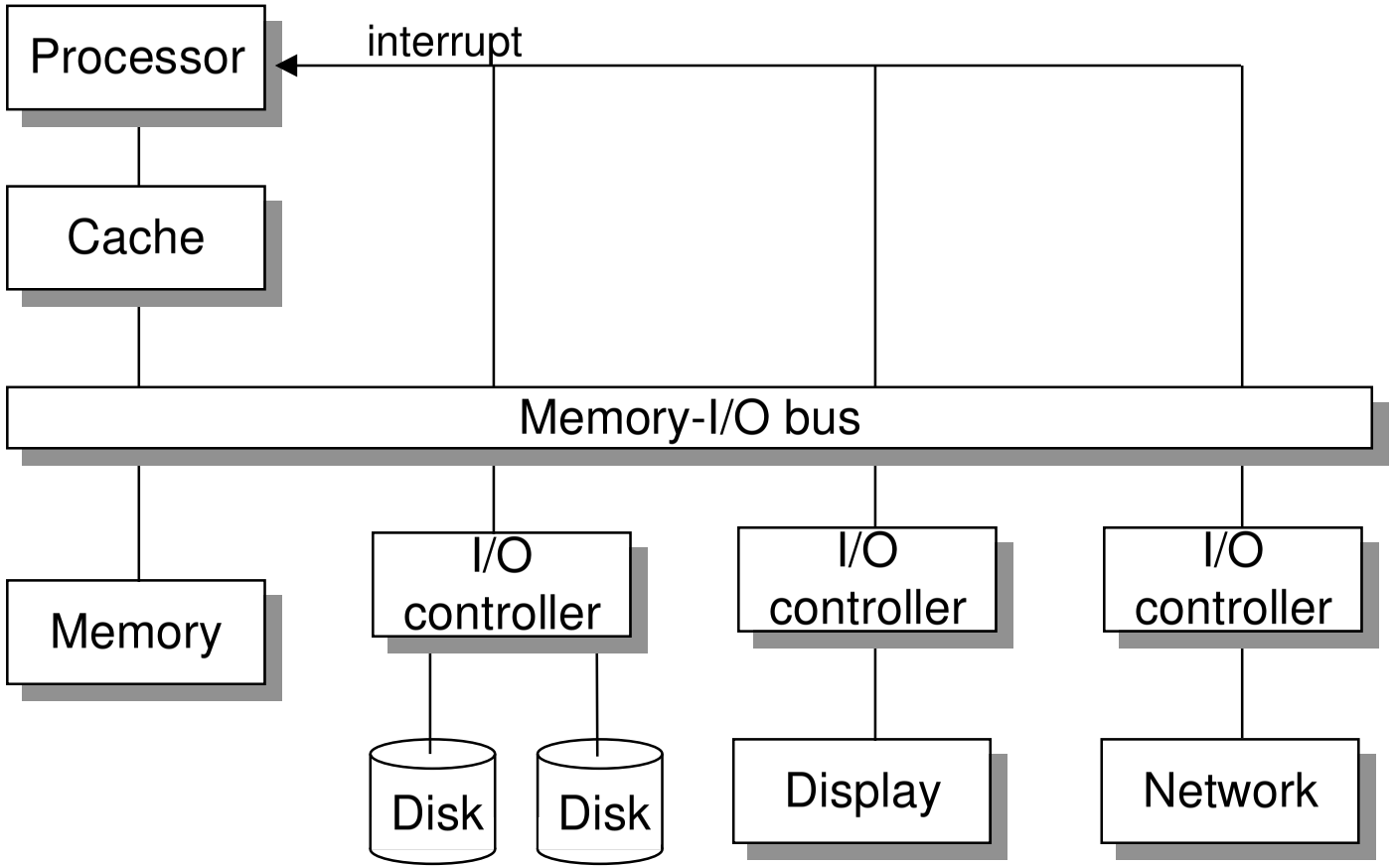
15-213

Caches
Oct. 22, 1998

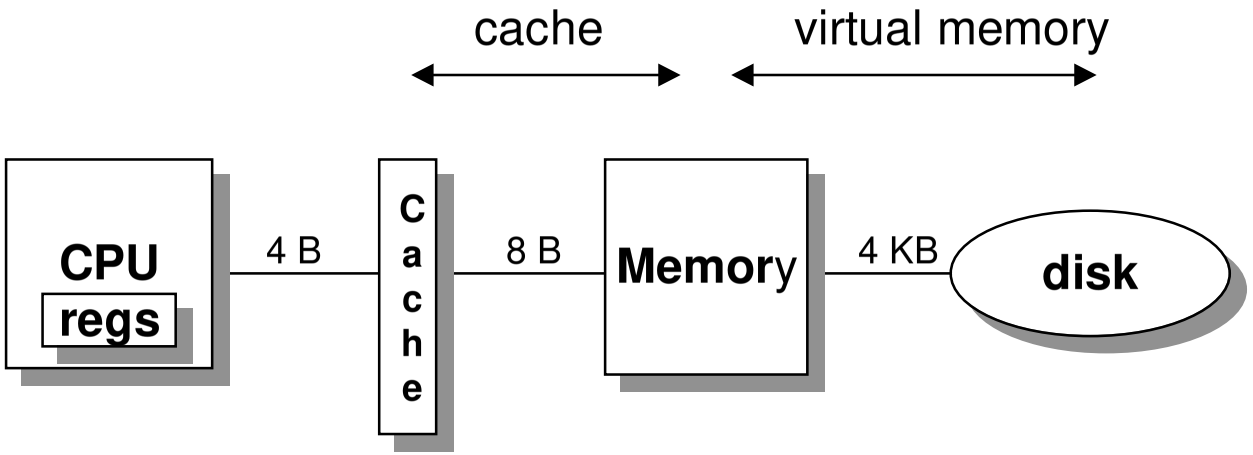
Topics

- **Memory Hierarchy**
- **Locality of Reference**
- **Cache Design**
  - Direct Mapped
  - Associative

Computer System



Levels in Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	200 B	32 KB / 4MB	128 MB	20 GB
speed:	3 ns	6 ns	100 ns	10 ms
$/Mbyte:		$100/MB	$1.50/MB	$0.06/MB
block size:	4 B	8 B	4 KB	

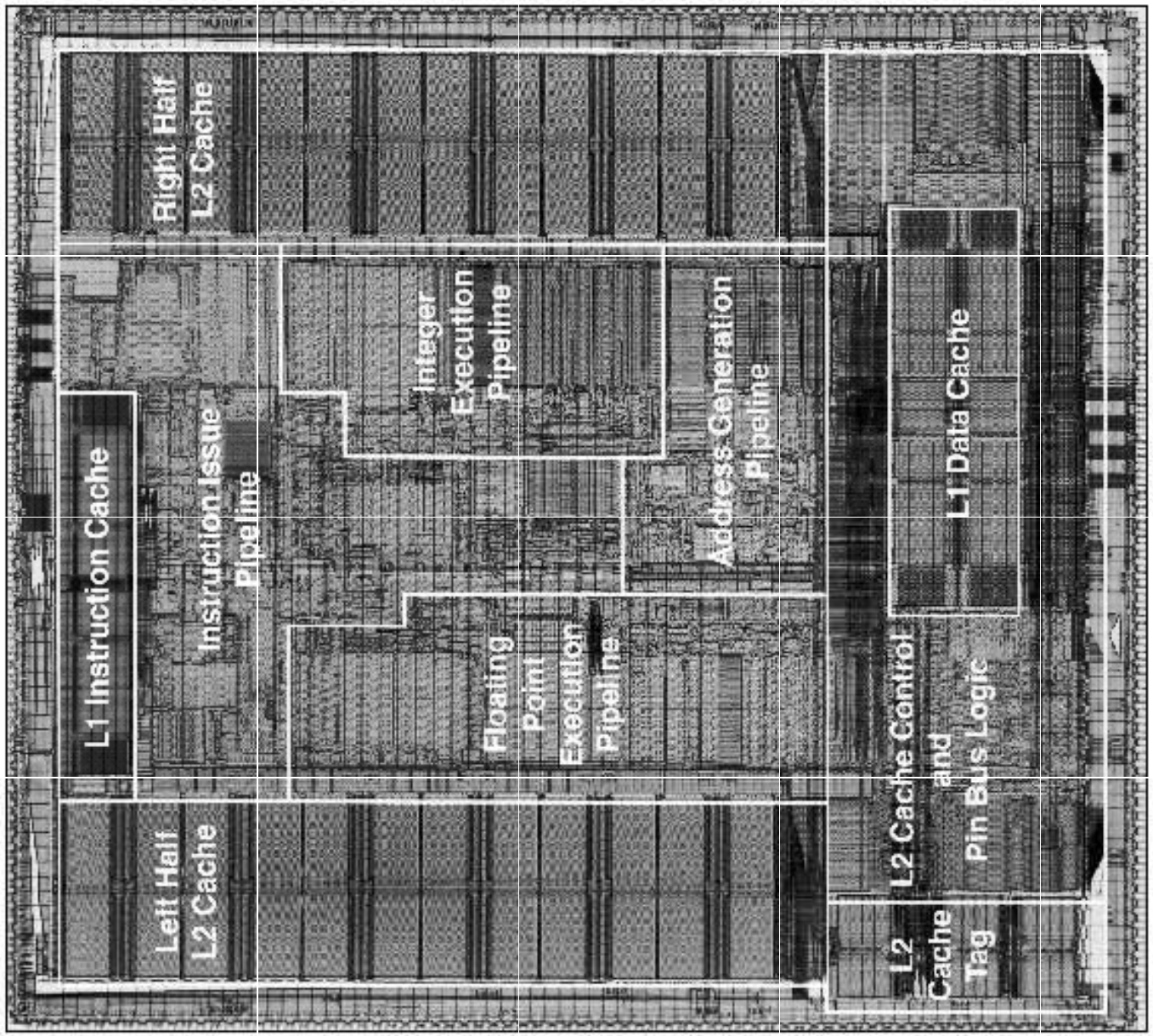
larger, slower, cheaper

Alpha 21164 Chip Photo

Microprocessor Report 9/12/94

Caches:

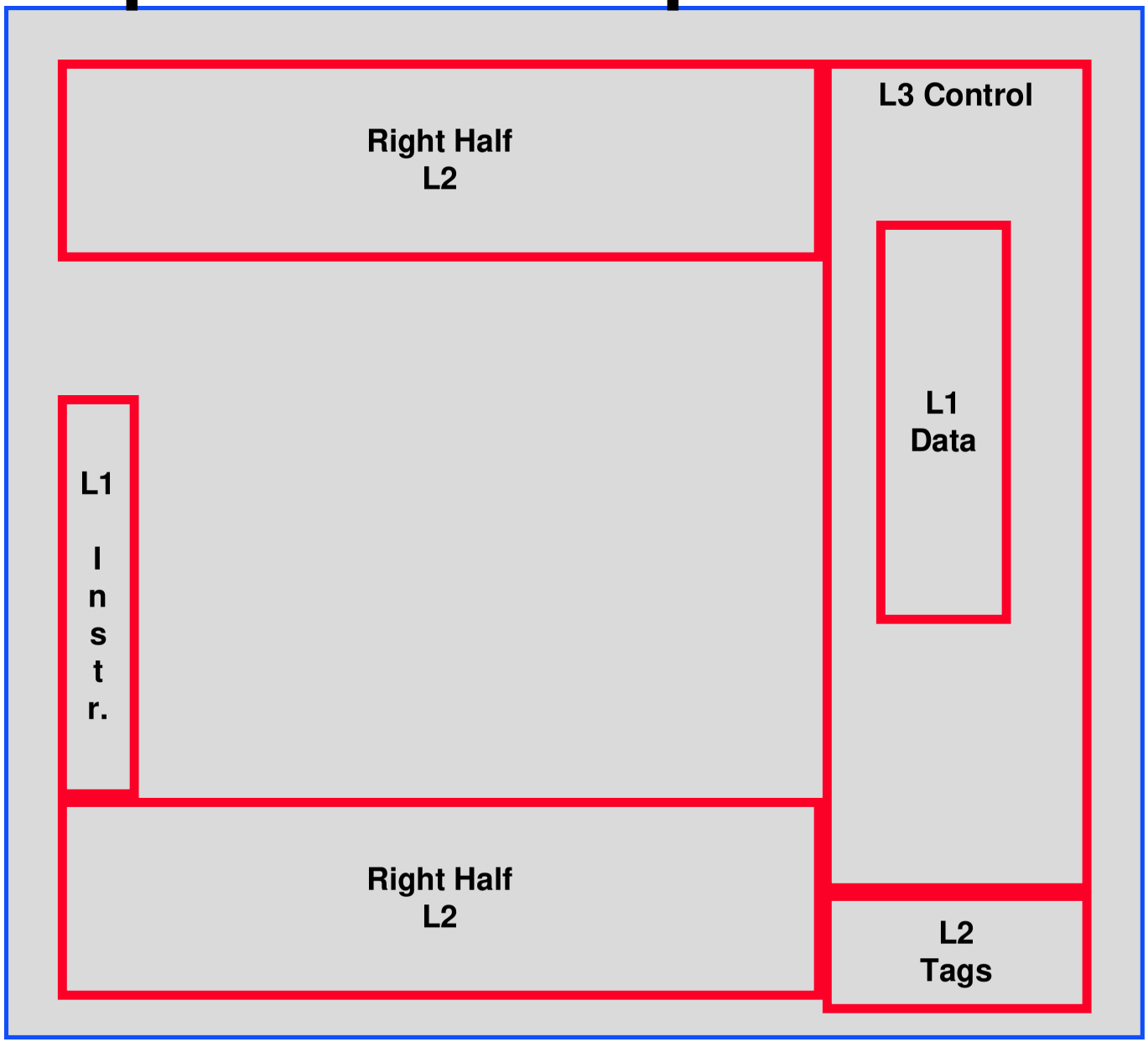
- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history



Alpha 21164 Chip Caches

Caches:

- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history



Locality of Reference

Principle of Locality

- **Programs tend to reuse data and instructions near those they have used recently.**
- ***Temporal locality:*** **recently referenced items are likely to be referenced in the near future.**
- ***Spatial locality:*** **items with nearby addresses tend to be referenced close together in time.**

Locality in Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- **Data**
  - Reference array elements in succession (spatial)
- **Instruction**
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)

Caching: The Basic Idea

Main Memory

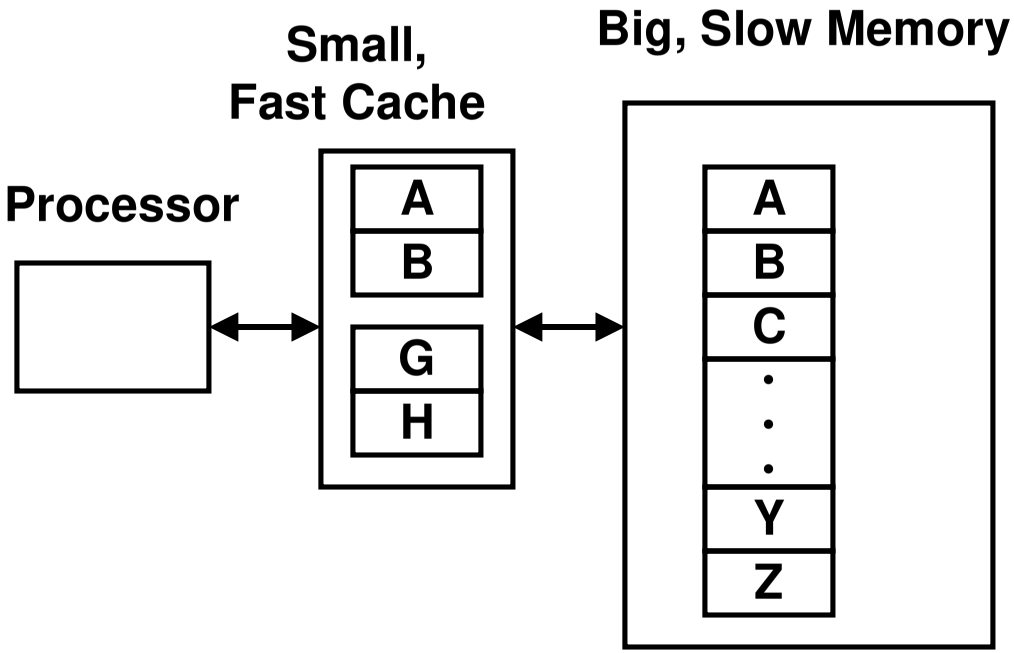
- **Stores words**
  A–Z in example

Cache

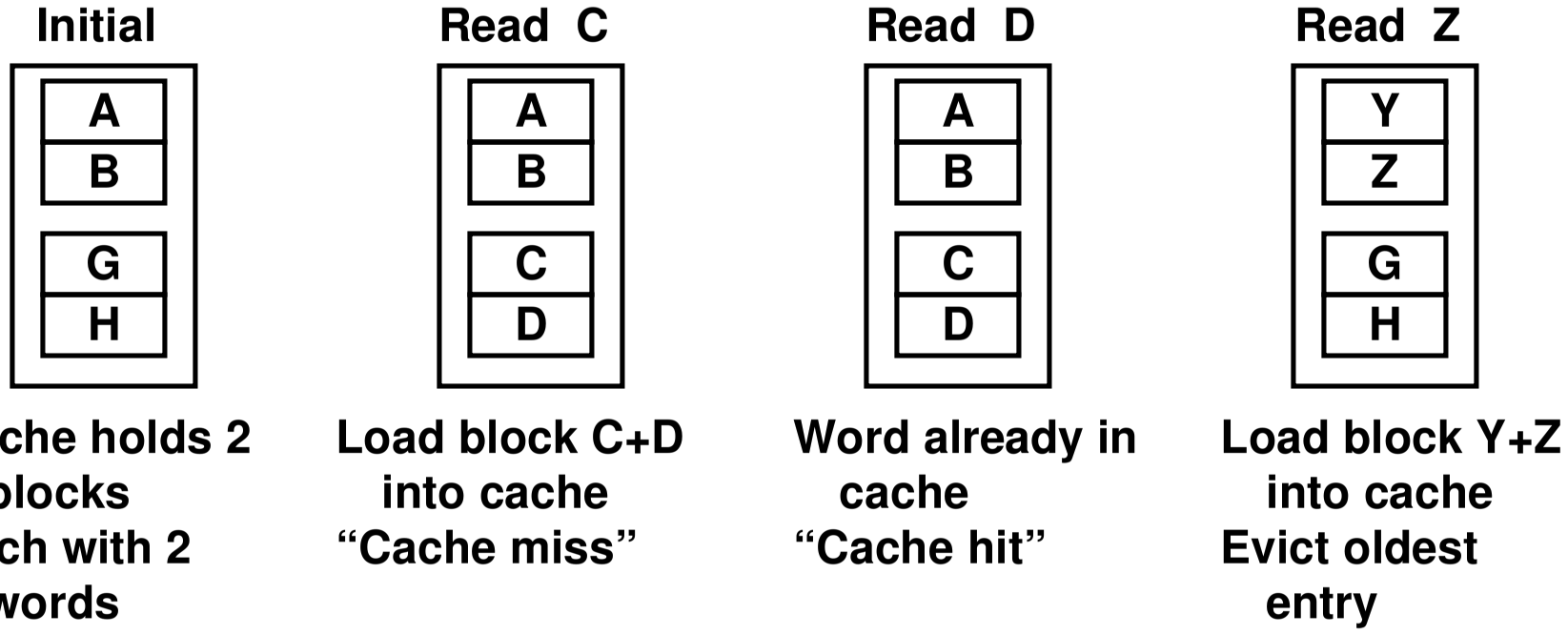
- **Stores subset of the words**
  4 in example
- **Organized in blocks**
  - Multiple words
  - To exploit spatial locality

Access

- **Word must be in cache for processor to access**



Basic Idea (Cont.)



Cache holds 2 blocks
Each with 2 words

Load block C+D into cache
"Cache miss"

Word already in cache
"Cache hit"

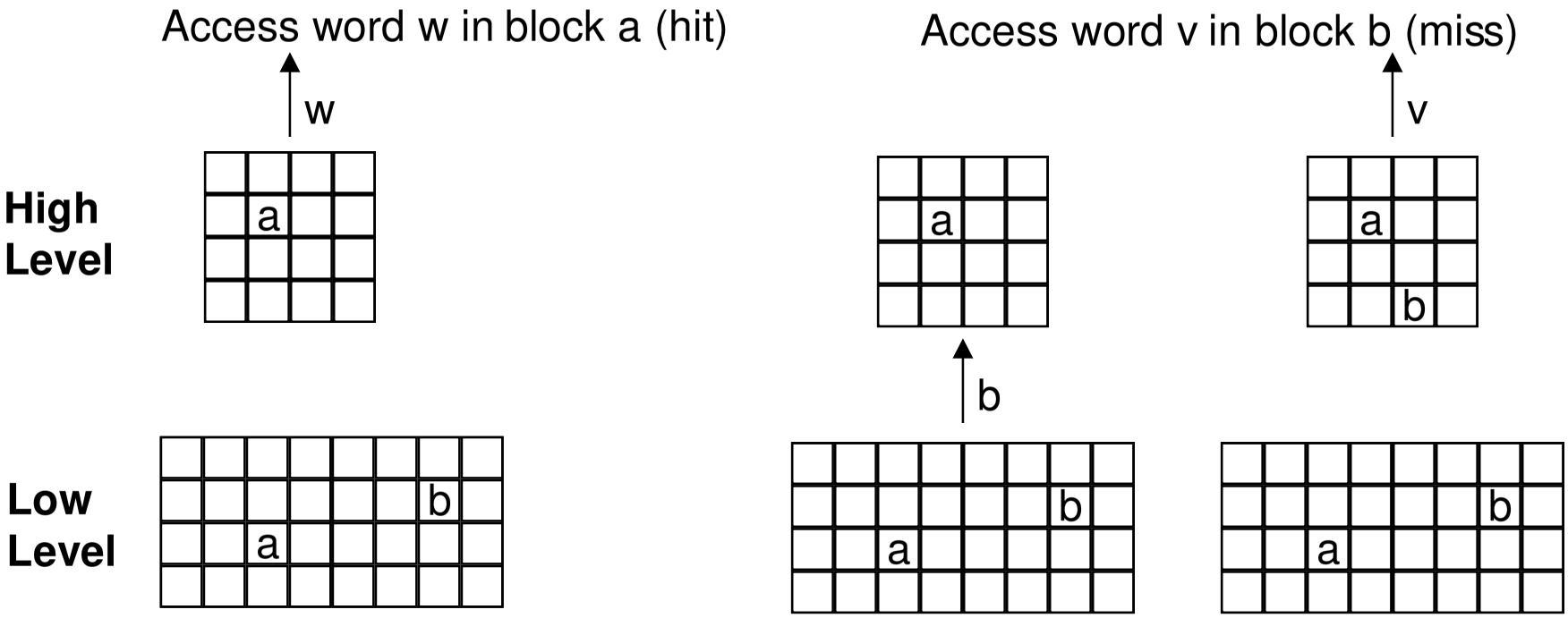
Load block Y+Z into cache
Evict oldest entry

Maintaining Cache

- **Every time processor performs load or store, bring block containing word into cache**
  - May need to evict existing block
- **Subsequent loads or stores to any word in block performed within cache**

Accessing Data in Memory Hierarchy

- **Between any two levels, memory divided into blocks.**
- **Data moves between levels on demand, in block-sized chunks.**
- **Invisible to application programmer**
  - Hardware responsible for cache operation
- **Upper-level blocks a subset of lower-level blocks.**



Design Issues for Caches

Key Questions

- **Where should a block be placed in the cache? (block placement)**
- **How is a block found in the cache? (block identification)**
- **Which block should be replaced on a miss? (block replacement)**
- **What happens on a write? (write strategy)**

Constraints

- **Design must be very simple**
  - Hardware realization
  - All decision making within nanosecond time scale
- **Want to optimize performance for "typical" programs**
  - Do extensive benchmarking and simulations
  - Many subtle engineering trade-offs

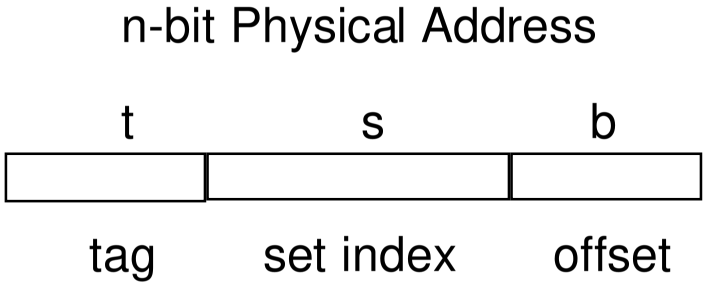
Direct-Mapped Caches

Simplest Design

- **Given memory block has unique cache location**

Parameters

- **Block size B = 2^b**
  - Number of bytes in each block
  - Typically 2X–8X word size
- **Number of Sets S = 2^s**
  - Number of blocks cache can hold
- **Total Cache Size = B*S = 2^{b+s}**

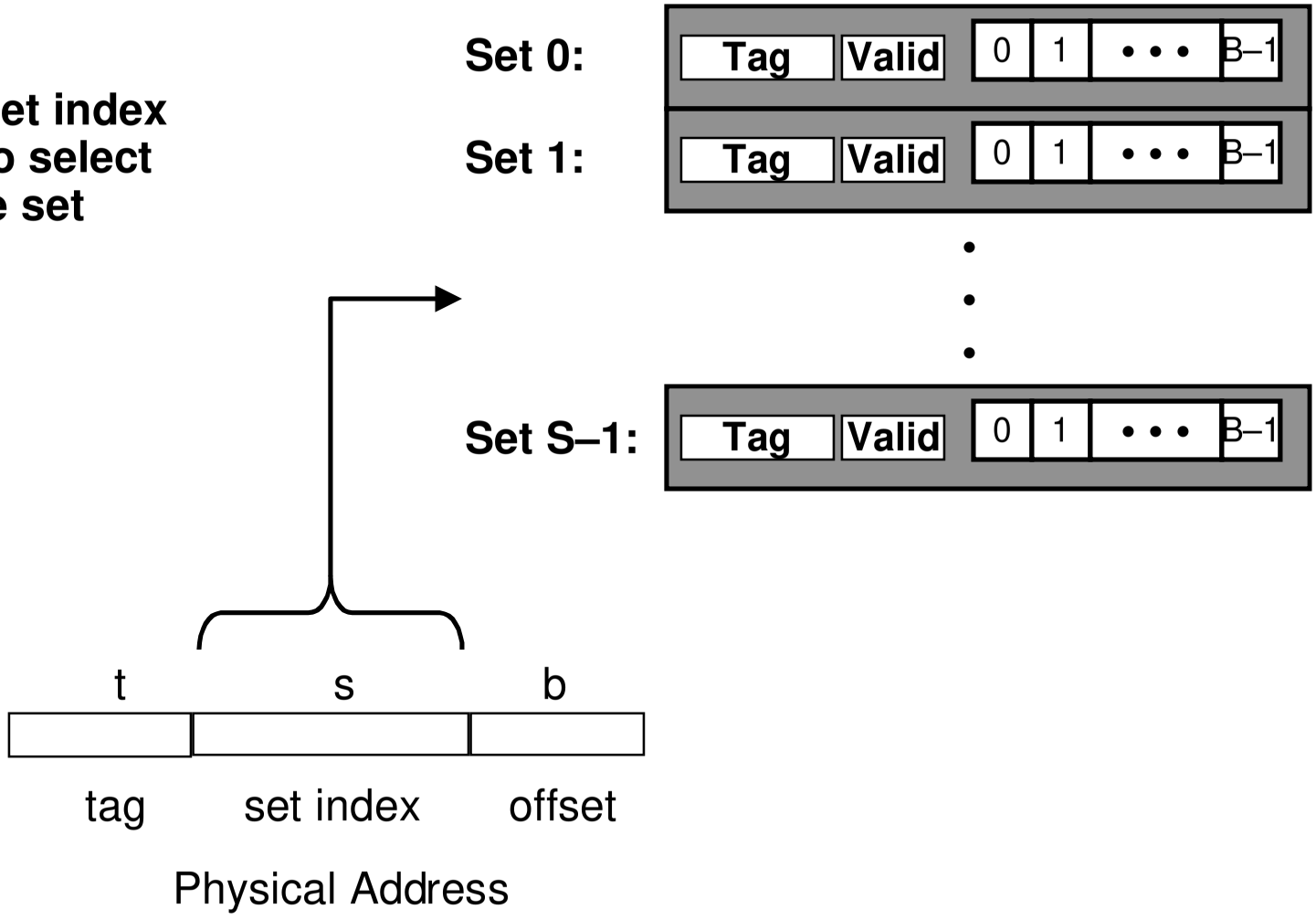


Physical Address

- **Address used to reference main memory**
- **n bits to reference N = 2^n total bytes**
- **Partition into fields**
  - Offset: Lower b bits indicate which byte within block
  - Set: Next s bits indicate how to locate block within cache
  - Tag: Identifies this block when in cache

Indexing into Direct-Mapped Cache

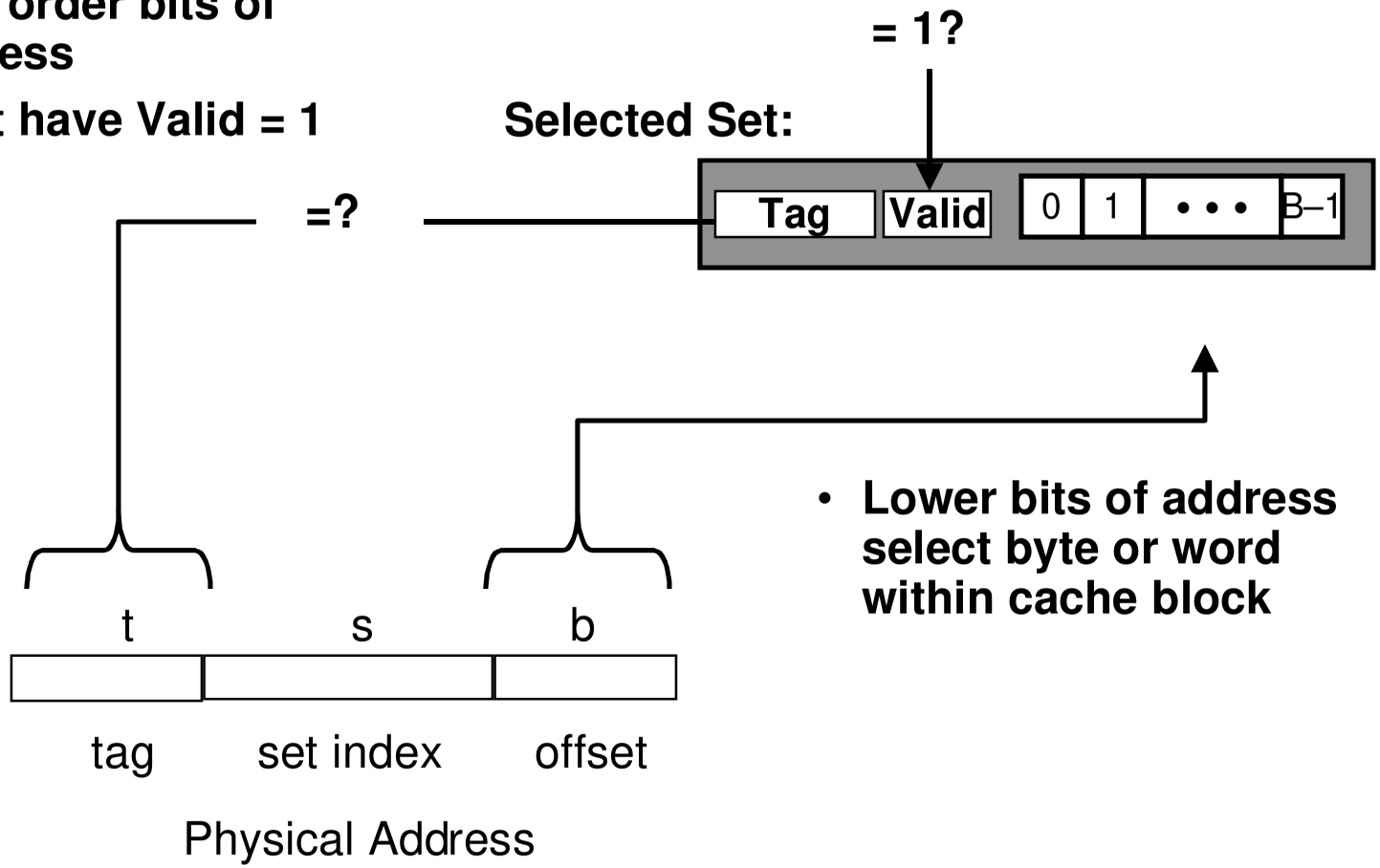
- **Use set index bits to select cache set**



Direct-Mapped Cache Tag Matching

Identifying Block

- **Must have tag match high order bits of address**
- **Must have Valid = 1**



- **Lower bits of address select byte or word within cache block**

Direct Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

N=16 byte addresses B=2 bytes/block S=4 sets E=1 entry/set

Address trace (reads):

0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

0 [0000] (miss)

(1)

v	tag	data

13 [1101] (miss)

(2)

v	tag	data

8 [1000] (miss)

(3)

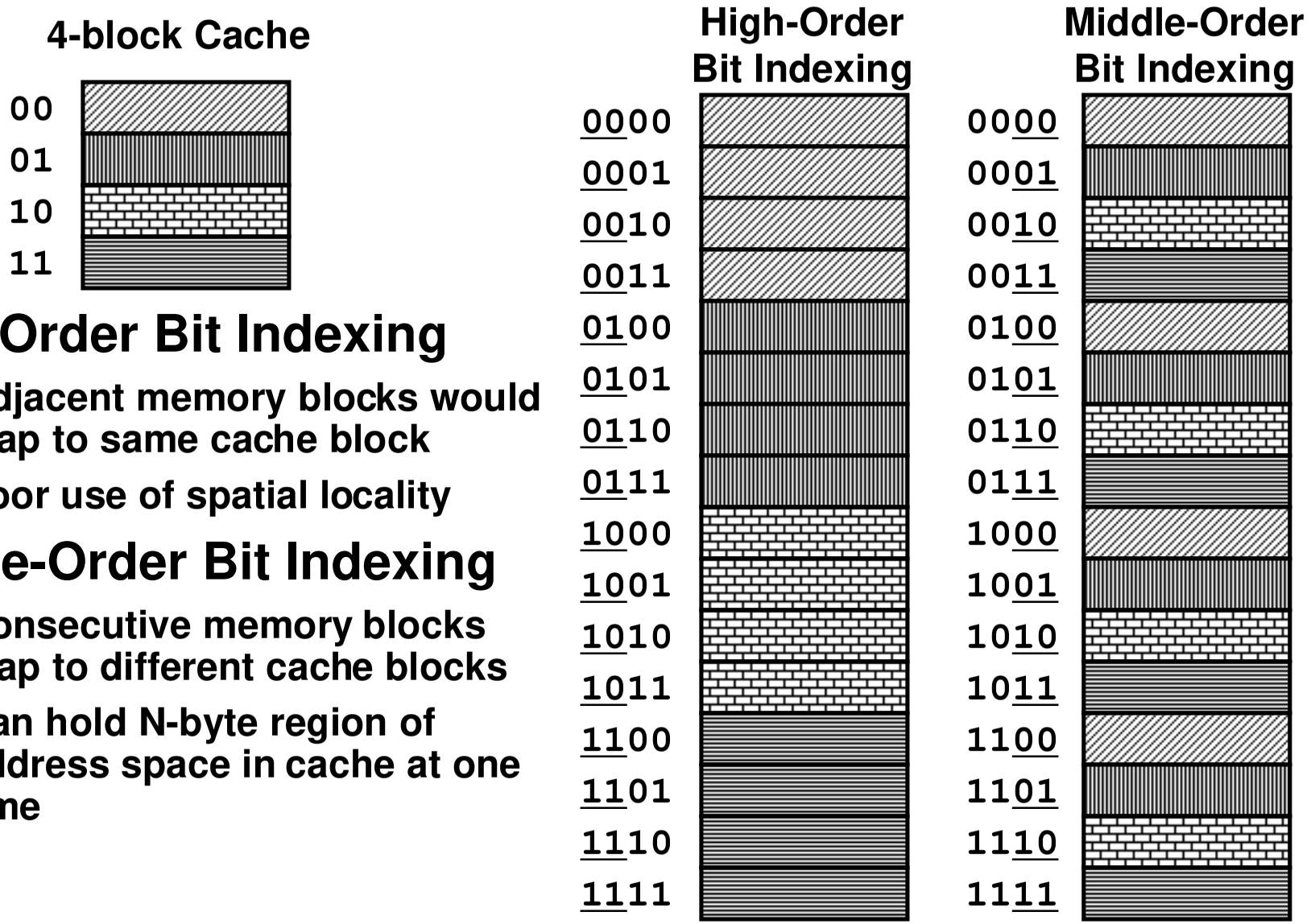
v	tag	data

0 [0000] (miss)

(4)

v	tag	data

Why Use Middle Bits as Index?



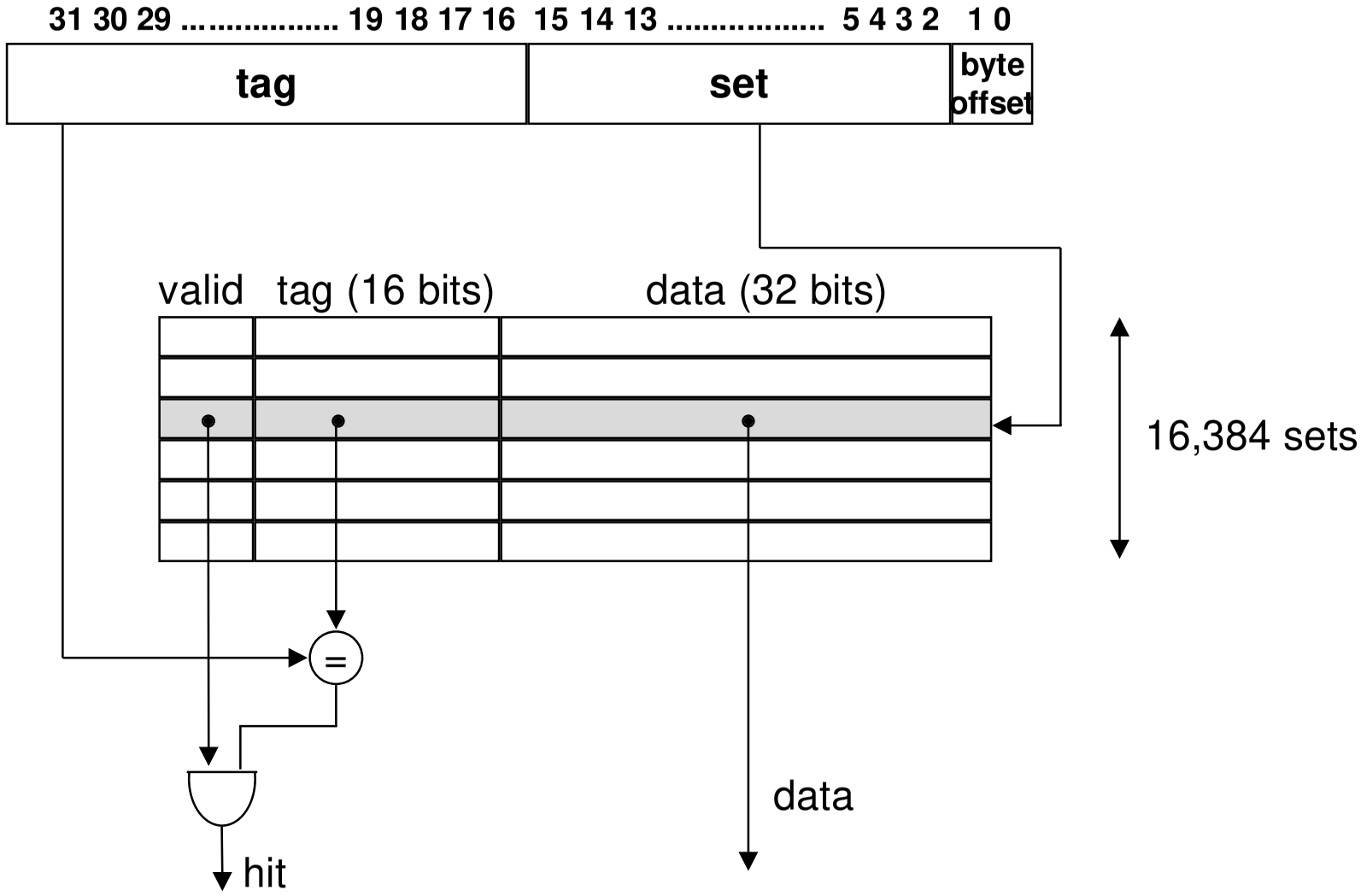
High-Order Bit Indexing

- Adjacent memory blocks would map to same cache block
- Poor use of spatial locality

Middle-Order Bit Indexing

- Consecutive memory blocks map to different cache blocks
- Can hold N-byte region of address space in cache at one time

Direct Mapped Cache Implementation (DECStation 3100)



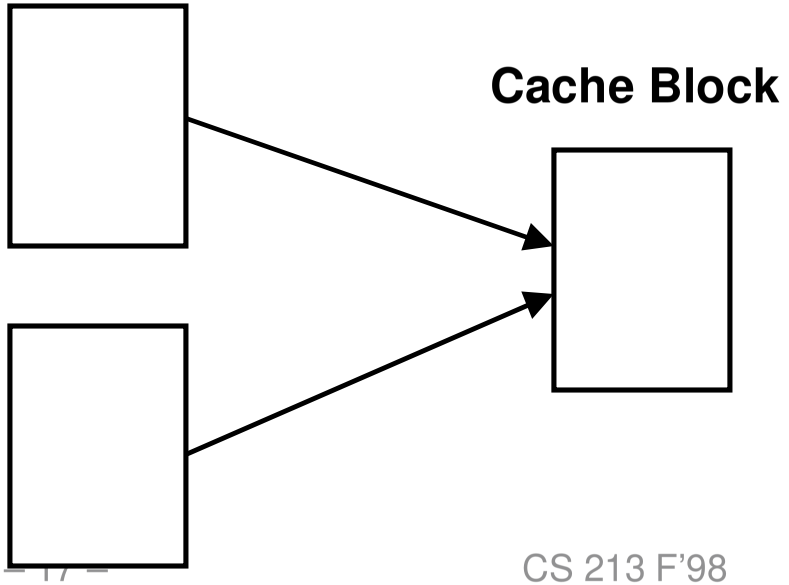
Properties of Direct Mapped Caches

Strength

- Minimal control hardware overhead
- Simple design
- (Relatively) easy to make fast

Weakness

- Vulnerable to thrashing
- Two heavily used blocks have same cache index
- Repeatedly evict one to make room for other



Vector Product Example

```
float dot_prod(float x[1024], y[1024])
{
  float sum = 0.0;
  int i;
  for (i = 0; i < 1024; i++)
    sum += x[i]*y[i];
  return sum;
}
```

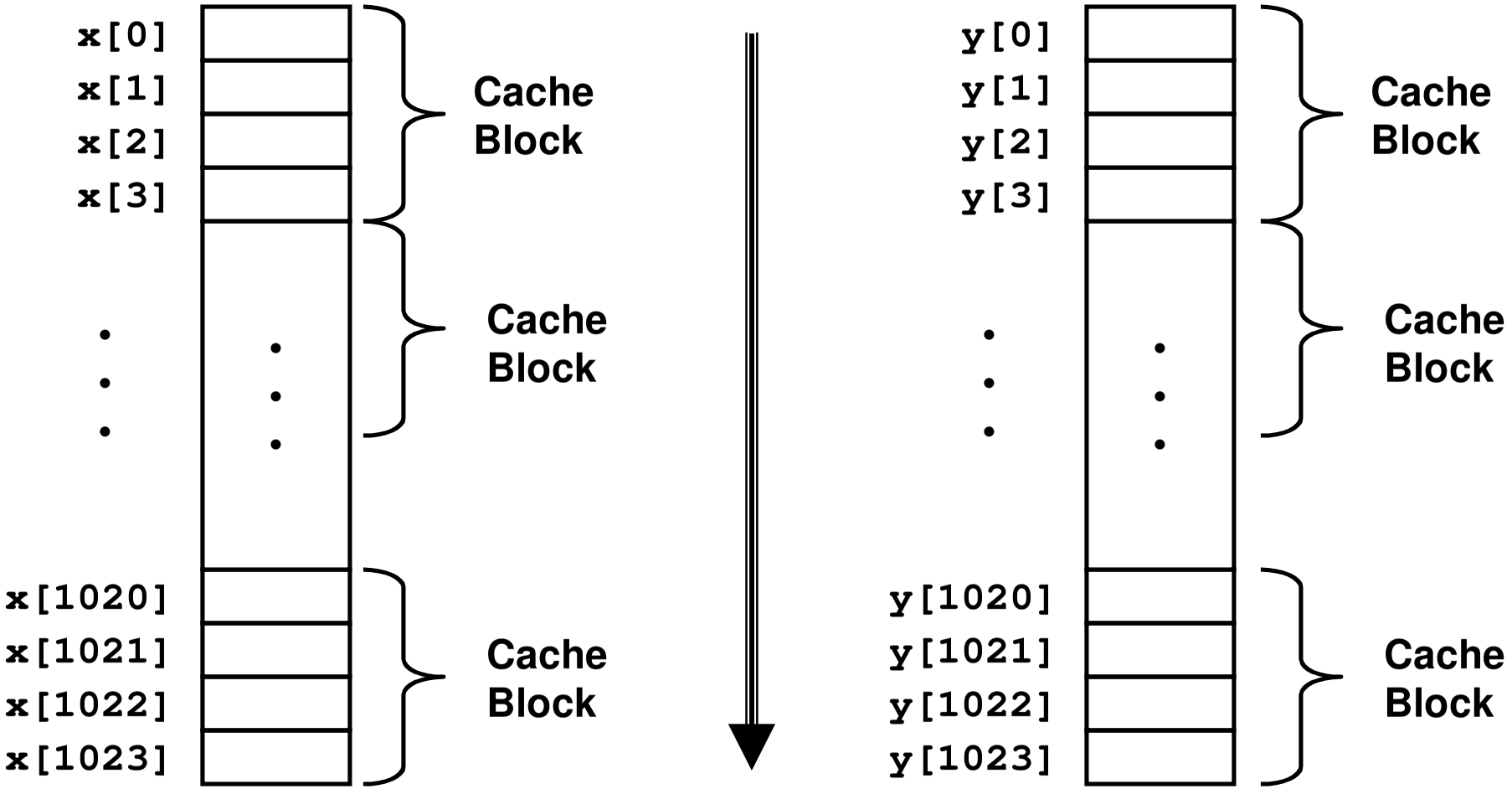
Machine

- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B block size

Performance

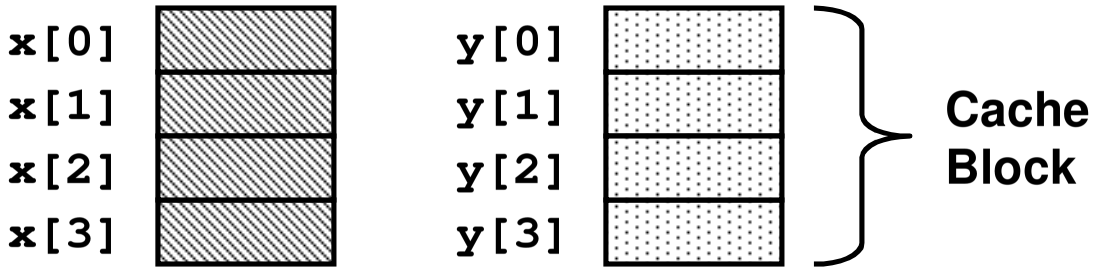
- Good case: 24 cycles / element
- Bad case: 66 cycles / element

Thrashing Example



- Access one element from each array per iteration

Thrashing Example: Good Case



Access Sequence

- **Read x[0]**
  - x[0], x[1], x[2], x[3] loaded
- **Read y[0]**
  - y[0], y[1], y[2], y[3] loaded
- **Read x[1]**
  - Hit
- **Read y[1]**
  - Hit
- **. . .**
- **2 misses / 8 reads**

Analysis

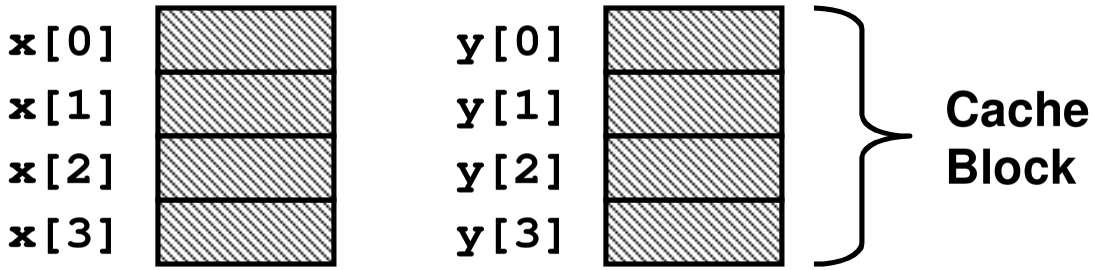
- **x[i] and y[i] map to different cache blocks**
- **Miss rate = 25%**
  - Two memory accesses / iteration
  - On every 4th iteration have two misses

Timing

- **10 cycle loop time**
- **28 cycles / cache miss**
- **Average time / iteration =**

  10 + 0.25 * 2 * 28

Thrashing Example: Bad Case



Access Pattern

- **Read x[0]**
  - x[0], x[1], x[2], x[3] loaded
- **Read y[0]**
  - y[0], y[1], y[2], y[3] loaded
- **Read x[1]**
  - x[0], x[1], x[2], x[3] loaded
- **Read y[1]**
  - y[0], y[1], y[2], y[3] loaded
- • •
- **8 misses / 8 reads**

Analysis

- **x[i] and y[i] map to same cache blocks**
- **Miss rate = 100%**
  - Two memory accesses / iteration
  - On *every* iteration have two misses

Timing

- **10 cycle loop time**
- **28 cycles / cache miss**
- **Average time / iteration =**

  10 + 1.0 * 2 * 28

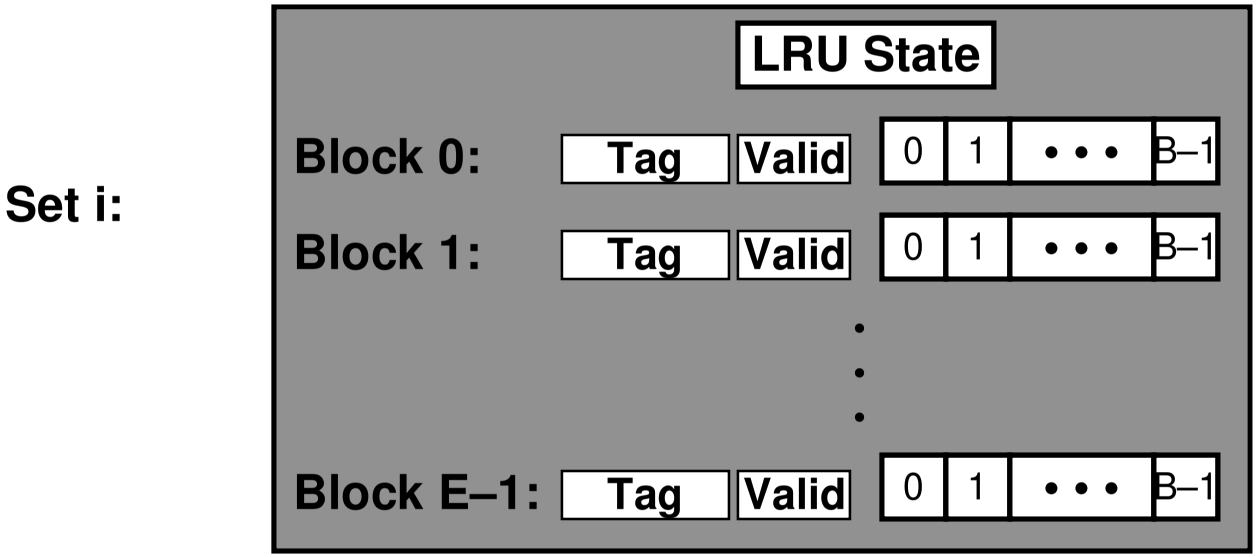
Set Associative Cache

Mapping of Memory Blocks

- **Each set can hold E blocks**
  - Typically between 2 and 8
- **Given memory block can map to any block in set**

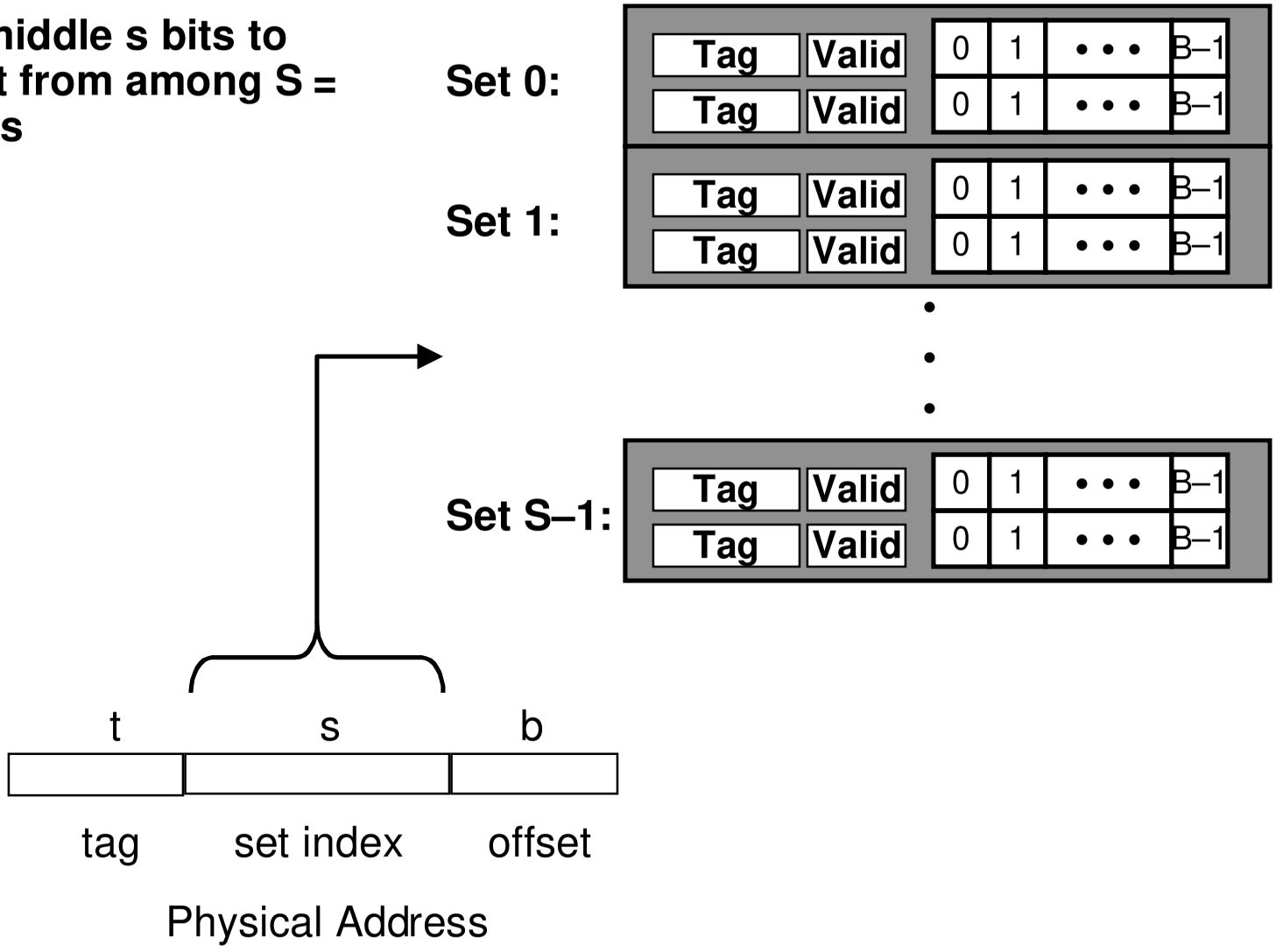
Eviction Policy

- **Which block gets kicked out when bring new block in**
- **Commonly "Least Recently Used" (LRU)**
  - Least-recently accessed (read or written) block gets evicted



Indexing into 2-Way Associative Cache

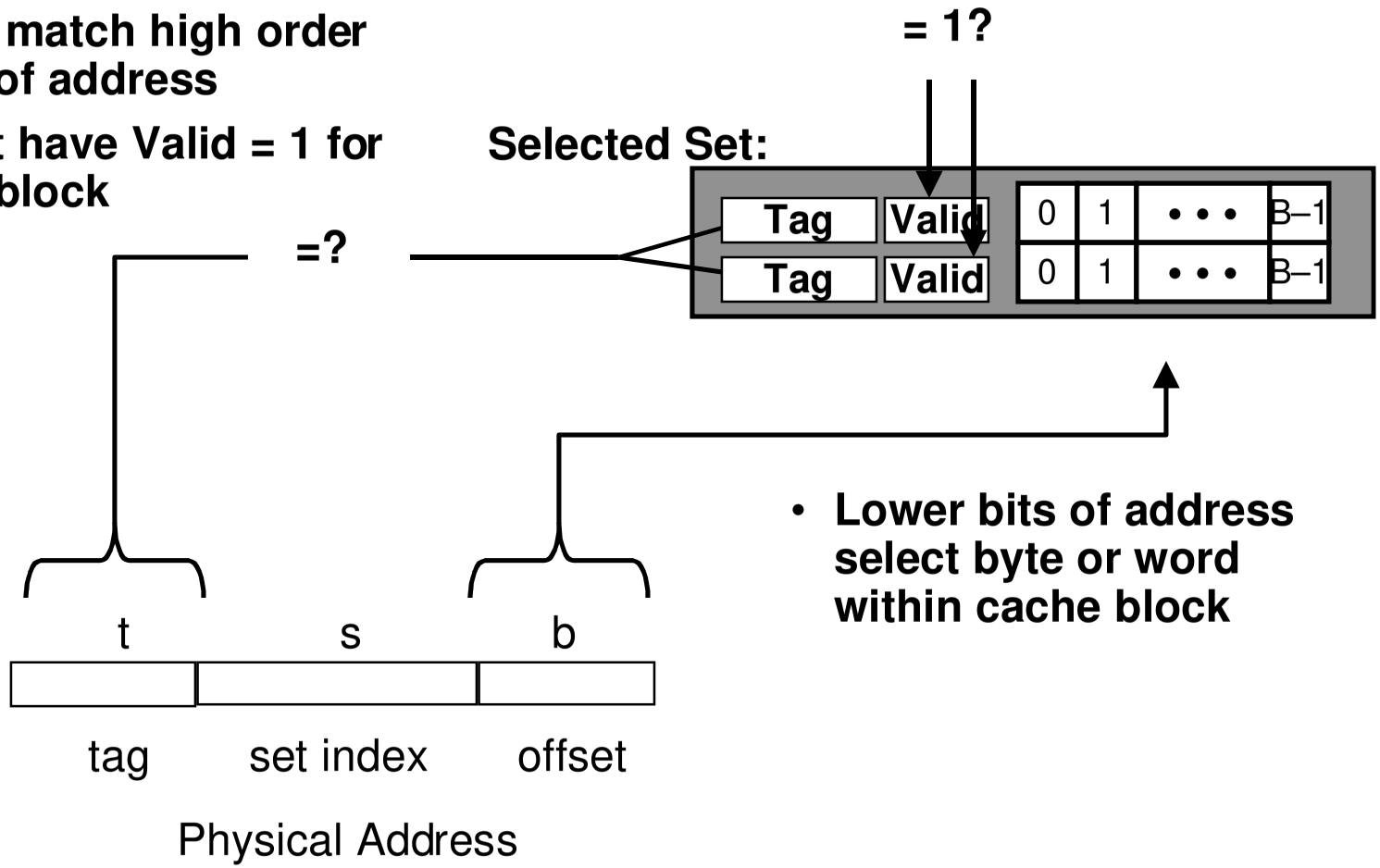
- **Use middle s bits to select from among S = 2^s sets**



2-Way Associative Cache Tag Matching

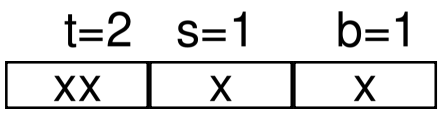
Identifying Block

- Must have one of the tags match high order bits of address
- Must have Valid = 1 for this block

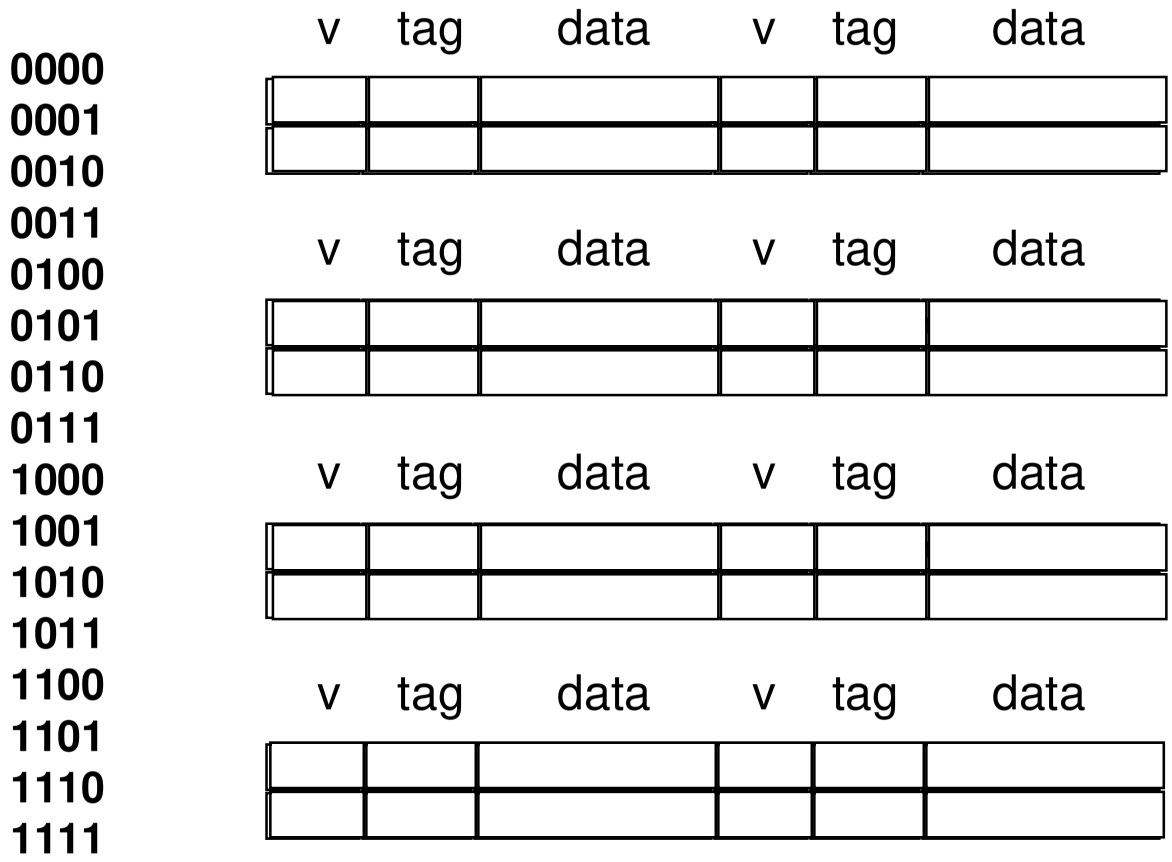


- Lower bits of address select byte or word within cache block

2-Way Set Associative Simulation



N=16 addresses B=2 bytes/line S=2 sets E=2 entries/set
Address trace (reads):
0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]



0 (miss)

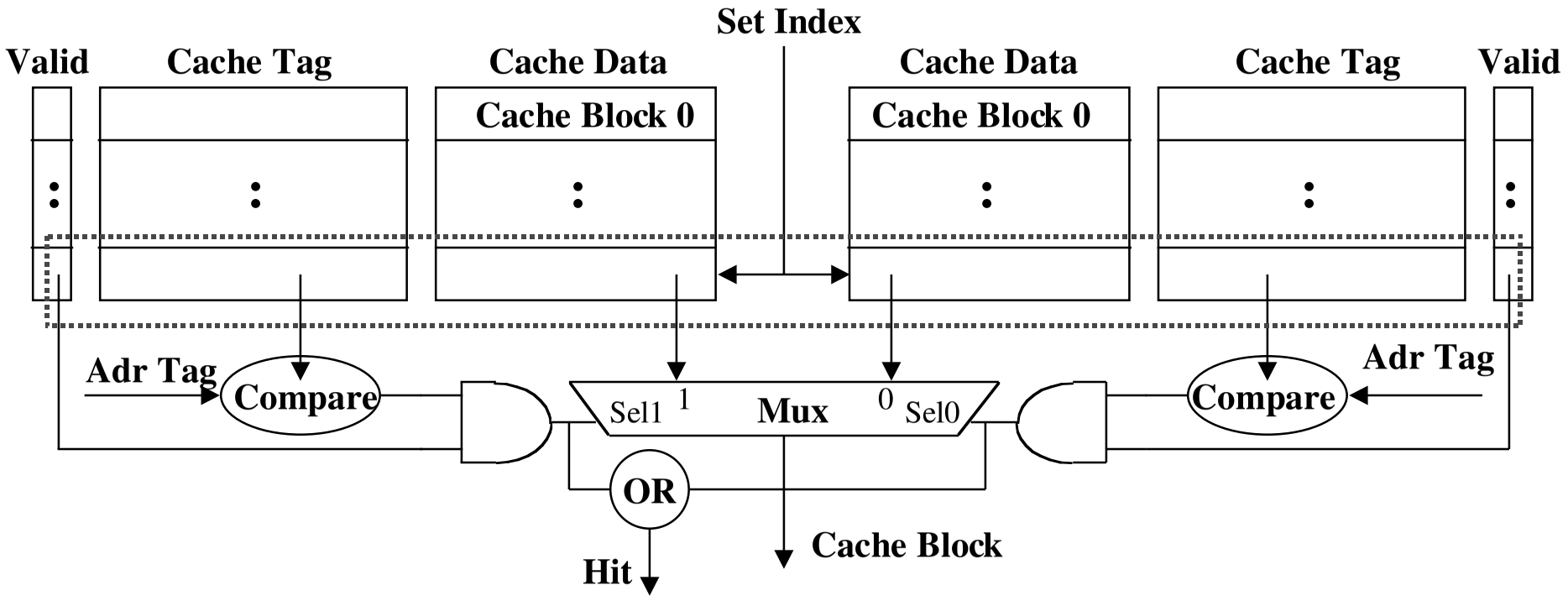
13 (miss)

8 (miss)
(LRU replacement)

0 (miss)
(LRU replacement)

Two-Way Set Associative Cache Implementation

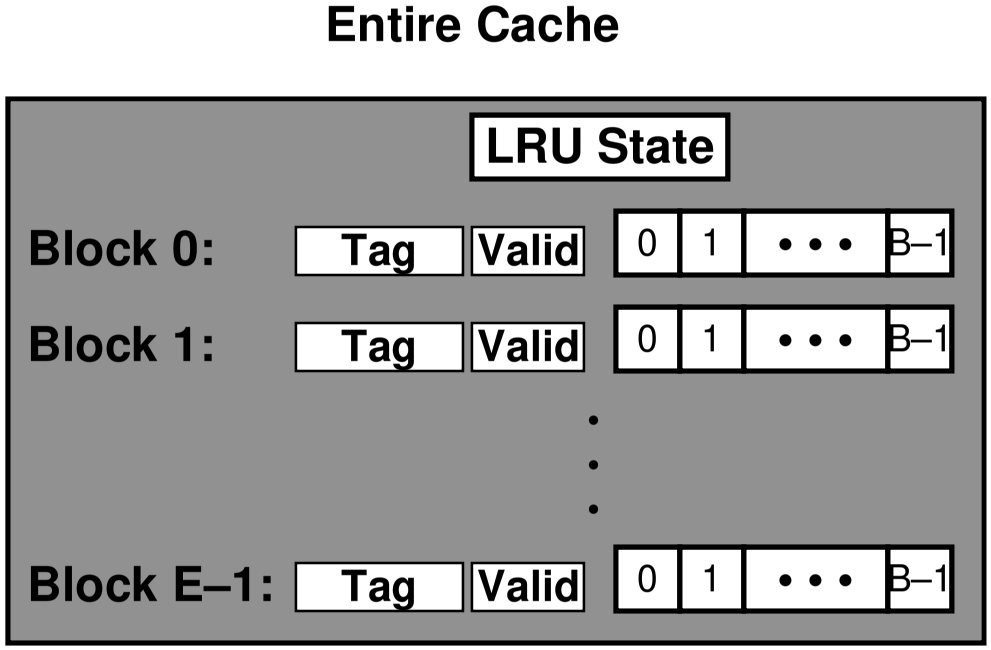
- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result



Fully Associative Cache

Mapping of Memory Blocks

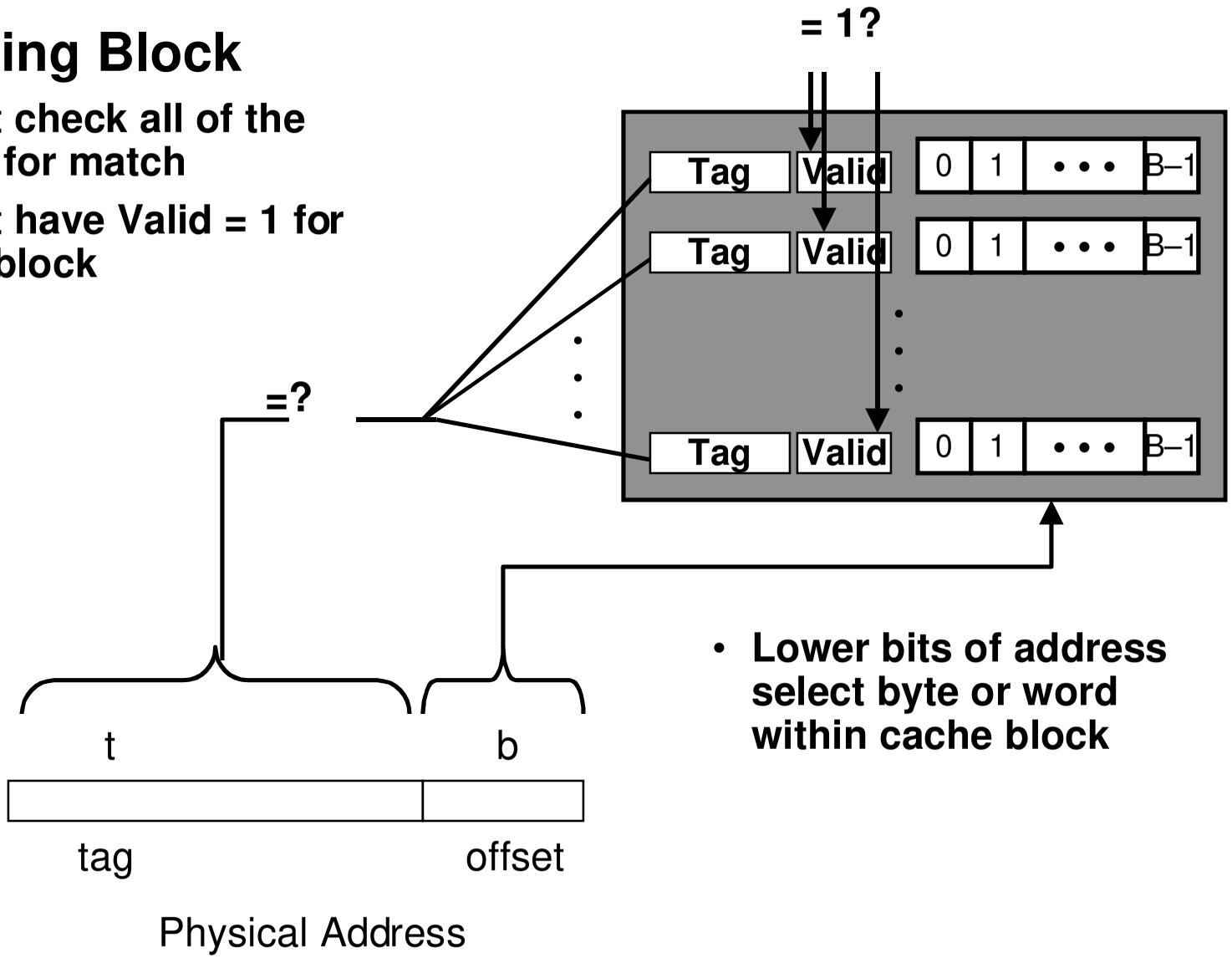
- Cache consists of single set holding E blocks
- Given memory block can map to any block in set
- Only practical for small caches



Fully Associative Cache Tag Matching

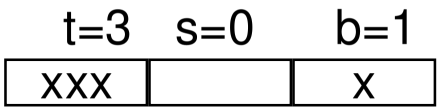
Identifying Block

- Must check all of the tags for match
- Must have Valid = 1 for this block



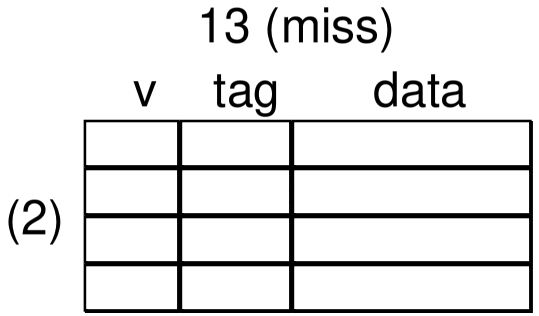
- Lower bits of address select byte or word within cache block

Fully Associative Cache Simulation



N=16 addresses B=2 bytes/line S=1 sets E=4 entries/set
Address trace (reads):
0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

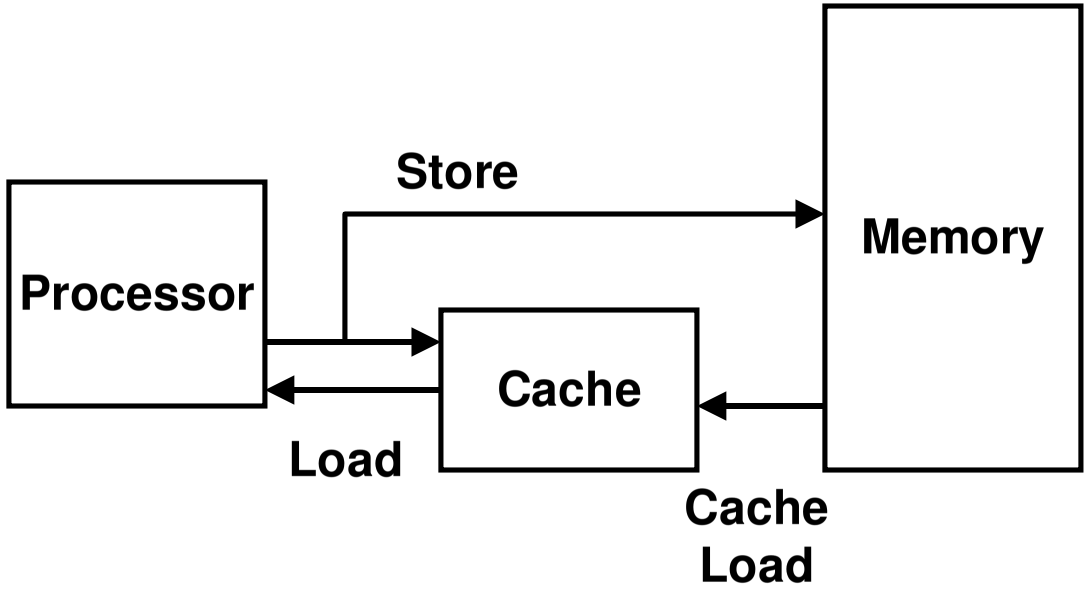


Write Policy

- What happens when processor writes to the cache?
- Should memory be updated as well?

Write Through

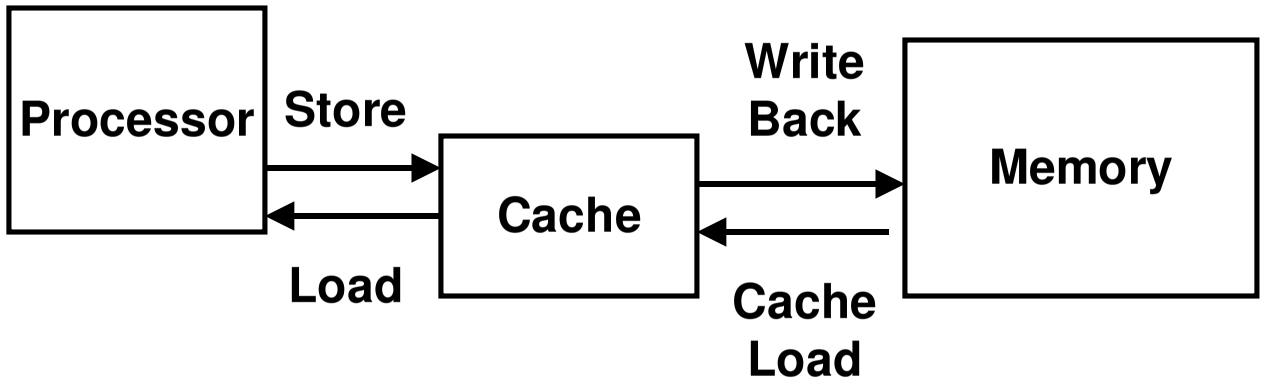
- Store by processor updates cache *and* memory.
- Memory always consistent with cache
- Never need to store from cache to memory
- Viable since ~2X more loads than stores



Write Strategies (Cont.)

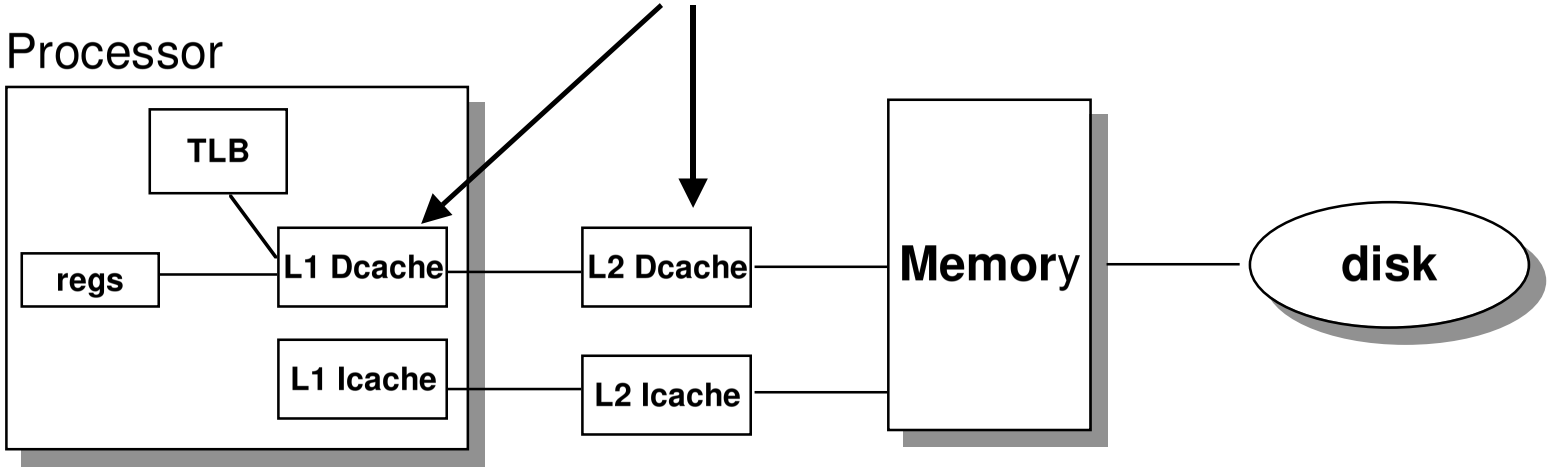
Write Back

- **Store by processor only updates cache block**
- **Modified block written to memory only when it is evicted**
  - Requires "dirty bit" for each block
    - » Set when modify block in cache
    - » Indicates that block in memory is stale
- **Memory not always consistent with cache**



Multi-Level Caches

Can have separate Icache and Dcache or *unified* Icache/Dcache

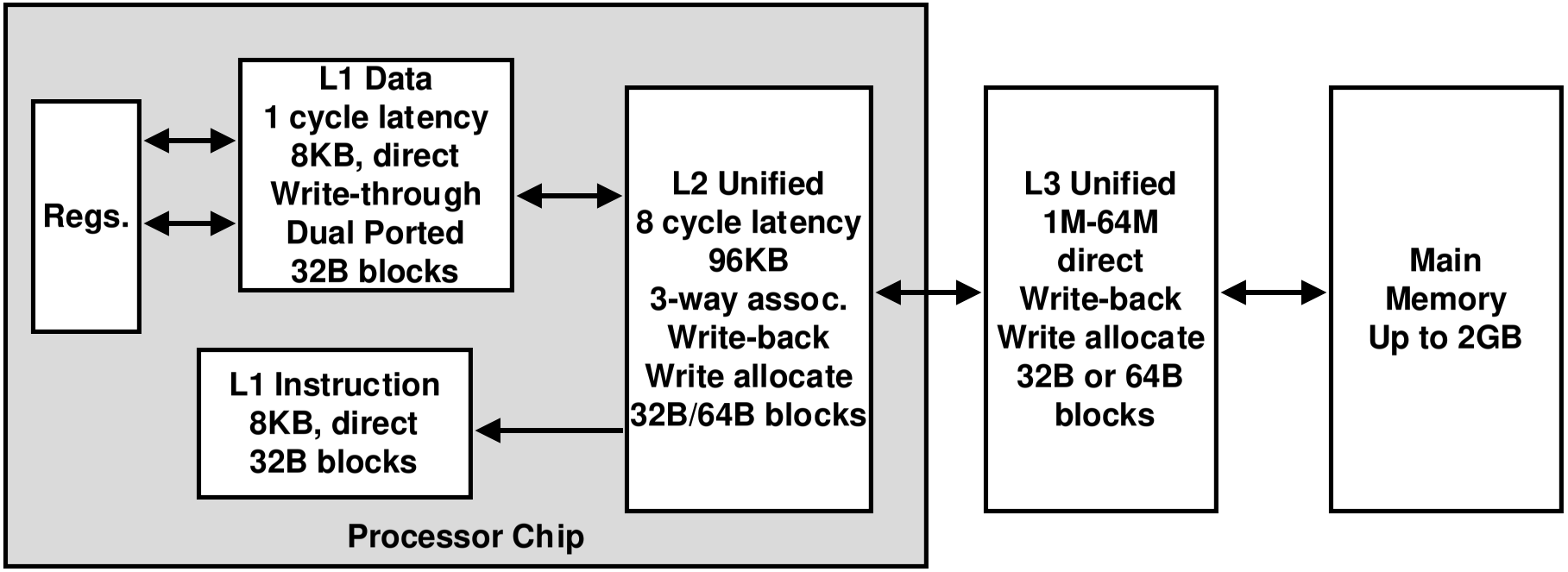


size:	200 B	8 KB	1M SRAM	128 MB DRAM	10 GB
speed:	5 ns	5 ns	6 ns	70 ns	10 ms
$/Mbyte:			$200/MB	$1.50/MB	$0.06/MB
block size:	4 B	16 B	32 B	4 KB	

larger, slower, cheaper

larger block size, higher associativity, more likely to write back

Alpha 21164 Hierarchy

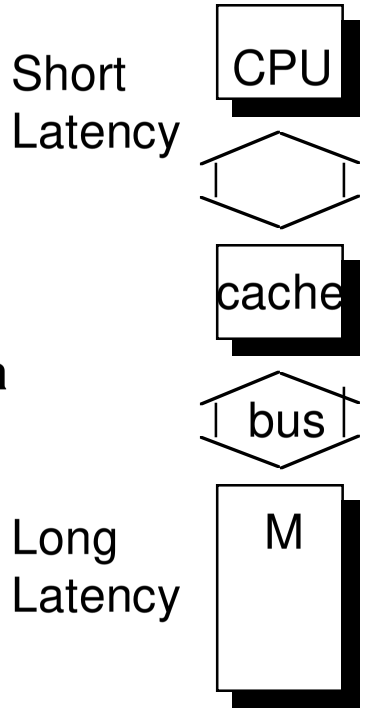


- Improving memory performance was a main design goal
- Earlier Alpha's CPUs starved for data

Bandwidth Matching

Challenge

- **CPU works with short cycle times**
- **DRAM (relatively) long cycle times**
- ***How can we provide enough bandwidth between processor & memory?***



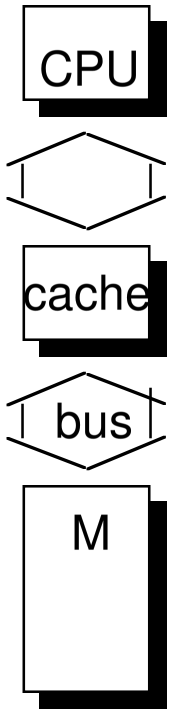
Effect of Caching

- **Caching greatly reduces amount of traffic to main memory**
- **But, sometimes need to move large amounts of data from memory into cache**

Trends

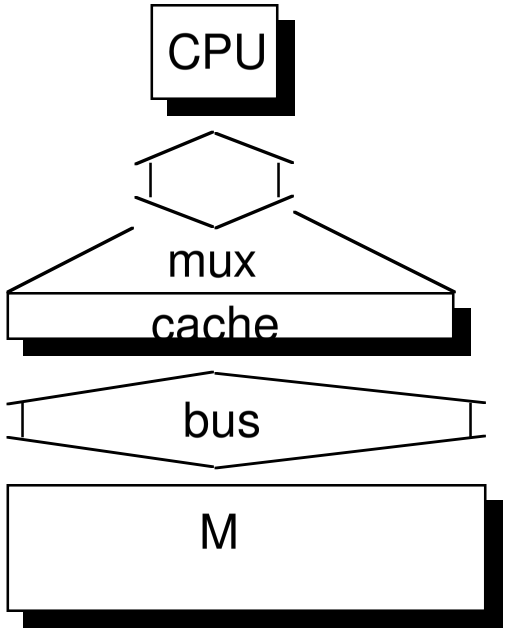
- **Need for high bandwidth much greater for multimedia applications**
  - Repeated operations on image data
- **Recent generation machines (e.g., Pentium II) greatly improve on predecessors**

High Bandwidth Memory Systems



Solution 1
High BW DRAM

Example:
Page Mode DRAM
RAMbus



Solution 2
Wide path between memory & cache

Example: Alpha AXP 21064
256 bit wide bus, L2 cache,
and memory.

Cache Performance Metrics

Miss Rate

- **fraction of memory references not found in cache (misses/references)**
- **Typical numbers:**

  5-10% for L1

  1-2% for L2

Hit Time

- **time to deliver a block in the cache to the processor (includes time to determine whether the block is in the cache)**
- **Typical numbers**

  1 clock cycle for L1

  3-8 clock cycles for L2

Miss Penalty

- **additional time required because of a miss**
  - Typically 10-30 cycles for main memory