

# **CSE 31**

# **Computer Organization**

**Bonus Lecture – Cache**



# Library Analogy

- ▶ Writing a report based on books on reserve
  - E.g., *History of most boring video games*
- ▶ Go to library to get reserved book and place on desk in your dorm room
- ▶ If need more, check them out and keep on desk
  - But don't return earlier books since might need them later
- ▶ You hope this collection of ~10 books on desk enough to write the report, despite 10 being only 0.00001% of books in the library
- ▶ We can do the same with memory use in a computer

# Big Idea: 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
- ▶ *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books on *History of most boring video games* since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

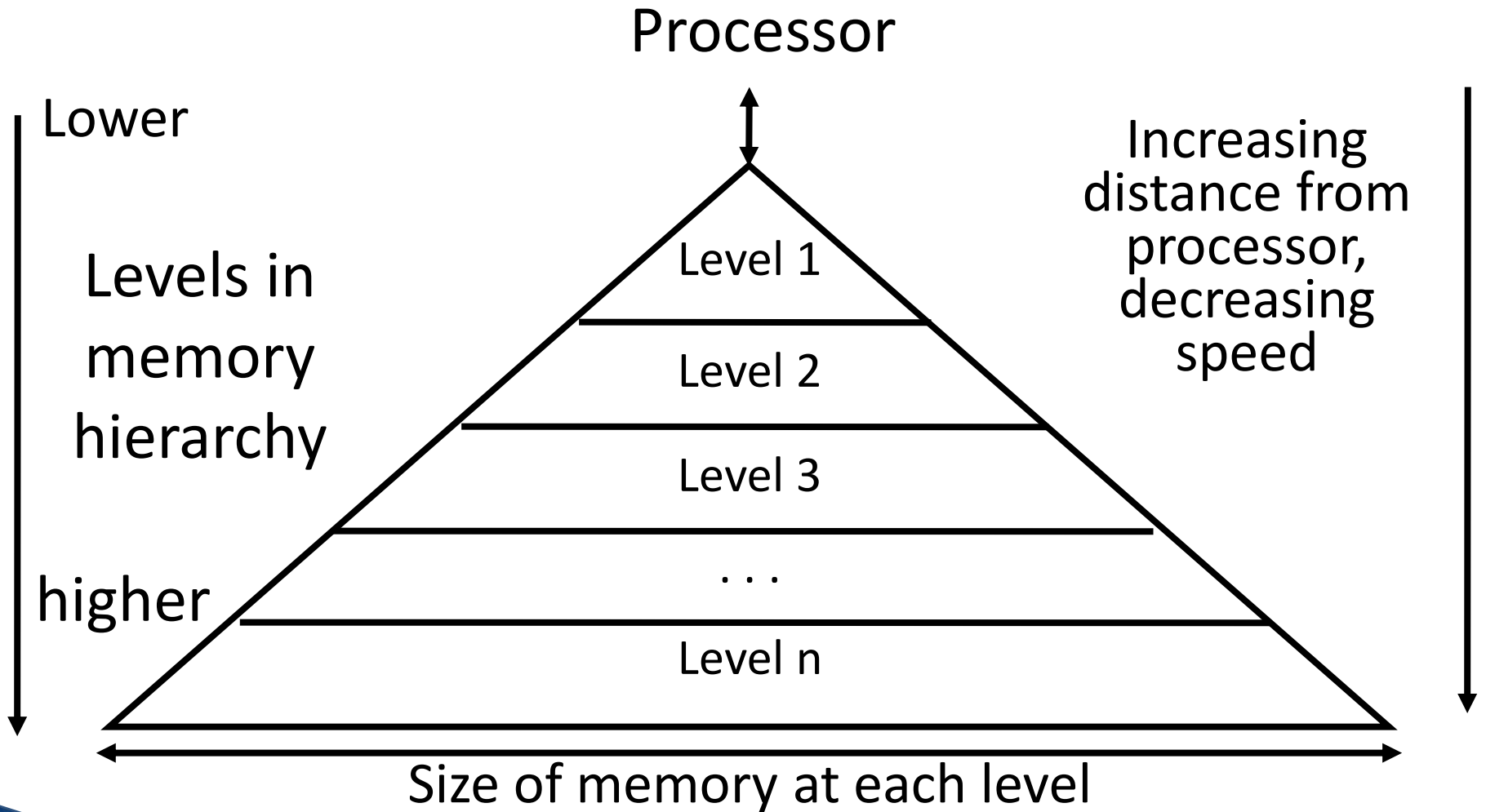
# Principle of Locality

- ▶ *Principle of Locality*: Programs access small portion of address space at any instant of time
- ▶ What **program** structures lead to temporal and spatial locality in code?
  - Temporal – Loops
  - Spatial – Sequential execution
- ▶ What about in **data**?
  - Temporal – Loop variable
  - Spatial – Arrays

# How does hardware exploit principle of locality?

- ▶ Offer a hierarchy of memories where
  - closest to processor is fastest  
(and most expensive per bit so smallest)
    - Books on your desk
  - furthest from processor is largest  
(and least expensive per bit so slowest)
    - Books in the library
- ▶ Goal is to create **illusion** of memory almost **as fast** as fastest memory and almost **as large** as biggest memory of the hierarchy

# Big Idea: Memory Hierarchy



*As we move to deeper levels the latency goes up and price per bit goes down. Why?*

# Cache Concept

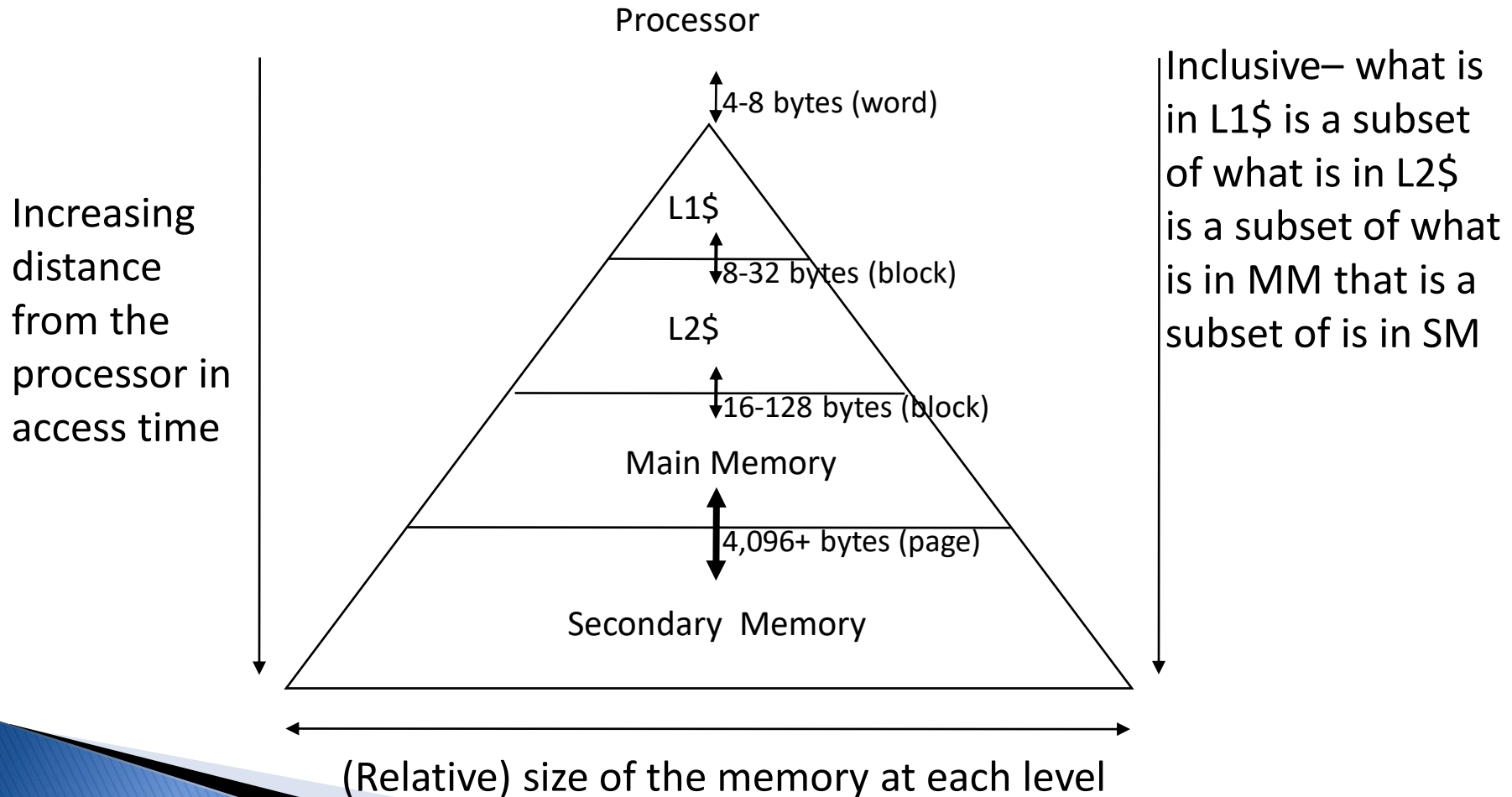
- ▶ Processor and memory speed mismatch leads us to add a new level: a memory *cache*
- ▶ Implemented with same integrated circuit processing technology as processor, integrated on-chip: faster but more expensive than DRAM memory
- ▶ *Cache is a copy of a subset of main memory*
- ▶ Modern processors have separate caches for instructions and data, as well as several levels of caches implemented in different sizes
- ▶ As a pun, often use \$ (“cash”) to abbreviate cache, e.g. D\$ = Data Cache, I\$ = Instruction Cache

# Memory Hierarchy Technologies

- ▶ Caches use SRAM (Static RAM) for speed and technology compatibility
  - Fast (typical access times of 0.5 to 2.5 ns)
  - Low density (6 transistor cells), higher power, expensive (\$1000 to \$2000 per GB in 2018)
  - Static: content will last as long as power is on
- ▶ Main memory uses DRAM (Dynamic RAM) for size (density)
  - Slower (typical access times of 50 to 70 ns)
  - High density (1 capacitor and 1 transistor cells), lower power, cheaper (~\$10 per GB in 2017)
  - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
    - Consumes 1% to 2% of the active cycles of the DRAM



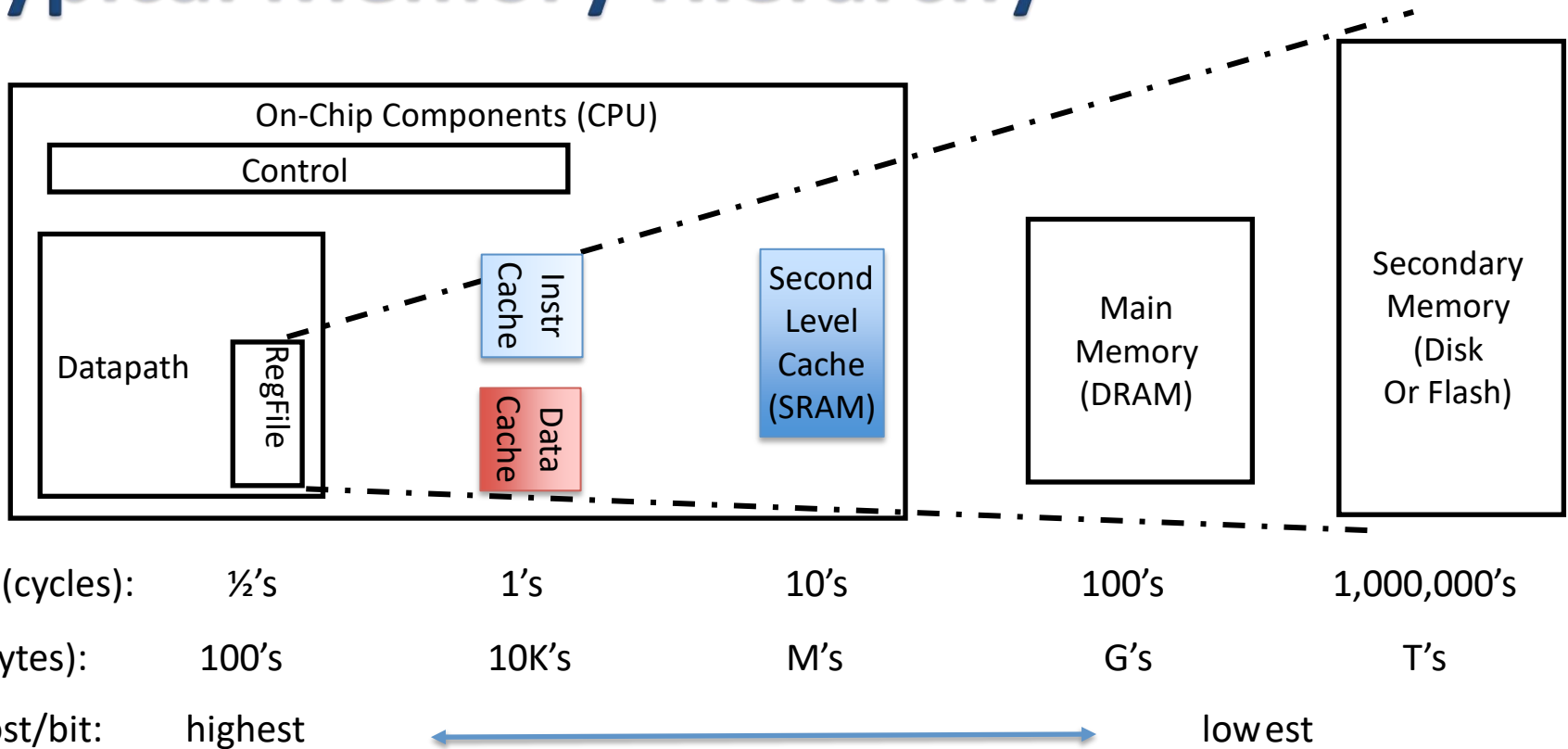
# Characteristics of the Memory Hierarchy



# How is the Hierarchy Managed?

- ▶ registers  $\leftarrow$  memory
  - By compiler (or assembly level programmer)
- ▶ cache  $\leftarrow$  main memory
  - By the cache controller hardware
- ▶ main memory  $\leftarrow$  disks (secondary storage)
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware (TLB)
  - By the programmer (files)
  - (Talk about it later in CSE140)

# Typical Memory Hierarchy

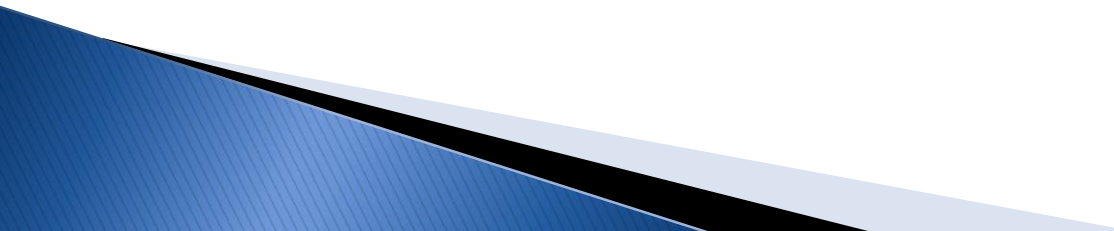


- ▶ **Principle of locality + memory hierarchy** presents programmer with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology

# So far ...

- ▶ Wanted: effect of a large, cheap, fast memory
- ▶ Approach: Memory Hierarchy
  - Successively lower levels contain “most used” data from next higher level
  - Exploits *temporal & spatial locality*
- ▶ Memory hierarchy follows 2 Design Principles: Smaller is faster and Do common case fast

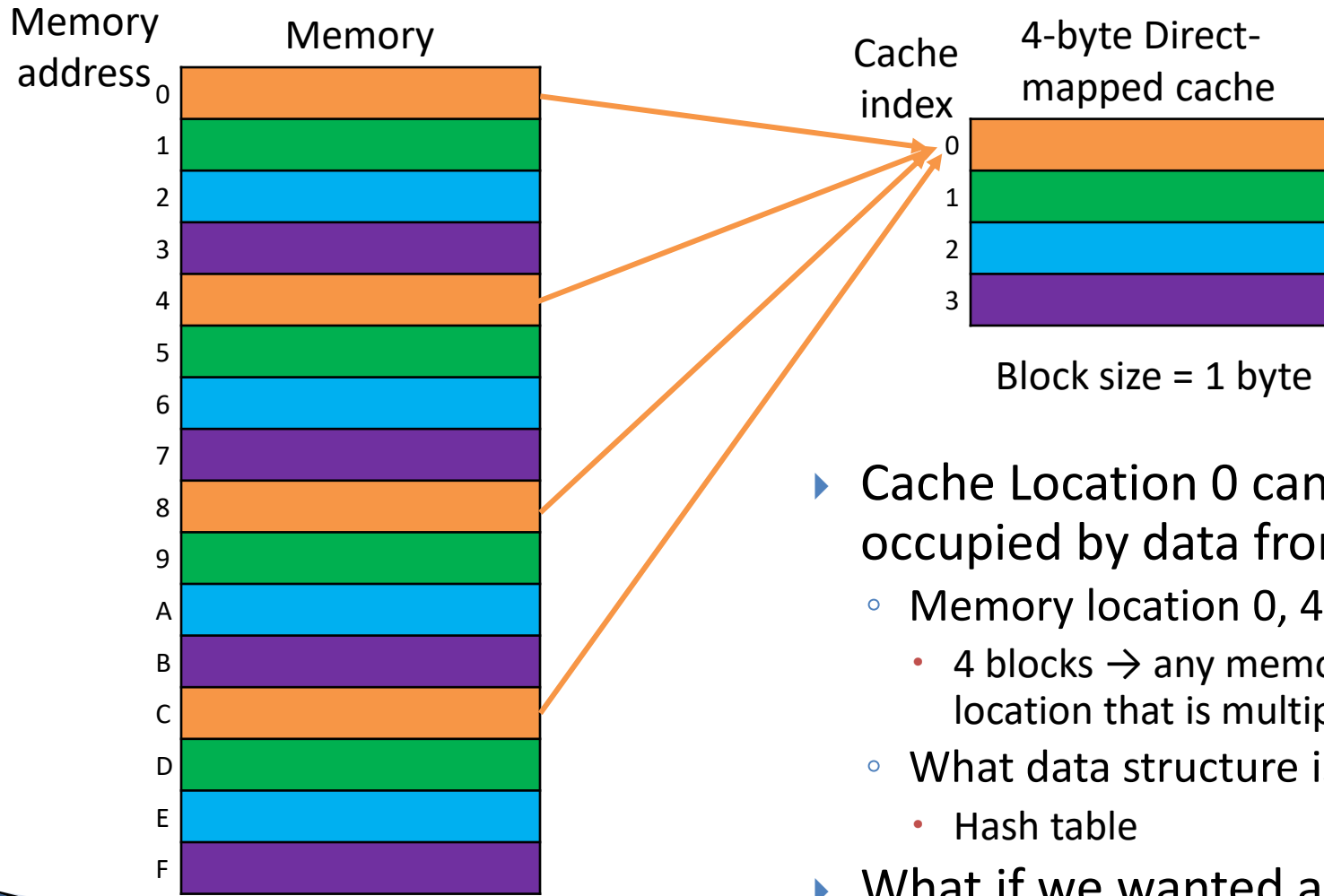
# Cache Design Questions

- ▶ How best to organize the memory blocks of the cache?
  - ▶ To which block of the cache does a given main memory address map?
    - Since the cache is a subset of memory, multiple memory addresses can be mapped to the same cache location
  - ▶ How do we know which blocks of main memory currently have a copy in cache?
  - ▶ How do we find these copies quickly?
- 

# Direct-Mapped Cache (1/4)

- ▶ Block is the unit of transfer between cache and memory
  - The size of a block must be pre-defined, and the memory will be framed in terms of blocks.
- ▶ 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

# Direct-Mapped Cache (2/4)



- ▶ Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
    - 4 blocks → any memory location that is multiple of 4
  - What data structure is it?
    - Hash table
- ▶ What if we wanted a block to be bigger than one byte?

# Direct-Mapped Cache (3/4)

Memory  
address

Memory	
0	0 1
2	2 3
4	4 5
6	6 7
8	8 9
A	...
C	
E	
10	
12	
14	
16	
18	
1A	
1C	1D
1E	

Cache  
index

8-byte Direct-  
mapped cache

0	
1	
2	
3	

Block size = 2 bytes

- ▶ When we ask for a byte, the system finds out the **right block** and **loads it all!**
  - How does it know the right block?
  - How do we select the byte?
    - Ex. Mem address **11101 (1D)**?
  - How does it know WHICH colored block it originated from?
  - What do you do at baggage claim?
    - Add a tag to each block!



# Direct-Mapped Cache (4/4)

Memory  
address

Memory

0	0	1
2	2	3
4	4	5
6	6	7
8	8	9
A	...	...
C		
F		
10		
12		
14		
16		
18		
1A		
1C		
1E		

Cache  
index

8-byte Direct-mapped  
cache w/ Tag

0	8		
1	2		
2	14		
3	1E		

Tag

Data

Block size = 2 bytes

- ▶ What should go in the tag?
  - Do we need the entire address?
    - What do all these tags have in common?

# Direct-Mapped Cache (4/4)

Memory  
address

Memory

0000 0000	0	1
0000 0010	2	3
0000 0100	4	5
0000 0110	6	7
0000 1000	8	9
0000 1010	...	...
0000 1100		
0000 1110		
0001 0000		
0001 0010		
0001 0100		
0001 0110		
0001 1000		
0001 1010		
0001 1100		
0001 1110		

0

1

2

3

Cache  
index

8-byte Direct-mapped  
cache w/ Tag

0	<del>8</del> 1		
1	<del>2</del> 0		
2	<del>14</del> 2		
3	<del>1E</del> 3		

Tag

Data

Block size = 2 bytes

- ▶ What should go in the tag?
  - Do we need the entire address?
    - What do all these tags have in common?
- ▶ Why not count by group #?
  - It's useful to draw memory with the same width as the block size

# Issues with Direct-Mapped

- ▶ Since multiple memory addresses map to same cache index, how can we tell which one is in there?
- ▶ If we have a block size  $> 1$  byte, how do we know where to find the data within the block?
  - Answer: divide memory address into three fields

t t t t t t t t t t t t t t t t t t	i i i i i i i i i i	o o o o
-------------------------------------	---------------------	---------

Tag to check if it has  
the correct block  
(Zip code)

Index to  
select  
block  
(Floor #)

Byte offset  
within block  
(Appt. #)

# Direct-Mapped Cache Terminology

- ▶ All fields are read as unsigned integers
- ▶ **Index**
  - specifies the cache index (which “row” or block of the cache we should look in)
- ▶ **Offset**
  - once we’ve found correct block, specifies which byte within the block we want
- ▶ **Tag**
  - the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

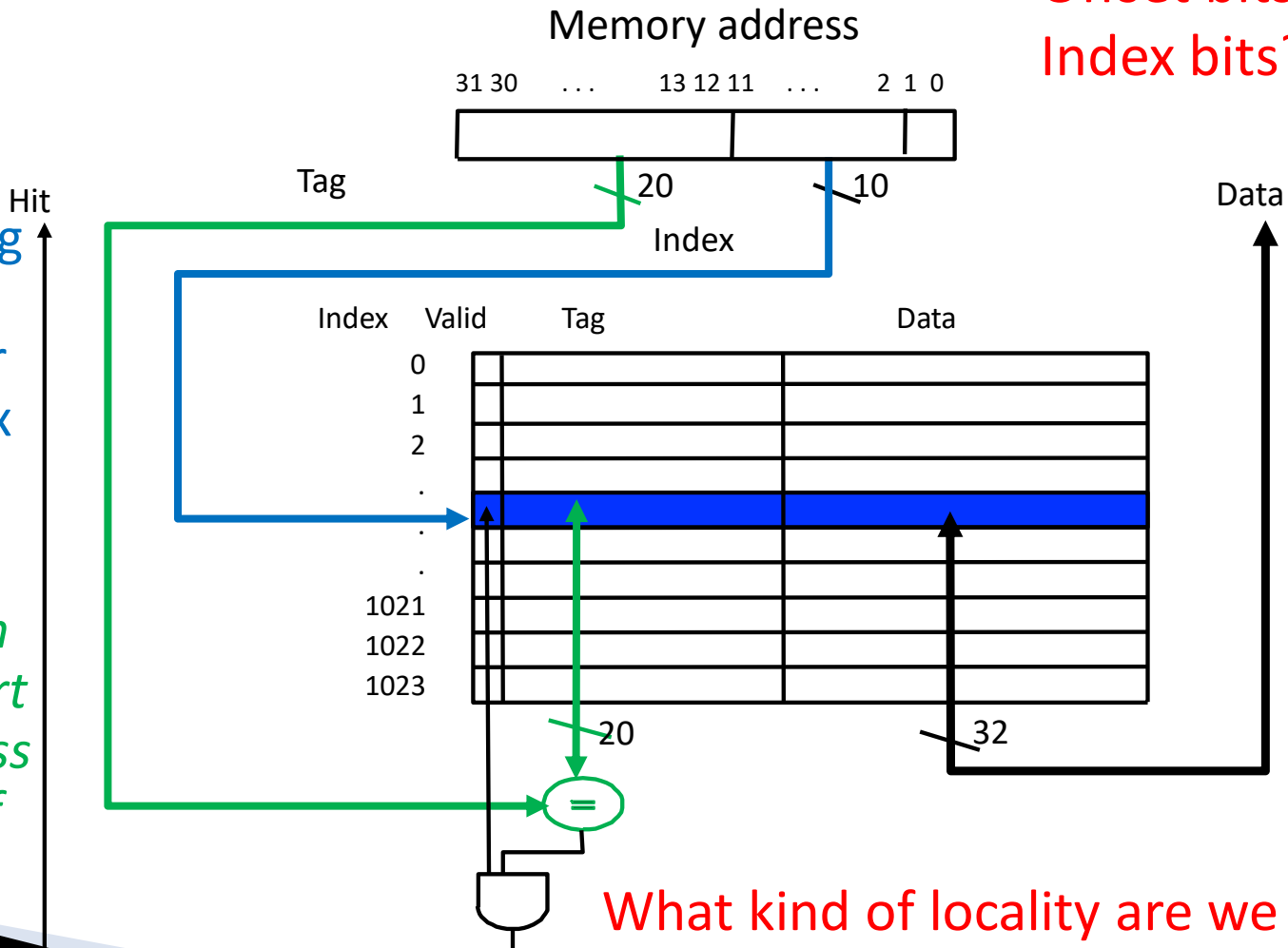
# Direct Mapped Cache Example

One word blocks, cache size = 1K words (or 4KB)

Offset bits? 2  
Index bits? 10

Valid bit  
ensures  
something  
useful in  
cache for  
this index

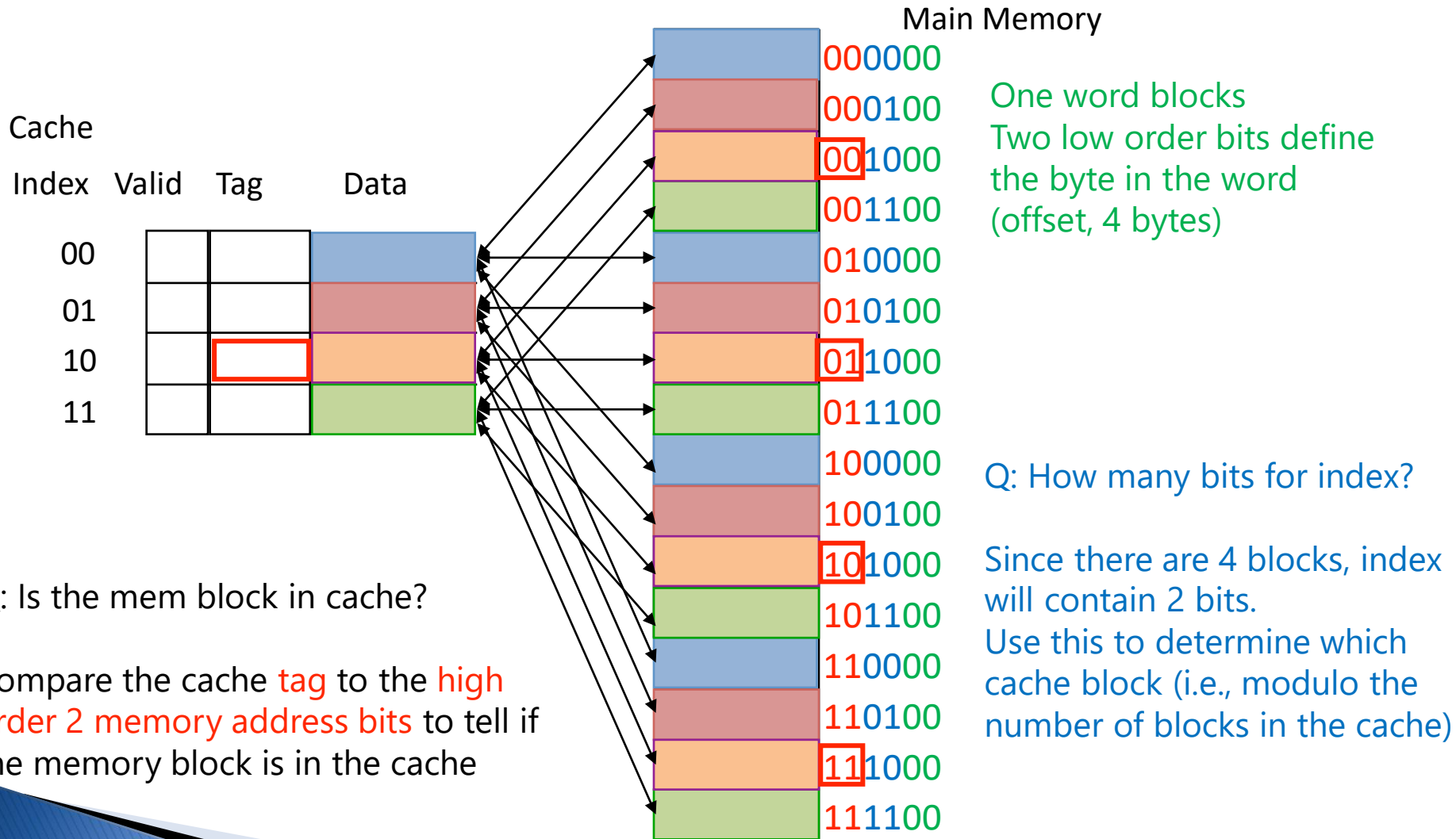
Compare  
Tag with  
"tag" part  
of Address  
to see if  
it's a Hit



*Read  
data  
from  
cache  
instead  
of  
memory  
if a Hit*

What kind of locality are we taking advantage of?

# Caching: A Simple First Example



# Caching Terminology

- ▶ When reading memory, 3 things can happen:
  - **cache hit:**
    - cache block is valid and contains proper address, so read desired word
  - **cache miss:**
    - nothing in cache at appropriate block, so fetch from memory
  - **cache miss, block replacement:**
    - wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

# Direct Mapped Cache

Consider the main memory word reference string (only the tag and index bits of the address are shown. The 2-bits for offset are not shown)

Start with an empty 4-word cache -  
all blocks initially marked as not valid

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

Address 0




# Direct Mapped Cache

Consider the main memory word reference string (only the tag and index bits of the address are shown. The 2-bits for offset are not shown)

Start with an empty 4-word 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

00	Mem(0)

- 1 requests, 1 miss

# Direct Mapped Cache

Consider the main memory word reference string (only the tag and index bits of the address are shown. The 2-bits for offset are not shown)

Start with an empty 4-word 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

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

<del>00</del>	<del>Mem(0)</del>
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
<del>00</del>	<del>Mem(3)</del>

11

15

- 8 requests, 6 misses

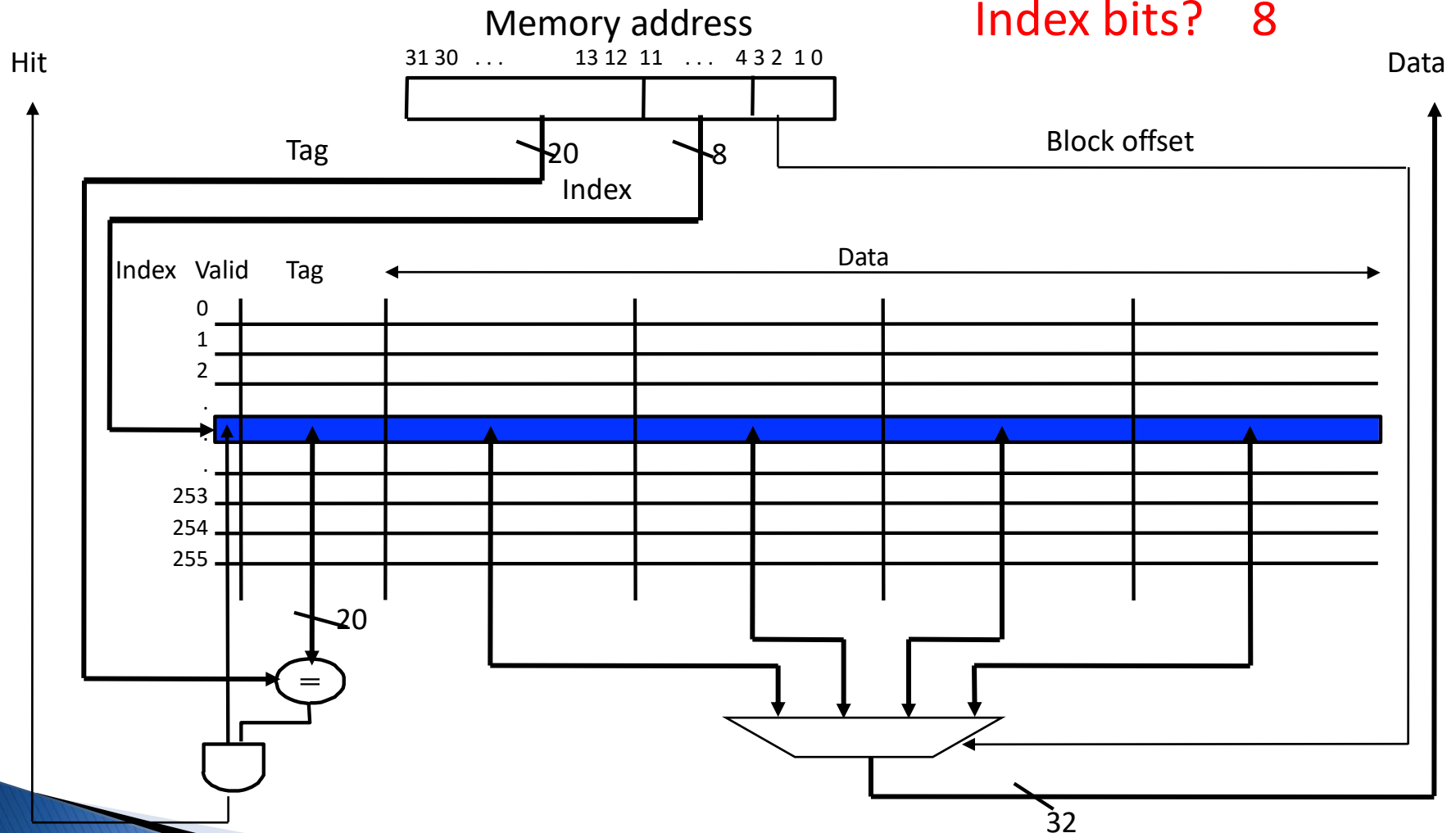
How can we reduce the number of misses?

# Multiword Block Direct Mapped Cache

Four words/block, cache size = 1K words

Offset bits? 4

Index bits? 8



What kind of locality are we taking advantage of?

# Taking Advantage of Spatial Locality

Let cache block hold more than one word

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

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

01      5      4 miss      4

<del>00</del>	<del>Mem(1)</del>	<del>Mem(0)</del>
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

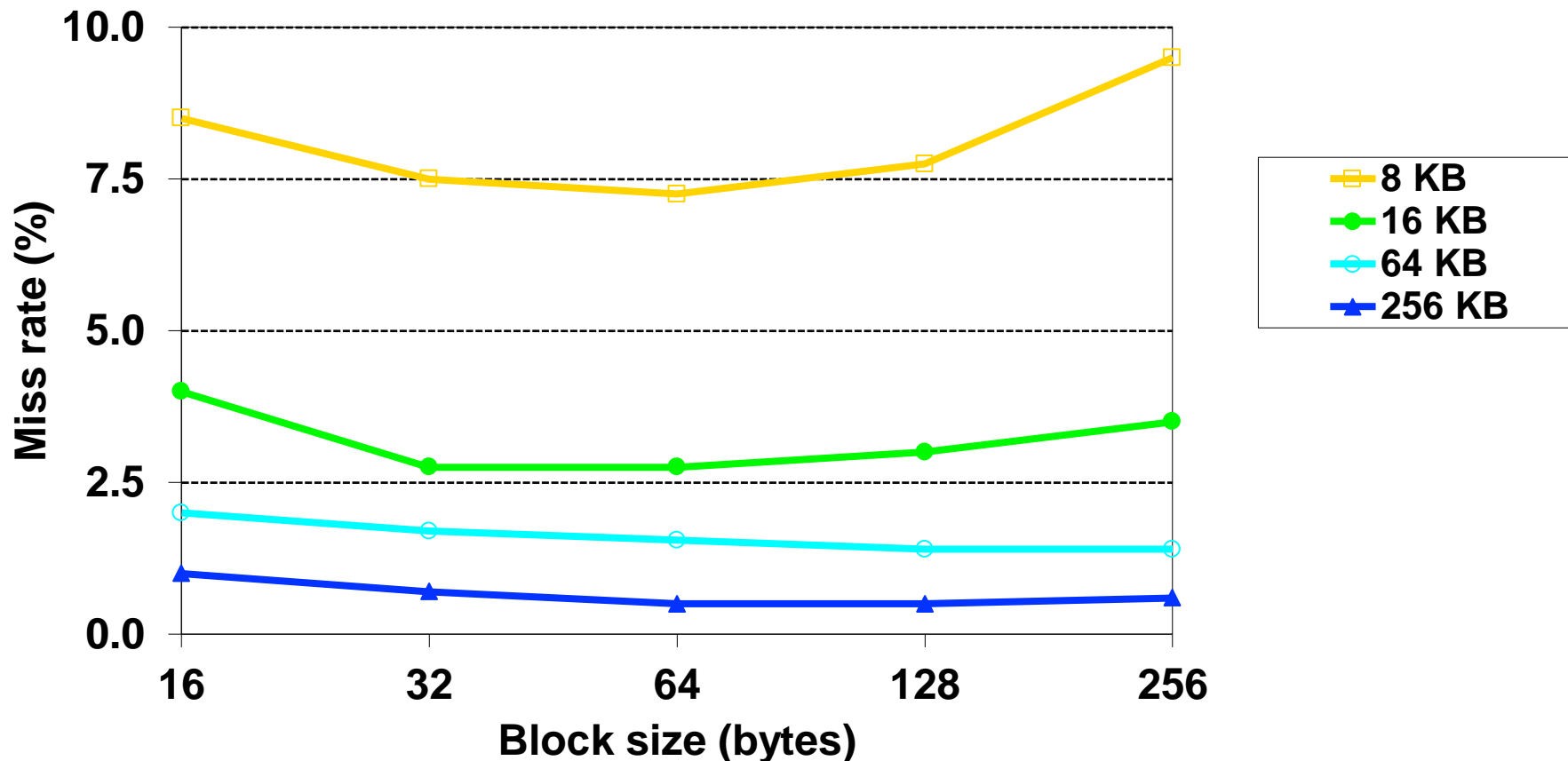
11

01	Mem(5)	Mem(4)
<del>00</del>	<del>Mem(3)</del>	<del>Mem(2)</del>

15      14

- 8 requests, 4 misses, with same size of cache!

# Miss Rate vs Block Size vs Cache Size



Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

# Block Size Tradeoff (1/2)

- ▶ Benefits of Larger Block Size
  - **Spatial Locality:** if we access a given word, we're likely to access other nearby words soon
  - Very applicable with Stored-Program Concept: if we execute a given instruction, it's likely that we'll execute the next few as well
  - Works nicely in sequential array accesses too

# Block Size Tradeoff (2/2)

- ▶ Drawbacks of Larger Block Size
  - Larger block size means **larger miss penalty**
    - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up

# Cache Performance

- ▶ Hit Time
  - time to find and retrieve data from current level cache
- ▶ Miss Penalty
  - average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- ▶ Hit Rate
  - % of requests that are found in current level cache
- ▶ Miss Rate
  - $1 - \text{Hit Rate}$
- ▶ In general, minimize  
**Average Memory Access Time (AMAT)**  
$$= \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$$



# Average Memory Access Time (AMAT)

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

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

- ▶ What is the AMAT for a processor with a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

$$1 + 0.02 \times 50 = 2 \text{ clock cycles}$$

$$\text{Or } 2 \times 200 = 400 \text{ psecs}$$

- ▶ Potential impact of much larger cache on AMAT?

- 1) Lower Miss rate

- 2) Longer Access time (Hit time): smaller is faster

At some point, increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance

# Extreme Example: One Big Block

Valid Bit

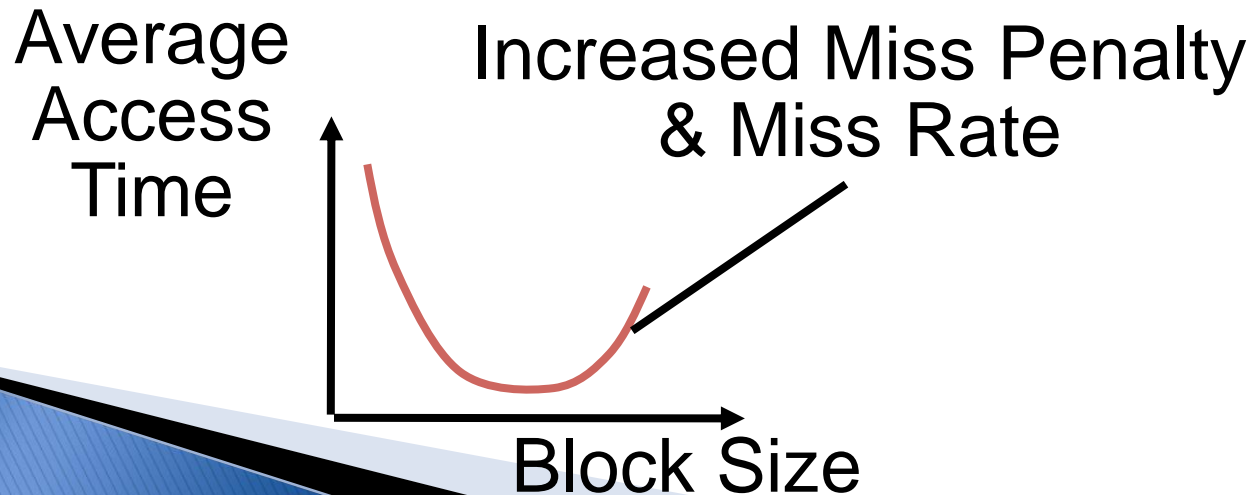
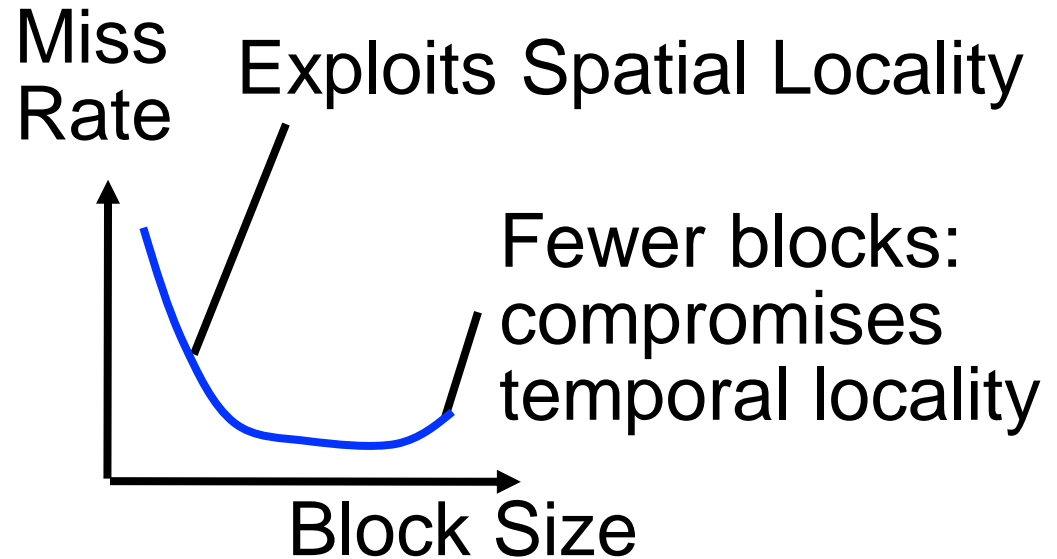
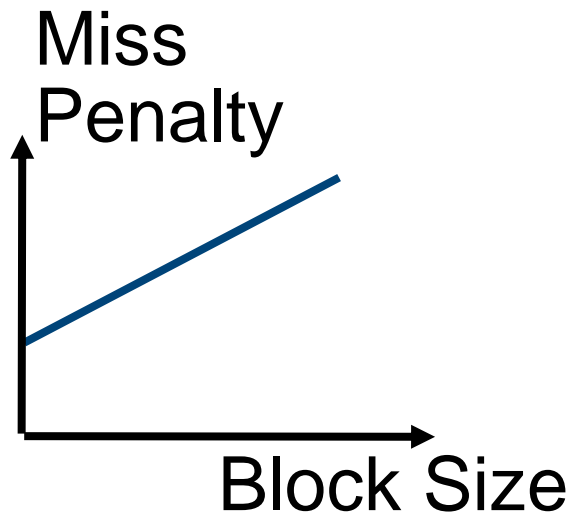
Tag

Cache Data



- ▶ Cache Size = 4 bytes      Block Size = 4 bytes
  - Only **ONE** entry (row) in the cache!
- ▶ If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- ▶ The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: **Ping Pong Effect**

# Block Size Tradeoff Conclusions



# What about Memory Write?

- ▶ When shall we update the memory when write?
  - Write-through
    - update the word in cache block and corresponding word in memory
  - Write-back
    - update word in cache block
    - allow memory word to be “stale”
    - add ‘dirty’ bit to each block indicating that memory needs to be updated before block is replaced
- ▶ Performance trade-offs?

# Types of Cache Misses (1/3)

- ▶ “Three Cs” Model of Misses
- ▶ 1<sup>st</sup> C: **Compulsory Misses**
  - occur when a program is first started
  - cache does not contain any of that program’s data yet, so misses are bound to occur (valid bit = 0)
  - can’t be avoided easily, so won’t focus on these in this course

# Types of Cache Misses (2/3)

## ▶ 2<sup>nd</sup> C: Conflict Misses

- miss that occurs because two distinct memory addresses map to the same cache location
- two blocks (which happen to map to the same location) can keep overwriting each other
- big problem in direct-mapped caches
- how do we lessen the effect of these?

## ▶ Dealing with Conflict Misses

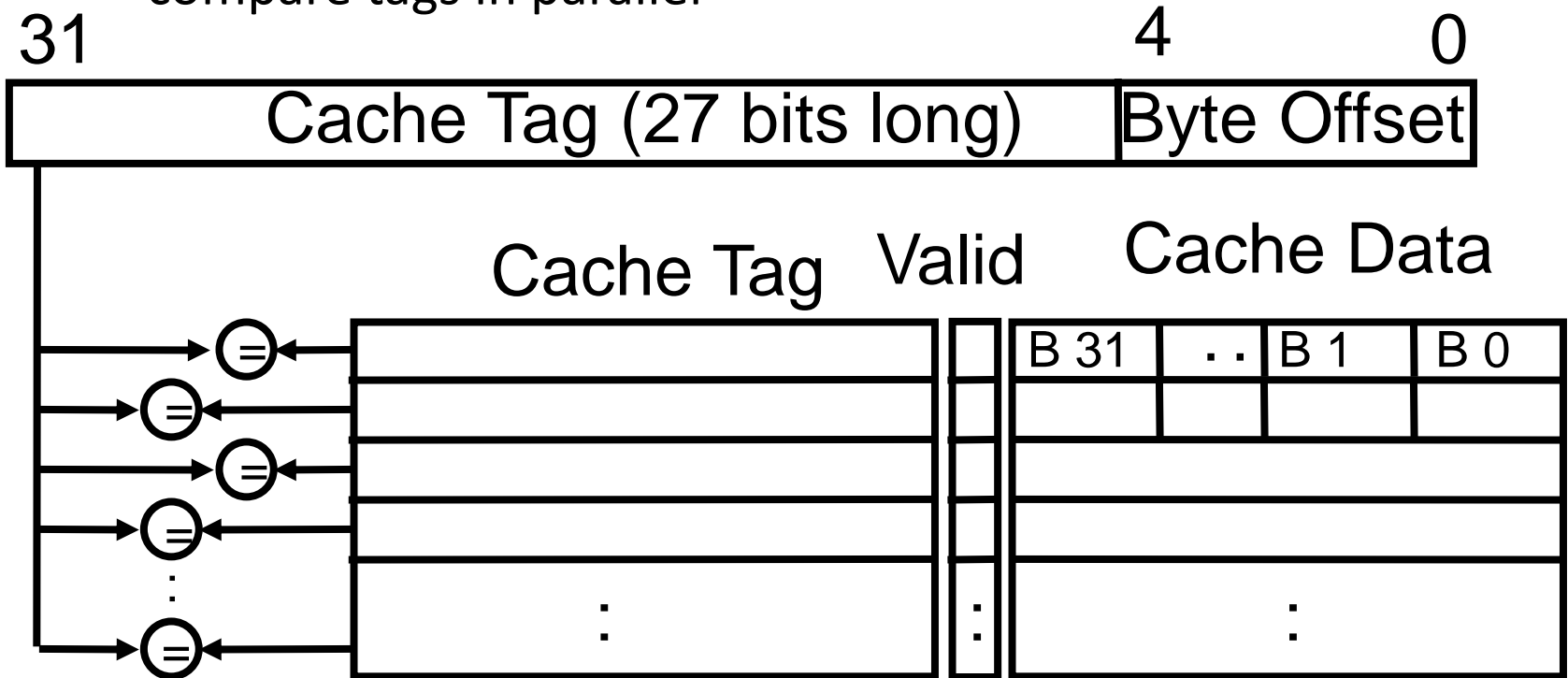
- Solution 1: Make the cache size bigger
  - Fails at some point
- Solution 2: Multiple distinct blocks can fit in the same cache Index
  - How???

# Fully Associative Cache (1/3)

- ▶ Memory address fields:
  - **Tag**: same as before
  - **Offset**: same as before
  - **Index**: non-existent
- ▶ What does this mean?
  - no “rows”: any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

## Fully Associative Cache (2/3)

- ▶ Fully Associative Cache (e.g., 32 B block)
  - compare tags in parallel





# Fully Associative Cache (3/3)

- ▶ Benefit of Fully Assoc. Cache
  - No Conflict Misses (since data can go anywhere)
- ▶ Drawbacks of Fully Assoc. Cache
  - Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: infeasibly high cost
  - Not practical for large cache!

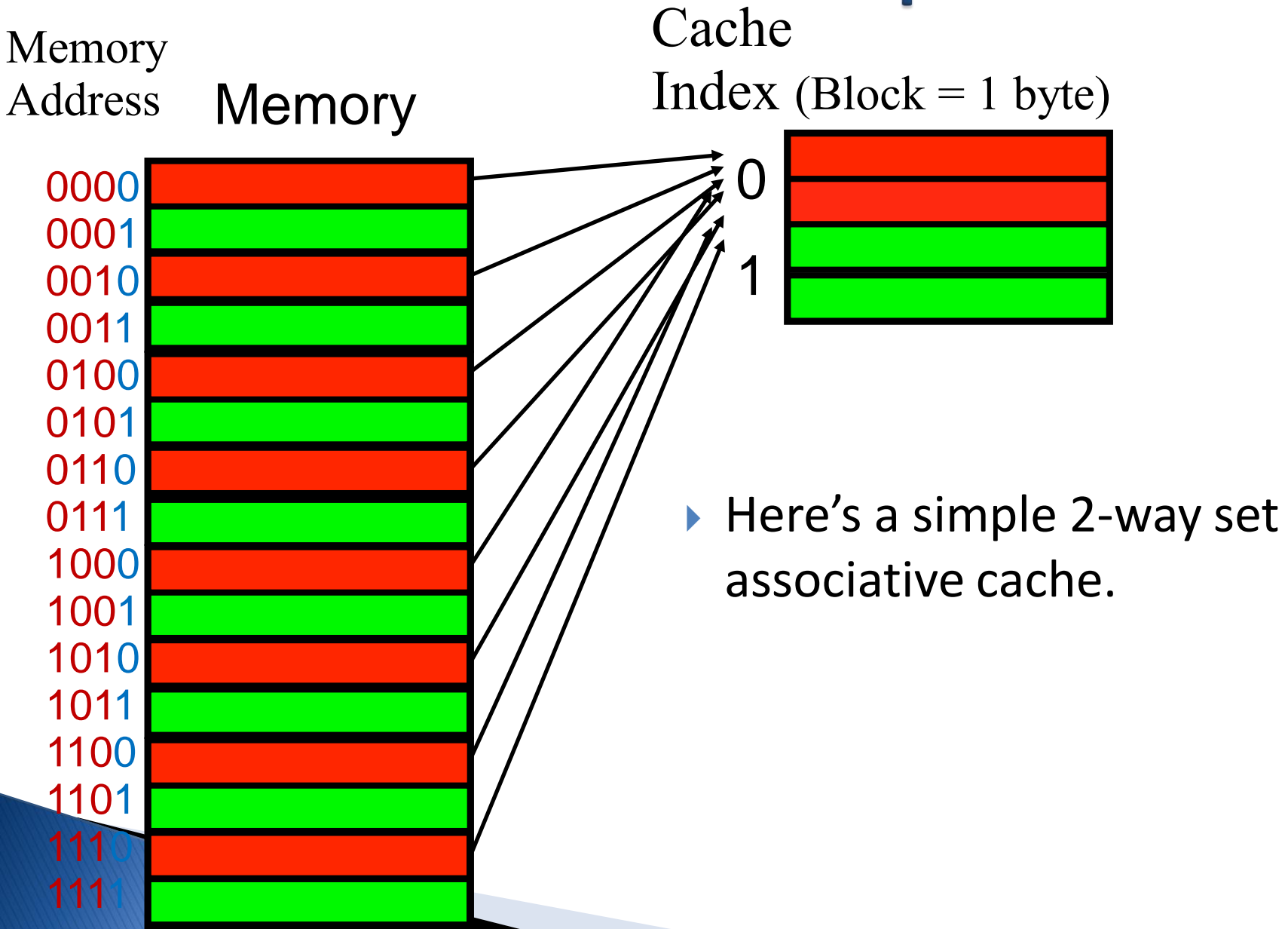
# Types of Cache Misses (3/3)

- ▶ 3<sup>rd</sup> C: Capacity Misses
  - miss that occurs because the cache has a limited size
  - miss that would not occur if we increase the size of the cache
  - sketchy definition, so just get the general idea
- ▶ This is the primary type of miss for Fully Associative caches.

# N-Way Set Associative Cache (1/3)

- ▶ Memory address fields:
  - **Tag**: same as before
  - **Offset**: same as before
  - **Index**: points us to the correct “row” (called a **SET** in this case)
- ▶ So what’s the difference?
  - each set contains multiple blocks
  - once we’ve found correct set, must compare with all tags in that set to find our data
  - Hybrid of direct-mapped and fully associative

# Associative Cache Example

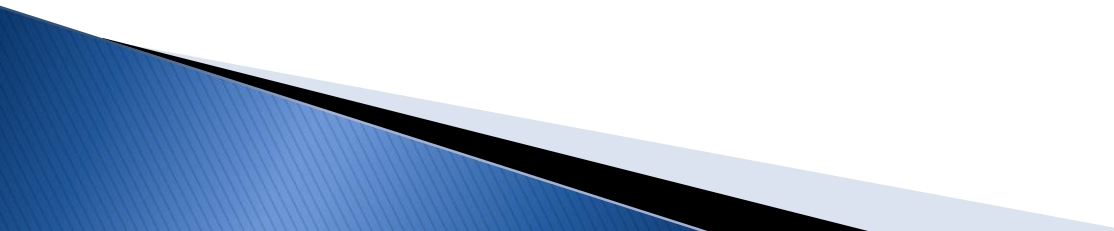


# N-Way Set Associative Cache (2/3)

## ▶ Basic Idea

- cache is direct-mapped w/respect to sets
- each set is fully associative with N blocks in it

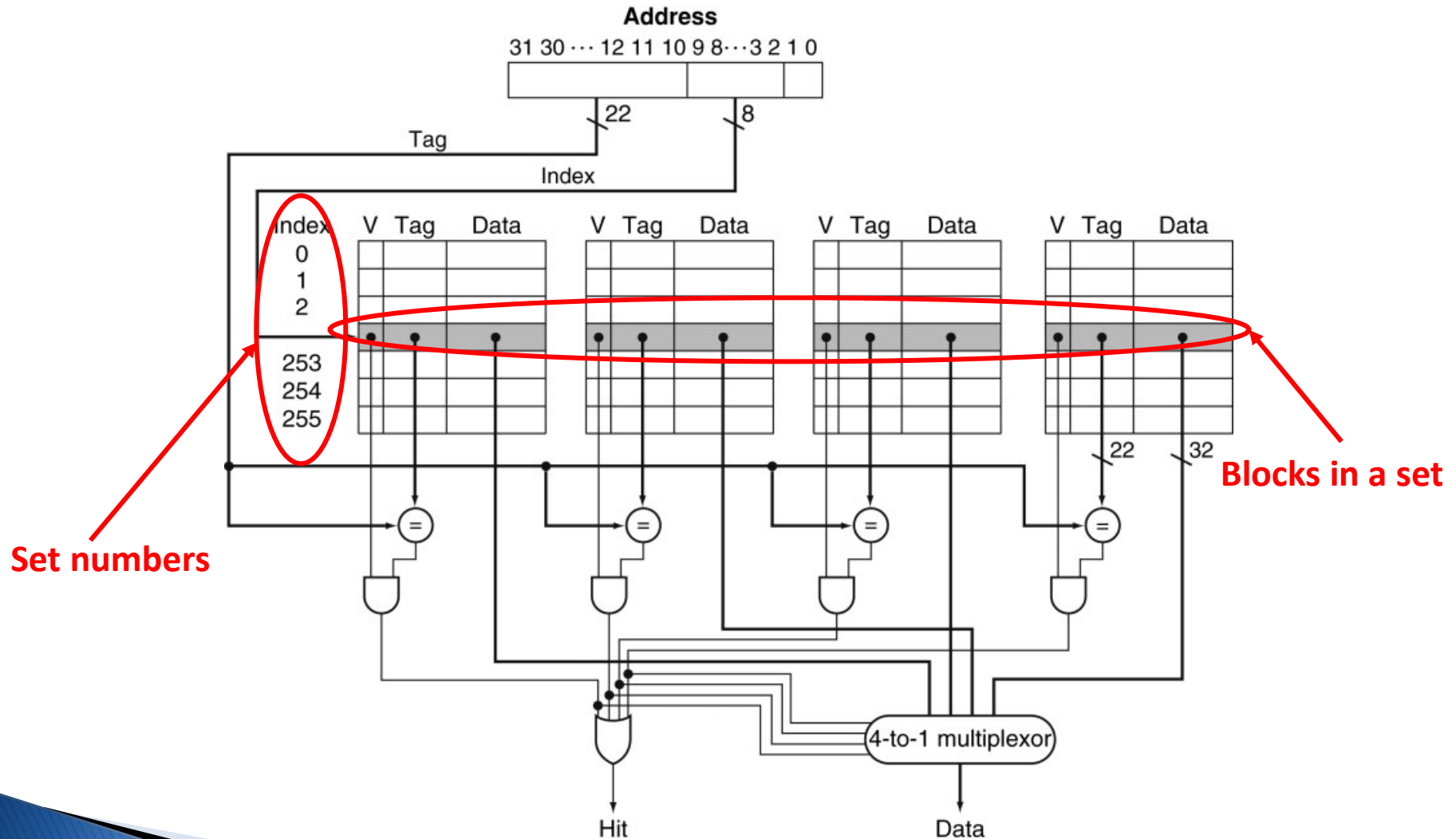
## ▶ Given memory address:

- Find correct set using Index value.
  - Compare Tag with all Tag values in the determined set.
  - If a match occurs, hit!, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the block.
- 

# N-Way Set Associative Cache (3/3)

- ▶ What's so great about this?
  - even a 2-way set assoc. cache avoids a lot of conflict misses
  - hardware cost isn't that bad: only need  $N$  comparators
- ▶ In fact, for a cache with  $M$  blocks,
  - it's **Direct-Mapped** if it's 1-way set assoc.
  - it's **Fully Assoc** if it's  $M$ -way set assoc.
  - so, these two are just special cases of the more general set associative design

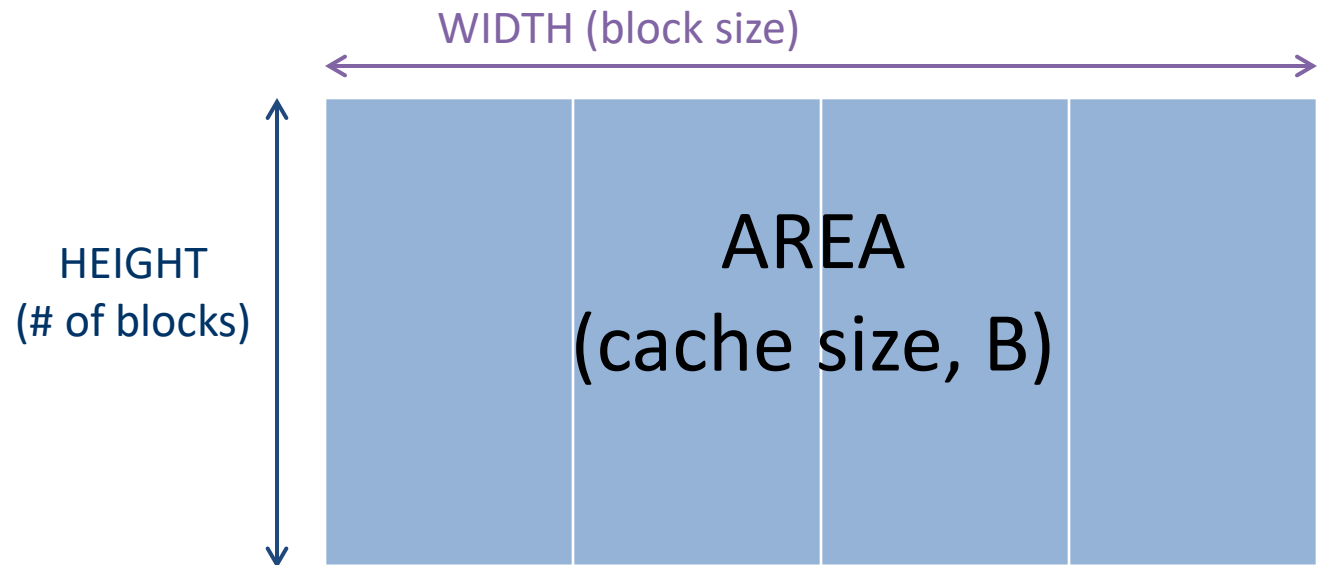
# 4-Way Set Associative Cache Circuit





AREA (cache size, B)  
= HEIGHT (# of blocks) \* WIDTH (block size)

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





# Block Replacement Policy

## ▶ Direct-Mapped Cache

- index completely specifies position which position a block can go in on a miss

## ▶ N-Way Set Assoc.

- index specifies a set, but block can occupy any position within the set on a miss

## ▶ Fully Associative

- block can be written into any position

## ▶ Question: if we have the choice, where should we write an incoming block?

- If there are any locations with valid bit off (empty), then usually write the new block into the first one.
- If all possible locations already have a valid block, we must pick a **replacement policy**: rule by which we determine which block gets “cached out” on a miss.

# Block Replacement Policy: LRU

- ▶ LRU (Least Recently Used)
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: **temporal locality** → recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

# Block Replacement Example

- ▶ We have a 2-way set associative cache with a four-word total capacity and one-word blocks. We perform the following word address accesses (ignore bytes for this problem):

0, 2, 0, 1, 4, 0, 2, 3, 5, 4

- ▶ How many hits and how many misses will there be for the LRU block replacement policy?

# Block Replacement Example: LRU

0: miss, bring into set 0 (blk 0)

2: miss, bring into set 0 (blk 1)

Addresses 0, 2, 0, 1, 4, 0, ...

0: hit

1: miss, bring into set 1 (blk 0)

4: miss, bring into set 0 (blk 1, replace 2)

0: hit

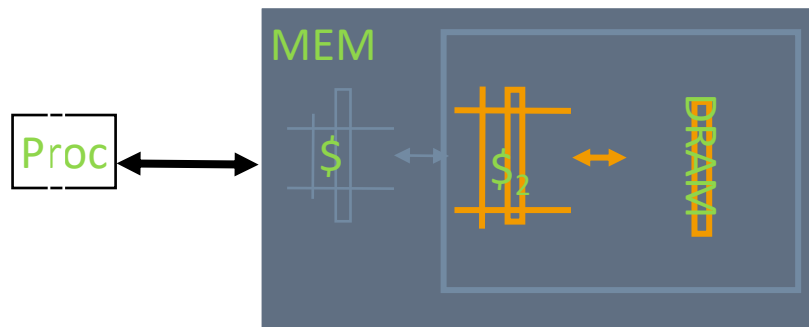
	blk 0	blk 1
set 0	0	<del>lru</del>
set 1		
set 0	<del>lru</del> 0	<del>lru</del> 2
set 1		
set 0	<del>lru</del> 0	<del>lru</del> 2
set 1		
set 0	0	<del>lru</del> 2
set 1	1	<del>lru</del>
set 0	<del>lru</del> 0	<del>lru</del> 4
set 1	1	<del>lru</del>
set 0	<b>0</b>	<del>lru</del> 4
set 1	1	<del>lru</del>

# Big Idea

- ▶ How to choose between associativity, block size, replacement & write policy?
- ▶ Design against a performance model
  - Minimize: Average Memory Access Time
    - $\text{= Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$
  - Influenced by technology & program behavior
- ▶ Create the illusion of a memory that is large, cheap, and fast - on average
- ▶ How can we improve miss penalty?

# Improving Miss Penalty

- ▶ When caches first became popular, Miss Penalty  
~ 10 processor clock cycles
- ▶ Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM  
~ 200 processor clock cycles!



**Solution: another cache between  
memory and the processor cache:  
Second Level (L2) Cache**

# Summary

- ▶ 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
  - ▶ Larger caches reduce Miss rate via Temporal and Spatial Locality, but can increase Hit time
  - ▶ Larger blocks to reduces Miss rate via Spatial Locality, but increase Miss penalty
  - ▶ AMAT helps balance Hit time, Miss rate, Miss penalty
- 