My CPU when the L1 cache misses

EECS 370

Cache Introduction

Announcements

- Project 3 Checkpoint due next Thursday
  - Pipeline working without data hazards
- Checkout pipeline simulator on website
  - Not exactly the same as project, doesn't have WBEND

Can We Improve Branch Performance?

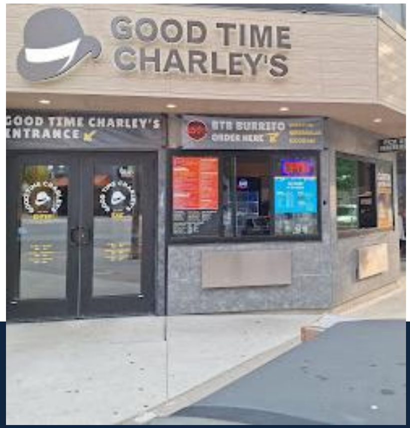
- CPI increases every time a branch is taken!
  - About 50%-66% of time
- Is that necessary?

- **No!** We can try to predict when branch is taken
  - But we would need to send target PC to memory before decoding branch
  - How do we:
    1. Know an instruction is a branch before decoding?
    2. Reliably guess whether it should be taken?
    3. Figure out the target PC before executing the branch?

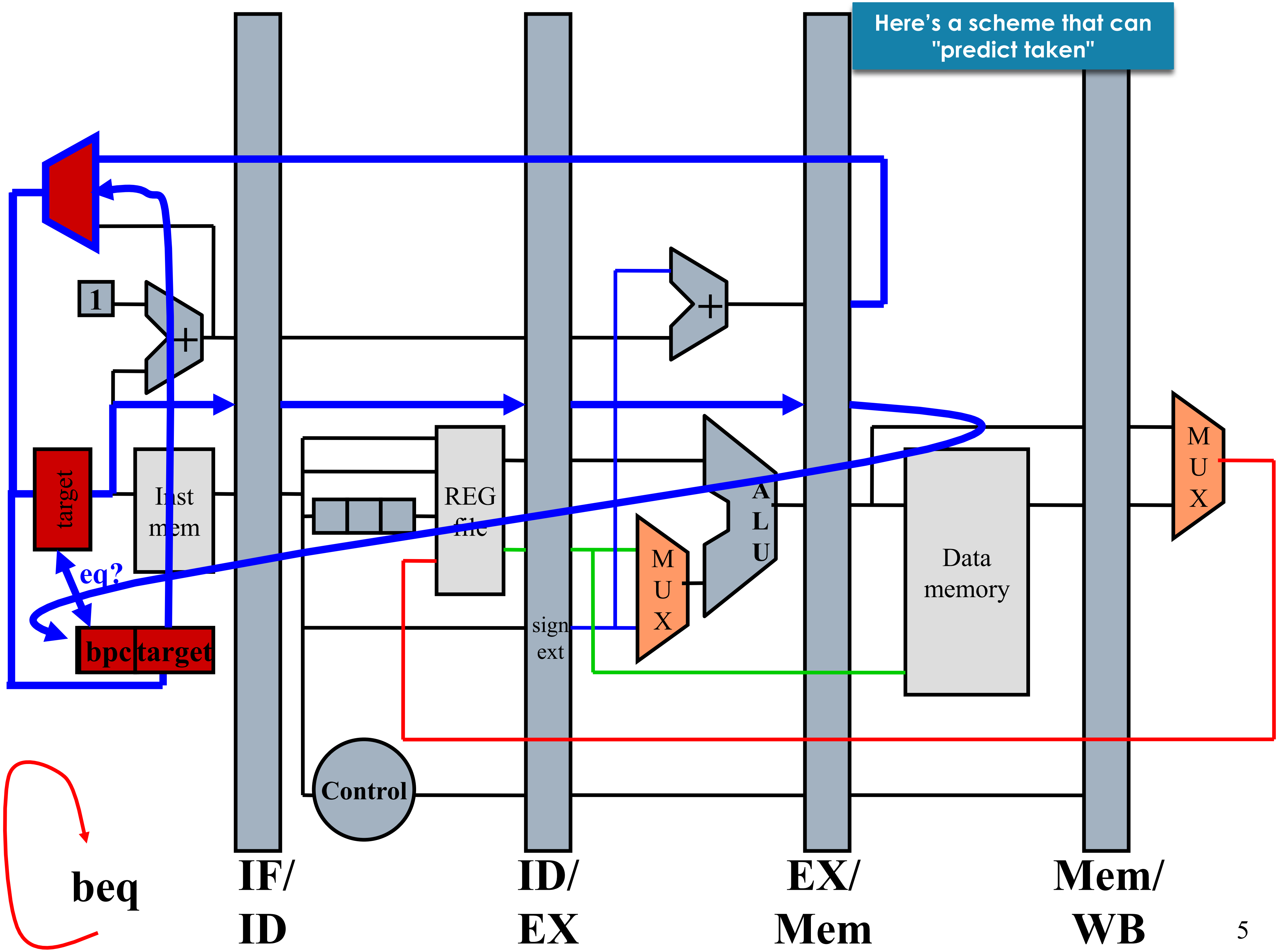
Sometimes predict taken?

- When fetching an instruction, need to predict 3 things:
  1. Whether the fetched instruction is a branch
  2. Branch direction (if conditional)
  3. Branch target address (if direction is taken)

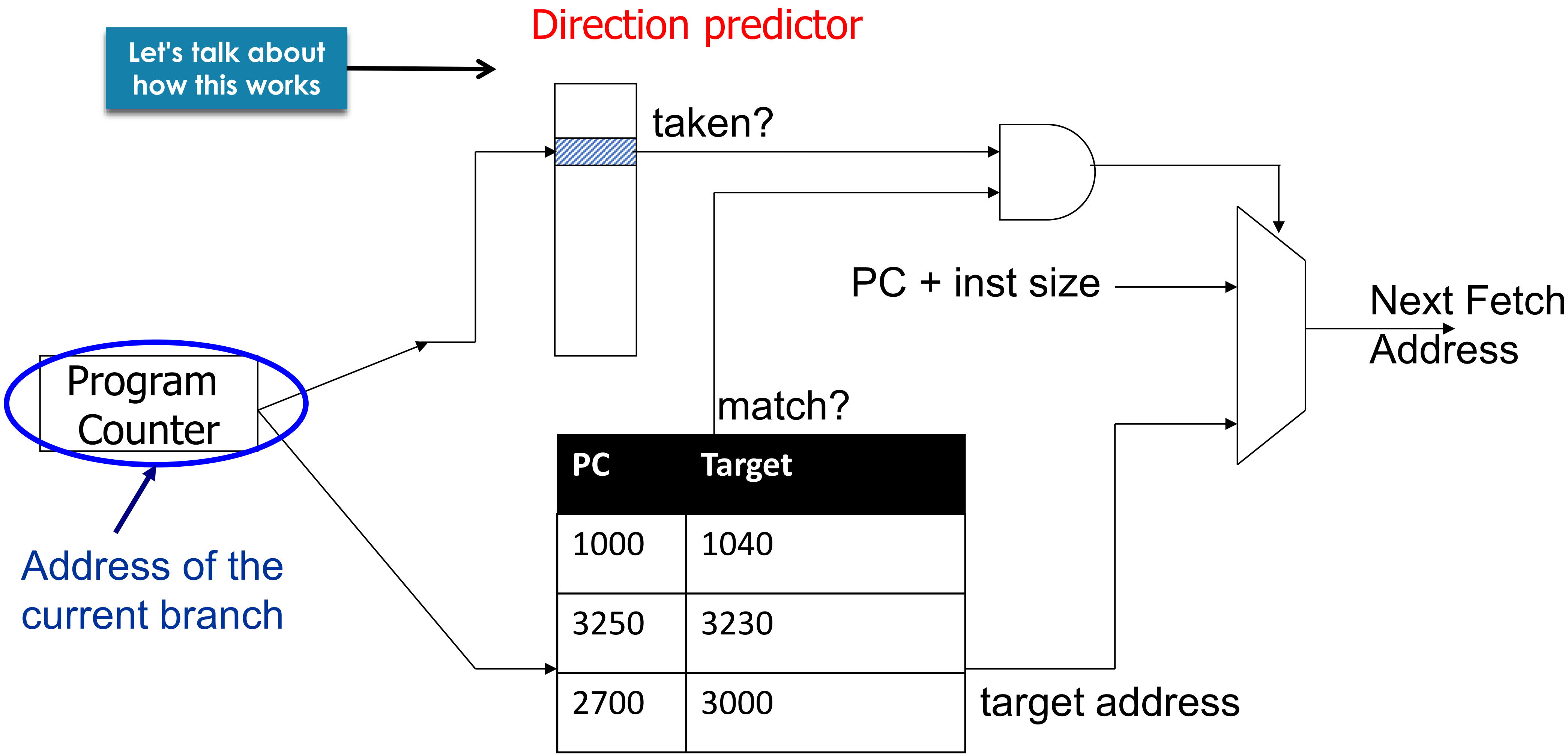
- Observation: Target address remains the same for conditional branch across multiple executions
  - Idea: store the target address of branch once we execute it, along with PC of instruction
  - Called Branch Target Buffer (BTB)



Here's a scheme that can "predict taken"



Sometimes predict taken?



PC	Target
1000	1040
3250	3230
2700	3000

"Cache" of Target Addresses (BTB: Branch Target Buffer)

Branch Direction Prediction

- "Branch direction" refers to whether the branch was taken or not
- Two methods for predicting direction:
  - Static - We predict once during compilation, and that prediction never changes
  - Dynamic - We predict (potentially) many times during execution, and the prediction may change over time

- *Static vs dynamic strategies are a very common topic in computer architecture*

Branch Direction Prediction (Static)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the "not-taken" path
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC
- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Branch Direction Prediction (Dynamic)

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed

    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

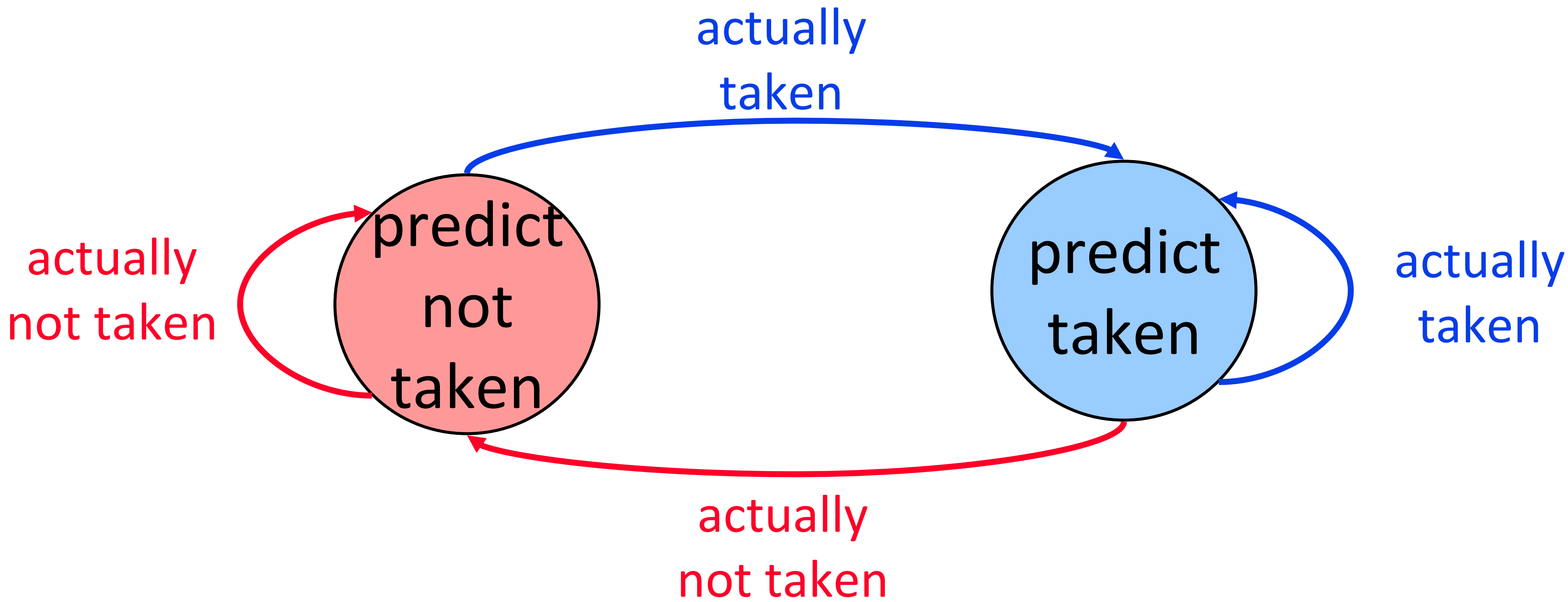
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

State Machine for Last-Time Prediction



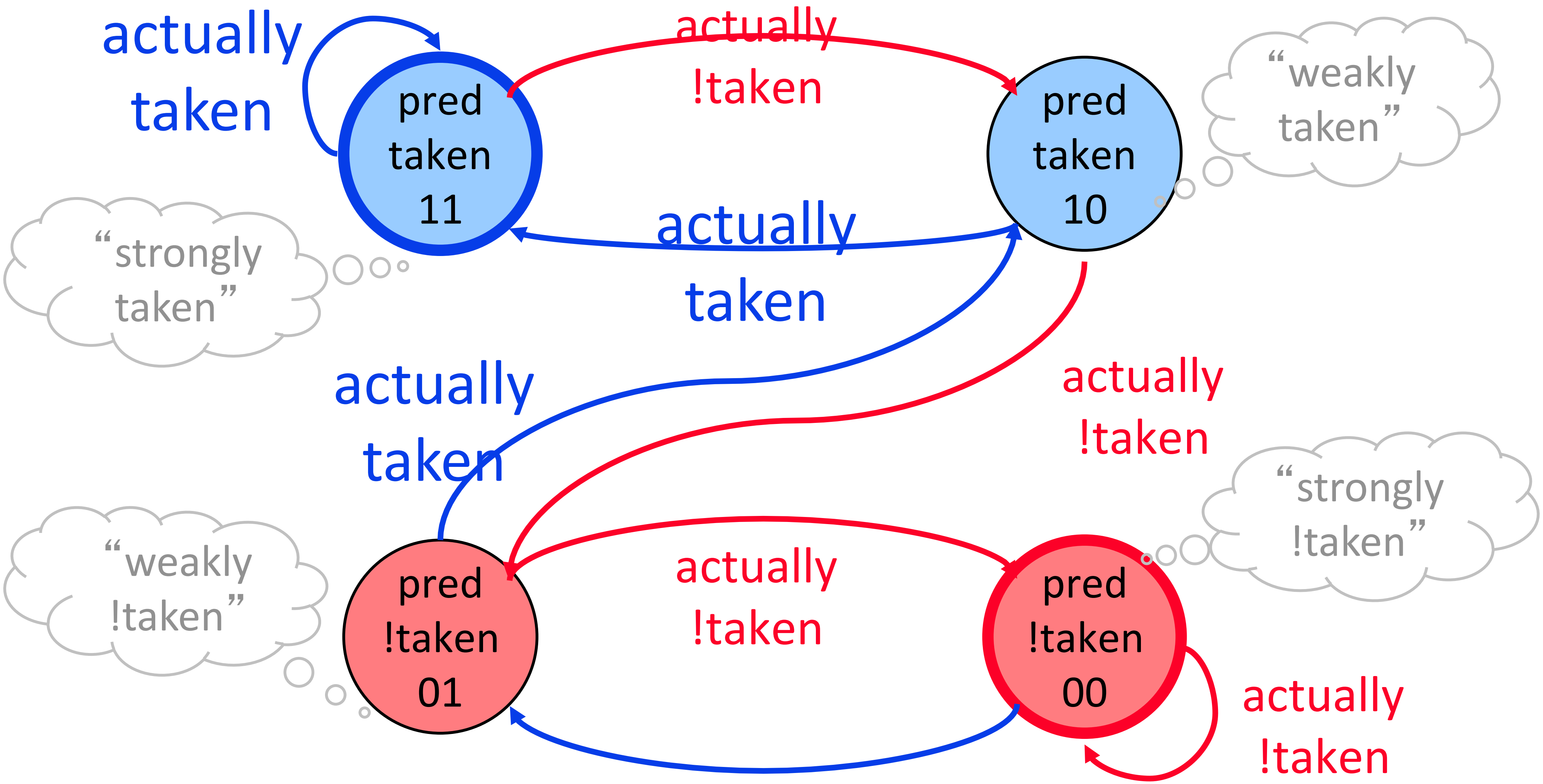
Agenda

- Improving Performance with Branch Predicting
- Simple Direction Predictor
- **Improving Direction Predictor**

Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - Even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

State Machine for 2-bit Saturating Counter



Two-Bit Counter Based Prediction

Poll: How many branches do we get wrong?

- What's the prediction accuracy of a branch with the following sequence of taken/not taken outcomes:
  - T T T T N T T N N N T N T N N

Br	T	T	T	T	N	T	T	N	N	N	T	N	T	N	N
State	10	11	11	11	[illegible]	10	11	[illegible]	[illegible]	01	[illegible]	01	[illegible]	01	00
Pred	T	T	T	T	T	T	T	T	T	N	N	N	N	N	N

Can We Do Better?

- Absolutely… take 470
  - Tons of sophisticated branch predictor designs
- I've worked on a few that found their way into some Chromebooks!

Branch Prediction

- Predict not taken: ~50% accurate
- Predict backward taken: ~65% accurate
- Predict same as last time: ~80% accurate

- Realistic designs: ~96% accurate

Remember this Example from Lecture 1?

- We know understand why sorting improves the inner-loop so much
  - The branch predictor is better at guessing what's gonna happen when data is sorted!

```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
std::sort(data, data + arraySize);

// Test
clock_t start = clock();
long long sum = 0;
// Primary loop
for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}

double elapsedTime =
  static_cast<double>(clock() - start);
```

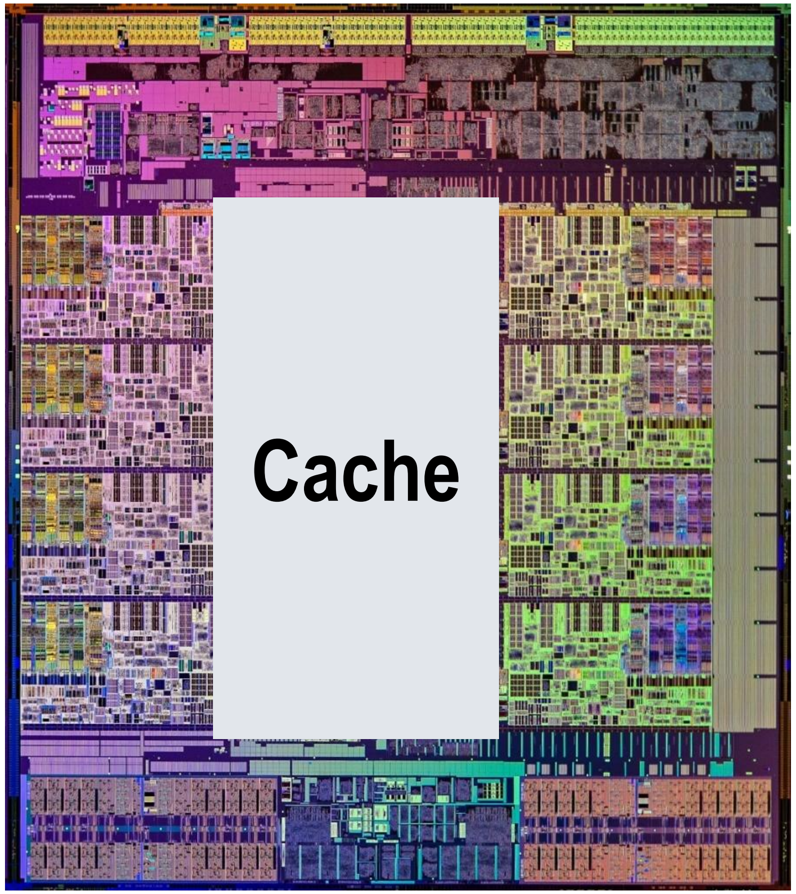
Agenda

- **Memory Types**
- Memory Hierarchy and Cache Principles
- Cache example
- How to improve cache

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** important



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);
}
```

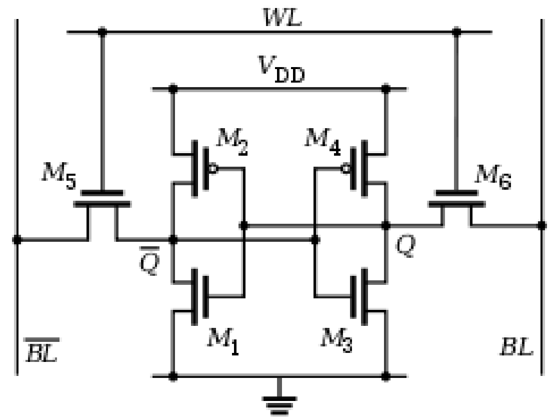
```c
#include<stdio.h>
#include<stdlib.h>

#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);
}
```

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

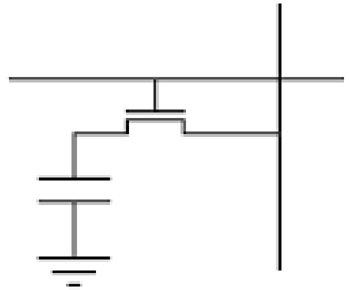
Option 1: SRAM

- Each bit is made of 6 transistors
- Volatile: need constant power to keep data
- Fast: ~1 ns read time
- Still rather large
    - Can only put ~MBs on chip
    - Impractical to scale up to GBs needed for memory



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
  - We can put up to ~16-64 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 (~100,000 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

Agenda

- Memory Types
- **Memory Hierarchy and Cache Principles**
- Cache example
- How to improve cache

Memory Goals

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

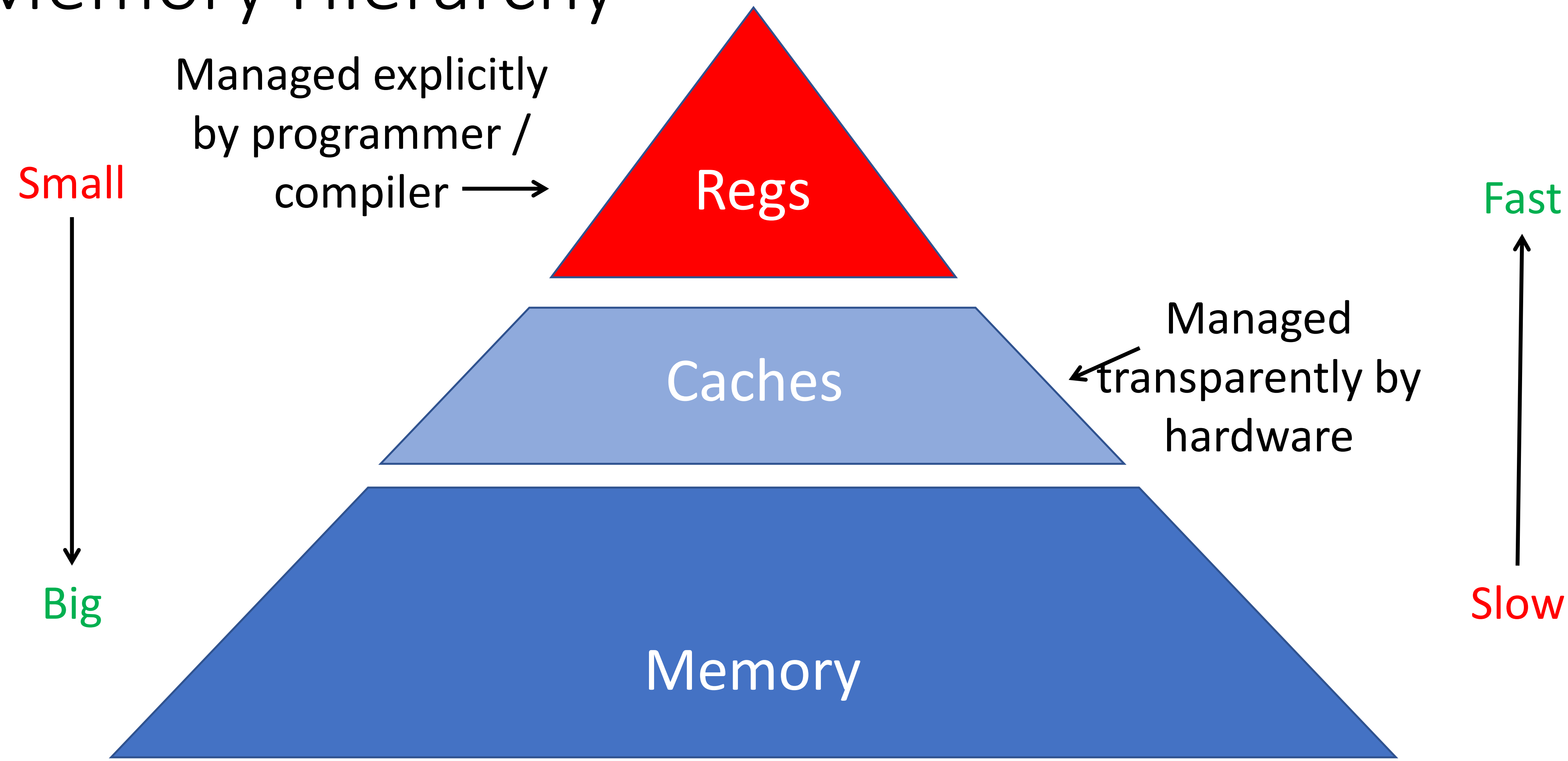
- DRAM, hard-drives and SSDs are too slow
- SRAM is too expensive

- How to get best properties of multiple memory technologies?

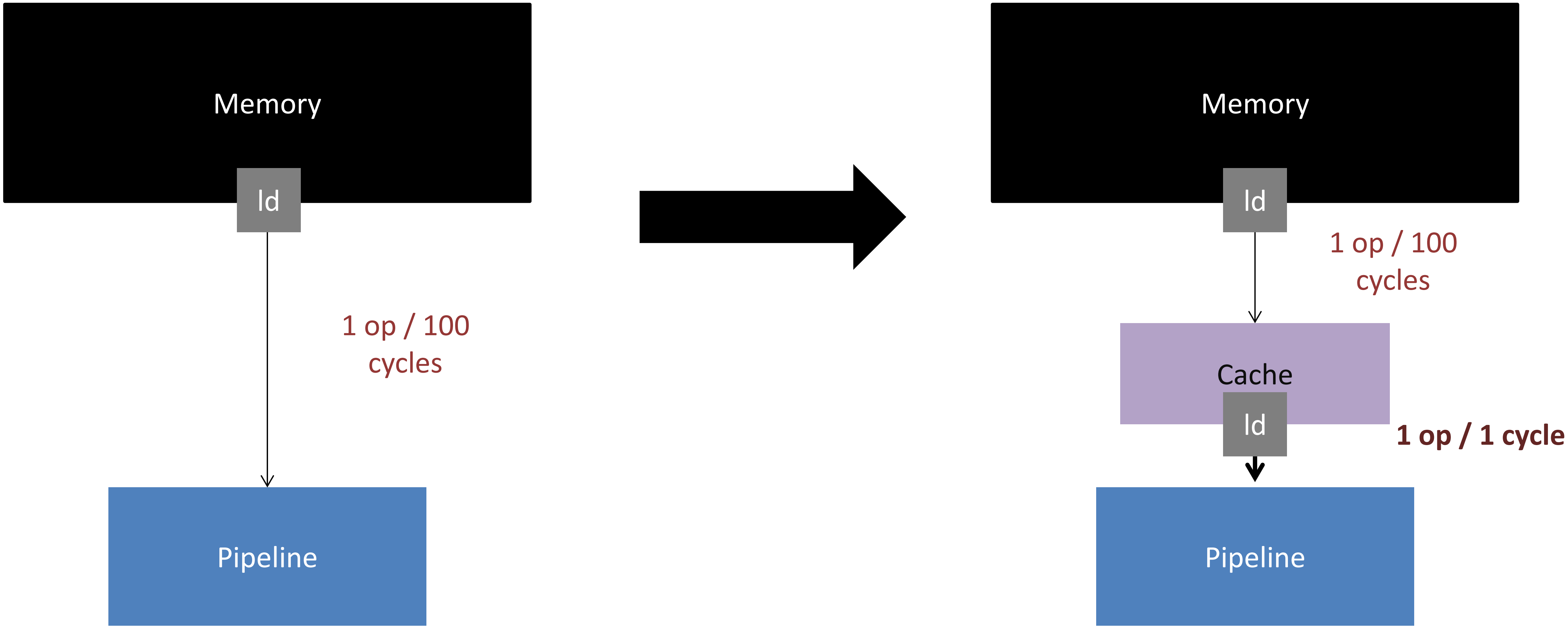
Memory Hierarchy

- 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

Memory Hierarchy



Caches - Overview

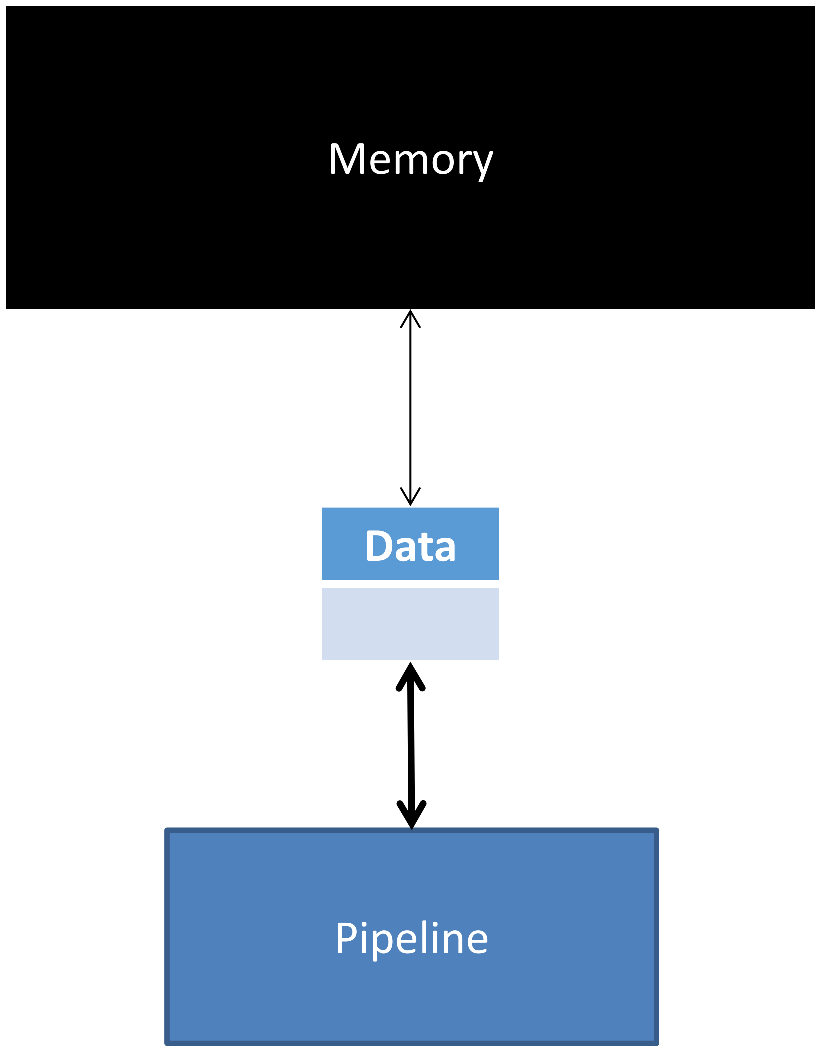


Function of the Cache

- 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?

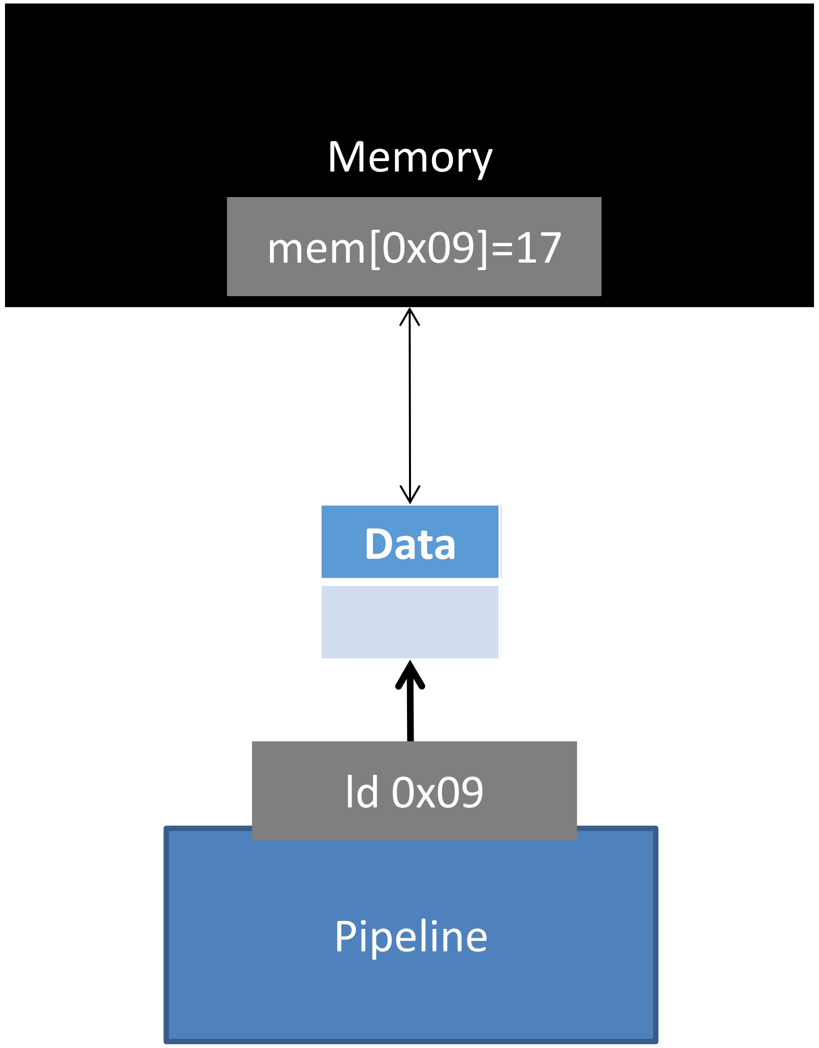
The simplest cache

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



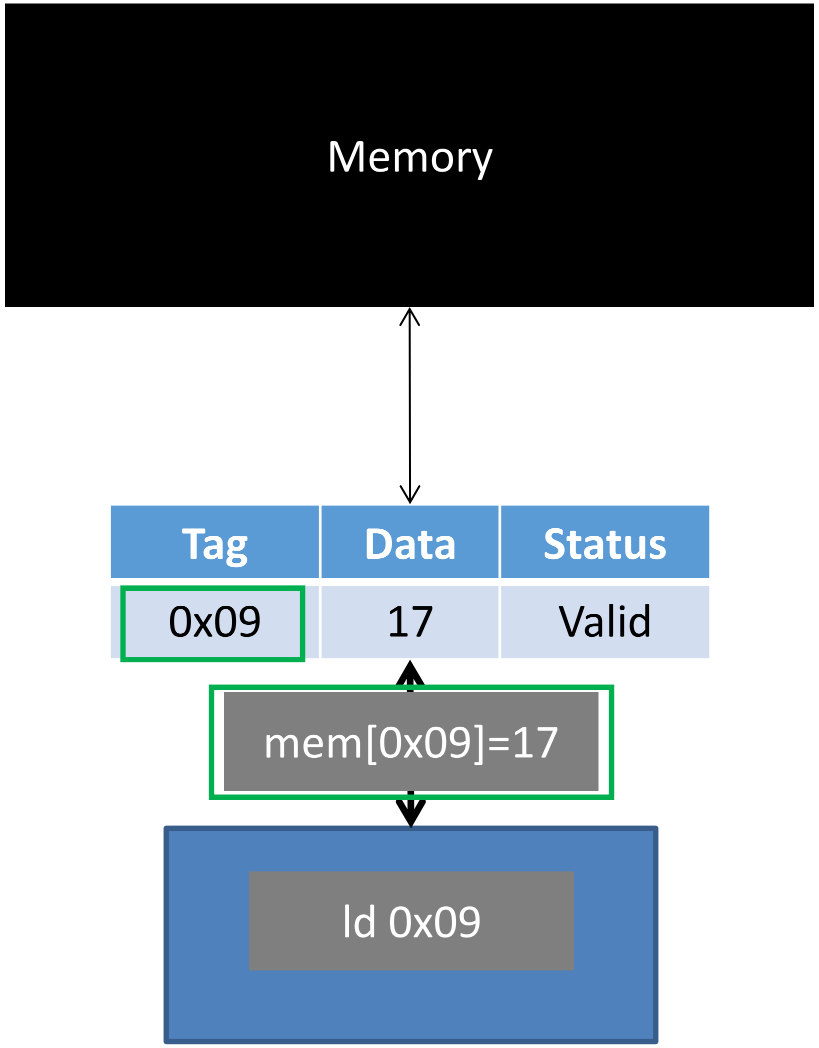
The simplest cache

- 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



Agenda

- Memory Types
- Memory Hierarchy and Cache Principles
- **Cache example**
- How to improve cache

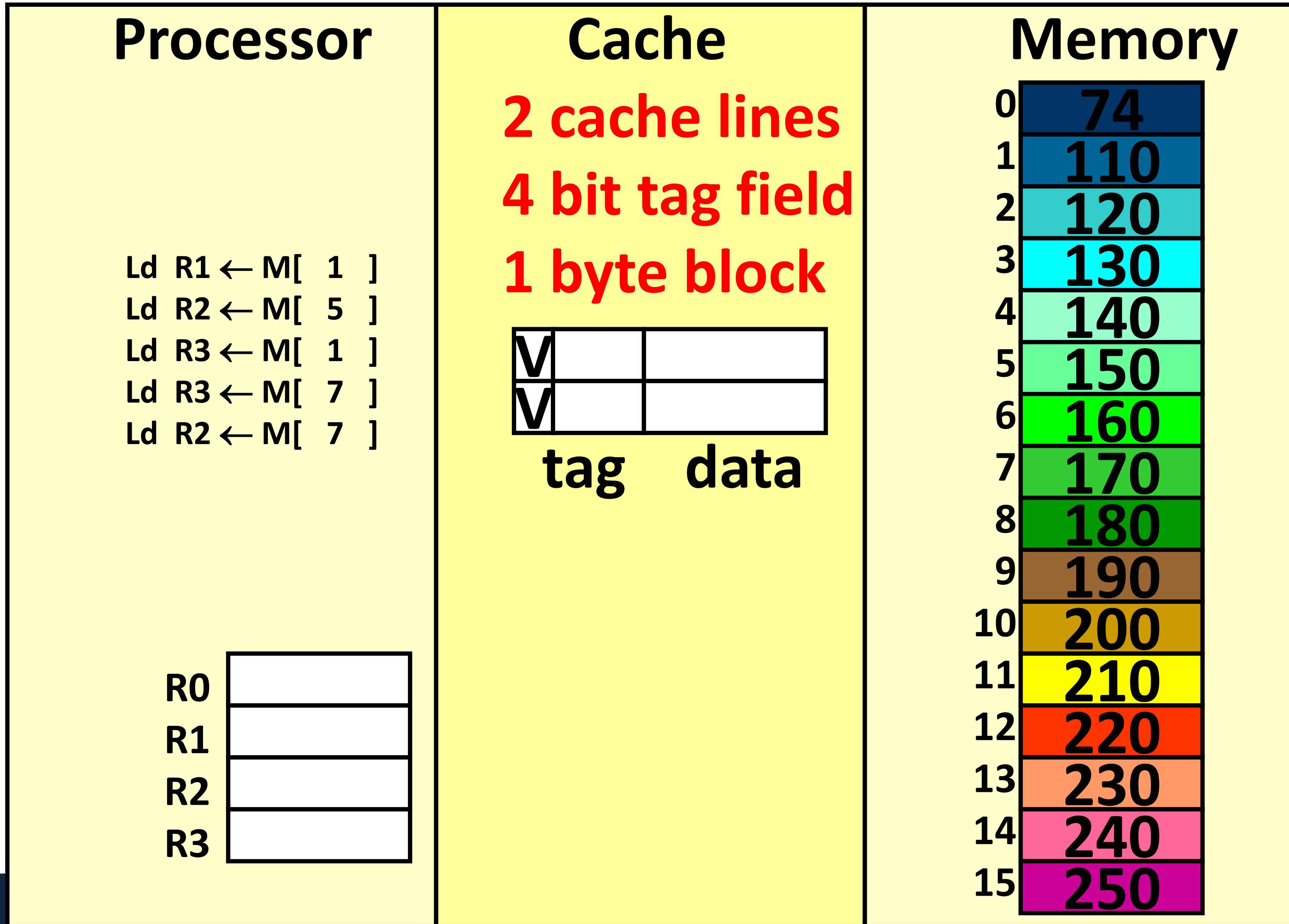
Scaling Up

- 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

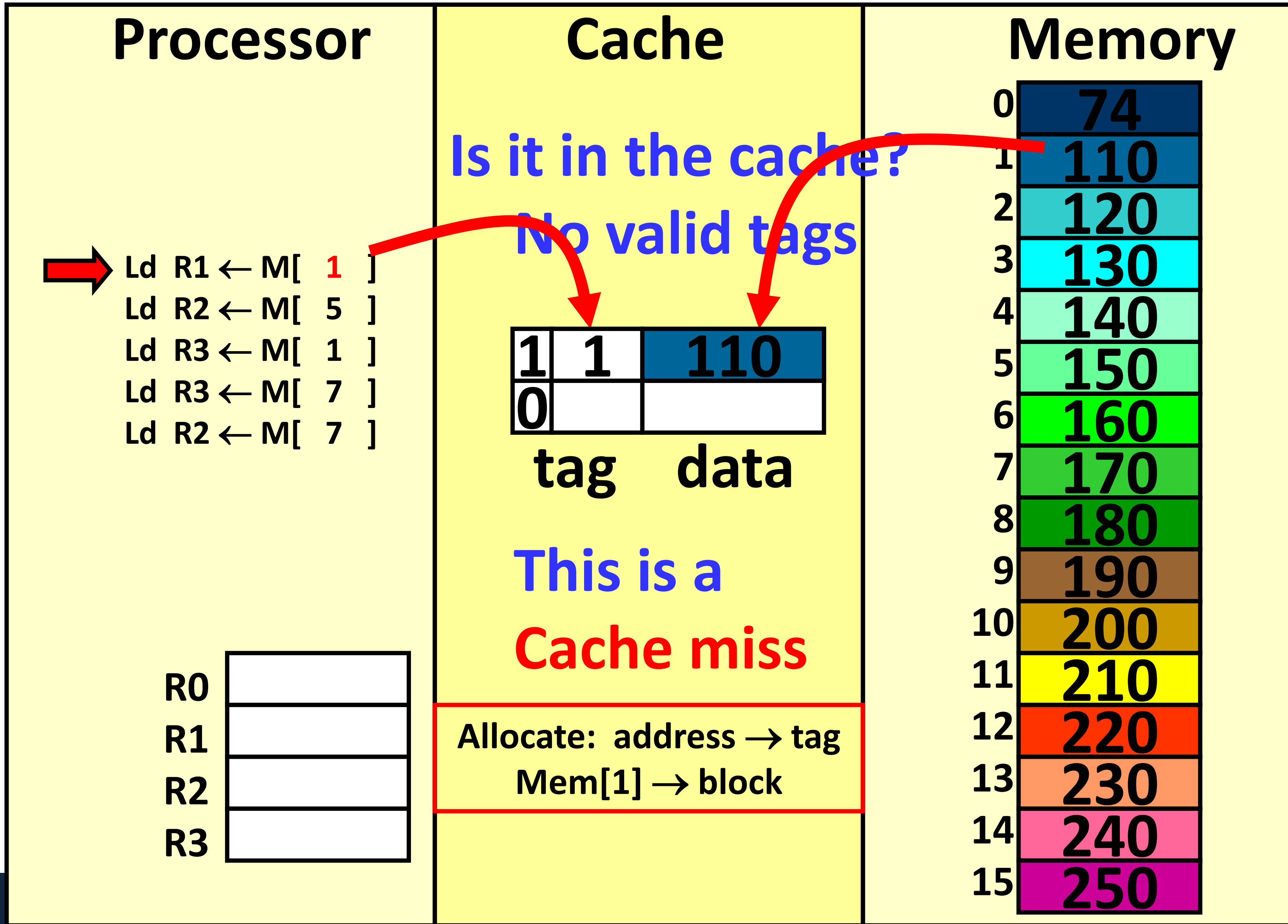
Temporal Locality

- 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)**

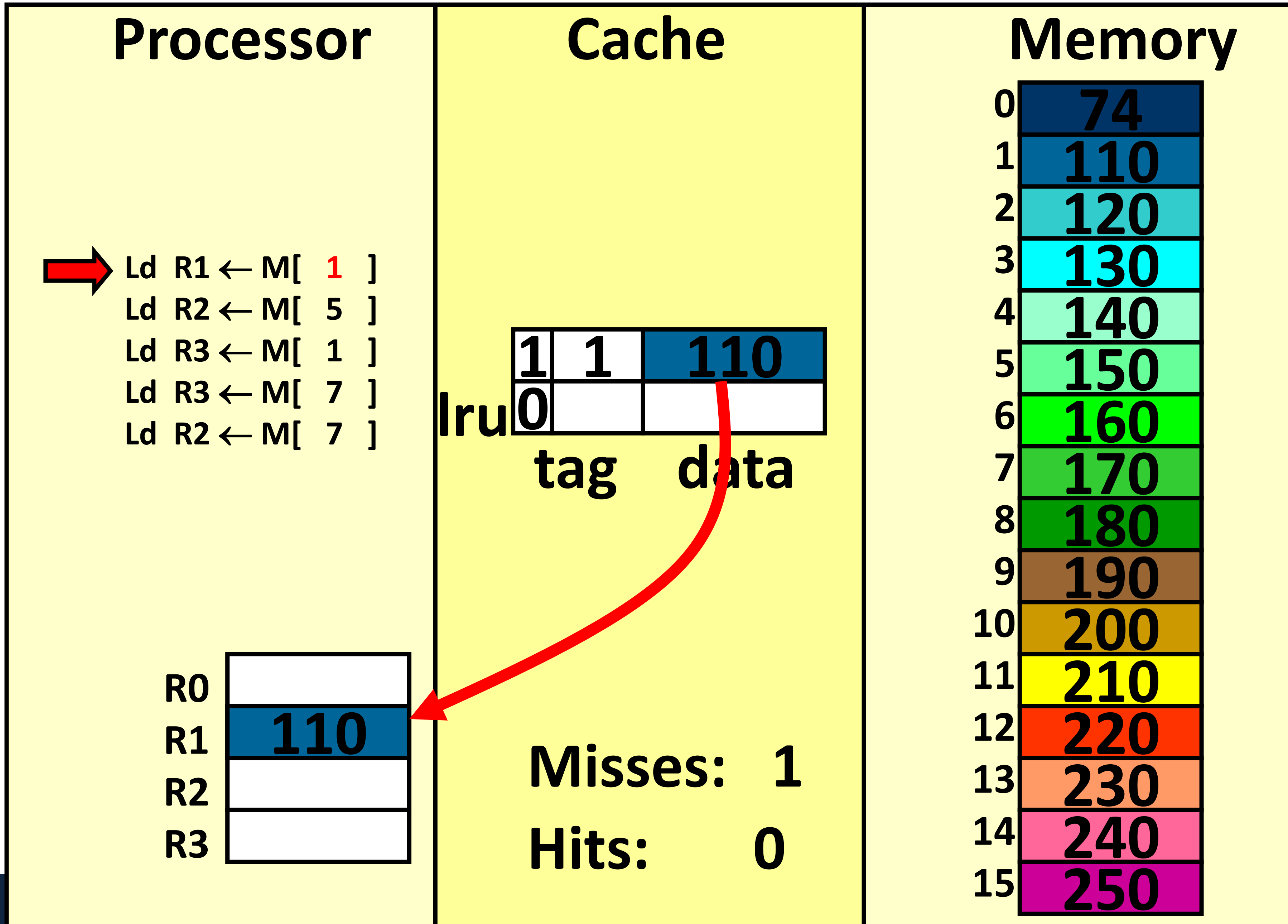
A Very Simple Memory System



A Very Simple Memory System

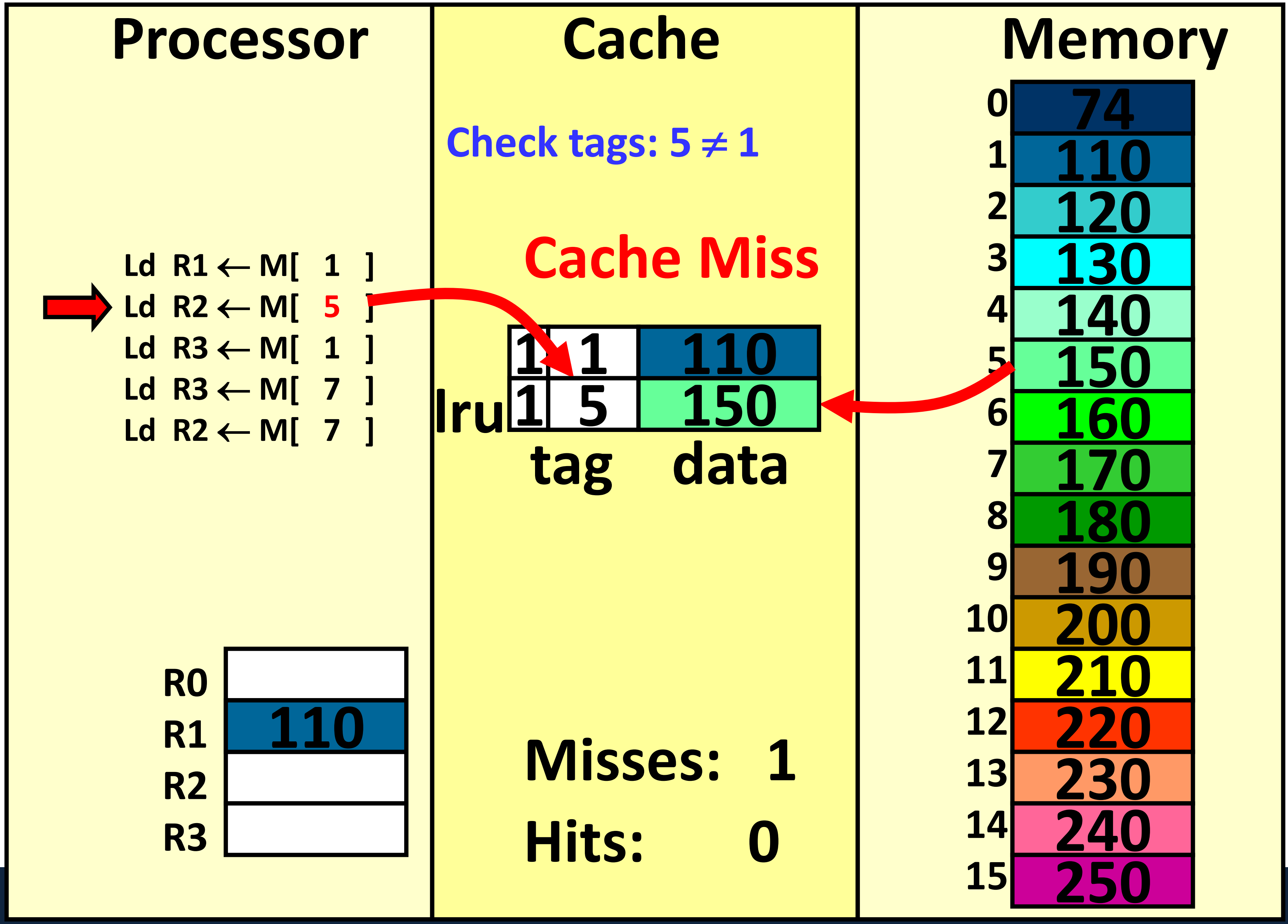


A Very Simple Memory System

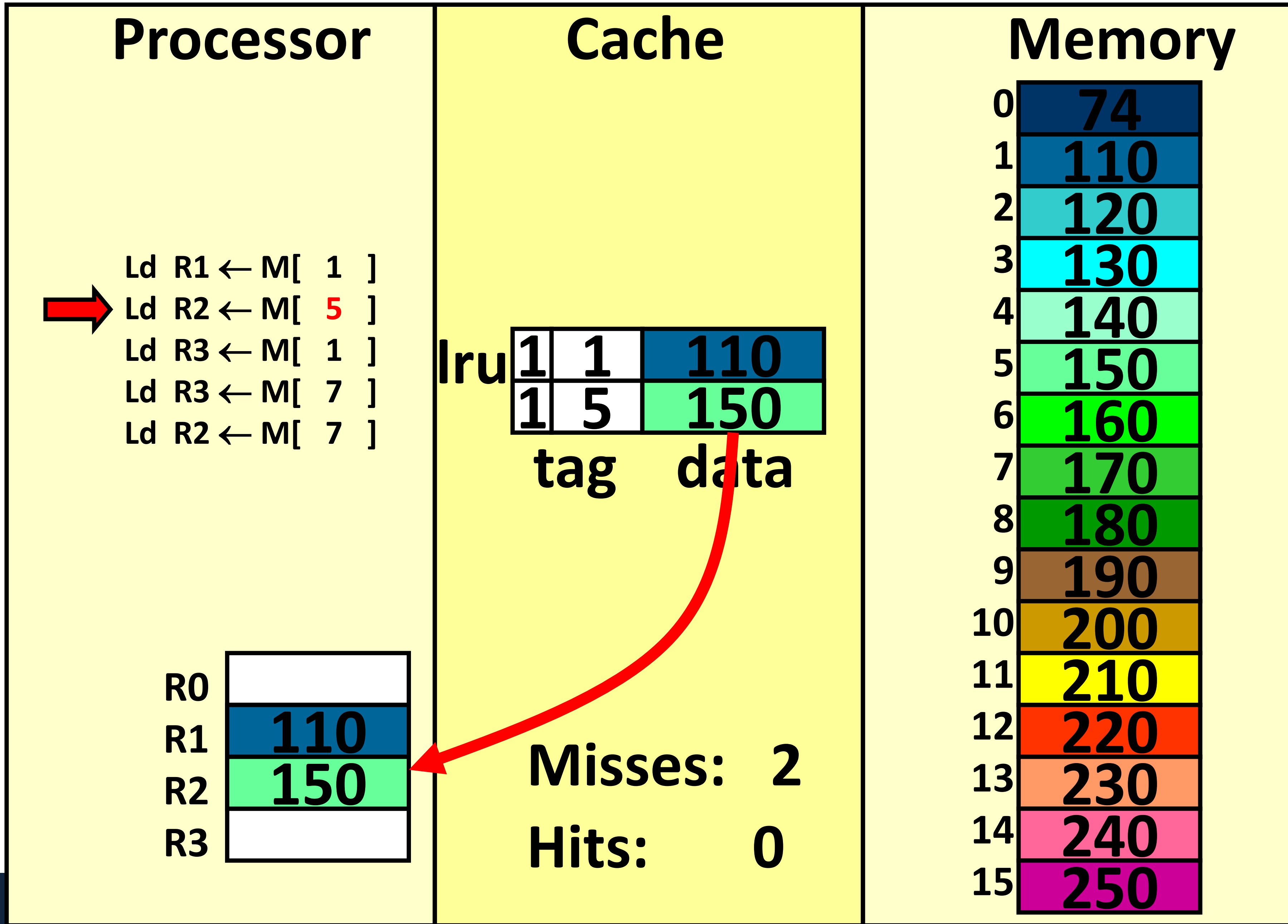


A Very Simple Memory System

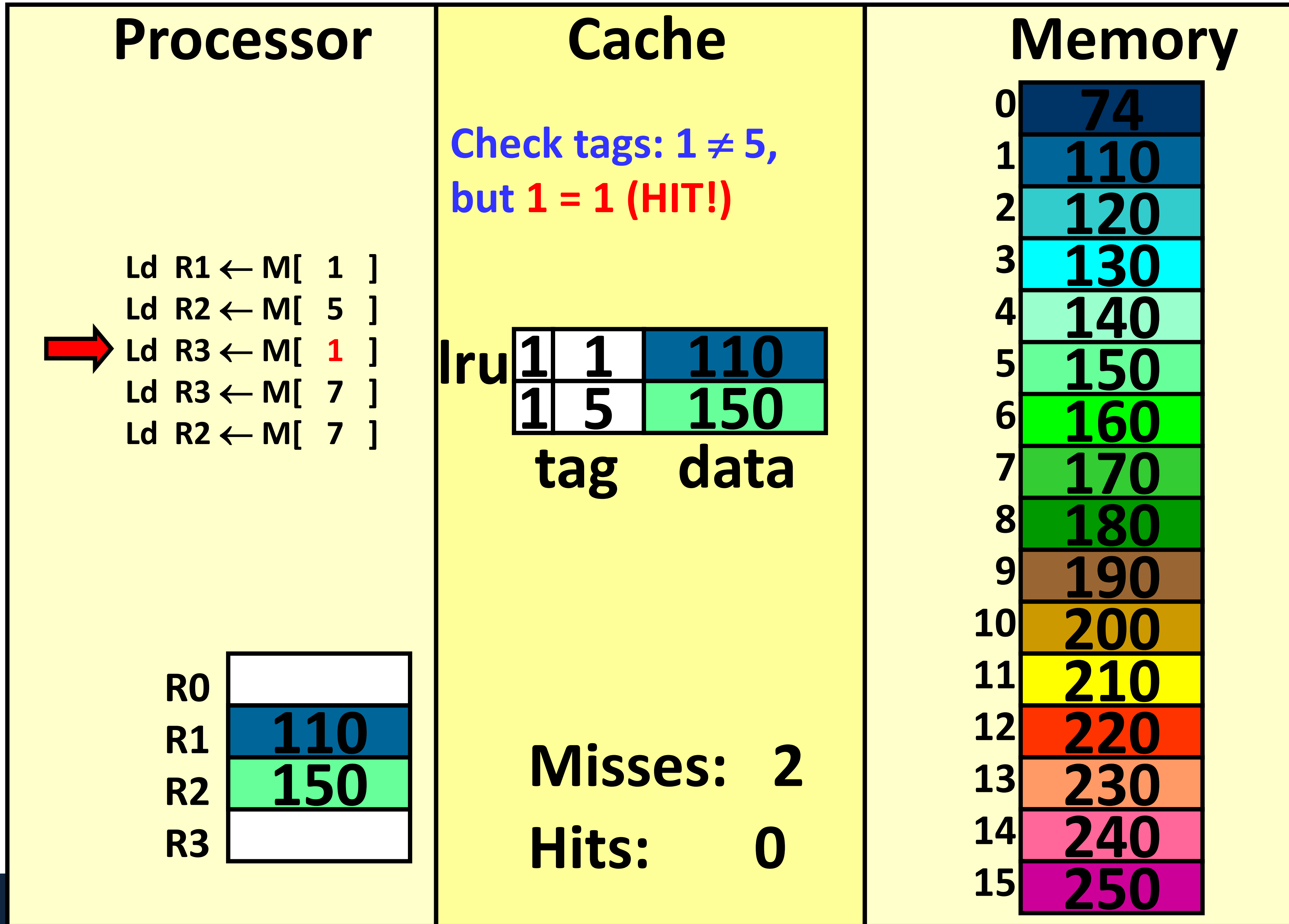
Tag comparison uses hardware called "content-addressable memory (CAM)"



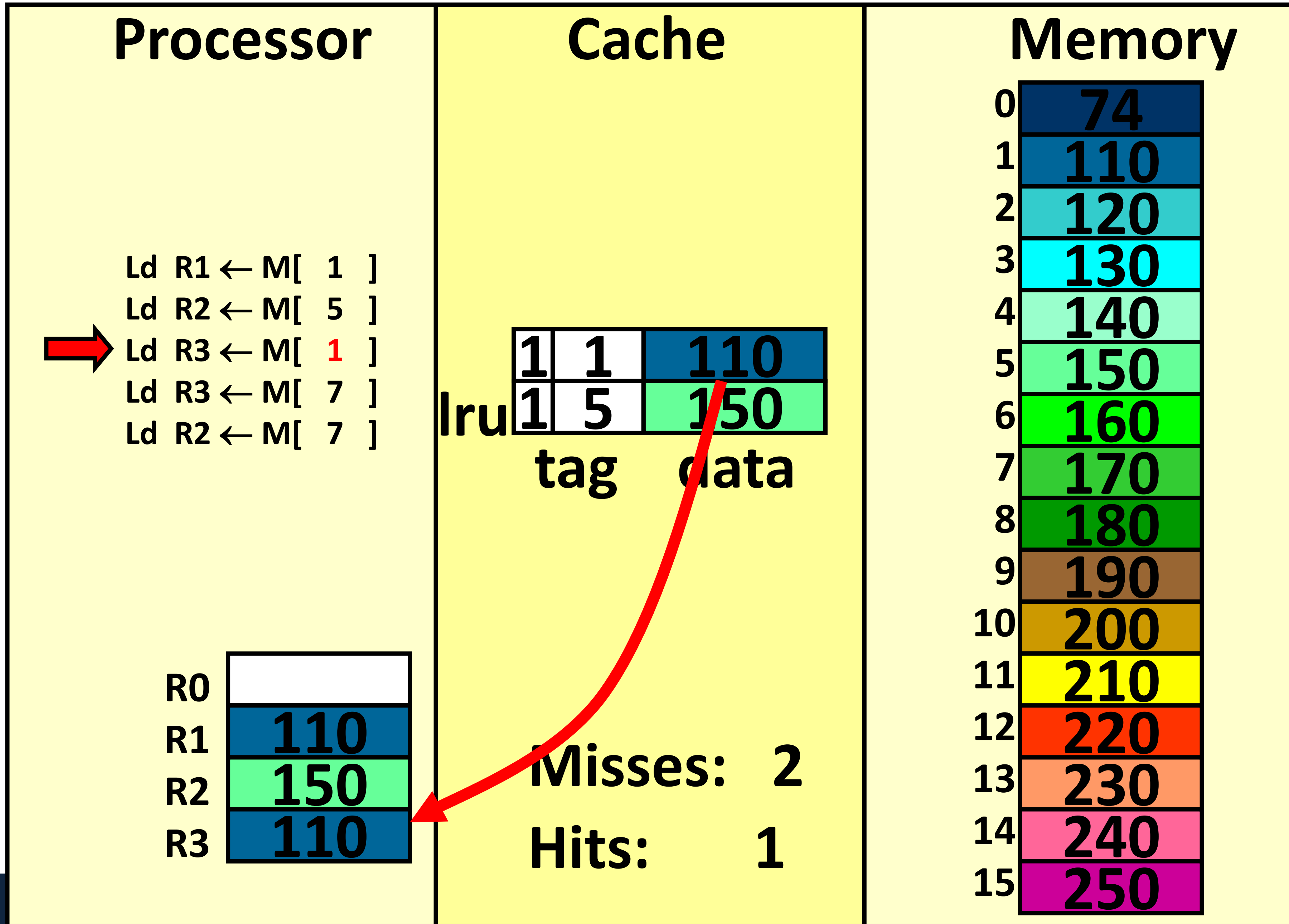
A Very Simple Memory System



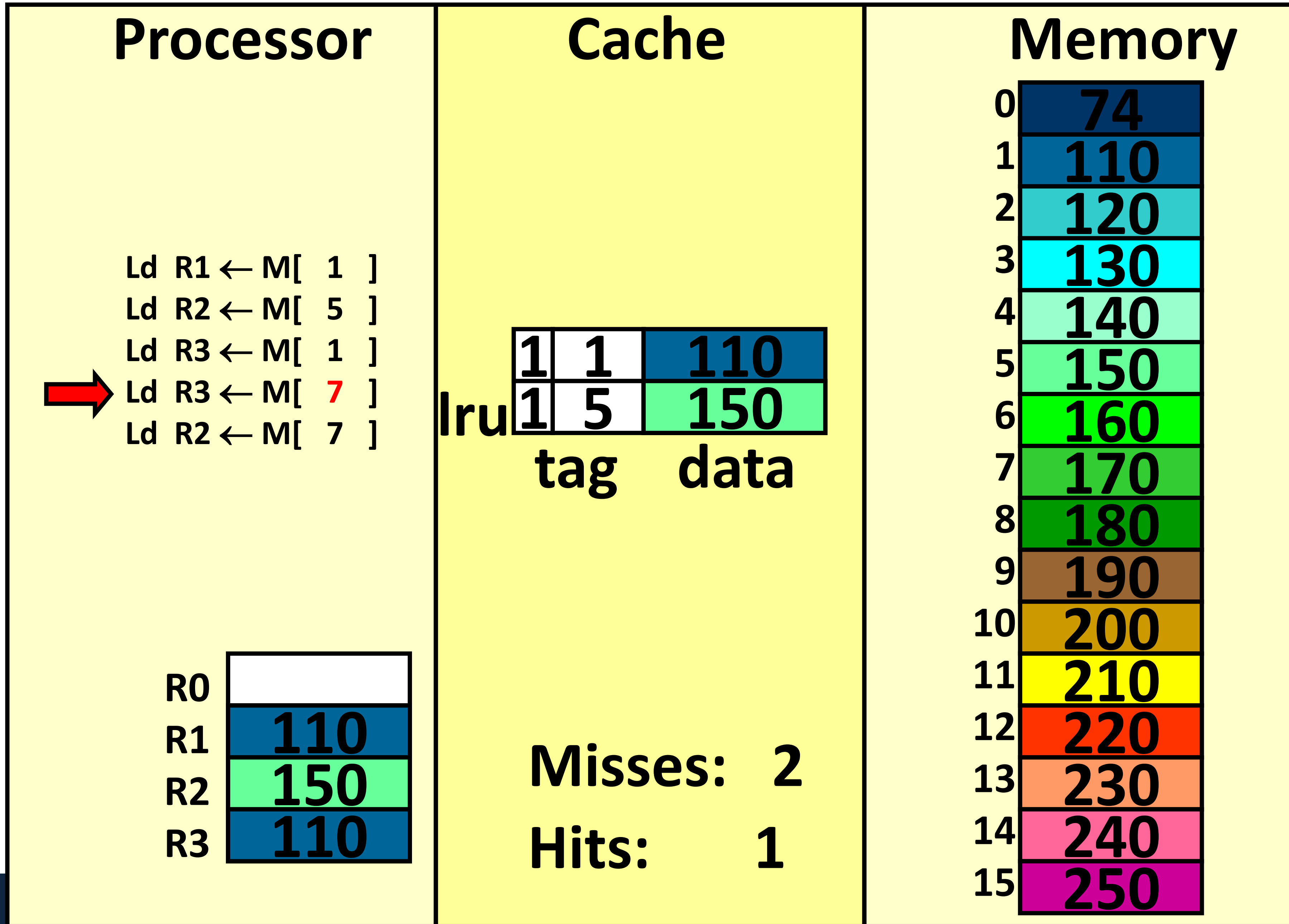
A Very Simple Memory System



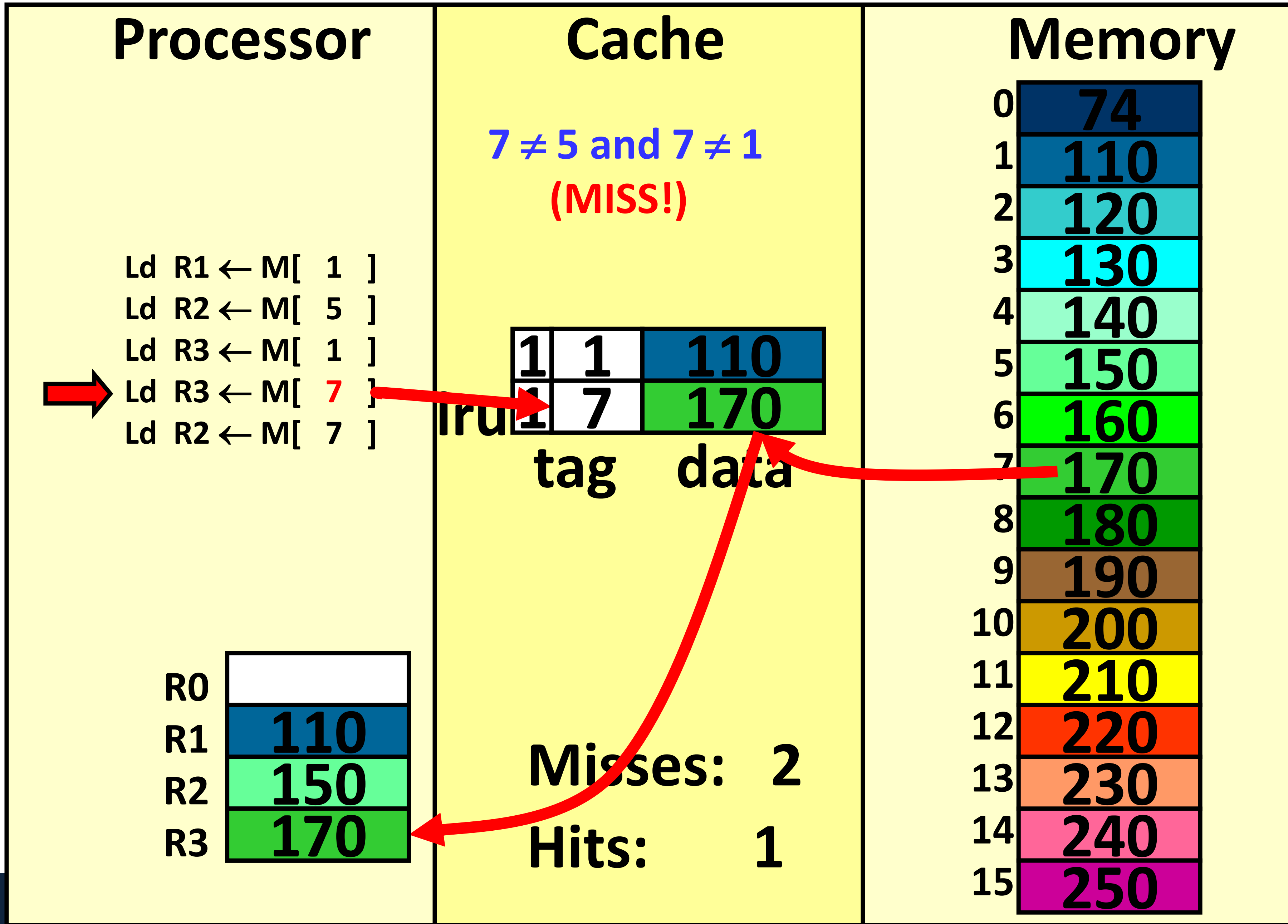
A Very Simple Memory System



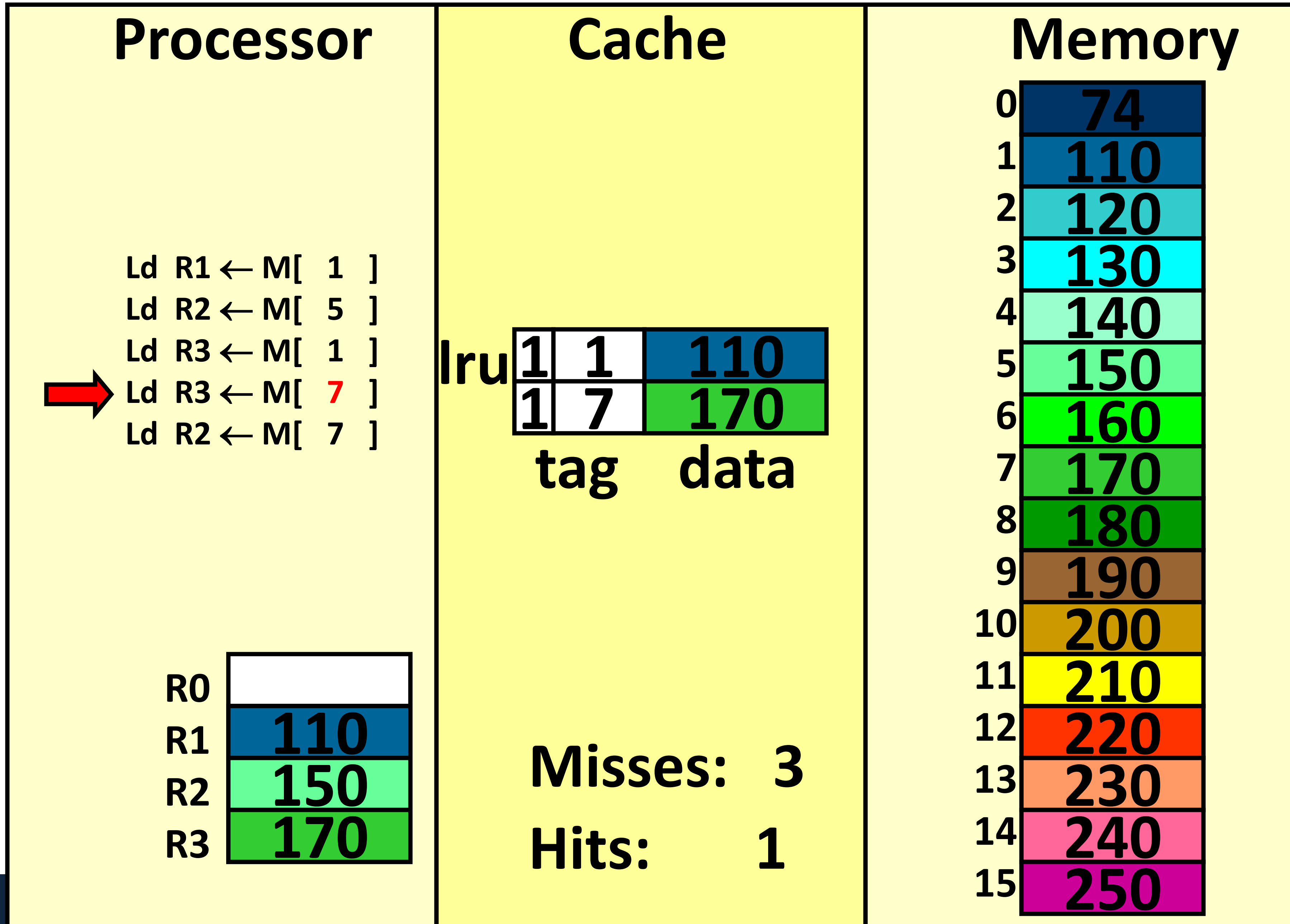
A Very Simple Memory System



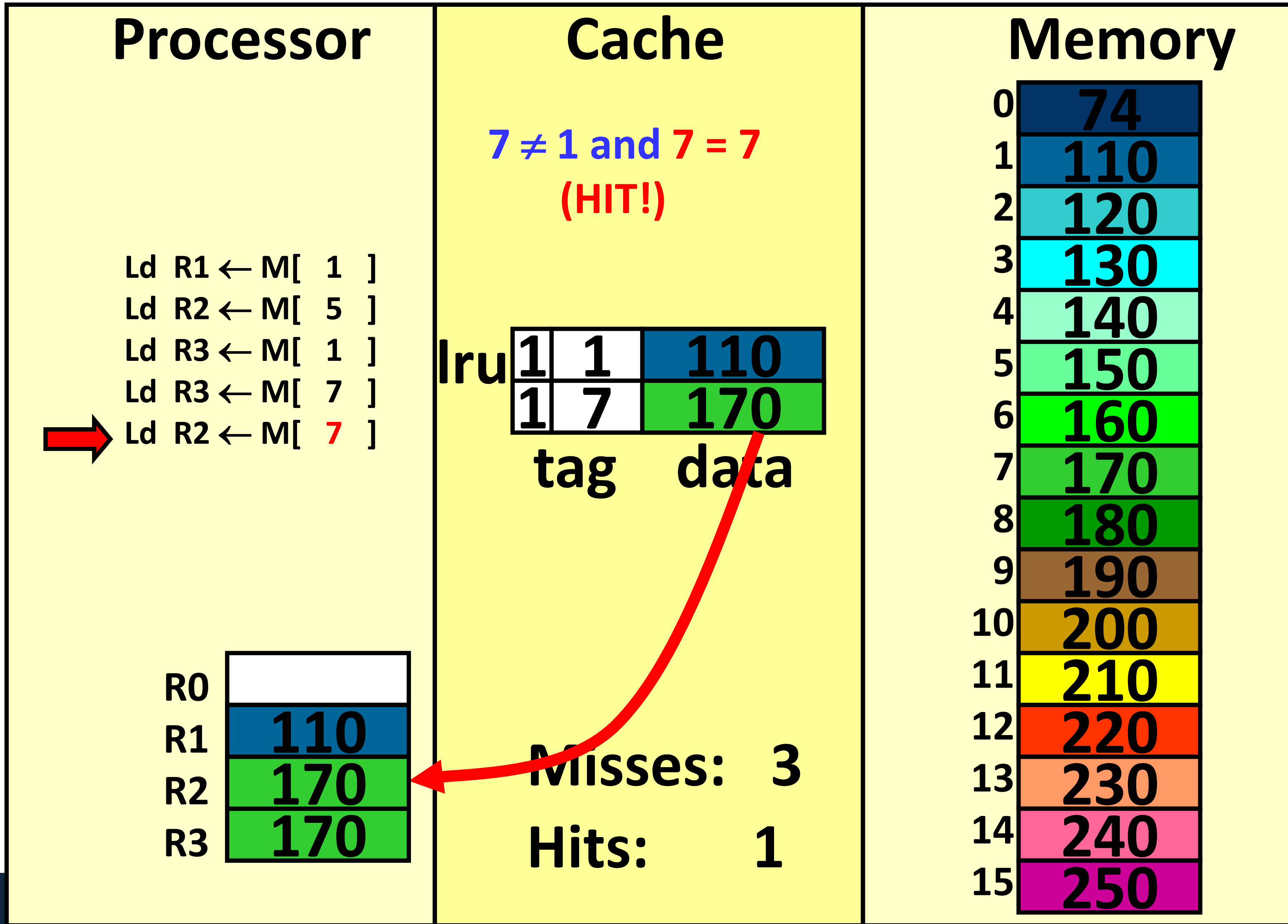
A Very Simple Memory System



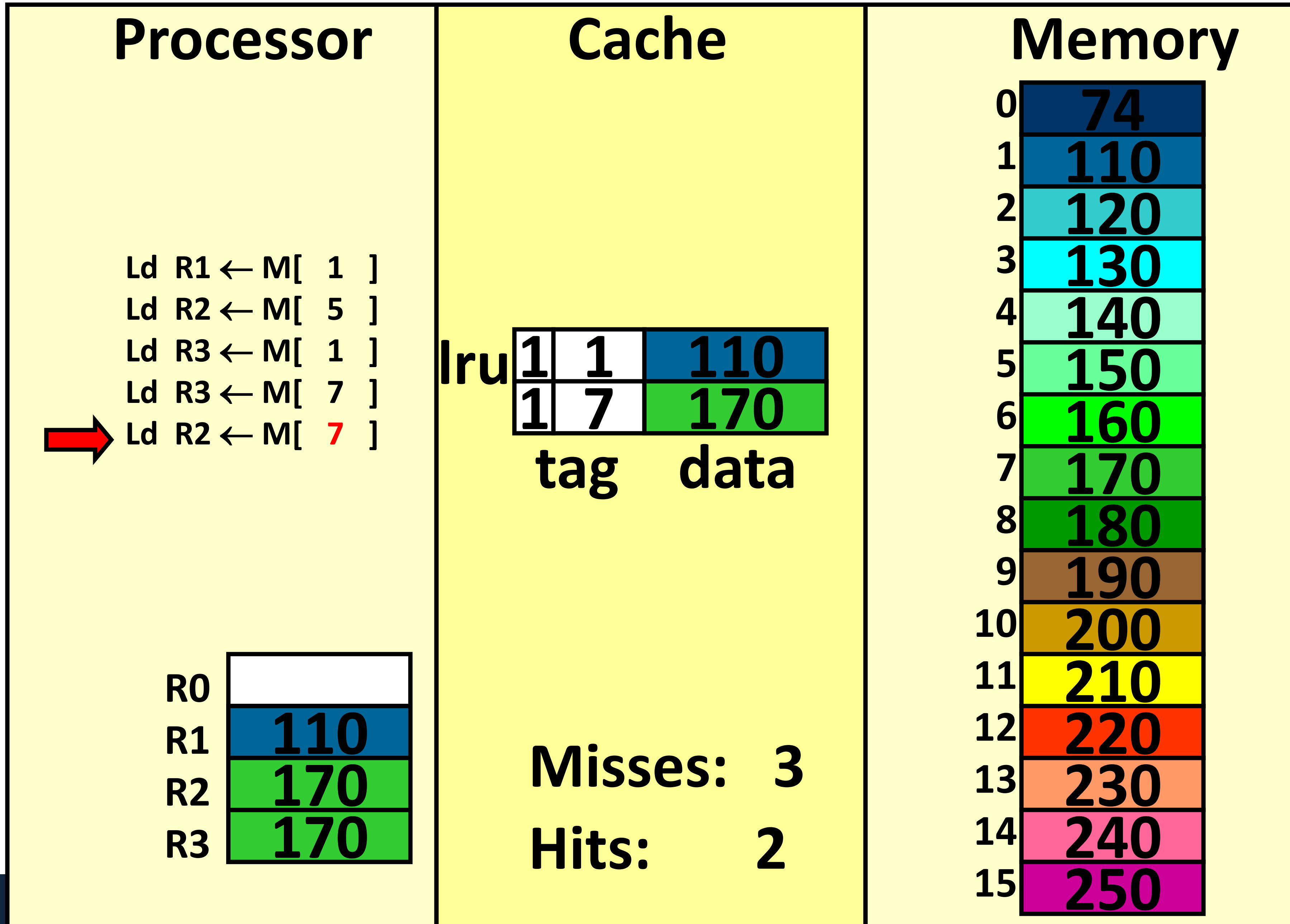
A Very Simple Memory System



A Very Simple Memory System



A Very Simple Memory System



Agenda

- Memory Types
- Memory Hierarchy and Cache Principles
- Cache example
- **How to improve cache**

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

Poll: **Average access time?**

Example Problem

- 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) = 11

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

Next time

- Making our caches bigger and better