CS3350B Computer Organization
Chapter 1: CPU and Memory
Part 3: Cache Implementations

Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Monday January 22 2024

Outline

How is the Hierarchy Managed?

Recall: CPU ← Registers ← L1 ← . . . ← Main Memory ← Disk

- Registers ↔ Cache Memory:
  - ↳ The compiler (sometimes with the programmer's help) decides which values are stored in which registers.
- Caches ↔ Main Memory:
  - ↳ The **cache controller** (thus the hardware) handles cache memory movement.
- Main Memory ↔ Disk:
  - ↳ The operating system (which controls the virtual memory).
  - ↳ the TLB (thus the hardware) which assists the virtual-to-physical address mapping.
  - ↳ The programmer (who organizes the data into files), if data comes from file.

Cache Design Questions

Q1 How best to organize the memory blocks (a.k.a lines) inside the cache?

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

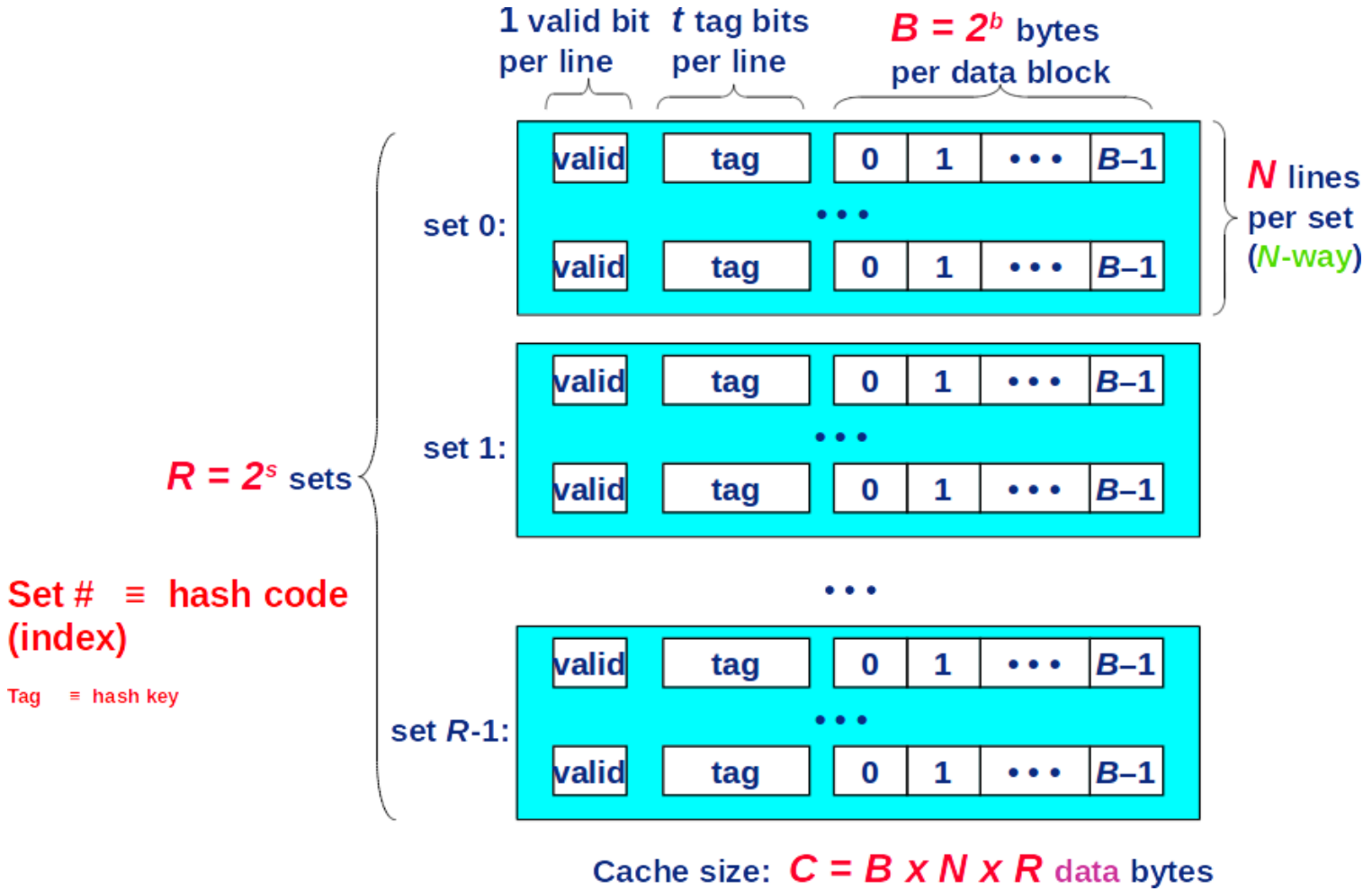
↳ *Note:* since the cache is a subset of the main memory, multiple memory addresses can map to the same cache location.

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

Q4 How do we quickly find a particular copy of main memory (memory address contents) in the cache?

General Organization of a Cache Memory (1/2)

- Cache is an array of $R = 2^s$ sets
- Each set contains $N \geq 1$ lines
- Each **cache line** (block) holds $B = 2^b$ bytes of data

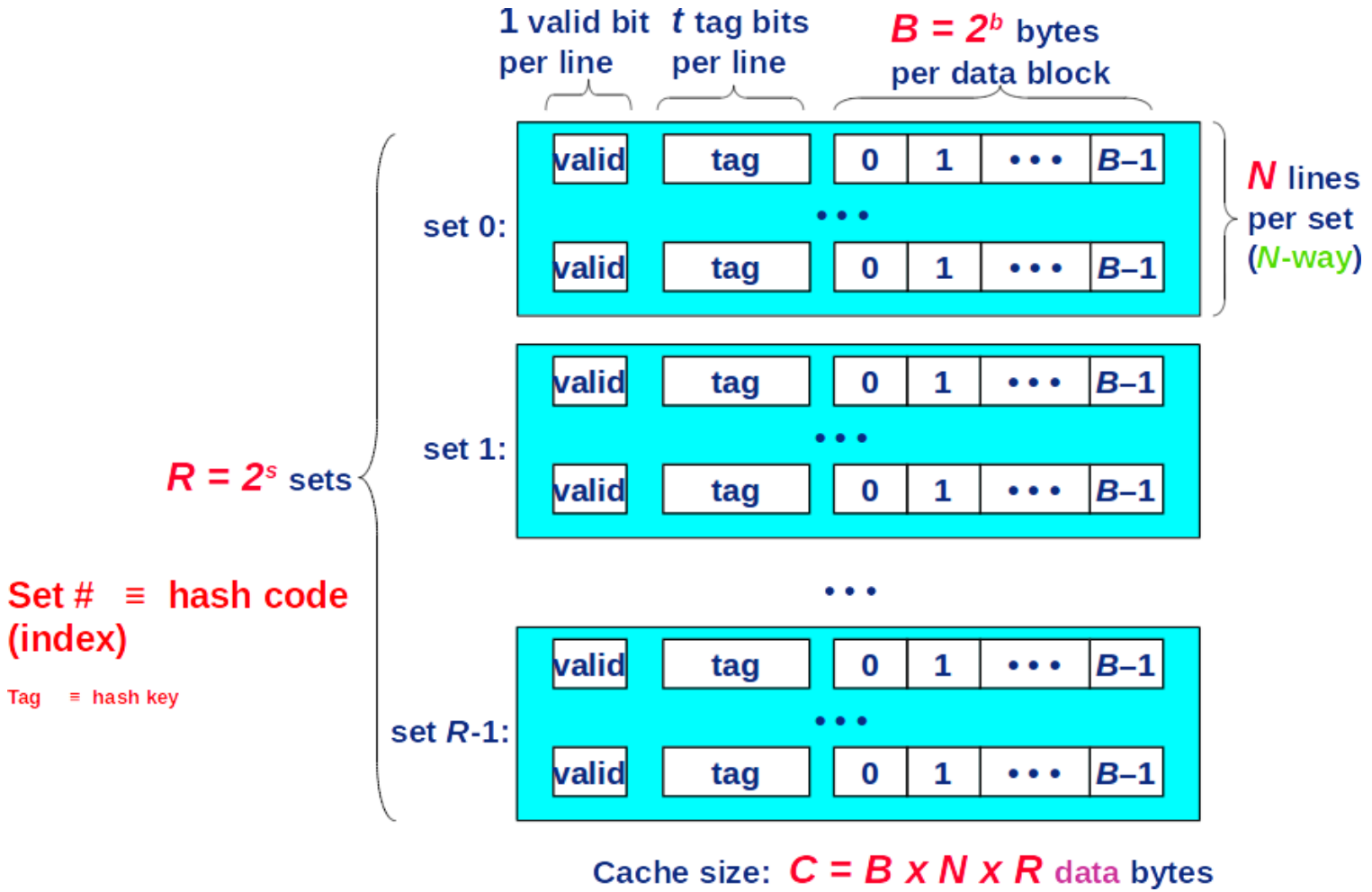


General Organization of a Cache Memory (2/2)

- **Cache capacity** = bytes per line × lines per set × number of sets

$$C = B \times N \times R$$

- All values are a power of 2.



Memory-Cache Mapping (Addressing Cache Memories)

- The data word at the m-bit **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

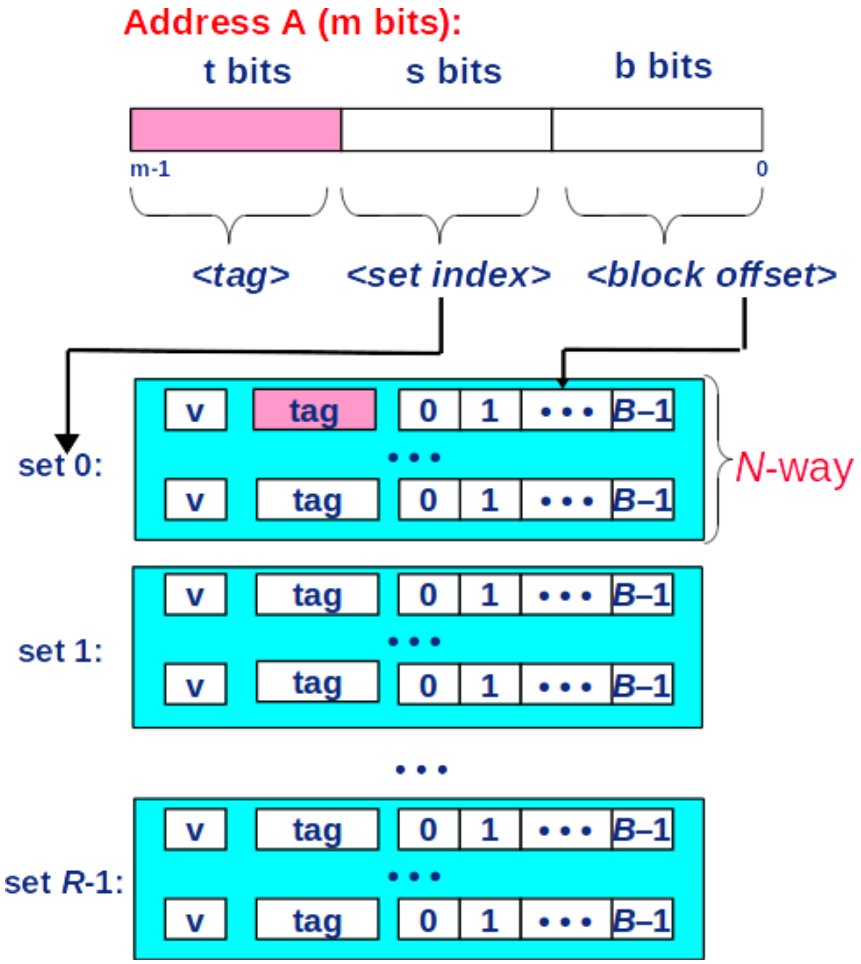
Address Mapping:

block address = <tag> || <set index>

set# = (block address) $\mod R$

↳ just take the "s bits" as set index.

Note: Main memory address space and cache size/implementation highly coupled.



$b = \log_2(B),\ R = C/(B * N)$

$s = \log_2(R),\ t = m - s - b$

Word Addresses and Byte Addresses

Often, a memory word for a computer will be more than 1 byte. In these cases a *byte memory address* and a *word memory address* are not the same.

- Consider an N-byte word with *word address* X.
- This word address is actually addressing the *byte* $N \times X$, and the word X has *byte address* $N \times X$.

Example

- Consider the word address 4 on a 16-bit computer.
- 4 in binary is 100, but we must account for the fact that words are 2 bytes long.
- The word address 4 is actually the byte address 8 ($=$ 1000 in binary).

Note: if not explicitly said, an address should be considered a byte address.

Outline

Types of Cache Organization

Direct-Mapped

- **N = 1**
  - ↳ One line per set.
  - ↳ Each memory address is mapped to exactly one line in the cache.
- $b = \log_2(B)$, $R = C/B$, $s = \log_2(R)$, t (tag size) $= m - s - b$.

Fully Associative

- R $= 1$ (allow a memory address to be mapped to any cache block).
- Tag is whole address except block offset.
- $b = \log_2(B)$, $N = C/B$, $s = 0$, $t = m - b$.

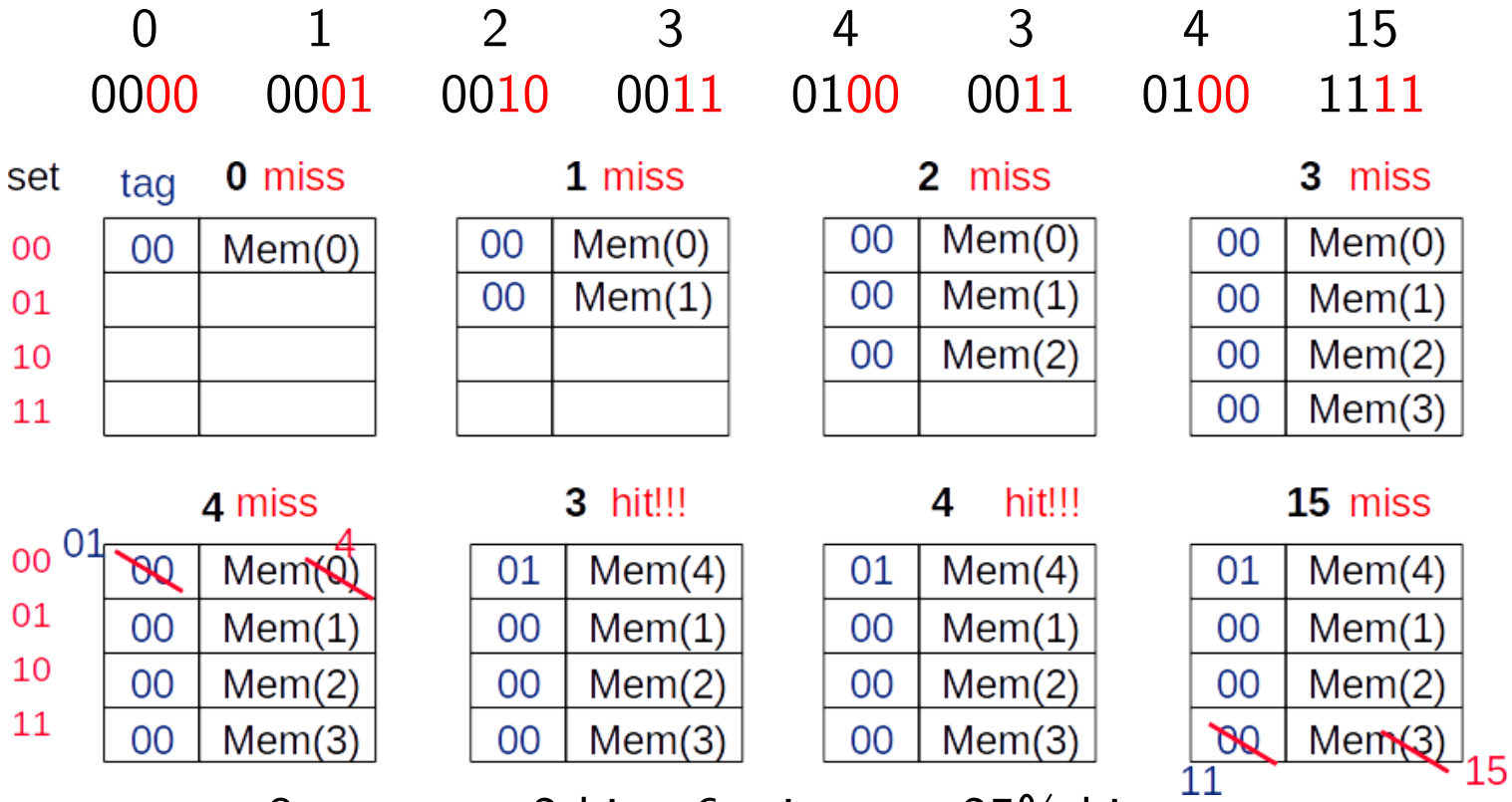
N-way set associative

- N is typically 2, 4, 8, or 16.
- A memory block maps to a specific set but can and can be placed in any **way** of that set (so there are N choices of mapping).
- $b = \log_2(B)$, $R = C/(B \times N)$, $s = \log_2(R)$, $t = m - s - b$.

Direct-Mapped Cache Example

- Direct-Mapped $\Longrightarrow$ $N = 1$. Let $B = 1$, $R = 4$.
- Therefore we have a 2-bit set index. Assume a 2-bit tag $\Longrightarrow$ $m = 4$.
- Start with an empty cache – blanks are considered invalid.

Consider the sequence of memory address accesses:

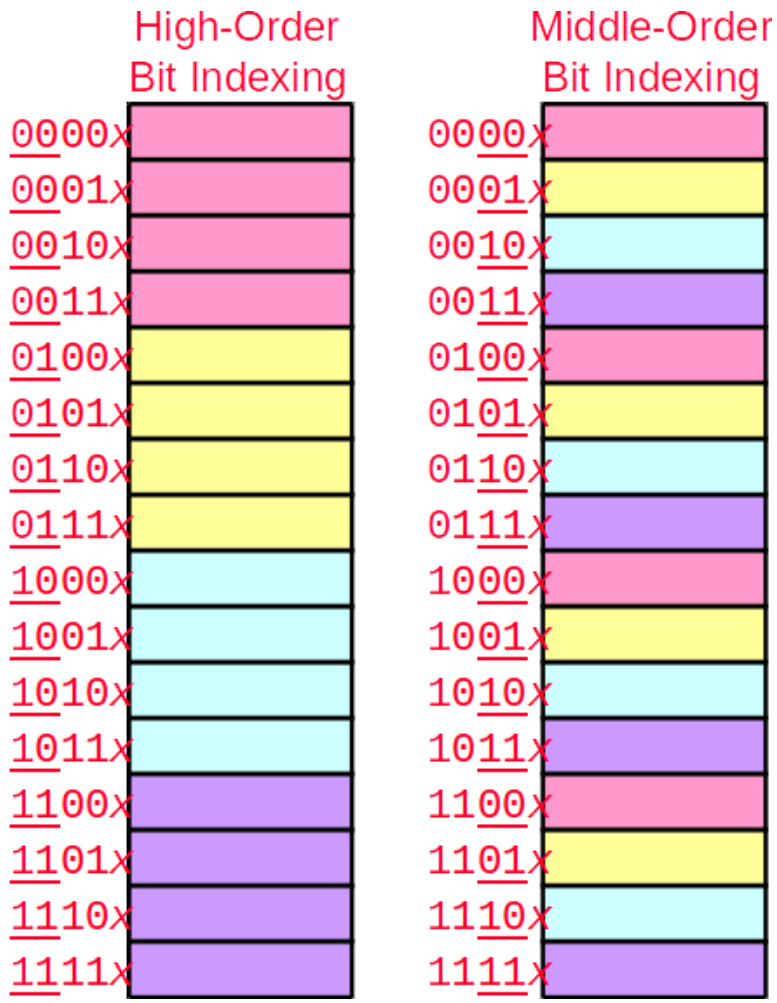


8 requests, 2 hits, 6 misses $=$ 25% hit rate

Why Middle Bits For Set Index?

Consider a 4-line direct-mapped cache; sets are indexed as 00, 01, 10, 11:

- High-Order Bit Indexing
  - ↳ Adjacent memory lines would map to same cache entry
  - ↳ Poor use of locality
- Middle-Order Bit Indexing
  - ↳ Consecutive memory lines map to different cache lines
  - ↳ Can hold C-bytes of contiguous memory in cache at one time

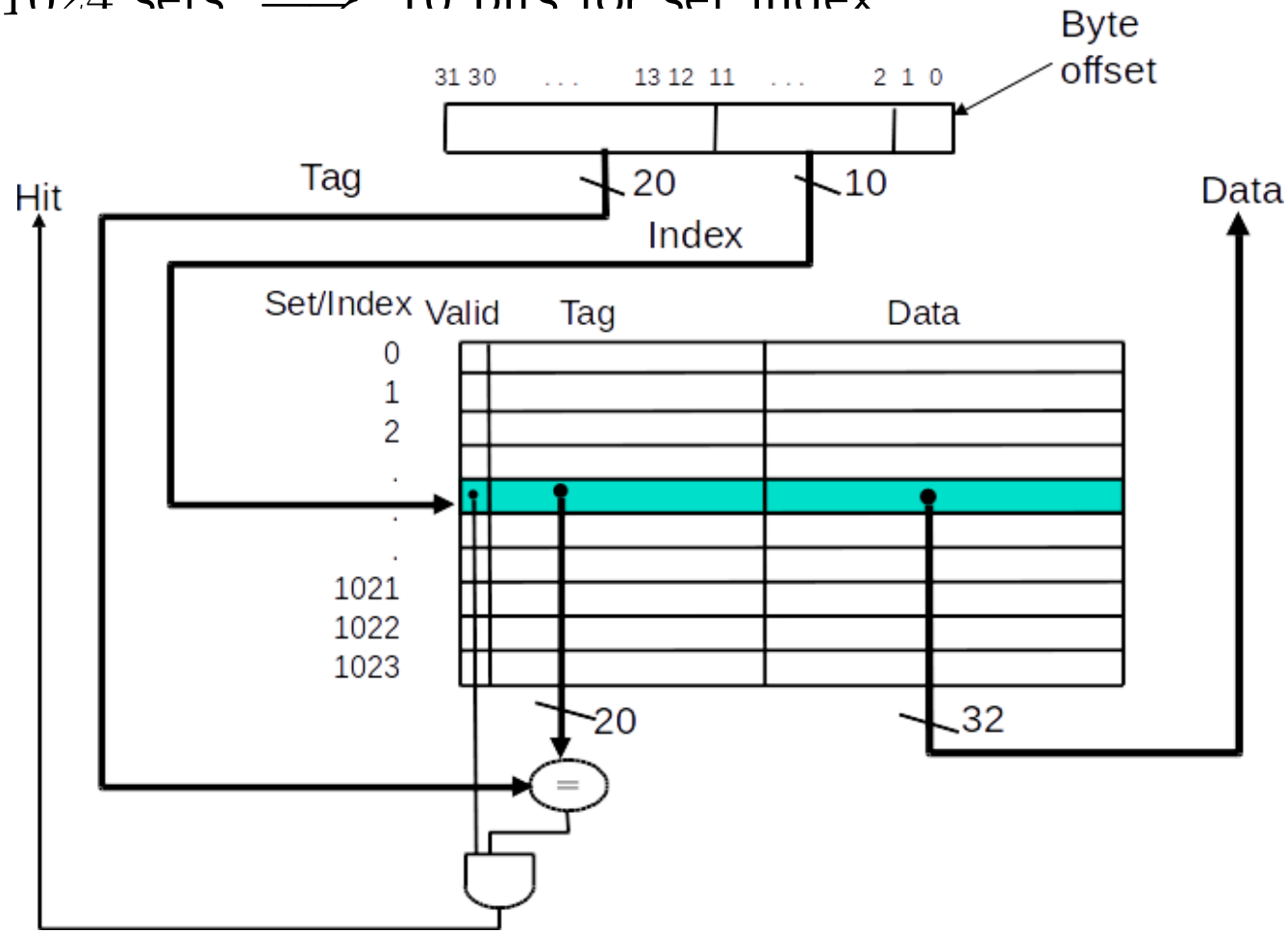


In this example (0000x, 0001x, . . .) with an m-bit address, x must be $m-4$ bits.

Direct-Mapped Cache Circuit Logic Example 1

In this example:

- One word (4 byte) cache lines ($B = 4$).
- Notice byte offset related to, but not exactly, block offset.
- $2^{10} = 1024$ sets $\implies$ 10 bits for set index

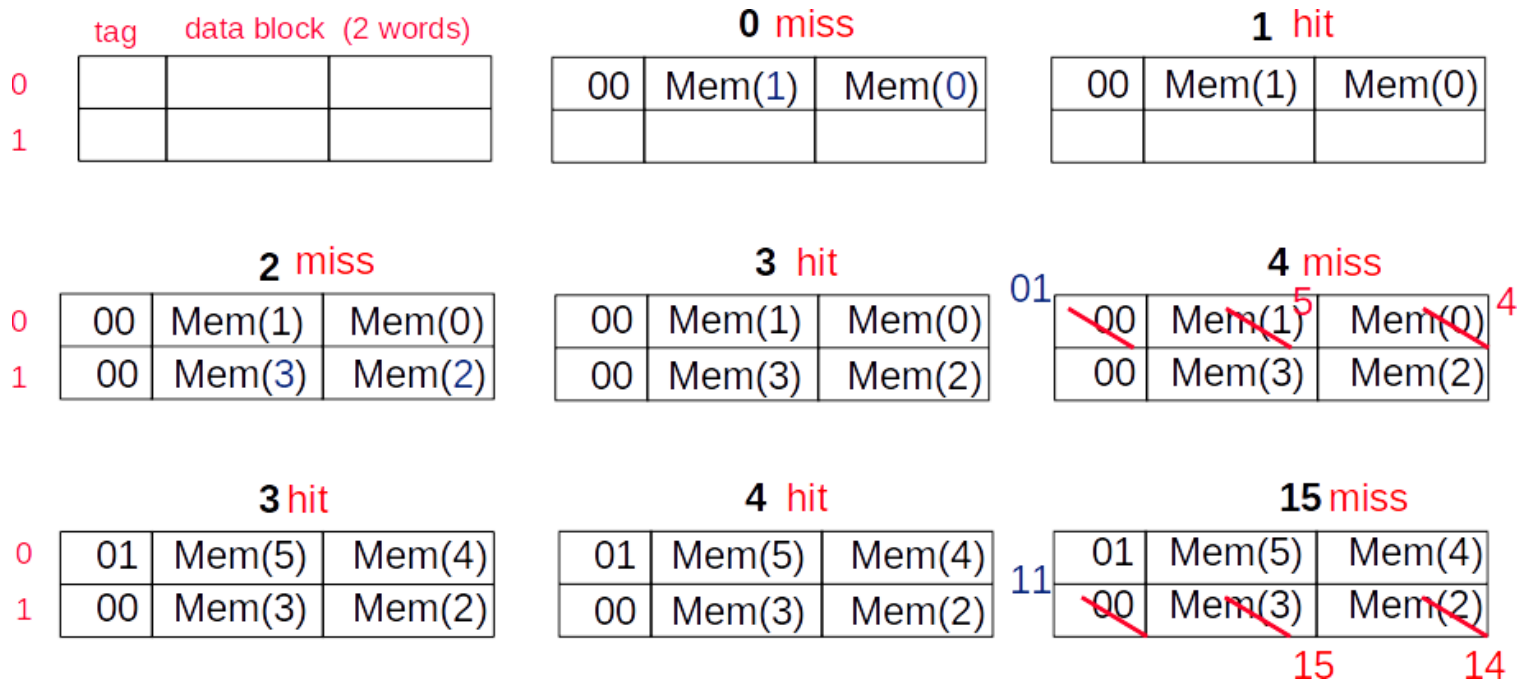


Direct-Mapped Cache Example 2: $(B > 1)$

- Let the cache line hold two words = bytes $(B = 2)$.
- Direct-Mapped $\Longrightarrow$ $N = 1$. $R = 2$.
- Therefore 1-bit set index. Assume a 2-bit tag again.
- Start with an empty cache – blanks are considered invalid.

Consider the sequence of memory address accesses:

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

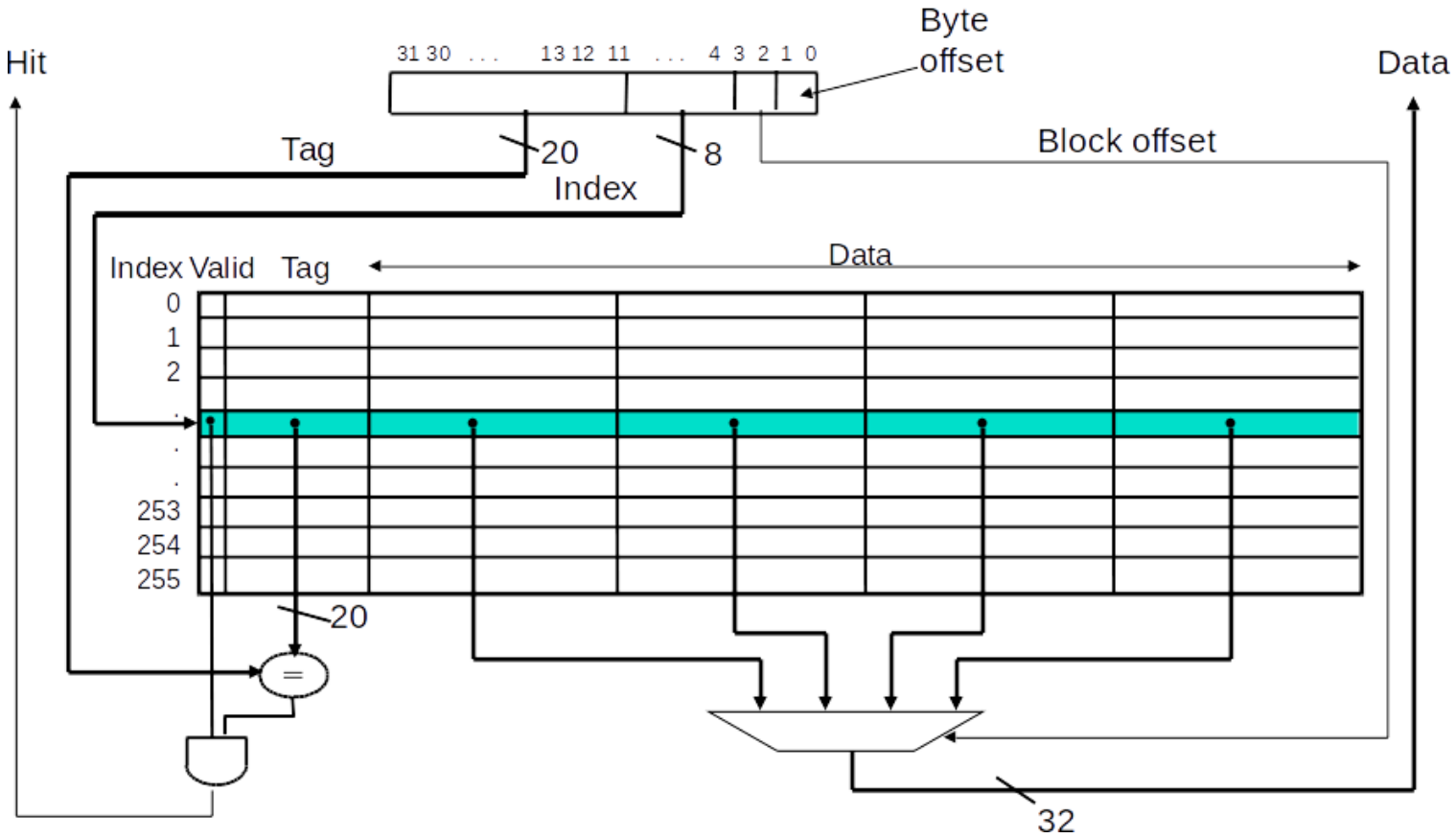


8 requests, 4 hits, 4 misses = 50% hit rate!

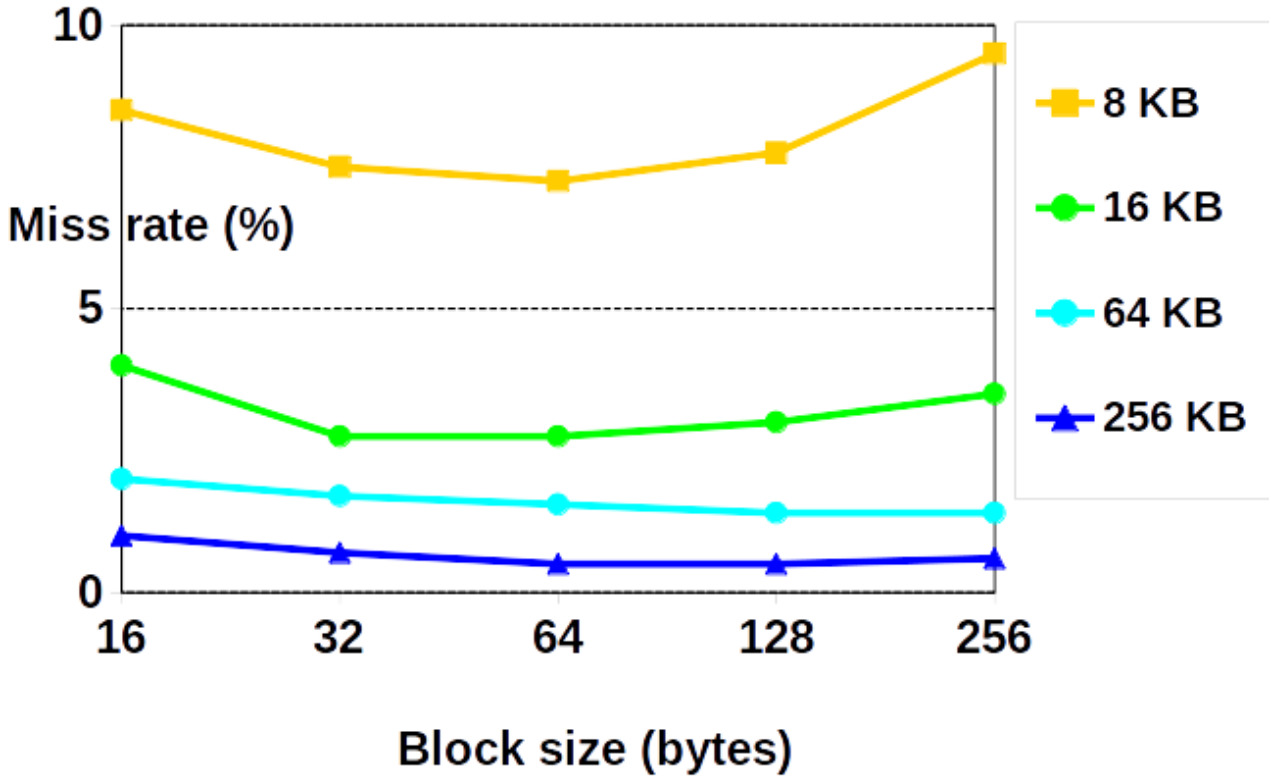
Direct-Mapped Cache Circuit Logic Example 2

In this example:

- Four data words/block ($B = 16$).
- $2^8 = 256$ sets $\implies$ 8 bits for set index.



Block Size & Cache Performance



Miss rate goes up if the block size becomes a significant fraction of the cache size.

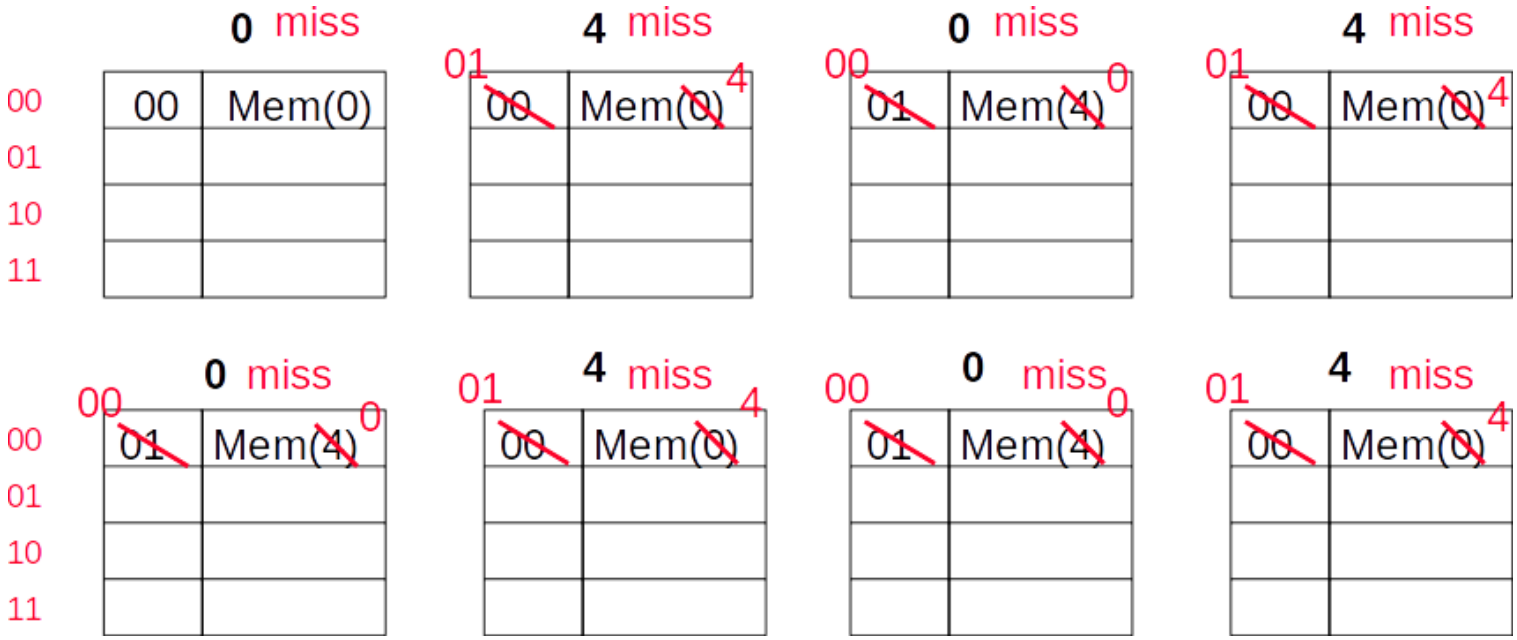
- ↳ For a fixed cache size, ↑ block size $\Longrightarrow$ ↓ number of blocks.
- ↳ Increases number of **capacity misses**.

Direct-Mapped Cache Worst-Case

Direct-Mapped $\Longrightarrow$ $N = 1$. Let $B = 1$, $R = 4$.

Consider the sequence of memory addresses accesses:

$0 = (00\mathbf{00}), 4 = (01\mathbf{00}), 0, 4, 0, 4, 0, 4, \ldots$



8 requests, 8 misses = 100% miss rate!

Thrasing!

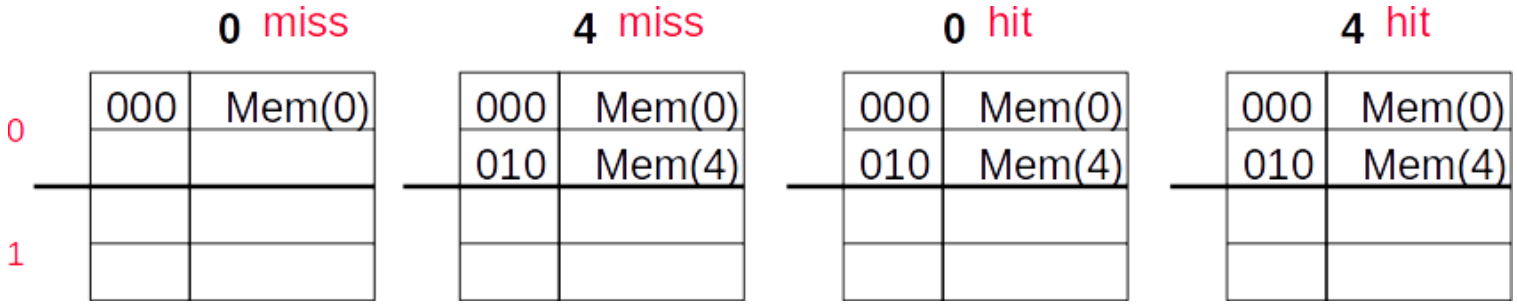
↳ **conflict misses** – two addresses map into the same cache block

Thrashing Fix with Set Associativity

2-way associative cache $\implies$ $N = 2$. Let $B = 1$, $R = 2$ now.

Consider the sequence of memory addresses accesses:

$$0 = (0000),\ 4 = (0100),\ 0,\ 4,\ 0,\ 4,\ 0,\ 4,\ \ldots$$



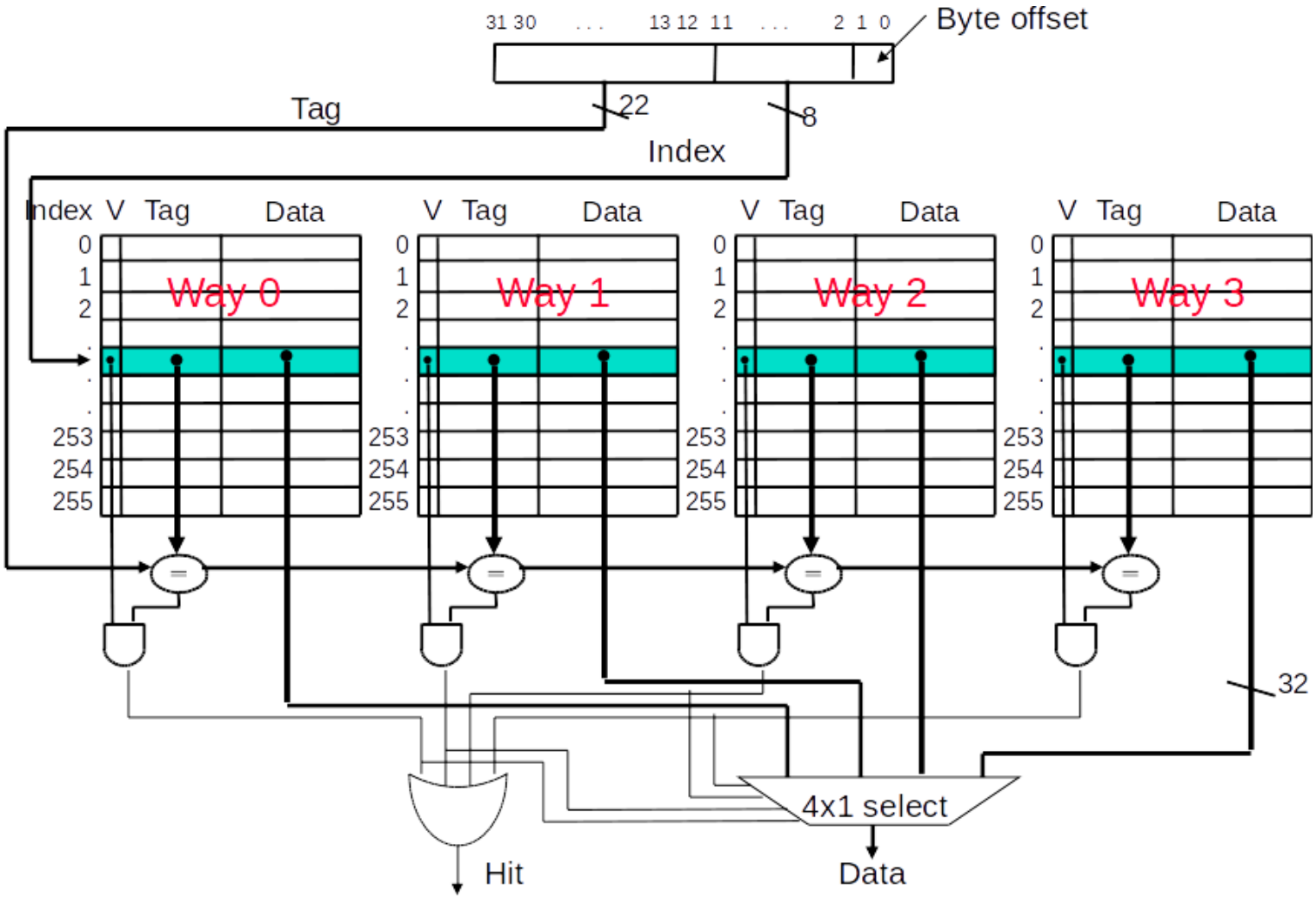
8 requests, 2 misses $=$ 75% hit rate

Solves thrashing of direct-mapped cache caused by *conflict* misses

↳ Now two memory locations that map to the same block can co-exist.

Four-way Set Associative Cache Circuit Logic Example

- $2^8 = 256$ sets each with four ways (each with one block of one word)



Set Associativity Costs Extra

When a miss occurs, which way do we pick for replacement?

- **Least Recently Used (LRU)**: the block replaced is the one that has been unused for the longest time.
  - ↳ Hardware must keep track of when each way was last 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).
- **First In First Out (FIFO)**
  - ↳ Can be implemented as a circular buffer or counter.

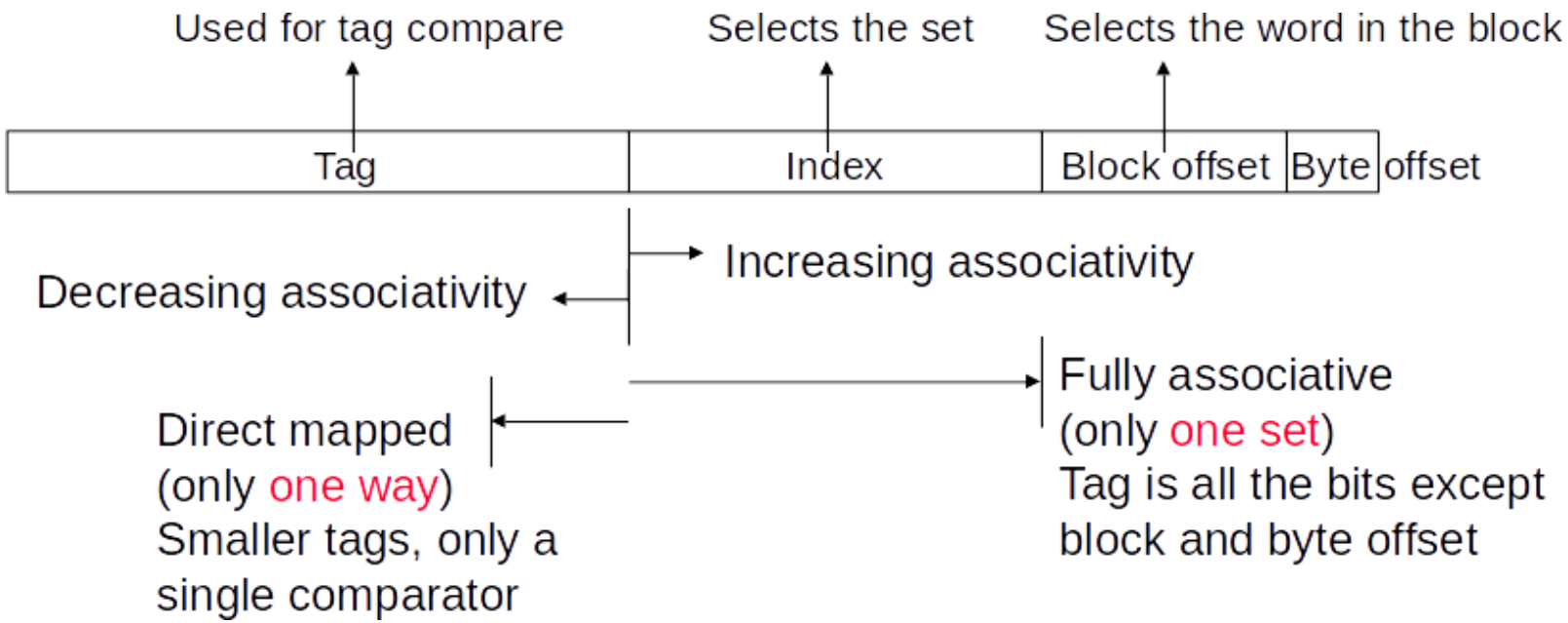
N-way set associative cache costs:

- N comparators for LRU policy (delay and physical area on chip).
- MUX delay (selecting the proper way) before data is available.
- In a direct mapped cache, the cache block is available *before* the hit/miss decision.
  - ↳ Impossible to assume a hit and recover later in set associative caches.

Range of Set Associative Caches

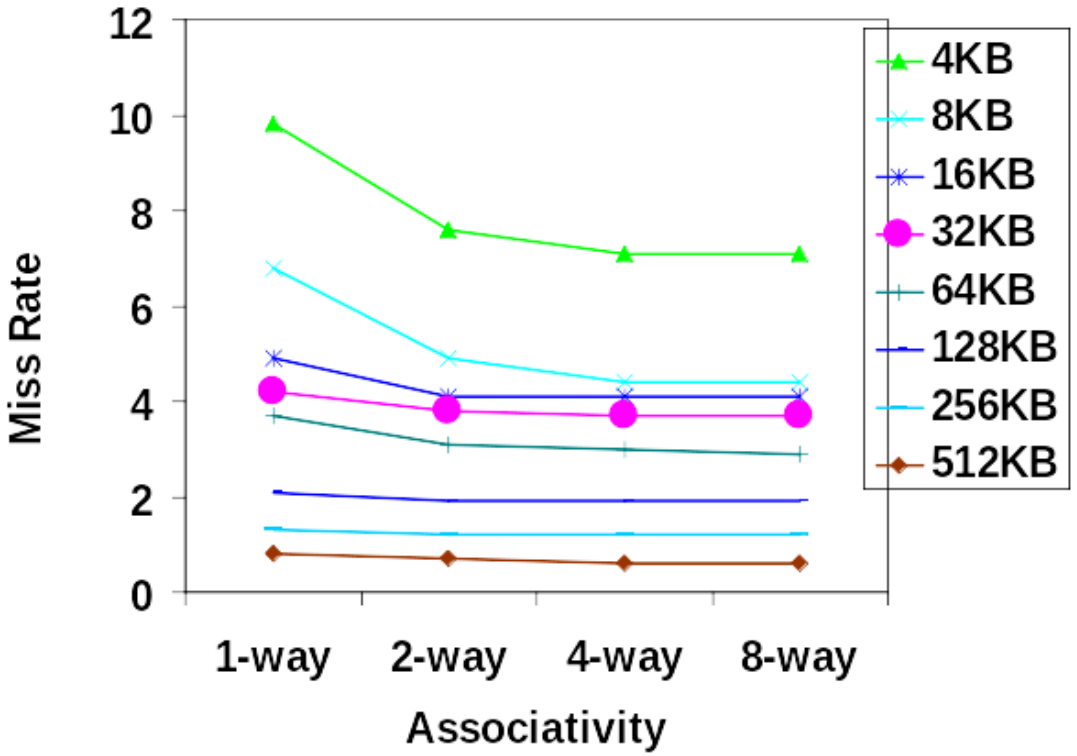
For a **fixed size cache** (and fixed size of cache lines) each doubling in associativity (ways):

- Doubles number of blocks per set,
- Halves number of sets,
- Decreases size of set-index by 1 bit,
- Increases size of tag by 1 bit.



Benefits of Set Associative Caches

The choice between *direct mapped* (1-way) and *set associative* 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).

Cache Structure in Different Processors

What do we know so far?

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

Outline

Handling Cache Hits

Read Hits (Instructions and Data)

- This is what we want! $\Longrightarrow$ Do nothing special.

Write Hits (Data only) has two policies:

- **Write-through:** Require the cache and backing memory to always be **consistent**.
  - ↳ When writing to a cache, also pass the value to the next lower level cache (or main memory) to update its copy.
  - ↳ In naïve implementation this is very slow. Speed of cache writes limited by the speed of the next lower level.
  - ↳ Use a **write buffer** between levels and stall only if buffer is full.
- **Write-back:** Allow cache and memory to be **inconsistent**
  - ↳ Write data into the cache block, this becomes a **dirty block**.
  - ↳ Only update the next lower level when a dirty block is evicted.
  - ↳ Requires $>$**two cycles** – one to check for evict/dirty and another to actually do write – or again use a **write buffer** and use only one cycle.

Handling Cache Misses

Read Misses (Instruction and Data)

- **Stall** the execution, fetch block from the next lower level of memory, write ("install") it into cache, pass it to next higher level of memory (or the processor), and let processor resume.

Write Misses (Data only) **stall** execution and perform one of two policies:

- **Write allocate:**
  - ↳ Fetch the block from the next level in the memory hierarchy,
  - ↳ Install it in the cache and write the updated word into the new block
  - ↳ Installing block and writing the updated word can be done simultaneously.
- **No-Write Allocate:** Skip fetching/installing block into cache, write directly into lower level of memory (or a write buffer).

then let the processor resume.

Dealing With Cache Misses with Hardware Design

Compulsory Misses:

- Caused by **cold** starts, process migrations or very first references.
- Reduce impact by

Capacity Misses:

- Caused by the cache becoming full; it cannot hold all blocks referenced by the program.
- Reduce impact by

Conflict Misses:

- Caused by multiple addresses mapping to the same cache block.
- Reduce impact by

Dealing With Cache Misses with Hardware Design

Compulsory Misses:

- Caused by **cold** starts, process migrations or very first references.
- Reduce impact by increasing block size. But this causes increased miss penalty and could increase miss rate.

Capacity Misses:

- Caused by the cache becoming full; it cannot hold all blocks referenced by the program.
- Reduce impact by increasing cache size (but may increase access time.

Conflict Misses:

- Caused by multiple addresses mapping to the same cache block.
- Reduce impact by increasing associativity or increasing cache/block size (but may increase access time)
  - ↳ Larger cache $\Longrightarrow$ more sets $\Longrightarrow$ fewer addresses map to same loc.

Outline

Reducing effects of Cache Miss

Reducing Cache Miss Rate $\Longrightarrow$ increase cache size

- With increasing technology (especially transistor size and density) there is more room for larger caches.

Reducing Cache Miss Penalty $\Longrightarrow$ use more levels of cache

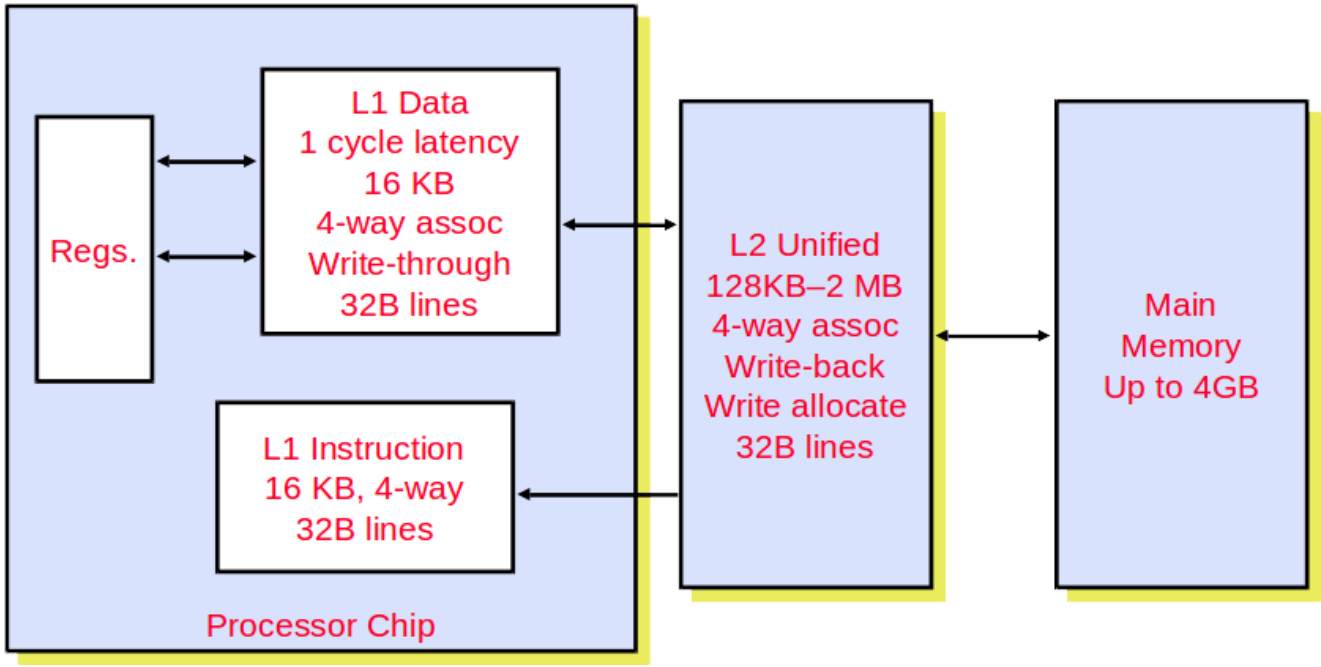
- L1 cache around for a *long* time.
- L2 cache first appeared with Intel's Celeron processors with only 128KB (1999).
- L3 was first seen in 2003 but prohibitively expensive. Became standard with Intel's Nehalem in 2008.

Recall: New AMAT 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 Memory Hierarchy



Multilevel Cache Design

Design considerations for L1 and L2 caches are very different:

- Primary cache should focus on **minimizing hit time** in support of faster processor clock.
  - ↳ Smaller capacity with smaller block sizes.
- Secondary cache(s) should focus on **reducing miss penalty** for L1 cache by reducing miss rate to main memory.
  - ↳ Larger capacity 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.
  - ↳ L1 cache can be smaller and faster (but results in higher miss rate).
- Hit time is less important for L2 cache than miss rate.
  - ↳ But, L2 hit time determines L1's miss penalty.
  - ↳ Presence of L3 cache greatly improves the situation, allowing L2 miss rate to be *slightly* less important.

It's all a balancing act.

Improving Cache Performance (1/2)

AMAT = Time for a Hit + Miss Rate * Miss Penalty

(1) Reduce the **time to hit** in the cache.

- ↳ Smaller cache size, smaller block size.
- ↳ Direct-Mapped
- ↳ For writes, two possible strategies:
  - No-Write Allocate: no "hit" on cache, just write to write buffer. Makes subsequent reads tricky.
  - Write Allocate: avoid 2 cycles using write buffer to write to lower cache.

(2) Reduce the **miss rate**.

- ↳ Larger cache, Larger block size (16 - 64 bytes typical).
- ↳ More flexible placement (increase associativity).
- ↳ Use a **victim-cache** – a small buffer holding most recently discarded blocks.

Improving Cache Performance (2/2)

AMAT = Time for a Hit + Miss Rate * Miss Penalty

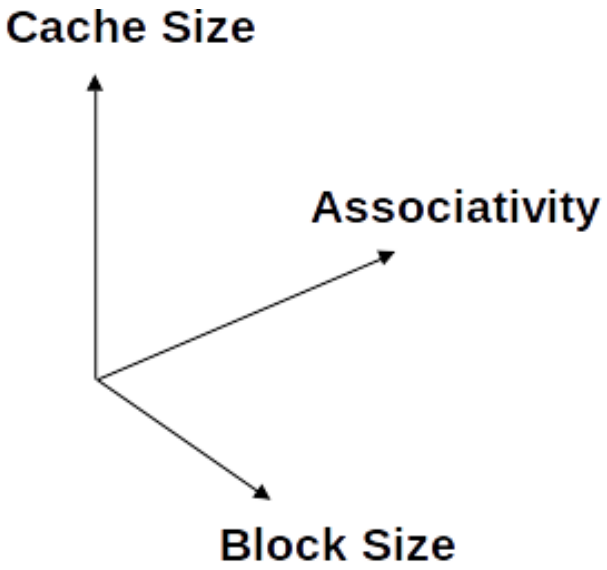
(3) Reduce the **miss penalty**

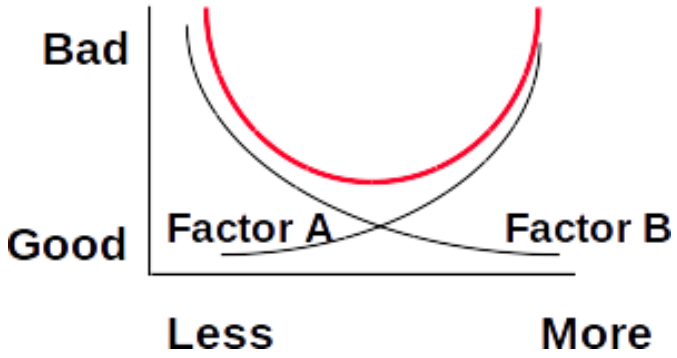
- ↳ Smaller blocks.
- ↳ Use a write-buffer.
- ↳ Check the write-buffer (or the victim-cache) on a read miss: *luck!*
- ↳ Pre-fetch critical word first, then rest of cache block.
- ↳ Use multiple cache levels.
- ↳ Faster backing store (= main memory).
- ↳ Improved **memory bandwidth** – amount and speed of memory transfer between levels (e.g. wider buses).

The Cache Design Space

It's all a balancing act.

- 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 (what the program is doing).
  - depends on technology / cost.
- *Simplicity* often wins.





Memory Hierarchy: Critical Aspects

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 types of **cache misses**:

- **Compulsory/Cold Misses**: Sad facts of life. The *first* reference to an address/block.
- **Conflict misses**: Increase cache size and/or associativity. **Thrashing!**
- **Capacity misses**: It turns out size does matter :(

Cache Design Space

- Total size, Block size, Associativity (& Replacement policy).
- Write-hit policy: **write-through**, **write-back**
- Write-miss policy: **(no)-write-allocate**. Use write buffers.

Questions to Consider

Q1 Where can an entry be placed or found in cache?

↳ Cache Organization, Direct-Mapping, Associativity.

Q2 Which entry should be replaced on a miss?

↳ Replacement Policies: LRU, FIFO.

Q3 What happens on a write?

↳ Write hit/miss strategies: write-through, write-back, write-allocate.

Typical Example: Given list of memory references describe the cache's state after each reference.

Q1: Where can an entry be placed?

	# of sets	Entries per set
Direct mapped	# of cache lines	1
Set associative	(# of cache lines) / 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	# of entries

Q2: 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)
  - ↳ FIFO (First In First Out)
- For 2-way set associative LRU is easy and ideal.
- Comparisons required for LRU can be too costly for high levels of associativity ($>$ 4-way).