EECS 370 - Lecture 17

Cache Introduction

Announcements

- P3 published
  - Checkpoint due **Thursday**
    - 5% of project – have pipeline working without data hazards or branches
- Midterm scores should be available later this week
- HW 4 posted, due next Monday

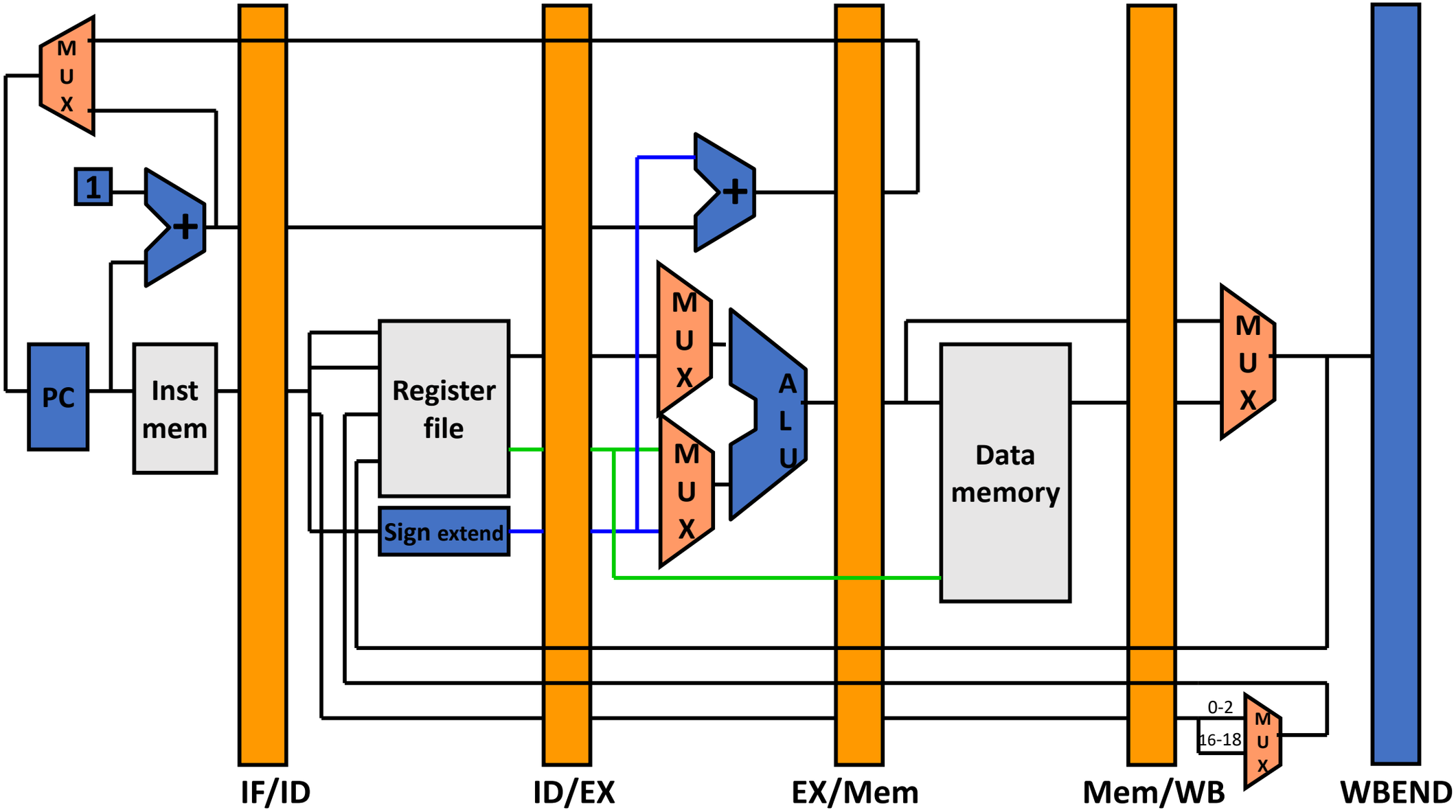
Resources

- Pipelining not clicking? Try playing with the "Pipeline Simulator" under "Resources" on the website
  - https://vhosts.eecs.umich.edu/370simulators/pipeline/simulator.html
  - Several pre-written programs you can step through to understand what's going on
  - Note that the project pipeline is slightly different

Project Pipeline

- Design is largely the same as lecture, except no "internal forwarding" through the register file
  - I.e. we need to explicitly forward from a new pipeline register WBEND if source instruction is in WB while dependent is in Decode

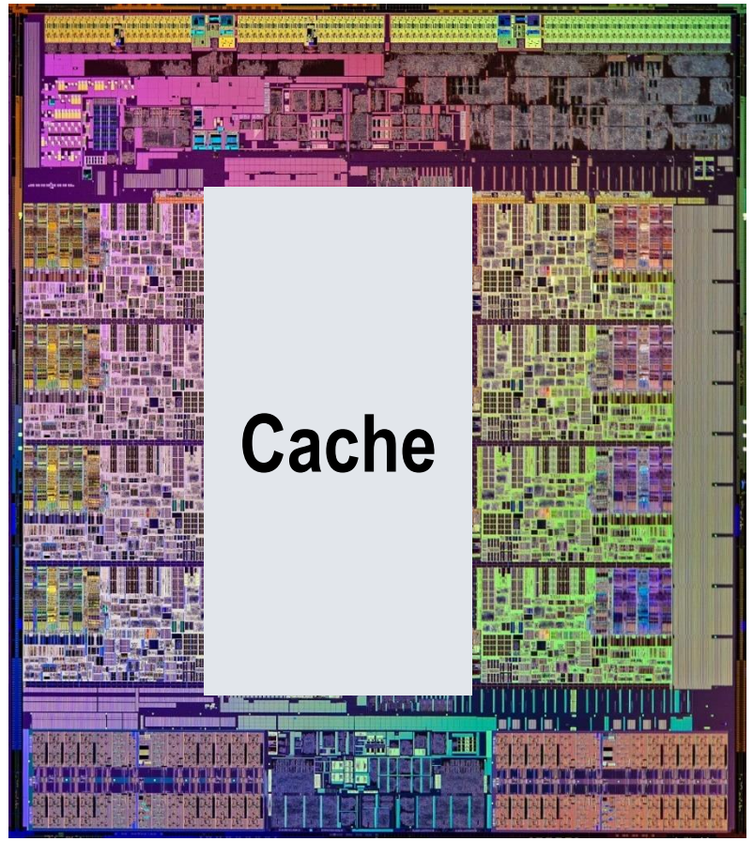
Project 3 Pipeline



EECS 370 Overview

- Part I: Software
- Part II: Processor Design
- Part III: Memory Design
  - Starting with: caches

- If we judge a component's value by how much space it takes up on a chip (not a terrible heuristic), caches are **very** valuable



Cache Aware vs Non-Aware Code

```c
#include<stdio.h>
#include<stdlib.h>

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
  int i, j;
  int count = 0;

  for(i=0; i< N; i++)
    for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[i][j] = 10;
    }

    printf("Count :%d\n", count);
}
```

```c
#include<stdio.h>
#include<stdlib.h>

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
  int i, j;
  int count = 0;

  for(i=0; i< N; i++)
    for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[j][i] = 10;
    }

    printf("Count :%d\n", count);
}
```

Live Poll + Q&A: slido.com #eecs370

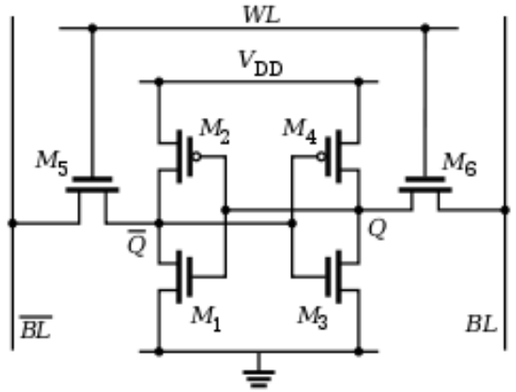
Memory

- So far, we have discussed two structures that hold data:
  - Register file (little array of words)
  - Memory (bigger array of words)

- How do we build this?
  - We need a lot of memory: 2^{18} for LC2K, a lot more for ARM
  - Bunch of flip-flops? Not practical – too many transistors, would be huge and power hungry
  - Other, clever ways of storing bits

Option 1: SRAM

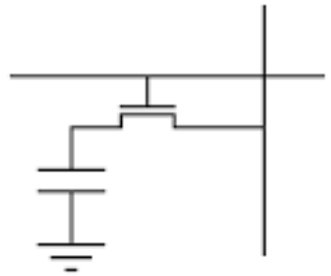
- Each bit is made of 6 transistors
- Volatile: need constant power to keep data
- Fast: ~1 ns read time
- Still rather large and expensive
  - ~$5 per megabyte
- Impractical to scale up to GBs needed for modern systems



Replace cost with how much we can fit on chip

Option 2: DRAM

- Each bit is made of a transistor and capacitor
- Volatile: need constant power to keep data
- Slower: ~50 ns access time
  - Must stall for dozens of cycles on each memory load



- Less expensive for SRAM
  - ~$0.004 per megabyte
  - We can put up to ~16 GB in current machines
  - Good for LC2K or 32-bit systems
  - But not for 64-bit systems

Option 3: Disks

- Hard-drives store bits as magnetic charges on spinning disks
  - Non-volatile – holds information even when no power is supplied
  - Obnoxiously slow compared to digital logic: 3,000,000 ns access time
- Recently, solid-state drives have replaced spinning disks with logic gates replacing mechanical systems
  - Also non-volatile
  - Much better speeds (3,000,000 ns), but still to slow to keep up with processor
- Cheap!
  - SSDs cost $0.0001 per megabyte
  - Scale up to terabytes – practical for modern computing

Memory Goals

- Fast: Ideally run at processor clock speed
  - 1 ns access
- Cheap: Ideally free
  - Not more expensive than rest of system

- DRAM, hard-drives and SSDs are too slow
- SRAM is too expensive

- How to get best properties of multiple memory technologies?

Memory Hierarchy

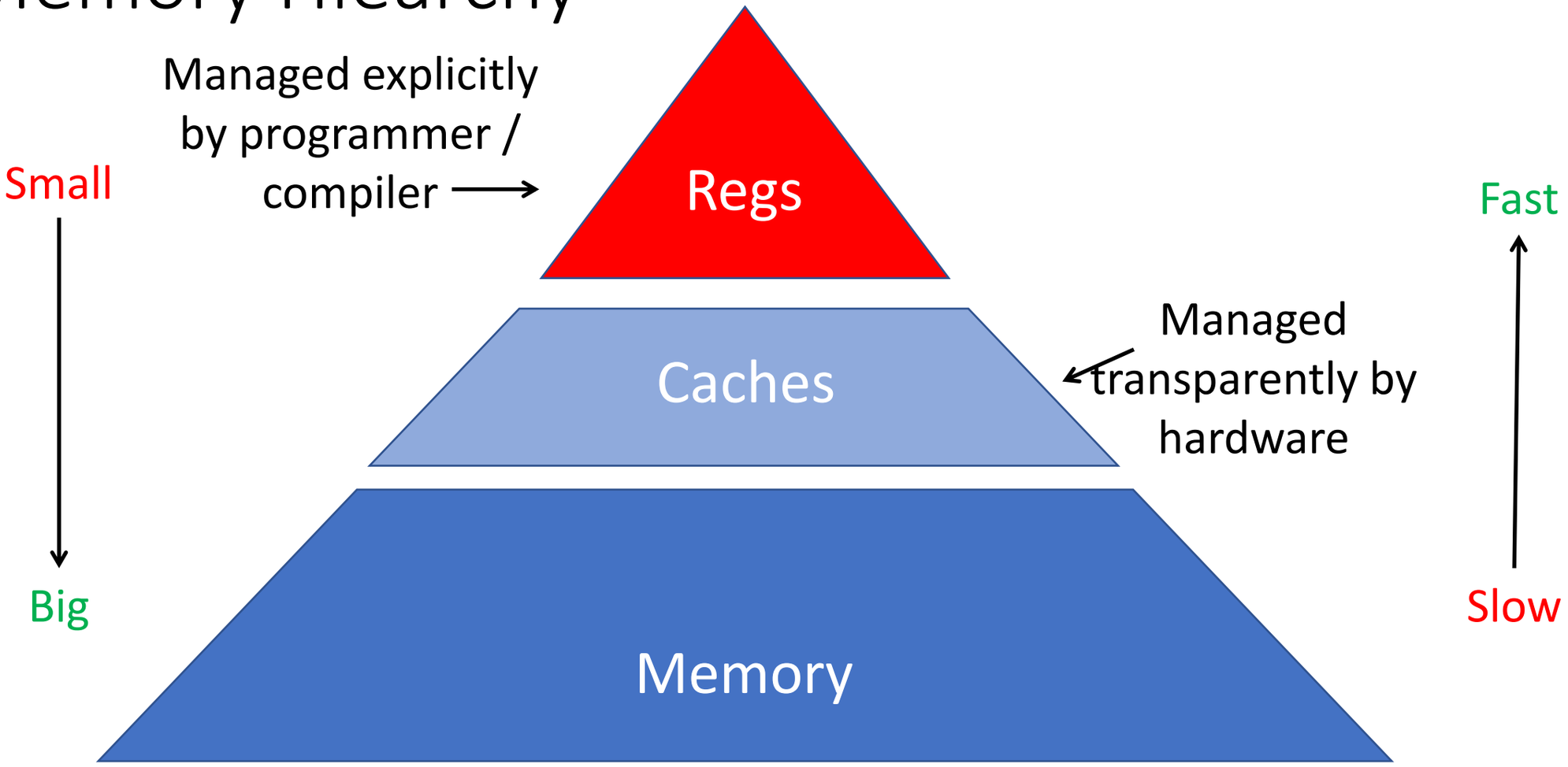
- Key observation: we only need to access a small amount of data at a time

- Let's use a small array of SRAM to hold data we need now
  - Call this the **cache**
  - Able to keep up with processor
  - Small (~Kilobytes), so it should be relatively cheap
- Use a large amount of DRAM for **main memory**
  - Can scale up to ~Gigabytes in size
- Everything else, store on disk
  - **Virtual Memory**
- Won't end up building 2^{64} of anything
  - Won't be needed in typical programs
  - Virtual memory (discussed in a couple weeks) will make memory look larger than it is

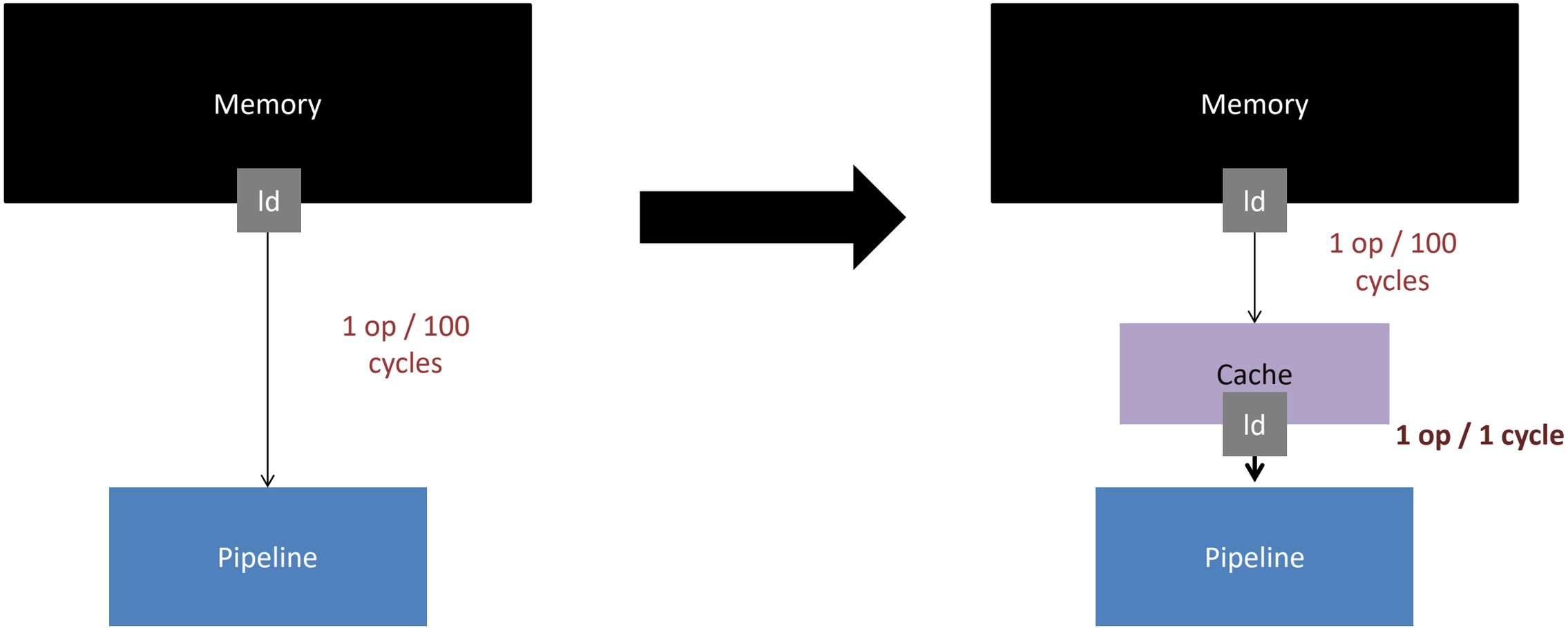
Cache Analogy

- Studying in the library
  - **Option 1: Every time you grab another book, return current book to shelf and get a new book from shelf**
    - Latency = 5 minutes
  - **Option 2: Keep 10 commonly-used books on shelf above desk**
    - Latency = 1 minute
  - **Option 3: Keep 3 books open on different locations on desk**
    - Latency = 10 seconds

Memory Hiearchy



Caches - Overview

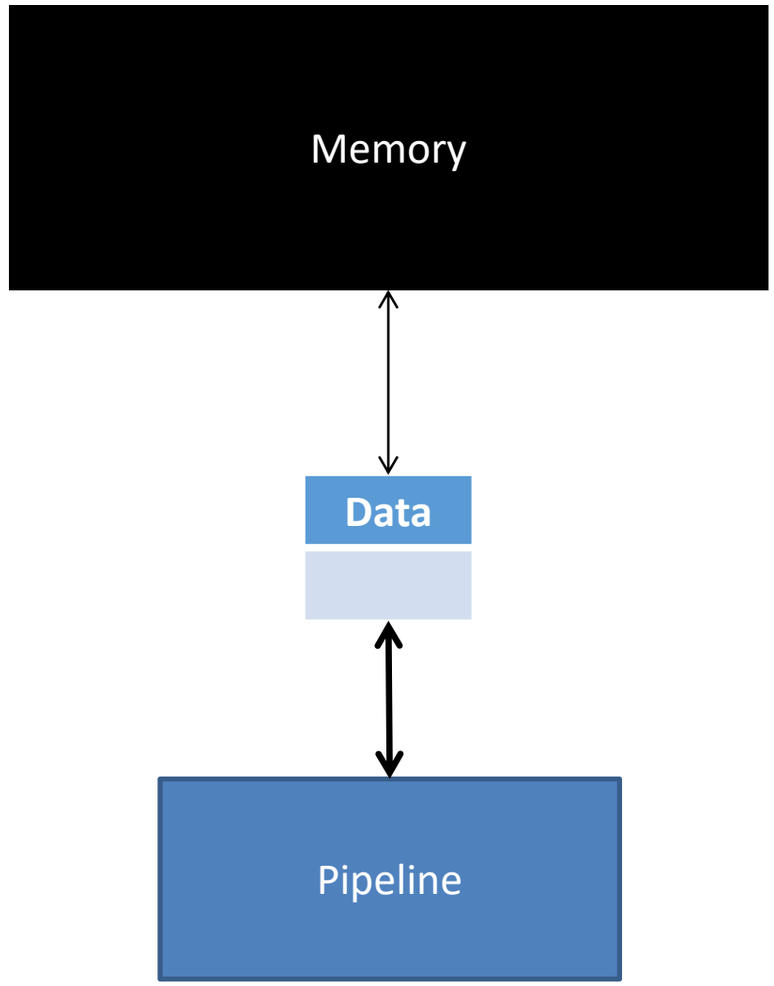


Function of the Cache

- The cache holds the data we think is **most likely** to be referenced
  - The more often the data we want is in the fast cache, the lower our **average memory access latency** is
  - How do we decide what the most likely accessed memory locations are?

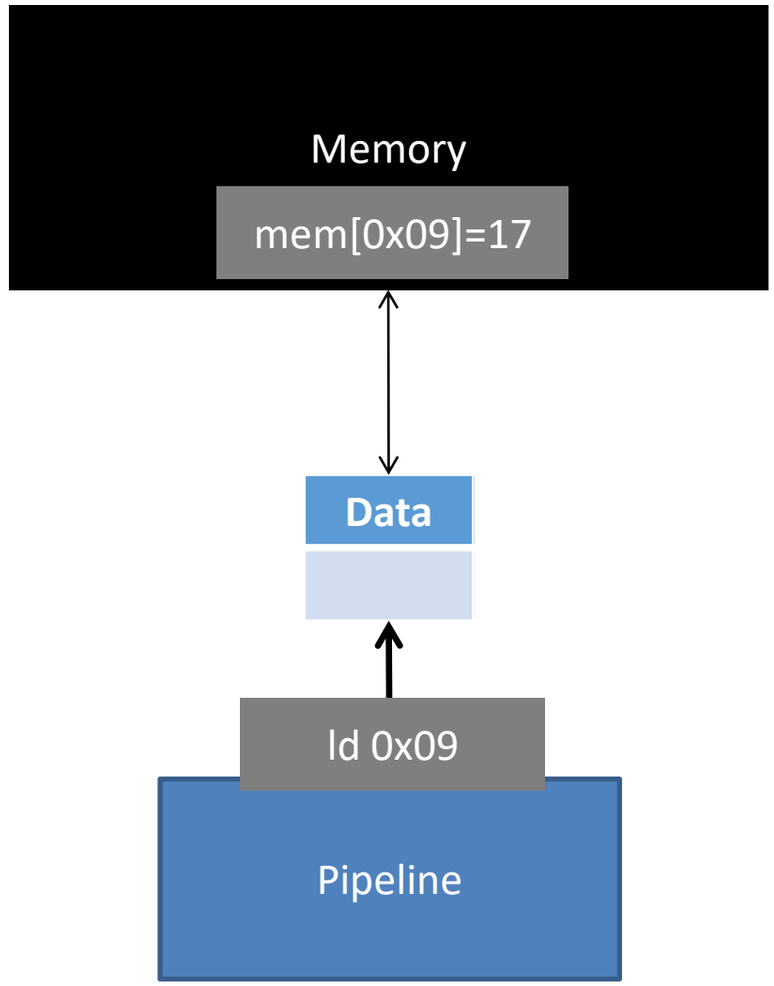
The simplest cache

- Only worry about load instructions for now
- Word-addressable address space
- Consists of a single, word-size storage location to remember last loaded value



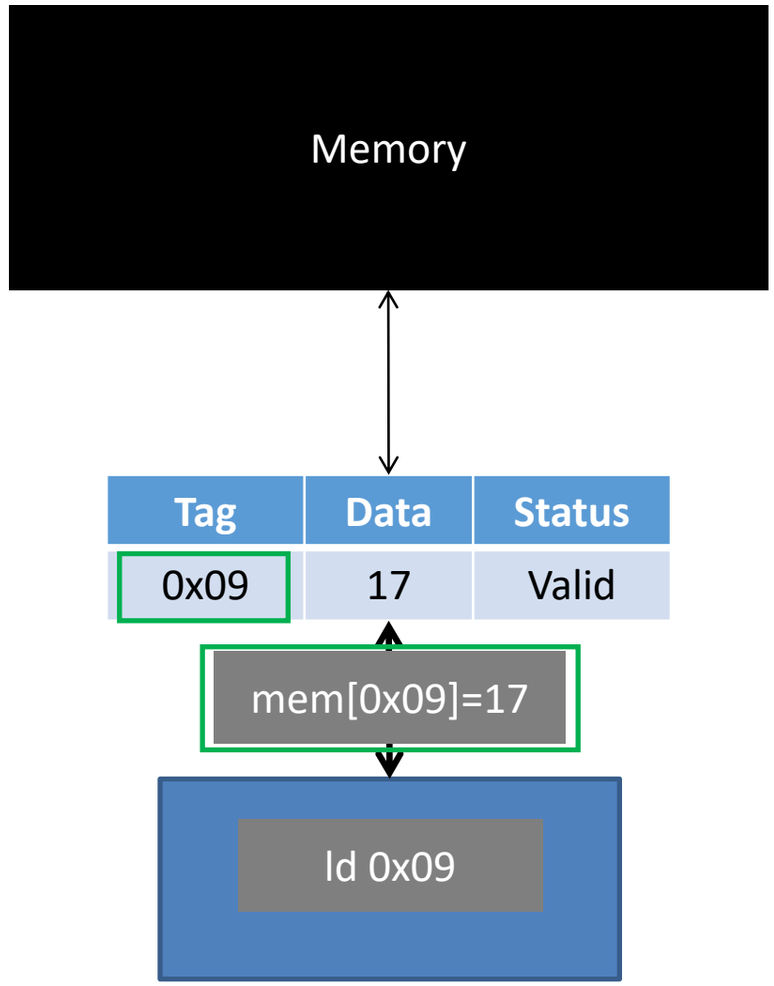
The simplest cache

- Whenever memory returns data, store it in the cache
- We'll also need to know what address this data corresponds to
  - Store that as "**tag**"
- Also include a "**valid**" status bit



The simplest cache

- Next memory access, first check if the tag matches address
  - Yes? Return cache data
  - No? Go to memory as before



Definitions

- Hit: when data for a memory access is found in the cache
- Miss: when data for a memory access is not found in the cache
- Hit/Miss rate: percentage of memory accesses that hit/miss in the cache

Example Problem

Poll: Average access time?

- Assume the following:
  - Cache has 1 cycle access time
  - If data is not found in cache, main memory is then accessed instead
  - Main memory has 100 cycle access time

- If we have a 90% hit rate in the cache, what is the average memory latency?

1 + 0.1 * (100) = 211

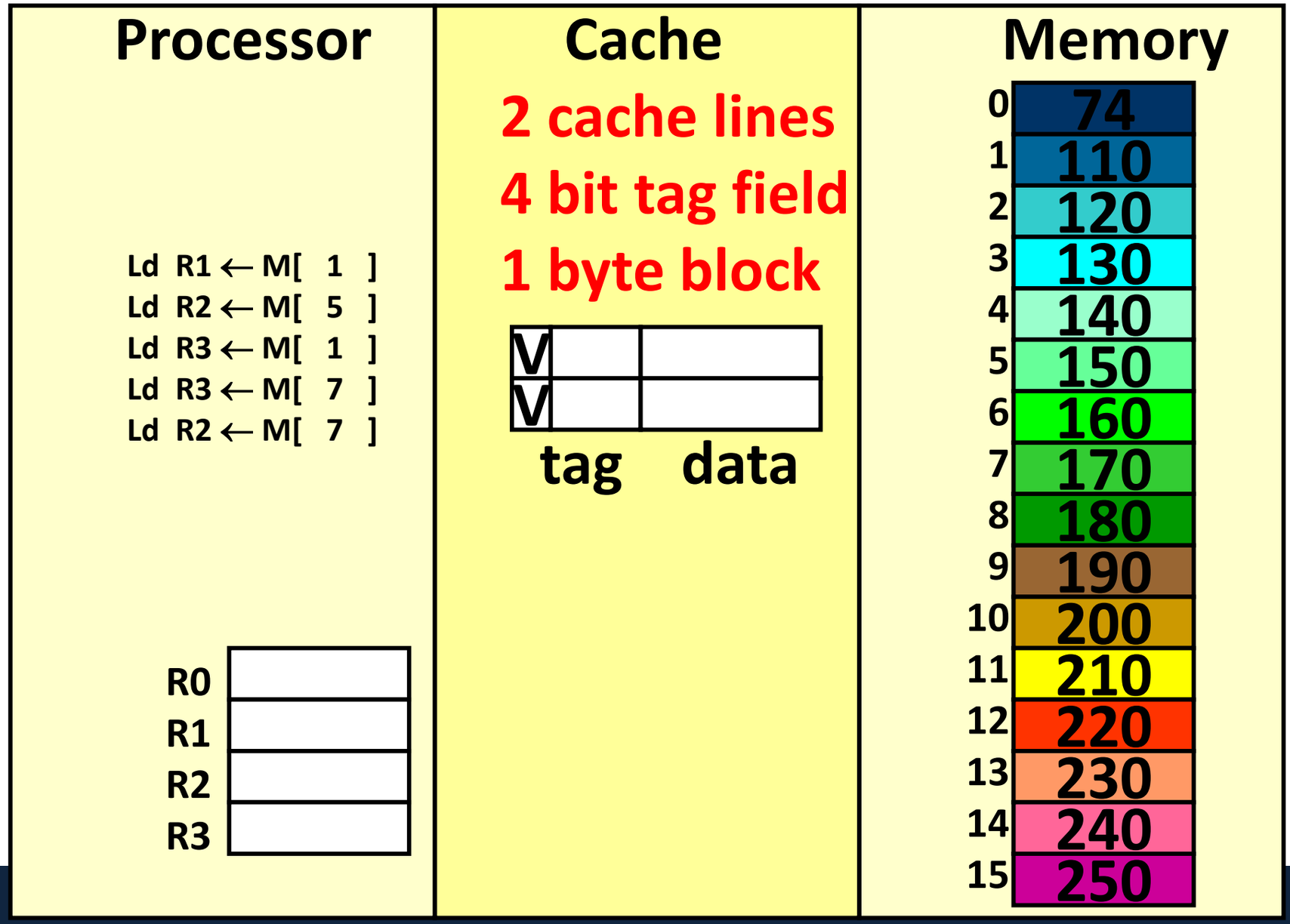
Scaling Up

- Of course, we don't just access one memory location
- What if we access many memory locations, and cache isn't large enough to hold all of them?
- How do we choose what to keep in the cache?
  - How does hardware predict what's most likely to be needed soon?
- Answer: **locality**
  - Temporal locality
  - Spatial locality

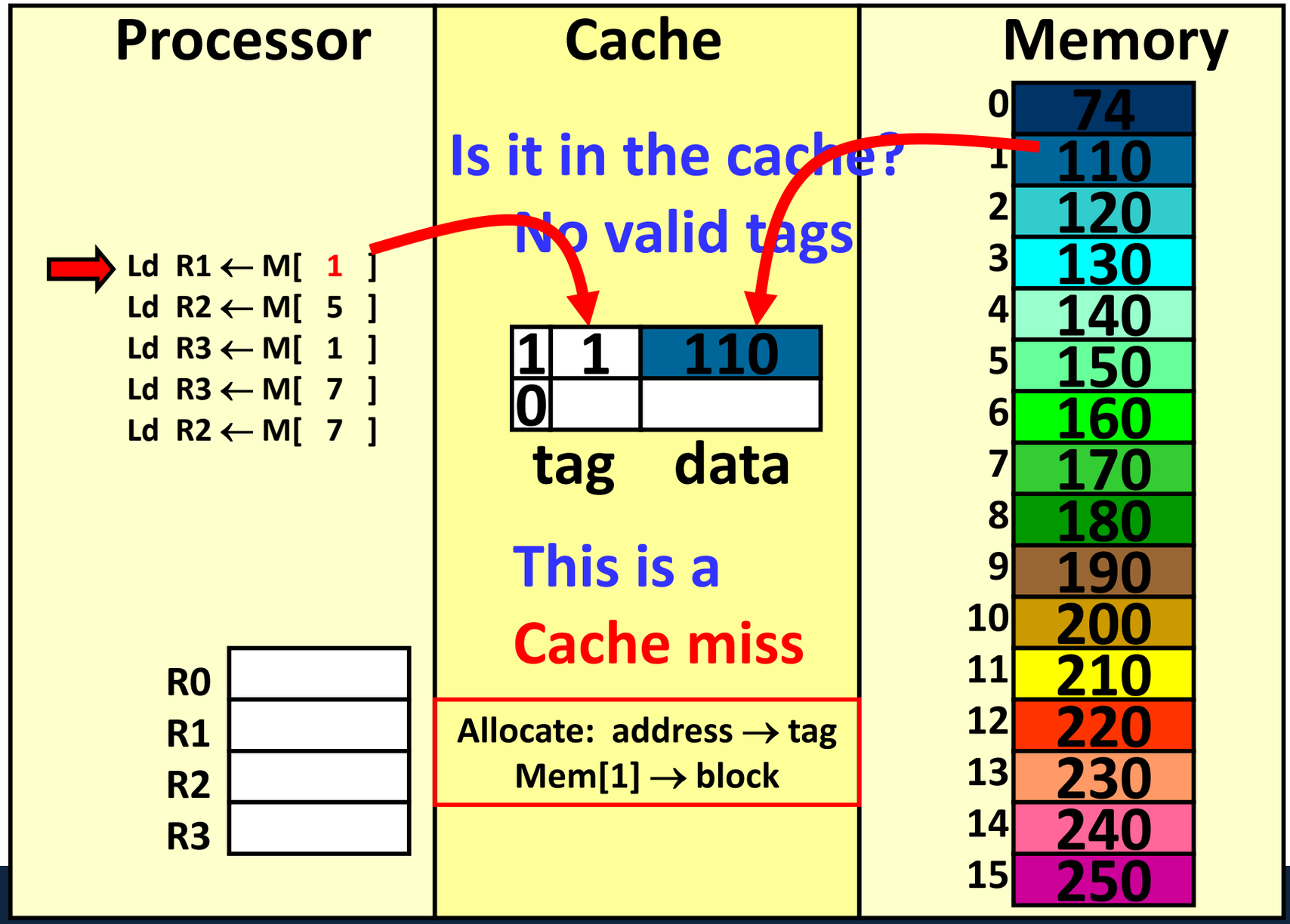
Temporal Locality

- Temporal locality: if a given memory location is referenced now, it will probably be used again in the near future
  - Why? Take a look at code you've written. You tend to use a variable multiple times
  - Corollary: if you haven't used a variable in a while, you probably won't need it very soon either
- Hardware should take advantage of this by:
  - Placing items we just accessed in the cache
  - When we need to evict something, evict whatever data was **least recently used (LRU)**

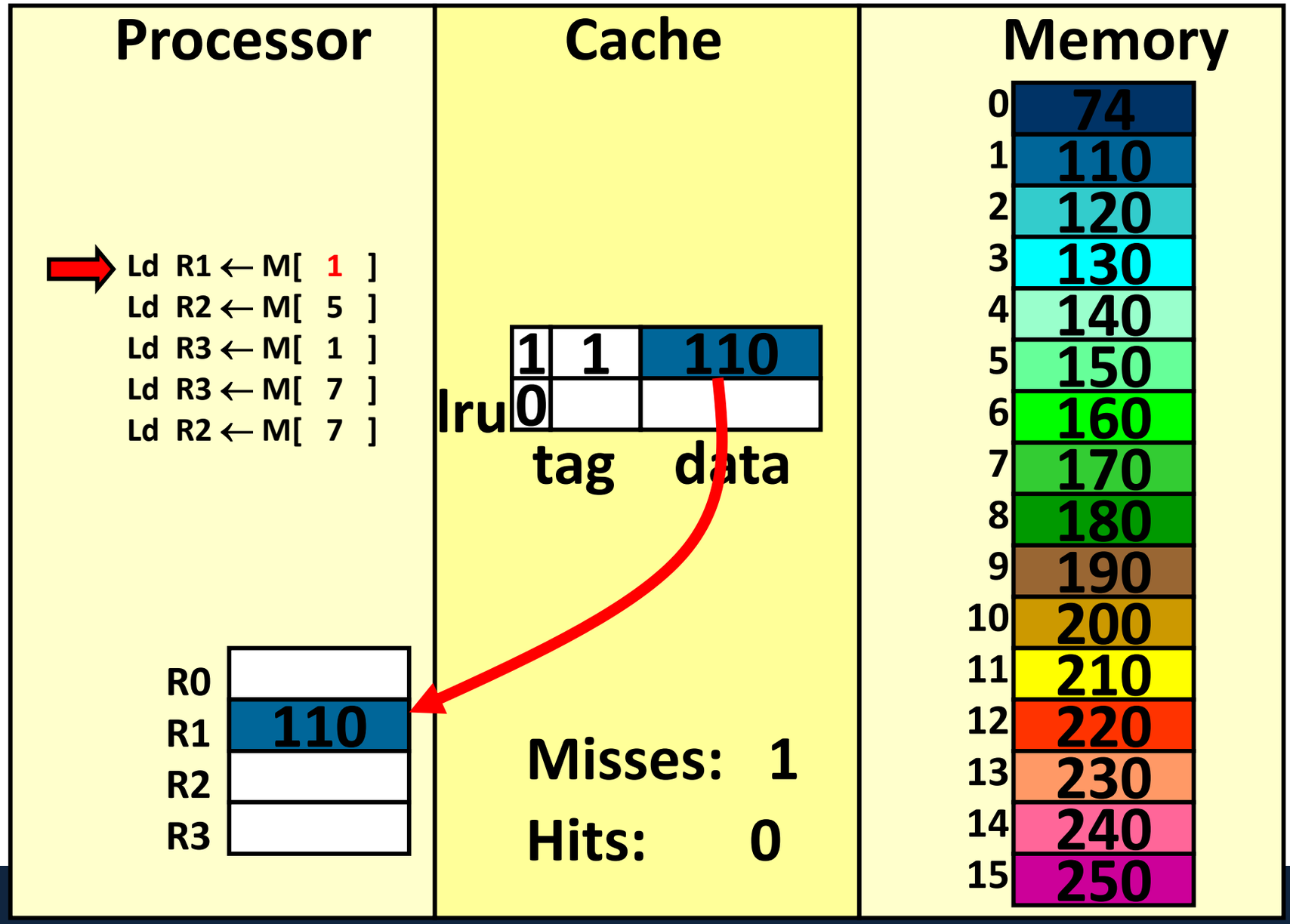
A Very Simple Memory System



A Very Simple Memory System

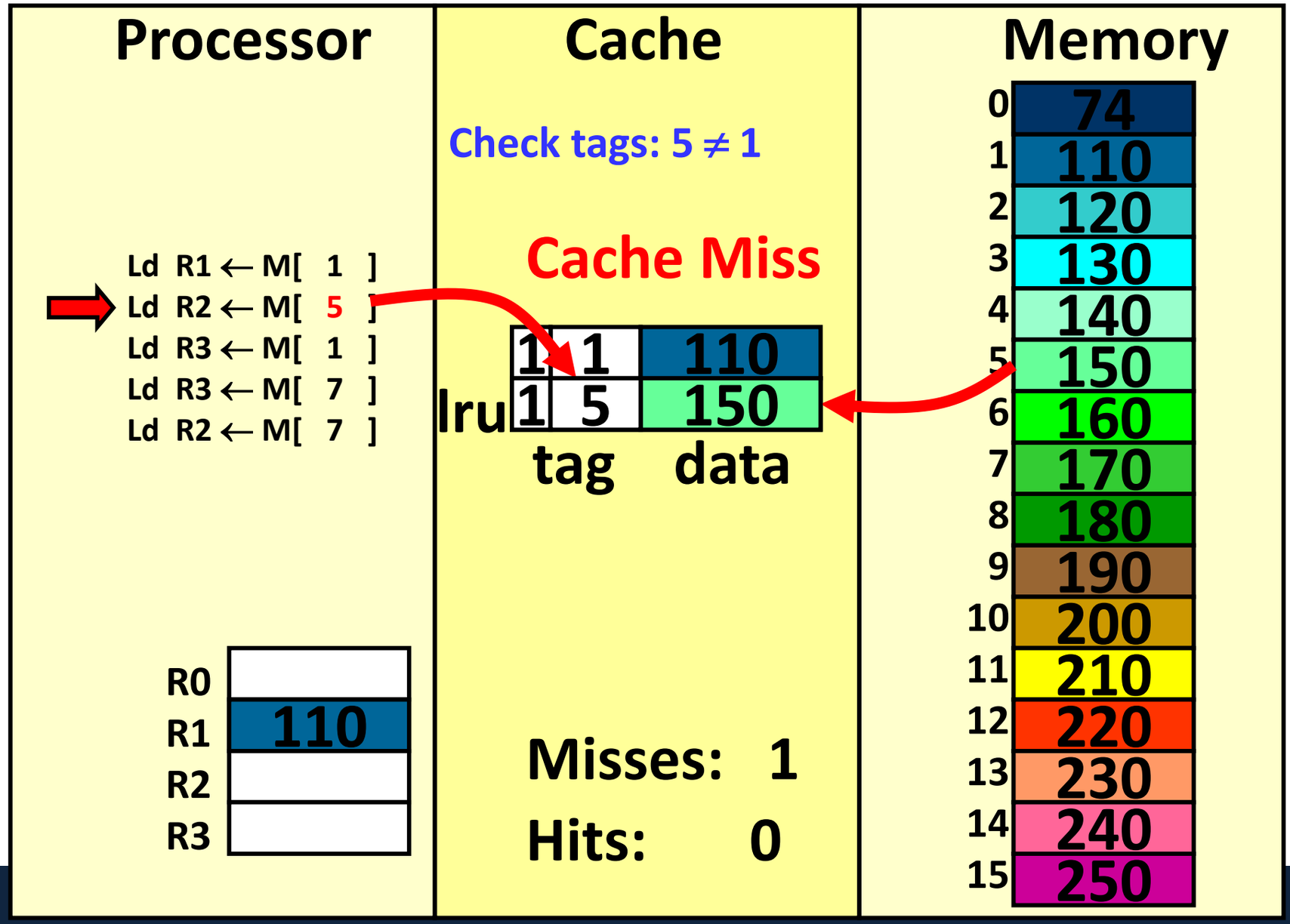


A Very Simple Memory System

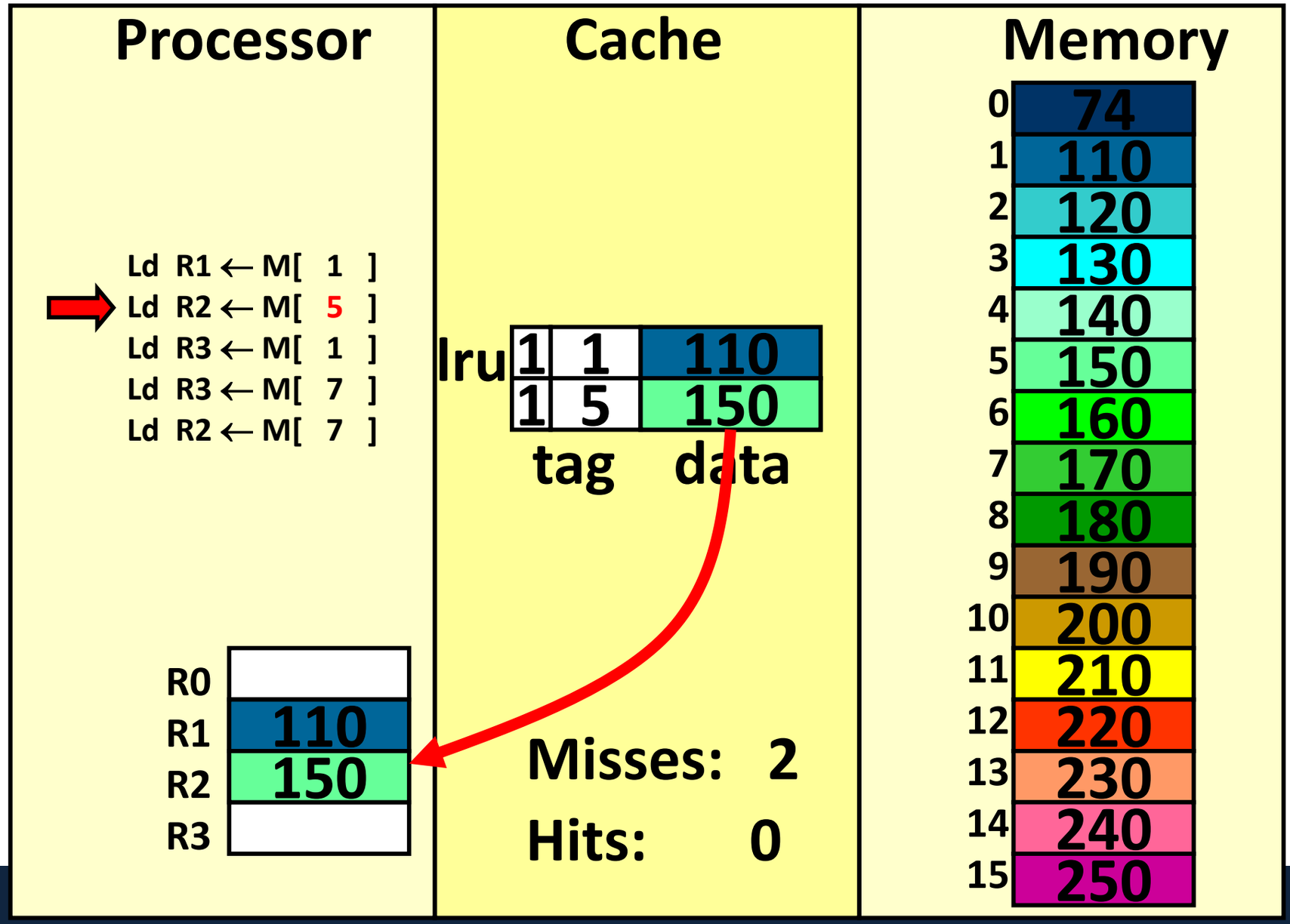


A Very Simple Memory System

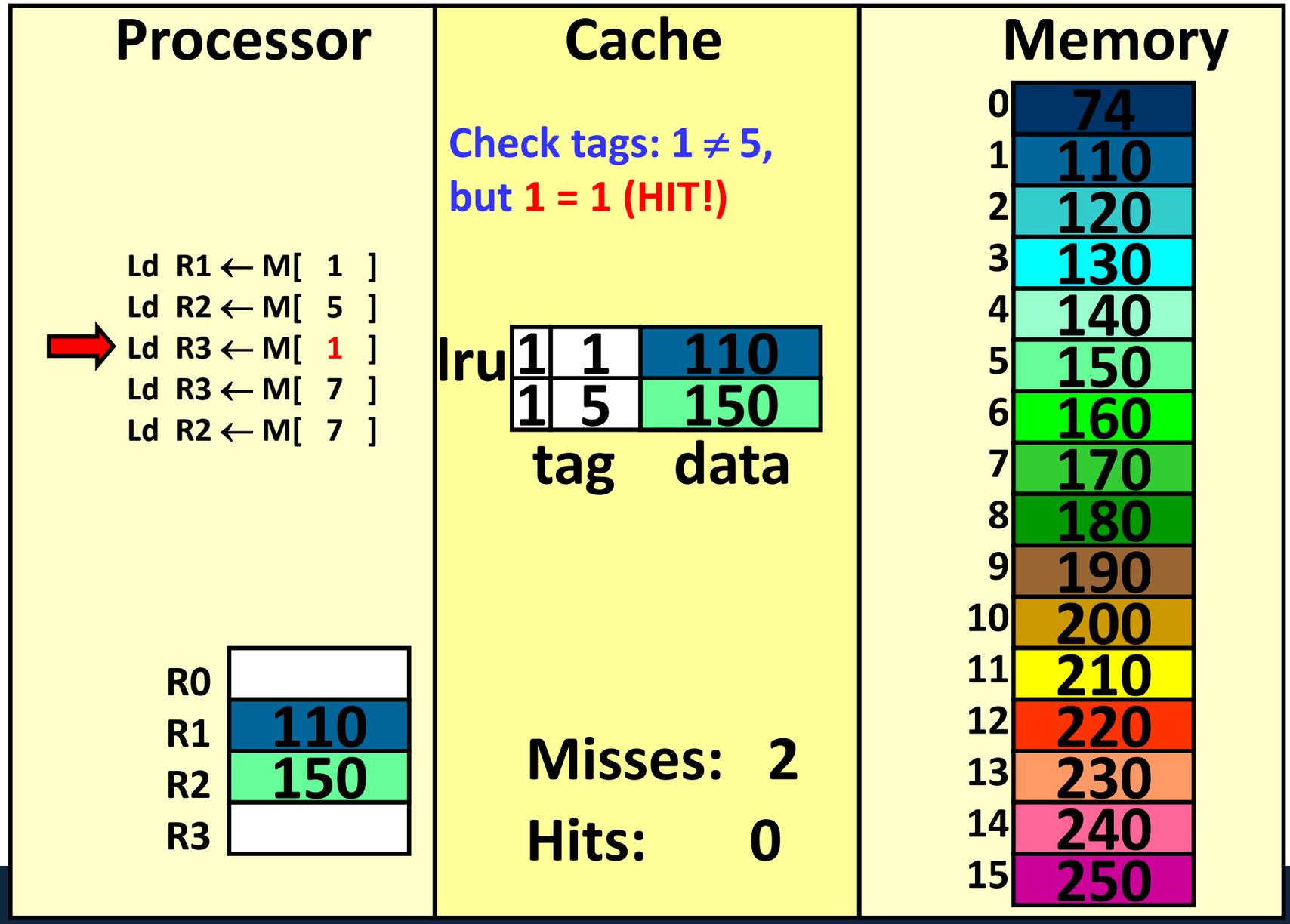
Tag comparison uses hardware called "content-addressable memory (CAM)"



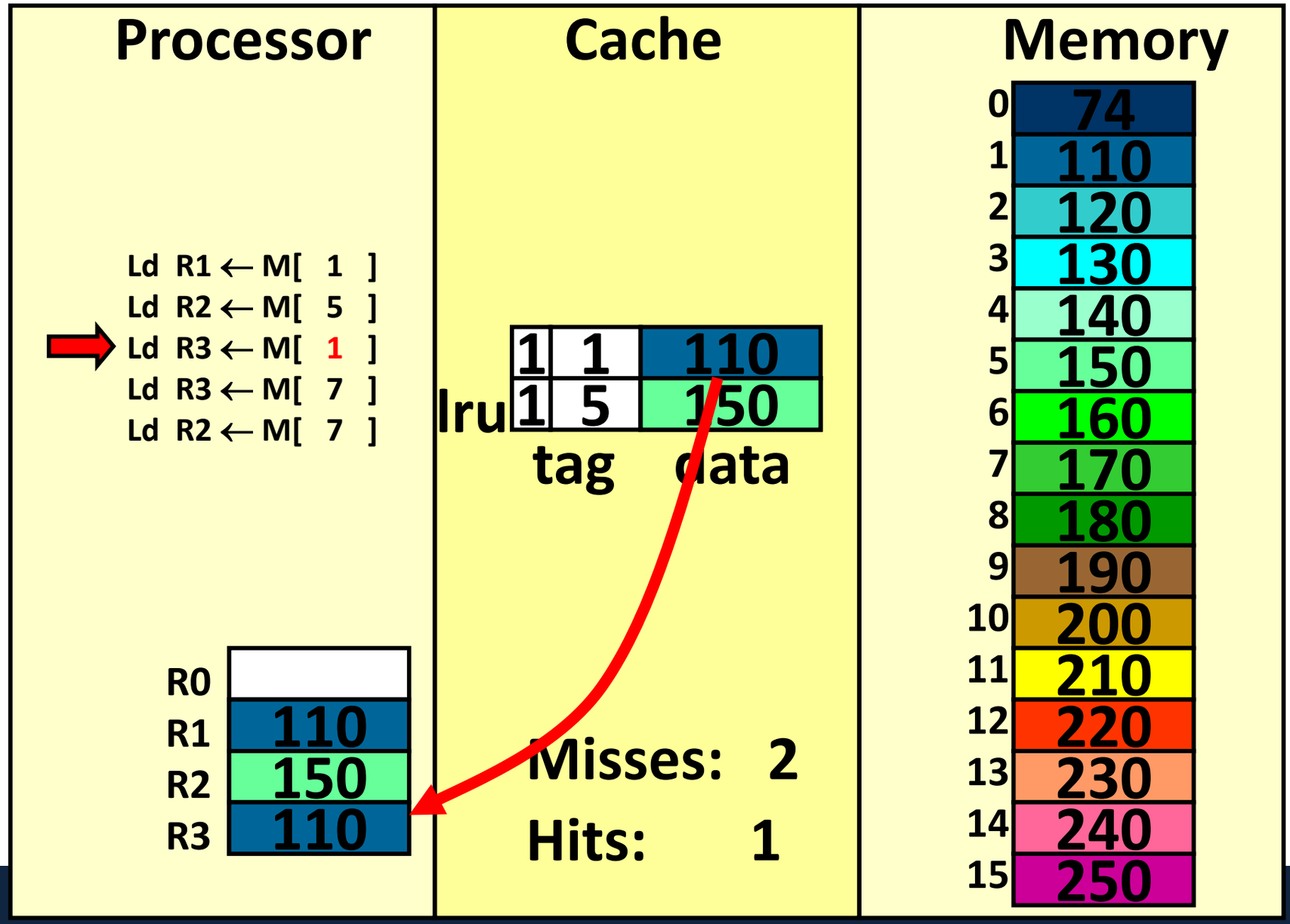
A Very Simple Memory System



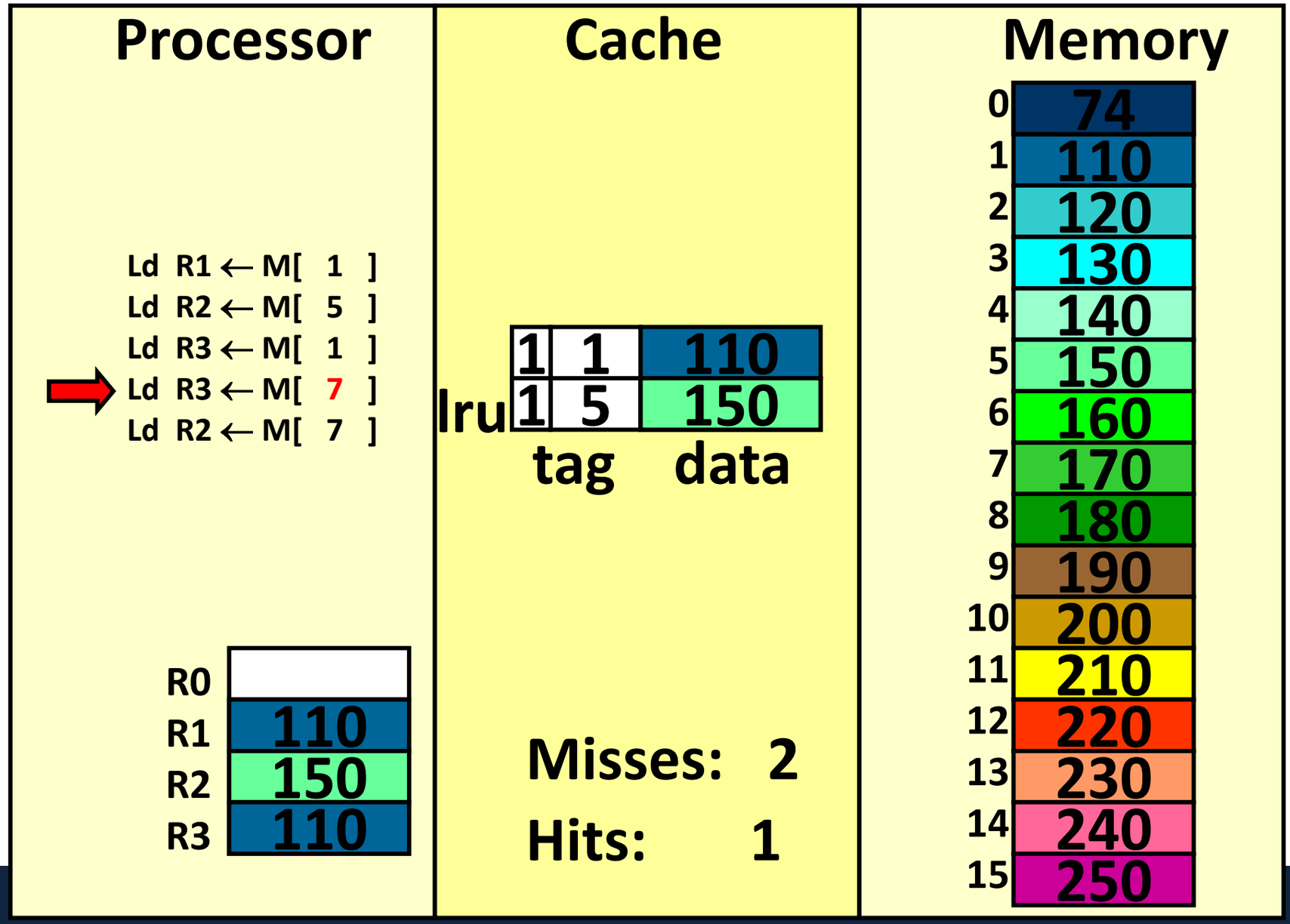
A Very Simple Memory System



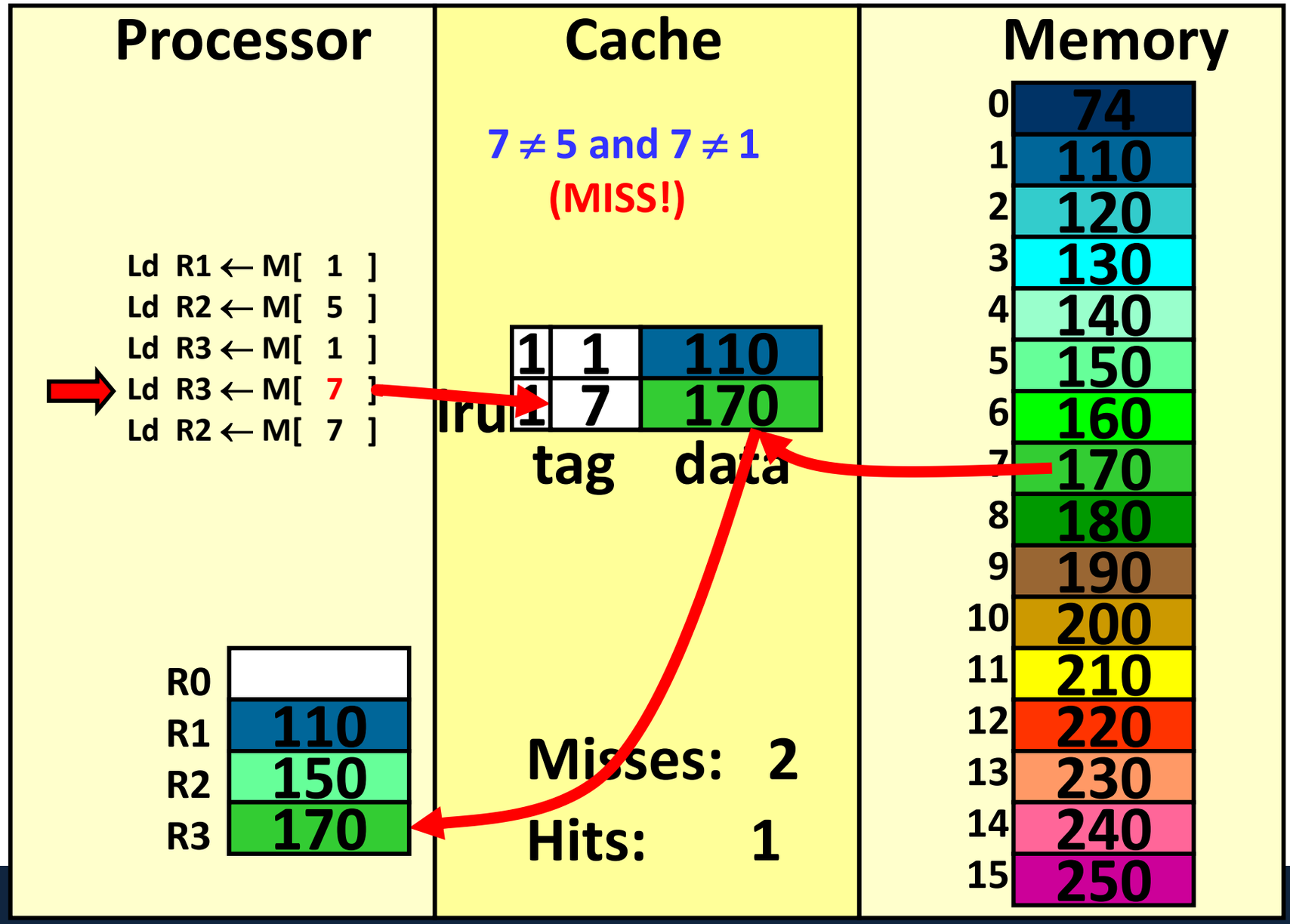
A Very Simple Memory System



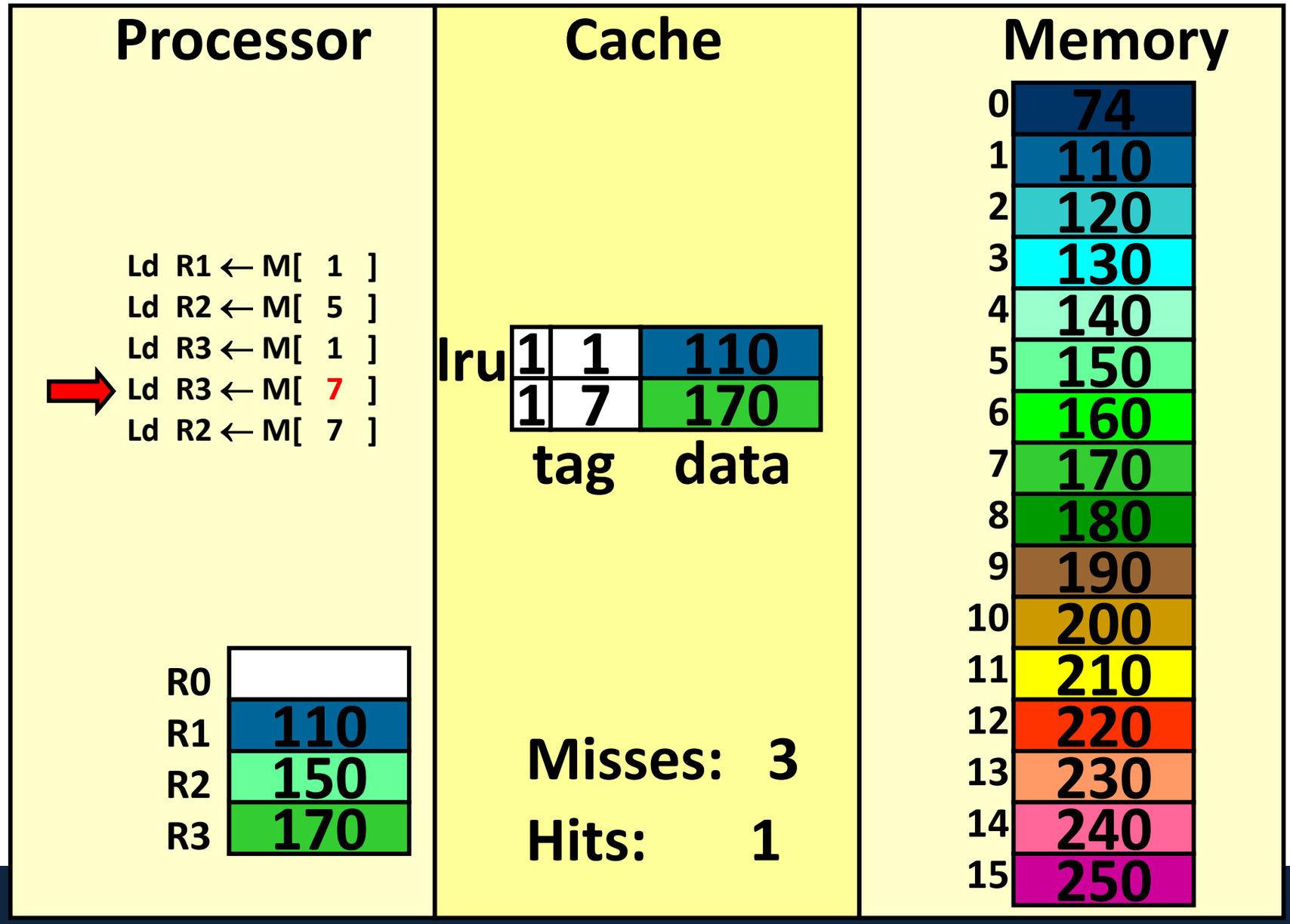
A Very Simple Memory System



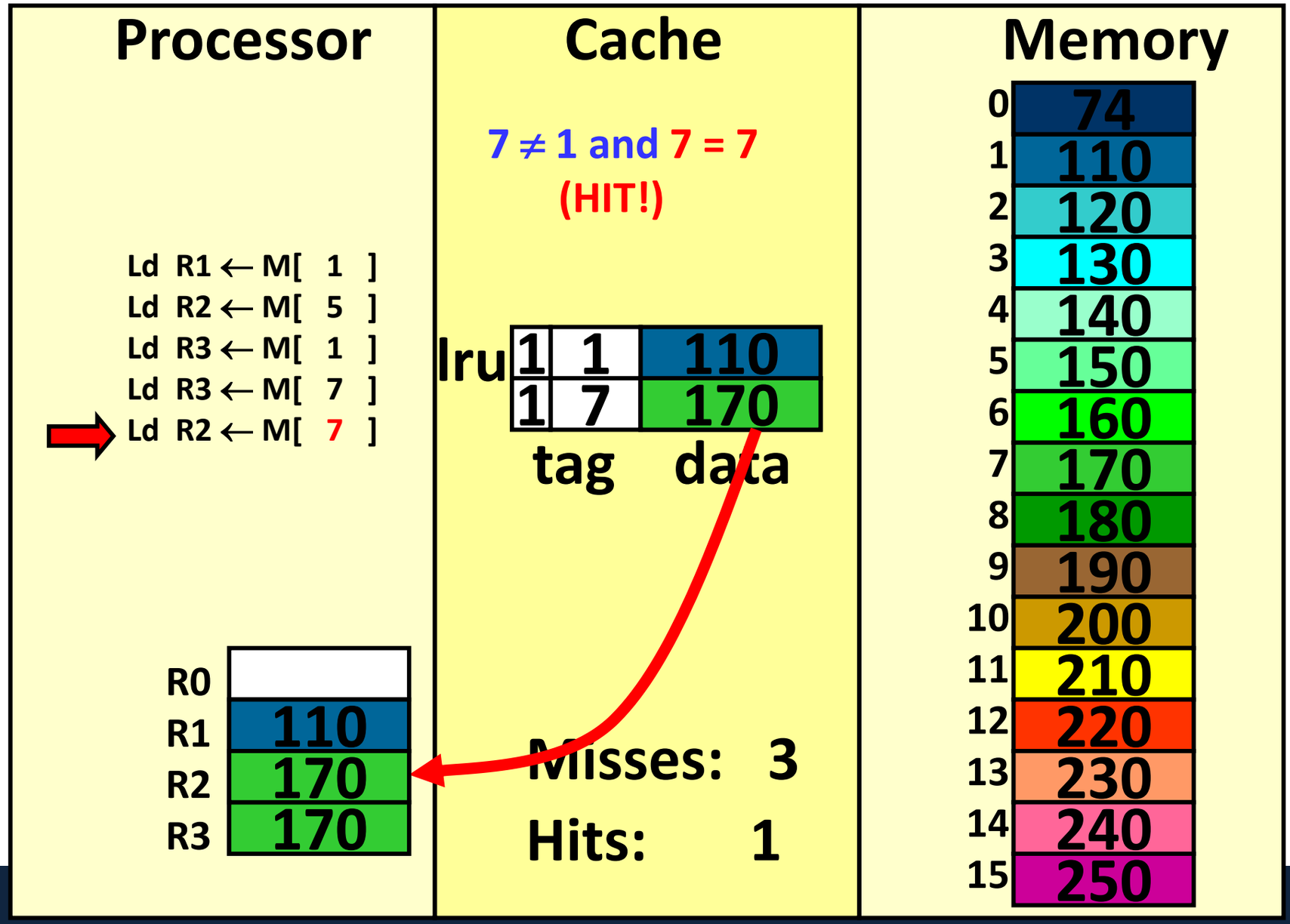
A Very Simple Memory System



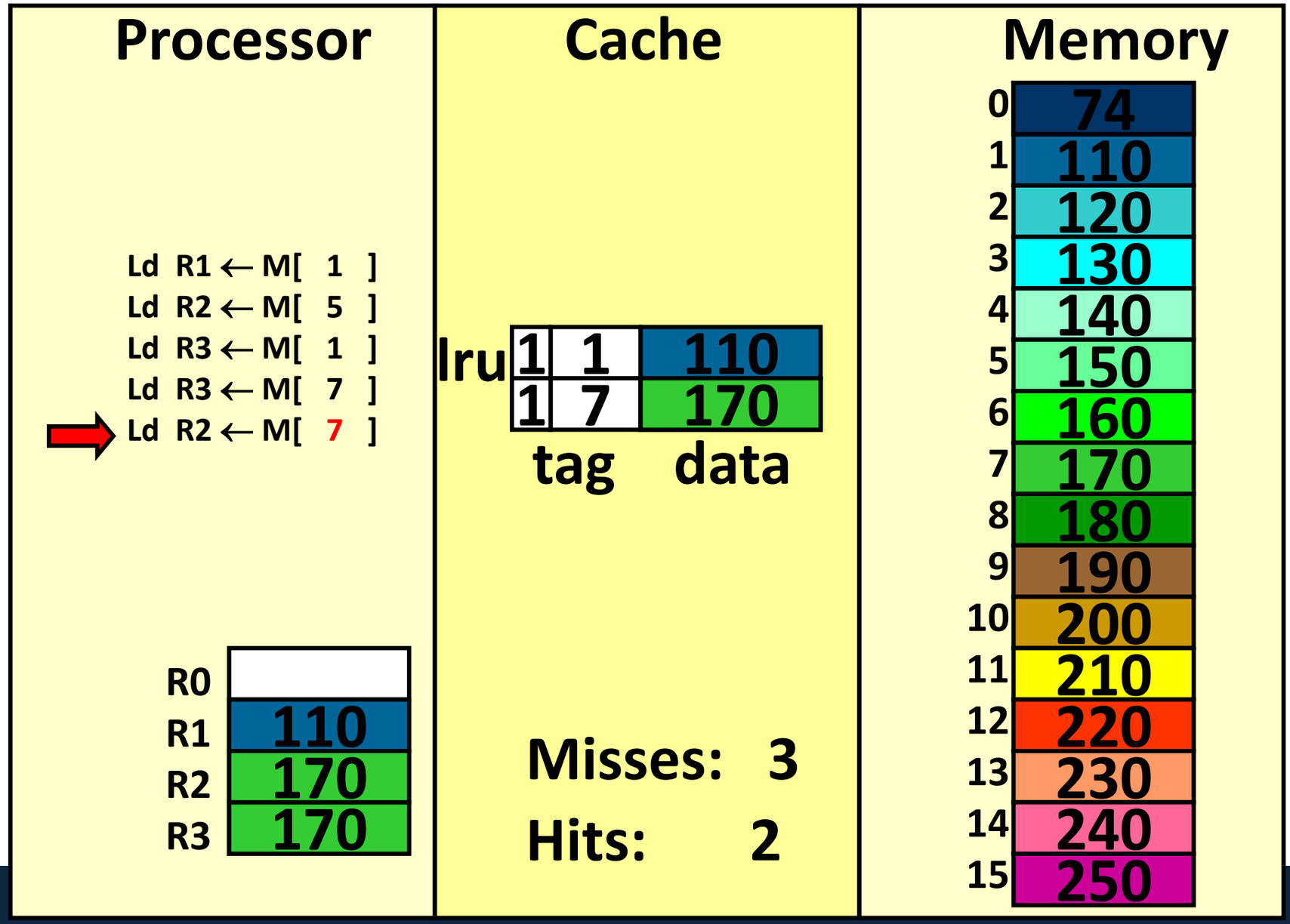
A Very Simple Memory System



A Very Simple Memory System



A Very Simple Memory System



Next time

- Making our caches bigger and better
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah

Calculating Average Access Latency

- Average Latency:
  - $cache\ latency + (memory\ latency \cdot miss\ rate)$
- Average latency for our example:
  - $1\ cycle + \left(15 \cdot \frac{3}{5}\right)$
  - $= 10\ cycles\ per\ reference$

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

Calculating Size

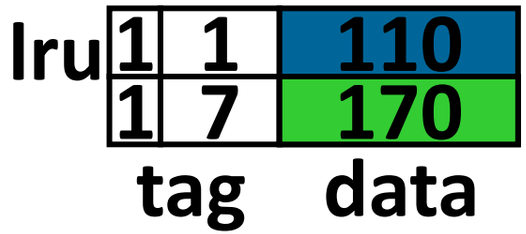
- How many bits is used in cache?
  - Storing data
    - 2 bytes of SRAM
  - Calculate overhead (non-data)
    - This cost is often forgotten for caches, but it drives up the cost of real designs!
    - 2 4-bit tags
    - 2 valid bits
- What is the storage requirement

Poll: Which of the following would reduce tag overhead (as an overall percentage)? (select all that apply)

a) Increase number of cache entries
b) Decrease number of cache entries
c) Use smaller addresses
d) Store more data in each cache entry

How can we reduce overhead?

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



Increasing Block Size

Case 1:

Block size: **1** bytes

V	tag	data (block)
1	0	74
1	6	160

How many bits needed per tag?

= $\log_2$(number of blocks in memory) = $\log_2(16)$

= 4 bits

Overhead = (4+1) / 8 = 62.5%

Case 2:

Block size: **2** bytes

V	tag	data (block)	
1	0	74	110
1	3	160	170

How many bits needed per tag?

= $\log_2$(number of blocks in memory) = $\log_2(8)$

= 3 bits

Overhead = (3+1) / 16 = 25%

	Memory	Tag (case 1)	Tag (case 2)
0	74	0	0
1	110	1	0
2	120	2	1
3	130	3	1
4	140	4	2
5	150	5	2
6	160	6	3
7	170	7	3
8	180	8	4
9	190	9	4
10	200	10	5
11	210	11	5
12	220	12	6
13	230	13	6
14	240	14	7
15	250	15	7

Poll: What will the overhead of this cache be?

Figuring out the tag

Poll: What's the pattern?

- If block size is N, what's the pattern for figuring out the tag from the address?
  - $tag = \left\lfloor \frac{addr}{block\ size} \right\rfloor$
- If block size is power of 2, then this is just everything except the $\log_2(block\ size)$ bits of the address in binary!
- E.g.

$$0d11 = 0b1011$$
$$Tag = 0b101 = 0d5$$
$$Block\ Offset = 1$$

- Remaining bits (block offset) tells us how far into the block the data is