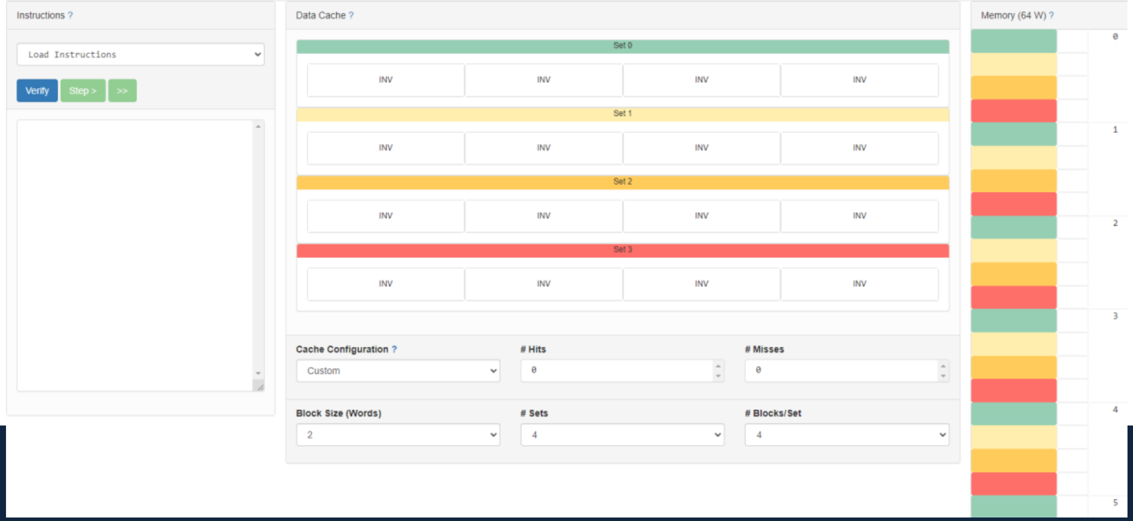
# EECS 370

## Classifying Cache Misses

# Announcements

- Don't forget pre-lab quiz tonight!
- P3 due tomorrow
- P4 released today
    - checkout cache simulator on website!
- HW 3 due Monday



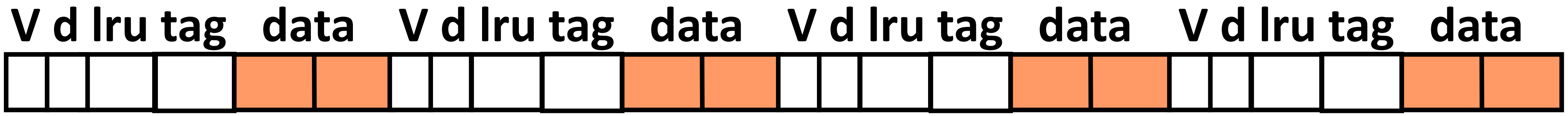
| Set | | | | |
|---|---|---|---|---|
| Set 0 | INV | INV | INV | INV |
| Set 1 | INV | INV | INV | INV |
| Set 2 | INV | INV | INV | INV |
| Set 3 | INV | INV | INV | INV |

Cache Configuration ?: Custom
# Hits: 0
# Misses: 0
Block Size (Words): 2
# Sets: 4
# Blocks/Set: 4

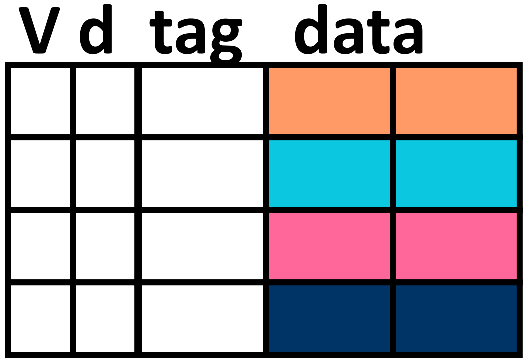
# Cache Organization Comparison

Block size = 2 bytes, total cache size = 8 bytes for all caches

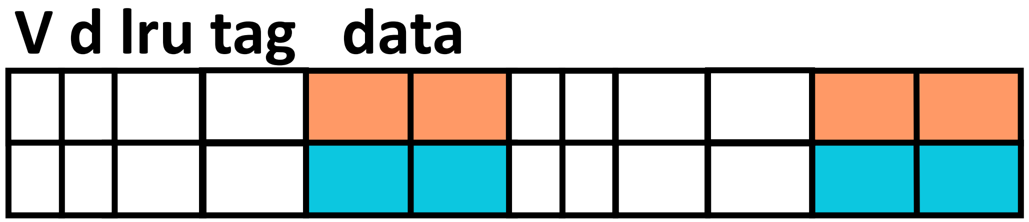
1. Fully associative (4-way associative)



2. Direct mapped



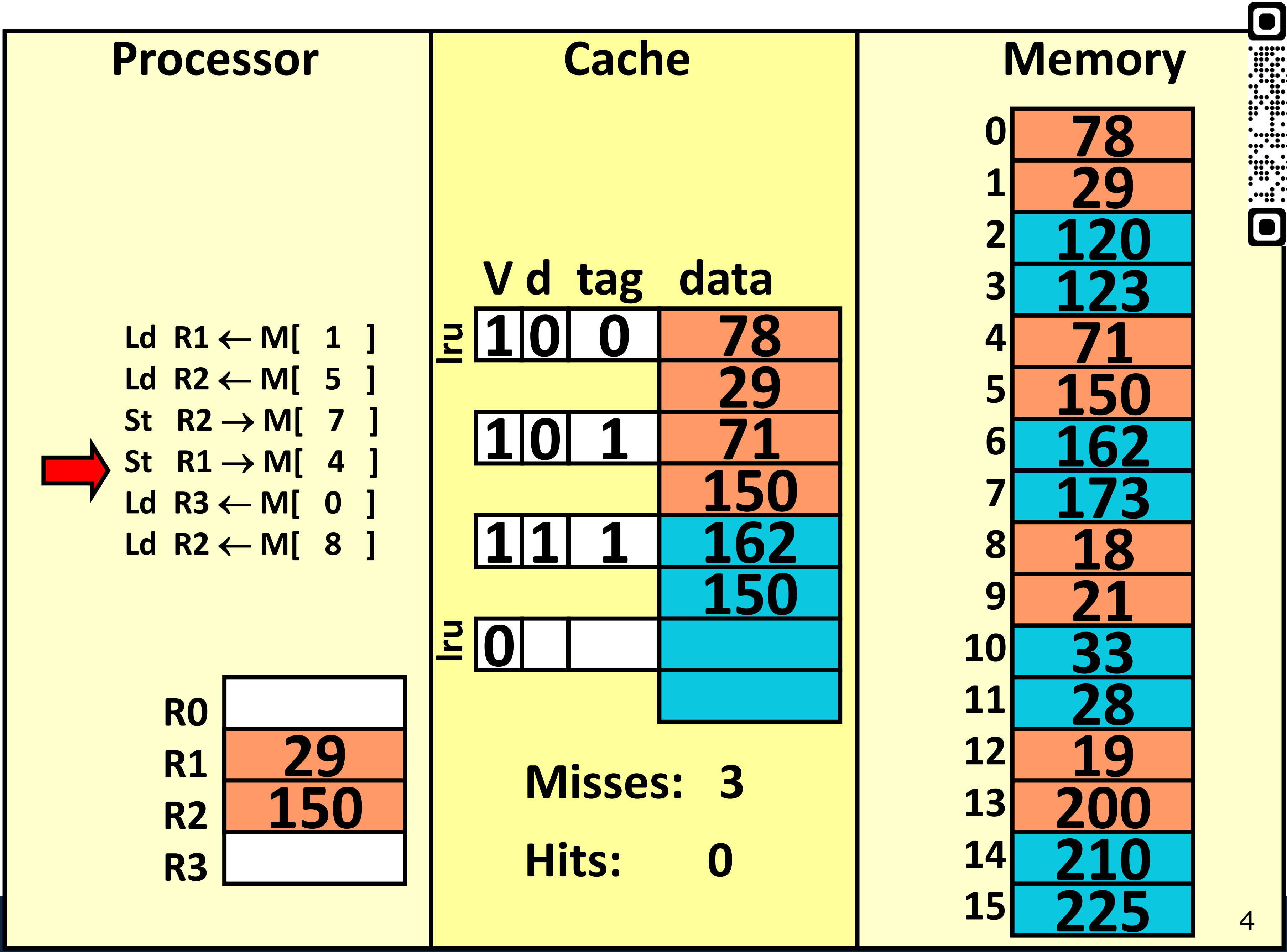
3. 2-way associative



# Set-associative cache (REF 4)

**Poll:** Finish the remaining references

https://bit.ly/3x7nKpP



## Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  7  ]
St  R1 → M[  4  ]
Ld  R3 ← M[  0  ]
Ld  R2 ← M[  8  ]
```

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 150 |
| R3 | |

## Cache

| | V | d | tag | data |
|---|---|---|---|---|
| lru | 1 | 0 | 0 | 78 |
| | | | | 29 |
| | 1 | 0 | 1 | 71 |
| | | | | 150 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

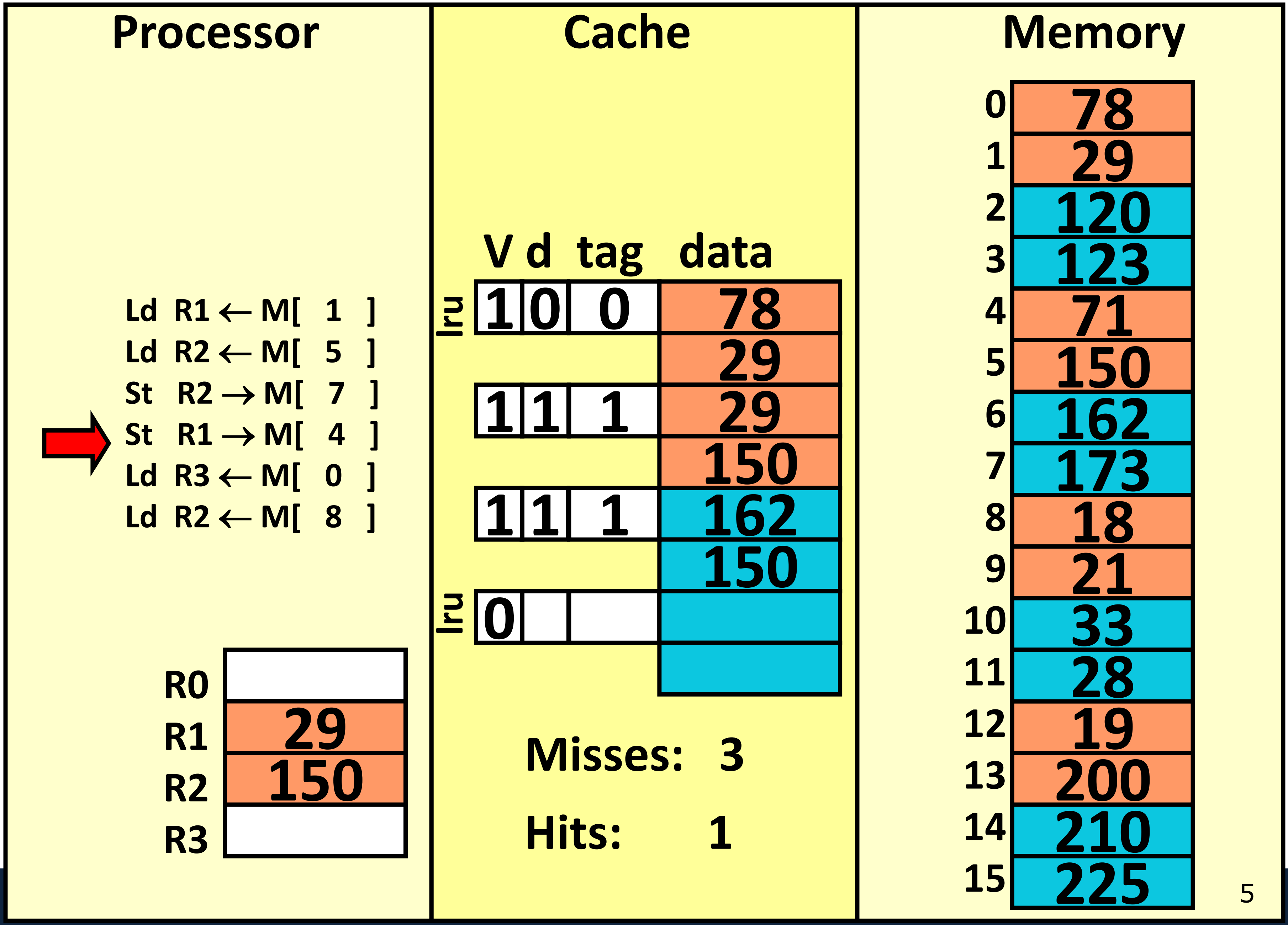
Misses: 3

Hits: 0

## Memory

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 71 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Set-associative cache (REF 4)



## Processor

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
    St  R2 → M[  7  ]
➡   St  R1 → M[  4  ]
    Ld  R3 ← M[  0  ]
    Ld  R2 ← M[  8  ]
```

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 150 |
| R3 | |

## Cache

| | V | d | tag | data |
|---|---|---|---|---|
| lru | 1 | 0 | 0 | 78 |
| | | | | 29 |
| | 1 | 1 | 1 | 29 |
| | | | | 150 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

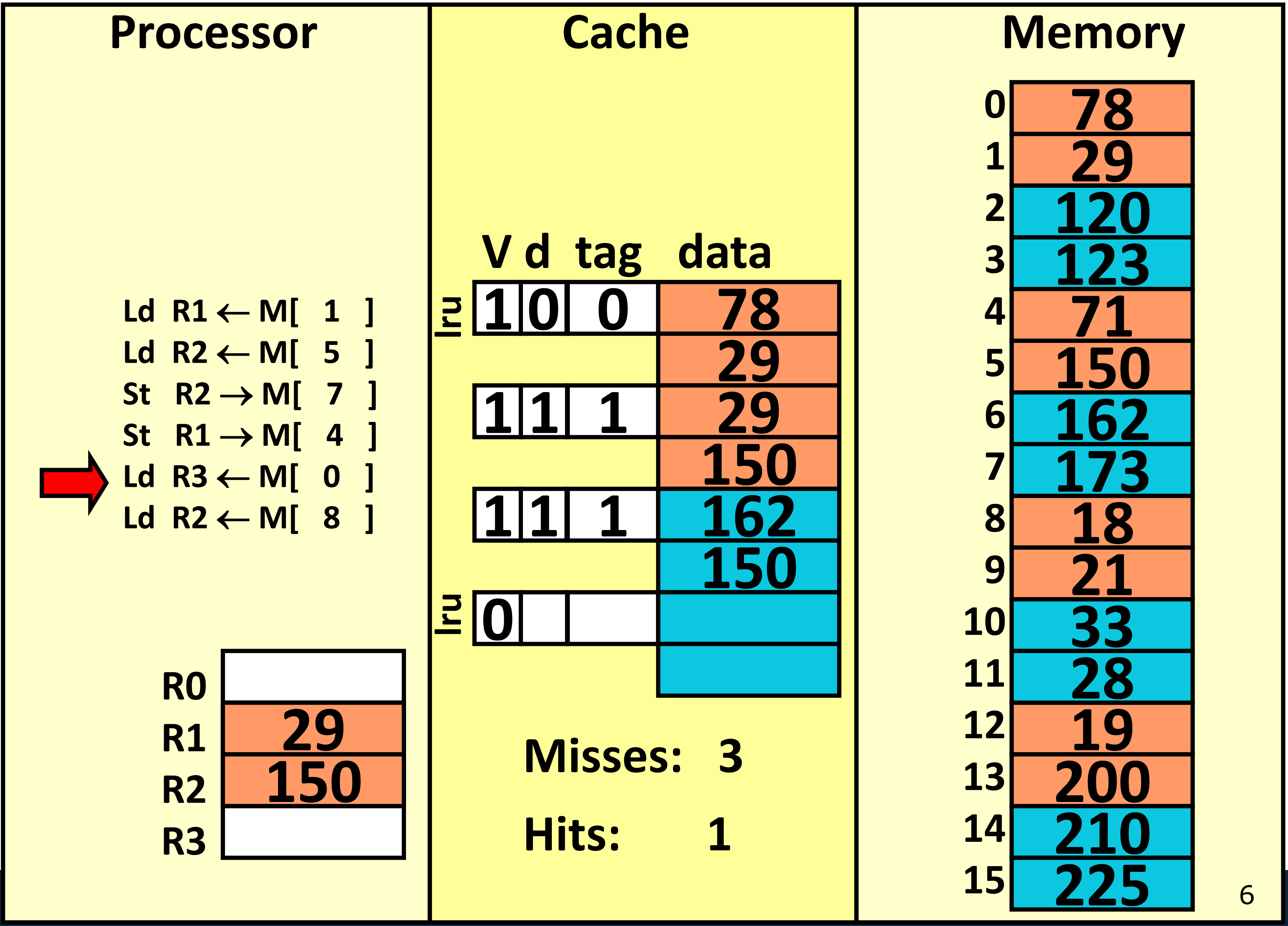
Misses: 3

Hits: 1

## Memory

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 71 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Set-associative cache (REF 5)



**Processor**

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
    St  R2 → M[  7  ]
    St  R1 → M[  4  ]
➡   Ld  R3 ← M[  0  ]
    Ld  R2 ← M[  8  ]
```

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 150 |
| R3 | |

**Cache**

| | V | d | tag | data |
|---|---|---|---|---|
| lru | 1 | 0 | 0 | 78 |
| | | | | 29 |
| | 1 | 1 | 1 | 29 |
| | | | | 150 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

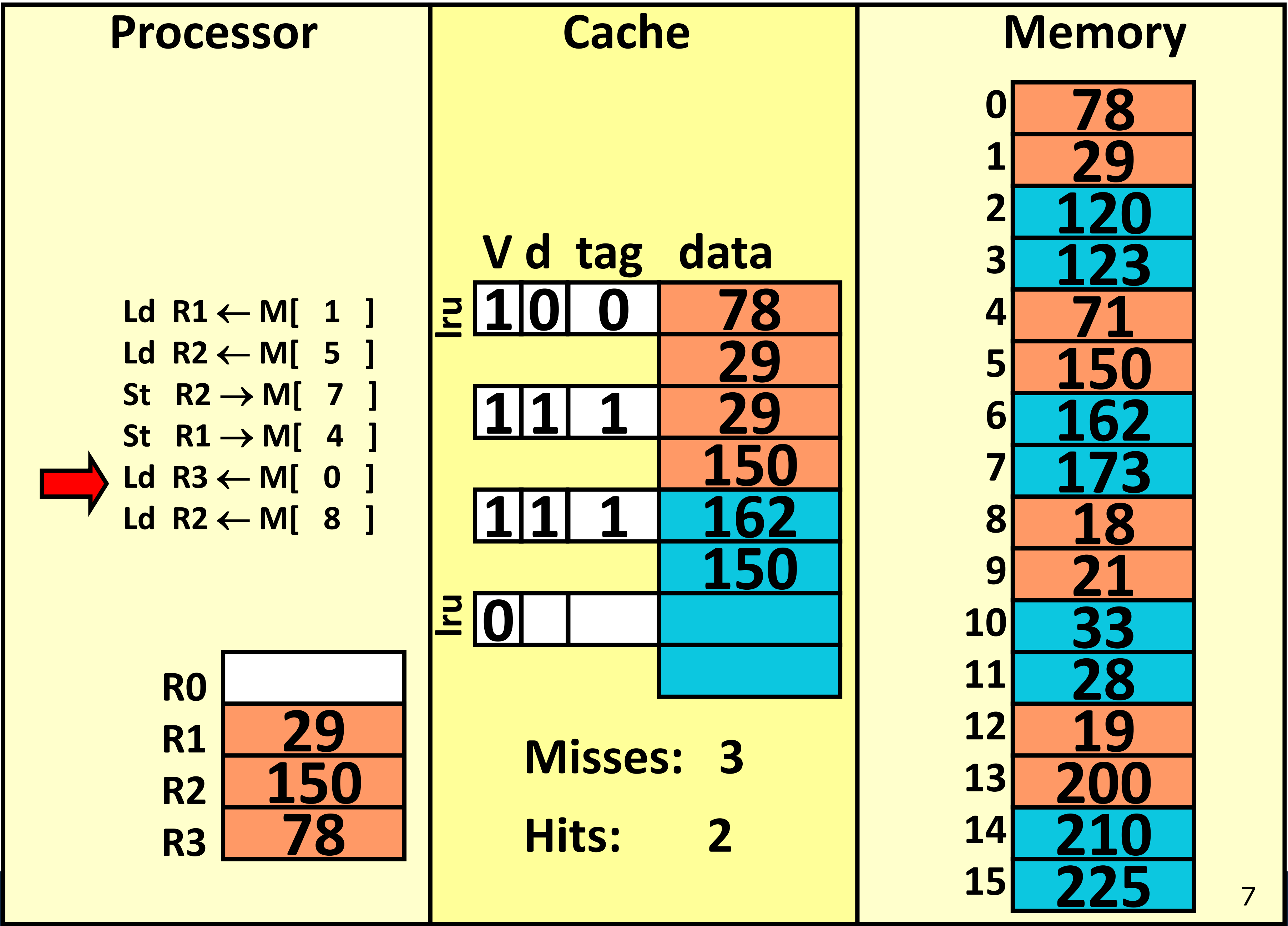
Misses: 3

Hits: 1

**Memory**

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 71 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Set-associative cache (REF 5)



Processor

Ld R1 ← M[ 1 ]
Ld R2 ← M[ 5 ]
St R2 → M[ 7 ]
St R1 → M[ 4 ]
➡ Ld R3 ← M[ 0 ]
Ld R2 ← M[ 8 ]

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 150 |
| R3 | 78 |

Cache

| | V | d | tag | data |
|---|---|---|---|---|
| lru | 1 | 0 | 0 | 78 |
| | | | | 29 |
| | 1 | 1 | 1 | 29 |
| | | | | 150 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

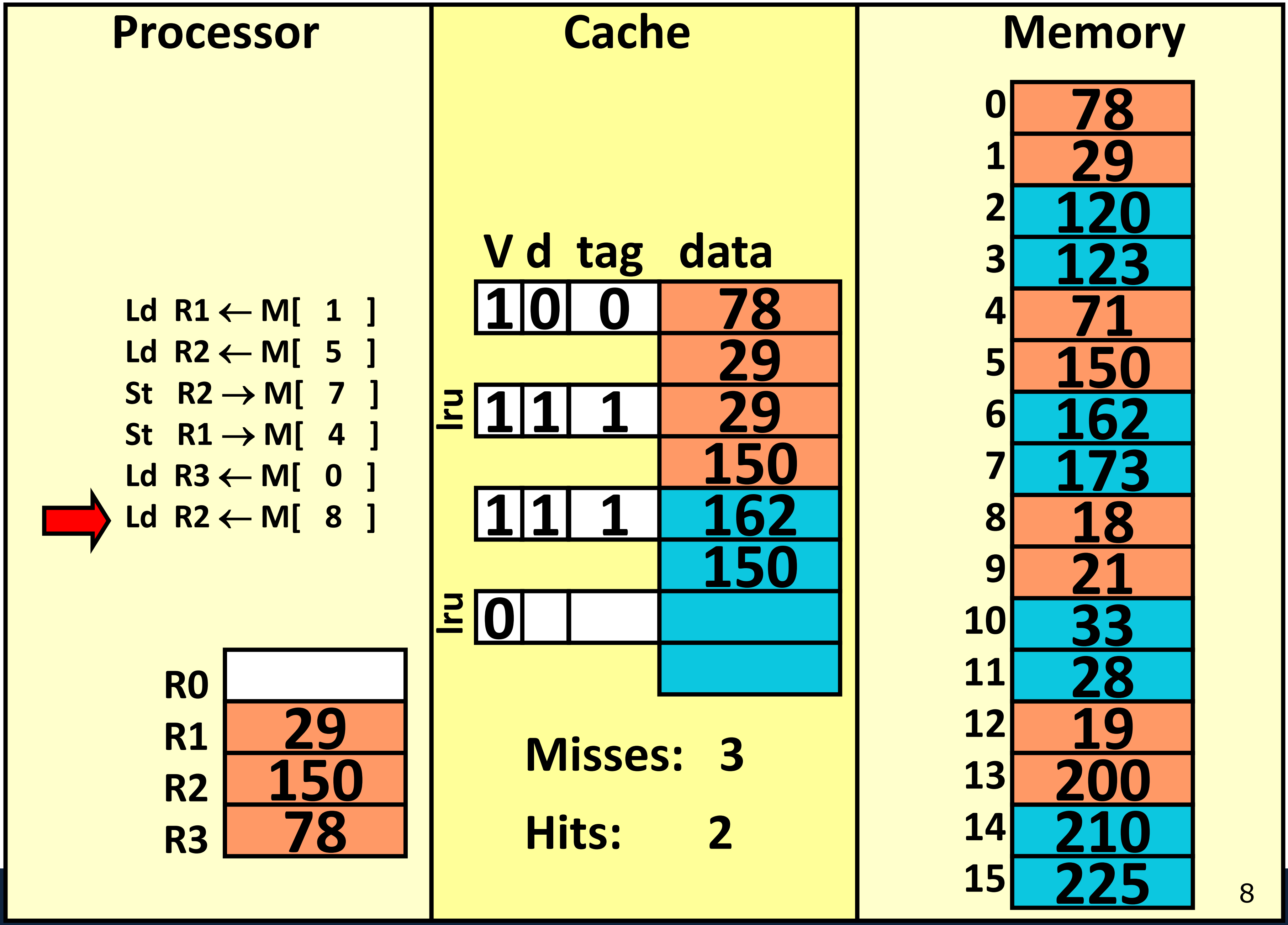
Misses: 3

Hits: 2

Memory

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 71 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Set-associative cache (REF 6)



Processor

```
  Ld  R1 ← M[  1  ]
  Ld  R2 ← M[  5  ]
  St  R2 → M[  7  ]
  St  R1 → M[  4  ]
  Ld  R3 ← M[  0  ]
➡ Ld  R2 ← M[  8  ]
```

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 150 |
| R3 | 78 |

Cache

| | V | d | tag | data |
|---|---|---|---|---|
| | 1 | 0 | 0 | 78 |
| | | | | 29 |
| lru | 1 | 1 | 1 | 29 |
| | | | | 150 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

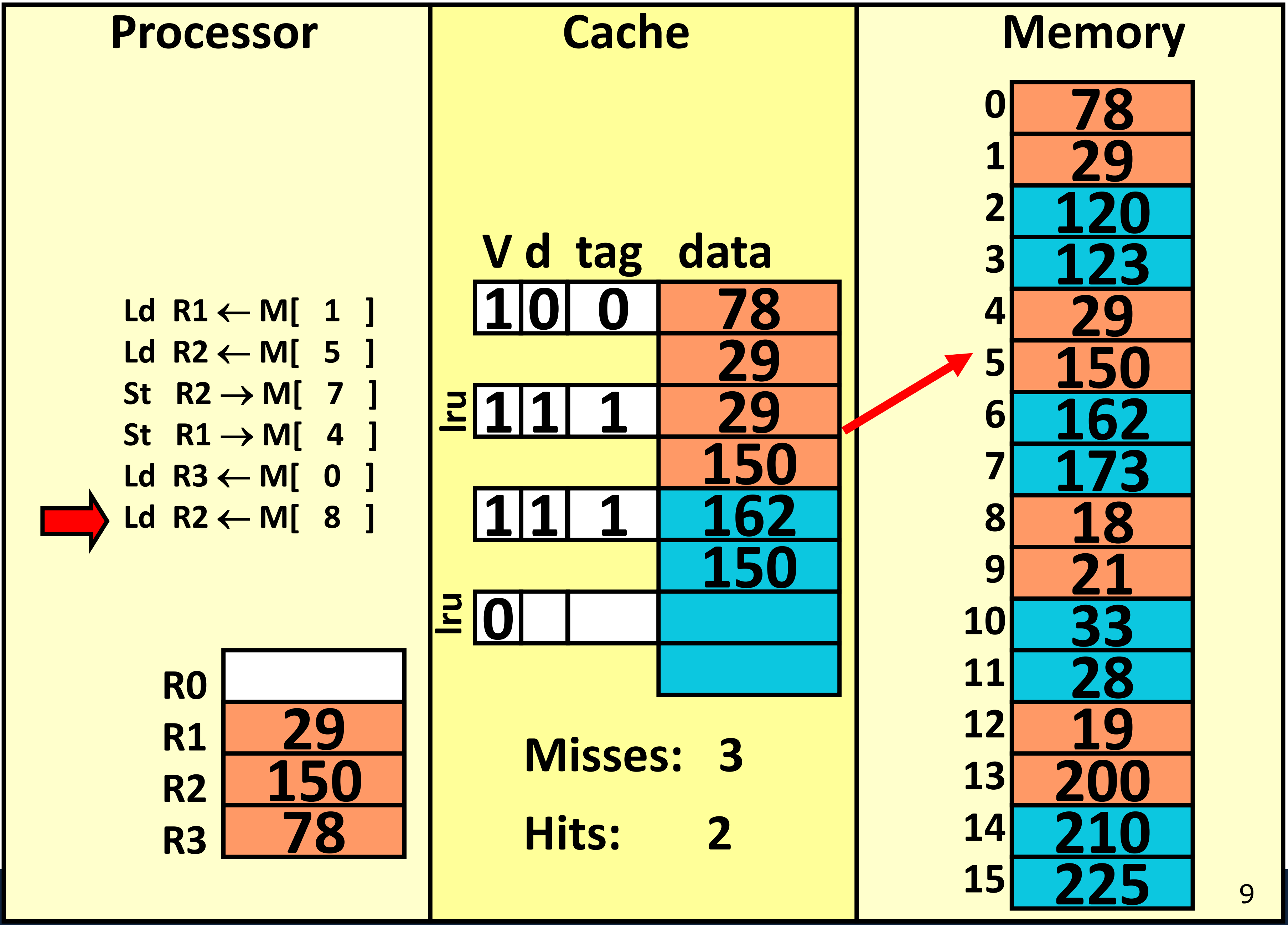
Misses: 3

Hits: 2

Memory

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 71 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Set-associative cache (REF 6)



## Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
St  R2 → M[  7  ]
St  R1 → M[  4  ]
Ld  R3 ← M[  0  ]
Ld  R2 ← M[  8  ]
```

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 150 |
| R3 | 78 |

## Cache

| | V | d | tag | data |
|---|---|---|---|---|
| | 1 | 0 | 0 | 78 |
| | | | | 29 |
| lru | 1 | 1 | 1 | 29 |
| | | | | 150 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

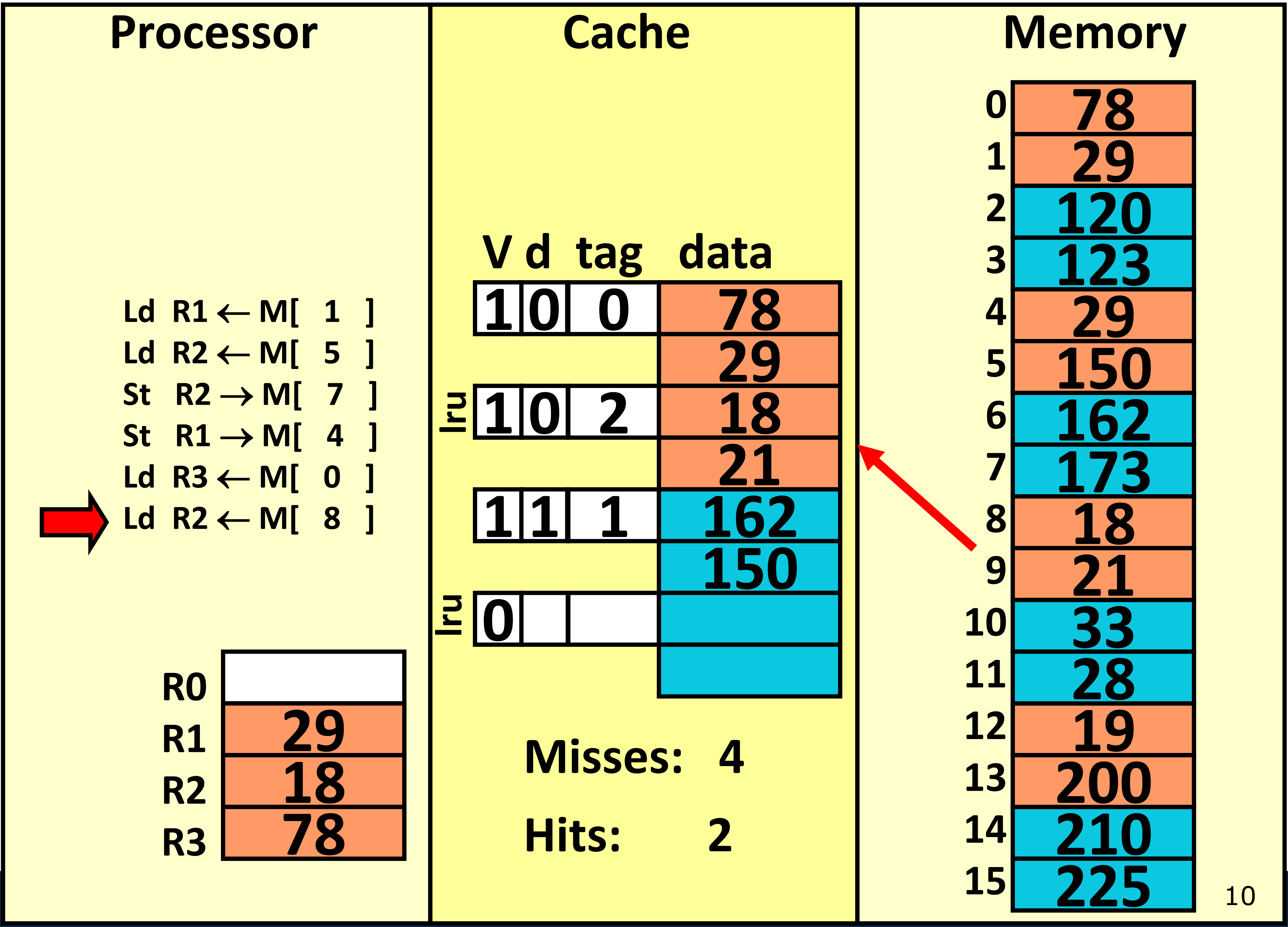
Misses: 3

Hits: 2

## Memory

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 29 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Set-associative cache (REF 6)



Ld R1 ← M[ 1 ]
Ld R2 ← M[ 5 ]
St R2 → M[ 7 ]
St R1 → M[ 4 ]
Ld R3 ← M[ 0 ]
Ld R2 ← M[ 8 ]

| | |
|---|---|
| R0 | |
| R1 | 29 |
| R2 | 18 |
| R3 | 78 |

| | V | d | tag | data |
|---|---|---|---|---|
| | 1 | 0 | 0 | 78 |
| | | | | 29 |
| lru | 1 | 0 | 2 | 18 |
| | | | | 21 |
| | 1 | 1 | 1 | 162 |
| | | | | 150 |
| lru | 0 | | | |
| | | | | |

Misses: 4

Hits: 2

| | |
|---|---|
| 0 | 78 |
| 1 | 29 |
| 2 | 120 |
| 3 | 123 |
| 4 | 29 |
| 5 | 150 |
| 6 | 162 |
| 7 | 173 |
| 8 | 18 |
| 9 | 21 |
| 10 | 33 |
| 11 | 28 |
| 12 | 19 |
| 13 | 200 |
| 14 | 210 |
| 15 | 225 |

# Agenda

- Set-associativity overview
- Example
- **Class problem**
- Integrating caches into our processor

# Class Problem 1

- For a 32-bit address and 16KB cache with 64-byte blocks, show the breakdown of the address for the following cache configuration:

**A) fully associative cache**

**B) 4-way set associative cache**

**C) Direct-mapped cache**

# Class Problem 1

- For a 32-bit address and 16KB cache with 64-byte blocks, show the breakdown of the address for the following cache configuration:

**A) fully associative cache**

**Block Offset = log2(64)=6 bits**
**Tag = 32 - 6 = 26 bits**

**C) Direct-mapped cache**

**Block Offset = 6 bits**
**#lines = 256 Line Index = 8 bits**
**Tag = 32 – 6 – 8 = 18 bits**

**B) 4-way set associative cache**

**Block Offset = 6 bits**
**#sets = #lines / ways = 64**
**Set Index = 6 bits**
**Tag = 32 - 6 - 6 = 20 bits**

# Agenda

- Set-associativity overview
- Example
- Class problem
- **Integrating caches into our processor**

# Multi-Level Caches

- We've been considering proc -> cache -> memory

- This works well if working data set is <= size of cache

- But if data set is a little larger that cache, performance can plummet

# Multi-Level Caches

- This is the motivation of multiple levels of caches
- L1 – smallest, fastest, closest to processor
- LN – biggest, slowest, closest to memory
- Allows for gradual performance degradation as data set size increases
- 3 levels of cache is pretty common in today's systems

# What about cache for instructions

- We've been focusing on caching loads and stores (i.e. data)
- Instructions should be cached as well
- We have two choices:
  1. Treat instruction fetches as normal data and allocate cache lines when fetched
  2. Create a second cache (called the instruction cache or ICache) which caches instructions only
     - More common in practice

How do you know which cache to use?

What are advantages of a separate ICache?

# Integrating Caches into Pipeline

- How are caches integrated into a pipelined implementation?
  - Replace instruction memory with Icache
  - Replace data memory with Dcache

- Issues:
  - Memory accesses now have variable latency
  - Both caches may miss at the same time

# Agenda

- **Motivation**
- Example
- How to optimize cache design
- Practice Problem 1
- Practice Problem 2
- Practice Problem 3
- Practice Problem 4

# Improving our Caches

- If our cache is getting a lot of misses, how do we improve it?
  - Depends on why the misses occurring
  - Is the cache too small? Is the associativity too restrictive? Something else?
- A decent first step is to **classify** the types of missing we are observing

# Classifying Cache Misses

- Cache misses happen for 3* reasons
  - The 3C's of Cache misses:
- Compulsory miss
  - We've never accessed this data before
- Capacity miss
  - Cache is not large enough to hold all the data
  - May have been avoided if we used a bigger cache
- Conflict miss
  - Cache is large enough to hold data, but was replaced due to overly restrictive associativity
  - May have been avoided if we used a higher-associative cache

**Note:** **On multi-core systems, there's a 4th C – take EECS 470/570 to learn more*

# Classifying Cache Misses

- Scenario: run given program on system with N-way cache of size M
    - Identify each miss
- We can classify each miss in a program by simulating on 3 different caches
    - If miss still occurs in cache where size >= memory size: compulsory miss
    - Else, if miss occurs in fully associative cache of size M: capacity miss
    - Else, if miss occurs in N-way cache of size M (original cache): conflict miss

# Agenda

- Motivation
- **Example**
- How to optimize cache design
- Practice Problem 1
- Practice Problem 2
- Practice Problem 3
- Practice Problem 4

# 3C's Sample Problem

Consider a cache with the following configuration: write-allocate, total size is 64 bytes, block size is 16 bytes, and 2-way associative. The memory address size is 16 bits and byte-addressable. The replacement policy is LRU. The cache is empty at the start.

For the following memory accesses, indicate whether the reference is a hit or miss, and the type of a miss (compulsory, conflict, capacity)

# 3 C's Practice Problem – 3 C's

64 bytes total, 16 byte blocks, 2-way, 2 sets

| Address | Infinite | FA | SA | 3Cs |
|---|---|---|---|---|
| 0x00 | | | | |
| 0x14 | | | | |
| 0x27 | | | | |
| 0x08 | | | | |
| 0x38 | | | | |
| 0x4A | | | | |
| 0x18 | | | | |
| 0x27 | | | | |
| 0x0F | | | | |
| 0x40 | | | | |

# 3 C's Practice Problem – 3 C's

**Poll:** How many blocks will be in a 64 byte FA cache?

| Address | Infinite | FA | SA | 3Cs |
|---|---|---|---|---|
| 0x00 | M | | | |
| 0x14 | M | | | |
| 0x27 | M | | | |
| 0x08 | H | | | |
| 0x38 | M | | | |
| 0x4A | M | | | |
| 0x18 | H | | | |
| 0x27 | H | | | |
| 0x0F | H | | | |
| 0x40 | H | | | |

# 3 C’s Practice Problem – 3 C’s

64 bytes total, 16 byte blocks, 2-way, 2 sets

| Address | Infinite | FA | SA | 3Cs |
|---|---|---|---|---|
| 0x00 | M | M | | |
| 0x14 | M | M | | |
| 0x27 | M | M | | |
| 0x08 | H | H | | |
| 0x38 | M | M | | |
| 0x4A | M | M | | |
| 0x18 | H | M | | |
| 0x27 | H | M | | |
| 0x0F | H | M | | |
| 0x40 | H | H | | |

# 3 C's Practice Problem – 3 C's

64 bytes total, 16 byte blocks, 2-way, 2 sets

| Address | Infinite | FA | SA | 3Cs |
|---|---|---|---|---|
| 0x00 | M | M | M | Compulsory |
| 0x14 | M | M | M | Compulsory |
| 0x27 | M | M | M | Compulsory |
| 0x08 | H | H | H | --- |
| 0x38 | M | M | M | Compulsory |
| 0x4A | M | M | M | Compulsory |
| 0x18 | H | M | H | --- |
| 0x27 | H | M | M | Capacity |
| 0x0F | H | M | M | Capacity |
| 0x40 | H | H | M | Conflict |

# Agenda

- Motivation
- Example
- **How to optimize cache design**
- Practice Problem 1
- Practice Problem 2
- Practice Problem 3
- Practice Problem 4

# How to reduce cache misses

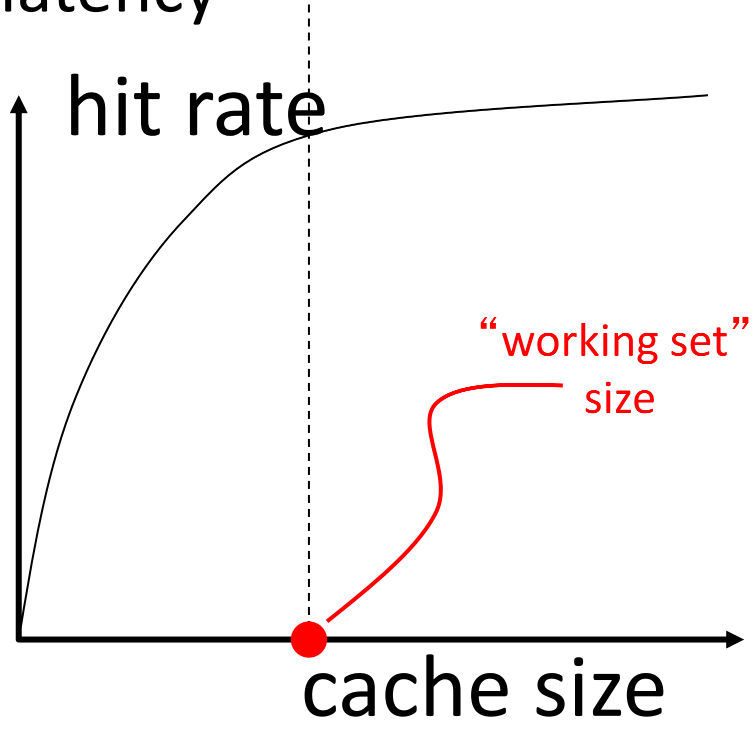
- Compulsory miss
  - Reduce by **increasing cache block size**
    - Reduces total number of blocks for given cache size ☹
  - Or by using prefetching (guess we'll need data based on previous memory patterns - discussed more in EECS 470)
- Capacity miss
  - Reduce by **building a bigger cache**
    - Increase access latency ☹
- Conflict miss
  - Reduce by **increasing associativity**
    - Increase access latency / overheads ☹

# Cache Performance

- How does changing these parameters affect performance?
  - Cache size
  - Block size
  - Associativity

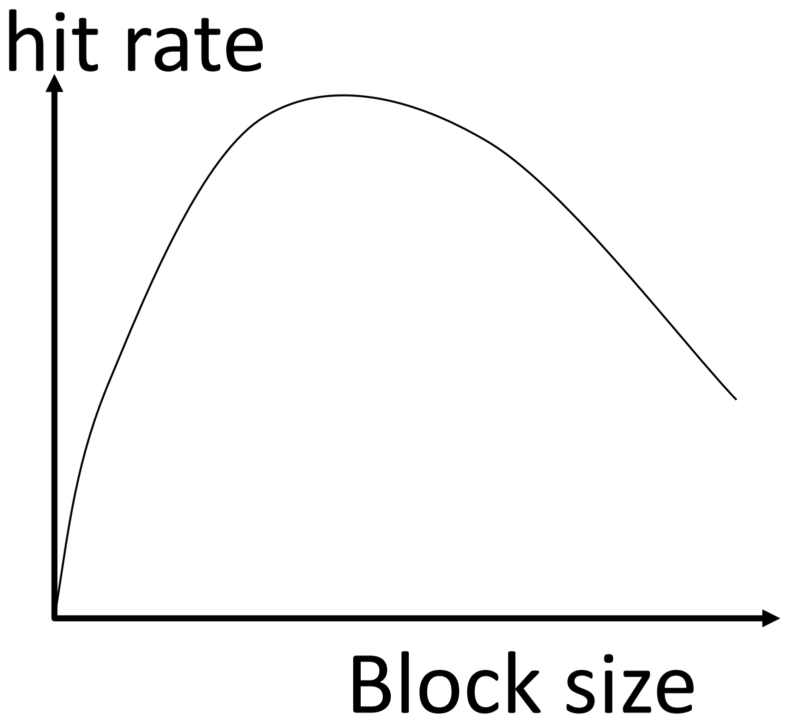
# Cache Size

- Cache size in the total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit & miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data executing application references
  - **Within a time interval**



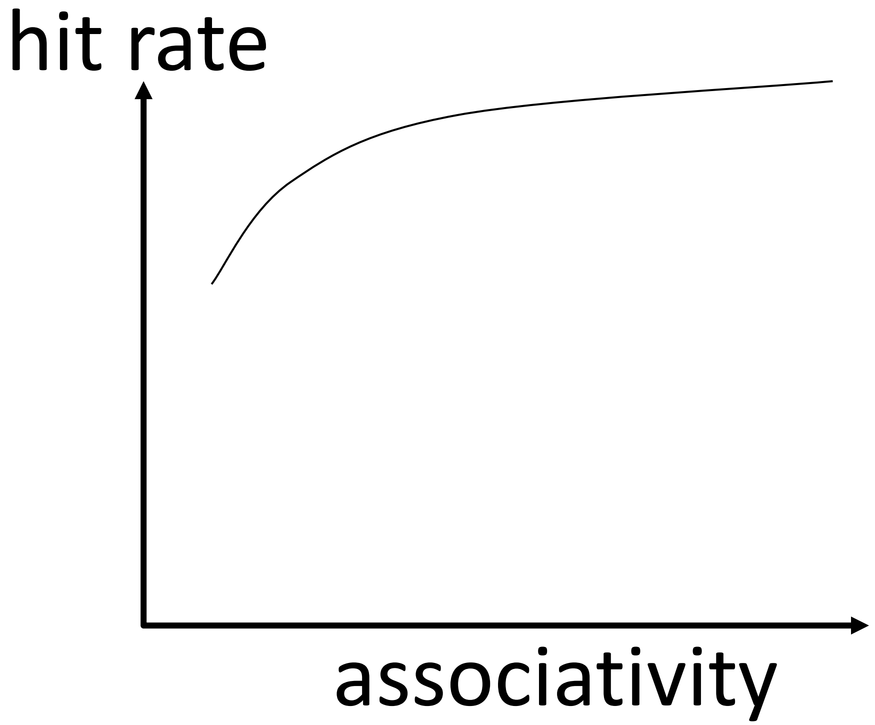
# Block size

- Block size is the data that is associated with an address tag
  - Sub-blocking: A block divided into multiple pieces (each with V bit)
    - Can improve “write” performance
    - Take 470 to learn more
- Too small blocks
  - don’t exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks
    - likely-useless data transferred
    - Extra bandwidth/energy consumed



# Associativity

- How many blocks can map to the same index (or set)?
- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns
- Smaller associativity
  - lower cost
  - faster hit time
    - Especially important for L1 caches



# Agenda

- Motivation
- Example
- How to optimize cache design
- **Practice Problem 1**
- Practice Problem 2
- Practice Problem 3
- Practice Problem 4

**Poll:** What is the CPI increase over base (1) due to hazards?

# Practice Problem 1: CPI with caches

The *blaster* application run on the LC2k with full data forwarding and all branches predicted not-taken has the following instruction frequencies:

45% R-type   20% Branches   15% Loads   20% Stores

In *blaster*, 40% of branches are taken and 50% of LWs are followed by an immediate use.

The I-cache has a miss rate of 3% and the D-cache has a miss rate of 6% (no overlapping of misses). On a miss, the main memory is accessed and has a latency of 100 ns. The clock frequency  is 500 MHz.

# Problem 1 Solution

The *blaster* application run on the LC2k with full data forwarding and all branches predicted not-taken has the following instruction frequencies:

45% R-type   20% Branches   15% Loads   20% Stores

In *blaster*, 40% of branches are taken and 50% of LWs are followed by an immediate use.

The I-cache has a miss rate of 3% and the D-cache has a miss rate of 6% (no overlapping of misses). On a miss, the main memory is accessed and has a latency of 100 ns. The clock frequency is 500 MHz.

What is the CPI of *blaster* on the LC2k?

Stalls per cache miss = 100 ns / 2ns = 50 cycles (500 Mhz ➜ 2ns cycle time)

CPI = 1 + data hazard stalls + control hazard stalls + icache stalls + dcache stalls

CPI = 1 + 0.15*0.50*1 + 0.20*0.40*3 + 1*0.03*50 + 0.35*0.06*50

# Agenda

- Motivation
- Example
- How to optimize cache design
- Practice Problem 1
- **Practice Problem 2**
- Practice Problem 3
- Practice Problem 4

# Practice Problem 2: Memory Usage

**Poll:** How many bytes are read/written if we have no cache?

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- How many bytes of memory would be read and written if:
  - We had no cache?
  - We had a write-though cache with a no-write allocate policy?
  - We had a write-back cache with a write-allocate policy? (Assume 25% of all misses result in a dirty eviction)

# Practice Problem 2: Memory Usage

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- Let's start with the no-cache case.
  - All stores go to memory and are 4 bytes each
    - Writes: 1 billion stores* 4 bytes = 4 billion bytes
  - All loads go to memory and are 4 bytes each.
    - Reads: 2 billion loads* 4 bytes = 8 billion bytes
- Write-through with no write-allocate?

# Practice Problem 2: Memory Usage

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- Write-though, no allocate.
  - All stores still go to memory and are still 4 bytes each.
    - Writes: 1 billion stores* 4 bytes = 4 billion bytes
  - Only loads that miss in the cache go to memory. But they read the full cache block.
    - Reads: 2 billion loads* 0.05* 32 bytes = 3.2 billion bytes

# Practice Problem 2: Memory Usage

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- Write-though, no allocate.

# Practice Problem 2: Memory Usage

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- Write-back, write-allocate (25% of all misses result in a dirty eviction)

# Practice Problem 2: Memory Usage

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- Write-back, write-allocate (25% of all misses result in a dirty eviction)
  - *Store* misses result in a cache block being read.
    - Reads: 1 billion stores* 0.05* 32 bytes = 1.6 billion bytes
  - *Load* misses result in a cache block being read.
    - Reads: 2 billion loads* 0.05* 32 bytes = 3.2 billion bytes
  - So that is 4.8 billion bytes of data read.

# Practice Problem 2: Memory Usage

- Say you have the following:
  - A program that generates 2 Billion loads and 1 Billion stores, each 4 bytes in size.
  - A cache with a 32-byte block which gets a 95% hit rate on that program.

- Write-back, write-allocate (25% of all misses result in a dirty eviction)
  - *Store* misses result in dirty eviction 1/4 of the time.
    - Reads: 1 billion stores* 0.05* 32 bytes*(.25) = 0.4 billion bytes
  - *Load* misses result in a cache block being read.
    - Reads: 2 billion loads* 0.05* 32 bytes*(.25) = 0.8 billion bytes

# Agenda

- Motivation
- Example
- How to optimize cache design
- Practice Problem 1
- Practice Problem 2
- **Practice Problem 3**
- Practice Problem 4

# Practice Problem 3: CPI w/ Caches 2

- Given a 200 MHz processor with 8KB instruction and data caches and a with memory access latency of 20 cycles. Both caches are 2-way associative. A program running on this processor has a 95% icache hit rate and a 90% dcache hit rate. On average, 30% of the instructions are loads or stores. The CPI of this system, if caches were ideal would be 1.
- Suppose you have 2 options for the next generation processor, which do you pick?
  - Option 1: Double the clock frequency—assume this will increase your memory latency to 40 cycles. Also assume a base CPI of 1 can still be achieved after this change.
  - Option 2: Double the size of your caches, this will increase the instruction cache hit rate to 98% and the data cache hit rate to 95%. Assume the hit latency is still 1 cycle.

# Practice Problem 3: Solution

Option 1: (double clock freq, base cycle time is 5 ns, so new cycle time is 2.5 ns)

CPI = baseCPI + IcacheStallCPI + DcacheStallCPI

CPI = 1.0 + 0.05*40 + 0.3*0.1*40 = 4.2

Execution time = 4.2 * Ninstrs * 2.5ns = 10.5ns * Ninstrs

Option 2 (icache/dcache miss rates lowered to 2% and 5%)

CPI = baseCPI + IcacheStallCPI + DcacheStallCPI

CPI = 1.0 + 0.02*20 + 0.3*0.05*20 = 1.7

Execution time = 1.7 * Ninstrs * 5ns = 8.5ns * Ninstrs

Therefore, Option 2 is the better choice

# Agenda

- Motivation
- Example
- How to optimize cache design
- Practice Problem 1
- Practice Problem 2
- Practice Problem 3
- **Practice Problem 4**

# Practice Problem 4: Guess that cache!

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

```
0x310 – Miss
0x30f – Miss
0x510 – Miss
0x31f – Hit
0x72d – Miss
0x72f – Hit
0x320 – Miss
0x520 – Miss
0x720 - Miss
```

**Block size: ?**

**Associativity: ?**

**Number of sets: ?**

# Practice Problem 4: Guess that cache!

Similar to homework! Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

```
0x310 – Miss
0x30f – Miss
0x510 – Miss
0x31f – Hit
0x72d – Miss
0x72f – Hit
0x320 – Miss
0x520 – Miss
0x720 - Miss
```

# Practice Problem 4: Guess that cache!

Similar to homework! Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

```
0x310 – Miss
0x30f – Miss
0x510 – Miss
0x31f – Hit
0x72d – Miss
0x72f – Hit
0x320 – Miss
0x520 – Miss
0x720 - Miss
```

# Practice Problem 4: Guess that cache!

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 – Miss

0x30f – Miss

0x510 – Miss

0x31f – Hit

0x72d – Miss

0x72f – Hit

0x320 – Miss

0x520 – Miss

0x720 - Miss

<u>Determine block size</u>

First hit must be brought in by another miss

Take closest address: 0x310, so know block size must be at least 16 bytes so 0x31f brought in when 0x310 miss occurs

Now, is the block size larger? Know that 0x30f was a miss, thus 0x310 and 0x30f not in the same block. Thus, block size must be <= 16 bytes

Thus Block Size = 16 bytes

# Practice Problem 4: Guess that cache!

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 – Miss 🟩

0x30f – Miss

0x510 – Miss 🟩

0x31f – Hit 🟩

0x72d – Miss

0x72f – Hit 🟨

0x320 – Miss 🟧

0x520 – Miss 🟧

0x720 - Miss 🟨

Determine associativity

Assume direct mapped: 3-bit tag, 5-bit index, 4-bit offset.
If DM, 0x310 and 0x510 would both map to index 17,
Thus 0x31f could not be a hit. So, not direct mapped.

Assume 2-way associative: 4-bit tag, 4-bit index, 4-bit offset
This fixes the green accesses, and allows 0x31f to be a hit.

What about > 2-way associative?
Now we also know that 0x720 is a miss even though 3 accesses earlier 0x72f was a hit, and thus it is in the cache. The intervening 2 accesses must kick it out, 0x320 and 0x520. Both go to set 2. If the associativity was > 2, then 0x720 would be a hit. So, must conclude that cache is 2-way associative.

Lastly, number of sets = 512 / (2 * 16) = 16

# Next time

- Completing the hierarchy: virtual memory