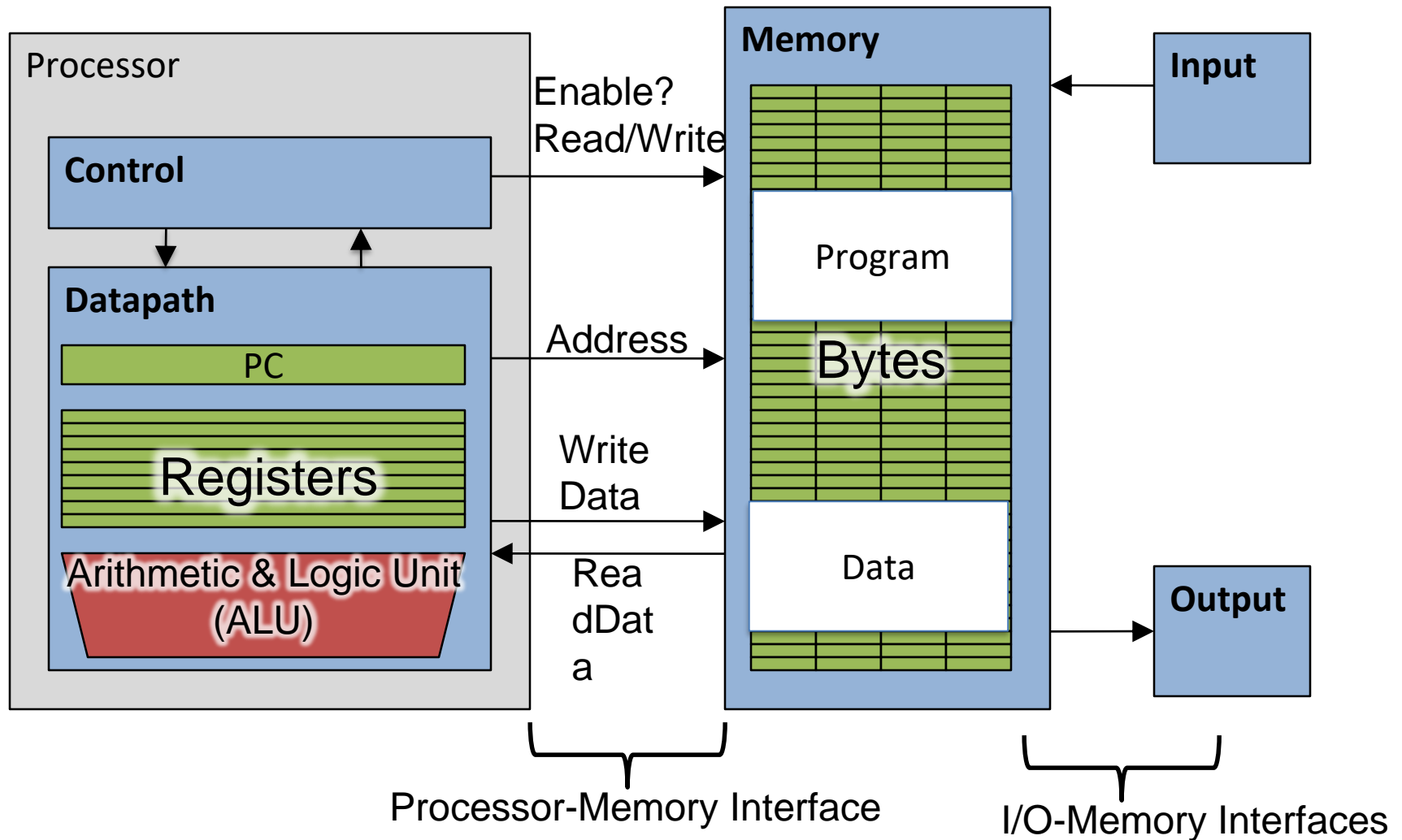


# Chapter 5

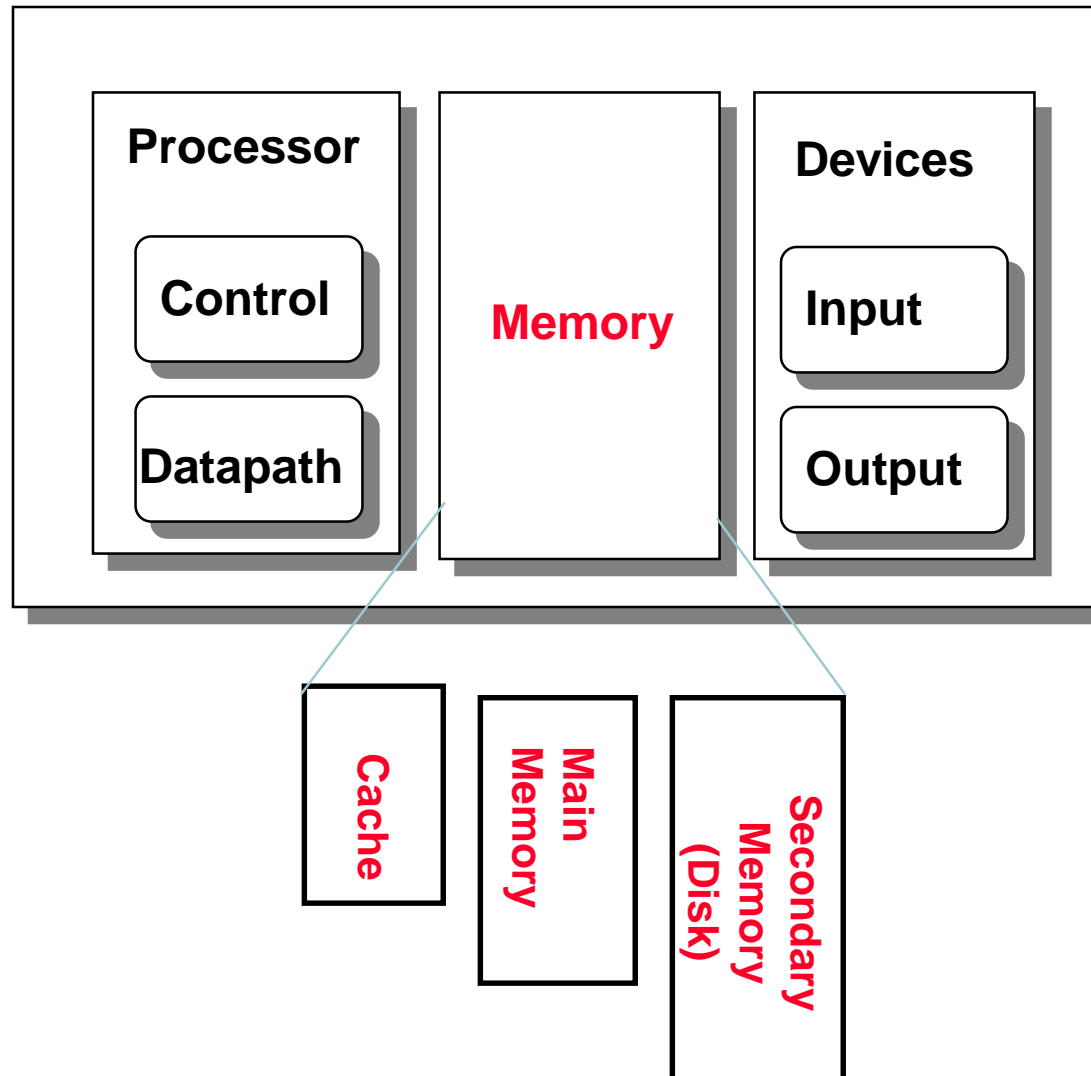
## Memory Hierarchy

מקורות השקפים :  
אתר הספר  
ד"ר רם בוסני  
ד"ר ג'ון מרברג  
אוניברסיטת ברקלי

# Components of a Computer



# Review: Major Components of a Computer



# RAM – Random Access Memory

- Principle of Random Access
  - Access to any (random) address takes the same time
- Two major technologies
  - SRAM - Static RAM
  - DRAM - Dynamic RAM
- Difference is how bits are stored
  - SRAM uses more logic (more transistors)
  - DRAM requires refresh every few ns (dynamic)
- Some practical impacts of the technology
  - SRAM is faster
  - SRAM consumes less power
  - DRAM is cheaper per bit
  - DRAM has higher bit density (more bits per area)

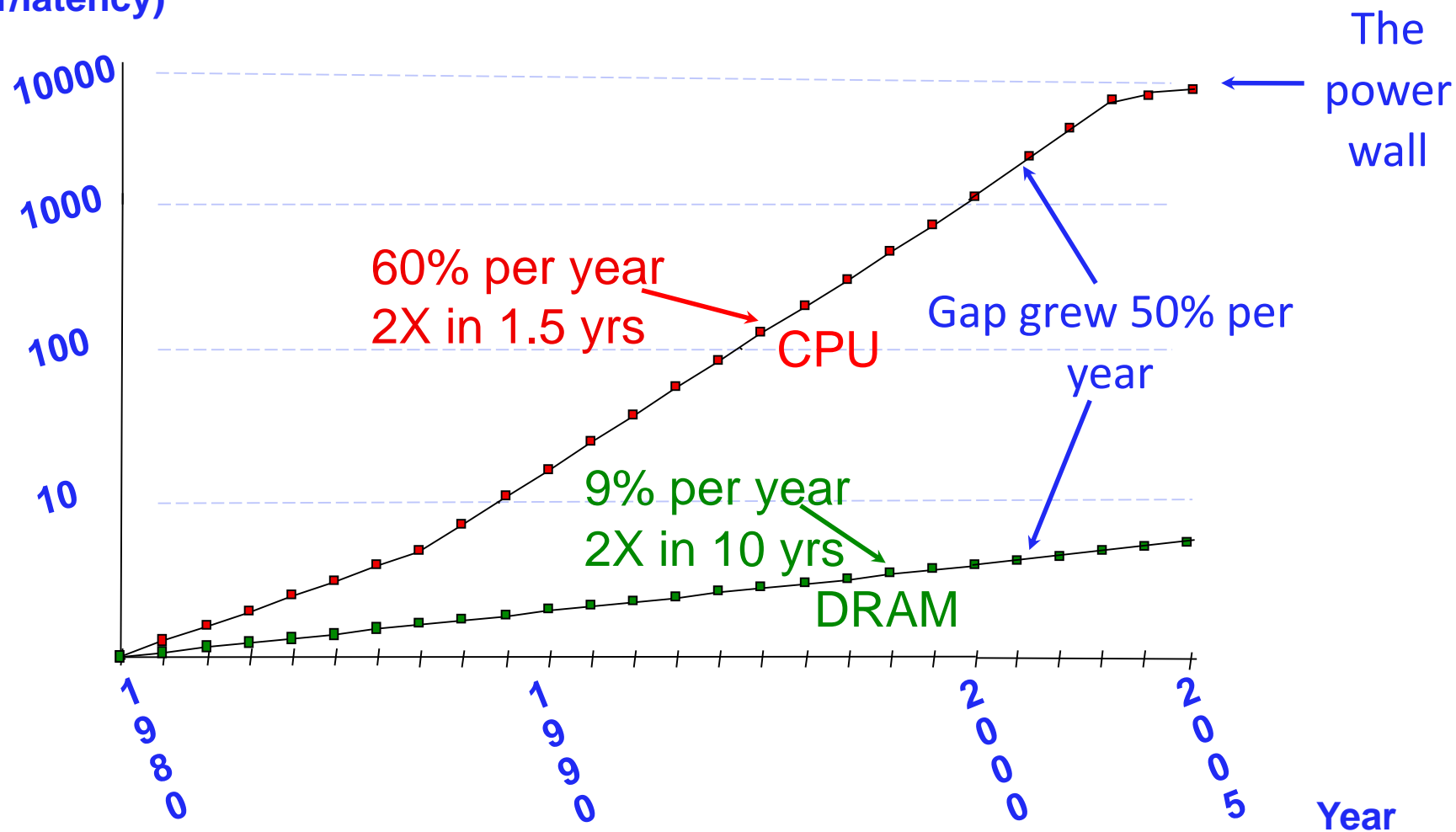
Technology	Access Time (ns) (in 2008)	Price per GB (in 2008)
SRAM	0.5-1.5 ns	2000-5000 \$
DRAM	50-70 ns	20-70 \$
Magnetic Tape	5-20 million ns	0.2-2 \$

# CPU-DRAM Speed gap

Q. How do architects address this gap?

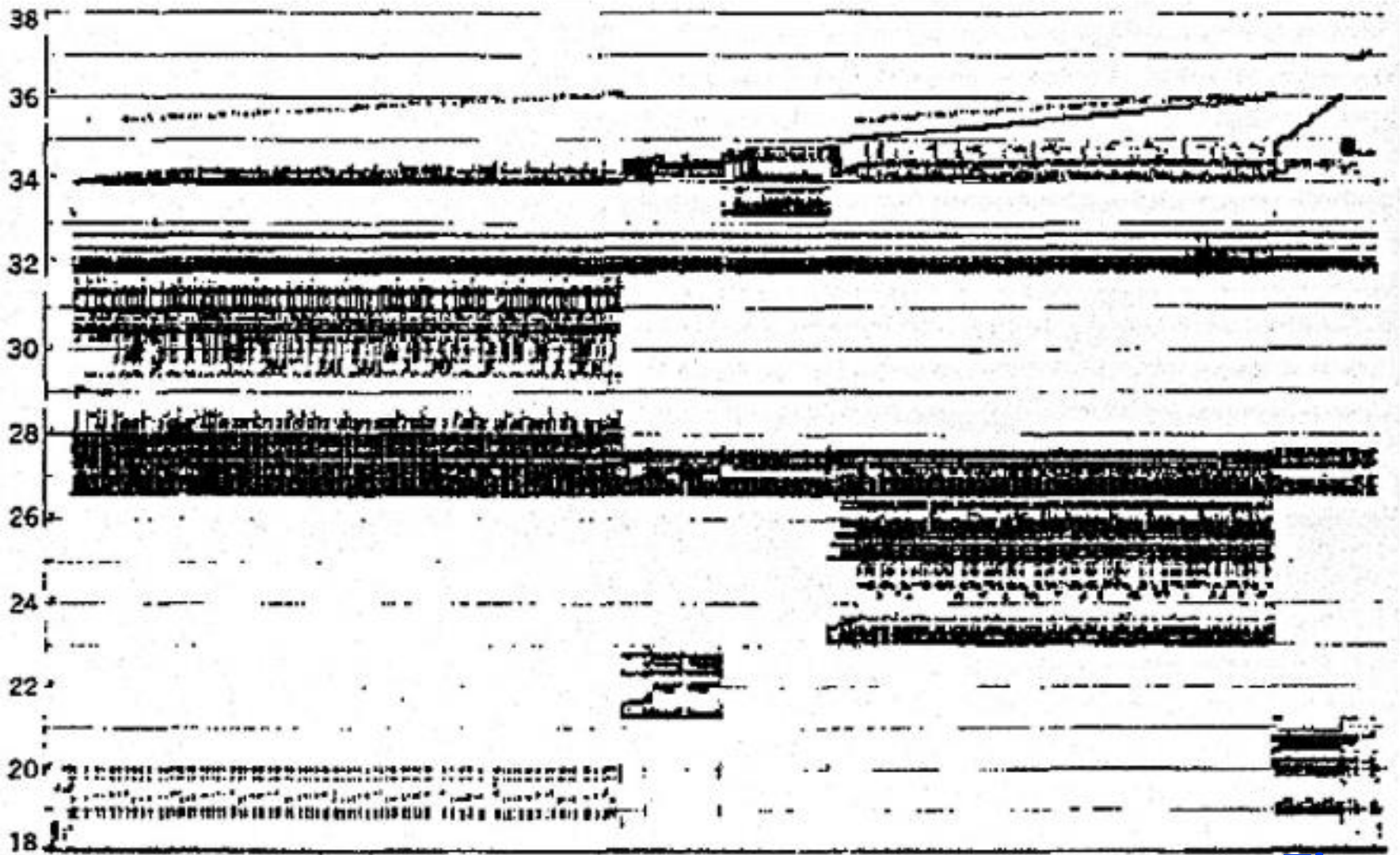
A. Put smaller, faster SRAM “cache” memory between CPU and DRAM

Performance  
(1/latency)



# Real Memory Reference Patterns

Memory Address (one dot per access)



Time

Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

# Principle of Locality

- Programs spend much of their time on accessing memory
- But – in any timeframe, programs access only a small portion of their address space
- **Temporal locality** (time)
  - Addresses accessed recently are likely to be accessed again soon
  - E.g., program variables, instructions in a loop
- **Spatial locality** (space)
  - Addresses near those accessed recently are likely to be accessed next
  - E.g., array data, sequential instructions

# Example: Principle of Locality

- Consider an array  $A[]$  of size 10000
  - The array is stored sequentially in memory
- Access to  $A[]$  with high spatial, low temporal locality

```
for (i=0; i<10000; i++)
    A[i]=17;
```
- Access to  $A[]$  with low spatial, high temporal locality

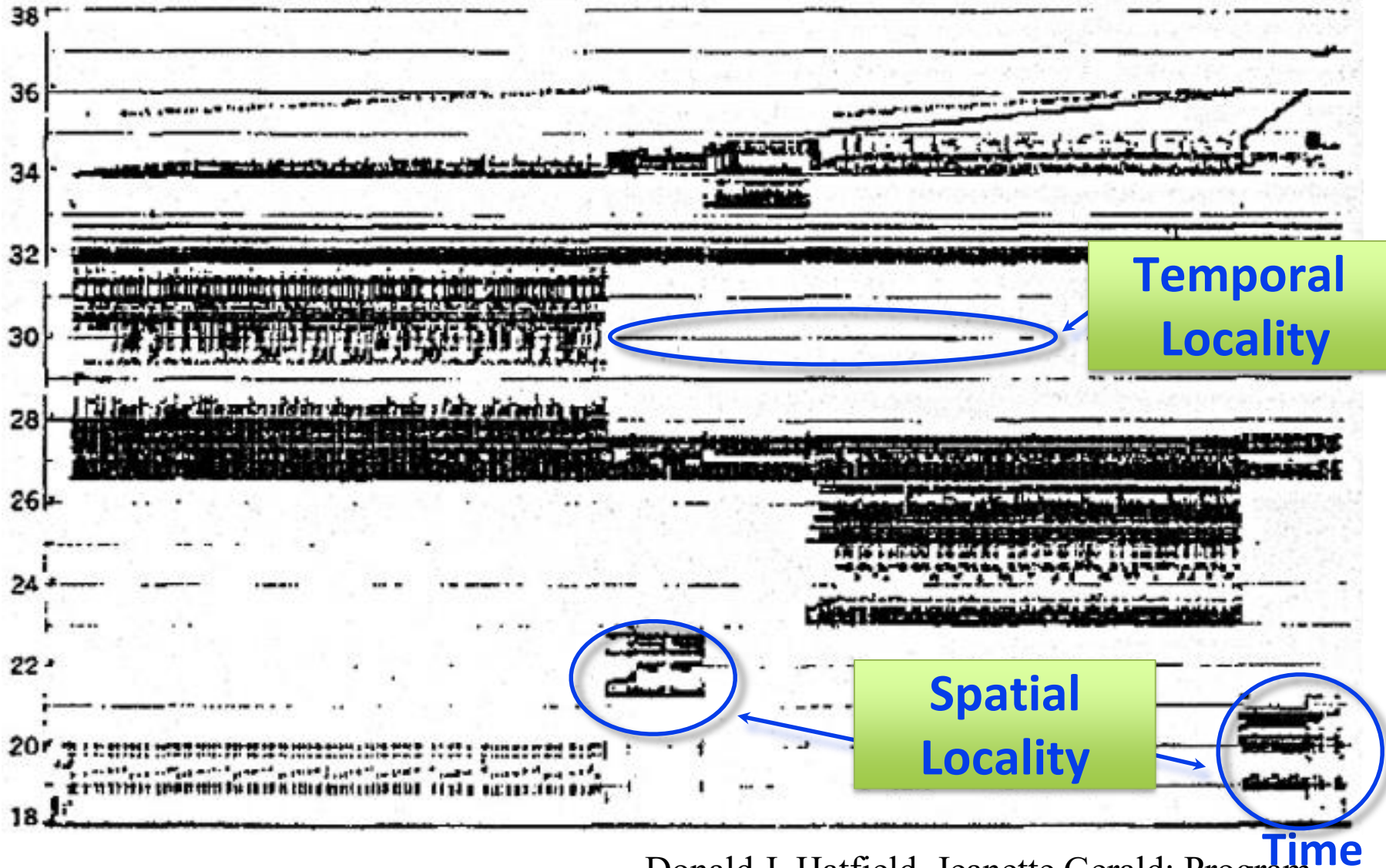
```
for (i=0; i<10000; i++)
    print(A[(i%50)*200]+i);
```
- Access to  $A[]$  with high spatial and high temporal locality

```
for (i=0; i<10000; i++)
    B[i]=A[i%100];
```



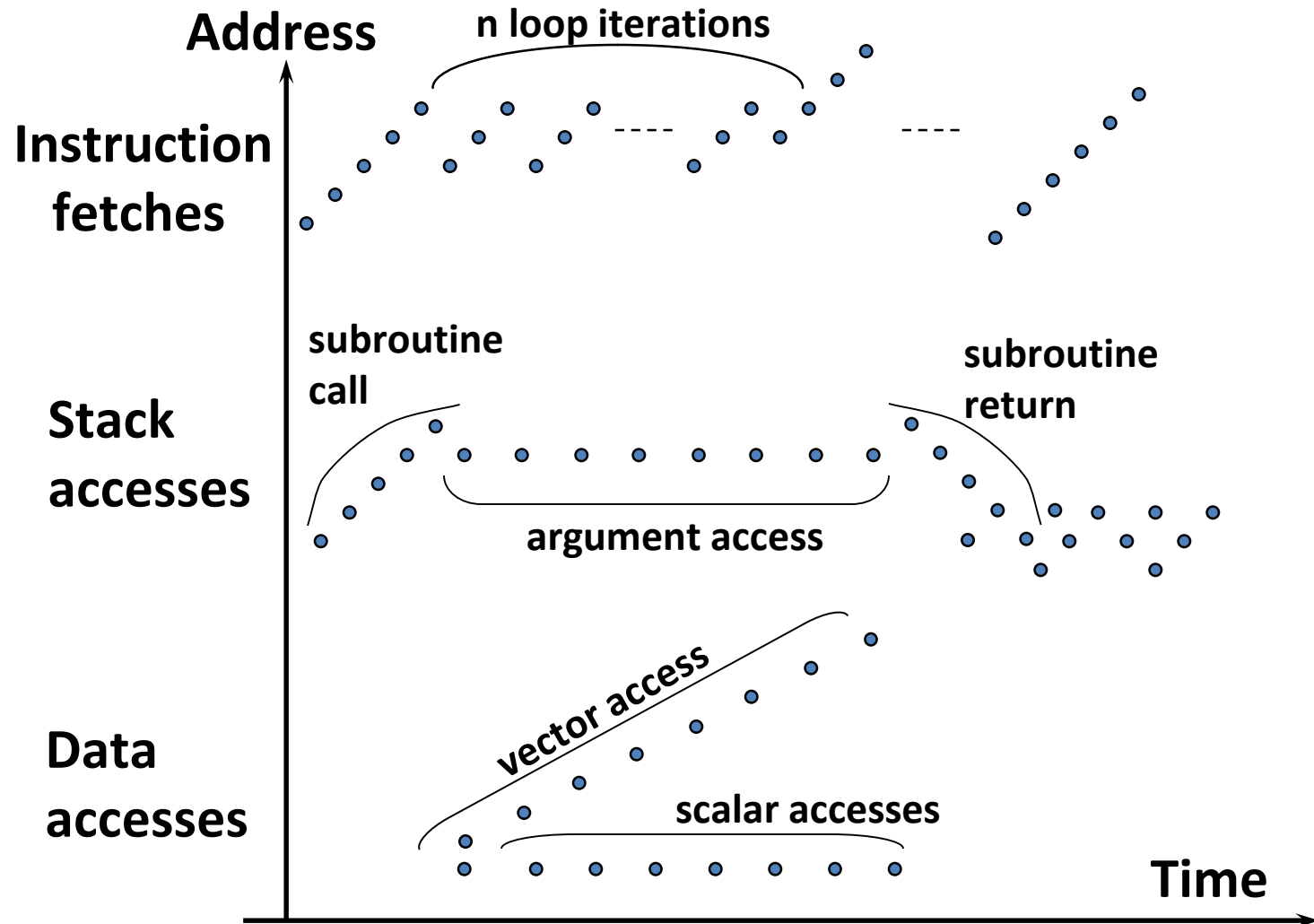
# Memory Reference Patterns

Memory Address (one dot per access)



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

# Memory Reference Patterns

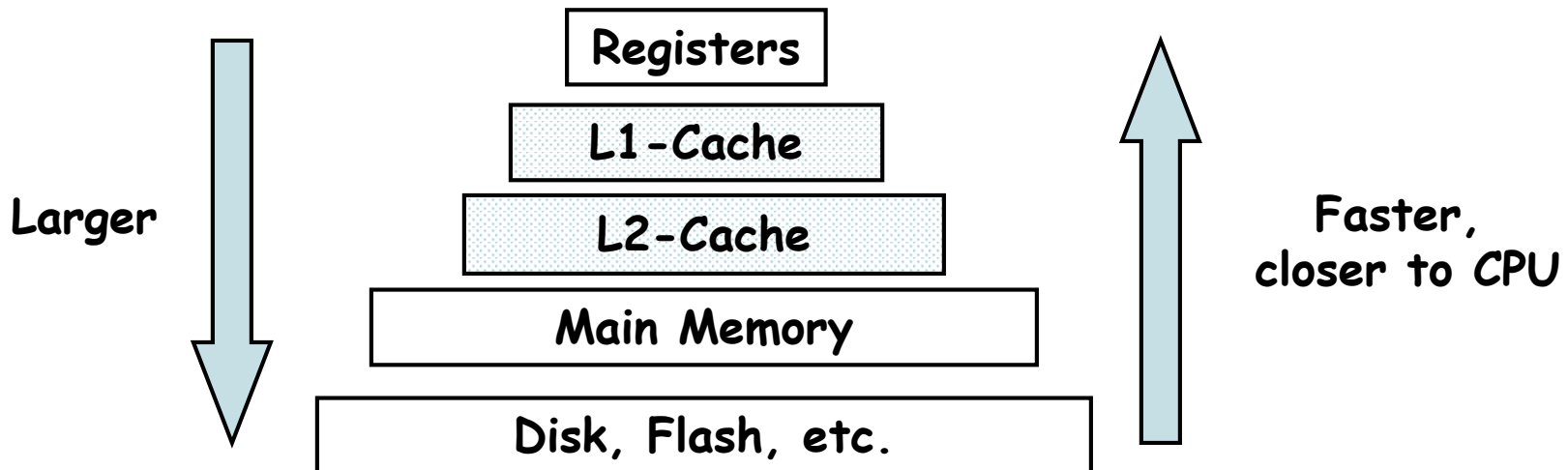


# Memory Hierarchy

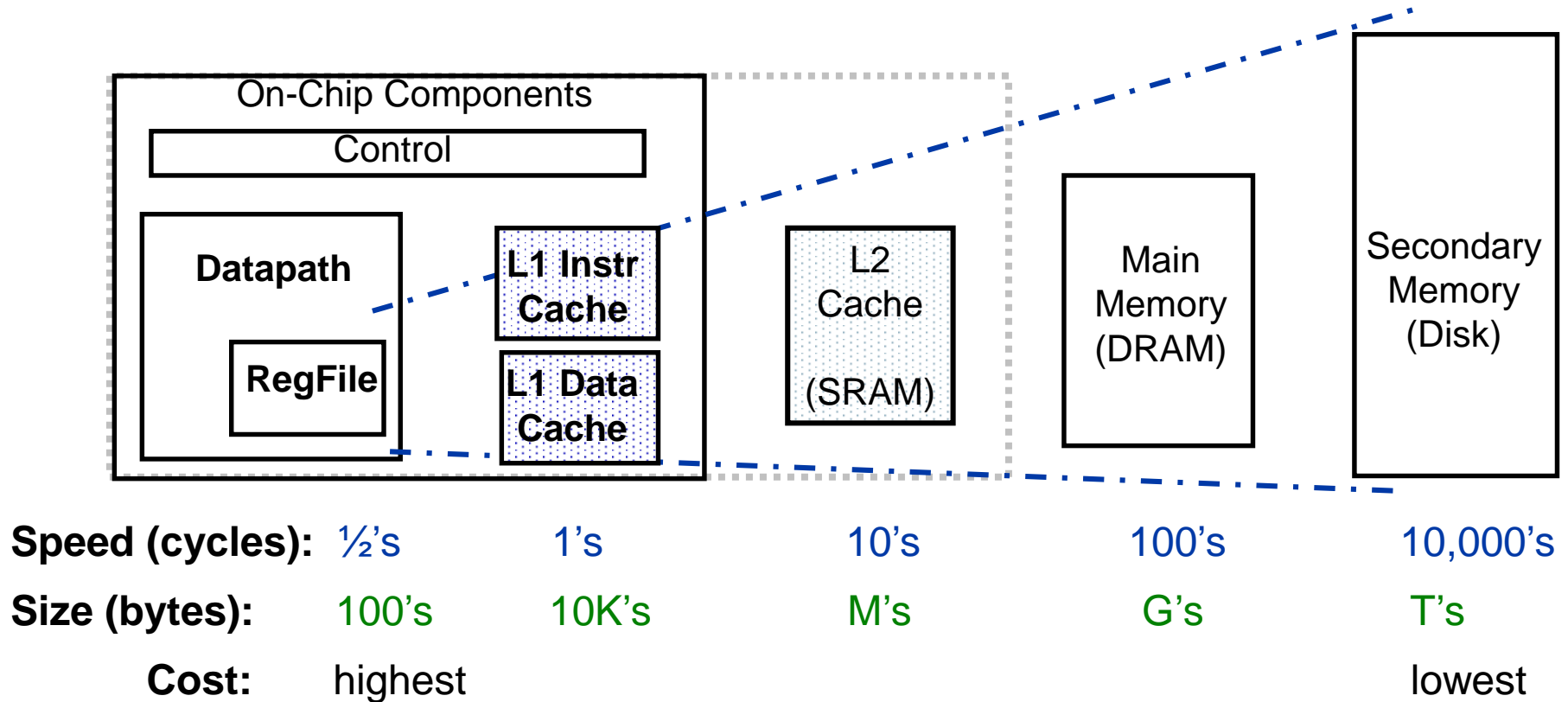
- Goal: take advantage of the **principle of locality** to present the user with a memory that has:
  - The **size** affordable with the *cheapest* technology
  - The **speed** offered by the *fastest* technology
- Approach:
  - a. Store everything on disk  
→ **Secondary storage**
  - b. Copy recently accessed (and nearby) locations from disk to smaller DRAM memory  
→ **Main memory**
  - c. Copy more recently accessed (and nearby) locations from DRAM to smaller SRAM memory  
→ **Cache memory**

# What is a Cache

- A cache is faster (and smaller) memory intended to improve average access time to slower (and larger) memory
- Exploiting temporal and spatial locality, the faster memory copies (“caches”) some of the contents of the slower memory
- Each memory level caches parts of the next level
  - Registers cache program variables
  - L1 caches L2
  - L2 caches the main memory
  - Main memory caches the disk/flash (virtual memory)
- We focus mainly on L1 cache – closest to the CPU (on-chip)



# Typical Memory Hierarchy Performance



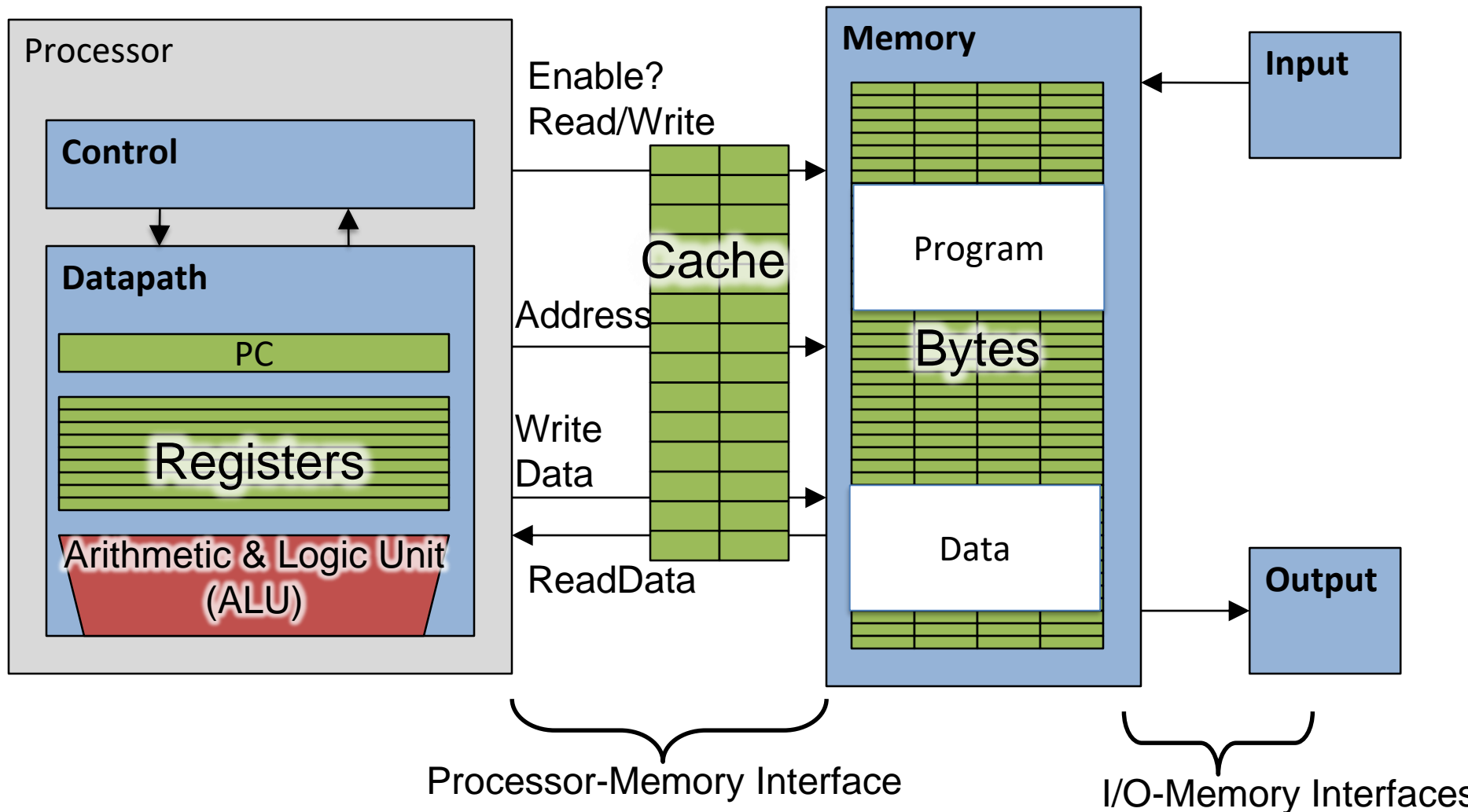
# Cache Philosophy

- Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - Works fine even if programmer has no idea what a cache is
  - However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache

# Memory Access without Cache

- Load word instruction: `lw $t0, 0($t1)`
- `$t1` contains  $1022_{\text{ten}}$ , `Memory[1022] = 99`
  1. Processor issues address  $1022_{\text{ten}}$  to Memory
  2. Memory reads word at address  $1022_{\text{ten}}$  (99)
  3. Memory sends 99 to Processor
  4. Processor loads 99 into register `$t0`

# Adding Cache to Computer





# Memory Access with Cache

- Load word instruction: `lw $t0, 0($t1)`
- `$t1` contains `1022ten`, `Memory[1022] = 99`
- With cache (similar to a hash)
  1. Processor issues address `1022ten` to Cache
  2. Cache checks to see if has copy of data at address `1022ten`
    - 2a. If finds a match (Hit): cache reads 99, sends to processor
    - 2b. No match (Miss): cache sends address 1022 to Memory
      - I. Memory reads 99 at address `1022ten`
      - II. Memory sends 99 to Cache
      - III. Cache replaces word with new 99
      - IV. Cache sends 99 to processor
  3. Processor loads 99 into register `$t0`

# Cache Terminology

- **Block** (aka **Line**): the basic amount of data copied between the cache and the next memory level
- **Block Size**: Typically measured in number of **words**
  - Number of **words** in block:  $2^m$  (typically  $0 \leq m \leq 4$ )
  - Number of **bytes** in block:  $2^{m+2}$
- **Cache size**: Typically measured in number of **blocks**
  - Number of **blocks** in cache:  $2^n$  (typically  $2 \leq n \leq 12$ )
  - Number of **words** in cache:  $2^{n+m}$
  - Number of **bytes** in cache:  $2^{n+m+2}$
- **Index**: the sequential number of a cache block
  - $0 \leq \text{index} < 2^n$

# Cache Terminology (Cont.)

- **Hit:** successful access to data in the cache
  - The requested memory block is **found** in the cache
  - Continue normal execution
- **Miss:** failed access to data in the cache
  - The requested memory block is **not found** in the cache
  - Need to retrieve the block from the next memory level and insert it into the cache
  - Execution is **stalled** until the block is in the cache

# Cache Terminology (Cont.)

- **Hit Rate**: the fraction of memory accesses that are a hit
- **Hit Time**: time to access data in the cache
  - Consists of: time to access the block  
+ time to determine hit/miss
- **Miss Rate**: fraction of memory accesses that are a miss
  - Equals  $1 - \text{HitRate}$
- **Miss Penalty**: time to bring a block into the cache from the next level
  - Consists of: time to access the block in the next level  
+ time to transmit the block to the cache  
+ time to insert the block into the cache  
+ time to access the block (again) in the cache

Typically Hit Time  $\ll$  Miss Penalty

# Cache Organization

- Consider two closely related issues
  - How to determine whether or not the requested data is in the cache?
  - If the data is in the cache, how to find it?
- To handle these issues efficiently, cache content is organized in one of several ways
  - Direct-mapped cache
  - Fully associative cache
  - K-way set-associative cache

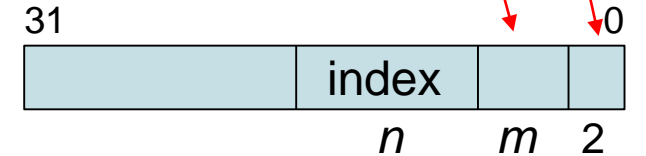
# Direct Mapped Cache

- Location of data in the cache is determined from:
  - Address in memory, block size, and number of blocks in cache
- Assume  $2^n$  cache blocks, each block is  $2^m$  words
- There is one fixed choice (direct mapping)

$$\text{blockAddress} = \text{address} \gg (m+2)$$

$$\text{index} = \text{blockAddress} \bmod 2^n$$

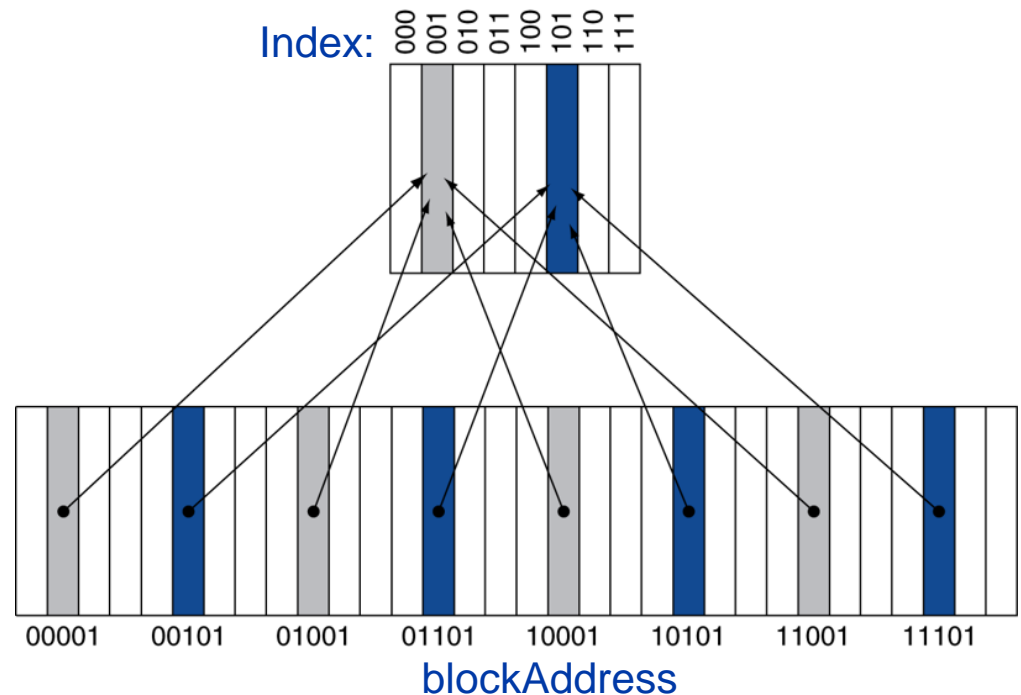
- Index designates the cache block where the data is located
  - Bits  $m+2$  to  $m+n+1$  in memory address are the index



Example:

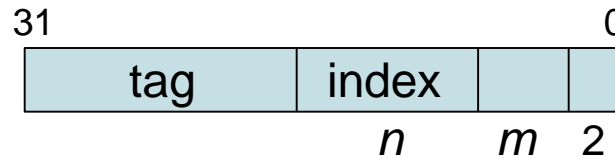
8 blocks, 1 word/block

→  $n=3, m=0$



# Tag and Valid Bit

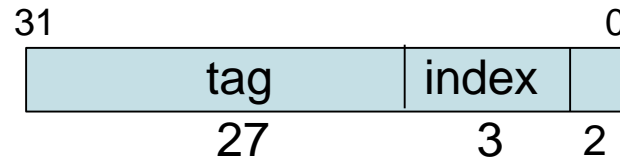
- How do we know which particular memory block is stored in each cache block?
  - Keep high order bits of address alongside data
  - This is called the **tag**



- What if there is no data in a cache block?
  - Maintain a “**valid**” **bit** for each cache block
    - 1 = valid (“full”) 0 = invalid (“empty”)
  - Initially valid=0
  - Can be used to invalidate the cache block

# Cache Organization Example

- 8 blocks, 1 word/block, direct mapped



Index	Valid	Tag (27 bits)	Data (32 bits)
000			
001			
010			
011			
100			
101			
110			
111			



# Accessing Data in Direct-Mapped Cache

- Given a memory address, access the corresponding block in a direct-mapped cache

## Step 1: Obtain block index

- Use the index field of the memory address

## Step 2: Check the valid bit of the cache entry

## Step 3: Check if the entry at the given index in the cache contains the relevant memory block

- Compare the tag in the cache entry with the tag field of the memory address

## Step 4: Announce whether hit or miss

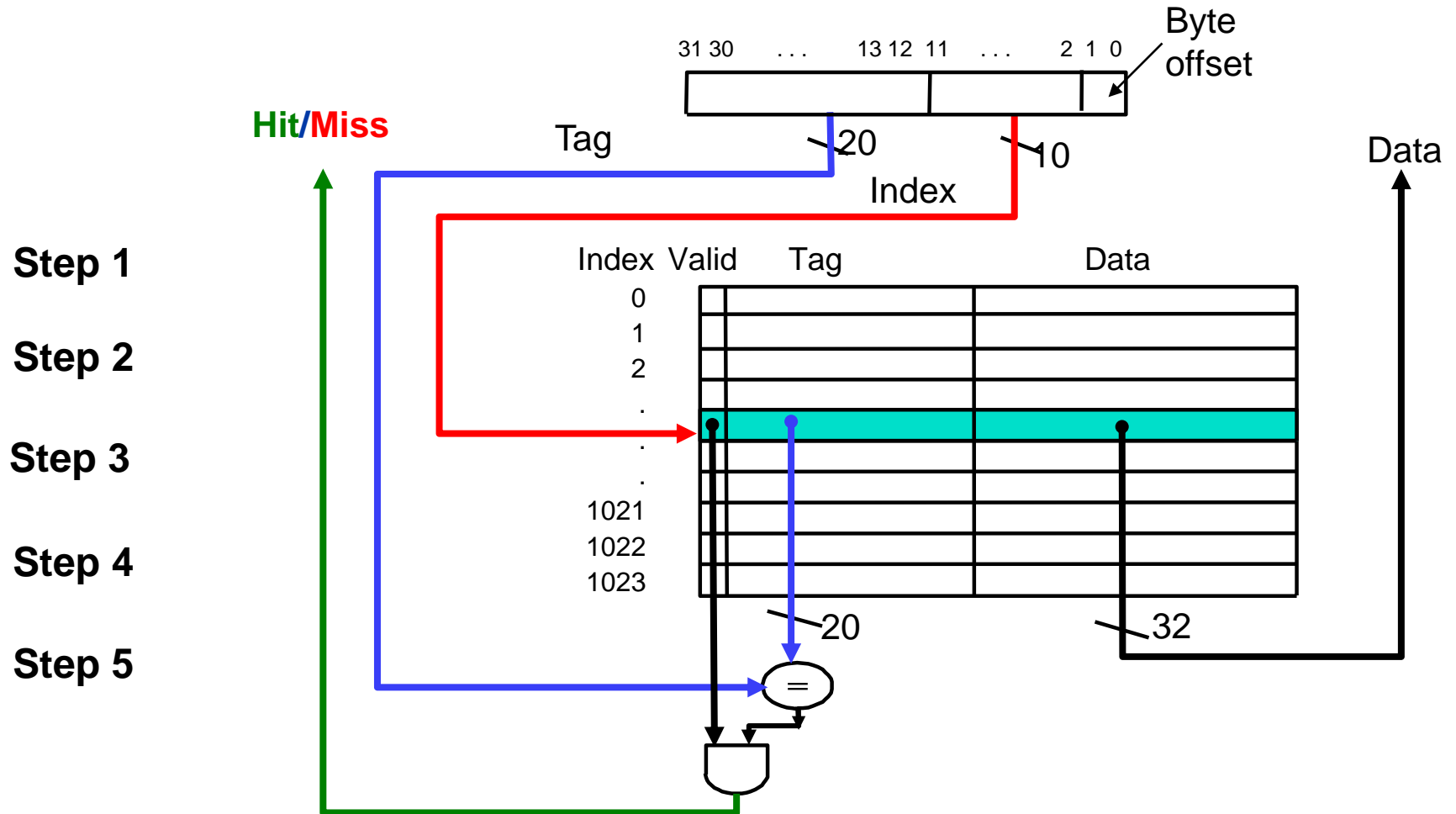
- If tags are equal and valid bit is 1 → it's a **hit**
- Otherwise → it's a **miss**

## Step 5: Access Data

- If a **hit**, access data in cache
- If a **miss**, bring data to cache

# Accessing Data in Direct-Mapped Cache

1 word/block      1024 Blocks



# Direct Mapped Cache: Temporal Example

0...010**110**00

Miss: valid

lw \$1,88(\$0)

0...011**010**00

Miss: valid

lw \$2,104(\$0)

0...010**110**00

Hit!

lw \$3,88(\$0)

Index	Valid	Tag (27 bits)	Data (1 word)
000	N		
001	N		
010	Y	0...011	Memory[01101000]
011	N		
100	N		
101	N		
110	Y	0...010	Memory[01011000]
111	N		

# Direct Mapped Cache: Temporal Example

0...010**110**00

Miss: valid

lw \$1,88(\$0)

0...011**110**00

Miss: tag

lw \$2,120(\$0)

0...000**110**00

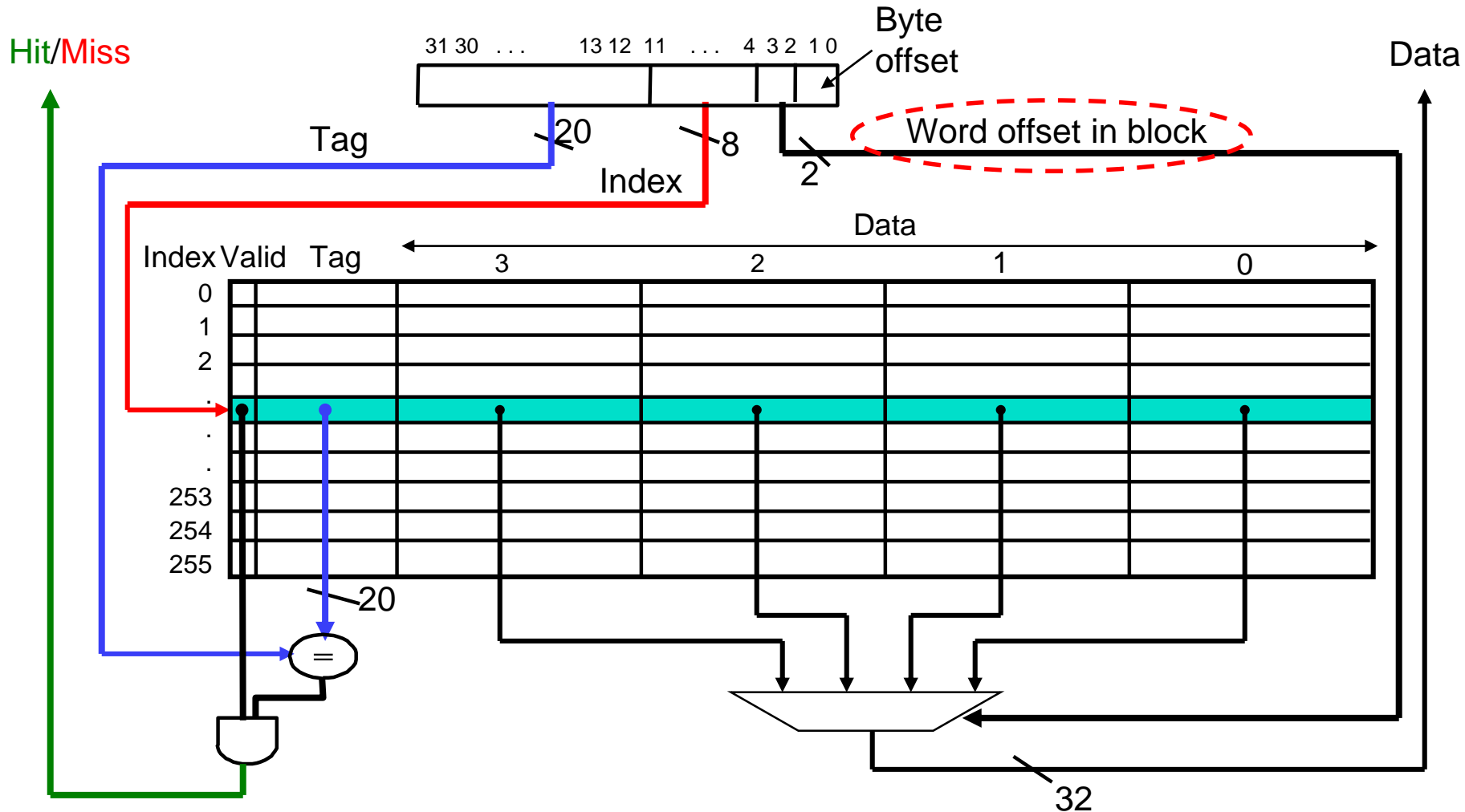
Miss: tag

lw \$3,24(\$0)

Index	Valid	Tag (27 bits)	Data (1 word)
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	0...00	Memory[00011000]
111	N		

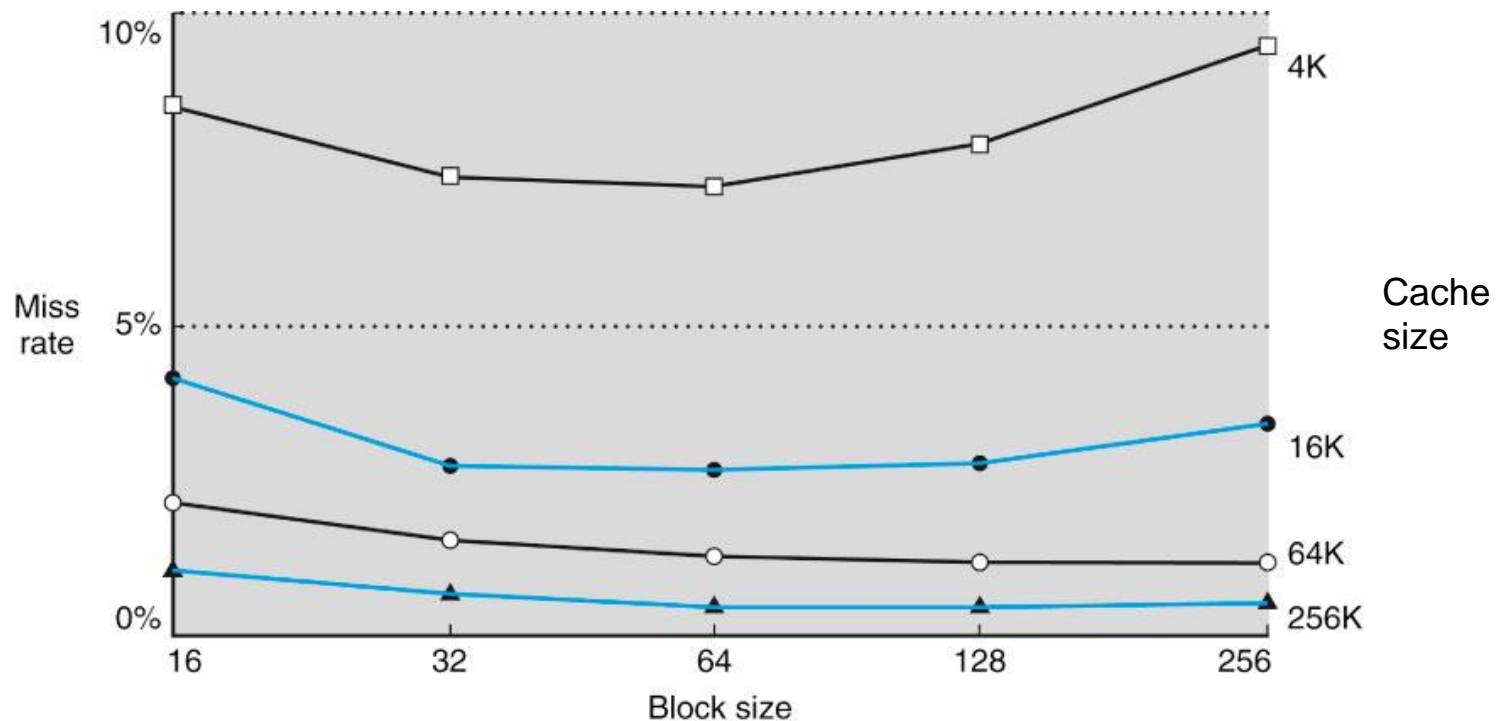
# Multi-word Block in Direct-Mapped Cache

4 words/block    256 blocks



# Block Size Tradeoff

- Larger blocks
  - should **reduce miss rate** due to spatial locality
- But - for a given (fixed) cache size
  - Larger blocks → fewer of them
    - More competition → may **increase miss rate**
  - Larger blocks → **bigger miss penalty**

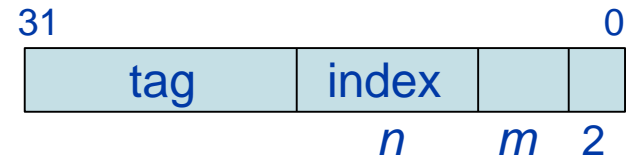


Miss rate goes up when block size becomes a significant fraction of cache size

# Total Size of Cache

- The total size of the cache includes both the space for data and the space for the extra fields

- For a data block of  $2^m$  words
  - $m$  bits are used for word offset within the block
  - 2 bits are used for byte offset within the word
- For a direct mapped cache with  $2^n$  blocks
  - $n$  bits are used for the index
  - Assuming 32-bit address space
    - ➔ Tag size is  $32 - (n + m + 2)$  bits
  - 1 bit is used for “valid”



- The total number of bits in a direct mapped cache is:  
 $((\text{\#words per block}) \times 32 + (\text{tag size}) + 1) \times \text{\#blocks}$

$$\text{bitsInCache} = (2^m \times 32 + (32 - (n + m + 2)) + 1) \times 2^n$$

- **Example:** What is the total size of a direct mapped cache with 16 KBytes of data and 4-word blocks?
  - $\text{\#blocks} = 2^n = 16\text{KBytes} / ((4 \text{ words per block}) \times (4 \text{ bytes})) = 1024$
  - $n = \log_2(1024) = 10$
  - $m = \log_2(4) = 2$
  - ➔ Total cache size = 150,528 bits = 18.37 KBytes

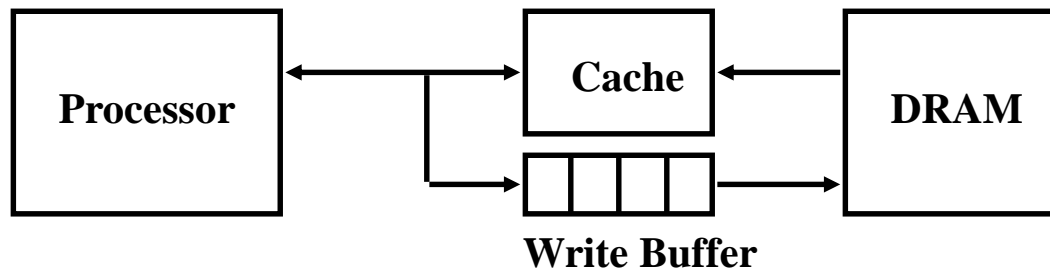
# Handling Cache Hit

- Hit on read
  - CPU pipeline continues normally
    - No stall needed for memory access
- Hit on write – need to update next-level memory
  - Alternative 1: **Write through**
    - Always write in both places: cache block and memory
    - Cache and memory are always consistent
    - ➔ Must stall on every write (wait for slower memory)
  - Alternative 2: **Write back**
    - Write only to cache block
    - Write entire block to memory only when replaced in cache
    - ➔ Only need to stall on replacement



# Write-Through

- **Problem:** Write is slow – need to **stall on every write**
  - Example: Base CPI = 1,  
10% of instructions are stores  
Write to memory takes 100 cycles
    - ➔ Need to stall 100 cycles on each write
    - ➔ Effective CPI =  $1 + 0.1 \times 100 = 11$
- **Solution:** engage a write-buffer
  - Buffer holds data waiting to be written to memory
  - CPU continues immediately after copy to buffer
  - Memory controller writes from buffer to memory
    - ➔ Only need to **stall if write-buffer is full**



# Write-Back

- Keep track whether the block has changed
  - Add a “dirty” bit to each cache block
- When the block is replaced in the cache
  - If block is dirty, write it back to memory
  - Engage a write-buffer (similar to write-through)
- Only need to stall if:
  - Block is being replaced
  - & Block is dirty
  - & Write-buffer is full

# 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

# Handling Cache Miss

- **Stall** and freeze the CPU pipeline
  - Fetch block into cache from next level memory
  - Resume the pipeline
    - Instruction is re-fetched (fixes ICache miss)
    - Data access is repeated (fixes DCache miss)
- 
- Why does the cache have a read-enable control (MemRead) ?
    - Avoid miss penalty for unused read
    - Prevent unnecessary cache block replacement

# Measuring Cache Performance

- CPU time is composed of:
  - Program execution cycles
    - This includes cache hit time
  - Memory stall cycles
    - Due to cache misses
- Simplifying assumptions
  - Same penalty for read and write miss
  - No stall on write-buffer

Memory stall Cycles =  
(IMemory stall cycles) + (DMemory stall cycles)

IMemory stall cycles =  
 $IC \times (ICache \text{ miss rate}) \times (\text{miss penalty})$

DMemory stall cycles =  
 $IC \times (\text{load\&store rate}) \times (DCache \text{ miss rate}) \times (\text{miss penalty})$

# Cache Performance Example

## ■ Rates

- ICache miss rate = 2%
- DCache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI (“ideal” cache) = 2
- 36% of instructions are load/store

## ■ What is the effective CPI?

- Stall cycles

$$\text{ICache: } IC \times 0.02 \times 100 = 2 \times IC$$

$$\text{DCache: } IC \times 0.36 \times 0.04 \times 100 = 1.44 \times IC$$

- Effective CPI = 2 + 2 + 1.44 = 5.44

## ■ ➔ Slowdown: $5.44/2=2.72$

Effective CPI is 2.72 times larger than base CPI

# Average Memory Access Time (AMAT)

- Hit time is also relevant to memory performance
  - A **larger cache** has a smaller miss rate, but also **larger hit time**
  - Increase in hit time will likely add another stage to the pipeline
  - The increase in hit time might outweigh the decrease in miss rate

- Average memory access time (AMAT)

- Performance measure that includes hit time

$$\text{AMAT} = (\text{hit time}) + ((\text{miss rate}) \times (\text{miss penalty}))$$

- Example

- Hit time = 1 cycle
  - ICache miss rate = 2%
  - DCache miss rate = 4%
  - Miss penalty = 100 cycles
  - Clock rate = 1.25 GHz
  - Assume same penalty for read/write, no other stalls

$$\text{Cycle time} = 1/1.25 \text{ GHz} = 0.8 \text{ ns}$$

$$\rightarrow \text{ICache AMAT} = 1 + (0.02 \times 100) = 3 \text{ cycles} = 2.4 \text{ ns}$$

$$\rightarrow \text{DCache AMAT} = 1 + (0.04 \times 100) = 5 \text{ cycles} = 4 \text{ ns}$$

# Example: Direct-Mapped \$ with 4 Single-Word Lines, Worst-Case 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

**0**


**4**


**0**


**4**


**0**


**4**


**0**


**4**

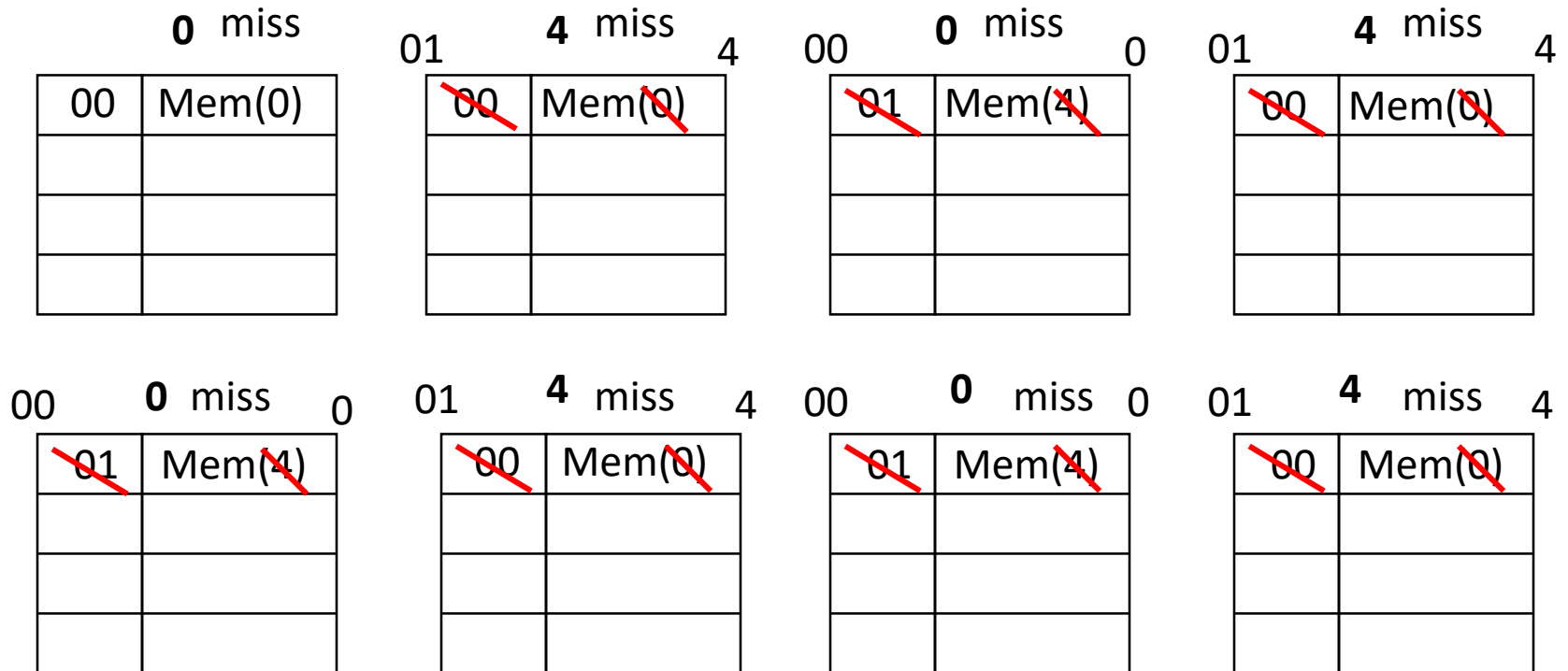



# Example: Direct-Mapped \$ with 4 Single-Word Lines, Worst-Case Reference String

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

# Associative Cache

- Goal: reduce cache miss rate
- Method: allow more flexible block placement

## ■ Fully associative

- Allow any memory block to go in any cache block
- On access, search all cache blocks in parallel
  - Need comparator per each cache block (expensive)
- There is **no index**



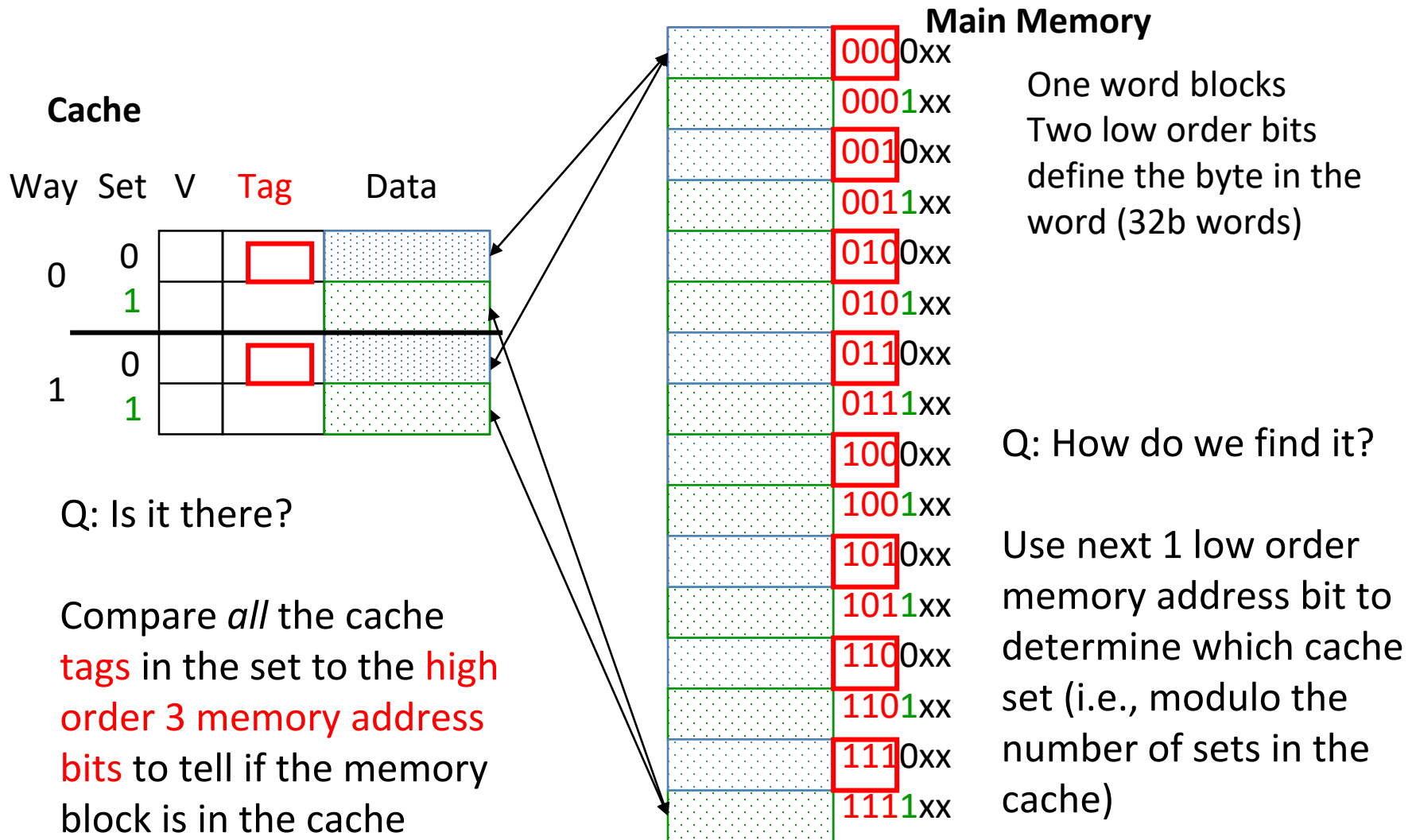
## ■ *k*-way set associative

- Cache is partitioned into  $2^n$  sets
- Each set contains  $k$  blocks (aka “ $k$  ways”)
- Cache is indexed by set number
- Index (set number) is determined from memory address
$$index = (address \gg (m+2)) \bmod 2^n$$
- Put memory block in any cache block in its designated set
- On access, search in parallel all blocks in the set
  - Need only  $k$  comparators (less expensive than fully associative)



# Example: 2-Way Set Associative \$

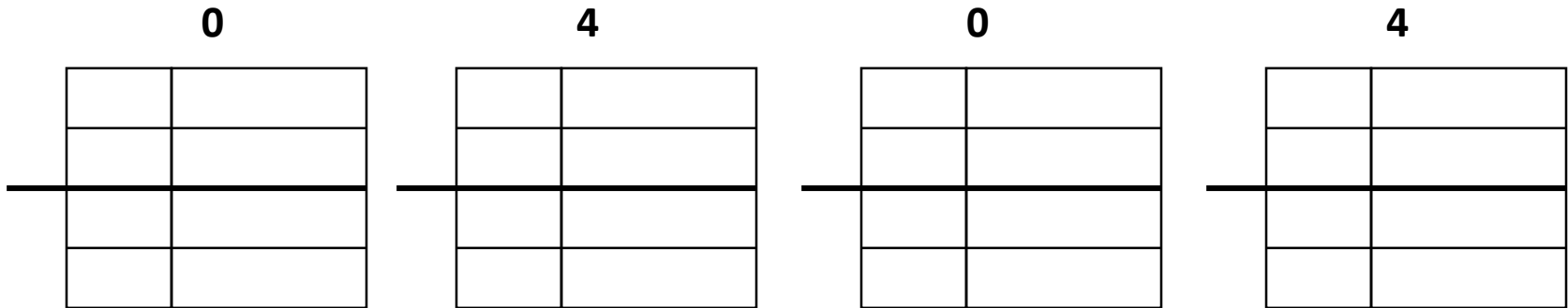
(4 words = 2 sets x 2 ways per set)



# 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



# 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

**0** miss

000	Mem(0)

**4** miss

000	Mem(0)
010	Mem(4)

**0** hit

000	Mem(0)
010	Mem(4)

**4** hit

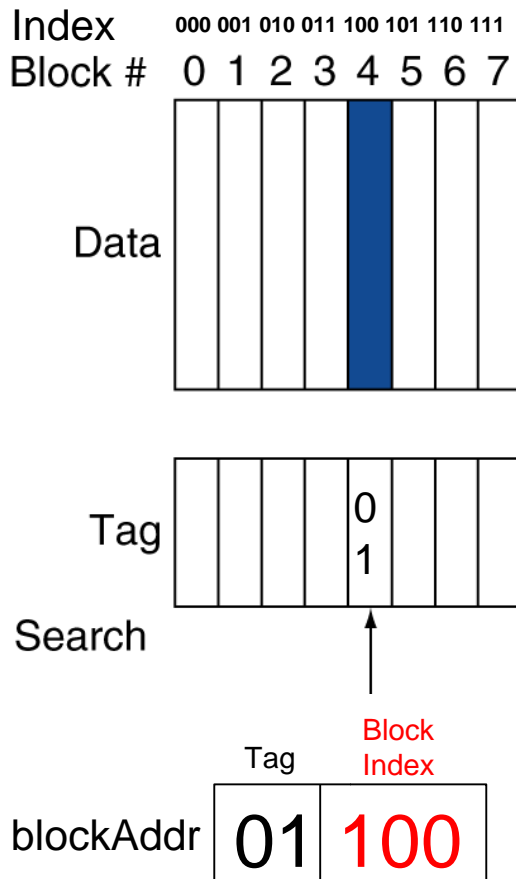
000	Mem(0)
010	Mem(4)

- 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!

# Associative Cache Example

- Total 8 blocks in cache
- Show cache location for blockAddress  $01100_2 = 12_{10}$

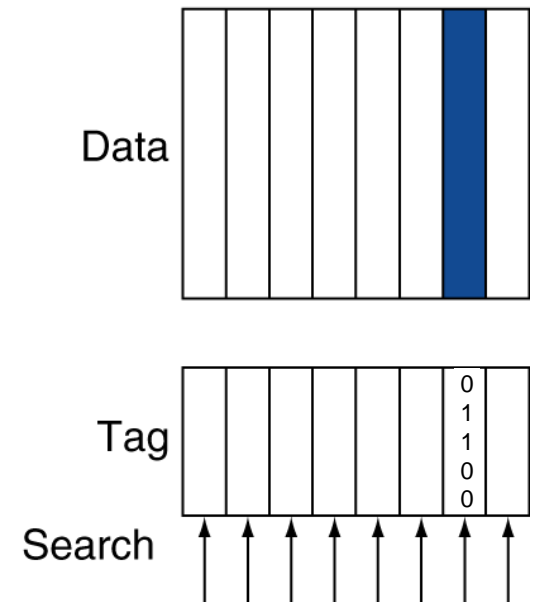
**Direct mapped**



**2-way set associative**

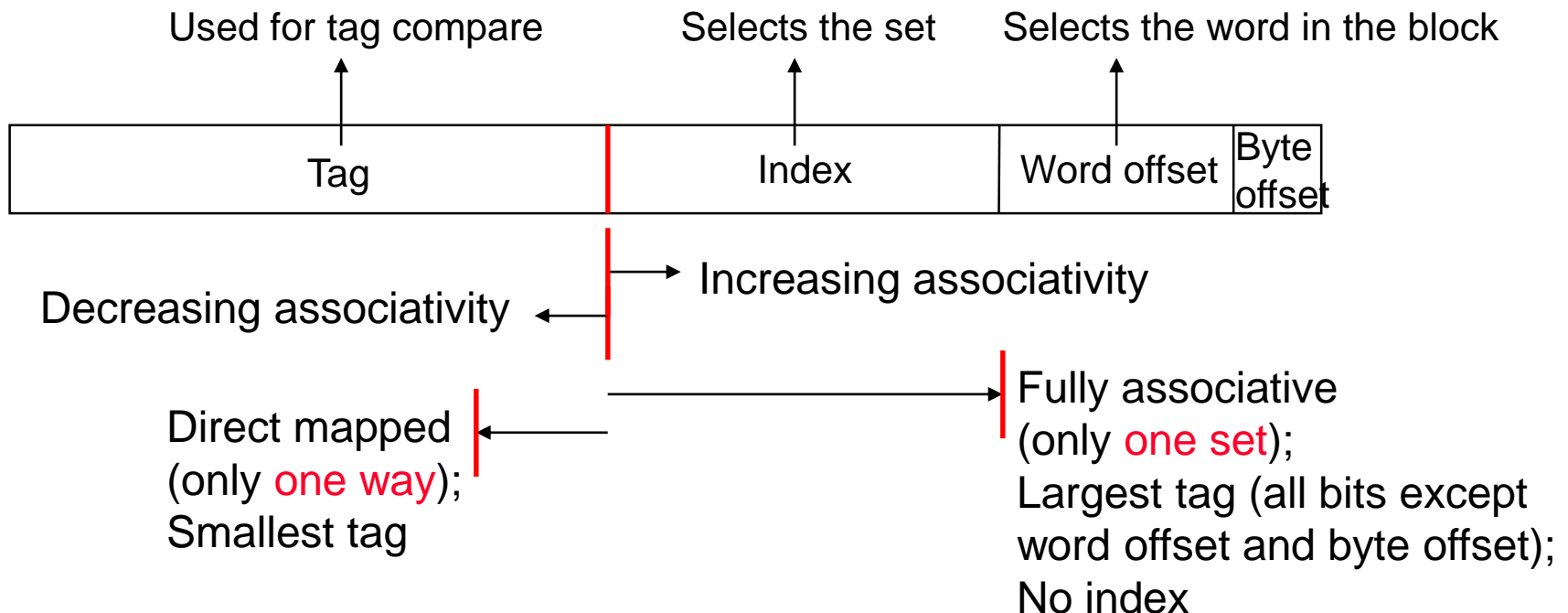


**Fully associative**



# Spectrum of Associativity

- For a given cache size, increasing associativity from  $k$  to  $2k$ 
  - Doubles the number of blocks per set
  - Halves the number of sets
    - Decreases the width of the index field by 1 bit
    - Increases the width of the tag field by 1 bit



# Different Organizations of an Eight-Block Cache

## One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

## Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

## Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

## Eight-way set associative (fully associative)

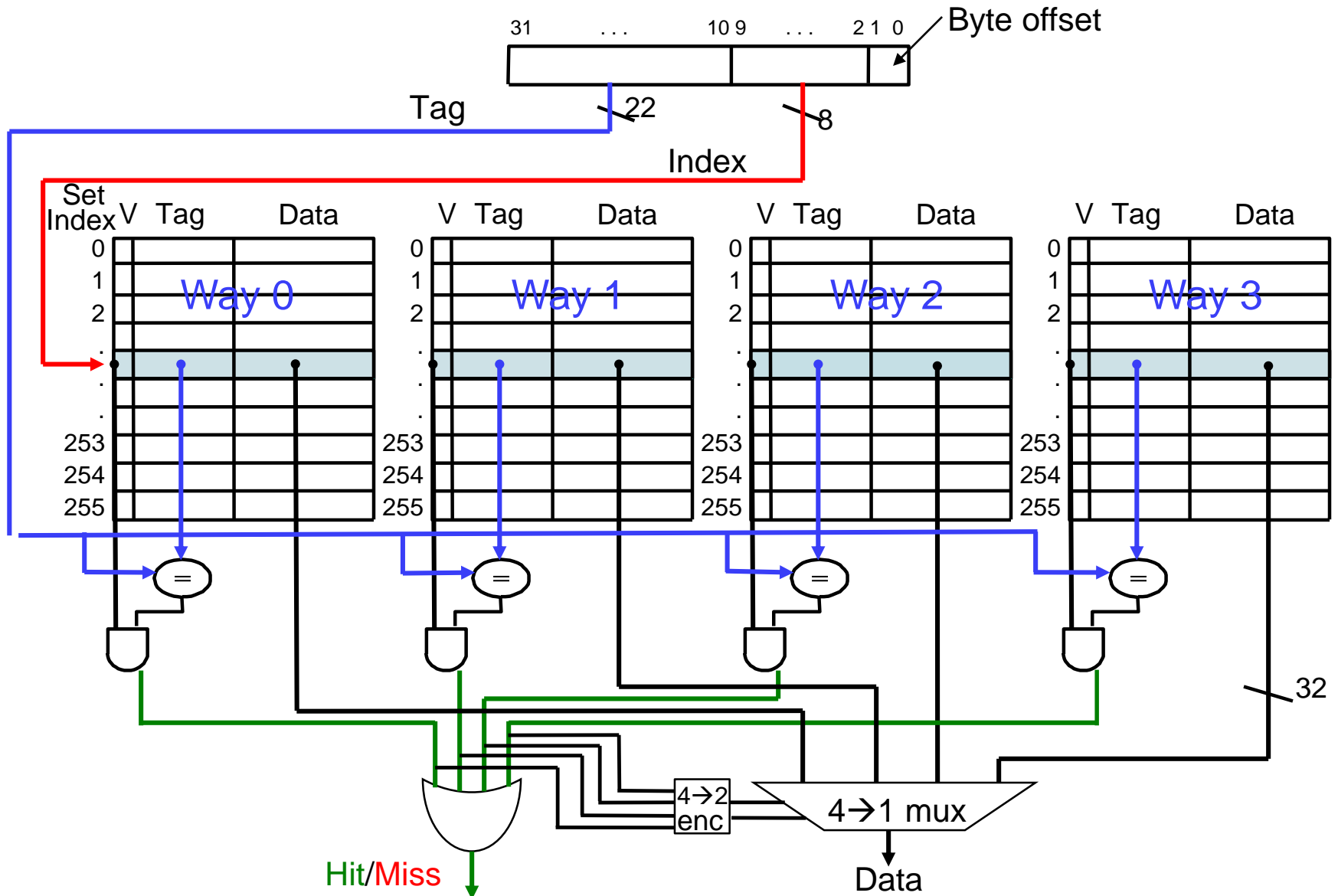
Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Total size of \$ in blocks is equal to *number of sets x associativity*. For fixed \$ 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 \$.



# Accessing Data in a Set Associative Cache

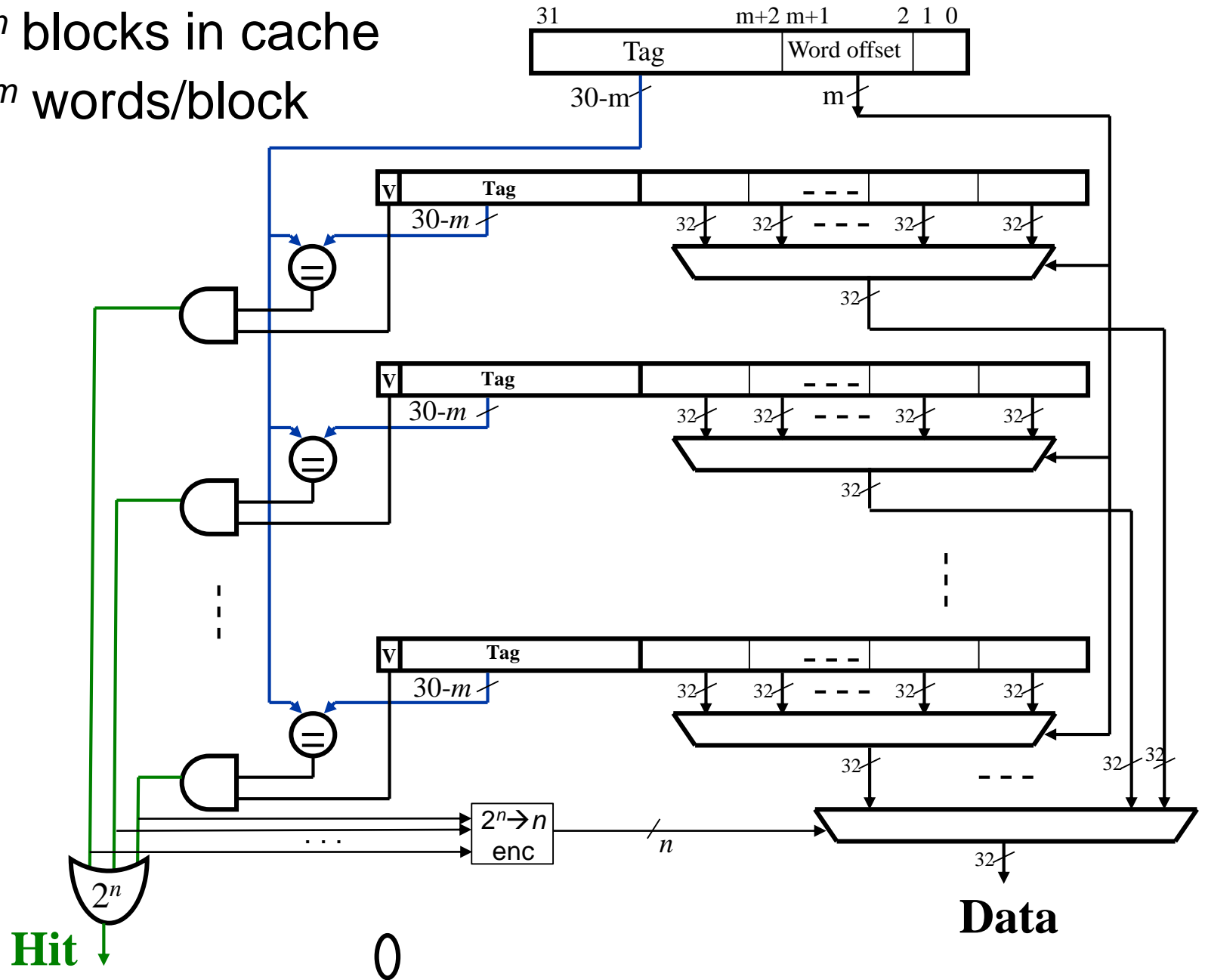
- 4 ways, 256 sets, 1 word/block



# Accessing Data in Fully Associative Cache

$2^n$  blocks in cache

$2^m$  words/block



# Which Cache Block to Replace?

- Direct mapped
  - No flexibility
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among blocks in the set using a well-defined replacement policy

# Replacement Policy

In an associative cache, which line 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 line or lines

*Replacement only happens on misses*

# Costs of Set-Associative Caches

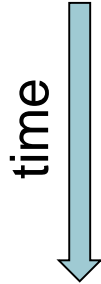
- N-way set-associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection (and Hit/Miss decision).  
DM \$: block is available before the Hit/Miss decision
    - In Set-Associative, not possible to just assume 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")

# Associativity Example

- Sequence of memory blocks accessed : 0, 8, 0, 6, 8
- Cache: 4 blocks, LRU policy
- Compare miss/hit behavior for given sequence
  - Direct mapped
  - 2-way set associative
  - Fully associative

Direct mapped

(5 miss, 0 hit)

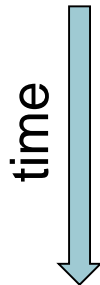


Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	Miss:valid	Mem[0]			
8	0	Miss:tag	Mem[8]			
0	0	Miss:tag	Mem[0]			
6	2	Miss:valid	Mem[0]		Mem[6]	
8	0	Miss:tag	Mem[8]		Mem[6]	

# Associativity Example

2-way set associative

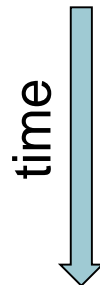
(4 miss, 1 hit)



Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	Miss:valid	<b>Mem[0]</b>			
8	0	Miss:valid	Mem[0]	<b>Mem[8]</b>		
0	0	Hit	<b>Mem[0]</b>	Mem[8]		
6	0	Miss:tag	Mem[0]	<b>Mem[6]</b>		
8	0	Miss:tag	<b>Mem[8]</b>	Mem[6]		

Fully associative

(3 miss, 2 hit)



Block address		Hit/miss	Cache content after access			
0		miss:valid	<b>Mem[0]</b>			
8		Miss:valid	Mem[0]	<b>Mem[8]</b>		
0		hit	<b>Mem[0]</b>	Mem[8]		
6		Miss:valid	Mem[0]	Mem[8]	<b>Mem[6]</b>	
8		hit	Mem[0]	<b>Mem[8]</b>	Mem[6]	

# How Much Associativity?

- Increased associativity reduces miss rate
  - But with diminishing returns
  - Best gain is when going from direct-mapped to 2-way
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - Direct: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%



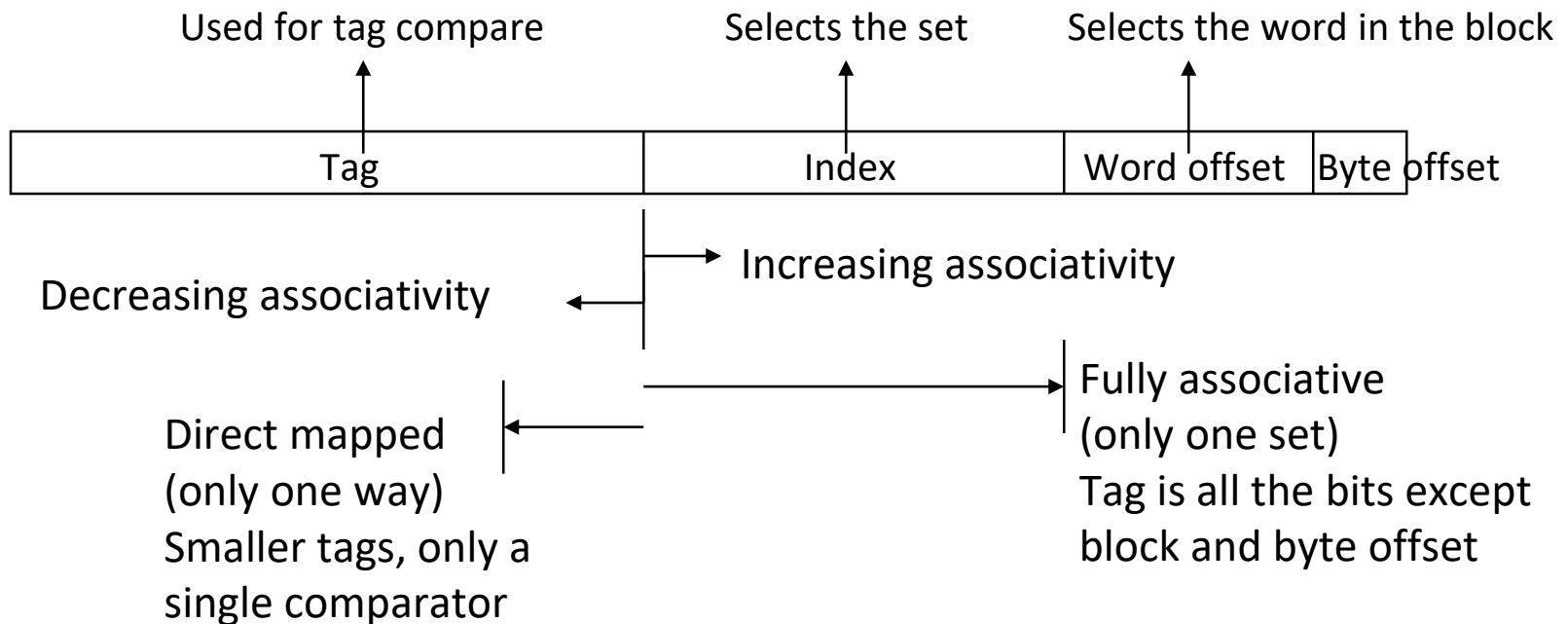
# Range of Set-Associative Caches

- For a fixed-size cache, 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, 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



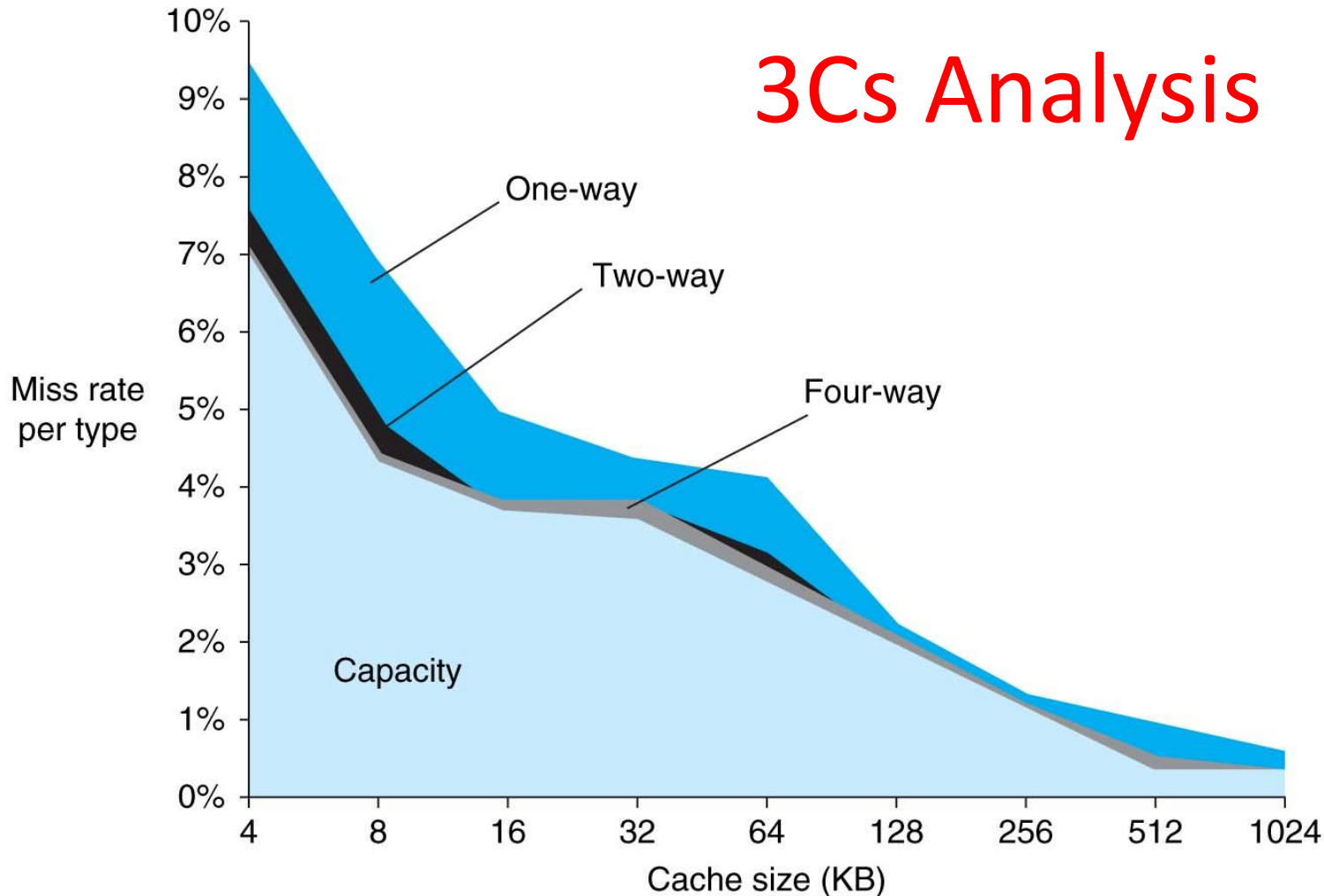
# Understanding Cache Misses: The 3Cs

- **Compulsory** (cold start or process migration, 1<sup>st</sup> reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- **Capacity**:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- **Conflict** (*collision*):
  - *Multiple memory locations mapped to the same cache location*
  - *Solution 1: increase cache size*
  - *Solution 2: increase associativity (may increase access time)*

# How to Calculate 3C's using Cache Simulator

1. *Compulsory*: set cache size to infinity and fully associative, and count number of misses
2. *Capacity*: Change cache size from infinity, usually in powers of 2, and count misses for each reduction in size
  - 16 MB, 8 MB, 4 MB, ... 128 KB, 64 KB, 16 KB
3. *Conflict*: Change from fully associative to n-way set associative while counting misses
  - Fully associative, 16-way, 8-way, 4-way, 2-way, 1-way

# 3Cs Analysis



- Three sources of misses (SPEC2000 integer and floating-point benchmarks)
  - Compulsory misses 0.006%; not visible
  - Capacity misses, function of cache size
  - Conflict portion depends on associativity and cache size

# Improving Cache Performance

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

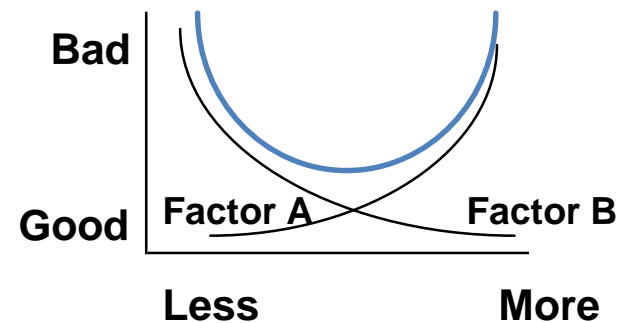
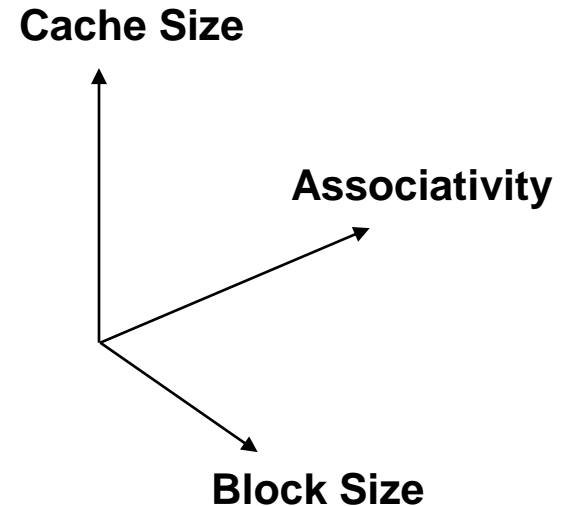
- Reduce the time to hit in the cache
  - E.g., Smaller cache
- Reduce the miss rate
  - E.g., Bigger cache
- Reduce the miss penalty
  - E.g., Use multiple cache levels

# Impact of Larger Cache on AMAT?

- 1) Reduces misses (what kind(s)?)
- 2) Longer Access time (Hit time): smaller is faster
  - Increase in hit time will likely add another stage to the pipeline
- At some point, increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance
- Computer architects expend considerable effort optimizing organization of cache hierarchy – big impact on performance and power!

# Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write allocation
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins





# Increasing Block Size?

- Hit time as block size increases?
  - Hit time unchanged, but might be slight hit-time reduction as number of tags is reduced, so faster to access memory holding tags
- Miss rate as block size increases?
  - Goes down at first due to spatial locality, then increases due to increased conflict misses due to fewer blocks in cache
- Miss penalty as block size increases?
  - Rises with longer block size, but with fixed constant initial latency that is amortized over whole block

# Increasing Associativity?

- Hit time as associativity increases?
  - Increases, with large step from direct-mapped to  $\geq 2$  ways, as now need to mux correct way to processor
  - Smaller increases in hit time for further increases in associativity
- Miss rate as associativity increases?
  - Goes down due to reduced conflict misses, but most gain is from 1- $\rightarrow$ 2- $\rightarrow$ 4-way with limited benefit from higher associativities
- Miss penalty as associativity increases?
  - Unchanged, replacement policy runs in parallel with fetching missing line from memory

# Increasing #Entries?

- Hit time as #entries increases?
  - Increases, since reading tags and data from larger memory structures
- Miss rate as #entries increases?
  - Goes down due to reduced capacity and conflict misses
  - *Architects rule of thumb: miss rate drops  $\sim 2x$  for every  $\sim 4x$  increase in capacity (only a gross approximation)*
- Miss penalty as #entries increases?
  - Unchanged

# Impact of longer cache blocks on misses?

- For fixed total cache capacity and associativity, what is effect of longer blocks on each type of miss:
  - A: Decrease, B: Unchanged, C: Increase
- Compulsory?
- Capacity?
- Conflict?

# Impact of longer blocks on AMAT

- For fixed total cache capacity and associativity, what is effect of longer blocks on each component of AMAT:
  - A: Decrease, B: Unchanged, C: Increase
- Hit Time?
- Miss Rate?
- Miss Penalty?

# Multi-level Cache

- **Goal:** Better cache design for faster CPU
- **Method:** Engage multiple levels of cache
- **L1 (primary) cache:** attached to CPU
  - Small, but very fast
  - Focused on **reducing hit time**
    - ➔ Attain smaller cycle time, or fewer pipeline stages
- **L2 cache:** services L1 cache misses
  - Larger & slower than L1, but faster than main memory
    - Has bigger blocks, more associativity
  - Focused on **reducing miss rate**
- **Main memory:** services L2 cache misses
  - Some high-end systems even have L3 cache

# Multi-level Cache Example

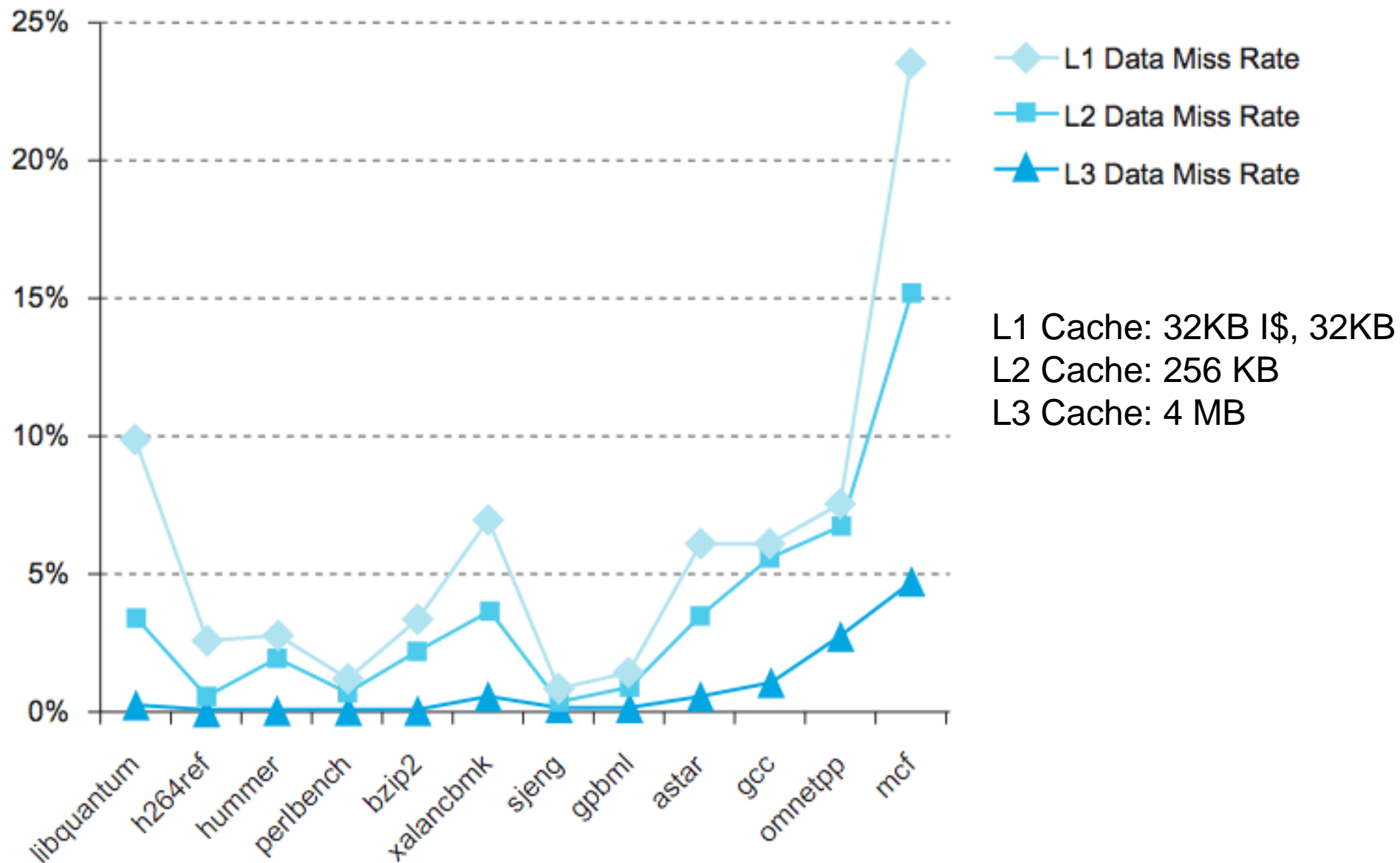
- Rates
  - CPU base CPI = 1, clock rate = 4GHz
  - Cache miss rate = 2%
  - Main memory access time = 100ns
- What is the Effective CPI?
- I. Use only single-level cache
  - Clock cycle =  $1/4\text{GHz} = 0.25\text{ns}$
  - Miss penalty =  $100[\text{ns}]/0.25[\text{ns/cycle}] = 400 \text{ cycles}$
  - Effective CPI =  $1 + 0.02 \times 400 = 9$

# Mutli-level Cache Example (cont.)

## II. Now add L2 cache

- L2 access time = 5ns
- Global miss rate to main memory = 0.5%
- L1 miss with L2 hit
  - Penalty =  $5\text{ns}/0.25(\text{cycle time}) = 20 \text{ cycles}$
- L1 miss with L2 miss
  - Extra penalty = 400 cycles (same as before)
- Effective CPI =  $1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- 2-level cache is  $9/3.4=2.6$  times faster than single level





**FIGURE 5.47** The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPEC CPU2006 benchmarks.

# Local vs. Global Miss Rates

- *Local miss rate* – the fraction of references to one level of a cache that miss
- Local Miss rate L2\$ = L2\$ Misses / L1\$ Misses
- *Global miss rate* – the fraction of references that miss in all levels of a multilevel cache
  - L2\$ local miss rate >> than the global miss rate
- Global Miss rate = L2\$ Misses / Total Accesses  
= L2\$ Misses / L1\$ Misses x L1\$ Misses / Total Accesses  
= Local Miss rate L2\$ x Local Miss rate L1\$
- AMAT = Time for a hit + Miss rate x Miss penalty
- AMAT = Time for a L1\$ hit + (local) Miss rate L1\$ x (Time for a L2\$ hit + (local) Miss rate L2\$ x L2\$ Miss penalty)

## And In Conclusion, ...

- Principle of Locality
- Hierarchy of Memories (speed/size/cost per bit) to Exploit Locality
- Cache – copy of data lower level in memory hierarchy
- Direct Mapped to find block in cache using Tag field and Valid bit for Hit
- Cache design choice:
  - Write-Through vs. Write-Back

# And, In Conclusion ...

- Name of the Game: Reduce AMAT
  - Reduce Hit Time
  - Reduce Miss Rate
  - Reduce Miss Penalty
- Balance cache parameters

# Primary Cache Parameters

- Block size (aka line size)
  - how many bytes of data in each cache entry?
- Associativity
  - how many ways in each set?
  - Direct-mapped  $\Rightarrow$  Associativity = 1
  - Set-associative  $\Rightarrow 1 < \text{Associativity} < \text{\#Entries}$
  - Fully associative  $\Rightarrow \text{Associativity} = \text{\#Entries}$
- Capacity (bytes) = Total  $\text{\#Entries}$  \* Block size
- $\text{\#Entries} = \text{\#Sets} * \text{Associativity}$

# Other Cache Parameters

- Write Policy
- Replacement policy