

17. Caches: Writes and Blocks

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan, Ann Arbor

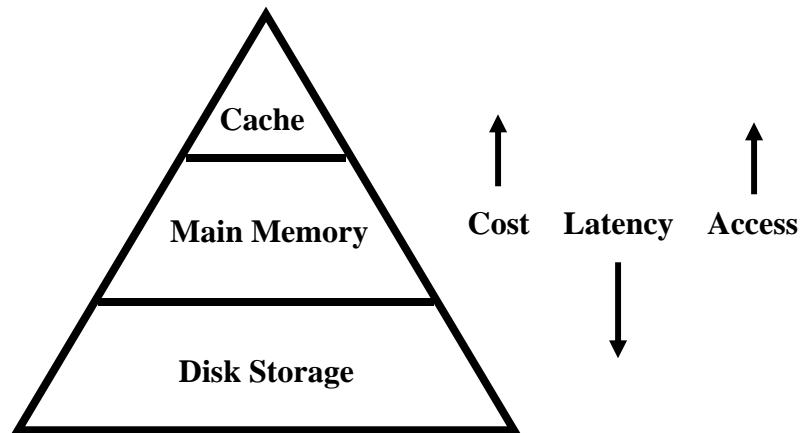
© Austin & Papaefthymiou, 2009

The material in this presentation cannot be
copied in any form without our written permission

Announcements

- ❑ Homework 5
 - Posted

What is a "Memory Hierarchy"?



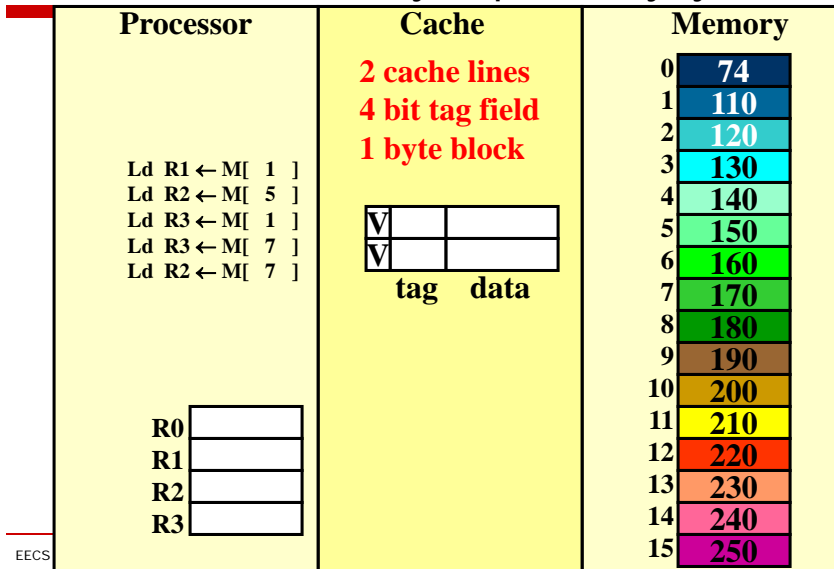
Basic Cache Design

- 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 they don't match, get the data from main memory

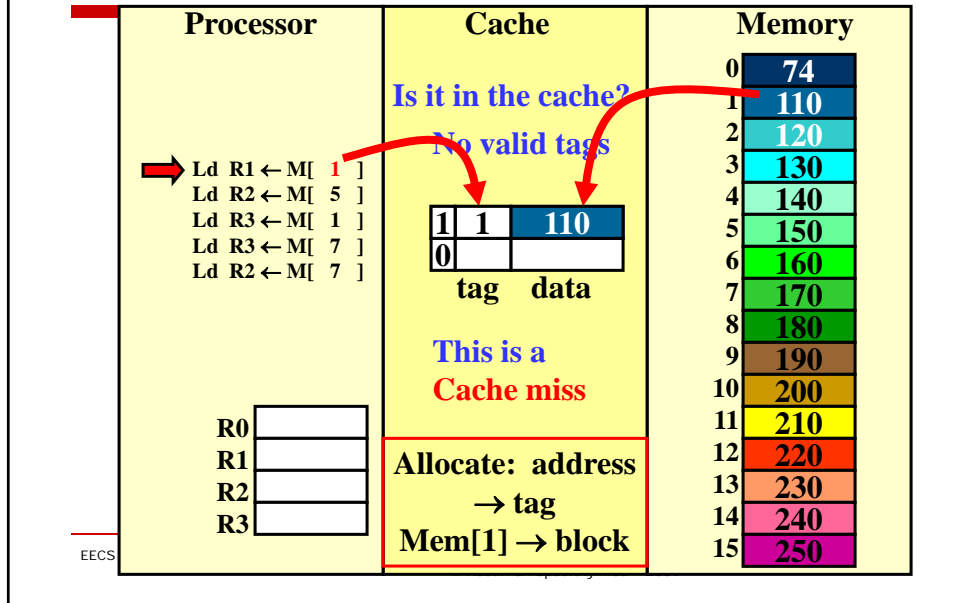
Temporal Locality

- ❑ 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.
- ❑ 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.

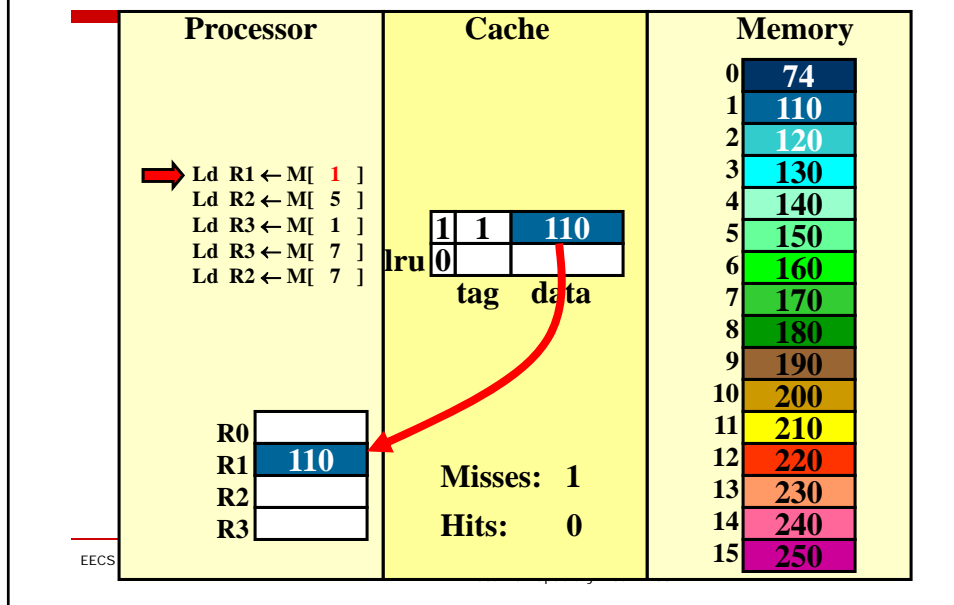
From Last Lecture: A Very Simple Memory System



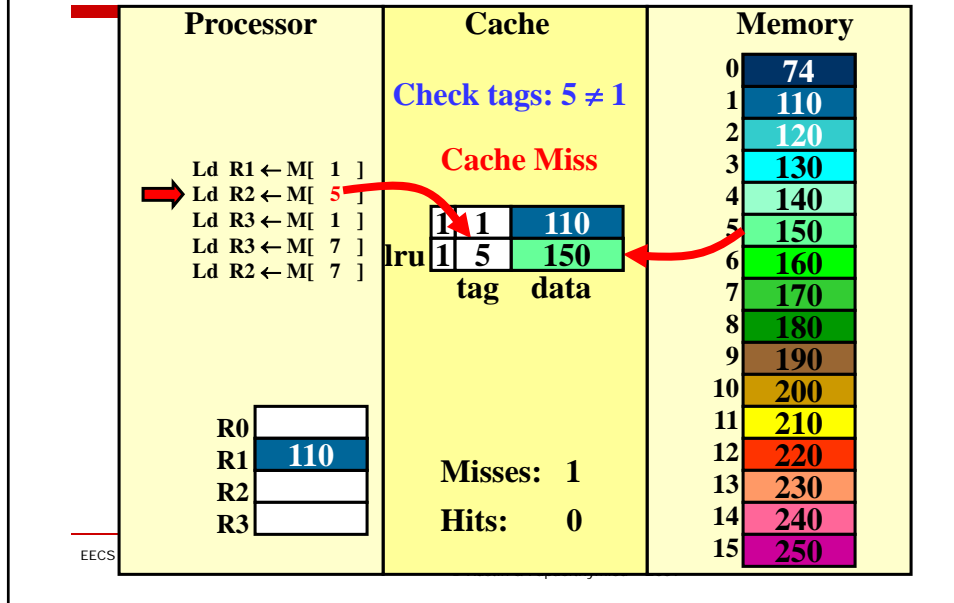
A Very Simple Memory System



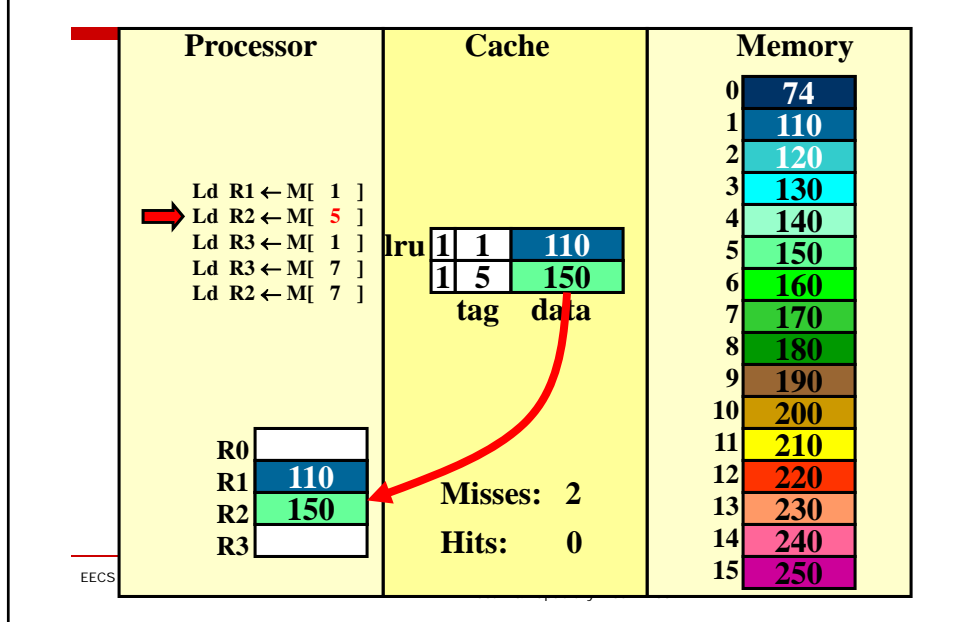
A Very Simple Memory System



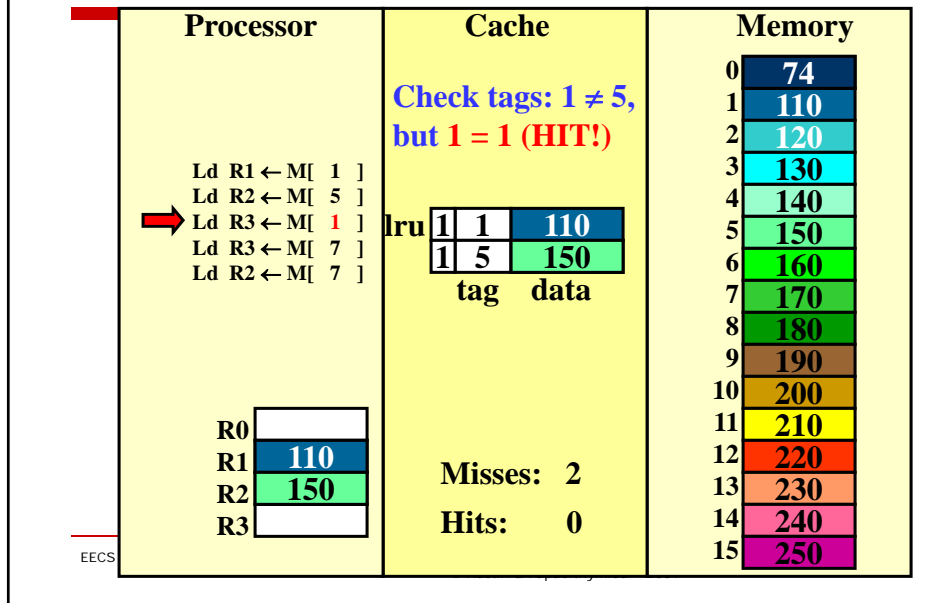
A Very Simple Memory System



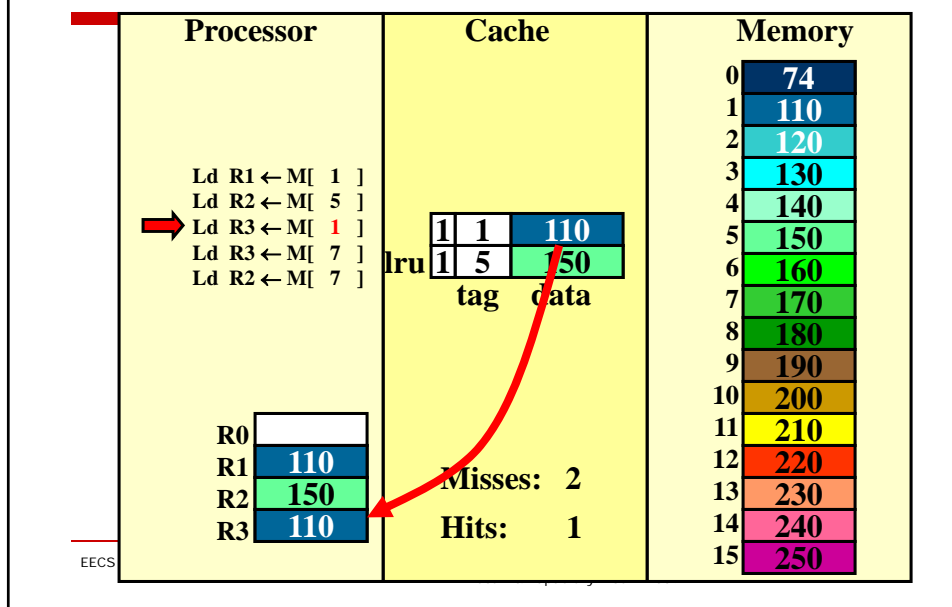
A Very Simple Memory System



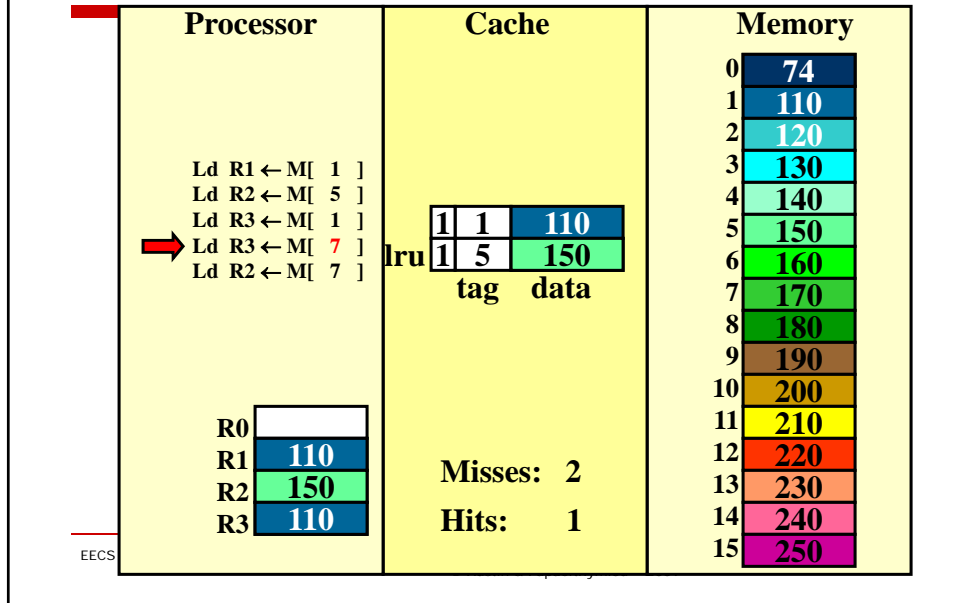
A Very Simple Memory System



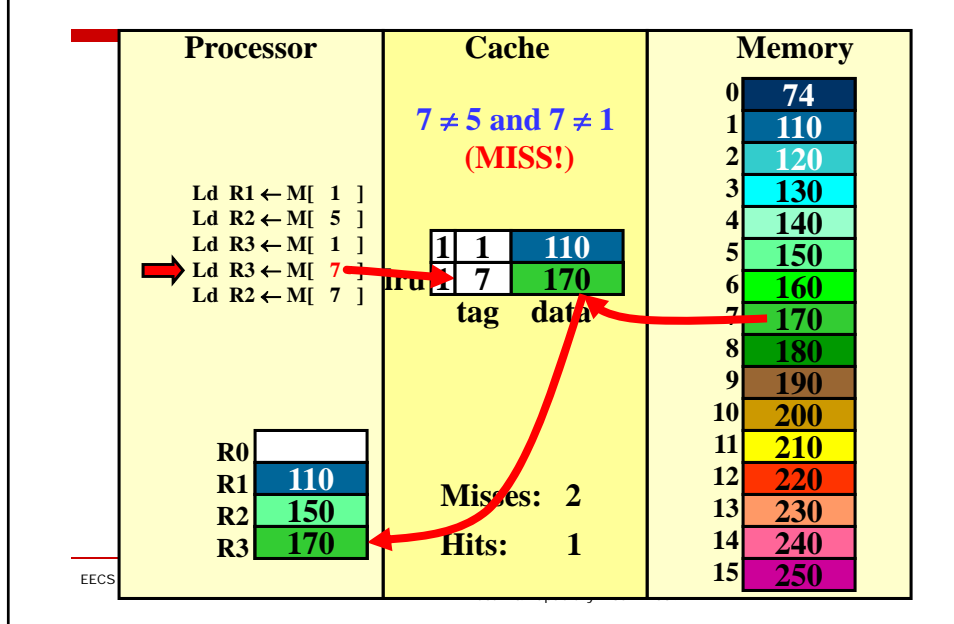
A Very Simple Memory System



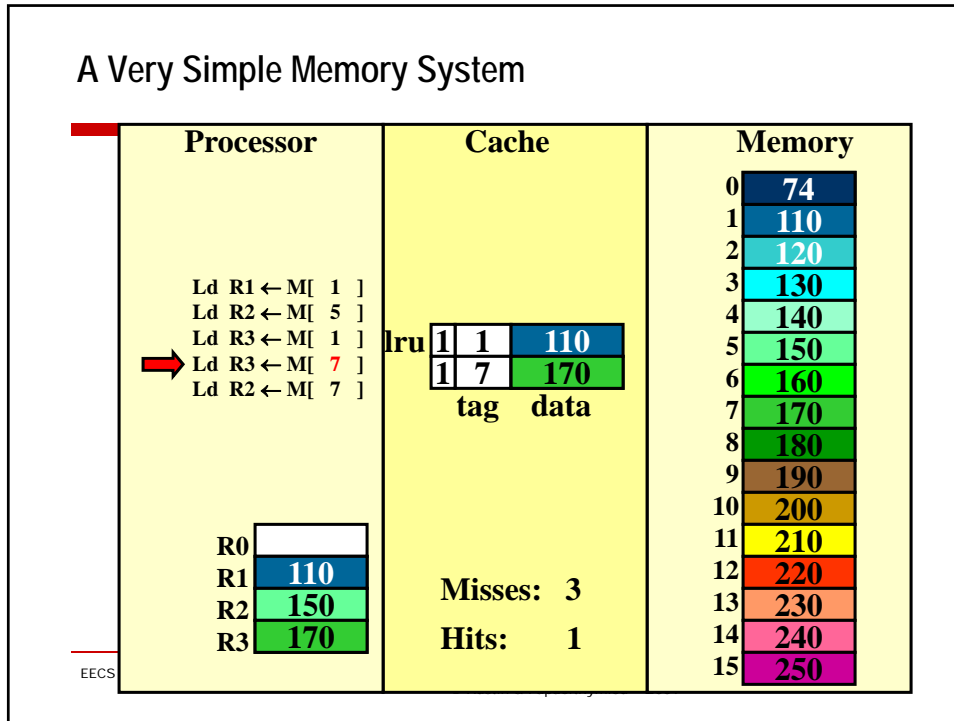
A Very Simple Memory System



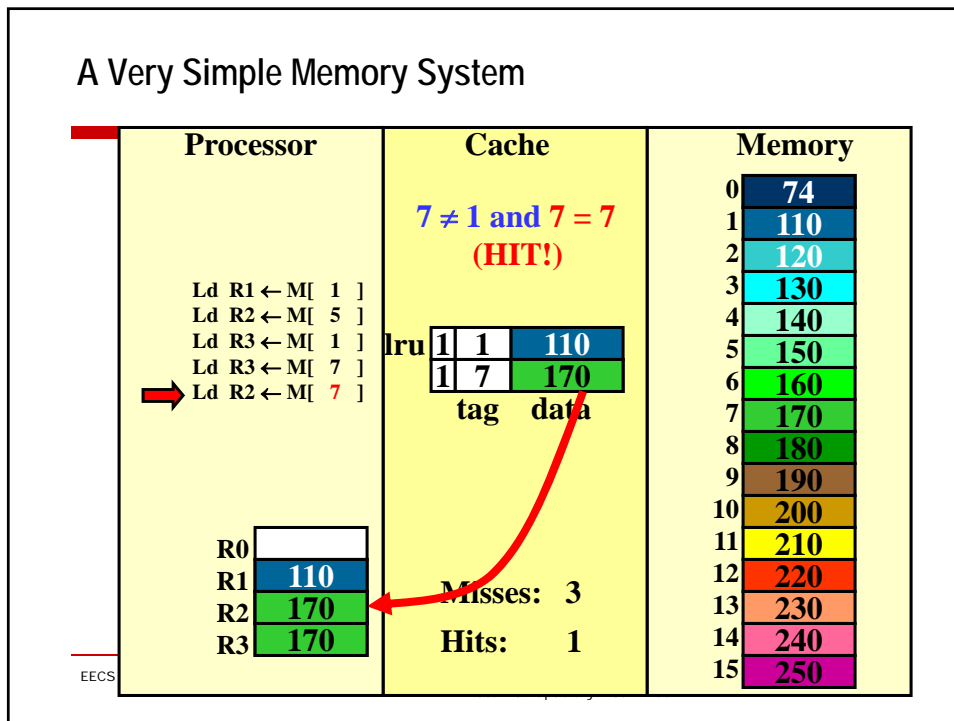
A Very Simple Memory System



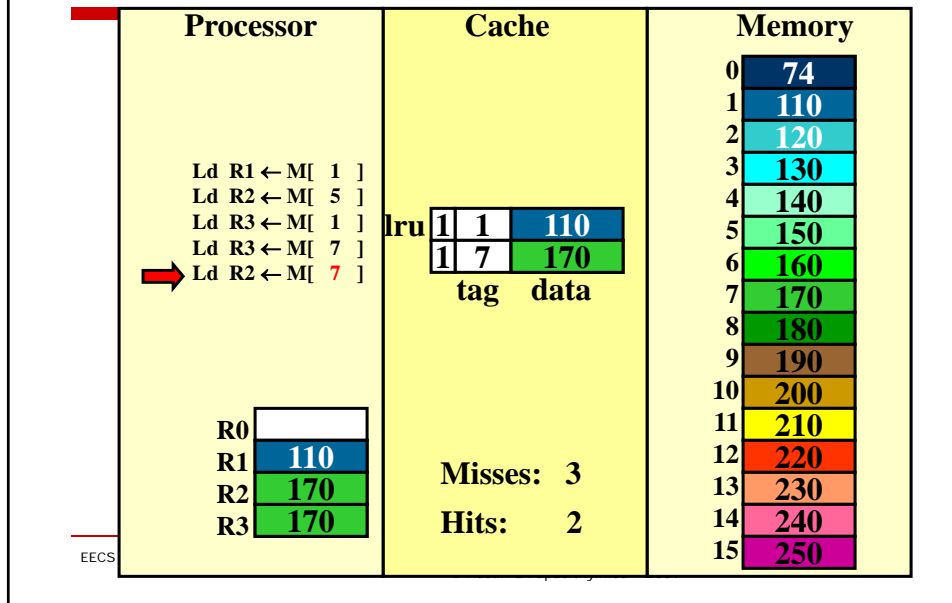
A Very Simple Memory System



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

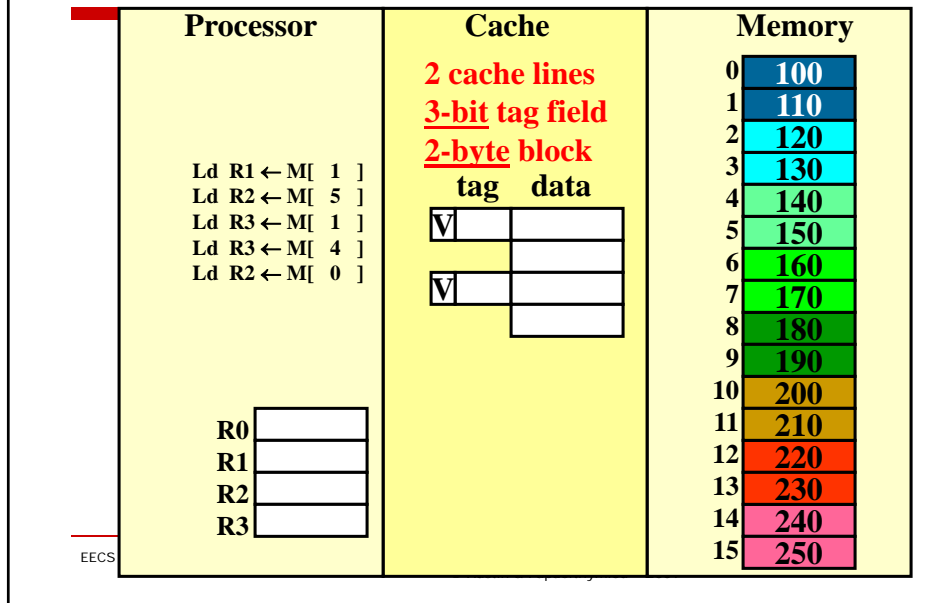
Calculating Cost

- ❑ 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?

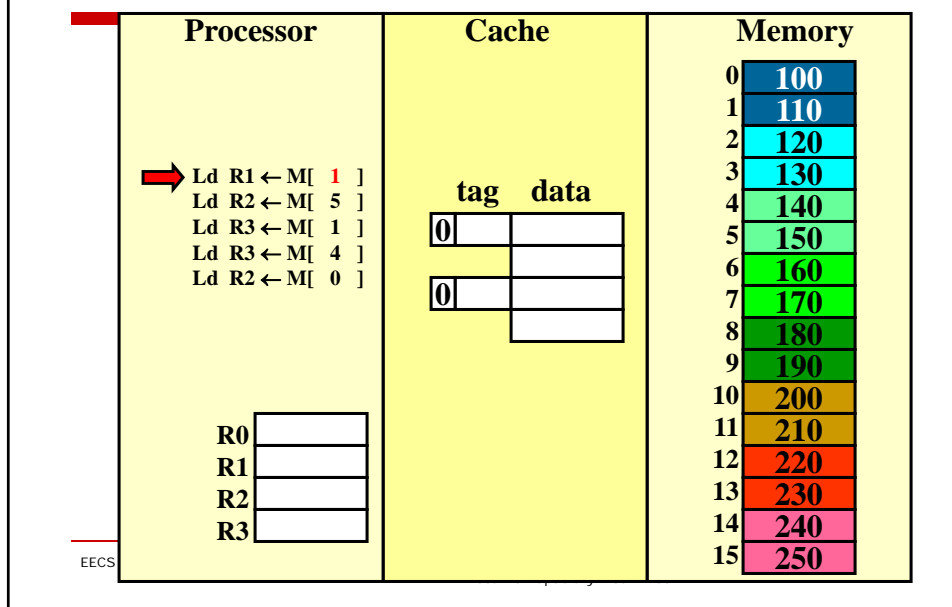
How can we reduce the overhead?

- ❑ 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.

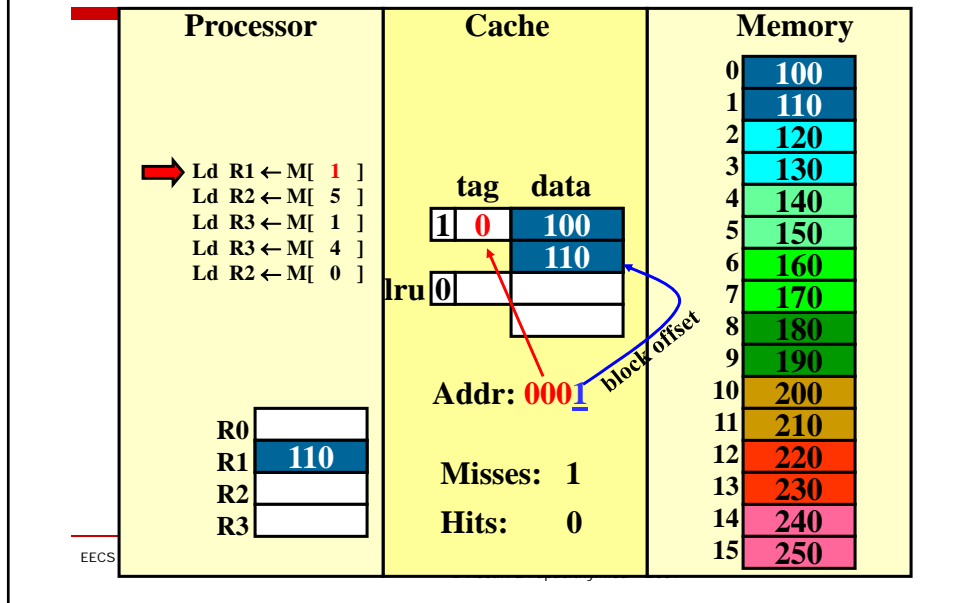
Block size for caches



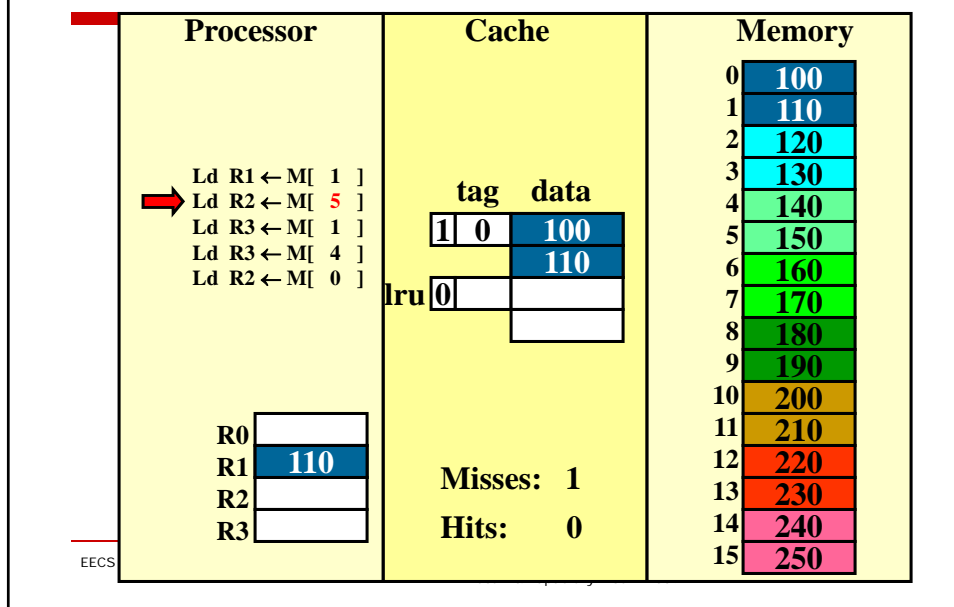
Block size for caches



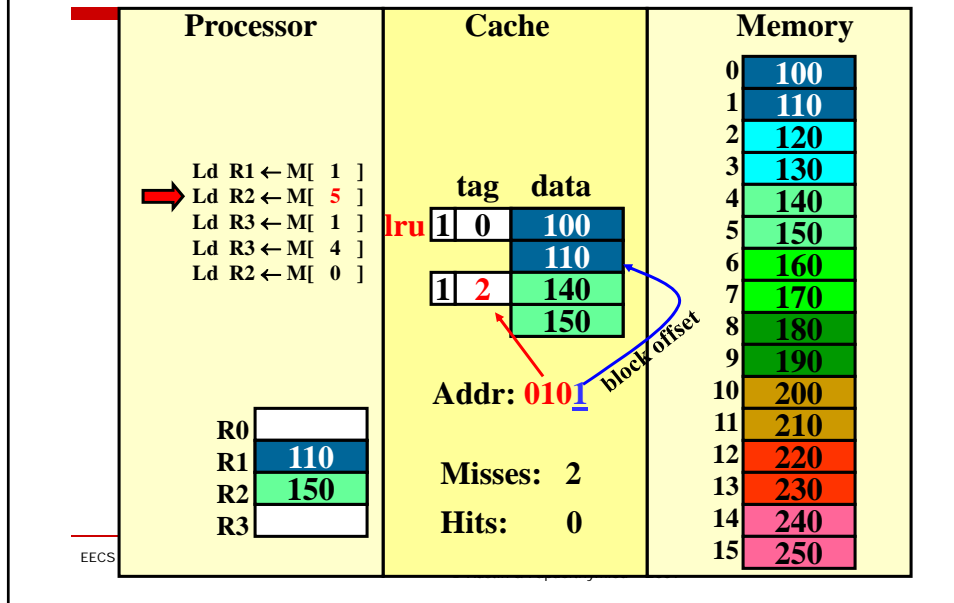
Block size for caches



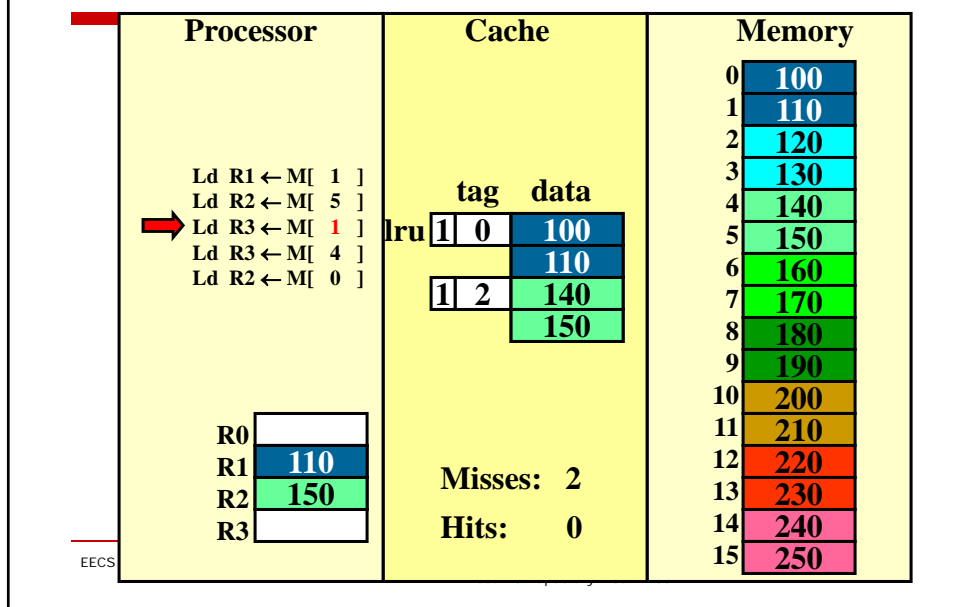
Block size for caches



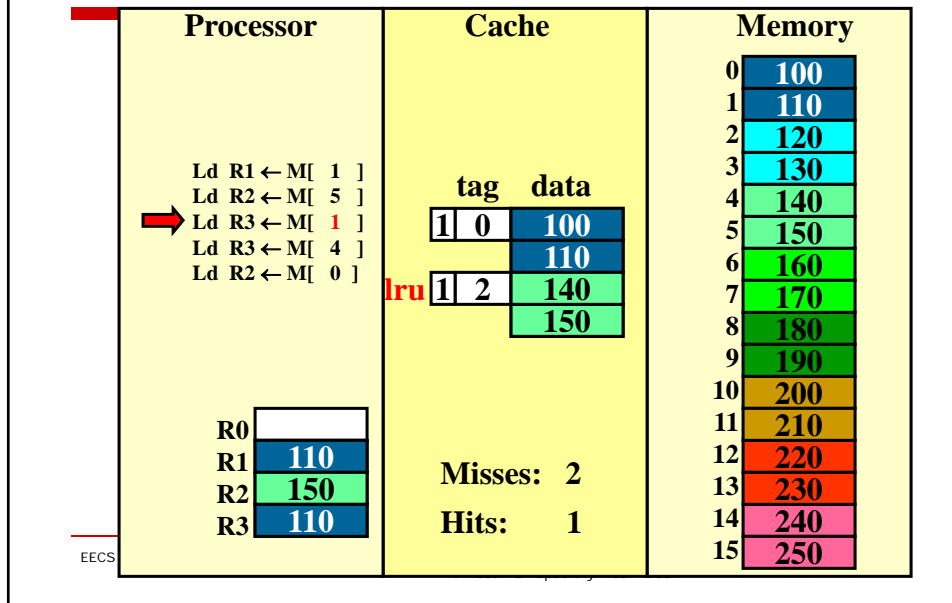
Block size for caches



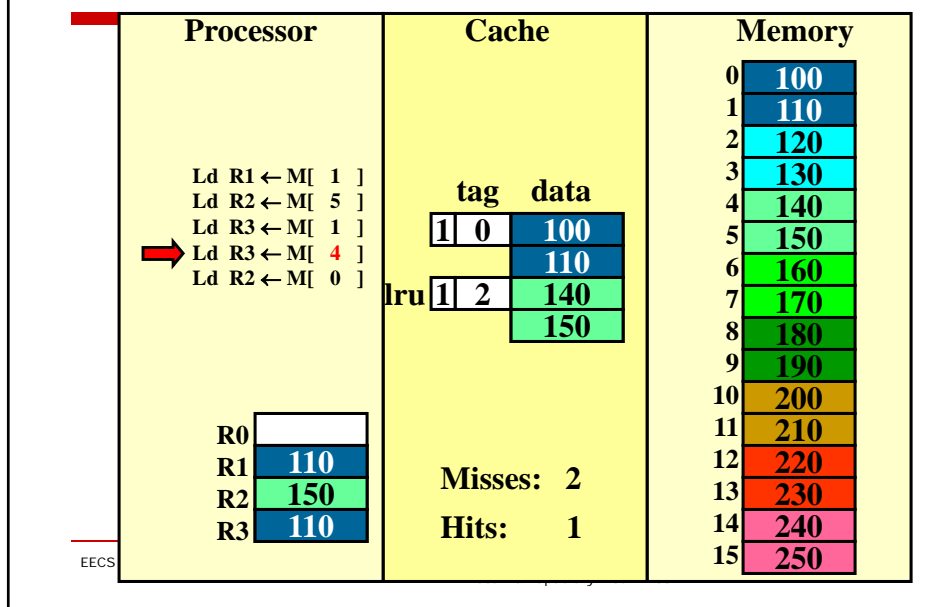
Block size for caches



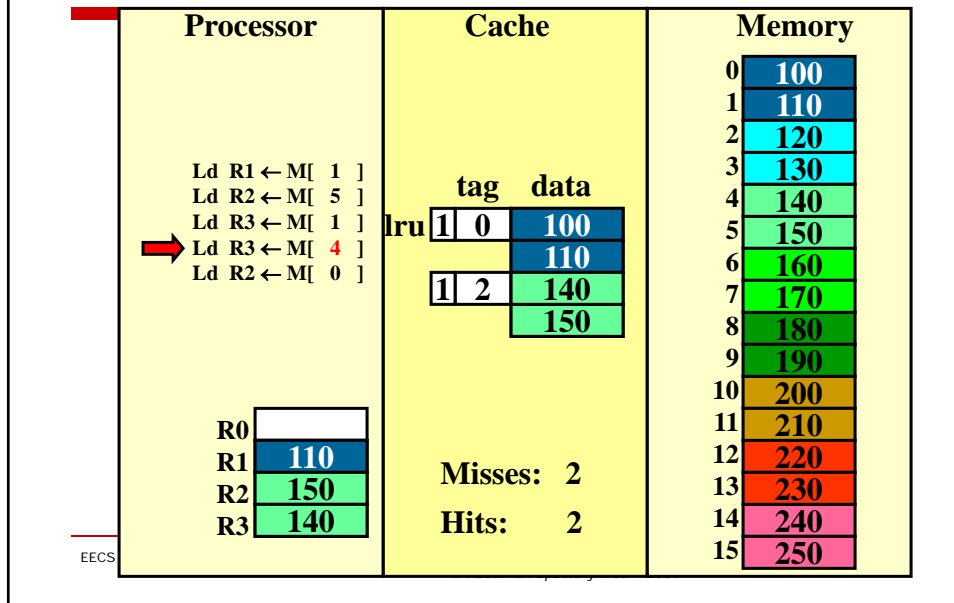
Block size for caches



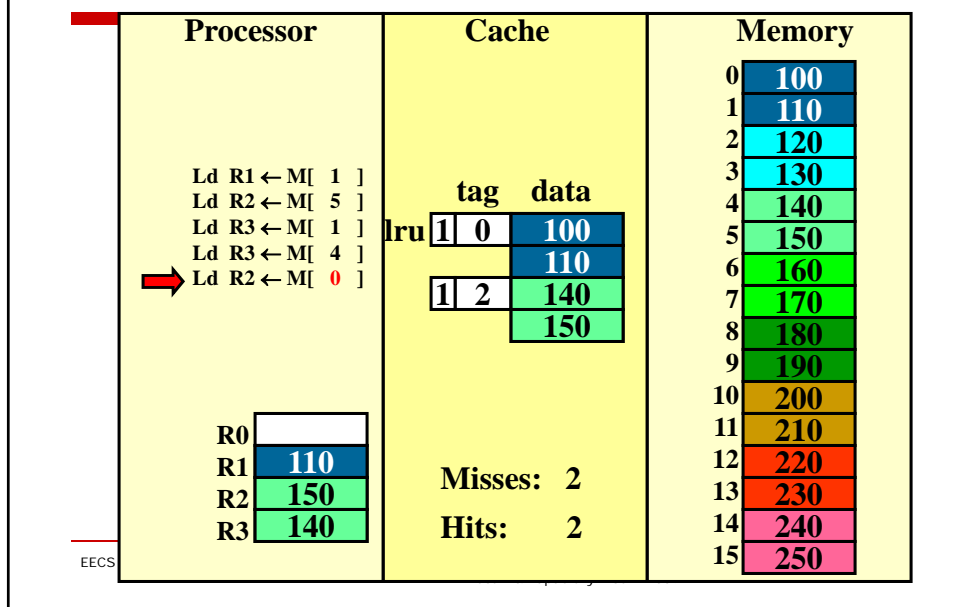
Block size for caches



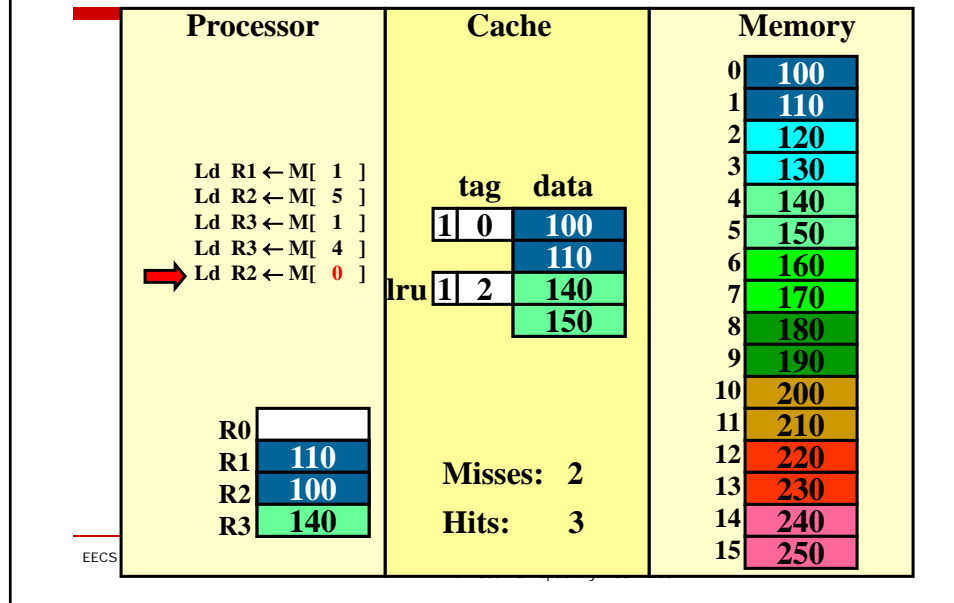
Block size for caches



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.

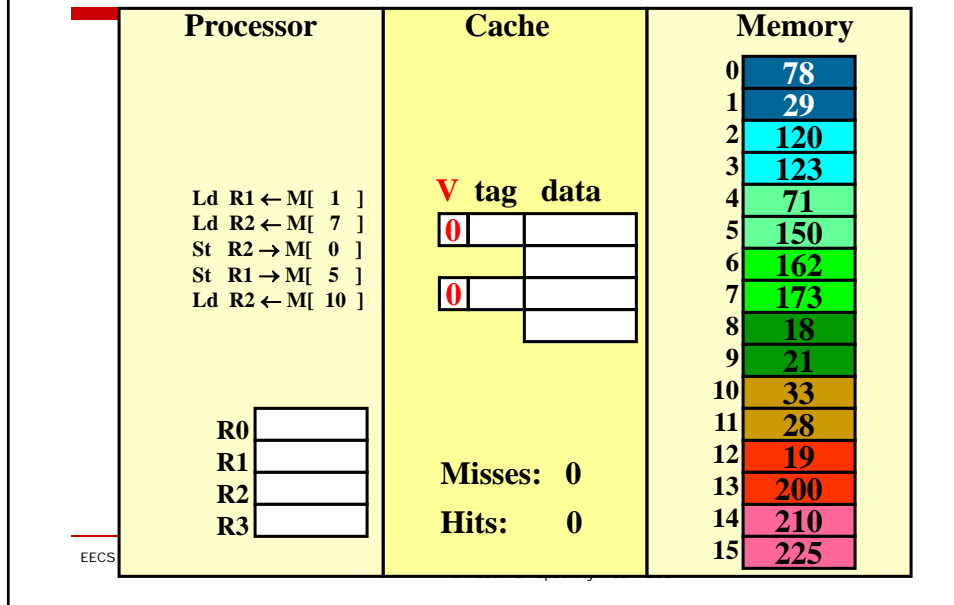
Questions to ask about a cache

- ❑ What is the block size?
- ❑ How many lines?
- ❑ How many bytes of data storage?
- ❑ How much overhead storage?
- ❑ What is the hit rate?
- ❑ What is the latency of an access?
- ❑ What is the replacement policy ?
 - LRU? LFU? FIFO? Random?

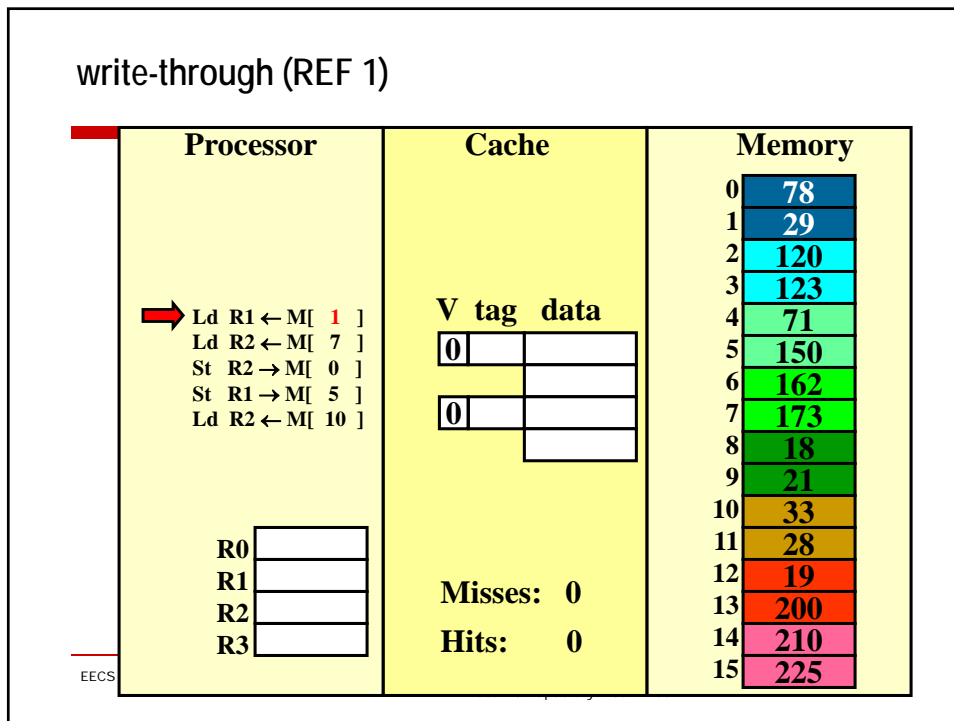
What about stores?

- ❑ Where should you write the result of a store?
 - If that memory location is in the cache?
 - Send it to the cache.
 - Should we also send it to memory?
(write-through policy)
 - If it is not in the cache?
 - Allocate the line (put it in the cache)?
(allocate-on-write policy)
 - Write it directly to memory without allocation?

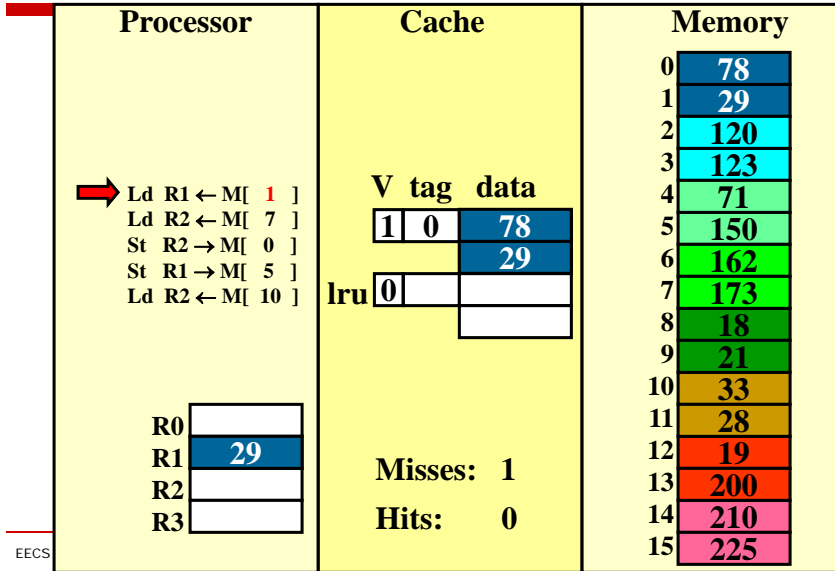
Handling stores (write-through)



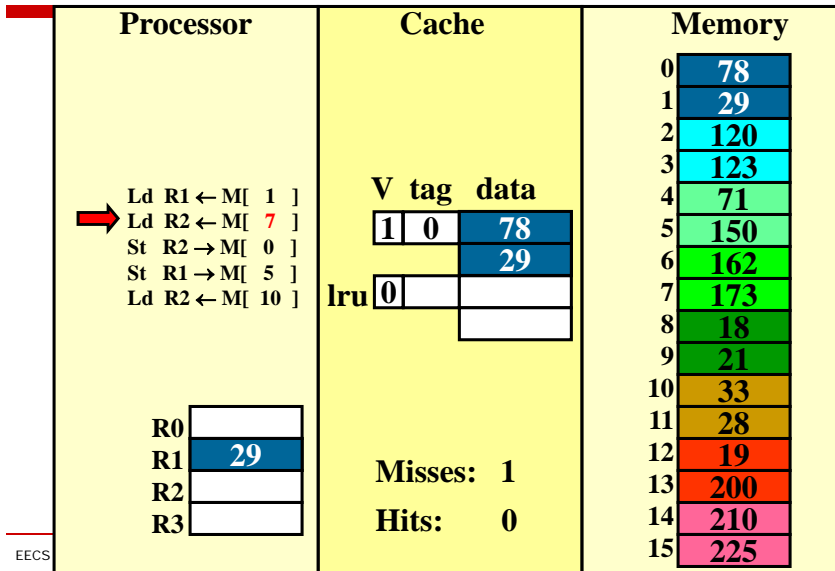
write-through (REF 1)



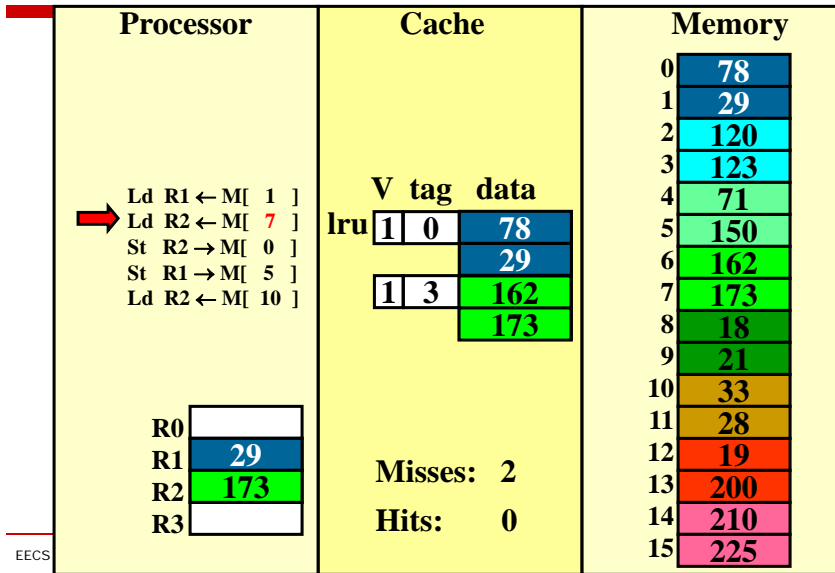
write-through (REF 1)



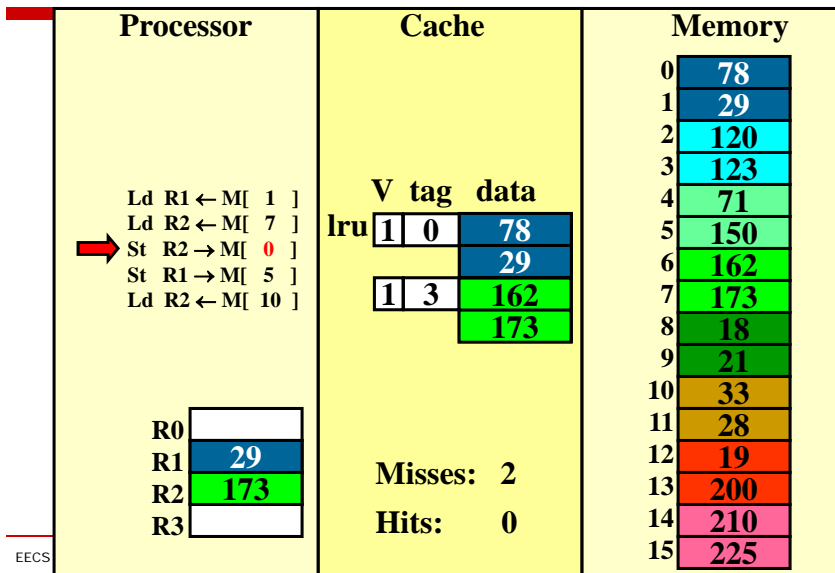
write-through (REF 2)



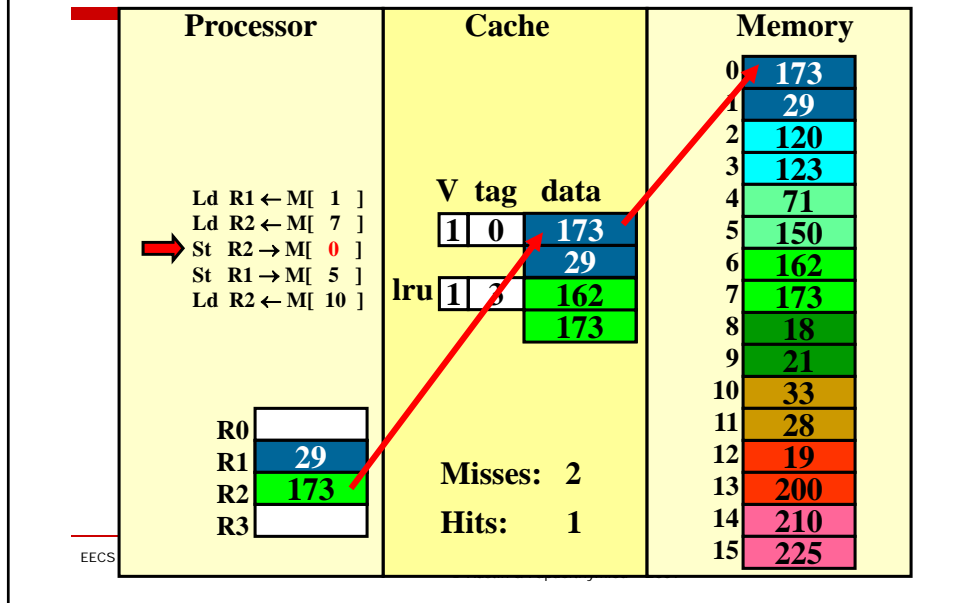
write-through (REF 2)



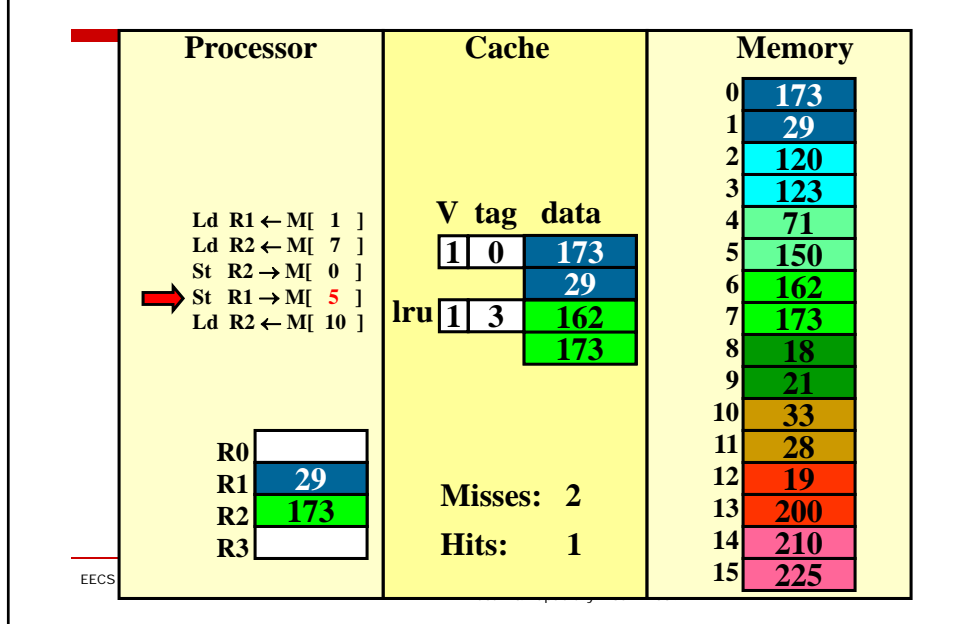
write-through (REF 3)



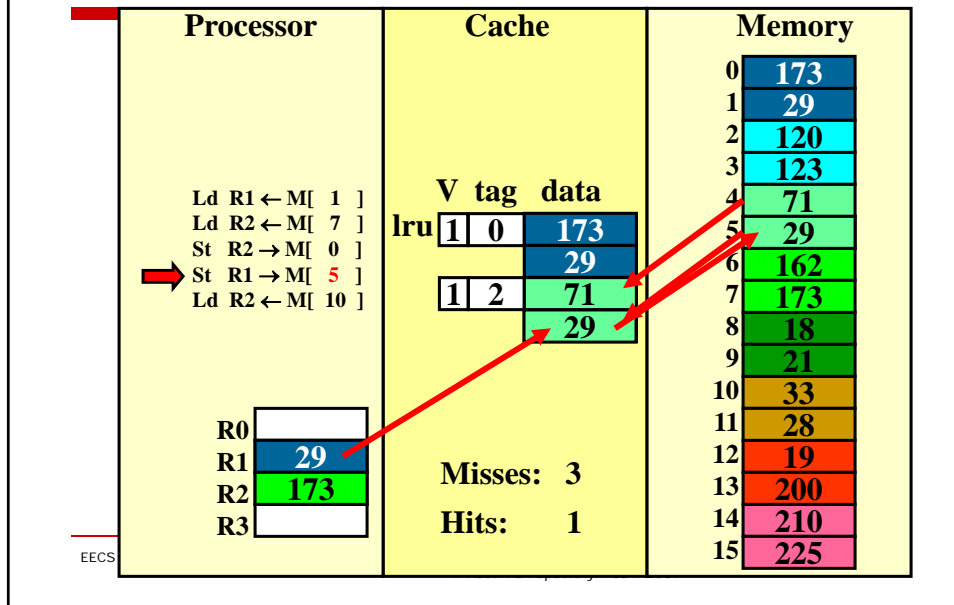
write-through (REF 3)



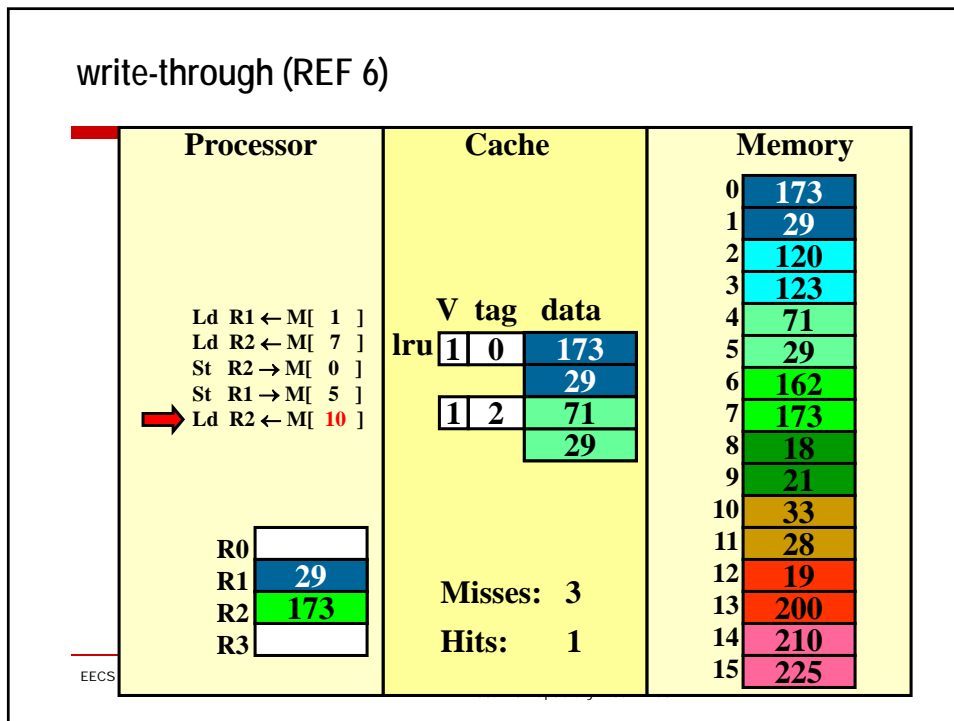
write-through (REF 4)



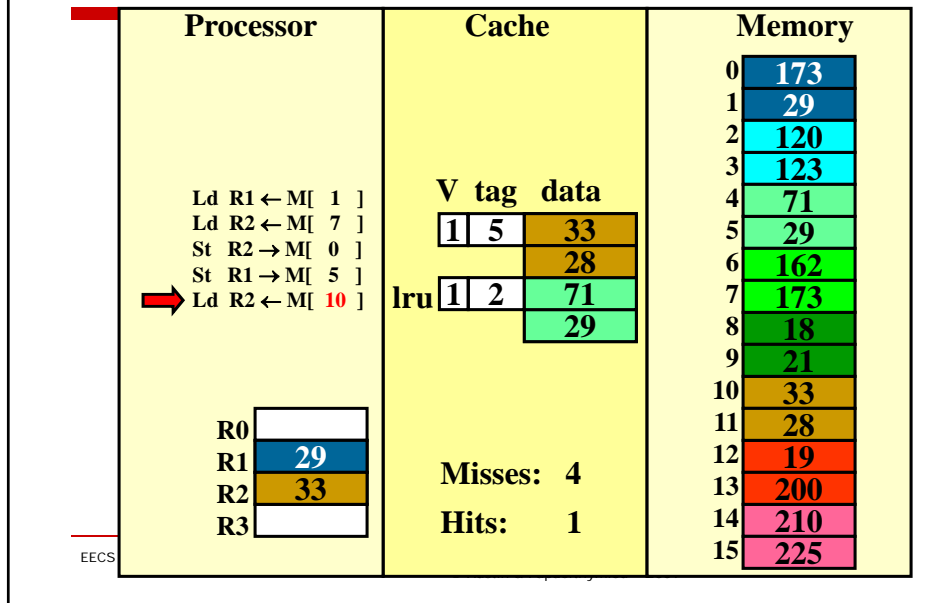
write-through (REF 4)



write-through (REF 6)



write-through (REF 6)



How many memory references?

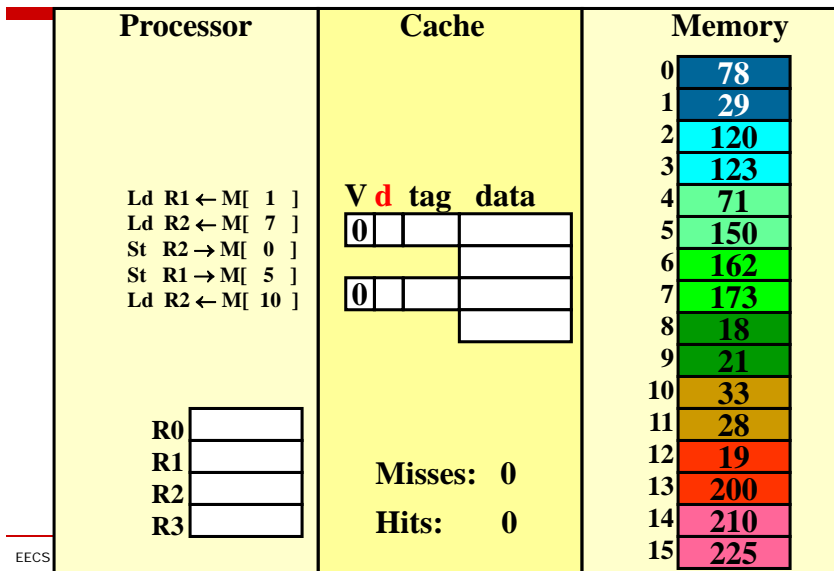
- Each miss reads a block
 - 2 bytes in this cache
- Each store writes a byte
- Total reads: 8 bytes
- Total writes: 2 bytes

but caches generally miss < 20%

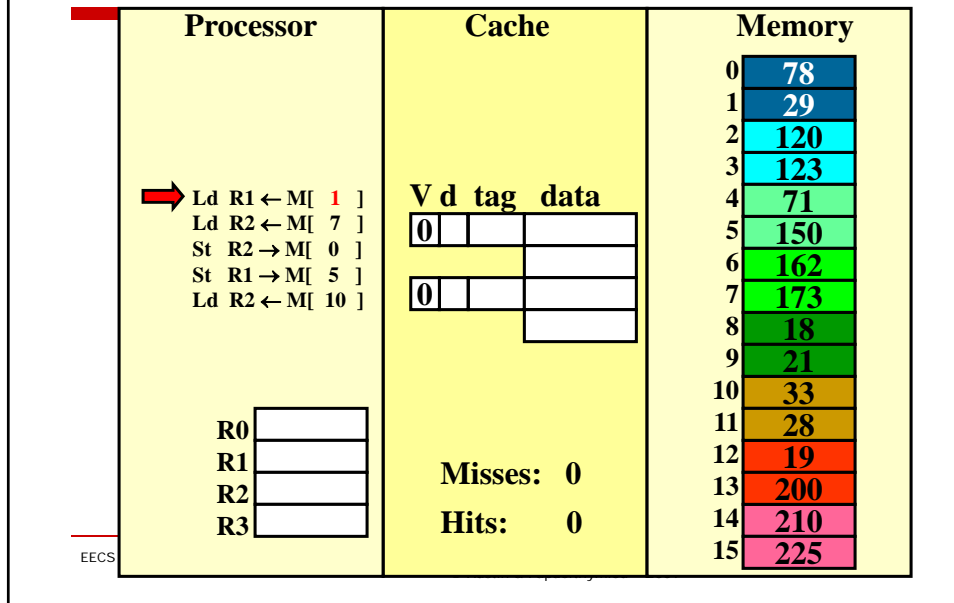
Write-through vs. write-back

- ❑ 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.

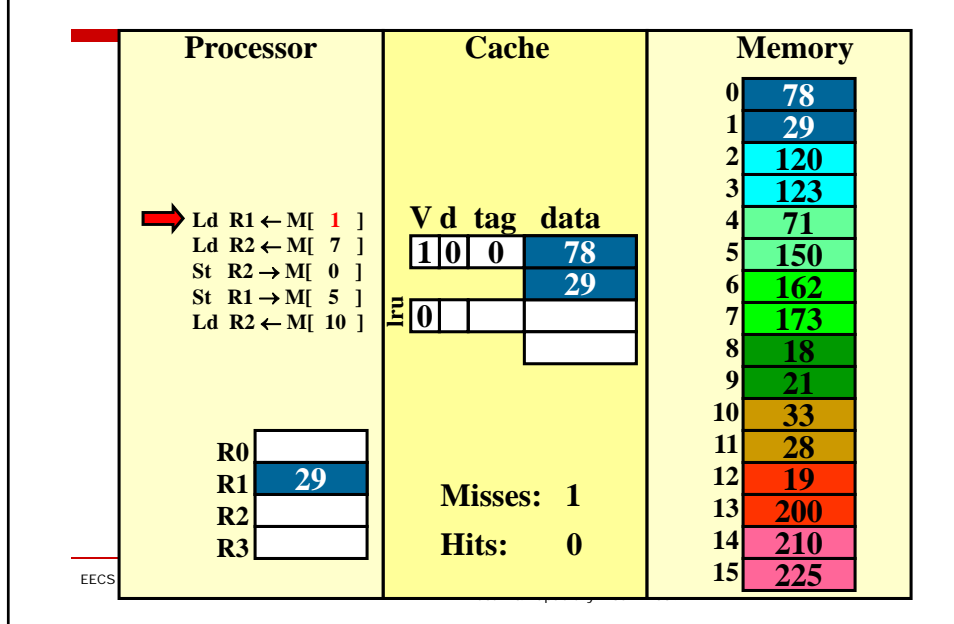
Handling stores (write-back)



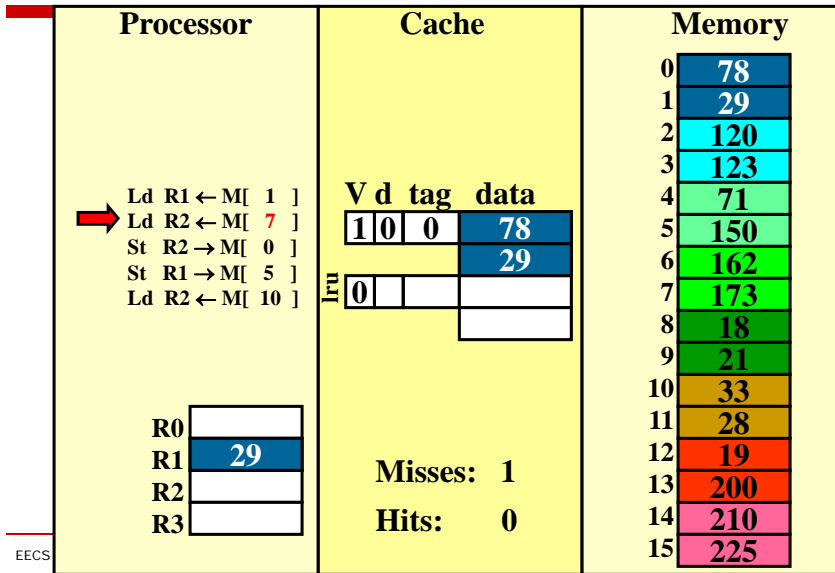
write-back (REF 1)



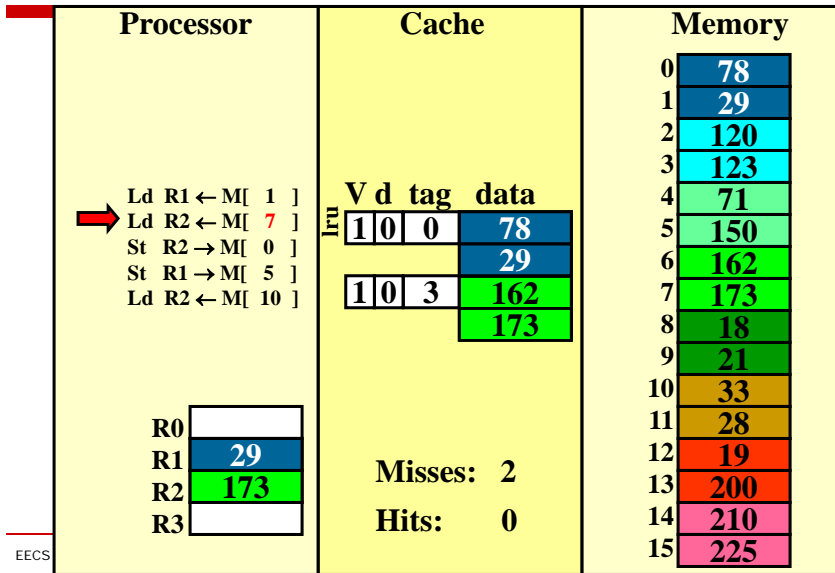
write-back (REF 1)



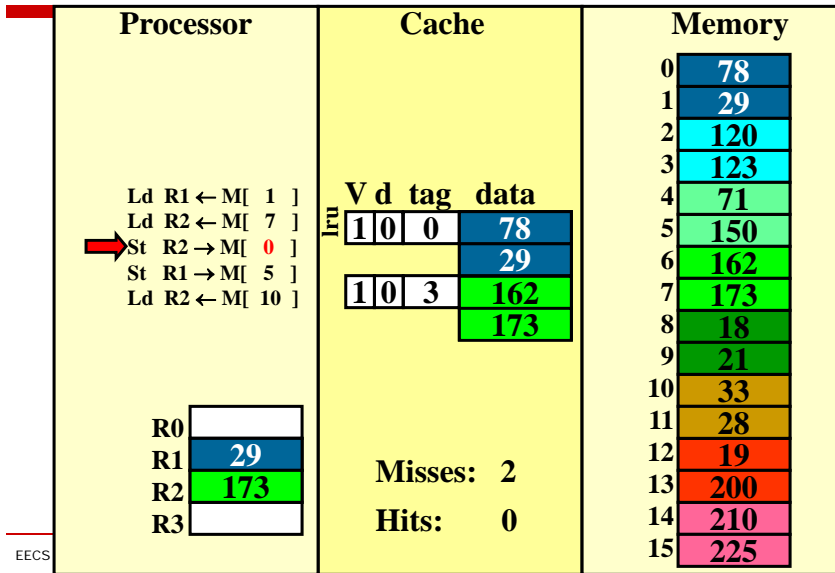
write-back (REF 2)



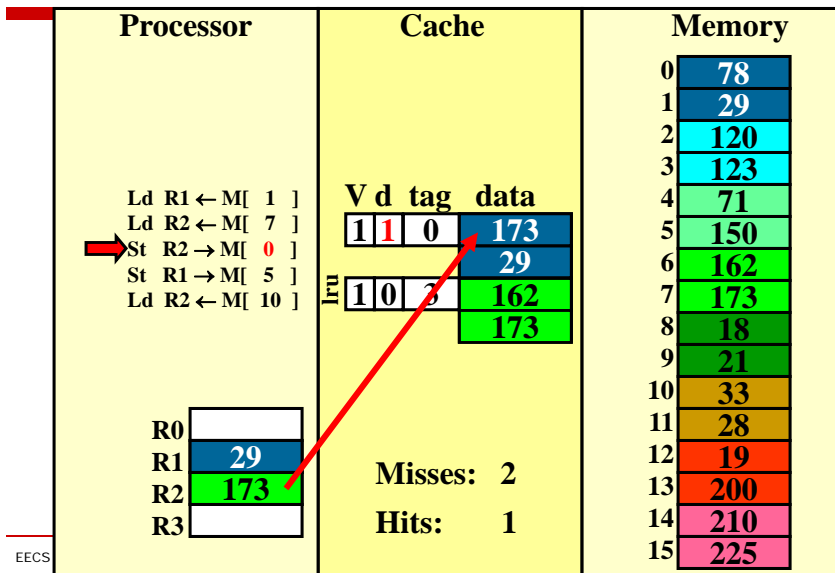
write-back (REF 2)



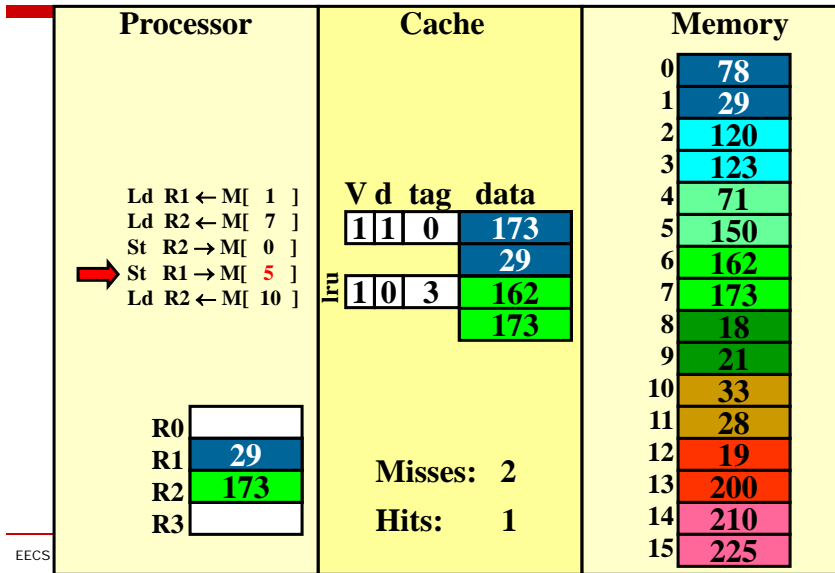
write-back (REF 3)



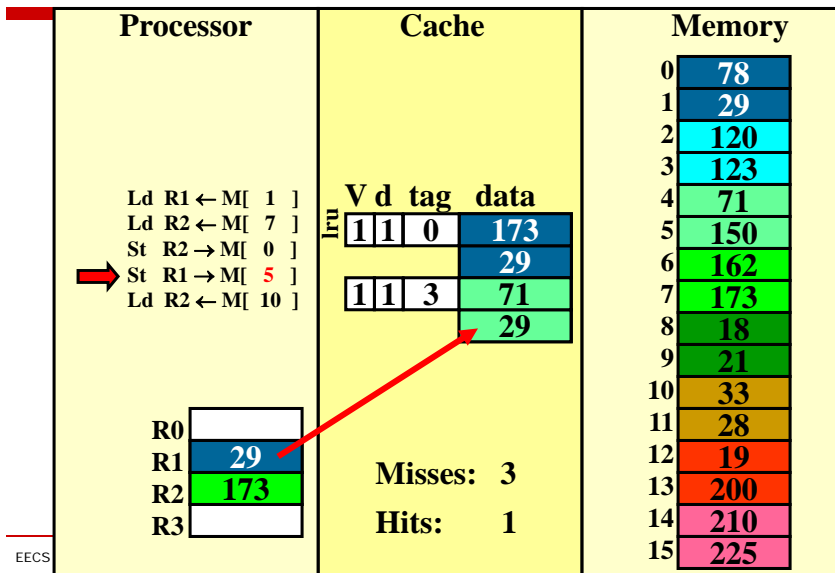
write-back (REF 3)



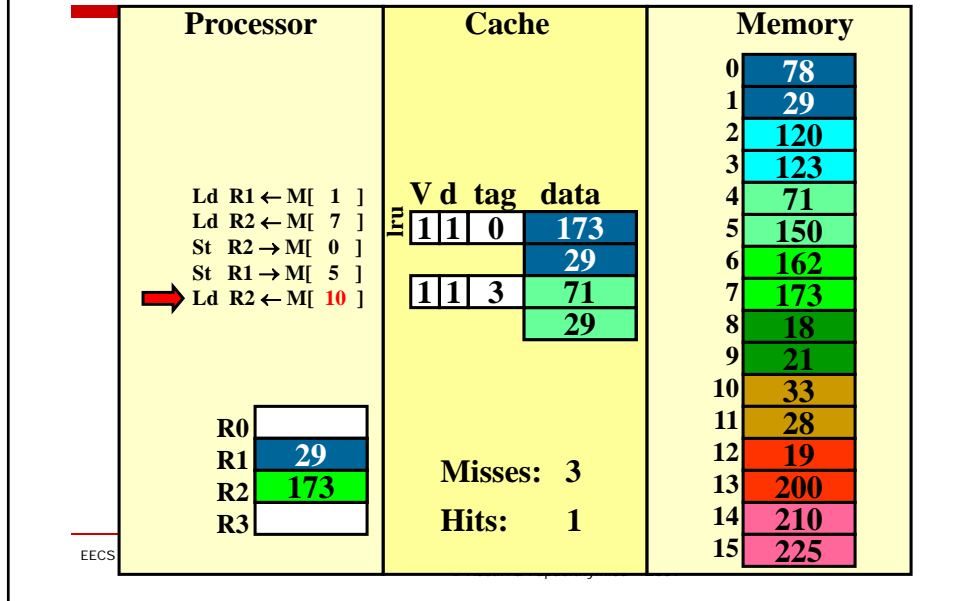
write-back (REF 4)



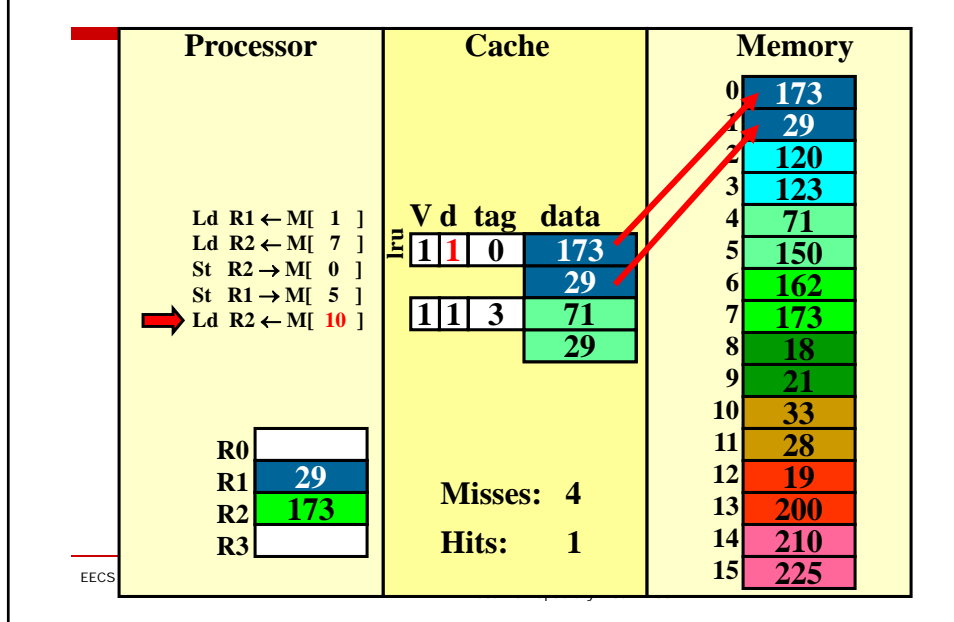
write-back (REF 4)



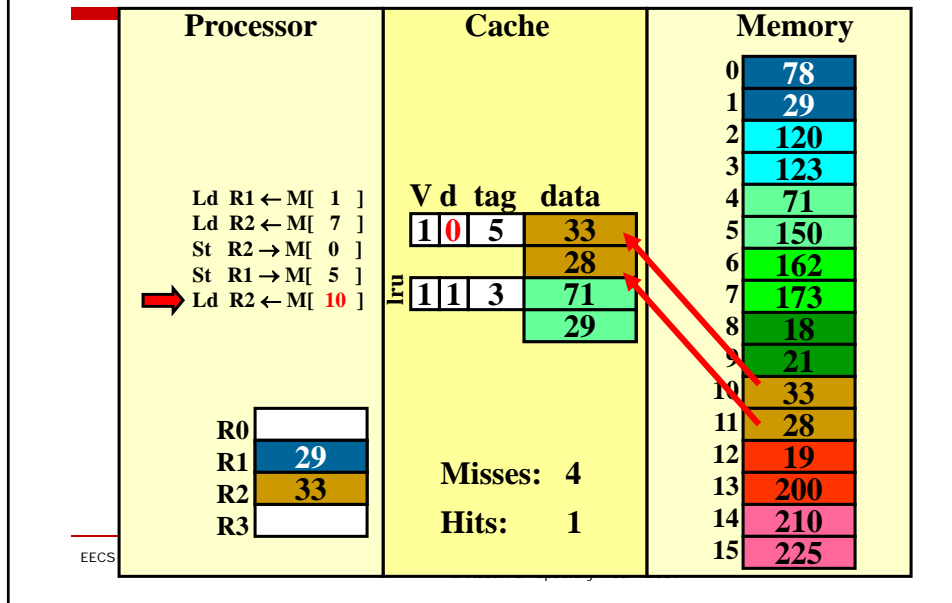
write-back (REF 5)



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?