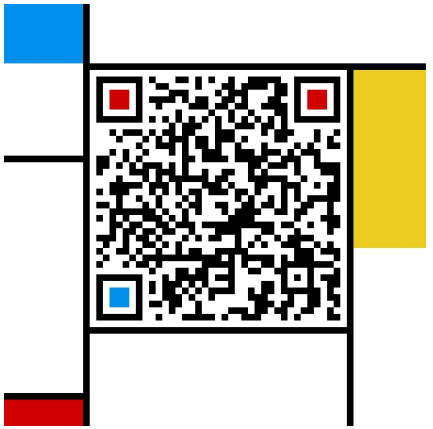
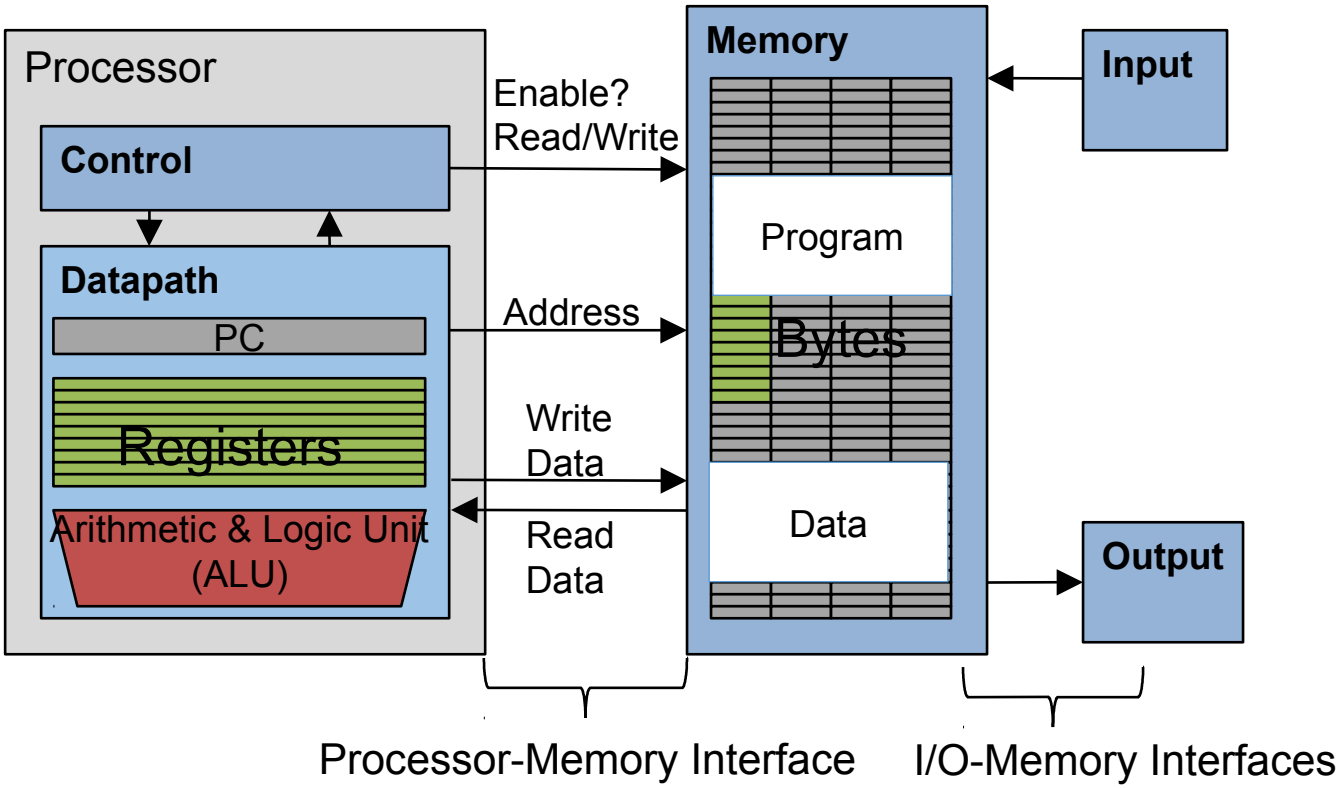
计算机组成与系统结构
Computer Organization & System Architecture

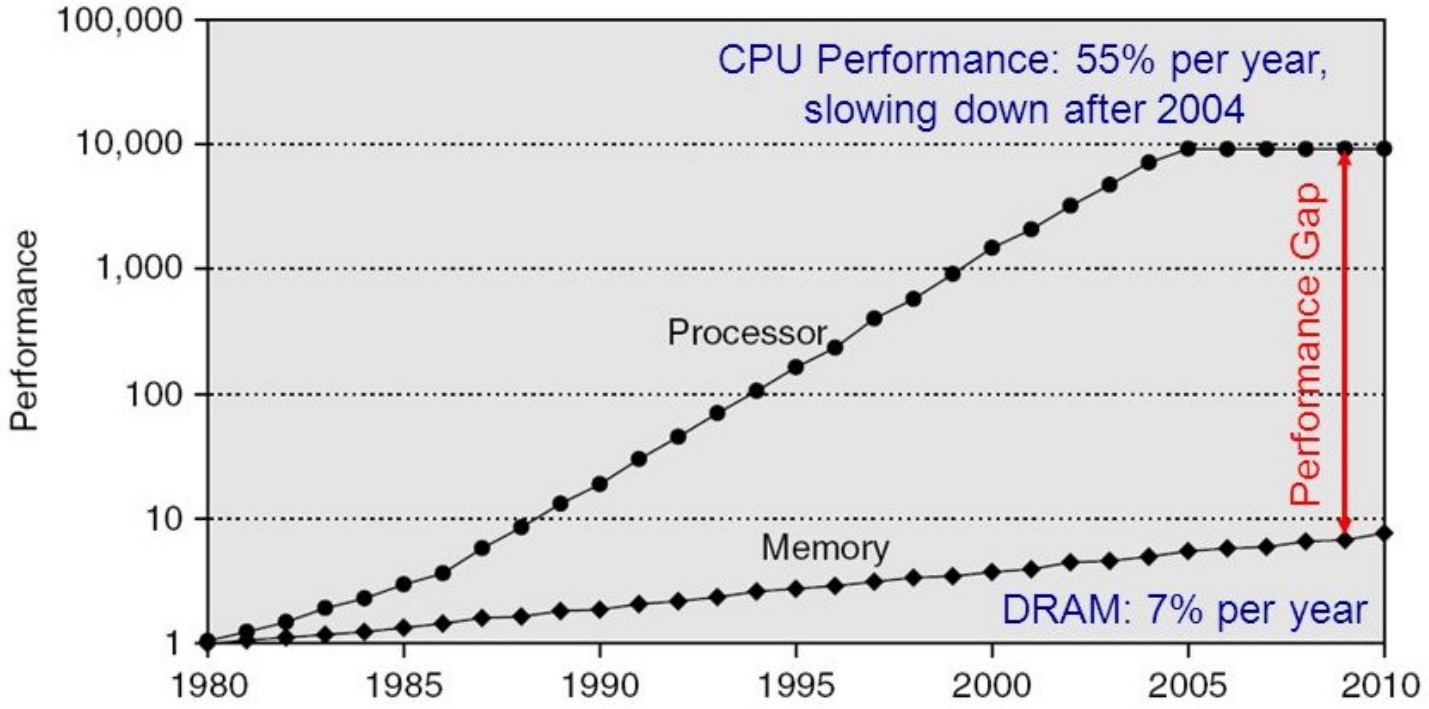
Huang Kejie (黄科杰) 百人计划研究员
Office: 玉泉校区老生仪楼 304
Email address: huangkejie@zju.edu.cn
HP: 17706443800



Components of a Computer



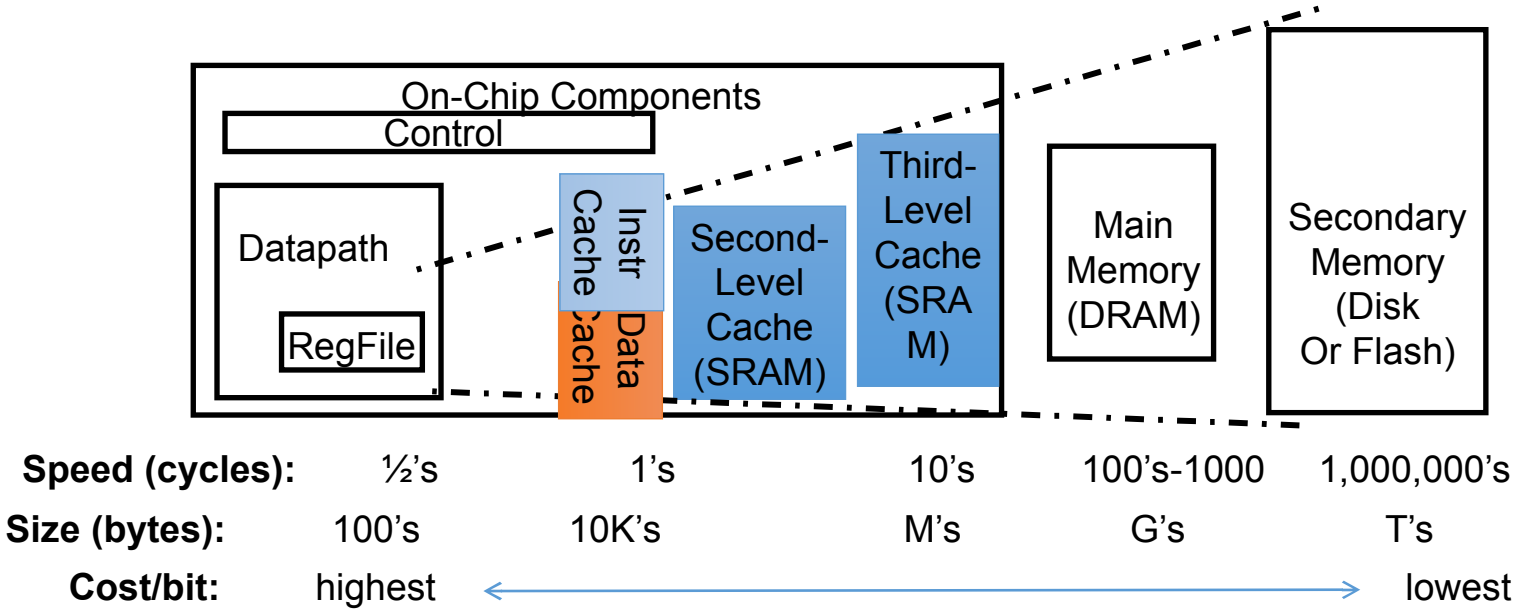
Processor-DRAM Gap (Latency)



1980 microprocessor executes **~one instruction** in same time as DRAM access
2017 microprocessor executes **~1000 instructions** in same time as DRAM access

Slow DRAM access has disastrous impact on CPU performance!

Typical Memory Hierarchy



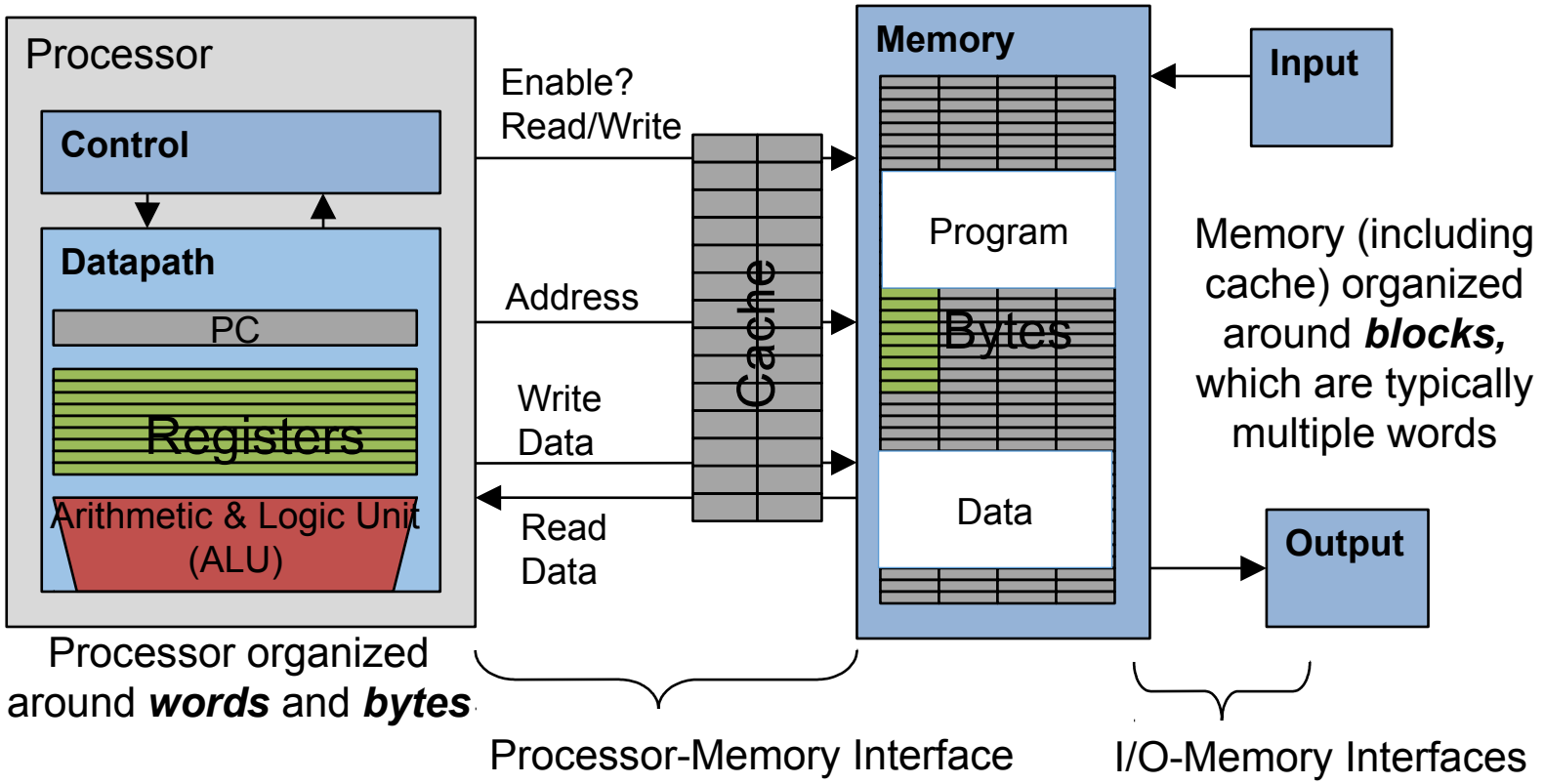
- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

Big Idea: Locality

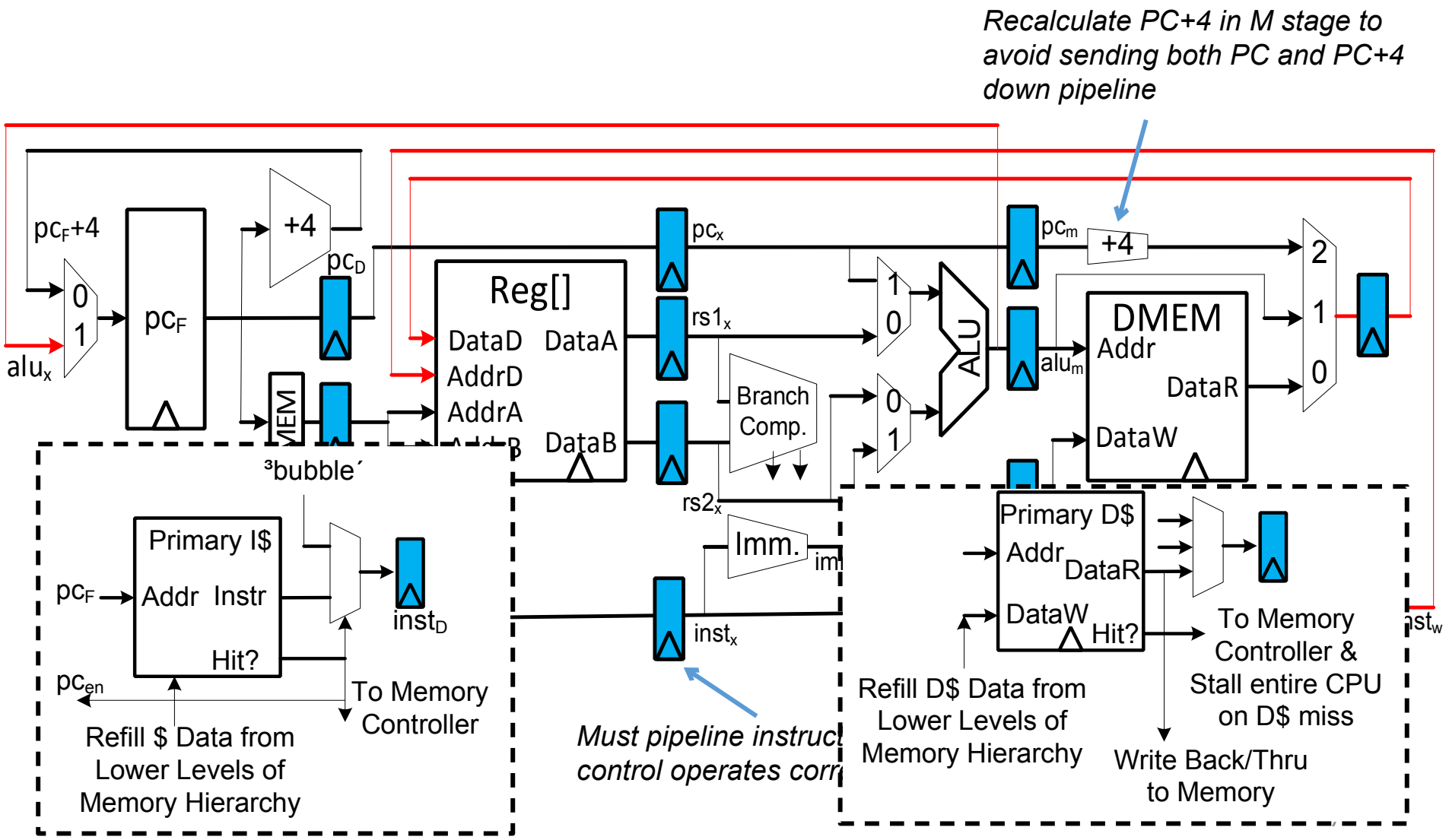
- *Temporal Locality* (locality in time)
  - Go back to same book on desk multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon

- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books on J.D. Salinger since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

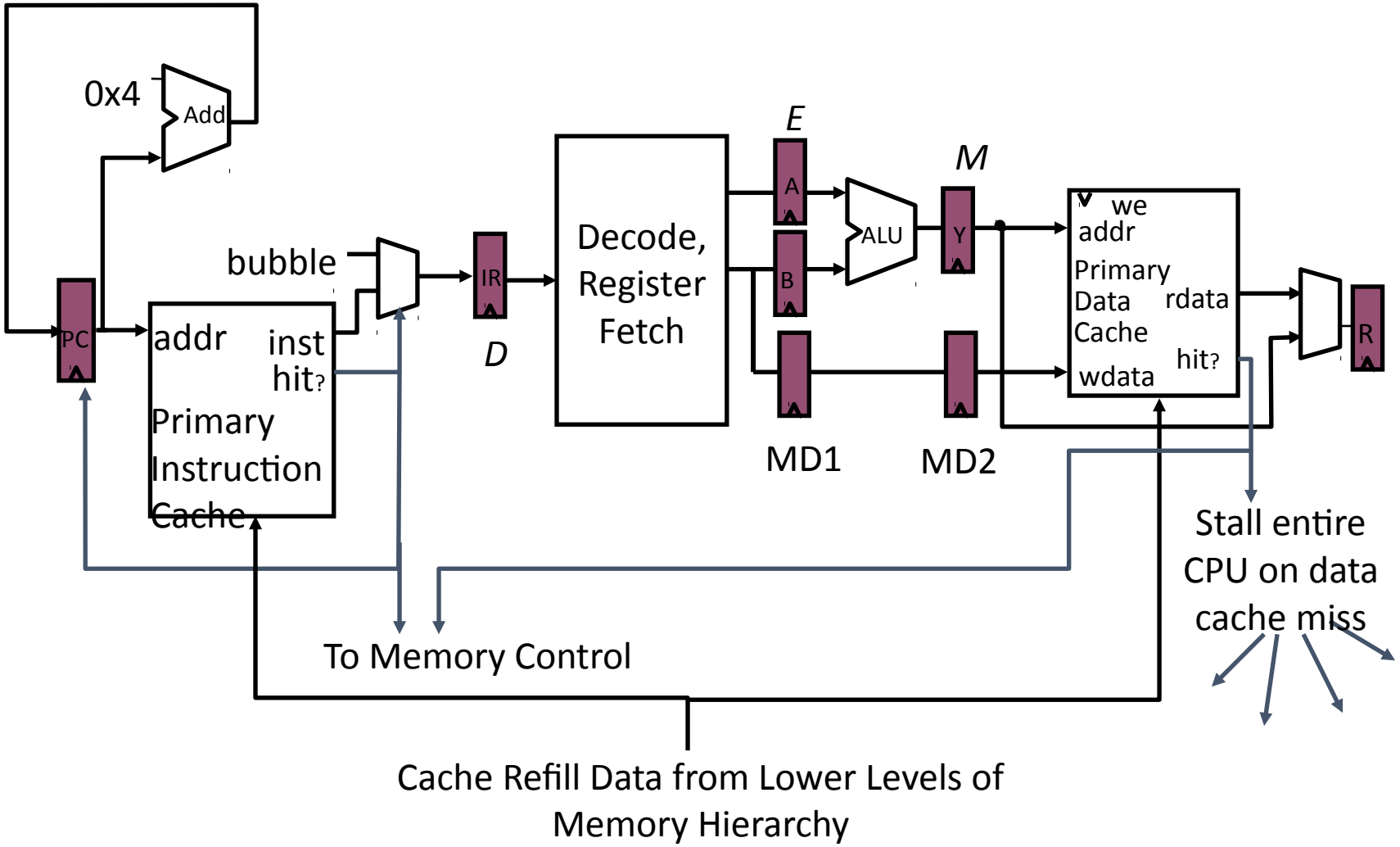
Adding Cache to Computer



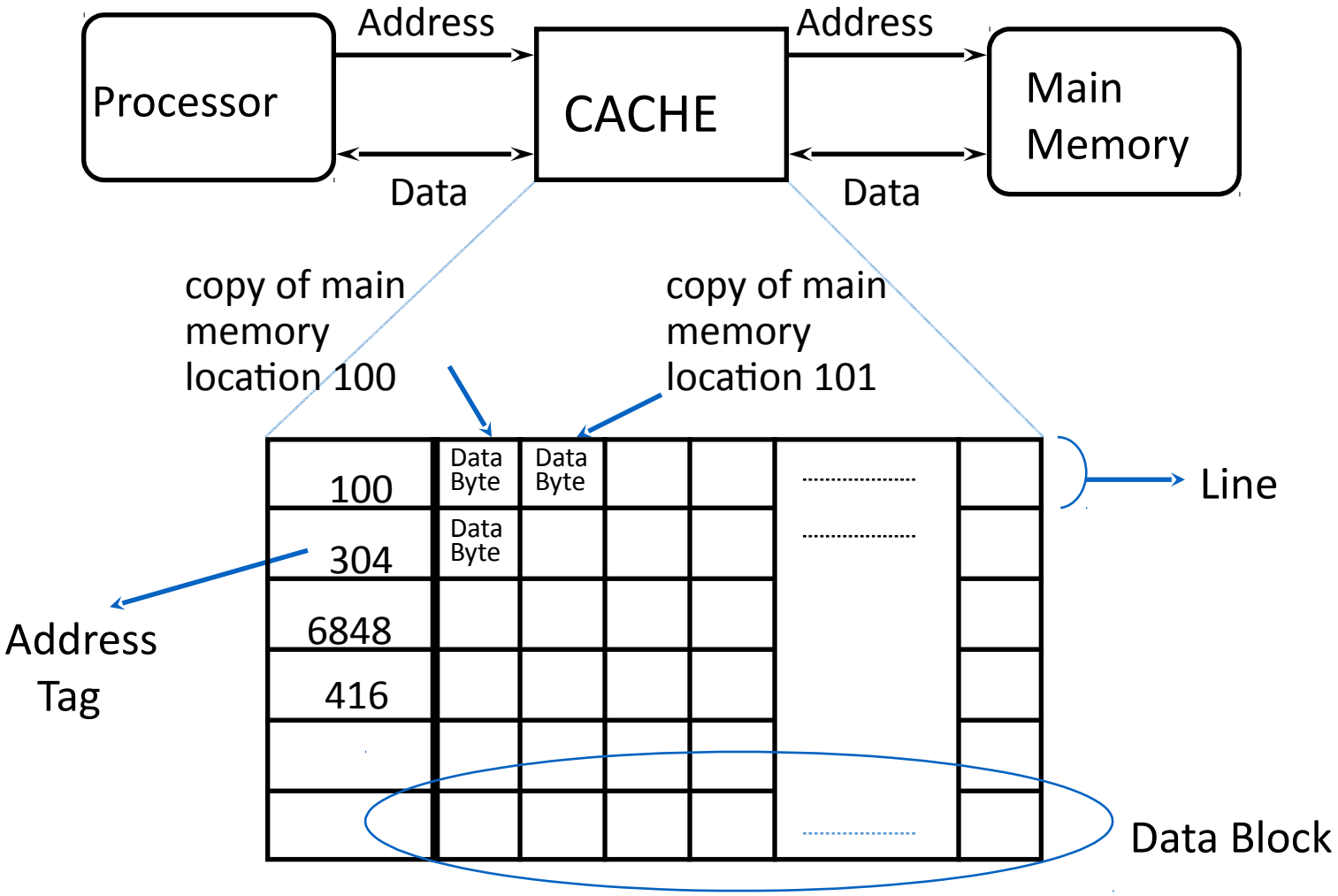
Pipelined RISC-V RV32I Datapath (with I,D$)



CPU-Cache Interaction (5-stage pipeline)



Inside a Cache



Cache Terminology

- **hit**: data appears in some block
  - **hit rate**: the fraction of accesses found in the level
  - **hit time**: time to access the level (consists of RAM access time + time to determine hit)

- **miss**: data needs to be retrieved from a block in the lower level (e.g., block Y)
  - **miss rate** = 1 - (hit rate)
  - **miss penalty**: time to replace a block in the upper level + time to deliver the block to the processor

- hit time << miss penalty

- 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!

- miss penalty: time to fetch a block from lower memory level
  - access time: function of latency
  - transfer time: function of bandwidth b/w levels
    - transfer one “cache line/block” at a time
    - transfer at the size of the memory-bus width

Peer Instruction

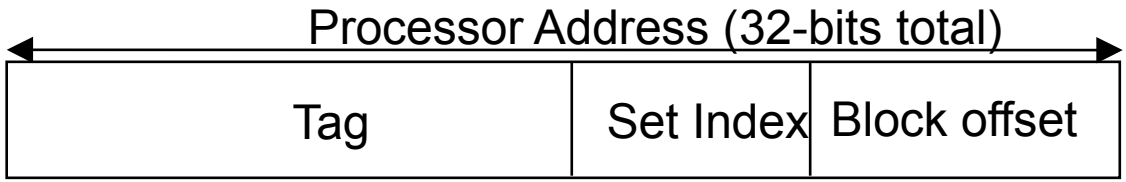
AMAT = Time for a hit + Miss rate × Miss penalty

- 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

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)

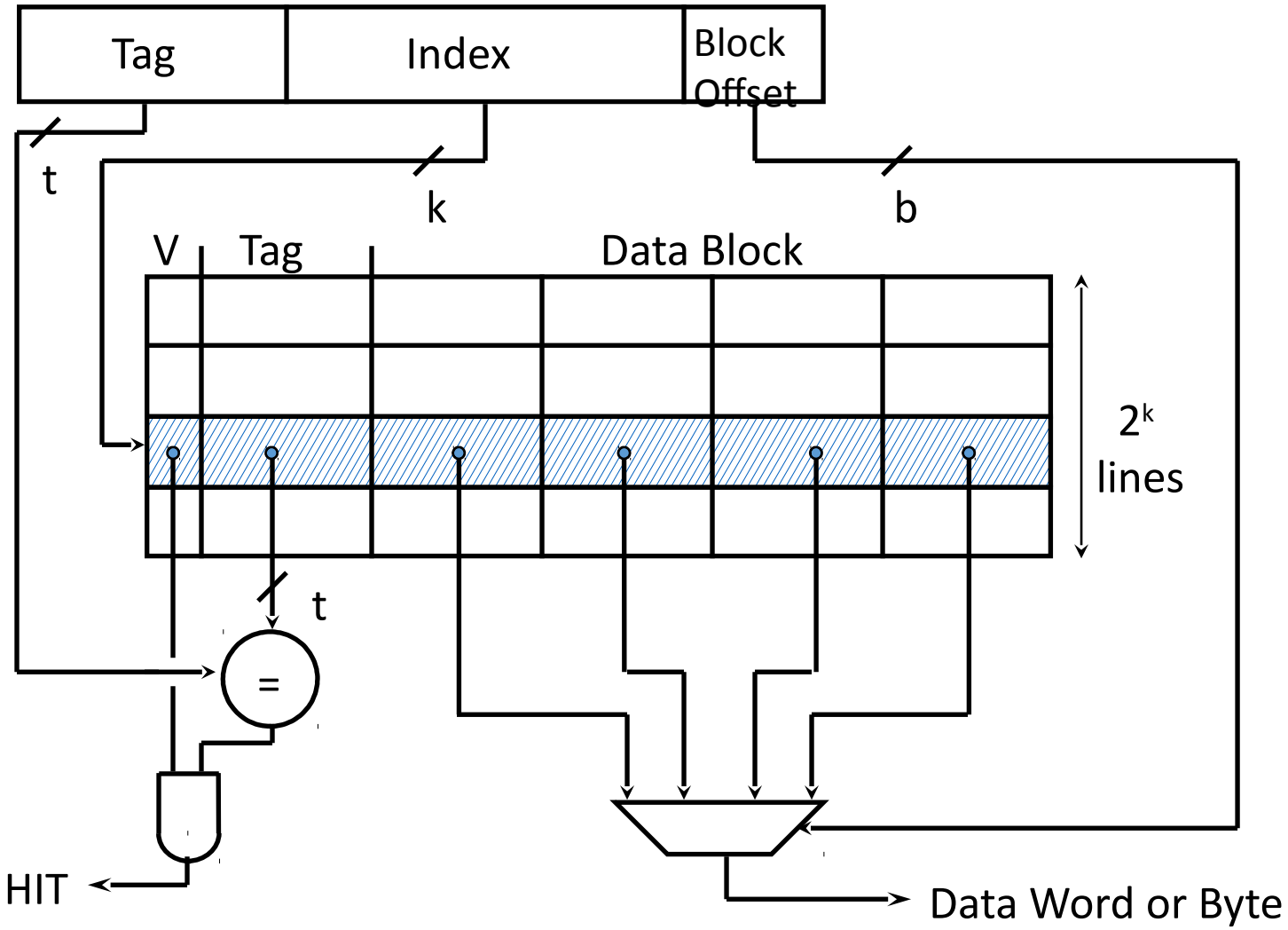
What Limits Number of Sets?

- For a given total number of blocks, we save comparators if have more than two sets

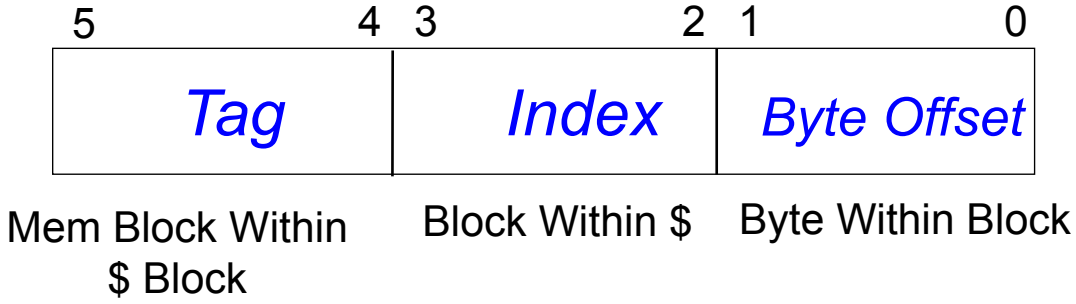
- Limit: As Many Sets as Cache Blocks => only one block per set – only needs one comparator!

- Called “Direct-Mapped” Design

Direct-Mapped Cache



Direct Mapped Cache Example: Mapping a 6-bit Memory Address



- In example, block size is 4 bytes/1 word
- Memory and cache blocks always the same size, unit of transfer between memory and cache
- # Memory blocks >> # Cache blocks
  - 16 Memory blocks = 16 words = 64 bytes => 6 bits to address all bytes
  - 4 Cache blocks, 4 bytes (1 word) per block
  - 4 Memory blocks map to each cache block
- Memory block to cache block, aka index: middle two bits
- Which memory block is in a given cache block, aka tag: top two bits

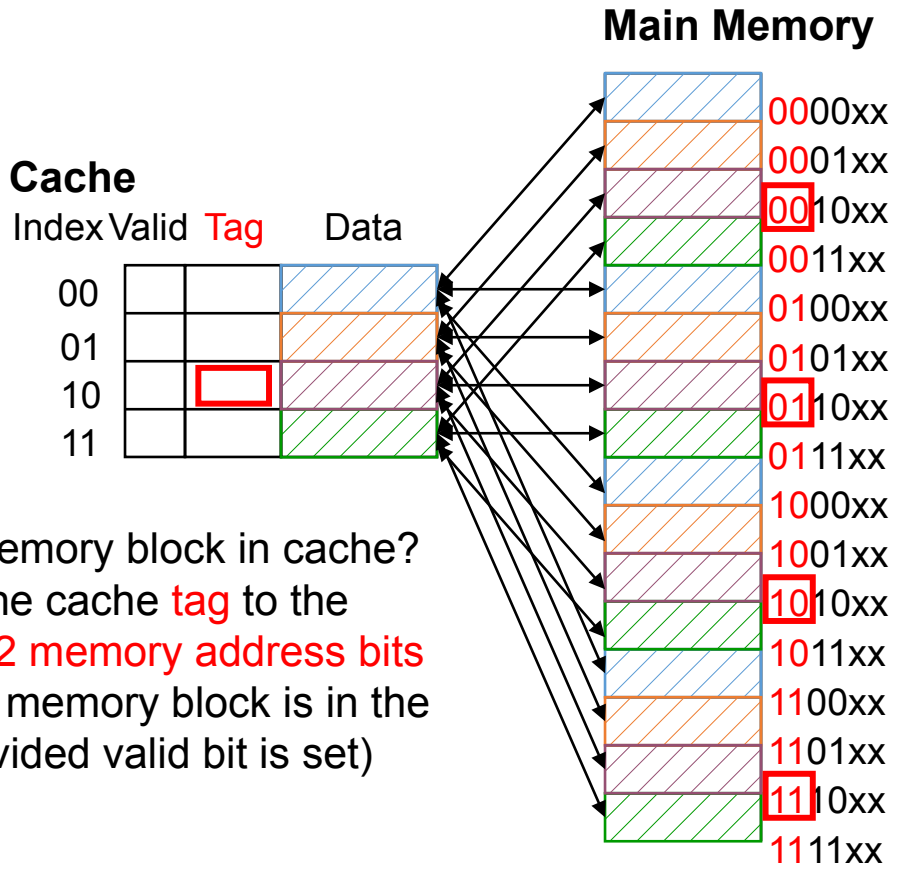
One More Detail: Valid Bit

- When start a new program, cache does not have valid information for this program

- Need an indicator whether this tag entry is valid for this program

- Add a “valid bit” to the cache tag entry
  - 0 => cache miss, even if by chance, address = tag
  - 1 => cache hit, if processor address = tag

Cache Organization: Simple First Example



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

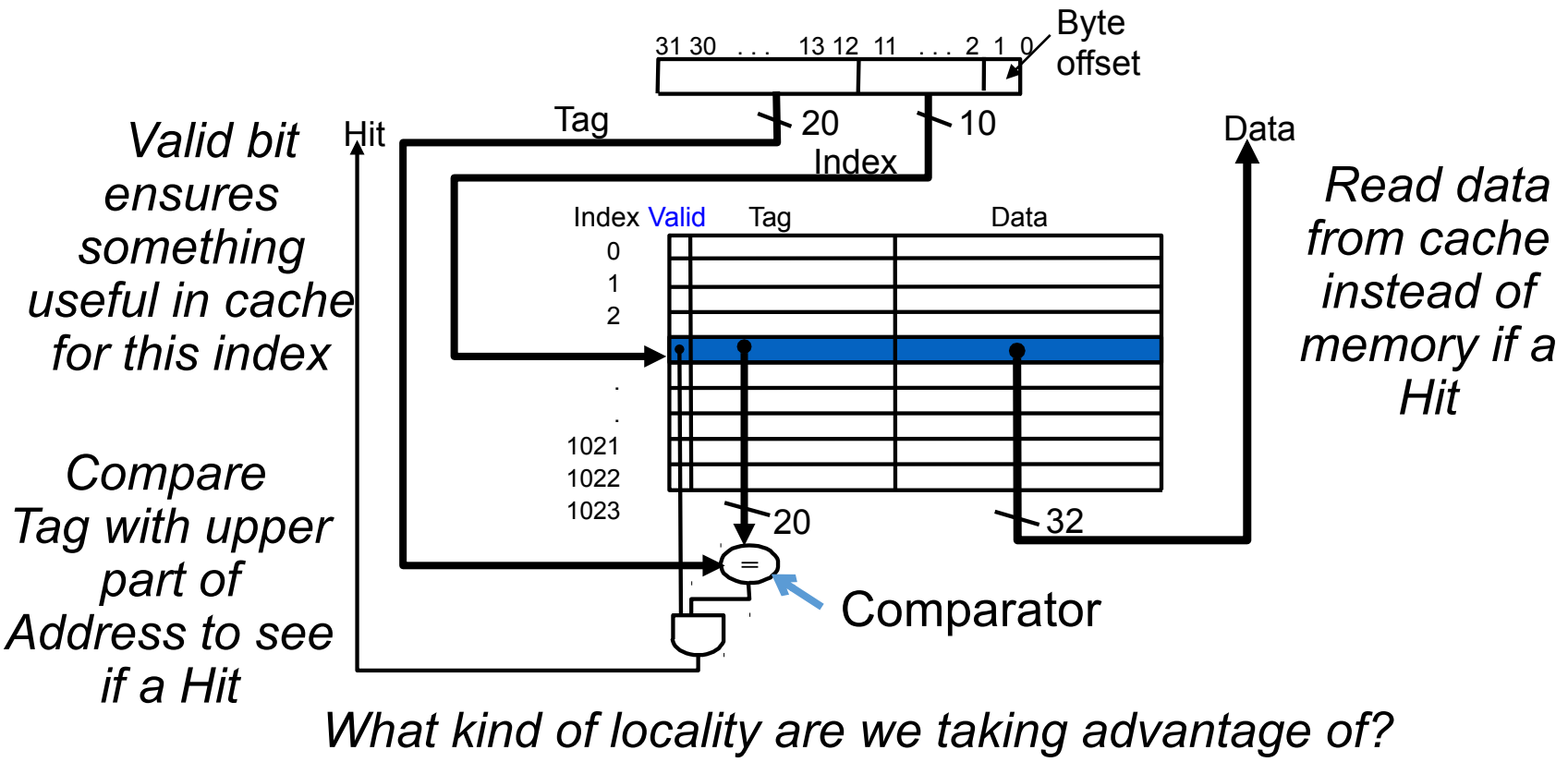
Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

Q: Where in the cache is the mem block?

Use next 2 low-order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

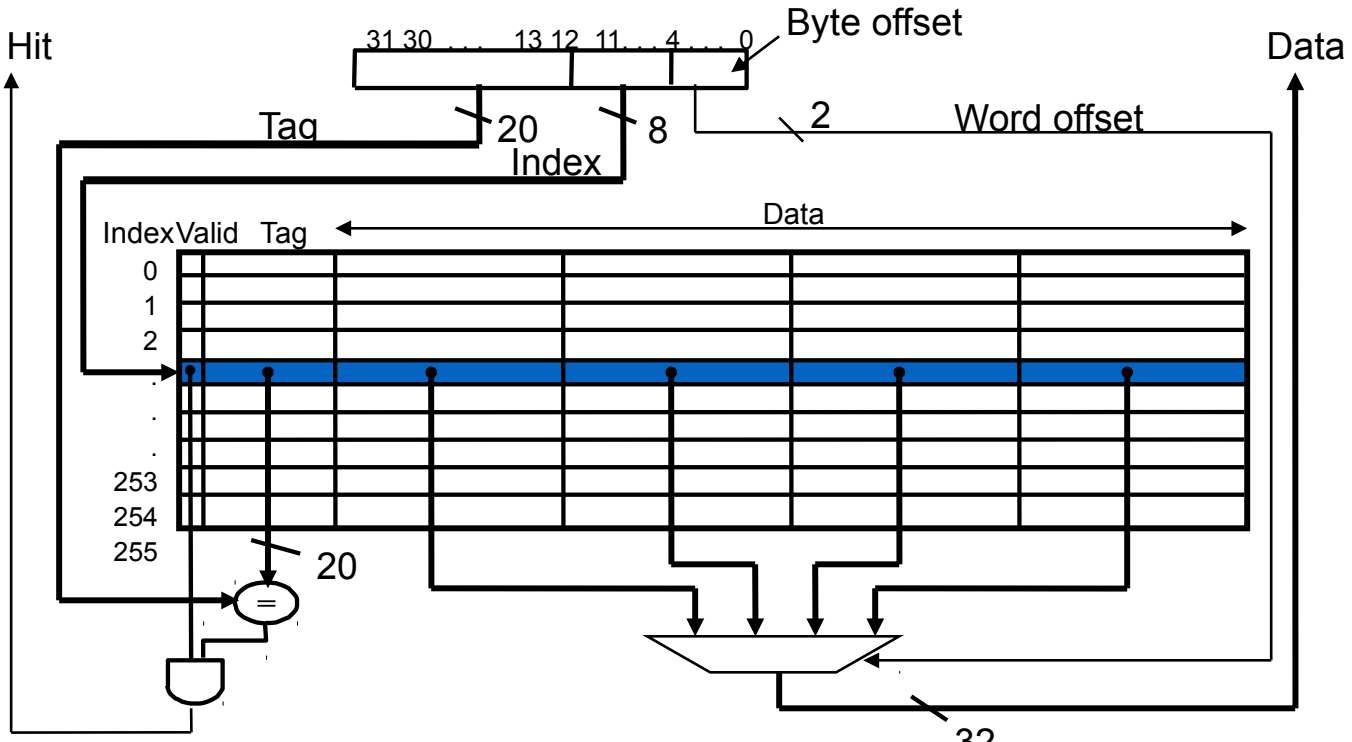
Direct-Mapped Cache Example

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



Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 1K words



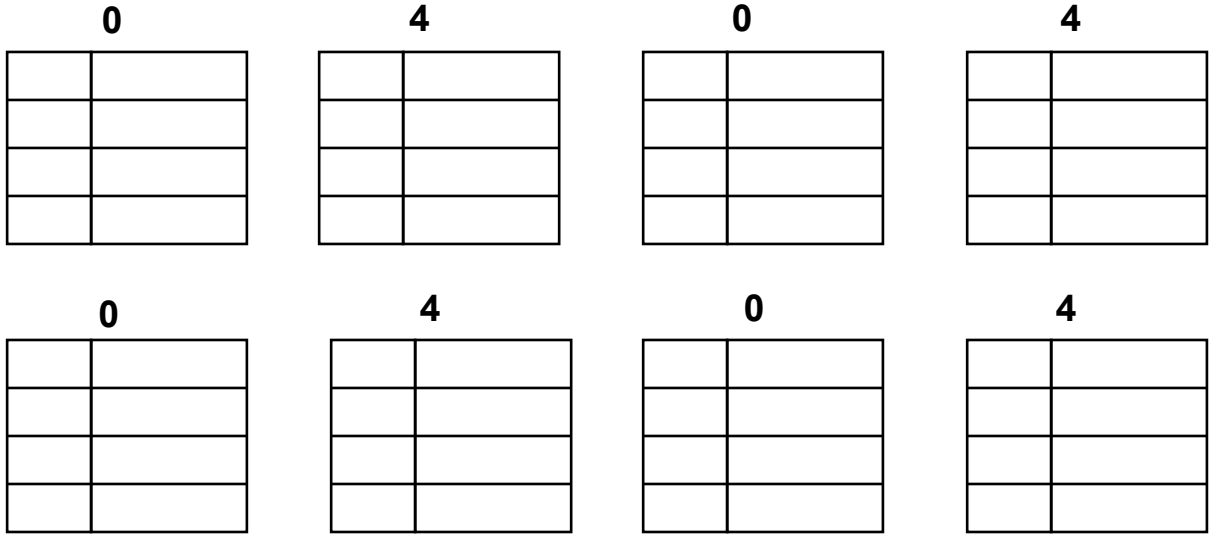
What kind of locality are we taking advantage of?

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

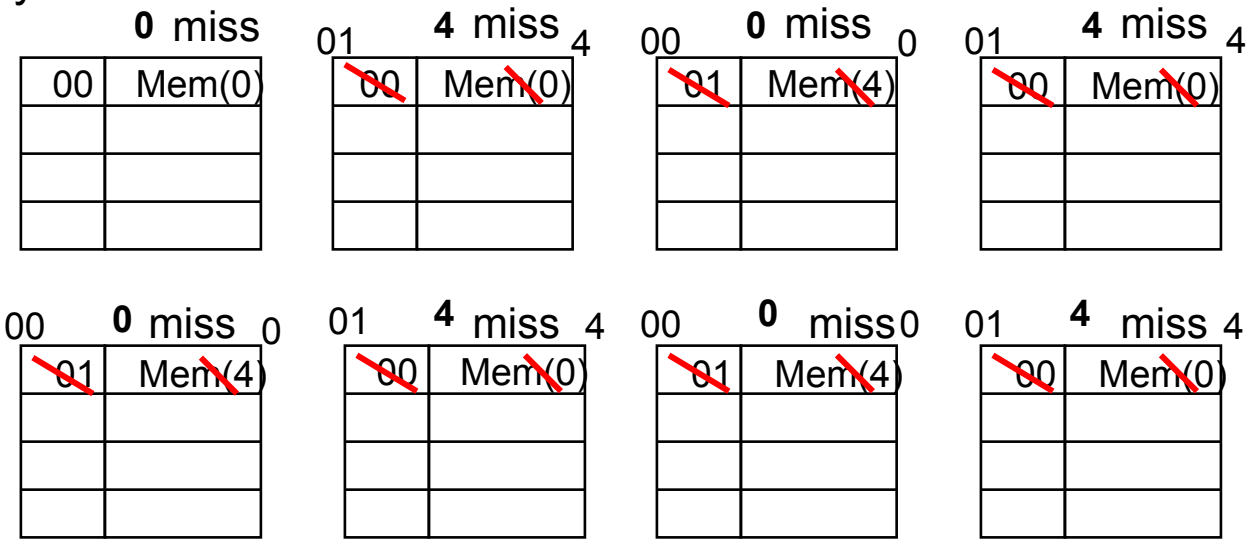


Ping Pong Cache Example: Direct-Mapped Cache w/4 Single-Word Blocks, Worst-Case Reference String

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

0 4 0 4 0 4 0 4

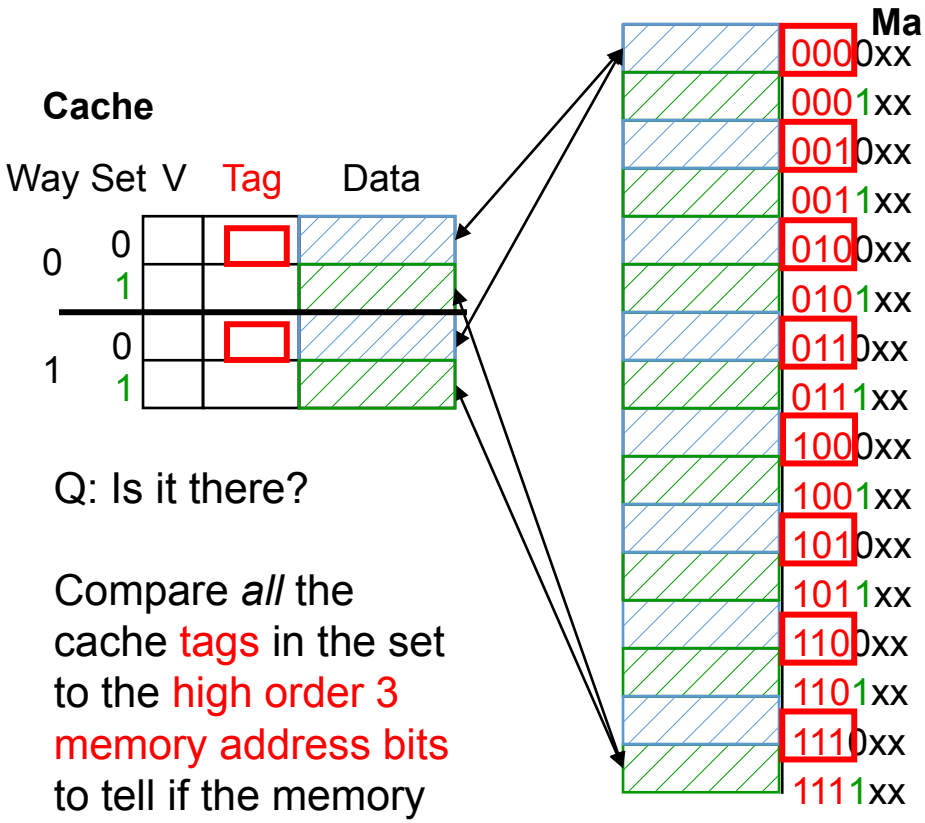
Start with an empty cache - all blocks initially marked as not valid



- 8 requests, 8 misses

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

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



Q: Is it there?

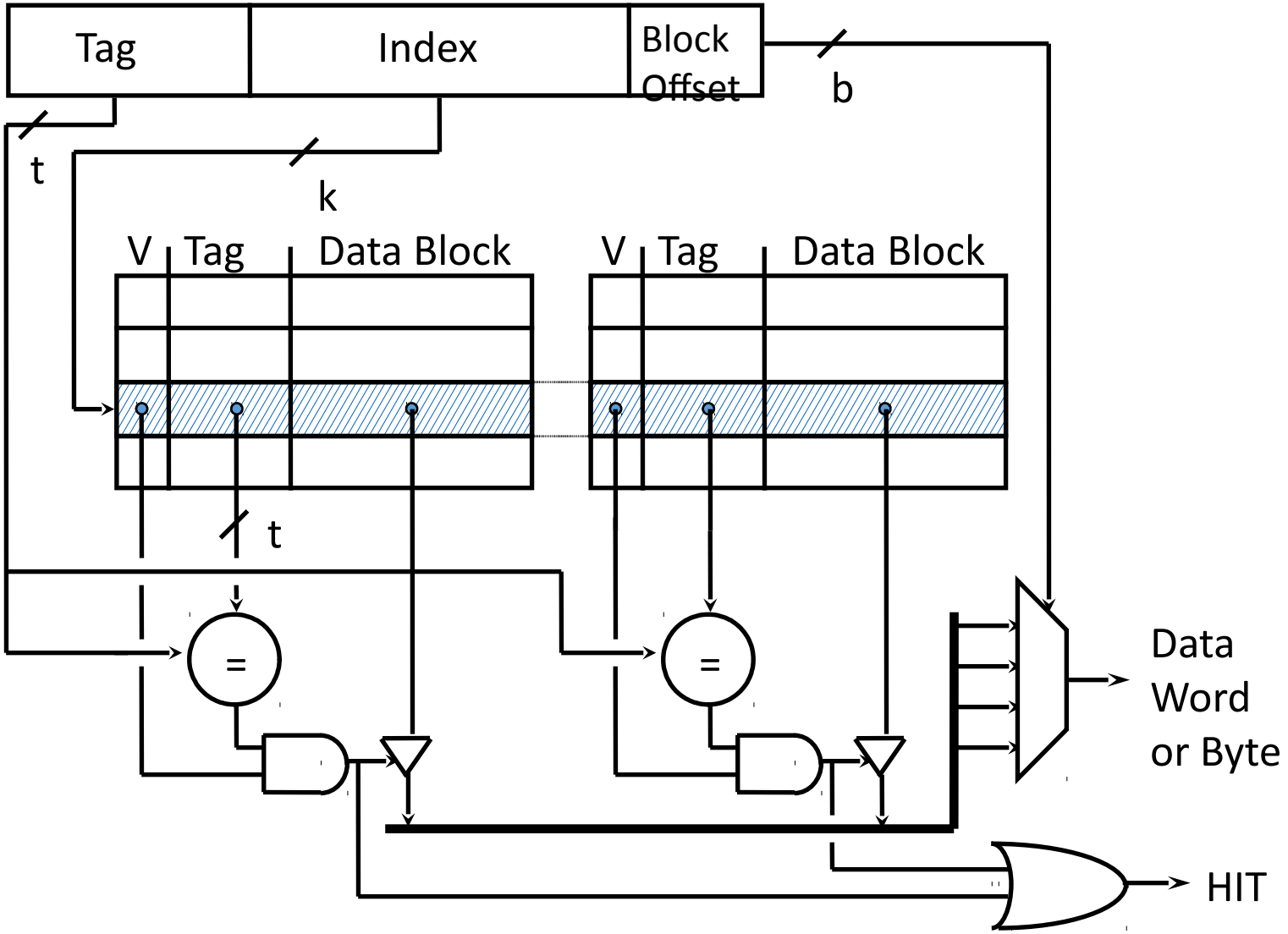
Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

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)

2-Way Set-Associative Cache

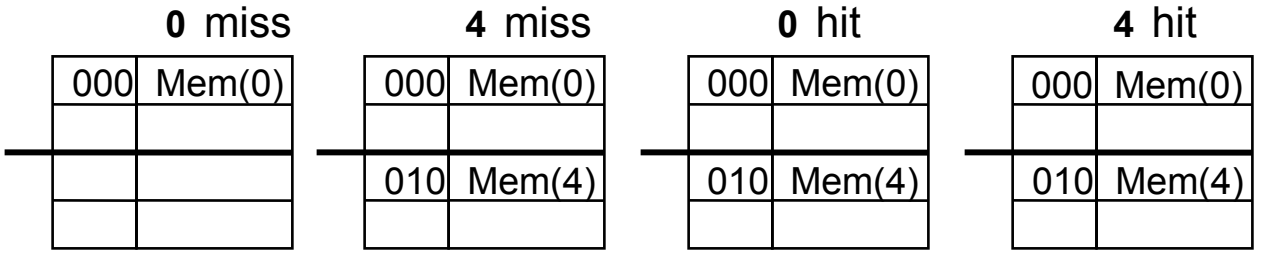


Ping Pong Cache Example: 4 Word 2-Way SA $, Same Reference String

- Consider the main memory word reference string

0 4 0 4 0 4 0 4

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

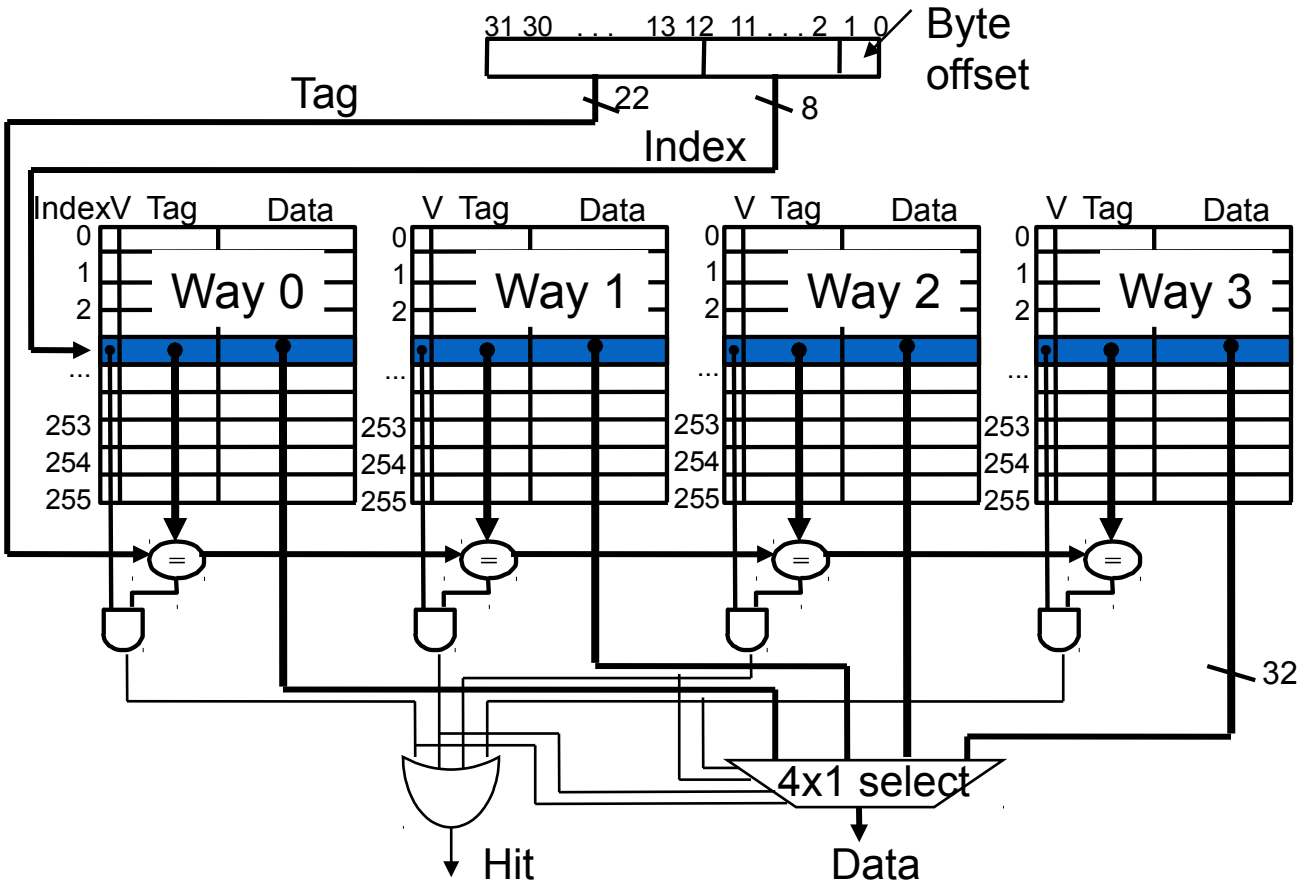


  - 8 requests, 2 misses

- 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 Cache Organizations

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

Types of Caches

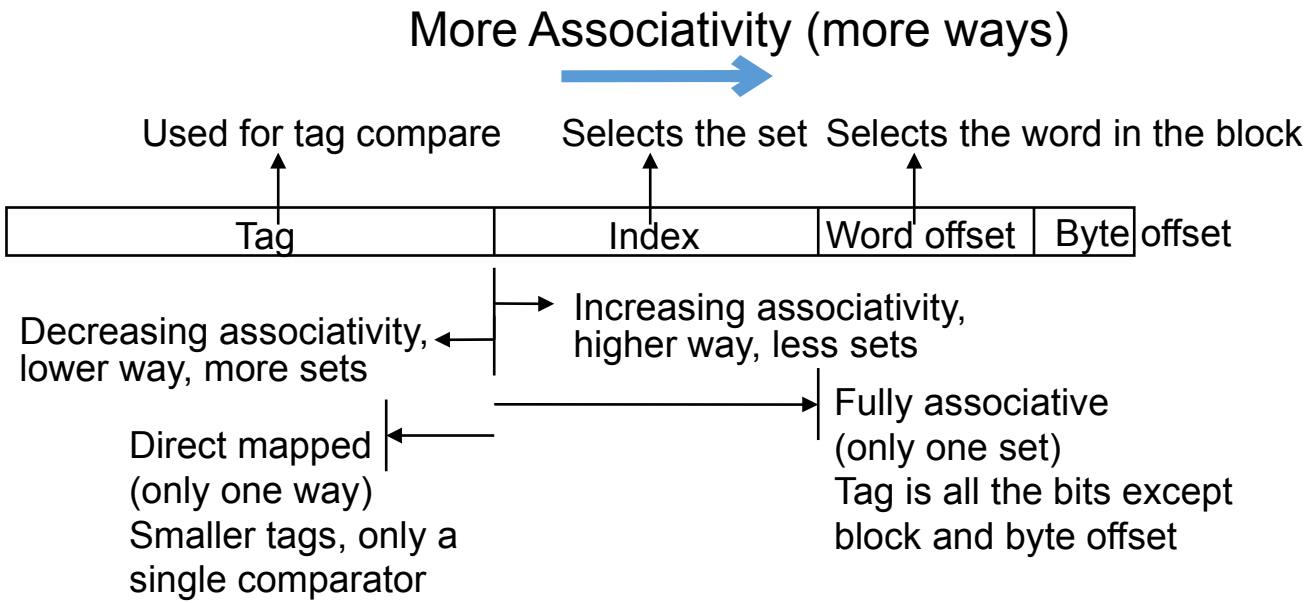
type of cache	mapping of data from memory to cache	complexity of searching the cache
Direct mapped (DM)	a memory value can be placed at **a single corresponding location** in the cache	fast indexing mechanism
set-associative (SA)	a memory value can be placed in **any of a set of locations** in the cache	slightly more involved search mechanism
fully-associative (FA)	a memory value can be placed in **any location** in the cache	extensive hardware resources required to search (CAM)

Describing Caches: 7 Parameters

- access time: T_{hit}
- capacity
  - total amount of data the cache can hold
    - # of blocks × block size
- block (line) size
  - the amount of data that gets moved into or out of the cache as a chunk
    - analogous to page size in virtual memory
- replacement policy
  - what data is replaced on a miss?
- associativity
  - how many locations in the cache is a given address eligible to be placed in?
- unified, instruction, data
  - what type of data is kept in the cache? We'll cover this in more detail later

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 of 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



Example: Alternatives in an 8 Block Cache

- Direct Mapped: 8 blocks, 1 way, 1 tag comparator, 8 sets
- Fully Associative: 8 blocks, 8 ways, 8 tag comparators, 1 set
- 2 Way Set Associative: 8 blocks, 2 ways, 2 tag comparators, 4 sets
- 4 Way Set Associative: 8 blocks, 4 ways, 4 tag comparators, 2 sets

Peer Instruction

- For a cache with constant total capacity, if we increase the number of ways by a factor of two, which statement is false:

A : The number of sets could be doubled
B : The tag width could decrease
C : The block size could stay the same
D : The block size could be halved

Example of Caches

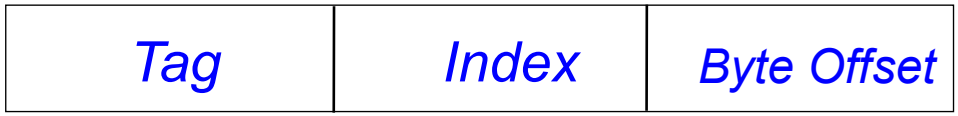
- given a 2MB, direct-mapped physical caches, line size=64bytes, and 52-bit physical address
  - tag size?
  - now change it to 16-way, tag size?
  - how about if it's fully associative, tag size?

Total Cache Capacity =

Associativity * # of sets * block_size

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

*C = N * S * B*



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?

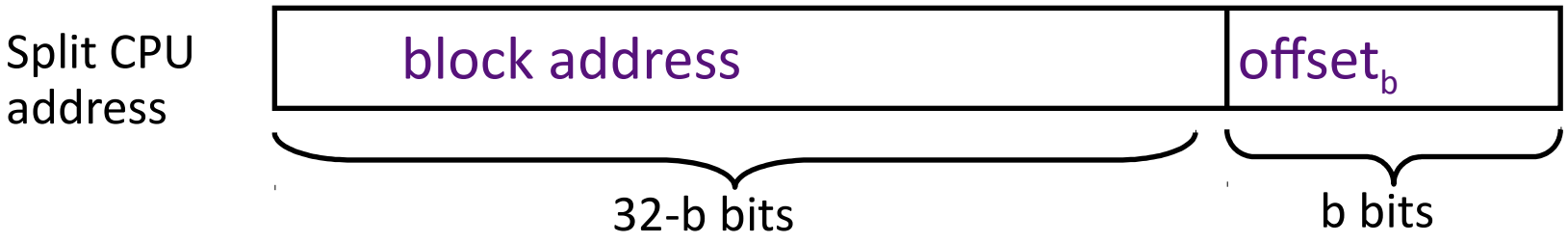
Double the Sets: Associativity?

tag_size? index_size? # comparators?

Block Size and Spatial Locality

- Block is unit of transfer between the cache and memory

4 word block, b=2



2^b = block size *a.k.a* line size (in bytes)

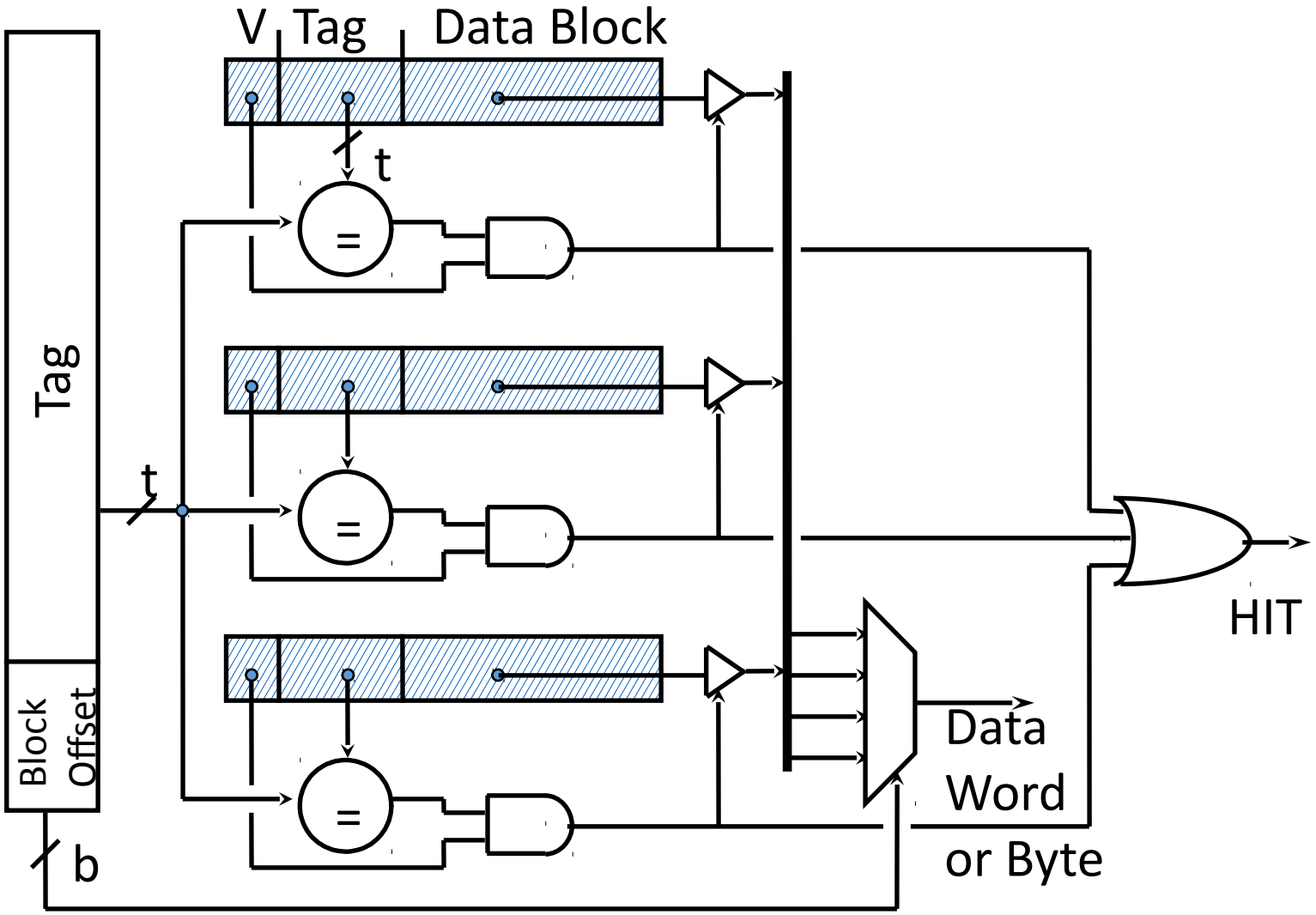
Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

What are the disadvantages of increasing block size?

Fewer blocks => more conflicts. Can waste bandwidth.

Fully Associative Cache



Your Turn

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

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

Peer Instruction

- For a cache with constant total capacity, if we increase the number of ways by a factor of two, which statement is **false**:

A : The number of sets is halved
B : The tag width decreases
C : The block size stays the same
D : The set index decreases

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
D : All four statements are true

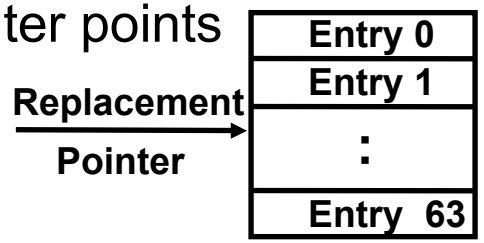
Costs of Set-Associative Caches

- 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)



Replacement Policy

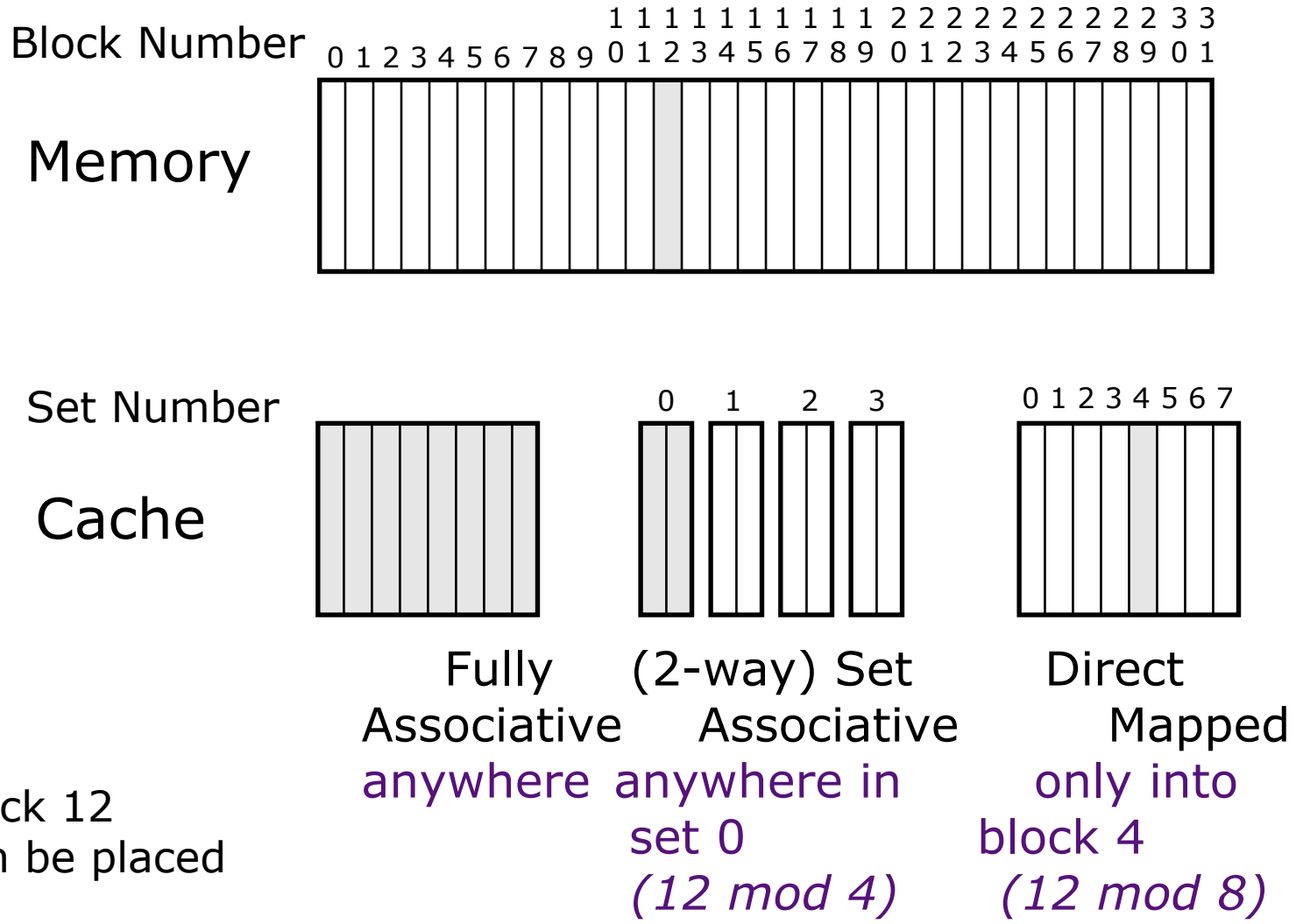
In an associative cache, which block from a set should be evicted when the set becomes full?

- Random
- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree often used for 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches
- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used block or blocks

This is a second-order effect. Why?

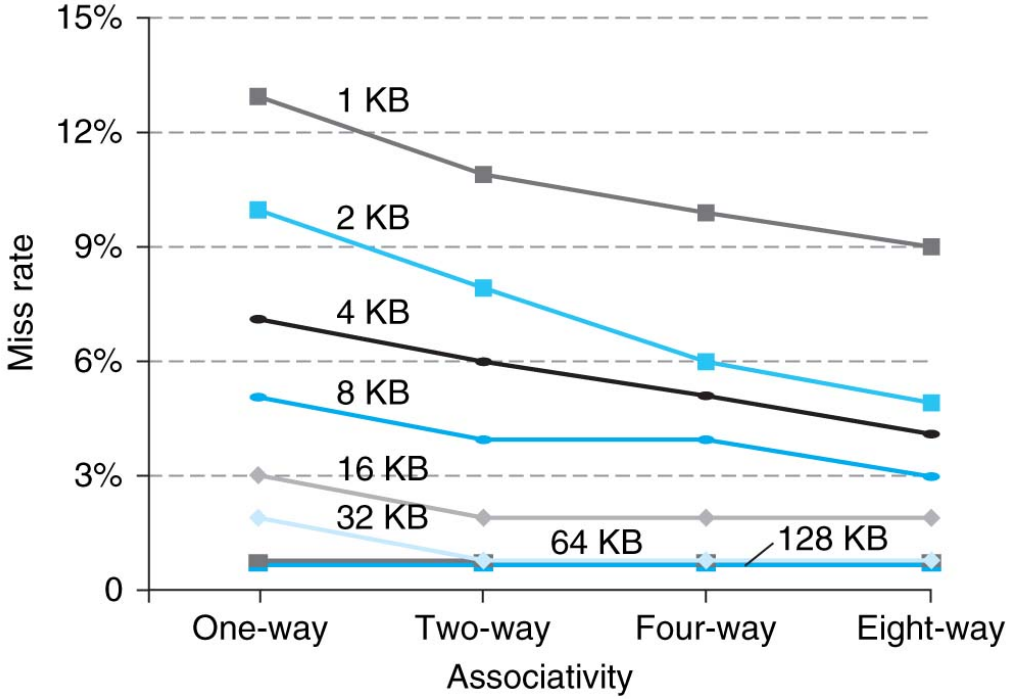
Replacement only happens on misses

Placement 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)

Miss Rate vs. Cache Size on the Integer Portion of SPECCPU2000