My CPU when the L1 cache misses

EECS 370

Improving Caches

Announcements

- My office hours on Wed 11/8 are cancelled
- Lab 9 meets Fr/M
- P3 Checkpoint due tonight
  - Full P3 due Thu 11/9
- Homework 3 due Mon 11/6

Agenda

- **Larger Cache Blocks**
- Extra Problems
- LRU with More than Two Blocks
- Write-Through Cache
- Write-Back Cache

Calculating Size

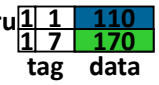
- How many bits is used in cache?
  - Storing data
    - 2 bytes of SRAM
  - Calculate overhead (non-data)
    - This cost is often forgotten for caches, but it drives up the cost of real designs!
    - 2 4-bit tags
    - 2 valid bits
- What is the storage requirement

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

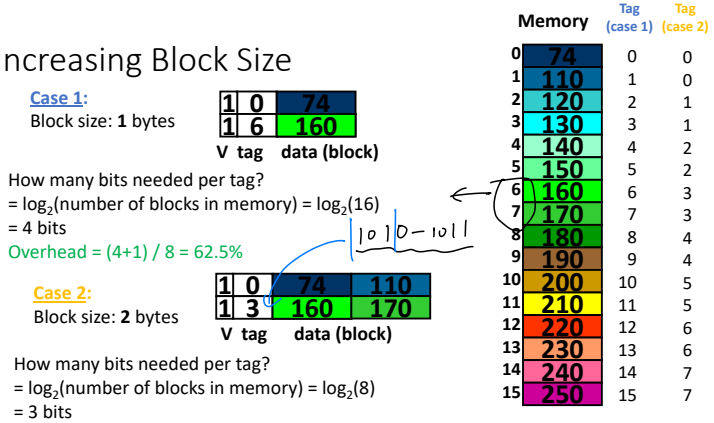
a) Increase number of cache entries
b) Decrease number of cache entries
c) Use smaller addresses
d) Store more data in each cache entry

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%

1010-1011

Case 2:
Block size: **2** bytes

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

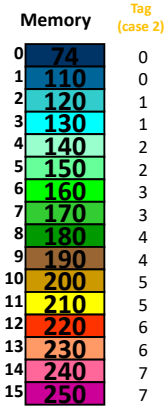
How many bits needed per tag?
= $\log_2$(number of blocks in memory) = $\log_2(8)$
= 3 bits
Overhead = (3+1) / 16 = 25%

Figuring out the tag

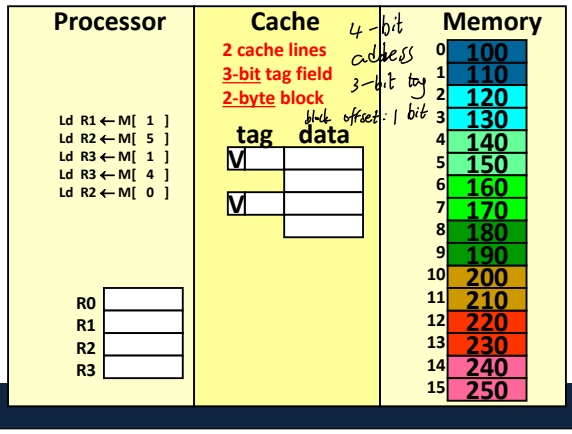
- If block size is N, what's the pattern for figuring out the tag from the address?
  - $tag = \left\lfloor \frac{addr}{block\ size} \right\rfloor$
- If block size is power of 2, then this is just everything except the $\log_2(block\ size)$ bits of the address in binary!
- E.g.

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

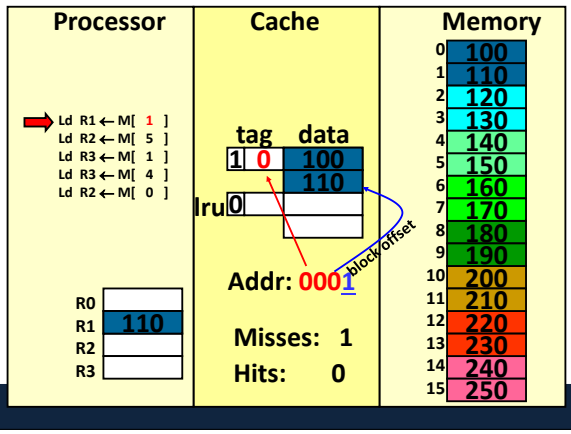
- Remaining bits (block offset) tells us how far into the block the data is



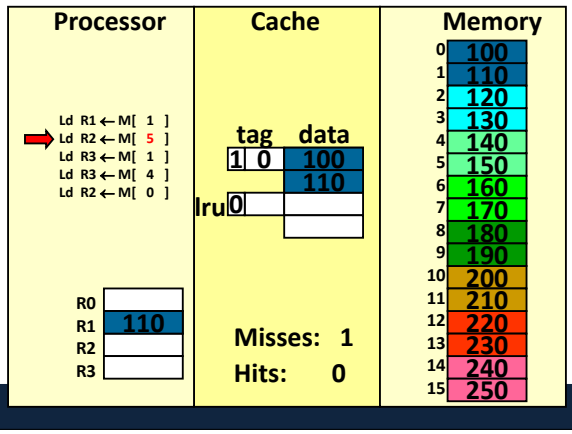
Block size for caches



Block size for caches

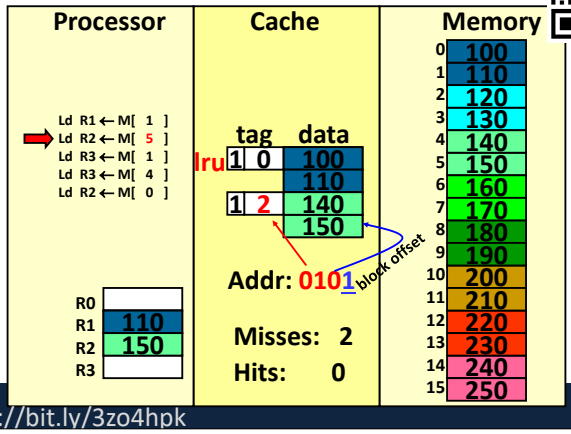


Block size for caches



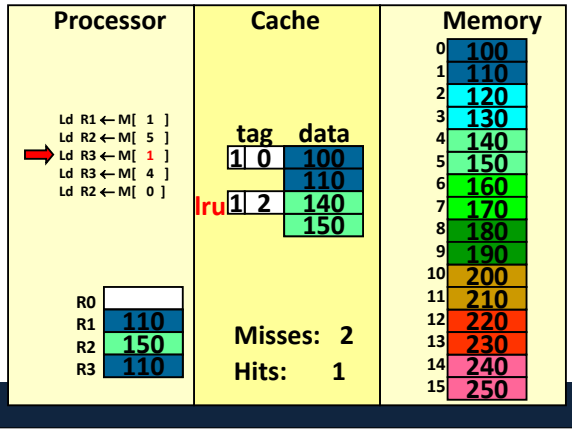
Block size for caches

Poll: Complete the last 3 instructions yourself

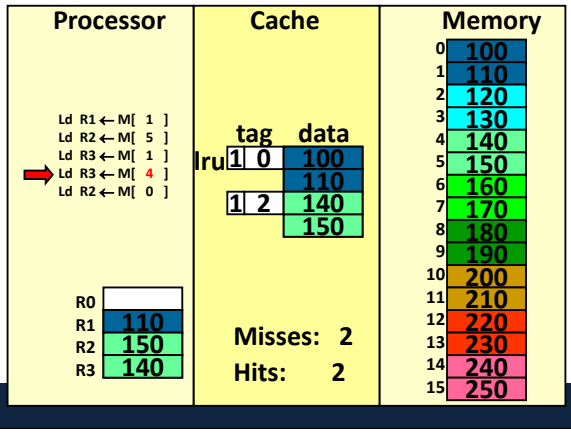


https://bit.ly/3zo4hpk

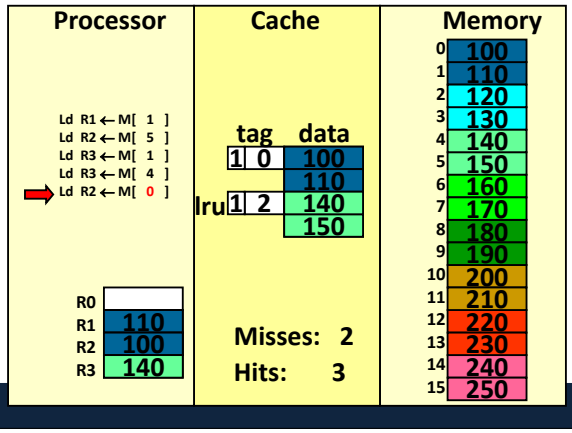
Block size for caches



Block size for caches



Block size for caches



Spatial Locality

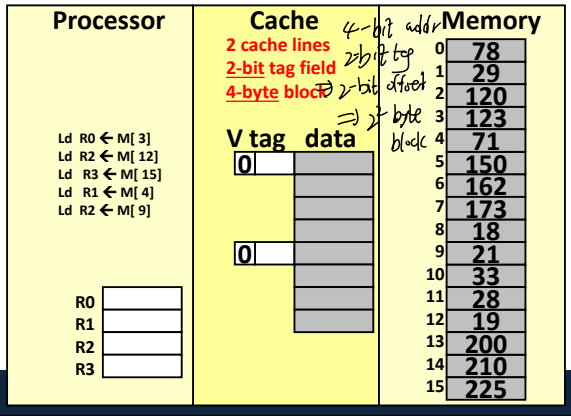
- Notice that when we accessed address 1, we also brought in address 0
  - This turned out to be a good thing, since we later referenced address 0 and found it in the cache
- This is taking advantage of **spatial locality:**
  - If we access a memory location (e.g. 1000), we are more likely to access a location near it (e.g. 1001) than some random location
  - Arrays and structs are a big reason for this

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

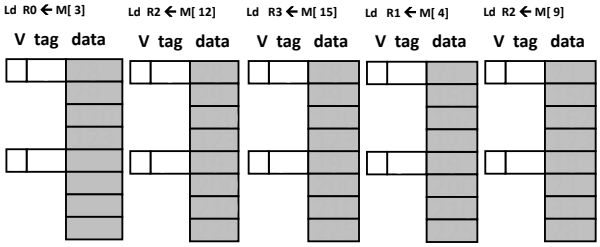

Agenda

- Larger Cache Blocks
- **Extra Problems**
- LRU with More than Two Blocks
- Write-Through Cache
- Write-Back Cache

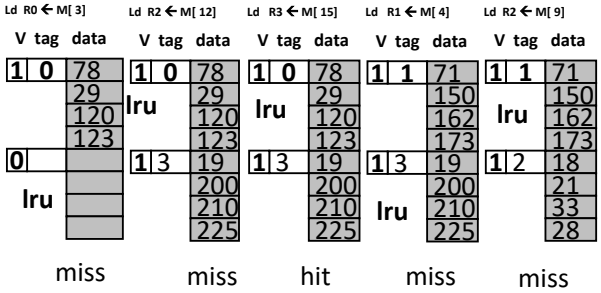
Extra Practice Problem



Solution to Practice Problem



Solution to Practice Problem



Agenda

- Larger Cache Blocks
- Extra Problems
- **LRU with More than Two Blocks**
- Write-Through Cache
- Write-Back Cache

LRU with more than 2 entries

- If we have more than 2 things we're keeping track of…
  - Can't **just** track LRU
  - Once we access that element, how do we know which of the other elements are LRU?
  - Must track the *full ordering* of when elements were accessed*
- Each element must store a number [0-(N-1)] -> $\log_2(N)$ bits
- 0 is LRU, 1 is 2nd LRU… N-1 is most recently used

LRU with more than 2 entries

- If we have more than 2 things we're keeping track of…
  - Can't **just** track LRU
  - Once we access that element, how do we know which of the other elements are LRU?
  - Must track the *full ordering* of when elements were accessed*
- Each element must store a number [0-(N-1)] -> $\log_2(N)$ bits
- 0 is LRU, 1 is 2nd LRU… N-1 is most recently used
- When element i is used:

```
X = counter[i]
counter[i] = N-1
for (j=0  to N-1)
    if ((j != i) AND (counter[j]>X)) counter[j]—
```

- Evict element with counter = 0 when needed
- Get's expensive for moderate to large N

Initial State				
Element	0	1	2	3
Count	0	1	2	3

Access Element 2				
Element	0	1	2	3
Count	0	1	3	2

Access Element 0				
Element	0	1	2	3
Count	3	0	2	1

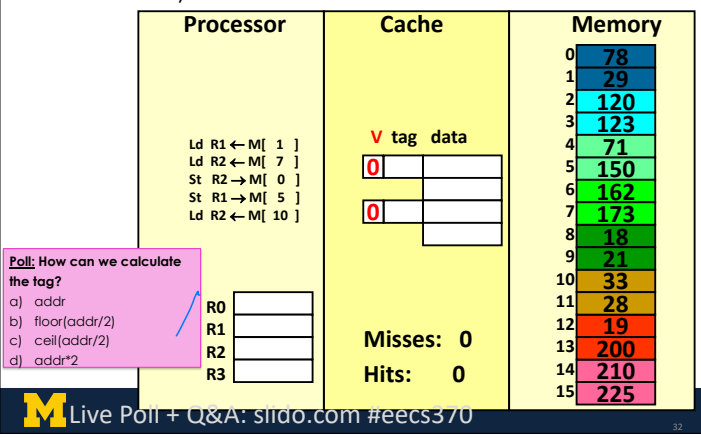
Agenda

- Larger Cache Blocks
- Extra Problems
- LRU with More than Two Blocks
- **Write-Through Cache**
- Write-Back Cache

What about stores?

- Where should you write the result of a store?
  - If that memory location is in the cache:
    - Send it to the cache.
    - Should we also send it to memory? (write-through policy)
  - If it is not in the cache:
    - Allocate the line (put it in the cache)? (allocate-on-write policy)
    - Write it directly to memory without allocation? (no allocate-on-write policy)

Handling stores (write-through, allocate on write)

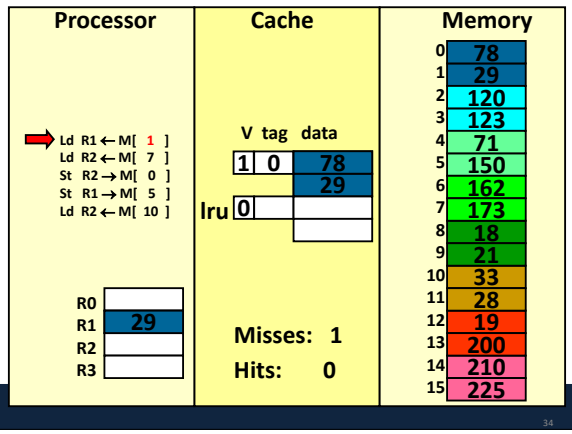


Poll: How can we calculate the tag?

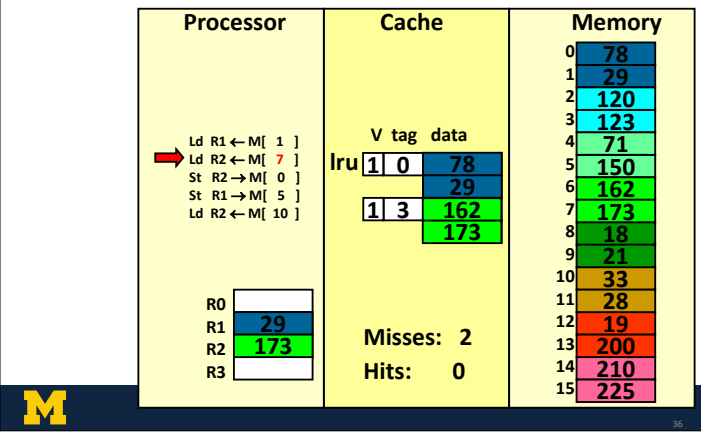
a) addr
b) floor(addr/2)
c) ceil(addr/2)
d) addr*2

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

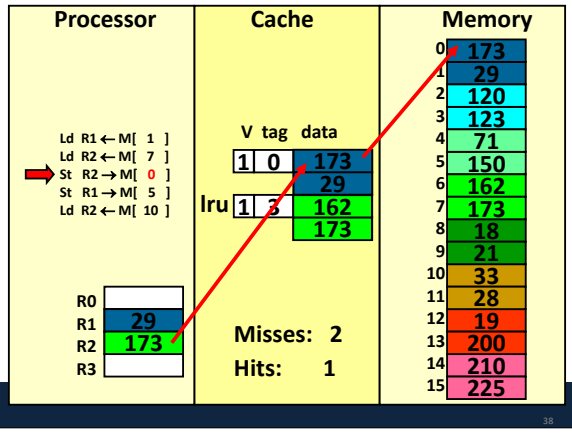
write-through, allocate on write (REF 1)



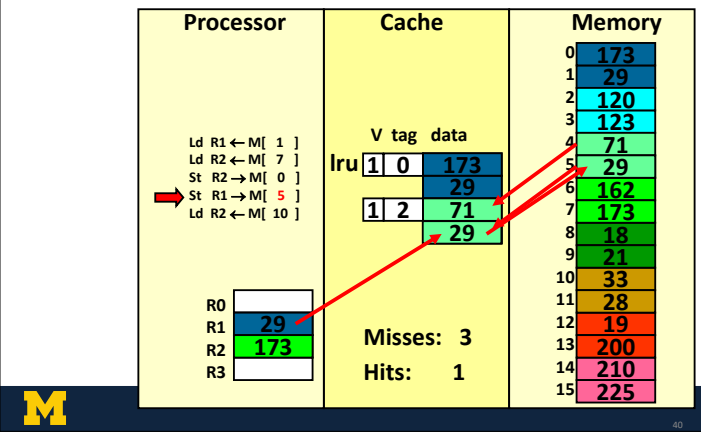
write-through, allocate on write (REF 2)



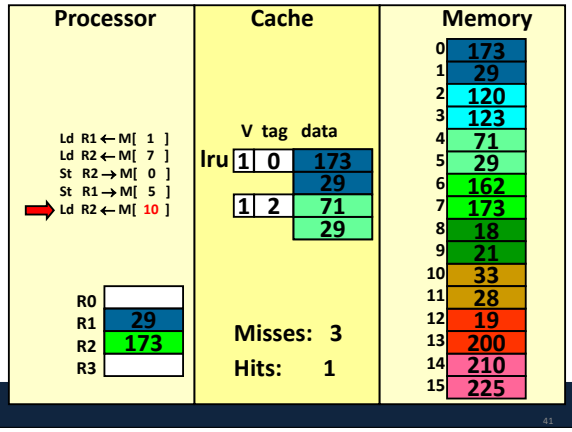
write-through, allocate on write (REF 3)



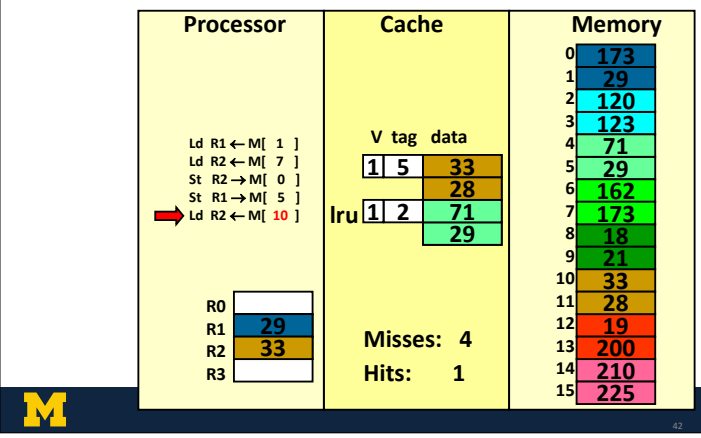
write-through, allocate on write (REF 4)



write-through, allocate on write (REF 6)



write-through, allocate on write (REF 6)



How many memory references?

- Each miss reads a block
  - 2 bytes in this cache
- Each store writes a byte
- Total reads: 8 bytes
- Total writes: 2 bytes

- but caches generally miss < 20%
  - Can we take advantage of that?
  - Multi-core processors have limited bandwidth between caches and memory
  - Extra stores also cost power

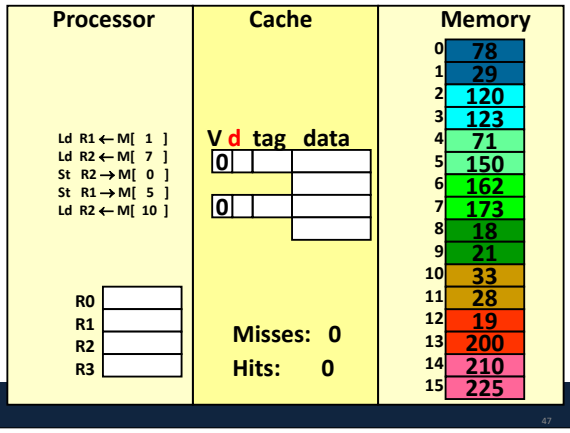
Agenda

- Larger Cache Blocks
- Extra Problems
- LRU with More than Two Blocks
- Write-Through Cache
- **Write-Back Cache**

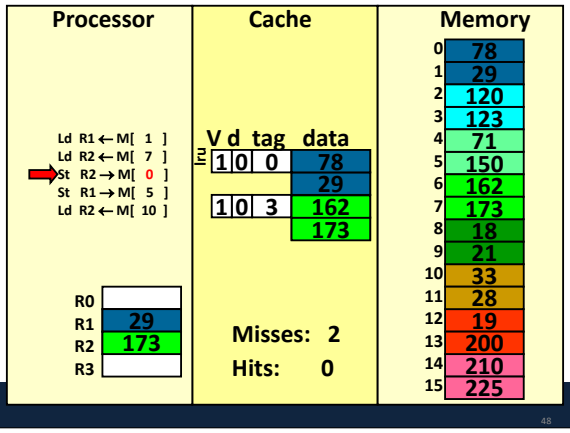
Write-through vs write-back

- Can we design the cache to NOT write all stores to memory immediately?
  - Keep the most recent copy in the cache and update the memory **only when** that data is evicted from the cache (write-back)
  - Do we need to write-back all evicted lines?
    - No, only blocks that have been modified
    - Keep a "dirty bit", reset when the line is allocated, set when the block is stored into. If a block is "dirty" when evicted, write its data back into memory.

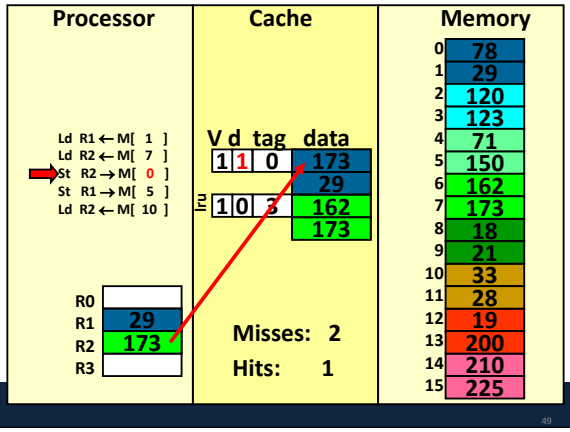
Handling stores (write-back)



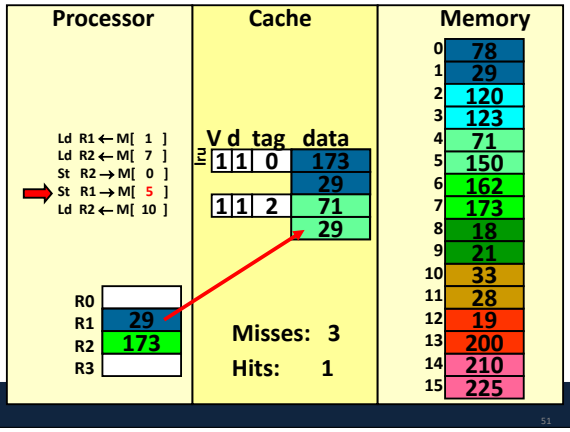
write-back (REF 3)



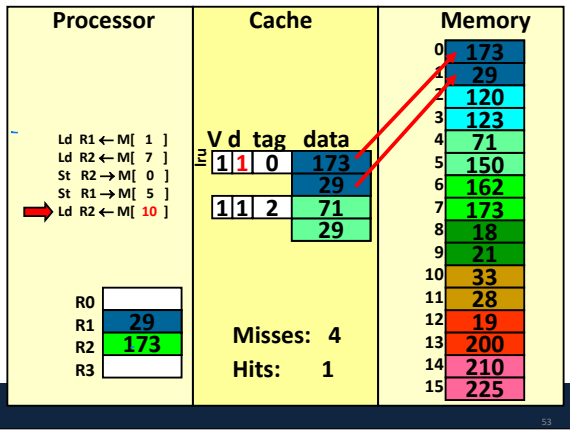
write-back (REF 3)



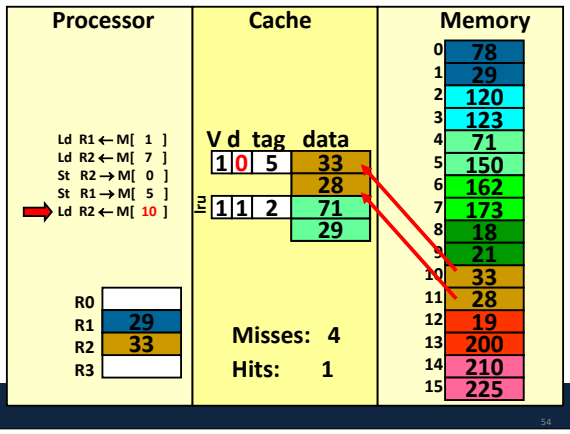
write-back (REF 4)



write-back (REF 5)



write-back (REF 5)



How many memory references?

- Each miss reads a block
  - 2 bytes in this cache
- Each evicted dirty cache line writes a block
- Total reads: 8 bytes
- Total writes: 4 bytes (after final eviction)

For this example, would you choose write-back or write-through?

Write-back works best when we write to a particular address multiple times before evicting

Review: Writes

Store w No Allocate	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache **+ Memory**
Miss?	Write to Memory	Write to Memory
Replace block?	If evicted block is dirty, write to Memory	Do Nothing
Store w Allocate	**Write-Back**	**Write-Through**
Hit?	Write Cache	Write to Cache **+ Memory**
Miss?	Read from Memory to Cache, Allocate to LRU block Write to Cache	Read from Memory to Cache, Allocate to LRU block Write to Cache **+ Memory**
Replace block?	If evicted block is dirty, write to Memory	Do Nothing

Next time

- Direct-mapped vs associative caches.