

# CSC 112: Computer Operating Systems

## Lecture 7

### Memory System I: Cache

Department of Computer Science,  
Hofstra University

# Outline

- Cache Introduction
- Cache Organization
- Cache Performance Analysis

# Why are Large Memories Slow?

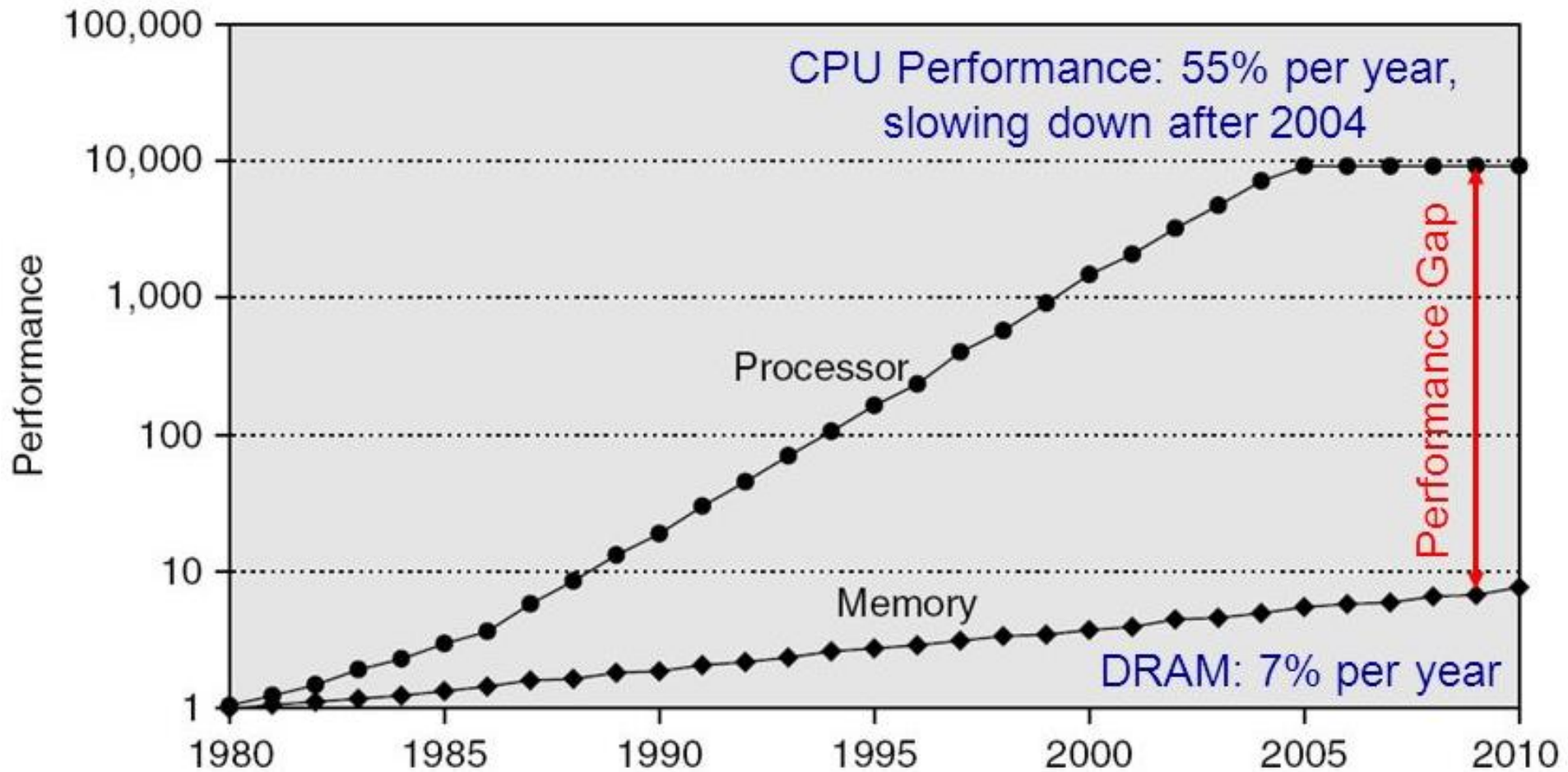
## Library Analogy

- Time to find a book in a large library
  - Search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book
- Both delays become larger for larger libraries
- Computer memories have same issue, *plus* the technologies used to store a bit slow down as density increases (e.g., SRAM vs. DRAM vs. Disk)



***However, what we want is a large yet fast memory!***

# Processor-DRAM Gap (Latency)



1980 microprocessor executes **~one instruction** in same time as DRAM access

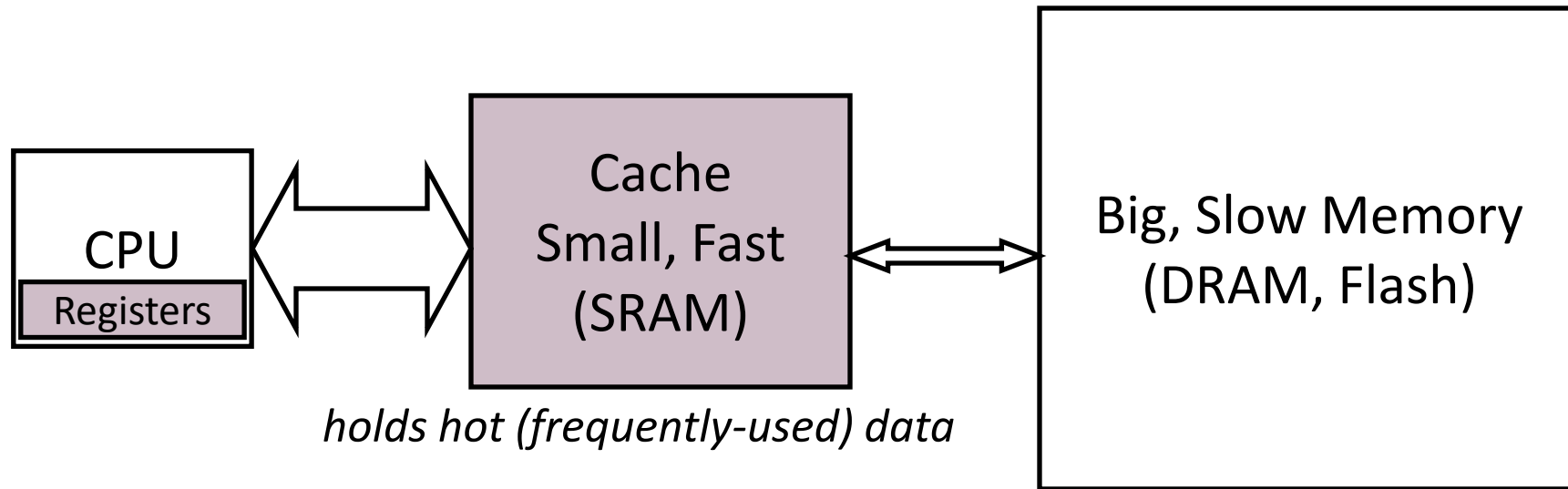
2017 microprocessor executes **~1000 instructions** in same time as DRAM access

***Memory wall: memory access likely to be the performance bottleneck***

# What To Do: Library Analogy

- Write a report using library books
- Go to library (main memory), look up relevant books, fetch from stacks, and place on your desk (cache)
- If need more, check them out and keep them on your desk
  - But don't return earlier books since might need them
- You hope this collection of a few books on your desk enough to write report, even though they are a tiny fraction of all books in the library

# Memory Hierarchy



- *Capacity*: register  $\ll$  cache (typically on-chip)  $\ll$  memory (off-chip)
- *Latency*: register  $\ll$  cache (typically on-chip)  $\ll$  memory (off-chip)

On a data access:

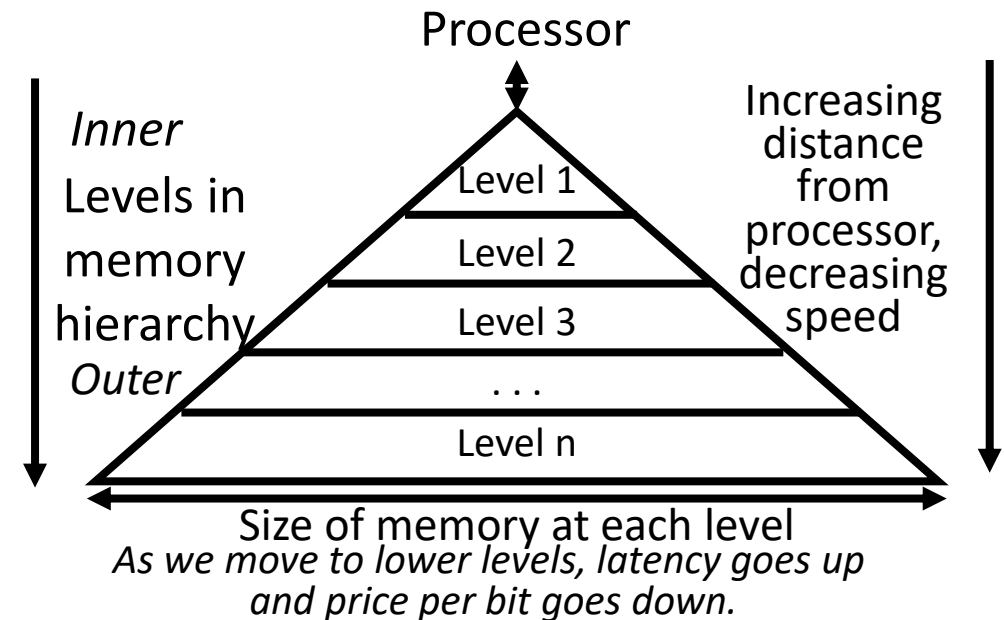
*if data  $\in$  cache  $\Rightarrow$  cache hit  $\Rightarrow$  low latency access (SRAM)*

*if data  $\notin$  cache  $\Rightarrow$  cache miss  $\Rightarrow$  high latency access (DRAM, Flash)*

Goal: create the illusion of accessing as much memory as is available in the slow memory at the speed of the fast cache

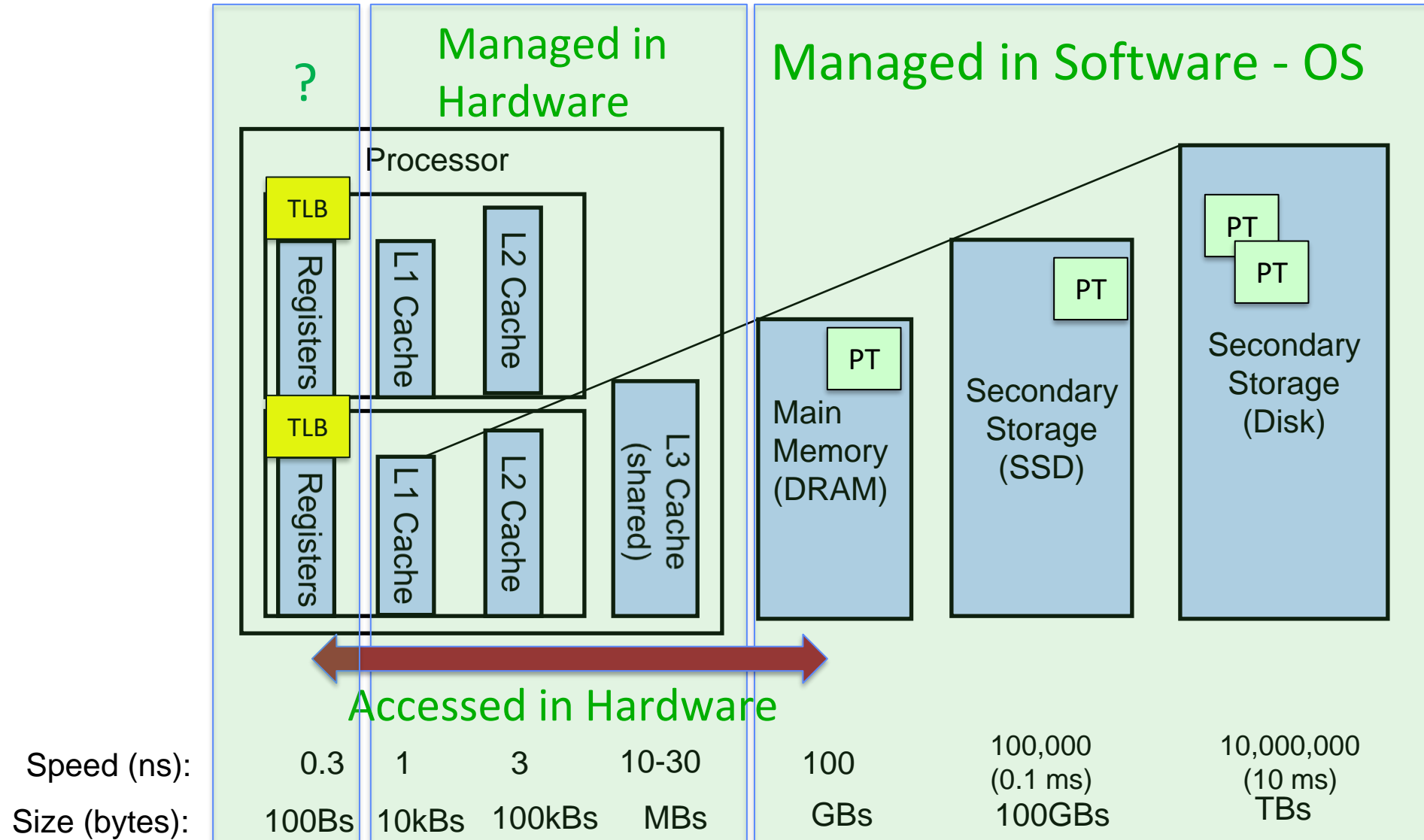
# Memory Hierarchy Technologies

- Caches use SRAM (Static Random Access Memory) for speed and technology compatibility
  - Fast (typical access times of 0.5 to 2.5 ns)
  - Low density (6 transistor cells), higher power, expensive
  - Static: content will last as long as power is on
- Main memory uses DRAM (Dynamic RAM) for size and density
  - Slower (typical access times of 50 to 70 ns)
  - High density (1 transistor cells), lower power, cheaper
  - Dynamic: needs to be “refreshed” regularly (every ~8 ms)
    - Consumes 1% to 2% of the active cycles of the DRAM



# The Complete Memory Hierarchy

TLB: stores mappings of virtual addresses to physical addresses  
PT: Page Table





# How is the Hierarchy Managed?

- Registers  $\leftrightarrow$  memory hierarchy
  - By compiler (or assembly programmer)
- Cache  $\leftrightarrow$  main memory
  - By the cache controller hardware
  - Focus of this lecture
- Main memory  $\leftrightarrow$  disks (secondary storage)
  - By the operating system (virtual memory)
  - By the programmer (files)

# Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time (spatial locality) and repeatedly access that portion (temporal locality)
- *Temporal Locality* (locality in time)
  - Go back to same book on desk multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon
    - Keep recently-accessed blocks in the cache
- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books around the book you want, since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
    - When fetching a block into cache, also fetch blocks around it
- If the program has poor temporal or spatial locality, then lots of useless junk may be brought into cache

# What locality does this program have?

```
int sum = 0, a[n];  
...  
for (i = 0; i < n; i++) {  
    sum += a[i];  
}  
return sum;
```

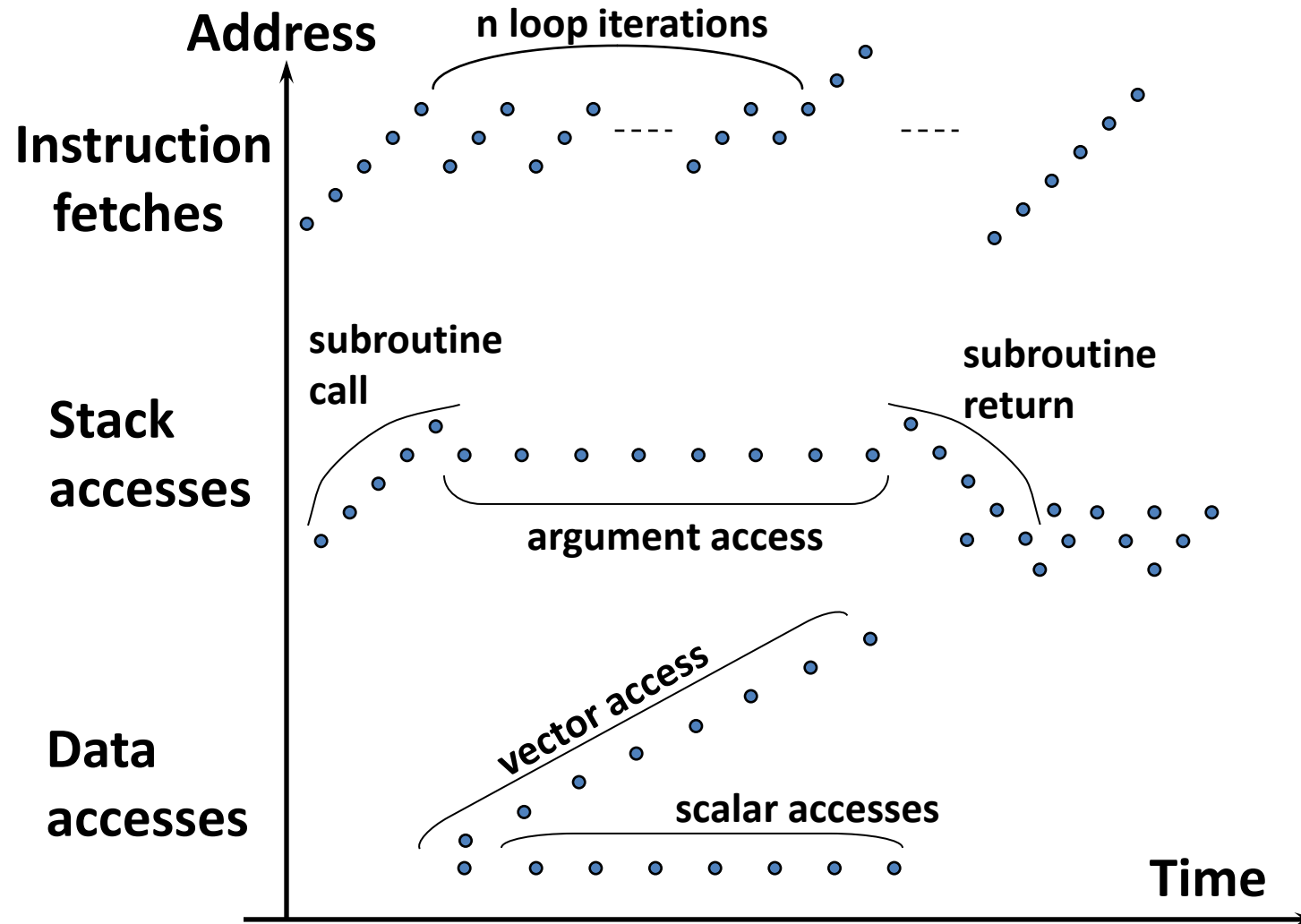
- Data:
  - Temporal locality: variable **sum** is referenced in every iteration
  - Spatial locality: array **a[]** is accessed with stride 1 in each iteration (assuming a[] is stored in contiguous addresses in memory)
- Instructions:
  - Temporal locality: the loop body is executed repeatedly for n times
  - Spatial locality: instructions are accessed sequentially (with 1 branch in each iteration) (assuming instructions are stored in contiguous addresses in memory)

# Memory Reference Patterns



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory.  
IBM Systems Journal 10(3): 168-192 (1971)

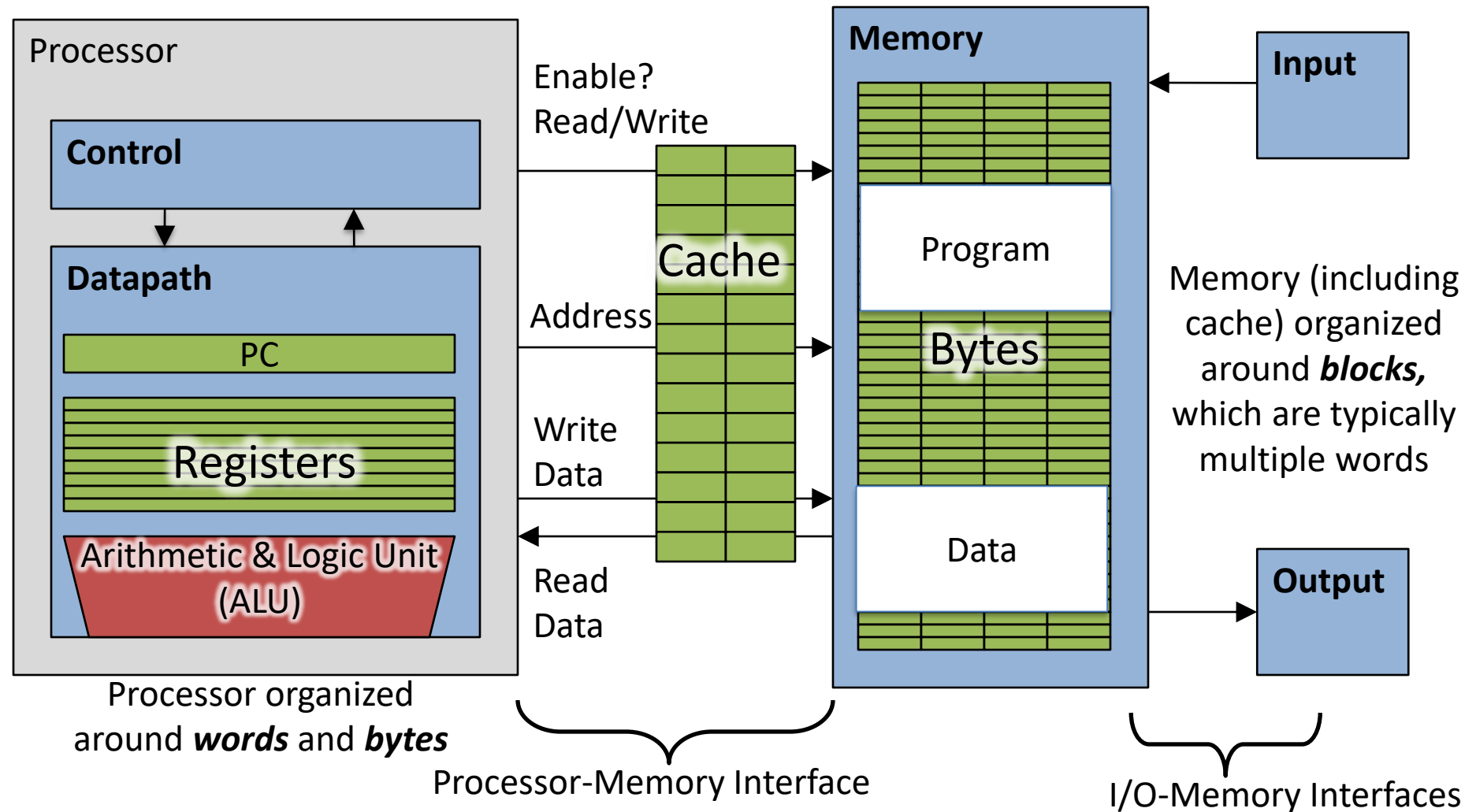
# Memory Reference Patterns



# Outline

- Cache Introduction
- Cache Organization
- Cache Performance Analysis

# Processor with Cache

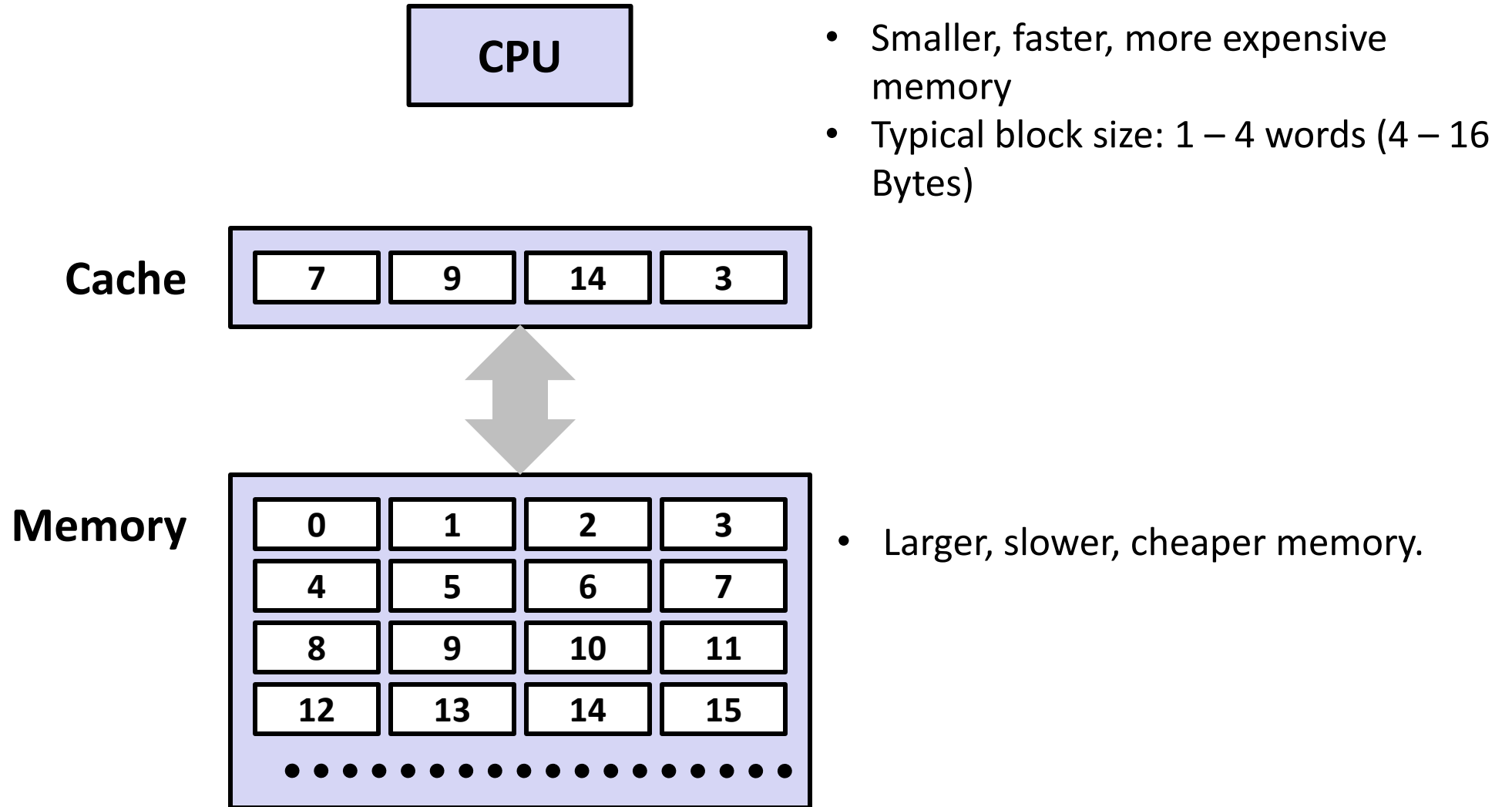


# Cache vs. Memory

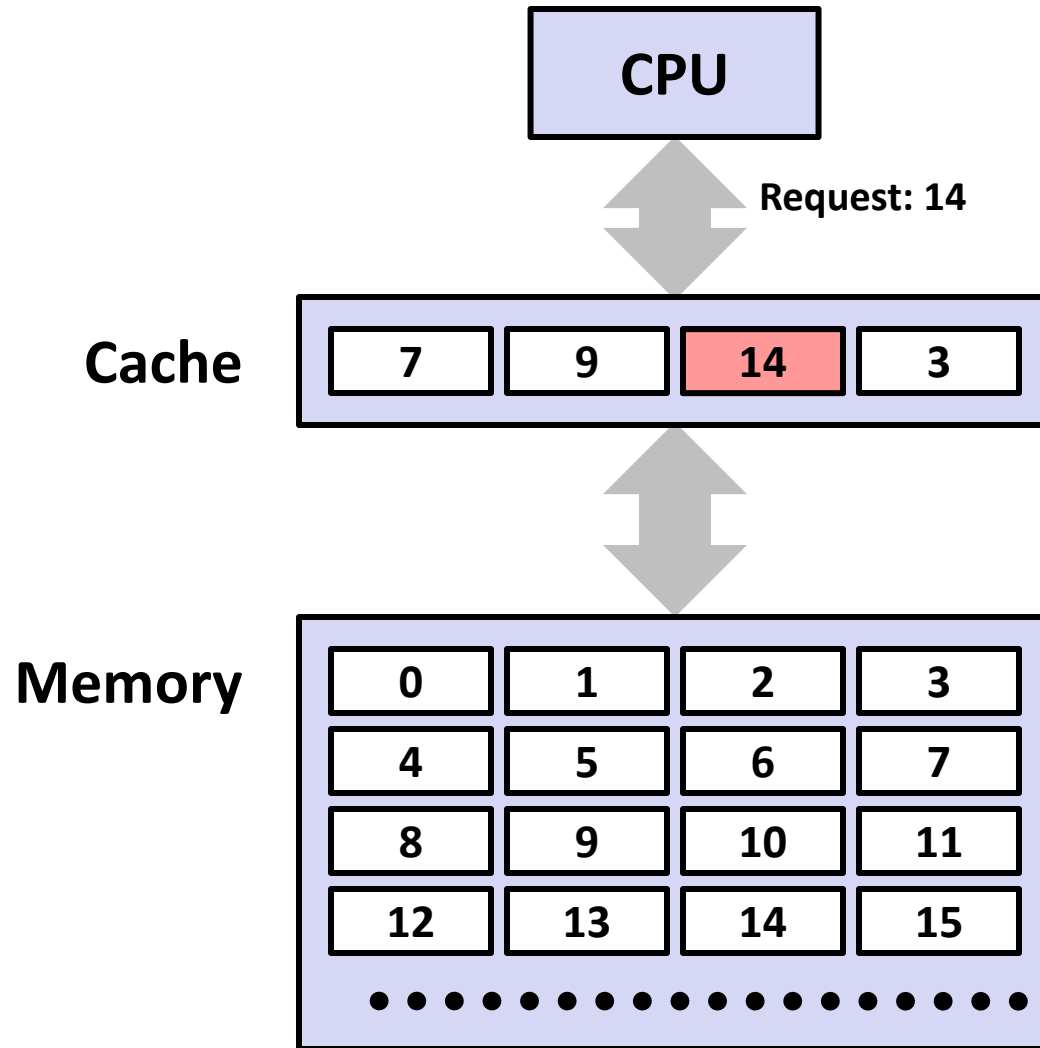
- Cache size  $\ll$  memory size
  - Smaller cache is faster
- 1-to-many correspondence between cache blocks and memory blocks
  - Use *Tags* in the cache to match cache and memory blocks
- A **cache block** is also called a **cache line**
- Blocks are aligned in memory:
  - if each cache block is 4 Bytes (1 word), then binary address of each cache block always ends in 00
  - If each cache block is 8 Bytes (2 words), then binary address each cache block always ends in 000



# Cache Blocks



# General Cache Concepts: Hit



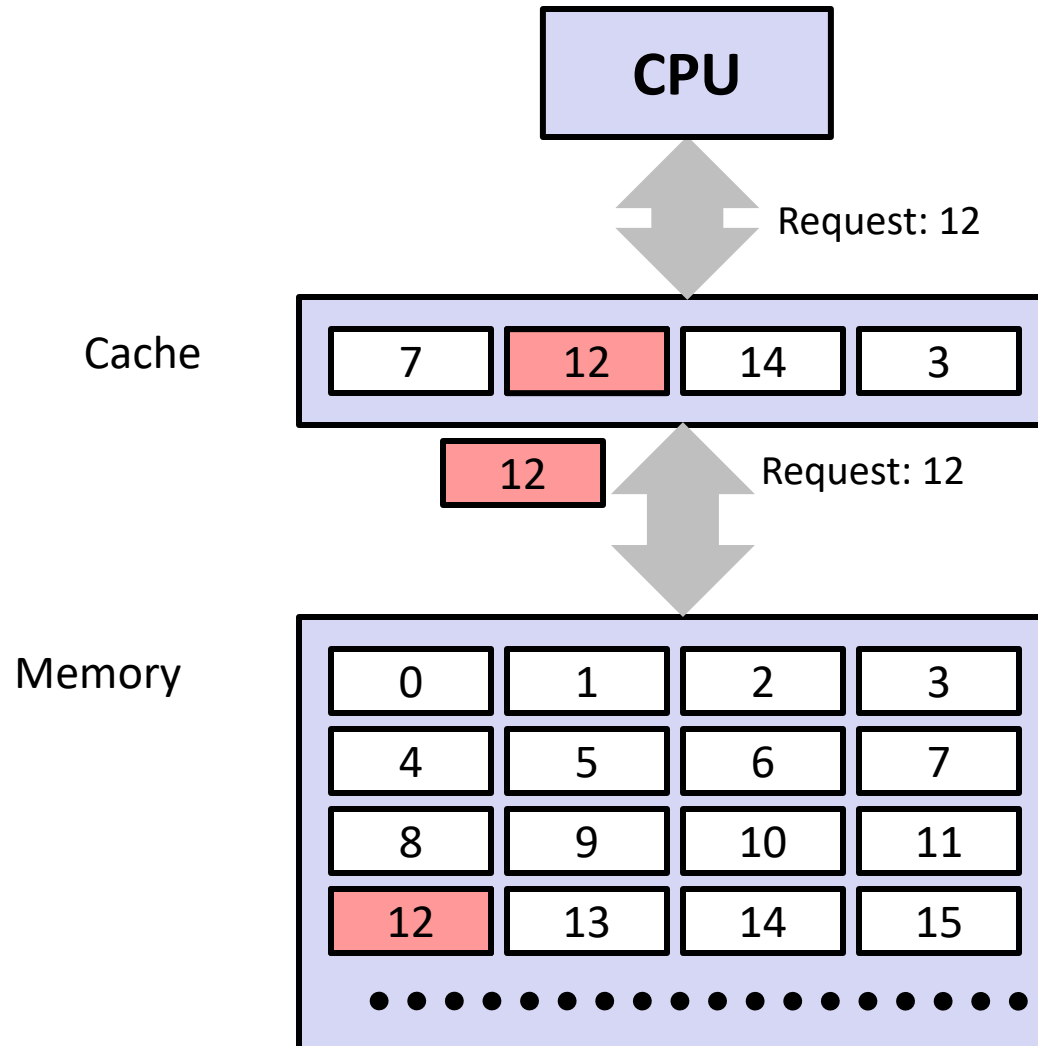
*Data in block 14 is needed*

*Block 14 is in cache:*

*Hit!*

*Data is loaded from cache into CPU register*

# General Cache Concepts: Miss



*Data in block 12 is needed*

*Block 12 is not in cache:*  
**Miss!**

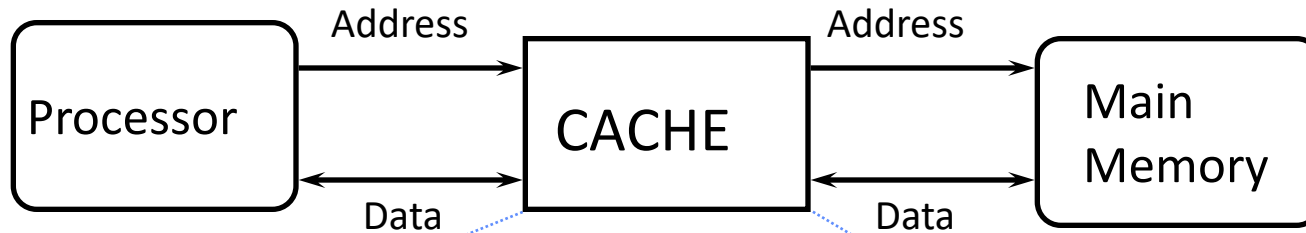
*Block 12 is fetched from  
memory*

*Block 12 is stored in cache*

- **Placement policy:**  
determines where the new block goes
- **Replacement policy:**  
determines which old block  
gets evicted (victim)

*Data is loaded from cache into CPU register*

# Inside a Cache



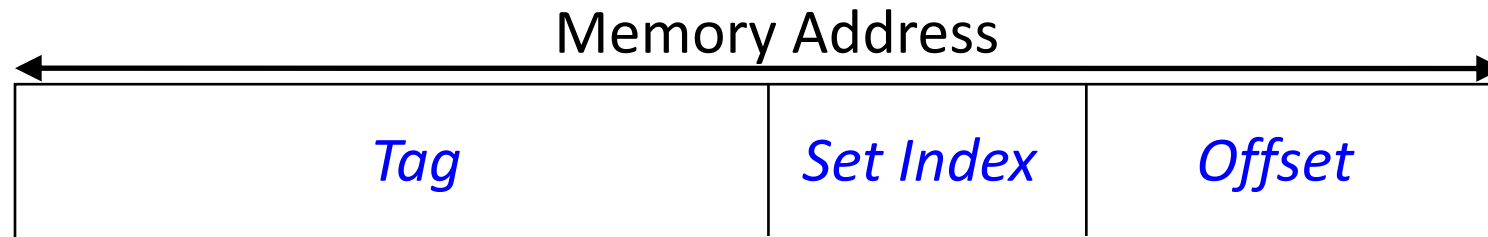
Valid	Tag	Data Byte					
0	100					-----	
1	304					-----	
0	6848						
1	416						
1							
1						-----	

A Cache Block

- A “**valid bit**” indicates if a cache block contains valid data
  - e.g., upon startup, the cache is “cold”: all cache blocks are invalid
  - The cache is “warmed-up” gradually by bringing content into the cache
- A **tag** helps identify the memory block contained in the cache block
  - Disambiguate among multiple possible memory blocks that may be mapped to the same cache block
- **Cache capacity** refers to the total size of cache blocks (not including Tag and Valid bits)

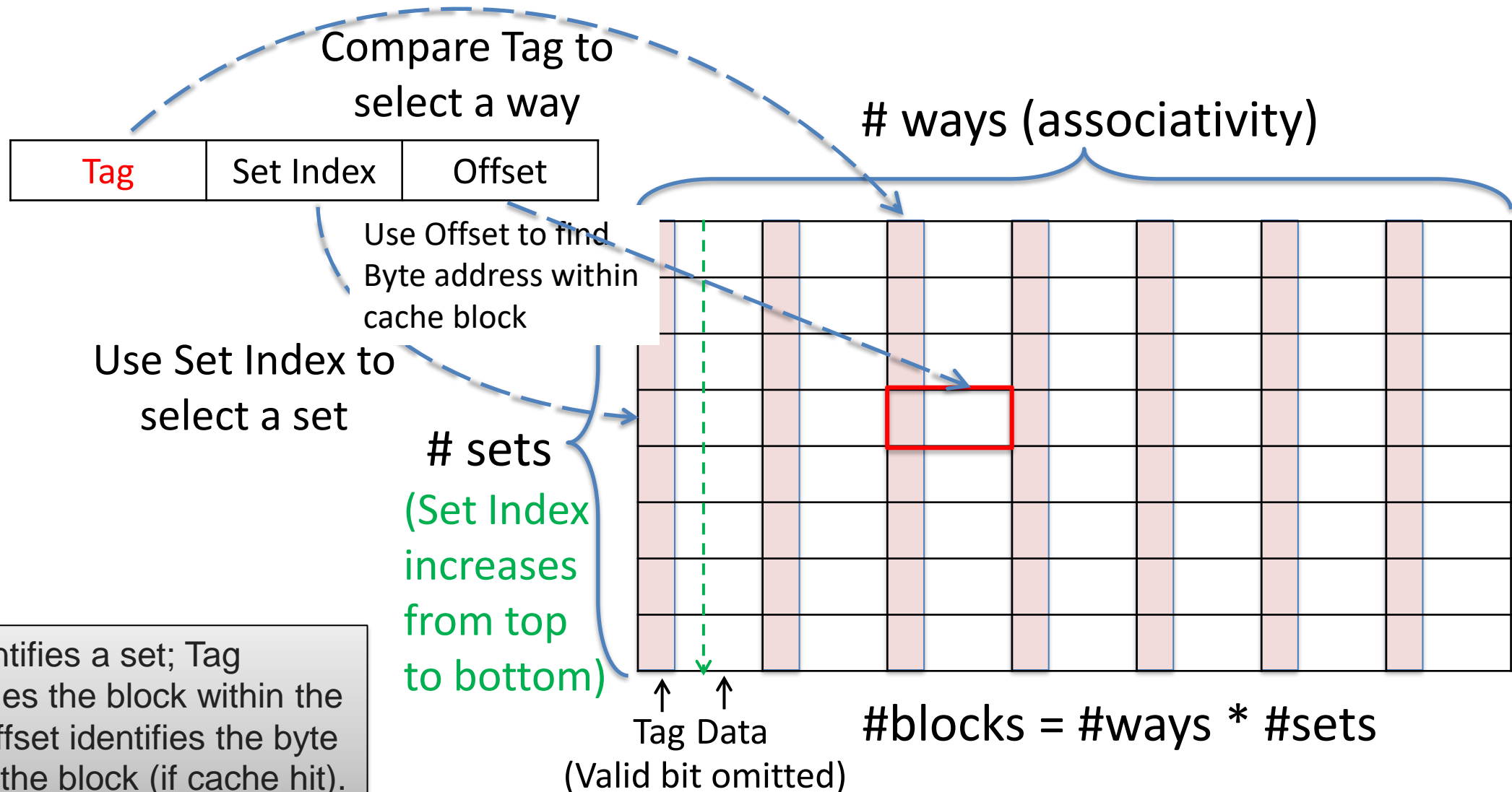
# Memory Address Fields

- **Offset**: Byte address within a cache block
- **Set Index**: Selects which set
- **Tag**: Remaining portion of processor address



- Size of Set Index (SI) =  $\log_2(\text{number of sets})$
- Size of Offset =  $\log_2(\text{number of bytes/block})$
- Size of Tag = Address size – Size of SI - Size of Offset

# Cache Organization



# Sources of Cache Misses (3 C's)

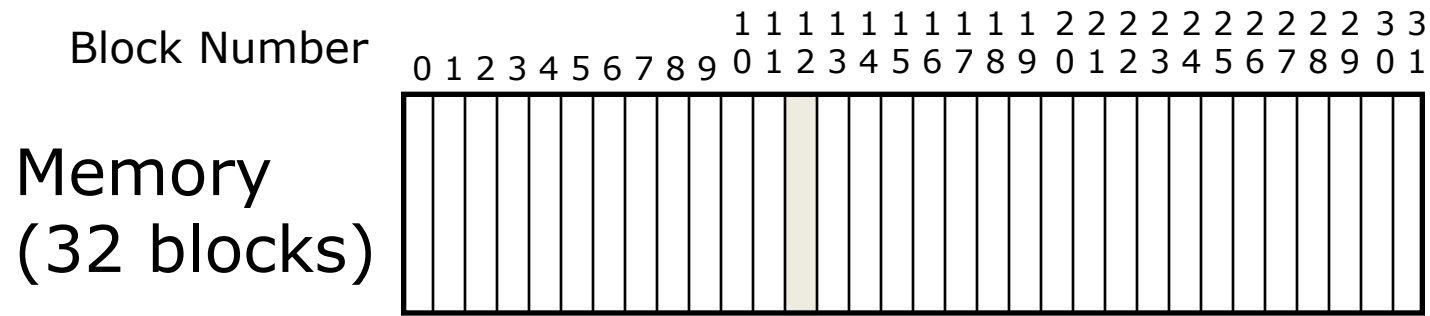
- *Compulsory*: cold start, first access to a block
  - Unavoidable misses that would occur even with infinite cache
  - Can be reduced by increasing block size
- *Capacity*: cache is too small to hold all data needed by the program
  - Misses that would occur even under perfect replacement policy
  - Can be reduced by increasing cache capacity
- *Conflict*: collisions due to multiple memory addresses mapped to same cache set
  - Can be reduced by increasing associativity and/or increasing cache capacity

# Alternative Cache Organizations

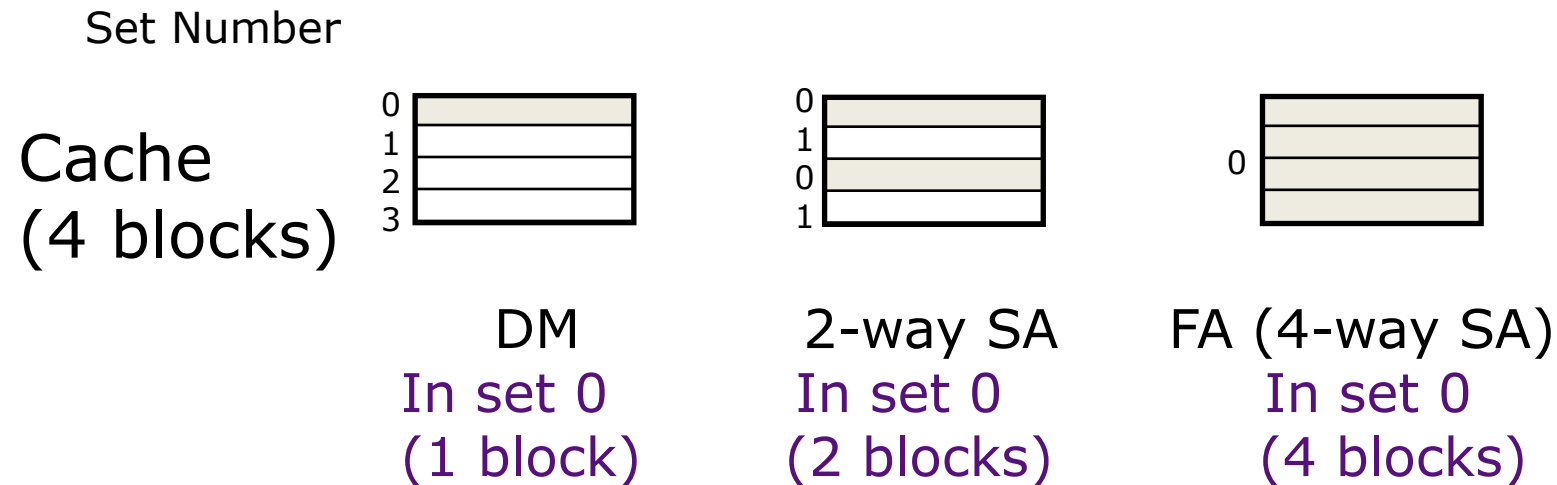
- A memory block is mapped to one **cache set**, which may contain one or more cache blocks
- **Direct Mapped (DM)**
  - Each cache set has 1 cache block; # cache sets = # cache blocks
  - A memory block is mapped to 1 possible cache block
- **N-way Set Associative (SA)**
  - Associativity =  $N \rightarrow$  Each cache set has  $N$  cache blocks; # cache sets = # cache blocks/ $N$
  - A memory block can be mapped to one of  $N$  possible cache blocks
- **Fully Associative (FA)**
  - A single cache set contains all cache blocks; # cache sets = 1
  - A memory block can be mapped to any cache block
- DM and FA are special cases of SA
  - DM = 1-way SA ( $N = 1$ )
  - FA =  $N$ -way SA ( $N =$  cache capacity (total # cache blocks))



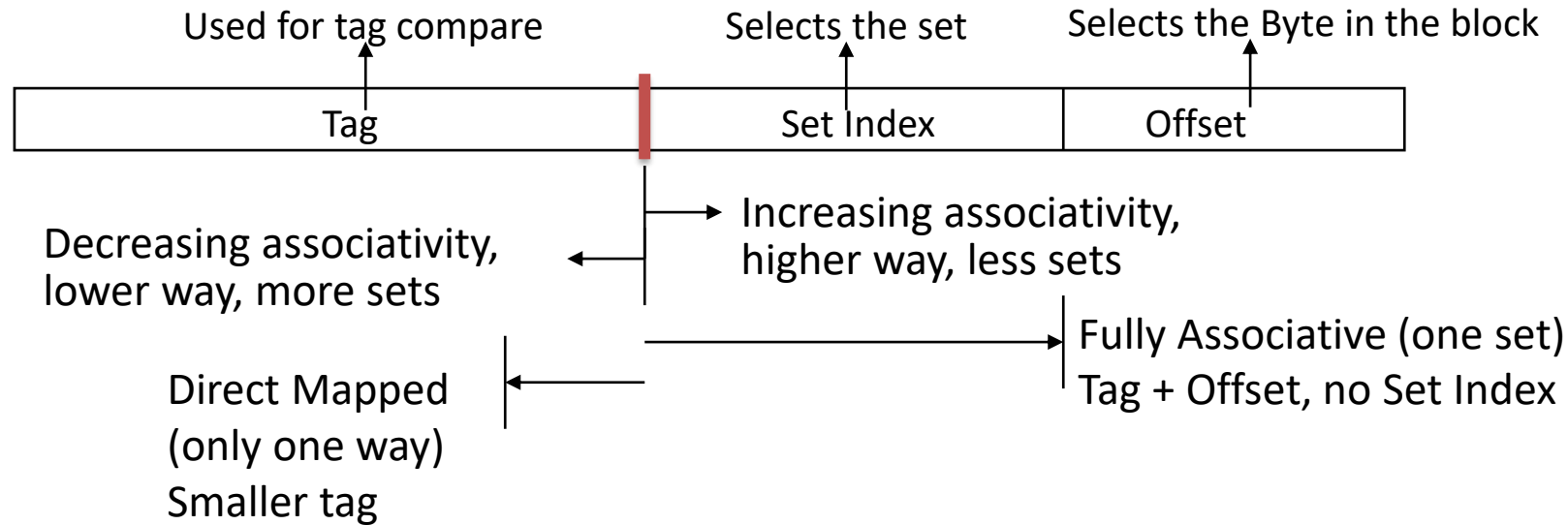
# Alternative Cache Organizations (4-block cache)



Where are possible locations in cache that block #12 in memory can be placed?



# Range of SA Cache Organizations



- $\text{memory\_address\_size} = \text{tag\_size (T)} + \text{set\_index\_size (SI)} + \text{block\_offset\_size (O)}$
- For fixed cache size, increasing associativity decreases number of sets while increasing number of blocks per set.
- If we decrease index by 1 bit and increase tag by 1 bit (pushing the red bar to the right by 1 bit) :
  - Doubled: #ways = #blocks per cache set = associativity
  - Halved: #cache sets

Way	Set	Valid	Tag	Data
0	0	0	01	
1	1	1	11	
2	1	1	10	
3	1	1	00	

**Direct Mapped**

5	4	3	2	1	0
Tag			Set Index		Offset

Way	Set	Valid	Tag	Data
0	0	0	010	
1	1	1	111	
0	1	1	101	
1	1	1	001	

**2-Way Set Associative**

5	4	3	2	1	0
Tag			Index	Offset	

Way	Set	Valid	Tag	Data
0	0	0	0101	
1	1	1	1110	
2	1	1	1010	
3	1	1	0011	

**Fully Associative**

5	4	3	2	1	0
Tag				Offset	

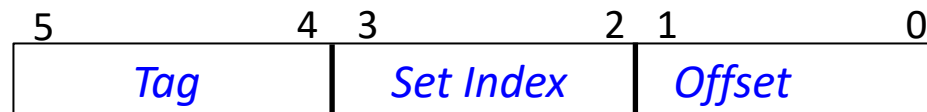
# 4-Block Cache (Valid Bit Omitted)

Higher associativity  $\rightarrow$  More ways  $\rightarrow$  fewer cache sets  $\rightarrow$  cache structure is more “short (vertically) and fat (horizontally)”

Lower associativity  $\rightarrow$  Fewer ways  $\rightarrow$  more cache sets  $\rightarrow$  cache structure is more “tall (vertically) and skinny (horizontally)”

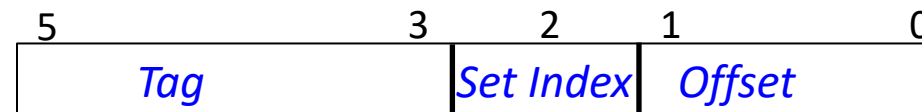
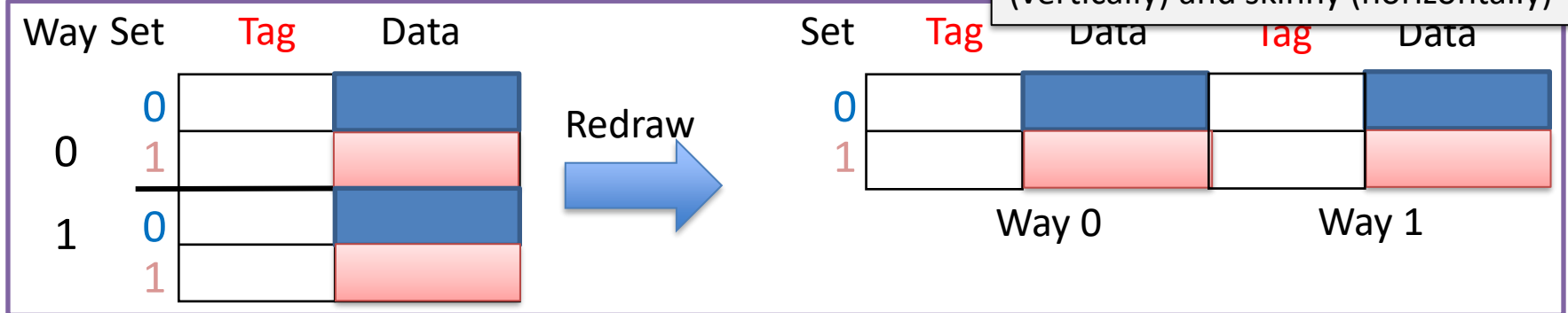
**Direct-Mapped**

Way	Set	Tag	Data
0	0		Blue
	1	Green	Green
	2	Red	Red
	3		Grey



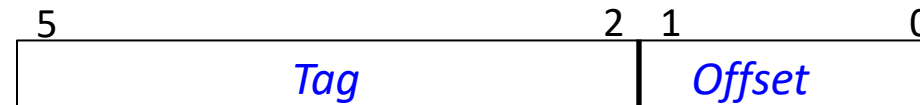
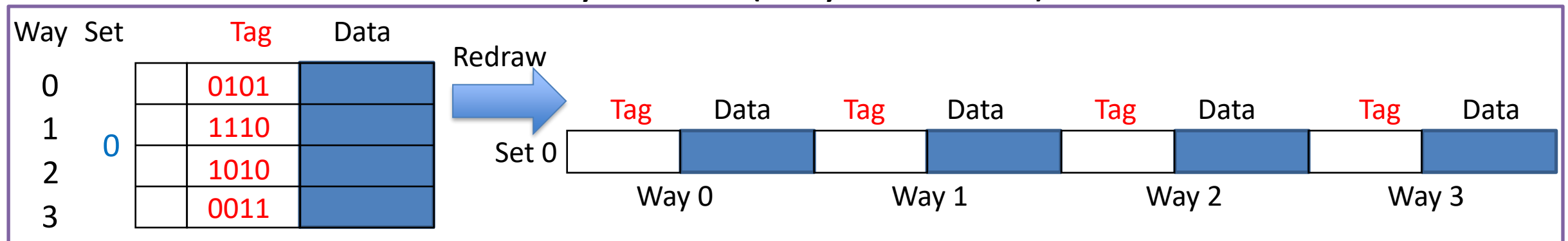
# cache blocks = 1 way \* 4 sets = 4

**2-way Set Associative**



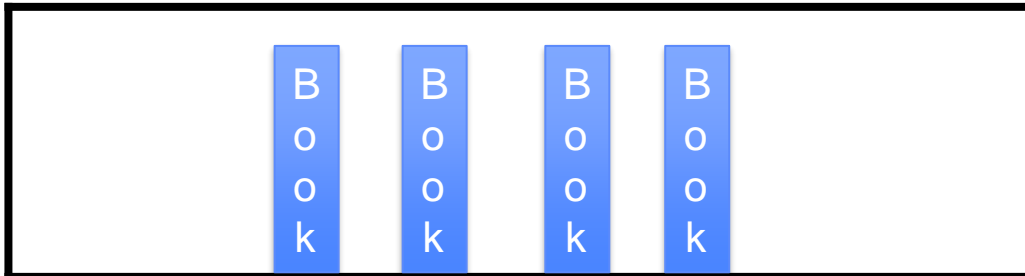
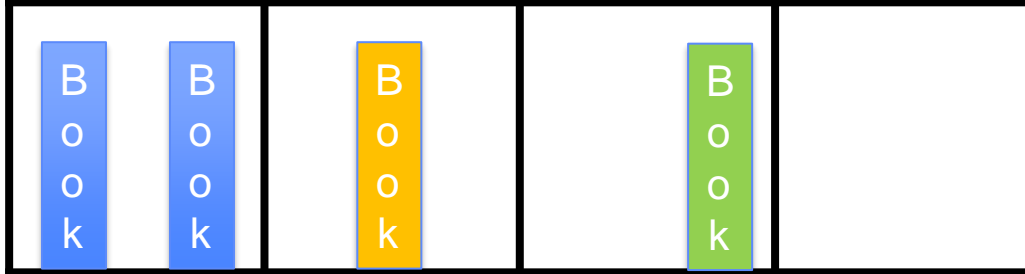
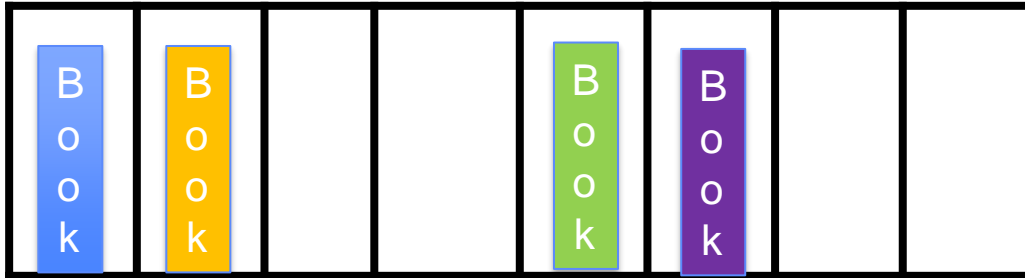
# cache blocks = 2 ways \* 2 sets = 4

**Fully Associative (4-way Set Associative)**



# cache blocks = 4 ways \* 1 set = 4

# Bookshelf Analogy



Tag	Index	Offset
-----	-------	--------

**Direct Mapped (associativity=1):** Each cache set contains one block. A cache block can only go in one position in the cache. It makes a cache block easy to find, but it's not inflexible about where to put it.

Tag	Index	Offset
-----	-------	--------

**Set Associative w/ low associativity:** Each cache set contains 2 blocks. The index is used to find the set, and the tag is used to find the block within the set in case of cache hit.

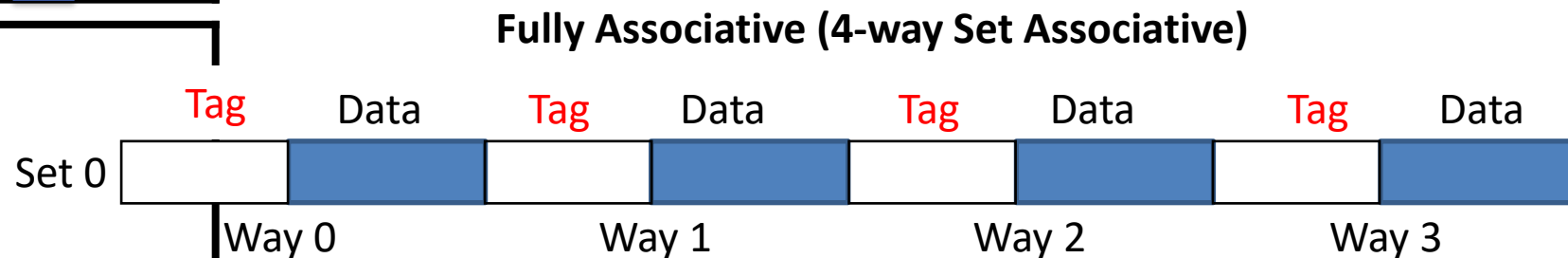
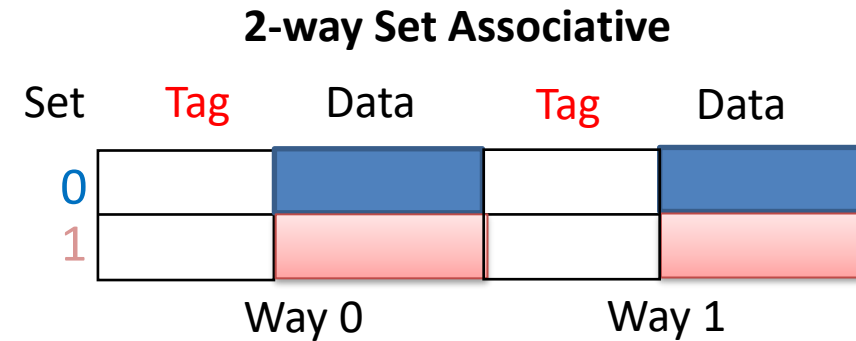
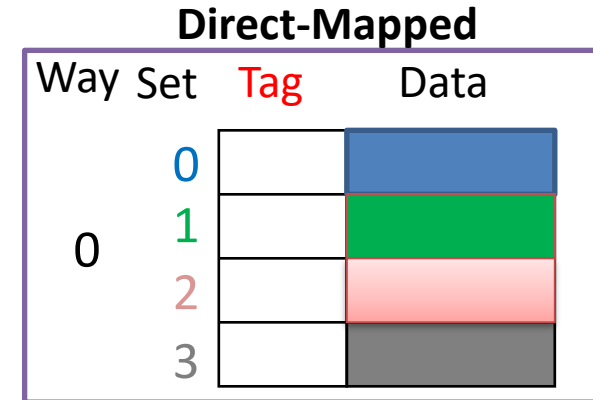
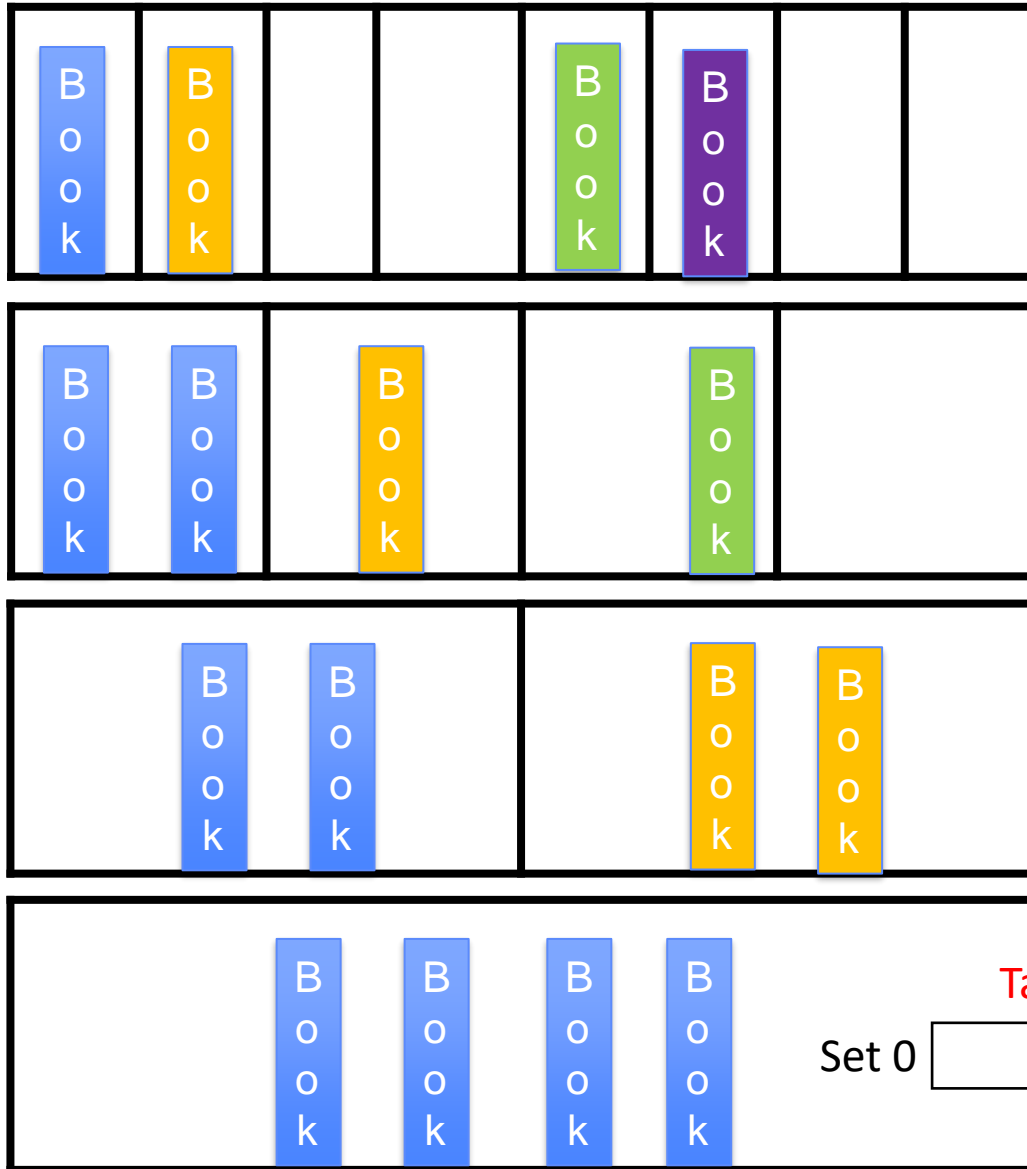
Tag	Index	Offset
-----	-------	--------

**Set Associative w/ high associativity:** Each cache set contains 4 blocks, so there are fewer sets. As such, fewer index bits are needed.

Tag	Offset
-----	--------

**Fully Associative:** No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible.

# Bookshelf Analogy



# Associativity and Performance

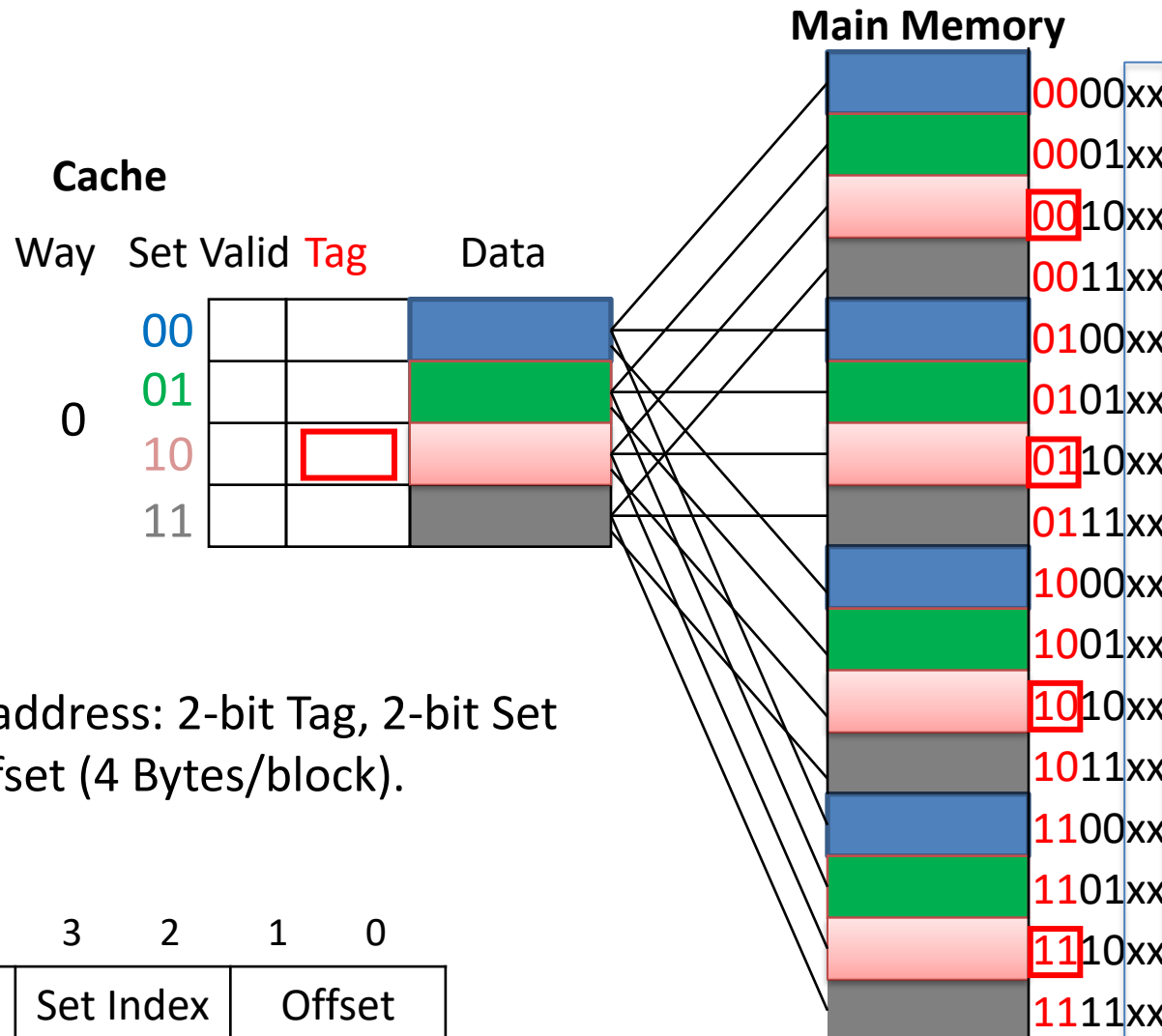
## ■ Increasing associativity:

- **increases** hit time because there are **more** cache blocks per set (books per partition) to compare the tags with to find a match
- **reduces** miss rate by **reducing** conflict misses, since block placement within each set (book placement within each partition) becomes **more** flexible

## ■ Decreasing associativity

- **reduces** hit time because there are **fewer** blocks per set (books per partition) to compare the tags with to find a match
- **increases** miss rate by **increasing** conflict misses, since block placement within each set (book placement within each partition) becomes **less** flexible (may cause Ping-Pong effect)

# Direct-Mapped Cache



Q: Given a memory block, which cache block is it mapped to?

A: Use **2 middle index bits** in memory address to determine which cache block it is mapped to

- 00: mapped to **blue** block in cache
- 01: mapped to **green** block in cache
- 10: mapped to **pink** block in cache
- 11: mapped to **grey** block in cache

Q: Is the memory block in the cache?

A: Compare **2 higher tag bits** in memory address to the **cache tag** to tell if the memory block is in the cache (provided valid bit is set)

Q: Which exact Byte address in the given cache block of 4 Bytes?

A: Use the **Offset**

# DM Cache Example

Q: Given memory address 001110, is it in the cache?

**Cache**

Way	Set	Valid	Tag	Data
0	0	0	01	
	1	1	11	
	2	1	10	
	3	1	00	

6-bit memory address: 2-bit Tag, 2-bit Set Index, 2-bit Offset (4 Bytes/block).

5	4	3	2	1	0
Tag		Set Index		Offset	

## Main Memory

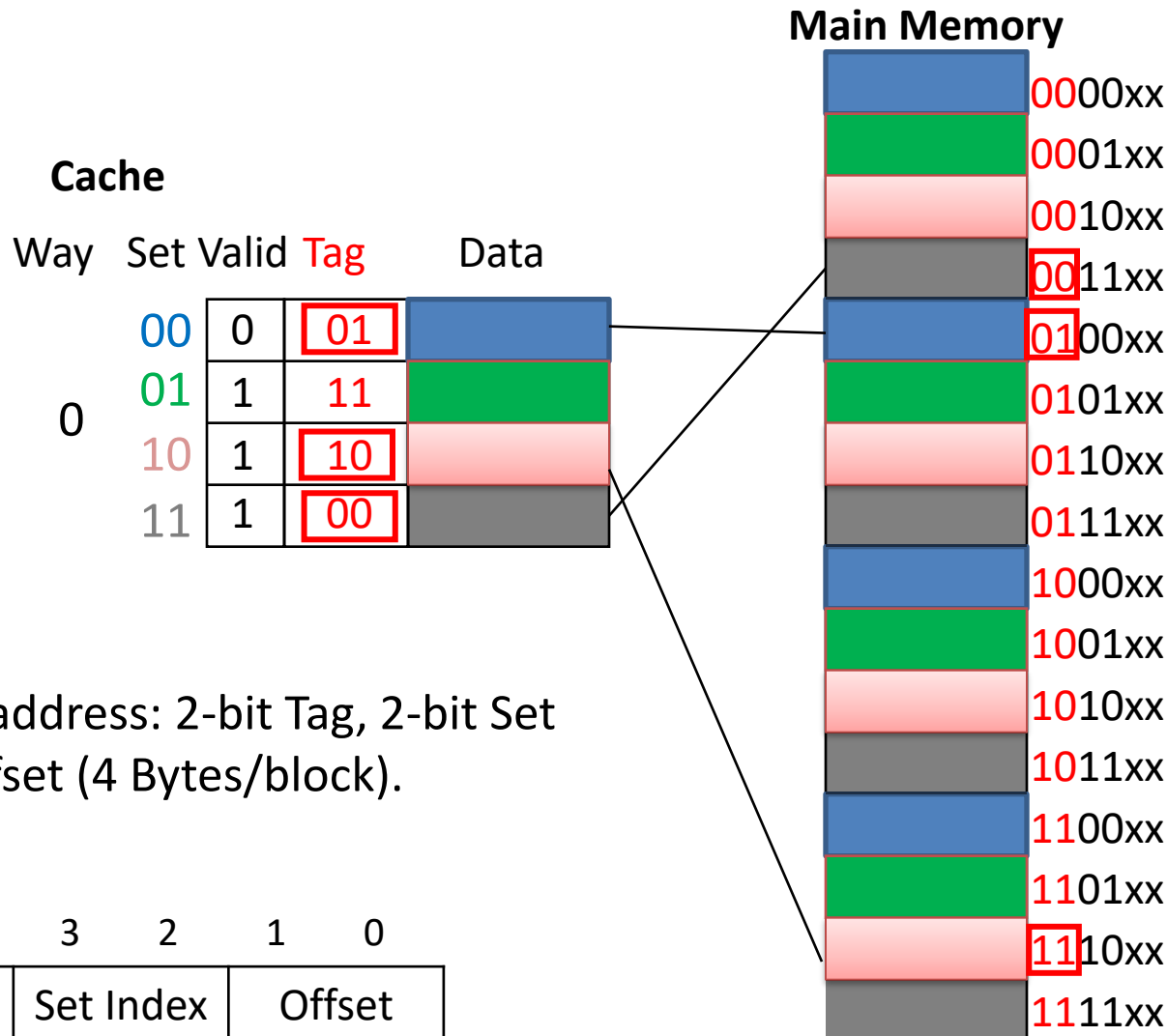
	0000xx
	0001xx
	0010xx
	0011xx
	0100xx
	0101xx
	0110xx
	0111xx
	1000xx
	1001xx
	1010xx
	1011xx
	1100xx
	1101xx
	1110xx
	1111xx

Q: Given memory address 0100xx, is it in the cache?

Q: Given memory address 1110xx, is it in the cache?



# DM Cache Example



Q: Given memory address 001110, is it in the cache?

A: Yes. First, 2 middle index bits (11) means that it is mapped to a grey block in cache; Second, the 2 higher tag bits (00) matches the tag in the grey block, with valid bit of 1; Finally, to get the exact Byte address, use the Offset of 10

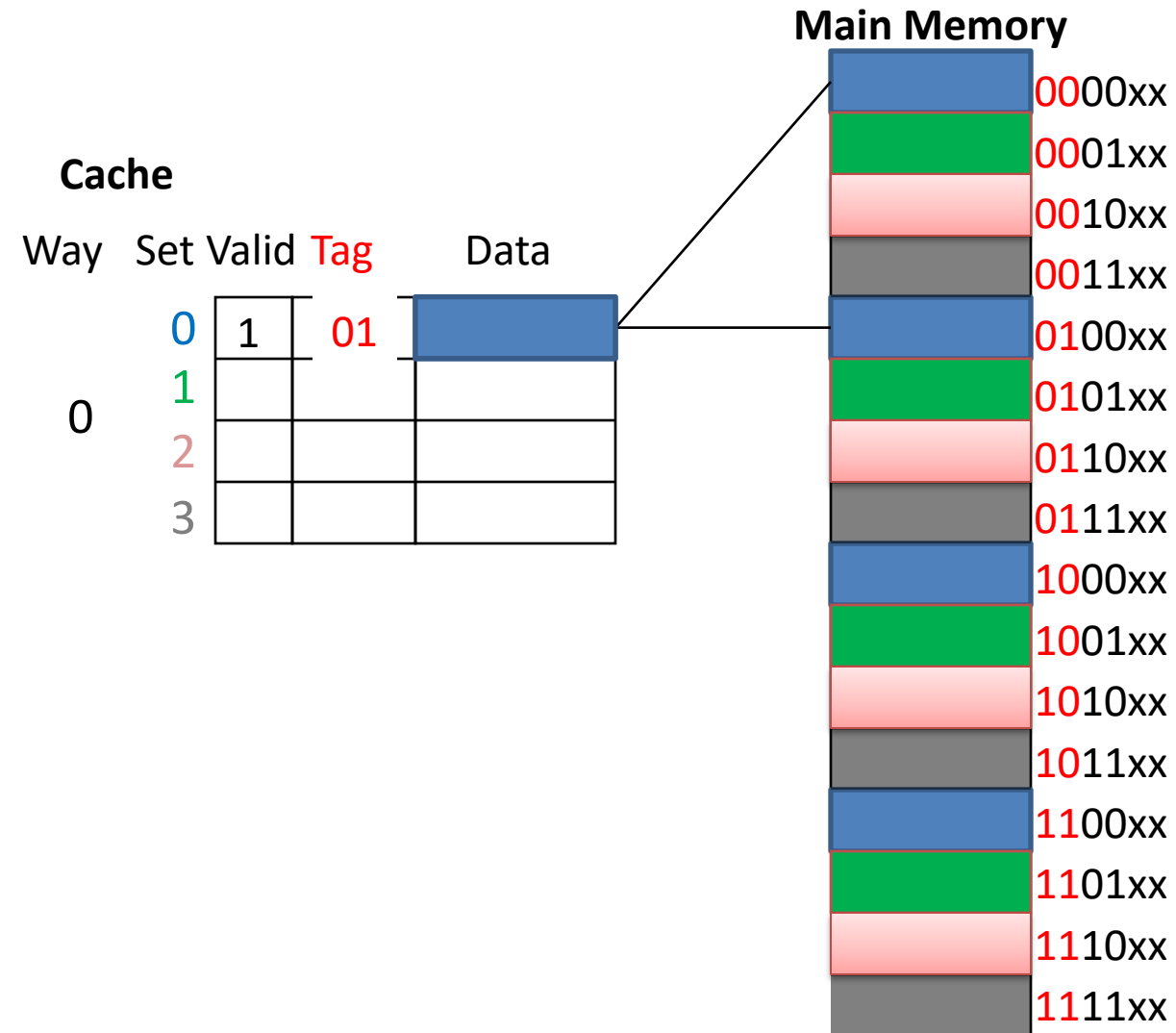
Q: Given memory address 0100xx, is it in the cache?

A: No. First, 2 middle bits (00) means that it is mapped to a blue block in cache; Second, the 2 higher tag bits (01) matches the tag in the blue block, with valid bit of 0, so cache block is invalid.

Q: Given memory address 1110xx, is it in the cache?

A: No. First, 2 middle index bits (10) means that it is mapped to a pink block in cache, with valid bit of 1; Second, the 2 higher tag bits (11) does not match the tag (10) in the pink block.

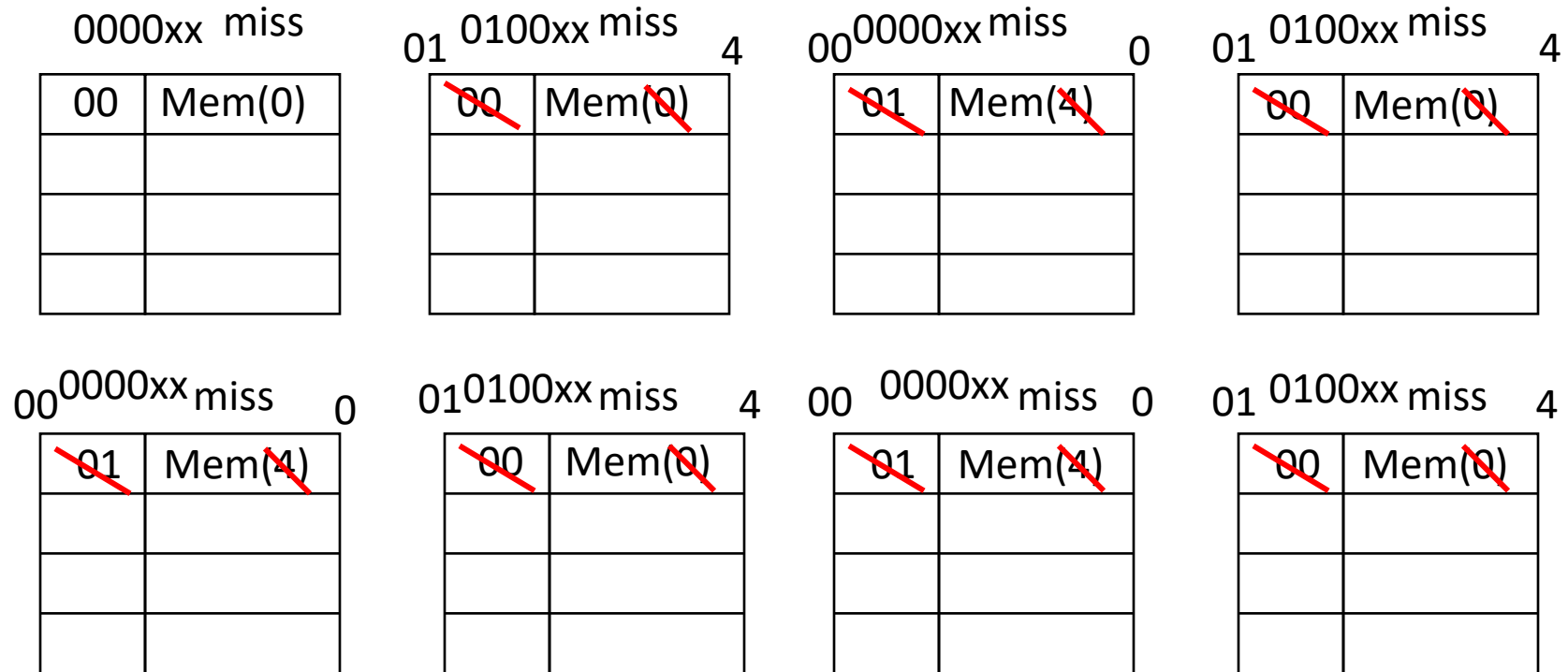
# DM Cache w/ Ping Pong Effect



- Consider the sequence of memory block addresses (0 and 4) referenced at runtime (Offset omitted):
  - 0000xx (0), 0100xx (4), 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx
- They all map to **Set 0**, which contains 1 cache block

# DM Cache w/ Ping Pong Effect

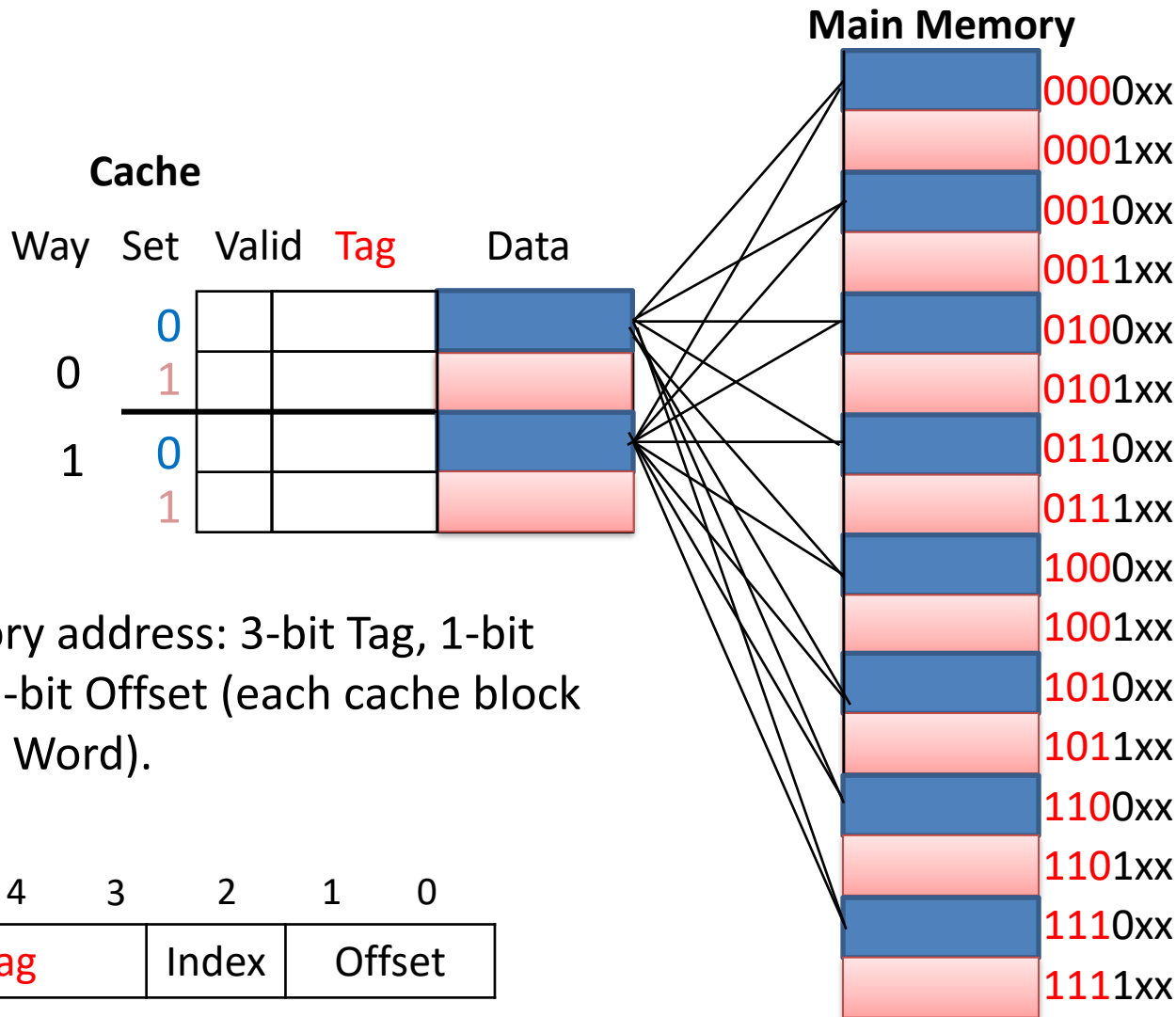
- Consider the sequence of memory addresses referenced at runtime: 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx. All mapped to Set 0.
- Start with an empty cache - all blocks initially marked as not valid



- 8 requests, 8 misses

Ping-pong effect due to conflict misses - two memory addresses that map into the same cache block

# 2-Way Set-Associative Cache



Q: Given a memory block, which cache set is it mapped to?

A: Use **1 middle index bits** in memory address to determine which cache set it is mapped to

0: mapped to **blue** set in cache

1: mapped to **pink** set in cache

Q: Is the memory block in the cache?

A: Compare **3 higher tag bits** in memory address to the **cache tag** to tell if the memory block is in the cache (provided valid bit is set)

# 2-Way SA Cache Example

Q: Given memory address 0111xx, is it in the cache?

Cache				
Way	Set	Valid	Tag	Data
0	0	0	010	
	1	1	111	
1	0	1	101	
	1	1	001	

6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).

5	4	3	2	1	0
Tag			Index	Offset	

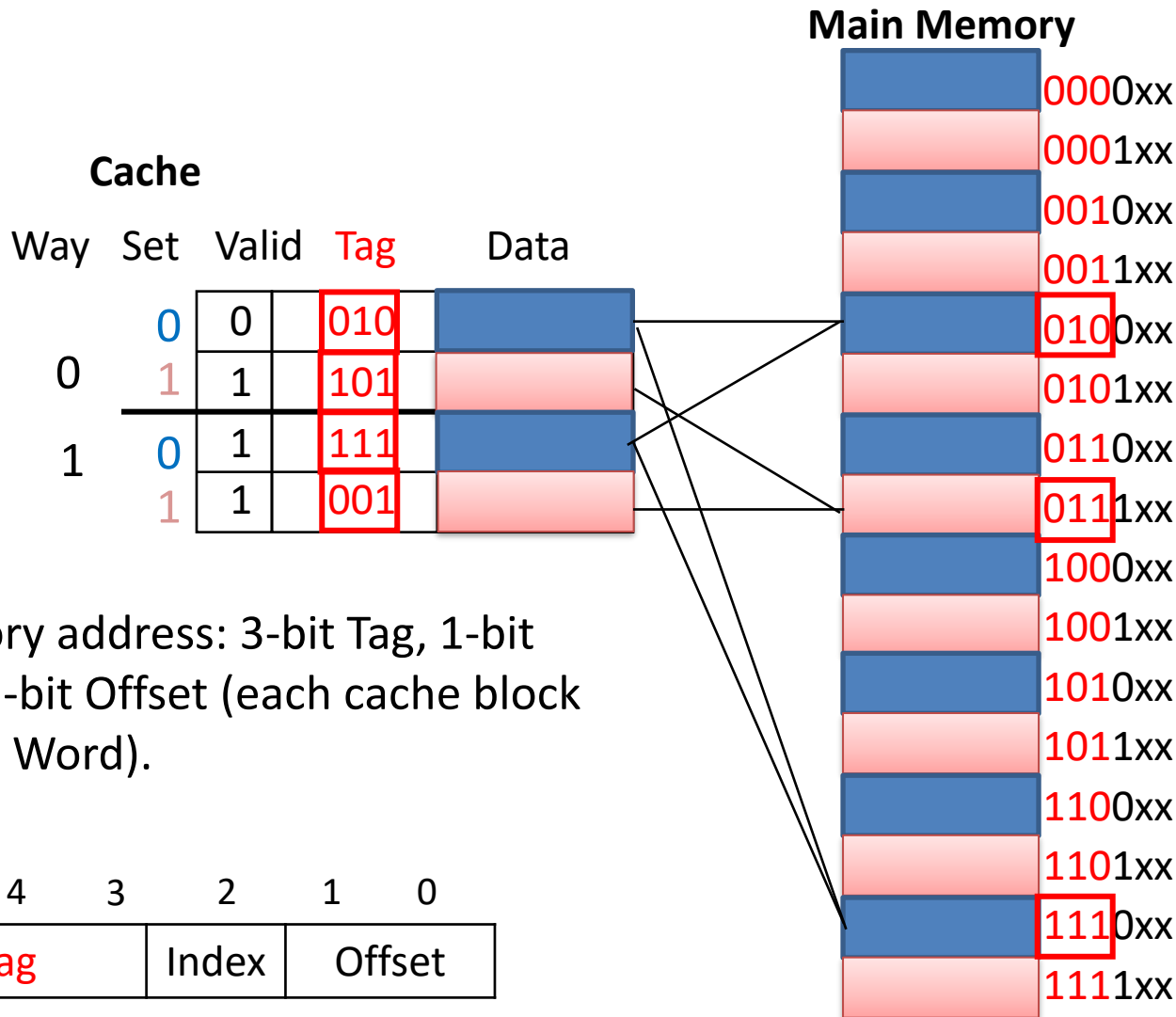
Main Memory

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

Q: Given memory address 0100xx, is it in the cache?

Q: Given memory address 1110xx, is it in the cache?

# 2-Way SA Cache Example



Q: Given memory address 0111xx, is it in the cache?

A: No. First, 1 middle index bit (1) means that it is mapped to **pink** set in cache; Second, the 3 higher tag bits (011) does not match any tag in the **pink** set (101 and 001).

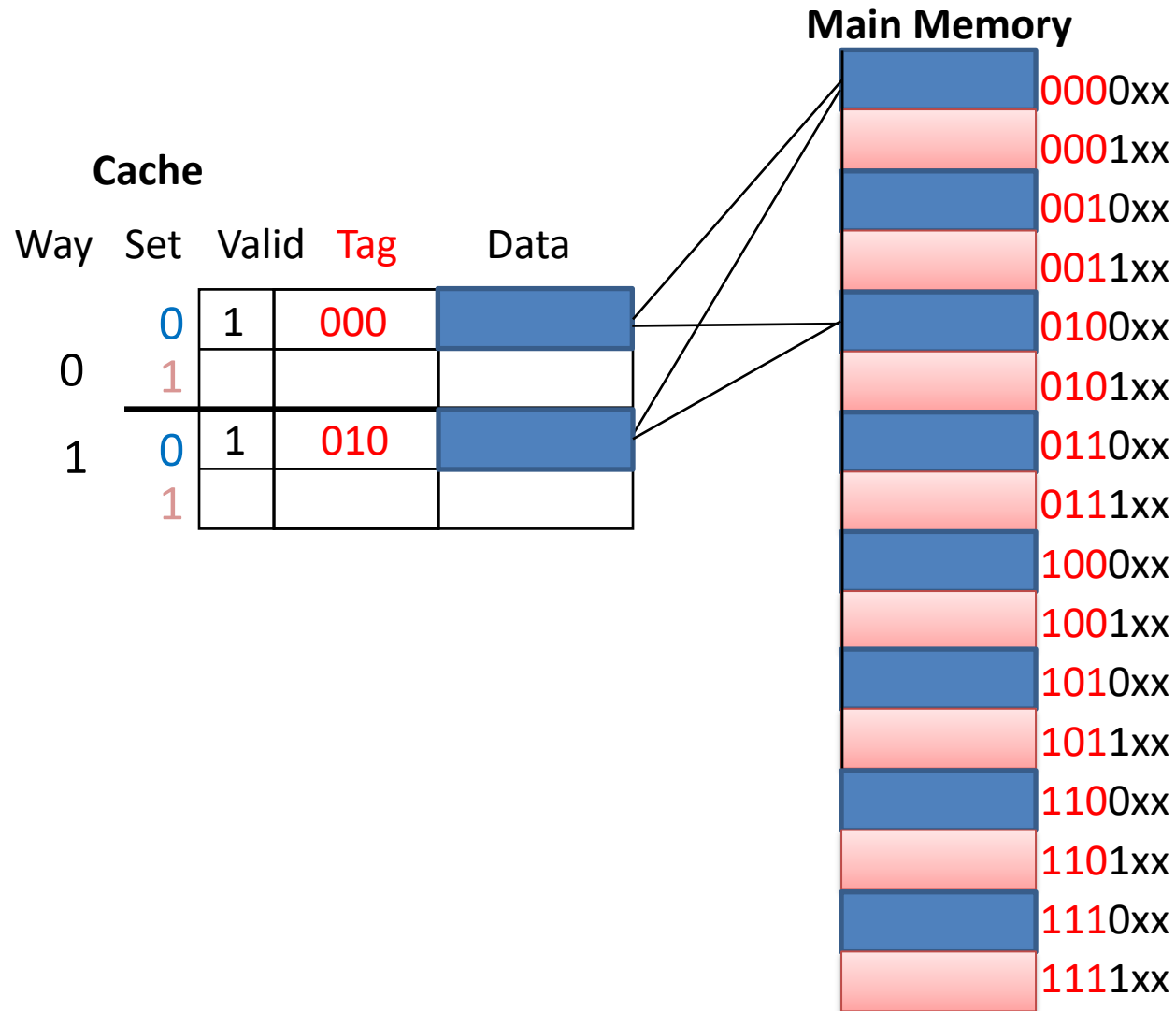
Q: Given memory address 0100xx, is it in the cache?

A: No. First, 1 middle index bit (0) means that it is mapped to **blue** set in cache; Second, the 3 higher tag bits (010) matches one of the tags in the **blue** set (010 and 111); Third, the valid bit of the corresponding cache block is 0.

Q: Given memory address 1110xx, is it in the cache?

A: Yes. First, 1 middle index bit (0) means that it is mapped to **blue** set in cache; Second, the 3 higher tag bits (111) matches one of the tags in the **blue** set (010 and 111); Third, the valid bit is 1.

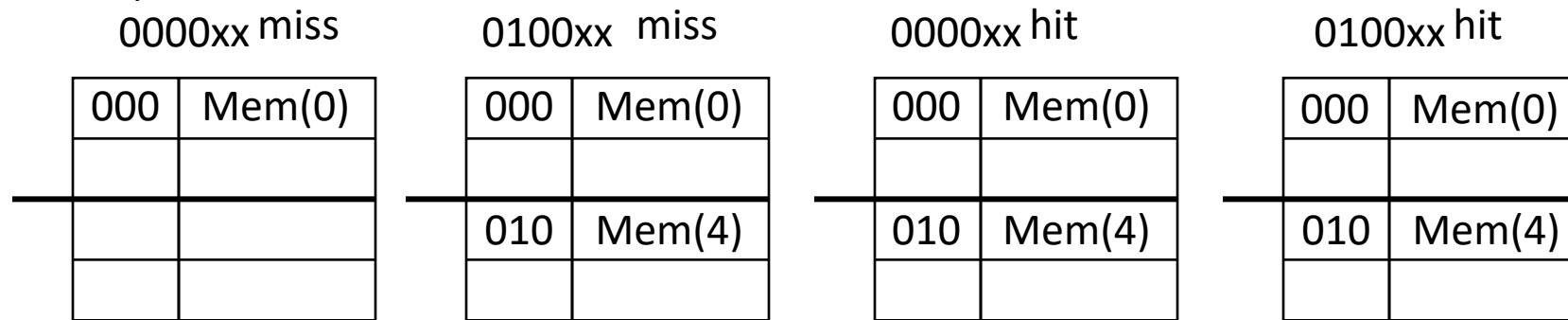
# 2-Way SA Cache w/o Ping Pong Effect



- Consider the sequence of memory block addresses (0 and 4) referenced at runtime (Offset omitted):
  - 0000xx (0), 0100xx (4), 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx
- They all map to **Set 0**, which contains 2 cache blocks. This avoids Ping Pong effect

# 2-Way SA Cache w/o Ping Pong Effect

- Consider the sequence of memory addresses referenced at runtime: 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx. All mapped to Set 0.
  - Start with an empty cache - all blocks initially marked as not valid

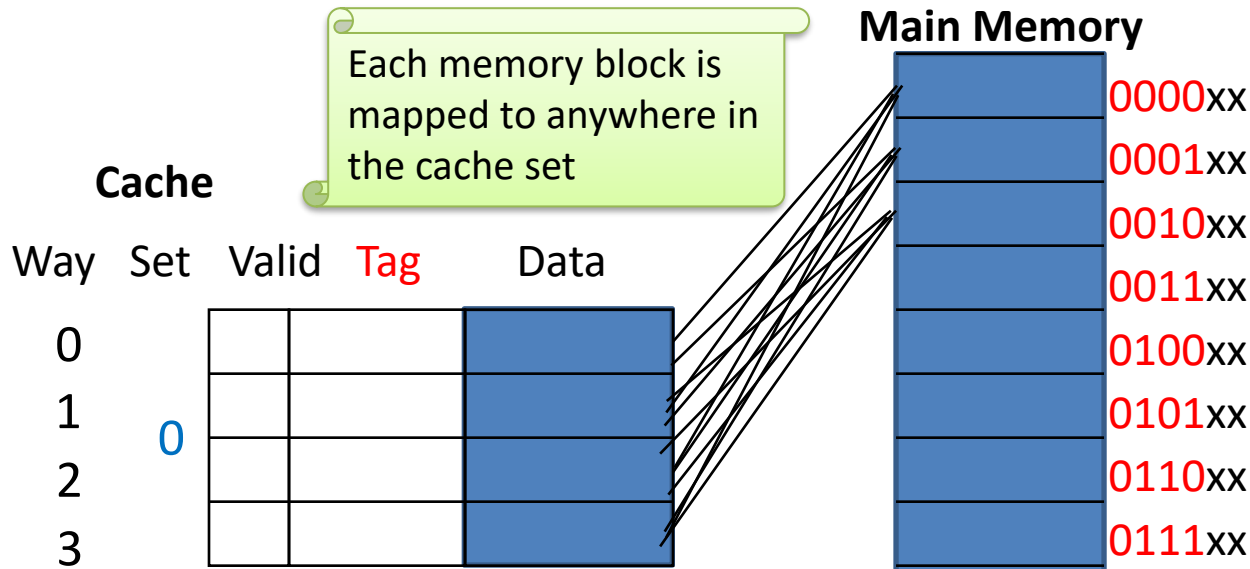


... ..

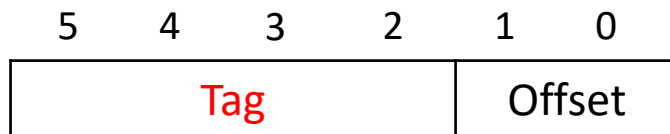
- 8 requests, 2 misses
- Two memory addresses that map into the same cache set can co-exist in the 2-way SA cache, removes the Ping-Pong effect of DM cache



# Fully-Associative Cache (4-way SA)



6-bit memory address: 4-bit Tag, 0-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



Q: Given a memory block, which cache set is it mapped to?

A: There is only a single blue set.

Q: Is the exact memory block in the cache?

A: Compare 4 higher tag bits in memory address to the cache tag to tell if the memory block is in the cache (provided valid bit is set)

Q: Which exact Byte address in a given cache block of 4 Bytes?

A: Use the Offset

# FA Cache Example

Cache				
Way	Set	Valid	Tag	Data
0	0	0	0101	
1		1	1110	
2		1	1010	
3		1	0011	

6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).

5	4	3	2	1	0
Tag				Offset	

Main Memory

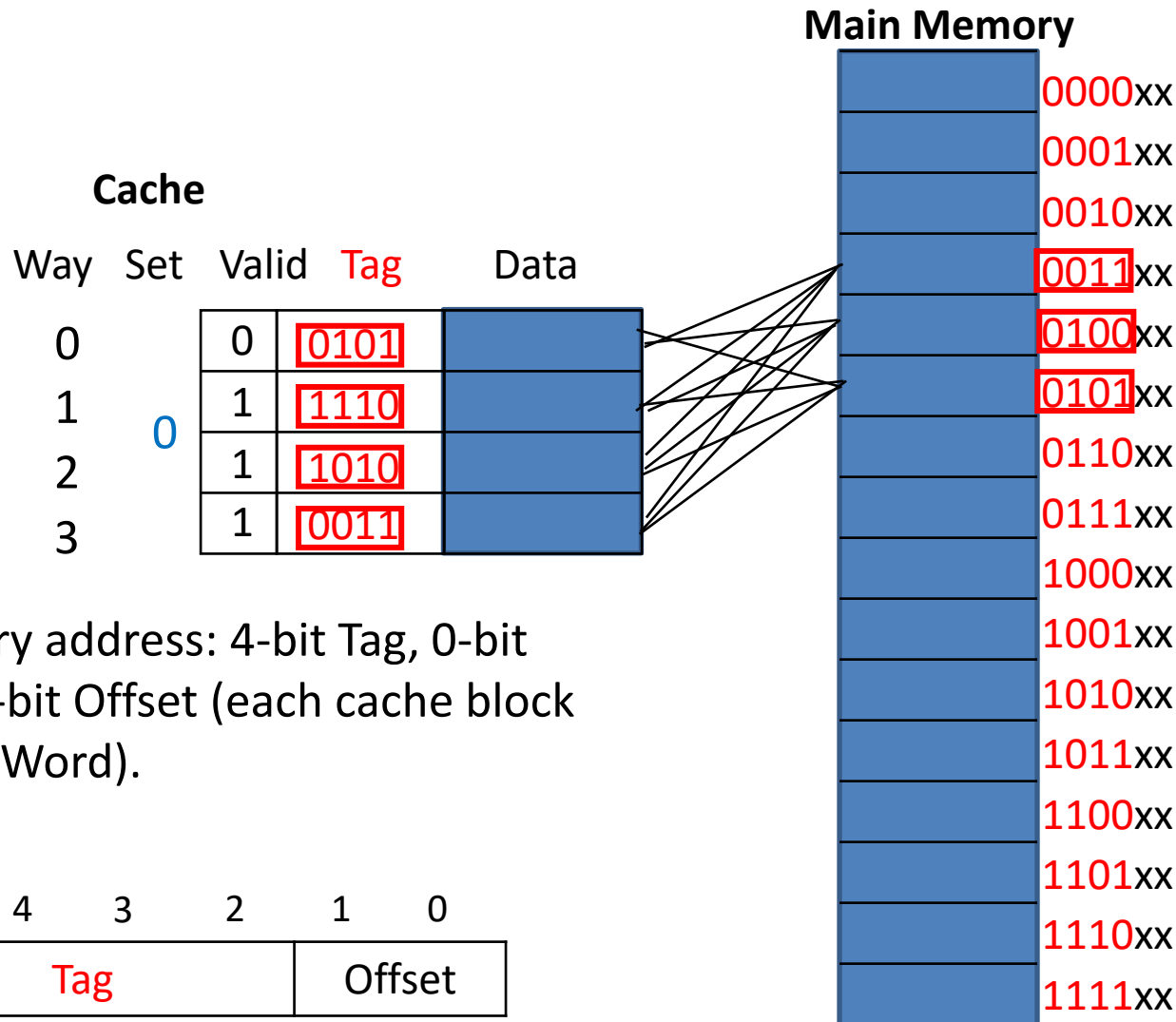
0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx

Q: Given memory address 0011xx, is it in the cache?

Q: Given memory address 0100xx, is it in the cache?

Q: Given memory address 0101xx, is it in the cache?

# FA Cache Example



Q: Given memory address 0011xx, is it in the cache?

A: Yes. The 4 higher tag bits (0011) matches one of the tags in the cache (the blue set), and the valid bit of the corresponding cache block is 1.

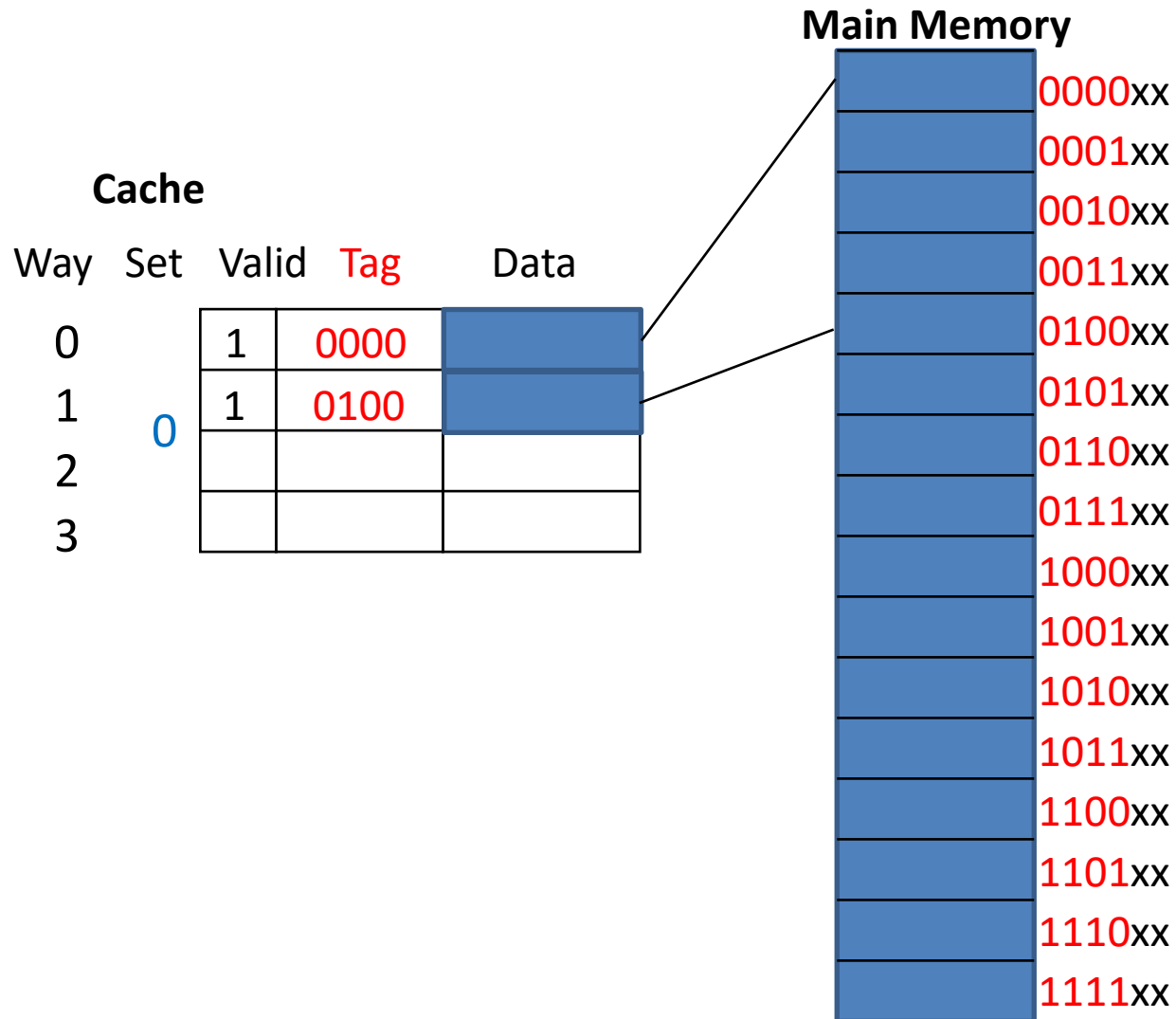
Q: Given memory address 0100xx, is it in the cache?

A: No. The 4 higher tag bits (0100) does not match any of the tags in the cache.

Q: Given memory address 0101xx, is it in the cache?

A: No. The 4 higher tag bits (0101) matches one of the tags in the blue set, but the valid bit of the corresponding cache block is 0.

# FA Cache w/o Ping Pong Effect



- Consider the sequence of memory block addresses (0 and 4) referenced at runtime (Offset omitted):
  - 0000xx (0), 0100xx (4), 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx
- They all map to **Set 0**, which contains 4 cache blocks. This avoids Ping Pong effect

# FA Cache w/o Ping Pong Effect

- Consider the sequence of memory addresses referenced at runtime: 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx, 0000xx, 0100xx. All mapped to Set 0.

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

0000xx miss

000	Mem(0)

0100xx miss

000	Mem(0)
010	Mem(1)

0000xx hit

000	Mem(0)
010	Mem(1)

0100xx hit

000	Mem(0)
010	Mem(1)

... ..

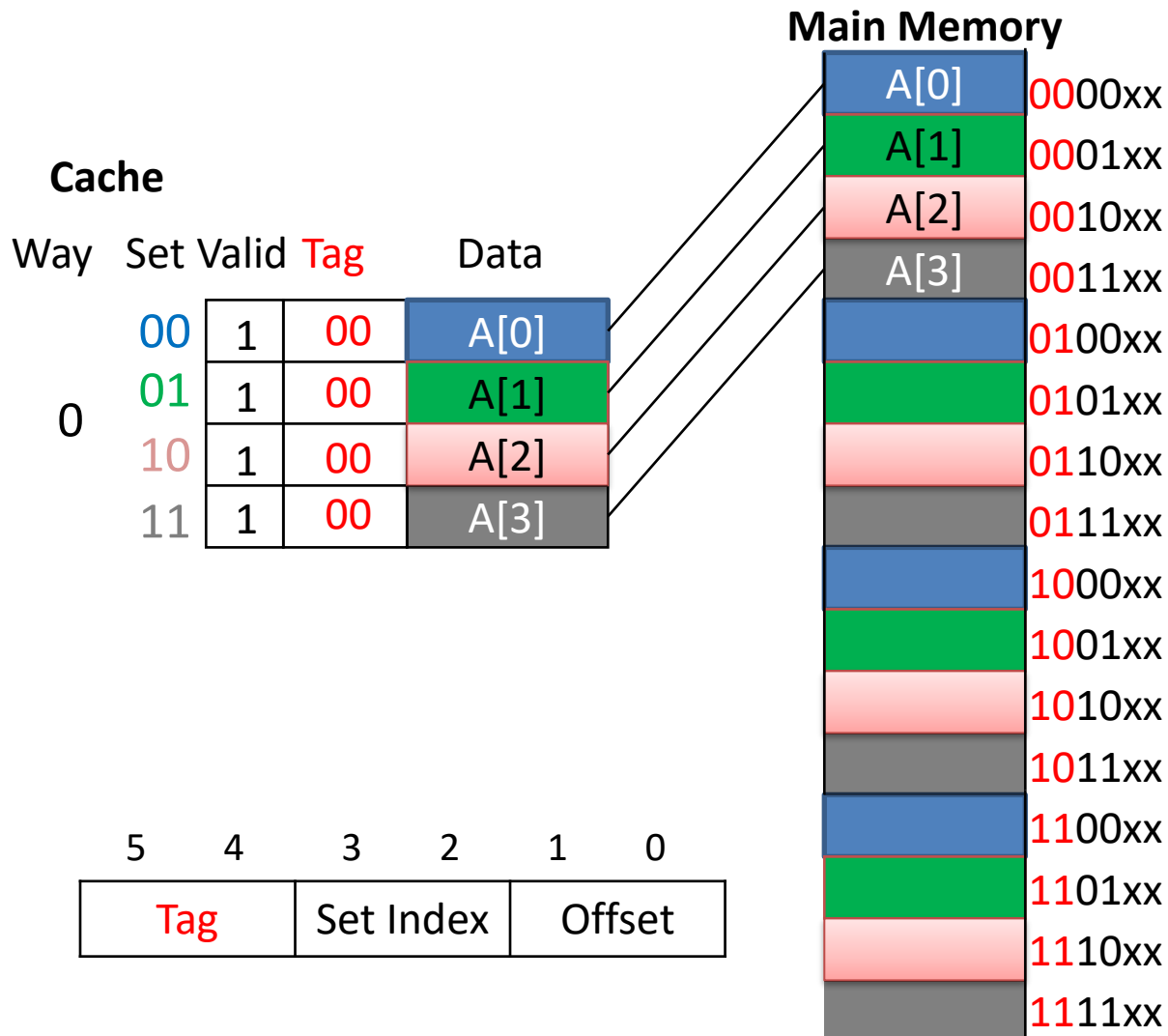
- 8 requests, 2 misses
- Two memory addresses that map into the same cache set can co-exist in the FA cache, removes the Ping-Pong effect of DM cache

## Example: Calculate Number of Cache Misses

- Consider the following program that repeatedly accesses an array A of four words (4 bytes each):  

```
for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}
```
- Suppose A[0] starts at memory address 000000, then A[0], A[1], A[2], A[3] start at memory addresses 000000, 000100, 001000, 001100, respectively, and this sequence of memory addresses are visited repeatedly in a cyclic manner.
- 1) How many cache misses occur due to reading array A[] for a **DM cache**?
- 2) Answer the same questions for a two-way **SA cache**.
- 3) Answer the same questions for a **FA cache**.

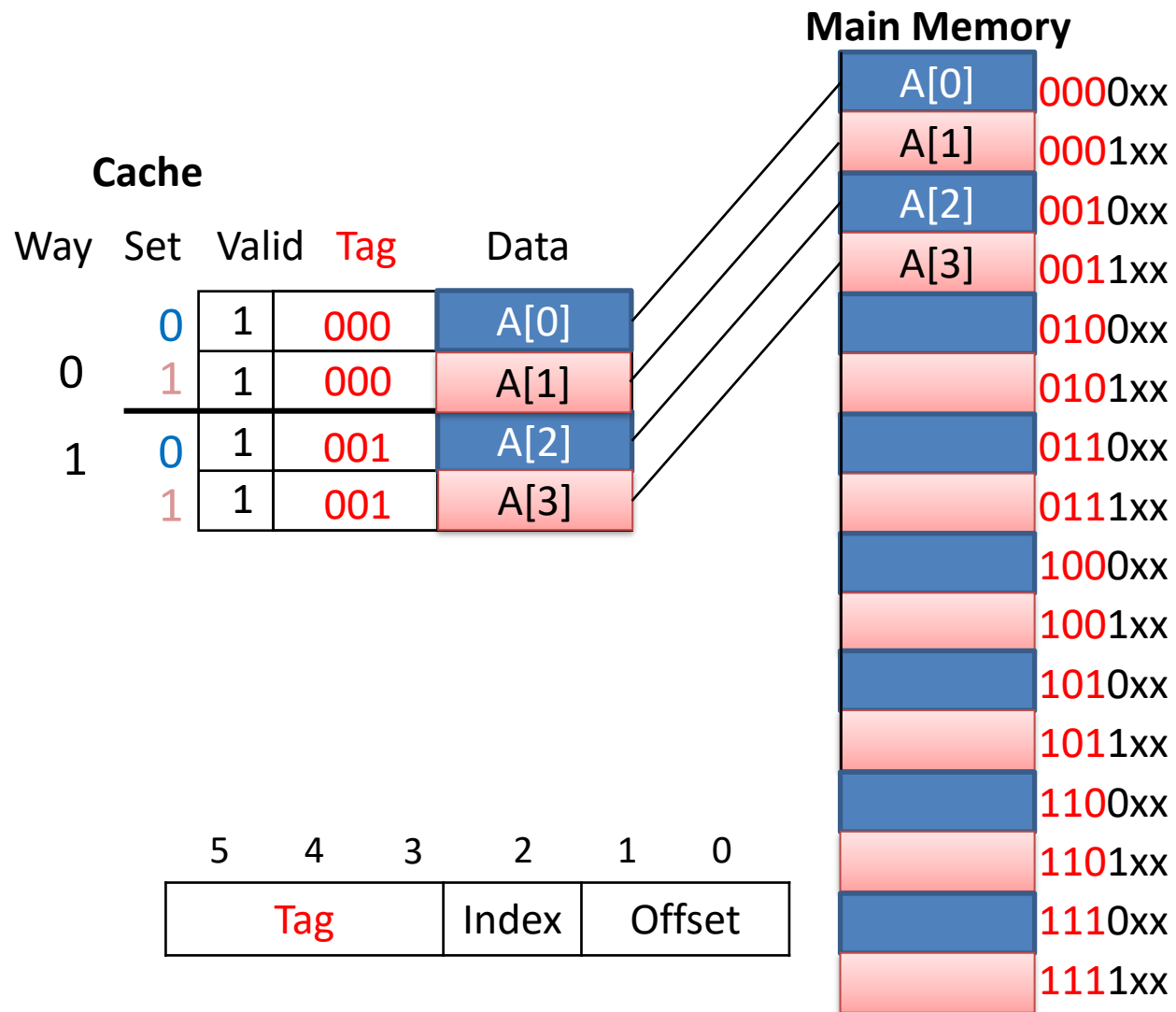
# Example: DM Cache



## DM cache: 4 misses

- 1<sup>st</sup> cache miss brings a block with address 0000xx into cache Set 0, which contains A[0]
- 2<sup>nd</sup> cache miss brings a block with address 0001xx into cache Set 1, which contains A[1]
- 3<sup>rd</sup> cache miss brings a block with address 0010xx into cache Set 2, which contains A[2]
- 4<sup>th</sup> cache miss brings a block with address 0011xx into cache Set 3, which contains A[3]
- All subsequent cache accesses are cache hits
- After 10000 iterations, 4 cache misses, 9996 cache hits.

# Example: 2-Way SA Cache

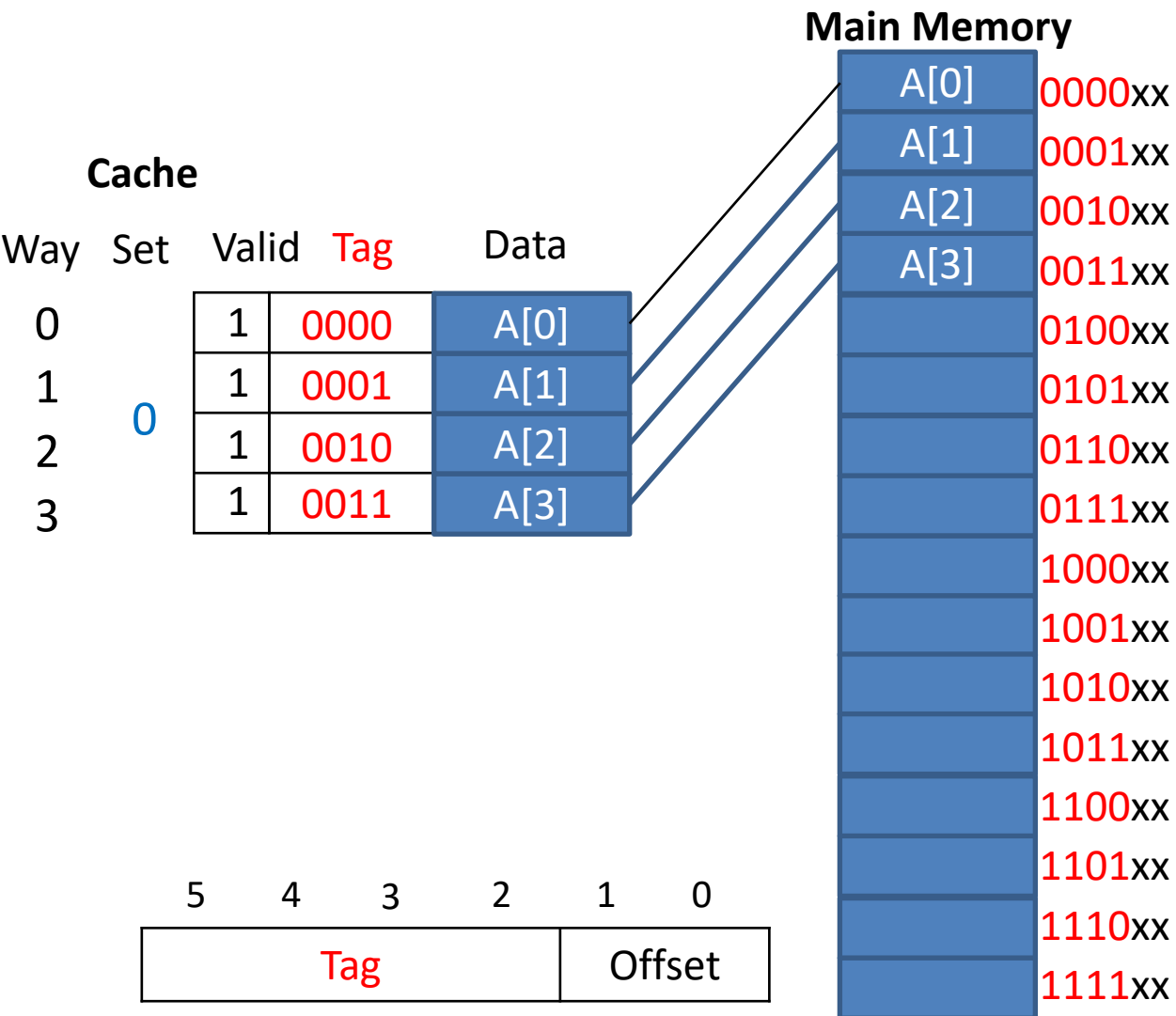


## 2-Way SA Cache: 4 misses

- 1<sup>st</sup> cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2<sup>nd</sup> cache miss brings into cache a block with address 0001xx into Set 1, which contains A[1]
- 3<sup>rd</sup> cache miss brings into cache a block with address 0010xx into Set 0, which contains A[2]
- 4<sup>th</sup> cache miss brings into cache a block with address 0011xx into Set 1, which contains A[3]
- All subsequent cache accesses are cache hits
- After 10000 iterations, 4 cache misses, 9996 cache hits



# Example: FA Cache



FA cache: 4 misses with LRU (Least-Recently-Used) replacement policy

- 1<sup>st</sup> cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2<sup>nd</sup> cache miss brings into cache a block with address 0001xx into Set 0, which contains A[1]
- 3<sup>rd</sup> cache miss brings into cache a block with address 0010xx into Set 0, which contains A[2]
- 4<sup>th</sup> cache miss brings into cache a block with address 0011xx into Set 0, which contains A[3]
- All subsequent cache accesses are cache hits
- After 10000 iterations, 4 cache misses, 9996 cache hits.

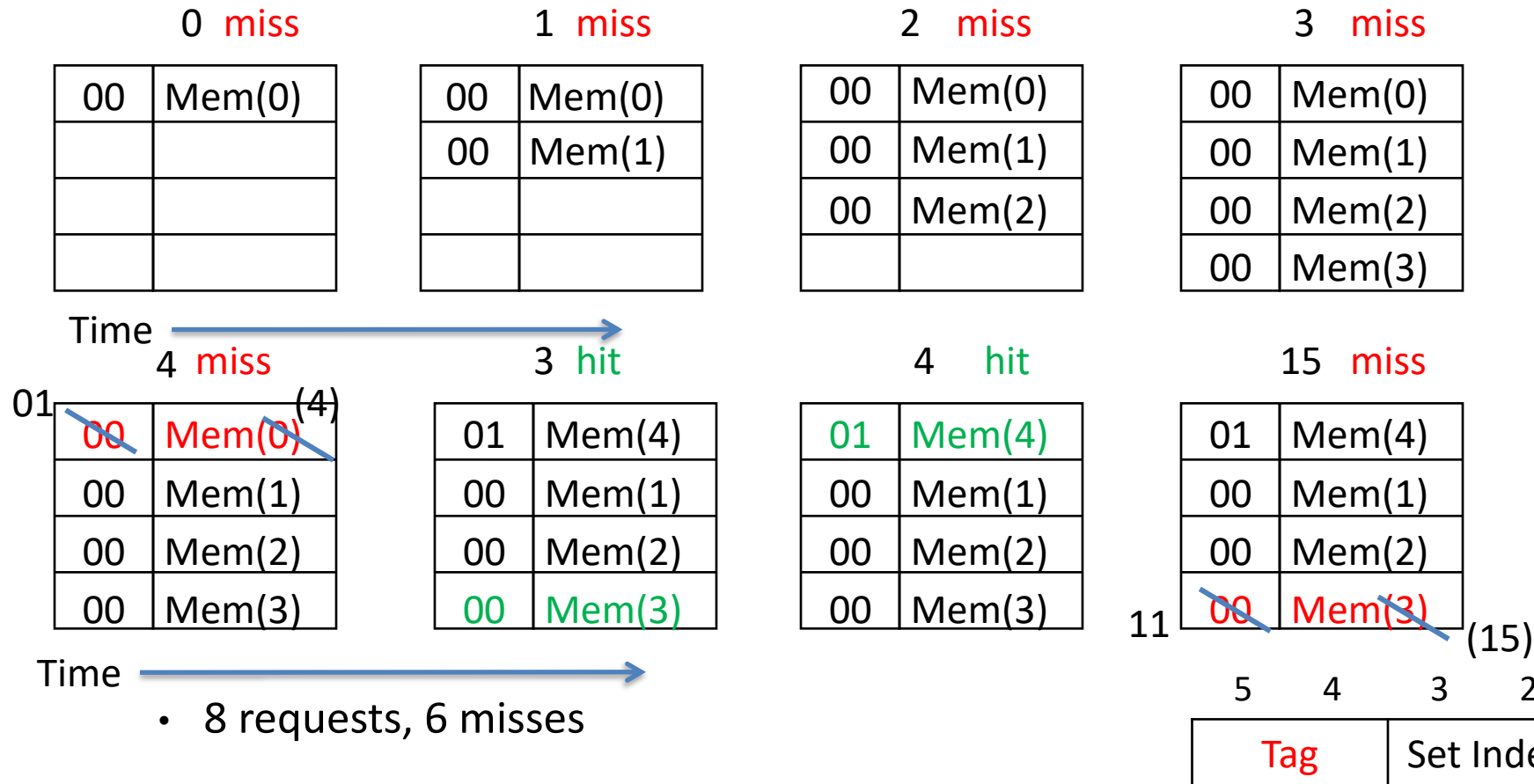
# Tradeoffs of Cache Block Sizes

- Smaller cache blocks → more fine-grained caching → take better advantage of temporal locality
  - A given data item stays in the cache longer before getting replaced
- Larger cache blocks → more coarse-grained caching, take better advantage of spatial locality
  - Fetching each cache block brings in lots of data at nearby addresses into the cache

# DM Cache: Memory Access Example

- Consider 6-bit memory address with Tag 2b; Index 2b; Offset 2b (We ignore Byte offset bits, and only consider 4-bit word addresses: Tag 2b; Index 2b)
- Start with an empty cache - all blocks initially marked as not valid. Fill in the cache state table below for the sequence of memory address accesses:

0 1 2 3 4 3 4 15  
0000 0001 0010 0011 0100 0011 0100 1111



# DM cache: larger block size helps take advantage of spatial locality

- Each cache block holds 2 words; so Tag 2b; Index 1b; Offset 3b for Byte address (we use Offset 1b to refer to 1 of 2 words in block, not Bytes.)

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

0	1	2	3	4	3	4	15
0000	0001	0010	0011	0100	0011	0100	1111
0 miss	1 hit					2 miss	
00	Mem(1)	Mem(0)	00	Mem(1)	Mem(0)	00	Mem(1)
							Mem(3)
							Mem(2)

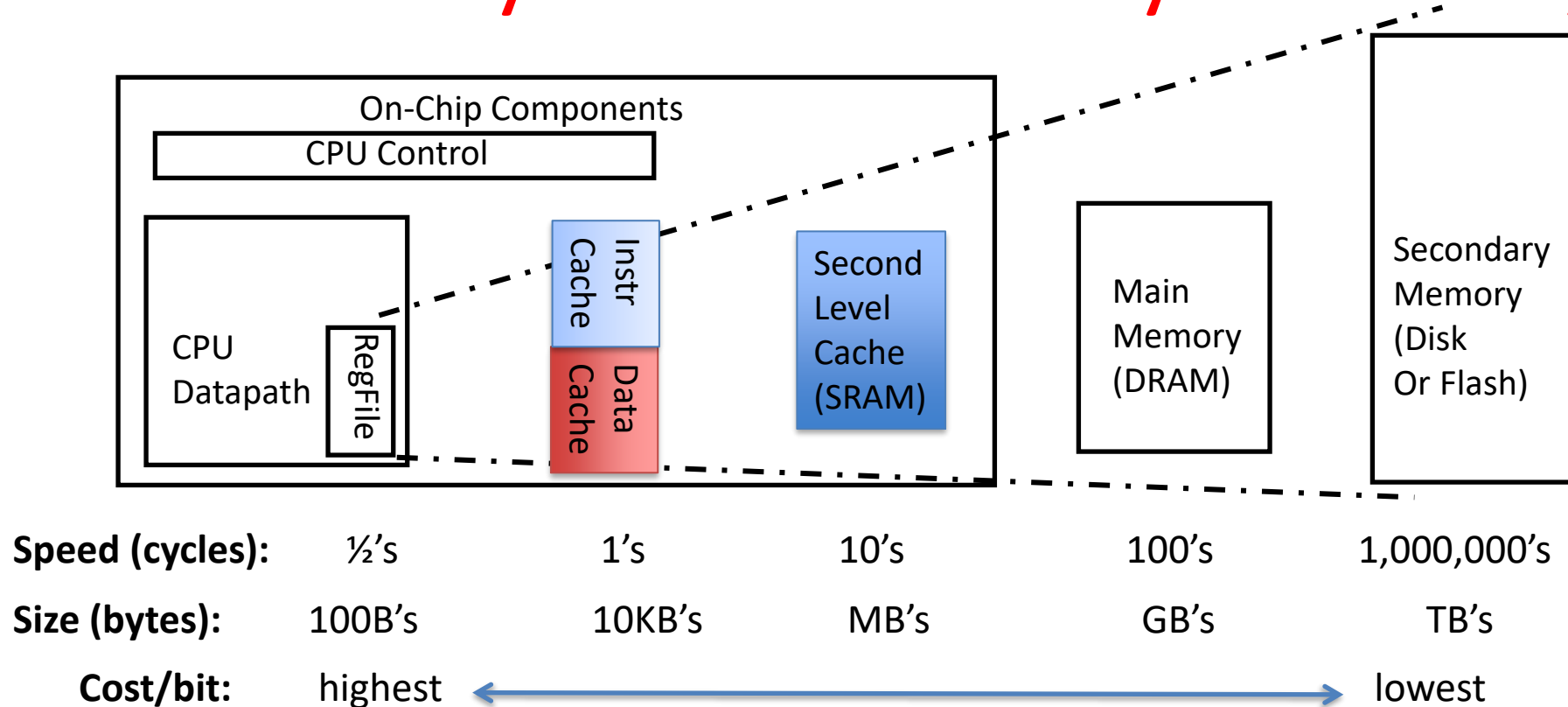
3 hit	4 miss	3 hit
00	01	01
Mem(1)	Mem(1)	Mem(5)
Mem(0)	Mem(1)	Mem(4)
00	00	00
Mem(3)	Mem(3)	Mem(3)
Mem(2)	Mem(2)	Mem(2)

4 hit	15 miss
01	11
Mem(5)	Mem(5)
Mem(4)	Mem(5)
00	00
Mem(3)	Mem(3)
Mem(2)	Mem(2)

- 8 requests, 4 misses

5	4	3	2	1	0
Tag	Index	Offset			

# Associativity in the Memory Hierarchy



- The closer to the CPU (e.g., L1/L2 cache), the lower associativity (e.g., Direct Mapped), to minimize hit time, since CPU needs to access cache contents fast
- The farther away from the CPU (e.g., Last-Level Cache), the higher associativity (e.g., Fully-Associative), to minimize miss rate, since cache misses will go to main memory, which is very slow
- Main memory can be viewed a “cache” for disk or Flash, and it is a Fully-Associative cache

# Summary of Cache Organizations

- A memory block is mapped to one **cache set**, which may contain one or more cache blocks
- **Direct Mapped (DM)**
  - Each cache set has 1 cache block;  $\# \text{ cache sets} = \# \text{ cache blocks}$
  - A memory block is mapped to 1 possible cache block
- **Fully Associative (FA)**
  - A single cache set contains all cache blocks;  $\# \text{ cache sets} = 1$
  - A memory block can be mapped to any cache block
- **N-way Set Associative (SA)**
  - Each cache set has N cache blocks;  $\# \text{ cache sets} = \# \text{ cache blocks} / N$
  - N is also called associativity
  - A memory block can be mapped to one of N possible cache blocks
- DM and FA are special cases of SA
  - DM = 1-way SA ( $N = 1$ )
  - FA = N-way SA ( $N = \text{total number of cache blocks}$ )

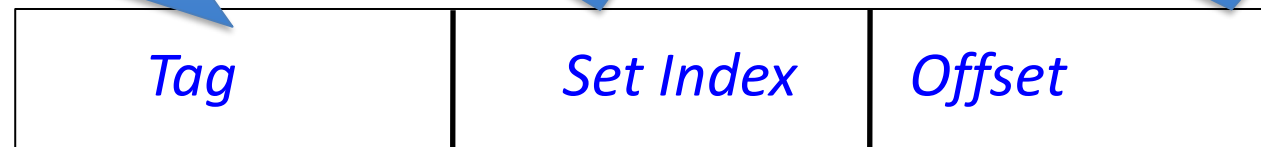
# Key Equations

$\# \text{ sets} = 2^{\text{SI size}}$ ;  $\# \text{ Bytes/block} = 2^{\text{Offset size}}$   
 $\# \text{ blocks} = \# \text{ ways (associativity)} * \# \text{ sets}$   
 $\text{cache capacity} = \# \text{ blocks} * \# \text{ Bytes/block}$

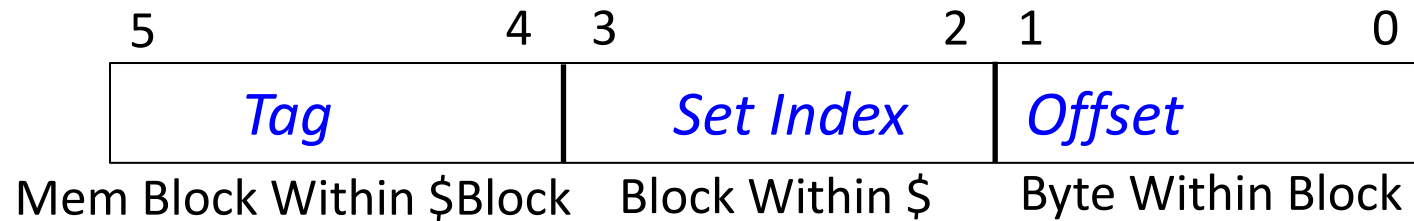
Tag size does not affect cache capacity; depends on memory address length

SI size determines  
 $\# \text{ sets} = 2^{\text{SI size}}$

Offset size determines  
 $\text{Bytes/block} = 2^{\text{Offset size}}$



# Cache Example

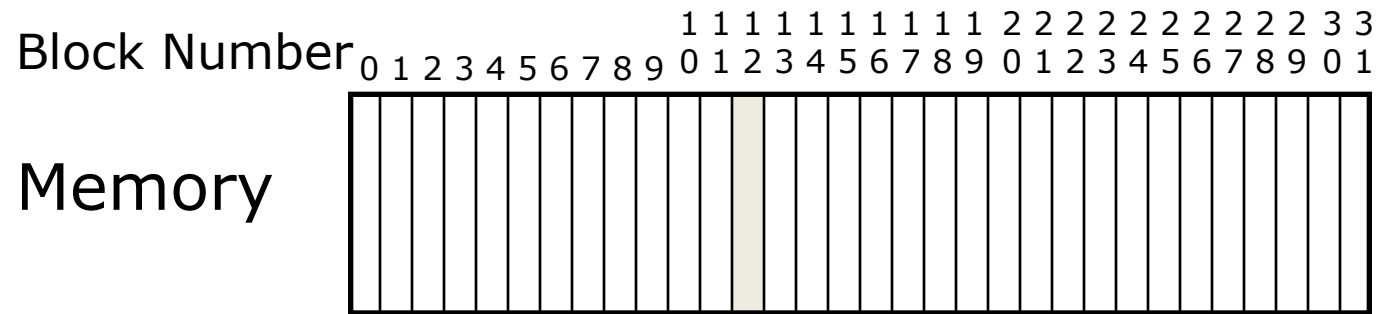


- Assume: DM cache; 6-bit memory address: 2-bit Tag, 2-bit index, 2-bit Offset. Compute cache capacity and memory size.
  - 2-bit Offset => Bytes/block = 4;
  - # sets =  $2^{\text{SI Size}} = 4$
  - # cache blocks = # ways \* # sets =  $1 * 4 = 4$
  - cache capacity = # cache blocks \* Bytes/block =  $4 * 4 = 16\text{B}$
- Memory size:  $2^4$  (2-bit tag + 2-bit SI) = 16 blocks = 64 Bytes

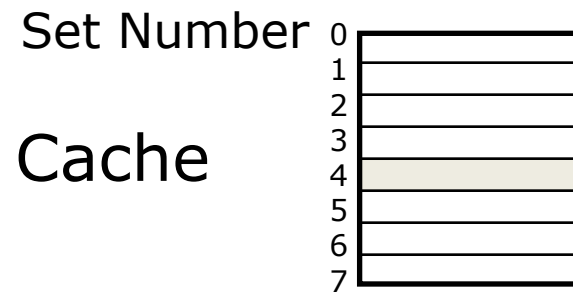
# sets =  $2^{\text{SI size}}$ ; # Bytes/block =  $2^{\text{Offset size}}$   
# blocks = # ways (associativity) \* # sets  
cache capacity = # blocks \* # Bytes/block



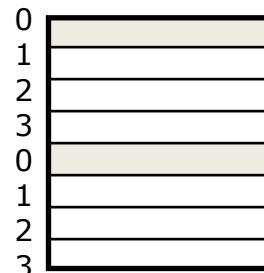
# Alternative Cache Organizations (8-block cache)



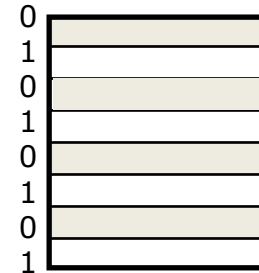
Where are possible locations in cache that block #12 in memory can be placed?



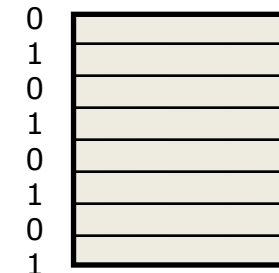
DM  
In set 4  
(1 block)



2-way SA  
In set 0  
(2 blocks)



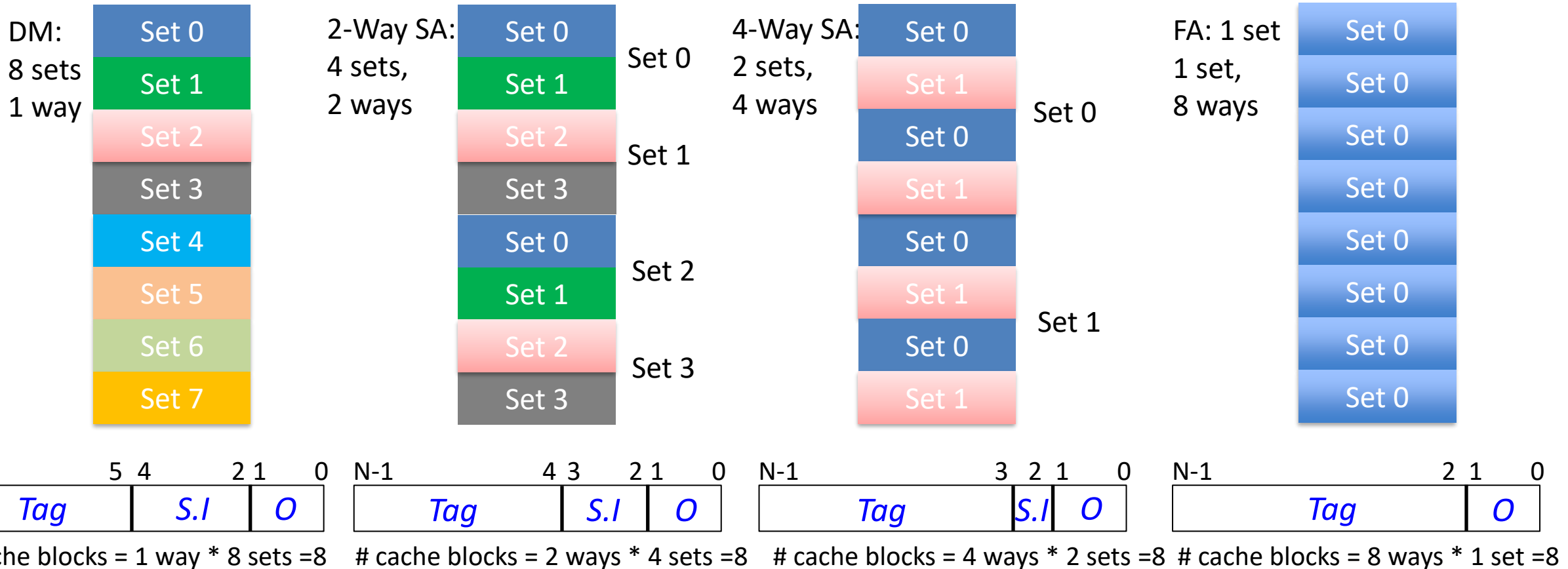
4-way SA  
In set 0  
(4 blocks)



FA(8-way SA)  
In set 0  
(8 blocks)

# 8-Block Cache

- Each color denotes a cache set



# 8-Block Cache Summary

## One-way set associative

(direct mapped)

Block	Tag	Data
0		Blue
1		Green
2		Red
3		Grey
4		Cyan
5		Orange
6		Light Green
7		Yellow

## Two-way set associative

Set	Tag	Data	Tag	Data
0		Blue		Blue
1		Green		Green
2		Red		Red
3		Grey		Grey

## Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0		Blue		Blue		Blue		Blue
1		Green		Green		Green		Green

## Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data
	Blue		Blue		Blue		Blue		Blue		Blue		Blue		Blue

- Cache: 8 blocks, each 4 Bytes. # cache blocks is equal to number of sets x associativity.
- For fixed cache size, increasing associativity decreases number of sets while increasing number of blocks per set.
  - DM (1-way SA): 8 sets x 1 block per set
  - 2-way SA: 4 sets x 2 blocks per set
  - 4-way SA: 2 sets x 4 blocks per set
  - FA (4-way SA): 1 set x 8 blocks per set
- Higher associativity → More ways → fewer cache sets → cache structure is more “short (vertically) and fat (horizontally)”
- Lower associativity → Fewer ways → more cache sets → cache structure is more “tall (vertically) and skinny (horizontally)”

# YouTube: How Cache Works Inside a CPU



How Cache Works Inside a CPU

<https://www.youtube.com/watch?v=zF4VMombo7U&list=PL38NNHQLqJqYnNrTenxBvGJSPCKv9EOWk&index=1>

# Outline

- Cache Introduction
- Cache Organization
- Cache Performance Analysis

# Cache (*Performance*) Terms

- **Hit rate**: fraction of accesses that hit in the cache
- **Miss rate**:  $1 - \text{Hit rate}$
- **Miss penalty**: time to access a block from lower level in memory hierarchy
- **Hit time**: time to access cache memory (including tag comparison)

# 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

$$\text{AMAT} = \text{Hit rate} * \text{Hit time} + \text{Miss rate} * \text{Miss time}$$

$$= (1 - \text{Miss rate}) * \text{Hit time} + \text{Miss rate} * (\text{Hit time} + \text{Miss penalty})$$

$$= \text{Hit time} + \text{Miss rate} * \text{Miss penalty}$$

- For single-level cache, Miss penalty = Memory access time, since Miss time = Memory access time + Hit time

# AMAT Example

**AMAT = Hit time + Miss rate \* Miss penalty**

- Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 2%, and a cache hit time of 1 clock cycle, what is AMAT?

**A** :  $\leq 200$  psec

**B** : 400 psec

**C** : 600 psec

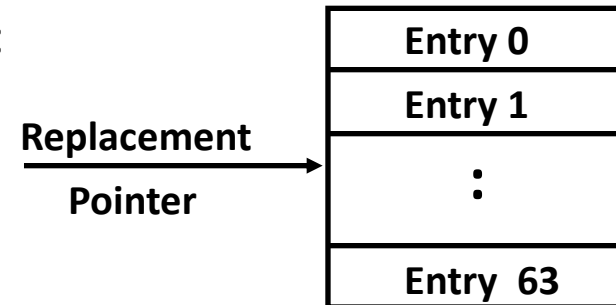
**D** :  $\geq 800$  psec

$$\text{AMAT} = (1 + 0.02 * 50) * 200 = 400 \text{ psec}$$



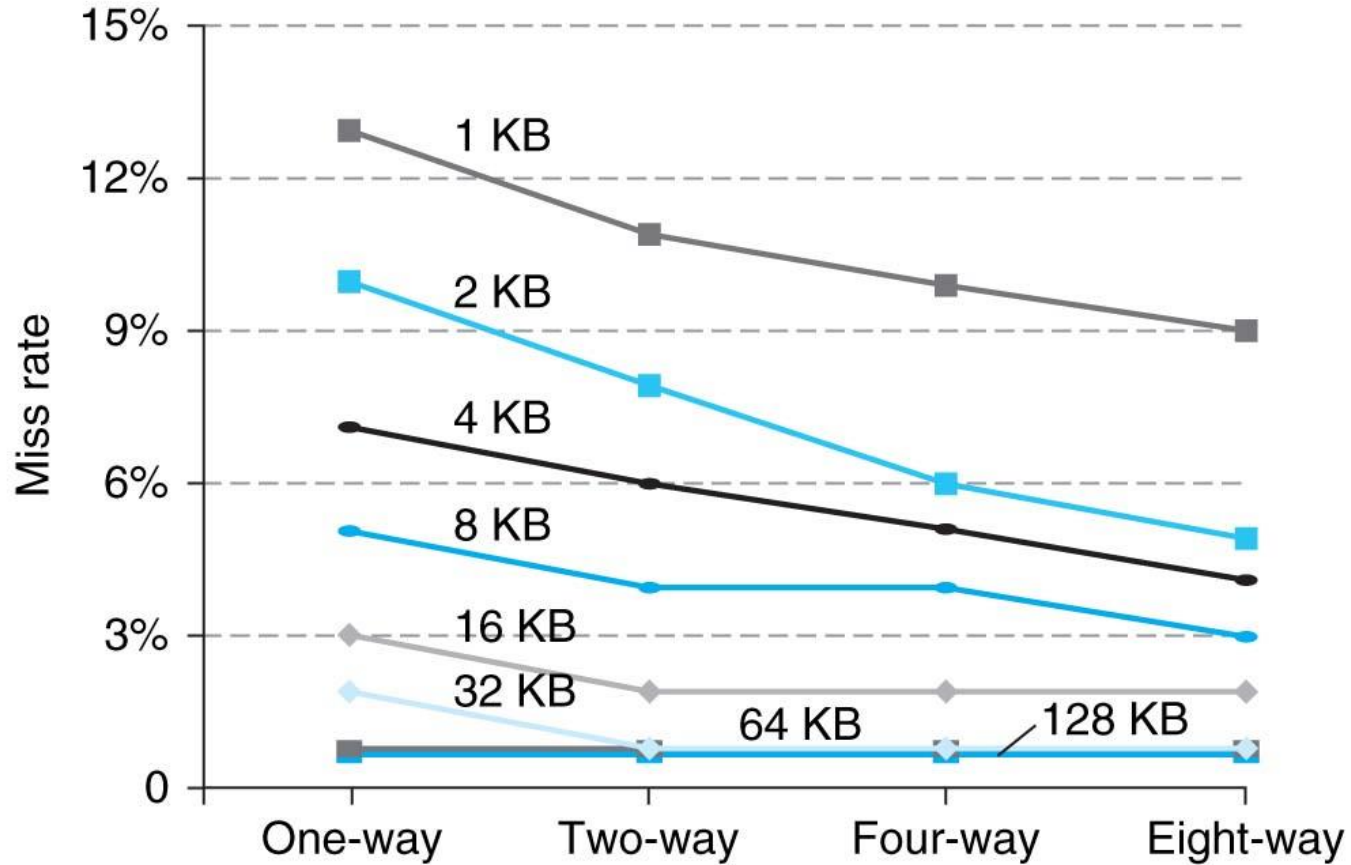
# Cache Replacement Policies

- Random Replacement
  - A cache block is randomly selected to 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 SA cache, one bit per set → set to 1 when a block is referenced; reset the other way's bit to 0; always replace the block with bit=0.
  - For N-way SA cache, can be expensive to implement
- Example of a simple “Pseudo” LRU Implementation for 64-way SA cache
  - 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)



# 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)

# Reduce AMAT

- Reduce hit time
  - e.g., smaller cache, lower associativity
- Reduce miss rate
  - e.g., larger cache, higher associativity
- Reduce miss penalty
  - e.g., multiple-level cache hierarchy
- Need to balance cache parameters (Capacity, associativity, block size)

# Recall: Sources of Cache Misses (3 C's)

- *Compulsory*: cold start, first access to a block
  - Misses that would occur even with infinite cache
  - Can be reduced by increasing block size
- *Capacity*: cache is too small to hold all data needed by the program
  - Misses that would occur even under perfect replacement policy
  - Can be reduced by increasing cache capacity
- *Conflict*: collisions due to multiple memory addresses mapped to same cache set
  - Recall the ping-pong cache example
  - Can be reduced by increasing associativity and/or increasing cache capacity

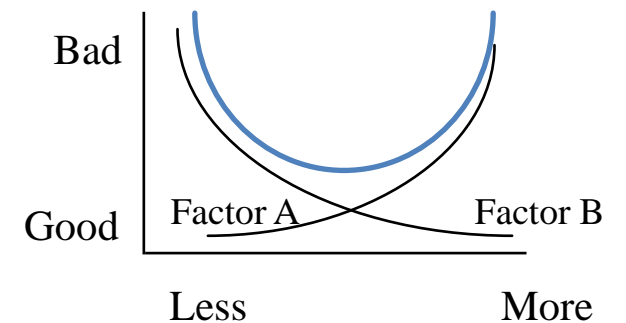
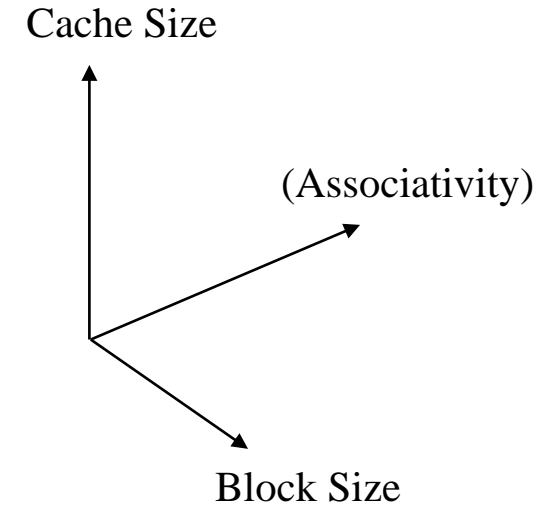
# Effect of Cache Parameters on Performance

- Larger cache size
  - + reduces capacity and conflict misses
  - Increases hit time
- Higher associativity
  - + reduces conflict misses, and the overall miss rate
  - increases hit time
- Larger block size
  - + reduces compulsory misses
  - increases conflict misses and miss penalty

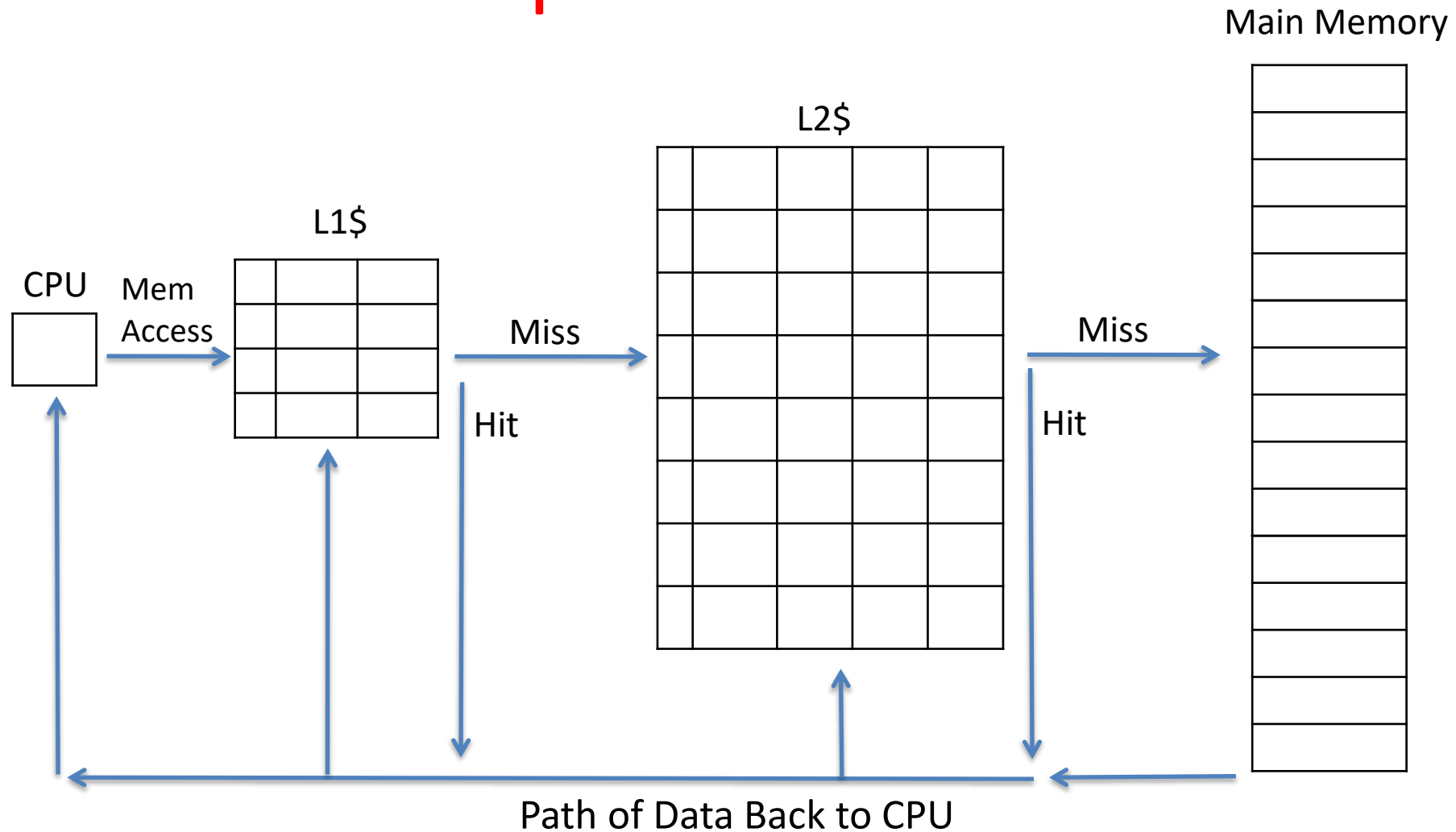
# Improving Cache Performance

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- Reduce hit time
  - e.g., smaller cache, lower associativity
- Reduce miss rate
  - e.g., larger cache, higher associativity
- Reduce miss penalty
  - e.g., multiple-level cache hierarchy
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost



# Multiple Cache Levels



L1 cache size < L2 cache size << memory size

# Local vs. Global Miss Rates

- *Local miss rate* – the fraction of references to one level of a cache that miss
  - $L2 \text{ Local Miss Rate} = L2 \text{ Misses} / L1 \text{ Misses}$
- *Global miss rate* – the fraction of references that miss in all levels of caches and must go to memory
  - $\text{Global Miss rate} = L2 \text{ Misses} / \text{Total Accesses}$
  - $= (L2 \text{ Misses} / L1 \text{ Misses}) \times (L1 \text{ Misses} / \text{Total Accesses})$
  - $= L2 \text{ Local Miss Rate} \times L1 \text{ Local Miss Rate}$
- With L1 cache only:
  - $\text{AMAT} = \text{Hit Time} + \text{Miss rate} \times \text{Miss penalty}$
- With 2-level cache hierarchy (L1+L2):
  - $L1 \text{ Miss Penalty} = L2 \text{ AMAT}; L2 \text{ Miss Penalty} = \text{Memory access time}$
  - $\text{AMAT} = L1 \text{ Hit Time} + L1 \text{ Local Miss rate} \times (L2 \text{ Hit Time} + L2 \text{ Local Miss rate} \times L2 \text{ Miss penalty})$
- With 3-level cache hierarchy (L1+L2+L3):
  - $L1 \text{ Miss Penalty} = L2 \text{ AMAT}; L2 \text{ Miss Penalty} = L3 \text{ AMAT}; L3 \text{ Miss Penalty} = \text{Memory access time}$
  - $\text{AMAT} = L1 \text{ Hit Time} + L1 \text{ Local Miss rate} \times (L2 \text{ Hit Time} + L2 \text{ Local Miss rate} \times (L3 \text{ Hit Time} + L3 \text{ Local Miss rate} \times L3 \text{ Miss penalty}))$



# AMAT Example

- L1 Hit Time: 1 cycle, L1 Miss Rate: 2%
- L2 Hit Time: 5 cycles, L2 Miss Rate: 5%
- Main Memory access time: 100 cycles
- No L2 Cache:
  - $AMAT = 1 + .02 * 100 = 3$
- With L2 Cache:
  - $AMAT = 1 + .02 * (5 + .05 * 100) = 1.2$

# Multilevel Cache Considerations

- Different design considerations for L1 Cache and LLC (Last Level Cache)
  - L1 Cache design should focus on **fast access**: minimize hit time to achieve shorter clock cycle, e.g., with smaller size, lower associativity; miss penalty is small thanks to L2 and lower caches, so higher miss rate is OK
  - LLC design should focus on **low miss rate**: miss penalty due to main memory access is very large, e.g., with larger size, higher associativity
- c.f., Slide “[Associativity in the Memory Hierarchy](#)”

# Summary

- Cache – copy of data in lower level of memory hierarchy
- Principle of locality:
  - Program likely to access a relatively small range of memory addresses at any instant of time.
    - Temporal locality vs. spatial locality
- Cache organizations:
  - Direct Mapped: 1 block per set
  - N-way Set Associative: N blocks per set, N possible places in cache to hold a given memory block
  - Fully Associative: all blocks in 1 set
- Increasing associativity helps to reduce miss rate, but increases runtime overhead
- Calculation of AMAT
- Three major categories of cache misses:
  - Compulsory Misses; Conflict Misses; Capacity Misses
- Multi-level caches
  - Optimize 1<sup>st</sup> level to minimize hit time
  - Optimize last level to minimize miss rate
- Lots of cache parameters (large design space)
  - Block size, cache size, associativity, etc.