

**CPS104**  
**Computer Organization and Programming**  
**Lecture 17: The Cache**

**Oct. 27, 1999**

**Dietolf (Dee) Ramm**

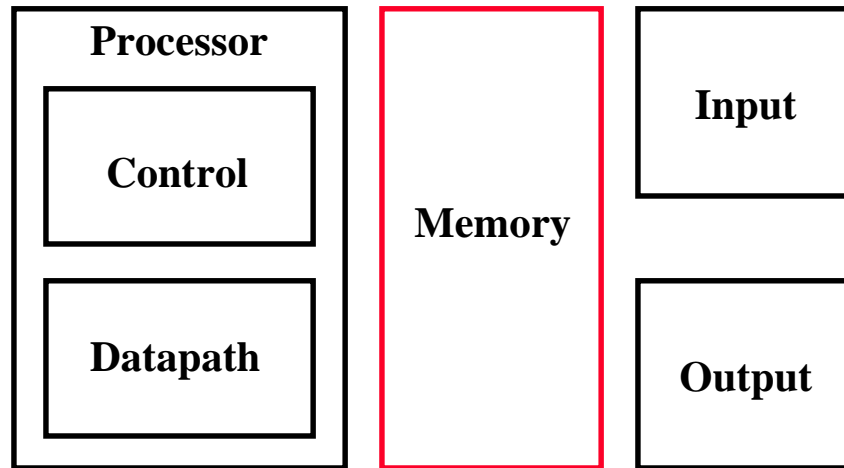
**<http://www.cs.duke.edu/~dr/cps104.html>**

# Outline of Today's Lecture

- **The Memory Hierarchy**
- **Direct Mapped Cache.**
- **Two-Way Set Associative Cache**
- **Fully Associative cache**
- **Replacement Policies**
- **Write Strategies**

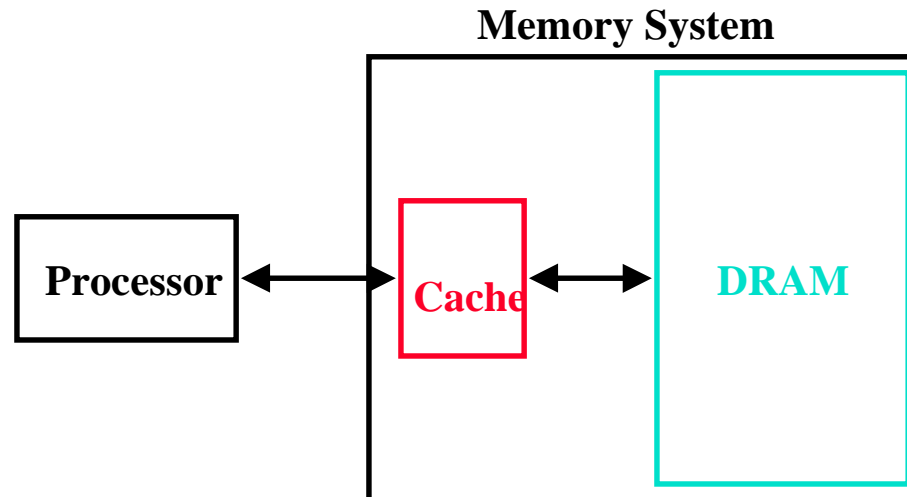
# The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



- Today's Topic: Memory System

# The Motivation for Caches



- **Motivation:**
  - Large memories (DRAM) are **slow**
  - Small memories (SRAM) are **fast**
- Make the *average access time* small by:
  - Servicing most accesses from a small, fast memory.
- Reduce the *bandwidth* required of the large memory

# Levels of the Memory Hierarchy

*Capacity*  
*Access Time*  
*Cost*

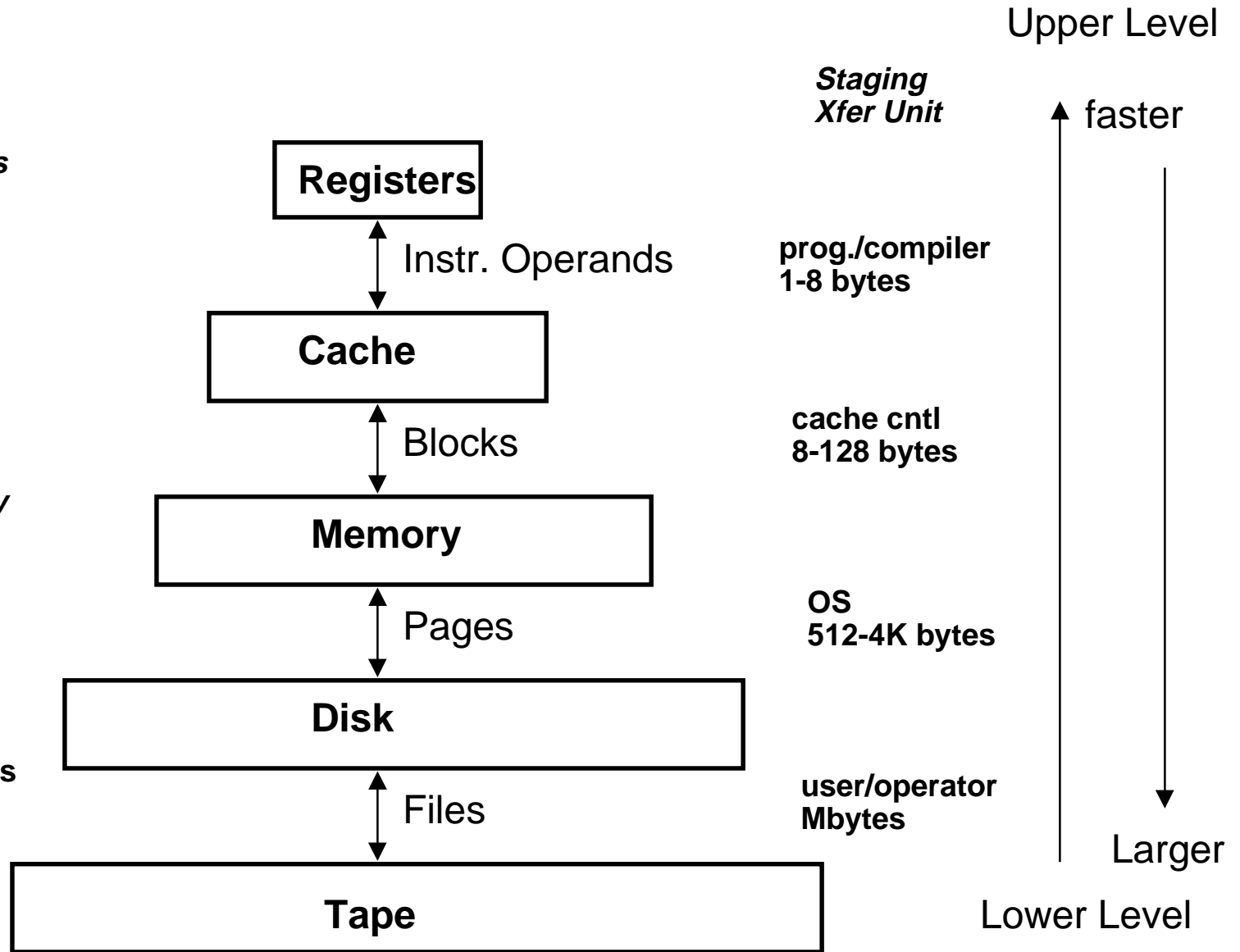
*CPU Registers*  
100s Bytes  
<10s ns

*Cache*  
K Bytes  
10-100 ns  
~\$.0005/bit

*Main Memory*  
M Bytes  
100ns-1us  
~\$.00001/bit

*Disk*  
G Bytes  
ms<sub>3</sub> - 10<sup>-4</sup>  
10<sup>-3</sup> - 10 cents

*Tape*  
infinite  
sec-min  
10<sup>-6</sup>



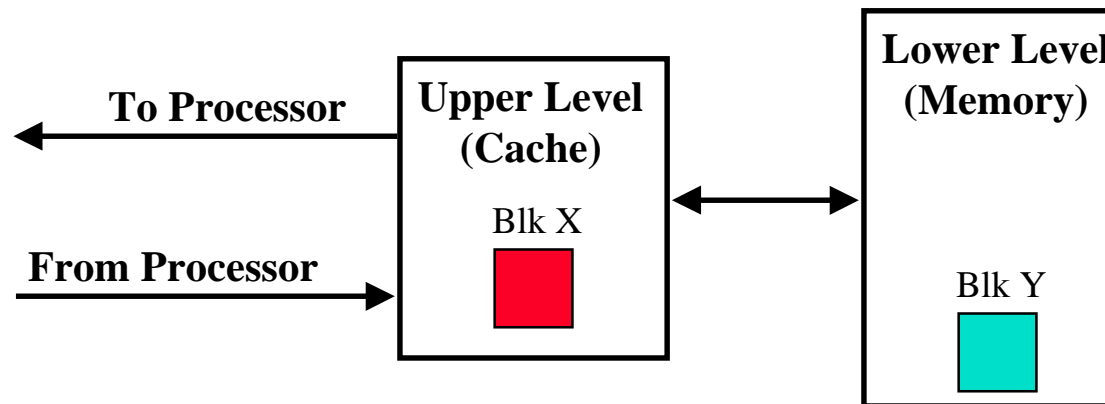
# The Principle of Locality



- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
  - Example: **90% of time in 10% of the code**
- Two Different Types of Locality:
  - **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
  - **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.

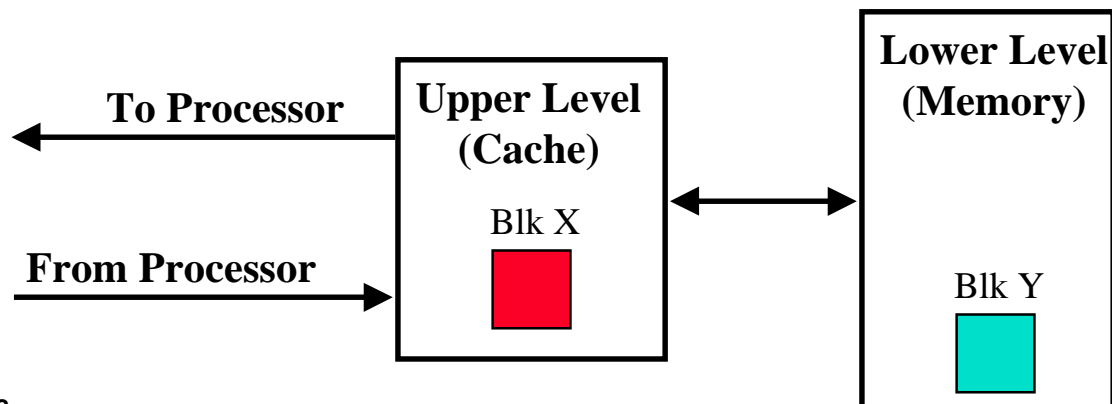
# Memory Hierarchy: Principles of Operation

- At any given time, data is copied between only 2 adjacent levels:
  - Upper Level (Cache) : the one closer to the processor
    - Smaller, faster, and uses more expensive technology
  - Lower Level (Memory): the one further away from the processor
    - Bigger, slower, and uses less expensive technology
- **Block:**
  - The minimum unit of information that can either be present or not present in the two level hierarchy



# Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
  - **Hit Rate**: the fraction of memory access found in the upper level
  - **Hit Time**: Time to access the upper level which consists of  
RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
  - **Miss Rate** =  $1 - (\text{Hit Rate})$
  - **Miss Penalty** = Time to replace a block in the upper level +  
Time to deliver the block the processor
- **Hit Time**  $\ll$  **Miss Penalty**



## Direct Mapped Cache

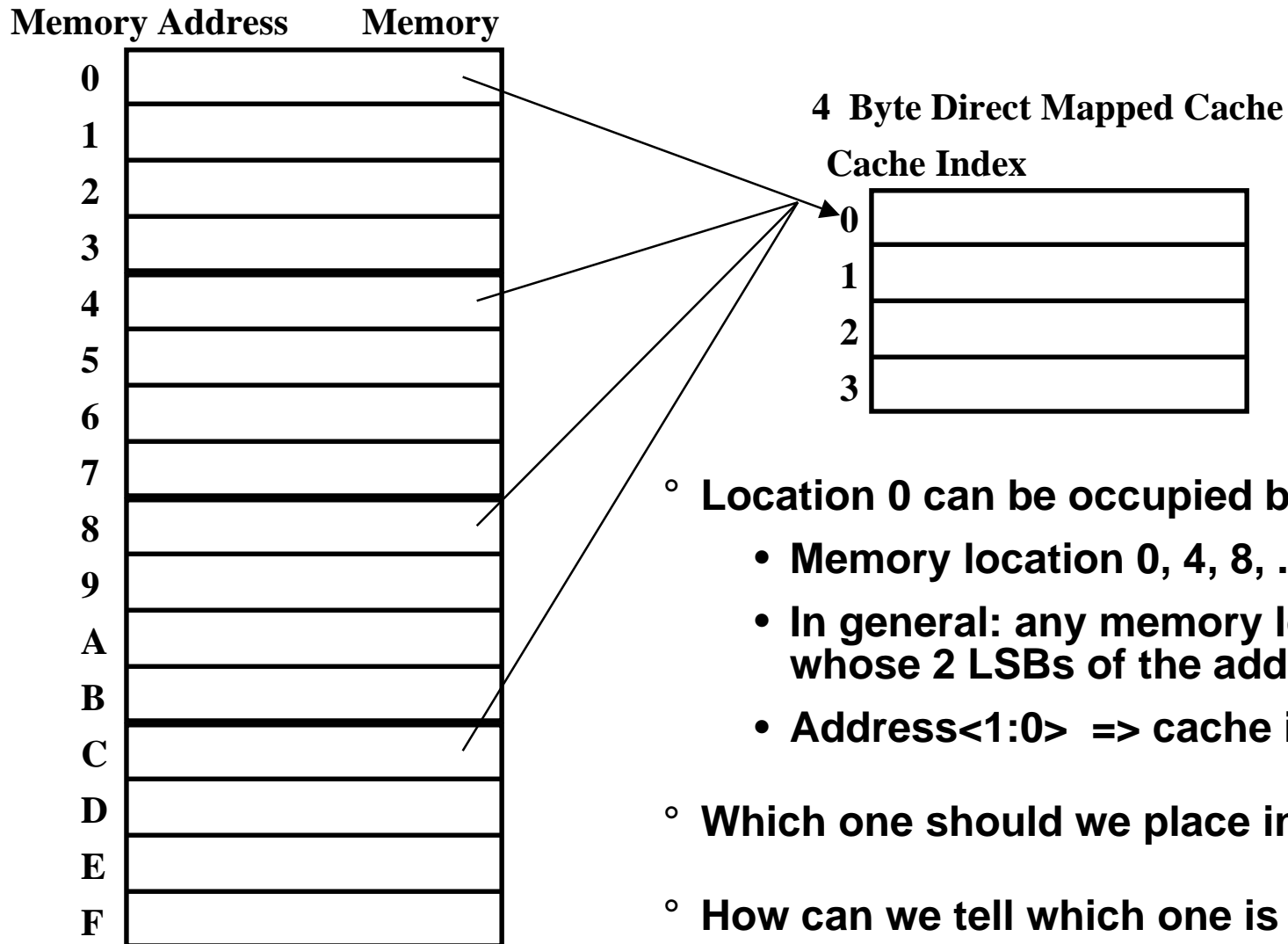
- Direct Mapped cache is an array of fixed size **blocks**. Each **block** holds consecutive bytes of main memory data.
- The **Tag Array** holds the **Block Memory Address**.
- A **valid bit** associated with each cache block tells if the data is valid.
  - **Cache Index**: The location of a block (and it's tag) in the cache.
  - **Block Offset**: The byte location in the cache block.

**Cache-Index** = ( $\langle \text{Address} \rangle \text{ Mod } (\text{Cache\_Size})$ ) / **Block\_Size**

**Block-Offset** =  $\langle \text{Address} \rangle \text{ Mod } (\text{Block\_Size})$

**Tag** =  $\langle \text{Address} \rangle / (\text{Cache\_Size})$

# The Simplest Cache: Direct Mapped Cache



- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index
- Which one should we place in the cache?
- How can we tell which one is in the cache?

## Direct Mapped Cache (Cont.)

For a Cache of  $2^M$  bytes with block size of  $2^L$  bytes

- There are  $2^{M-L}$  cache blocks,
- Lowest  $L$  bits of the address are **Block-Offset** bits
- Next  $(M - L)$  bits are the **Cache-Index**.
- The last  $(32 - M)$  bits are the **Tag** bits.



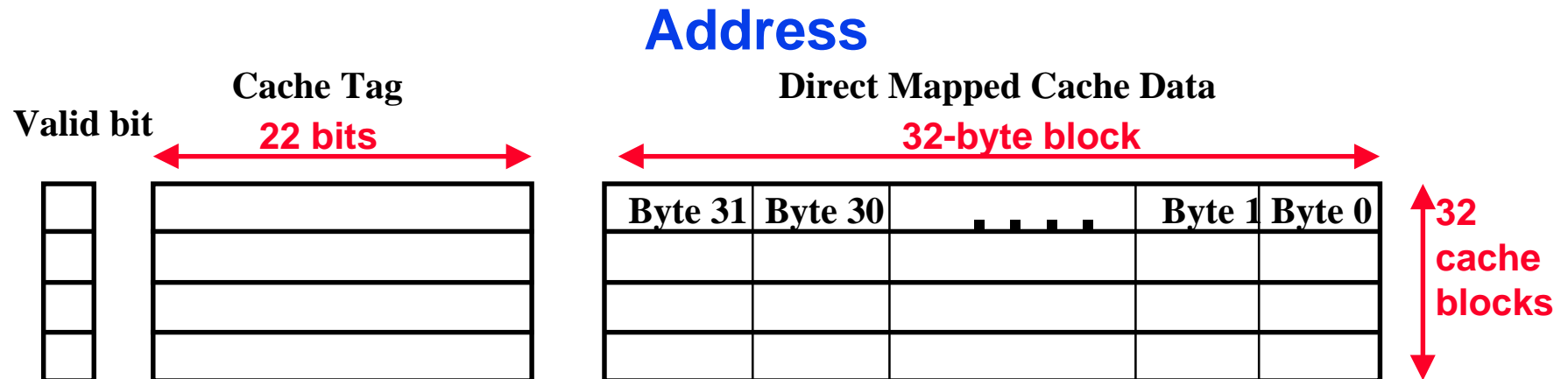
**Data Address**

## Example: 1-KB Cache with 32B blocks:

$$\text{Cache Index} = (\text{Address} \bmod 1024) / 32$$

$$\text{Block-Offset} = \text{Address} \bmod 32$$

$$\text{Tag} = \text{Address} / 1024$$

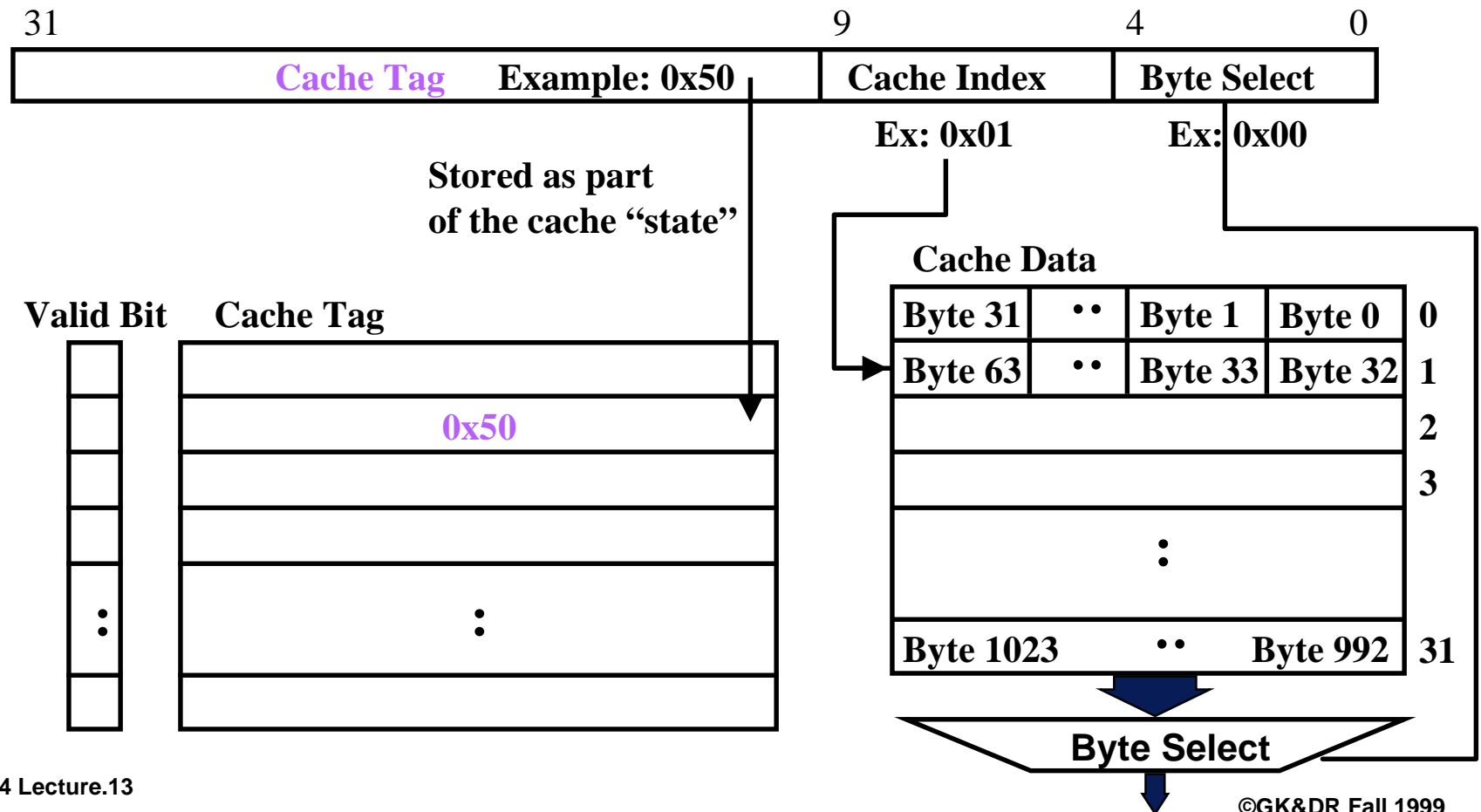


$$1\text{K} = 2^{10} = 1024$$

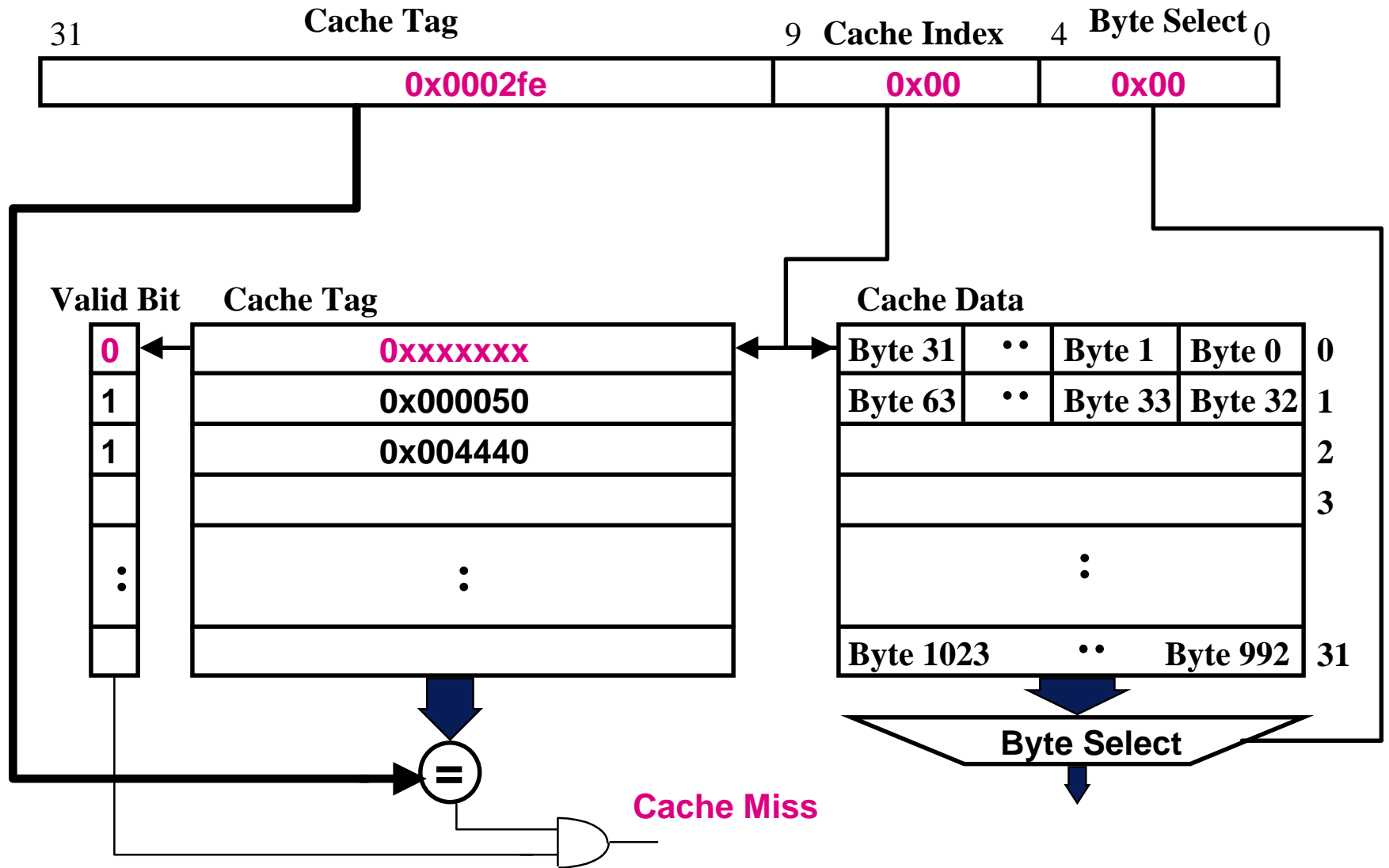
$$2^5 = 32$$

# Example: 1KB Direct Mapped Cache with 32B Blocks

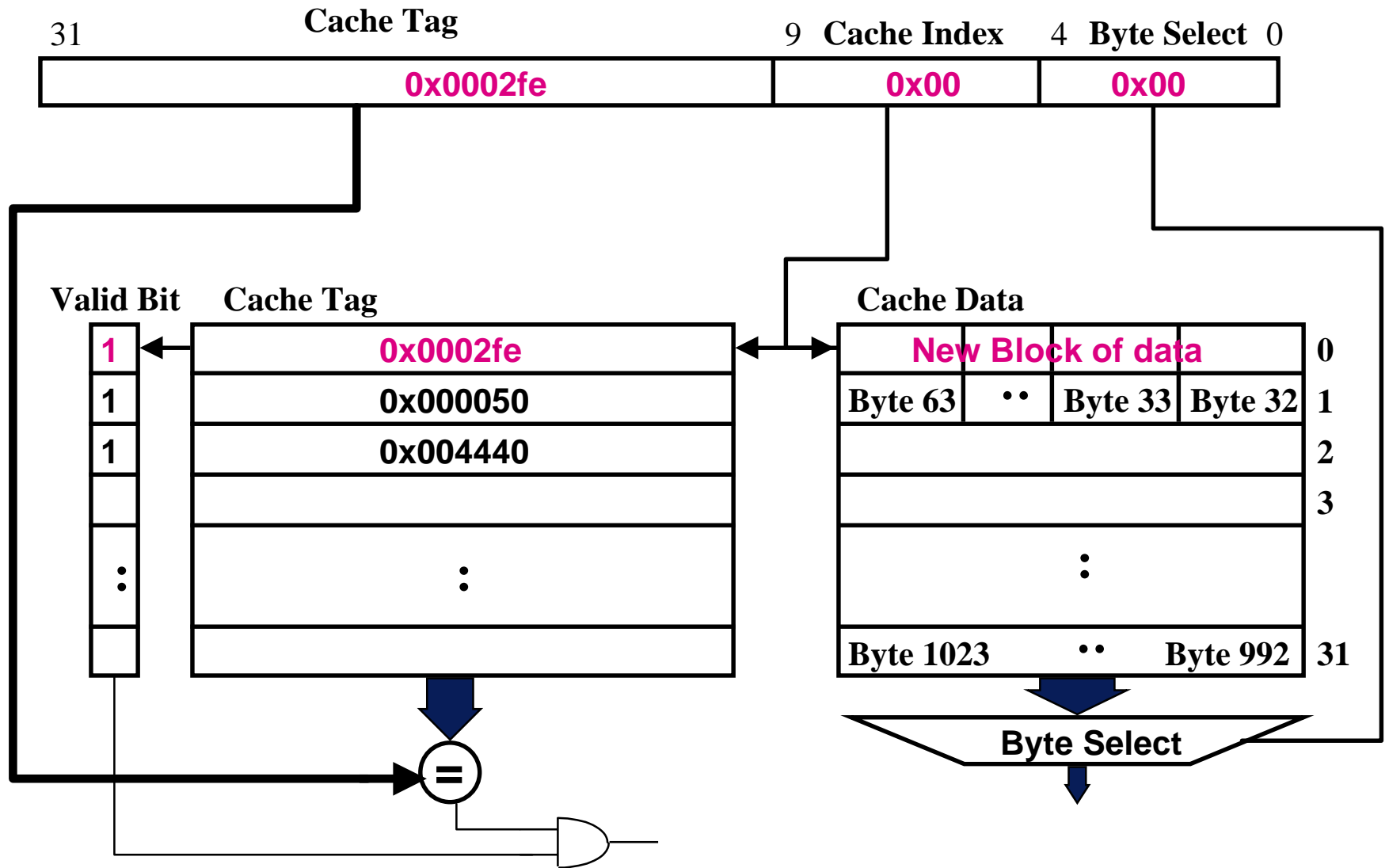
- For a **1024 ( $2^{10}$ )** byte cache with **32-byte blocks**:
  - The uppermost **22 = (32 - 10)** address bits are the **Cache Tag**
  - The lowest **5** address bits are the **Byte Select** (Block Size =  $2^5$ )
  - The next **5** address bits (**bit5 - bit9**) are the **Cache Index**



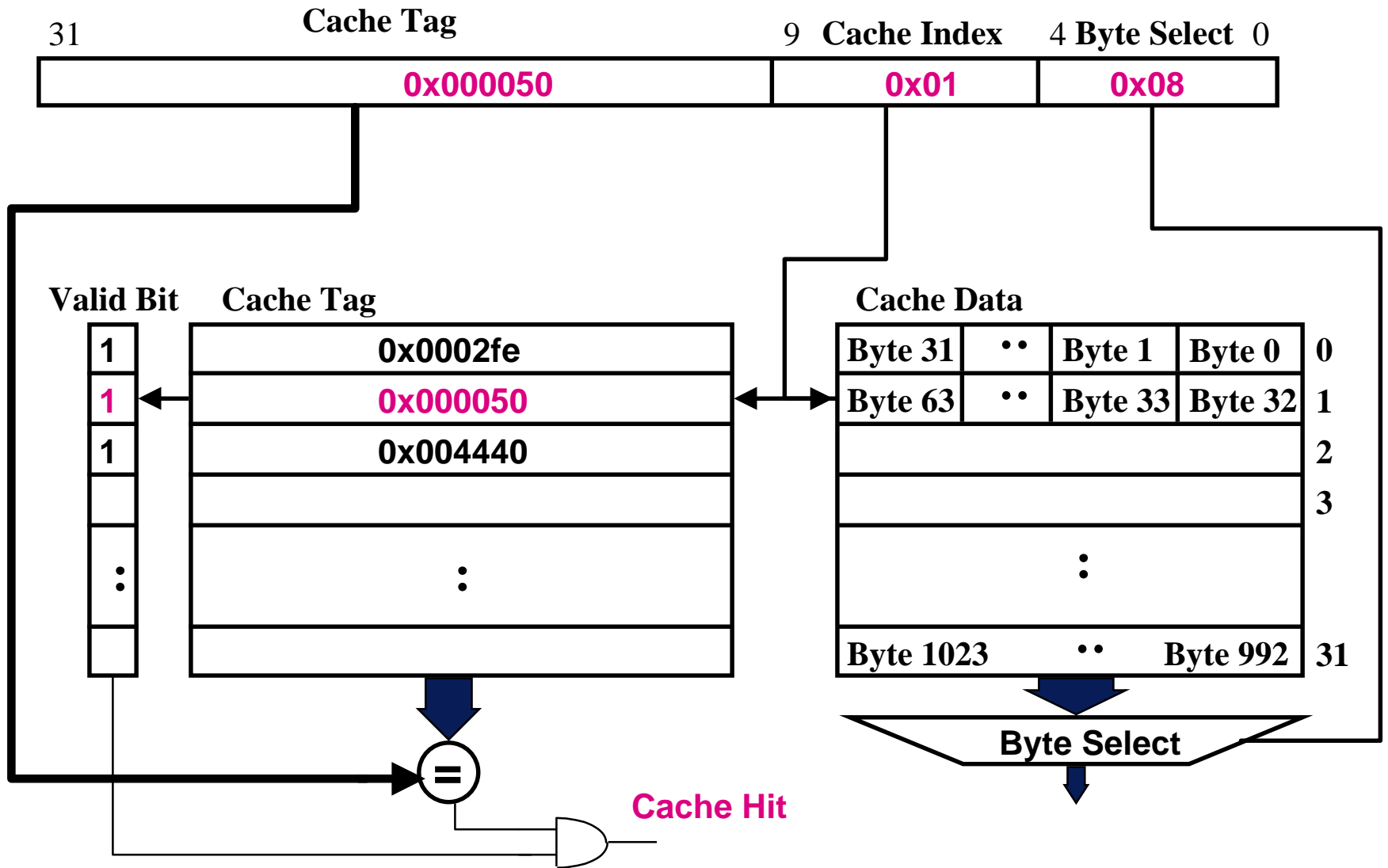
# Example: 1K Direct Mapped Cache



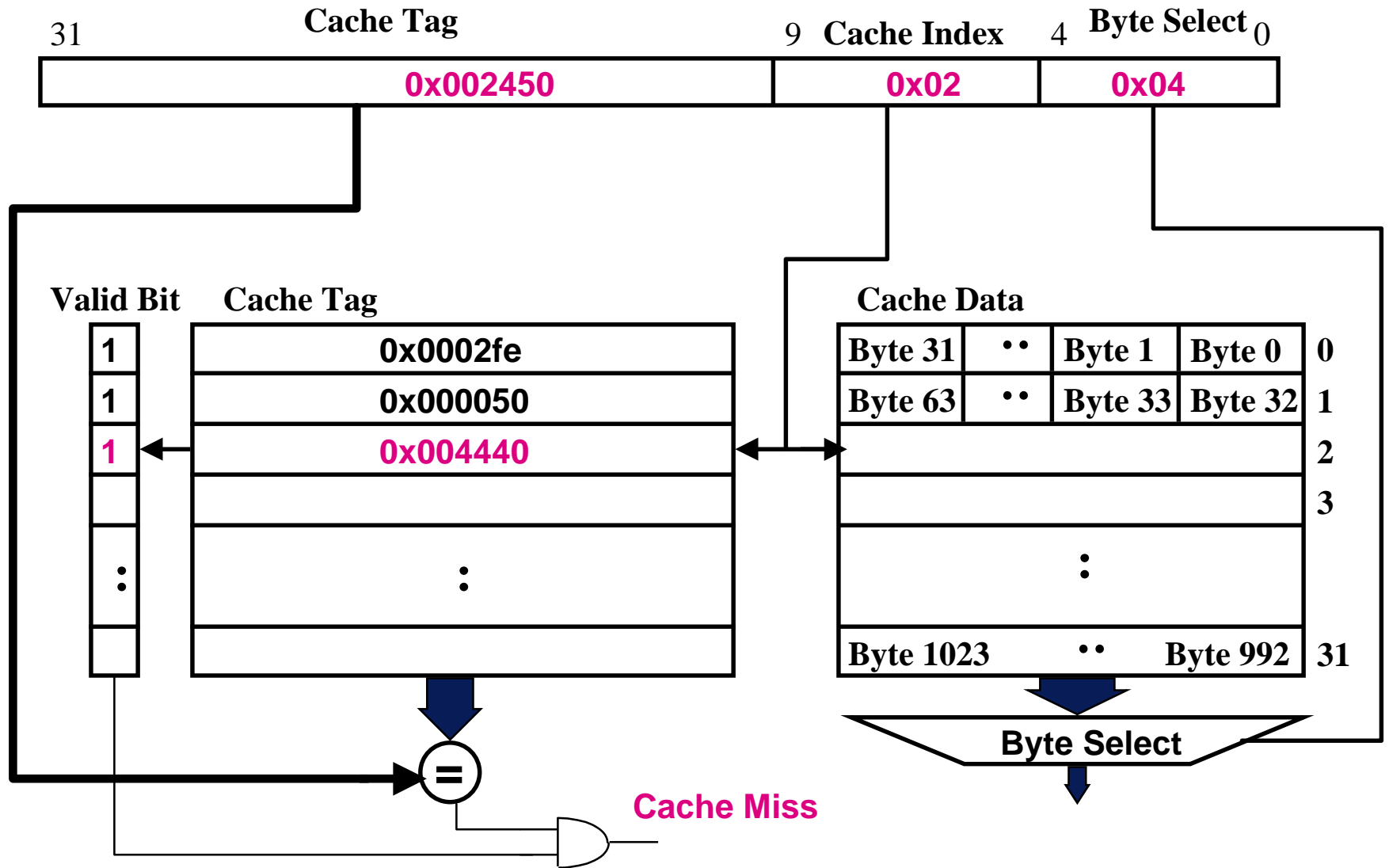
# Example: 1K Direct Mapped Cache



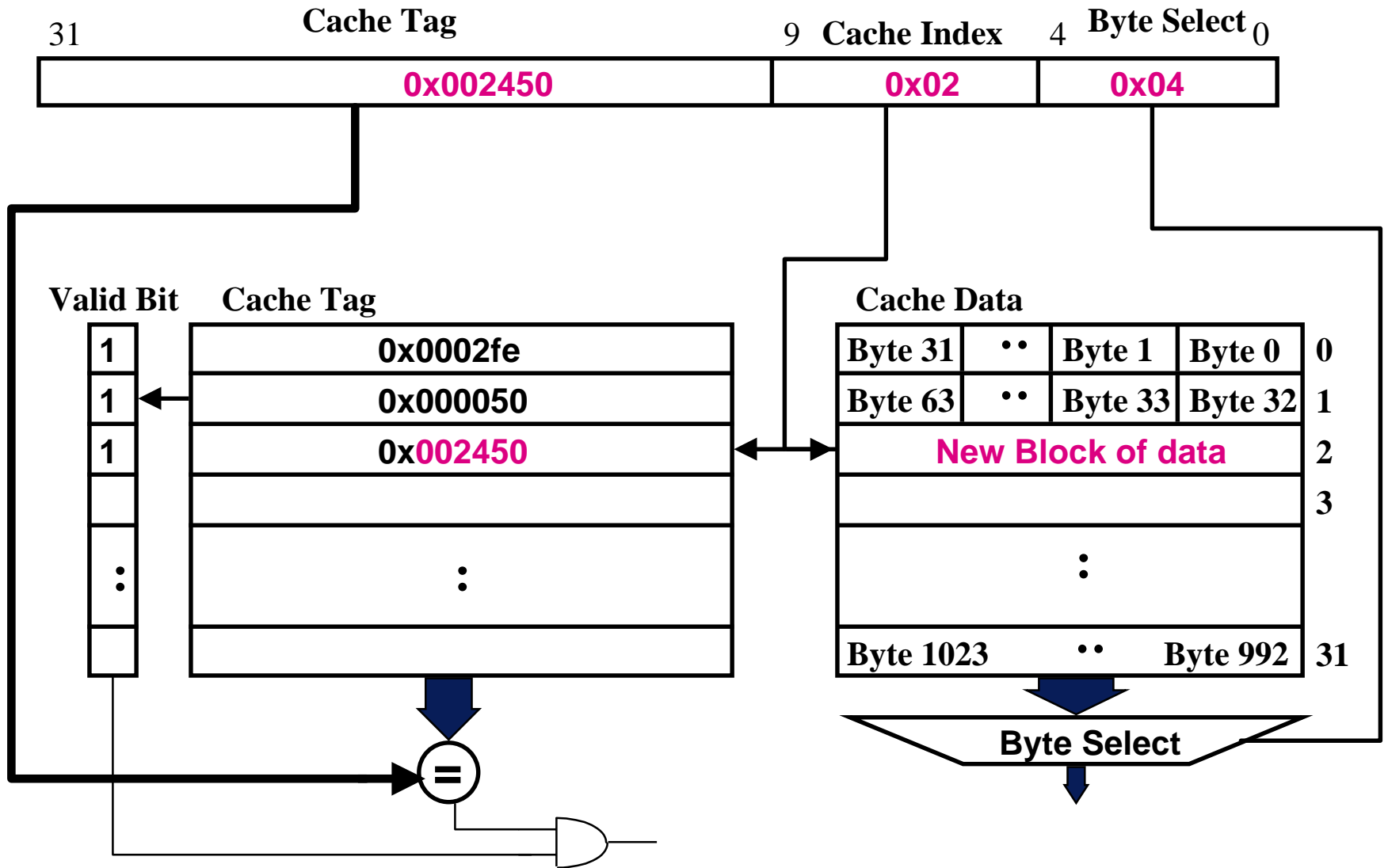
# Example: 1K Direct Mapped Cache



# Example: 1K Direct Mapped Cache

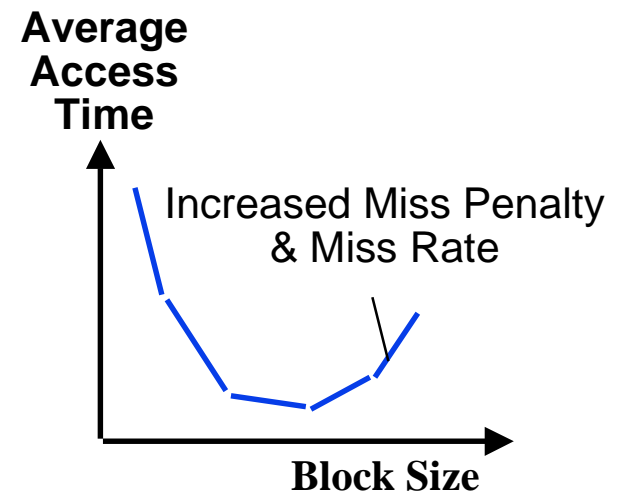
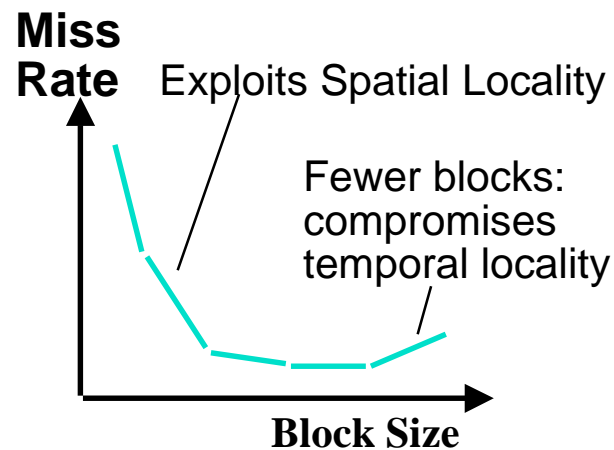
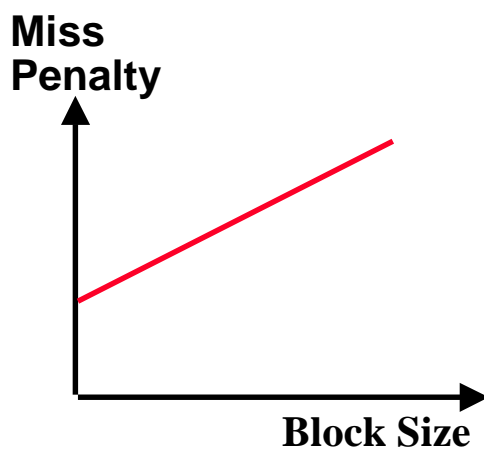


# Example: 1K Direct Mapped Cache



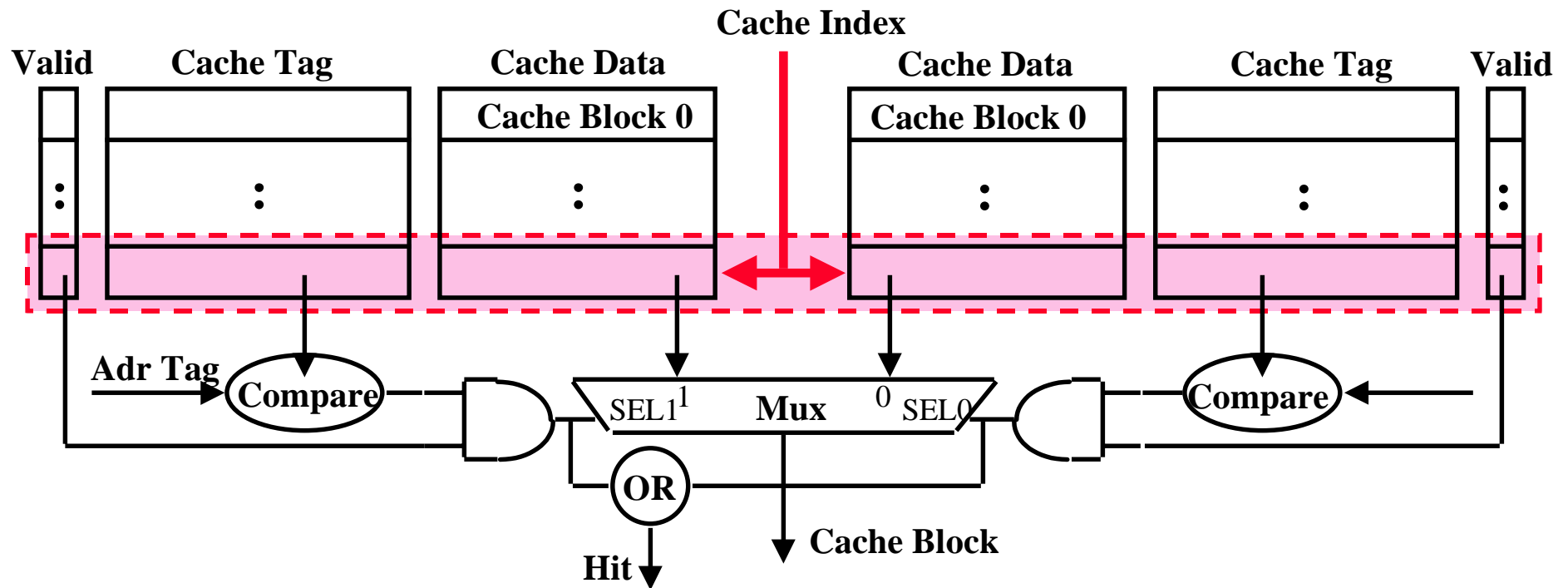
# Block Size Tradeoff

- In general, larger block size take advantage of spatial locality **BUT**:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- In general, Average Access Time:
  - **Hit Time x (1 - Miss Rate) + Miss Penalty x Miss Rate**



# A N-way Set Associative Cache

- **N-way set associative:** N entries for each Cache Index
  - N direct mapped caches operating in parallel
- **Example:** Two-way set associative cache
  - Cache 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

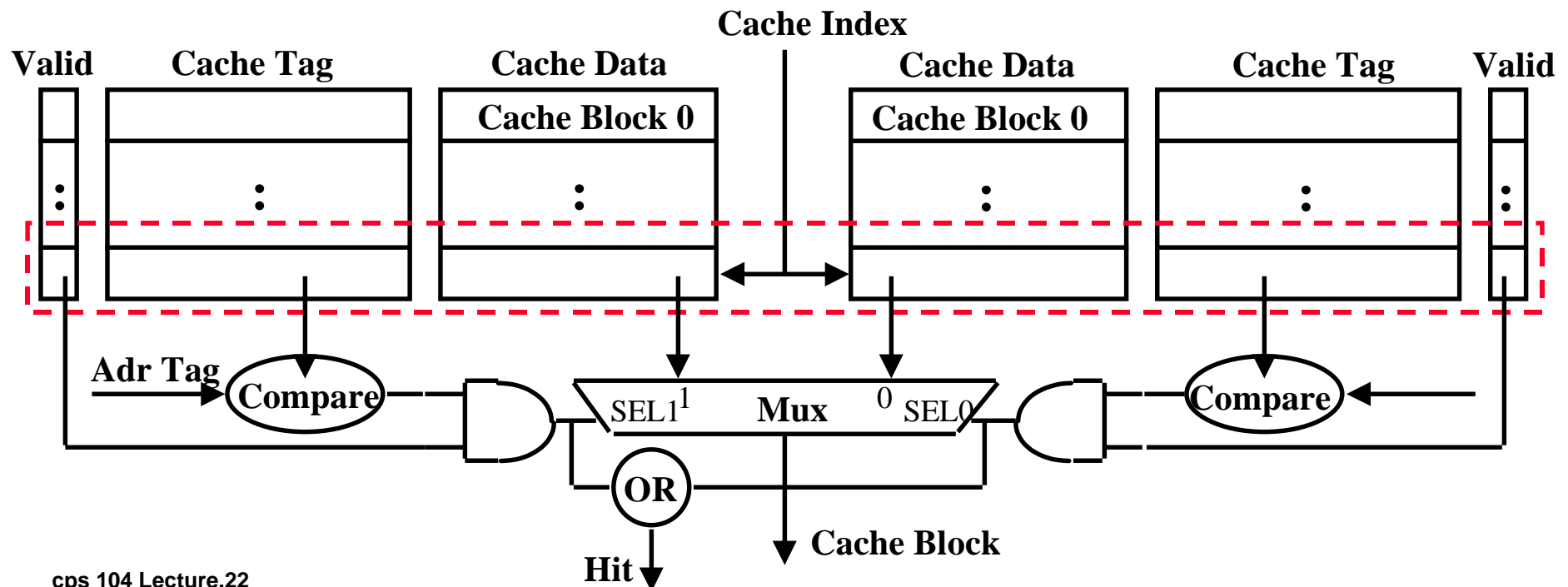


## Advantages of Set associative cache

- Higher **Hit rate** for the same cache size.
- Fewer **Conflict Misses**.
- Can have a larger cache but keep the index smaller  
(**same size as virtual page index**)

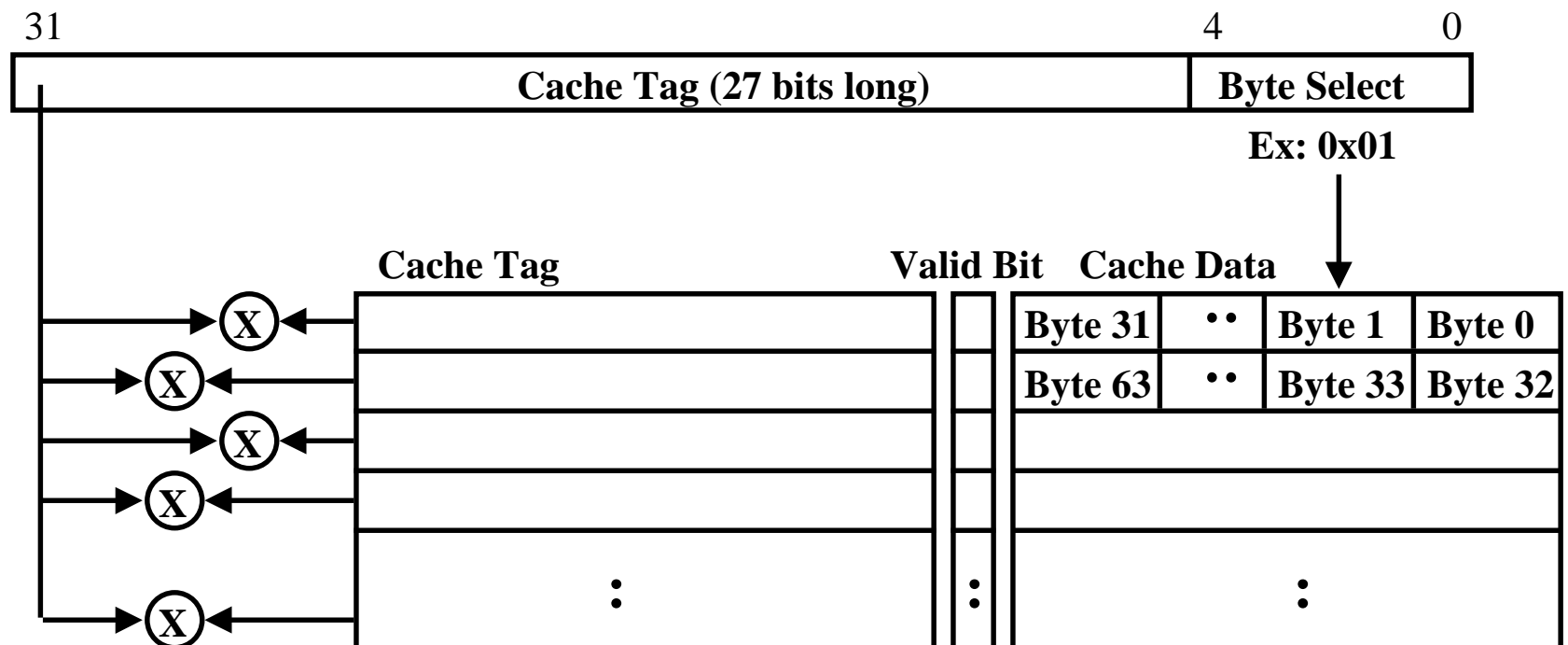
## Disadvantage of Set Associative Cache

- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes **AFTER** Hit/Miss decision and set selection
- In a direct mapped cache, Cache Block is available **BEFORE** Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



# And yet Another Extreme Example: Fully Associative cache

- **Fully Associative Cache** -- push the set associative idea to its limit!
  - Forget about the Cache Index
  - Compare the Cache Tags of **all cache entries** in parallel
  - Example: Block Size = 32B blocks, we need **N** 27-bit comparators
- By definition: **Conflict Miss = 0** for a fully associative cache



# Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase Associativity
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Invalidation**: other process (e.g., I/O) updates memory

## Sources of Cache Misses

	Direct Mapped	N-way Set Associative	Fully Associative
Cache Size	Big	Medium	Small
Compulsory Miss	Same	Same	Same
Conflict Miss	High	Medium	Zero
Capacity Miss	Low(er)	Medium	High
Invalidation Miss	Same	Same	Same

**Note:**

**If you are going to run “billions” of instruction, Compulsory Misses are insignificant.**

# The Need to Make a Decision!

- **Direct Mapped Cache:**
  - Each memory location can only mapped to 1 cache location
  - No need to make any decision :-)
    - Current item replaced the previous item in that cache location
- **N-way Set Associative Cache:**
  - Each memory location have a **choice of N** cache locations
- **Fully Associative Cache:**
  - Each memory location can be placed in **ANY** cache location
- **Cache miss in a N-way Set Associative or Fully Associative Cache:**
  - Bring in new block from memory
  - Throw out a cache block to make room for the new block
  - We need to make a decision on **which block to throw out!**

# Cache Block Replacement Policy

- **Random Replacement:**

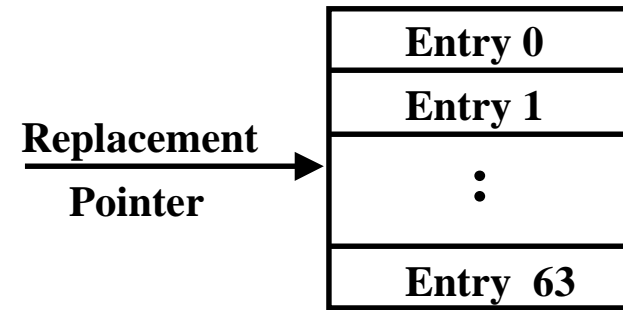
- ◆ Hardware randomly selects a cache block out of the set and replaces it.

- **Least Recently Used:**

- ◆ Hardware keeps track of the access history
- ◆ Replace the entry that has not been used for the longest time.
- ◆ For **two way set associative** cache one needs **one bit** for LRU replacement.

- **Example of a Simple “Pseudo” Least Recently Used Implementation:**

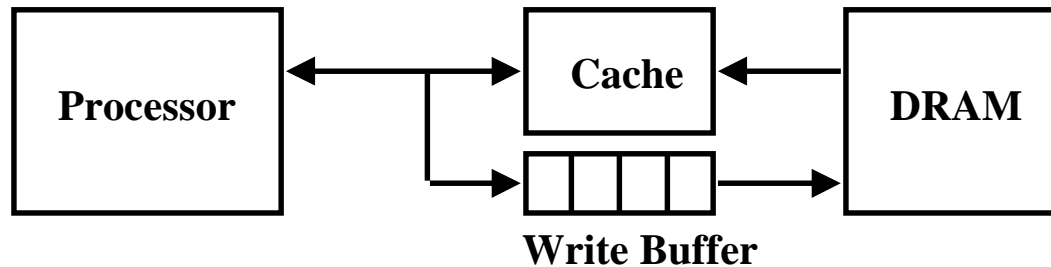
- ◆ Assume 64 Fully Associative Entries
- ◆ Hardware replacement pointer points to one cache entry
- ◆ Whenever an access is made to the entry the pointer points to:
  - Move the pointer to the next entry
  - Otherwise: do not move the pointer



# Cache Write Policy: Write Through versus Write Back

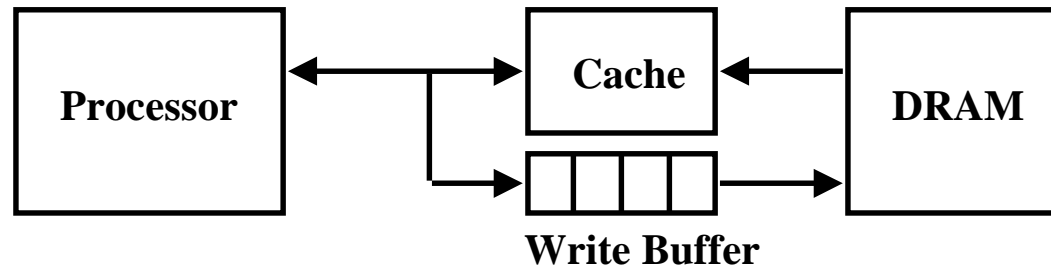
- Cache read is much easier to handle than cache write:
  - Instruction cache is much easier to design than data cache
- Cache write:
  - How do we keep data in the cache and memory consistent?
- Two options (decision time again :-)
  - **Write Back**: write to cache only. Write the cache block to memory when that cache block is being replaced on a cache miss.
    - Need a “**dirty bit**” for each cache block
    - Greatly reduce the memory bandwidth requirement
    - Control can be complex
  - **Write Through**: write to cache and memory at the same time.
    - What!!! How can this be? Isn't memory too slow for this?

## Write Buffer for Write Through

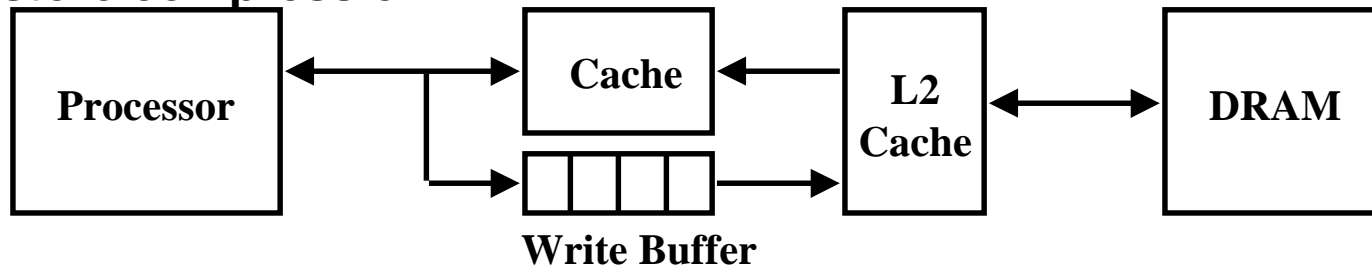


- **A Write Buffer is needed between the Cache and Memory**
  - **Processor:** writes data into the cache and the write buffer
  - **Memory controller:** write contents of the buffer to memory
- **Write buffer is just a FIFO:**
  - **Typical number of entries: 4**
  - **Works fine if: Store frequency (w.r.t. time)  $\ll 1 / \text{DRAM write cycle}$**
- **Memory system designer's nightmare:**
  - **Store frequency (w.r.t. time)  $> 1 / \text{DRAM write cycle}$**
  - **Write buffer saturation**

# Write Buffer Saturation

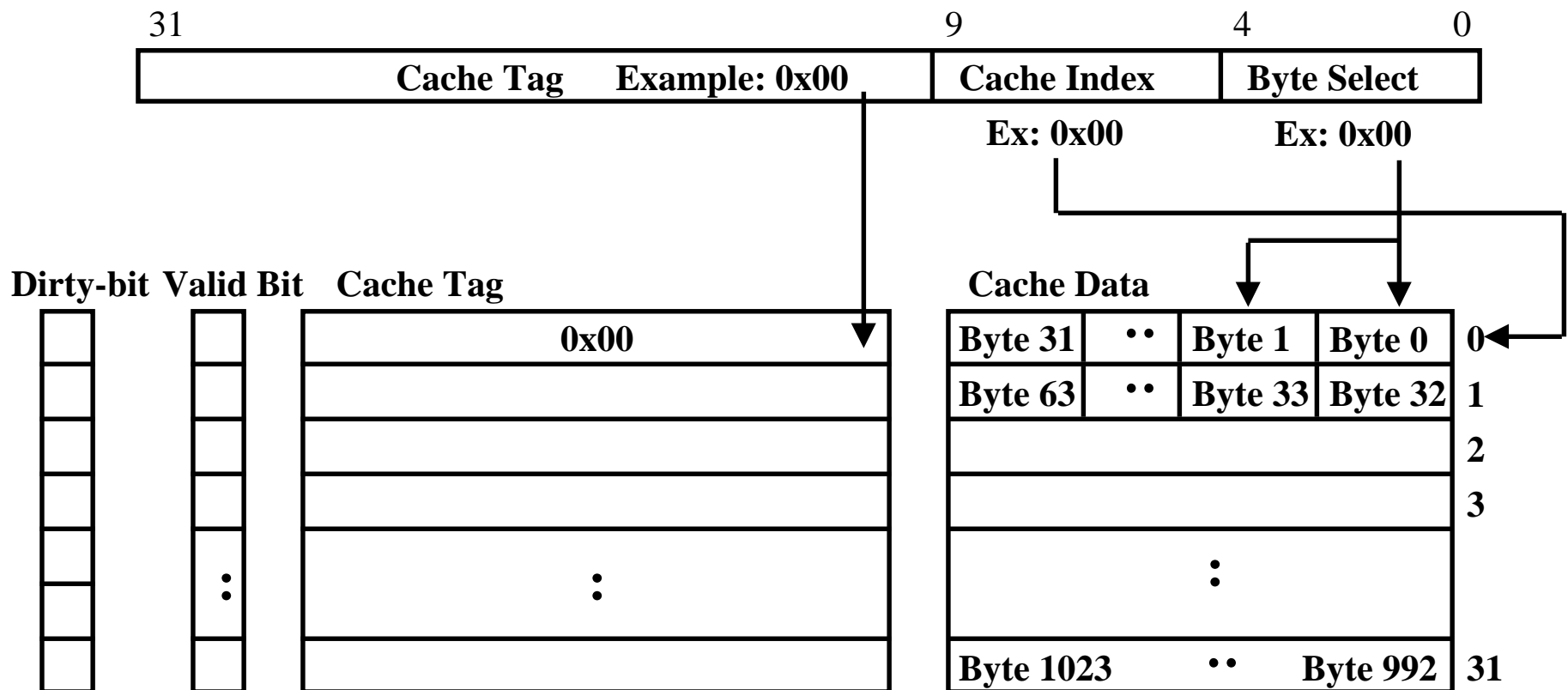


- Store frequency (w.r.t. time)  $\rightarrow 1 / \text{DRAM write cycle}$ 
  - If this condition exist for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - The CPU Cycle Time  $\ll$  DRAM Write Cycle Time
- Solution for write buffer saturation:
  - Use a write back cache
  - Install a second level (L2) cache:
  - store compression



# Write Allocate versus Not Allocate

- Assume: a 16-bit write to memory location 0x0 and causes a miss
  - Do we read in the block?
    - Yes: Write Allocate**
    - No: Write Not Allocate**



# Four Questions for Memory Hierarchy Designers

- **Q1:** Where can a block be placed in the upper level?  
*(Block placement)*
- **Q2:** How is a block found if it is in the upper level?  
*(Block identification)*
- **Q3:** Which block should be replaced on a miss?  
*(Block replacement)*
- **Q4:** What happens on a write?  
*(Write strategy)*

# What is a Sub-block?

- Sub-block:
  - Share one cache tag between all sub-blocks in a block
  - A unit within a block that has its own valid bit
  - Example: 1 KB Direct Mapped Cache, 32-B Block, 8-B Sub-block
    - Each cache entry will have:  $32/8 = 4$  valid bits
- Write miss: only the bytes in that sub-block is brought in.
  - **reduce cache fill bandwidth (penalty).**

