CS3350B
Computer Architecture
Winter 2015

Lecture 3.2: Exploiting Memory Hierarchy: How?

Marc Moreno Maza

www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on
Computer Organization and Design,
Patterson & Hennessy, 5th edition, 2014]

How is the Hierarchy Managed?

- registers ↔ cache memory
  - by compiler (programmer?)
- cache ↔ main memory
  - by the cache controller hardware
- main memory ↔ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

Cache Design Questions

Q1: How best to organize the memory blocks (lines) of the cache?

Q2: To which block (line) of the cache does a given main memory address **map**?

- Since the cache is a **subset** of memory, multiple memory addresses can map to the same cache location

Q3: How do we know if a block of main memory currently has a copy in cache?

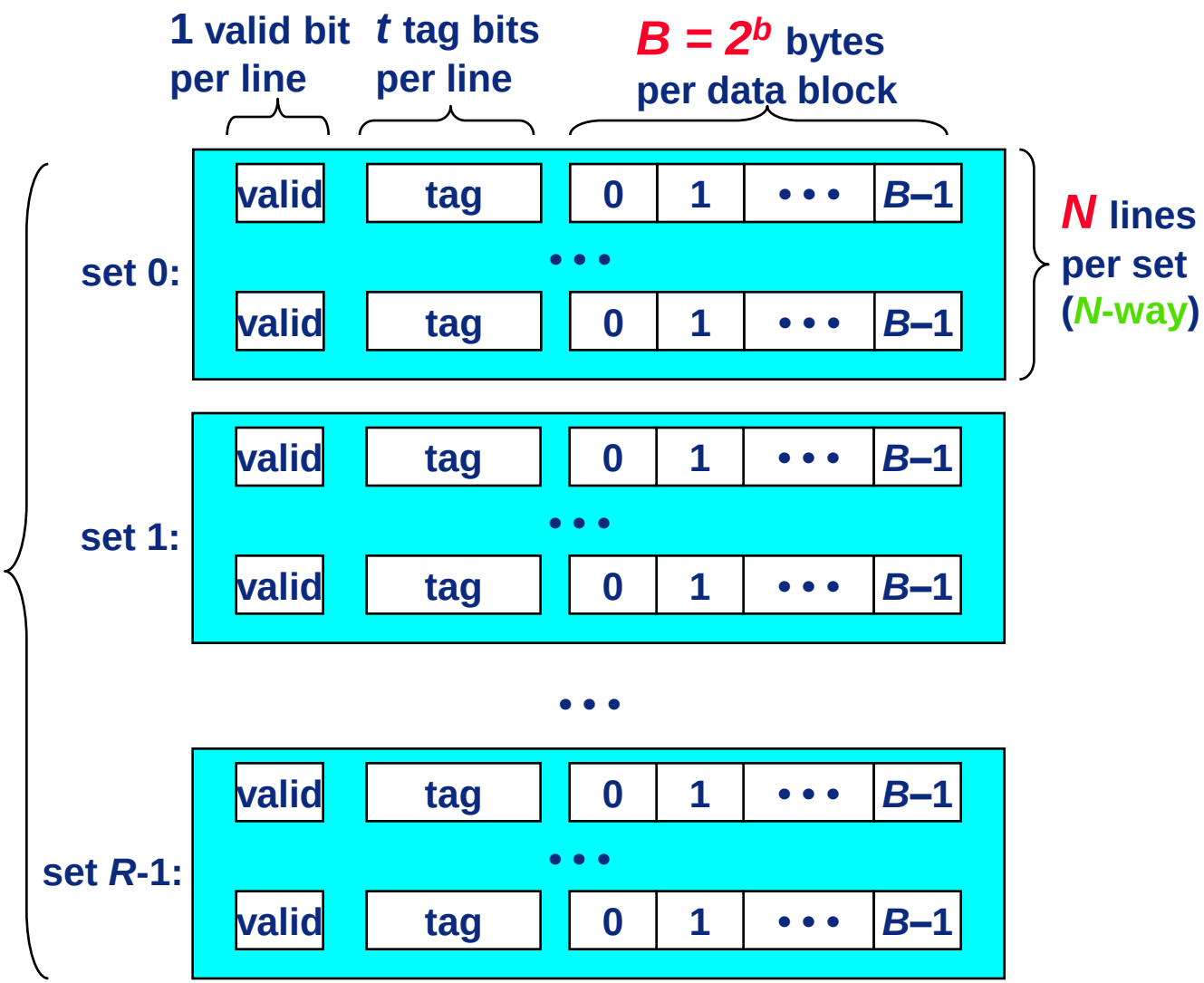
Q4: How do we find this copy quickly?

General Organization of a Cache Memory

Cache is an array of **sets**

Each set contains one or more **lines**

Each line holds a **block of data**



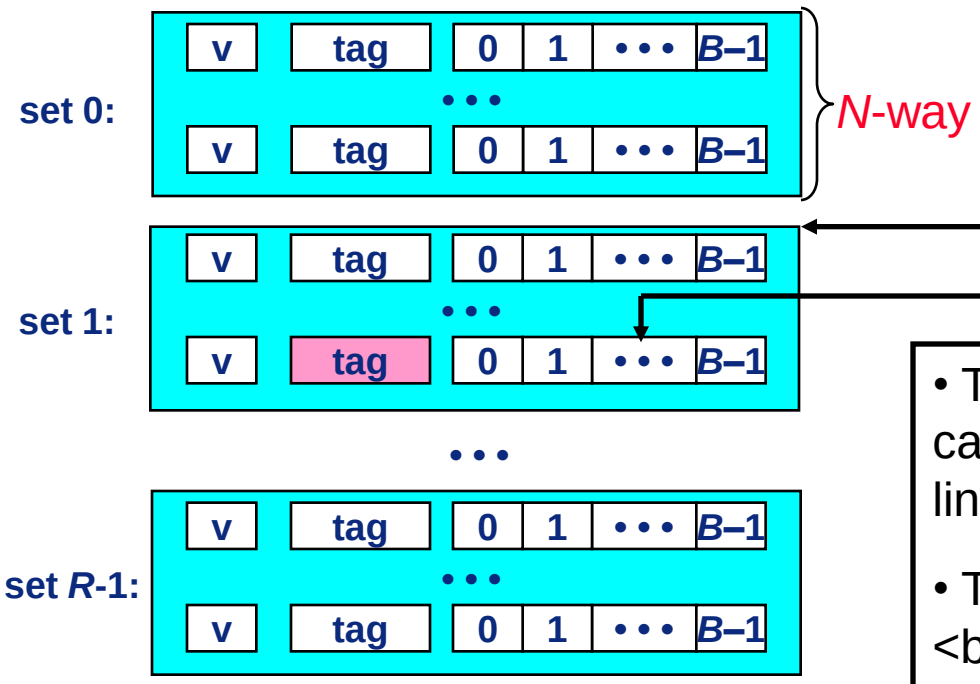
Set # ≡ hash code (index)

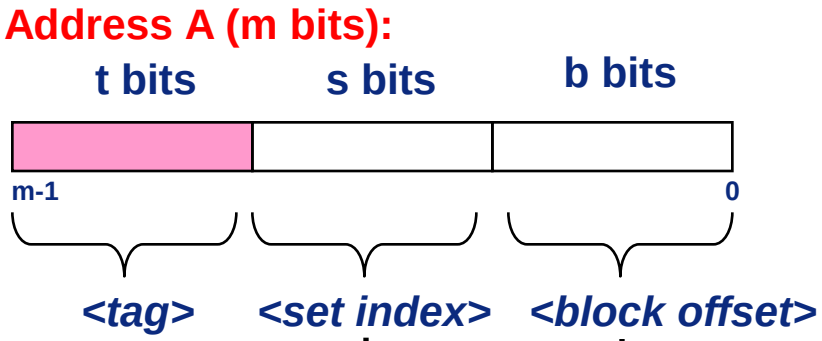
Tag ≡ hash key

Cache size: $C = B \times N \times R$ data bytes

Addressing Caches (Memory-Cache Mapping)

```
lw $t0,0($s1)  #$t0=Mem($s1)
sw $t0,0($s1)  #Mem($s1)=$t0
```





$b = \log_2(B)$
$R = C/(B*N)$
$s = \log_2(R)$
$t = m-s-b$

- The data word at **address A** is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>
- The word contents begin at offset <block offset> bytes from the beginning of the block

Address mapping:
set# = (block address) **modulo** (R)
block address =
<t bits> concatenate <s bits>

Types of Cache Organization

- **Direct-mapped**
  - N = 1
    - one line per set
    - each memory block is mapped to exactly one line in the cache)
  - b = $\log_2(B)$, R = C/B, s = $\log_2(R)$, t = m-s-b
- **Fully associative**
  - R = 1 (allow a memory block to be mapped to **any** cache block)
  - b = $\log_2(B)$, N = C/B, s = 0, t = m-b
- **n-way set associative**
  - N = n (2, 4, 8, or 16)
  - A memory block maps to a unique **set** (specified by the index field) and can be placed in any **way** of that set (so there are **n** choices)
  - b = $\log_2(B)$, R = C/(B*n), s = $\log_2(R)$, t = m-s-b

Direct Mapped Cache Example (1 word data block)

- Consider the sequence of memory address accesses

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

	tag	data block (1 word)
00		
01		
10		
11		

0, 1, 2, 3, 4, 3, 4, 15

0000, 0001, 0010, 0011, 0100, 0011, 0100, 1111

2-bits of tag, 2-bit of set address (index),
(2-bit of byte offset to data word is ignored)

0 miss

set	tag	
00	00	Mem(0)
01		
10		
11		

1 miss

00	00	Mem(0)
01	00	Mem(1)
10		
11		

2 miss

00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11		

3 miss

00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

4 miss

00	~~00~~ 01	~~Mem(0)~~ Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

3 hit!!!

00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

4 hit!!!

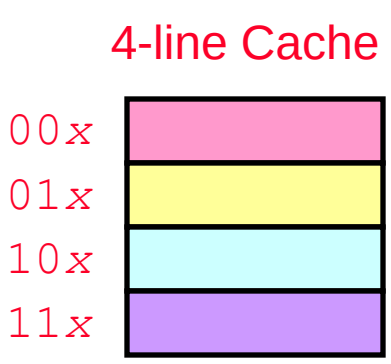
00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

15 miss

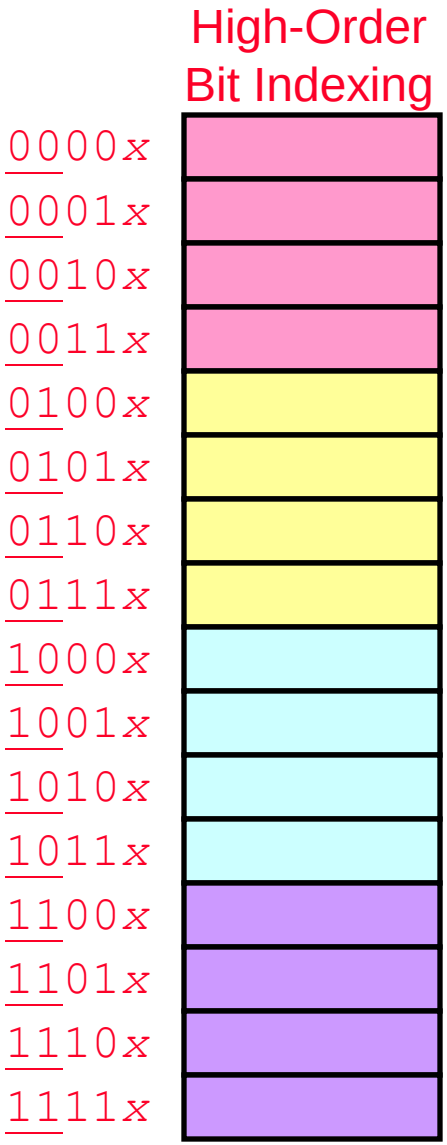
00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	~~00~~ 11	~~Mem(3)~~ Mem(15)

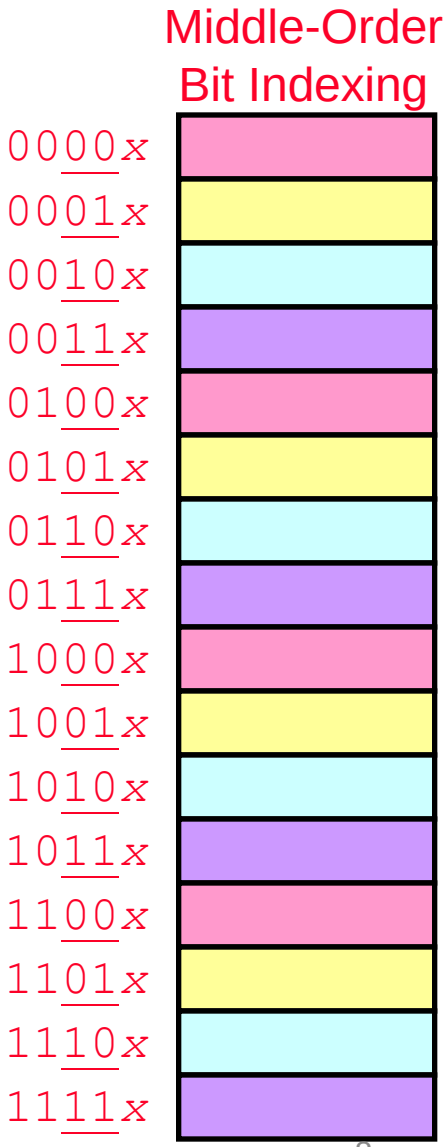
- 8 requests, 2 hits, 6 misses = 25% hit rate

Why Use Middle Bits as Index?



- High-Order Bit Indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- Middle-Order Bit Indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time
  - What type of locality?

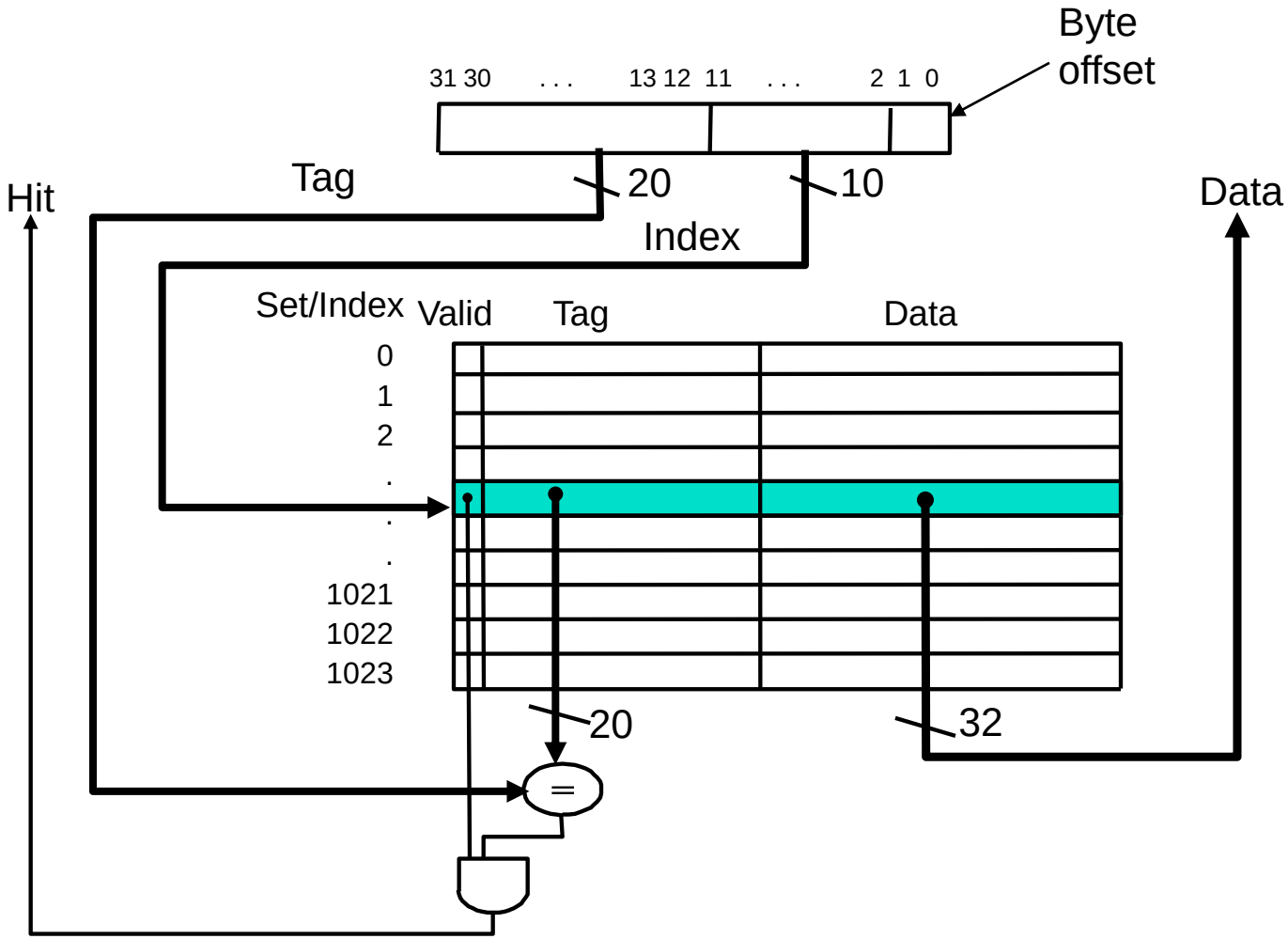




Direct Mapped Cache Example

$1024 = 2^{10}$

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



What kind of locality are we taking advantage of?

Taking Advantage of Spatial Locality

- Let cache block hold more than one word (say, two)

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

	tag	data block (2 words)	
0			
1			

0, 1, 2, 3, 4, 3, 4, 15

0000, 0001, 0010, 0011, 0100, 0011, 0100, 1111

2-bits of tag, 1-bit of set address (index), 1-bit of word-in-block select

0 miss

0	00	Mem(1)	Mem(0)
1			

1 hit

	00	Mem(1)	Mem(0)

2 miss

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

3 hit

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

4 miss

01	~~00~~	~~Mem(1)~~ 5	~~Mem(0)~~ 4
	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)

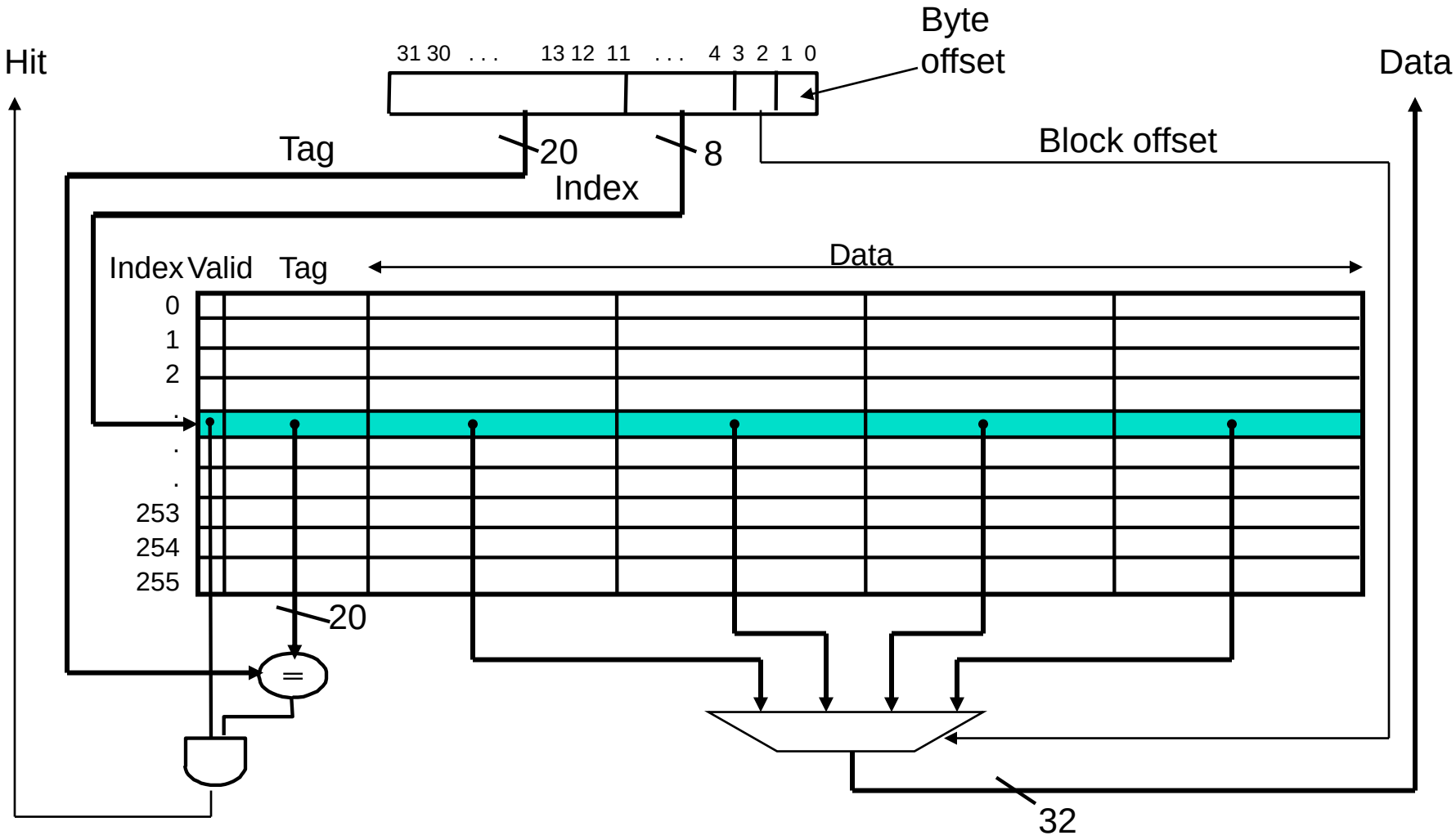
15 miss

11	01	Mem(5)	Mem(4)
	~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- 8 requests, 4 hits, 4 misses = 50% hit rate!

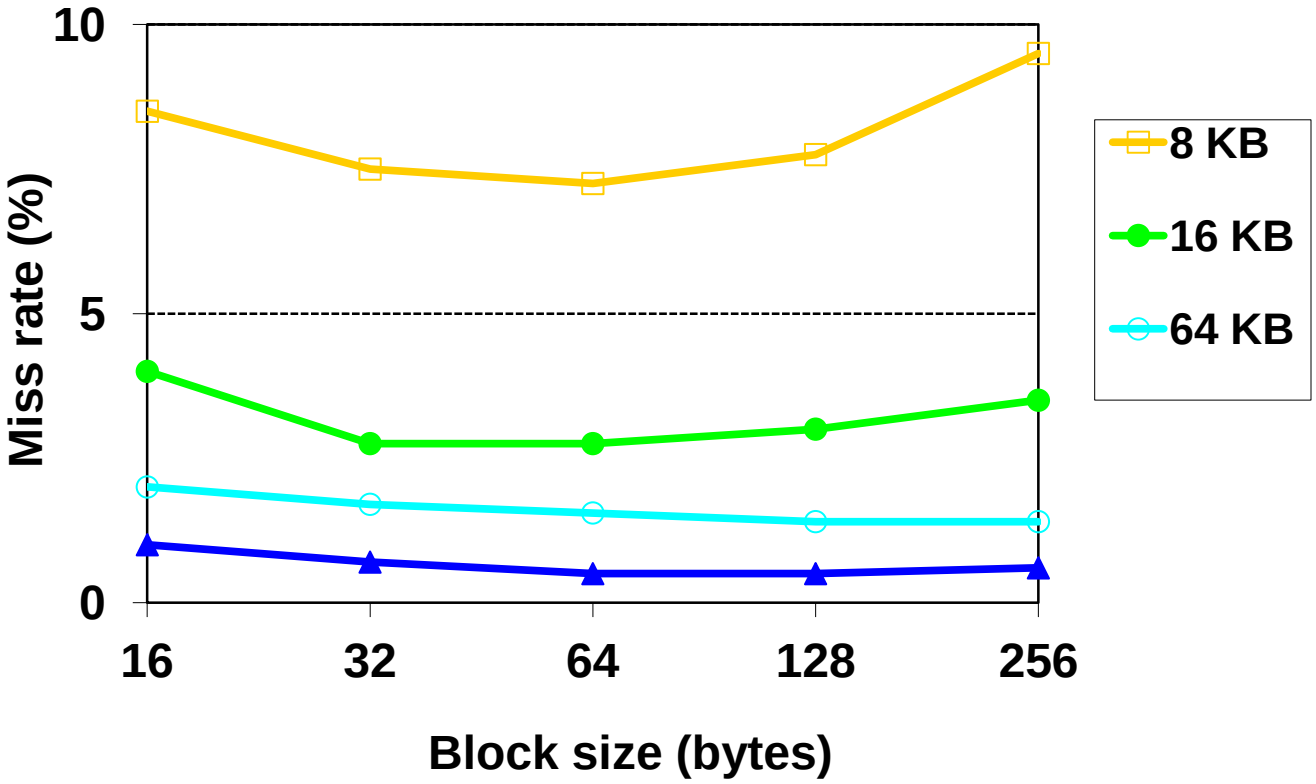
Multiword Block Direct Mapped Cache

Four data words/block, cache size = 1K words (256 blocks, 4KB total data)



What kind of locality are we taking advantage of?

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)

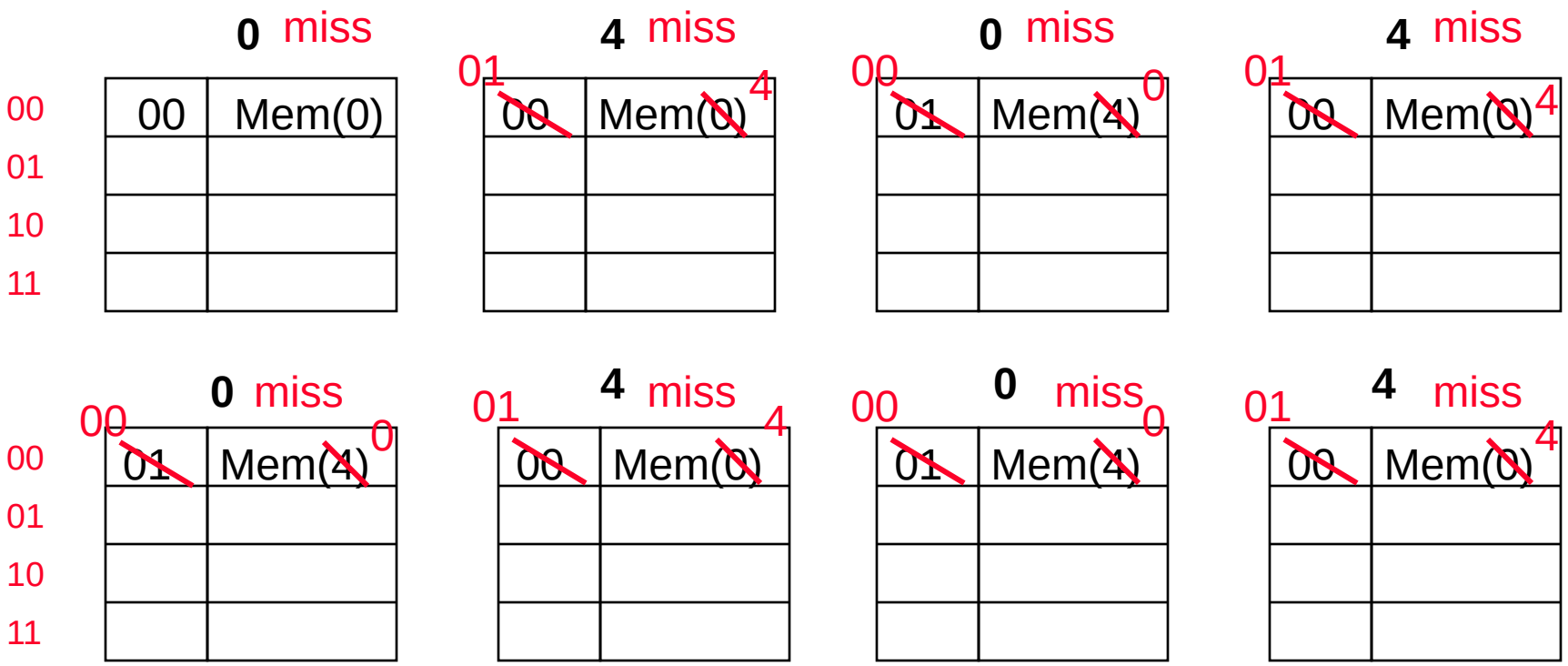
Exp: 4 Word Direct-Mapped $ for a Worst-Case Reference String

- Consider the main memory word reference string

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

0, 4, 0, 4, 0, 4, 0, 4

0000, 0100



- 8 requests, 8 misses

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

Exp: 4-Word 2-Way SA $ for the Same Reference String

- Consider the main memory word reference string

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

0, 4, 0, 4, 0, 4, 0, 4

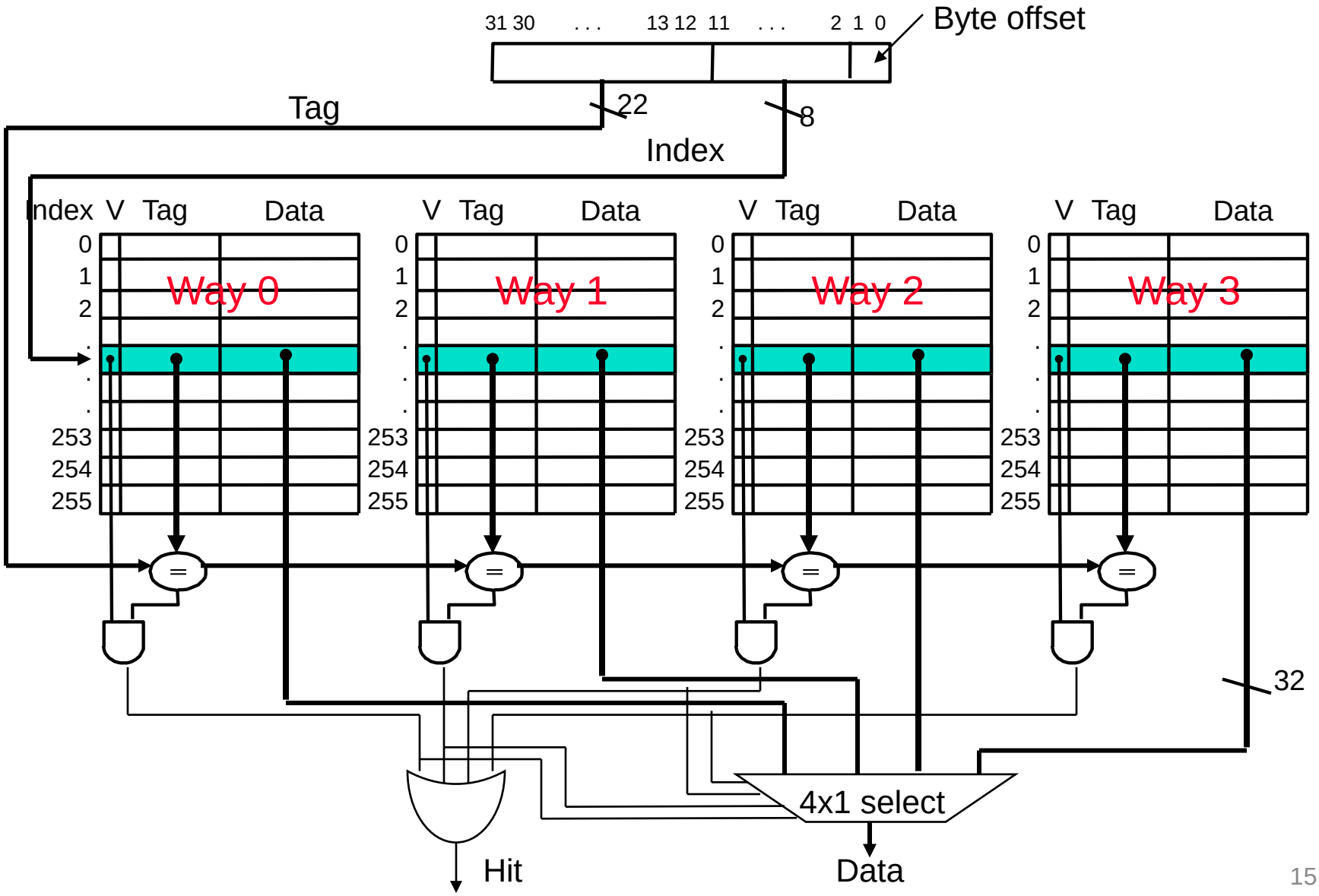
0000, 0100

- 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!

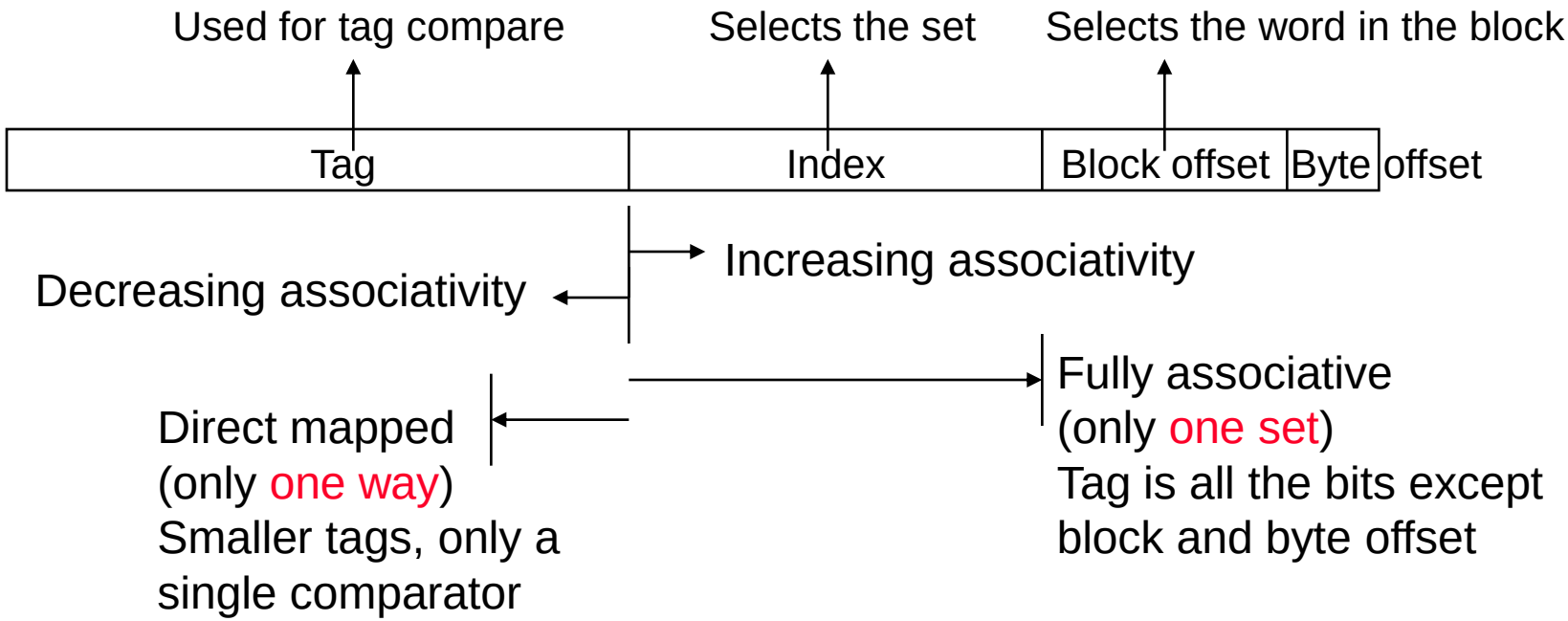
Four-Way Set Associative Cache

- 2^8 = 256 **sets** each with four ways (each with one block)



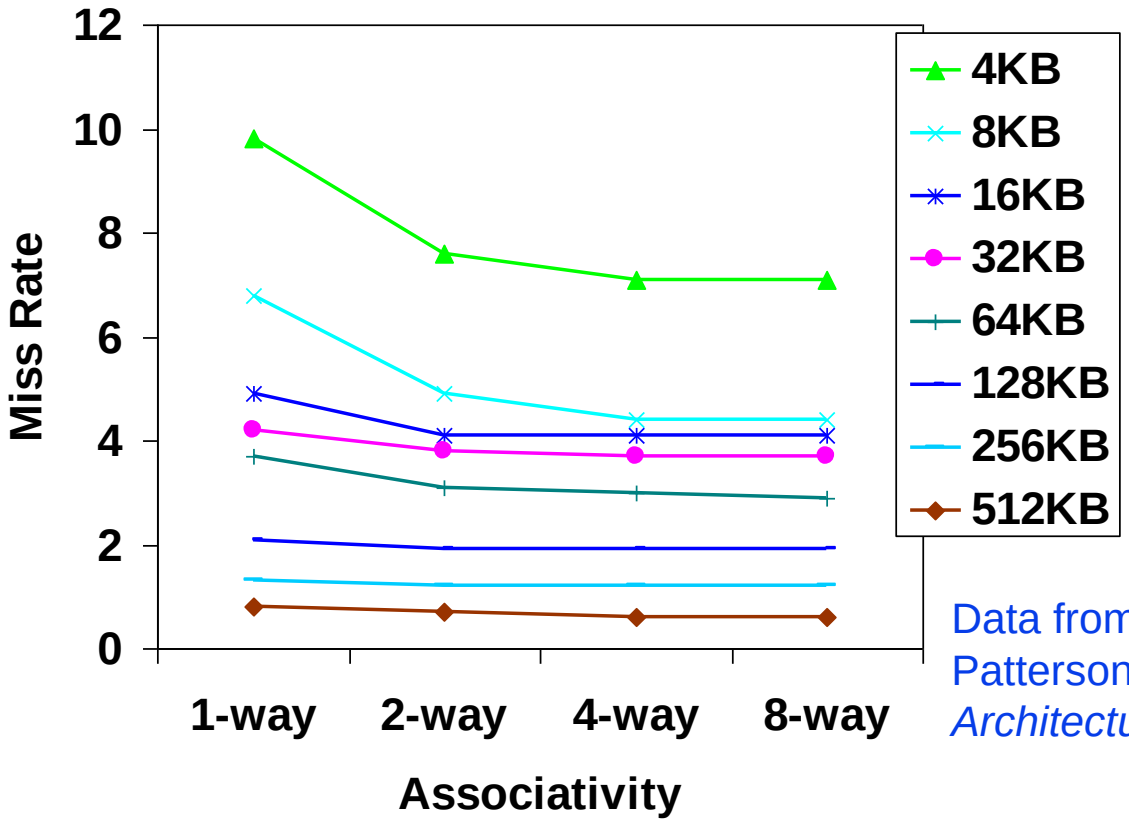
Range of Set Associative Caches

- For a **fixed size cache**, each increase by a factor of two 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



Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation



Data from Hennessy & Patterson, *Computer Architecture*, 2003

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Further Reducing Cache Miss Rates

Use multiple levels of caches

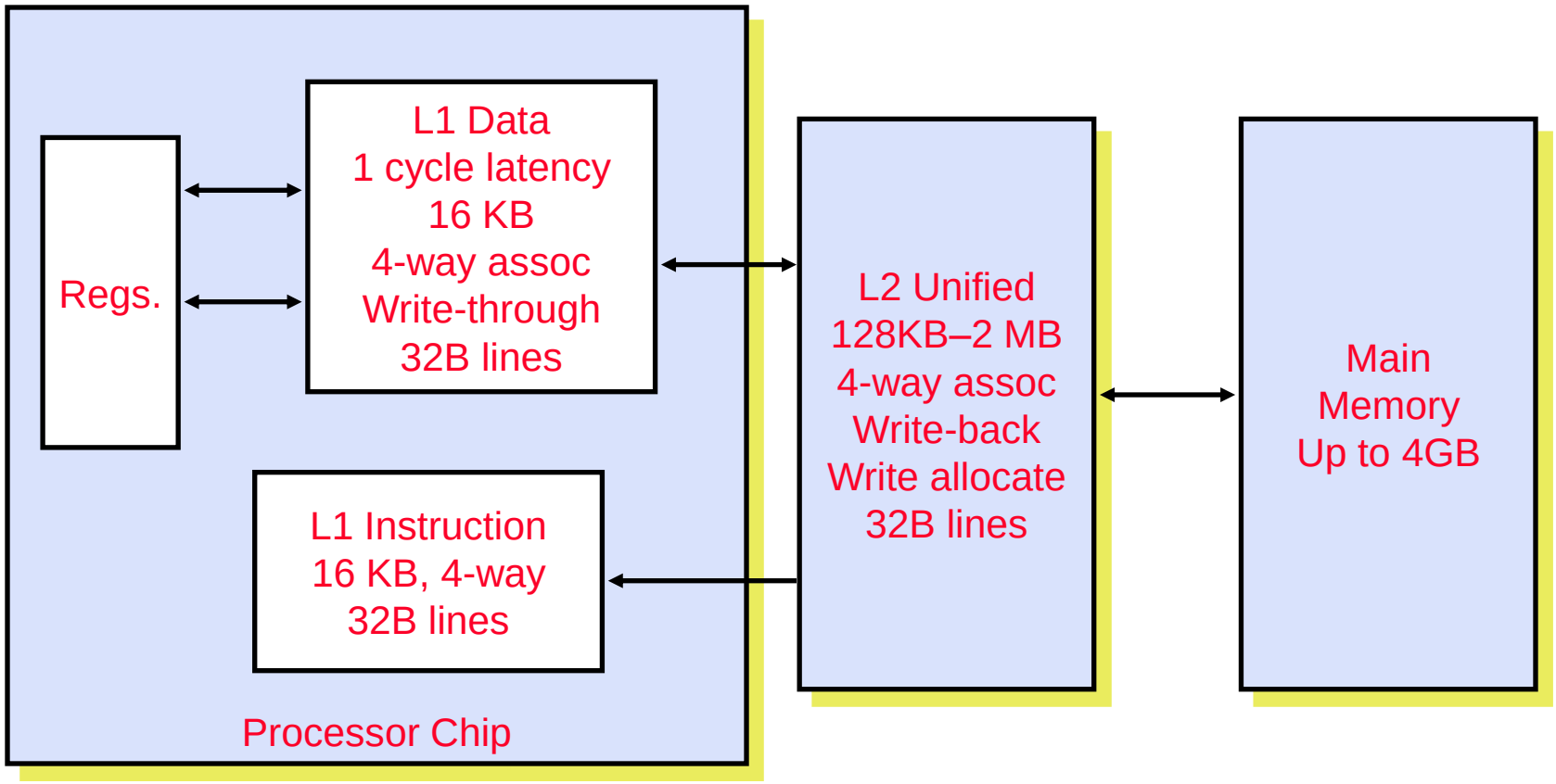
- With advancing technology have more than enough room on the die for bigger L1 caches *or* for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache

- New AMAT Calculation:

  AMAT = L1 Hit Time + L1 Miss Rate * **L1 Miss Penalty,**
  L1 Miss Penalty = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty, and so forth (final miss penalty is Main Memory access time)

- **Example**: 1 cycle L1 hit time, 2% L1 miss rate, 5 cycle L2 hit time, 5% L2 miss rate,100 cycle main memory access time
  - Without L2 cache:
    AMAT = 1 + .02*100 = 3
  - With L2 cache:
    AMAT = 1 + .02*(5 + .05*100) = 1.2

Intel Pentium Cache Hierarchy



Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller with smaller block sizes
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes
    - Higher levels of associativity
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

What parameters do you not know by far?

	Intel Nehalem	AMD Barcelona
L1 cache organization & size	Split I$ and D$; 32KB for each per core; 64B blocks	Split I$ and D$; 64KB for each per core; 64B blocks
L1 associativity	4-way (I), 8-way (D) set assoc.; ~LRU replacement	2-way set assoc.; LRU replacement
L1 write policy	write-back, write-allocate	write-back, write-allocate
L2 cache organization & size	Unified; 256MB (0.25MB) per core; 64B blocks	Unified; 512KB (0.5MB) per core; 64B blocks
L2 associativity	8-way set assoc.; ~LRU	16-way set assoc.; ~LRU
L2 write policy	write-back	write-back
L2 write policy	write-back, write-allocate	write-back, write-allocate
L3 cache organization & size	Unified; 8192KB (8MB) shared by cores; 64B blocks	Unified; 2048KB (2MB) shared by cores; 64B blocks
L3 associativity	16-way set assoc.	32-way set assoc.; evict block shared by fewest cores
L3 write policy	write-back, write-allocate	write-back; write-allocate

Handling Cache Hits

- **Read hits** (I$ and D$)
  - this is what we want!

- **Write hits** (D$ only)
  - require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (**write-through**)
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a **write buffer** and stall only if the write buffer is full
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (**write-back** the cache block to the next level in the memory hierarchy when that cache block is "evicted")
    - need a **dirty** bit for each data cache block to tell if it needs to be written back to memory when it is evicted – can use a **write buffer** to help "buffer" write-backs of dirty blocks

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. If you are going to run “millions” of instruction, compulsory misses are insignificant
  - 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)

Handling Cache Misses (Single Word Blocks)

- **Read misses** (I$ and D$)
  - stall the pipeline, fetch the block from the next level in the memory hierarchy, **install it in the cache** and send the requested word to the processor, then let the pipeline resume
- **Write misses** (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume

  or

  2. **Write allocate** – just write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall

  or

  3. **No-write allocate** – skip the cache write (but must invalidate that cache block since it will now hold stale data) and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full

Handling Cache Misses (Multiword Blocks)

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows. To reduce miss penalty:
    - **Early restart** – processor resumes execution as soon as the requested word of the block is returned
    - **Requested word first** – requested word is transferred from the memory to the cache (and processor) first
  - **Non-blocking cache** – allows the processor to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$)
  - If using write allocate must *first* fetch the block from memory and then write the word to the block (or could end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block)

Extra Costs of Set Associative Caches

- When a miss occurs, which way's block do we pick for replacement?
  - **Least Recently Used (LRU):** the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
    - For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit)
- 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). In a direct mapped cache, the cache block is available before the Hit/Miss decision
    - So its not possible to just assume a hit and continue and recover later if it was a miss

Summary: Improving Cache Performance

0. **Reduce the time to hit in the cache**
   - smaller cache
   - direct mapped cache
   - smaller blocks
   - for writes
     - no write allocate – no “hit” on cache, just write to write buffer
     - write allocate – to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

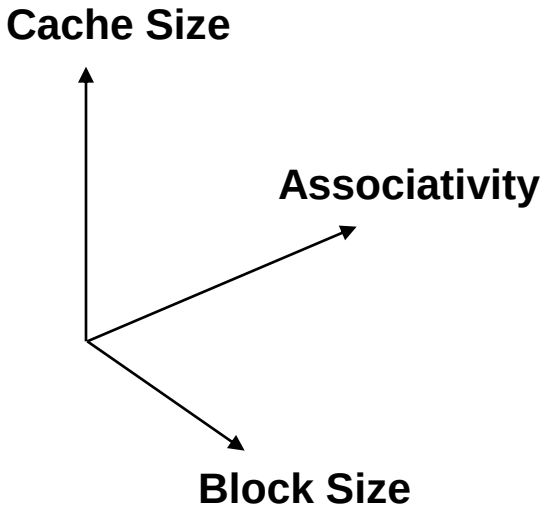
1. **Reduce the miss rate**
   - bigger cache
   - more flexible placement (increase associativity)
   - larger blocks (16 to 64 bytes typical)
   - victim cache – small buffer holding most recently discarded blocks

Summary: Improving Cache Performance

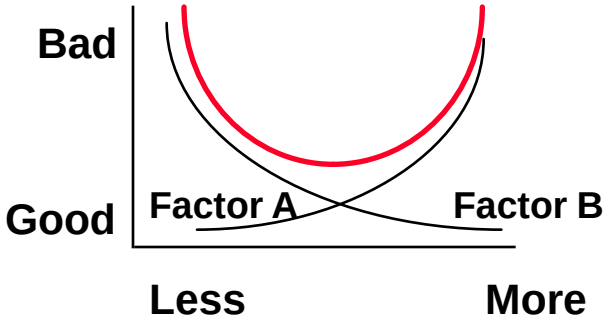
2. **Reduce the miss penalty**
   - smaller blocks
   - use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
   - check write buffer (and/or victim cache) on read miss – may get lucky
   - for large blocks fetch critical word first
   - use multiple cache levels – L2 cache not tied to CPU clock rate
   - faster backing store/improved memory bandwidth
     - wider buses
     - memory interleaving, DDR SDRAMs

Summary: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation



- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost



- Simplicity often wins

Takeaway

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three major categories of cache misses:
  - Compulsory misses: sad facts of life. Example: cold start misses
  - Conflict misses: increase cache size and/or associativity Nightmare Scenario: ping pong effect!
  - Capacity misses: increase cache size
- Cache design space
  - total size, block size, associativity (replacement policy)
  - write-hit policy (write-through, write-back)
  - write-miss policy (write allocate, write buffers)

Self-review Questions for the Memory Hierarchy

Q1: Where can a entry be placed in the upper level?
(Entry placement)

Q2: How is a entry found if it is in the upper level?
(Entry identification)

Q3: Which entry should be replaced on a miss?
(Entry replacement)

Q4: What happens on a write?
(Write strategy)

Q1&Q2: Where can an entry be placed/found?

	# of sets	Entries per set
Direct mapped	# of entries	1
Set associative	(# of entries)/ associativity	Associativity (typically 2 to 16)
Fully associative	1	# of entries

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all entries' tags Separate lookup (page) table	# of entries 0

Q3: Which entry should be replaced on a miss?

- Easy for direct mapped – only one choice
- Set associative or fully associative
  - Random
  - LRU (Least Recently Used)
- For a 2-way set associative, random replacement has a miss rate about 1.1 times higher than LRU
- LRU is too costly to implement for high levels of associativity (> 4-way) since tracking the usage information is costly

Q4: What happens on a write?

- **Write-through** – The information is written to the entry in the current memory level *and* to the entry in the next level of the memory hierarchy
  - Always combined with a write buffer so write waits to next level memory can be eliminated (as long as the write buffer doesn't fill)
- **Write-back** – The information is written only to the entry in the current memory level. The modified entry is written to next level of memory only when it is replaced.
  - Need a **dirty bit** to keep track of whether the entry is clean or dirty
  - Virtual memory systems always use write-back of dirty pages to disk
- Pros and cons of each?
  - Write-through: read misses don't result in writes (so are simpler and cheaper), easier to implement
  - Write-back: writes run at the speed of the cache; repeated writes require only one write to lower level