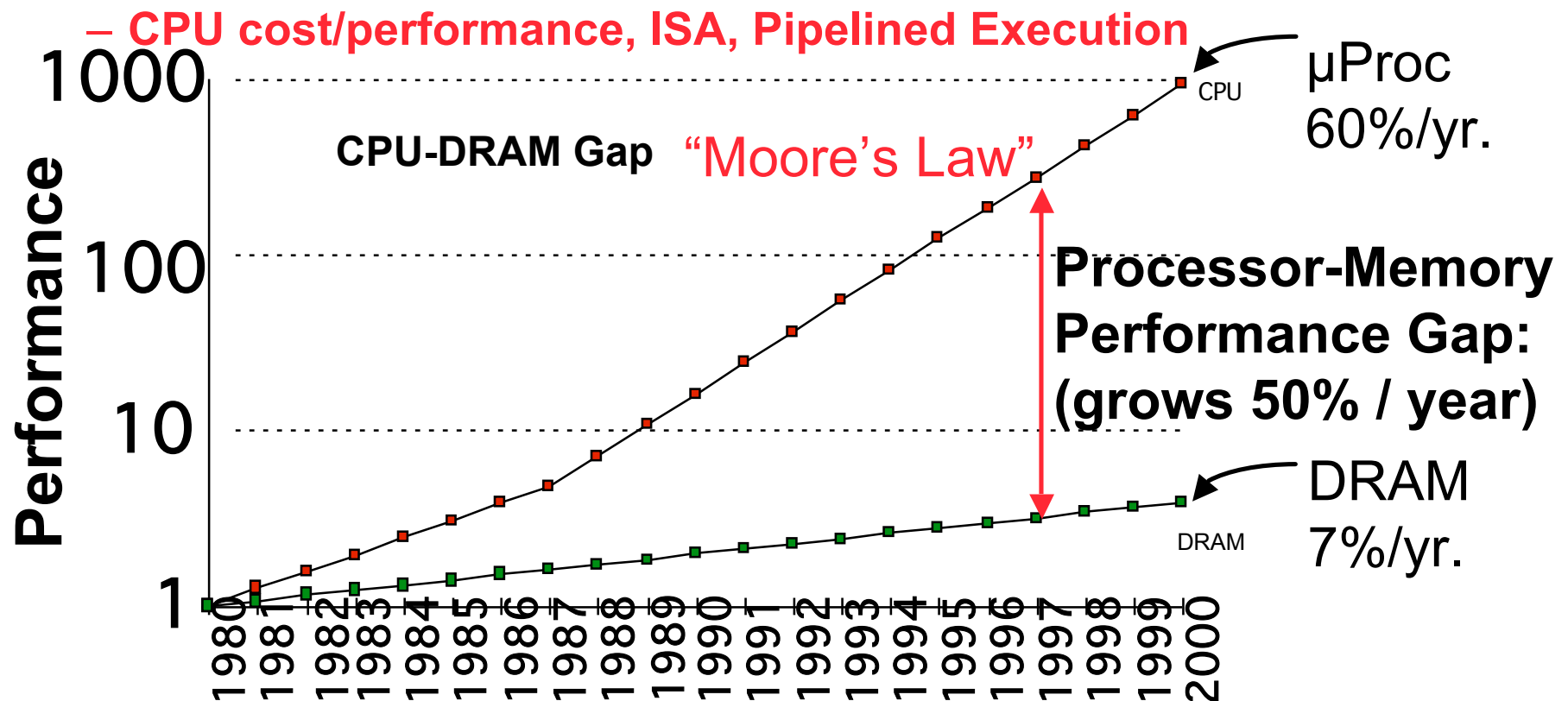


# Memory Hierarchy—Ways to Reduce Misses

# Review: Who Cares About the Memory Hierarchy?

- Processor Only Thus Far in Course:

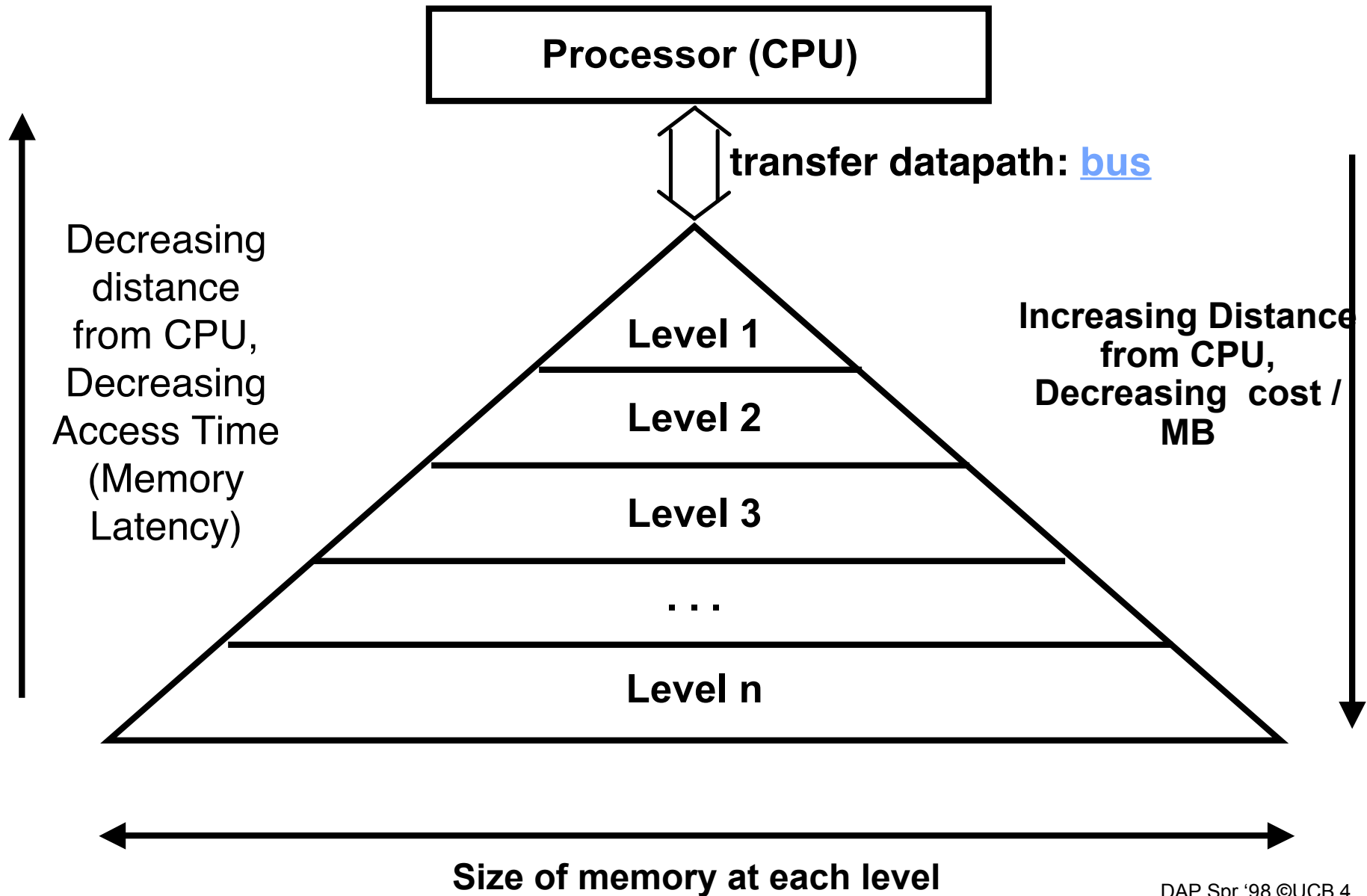


- 1980: no cache in  $\mu$ proc; 1995 2-level cache on chip (1989 first Intel  $\mu$ proc with a cache on chip)

# The Goal: Illusion of large, fast, cheap memory

- **Fact: Large memories are slow, fast memories are small**
- **How do we create a memory that is large, cheap and fast (most of the time)?**
- **Hierarchy of Levels**
  - **Uses smaller and faster memory technologies close to the processor**
  - **Fast access time in highest level of hierarchy**
  - **Cheap, slow memory furthest from processor**
- **The aim of memory hierarchy design is to have access time close to the highest level and size equal to the lowest level**

# Recap: Memory Hierarchy Pyramid



# Memory Hierarchy: Terminology

**Hit:** data appears in level X: **Hit Rate:** the fraction of memory accesses found in the upper level

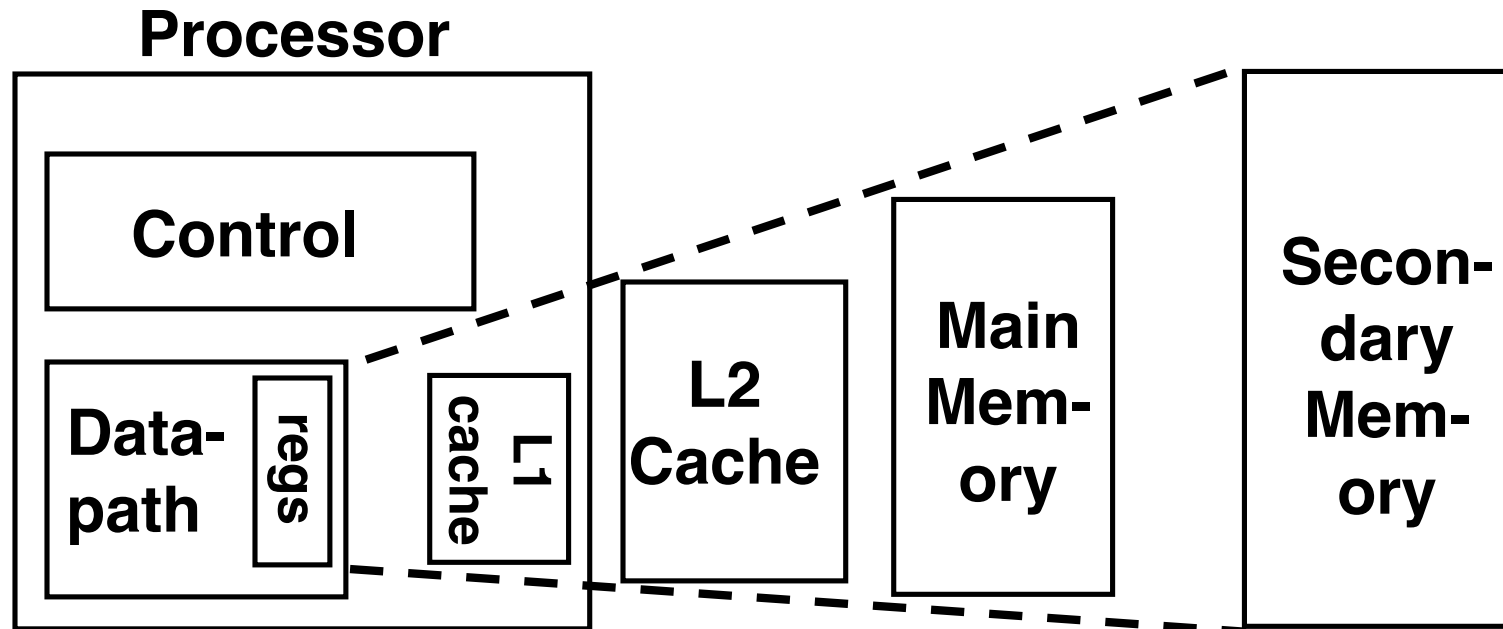
**Miss:** data needs to be retrieved from a block in the lower level (Block Y) **Miss Rate** = 1 - (Hit Rate)

**Hit Time:** Time to access the upper level which consists of Time to determine hit/miss + memory access time

**Miss Penalty:** Time to replace a block in the upper level + Time to deliver the block to the processor

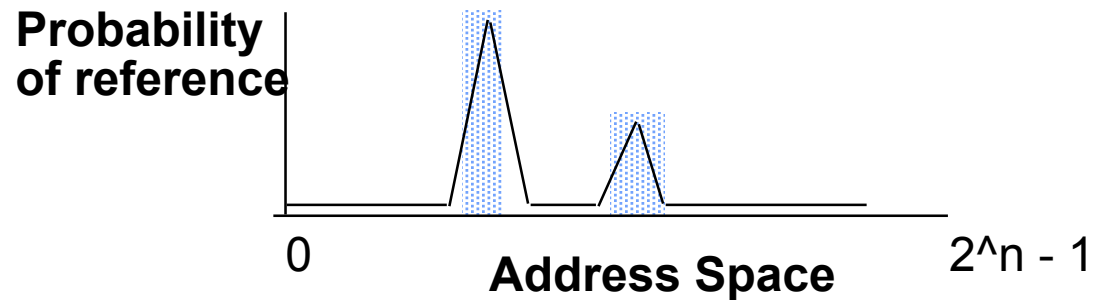
**Note:** Hit Time  $\ll$  Miss Penalty

# Current Memory Hierarchy



<b>Speed(ns):</b>	<b>0.5ns</b>	<b>2ns</b>	<b>6ns</b>	<b>100ns</b>	<b>10,000,000ns</b>
<b>Size (MB):</b>	<b>0.0005</b>	<b>0.05</b>	<b>1-4</b>	<b>100-1000</b>	<b>100,000</b>
<b>Cost (\$/MB):</b>	<b>--</b>	<b>\$100</b>	<b>\$30</b>	<b>\$1</b>	<b>\$0.05</b>
<b>Technology:</b>	<b>Regs</b>	<b>SRAM</b>	<b>SRAM</b>	<b>DRAM</b>	<b>Disk</b>

# Memory Hierarchy: Why Does it Work? Locality!

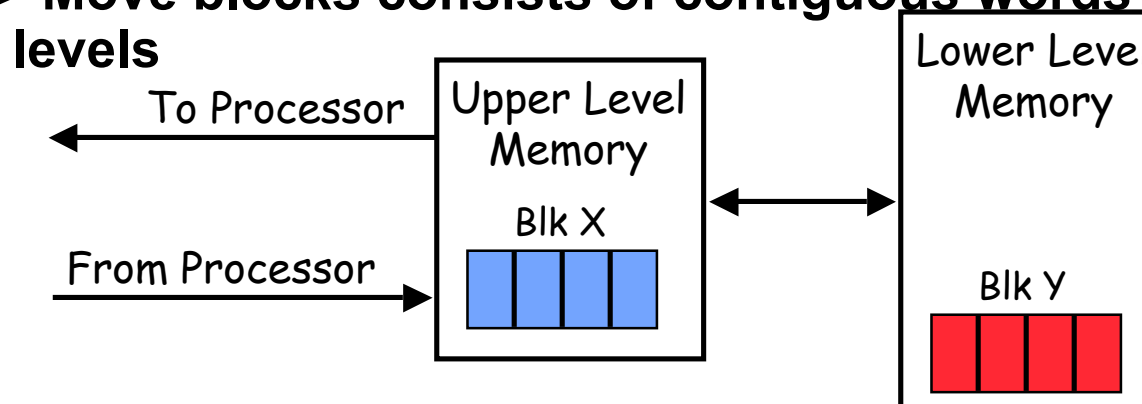


- **Temporal Locality (Locality in Time):**

=> Keep most recently accessed data items closer to the processor

- **Spatial Locality (Locality in Space):**

=> Move blocks consists of contiguous words to the upper levels



# Memory Hierarchy Technology

- **Random Access:**
  - “Random” is good: access time is the same for all locations
  - **DRAM:** Dynamic Random Access Memory
    - » High density, low power, cheap, slow
    - » Dynamic: need to be “refreshed” regularly
  - **SRAM:** Static Random Access Memory
    - » Low density, high power, expensive, fast
    - » Static: content will last “forever”(until lose power)
- **“Not-so-random” Access Technology:**
  - Access time varies from location to location and from time to time
  - Examples: Disk, CDROM
- **Sequential Access Technology: access time linear in location (e.g., Tape)**
- **We will concentrate on random access technology**
  - **The Main Memory: DRAMs + Caches: SRAMs**



# Introduction to Caches

- **Cache**

- is a small very fast memory (SRAM, expensive)
- contains copies of the most recently accessed memory locations (data and instructions): **temporal locality**
- is fully managed by hardware (unlike virtual memory)
- storage is organized in *blocks* of contiguous memory locations: **spatial locality**
- unit of transfer to/from main memory (or L2) is the cache block

- **General structure**

- *n blocks* per cache organized in *s sets*
- *b bytes* per block
- total cache size *n\*b bytes*

# Caches

- **For each block:**
  - an address *tag*: unique identifier
  - state bits:
    - » (in)valid
    - » modified
  - the data: *b* bytes
- **Basic cache operation**
  - every memory access is first presented to the cache
  - **hit**: the word being accessed is in the cache, it is returned to the cpu
  - **miss**: the word is not in the cache,
    - » a whole block is fetched from memory (L2)
    - » an “old” block is evicted from the cache (kicked out), which one?
    - » the new block is stored in the cache
    - » the requested word is sent to the cpu

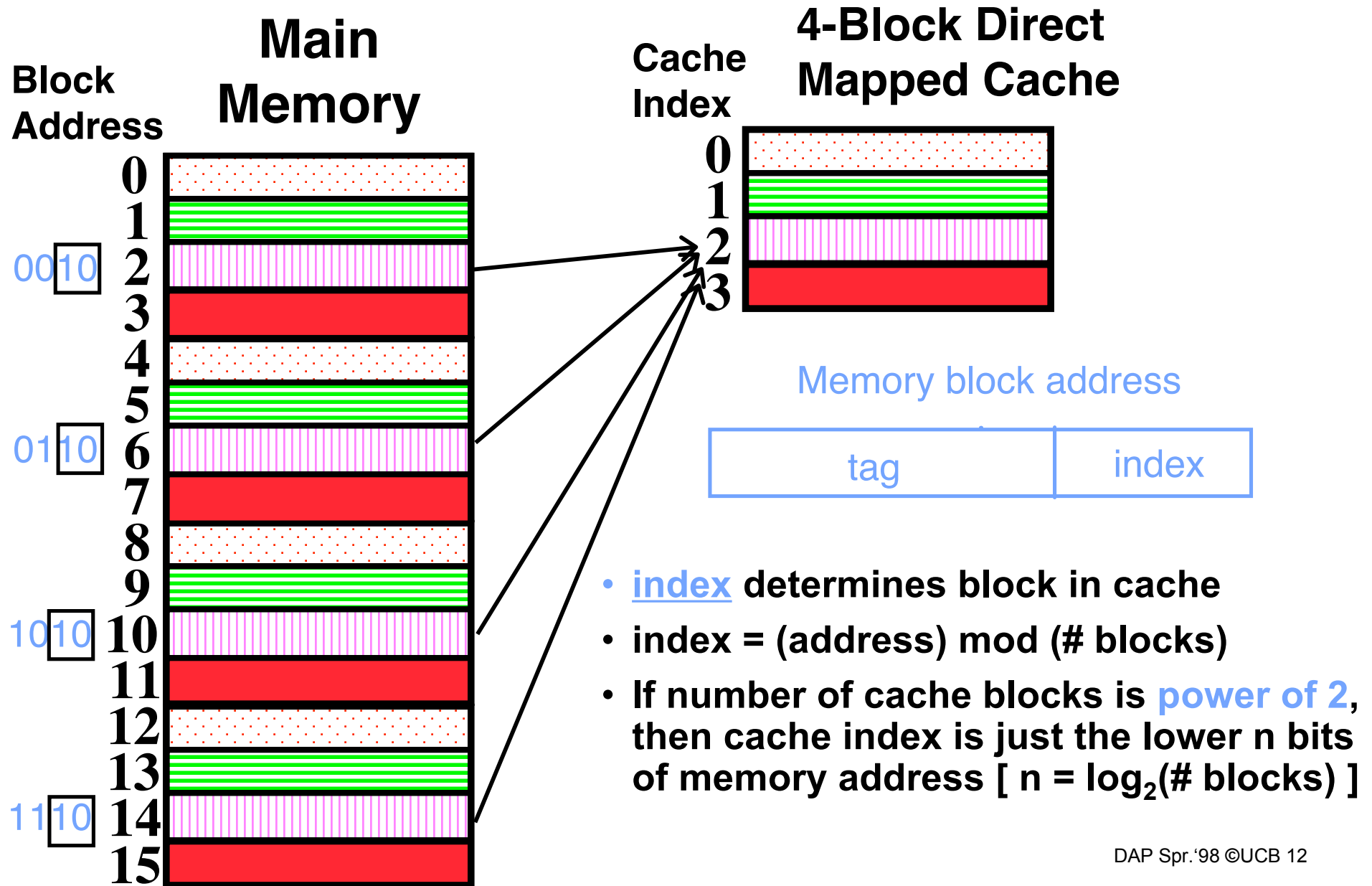
# Cache Organization

**(1) How do you know if something is in the cache?**

**(2) If it is in the cache, how to find it?**

- **Answer to (1) and (2) depends on type or organization of the cache**
- **In a direct mapped cache, each memory address is associated with one possible block within the cache**
  - **Therefore, we only need to look in a single location in the cache for the data if it exists in the cache**

# Simplest Cache: Direct Mapped



# Issues with Direct-Mapped

- If block size  $> 1$ , rightmost bits of index are really the offset within the indexed block

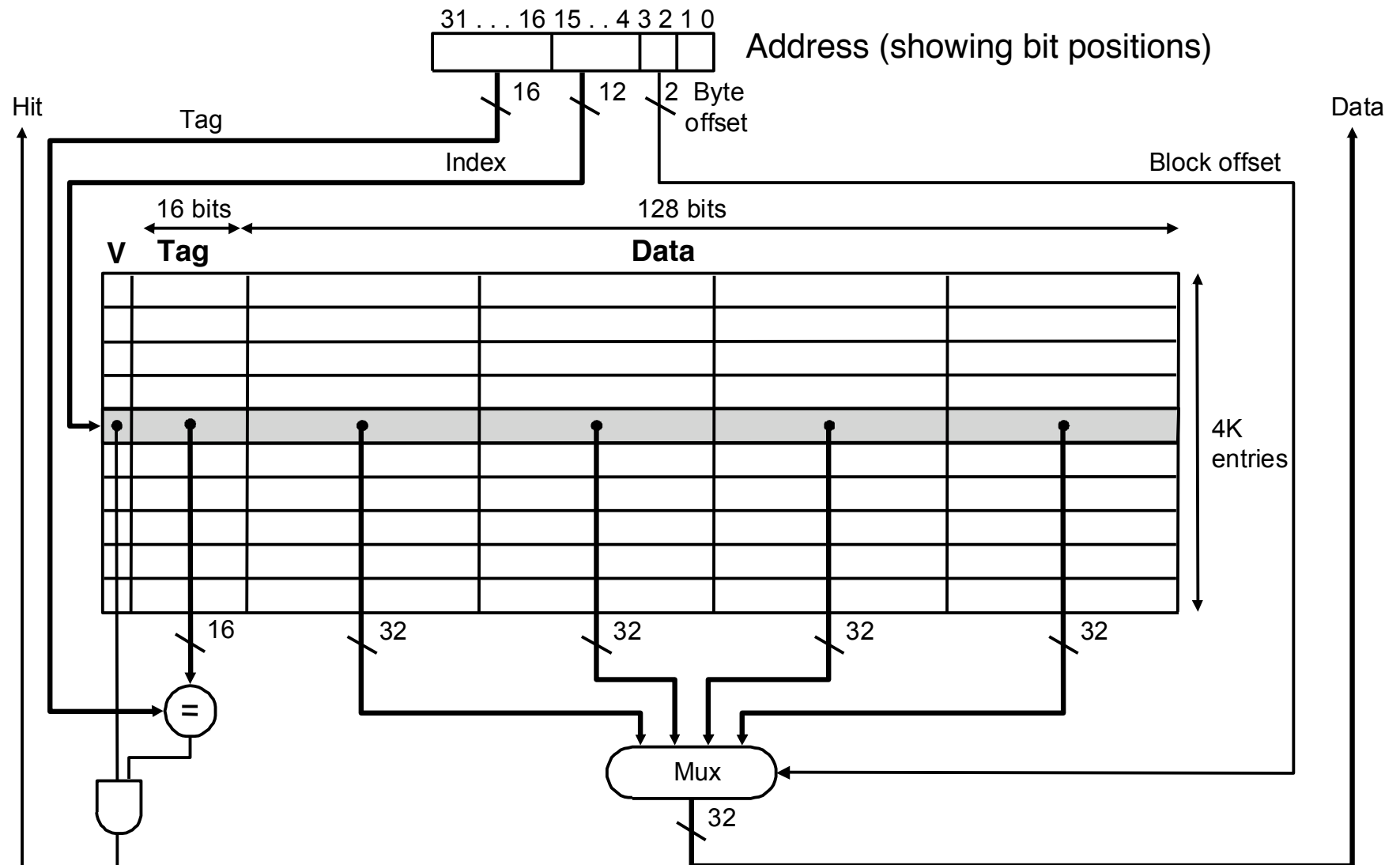


tag  
to check  
if have  
correct block

index  
to  
select  
block

byte  
offset  
within  
block

# 64KB Cache with 4-word (16-byte) blocks



# Direct-mapped Cache Contd.

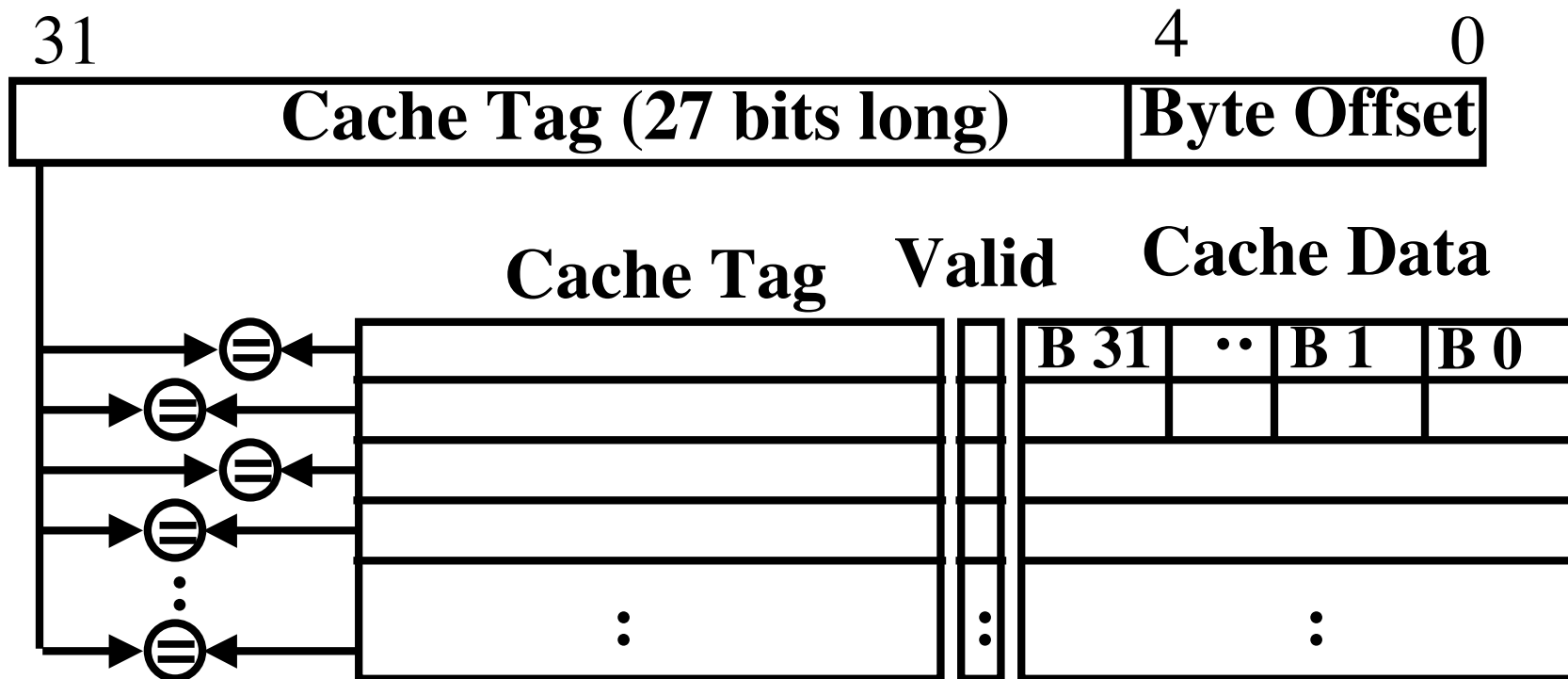
- The direct mapped cache is simple to design and its access time is fast (Why?)
- Good for L1 (on-chip cache)
- **Problem: Conflict Miss, so low hit ratio**

**Conflict Misses** are misses caused by accessing different memory locations that are mapped to the same cache index

In **direct mapped cache**, no flexibility in where memory block can be placed in cache, contributing to conflict misses

# Another Extreme: Fully Associative

- **Fully Associative Cache** (8 word block)
  - Omit cache index; place item in any block!
  - Compare all Cache Tags in parallel



- **By definition: Conflict Misses = 0 for a fully associative cache**



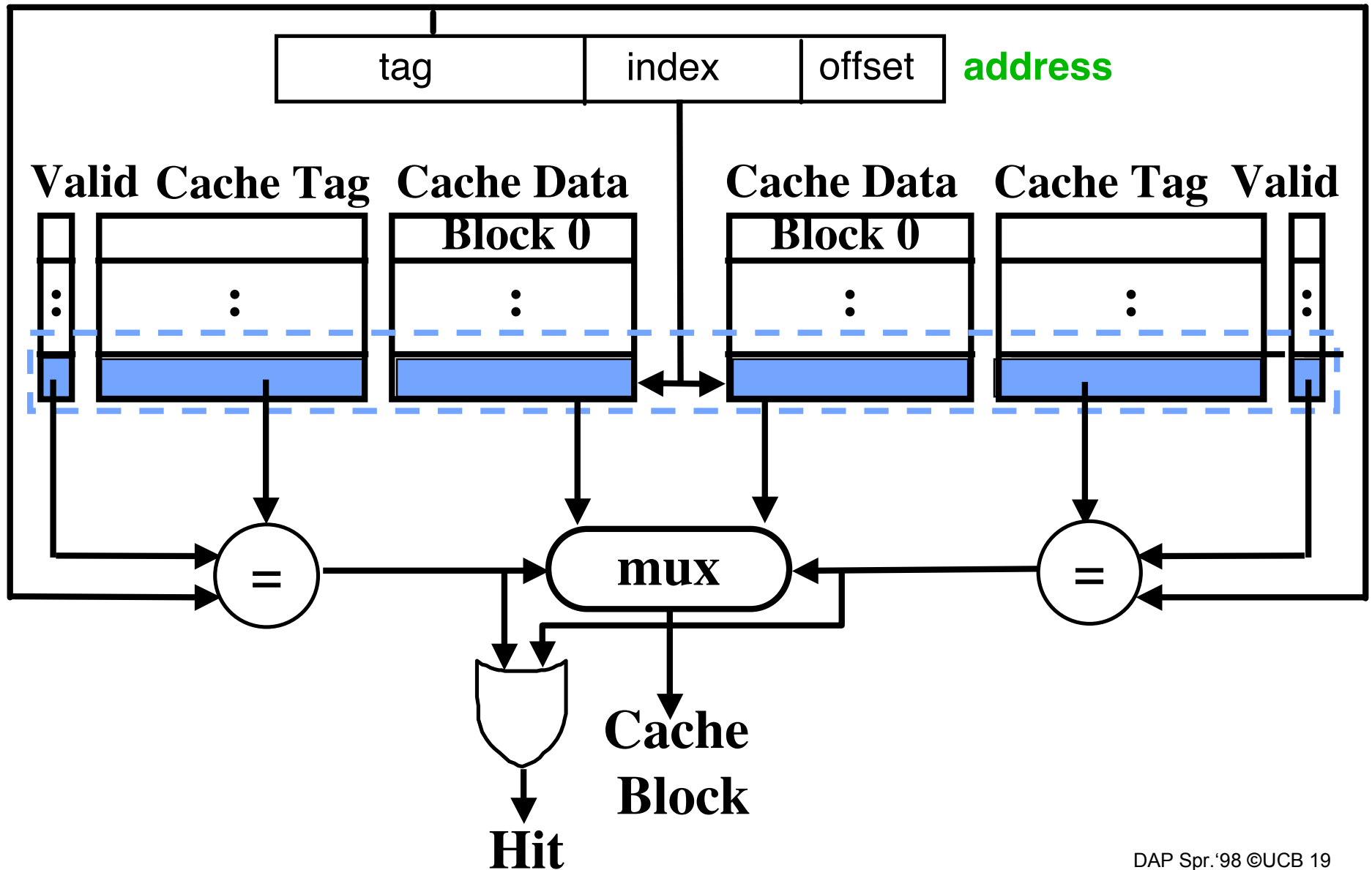
# Fully Associative Cache

- Must search all tags in cache, as item can be in any cache block
- Search for tag must be done by hardware in **parallel** (other searches too slow)
- But, the necessary parallel comparator hardware is very expensive
- Therefore, fully associative placement practical only for a very small cache

# Compromise: N-way Set Associative Cache

- **N-way set associative:**  
N cache blocks for each Cache Index
  - Like having N direct mapped caches operating in parallel
  - Select the one that gets a hit
- **Example: 2-way set associative cache**
  - Cache Index selects a “set” of 2 blocks from the cache
  - The 2 tags in set are compared in parallel
  - Data is selected based on the tag result (which matched the address)

# Example: 2-way Set Associative Cache



# Set Associative Cache Contd.

- Direct Mapped, Fully Associative can be seen as just variations of Set Associative block placement strategy
- Direct Mapped =  
    **1-way** Set Associative Cache
- Fully Associative =  
     **$n$ -way** Set associativity for a cache with exactly  $n$  blocks

# Addressing the Cache

- **Direct mapped cache: one block per set.**



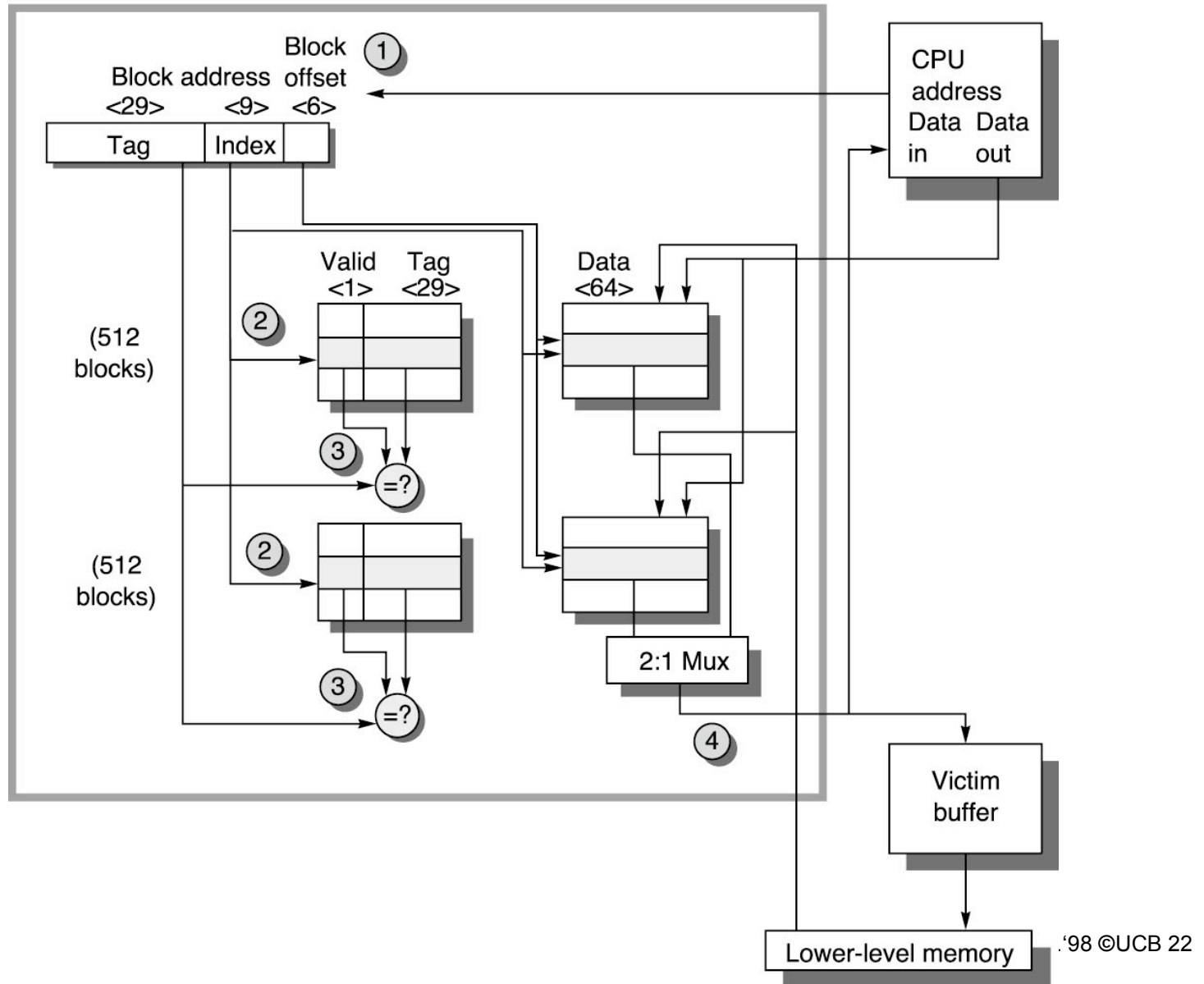
- **Set-associative mapping:  $n/s$  blocks per set.**



- **Fully associative mapping: one set per cache ( $s = n$ ).**



# Alpha 21264 Cache Organization



# Block Replacement Policy

- **N-way Set Associative or Fully Associative have choice where to place a block, (which block to replace)**
  - Of course, if there is an invalid block, use it
- **Whenever get a cache hit, record the cache block that was touched**
- **When need to evict a cache block, choose one which hasn't been touched recently: “Least Recently Used” (LRU)**
  - Past is prologue: history suggests it is least likely of the choices to be used soon
  - Flip side of temporal locality

# Review: Four Questions for Memory Hierarchy Designers

- **Q1: Where can a block be placed in the upper level?**  
*(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
- **Q2: How is a block found if it is in the upper level?**  
*(Block identification)*
  - Tag/Block
- **Q3: Which block should be replaced on a miss?**  
*(Block replacement)*
  - Random, LRU
- **Q4: What happens on a write?**  
*(Write strategy)*
  - Write Back or Write Through (with Write Buffer)