

## Lecture Plan for Next 2 Weeks

---

1. Today
  - Introduction to the memory systems
  - Basic cache design
2. then
  - Block size
  - Write-back vs write-through caches
3. then
  - Associativity
  - Performance calculation
4. finally
  - Finish up cache performance

## Memory Hierarchy

---

- We want to have lots of memory for our processor:
  - LC2K1 needs  $2^{16}$  words of memory
  - MIPS needs  $2^{32}$  bytes of memory
  - Alpha 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 is big
- It will cost LOTS of money
  - SRAM costs \$10 per megabyte
    - \$1.25 for LC2
    - \$40,960 for MIPS
    - \$175 trillion for Alpha

## Option 2: Build It Out of DRAM

---

- About 50 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 \$0.30 per megabyte
    - \$0.04 for LC2
    - \$1,200 for MIPS/Pentium-IV/Athlon-XP
    - \$5 trillion for Alpha/G5/Itanium/Athlon-64

## Option 3: Build It Using Disks

---

- About 8,000,000 ns access (snore!)
  - We could have stopped with the Intel 4004
- Costs are pretty reasonable
  - Disk storage costs \$0.001 per megabyte
    - Basically free for LC2K1
    - \$4 for MIPS
    - \$17 billion for Alpha (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 1-2 (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!

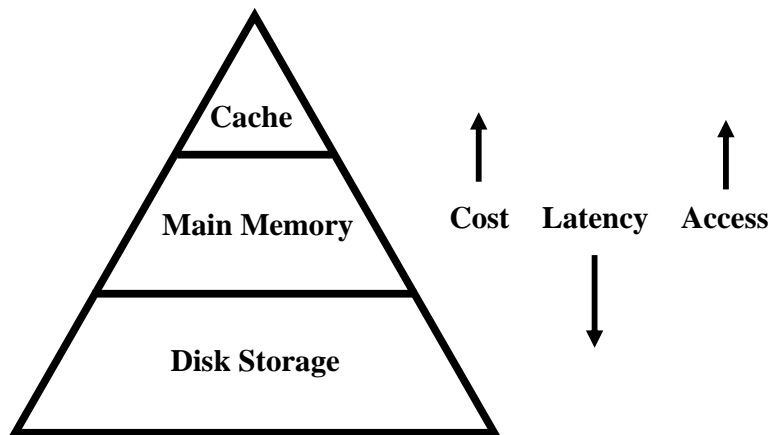
## Option 4: The Memory Hierarchy

---

- Use a small array of SRAM
  - For the [CACHE](#) (hopefully for most accesses)
- Use a bigger amount of DRAM
  - For the [Main memory](#)
- Use a really big amount of Disk storage
  - For the [Virtual memory](#)
- Don't try to buy  $2^{64}$  bytes of anything
  - Common sense!

## 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 → eat!
    - Latency = 1 minute
  - Option 2: go to store
    - Found → purchase, take home, eat!
    - Latency = 20-30 minutes
  - Option 3: grow food!
    - Plant, wait ... wait ... wait ... , harvest, eat!
    - Latency = ~250,000 minutes (~ 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 they don't match, get the data from main memory

## CAMs: Content Addressable Memories

---

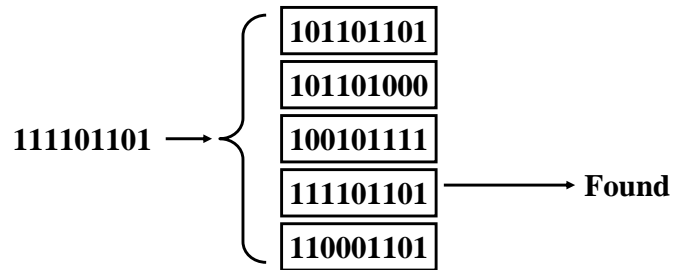
- Instead of thinking of memory as an array of data indexed by a memory address...
- Think of memory as a set of data matching a query.
  - Instead of an address, we send data to the memory, asking if any location contains that data.
  - Memory answers: yes/no (hit/miss for caches)

## Operations on CAMs

---

- Search: the primary way to access a CAM
  - Send data to CAM memory
  - Return “found” or “not found”
  - Alternatively, 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

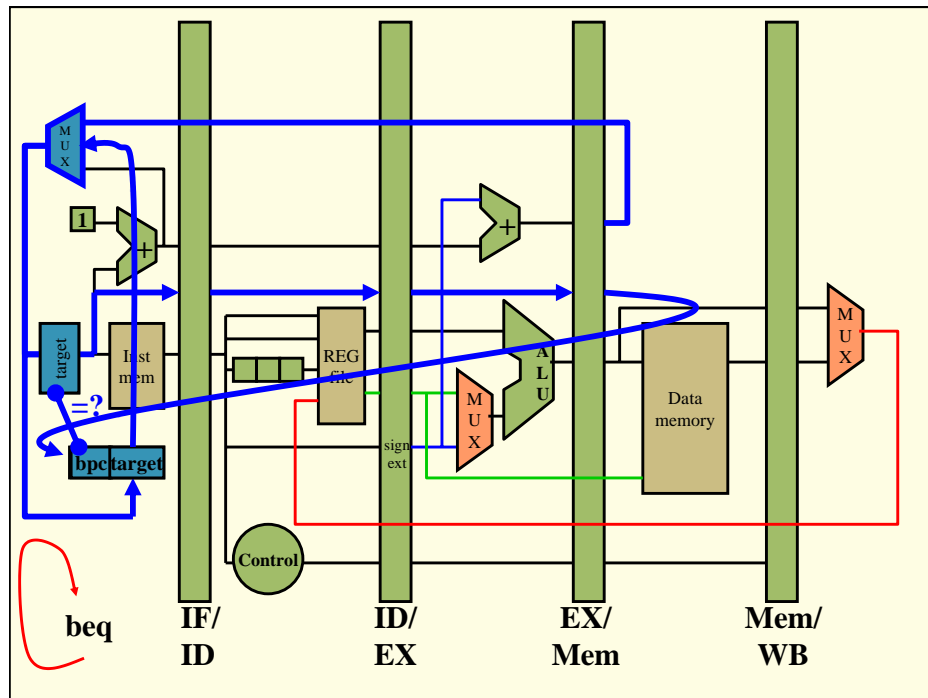
## CAM Array



5 storage element CAM array of 9 bits each

## Previous Use of CAMs

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



## 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

---

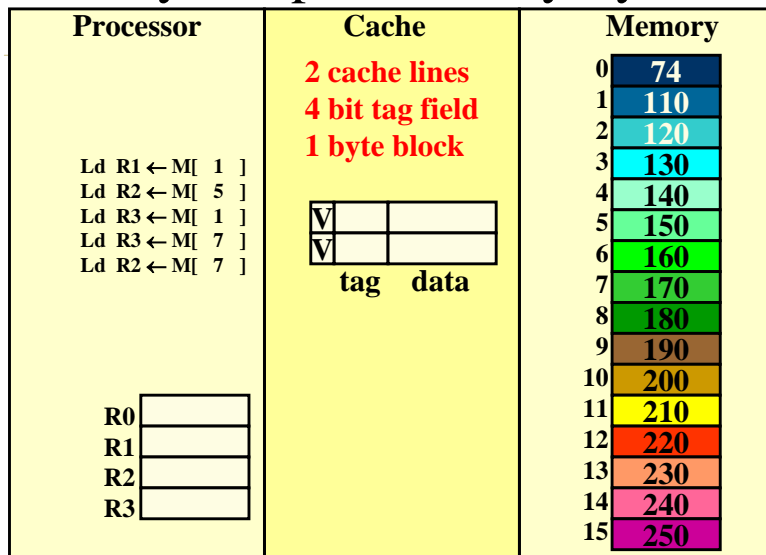
- The principle of **temporal locality** in program references says that if you access a memory location (e.g., 1000) you will be more likely to re-access that location than you will be to reference some other random location.

## Using Locality in the Cache

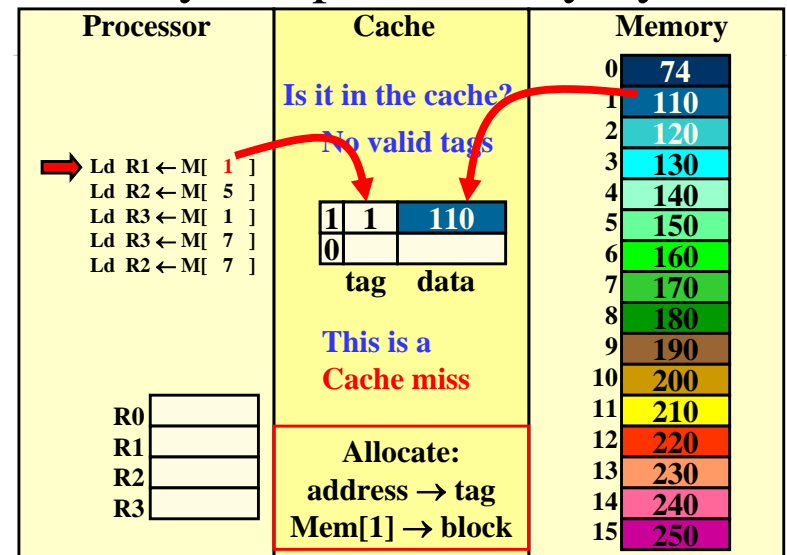
---

- 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.

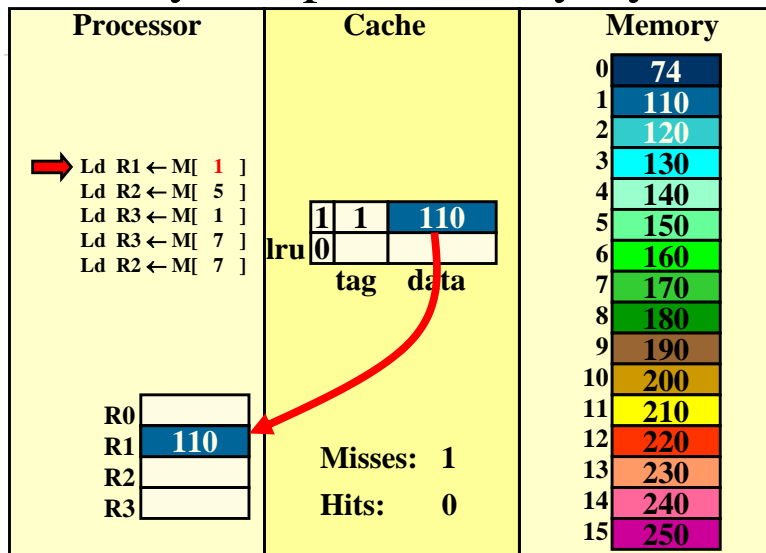
# A Very Simple Memory System



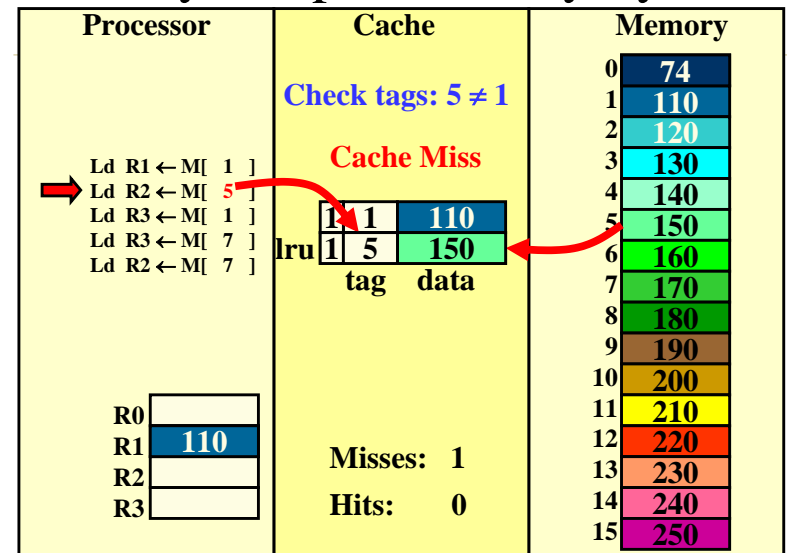
# A Very Simple Memory System



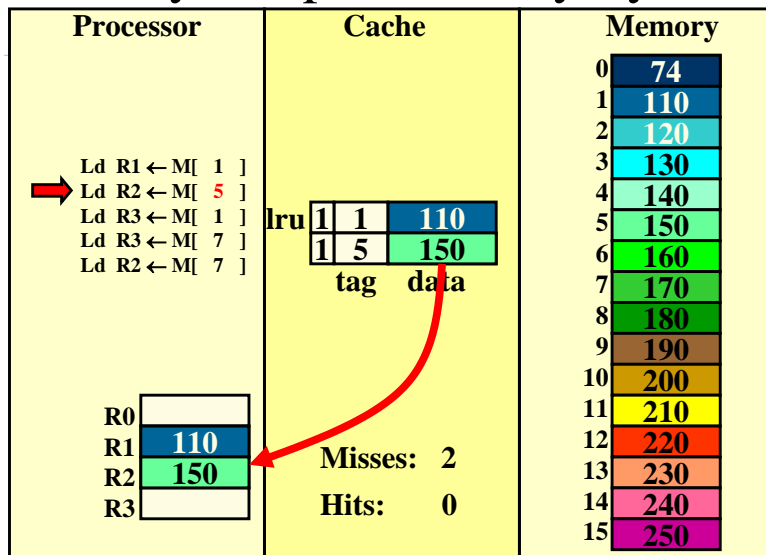
# A Very Simple Memory System



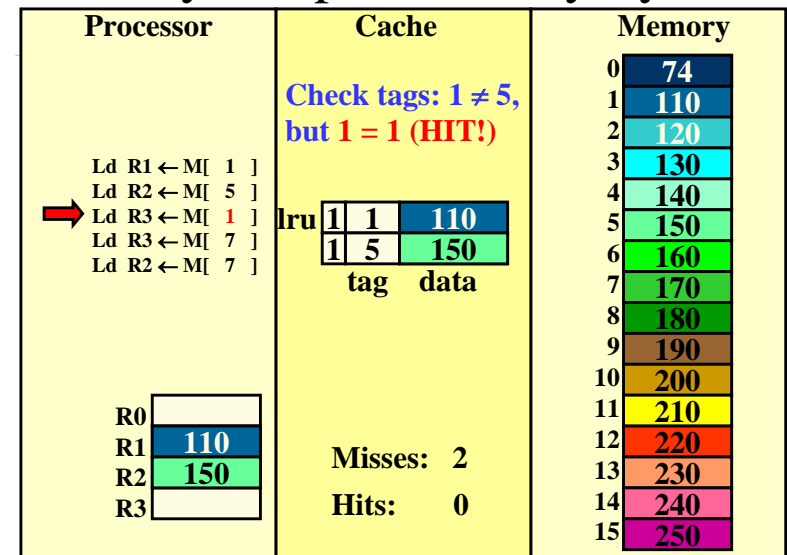
# A Very Simple Memory System



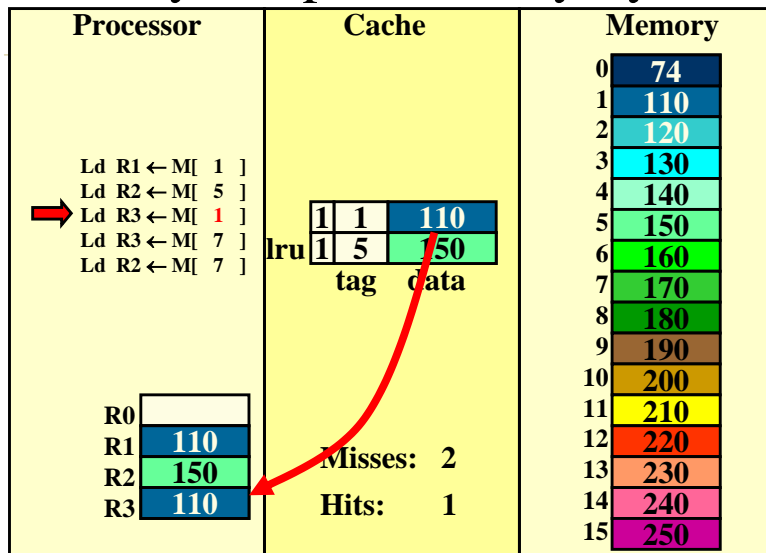
## A Very Simple Memory System



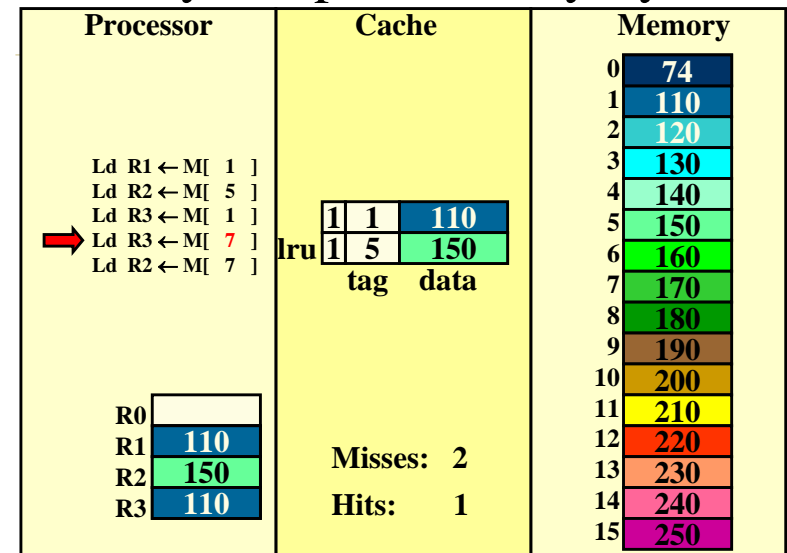
## A Very Simple Memory System



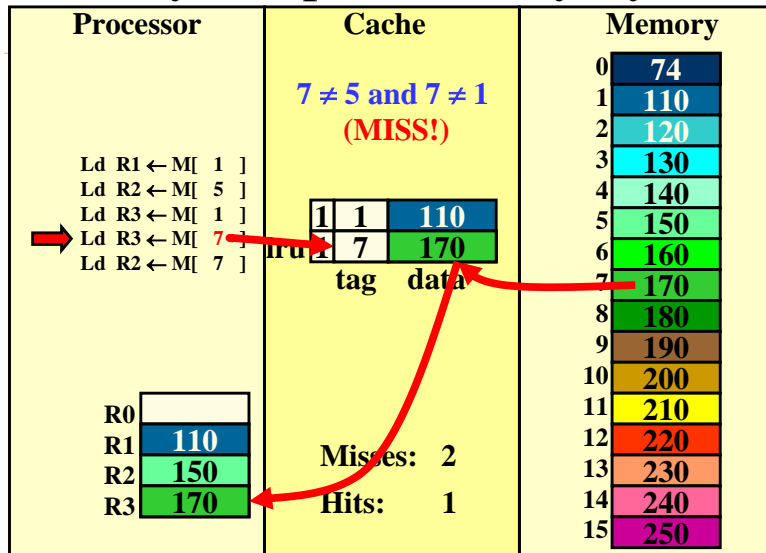
## A Very Simple Memory System



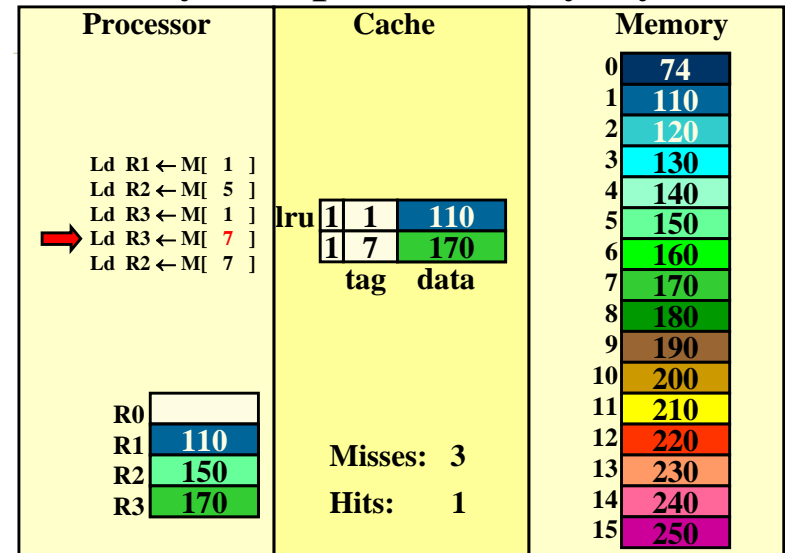
## A Very Simple Memory System



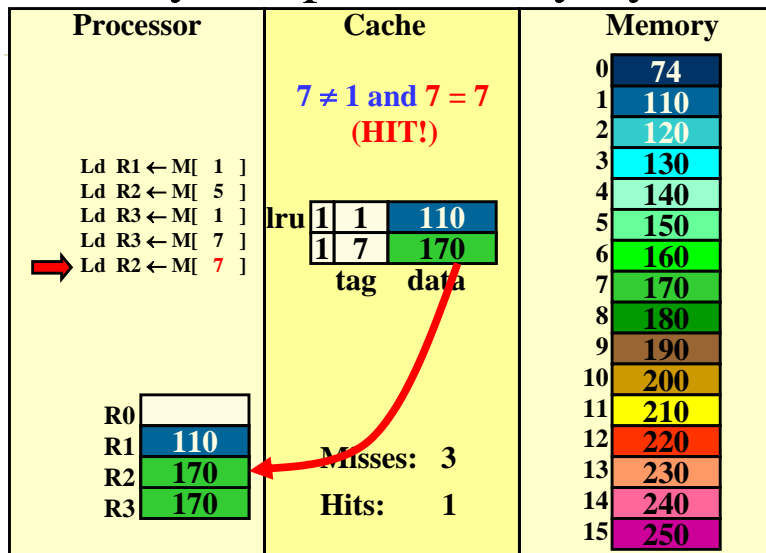
## A Very Simple Memory System



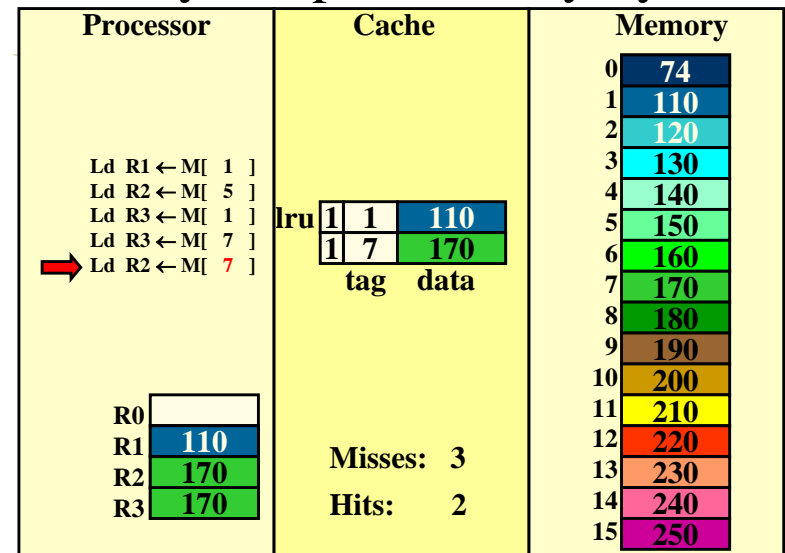
## A Very Simple Memory System



## A Very Simple Memory System



## A Very Simple Memory System



## Something To Think About

---

- Does an optimal replacement policy exist?
  - That is, given a choice of cache lines to replace, which one will result in the fewest total misses during program execution
  - Hint: a crystal ball will come in handy in solving this problem...
- Why would we care?