15. Cache organization: The basics

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

EECS Department
University of Michigan in Ann Arbor, USA

Memory

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

- We have discussed several methods of implementing storage devices:
  - Static memory (made with logic gates)
  - Dynamic memory (transistor and capacitor)
  - ROM, and other ROM-like storage,
    e.g., flash (floating gate transistors)

Memory Hierarchy

- We want a lot of memory
  - LC2 can handle 2^{18} bytes of memory
  - MIPS can handle 2^{32} bytes of memory
  - Athlon-64 or EM64T can handle 2^{64} bytes of memory

- What are our choices?
  - SRAM, DRAM, ROM, disk, tape, DVD?

Option 1: SRAM

- Fast: ~2ns access time or faster
  - Decoders are big
  - Array is big
    - Why?

- Expensive, high area requirement
  - SRAM: $5.0 per megabyte
    - $0.13 for LC2
    - $20,000 for MIPS
    - $88,000,000,000,000 for Athlon-64

Option 2: DRAM

- Slower: ~60ns access time
  - Hurry up and wait design philosophy doesn't work
  - Must stall for dozens of cycles on each memory load

- Less expensive than SRAM.
  - DRAM costs $0.012 per megabyte
    - $0.00 for LC2
    - $50 for MIPS/Pentium-IV/Athlon-XP
    - $210,000,000,000 for Alpha/G5/x86_64

Option 3: Flash

- Slower still: ~250ns access time
  - Hurry up and wait design philosophy doesn't work
  - Must stall for dozens of cycles on each memory load
- Less expensive than SRAM
  - Flash costs $0.0012 per megabyte
    - $0.00 for LC2
    - $4.9 for MIPS/Pentium-IV/Athlon-XP
    - $21,000,000,000 for Alpha/G5/x86_64
- Non-volatile

Option 4: Disks

- Obnoxiously slow: 3,000,000ns access time
  - We could have stopped with the Intel 4004
- Cheap
  - Disk storage costs $0.000043 per megabyte
    - $0.00 LC2
    - $0.18 for MIPS
    - $760,000,000 for Athlon-64
- Non-volatile

Memory Hierarchy Goals

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

- ❑ Options DRAM, flash, disks are too slow
- ❑ Option SRAM is too expensive

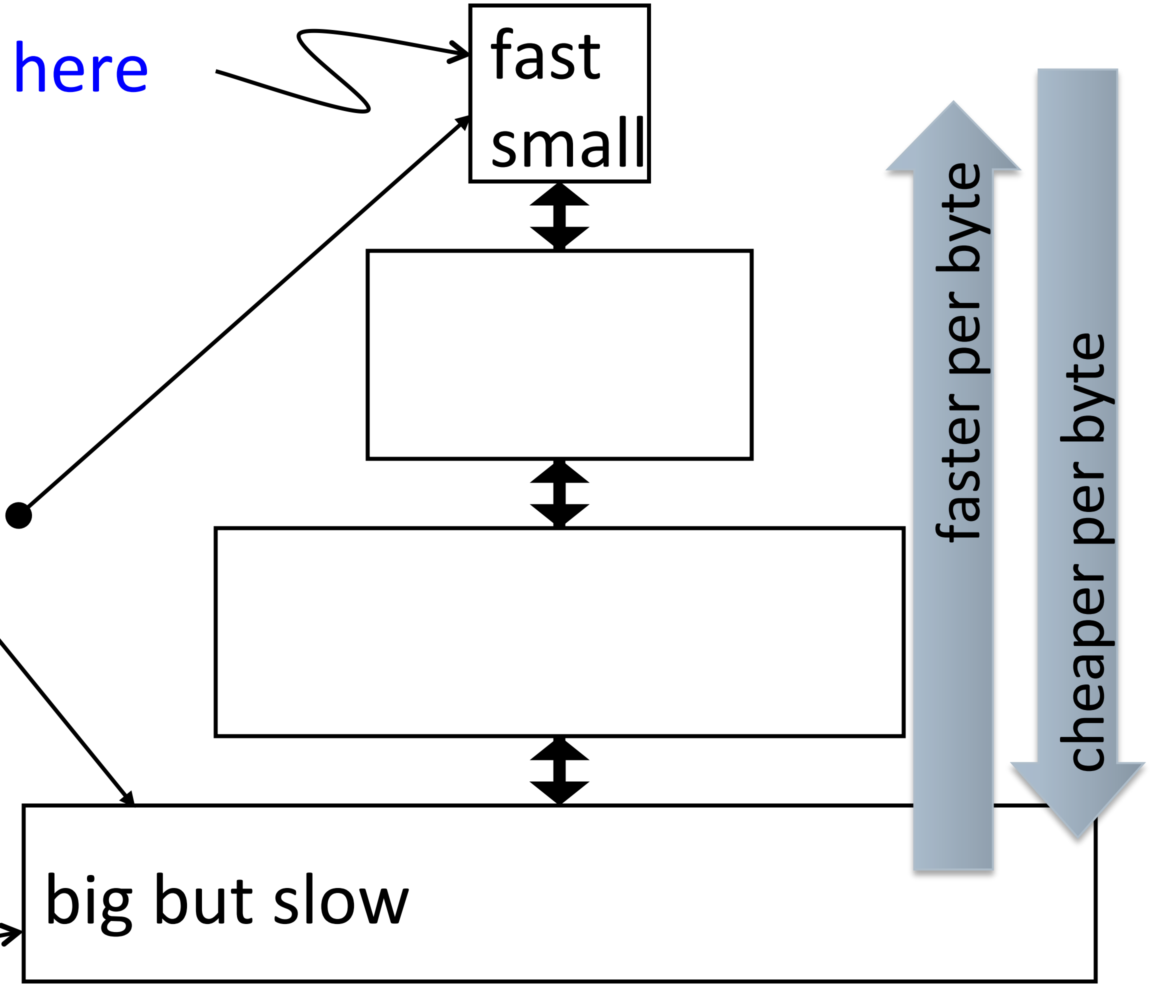
- ❑ How to get best properties of multiple memory technologies?

Memory Hierarchy Goals

move what you use here

With good locality of reference, memory appears as fast as and as large as

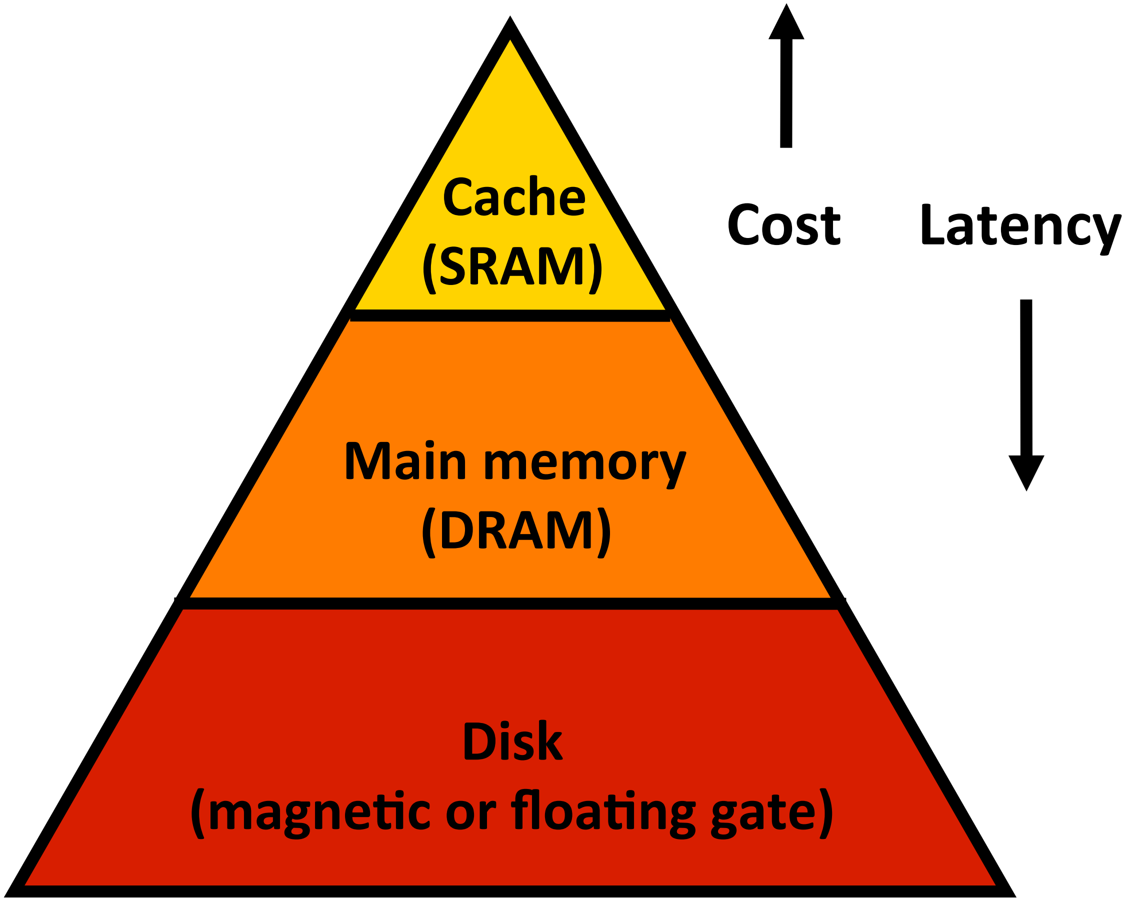
backup everything here



Memory Hierarchy

- Use a small array of SRAM
  - Small so fast and cheap
  - For the **cache** (hopefully covers most loads and stores)
- Use a larger amount of DRAM
  - Cheaper than SRAM, faster than flash/disk
  - For the **main memory**
- Use a lot of flash and/or disk
  - Non-volatile. Cheap. Big
  - For **Virtual memory**
- Don't try to buy 2^{64} bytes of anything
  - Use "virtual memory" to make it look like the entire address range is available
  - A few TB is enough for most desktop machines today, or a smartphone in a few years

Memory hierarchy



Definitions

- ❑ The **architectural** view of memory is
  - What the machine language (or programmer) sees
  - Just a big array
- ❑ Breaking up the memory system into different pieces – cache, main memory (made up of DRAM) and Disk – is **not architectural**
  - The machine language doesn't know about it
  - A new implementation may not break it up in the same way

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 locations are?

Cache Analogy

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

Example Problem

Given the following:

Cache has 1 cycle access time

Main memory has 100 cycle access time

Disk has 10,000 cycles access time

What is the average access time for 100 memory references if 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

addr	data
addr	data

TAG **BLOCK**

- Accessing the cache: compare 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,

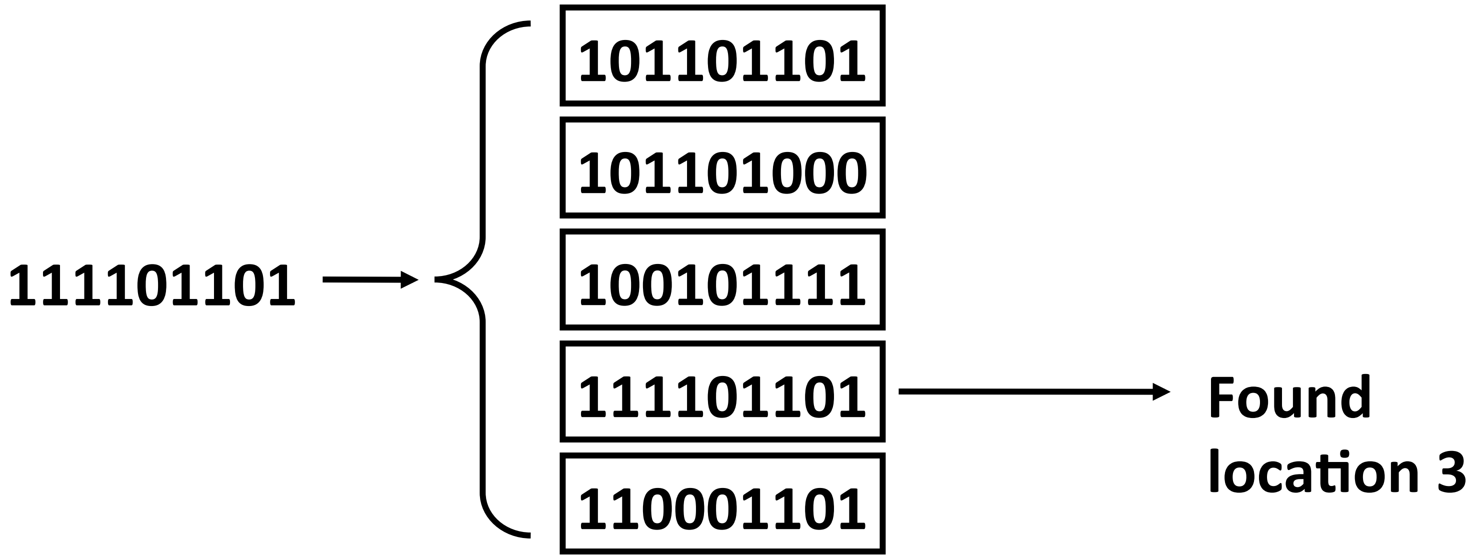
- ❑ Think of memory as a set of data matching a query
  - Instead of an address, we send a key to the memory, asking whether the key exists and, if so, what value it is associated with
  - Memory answers: yes/no (hit/miss for caches) and gives associated value (if there is one)

Operations on CAMs

- **Search**: the primary way to access a CAM
  - Send data to CAM memory
  - Return "found" or "not found"
  - If found, return location of where it was found or associated value

- **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 example



111101101

101101101

101101000

100101111

111101101

110001101

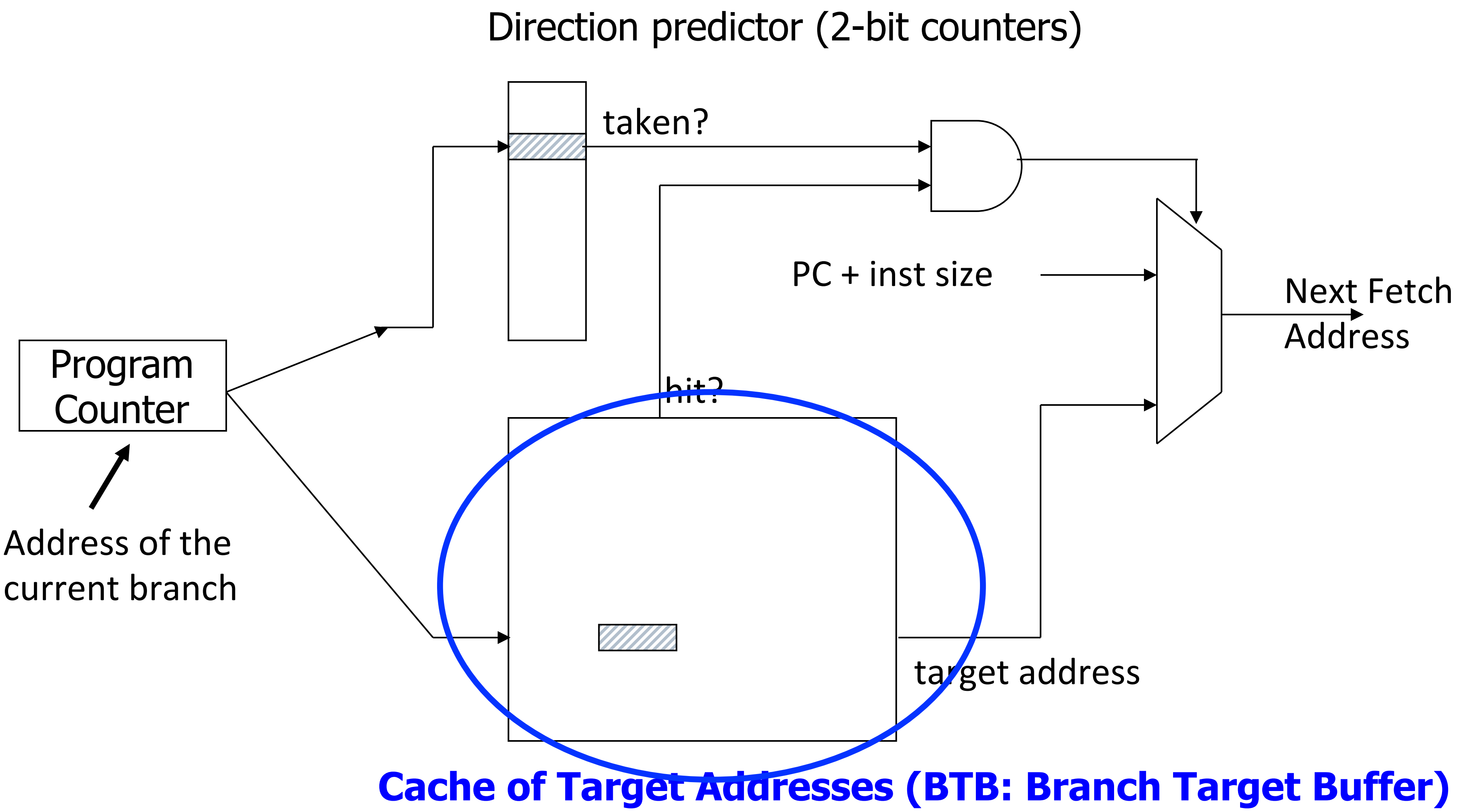
Found location 3

5 storage element CAM array of 9 bits each

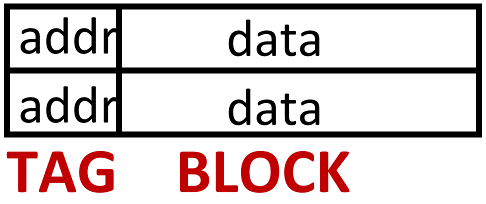
Previous use of CAMs

- You have seen a simple CAM used before. When?

Fetch Stage with Branch Prediction


Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
Program Counter
Address of the current branch
hit?
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Cache Organization



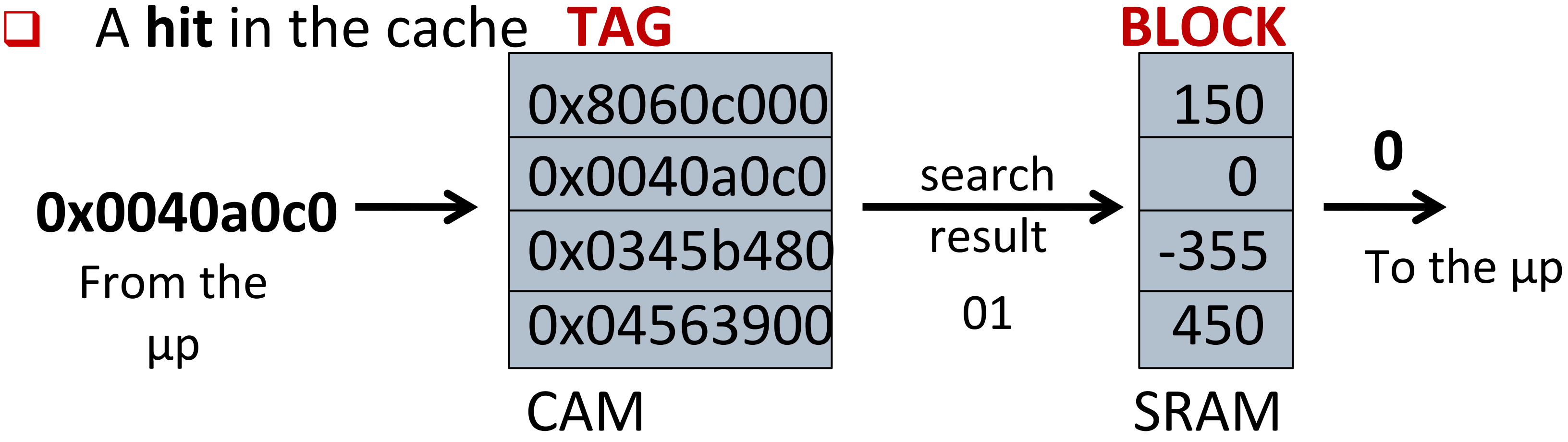
addr	data
addr	data
TAG	**BLOCK**

- 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

Caches: the hardware view

- A **hit** in the cache



TAG

BLOCK

0x0040a0c0

From the µp

CAM	SRAM
0x8060c000	150
0x0040a0c0	0
0x0345b480	-355
0x04563900	450

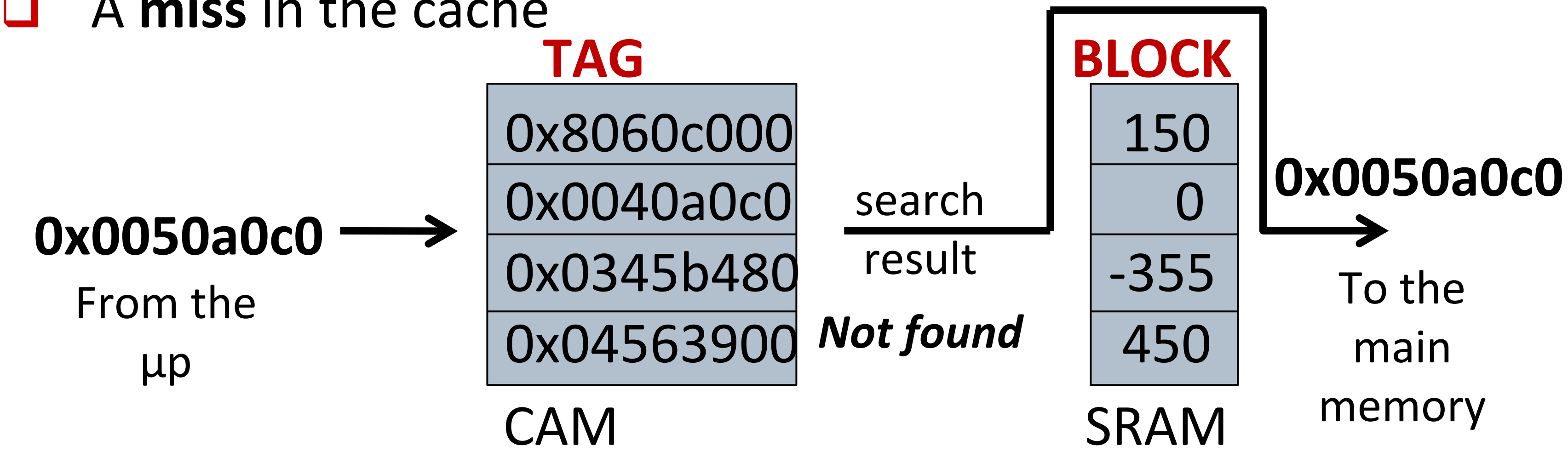
search result

01

0

To the µp

- A **miss** in the cache



TAG

BLOCK

0x0050a0c0

From the µp

CAM	SRAM
0x8060c000	150
0x0040a0c0	0
0x0345b480	-355
0x04563900	450

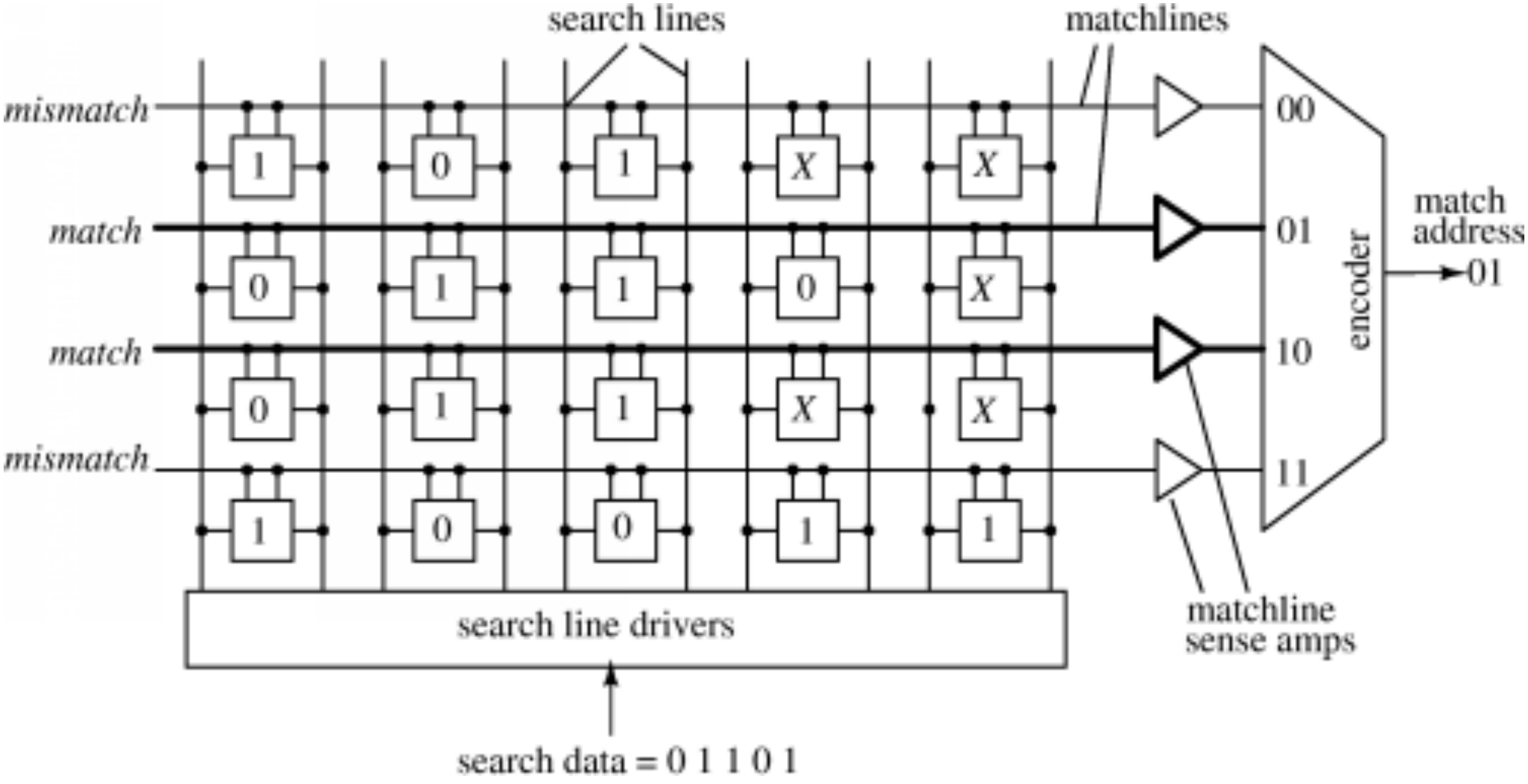
search result

Not found

0x0050a0c0

To the main memory

CAM = content addressable memory



When used in caches, all tags are fully specified (no X)

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?

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?

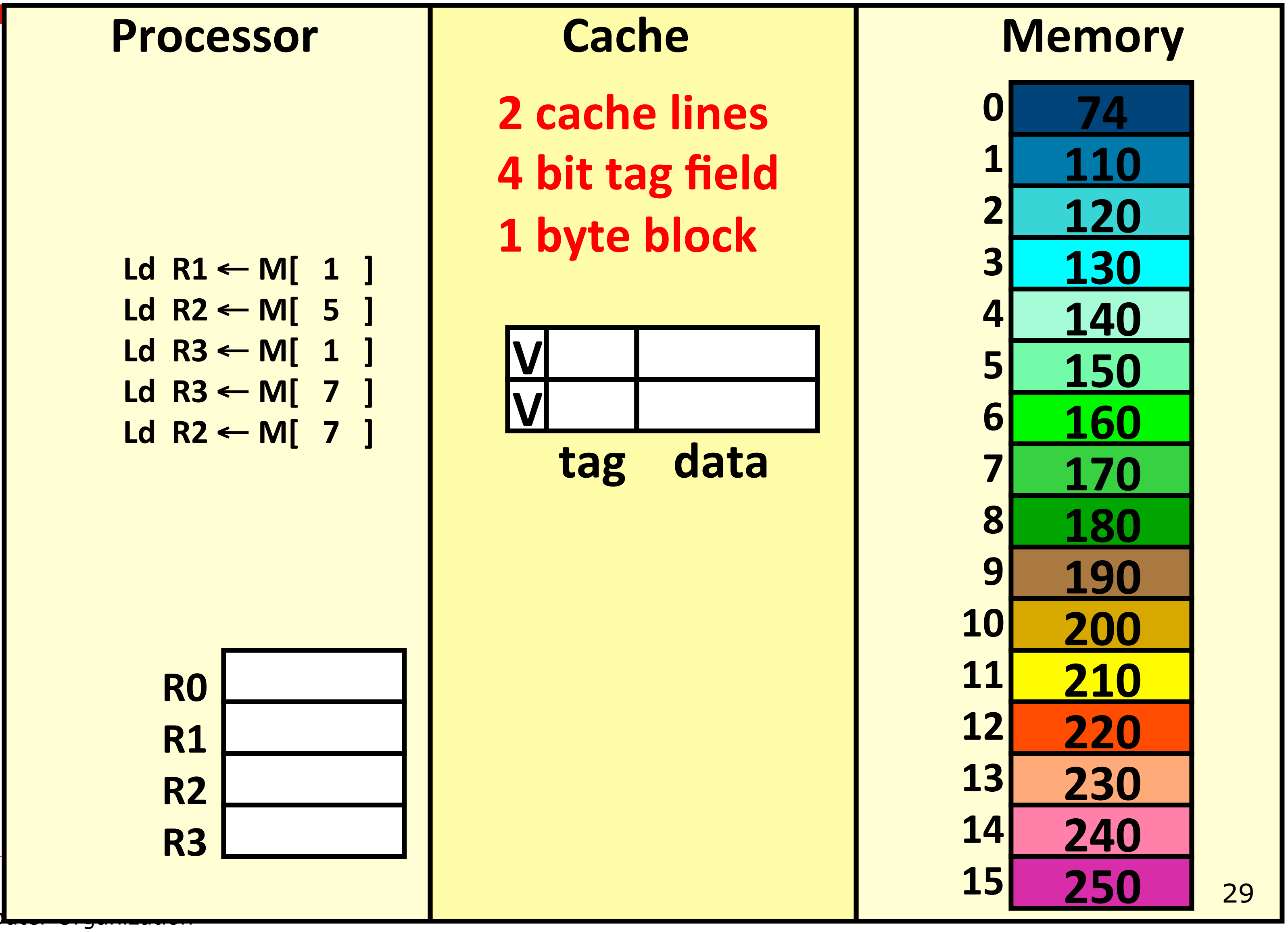
Picking the Most Likely Addresses

- 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
- 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



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	
R2	
R3	

Cache

2 cache lines

4 bit tag field

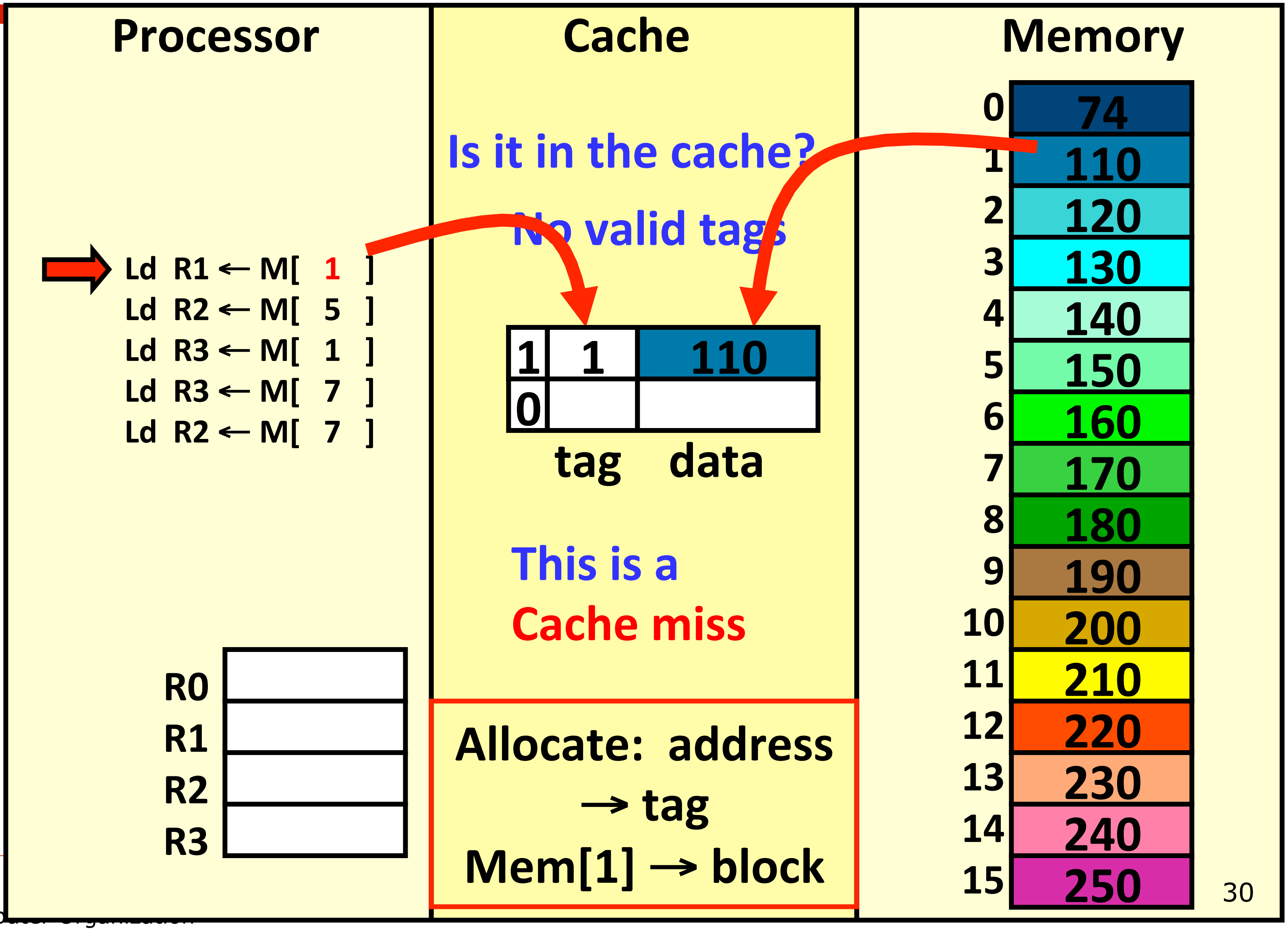
1 byte block

	tag	data
V		
V		

Memory

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

A Very Simple Memory System



Processor

```
Ld  R1 ← M[ 1 ]
Ld  R2 ← M[ 5 ]
Ld  R3 ← M[ 1 ]
Ld  R3 ← M[ 7 ]
Ld  R2 ← M[ 7 ]
```

R0	
R1	
R2	
R3	

Cache

Is it in the cache?

No valid tags

	tag	data
1	1	110
0		

This is a

Cache miss

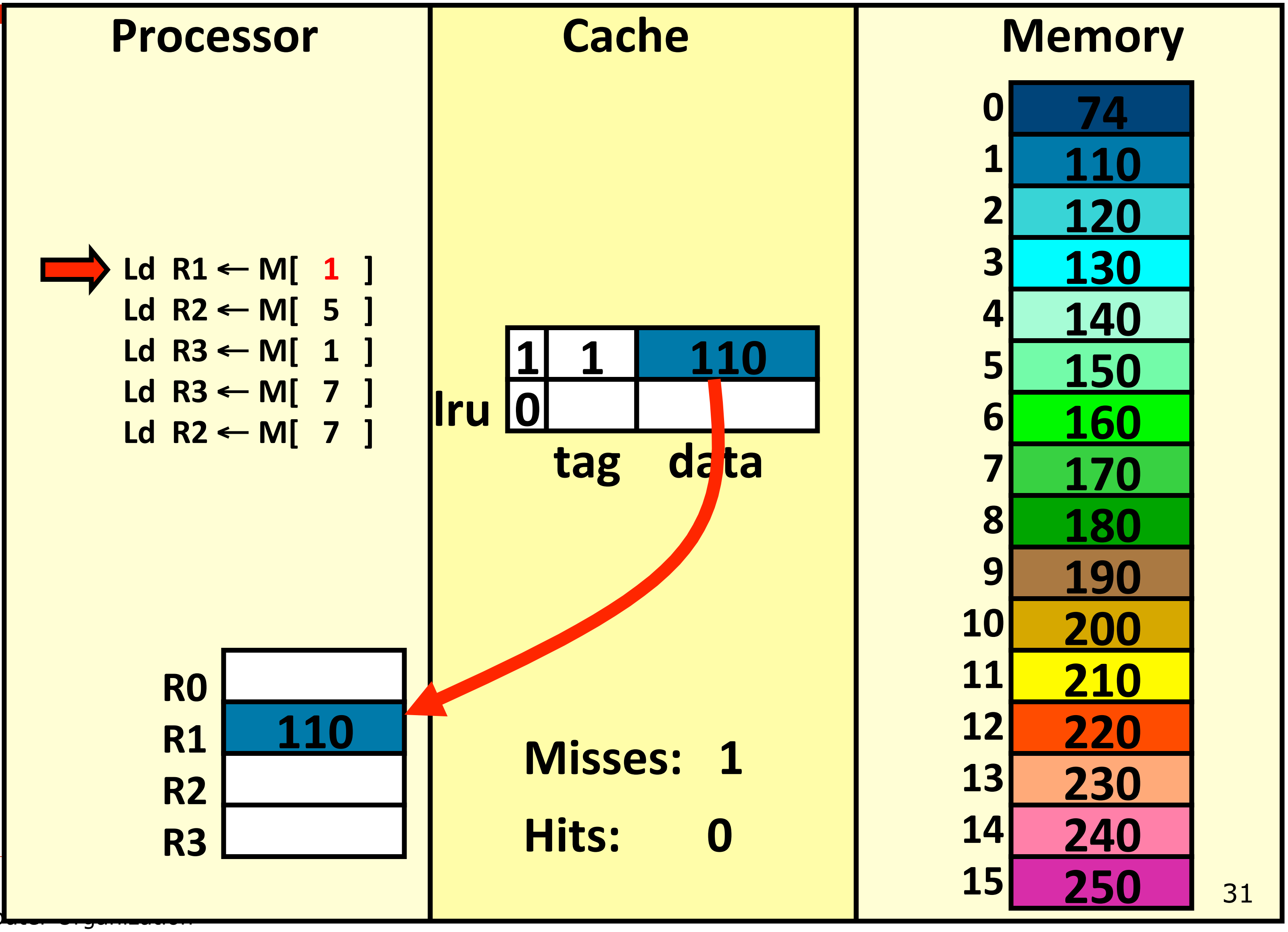
Allocate: address → tag

Mem[1] → block

Memory

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

A Very Simple Memory System



Processor

- Ld R1 ← M[1]
- Ld R2 ← M[5]
- Ld R3 ← M[1]
- Ld R3 ← M[7]
- Ld R2 ← M[7]

R0	
R1	110
R2	
R3	

Cache

lru		tag	data
	1	1	110
lru	0		

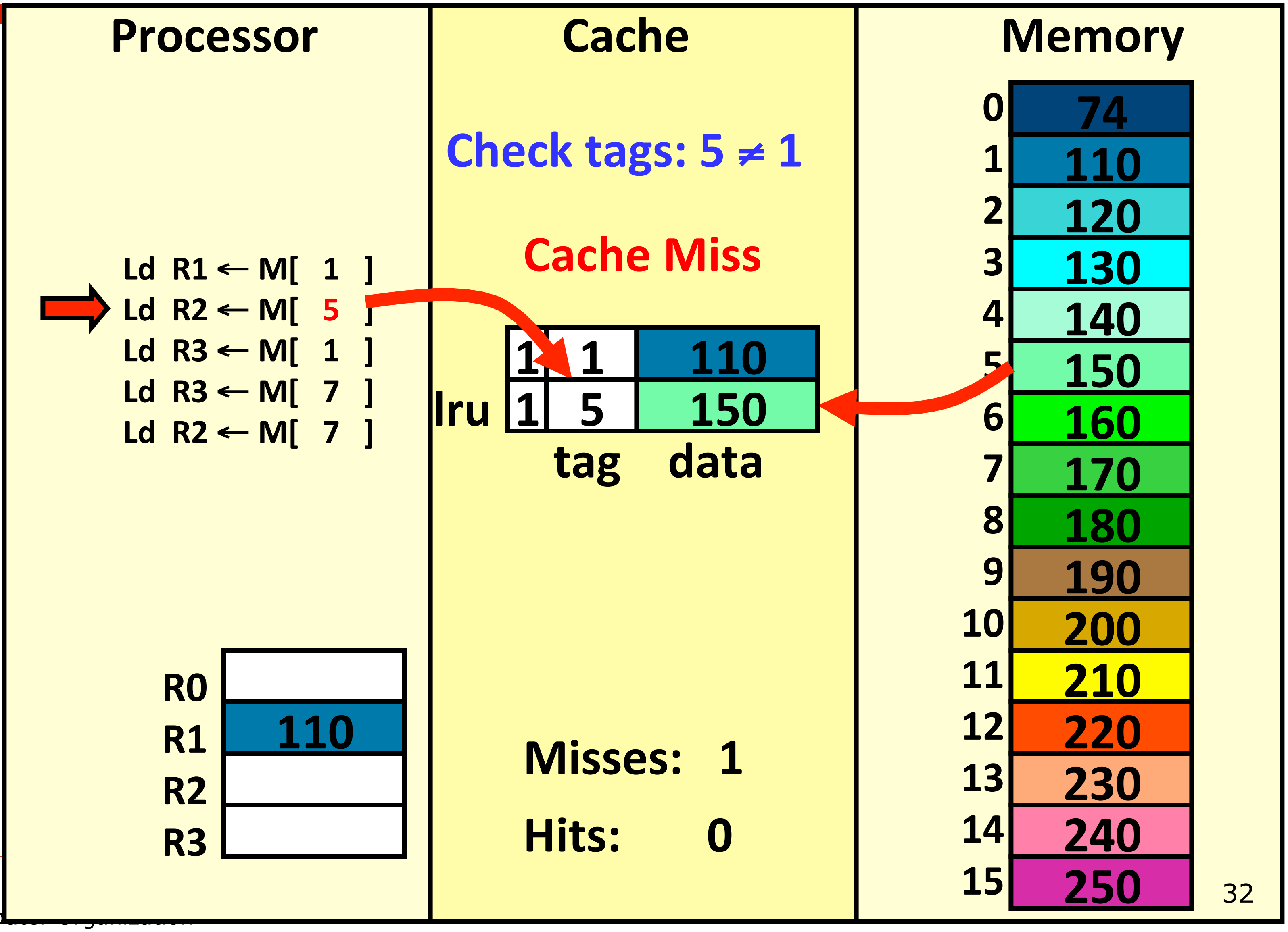
Misses: 1

Hits: 0

Memory

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

A Very Simple Memory System



Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	
R3	

Check tags: 5 ≠ 1

Cache Miss

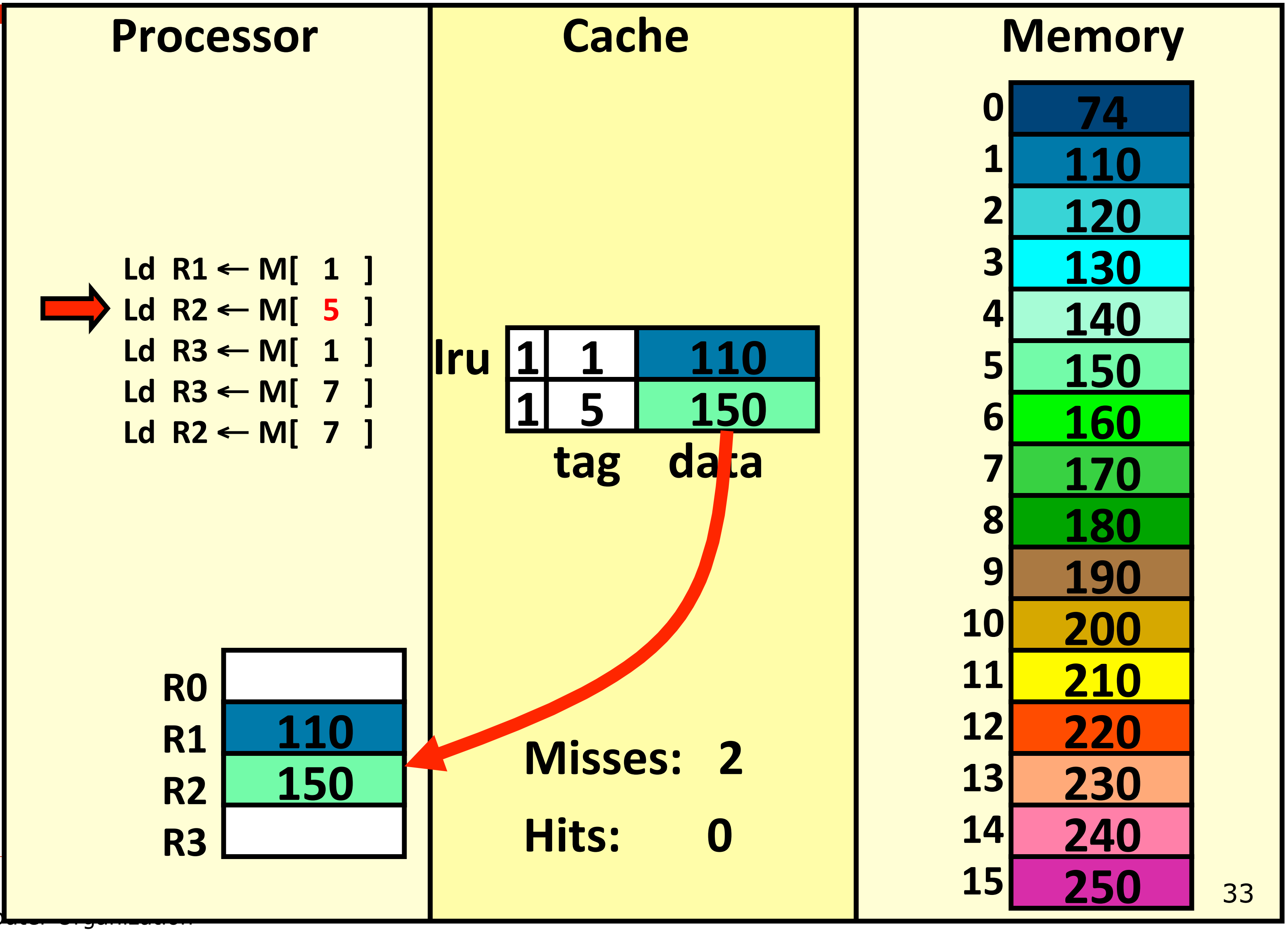
		tag	data
	1	1	110
lru	1	5	150

Misses: 1

Hits: 0

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

A Very Simple Memory System



Processor

Cache

Memory

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

lru	tag	data
1	1	110
1	5	150

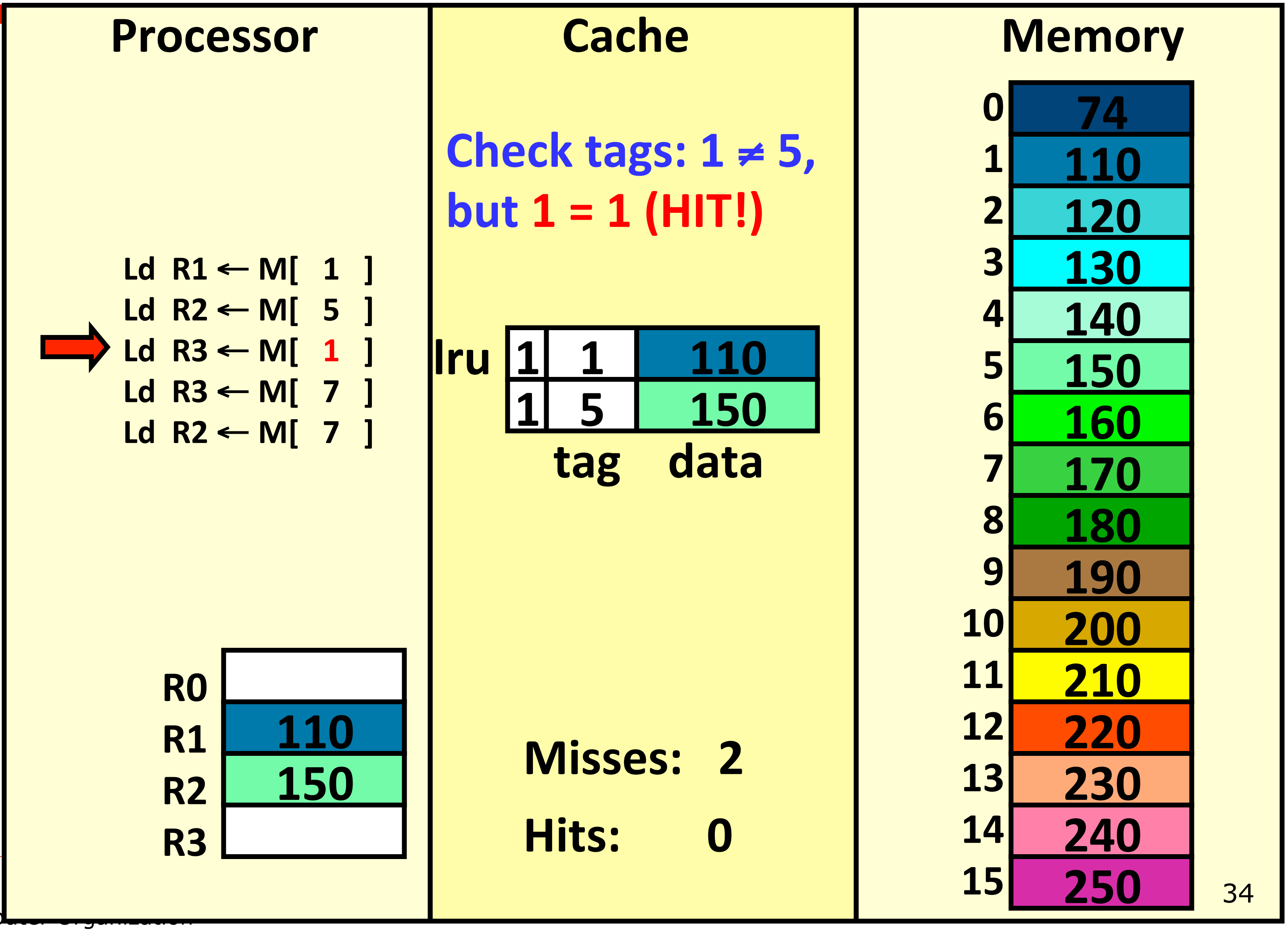
R0	
R1	110
R2	150
R3	

Misses: 2

Hits: 0

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

A Very Simple Memory System



Processor

```
  Ld  R1 ← M[  1  ]
  Ld  R2 ← M[  5  ]
➡ Ld  R3 ← M[  1  ]
  Ld  R3 ← M[  7  ]
  Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	

Cache

Check tags: 1 ≠ 5, but 1 = 1 (HIT!)

lru	tag	data
1	1	110
1	5	150

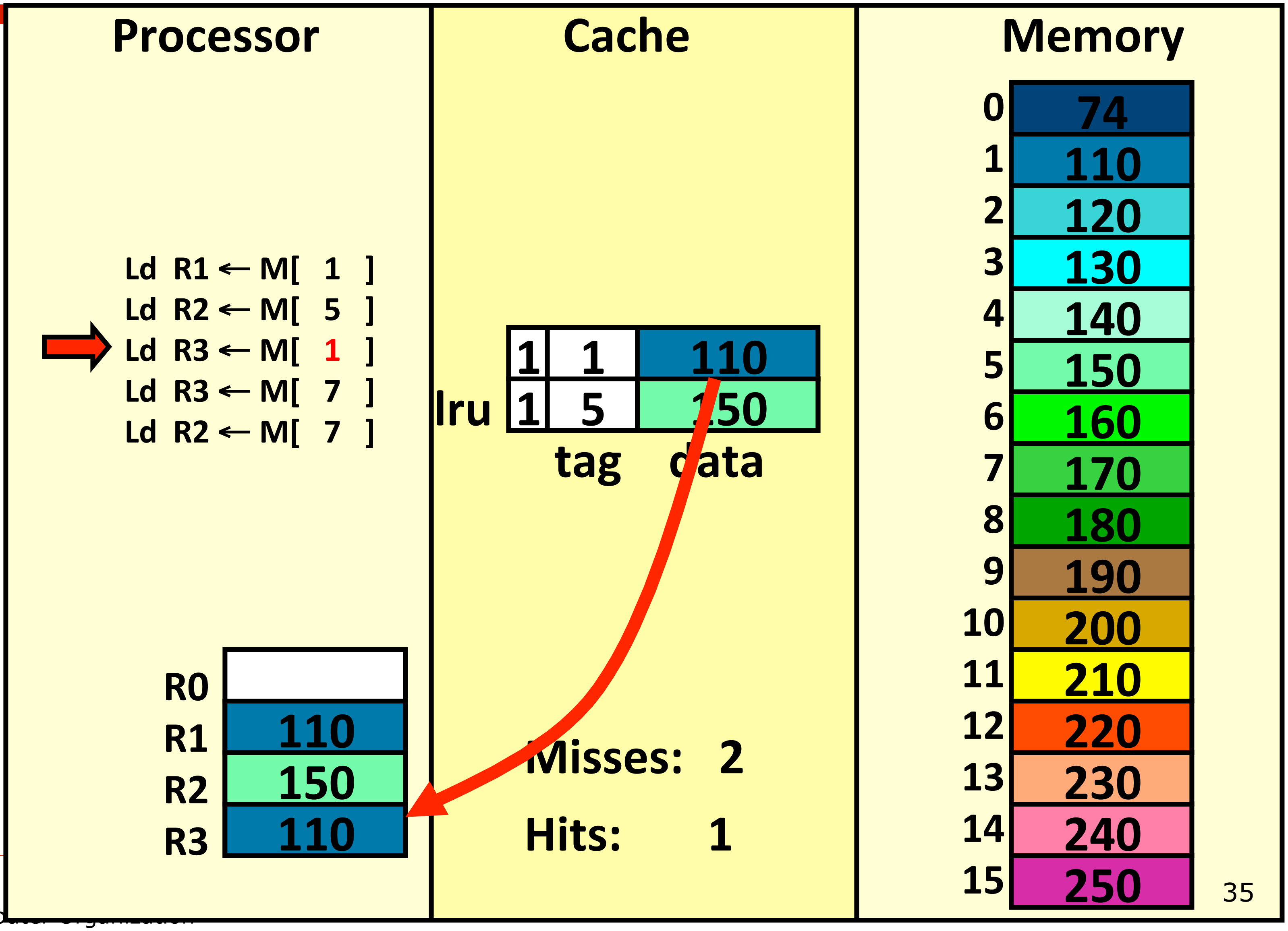
Misses: 2

Hits: 0

Memory

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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	110

Cache

		tag	data
	1	1	110
lru	1	5	150

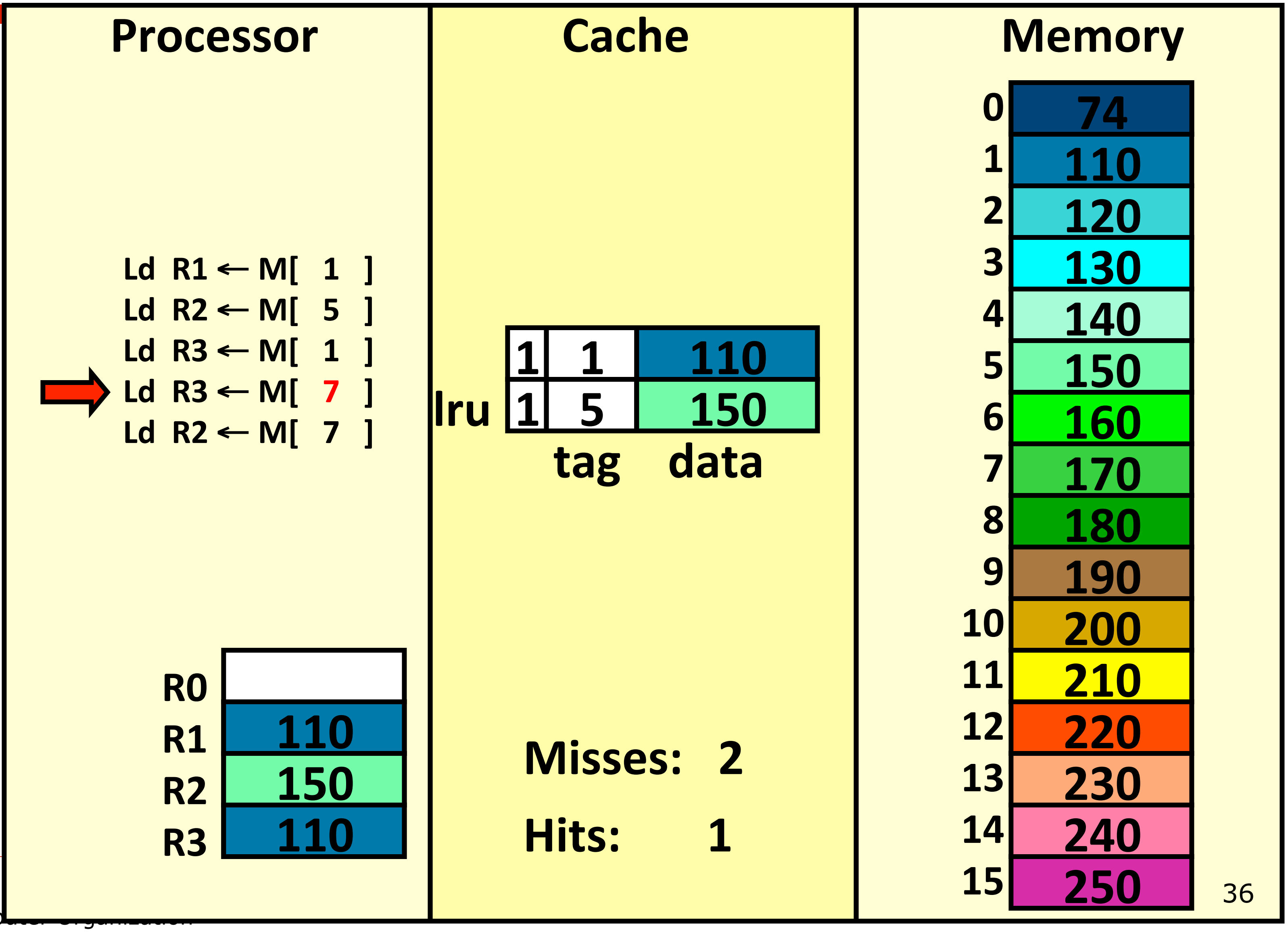
Misses: 2

Hits: 1

Memory

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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
➡ Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	110

Cache

		tag	data
lru	1	1	110
	1	5	150

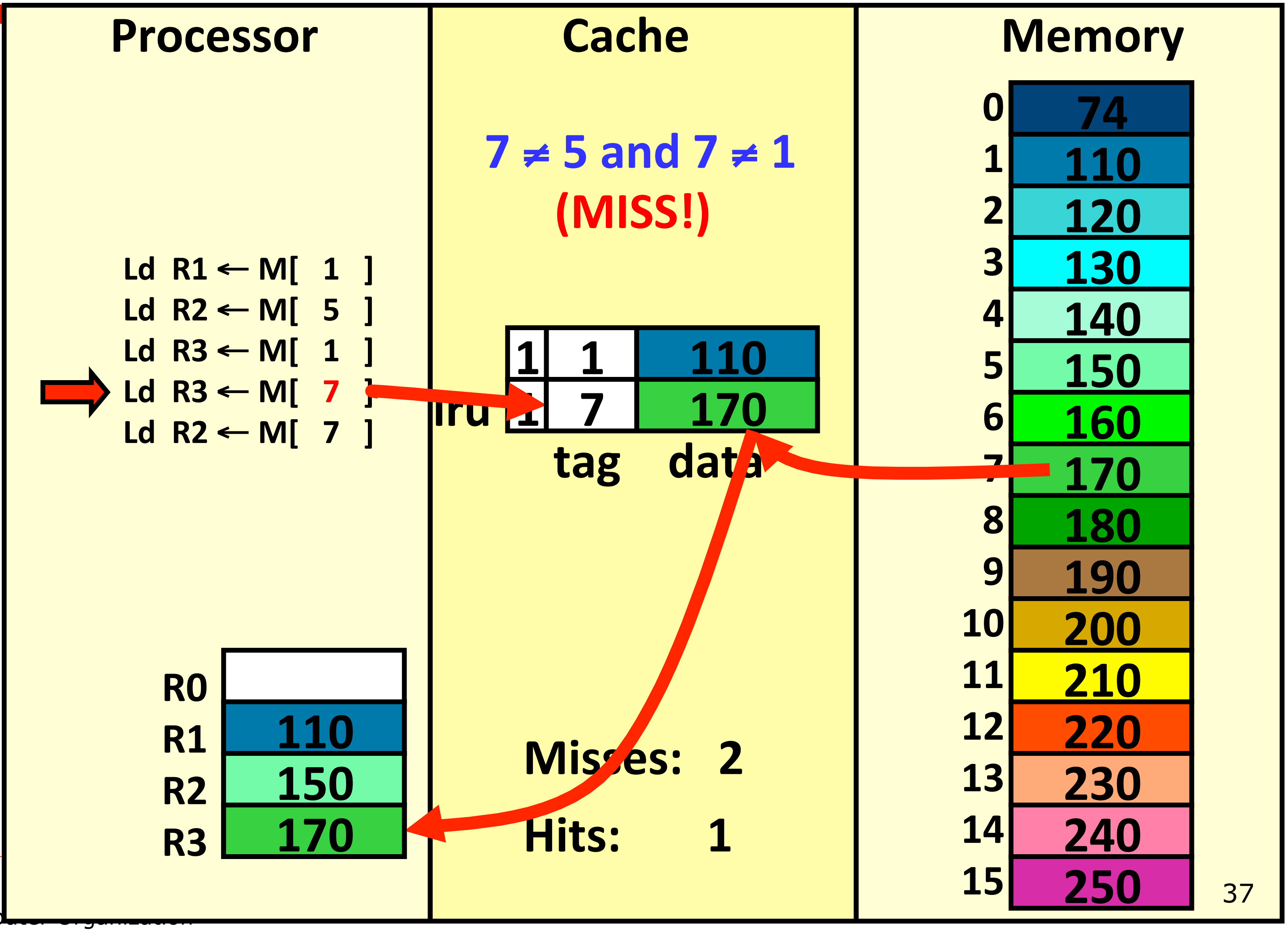
Misses: 2

Hits: 1

Memory

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

A Very Simple Memory System



```
Ld R1 ← M[ 1 ]
Ld R2 ← M[ 5 ]
Ld R3 ← M[ 1 ]
Ld R3 ← M[ 7 ]
Ld R2 ← M[ 7 ]
```

7 ≠ 5 and 7 ≠ 1

(MISS!)

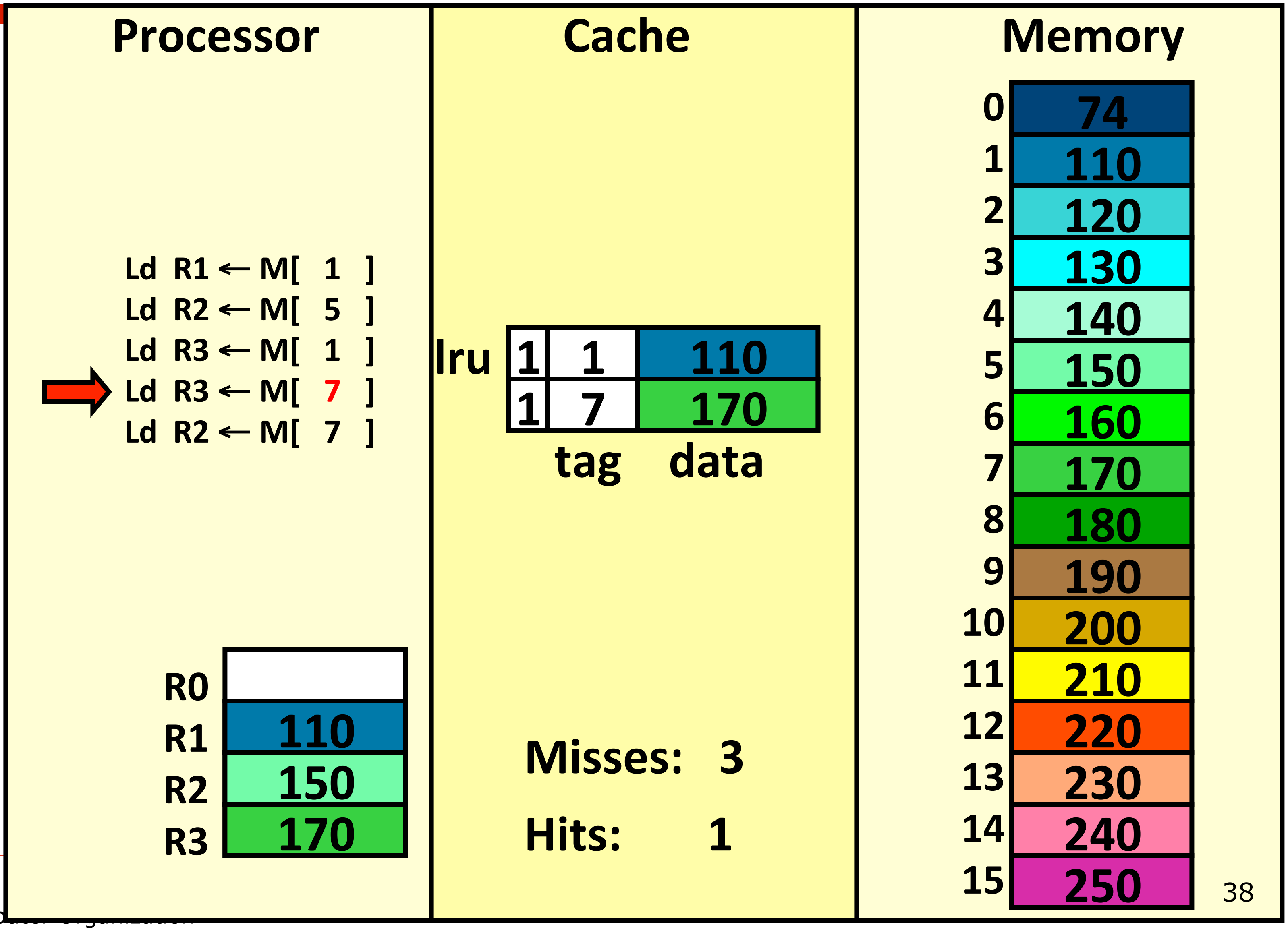
lru	tag	data
1	1	110
1	7	170

Misses: 2

Hits: 1

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

A Very Simple Memory System



Processor

```
  Ld  R1 ← M[  1  ]
  Ld  R2 ← M[  5  ]
  Ld  R3 ← M[  1  ]
➡ Ld  R3 ← M[  7  ]
  Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	170

Cache

lru		tag	data
	1	1	110
	1	7	170

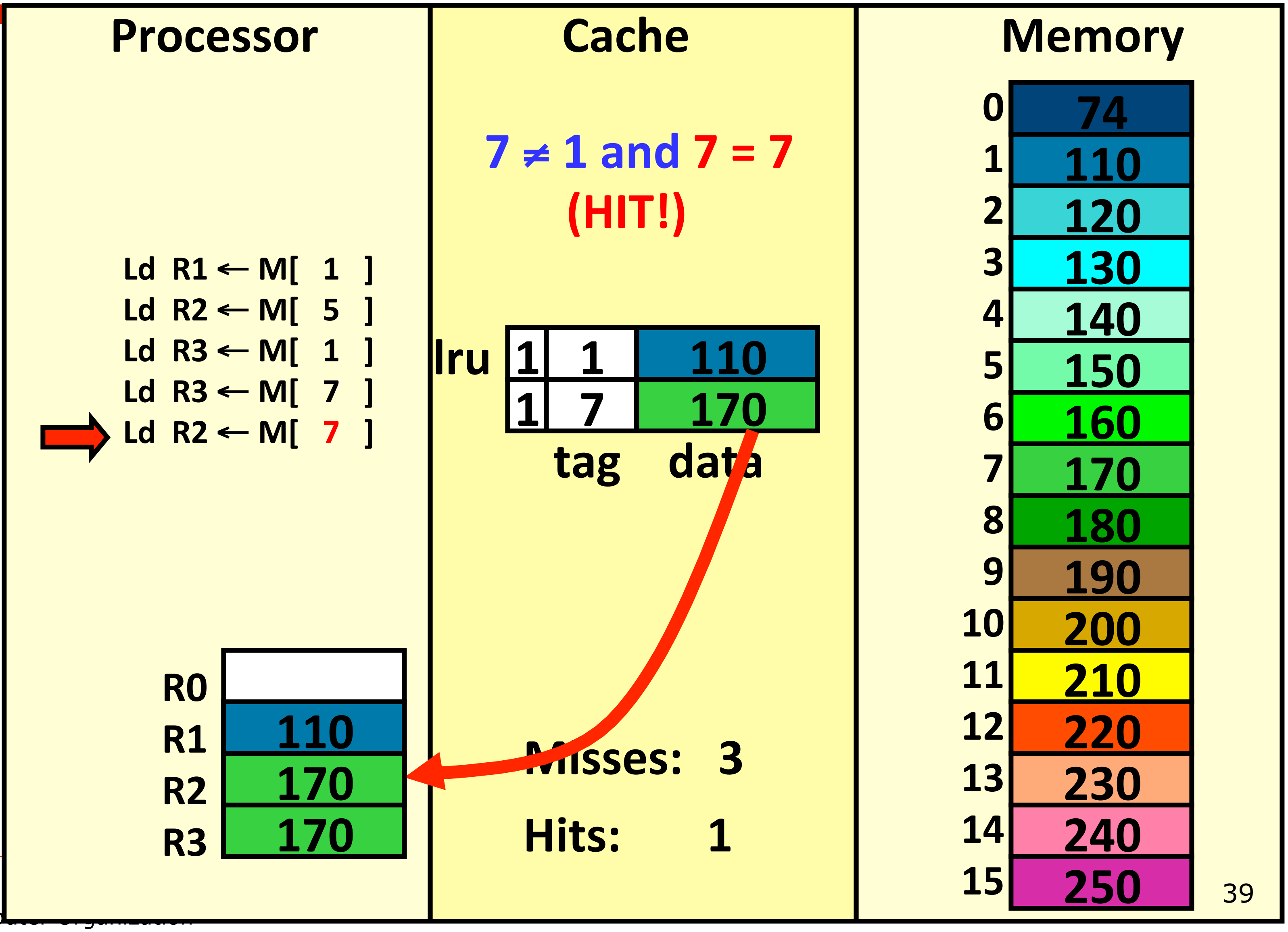
Misses: 3

Hits: 1

Memory

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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	170
R3	170

Cache

7 ≠ 1 and 7 = 7
(HIT!)

lru		tag	data
	1	1	110
	1	7	170

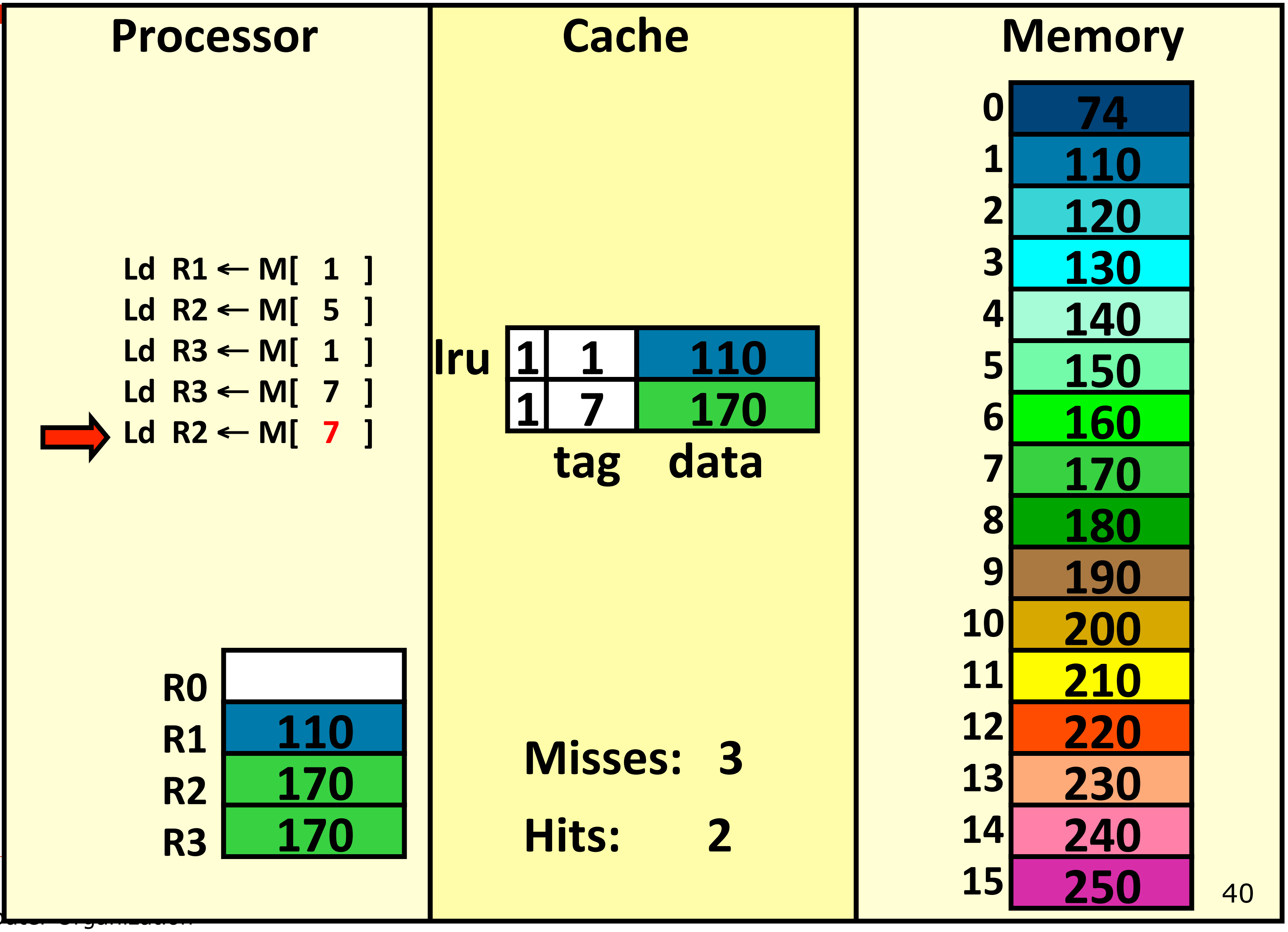
Misses: 3

Hits: 1

Memory

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

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	170
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

Misses: 3

Hits: 2

Memory

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

Calculating Average Access Latency

- ❑ Avg latency
  = cache latency × hit rate + memory latency × miss rate

- ❑ Avg latency for our example cache
  = 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 our example cache cost (in bits)?
  - 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
    - 2 valid bits
- 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 micro-architectural

- ❑ Cache bigger units than bytes
  - Each block has a single tag, and blocks can be whatever size we choose.

- ❑ To Be Continued…