15-213

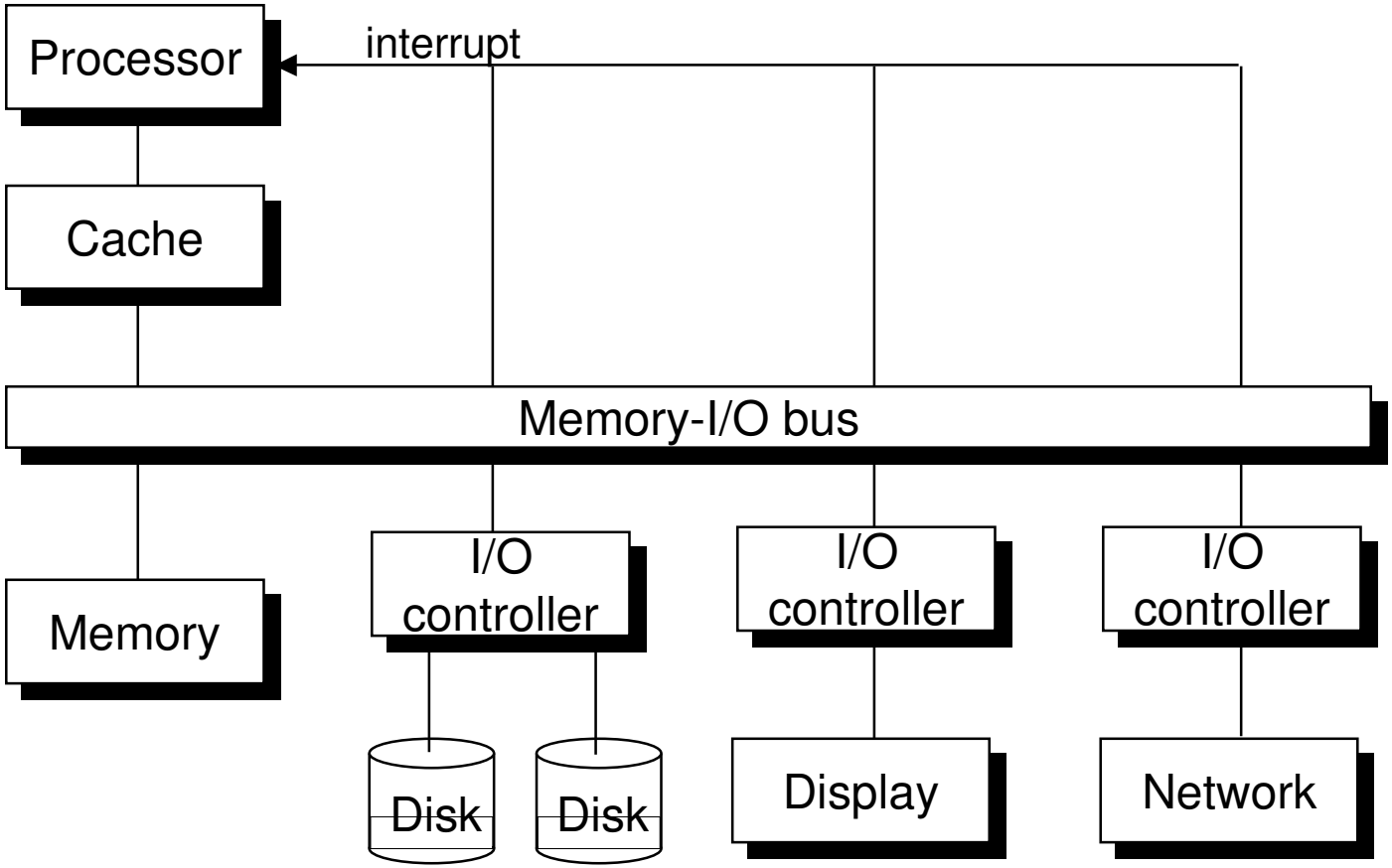
"The course that gives CMU its Zip!"

Caches
March 20, 2001

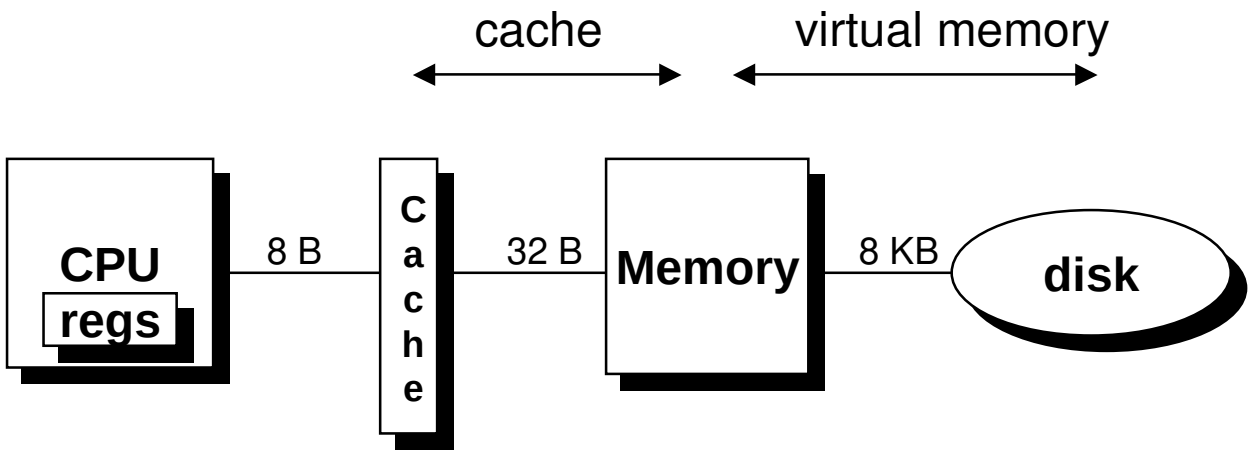
Topics

- **Memory Hierarchy**
  - Locality of Reference
- **SRAM Caches**
  - Direct Mapped
  - Associative

Computer System



Levels in Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	200 B	32 KB / 4MB	128 MB	30 GB
speed:	1 ns	2 ns	50 ns	8 ms
$/Mbyte:		$50/MB	$.50/MB	$0.05/MB
line size:	8 B	32 B	8 KB	

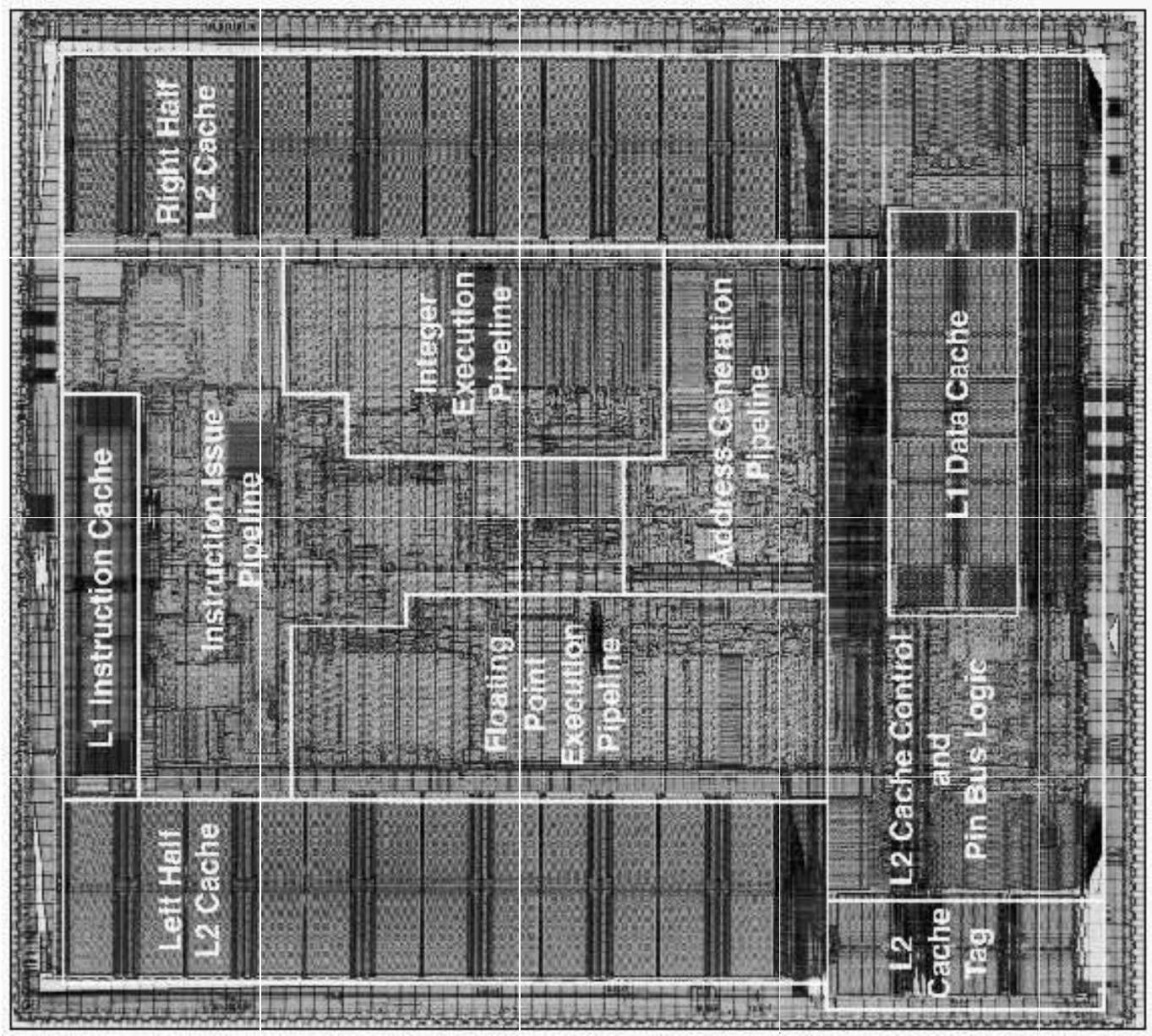
larger, slower, cheaper

Alpha 21164 Chip Photo

Microprocessor Report 9/12/94

Caches:

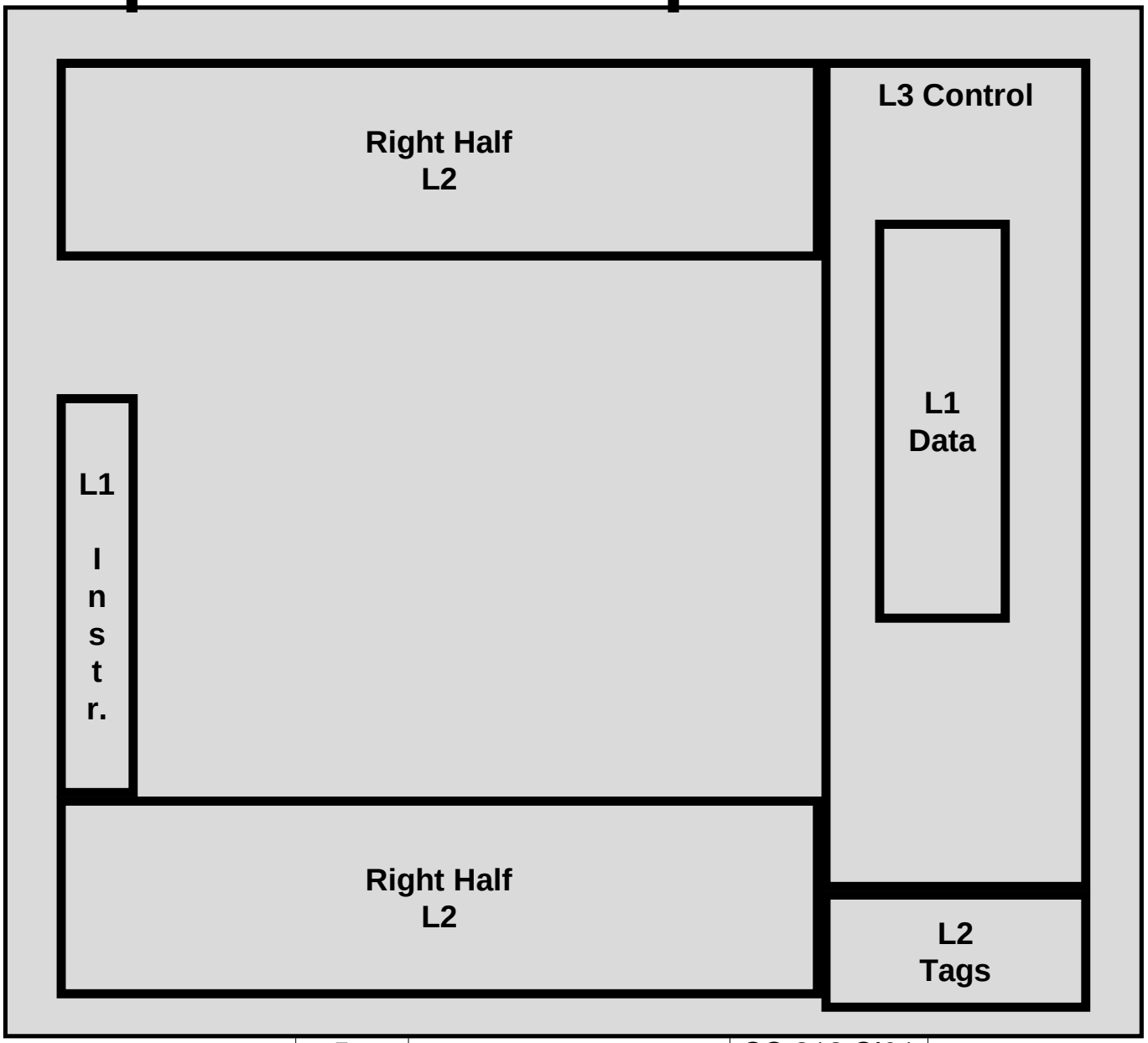
- **L1 data**
- **L1 instruction**
- **L2 unified**
- **TLB**
- **Branch history**



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)
- **Instructions**
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)

Caching: The Basic Idea

Main Memory

- **Stores words**

  A–Z in example

Cache

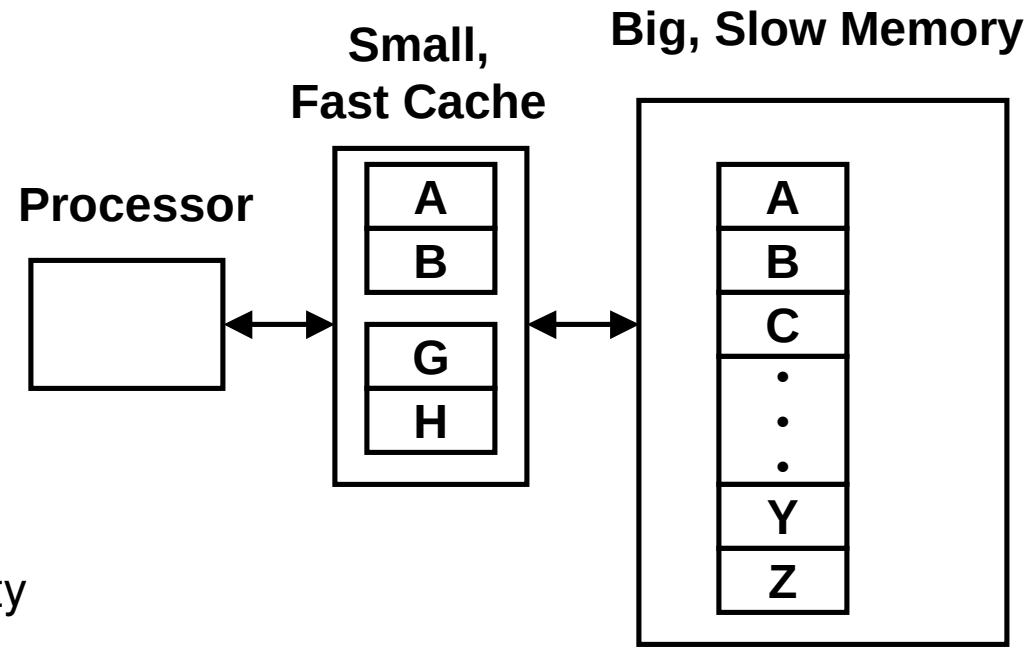
- **Stores subset of the words**

  4 in example

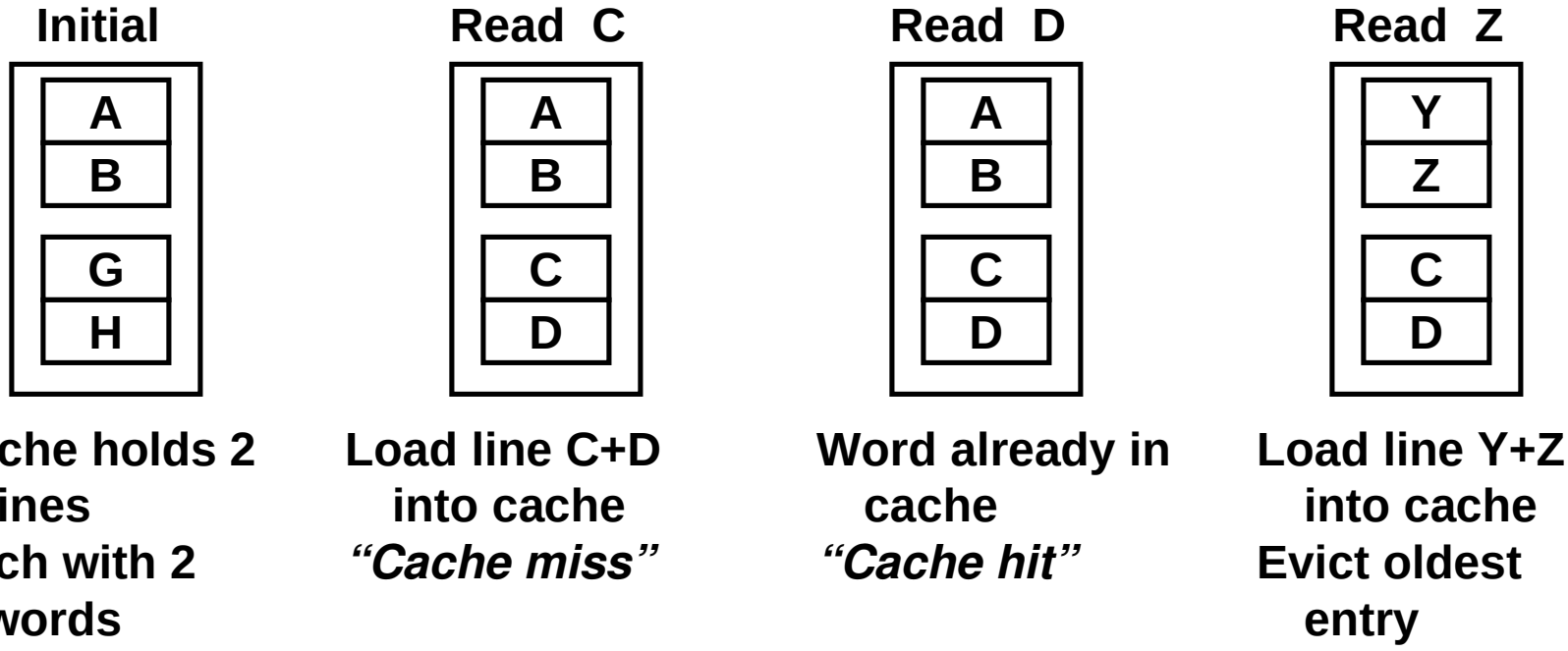
- **Organized in lines**
  - Multiple words
  - To exploit spatial locality

Access

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



Basic Idea (Cont.)



Cache holds 2 lines
Each with 2 words

Load line C+D into cache
"Cache miss"

Word already in cache
"Cache hit"

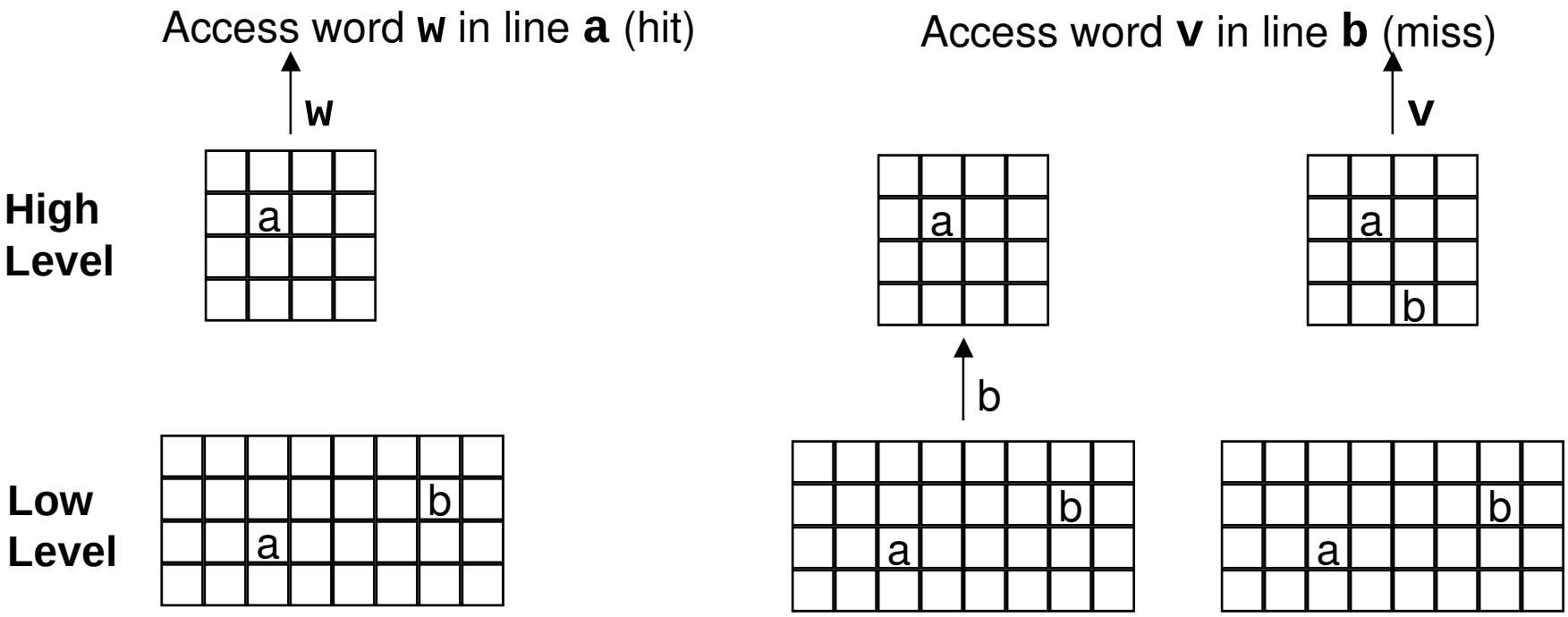
Load line Y+Z into cache
Evict oldest entry

Maintaining Cache:

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

Accessing Data in Memory Hierarchy

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



Design Issues for Caches

Key Questions:

- **Where should a line be placed in the cache? (line placement)**
- **How is a line found in the cache? (line identification)**
- **Which line should be replaced on a miss? (line 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 tradeoffs

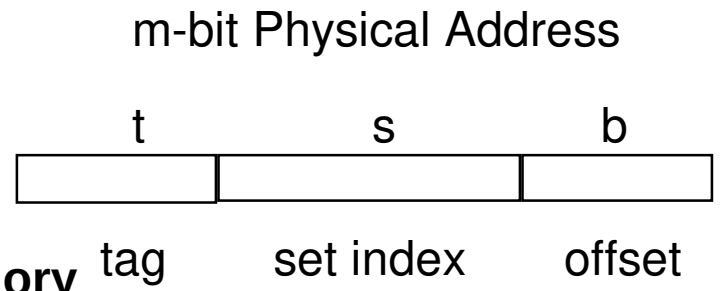
Direct-Mapped Caches

Simplest Design

- **Each memory line has a unique cache location**

Parameters

- **Line (or block) size B = 2^b**
  - Number of bytes in each line
  - Typically 2X–8X word size
- **Number of Sets S = 2^s**
  - Number of lines cache can hold
- **Total Cache Size = B*S = 2^{b+s}**

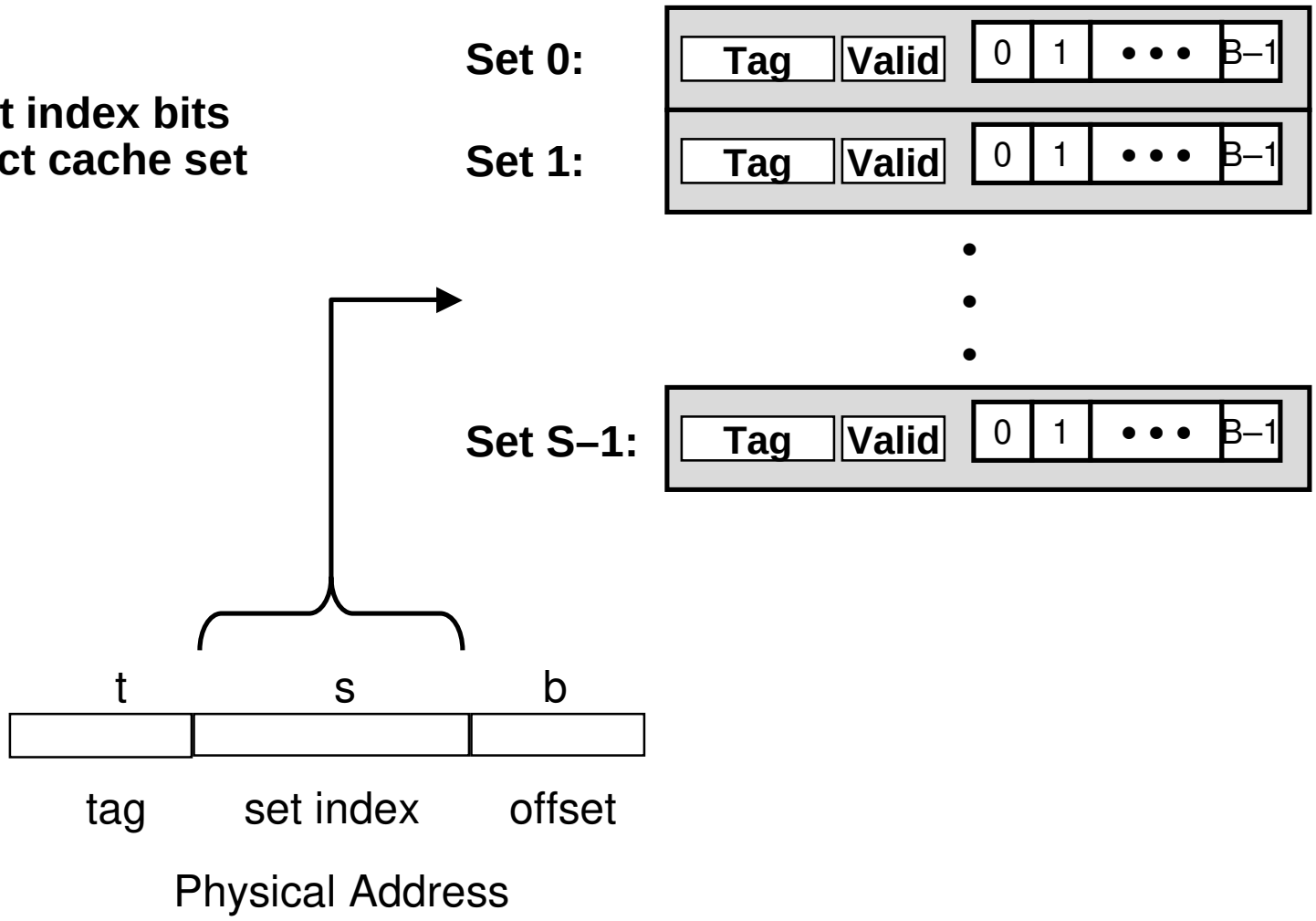


Physical Address

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

Indexing into Direct-Mapped Cache

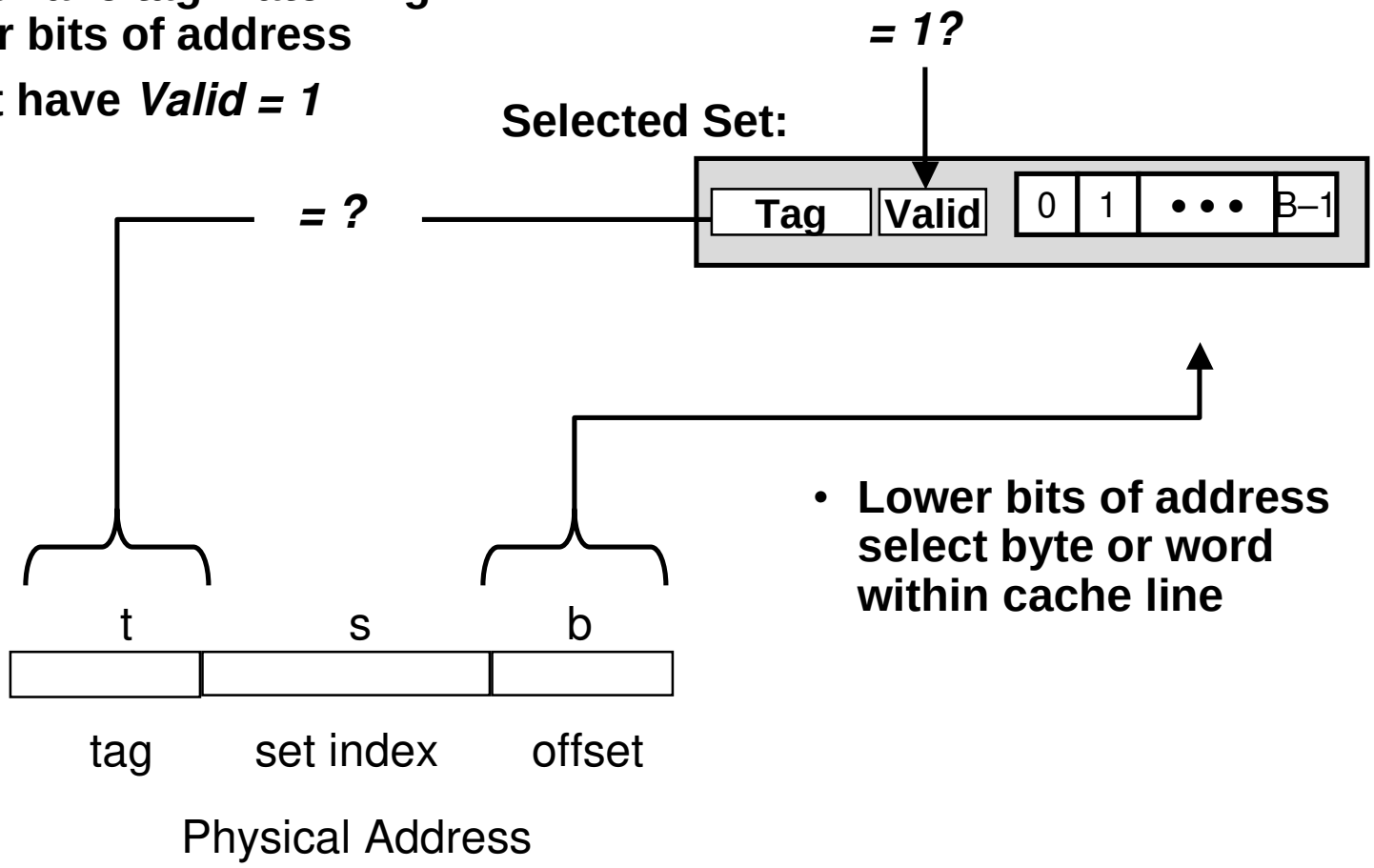
- **Use set index bits to select cache set**



Direct-Mapped Cache Tag Matching

Identifying Line

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



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

Direct Mapped Cache Simulation

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

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

0 [0000] (miss)

(1)

v	tag	data

13 [1101] (miss)

(2)

v	tag	data

8 [1000] (miss)

(3)

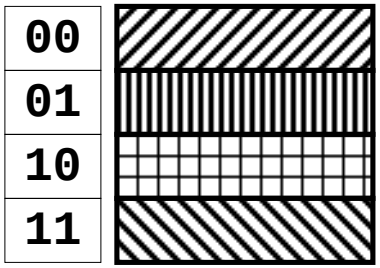
v	tag	data

0 [0000] (miss)

(4)

v	tag	data

Why Use Middle Bits as Index?

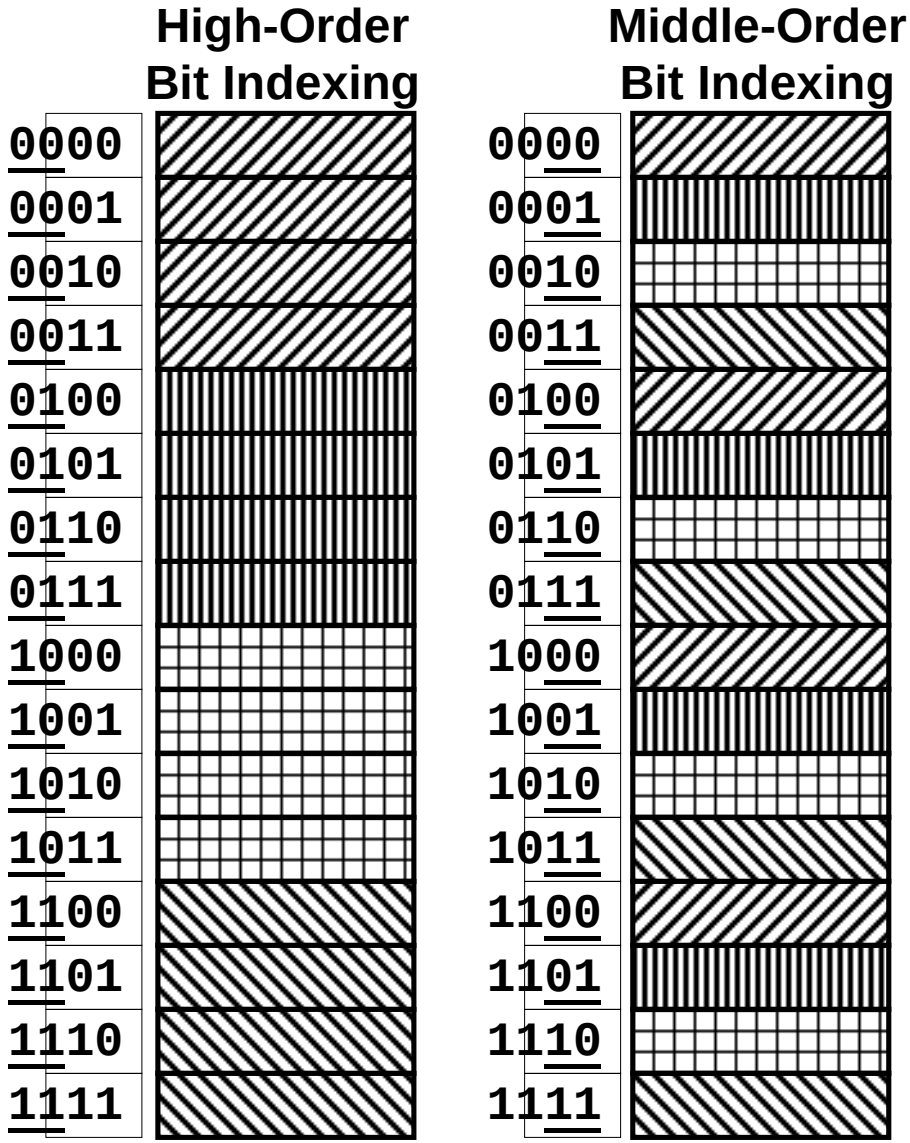


High-Order Bit Indexing

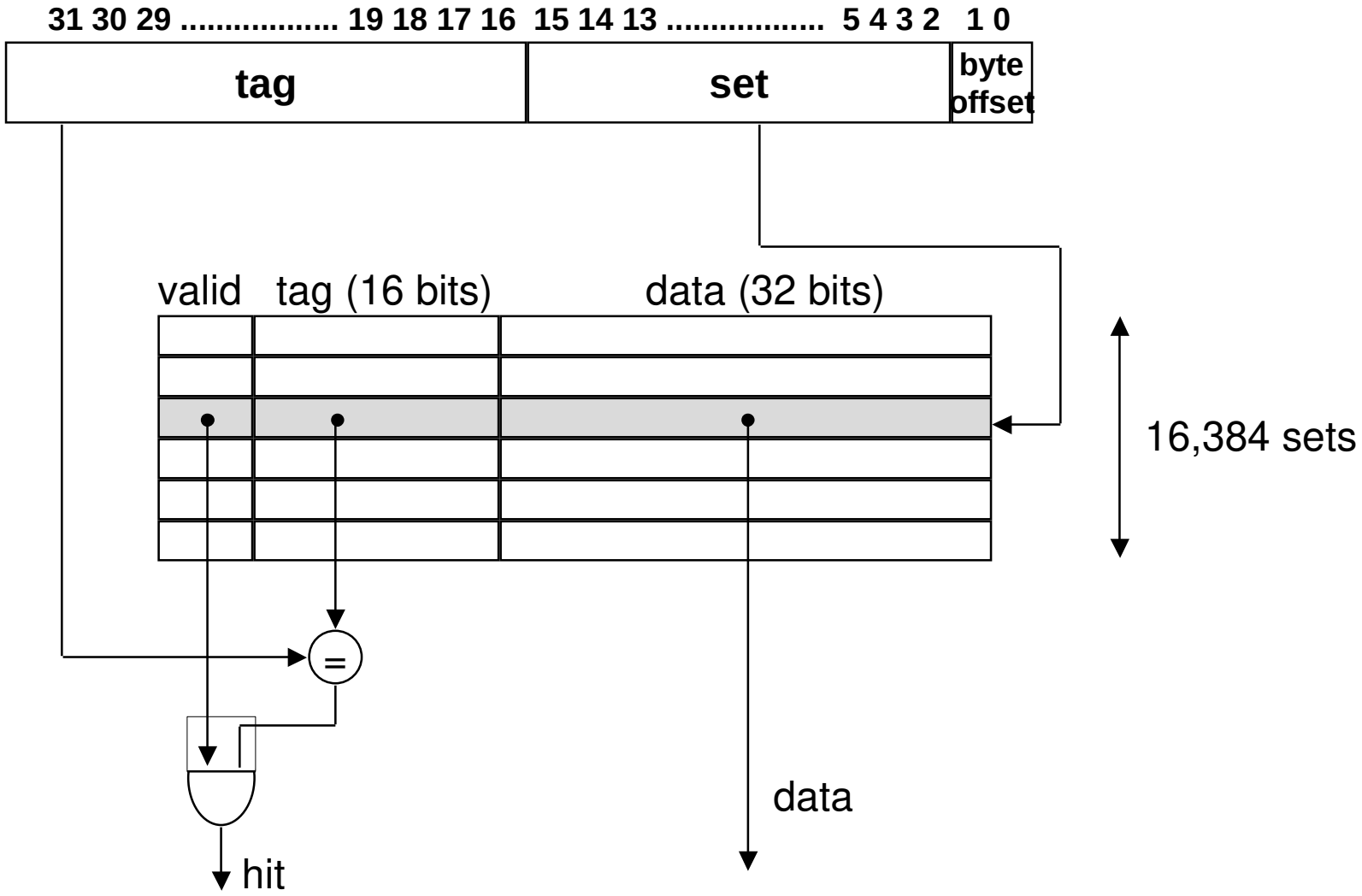
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing

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



Direct Mapped Cache Implementation (DECStation 3100)



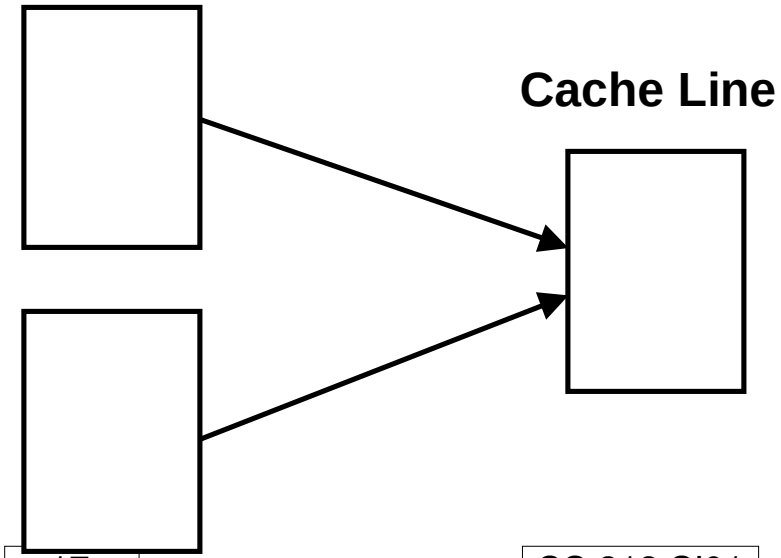
Properties of Direct Mapped Caches

Strength

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

Weakness

- Vulnerable to thrashing
- Two heavily used lines 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;
}
```

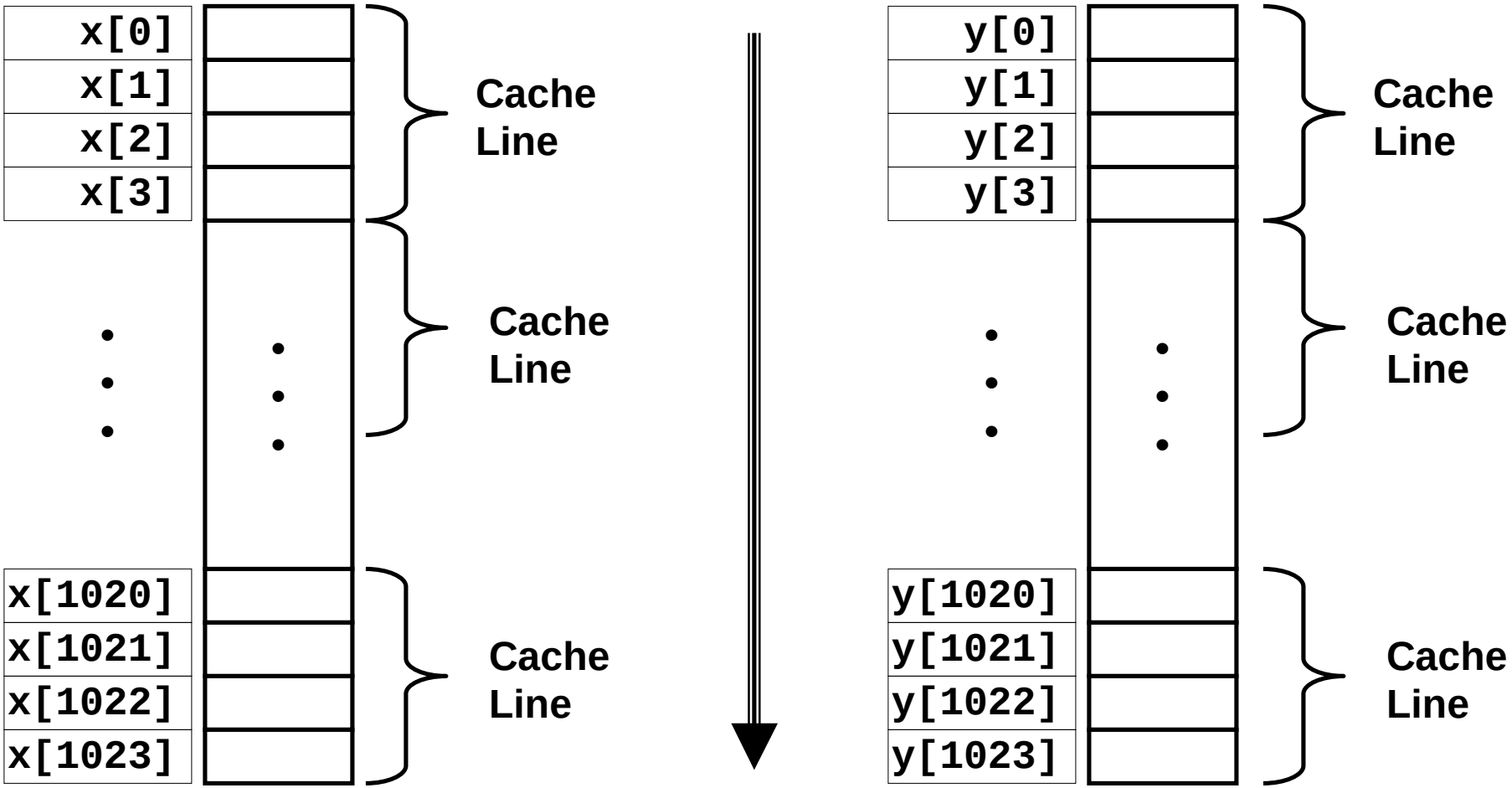
Machine

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

Performance

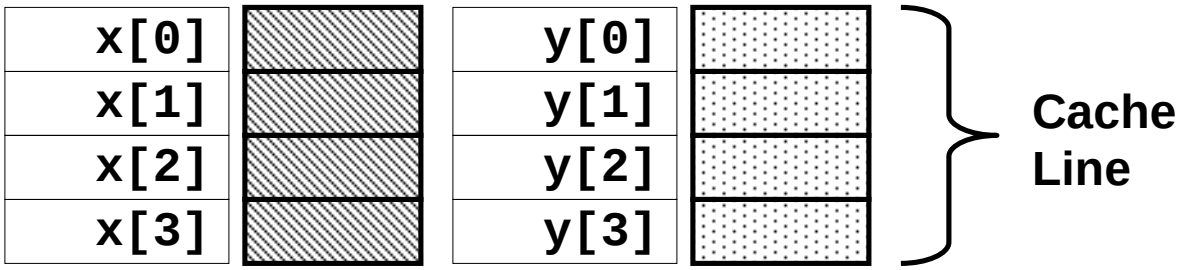
- Good case: 24 cycles / element
- Bad case: 66 cycles / element

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

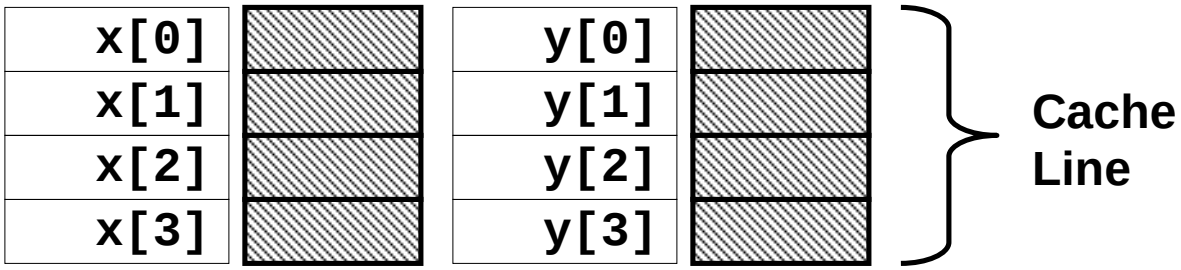
- **x[i] and y[i] map to different cache lines**
- **Miss rate = 25%**
  - Two memory accesses / iteration
  - On every 4th iteration have two misses

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 lines**
- **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

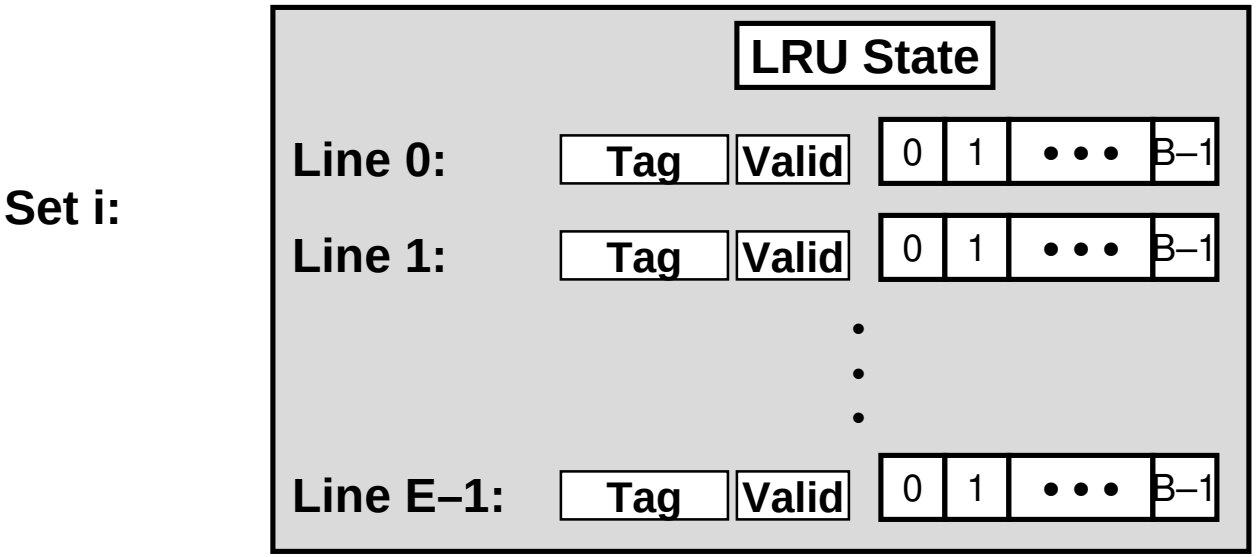
Set Associative Cache

Mapping of Memory Lines

- **Each set can hold E lines**
  - Typically between 2 and 8
- **Given memory line can map to any entry within its given set**

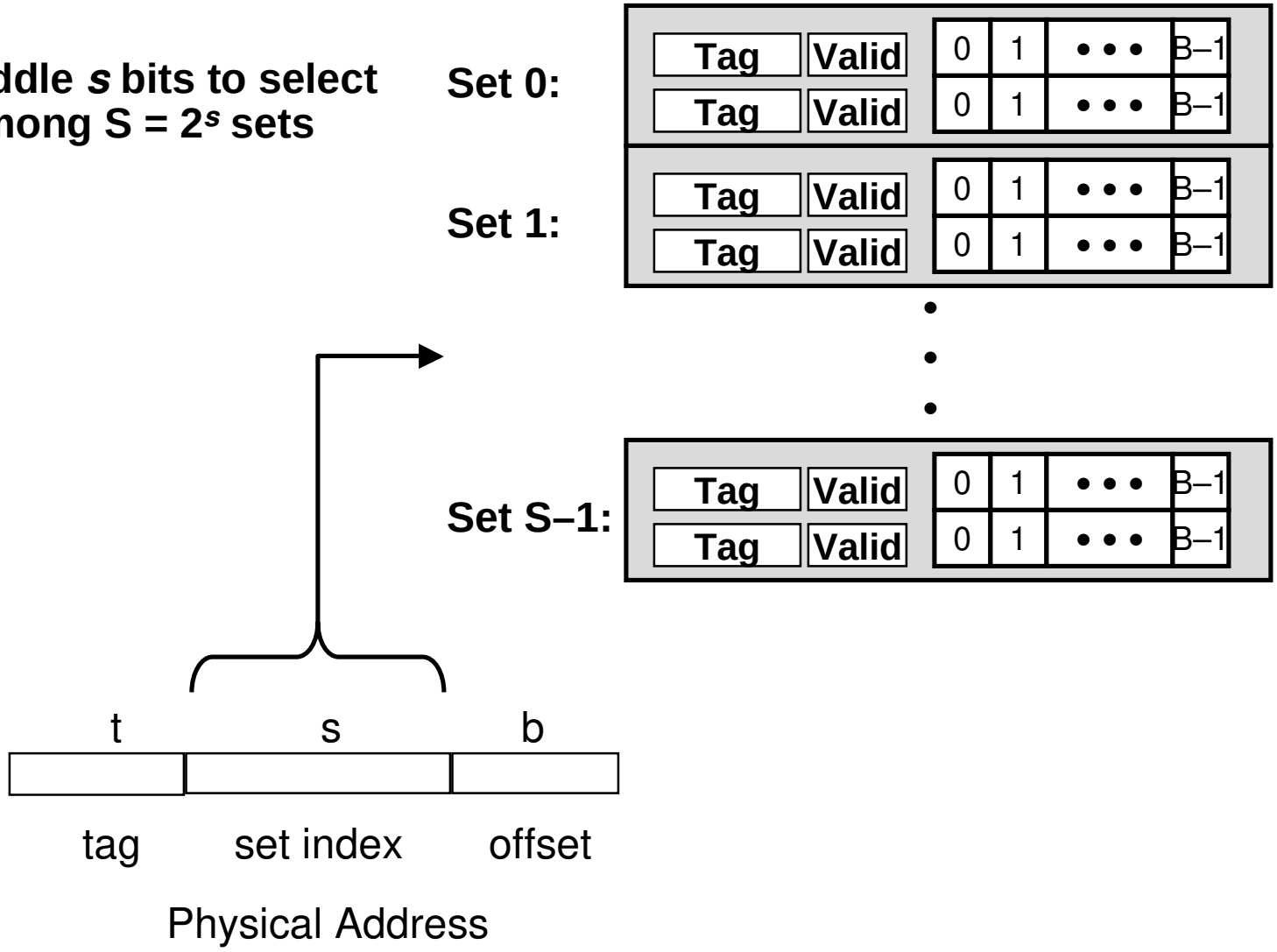
Eviction Policy

- **Which line gets kicked out when bring new line in**
- **Commonly either "Least Recently Used" (LRU) or pseudo-random**
  - LRU: least-recently accessed (read or written) line gets evicted



Indexing into 2-Way Associative Cache

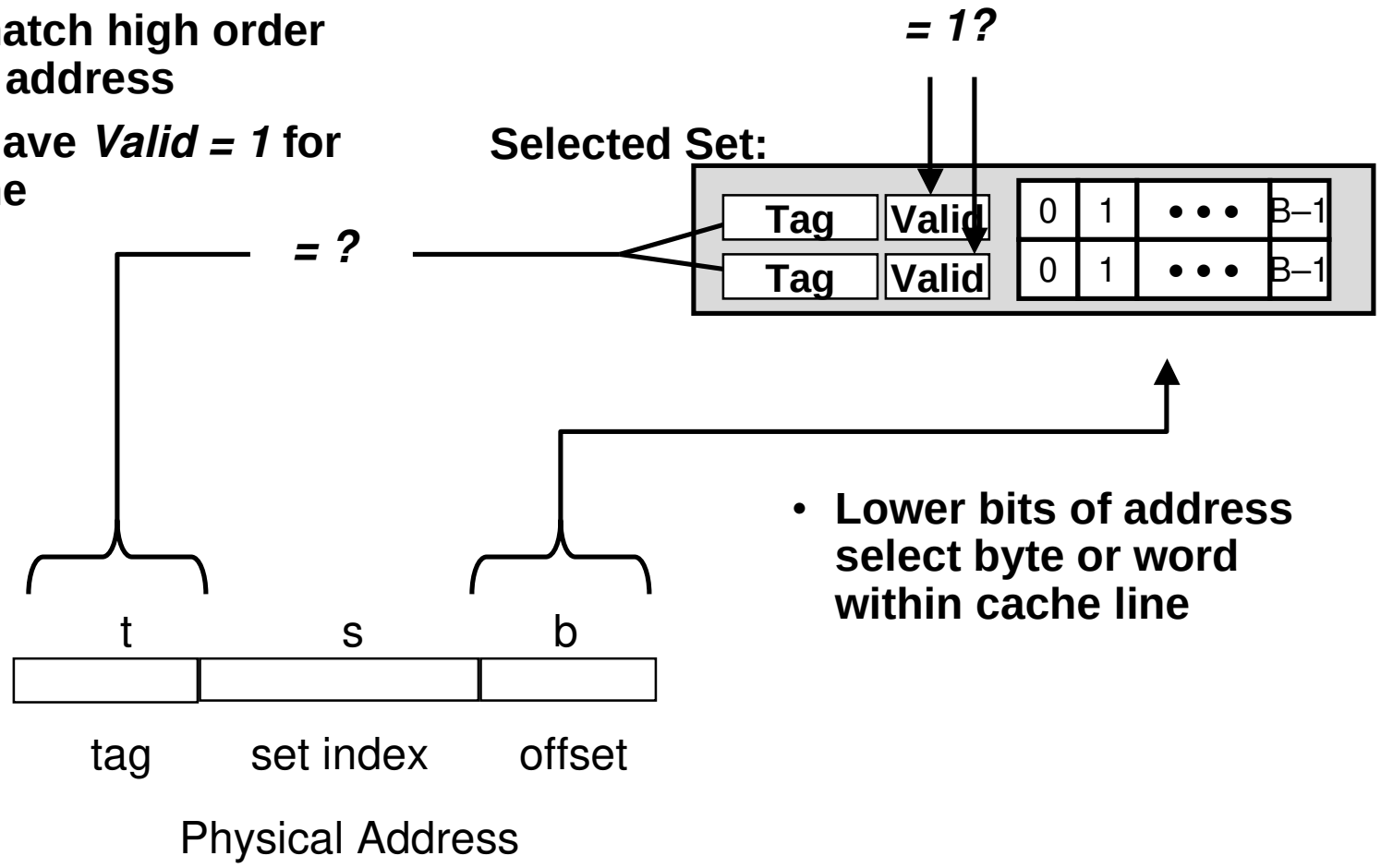
- **Use middle *s* bits to select from among S = 2^s sets**



2-Way Associative Cache Tag Matching

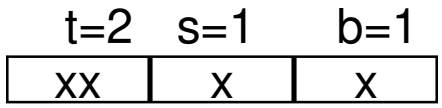
Identifying Line

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

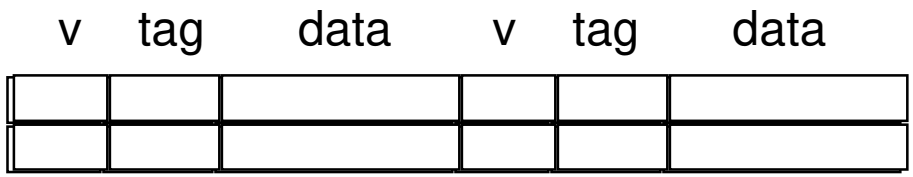


- **Lower bits of address select byte or word within cache line**

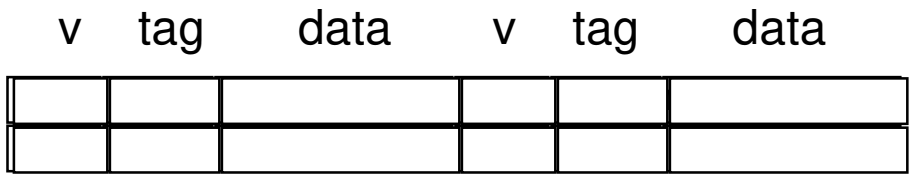
2-Way Set Associative Simulation



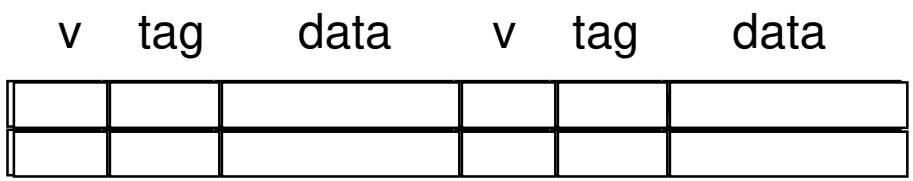
M=16 addresses, B=2 bytes/line, S=2 sets, E=2 entries/set
Address trace (reads):
0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]



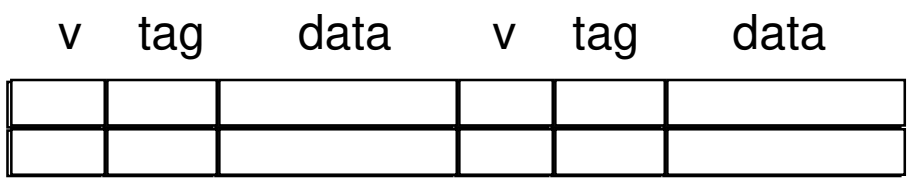
0 (miss)



13 (miss)



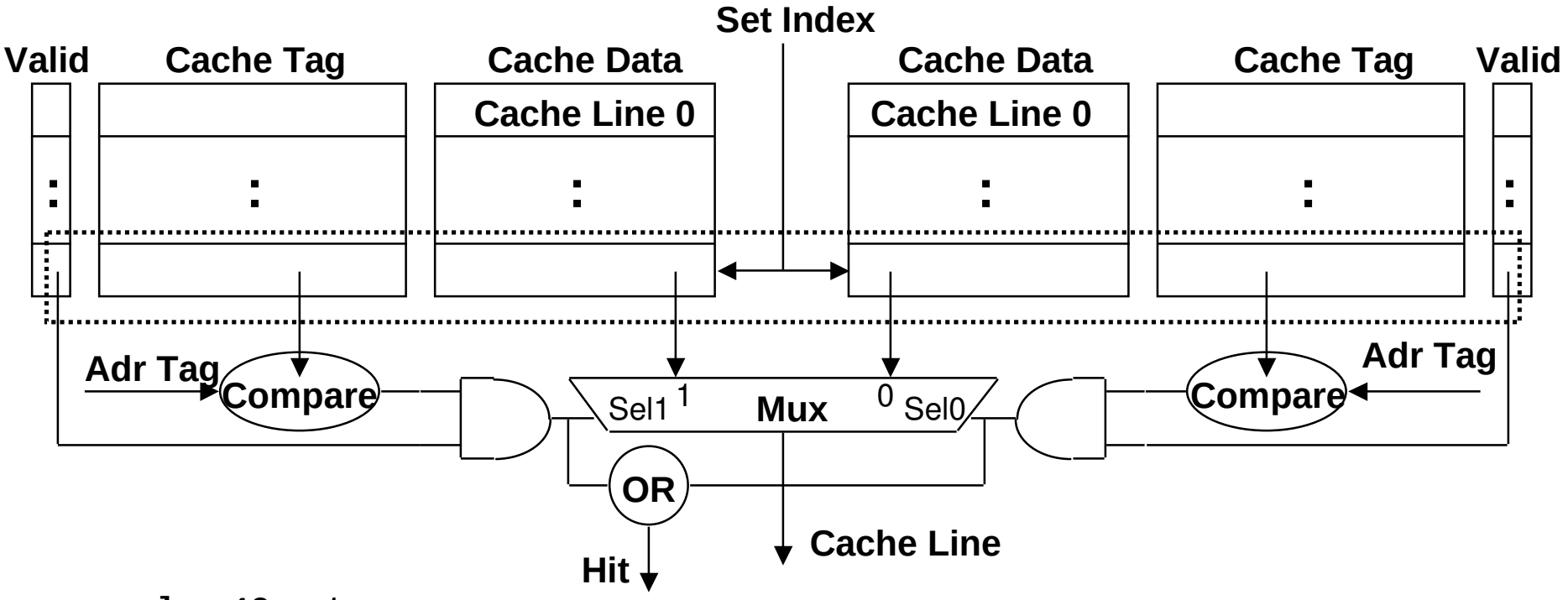
8 (miss)
(LRU replacement)



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

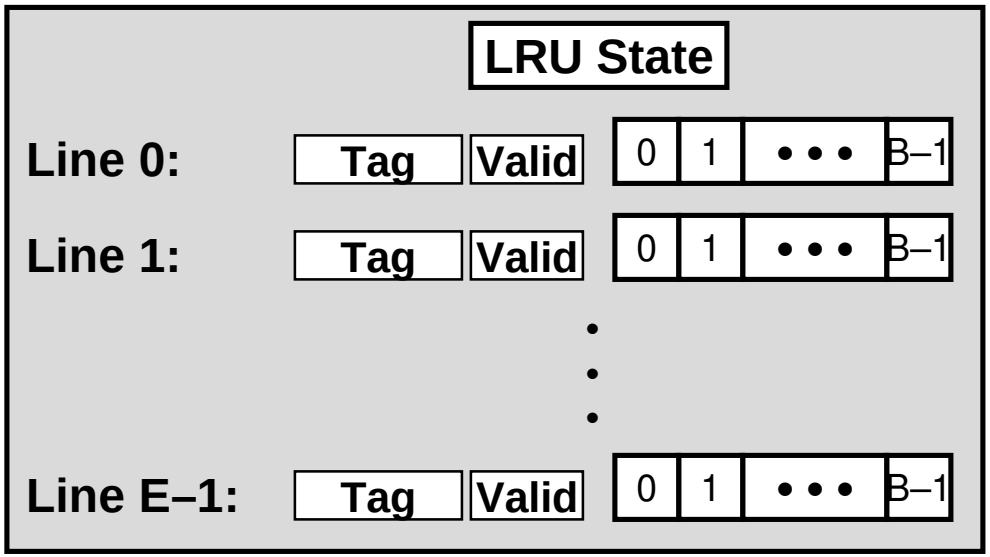


Fully Associative Cache

Mapping of Memory Lines

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

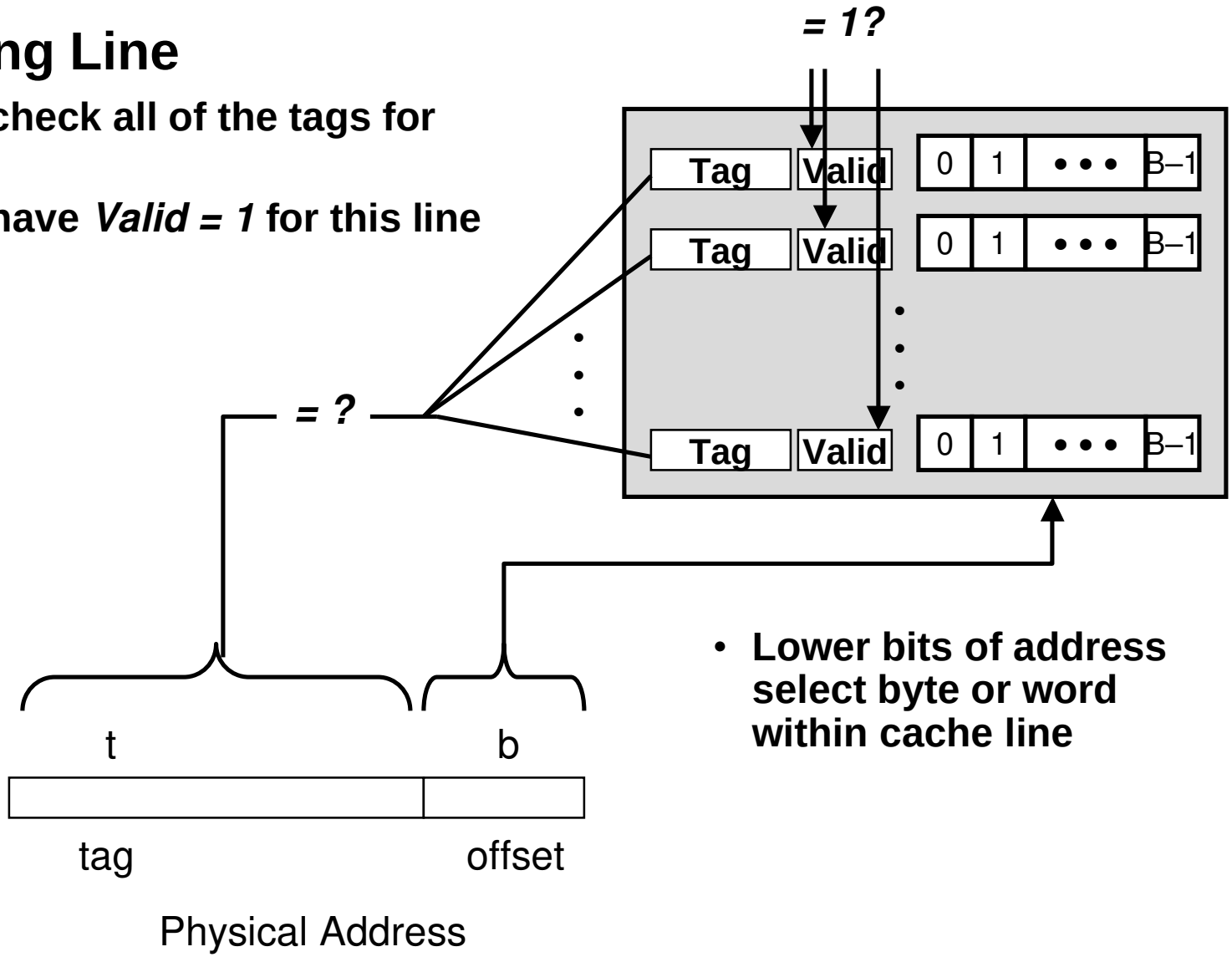
Entire Cache



Fully Associative Cache Tag Matching

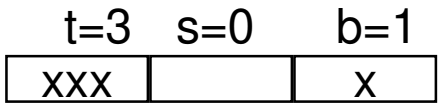
Identifying Line

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

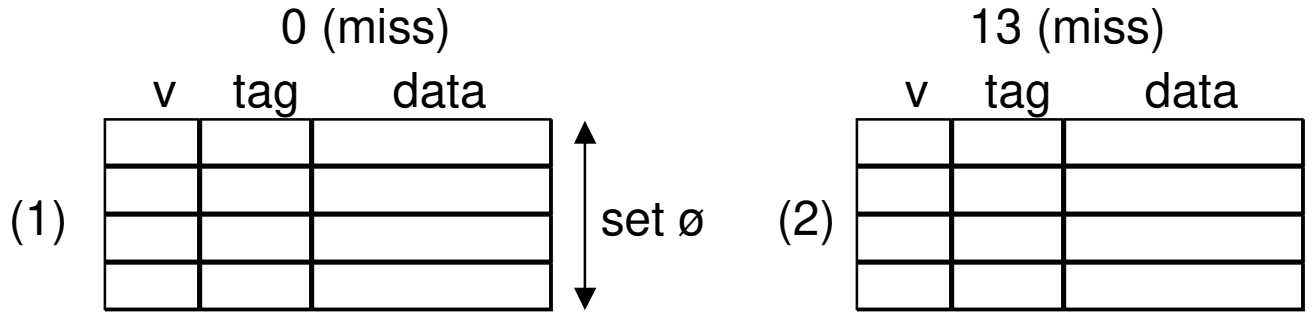


- **Lower bits of address select byte or word within cache line**

Fully Associative Cache Simulation



M=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]

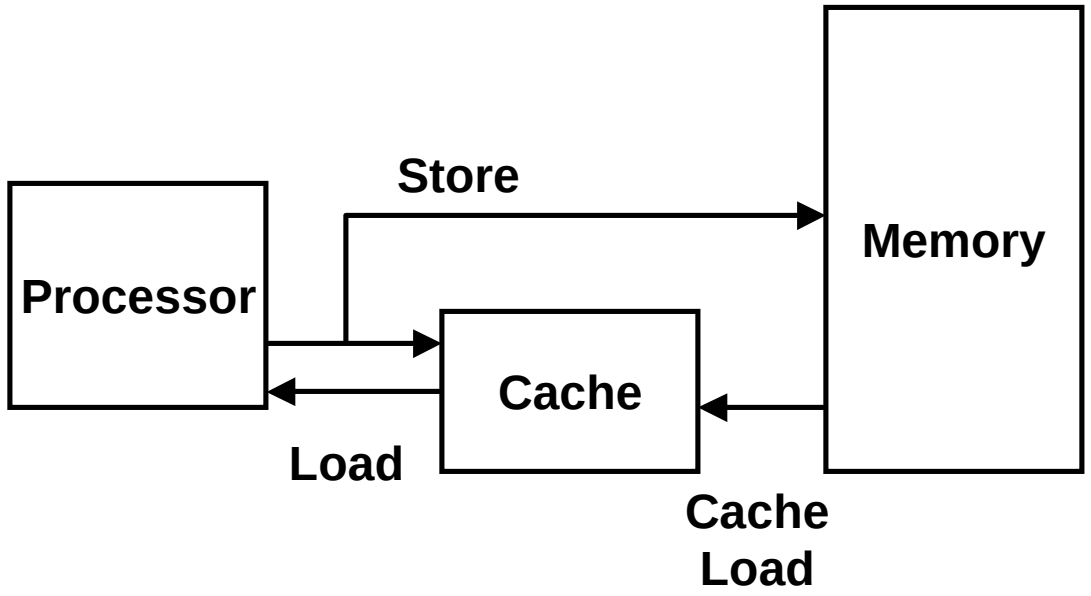


Write Policy

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

Write Through:

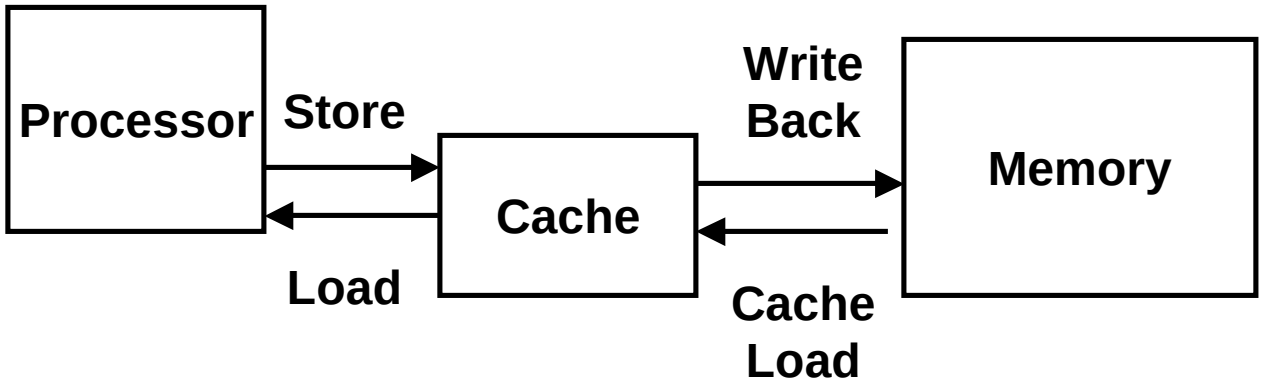
- Store by processor updates cache *and* memory.
- Memory always consistent with cache
- Never need to store from cache to memory
- ~2X more loads than stores



Write Strategies (Cont.)

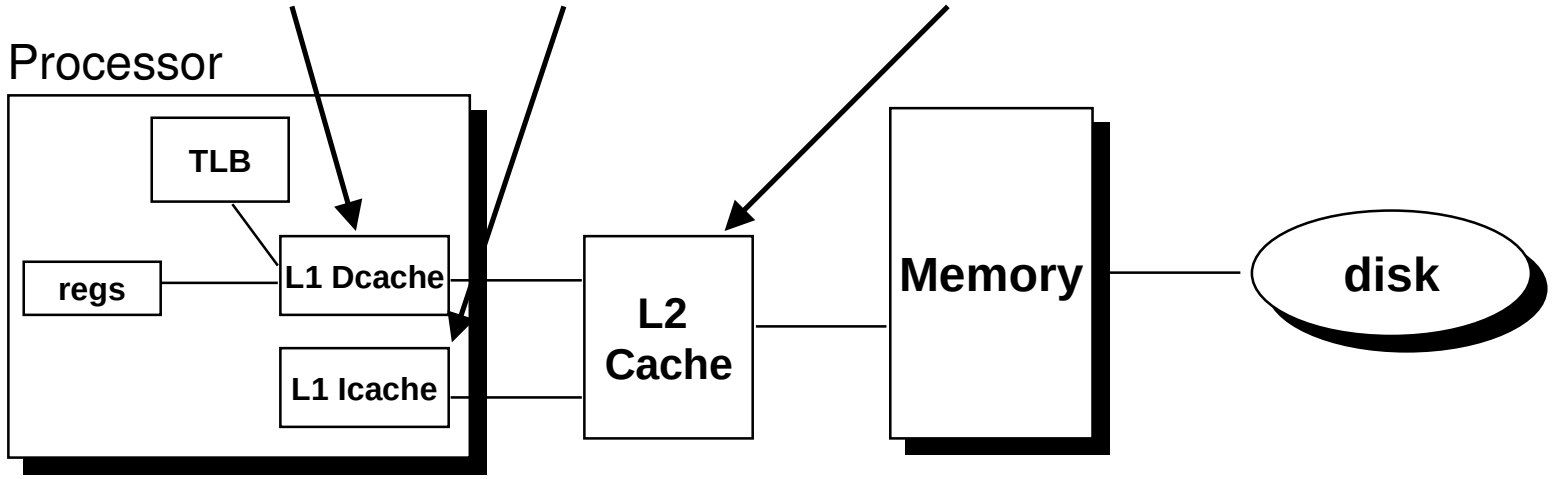
Write Back:

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



Multi-Level Caches

Options: ***separate*** data and instruction caches, or a ***unified*** cache

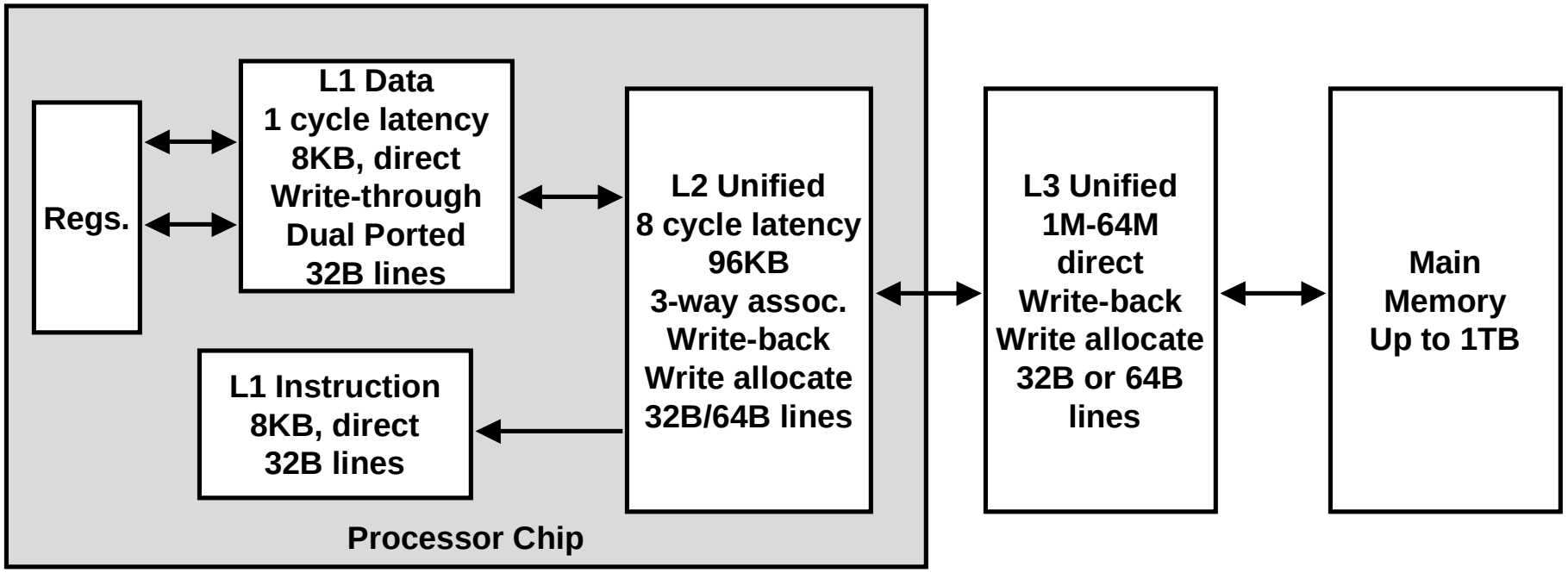


size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

larger, slower, cheaper

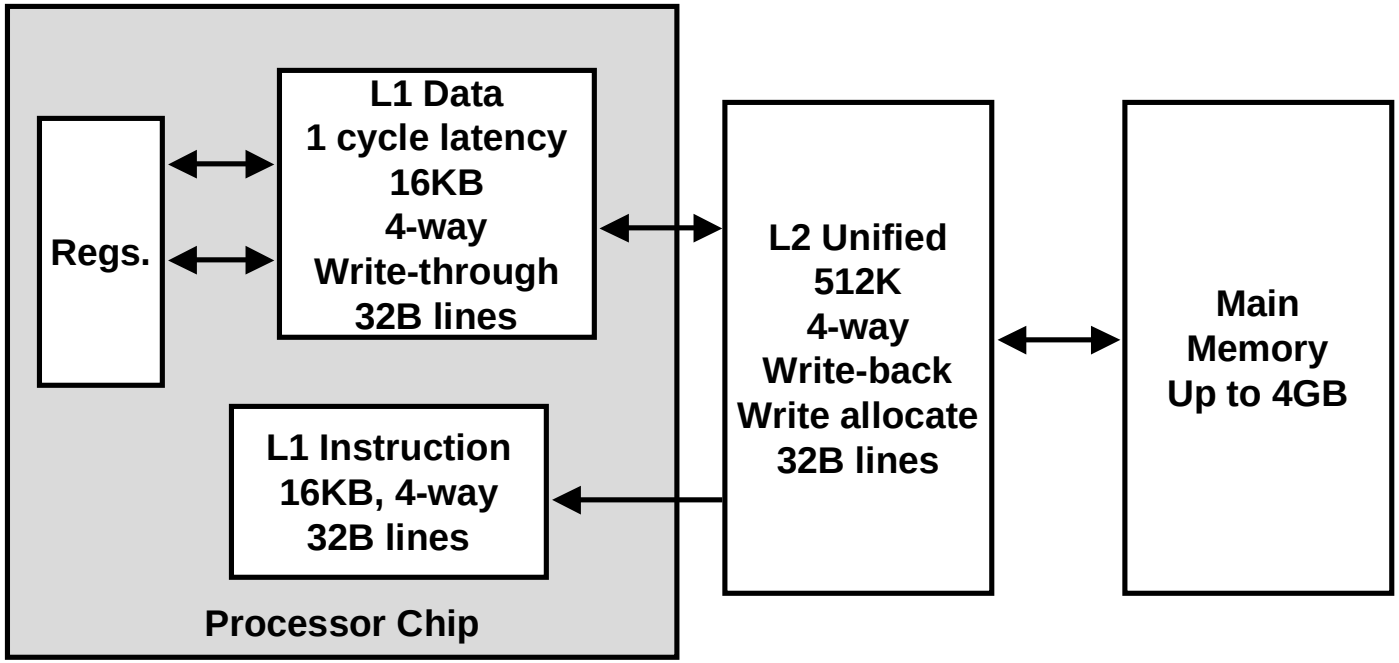
larger line size, higher associativity, more likely to write back

Alpha 21164 Hierarchy



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

Pentium III Xeon Hierarchy



Cache Performance Metrics

Miss Rate

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

  3-10% for L1

  can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

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

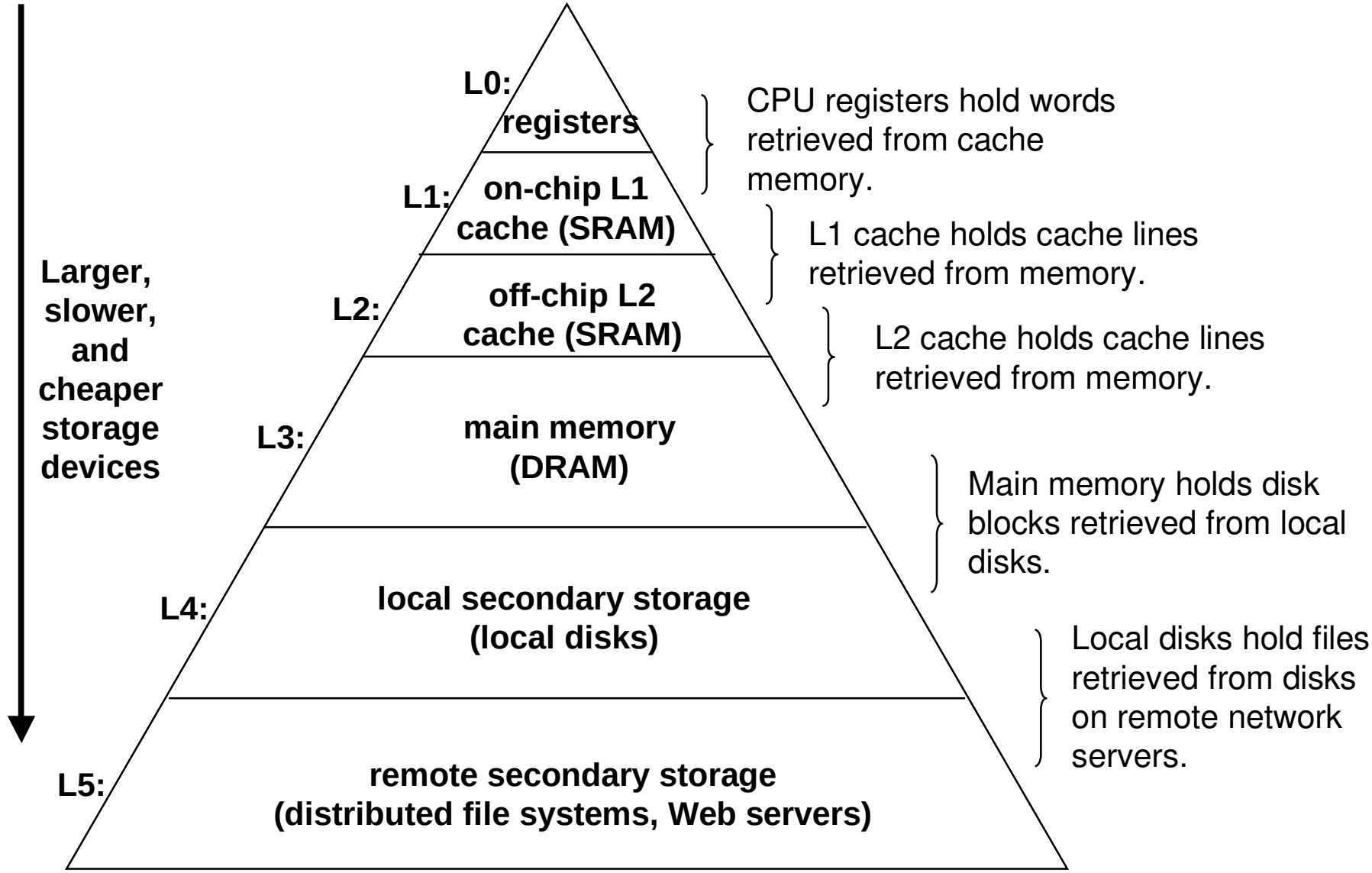
  1 clock cycle for L1

  3-8 clock cycles for L2

Miss Penalty

- **additional time required because of a miss**
  - Typically 25-100 cycles for main memory

Caching as a General Principle



Forms of Caching

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU Registers	0	Compiler
TLB	Address Translations	On-Chip TLB	0	Hardware
SRAM	32-byte block	On-Chip L1	1	Hardware
SRAM	32-byte block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main Memory	100	MMU+OS
Buffered Files	File Buffer	Main Memory	100	OS
Network File Cache	Parts of Files	Processor Disk	10,000,000	AFS Client
Browser Cache	Web Pages	Processor Disk	10,000,000	Browser
Web Cache	Web Pages	Server Disks	1,000,000,000	Akamai Server