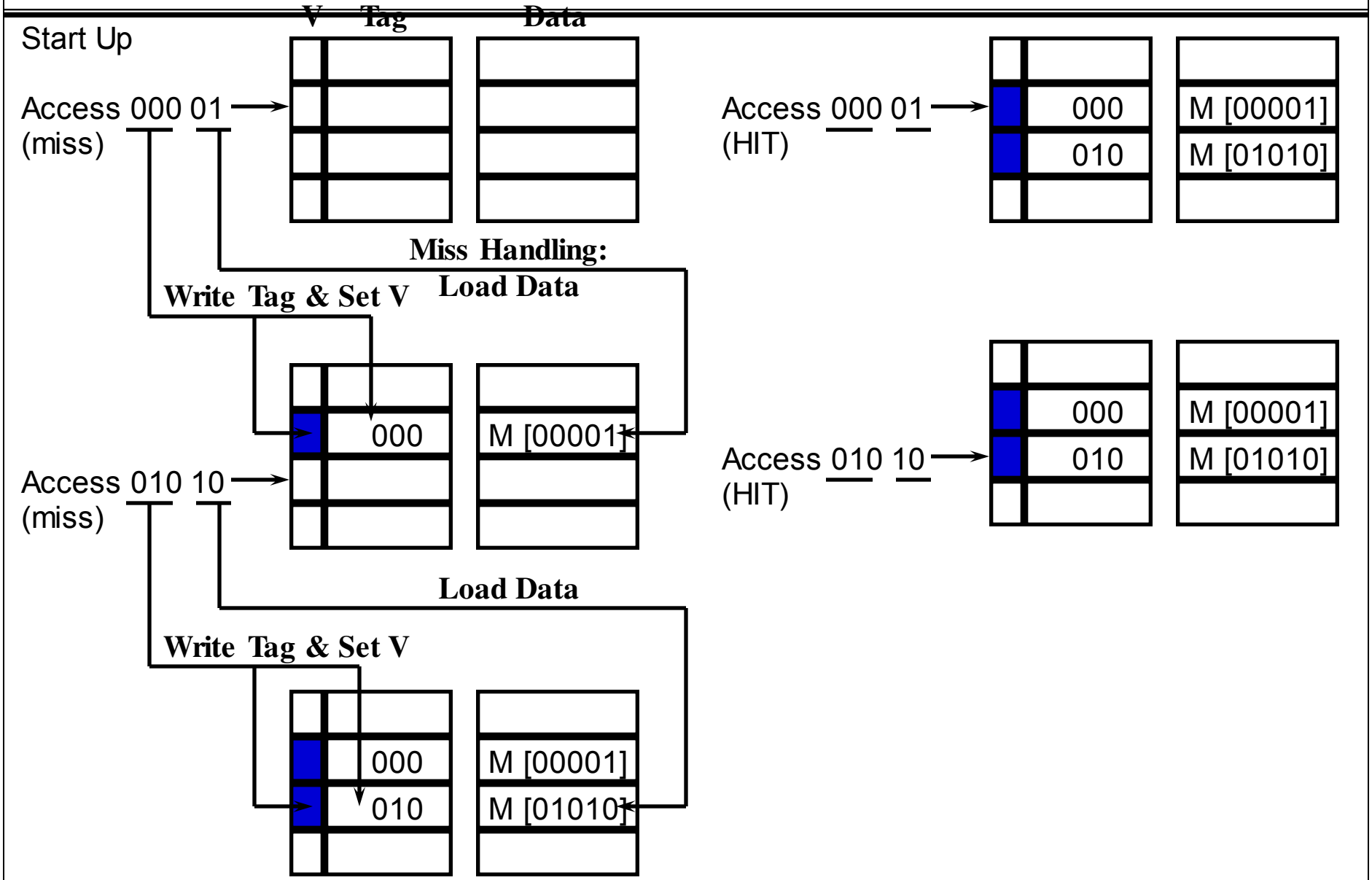
- CPU generates a memory request with (TAG;INDEX)
- Access Cache using INDEX ; (tag; data)
  Compare TAG and tag
- If matches -> Hit
  Provide Cache[INDEX](data) to CPU
- If not match -> Miss
  M[tag;INDEX] <- Cache[INDEX](data)
  Cache[INDEX] <- (TAG;M[TAG; INDEX])
  CPU <- Cache[INDEX](data)

## Direct Mapping with block size of 8 words

| | Index | tag | data |
|---|---|---|---|
| Block 0 | 000<br>007 | 0 1<br>0 1 | 3 4 5 0<br>6 5 7 8 |
| Block 1 | 010<br>017 | | |
| | | | |
| Block 63 | 770<br>777 | 0 2<br>0 2 | 6 7 1 0 |

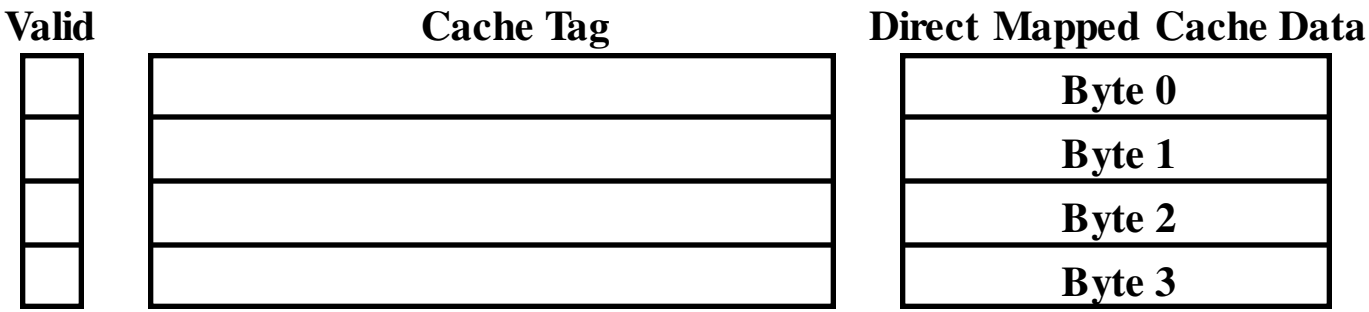


# Cache Access Example



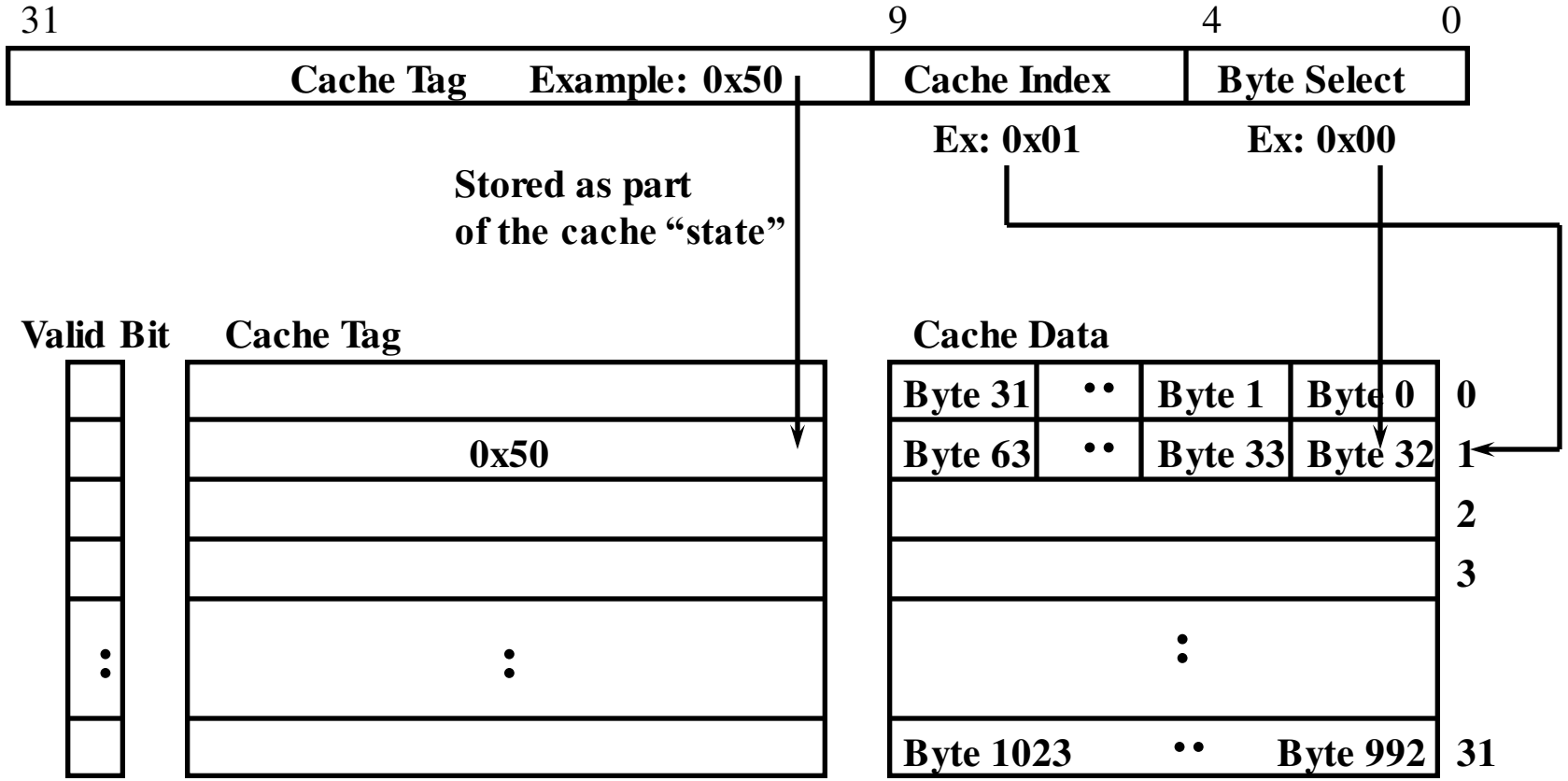
# Definition of a Cache Block

- Cache Block: the cache data that has in its own cache tag
  - 4-byte Direct Mapped Cache: Block Size = 1 Byte
  - Take advantage of Temporal Locality: If a byte is referenced, it will tend to be referenced soon.
  - Did not take advantage of Spatial Locality: If a byte is referenced, its adjacent bytes will be referenced soon.
- In order to take advantage of Spatial Locality: increase the block size



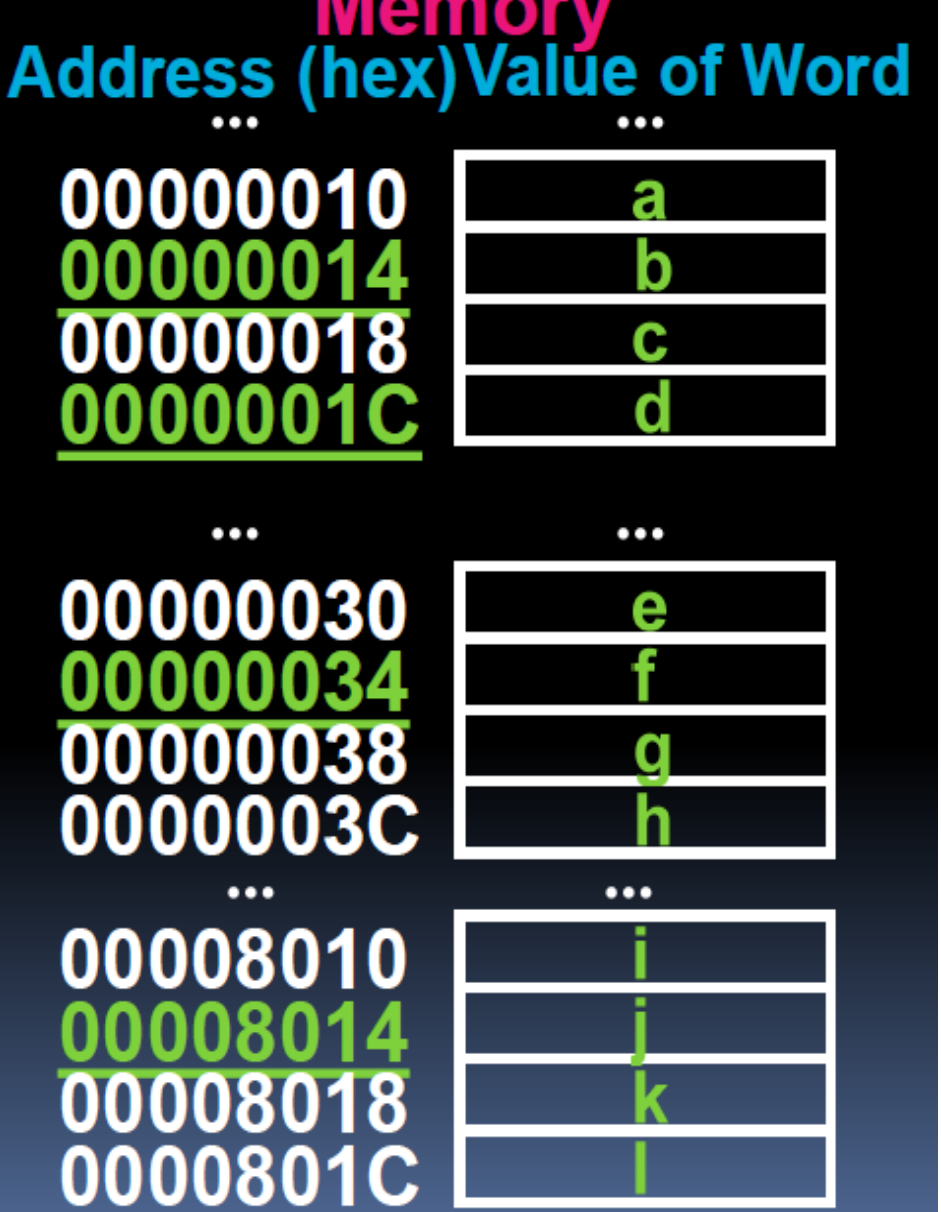
# Ex: 1 KB Direct Mapped Cache with 32 B Blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



# Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
  - Can you work out height, width, area?
- Read 4 addresses
  1. 0x00000014
  2. 0x0000001C
  3. 0x00000034
  4. 0x00008014
- Memory vals here:



# Accessing data in a direct mapped cache

- **4 Addresses:**
  - **0x00000014, 0x0000001C, 0x00000034, 0x00008014**
- **4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields**

**0000000000000000 0000000001 0100**

**0000000000000000 0000000001 1100**

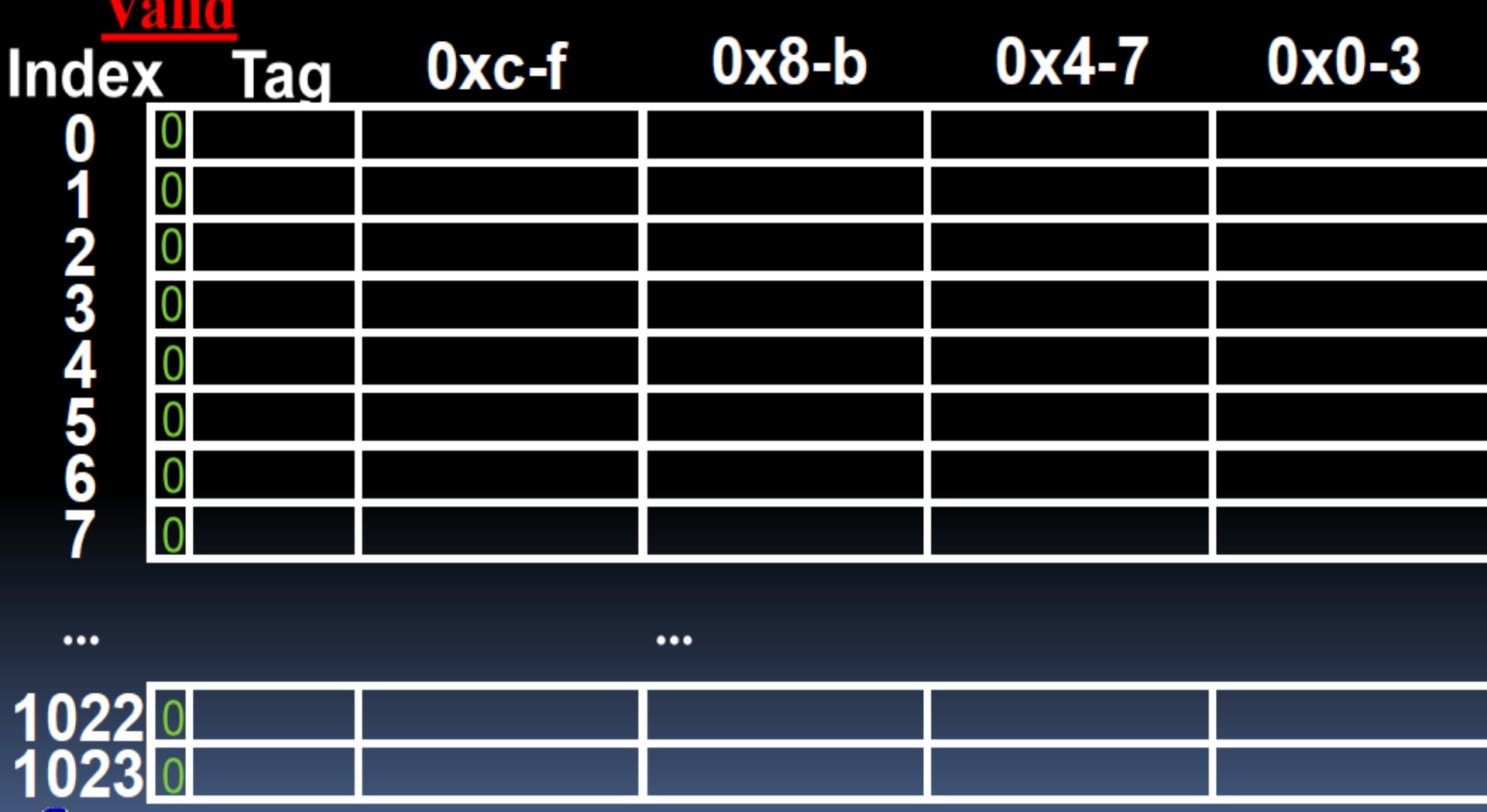
**0000000000000000 0000000011 0100**

**0000000000000010 0000000001 0100**

**Tag Index Offset**

# 16 KB Direct Mapped Cache, 16B blocks

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



# 1. Read 0x00000014

- 0000000000000000 0000000001 0100

| | Tag field | Index field | Offset |
|---|---|---|---|
| Index | Valid Tag 0xc-f | 0x8-b | 0x4-7 | 0x0-3 |
| 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)

- 0000000000000000 **0000000001** 0100

Tag field | Index field | Offset

| Index | Valid | Tag | 0xc-f | 0x8-b | 0x4-7 | 0x0-3 |
|---|---|---|---|---|---|---|
| 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

- 0000000000000000 0000000001 0100

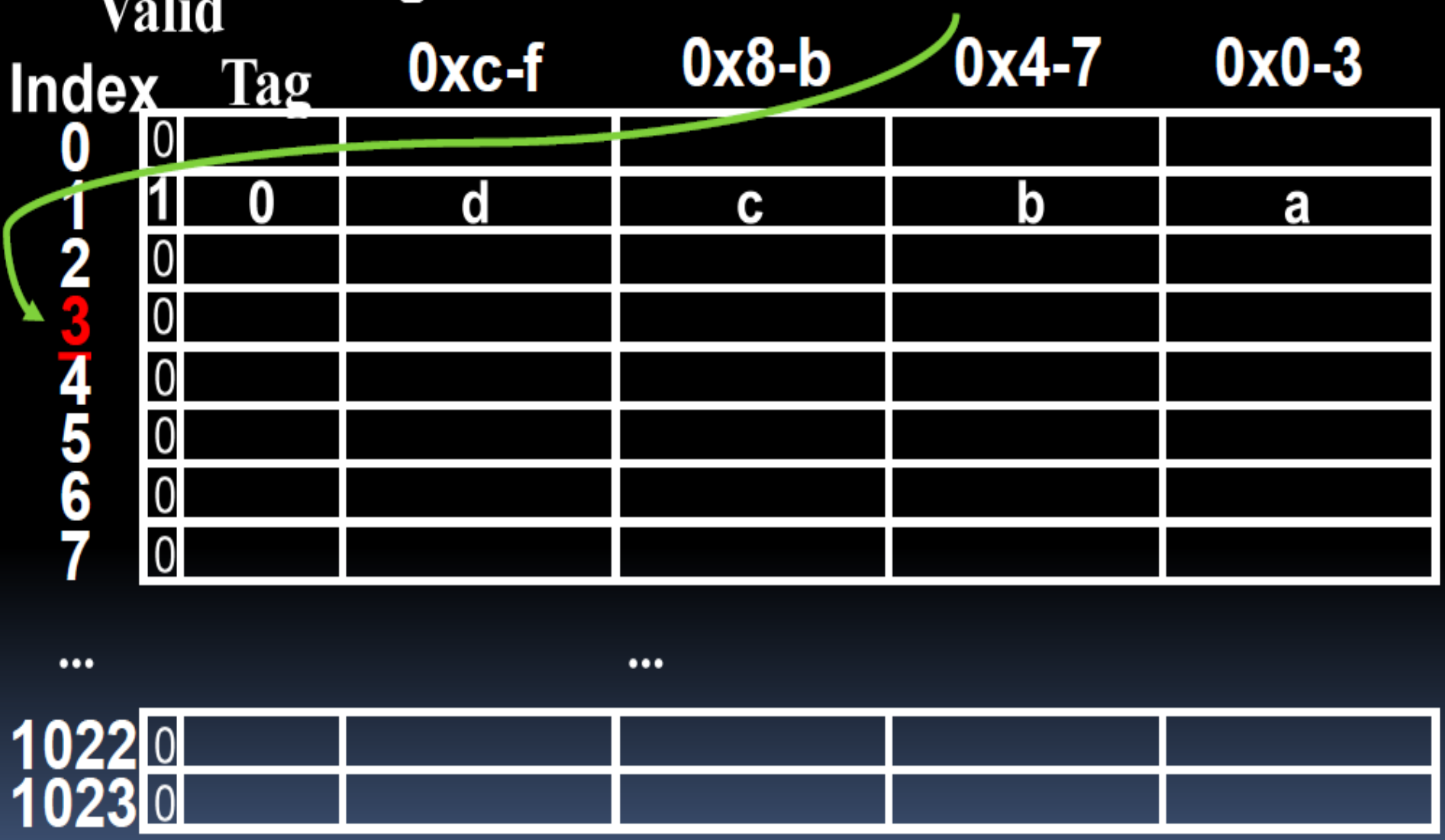
Tag field | Index field | Offset

0xc-f | 0x8-b | 0x4-7 | 0x0-3

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

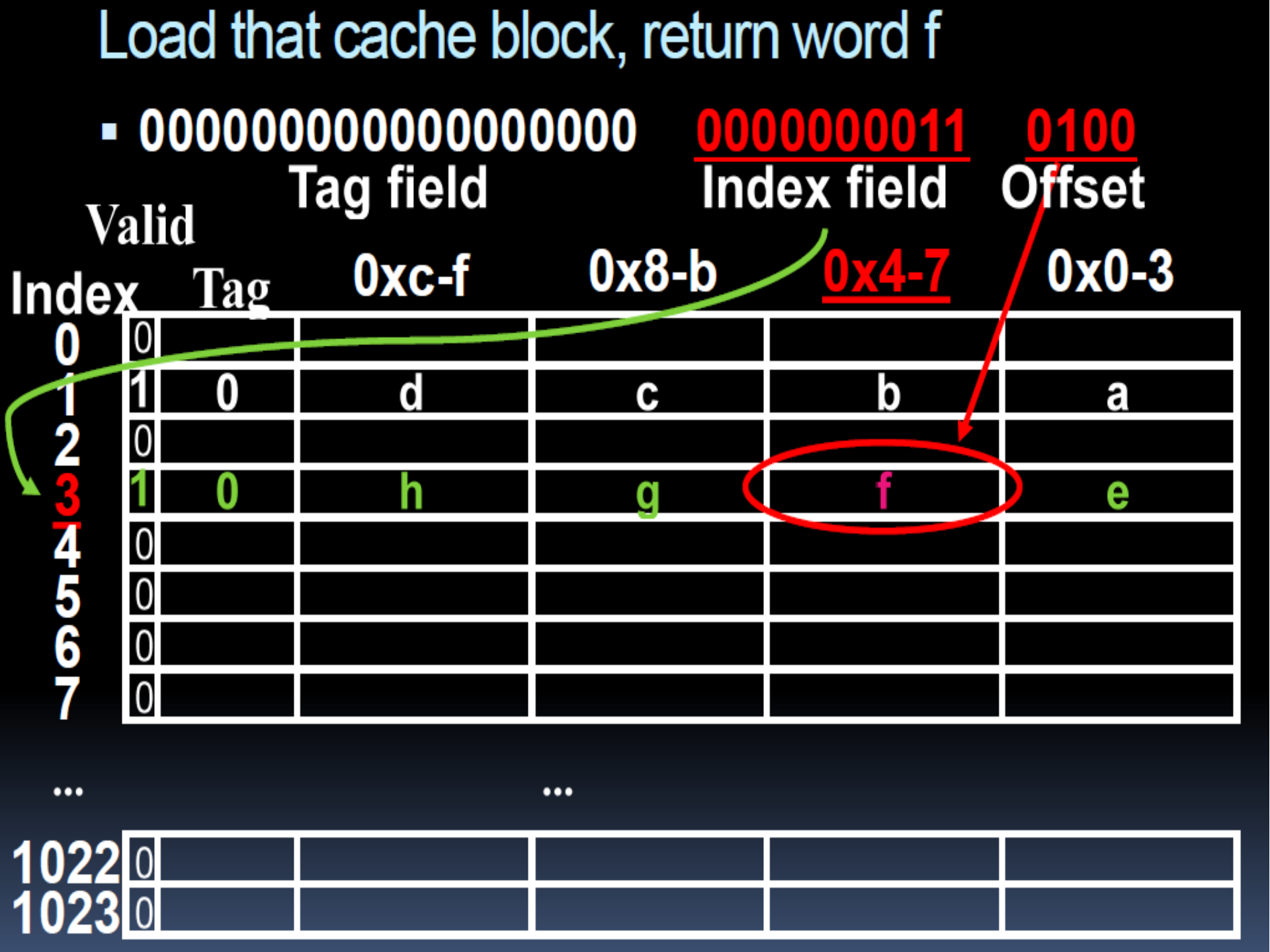
# So read block 3

- 00000000000000000000 **0000000011** 0100
  Tag field Index field Offset



# Load that cache block, return word f

- 00000000000000000000 0000000011 0100



# 4. Read 0x00008014 = 0…10 0..001 0100

- **0000000000000010 0000000001 0100**

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

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

# So read Cache Block 1, Data is Valid

- 0000000000000010 0000000001 0100

Tag field | Index field | Offset

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

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

- 0000000000000010 0000000001 0100

Tag field | Index field | Offset

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

# Miss, so replace block 1 with new data & tag

- 0000000000000010 0000000001 0100

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

# And return word J

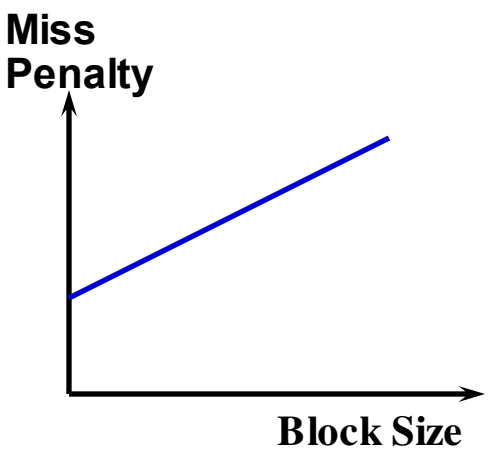
- 0000000000000010 0000000001 0100

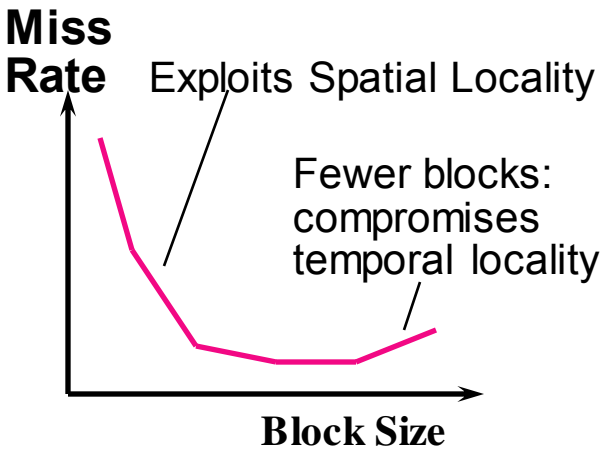
Tag field | Index field | Offset

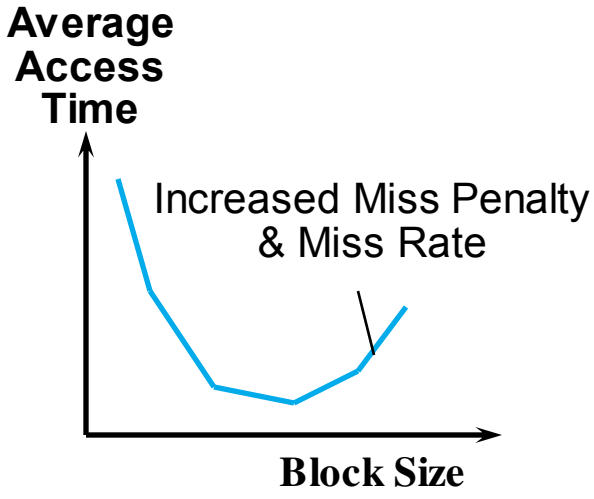
| Index | Valid | Tag | 0xc-f | 0x8-b | 0x4-7 | 0x0-3 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | l | k | j | i |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | h | g | f | e |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0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

# Block Size Tradeoff

- In general, larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - » Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
- Average Access Time:
  - = Hit Time x (1 - Miss Rate) + Miss Penalty x Miss Rate



# SET ASSOCIATIVE MAPPING -

- Each memory block has a set of locations in the Cache to load

Set Associative Mapping Cache with set size of two

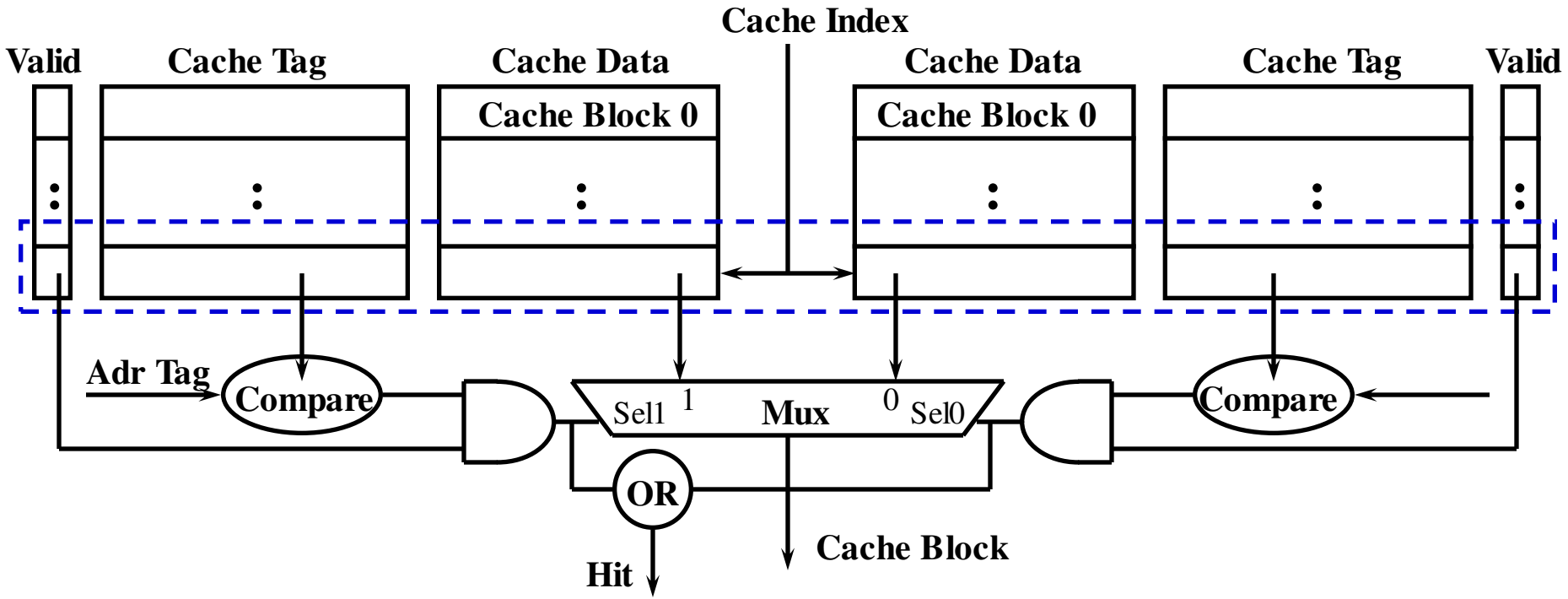
| Index | Tag | Data | Tag | Data |
|---|---|---|---|---|
| 000 | 0 1 | 3 4 5 0 | 0 2 | 5 6 7 0 |
| | | | | |
| 777 | 0 2 | 6 7 1 0 | 0 0 | 2 3 4 0 |

Operation

- CPU generates a memory address(TAG; INDEX)
- Access Cache with INDEX, (Cache word = (tag 0, data 0); (tag 1, data 1))
- Compare TAG and tag 0 and then tag 1
- If tag i = TAG -> Hit, CPU <- data i
- If tag i ≠ TAG -> Miss,
  Replace either (tag 0, data 0) or (tag 1, data 1),
  Assume (tag 0, data 0) is selected for replacement,
  M[tag 0, INDEX] <- Cache[INDEX](data 0)
  Cache[INDEX](tag 0, data 0) <- (TAG, M[TAG,INDEX]),
  CPU <- Cache[INDEX](data 0)

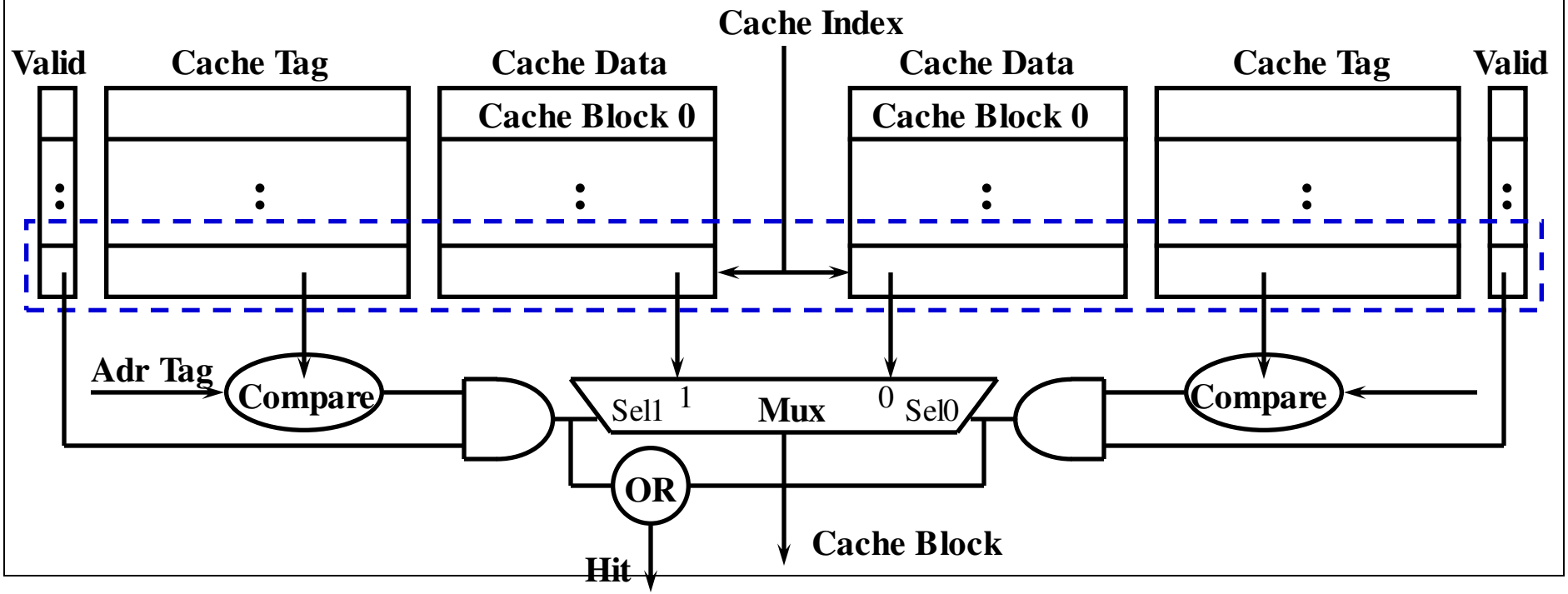
# A Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



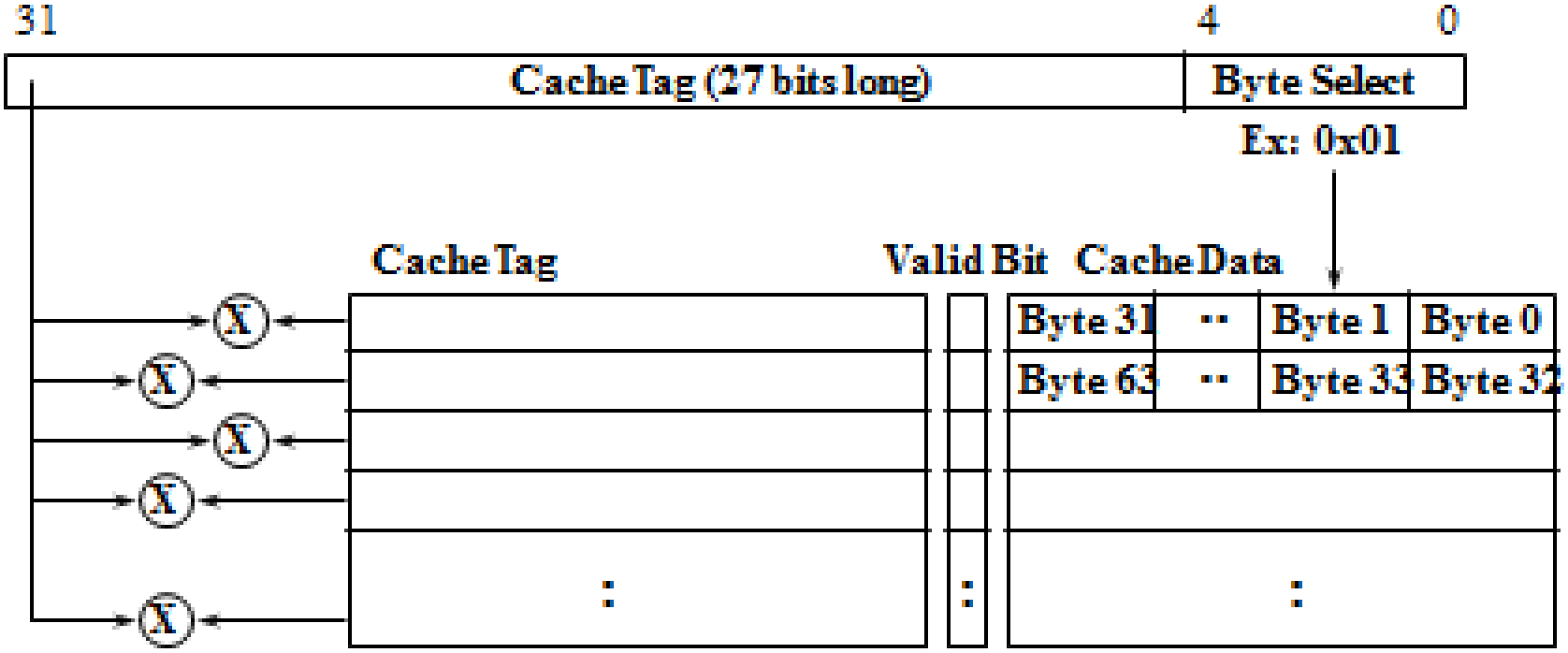
# Disadvantage of Set Associative Cache

- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue.



# And yet Another Extreme Example: Fully Associative

- Fully Associative Cache -- push the set associative idea to its limit!
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 2 B blocks, we need N 27-bit comparators
- By definition: Conflict Miss = 0 for a fully associative cache



# The Need to Make a Decision!

- Direct Mapped Cache:
  - Each memory location can only mapped to 1 cache location
  - No need to make any decision :-)
    - Current item replaced the previous item in that cache location
- N-way Set Associative Cache:
  - Each memory location have a choice of N cache locations
- Fully Associative Cache:
  - Each memory location can be placed in ANY cache location
- Cache miss in a N-way Set Associative or Fully Associative Cache:
  - Bring in new block from memory
  - Throw out a cache block to make room for the new block
  - Damn! We need to make a decision on which block to throw out!

# 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

# LRU(Least Recently Used)

Cache word = (tag 0, data 0, *U0*);(tag 1, data 1, *U1*), Ui = 0 or 1(binary)

Implementation of LRU in the Set Associative Mapping with set size = 2

Modifications

Initially all U0 = U1 = 1

When Hit to (tag 0, data 0, U0), U1 <- 1(least recently used)
(When Hit to (tag 1, data 1, U1), U0 <- 1(least recently used))

When Miss, find the least recently used one(Ui=1)

If U0 = 1, and U1 = 0, then replace (tag 0, data 0)
M[tag 0, INDEX] <- Cache[INDEX](data 0)
Cache[INDEX](tag 0, data 0, U0) <- (TAG,M[TAG,INDEX], 0); U1 <- 1
If U0 = 0, and U1 = 1, then replace (tag 1, data 1)
Similar to above; U0 <- 1

If U0 = U1 = 0, this condition does not exist
If U0 = U1 = 1, Both of them are candidates,
Take arbitrary selection

# CACHE WRITE

## Write Through

When writing into memory

If Hit, both Cache and memory is written in parallel
If Miss, Memory is written
For a read miss, missing block may be overloaded onto a cache block

Memory is always updated
-> Important when CPU and DMA I/O are both executing

Slow, due to the memory access time

## Write-Back (Copy-Back)

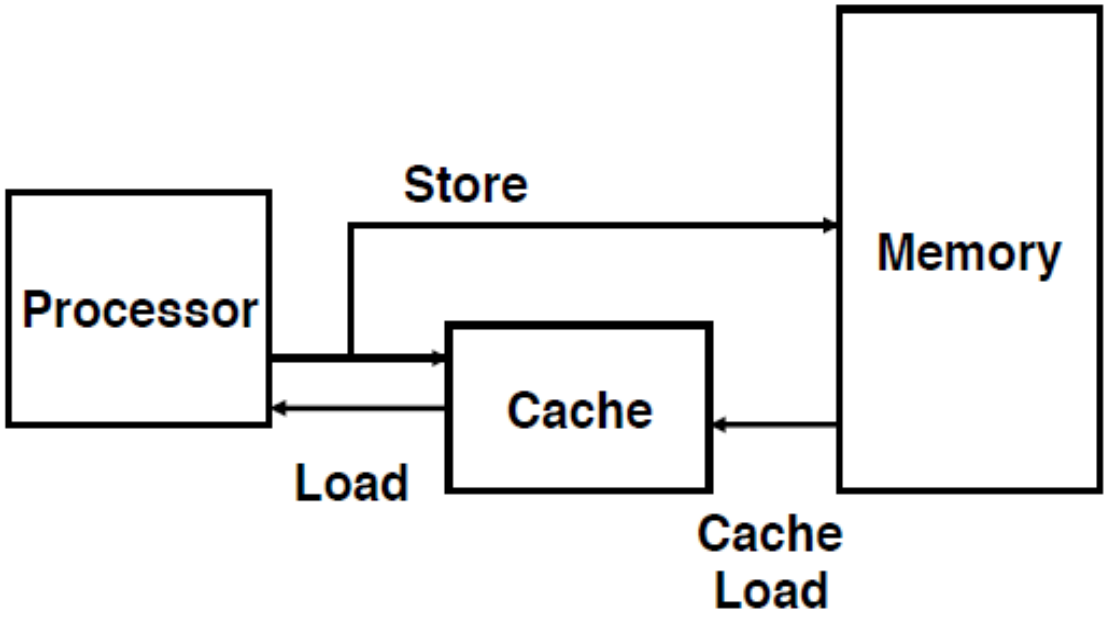
When writing into memory

If Hit, only Cache is written
If Miss, missing block is brought to Cache and write into Cache
For a read miss, candidate block must be written back to the memory

Memory is not up-to-date, i.e., the same item in Cache and memory may have different value

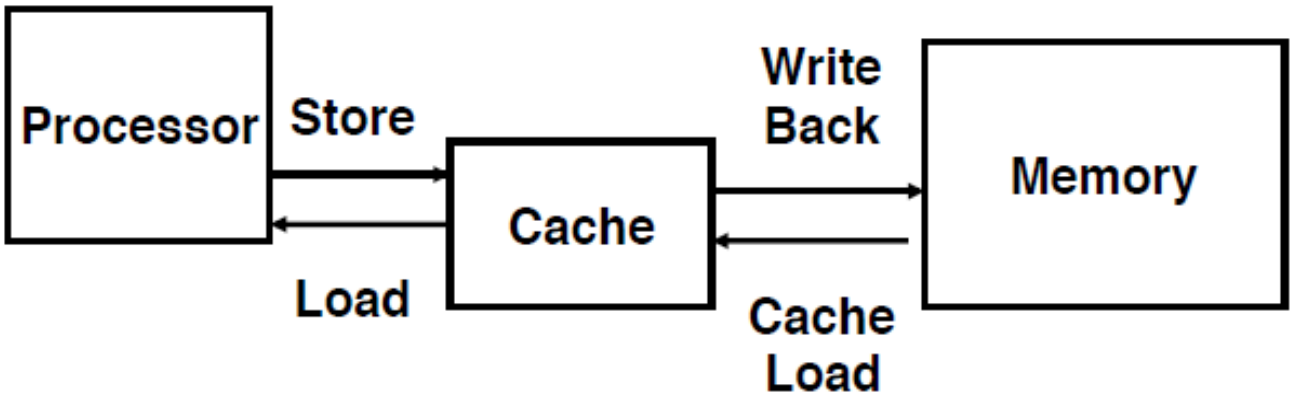
# Write Through

- 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

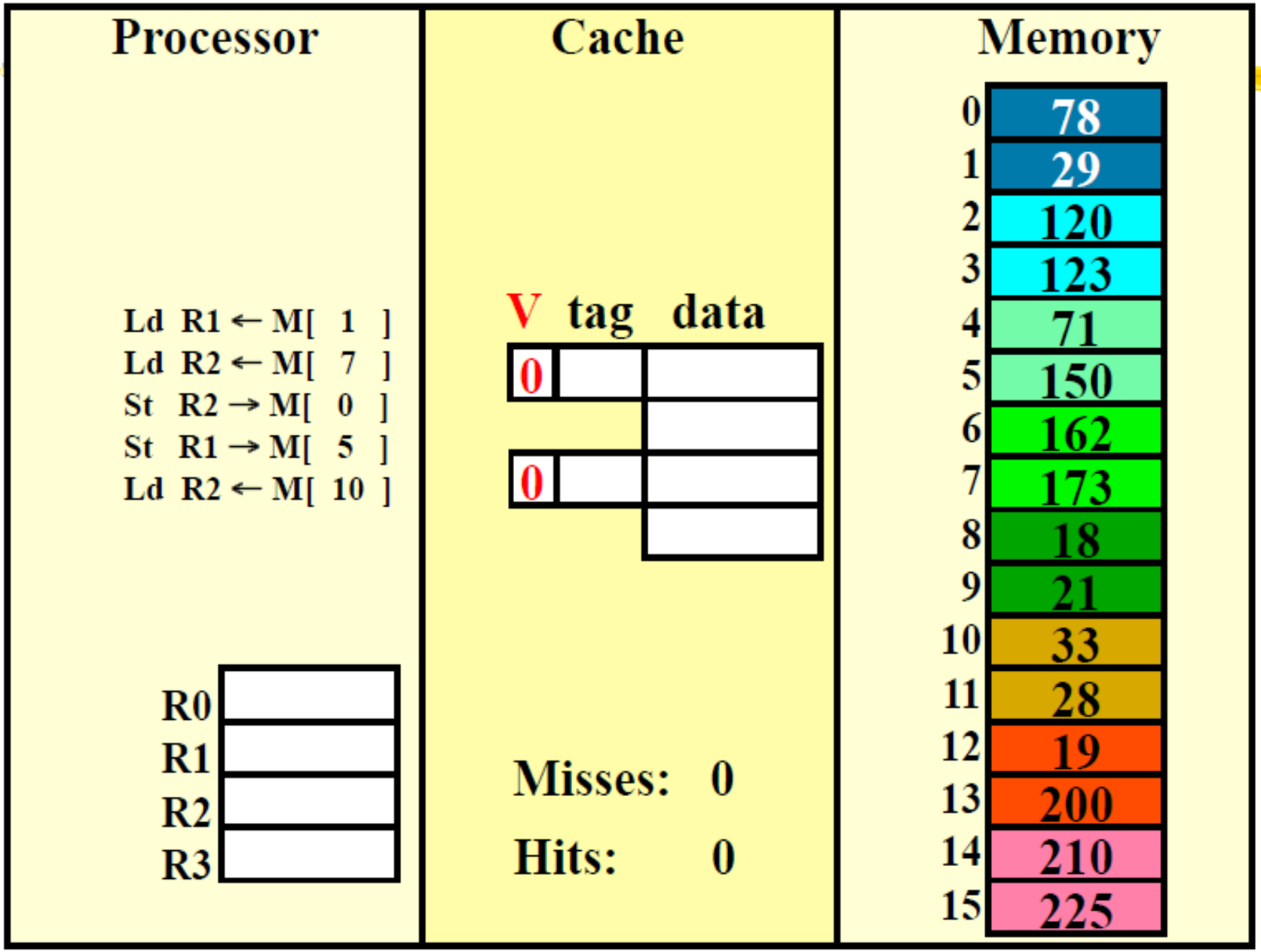


# Write Back

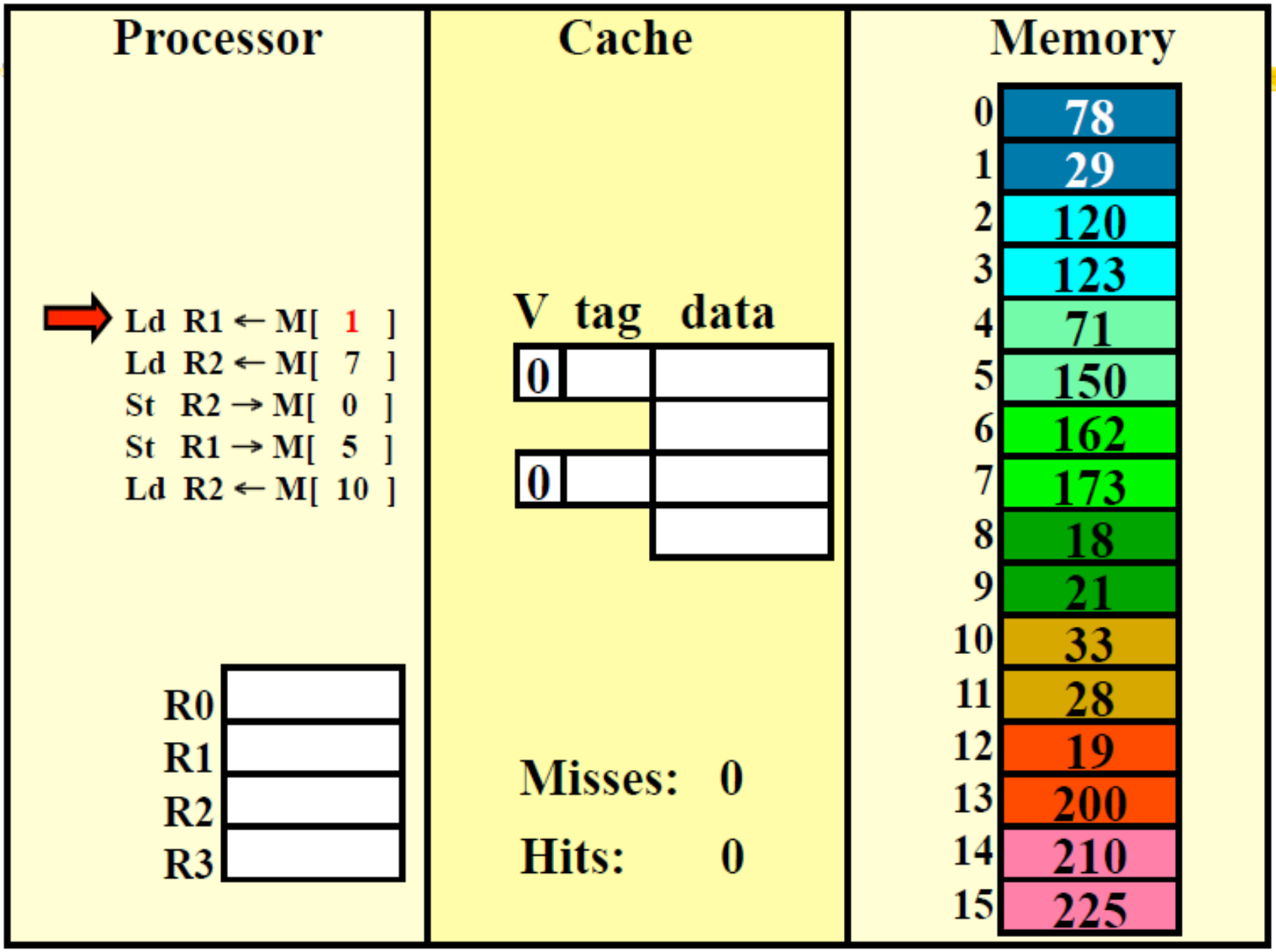
- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires “dirty bit” for each line
    - Set when line in cache is modified
    - Indicates that line in memory is stale
- Memory not always consistent with cache
- No writes of repeated writes



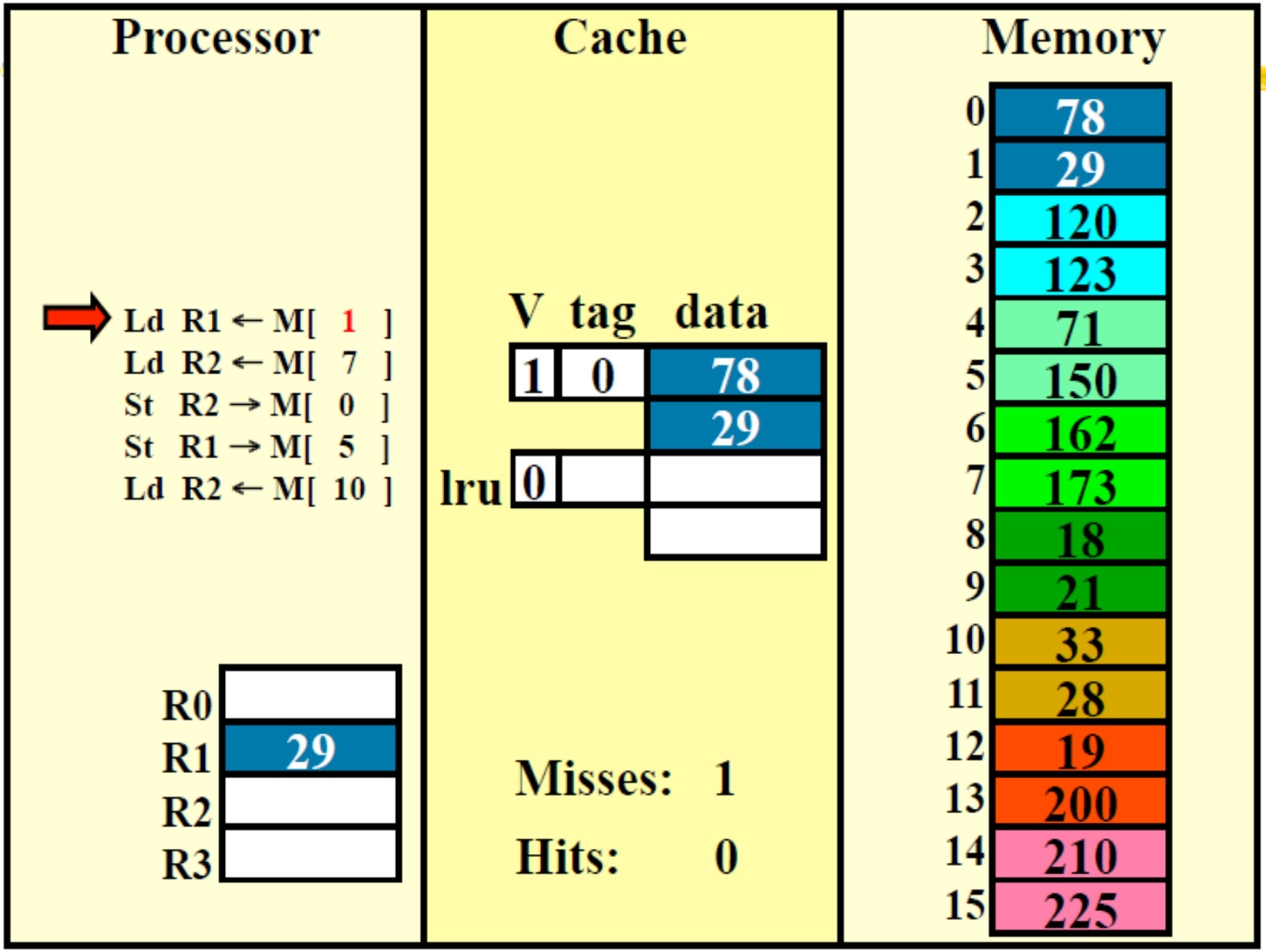
# Handling stores (write-through)



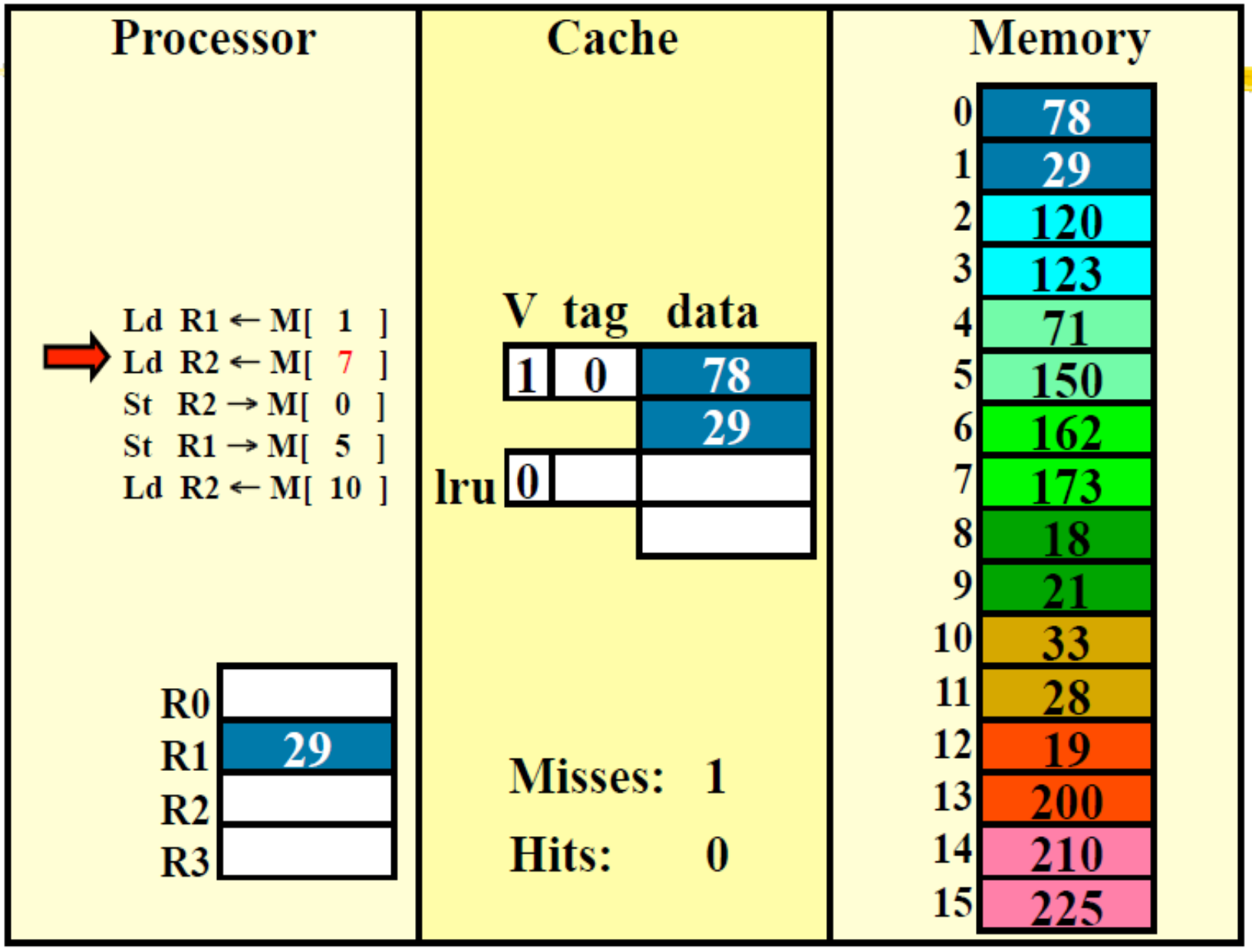
# write-through (REF 1)



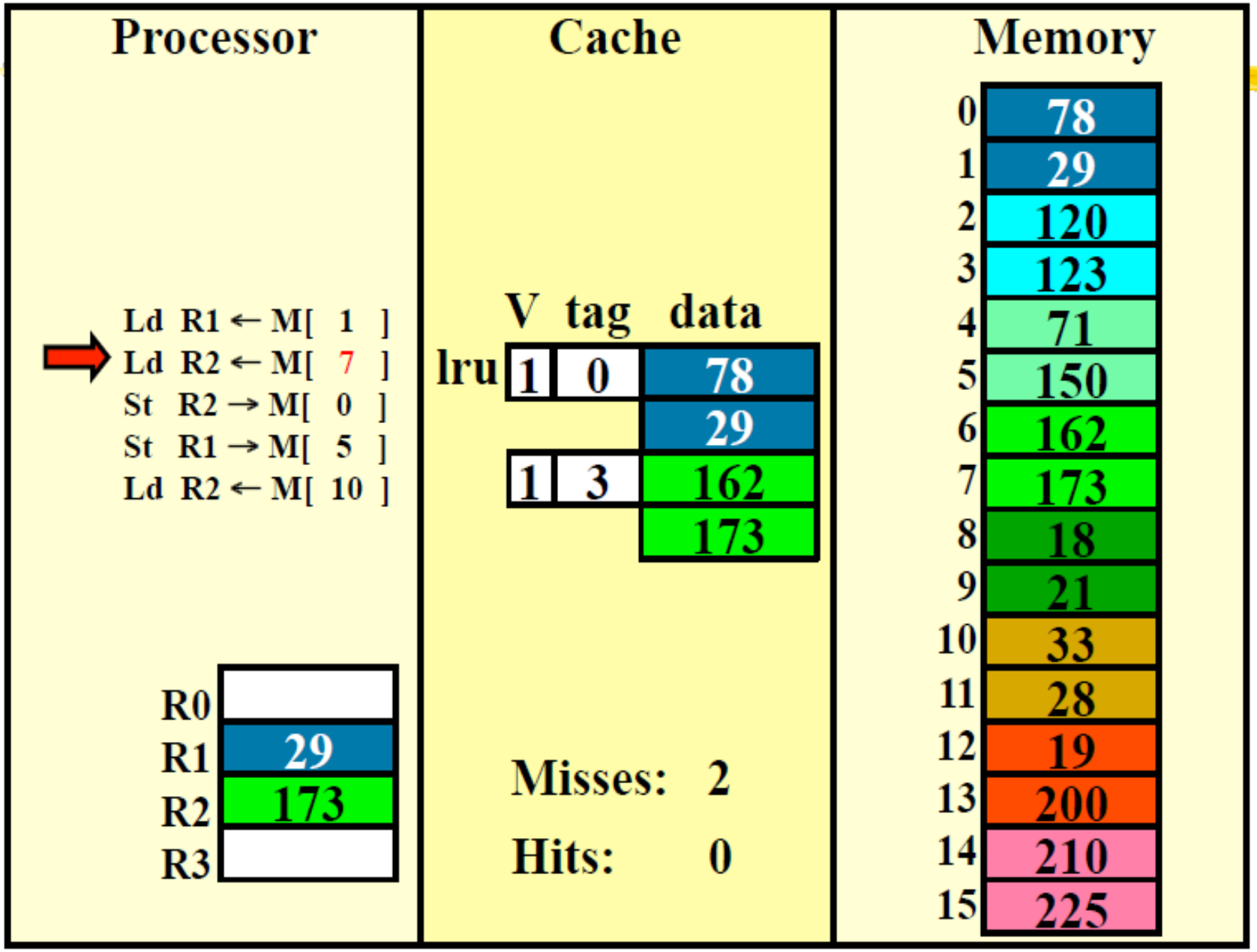
# write-through (REF 1)



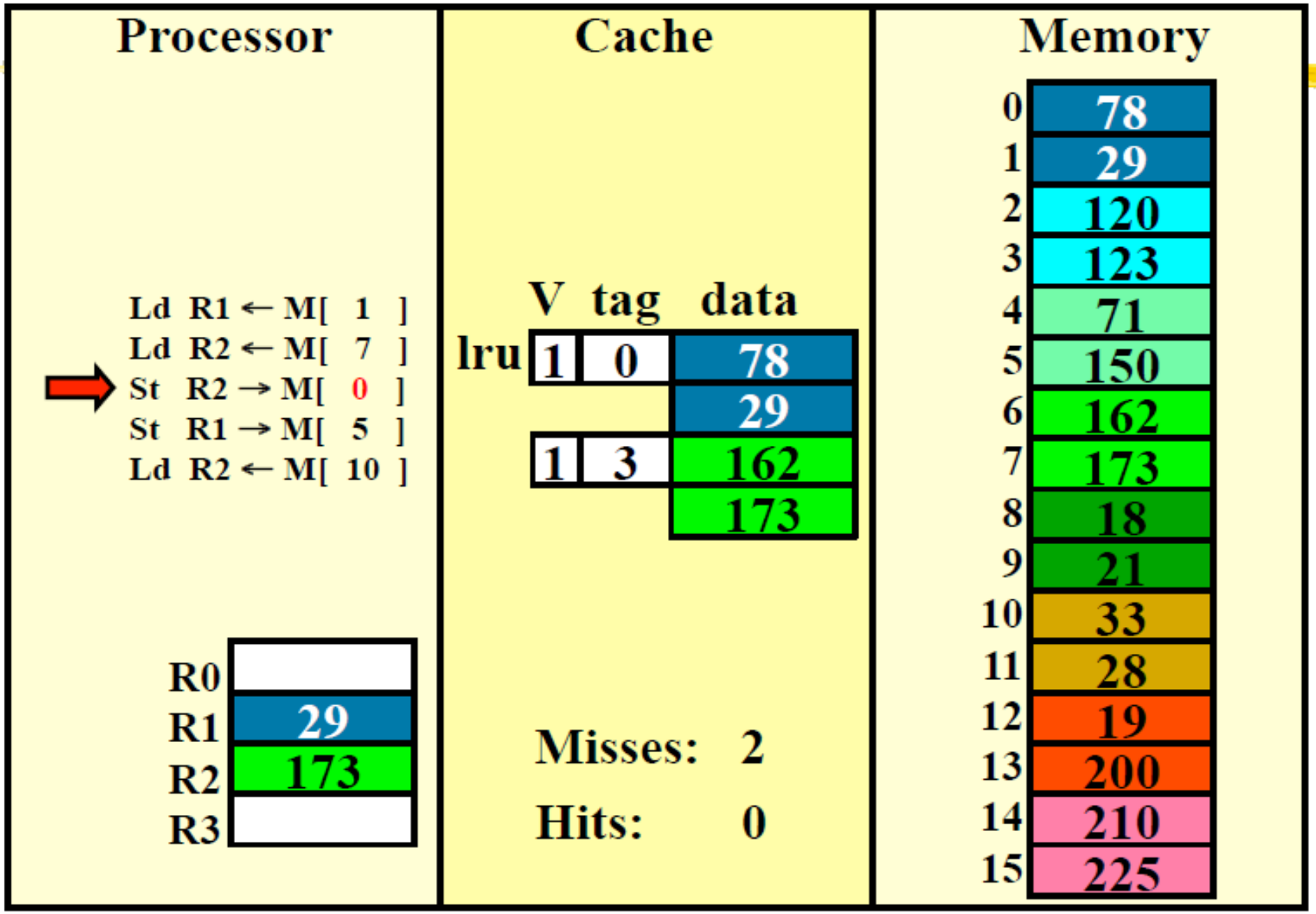
# write-through (REF 2)



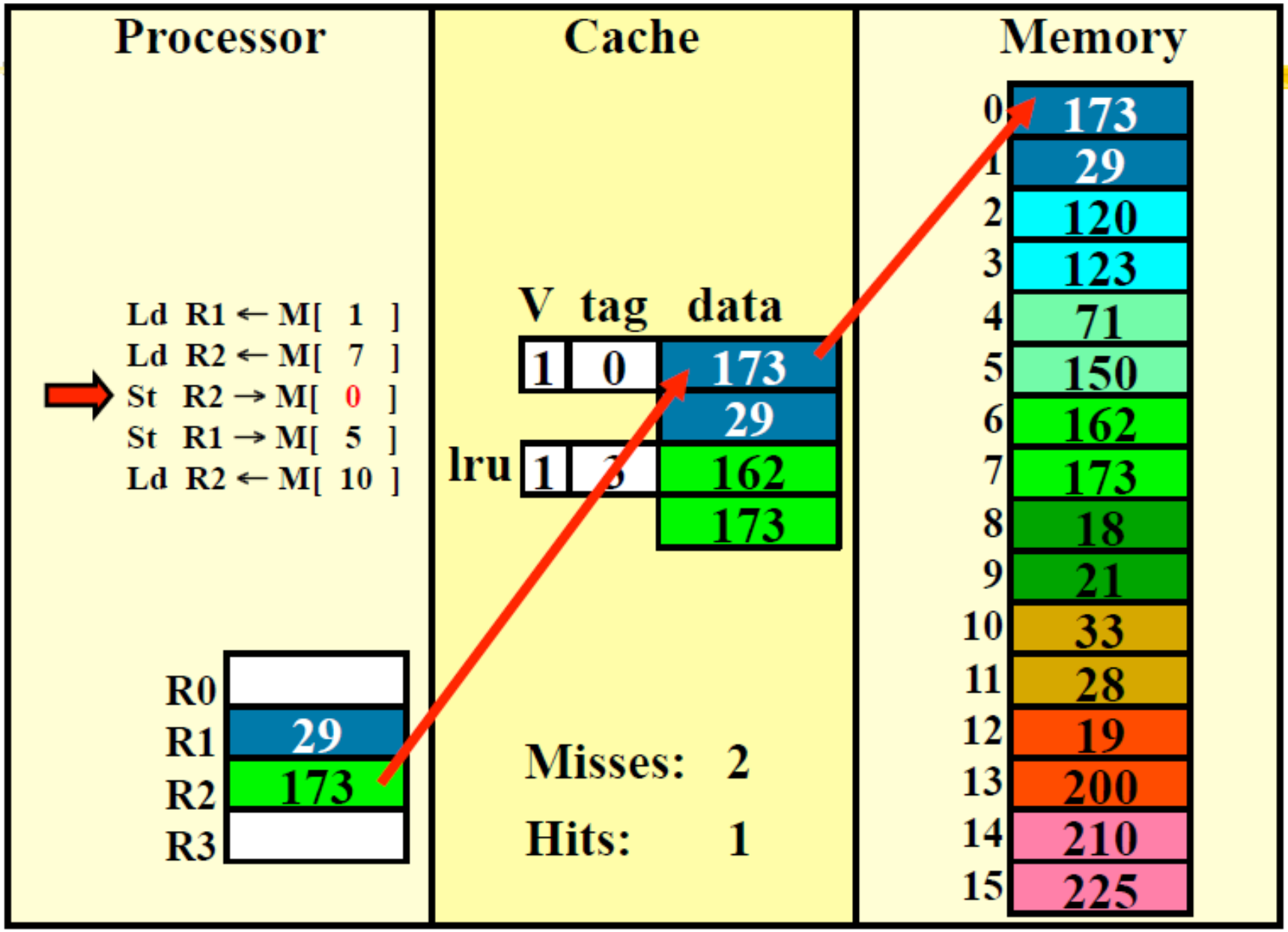
# write-through (REF 2)



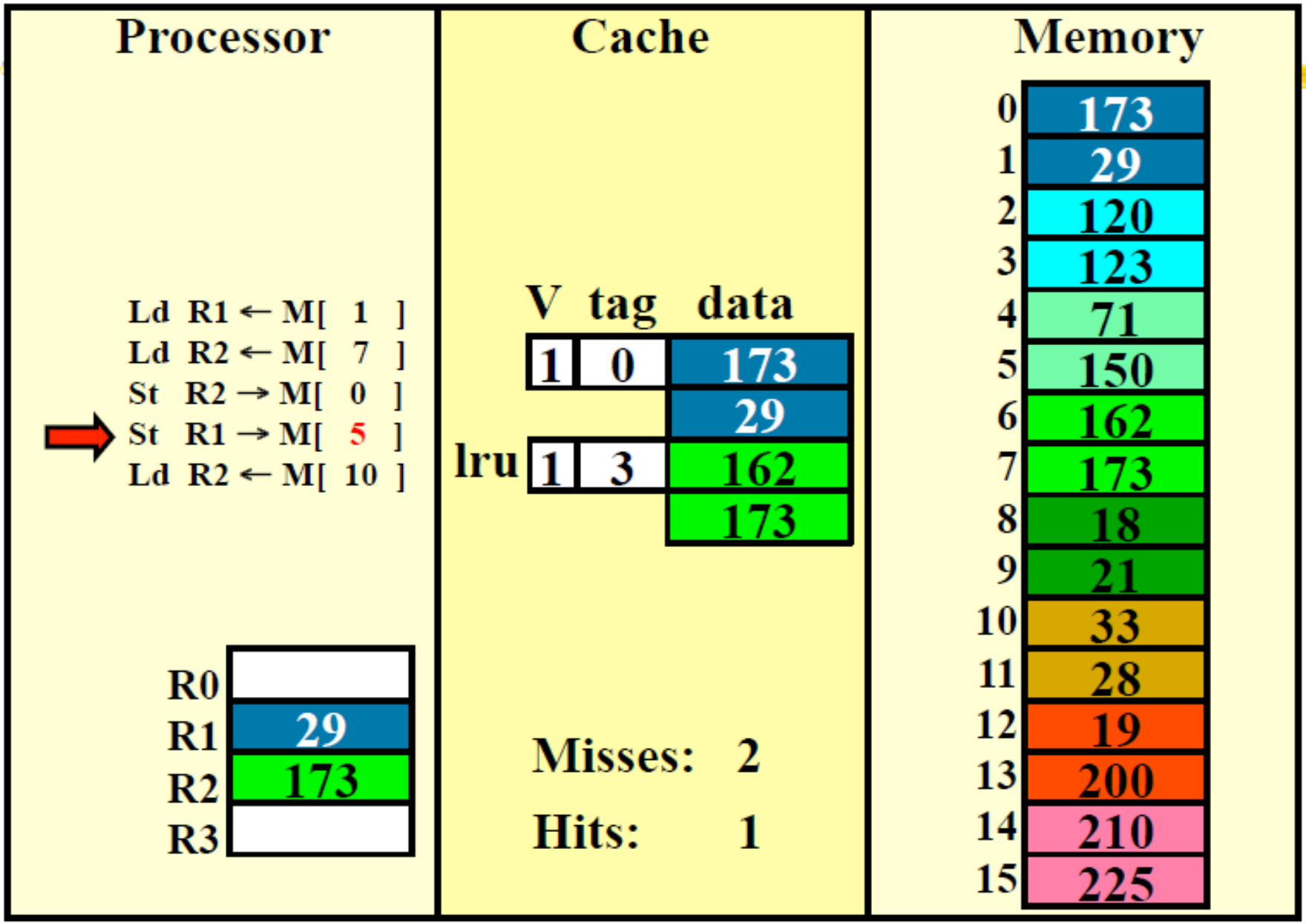
# write-through (REF 3)



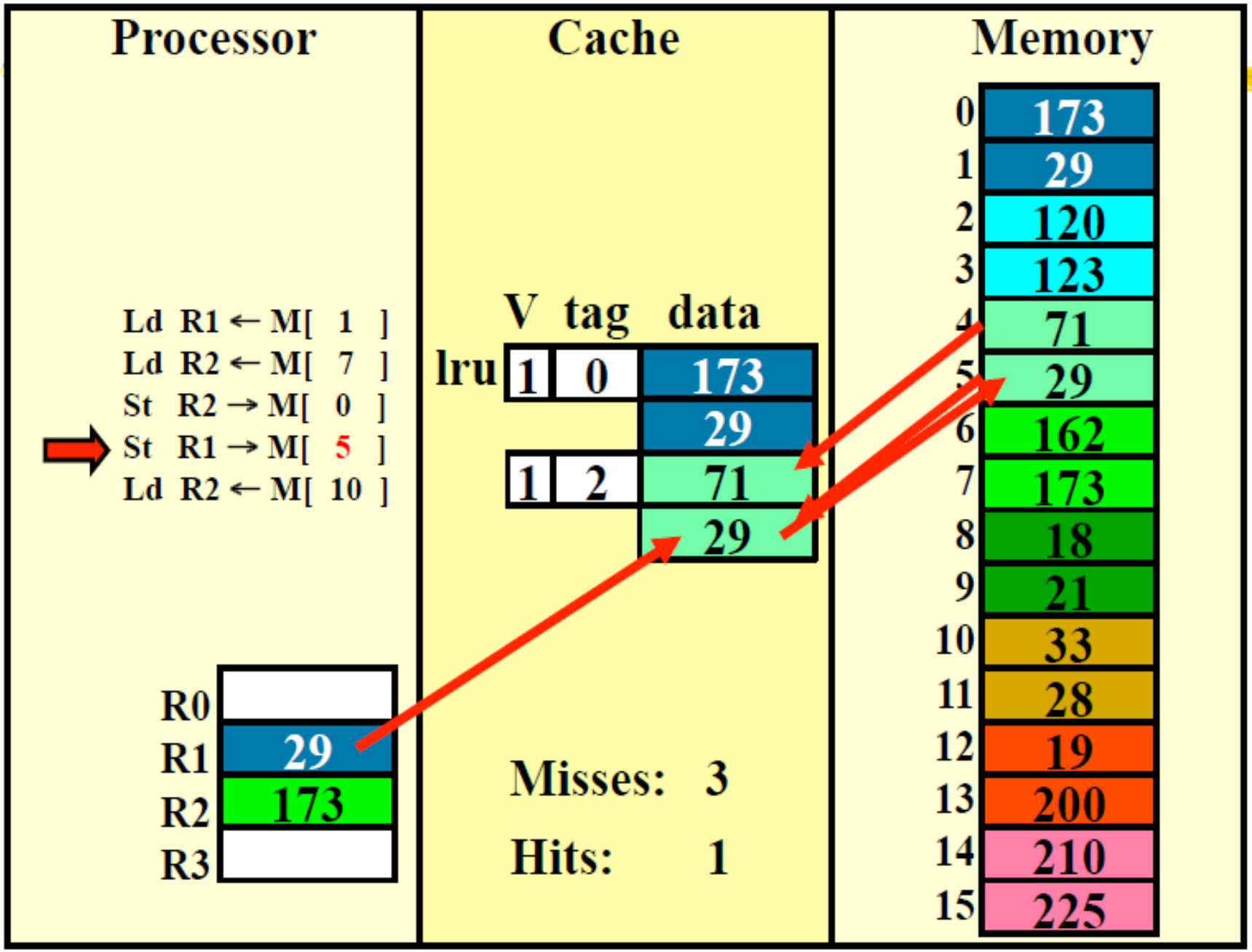
# write-through (REF 3)



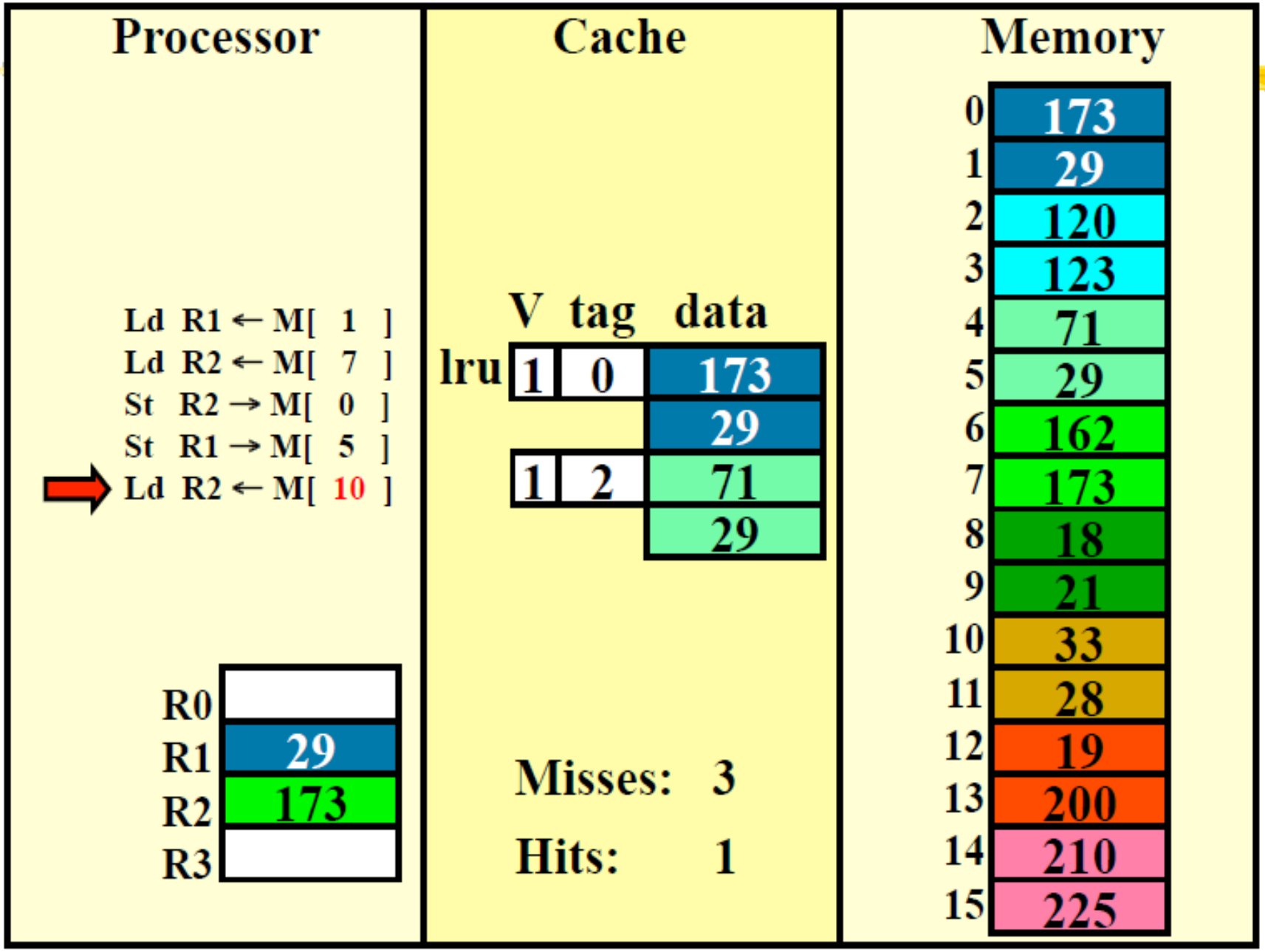
# write-through (REF 4)



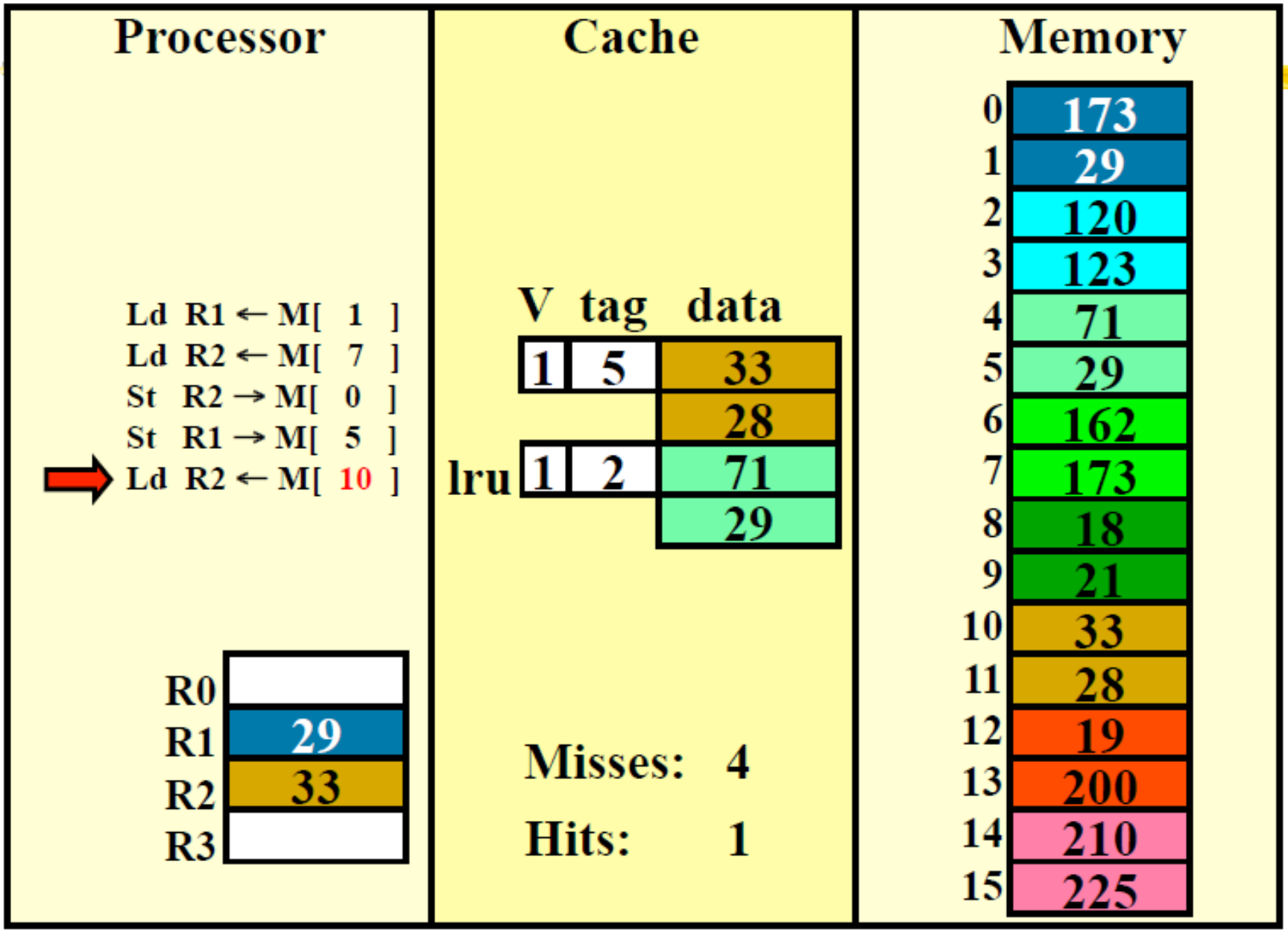
# write-through (REF 4)



# write-through (REF 5)



# write-through (REF 5)

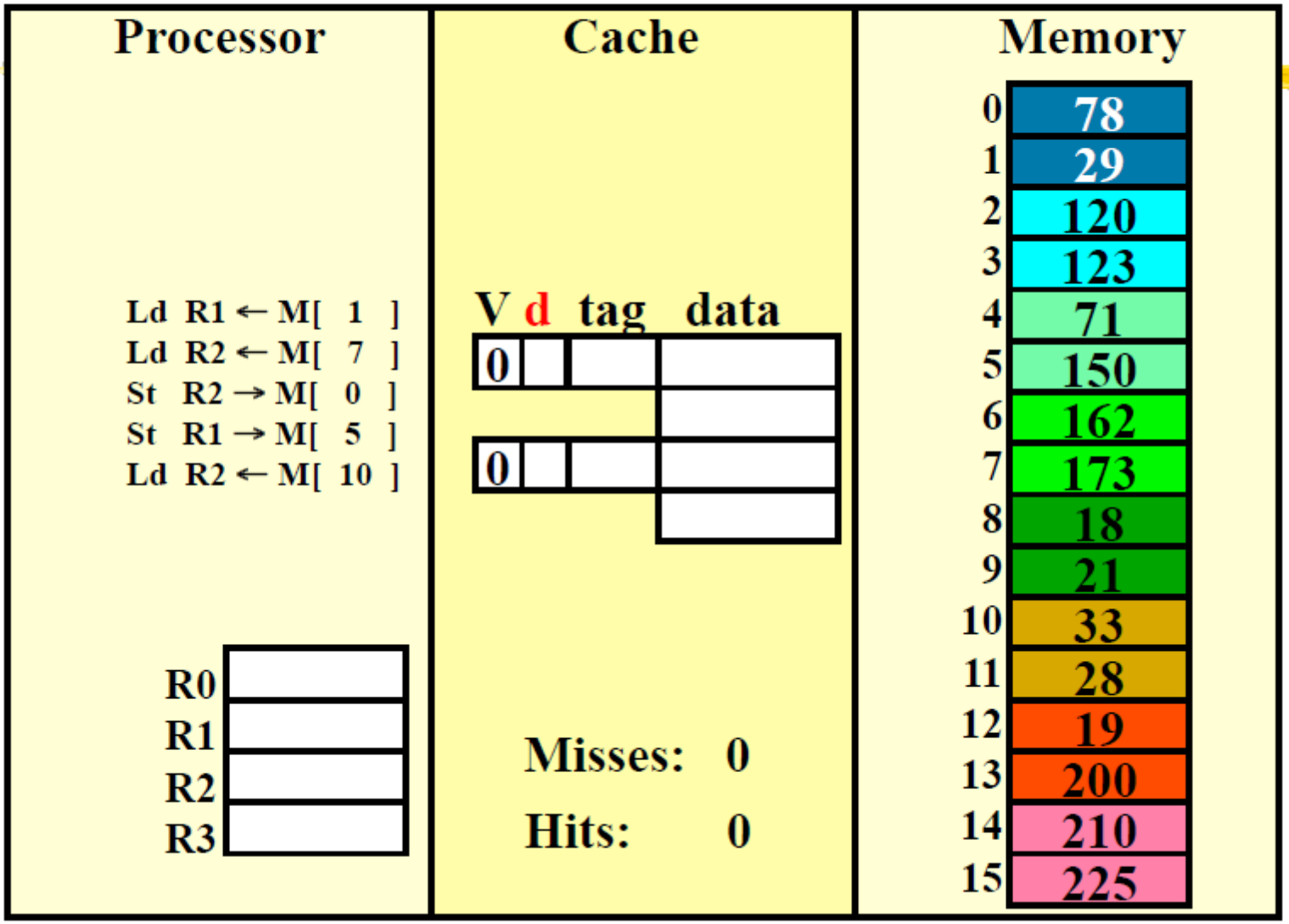


# How many memory references?

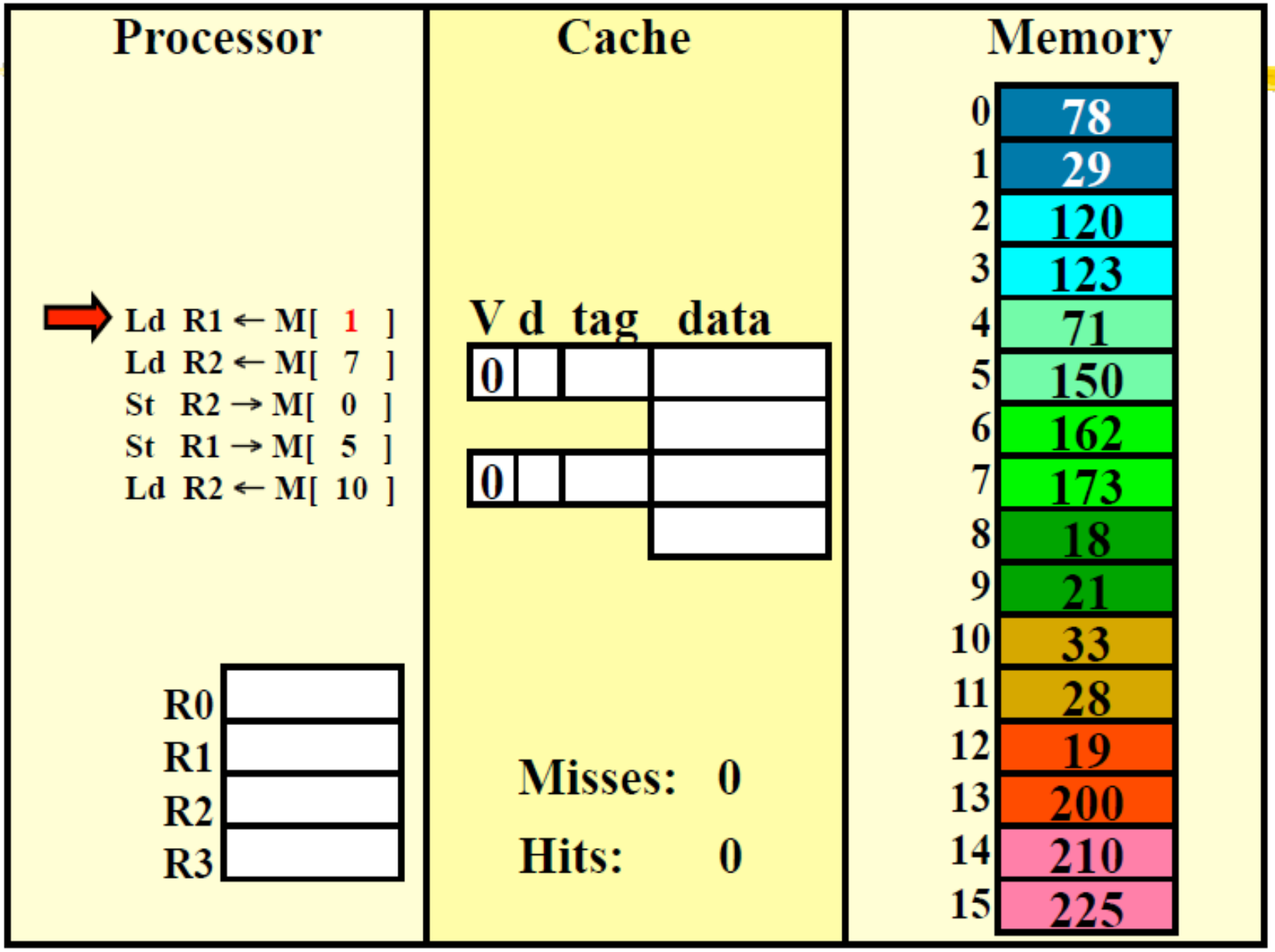
- Each miss reads a block
  - 2 bytes in this cache
- Each store writes a byte
- Total reads: 8 bytes
- Total writes: 2 bytes

but caches generally miss < 20%

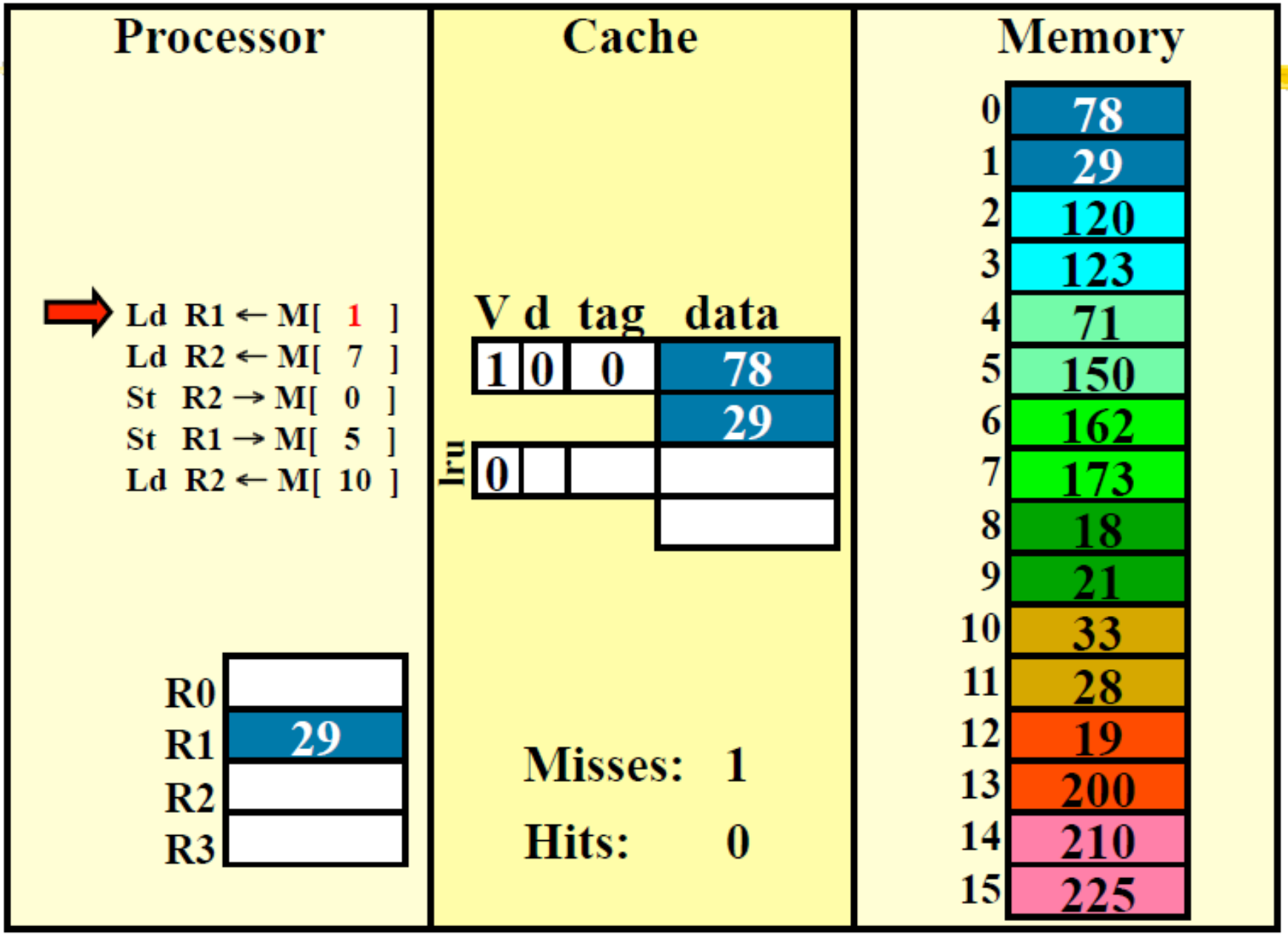
# Handling stores (write-back)



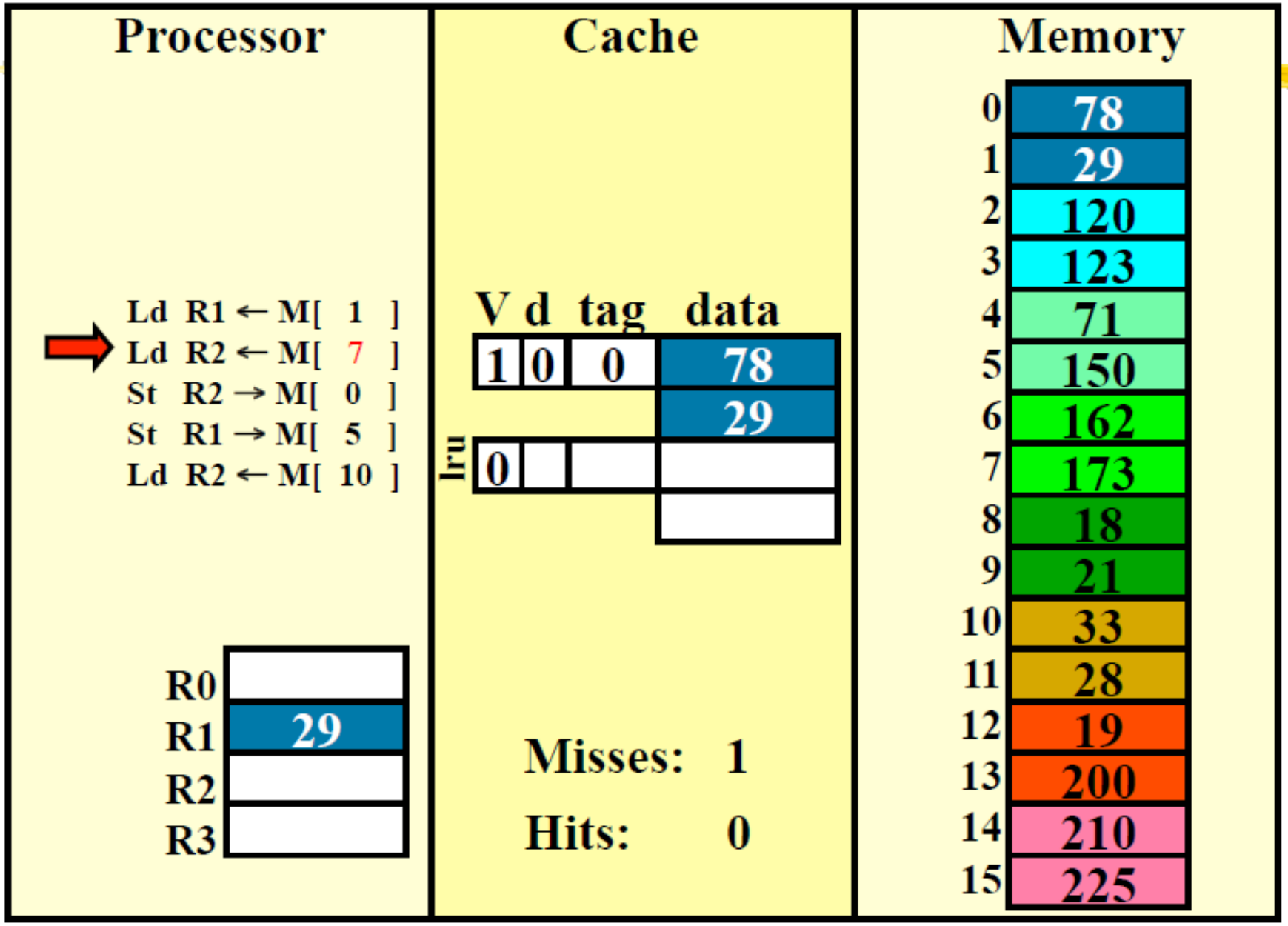
# write-back (REF 1)



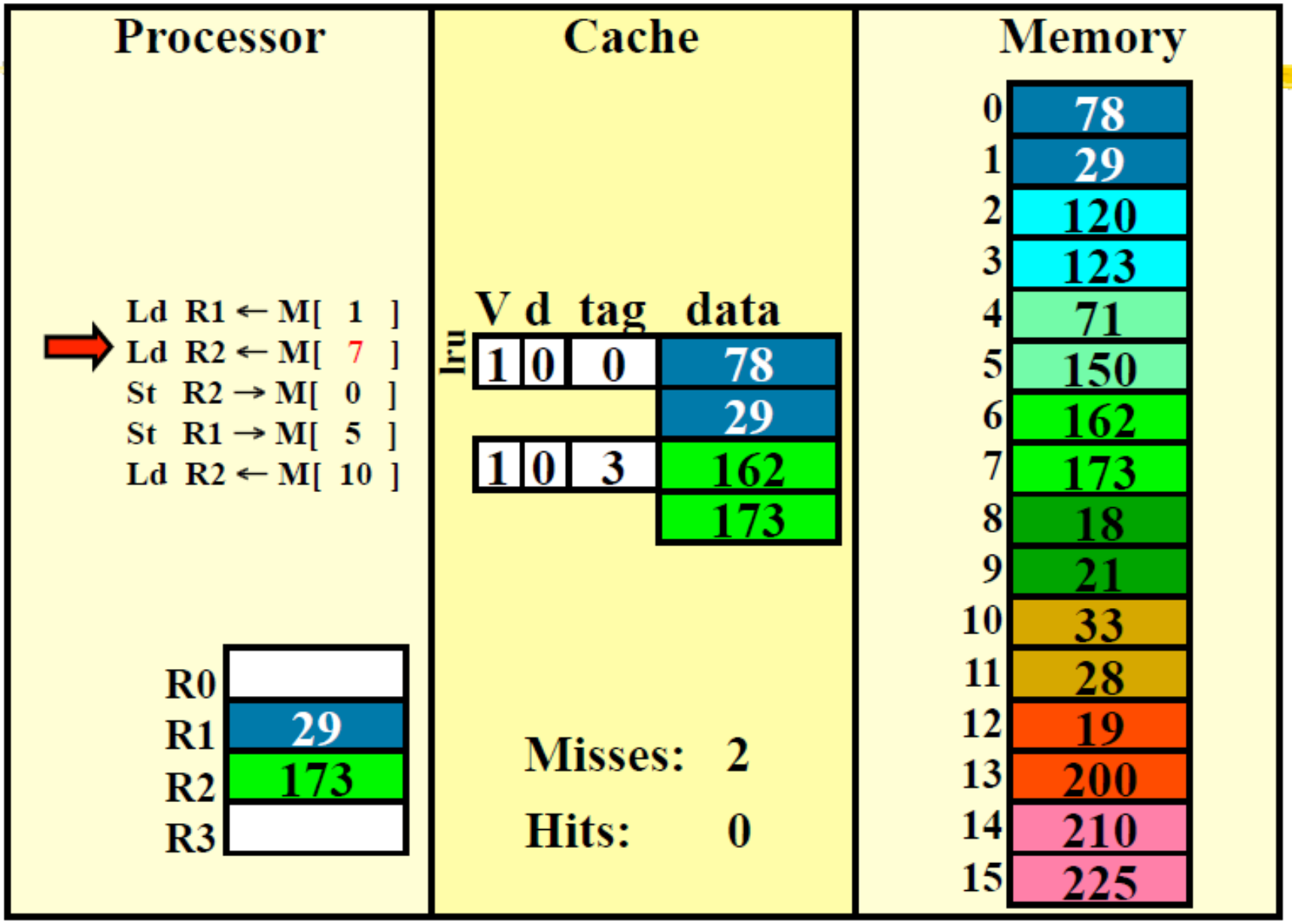
# write-back (REF 1)



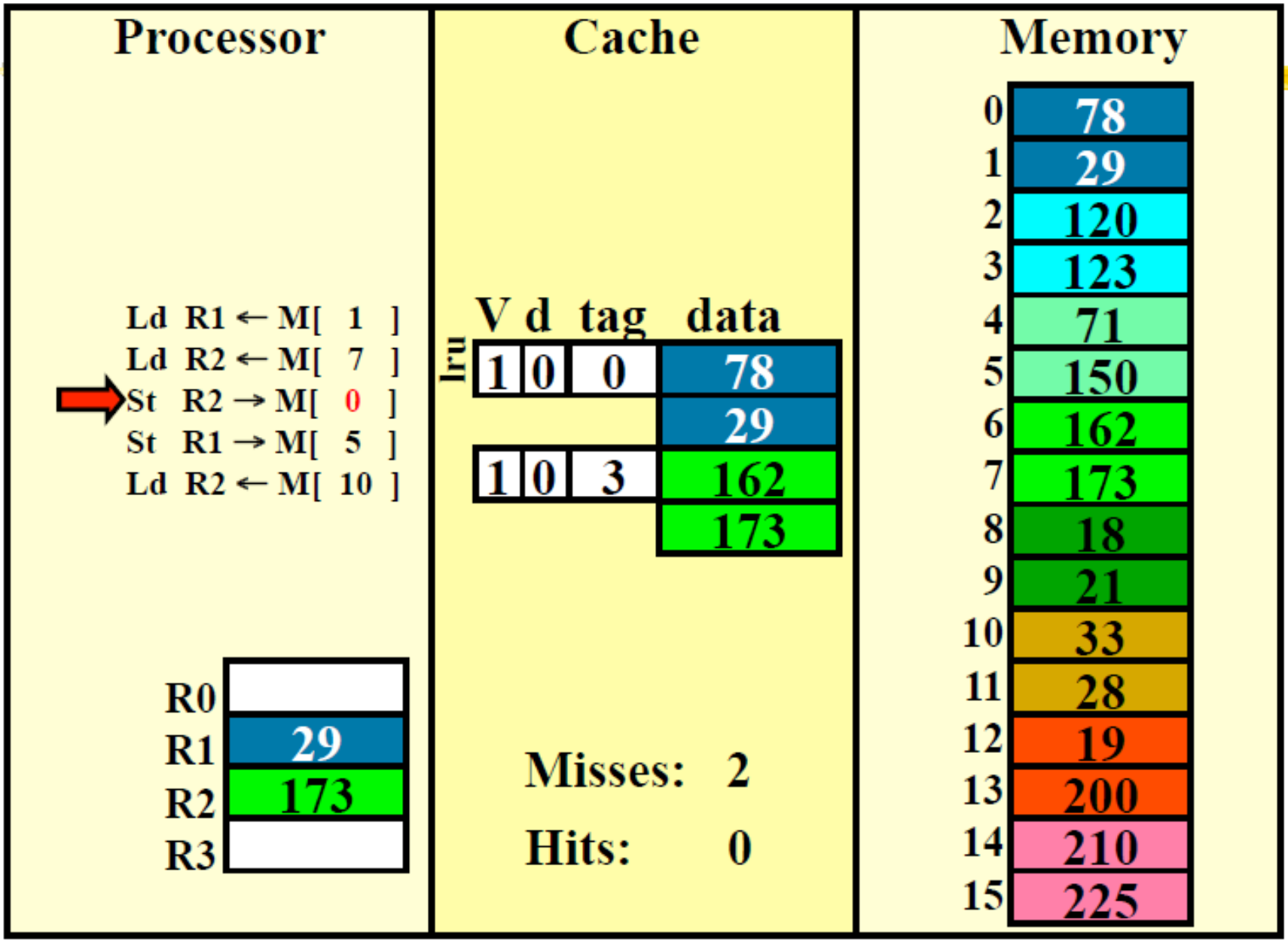
# write-back (REF 2)



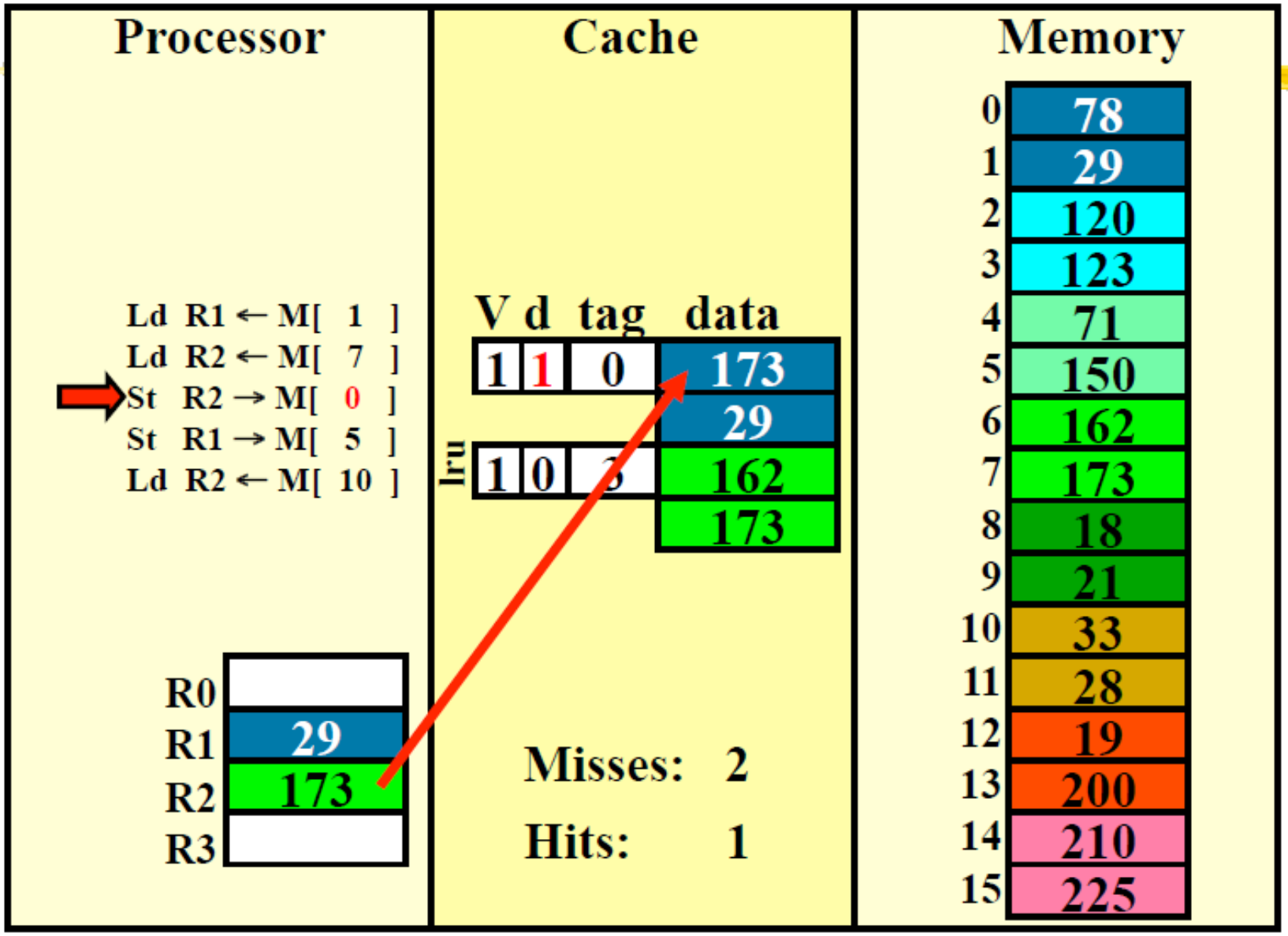
# write-back (REF 2)



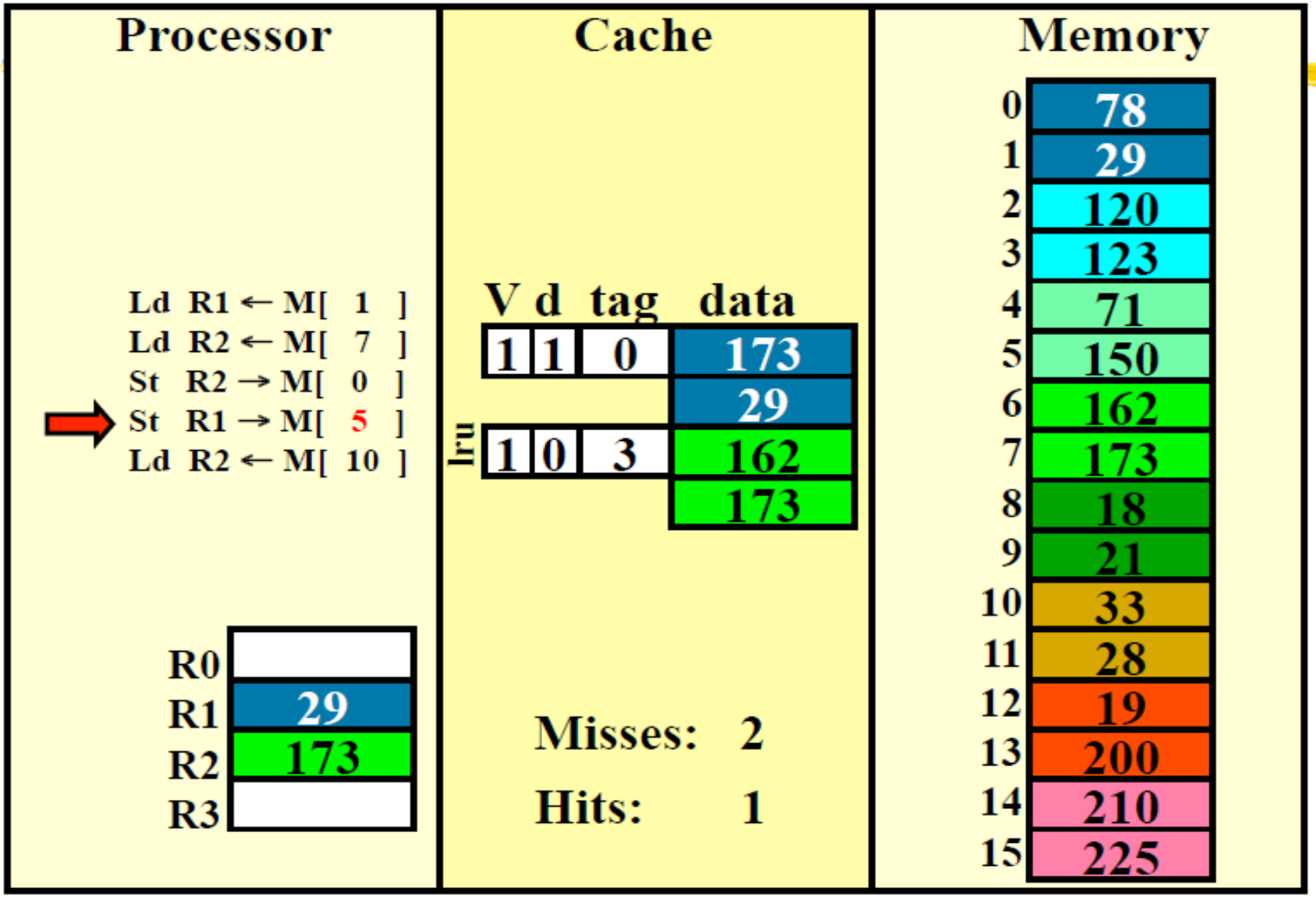
# write-back (REF 3)



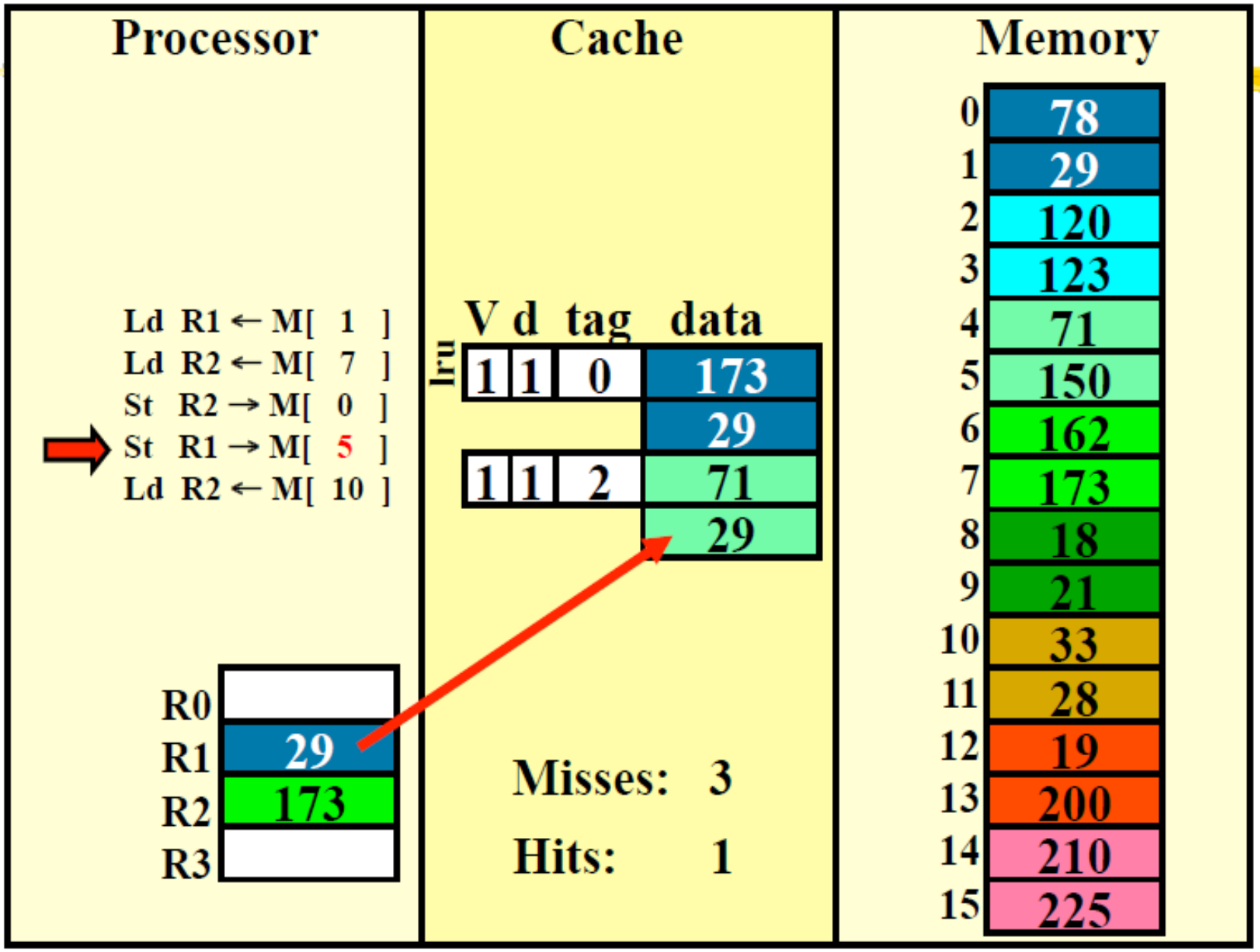
# write-back (REF 3)



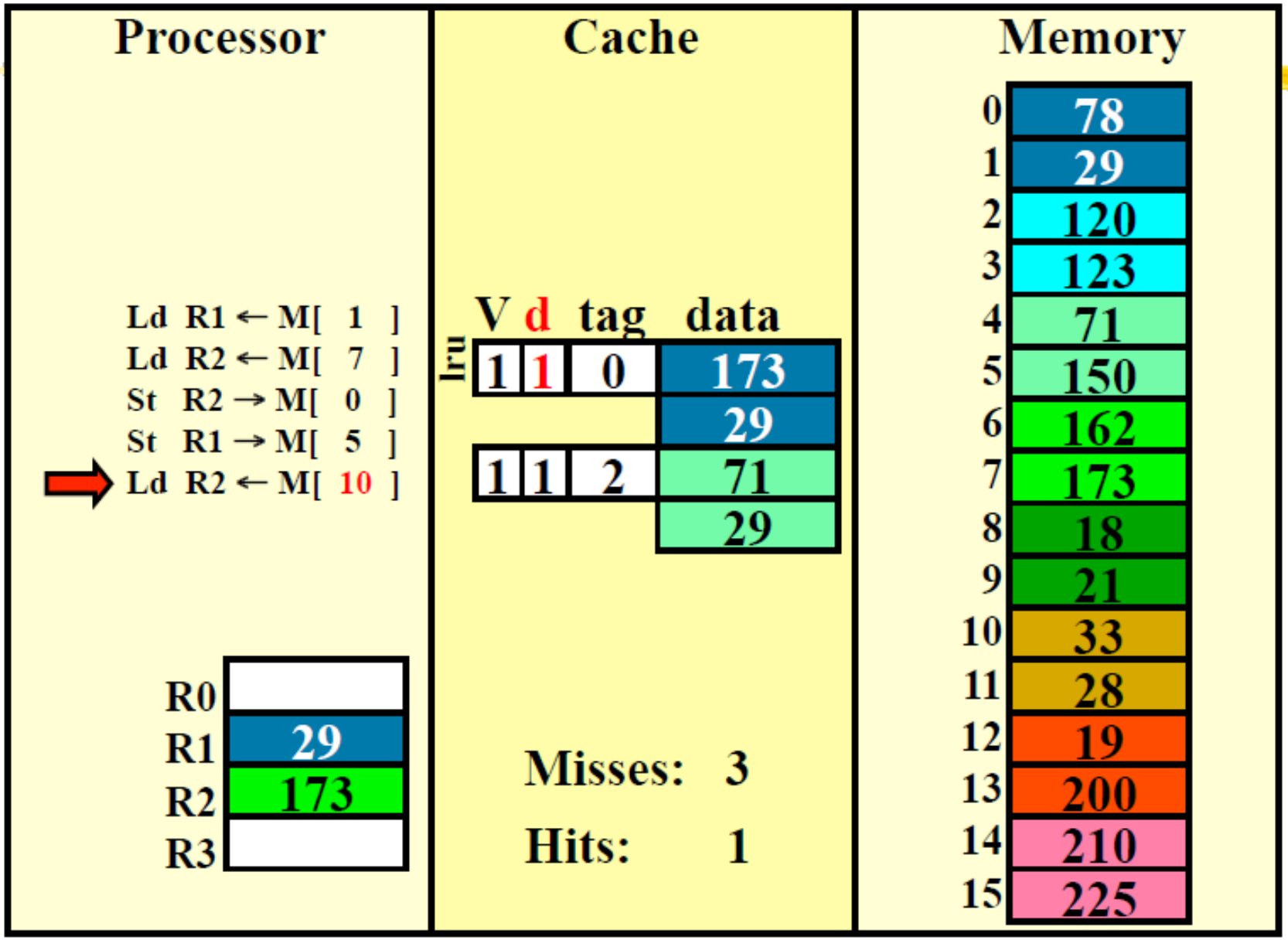
# write-back (REF 4)



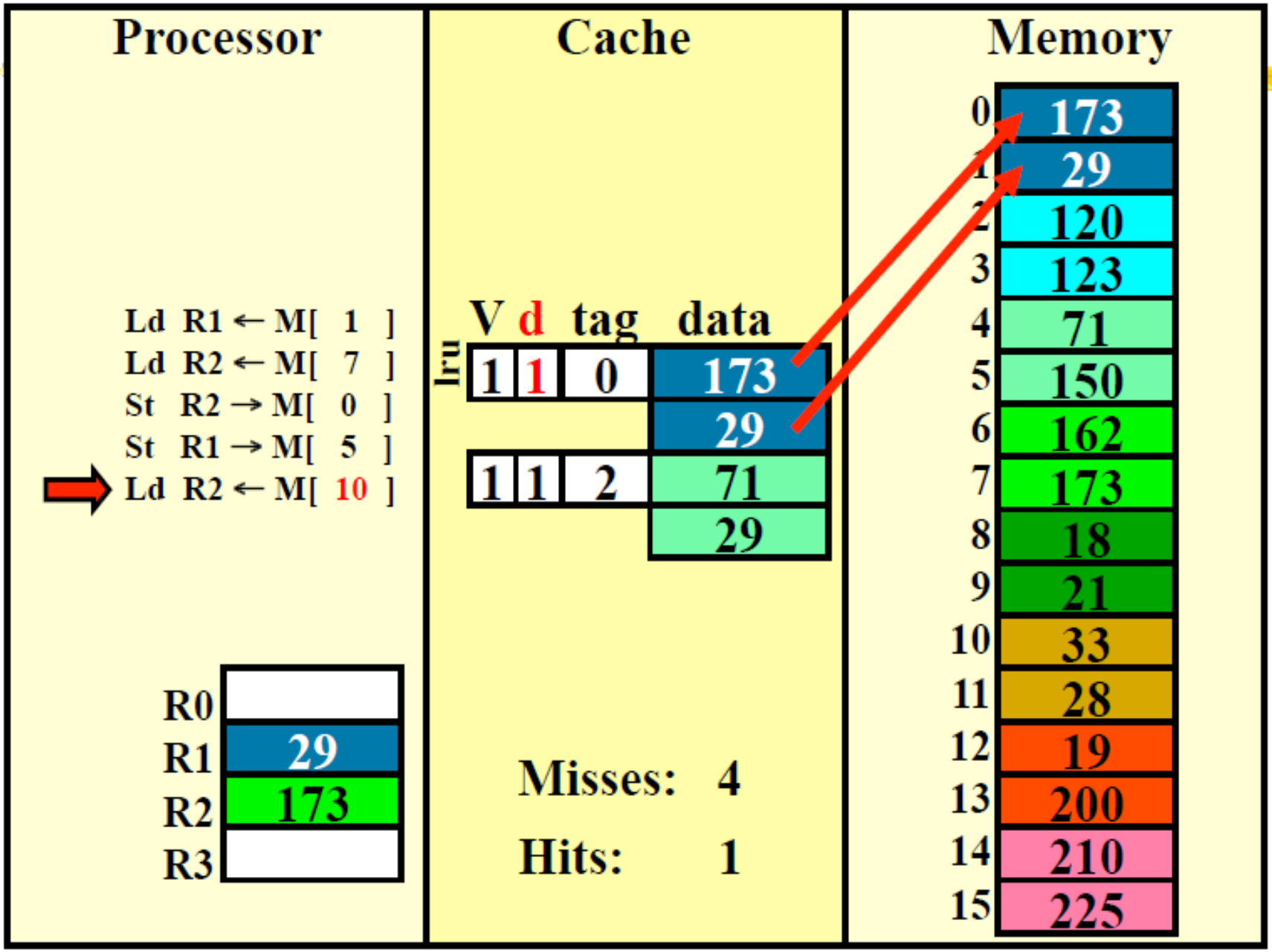
# write-back (REF 4)



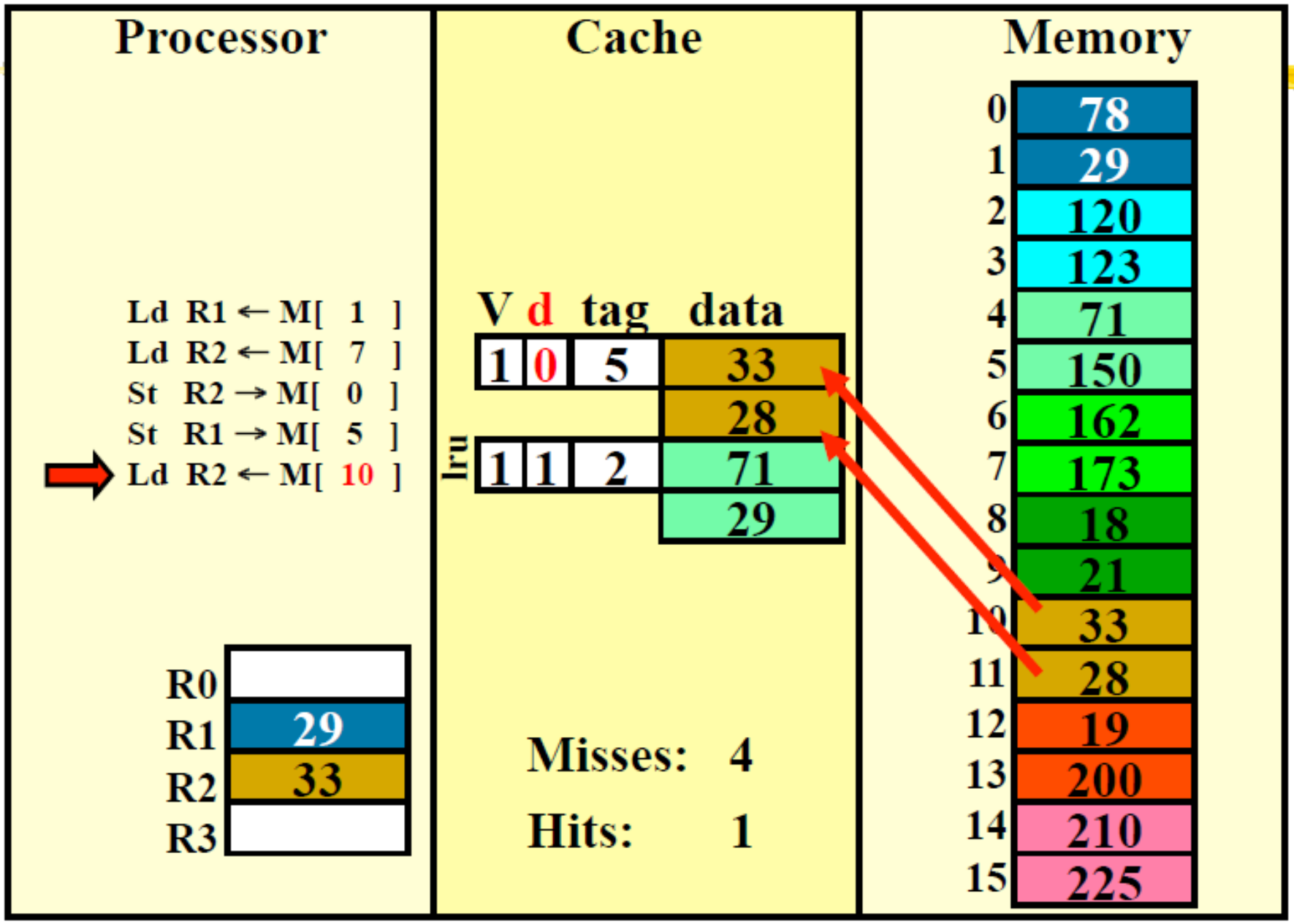
# write-back (REF 5)



# write-back (REF 5)



# write-back (REF 5)



# How many memory references?

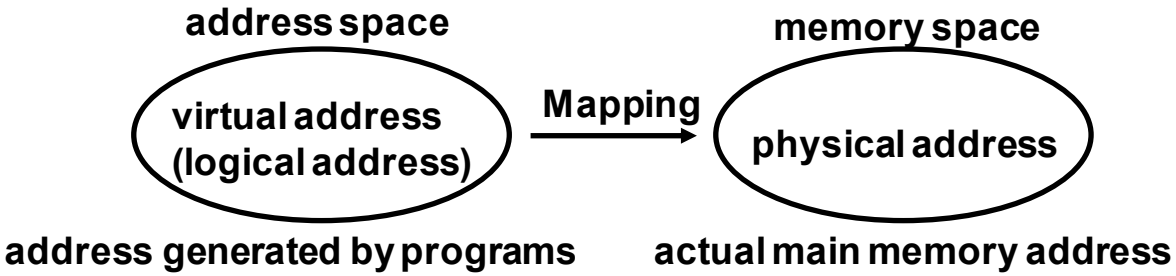
- Each miss reads a block
  - 2 bytes in this cache
- Each evicted dirty cache line writes a block
- Total reads: 8 bytes
- Total writes: 4 bytes (after final eviction)

Choose write-back or write-through?

# VIRTUAL MEMORY

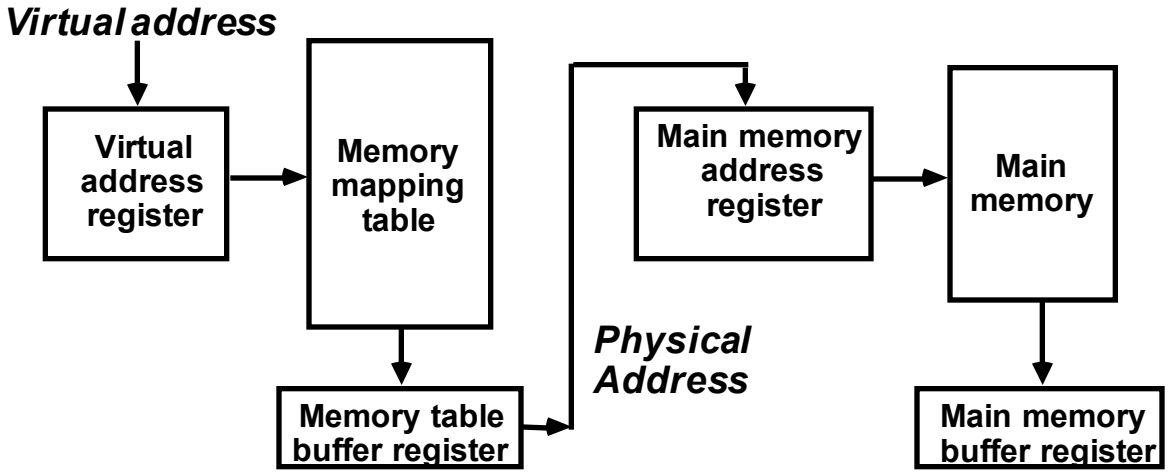
**Give the programmer the illusion that the system has a very large memory, even though the computer actually has a relatively small main memory**

## Address Space(Logical) and Memory Space(Physical)




## Address Mapping

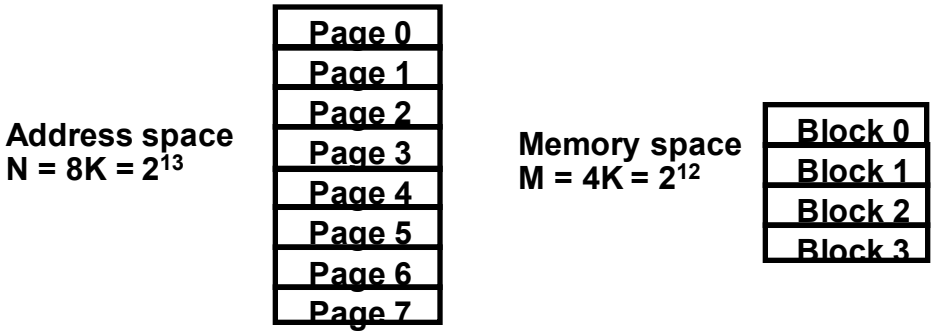
Memory *Mapping Table* for *Virtual Address -> Physical Address*



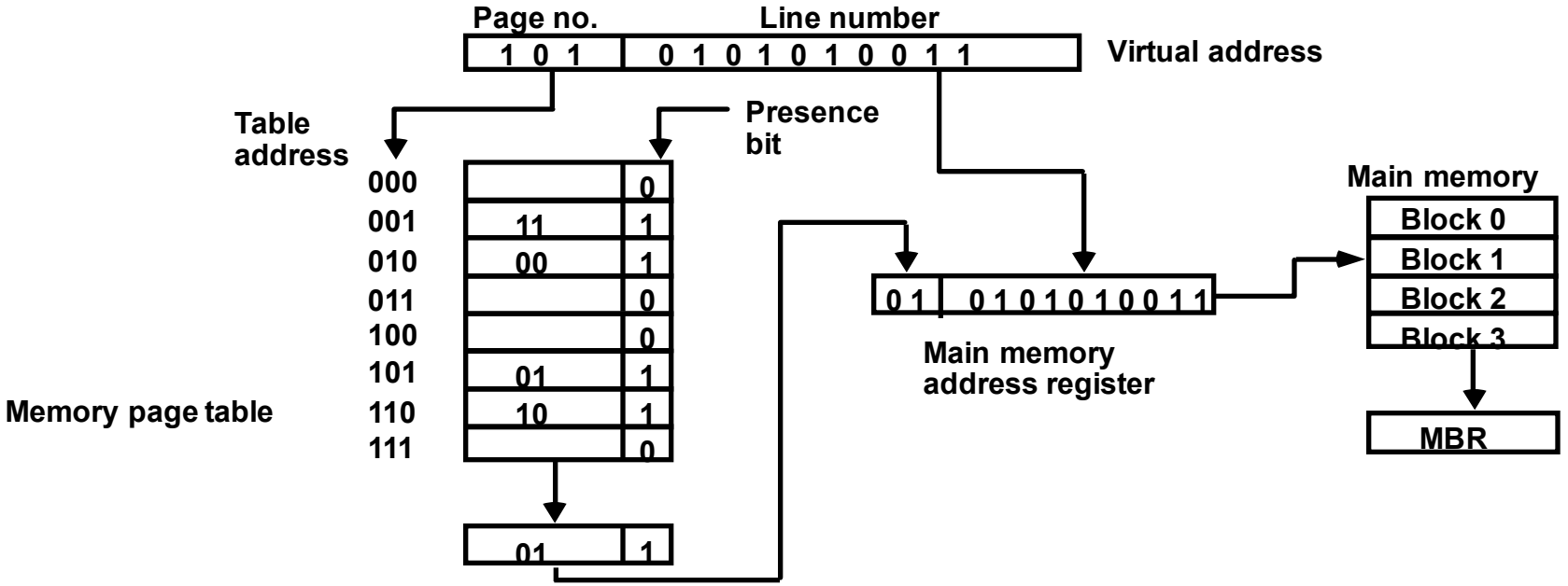
# ADDRESS MAPPING

**Address Space and Memory Space are each divided into fixed size group of words called *blocks* or *pages***

**1K words group**




**Organization of memory Mapping Table in a paged system**



# ASSOCIATIVE MEMORY PAGE TABLE

Assume that

Number of Blocks in memory = m

Number of Pages in Virtual Address Space = n

Page Table

- Straight forward design -> n entry table in memory

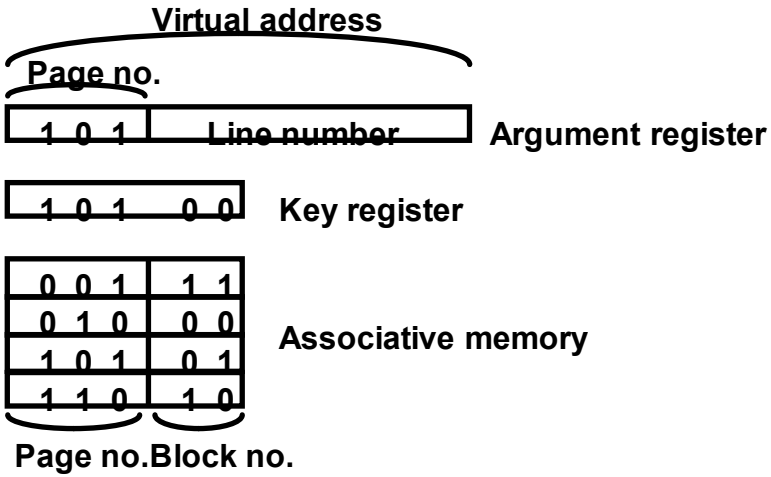
Inefficient storage space utilization

<- n-m entries of the table is empty

- More efficient method is m-entry Page Table

Page Table made of an Associative Memory

m words; (Page Number:Block Number)




Page Fault

Page number cannot be found in the Page Table

# MEMORY MANAGEMENT HARDWARE

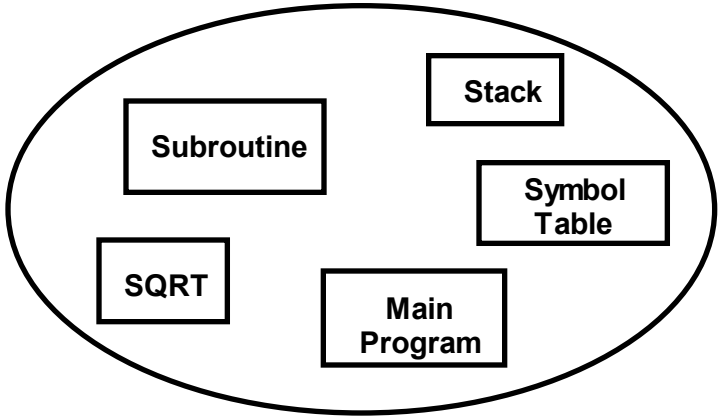
**Basic Functions of MM**

- *Dynamic Storage Relocation* - mapping logical memory references to physical memory references
- Provision for *Sharing* common information stored in memory by different users
- *Protection* of information against unauthorized access

**Segmentation**

- A segment is a set of logically related instructions or data elements associated with a given name
- Variable size

**User's view of memory**



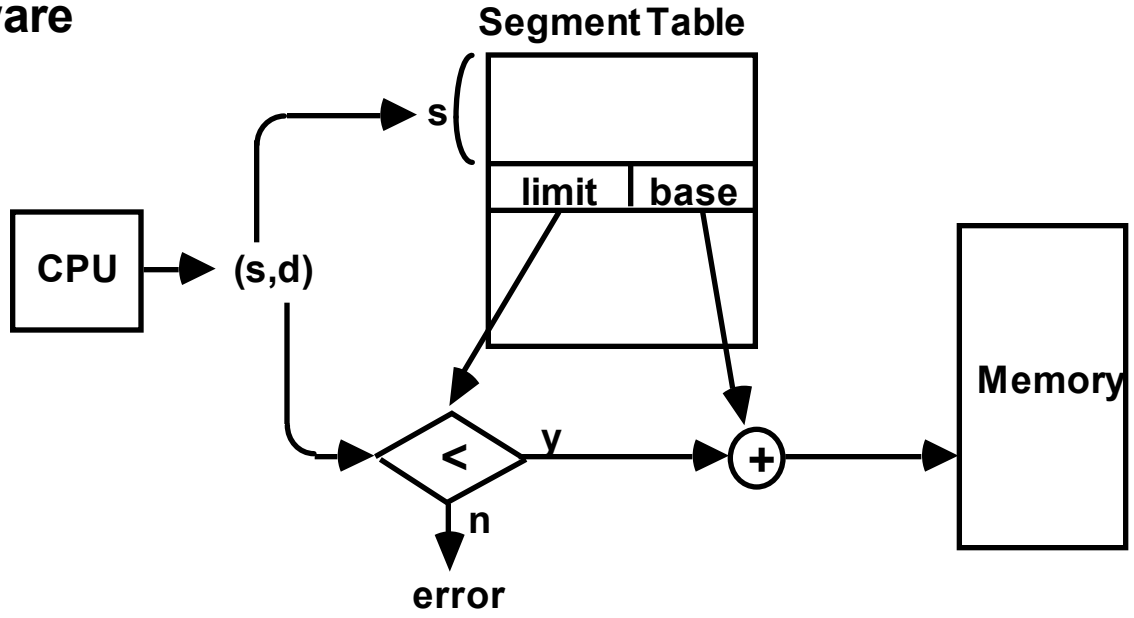

User's view of a program

The user does not think of memory as a linear array of words. Rather the user prefers to view memory as a collection of variable sized segments, with no necessary ordering among segments.

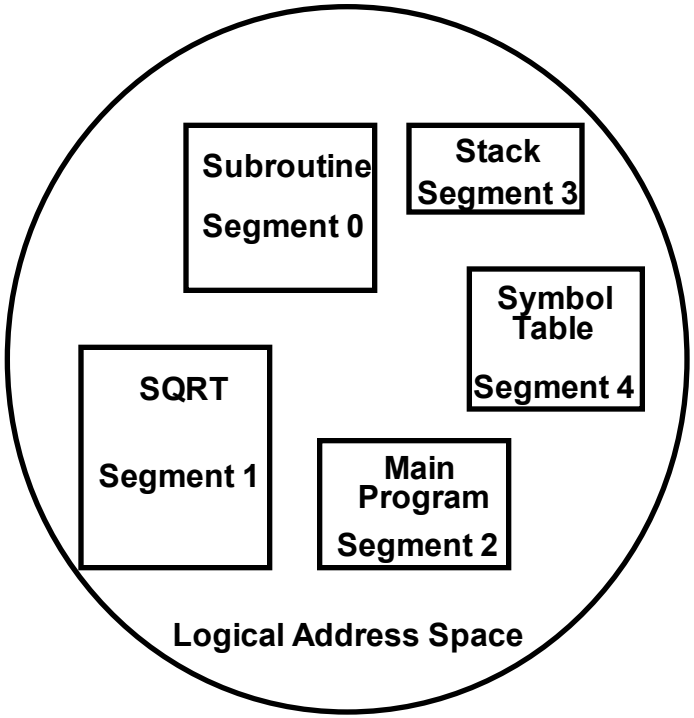
# SEGMENTATION

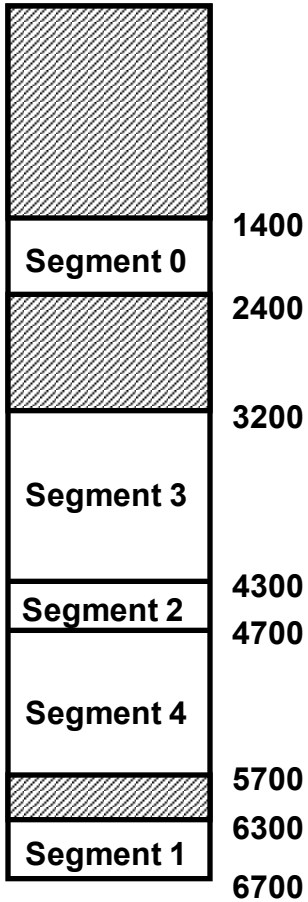
- A memory management scheme which supports user's view of memory
- A logical address space is a collection of segments
- Each segment has a name and a length
- Address specify both the segment name and the offset within the segment.
- For simplicity of implementations, segments are numbered.

**Segmentation Hardware**



# SEGMENTATION EXAMPLE

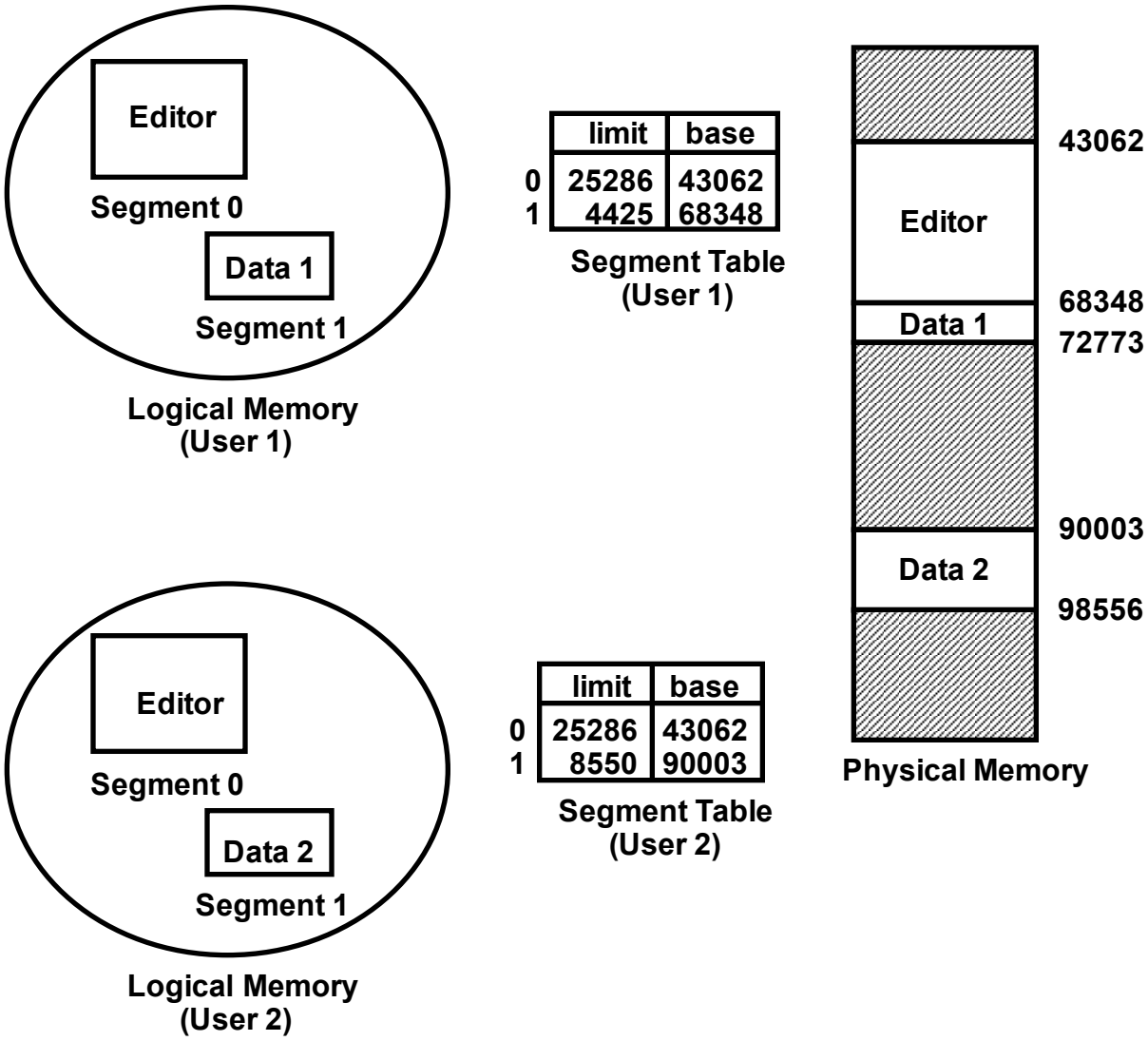


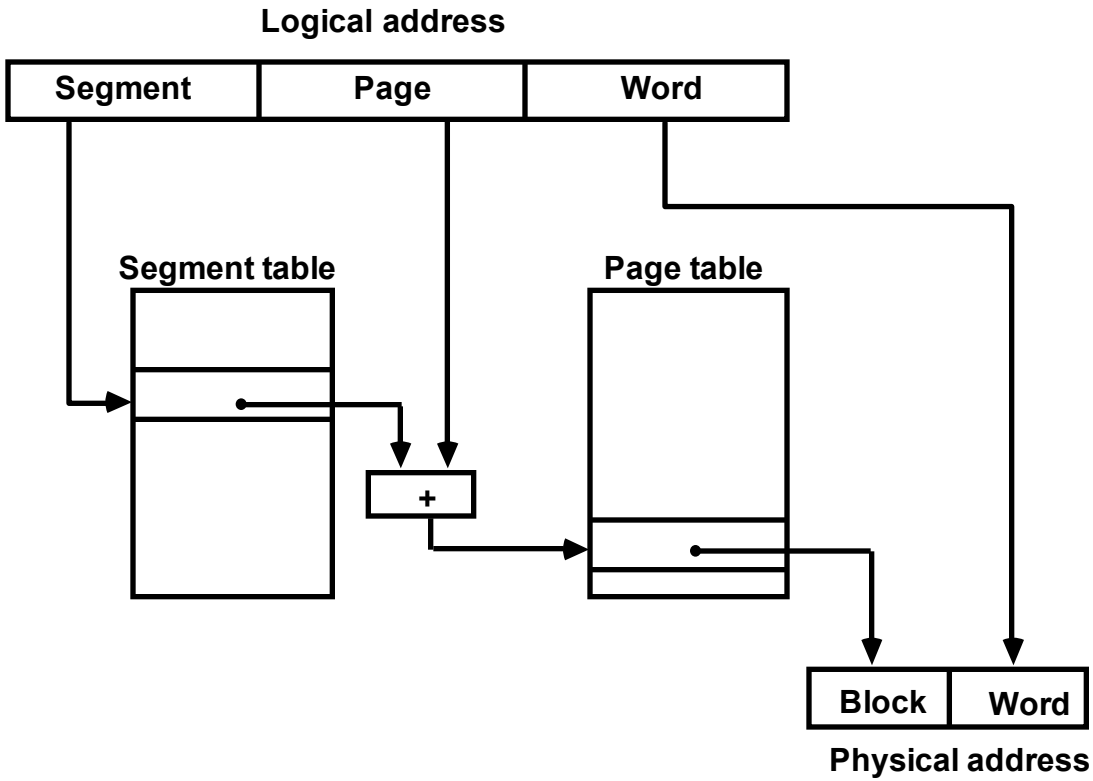

Segment Table

| | limit | base |
|---|---|---|
| 0 | 1000 | 1400 |
| 1 | 400 | 6300 |
| 2 | 400 | 4300 |
| 3 | 1100 | 3200 |
| 4 | 1000 | 4700 |

# SHARING OF SEGMENTS



# SEGMENTED PAGE SYSTEM



# IMPLEMENTATION OF PAGE AND SEGMENT TABLES

**Implementation of the Page Table**

- Hardware registers (if the page table is reasonably small)
- Main memory
  - Page Table Base Register(PTBR) points to PT
  - Two memory accesses are needed to access a word; one for the page table, one for the word
- Cache memory
  - To speedup the effective memory access time, a special small memory called associative memory, or cache is used

**Implementation of the Segment Table**

Similar to the case of the page table

# EXAMPLE

## Logical and Physical Addresses

Logical address format: 16 segments of 256 pages each, each page has 256words

| 4 | 8 | 8 |
|---|---|---|
| Segment | Page | Word |

$2^{20}$ x 32 Physical memory

Physical address format: 4096 blocks of 256 words each, each word has 32bits

| 12 | 8 |
|---|---|
| Block | Word |

## Logical and Physical Memory Address Assignment

| Hexa address | Page number |
|---|---|
| 60000 | Page 0 |
| 60100 | Page 1 |
| 60200 | Page 2 |
| 60300 | Page 3 |
| 60400<br>604FF | Page 4 |

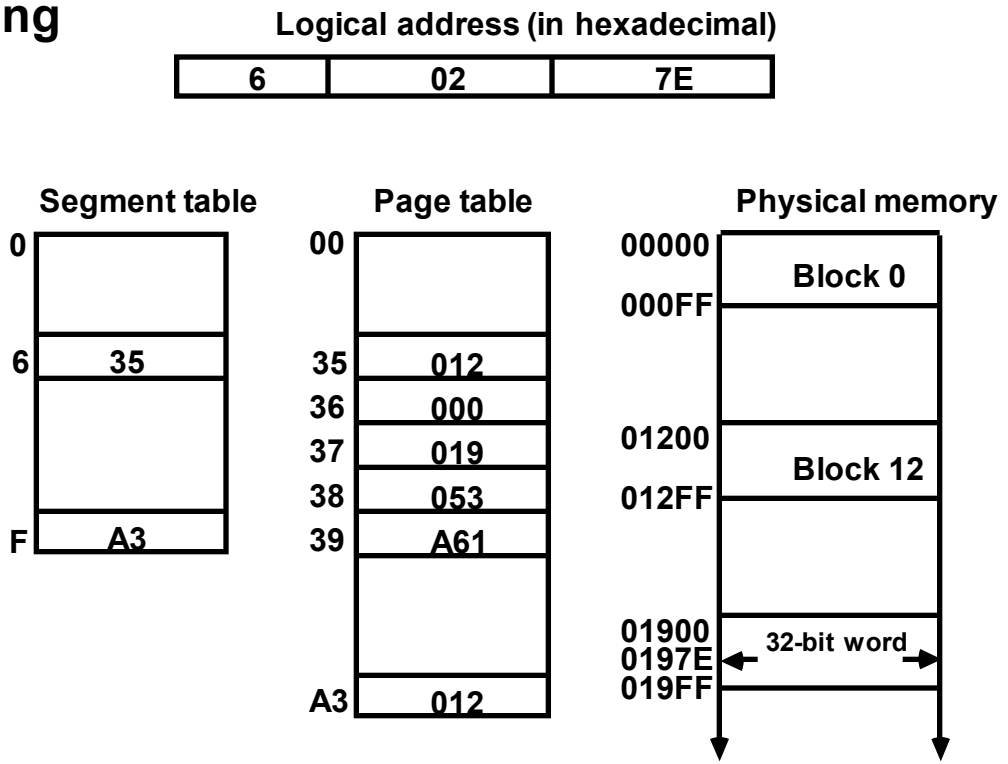
(a) Logical address assignment

| Segment | Page | Block |
|---|---|---|
| 6 | 00 | 012 |
| 6 | 01 | 000 |
| 6 | 02 | 019 |
| 6 | 03 | 053 |
| 6 | 04 | A61 |

(b) Segment-page versus memory block assignment

# LOGICAL TO PHYSICAL MEMORY MAPPING

## Segment and page table mapping




## Associative memory mapping

| Segment | Page | Block |
|---|---|---|
| 6 | 02 | 019 |
| 6 | 04 | A61 |
| | | |