CS 61C: Great Ideas in Computer Architecture (Machine Structures) Caches Part 2

Instructors:

Bernhard Boser & Randy H. Katz

http://inst.eecs.berkeley.edu/~cs61c/

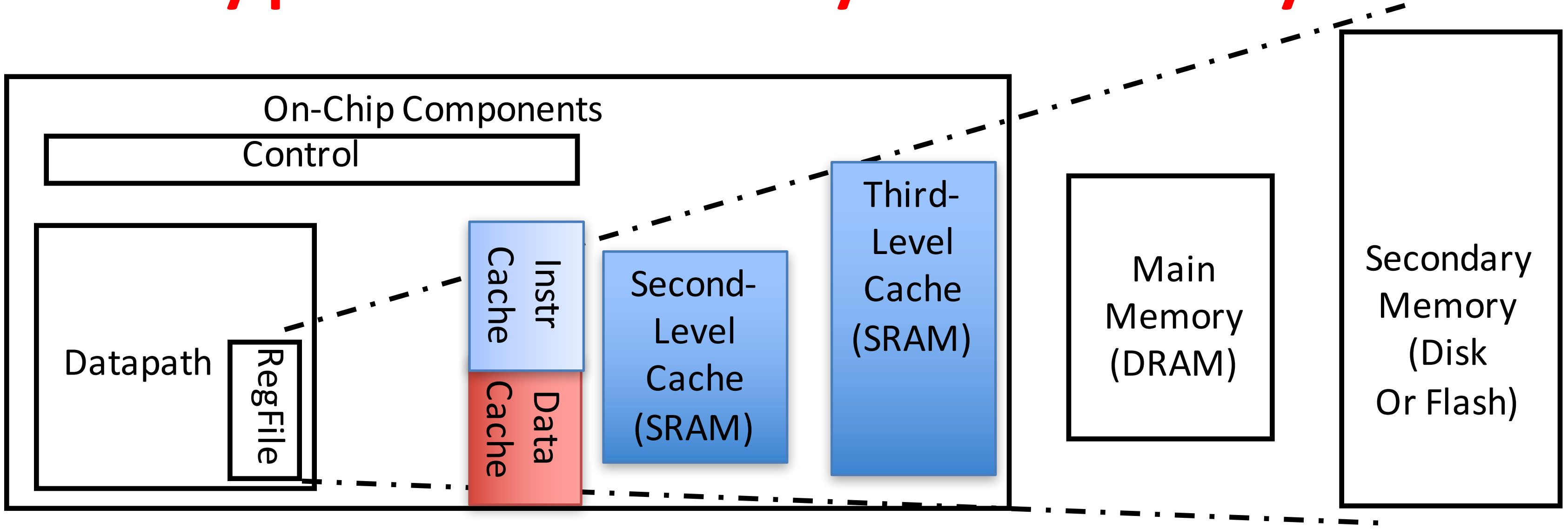
Outline

- Cache Organization and Principles
- Write Back vs. Write Through
- Cache Performance
- Cache Design Tradeoffs
- And in Conclusion …

Outline

- Cache Organization and Principles
- Write Back vs. Write Through
- Cache Performance
- Cache Design Tradeoffs
- And in Conclusion …

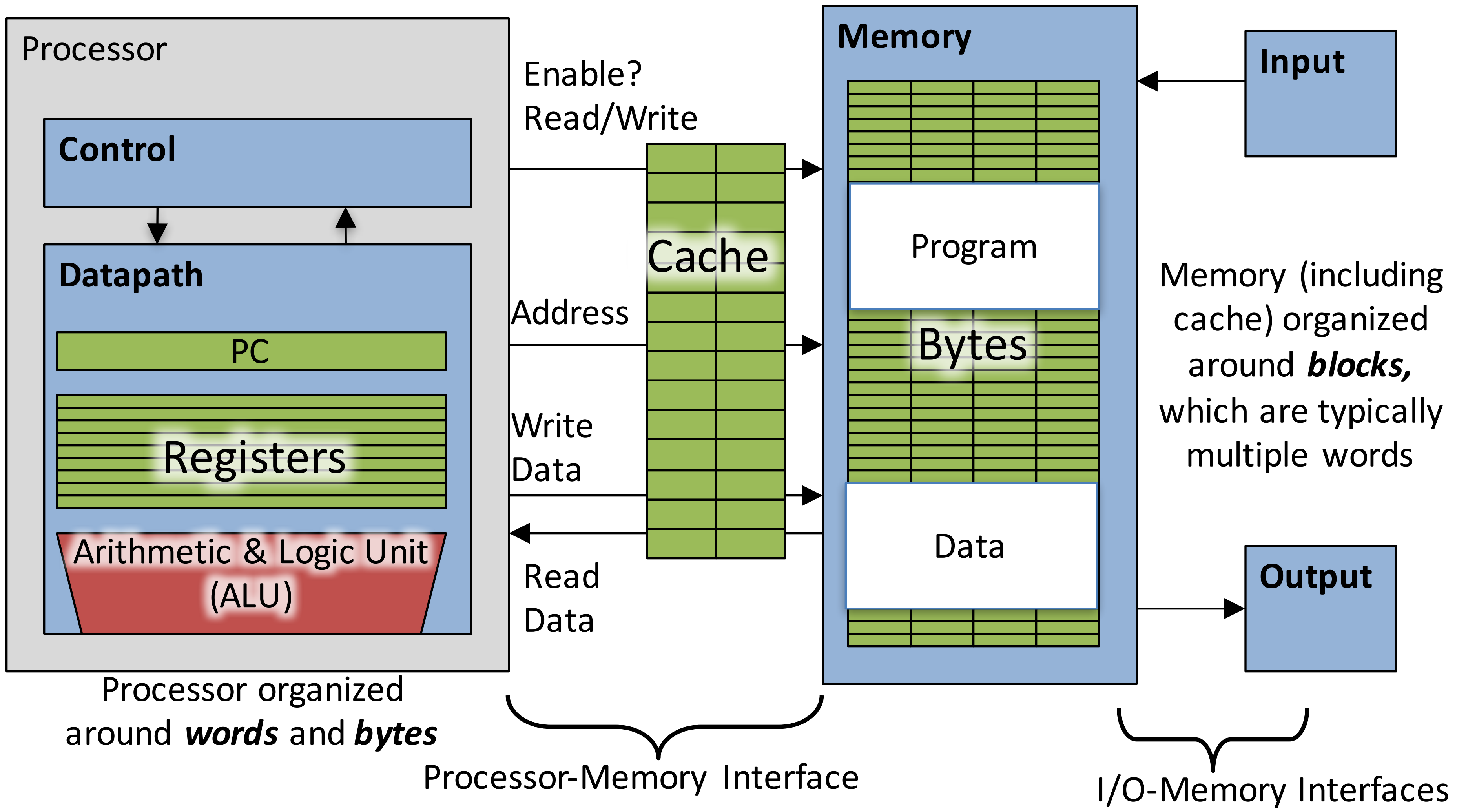
Typical Memory Hierarchy



Speed (cycles):	½'s	1's	10's	100's-1000	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest	⟷			lowest

- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

Adding Cache to Computer



Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data
Read Data
Cache
Memory
Program
Bytes
Data
Input
Output
Memory (including cache) organized around *blocks,* which are typically multiple words
Processor organized around *words* and *bytes*
Processor-Memory Interface
I/O-Memory Interfaces

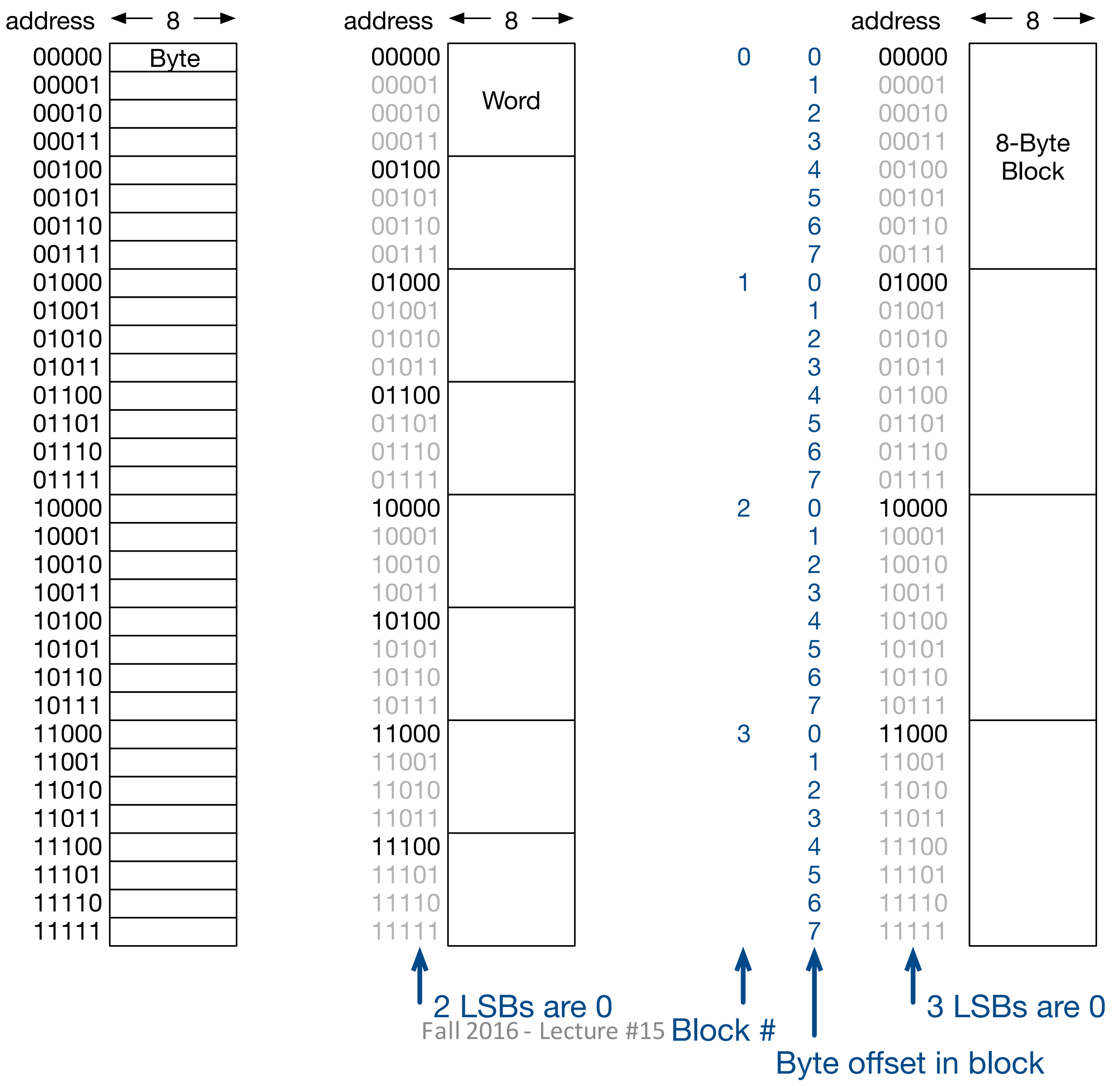
Key Cache Concepts

- Principle of Locality
  - Temporal Locality and Spatial Locality
- Hierarchy of Memories (speed/size/cost per bit) to exploit locality
- Cache – copy of data in lower level of memory hierarchy
- Direct Mapped to find block in cache using Tag field and Valid bit for Hit
- Cache Design Organization Choices:
  - Fully Associative, Set-Associative, Direct-Mapped

Cache Organizations

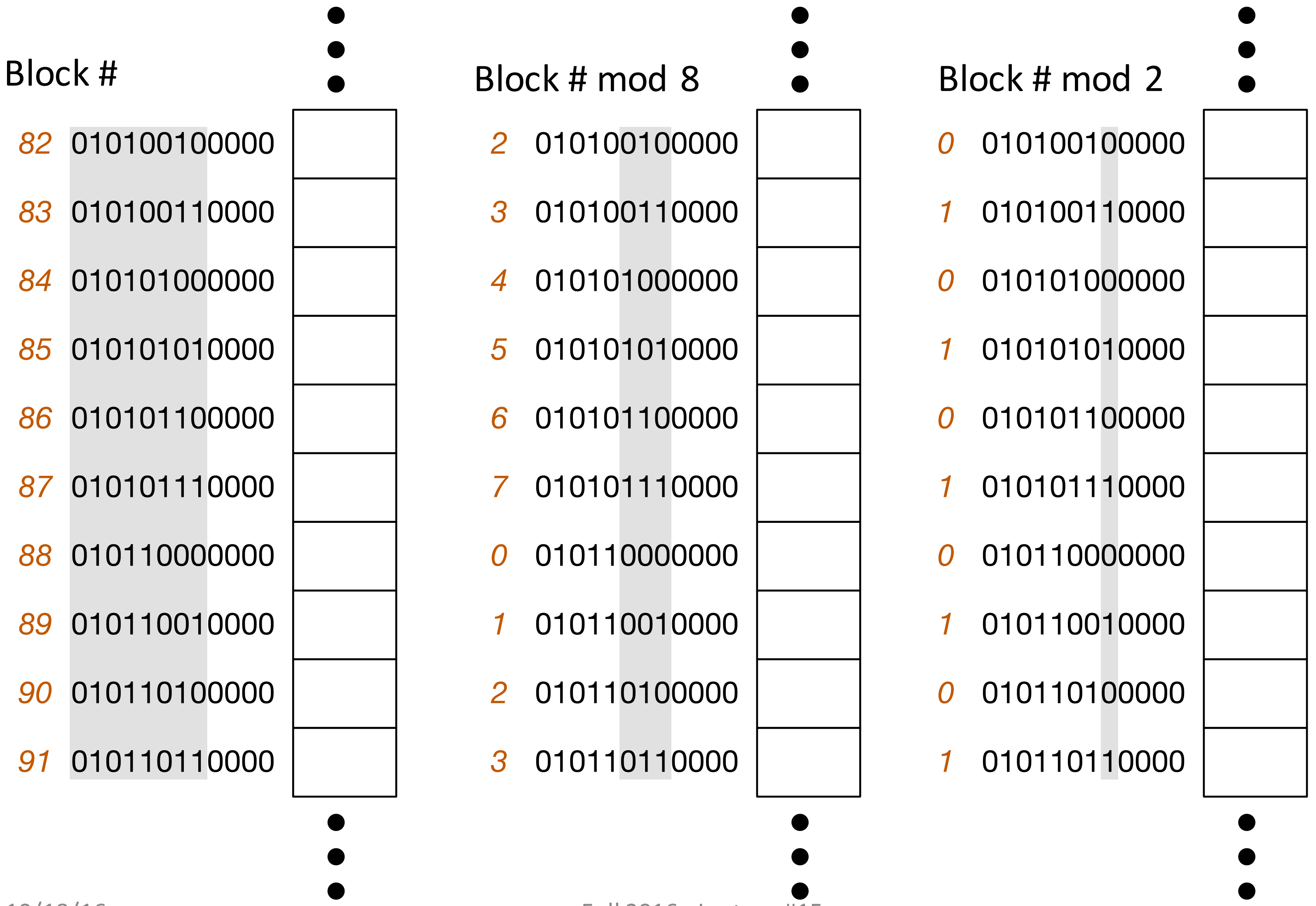
- "Fully Associative": Block placed anywhere in cache
  - First design last lecture
  - Note: No Index field, but one comparator/block
- "Direct Mapped": Block goes *only one place* in cache
  - Note: Only one comparator
  - Number of sets = number blocks
- "N-way Set Associative": N places for block in cache
  - Number of sets = Number of Blocks / N
  - N comparators
  - ***Fully Associative: N = number of blocks***
  - ***Direct Mapped: N = 1***

Memory Block vs. Word Addressing



Memory Block Number Aliasing

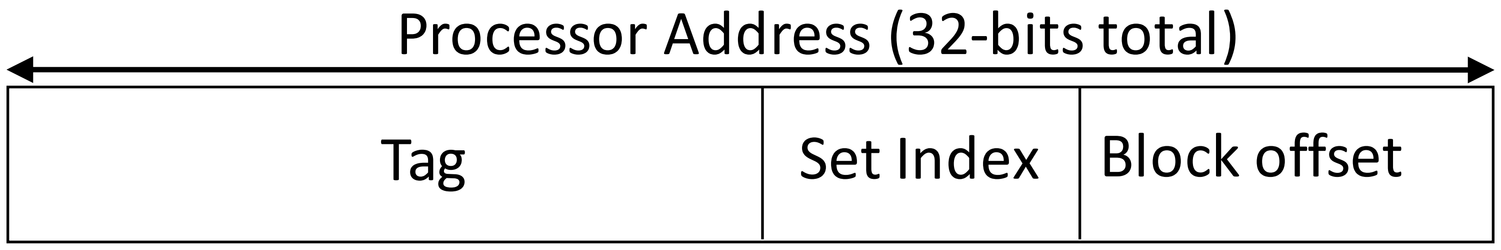
12-bit memory addresses, 16 Byte blocks



Block #		Block # mod 8		Block # mod 2	
82	010100100000	*2*	010100100000	*0*	010100100000
83	010100110000	*3*	010100110000	*1*	010100110000
84	010101000000	*4*	010101000000	*0*	010101000000
85	010101010000	*5*	010101010000	*1*	010101010000
86	010101100000	*6*	010101100000	*0*	010101100000
87	010101110000	*7*	010101110000	*1*	010101110000
88	010110000000	*0*	010110000000	*0*	010110000000
89	010110010000	*1*	010110010000	*1*	010110010000
90	010110100000	*2*	010110100000	*0*	010110100000
91	010110110000	*3*	010110110000	*1*	010110110000

Processor Address Fields used by Cache Controller

- Block Offset: Byte address within block
- Set Index: Selects which set
- Tag: Remaining portion of processor address



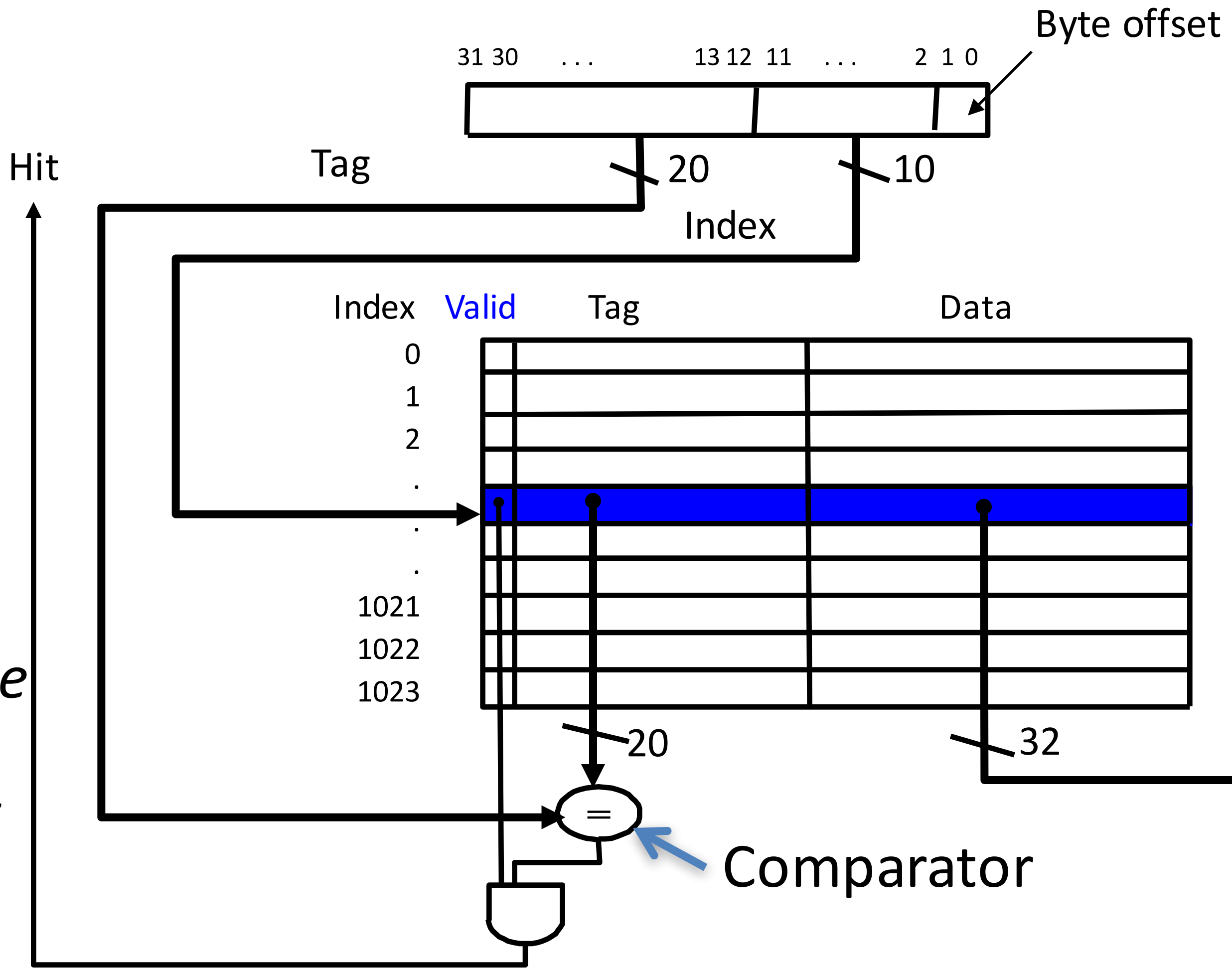
- Size of Index = $\log_2$(number of sets)
- Size of Tag = Address size – Size of Index – $\log_2$(number of bytes/block)

Direct-Mapped Cache Revisted

- One word blocks, cache size = 1K words (or 4KB)

Valid bit ensures something useful in cache for this index

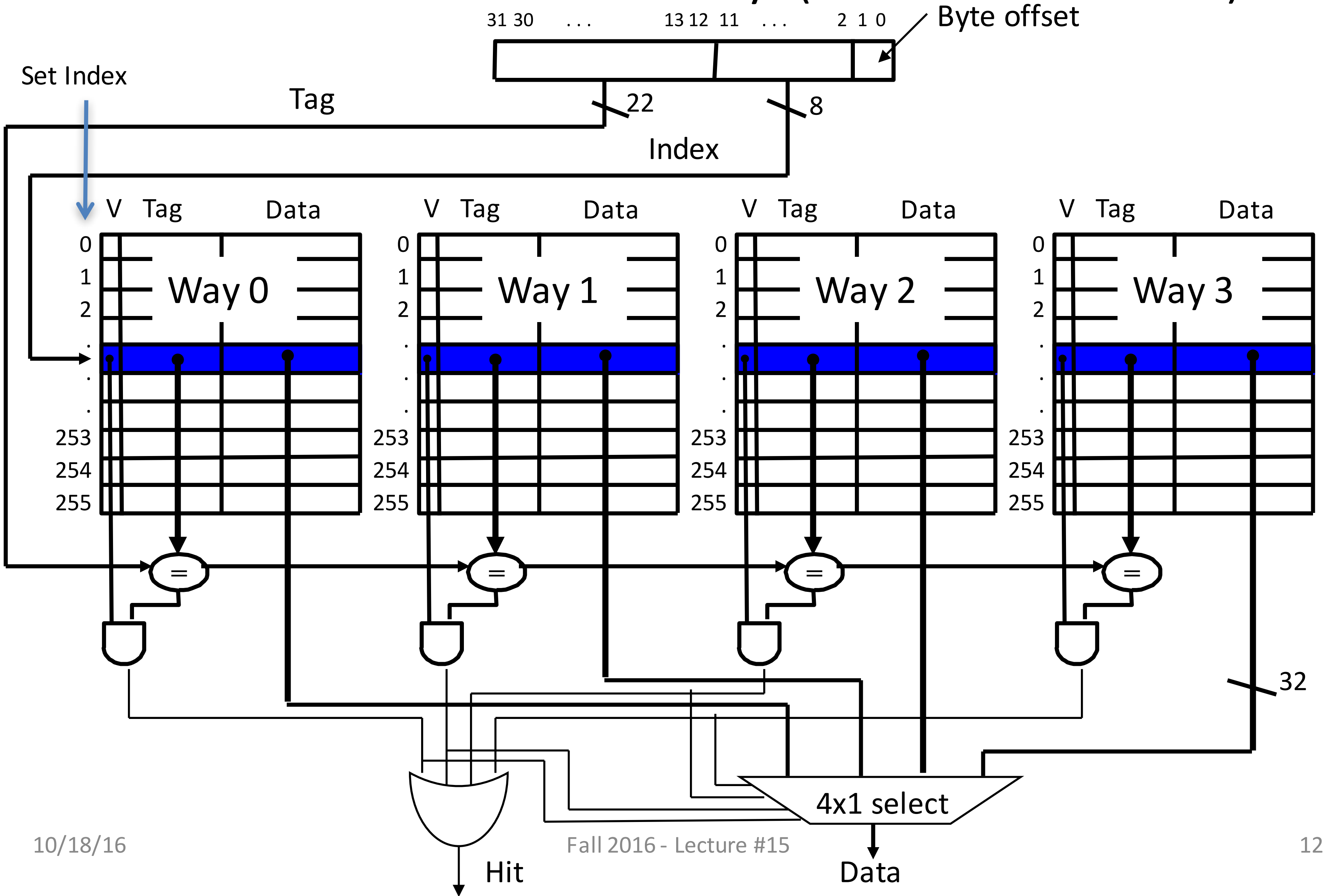
Compare Tag with upper part of Address to see if a Hit



Read data from cache instead of memory if a Hit

Four-Way Set-Associative Cache

- 2^8 = 256 sets each with four ways (each with one block)



Outline

- Cache Organization and Principles
- Write Back vs. Write Through
- Cache Performance
- Cache Design Tradeoffs
- And in Conclusion …

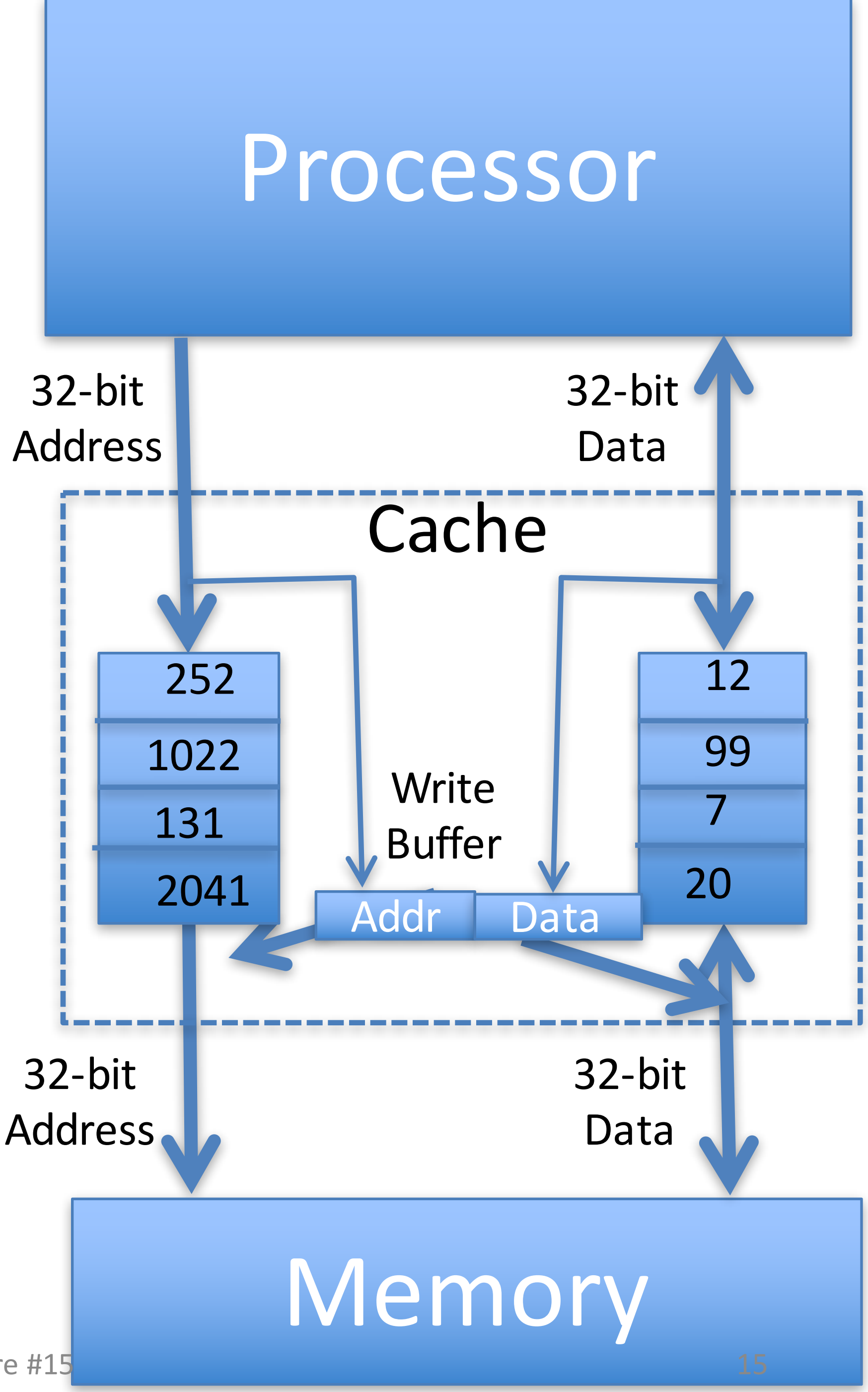
Handling Stores with Write-Through

- Store instructions write to memory, changing values
- Need to make sure cache and memory have same values on writes: two policies

1) Write-Through Policy: write cache and write *through* the cache to memory
   - Every write eventually gets to memory
   - Too slow, so include Write Buffer to allow processor to continue once data in Buffer
   - Buffer updates memory in parallel to processor

Write-Through Cache

- Write both values in cache and in memory
- Write buffer stops CPU from stalling if memory cannot keep up
- Write buffer may have multiple entries to absorb bursts of writes
- What if store misses in cache?



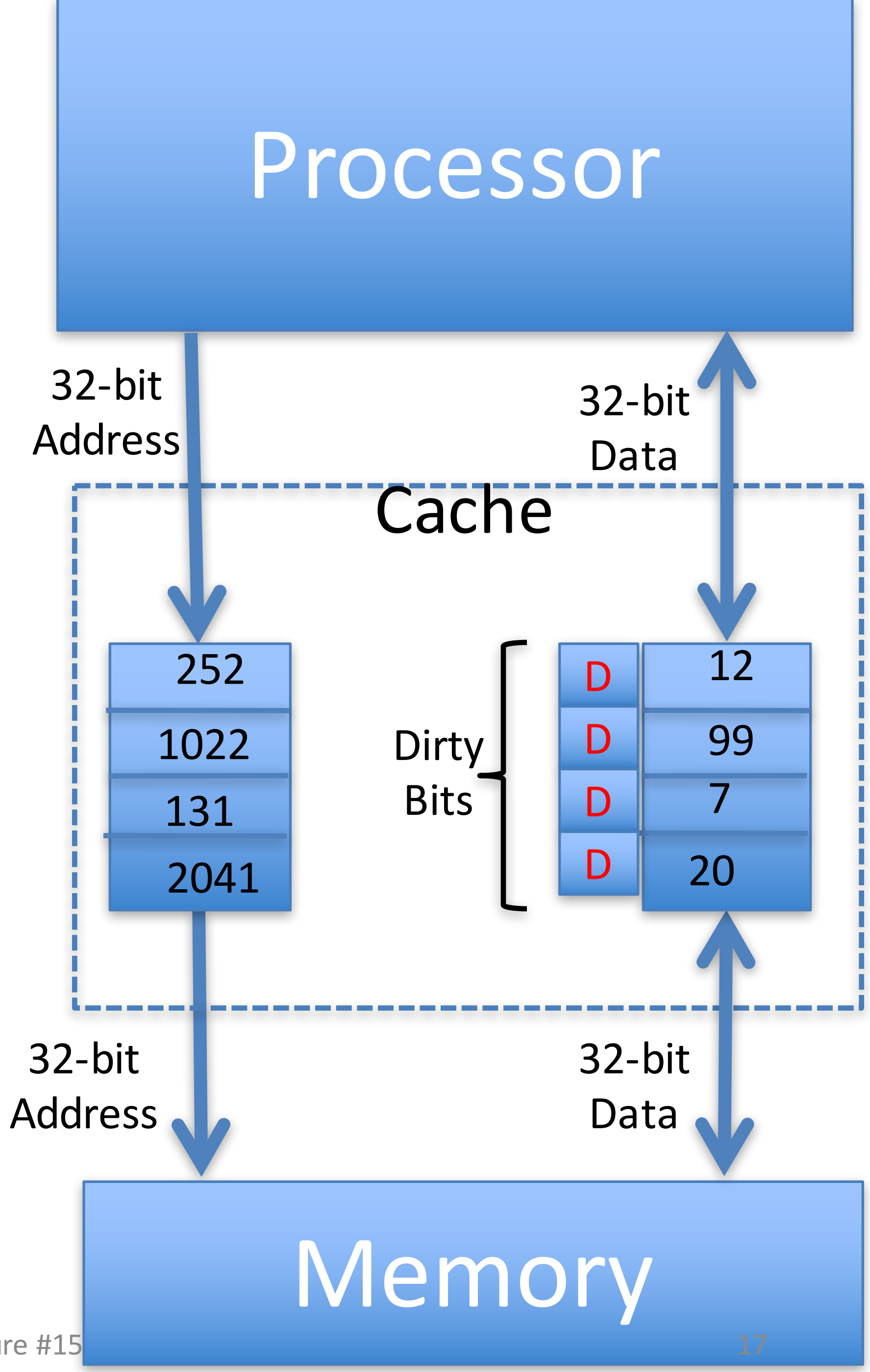
Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache

- Writes collected in cache, only single write to memory per block
- Include bit to see if wrote to block or not, and then only write back if bit is set
  - Called "Dirty" bit (writing makes it "dirty")

Write-Back Cache

- Store/cache hit, write data in cache *only* and set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - "Write-allocate" policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit



Write-Through vs. Write-Back

- Write-Through:
  - Simpler control logic
  - More predictable timing simplifies processor control logic
  - Easier to make reliable, since memory always has copy of data (big idea: Redundancy!)

- Write-Back
  - More complex control logic
  - More variable timing (0,1,2 memory accesses per cache access)
  - Usually reduces write traffic
  - Harder to make reliable, sometimes cache has only copy of data

Administrivia

- Midterm #2 2 weeks away! November 1!
  - In class! 3:40-5 PM
  - Synchronous digital design and Project 3 (processor design) included
  - Pipelines and Caches
  - ONE Double sided Crib sheet
  - Review Session, Sunday, 10/30, 1-3 PM, 10 Evans

iClicker Saga

iClicker and EPA

- No longer taking attendance in lecture – but we hope you will continue to come anyway
- Continue to use Clicker questions in lecture to help you test your understanding
- E**P**A will be based on a "holistic" assessment of lecture, piazza, guerrilla and tutoring sessions, office hours, discussion, and lab participation
- EPA will be calculated so as to only help your course grade, never hurt it

Outline

- Cache Organization and Principles
- Write Back vs. Write Through
- Cache Performance
- Cache Design Tradeoffs
- And in Conclusion …

Cache (*Performance*) Terms

- Hit rate: fraction of accesses that hit in the cache
- Miss rate: 1 – Hit rate
- Miss penalty: time to replace a block from lower level in memory hierarchy to cache
- Hit time: time to access cache memory (including tag comparison)

- Abbreviation: "$" = cache (a Berkeley innovation!)

Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses in the cache

AMAT = Time for a hit + Miss rate × Miss penalty

Important Equation!

Clickers/Peer Instruction

AMAT = Time for a hit + Miss rate x Miss penalty

- Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

A: ≤200 psec

B: 400 psec

C: 600 psec

D: ≥ 800 psec

Clickers/Peer Instruction

AMAT = Time for a hit + Miss rate x Miss penalty

- Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

  A: ≤200 psec

  [✓ B: 400 psec]

  C: 600 psec

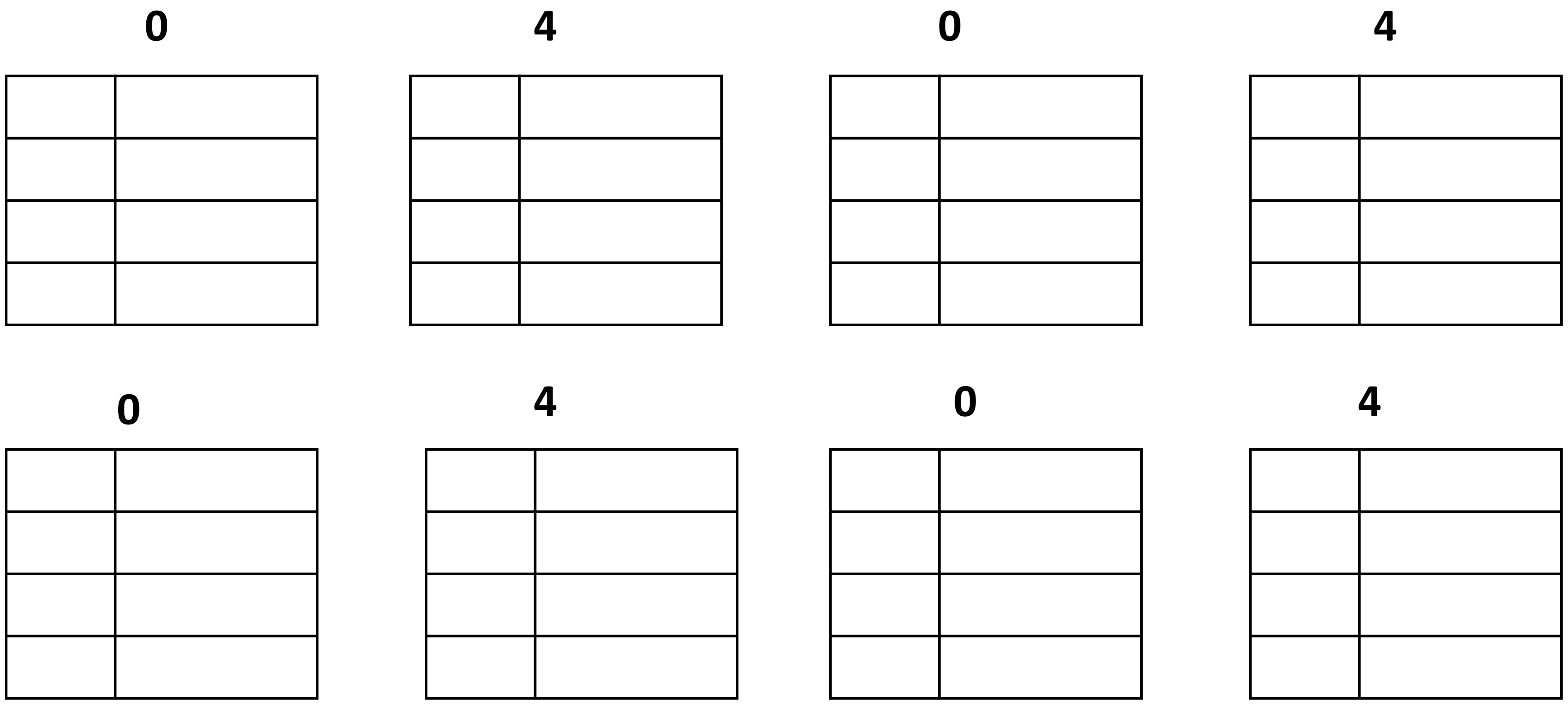
  D: ≥ 800 psec

1 clock cycle + .02 * 50 clock cycles = 2 clock cycles

Ping Pong Cache Example: Direct-Mapped Cache w/4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

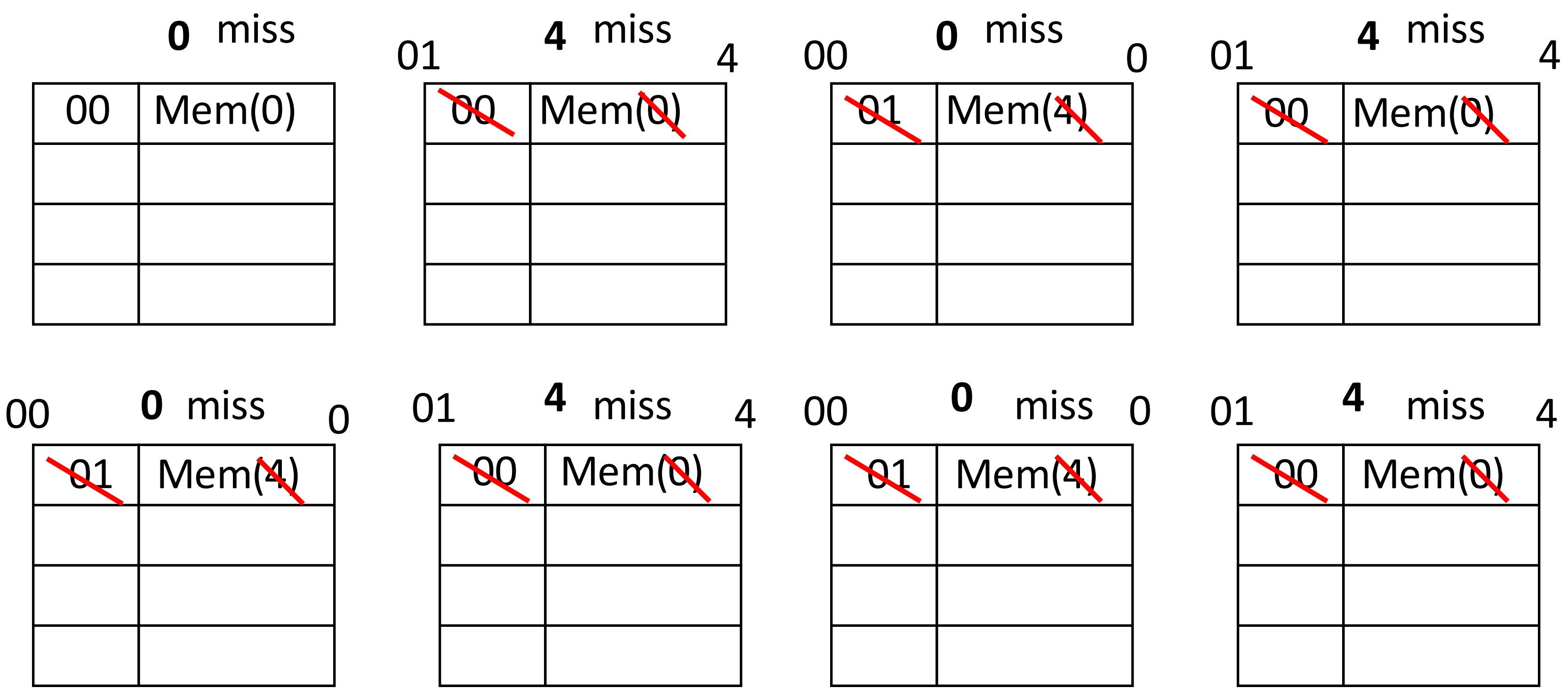
Start with an empty cache - all blocks initially marked as not valid



Ping Pong Cache Example: Direct-Mapped Cache w/4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid



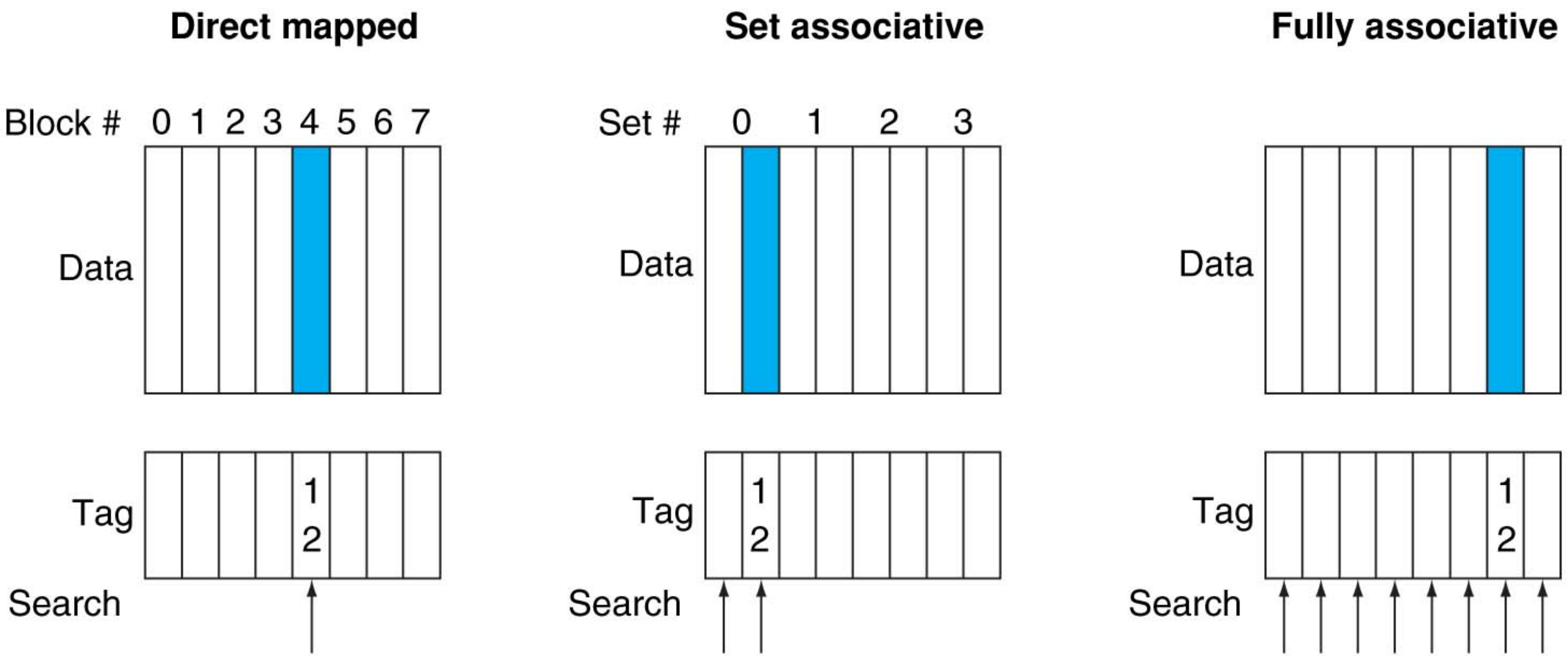
- 8 requests, 8 misses

Ping-pong effect due to conflict misses - two memory locations that map into the same cache block

Outline

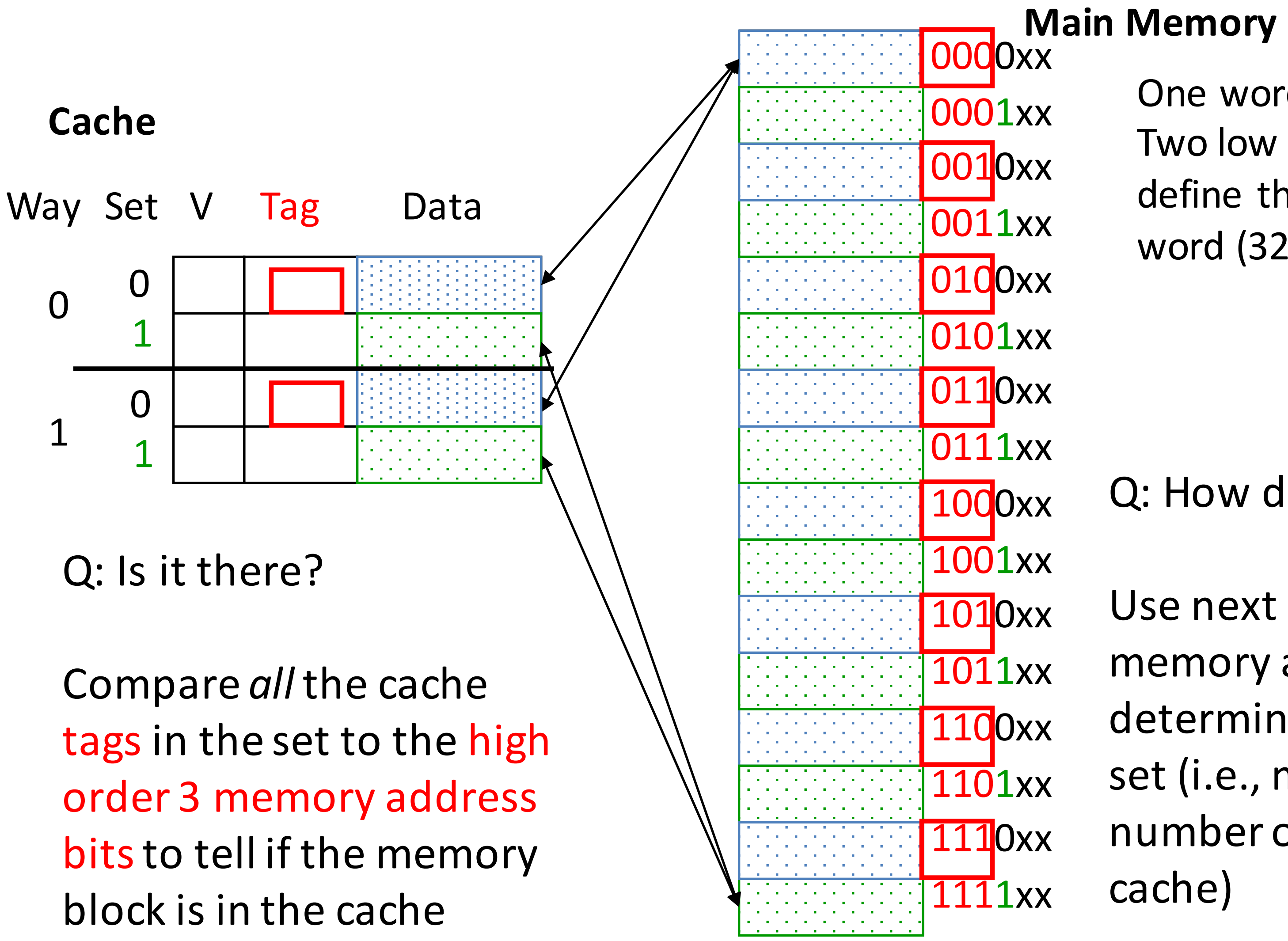
- Cache Organization and Principles
- Write Back vs. Write Through
- Cache Performance
- Cache Design Tradeoffs
- And in Conclusion …

Alternative Block Placement Schemes



- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- FA placement: mem block 12 can appear in any cache blocks

Example: 2-Way Set Associative $ (4 words = 2 sets x 2 ways per set)



One word blocks
Two low order bits define the byte in the word (32b words)

Q: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Q: Is it there?

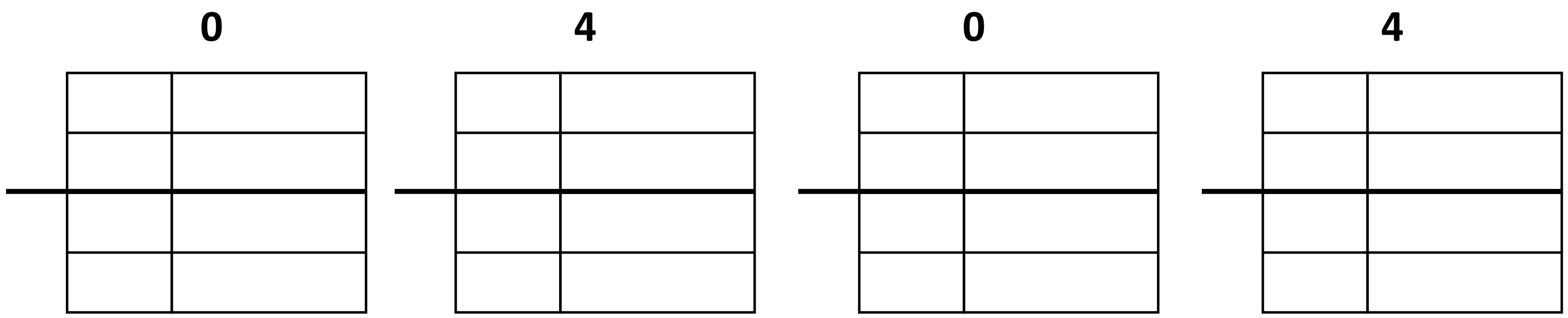
Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

Ping Pong Cache Example: 4 Word 2-Way SA $, Same Reference String

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

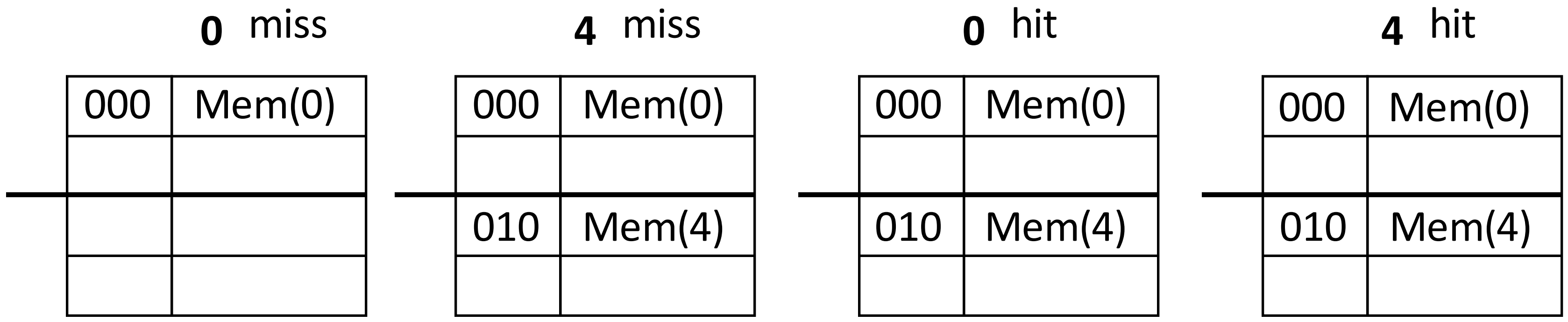


Ping Pong Cache Example: 4-Word 2-Way SA $, Same Reference String

- Consider the main memory address reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

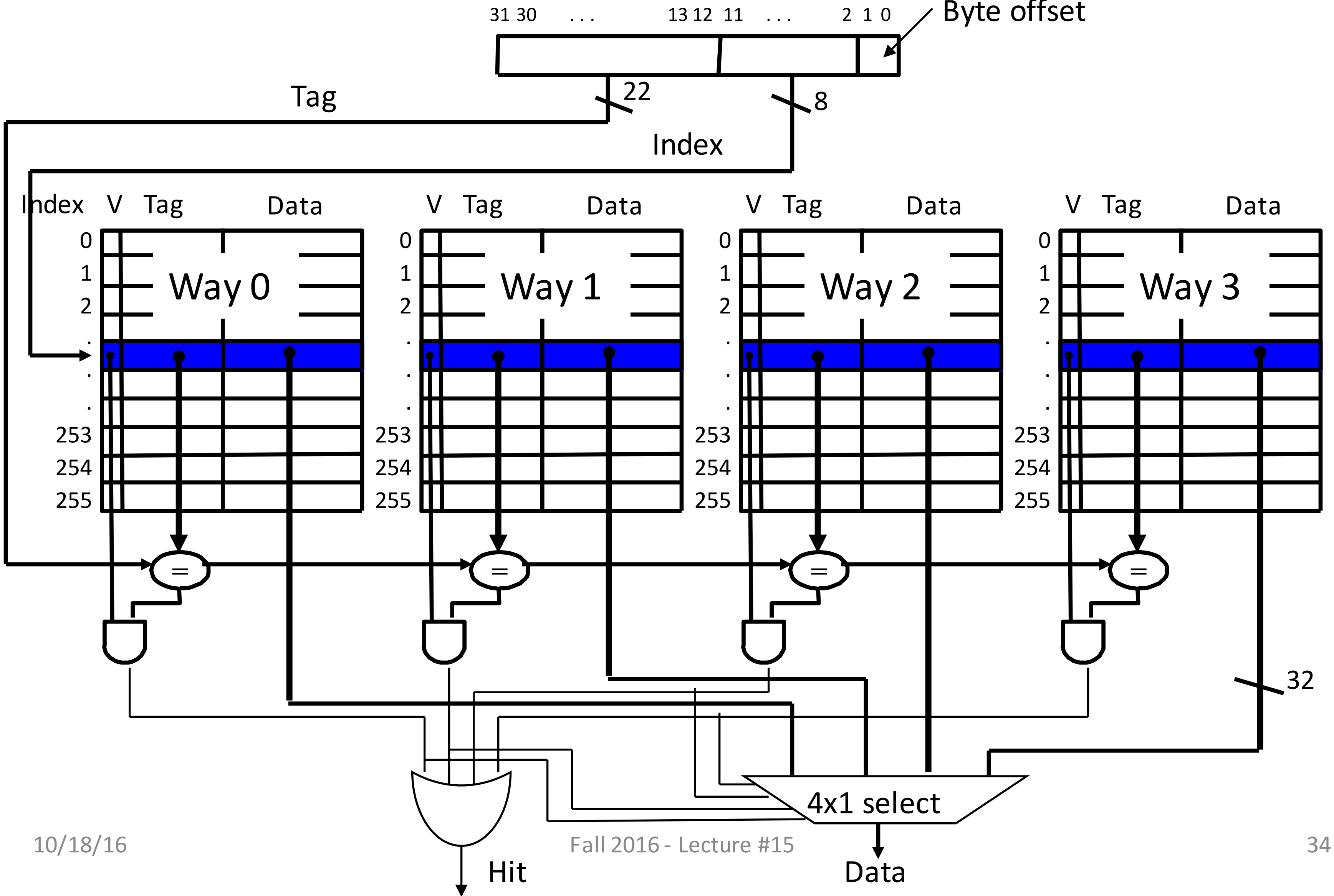


  - 8 requests, 2 misses

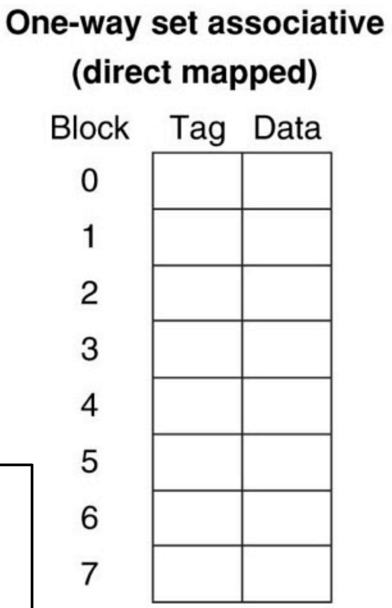
- Solves the ping-pong effect in a direct-mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

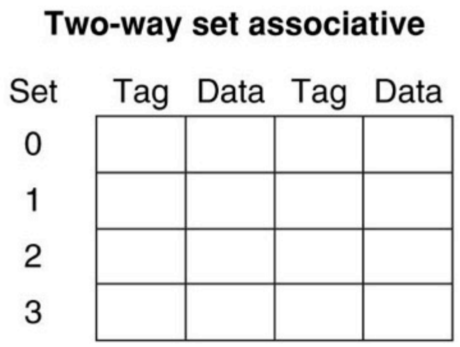
Four-Way Set-Associative Cache

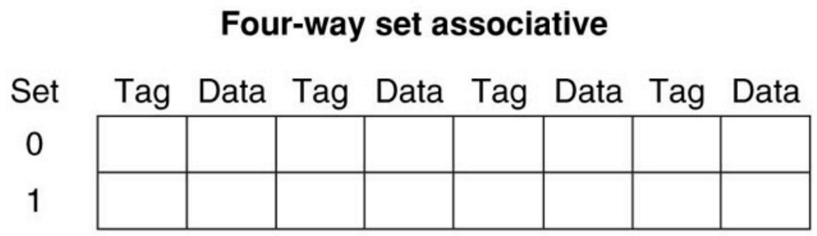
- 2^8 = 256 sets each with four ways (each with one block)

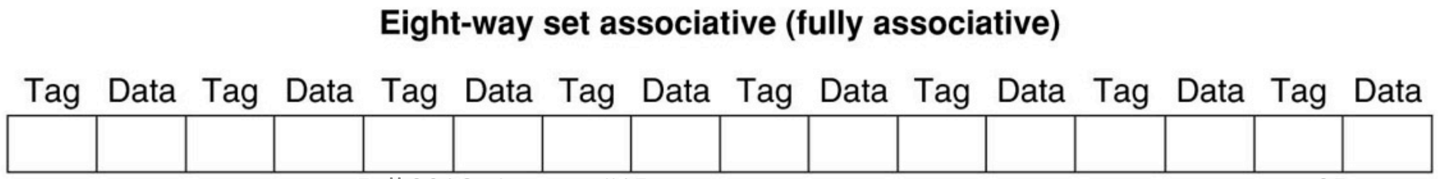


Alternative Organizations of an Eight-Block Cache









Total size of $ in blocks is equal to *number of sets* × *associativity*. For fixed $ size and fixed block size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $ is same as a fully associative $.

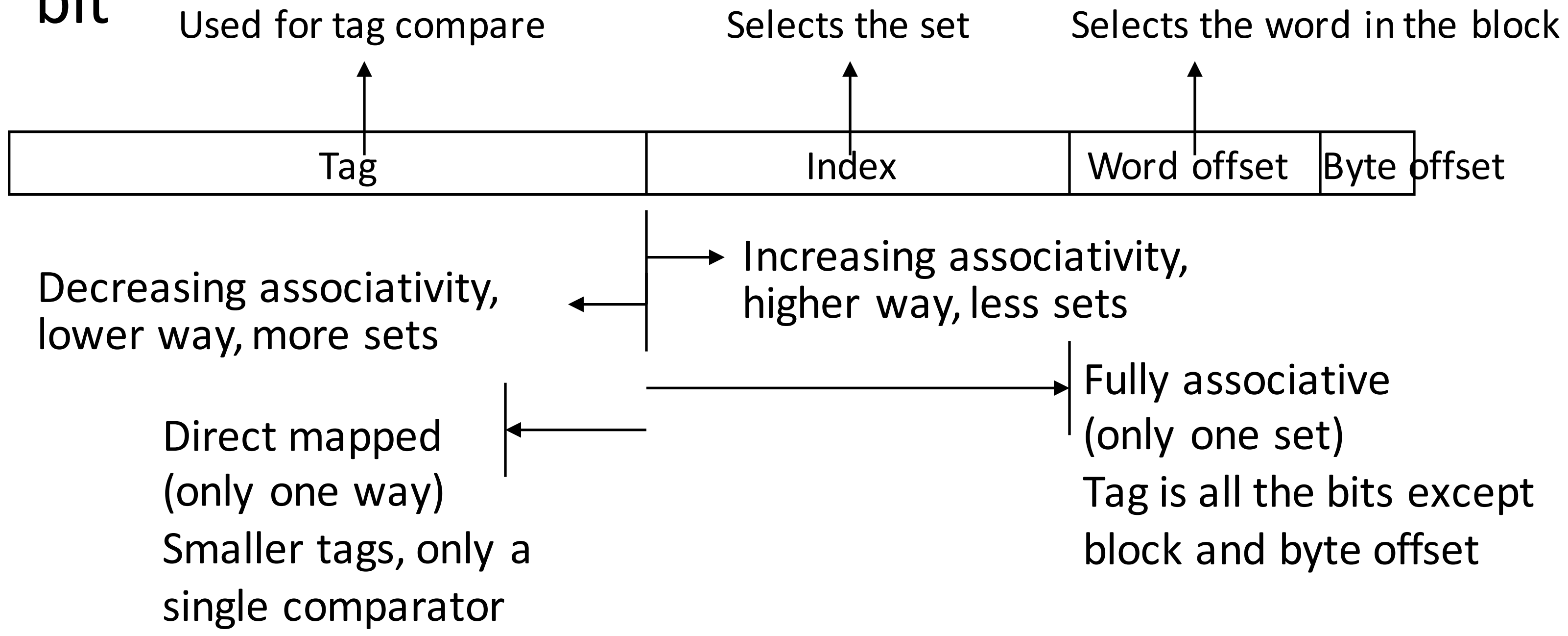
Range of Set-Associative Caches

- For a *fixed-size* cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

Tag	Index	Word offset	Byte offset

Range of Set-Associative Caches

- For a *fixed-size* cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit


Used for tag compare
Selects the set
Selects the word in the block
Tag
Index
Word offset
Byte offset
Decreasing associativity, lower way, more sets
Increasing associativity, higher way, less sets
Direct mapped (only one way)
Smaller tags, only a single comparator
Fully associative (only one set)
Tag is all the bits except block and byte offset

Total Cache Capacity =

Associativity × # of sets × block_size

Bytes = blocks/set × sets × Bytes/block

$$C = N \times S \times B$$

Tag	*Index*	*Byte Offset*

address_size = tag_size + index_size + offset_size

= tag_size + $\log_2(S)$ + $\log_2(B)$

Total Cache Capacity =

Associativity * # of sets * block_size

*Bytes = blocks/set * sets * Bytes/block*

$C = N * S * B$

Tag	*Index*	*Byte Offset*

address_size = tag_size + index_size + offset_size

= tag_size + $\log_2(S)$ + $\log_2(B)$

Double the Associativity: Number of sets?

tag_size? index_size? # comparators?

Double the Sets: Associativity?

tag_size? index_size? # comparators?

Total Cache Capacity =

Associativity * # of sets * block_size

*Bytes = blocks/set * sets * Bytes/block*

$C = N * S * B$

Tag	*Index*	*Byte Offset*

address_size = tag_size + index_size + offset_size

= tag_size + $\log_2(S)$ + $\log_2(B)$

Double the Associativity: Halve the number of sets

tag_size + 1 while index_size – 1, 2 x comparators

Double the Sets: Halve the associativity

tag_size - 1 while index_size + 1, ½ x comparators

Your Turn

- For a cache of 64 blocks, each block four bytes in size:

1. The capacity of the cache is: 256 bytes.
2. Given a 2-way Set Associative organization, there are 32 sets, each of 2 blocks, and 2 places a block from memory could be placed.
3. Given a 4-way Set Associative organization, there are 16 sets each of 4 blocks and 4 places a block from memory could be placed.
4. Given an 8-way Set Associative organization, there are 8 sets each of 8 blocks and 8 places a block from memory could be placed.

Clicker/Peer Instruction

- For S sets, N ways, B blocks, which statements hold?
  (i) The cache has B tags
  (ii) The cache needs N comparators
  (iii) B = N x S
  (iv) Size of Index = $\text{Log}_2(S)$

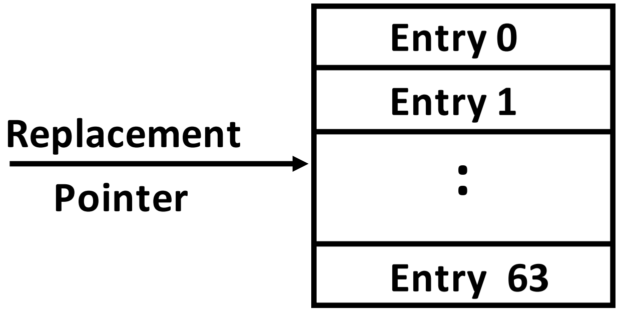
  A: (i) only
  B: (i) and (ii) only
  C: (i), (ii), (iii) only
  $\boxed{\text{D: All four statements are true}}$
  E: None are true

Costs of Set-Associative Caches

- N-way set-associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection (and Hit/Miss decision). DM $: block is available before the Hit/Miss decision
    - In Set-Associative, not possible to just assume a hit and continue and recover later if it was a miss
- When miss occurs, which way's block selected for replacement?
  - Least Recently Used (LRU): one that has been unused the longest (principle of temporal locality)
    - Must track when each way's block was used relative to other blocks in the set
    - For 2-way SA $, one bit per set → set to 1 when a block is referenced; reset the other way's bit (i.e., "last used")

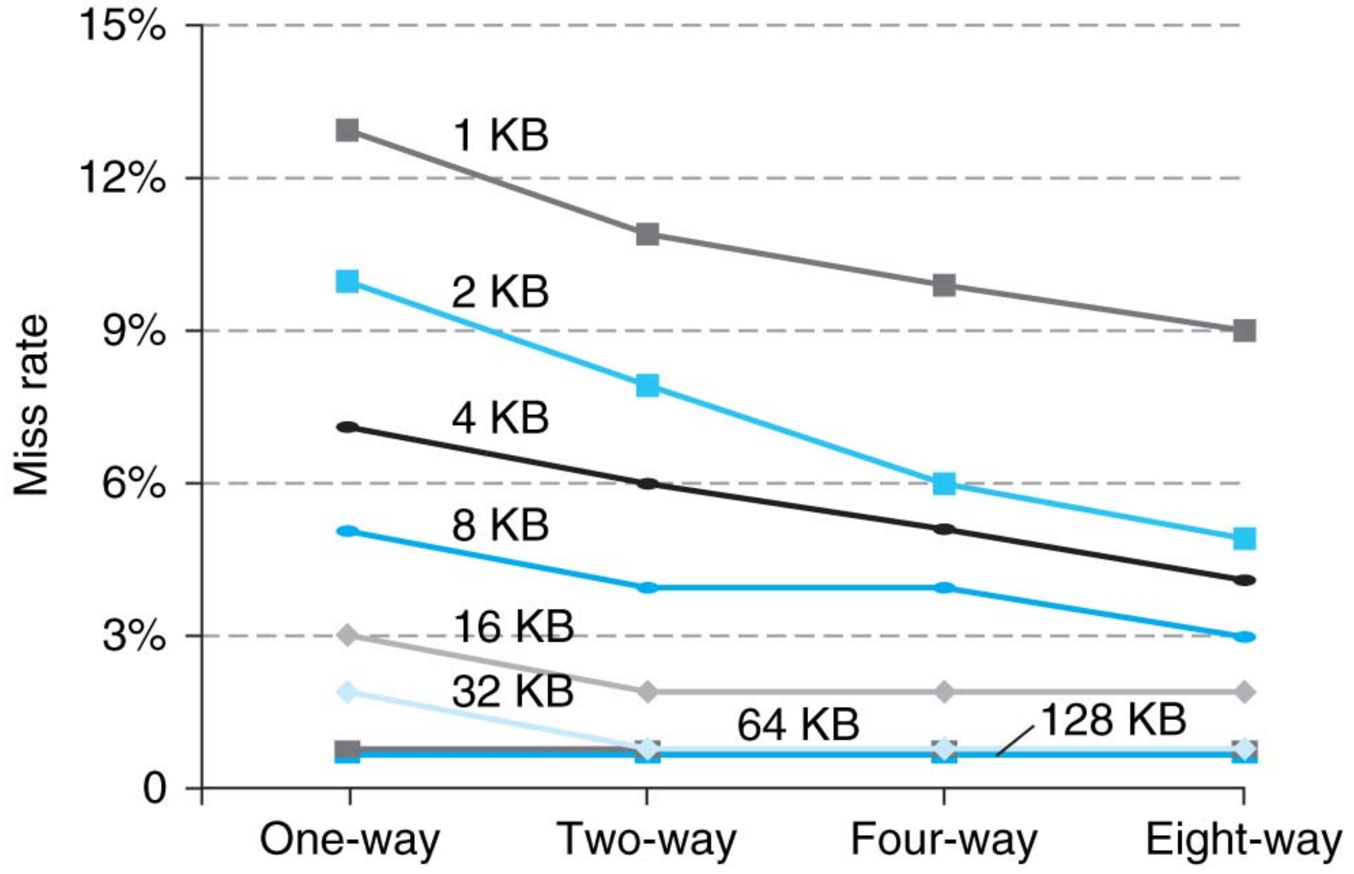
Cache Replacement Policies

- Random Replacement
  - Hardware randomly selects a cache evict
- Least-Recently Used
  - Hardware keeps track of access history
  - Replace the entry that has not been used for the longest time
  - For 2-way set-associative cache, need one bit for LRU replacement
- Example of a Simple "Pseudo" LRU Implementation
  - Assume 64 Fully Associative entries
  - Hardware replacement pointer points to one cache entry
  - Whenever access is made to the entry the pointer points to:
    - Move the pointer to the next entry
  - Otherwise: do not move the pointer
  - (example of "not-most-recently used" replacement policy)



Benefits of Set-Associative Caches

- Choice of DM $ versus SA $ depends on the cost of a miss versus the cost of implementation



- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Outline

- Cache Organization and Principles
- Write Back vs. Write Through
- Cache Performance
- Cache Design Tradeoffs
- And in Conclusion …

And in Conclusion …

- Name of the Game: Reduce AMAT
  - Reduce Hit Time
  - Reduce Miss Rate
  - Reduce Miss Penalty
- Balance cache parameters (Capacity, associativity, block size)