Lec 17: Caches

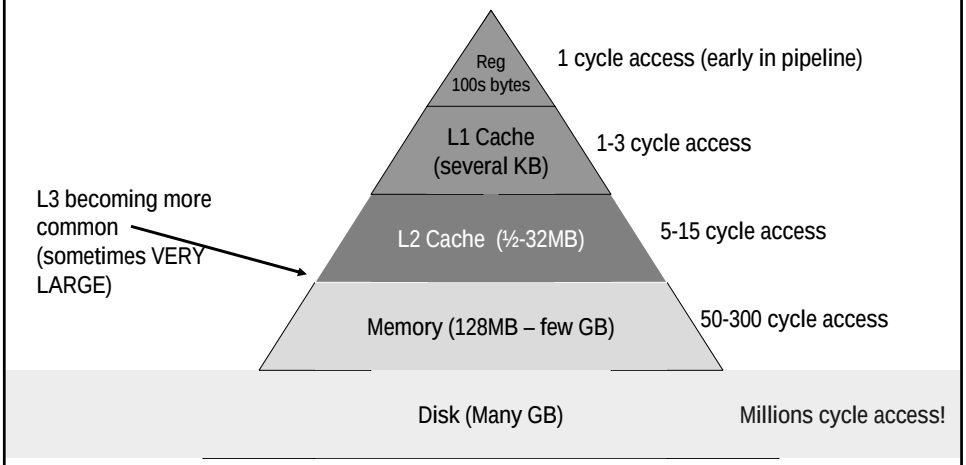
Kavita Bala
CS 3410, Fall 2008
Computer Science
Cornell University

Announcements

- Prelim: graded
- PA 2: graded
- HW 2: graded

- HW 3 out tonight: cache simulation
  - Recitations this week on C/Unix/etc.

Cache Design 101



These are rough numbers: mileage may vary for latest/greatest
Caches USUALLY made of SRAM

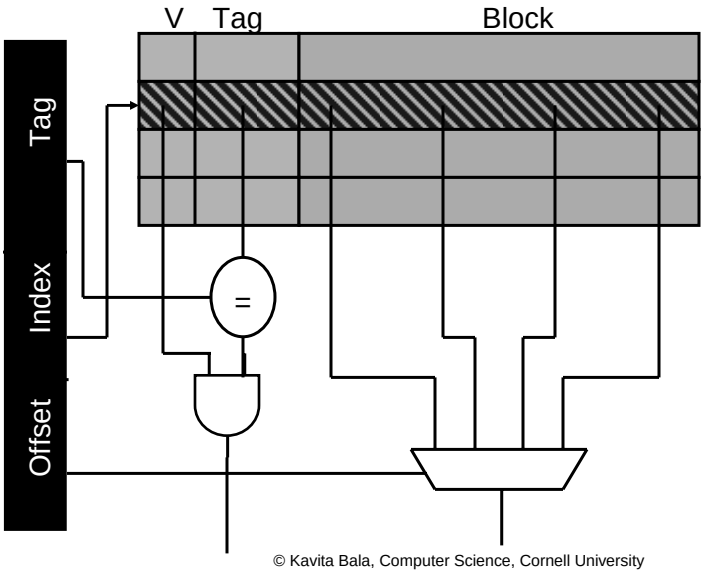
Insight of Caches

- Exploit locality
  - Two types: temporal and spatial
- Temporal locality
  - If memory location X is accessed, then it is more likely to be accessed again in the near future than some random location Y
  - Caches exploit temporal locality by placing a memory element that has been referenced into the cache
- Spatial locality
  - If memory location X is accessed, then locations near X are more likely to be accessed in the near future than some random location Y
  - Caches exploit spatial locality by allocating a cache line of data (including data near the referenced location)

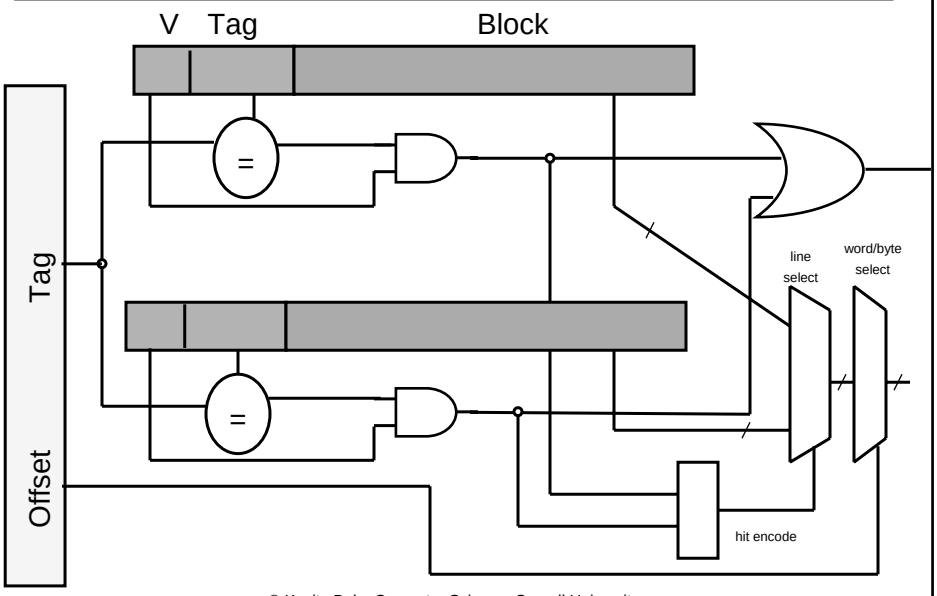
Cache Lookups (Read)

- Look at address issued by processor
- Search cache to see if that block is in the cache
  - Hit: Block is in the cache
    - return requested data
  - Miss: Block is not in the cache
    - read line from memory
    - evict an existing line from the cache
    - place new line in cache
    - return requested data

Direct Mapped Cache



Fully Associative Cache



Cache Organization

- Three common designs
  - Fully associative: Block can be anywhere in the cache
  - Direct mapped: Block can only be in one line in the cache
  - Set-associative: Block can be in a few (2 to 8) places in the cache

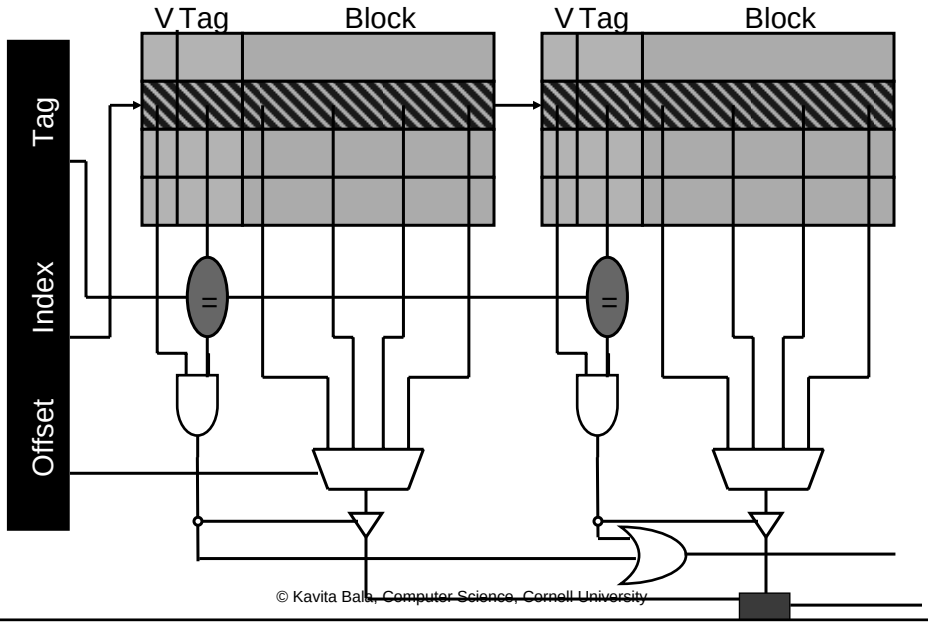
Eviction

- Which cache line should be evicted from the cache to make room for a new line?
  - Direct-mapped
    - no choice, must evict line selected by index
  - Associative caches
    - random: select one of the lines at random
    - round-robin: similar to random
    - FIFO: replace oldest line
    - LRU: replace line that has not been used in the longest time

Compromise

- Set-associative cache
- Like a direct-mapped cache
  - Index into a location
  - Fast
- Like a fully-associative cache
  - Can store multiple entries
    - decreases thrashing in cache
  - Search in each element

2-Way Set-Associative Cache



Direct Mapped

Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[4]
Ld R2 ← M[0]
Ld R2 ← M[12]
Ld R2 ← M[5]
⇒ Ld R2 ← M[12]
Ld R2 ← M[5]
Ld R2 ← M[12]
Ld R2 ← M[5]

Cache

4 cache lines
1 bit tag field
2 word block

	tag	data
1	0	100
		110
0		
1	0	140
		150
0		

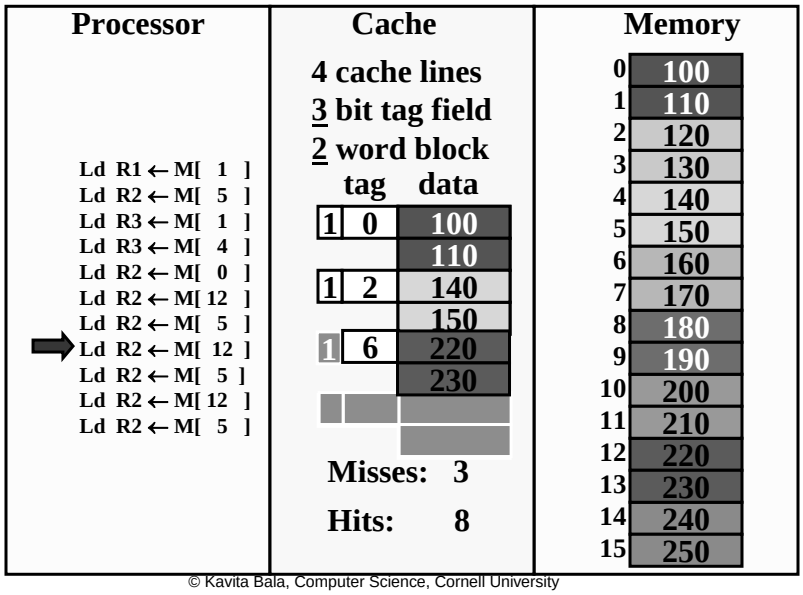
Misses: 4+2+2

Hits: 3

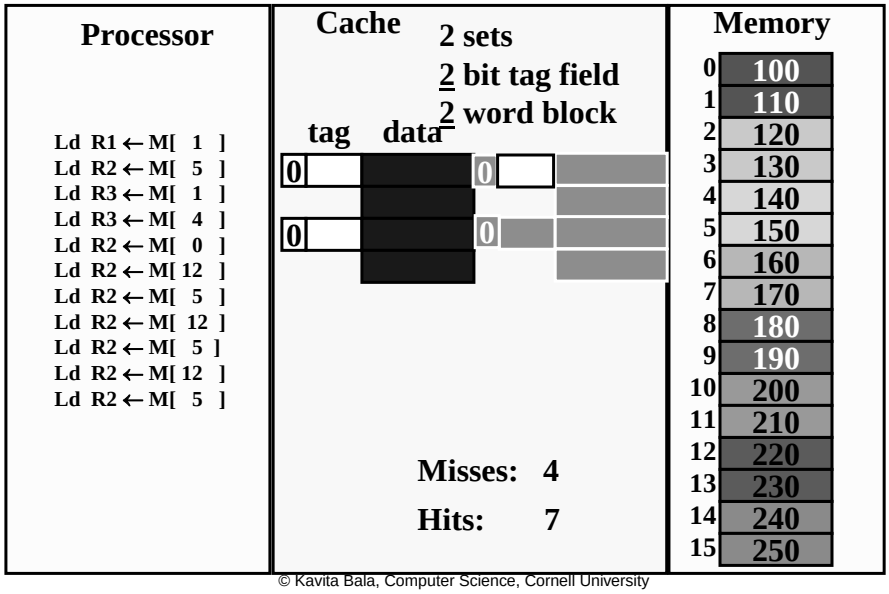
Memory

0	100
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

Fully Assoc



2 Way Set Assoc



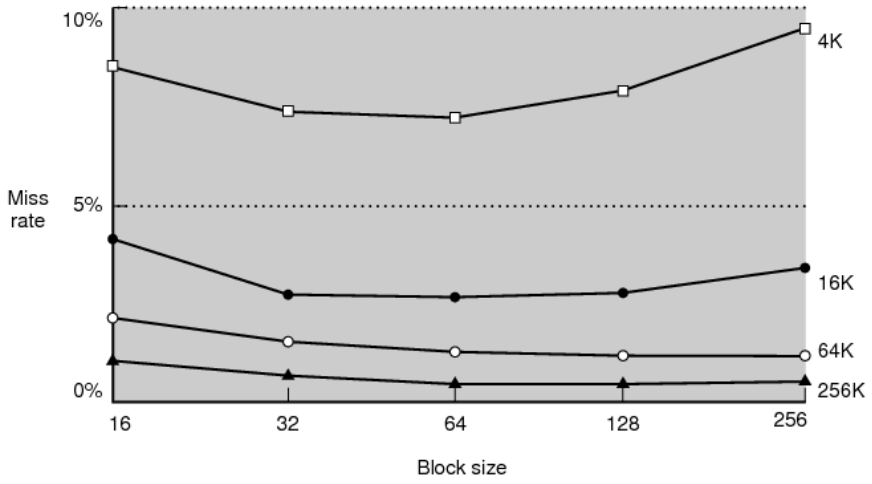
Cache Design

- Need to determine parameters
  - Block size
  - Number of ways of set-associativity
  - Eviction policy
  - Write policy
  - Separate I-cache from D-cache

Basic Cache Organization

Decide on the block size

- How? Simulate lots of different block sizes and see which one gives the best performance
- Most systems use a block size between 32 bytes and 128 bytes



Tradeoff

- Larger sizes reduce the overhead by
  - Reducing the number of tags
  - Reducing the size of each tag
- But
  - Have fewer blocks available
  - And the time to fetch the block on a miss is longer

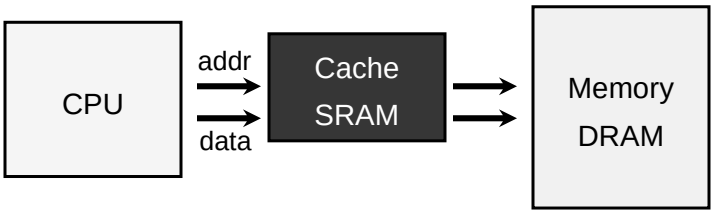
Valid Bits

- Valid bits indicate whether cache line contains an up-to-date copy of the values in memory
  - Must be 1 for a hit
  - Reset to 0 on power up
- An item can be removed from the cache by setting its valid bit to 0

Eviction

- Which cache line should be evicted from the cache to make room for a new line?
  - Direct-mapped
    - no choice, must evict line selected by index
  - Associative caches
    - random: select one of the lines at random
    - round-robin: similar to random
    - FIFO: replace oldest line
    - LRU: replace line that has not been used in the longest time

Cache Writes

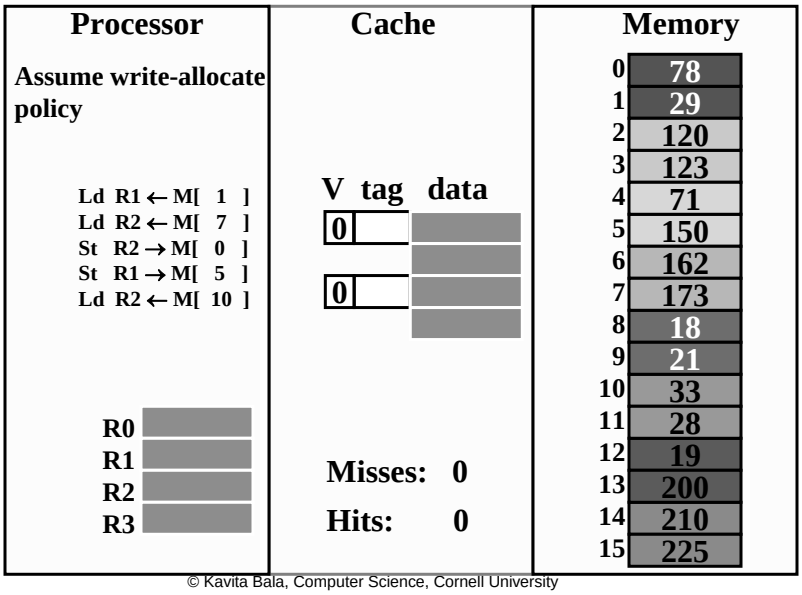


- No-Write
  - writes invalidate the cache and go to memory
- Write-Through
  - writes go to main memory and cache
- Write-Back
  - write cache, write main memory only when block is evicted

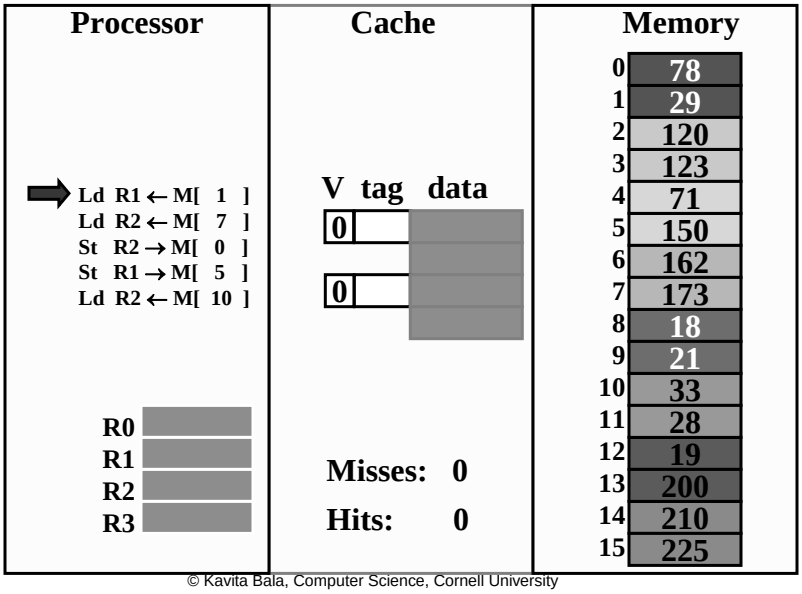
What about Stores?

- 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 right away? (write-through policy)
    - Wait until we kick the block out (write-back policy)
  - If it is not in the cache?
    - Allocate the line (put it in the cache)? (write allocate policy)
    - Write it directly to memory without allocation? (no write allocate policy)

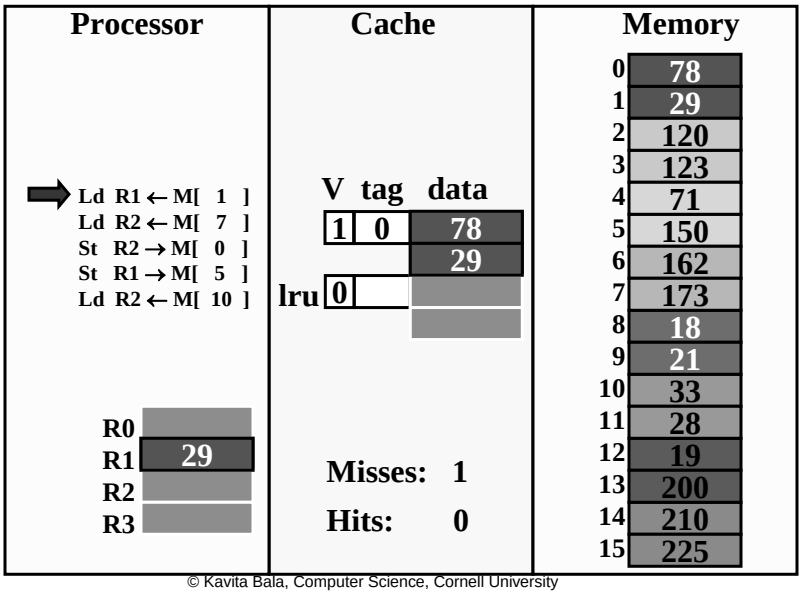
Handling Stores (Write-Through)



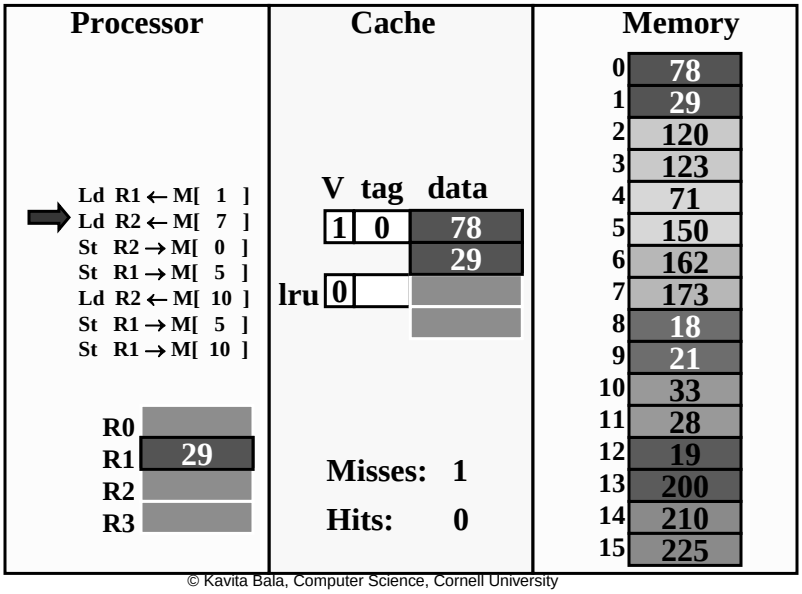
Write-Through (REF 1)



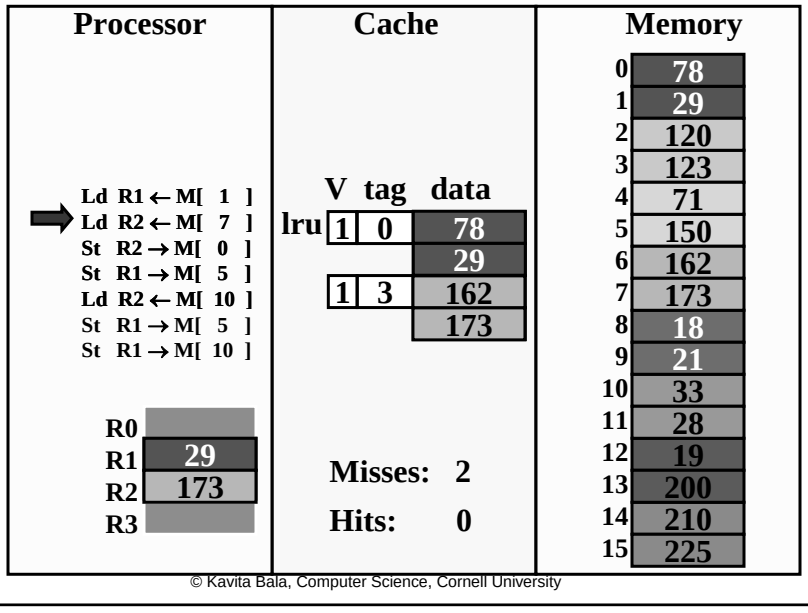
Write-Through (REF 1)



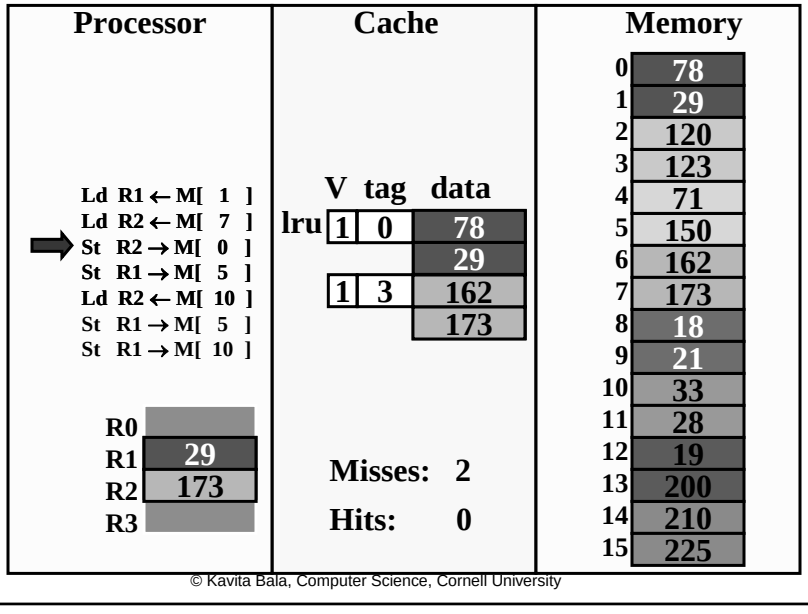
Write-Through (REF 2)



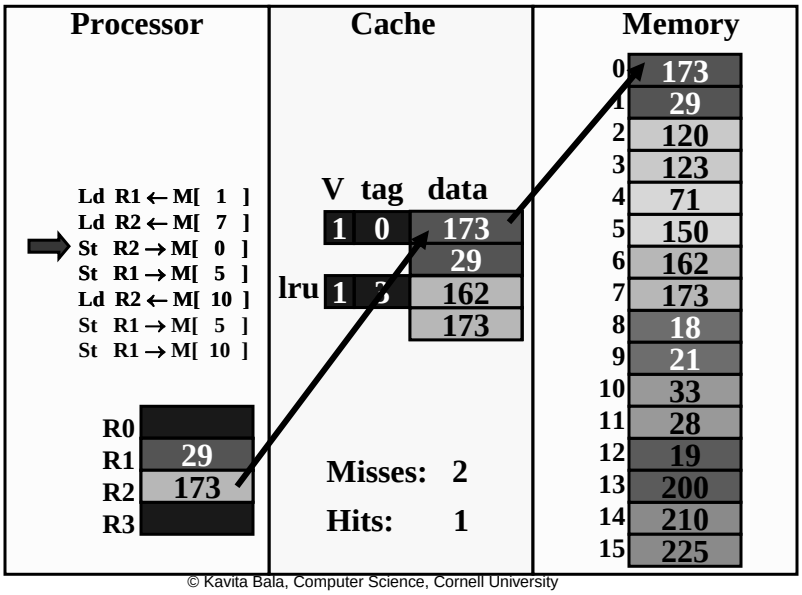
Write-Through (REF 2)



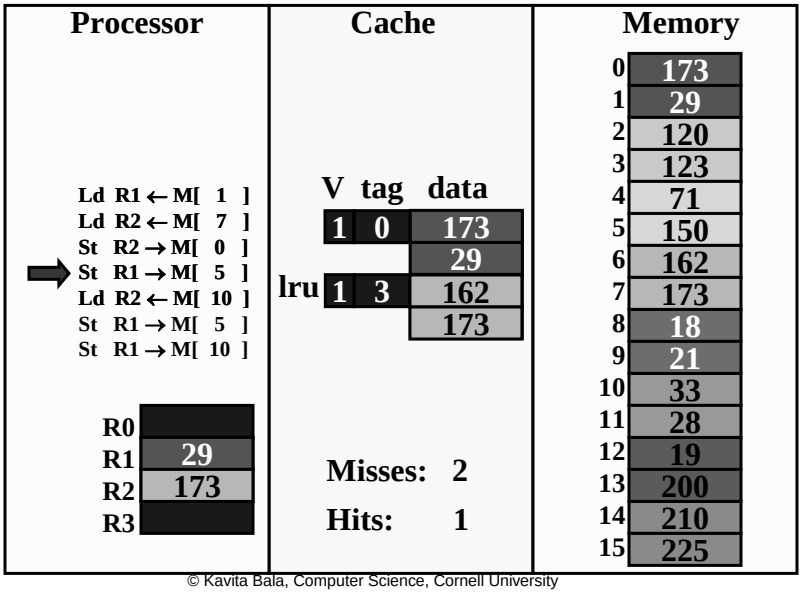
Write-Through (REF 3)



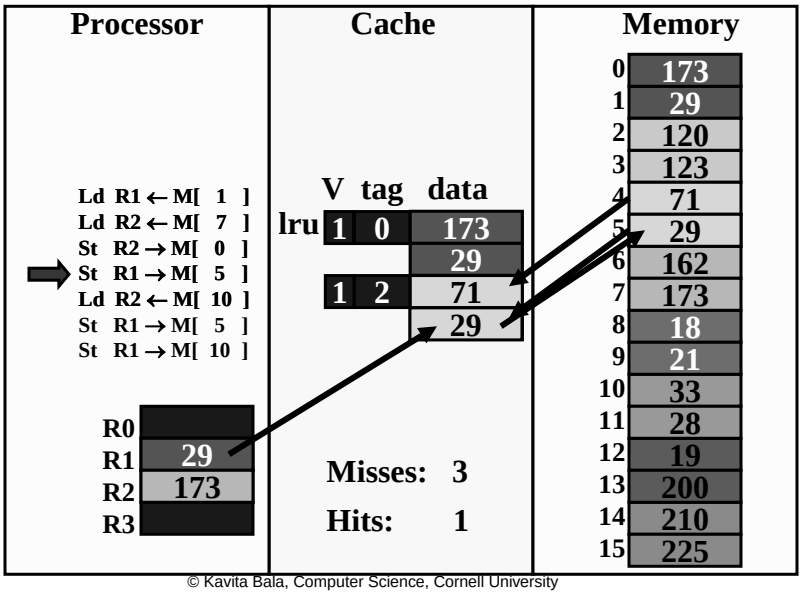
Write-Through (REF 3)



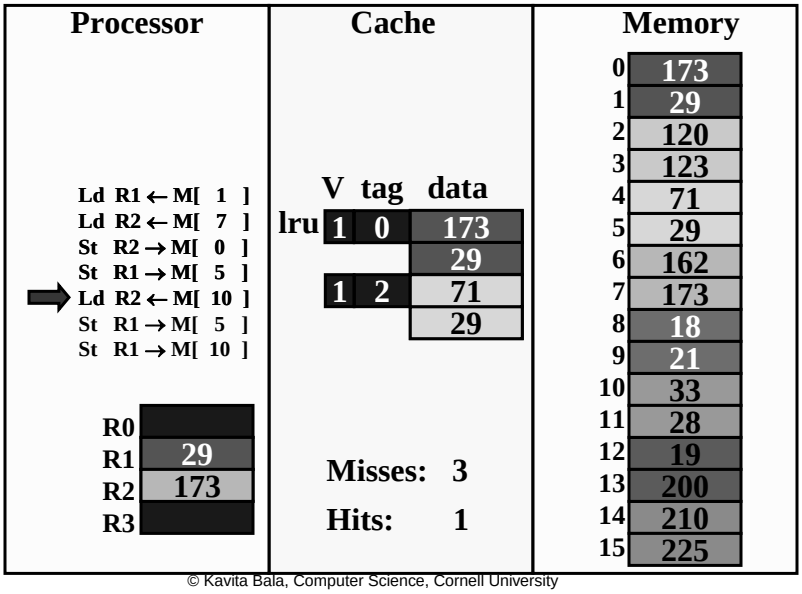
Write-Through (REF 4)



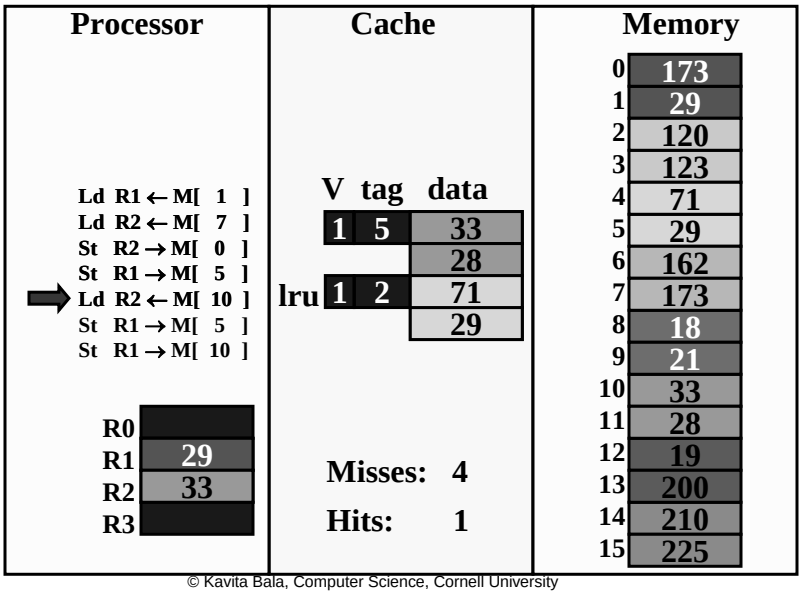
Write-Through (REF 4)



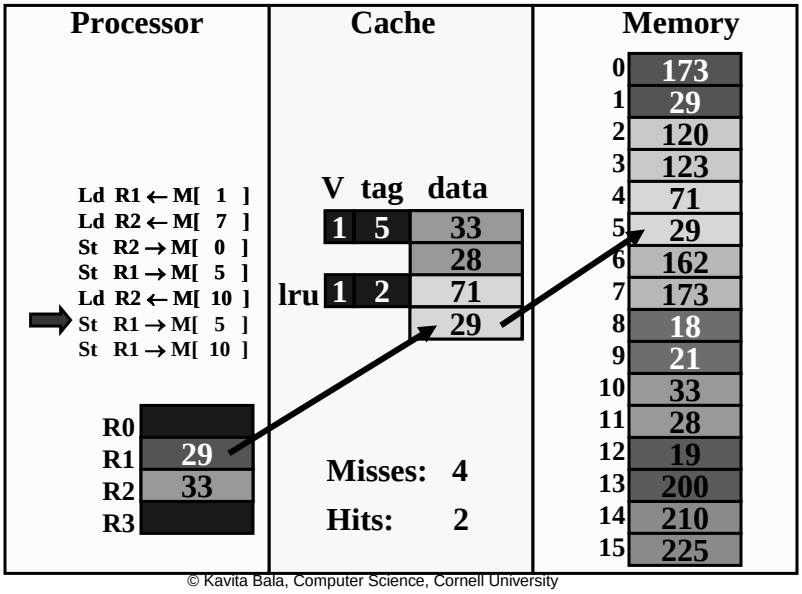
Write-Through (REF 5)



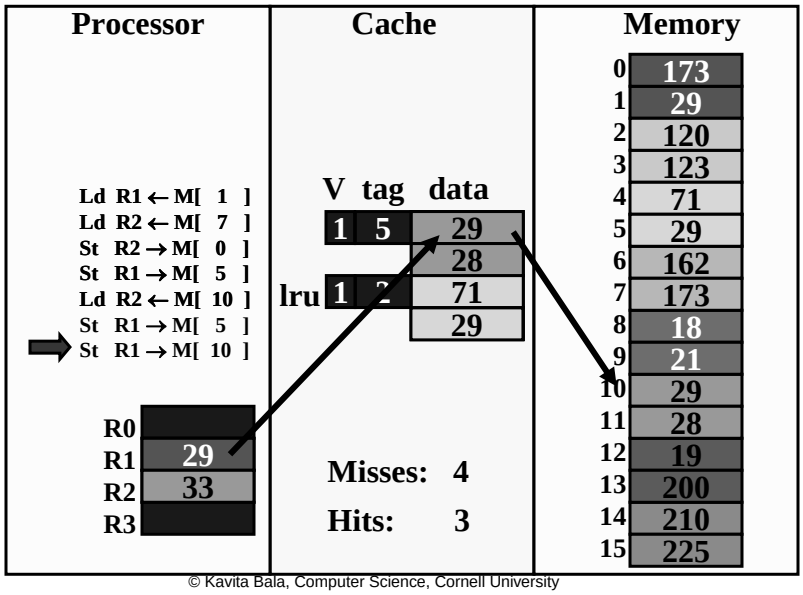
Write-Through (REF 5)



Write-Through (REF 6)



Write-Through (REF 7)

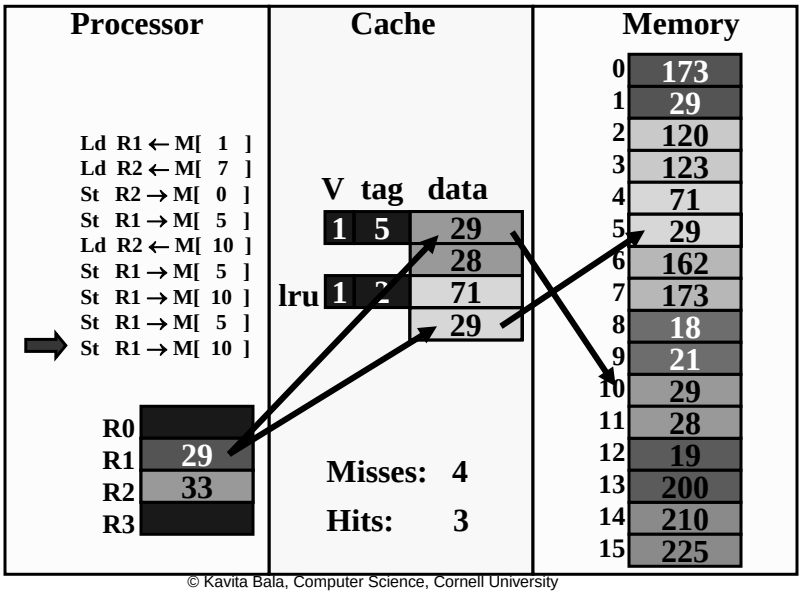


How Many Memory References?

- Each miss reads a block
  (only two words in this cache)
- Each store writes a word
- Total reads: eight words
- Total writes: four words

but caches generally miss < 20%
usually much lower miss rates . . . but depends on both cache and application!

Write-Through (REF 7)



How Many Memory References?

- Each miss reads a block
  (only two words in this cache)
- Each store writes a word
- Total reads: eight words
- Total writes: six words, eight words, etc.

but caches generally miss < 20%
usually much lower miss rates . . . but depends on both cache and application!

Write-Through vs. Write-Back

Can we also design the cache NOT to write all stores immediately to memory?

- Keep the most current copy in cache, and update memory when that data is evicted (write-back policy)
- Do we need to write-back all evicted lines?
- No, only blocks that have been stored into (written)

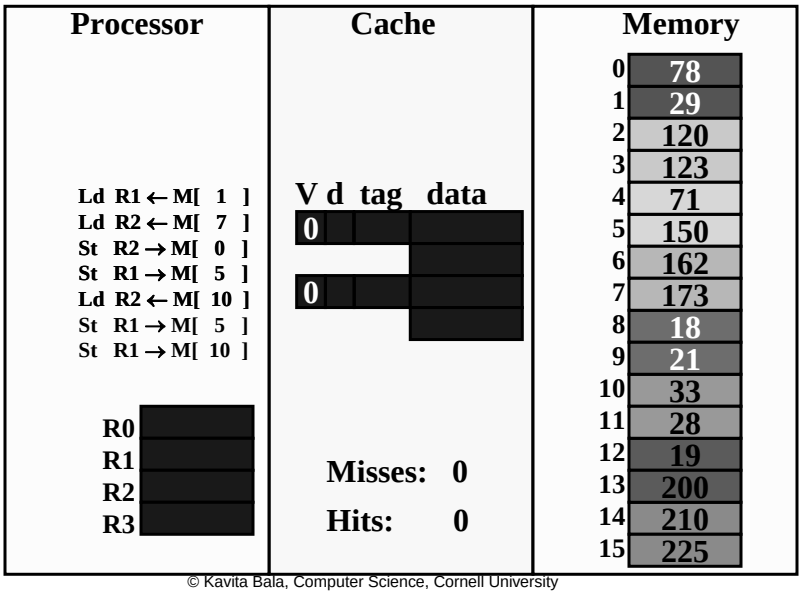
Dirty Bits and Write-Back Buffers

V	D	Tag	Data Byte 0, Byte 1 … Byte N
1	0		
1	1		
1	0		

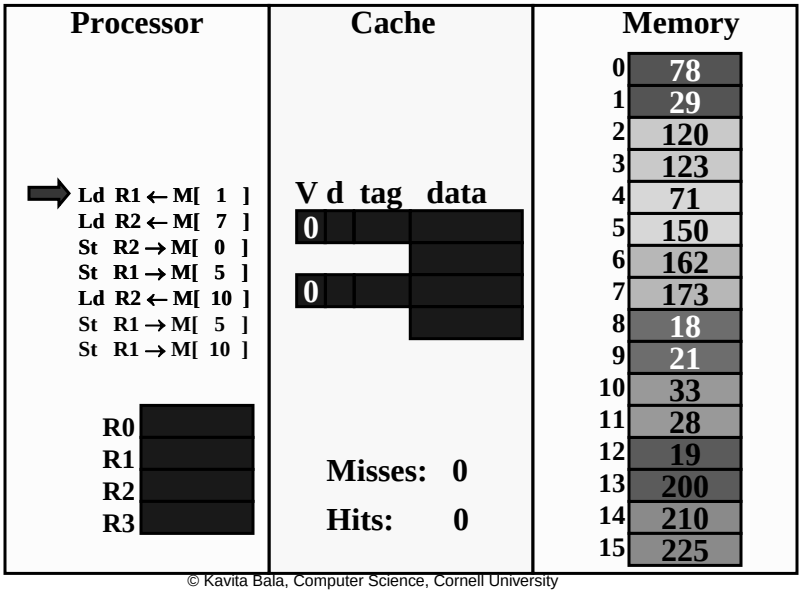
Line

- Dirty bits indicate which lines have been written
- Dirty bits enable the cache to handle multiple writes to the same cache line without having to go to memory
- Dirty bit reset when line is allocated
- Set when block is written
- Write-back buffer
  - A queue where dirty lines are placed
  - Items added to the end as dirty lines are evicted from the cache
  - Items removed from the front as memory writes are completed

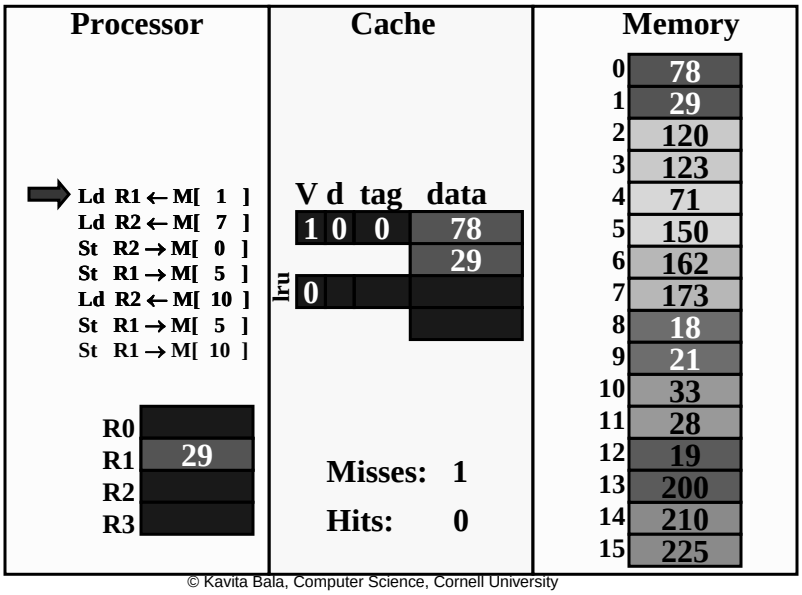
Handling Stores (Write-Back)



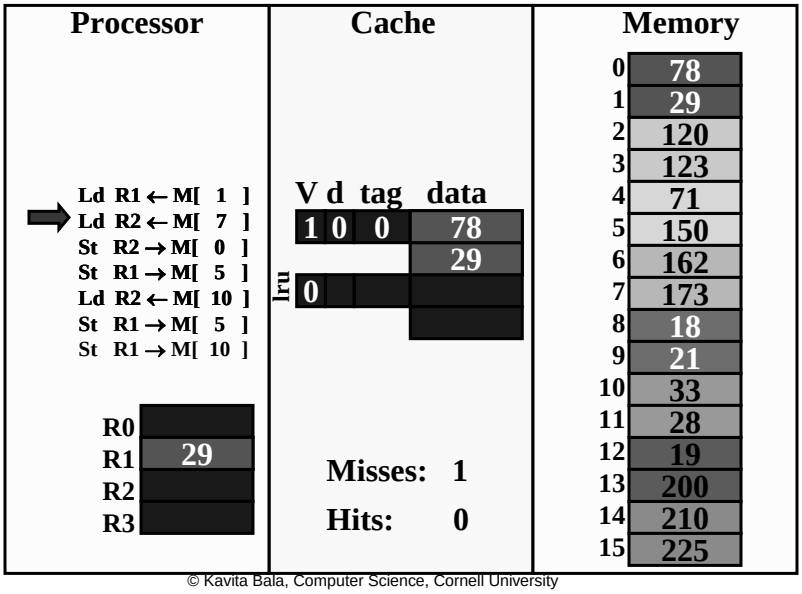
Write-Back (REF 1)



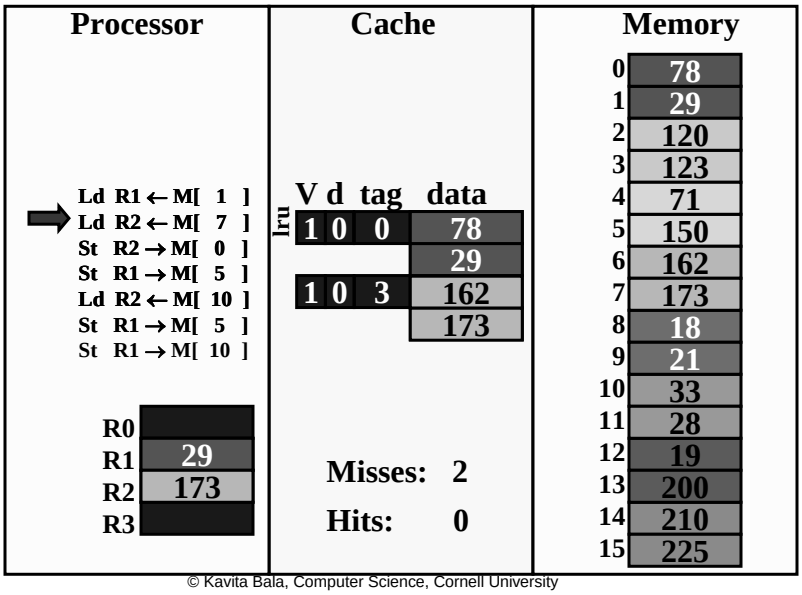
Write-Back (REF 1)



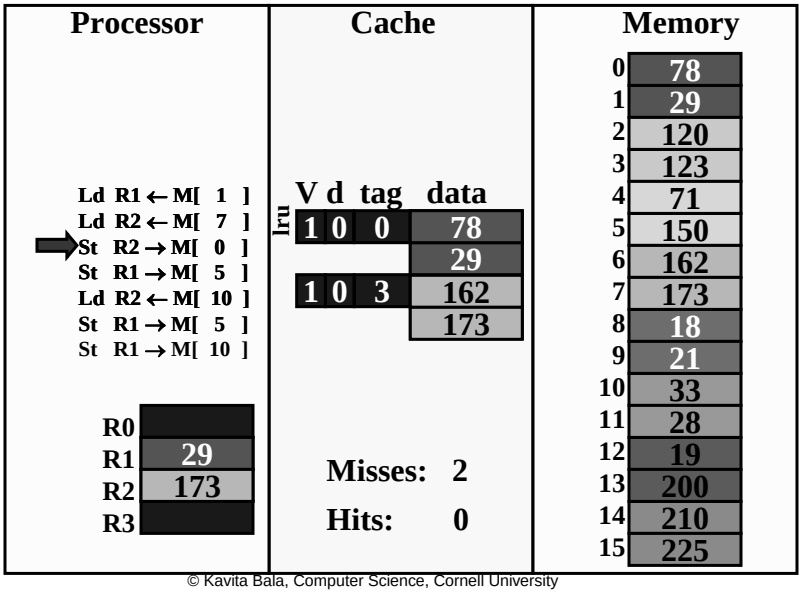
Write-Back (REF 2)



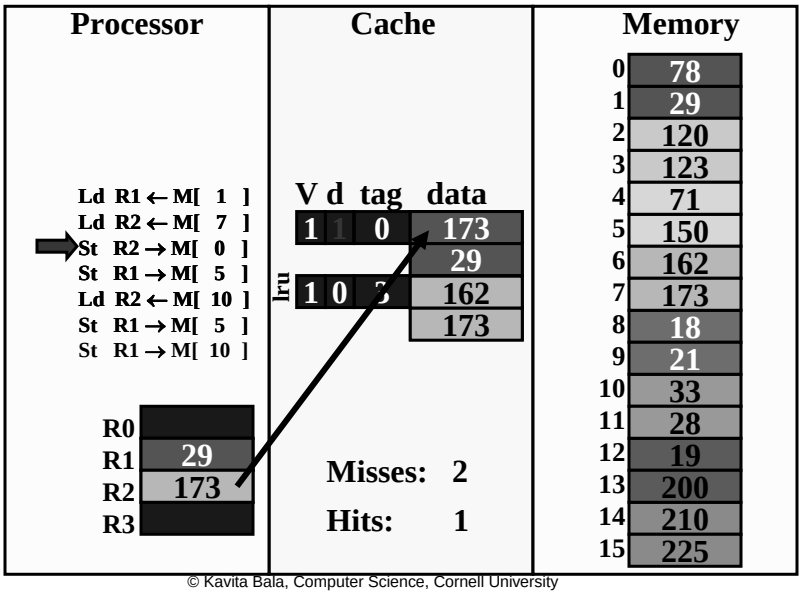
Write-Back (REF 2)



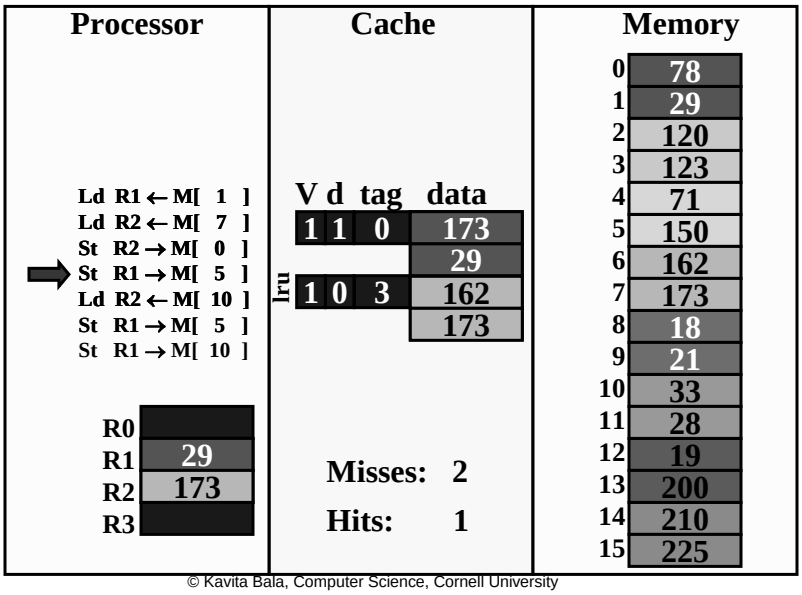
Write-Back (REF 3)



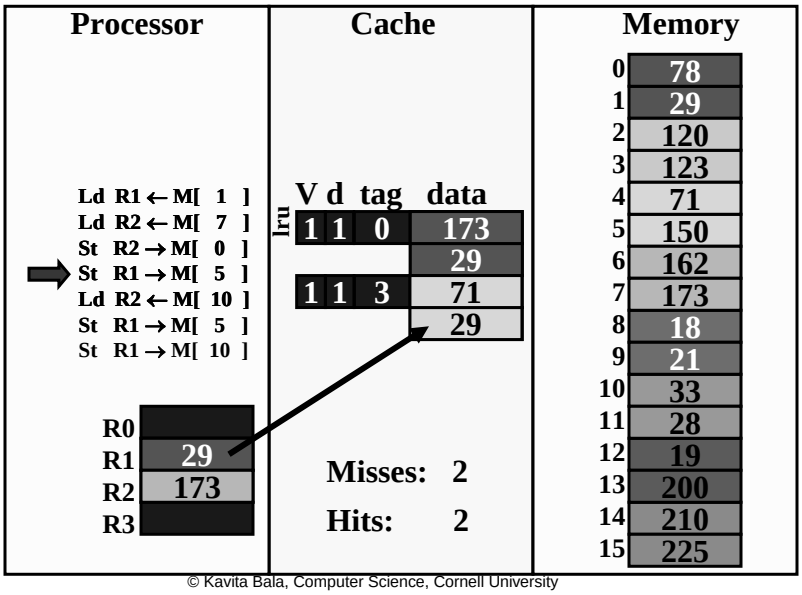
Write-Back (REF 3)



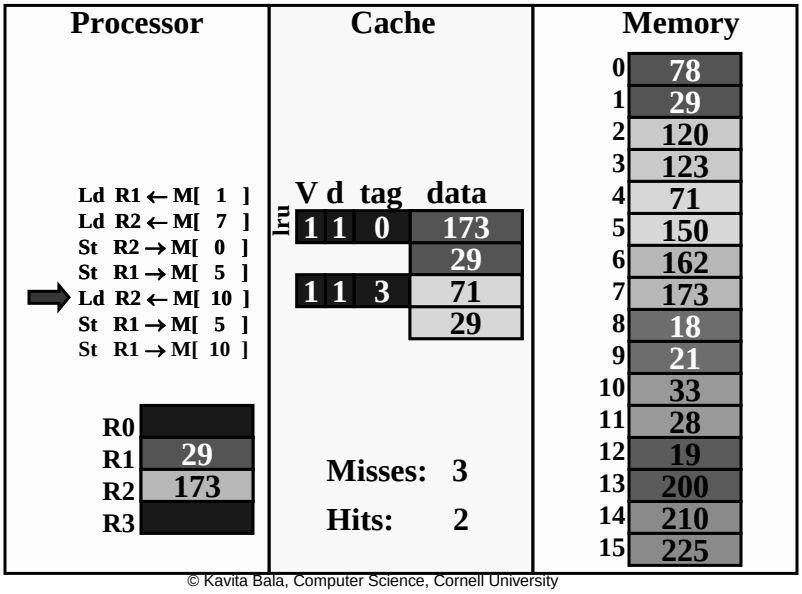
Write-Back (REF 4)



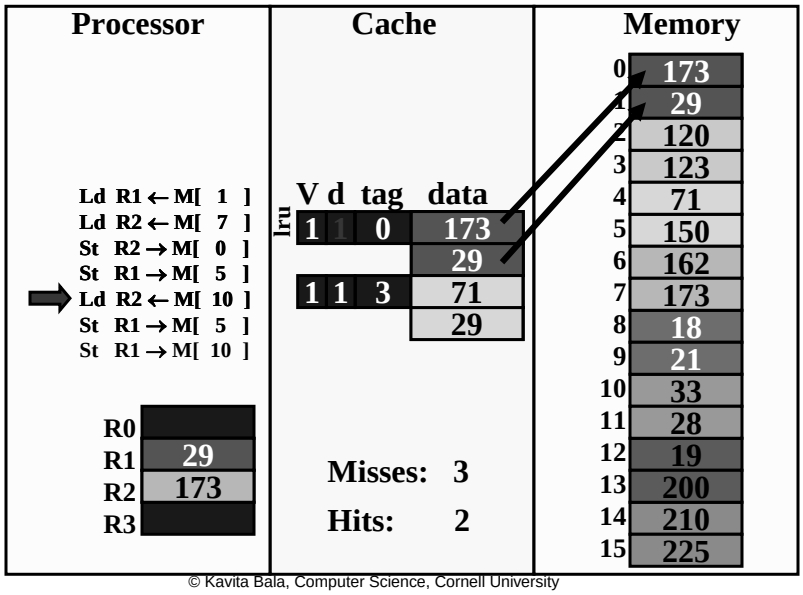
Write-Back (REF 4)



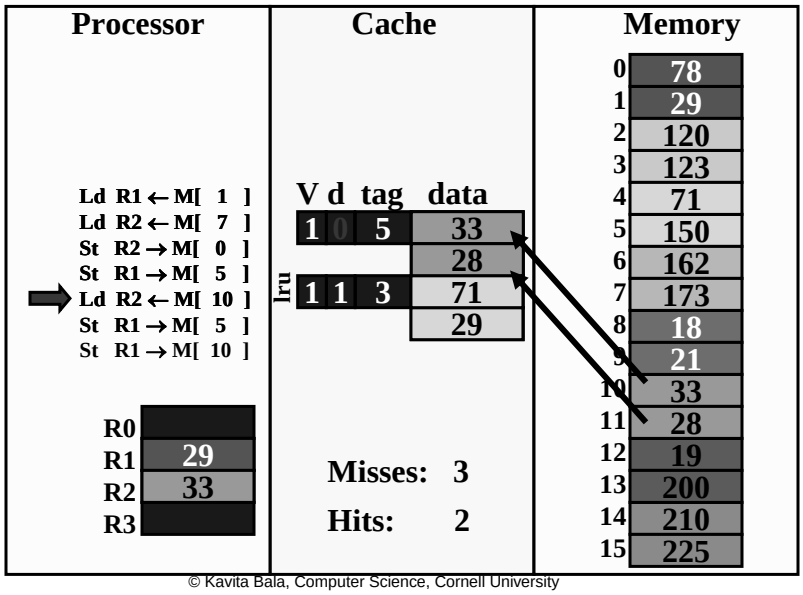
Write-Back (REF 5)



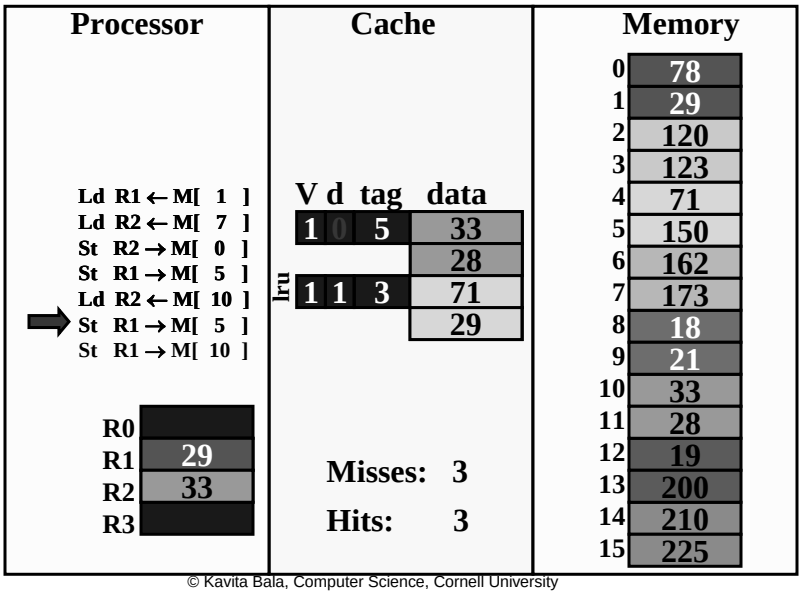
Write-Back (REF 5)



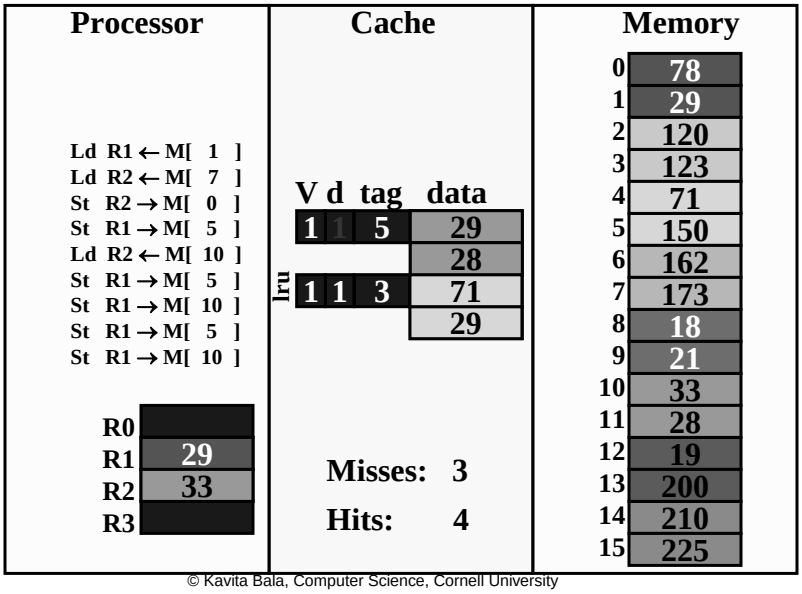
Write-Back (REF 5)



Write-Back (REF 6)



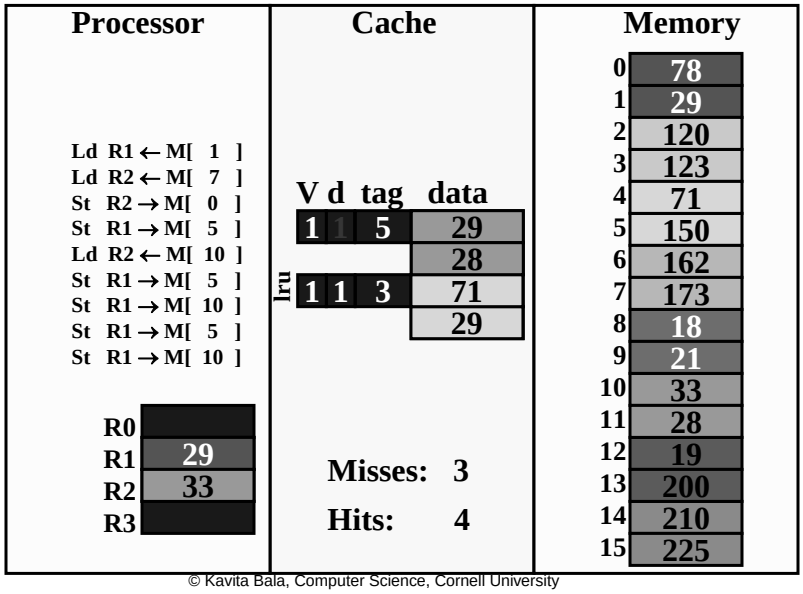
Write-Back (REF 7)



How many memory references?

- Each miss reads a block
  Two words in this cache
- Each evicted dirty cache line writes a block
- Total reads: six words
- Total writes: 4/6 words (after final eviction)

Write-Back (REF 8,9)



How many memory references?

- Each miss reads a block
  Two words in this cache
- Each evicted dirty cache line writes a block
- Total reads: six words
- Total writes: 4/6 words (after final eviction)
- By comparison write-through was
  - Reads: eight words
  - Writes: 6/8/10 etc words
- Write-through or Write-back?

Write-through vs. Write-back

- Write-through is slower
  - But cleaner (memory always consistent)
- Write-back is faster
  - But complicated when multi cores sharing memory