CS 110
Computer Architecture

Caches Part II

Instructors:

Sören Schwertfeger & Chundong Wang
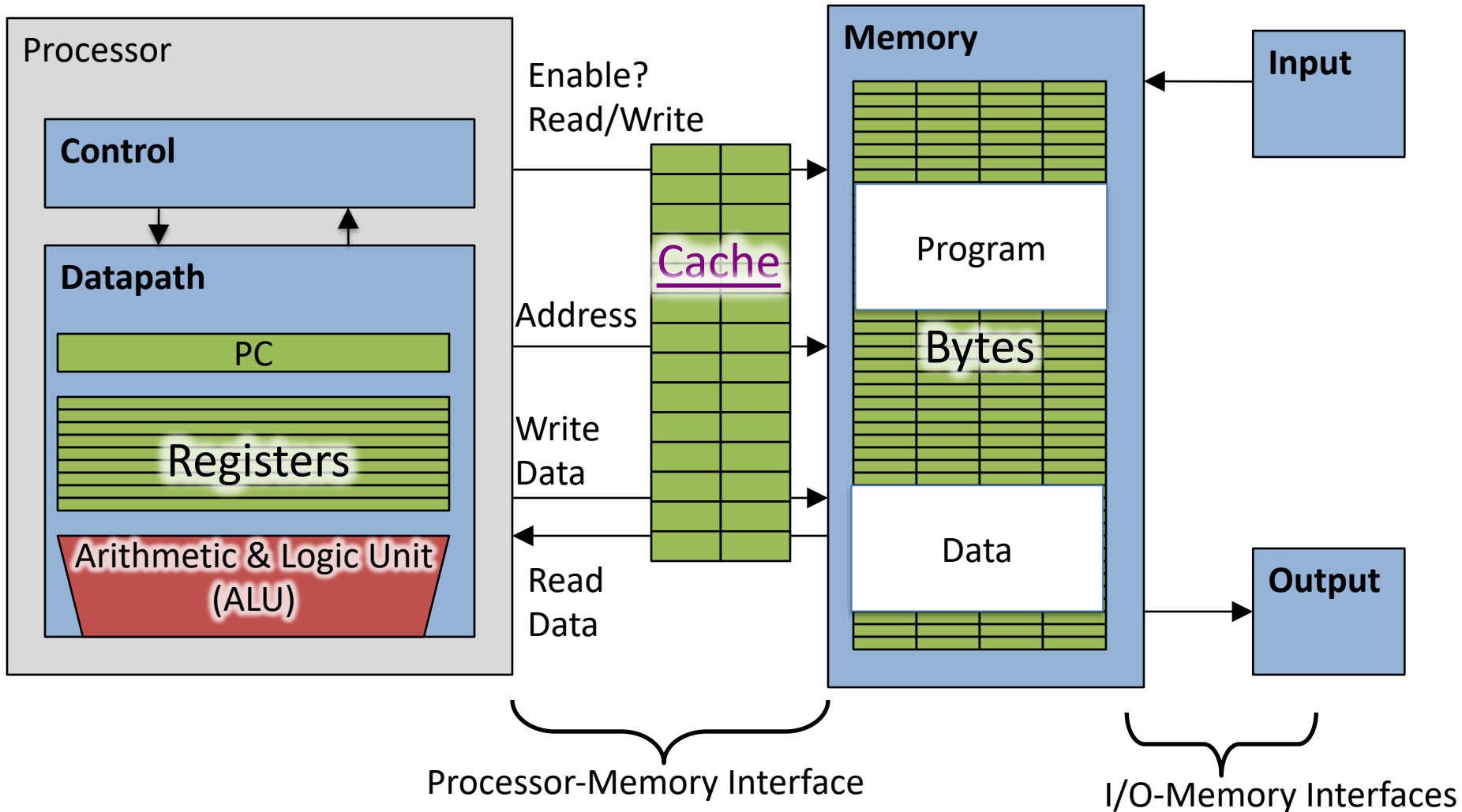
https://robotics.shanghaitech.edu.cn/courses/ca/22s/

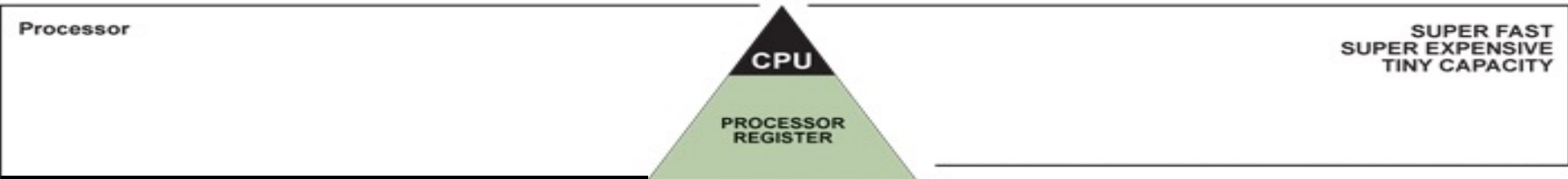**School of Information Science and Technology SIST**

ShanghaiTech University

Slides based on UC Berkeley's CS61C

Adding Cache to Computer

Great Idea #3: Principle of Locality / Memory Hierarchy

Processor

SUPER FAST
SUPER EXPENSIVE
TINY CAPACITY

CPU

PROCESSOR
REGISTER

EDO, SD-RAM, DDR-SDRAM, RD-RAM
and More...

PHYSICAL MEMORY

RAMDOM ACCESS MEMORY (RAM)

FAST
**PRICED REASONABLY**
AVERAGE CAPACITY

Big Idea: Locality

- *Temporal Locality* (locality in time)
  - If a memory location is referenced, then it will tend to be referenced again soon

- *Spatial Locality* (locality in space)
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

```
// Sample code for CS110@Spring 2022 -- Chundong
for (i = 0, sum = 0; i < n; sum += a[i], ++i);
```

4

Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 1K words
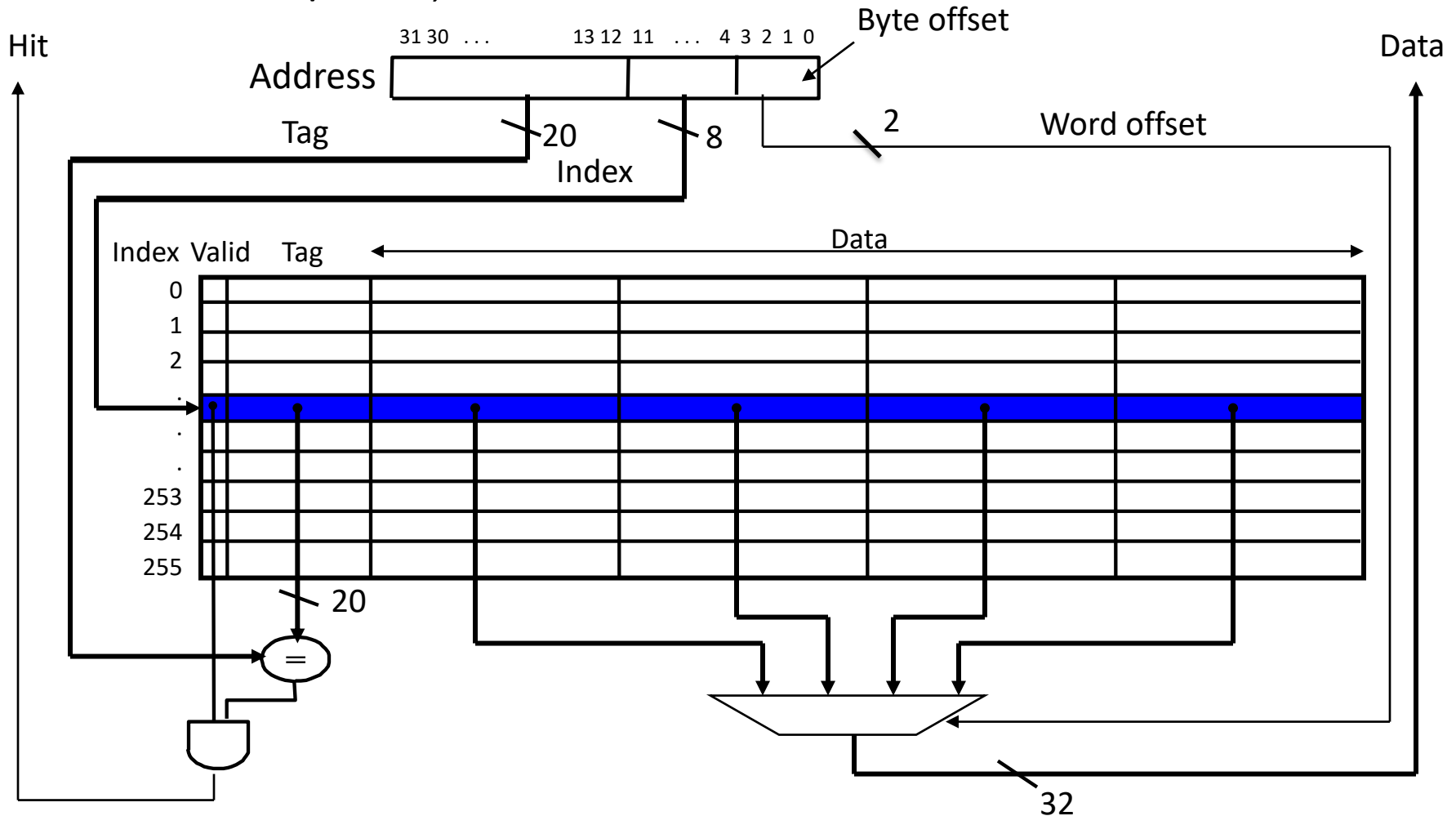
Cache Names for Each Organization

- "Fully Associative": Line can go anywhere
  - First design in lecture
  - Note: No Index field, but 1 comparator/ line
- "Direct Mapped": Line goes one place
  - Note: Only 1 comparator
  - Number of sets = number blocks
- "N-way Set Associative": N places for a line
  - Number of sets = number of lines/ N
  - N comparators
  - *Fully Associative: N = number of lines*
  - *Direct Mapped: N = 1*

Range of Set-Associative Caches

- For a fixed-size cache, and a given block size, each increase by a factor of 2 in associativity doubles the number of blocks per set (i.e., the number of "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

More Associativity (more ways)

Tag	Index	Block offset

Total Cache Capacity =

Associativity * # of sets * block_size

*Bytes = blocks/set * sets * Bytes/block*

*C = N * S * B*

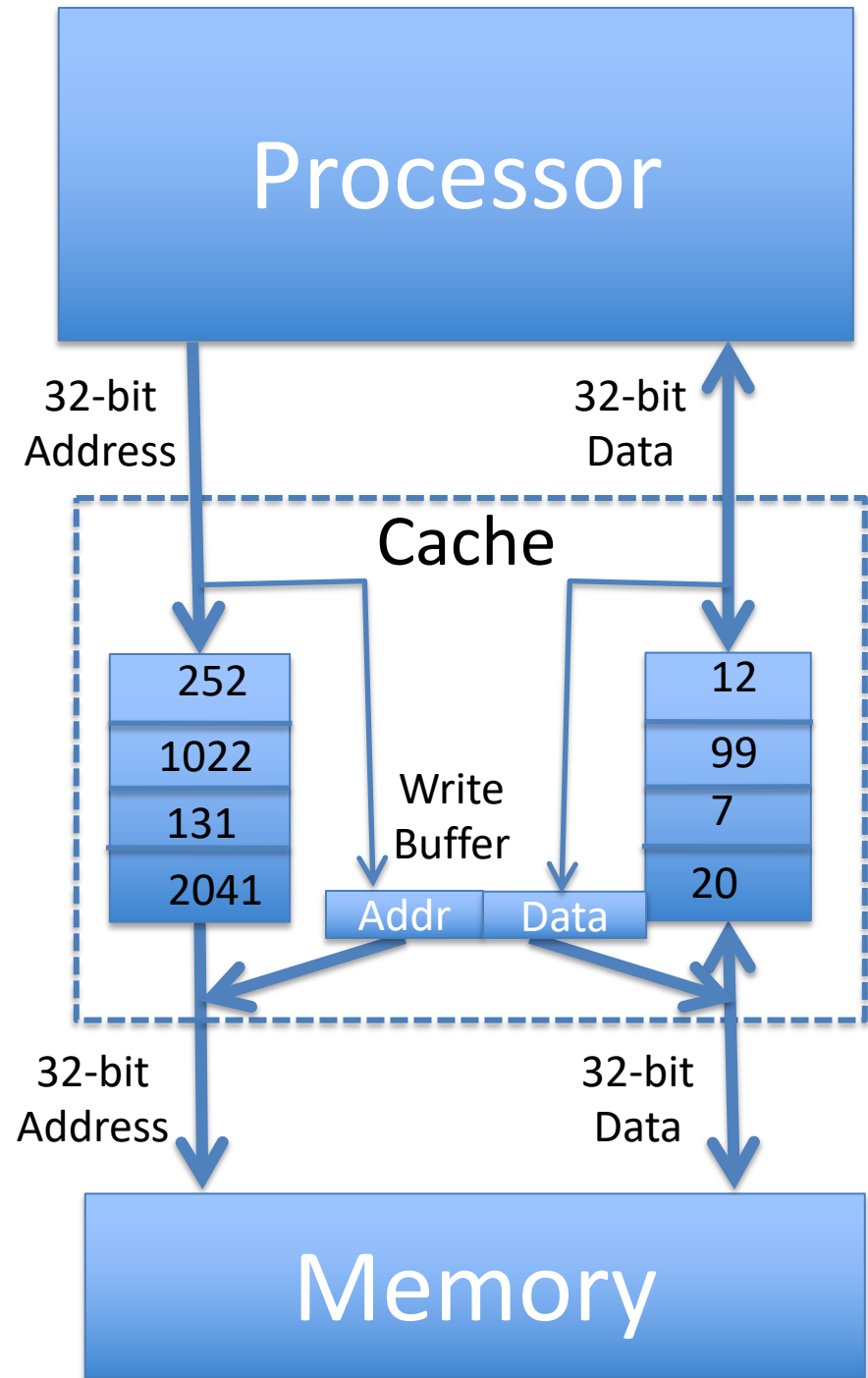| Tag	Index	Byte Offset

address_size = tag_size + index_size + offset_size
= tag_size + log2(S) + log2(B)

Handling Stores with Write-Through

- Store instructions write to memory, changing values

- Need to make sure cache and memory have same values on writes: 2 policies

1) Write-Through Policy: write cache and write *through* the cache to memory
   - Every write eventually gets to memory
   - Too slow, so include Write Buffer to allow processor to continue once data in Buffer
   - Buffer updates memory in parallel to processor

Write-Through Cache

- Write both values in cache and in memory

- Write buffer stops CPU from stalling if memory cannot keep up

- Write buffer may have multiple entries to absorb bursts of writes

- What if store misses in cache?

Processor

32-bit Address

32-bit Data

Cache

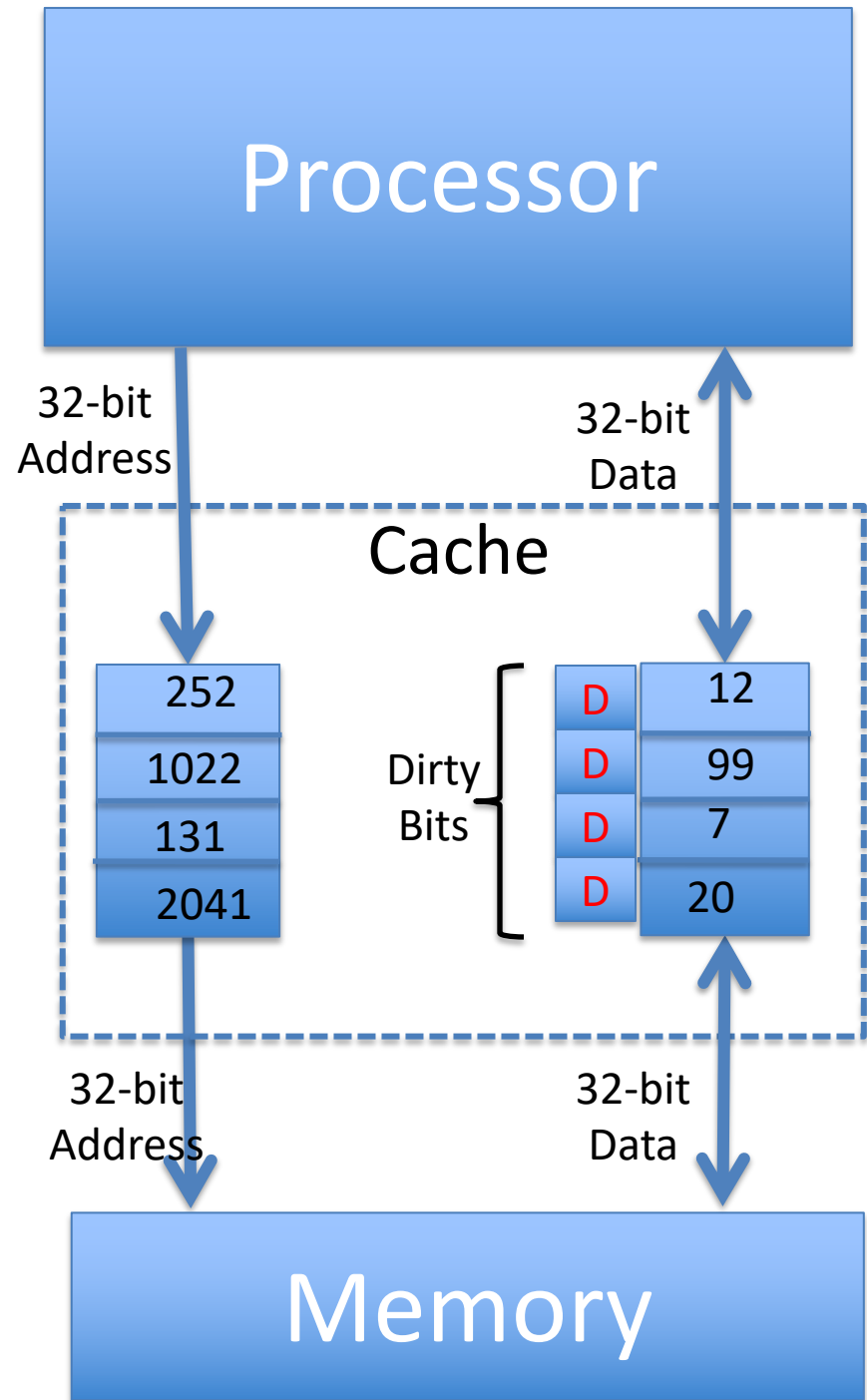| 252 | | 12 |
1022	Write Buffer	99
131		7
2041	Addr Data	20

32-bit Address

32-bit Data

Memory

Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache

– Writes collected in cache, only single write to memory per block

– Include bit to see if wrote to block or not, and then only write back if bit is set

• Called "Dirty" bit (writing makes it "dirty")

Write-Back Cache

- Store/cache hit, write data in cache *only* & set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - "Write-allocate" policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit.

Processor

32-bit Address

32-bit Data

Cache

252		D	12
1022	Dirty Bits	D	99
131		D	7
2041		D	20

32-bit Address

32-bit Data

Memory

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

Cache (*Performance)* Terms

- Hit rate: fraction of accesses that hit in the cache

- Miss rate: 1 − Hit rate

- Miss penalty: time to replace a line/ block from lower level in memory hierarchy to cache

- Hit time: time to access cache memory (including tag comparison)

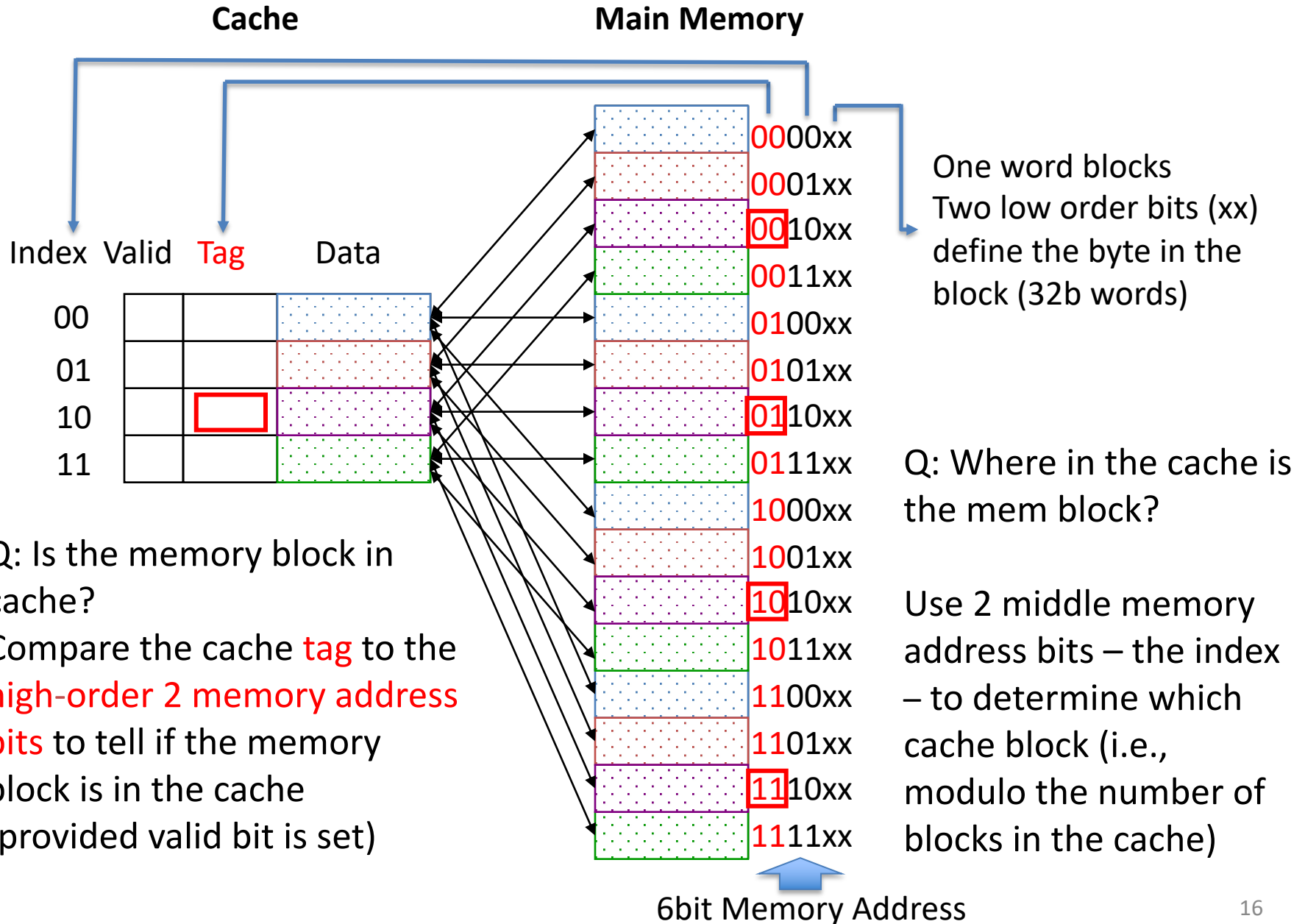- Abbreviation: "$" = cache (cash …)

Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses in the cache

AMAT =   Time for a hit

+ Miss rate × Miss penalty

Direct Mapped Cache Example

Cache   **Main Memory**

Index Valid Tag   Data

00			
01			
10			
11			

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

One word blocks
Two low order bits (xx)
define the byte in the
block (32b words)

Q: Where in the cache is
the mem block?

Use 2 middle memory
address bits – the index
– to determine which
cache block (i.e.,
modulo the number of
blocks in the cache)

Q: Is the memory block in
cache?
Compare the cache tag to the
high-order 2 memory address
bits to tell if the memory
block is in the cache
(provided valid bit is set)

6bit Memory Address
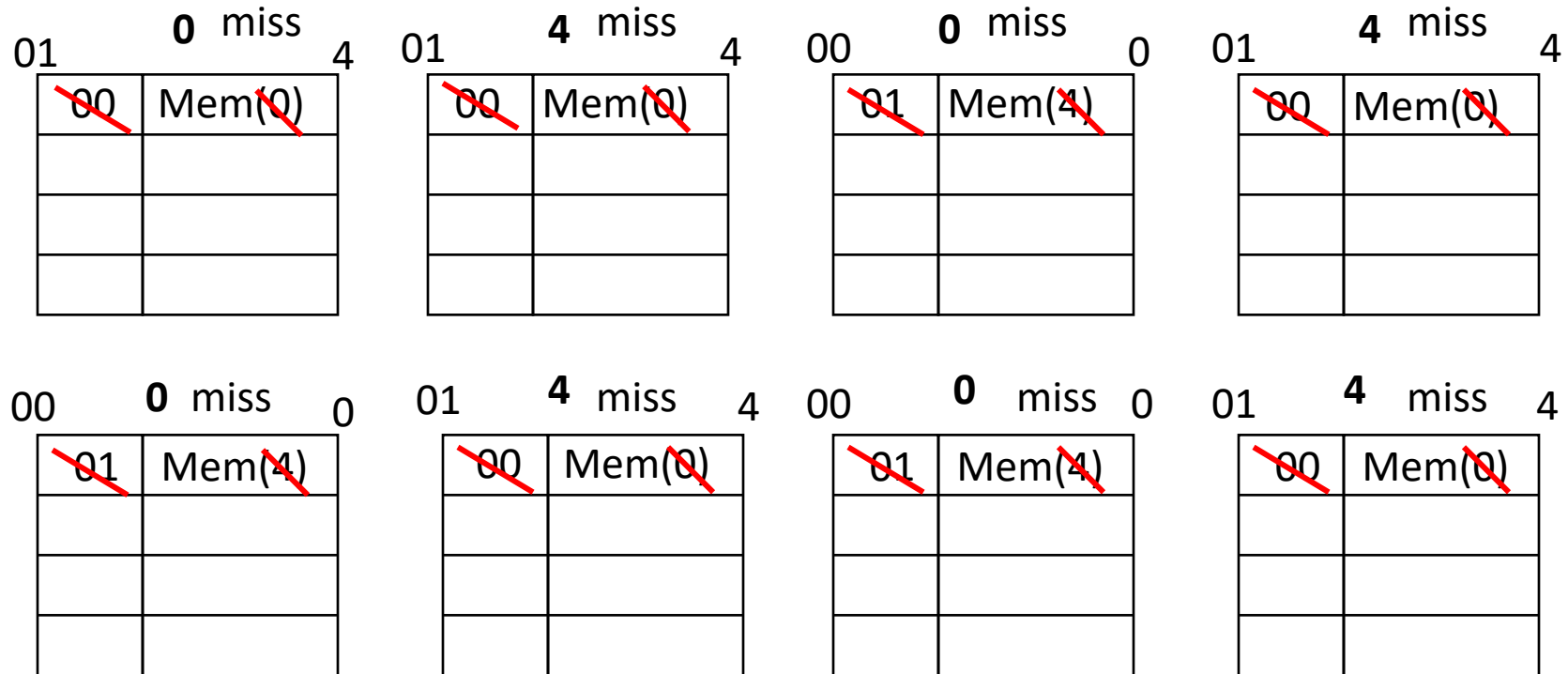
16

Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

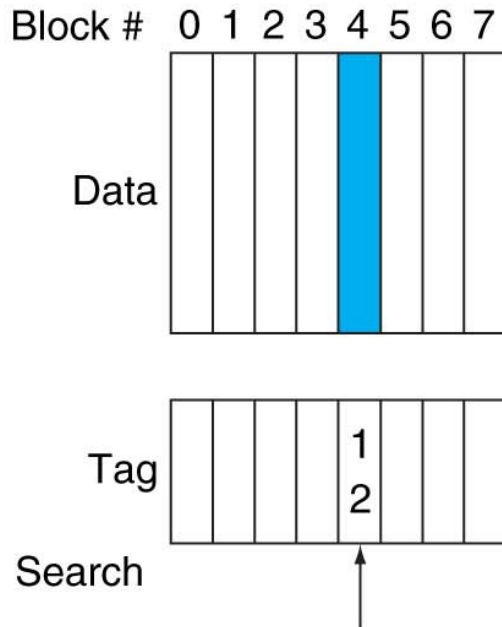- Consider the main memory address (words) reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks
initially marked as not valid

01   **0** miss   4   01   **4** miss   4   00   **0** miss   0   01   **4** miss   4

00	Mem(0)

00	Mem(0)

01	Mem(4)

00	Mem(0)

00   **0** miss   0   01   **4** miss   4   00   **0** miss   0   01   **4** miss   4

01	Mem(4)

00	Mem(0)

01	Mem(4)

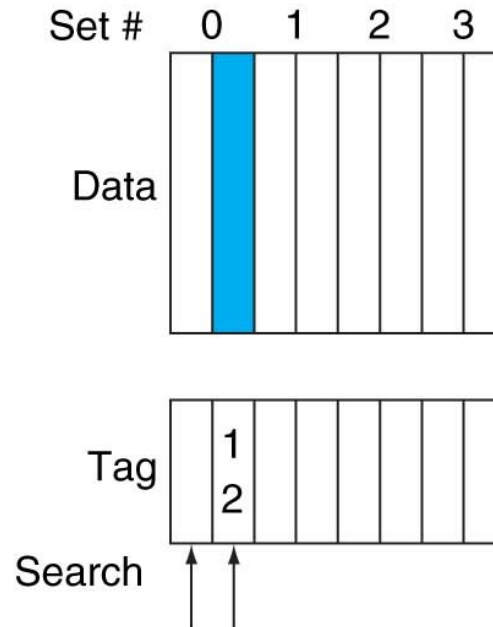| 00	Mem(0)

- 8 requests, 8 misses
- Ping-pong effect due to conflict misses - two memory locations that map into the same cache block
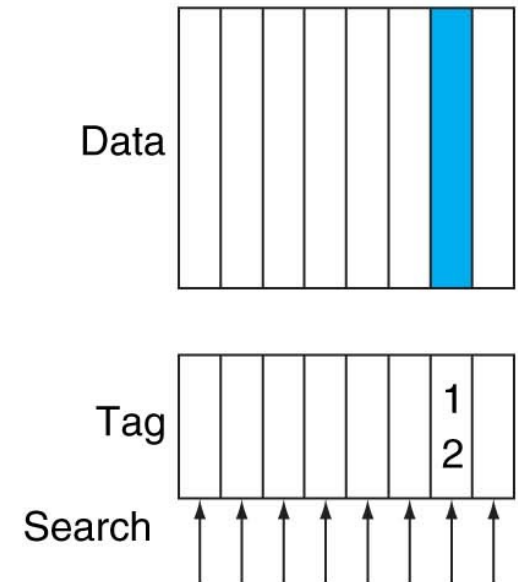
Alternative Block Placement Schemes
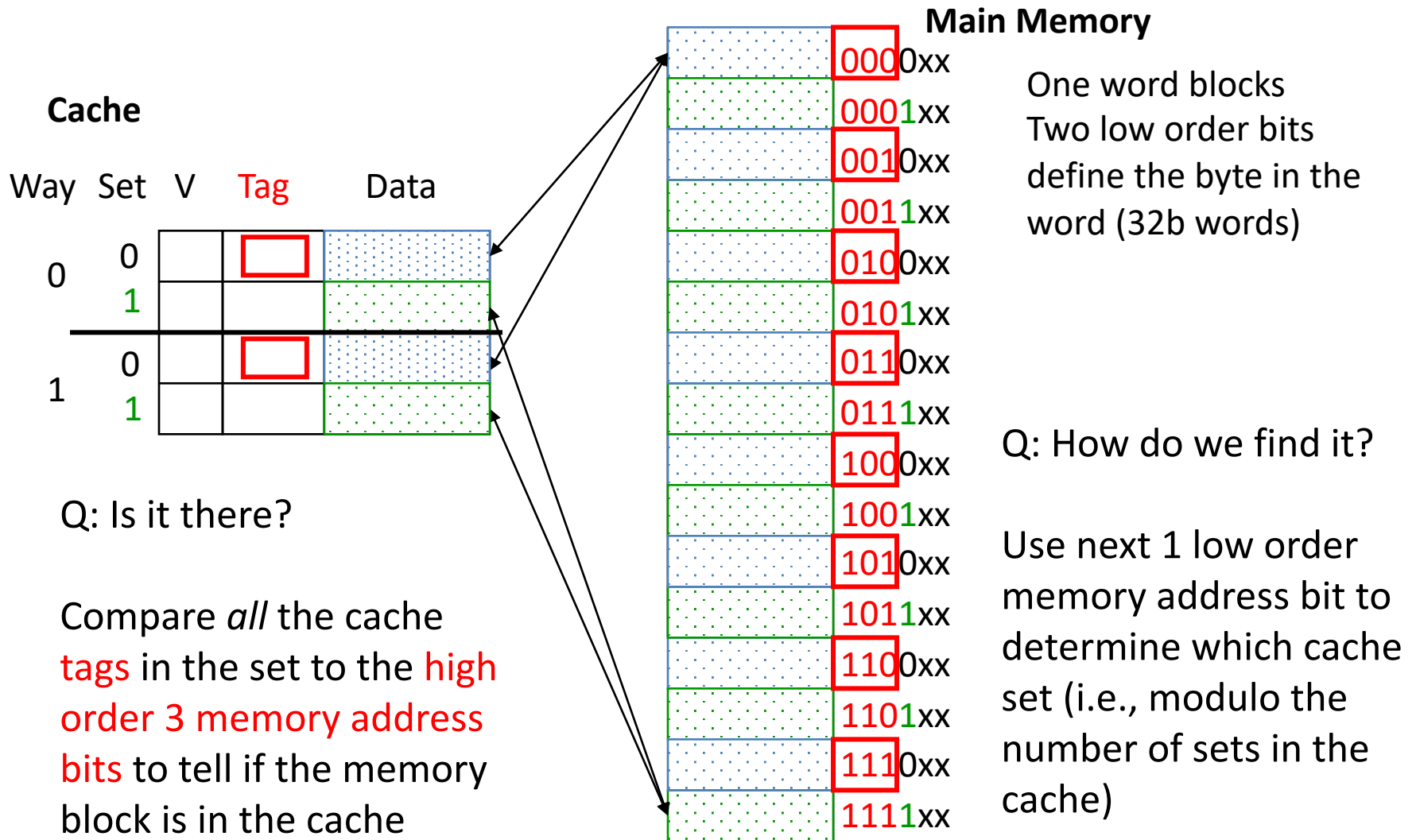
Direct mapped | **Set associative** | **Fully associative**

Block # 0 1 2 3 4 5 6 7   Set #   0   1   2   3

Data

Tag   1 2

Search

- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
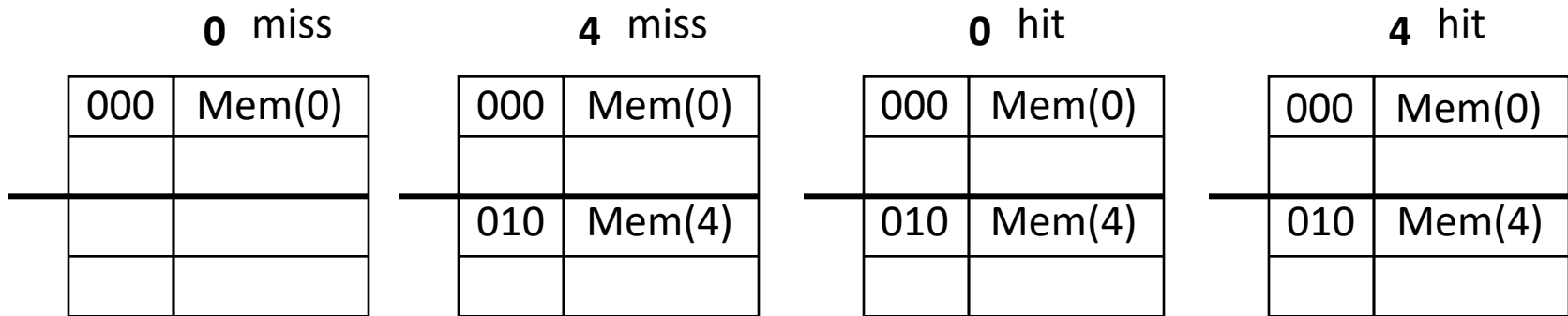- FA placement: mem block 12 can appear in any cache blocks

Example: 2-Way Set Associative $
(4 words = 2 sets x 2 ways per set)

Main Memory

Cache

Way	Set	V	Tag	Data
0	0		▭	
	1			
1	0		▭	
	1			

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

One word blocks
Two low order bits
define the byte in the
word (32b words)

Q: How do we find it?

Use next 1 low order
memory address bit to
determine which cache
set (i.e., modulo the
number of sets in the
cache)

Q: Is it there?

Compare *all* the cache
tags in the set to the high
order 3 memory address
bits to tell if the memory
block is in the cache

Example: 4-Word 2-Way SA $ Same Reference String

- Consider the main memory address (word) 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!

Four-Way Set-Associative Cache

- 2^8 = 256 sets each with four ways (each with one block)
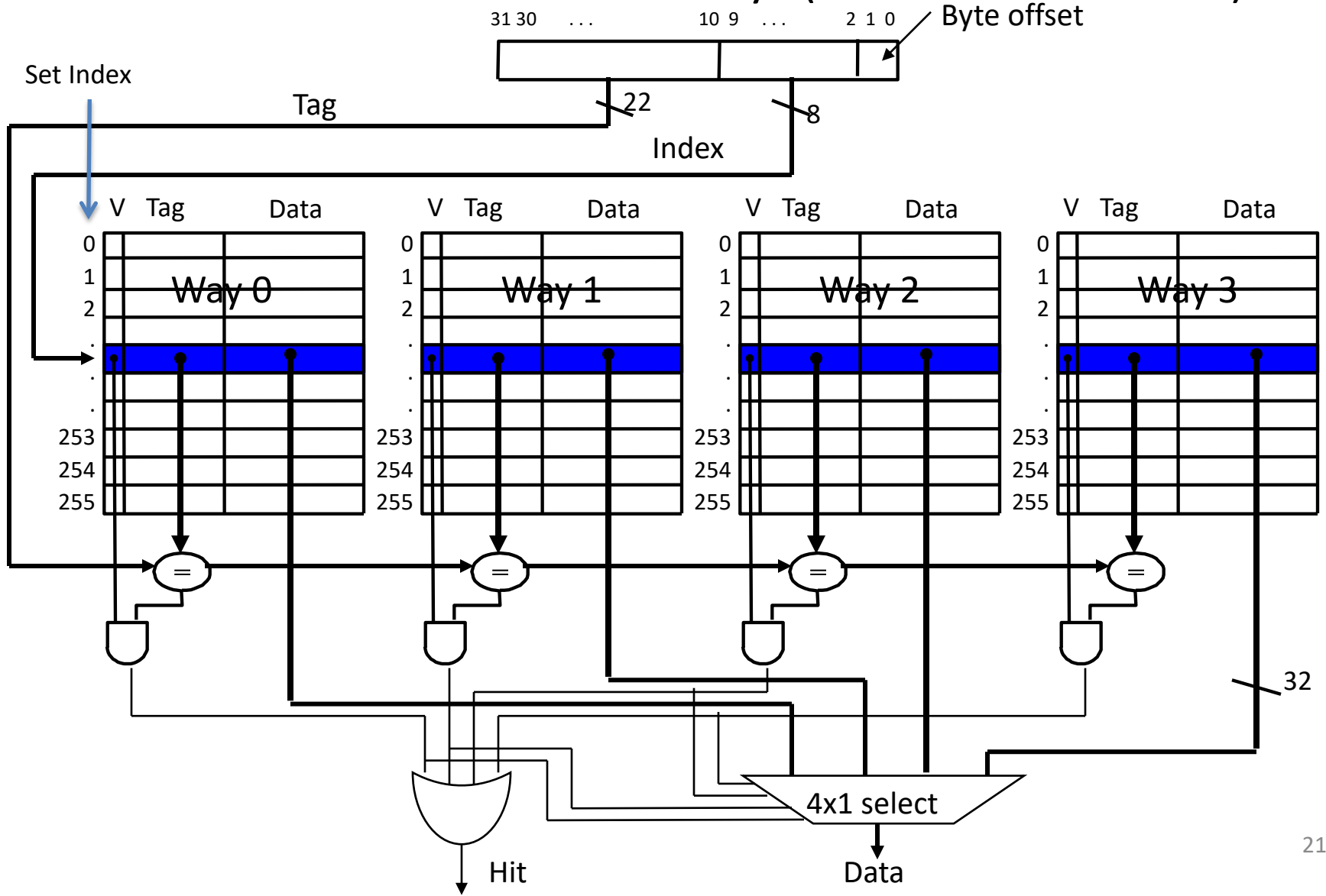
Different Organizations of an Eight-Block Cache

**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				

Total size of $ in blocks is equal to *number of sets × associativity*. For fixed $ size and fixed block 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 $.

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

Range of Set-Associative Caches

- For a *fixed-size* cache and fixed block size, 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

Used for tag compare   Selects the set   Selects the word in the block

Tag	Index	Word offset	Byte offset

Increasing associativity

Decreasing associativity

Direct mapped
(only one way)
Smaller tags, only a
single comparator

Fully associative
(only one set)
Tag is all the bits except
block and byte offset
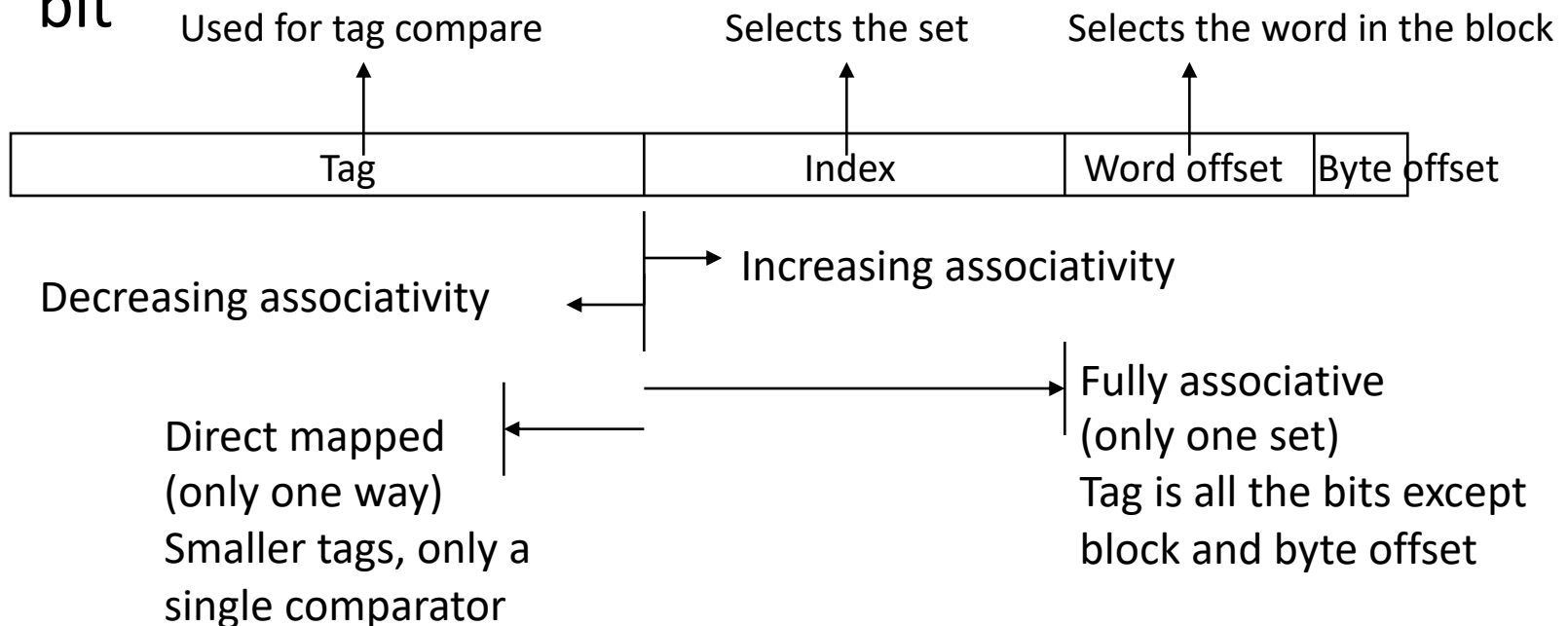
Costs of Set-Associative Caches

- 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")

Cache Replacement Policies

- Random Replacement
  - Hardware randomly selects a cache evict
- Least-Recently Used
  - Hardware keeps track of access history
  - Replace the entry that has not been used for the longest time
  - For 2-way set-associative cache, need one bit for LRU replacement
- Example of a Simple "Pseudo" LRU Implementation
  - Assume 64 Fully Associative entries
  - Hardware replacement pointer points to one cache entry
  - Whenever access is made to the entry the pointer points to:
    - Move the pointer to the next entry
  - Otherwise: do not move the pointer
  - (example of "not-most-recently used" replacement policy)

Replacement	**Entry 0**
	Entry 1
Pointer →	**:**
	Entry 63
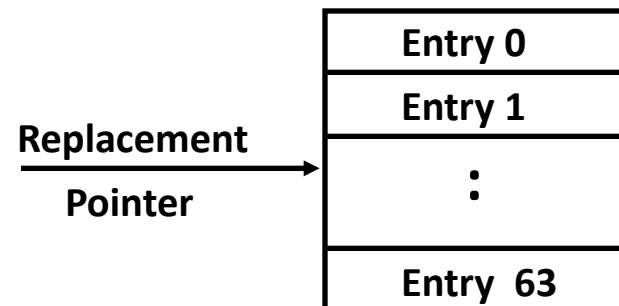
Benefits of Set-Associative Caches

- Choice of DM $ versus SA $ depends on the cost of a miss versus the cost of implementation
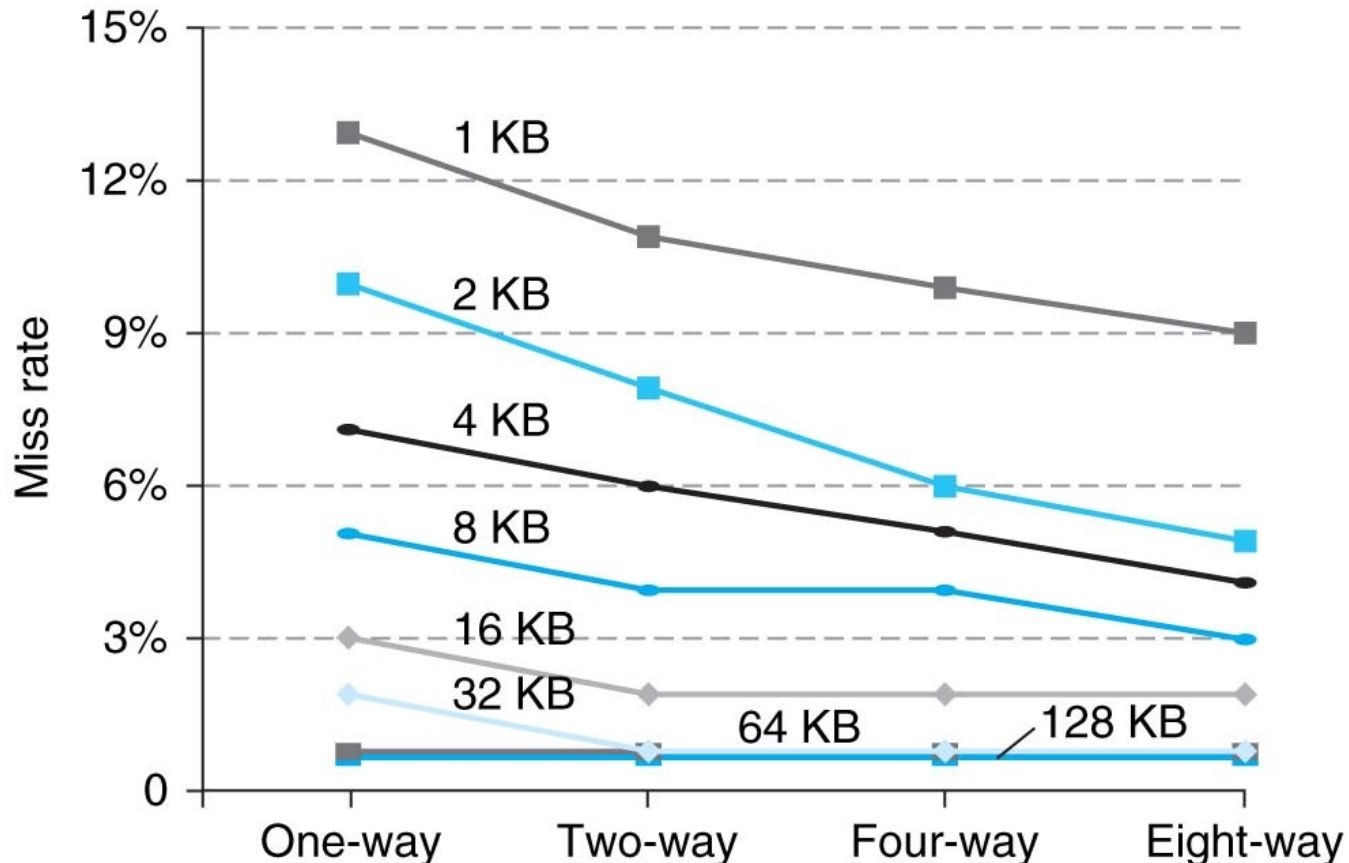
- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st 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)*

Prefetching...

- Programmer/Compiler: I know that, later on, I will need this data...

- Tell the computer to **_prefetch_** the data
  - Can be as an explicit prefetch instruction
  - Or an implicit instruction: **lw x0 0(t0)**
    - Won't stall the pipeline on a cache miss: The processor control logic recognizes this situation

- Allows you to hide the cost of compulsory misses
  - You still need to fetch the data however

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

And In Conclusion, …

- Principle of Locality for Computer Memory
- 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
  - FA, DM, and SA
  - 3C