CSC 112: Computer Operating Systems
Lecture 7

Cache

Department of Computer Science,

Hofstra University

Agenda

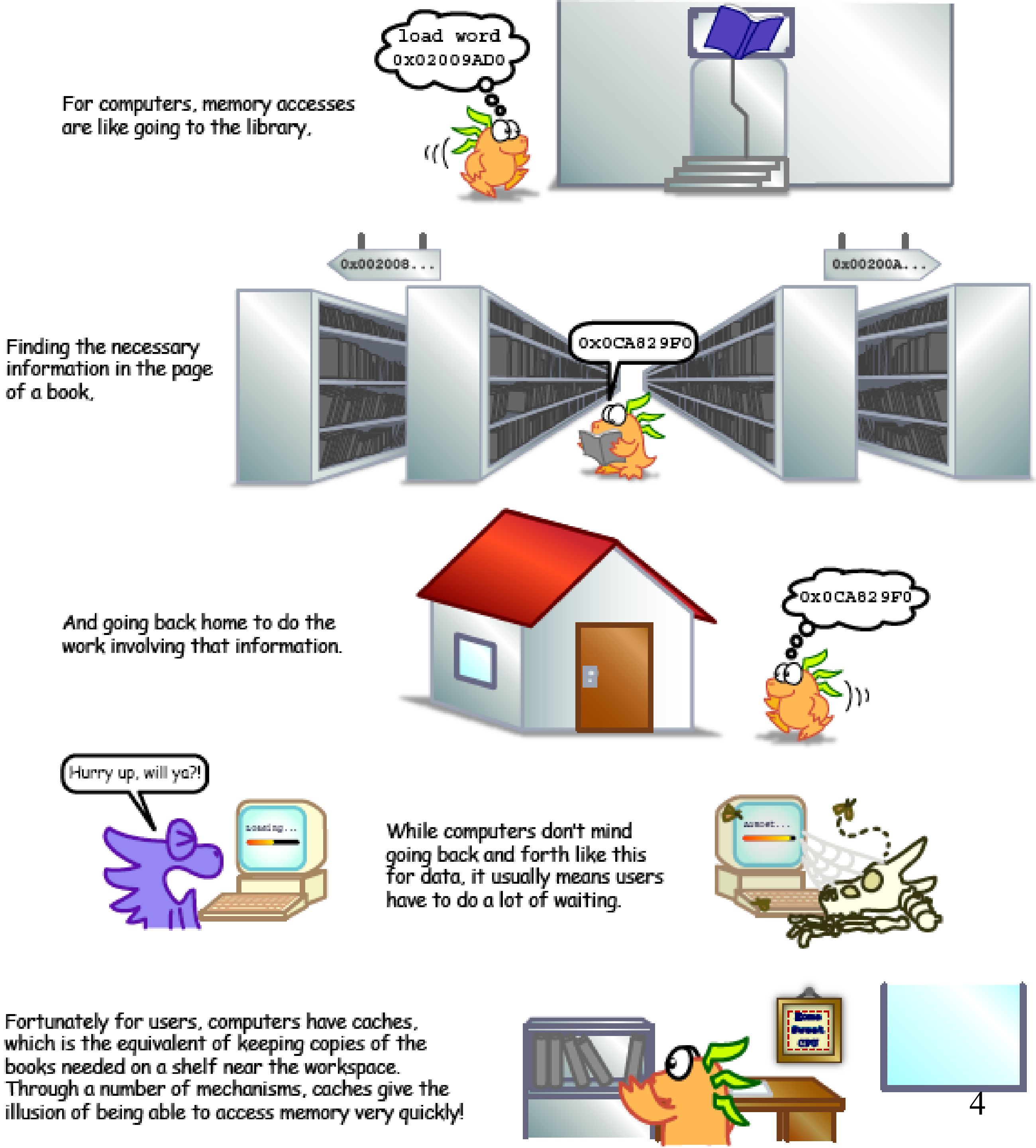
- Memory Hierarchy
- Cache Performance
- Direct Mapped Caches
- Set Associative Caches
- Multiprocessor Cache Consistency

Storage in a Computer

- Processor
  - – holds data in register file (~100 Bytes)
  - – Registers accessed on sub-nanosecond timescale
- Memory (we'll call "main memory")
  - – More capacity than registers (~Gbytes)
  - – Access time ~50-100 ns
  - – Hundreds of clock cycles per memory access?!

Library Analogy

- Writing a report on a specific topic.
- While at library, check out books and keep them on desk.
- If need more, check them out and bring to desk.
  - But don’t return earlier books since might need them
  - Limited space on desk; Which books to keep?
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in the library


For computers, memory accesses are like going to the library.
load word 0x02009AD0
0x002008...
0x00200A...
Finding the necessary information in the page of a book.
0x0CA829F0
And going back home to do the work involving that information.
0x0CA829F0
Hurry up, will ya?!
While computers don't mind going back and forth like this for data, it usually means users have to do a lot of waiting.
Fortunately for users, computers have caches, which is the equivalent of keeping copies of the books needed on a shelf near the workspace. Through a number of mechanisms, caches give the illusion of being able to access memory very quickly!

Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time
- *Temporal Locality* (locality in time)
  - If a memory location is referenced then it will tend to be referenced again soon
- *Spatial Locality* (locality in space)
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

Does Locality Exist?

- Instruction accesses spend a lot of time on the same page (since accesses mostly sequential, loop body tends to be small)
- Data accesses have less locality, but still some, depending on application…
- Stack accesses (push, pop…) have definite locality of reference

How does hardware exploit principle of locality?

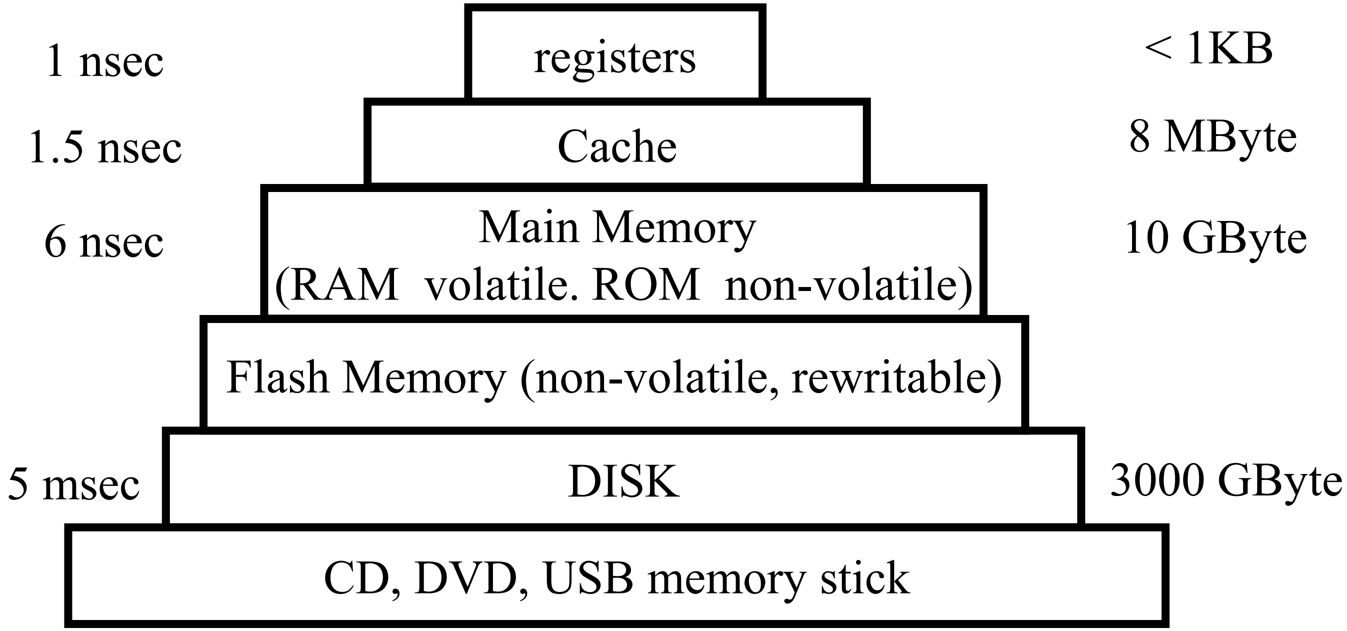
- Offer a hierarchy of memories where
  - closest to processor is fastest
    (and most expensive per bit so smallest)
  - furthest from processor is largest
    (and least expensive per bit so slowest)
- Goal is to create illusion of memory almost as fast as fastest memory and almost as large as biggest memory of the hierarchy
  - Processor and memory speed mismatch leads to the addition of *cache*

Memory Hierarchy

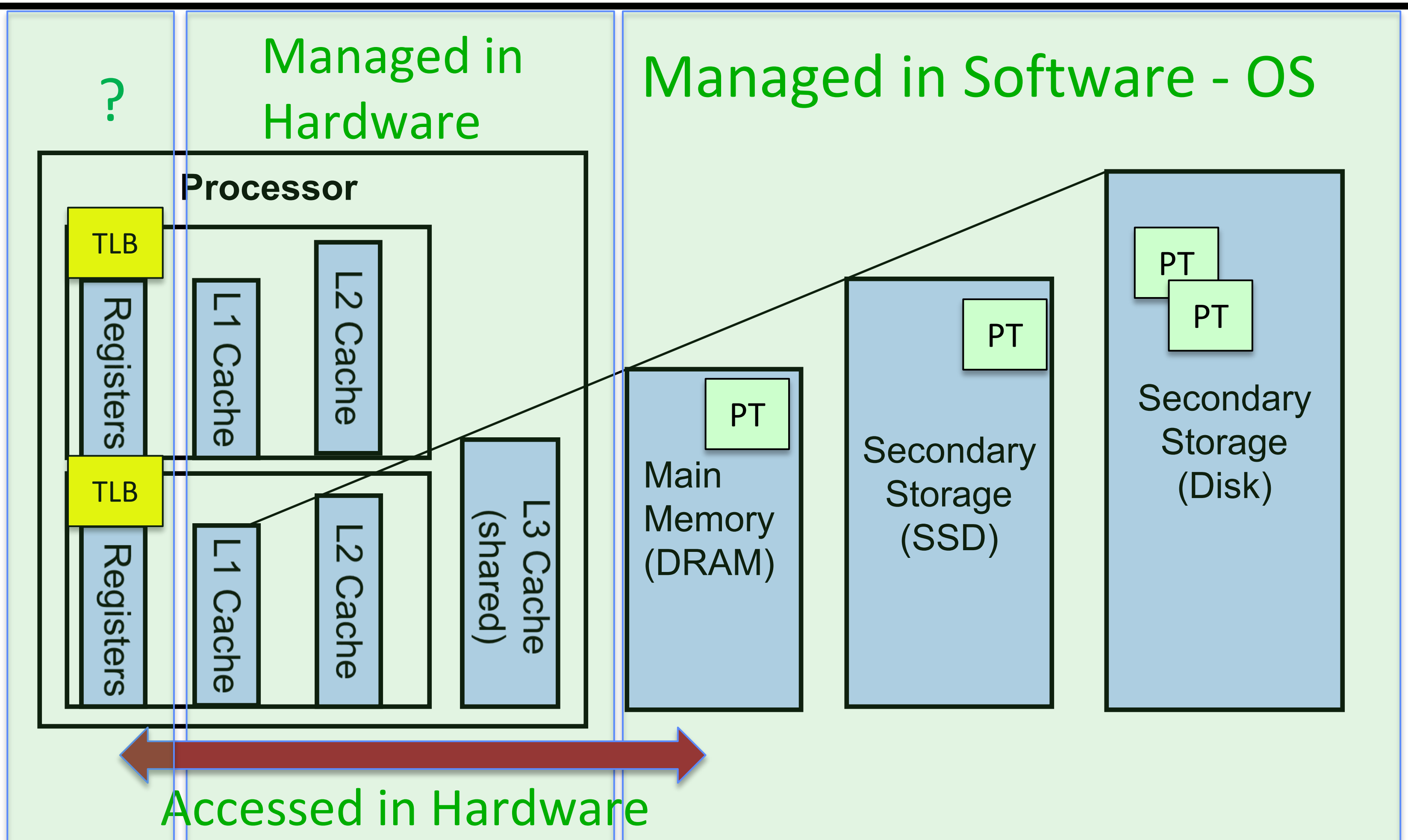
- Wanted: size of the largest memory available, speed of the fastest memory available
- Approach: Memory Hierarchy
  - Successively higher levels contain “most used” data from next lower level
  - Exploit temporal & spatial locality to present programmer with as much memory as is available in the cheapest technology at the speed offered by the fastest technology
- As we go down the memory hierarchy
  - Decreasing cost per bit, Increasing capacity
  - Increasing access time, Decreasing frequency of access

– The fastest memory is usually volatile (loses information when the power is off). Examples: register, cache, main memory. Referred to as primary memory.

– Non-volatile memory (continues to store information when the power is off) is usually slower. Examples: flash memory, internal and external hard drives, SSD, tape, etc. Referred to as secondary or auxiliary memory.


1 nsec
registers
< 1KB
1.5 nsec
Cache
8 MByte
6 nsec
Main Memory
(RAM volatile. ROM non-volatile)
10 GByte
Flash Memory (non-volatile, rewritable)
5 msec
DISK
3000 GByte
CD, DVD, USB memory stick

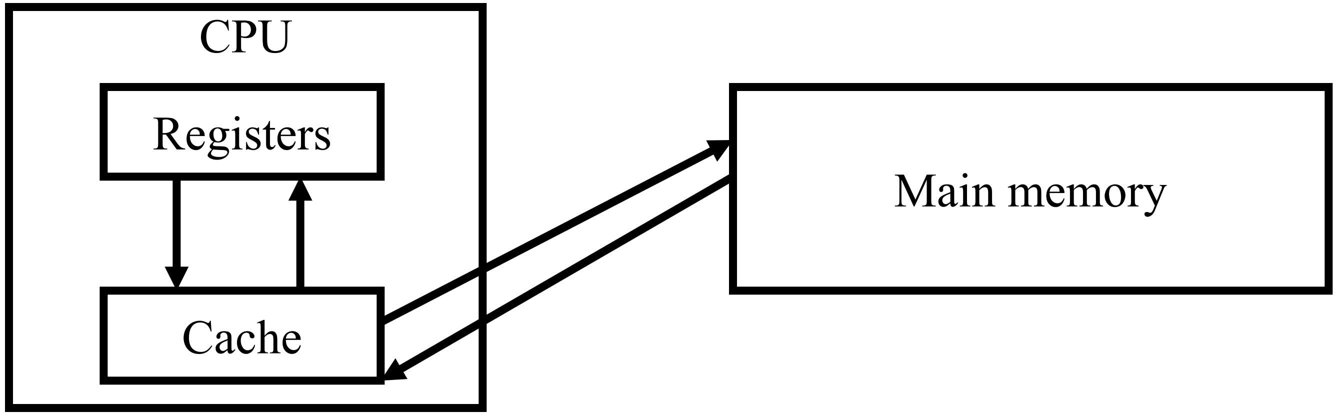
Typical Memory Hierarchy



Speed (ns):	**0.3**	**1**	**3**	**10-30**	**100**	**100,000** **(0.1 ms)**	**10,000,000** **(10 ms)**
Size (bytes):	**100Bs**	**10kBs**	**100kBs**	**MBs**	**GBs**	**100GBs**	**TBs**

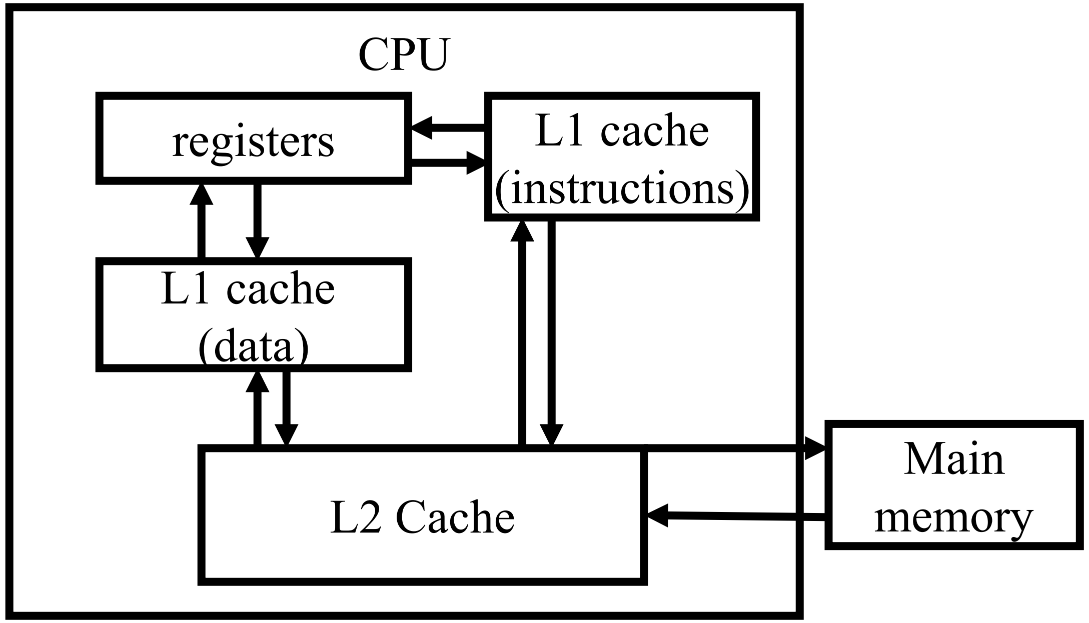
Registers and Cache

- CPU registers are on-chip resources, with access speed comparable to CPU clock speed
  - Size 64 bits for 64-bit machines, size 32 bits for 32-bit machines, etc.
- Cache may be on-chip or off-chip. It is slower and larger than registers, but much faster than main memory
  - The next few instructions, and data that will be needed will be loaded into cache in anticipation of faster access by the processor.
  - Cache contains a subset of content of main memory
- As a pun, often use $ (“cash”) as abbreviation for cache, e.g. D$ = Data Cache, I$ = Instruction Cache

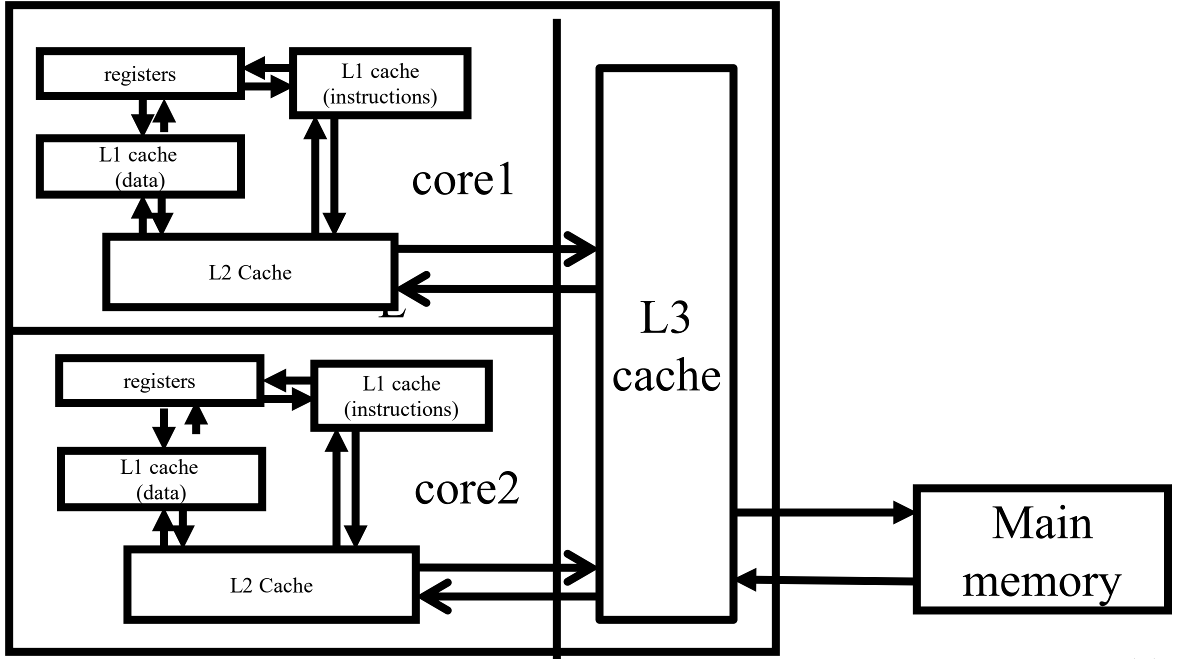


Cache Architectures

- Modern processors have an on-chip cache called an L1 cache
  - Typically separate L1 caches for instructions and data
- Most processors have multiple levels of caches between the L1 cache and the main memory with different sizes, e.g., 2nd and 3rd level of cache, called the L2 (L3) cache.
  - L2 cache can be on-chip or off-chip (connected to the CPU via a bus)
  - L3 cache is typically off-chip
  - Unified cache (containing both instructions and data)



Example cache config (uniprocessor)



Example cache config (multicore processor)

Memory Hierarchy Technologies

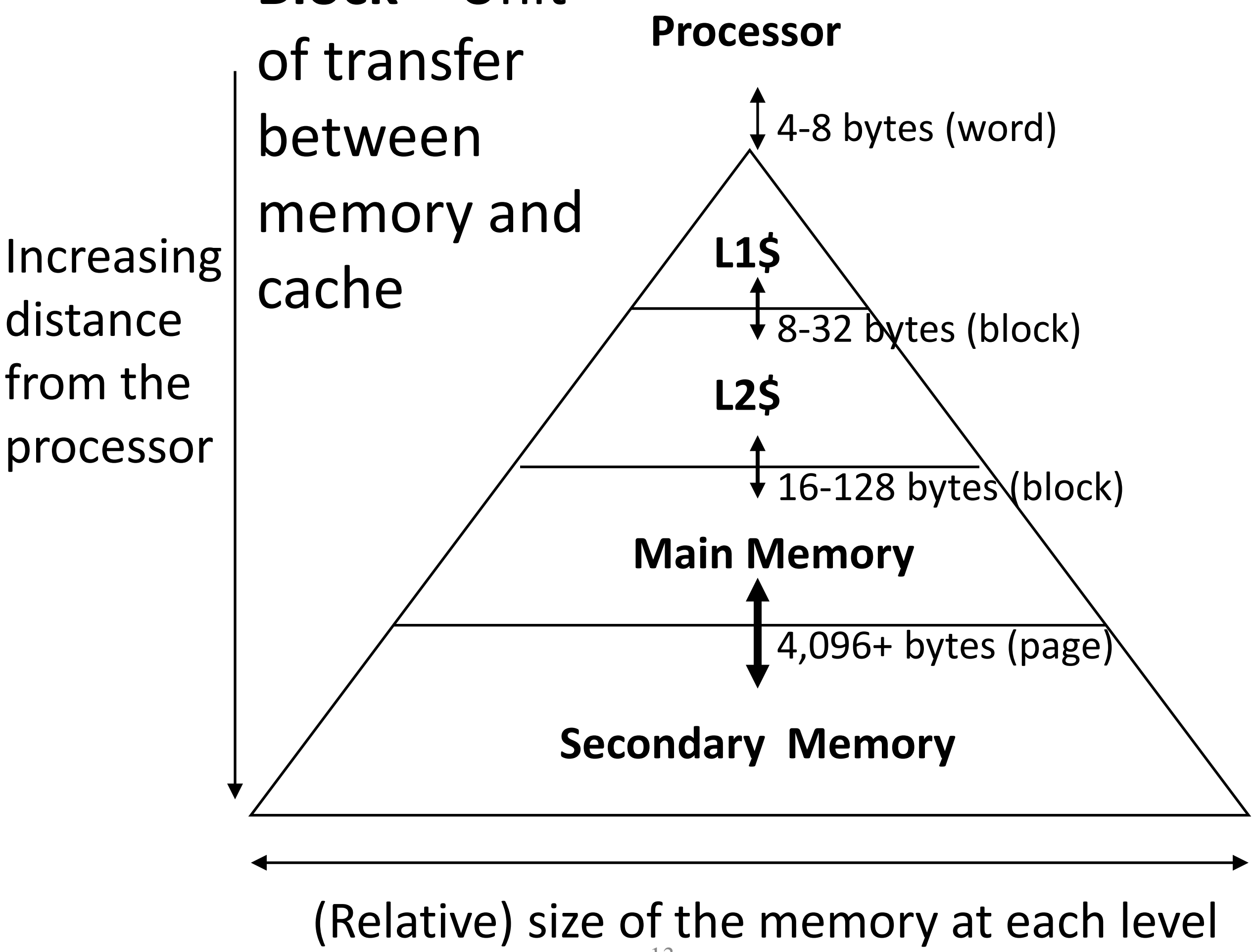
- Caches use SRAM (Static RAM) for speed and technology compatibility
  - Fast (typical access times of 0.5 to 2.5 ns)
  - Low density (6 transistor cells), higher power, expensive
  - Static: content will last as long as power is on
- Main memory uses DRAM (Dynamic RAM) for size (density)
  - Slower (typical access times of 50 to 70 ns)
  - High density (1 transistor cells), lower power, cheaper ($20 to $40 per GB in 2011)
  - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
    - Consumes 1% to 2% of the active cycles of the DRAM

Cache vs RAM: The Speed Difference Explained
https://www.youtube.com/watch?v=ammQYLcyebA&list=PL38NNHQLqJqYnNrTenxBvGJSPCkV9EOWk&index=5

How is the Hierarchy Managed?

- Registers ↔ memory
  - By compiler (or assembly level programmer)
- Cache ↔ main memory
  - Cache has a number of slots, each N Byes long, and can hold one N-Byte cache block (also called a cache line)
  - The cache controller hardware moves instructions (and data) between main memory and cache in blocks
  - Each time a copy of a new cache block is loaded into a cache slot, the original content of the block at that address is overwritten
- Main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
    - Virtual to physical address mapping with page tables
  - By the programmer (files)

Block – Unit of transfer between memory and cache

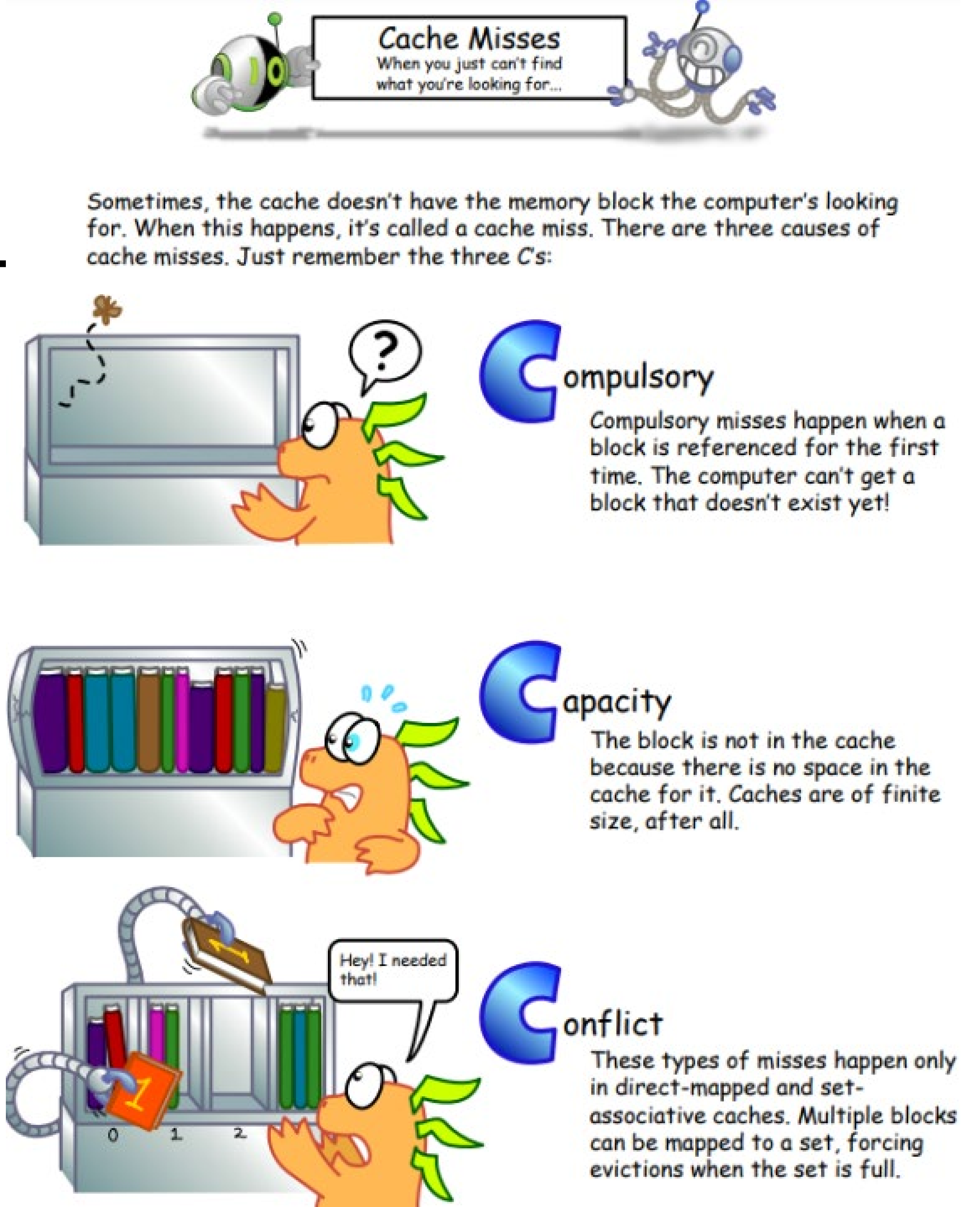


Caching Hit & Cache Miss

- When accessing a memory address:
  - cache hit: the requested memory address is found in the cache, i.e., cache block is valid and contains the proper memory address, so read from cache (fast). Proportion of accesses that are hits is called the hit rate
  - cache miss: the requested memory address is not in the cache, i.e., cache block is invalid, or refers to the wrong memory address, so read from memory (slow). Proportion of accesses that are misses is called the miss rate = 1 – hit rate

Sources of Cache Misses:3Cs

- Compulsory misses (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity misses:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- Conflict misses (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)



Cache Misses

When you just can't find what you're looking for...

Sometimes, the cache doesn't have the memory block the computer's looking for. When this happens, it's called a cache miss. There are three causes of cache misses. Just remember the three C's:

Compulsory

Compulsory misses happen when a block is referenced for the first time. The computer can't get a block that doesn't exist yet!

Capacity

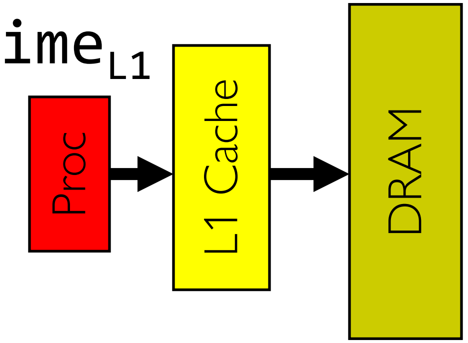
The block is not in the cache because there is no space in the cache for it. Caches are of finite size, after all.

Conflict

These types of misses happen only in direct-mapped and set-associative caches. Multiple blocks can be mapped to a set, forcing evictions when the set is full.

Average Memory Access Time (AMAT)

$\text{AMAT} = \text{Hit Rate}_{L1} \times \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Time}_{L1}$



$\text{Hit Rate}_{L1} + \text{Miss Rate}_{L1} = 1$

Hit Time_{L1} = Time to get value from L1 cache.

$\text{Miss Time}_{L1} = \text{Hit Time}_{L1} + \text{Miss Penalty}_{L1}$

Miss Penalty_{L1} = AVG Time to get value from lower level (DRAM)

So, $\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$

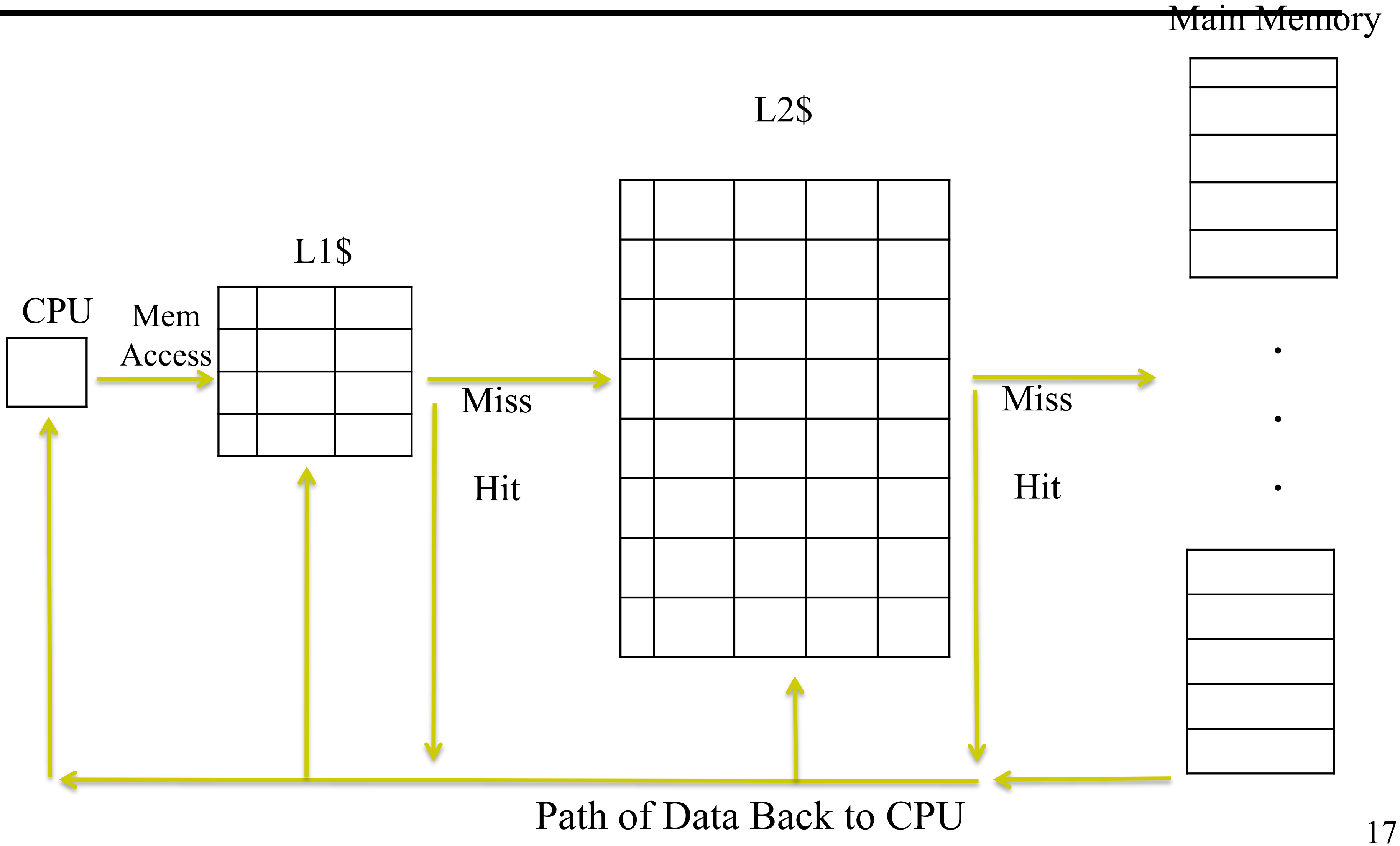
Example: Assume one level of cache. One cache access takes 1 clock cycle, i.e., Hit Time = 1 cycle. One main memory access takes 100 cycles. Any instruction or data not in cache will have to be accessed in main memory and moved to cache before being accessed: i.e., Miss penalty = 100 cycles (main memory access time), Miss Time =100+1=101 cycles (main memory access time + cache access time)

For miss rate of x, AMAT is a linearly increasing function with x:

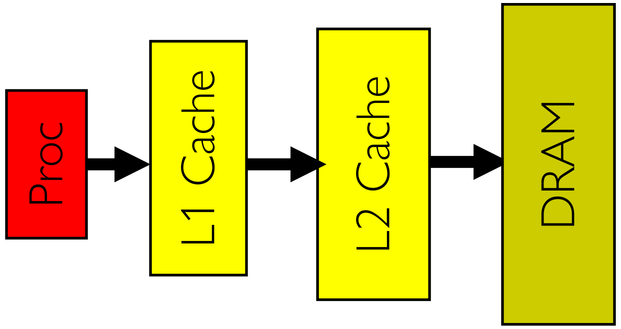
AMAT = (1-x)*1 + x*101 = 1 + x*100 cycles

Multiple Cache Levels

- How to reduce Miss Penalty?
- Multiple cache levels to reduce expensive main memory accesses



AMAT for Multiple Levels of Caches



Proc

L1 Cache

L2 Cache

DRAM

AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

Miss Penalty$_{L1}$ = AVG time to get value from lower level (L2)
= Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$

Miss Penalty$_{L2}$ = Average Time to fetch from below L2 (DRAM)

AMAT = Hit Time$_{L1}$ +
$\underline{\text{Miss Rate}_{L1}}$ x (Hit Time$_{L2}$ + $\underline{\text{Miss Rate}_{L2}}$ x Miss Penalty$_{L2}$)

And so forth (final miss penalty is Main Memory access time).

Example: L1 Hit Time 1 cycle, L1 Miss Rate 0.02, L2 Hit Time 5 cycles, L2 Miss Rate 0.05, Main Memory access time 100 cycles.

L1 Cache only, no L2: AMAT = 1 + 0.02*100 = 3 cycles

With L2 Cache: AMAT = 1 + 0.02*(5 + .05*100) = 1.2 cycles

Cache Design Considerations

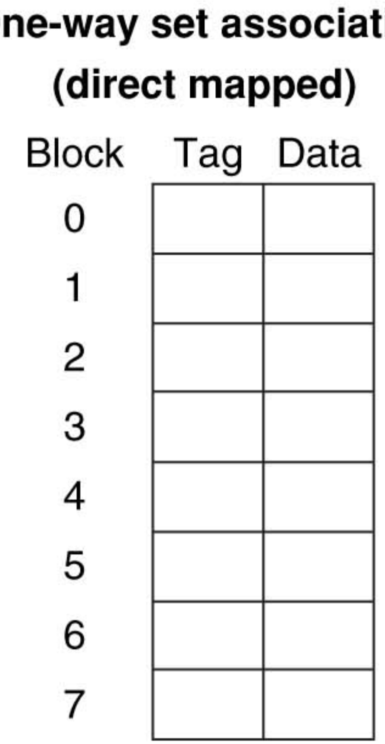
- Different design considerations for L1$ and L2$
  - L1$ focuses on fast access: minimize hit time to achieve shorter clock cycle, e.g., smaller $.
    - Since miss penalty of L1$ is significantly reduced by presence of L2$, so can be smaller/faster even with higher miss rate
  - L2$ (and L3$) focus on low miss rate: reduce penalty of long main memory access times: e.g., Larger $ with larger block sizes/higher levels of associativity
    - Since fast hit time is less important than low miss rate due to high miss penalty
- Global miss rate – the fraction of references that miss in all levels of a multilevel cache. It dictates how often we must access the main memory.

Cache Associativity

- *Direct Mapped (DM)*: memory block maps to exactly one cache block
- *Fully Associative (FA)*: allow a memory block to be mapped to any cache block
- *n-way Set Associative (SA)*: divide cache into sets, each of which consists of n addresses ("ways") to place memory block
  - Cache set index=(block address) modulo (# sets in the cache)
  - A block can be placed in any of the n addresses (ways) of the cache set it belongs to.

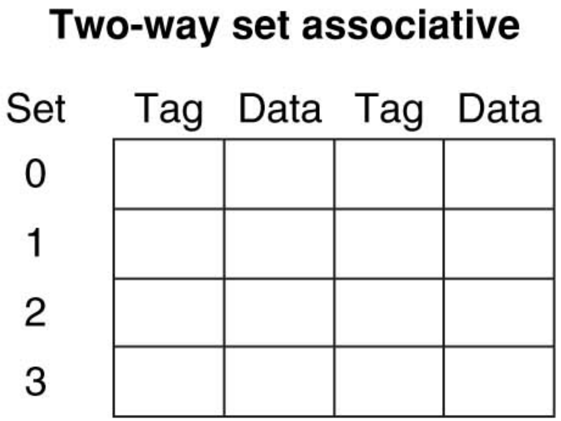
Example: 8-Block cache with different organizations

- Cache: 8 blocks, each 4 Bytes. # cache blocks is equal to number of sets x associativity.
- 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
- For fixed cache size, increasing associativity decreases number of sets while increasing number of blocks per set.



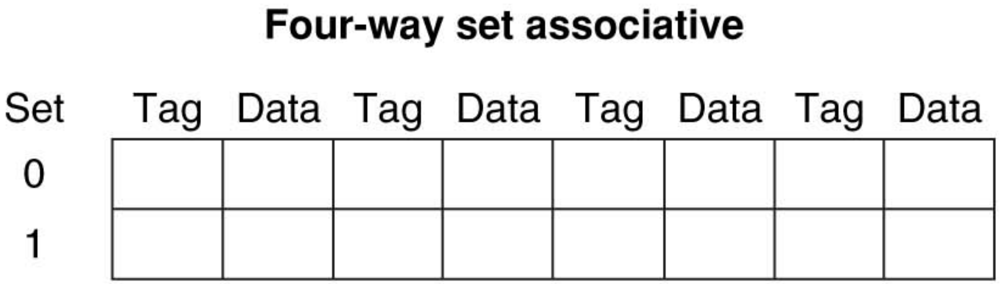
One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		



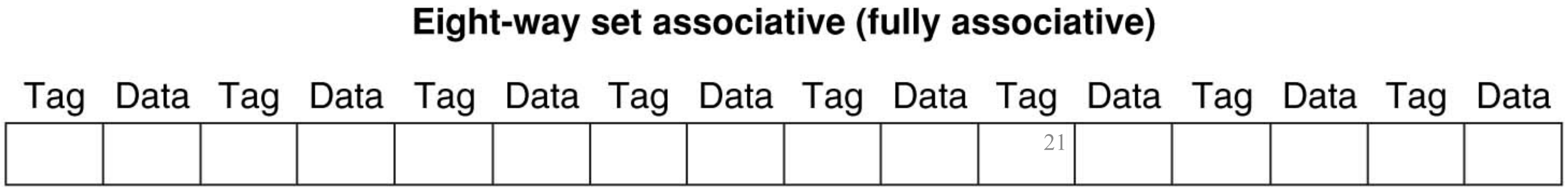
Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				



Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

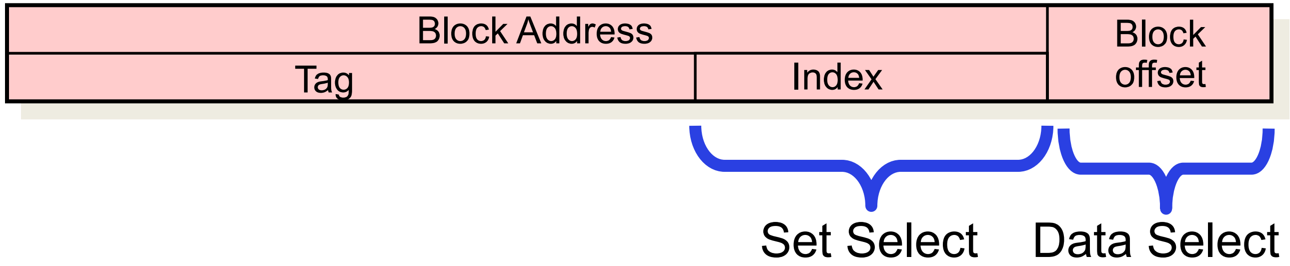


Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

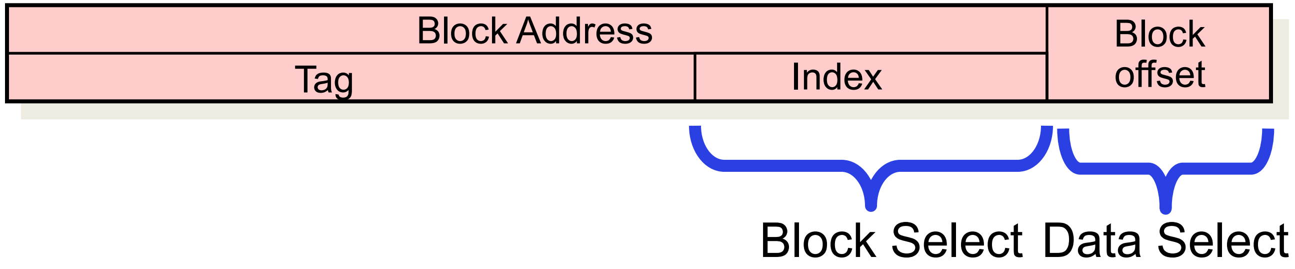
Tag, Index, Offset

- Index used to identify the cache set
  - Index = (memory *block* address (Tag+Index)) modulo (# *sets* in the cache)
  - I bits <=> 2^I sets in cache
- Tag used to distinguish between multiple memory blocks that map to a given set index
  - If no cache block in the set matches the tag, then declare cache miss.
- Offset used to select data within a block; *which byte within a block* is referenced?
  - O bits <=> 2^O bytes per block



DM cache: Tag, Index, Offset

- Each block in memory maps to one block in the cache.
- Index: which cache block (row in the cache) is the memory block mapped to?
- Tag: which block in memory did the cache block come from?
- Offset: which column of the cache block is the Byte stored?



Block Address

Tag

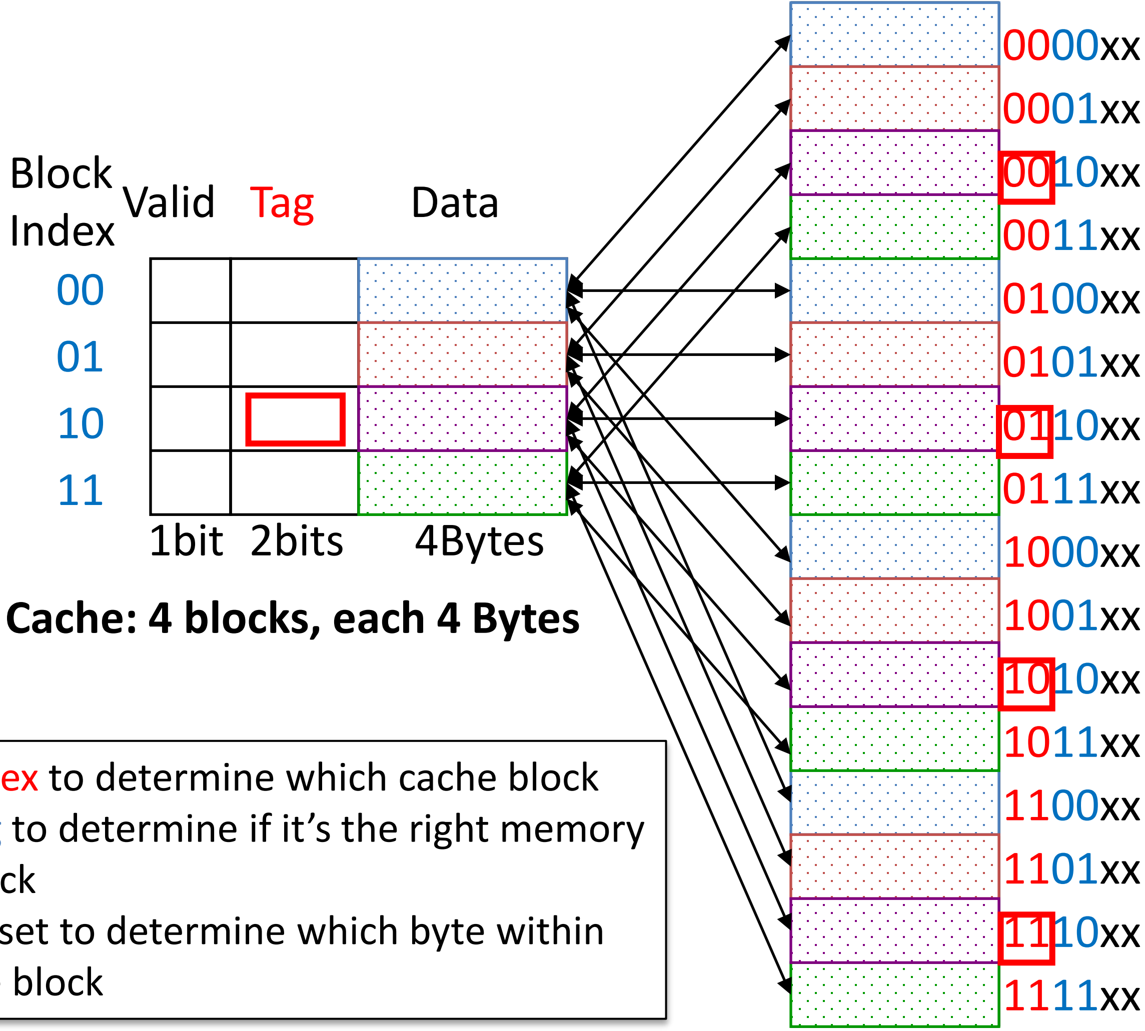
Index

Block offset

Block Select

Data Select

DM cache: Example



Cache: 4 blocks, each 4 Bytes

Main Memory - 6-bit addresses (16 blocks, each 4 Bytes)

Each memory block is mapped to exactly one block in the cache.

Tag: upper 2 bits → 4 memory addresses mapped to the same cache block index. Compare the cache tag to the high-order 2 memory address bits (Tag) to tell if the memory block is in the cache

Index: middle 2 bits → 4 blocks in cache; Index defines which cache block index the mem address is mapped to. Index = (block address) modulo (# blocks in the cache (4)).

Offset: lower 2 bits → 4 bytes (1 word) per block; Offset defines the byte within the cache block.

Index to determine which cache block
Tag to determine if it’s the right memory block
Offset to determine which byte within the block

Valid bit indicates whether an entry contains valid information – if the bit is not set, there cannot be a match for this block

Word vs. Byte

- Most modern CPUs, e.g., MIPS, operate with words (4 bytes), since it is convenient for 32-bit arithmetic. Entire words will be transferred to and from the CPU. Hence the only possible byte offsets are multiples of 4.
- When we use words as the unit of memory size, we leave out the last 2 bits of offset for byte addressing

DM cache: Example 2

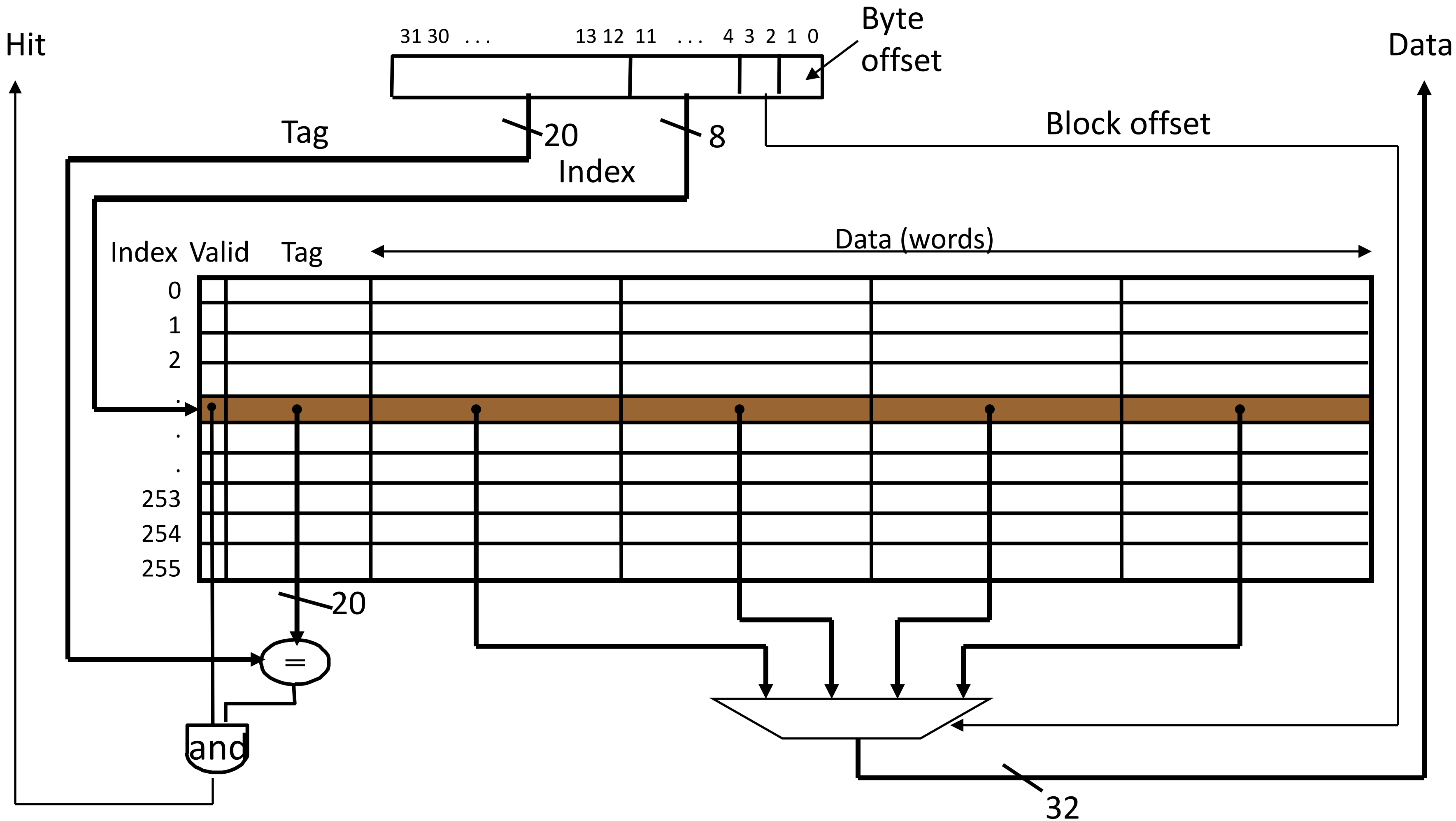
- 8-bit address space. Tag: 3 bits, Index: 2 bits, Offset: 3 bits
- Index 2 bits → 2^2=4 cache blocks
- Tag 3 bits → 2^3=8 memory addresses mapped to each cache index
- Offset 3 bits → 8 Bytes/block,
- Cache size = 4 blocks * 8 Bytes/block = 32 Bytes (8 words)

Byte Offset

Index	V	Tag	000	001	010	011	100	101	110	111
00			1 WORD				1 WORD			
01										
10										
11										

DM cache: Example 3

- 32-bit address space. Tag: 20 bits; Index: 8 bits; Offset: 4 bits
- Index 8 bits → 2^8=256 cache blocks
- Tag 20 bits → 2^20=1 million memory addresses mapped to each cache index
- Offset 4 bits → 2^4=16 Bytes (4 words) /block
- Cache size = 256 blocks * 16 Bytes/block = 4K Bytes (1K words)

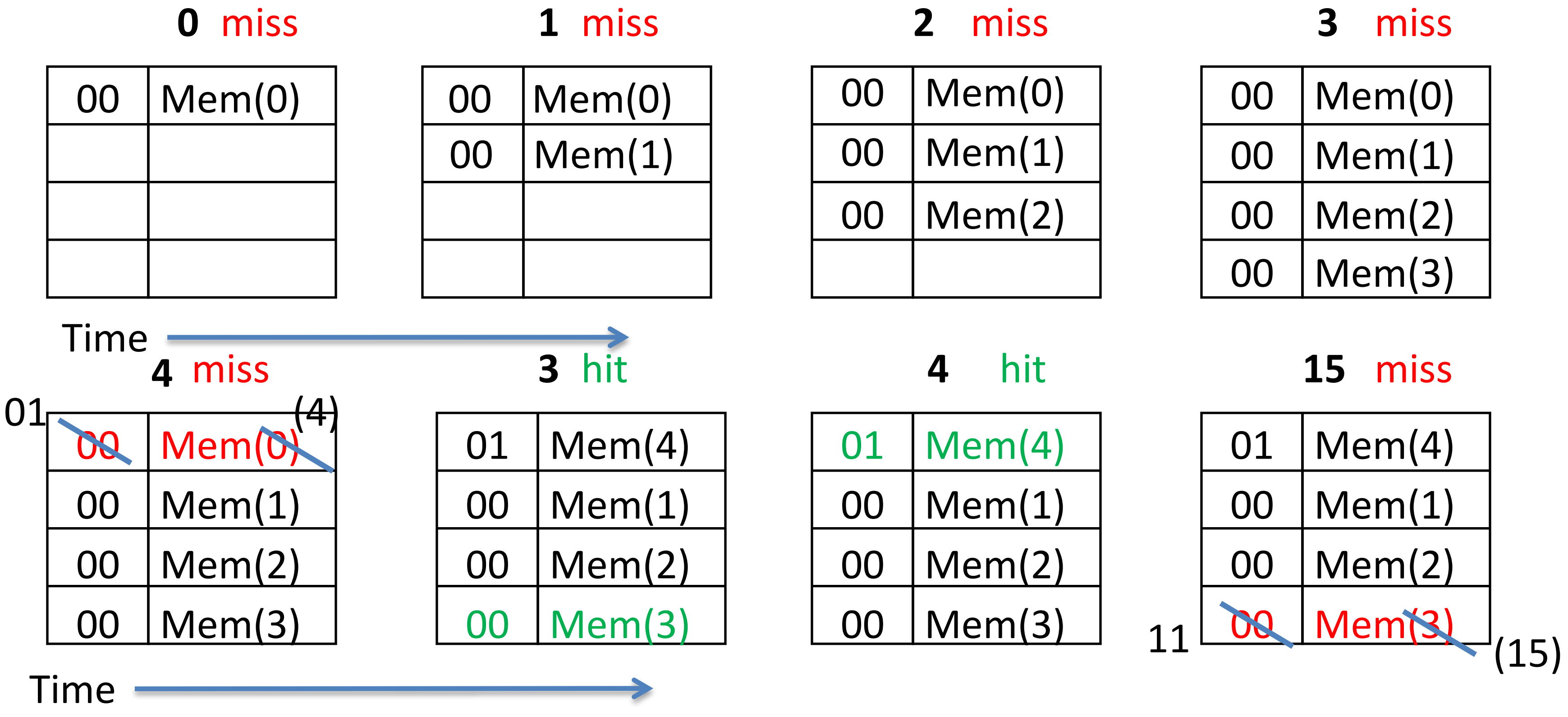


DM cache: Memory Access Example

- Consider the 6-bit memory address example; sequence of memory address accesses (Byte offset bits are ignored, so only 4-bit word addresses: Tag 2b; Index 2b)

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

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)

Time

4 miss

01 ~~00~~	~~Mem(0)~~ (4)
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)
11 ~~00~~	~~Mem(3)~~ (15)

Time

- 8 requests, 6 misses

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

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

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

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)

4 miss

01 ~~00~~	(5) ~~Mem(1)~~	~~Mem(0)~~ (4)
00	Mem(3)	Mem(2)

3 hit

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

4 hit

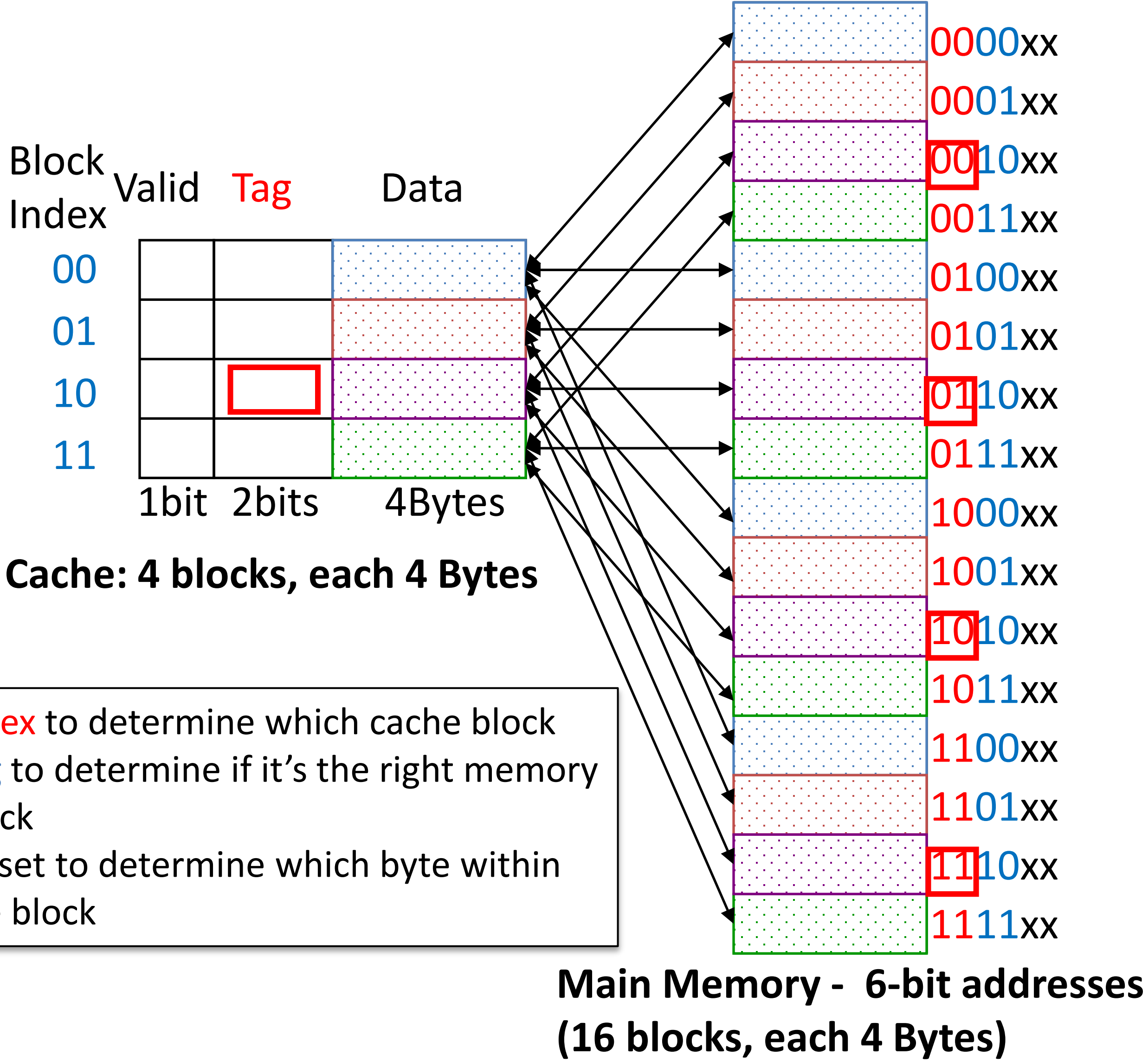
01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

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

- 8 requests, 4 misses

Recall: DM cache: Example



Cache: 4 blocks, each 4 Bytes

Main Memory - 6-bit addresses (16 blocks, each 4 Bytes)

Index to determine which cache block
Tag to determine if it's the right memory block
Offset to determine which byte within the block

Each memory block is mapped to exactly one block in the cache.

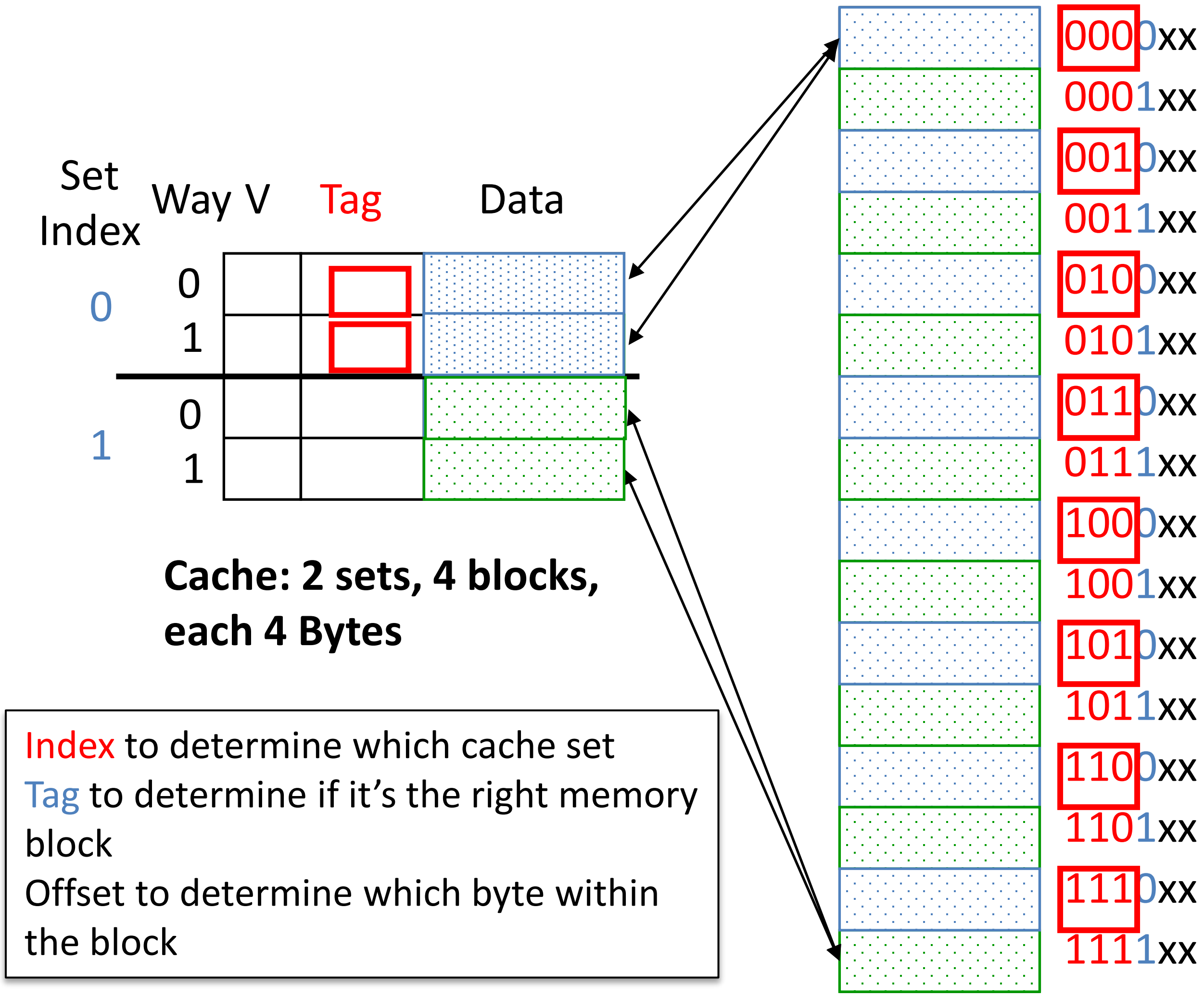
Tag: upper 2 bits → 4 memory addresses mapped to the same cache block index. Compare the cache tag to the high-order 2 memory address bits (Tag) to tell if the memory block is in the cache

Index: middle 2 bits → 4 blocks in cache; Index defines which cache block index the mem address is mapped to. Index = (block address) modulo (# blocks in the cache (4)).

Offset: lower 2 bits → 4 bytes (1 word) per block; Offset defines the byte within the cache block.

Valid bit indicates whether an entry contains valid information – if the bit is not set, there cannot be a match for this block

2-Way SA cache: Example



Cache: 2 sets, 4 blocks, each 4 Bytes

Main Memory - 6-bit addresses (16 blocks, each 4 Bytes)

Index to determine which cache set
Tag to determine if it’s the right memory block
Offset to determine which byte within the block

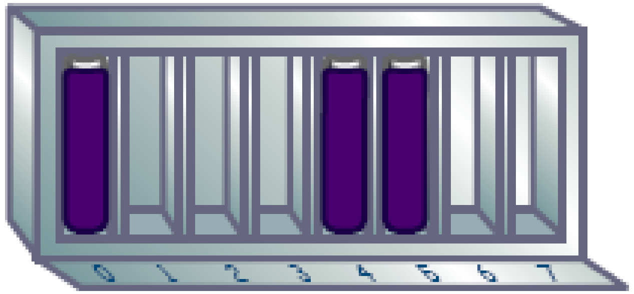
Each memory block is mapped to a cache set with two blocks.

Tag: upper 3 bits → 8 memory addresses mapped to the same cache set index. Compare all the cache tags in the set to the high-order 3 memory address bits (Tag) to tell if the memory block is in the cache

Index: middle 1 bit → 2 **sets** of blocks in cache; Index defines which cache set index the mem address is mapped to. Index = (block address) modulo (# sets in the cache (2)).

Offset: lower 2 bits → 4 bytes (1 word) per block; Offset defines the byte within the cache block.

Direct Mapped

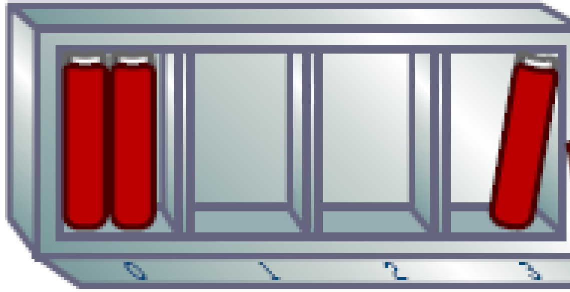


Memory Address

Tag	Index	Offset

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

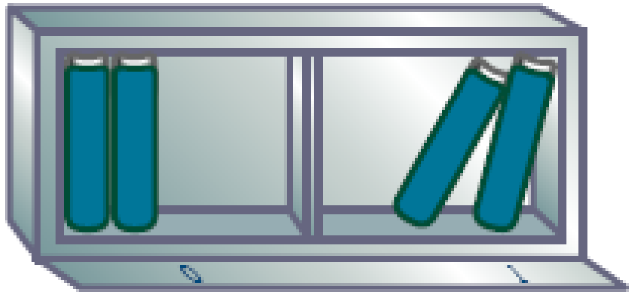
2-Way Set Associative



Tag	Index	Offset

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

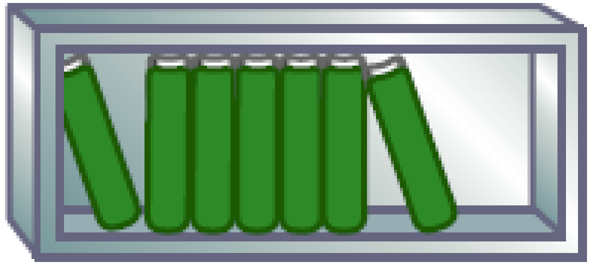
4-Way Set Associative



Tag	Index	Offset

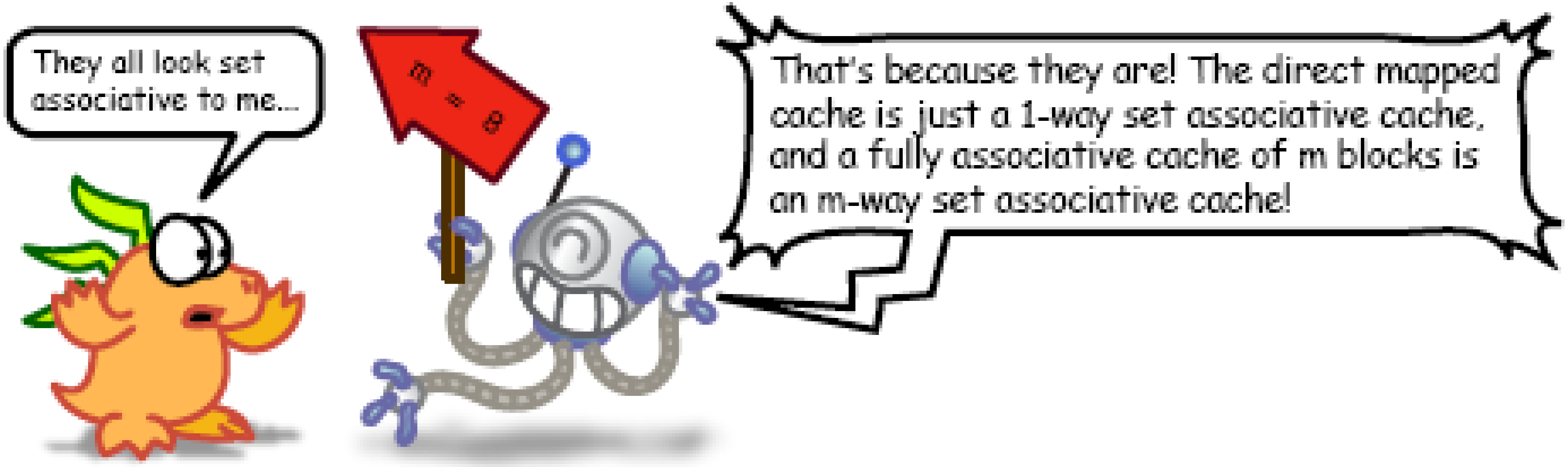
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

Fully Associative



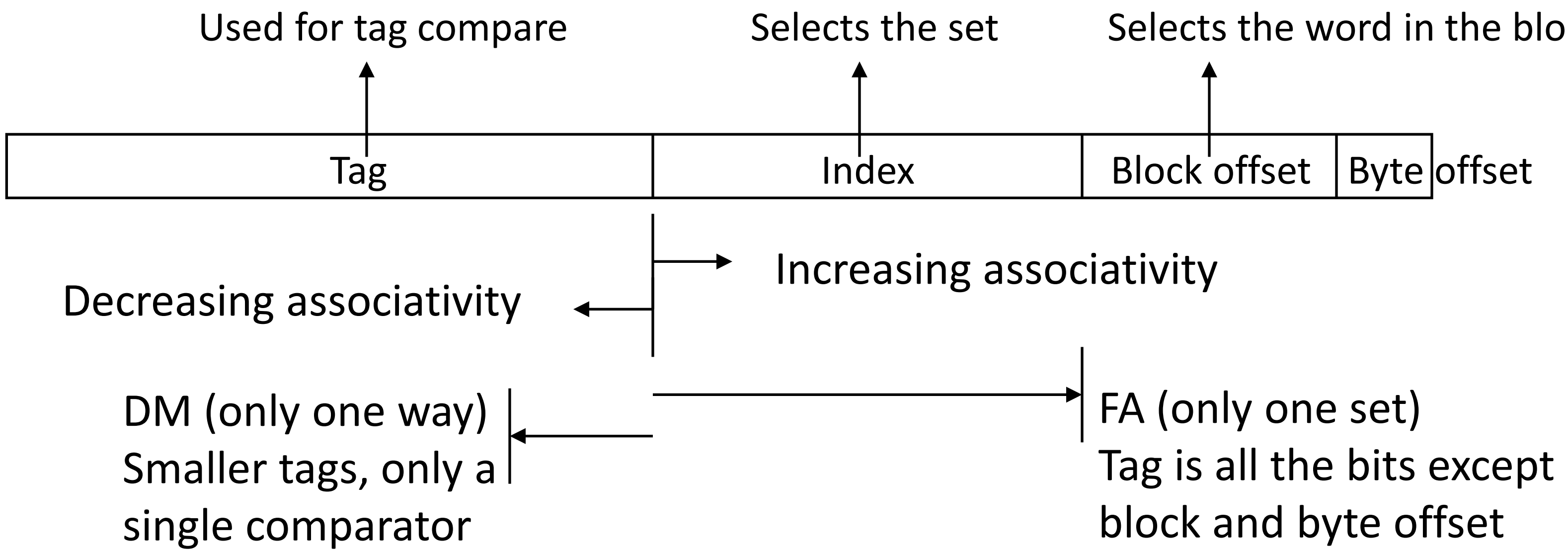
Tag	Offset

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!


They all look set associative to me...
m = 8
That's because they are! The direct mapped cache is just a 1-way set associative cache, and a fully associative cache of m blocks is an m-way set associative cache!

Range of SA Caches

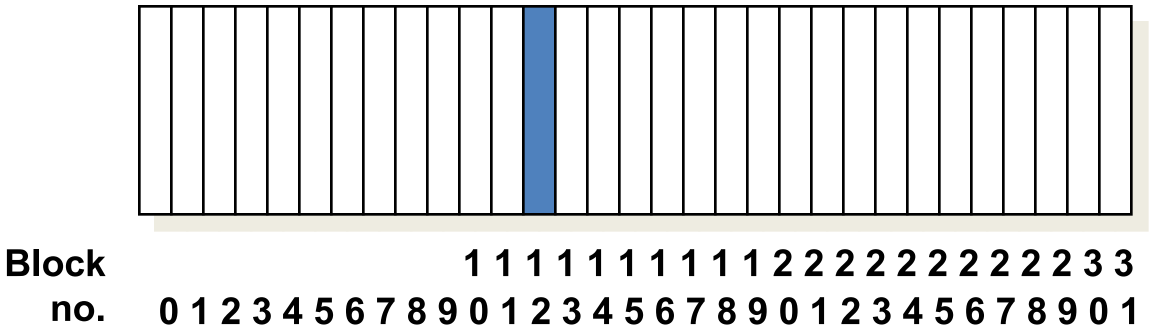
- For a fixed-size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Cache associativity example

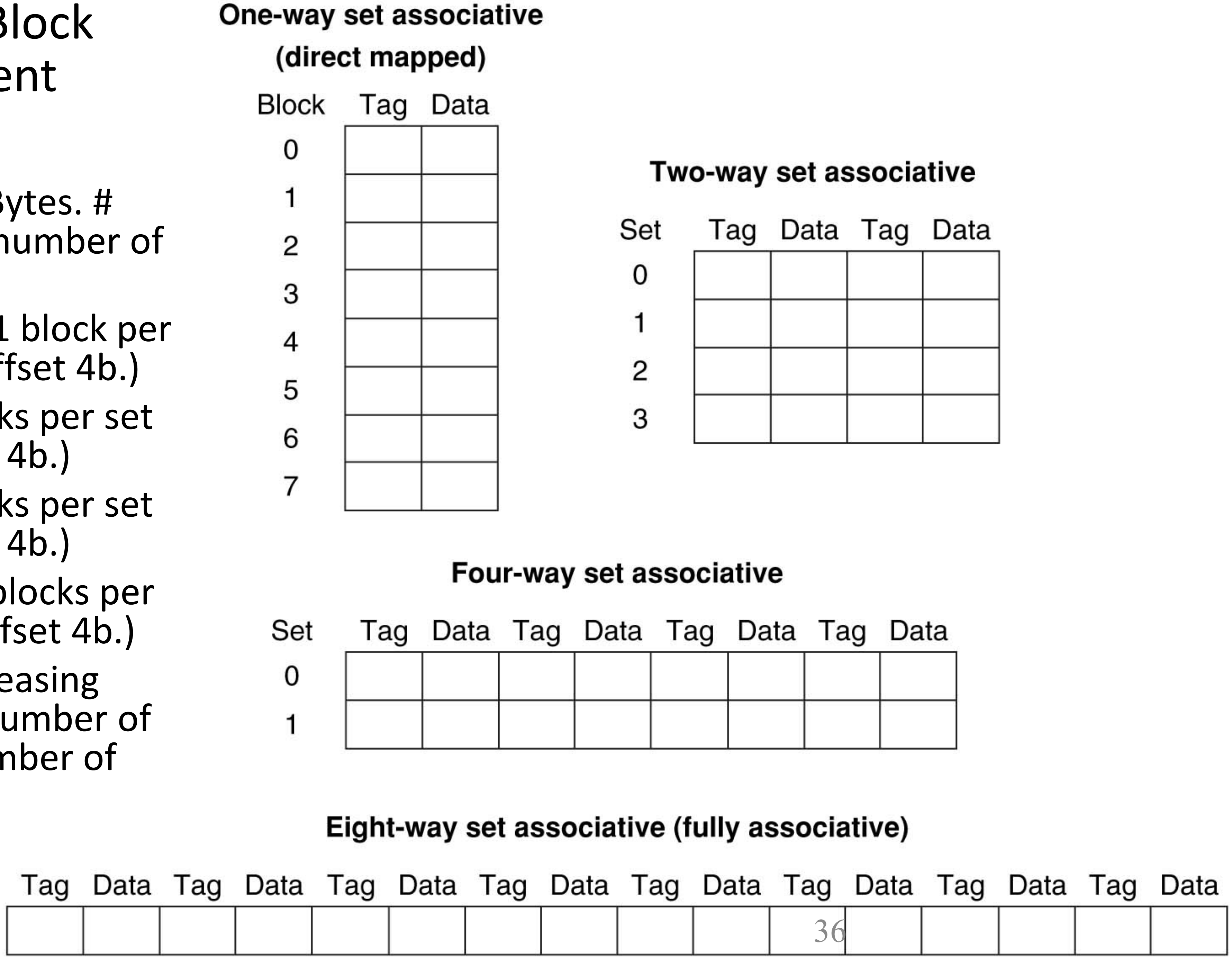
- Consider 5-bit memory address, with 2^5=32 memory blocks. Where can memory block 12 be placed in a cache with 8 blocks?

32-Block Address Space:

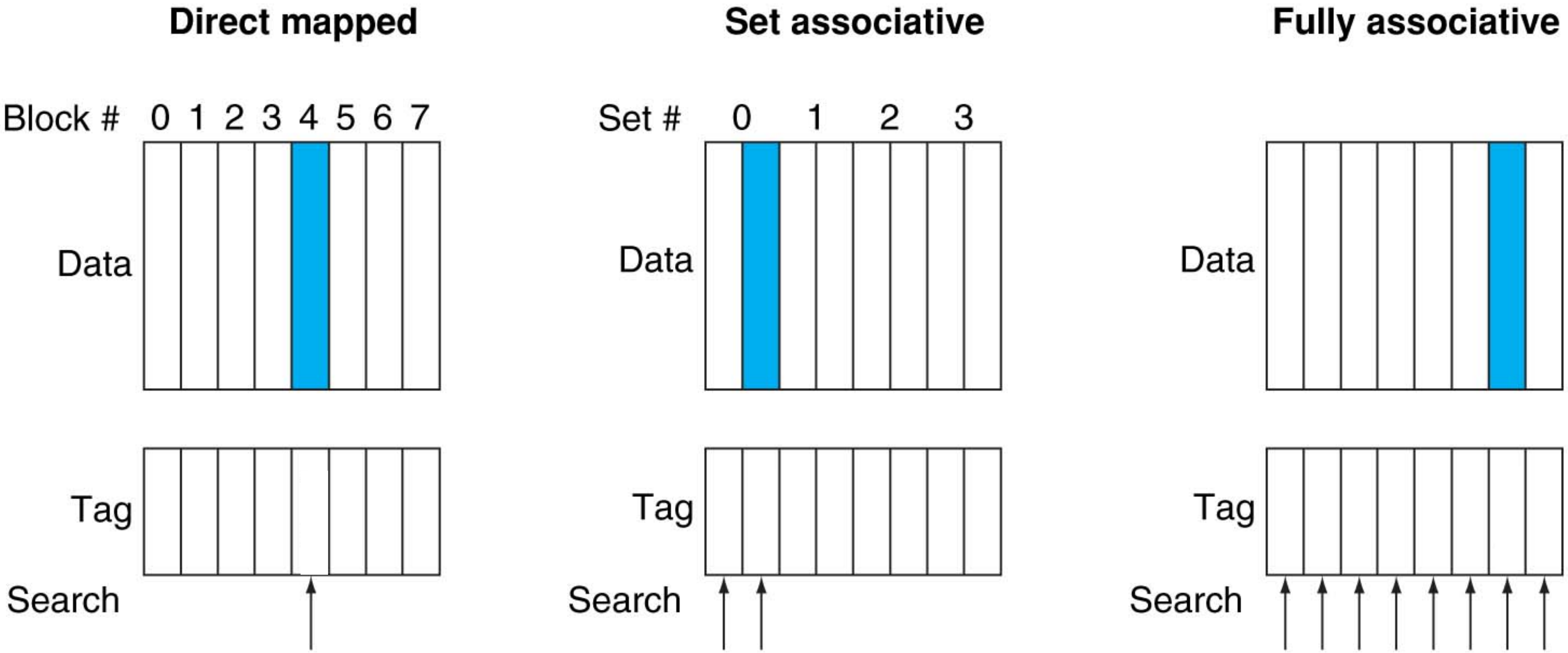


Recall Example: 8-Block cache with different organizations

- Cache: 8 blocks, each 4 Bytes. # cache blocks is equal to number of sets x associativity.
- DM (1-way SA): 8 sets x 1 block per set. (Tag 2b, Index 3b, Offset 4b.)
- 2-way SA: 4 sets x 2 blocks per set (Tag 3b, Index 2b, Offset 4b.)
- 4-way SA: 2 sets x 4 blocks per set (Tag 4b, Index 1b, Offset 4b.)
- FA (4-way SA): 1 set x 8 blocks per set (Tag 5b, Index 0b, Offset 4b.)
- For fixed cache size, increasing associativity decreases number of sets while increasing number of blocks per set.


One-way set associative
(direct mapped)
Block Tag Data
0
1
2
3
4
5
6
7
Two-way set associative
Set Tag Data Tag Data
0
1
2
3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0
1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

Cache associativity example con’t



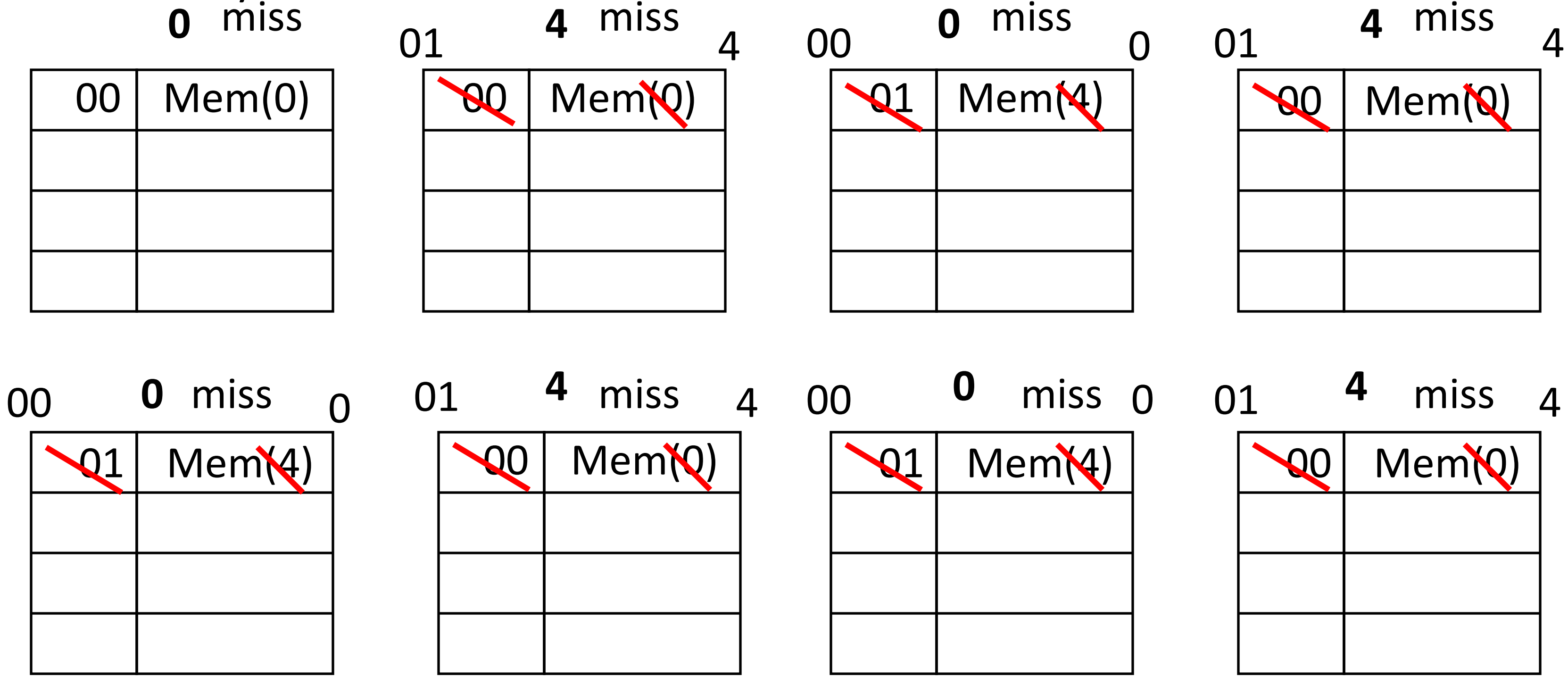
- DM: 8 sets x 1 block per set. Mem block 12 is always mapped to cache block index = 12 mod 8 = 4.
- 2-way SA: 4 sets x 2 blocks per set. Mem block 12 is mapped to cache set index = 12 mod 4 = 0; can be placed in either one of the 2 blocks in the mapped set.
- FA: 1 set x 8 blocks per set. Mem block 12 can be placed in any cache block.

Example: 4-Word DM cache, worst-case memory reference sequence

- Consider the sequence of memory accesses

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

0(0000) 4(0100) 0 4 0 4 0 4



0 miss

00 | Mem(0)

01 **4** miss 4

~~00~~ | Mem(~~0~~)

00 **0** miss 0

~~01~~ | Mem(~~4~~)

01 **4** miss 4

~~00~~ | Mem(~~0~~)

00 **0** miss 0

~~01~~ | Mem(~~4~~)

01 **4** miss 4

~~00~~ | Mem(~~0~~)

00 **0** miss 0

~~01~~ | Mem(~~4~~)

01 **4** miss 4

~~00~~ | Mem(~~0~~)

- 8 requests, 8 misses

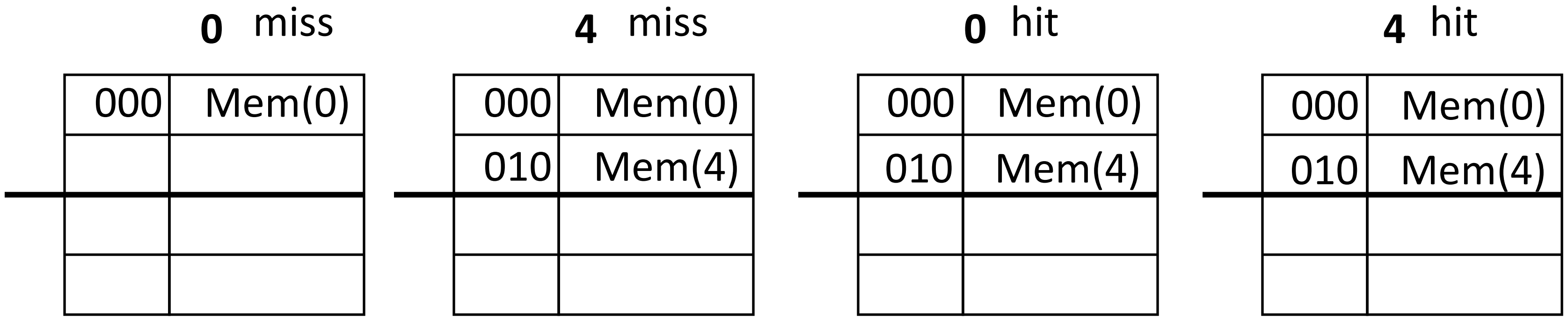
- Ping pong effect due to conflict misses - two memory locations that map into the same cache block.

Example: 4-Word 2-Way SA cache with same memory reference sequence

- Consider the sequence of memory accesses

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

0(0000) 4(0100) 0 4 0 4 0 4



0 miss

000	Mem(0)

4 miss

000	Mem(0)
010	Mem(4)

0 hit

000	Mem(0)
010	Mem(4)

4 hit

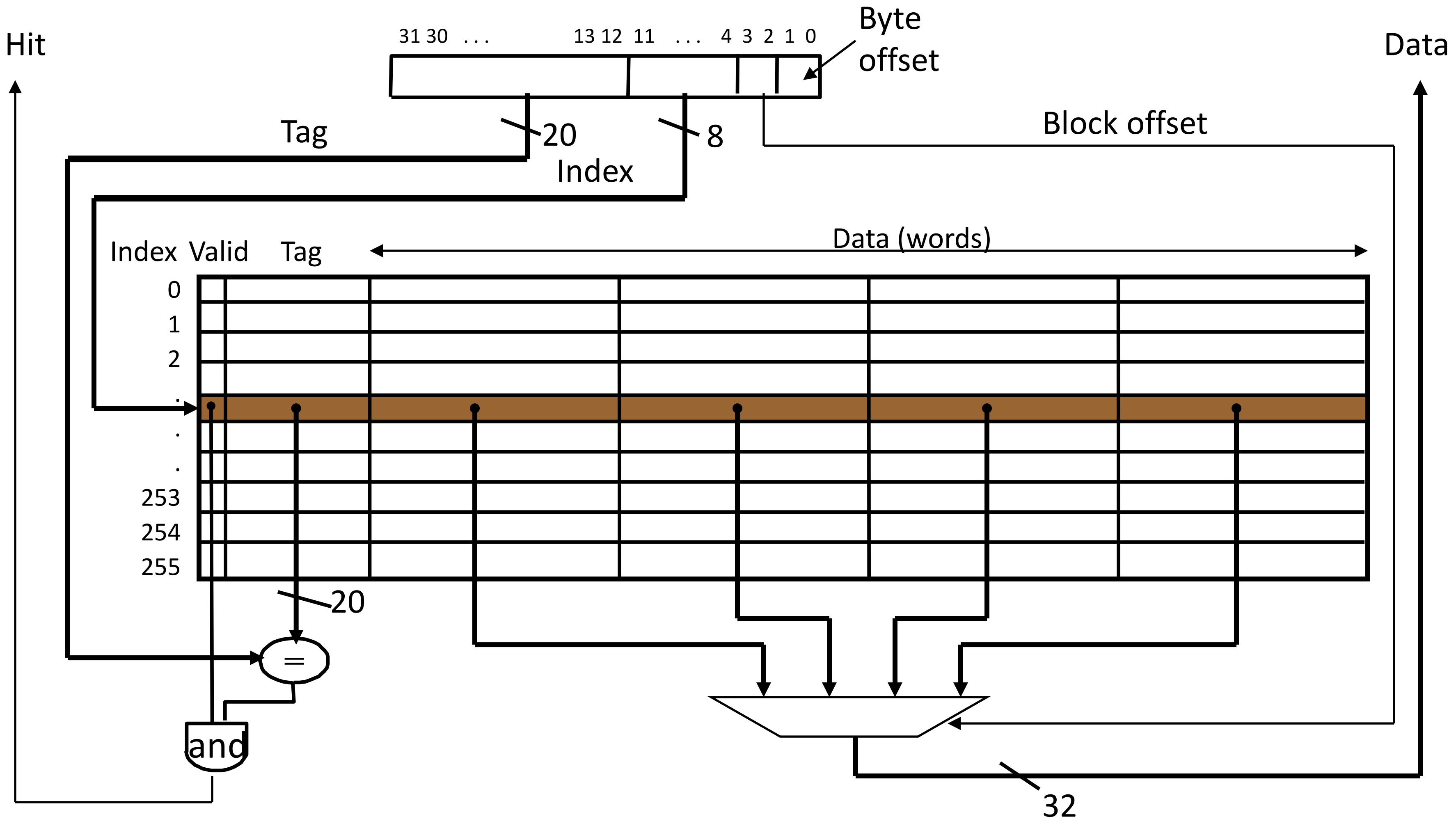
000	Mem(0)
010	Mem(4)

- 8 requests, 2 misses

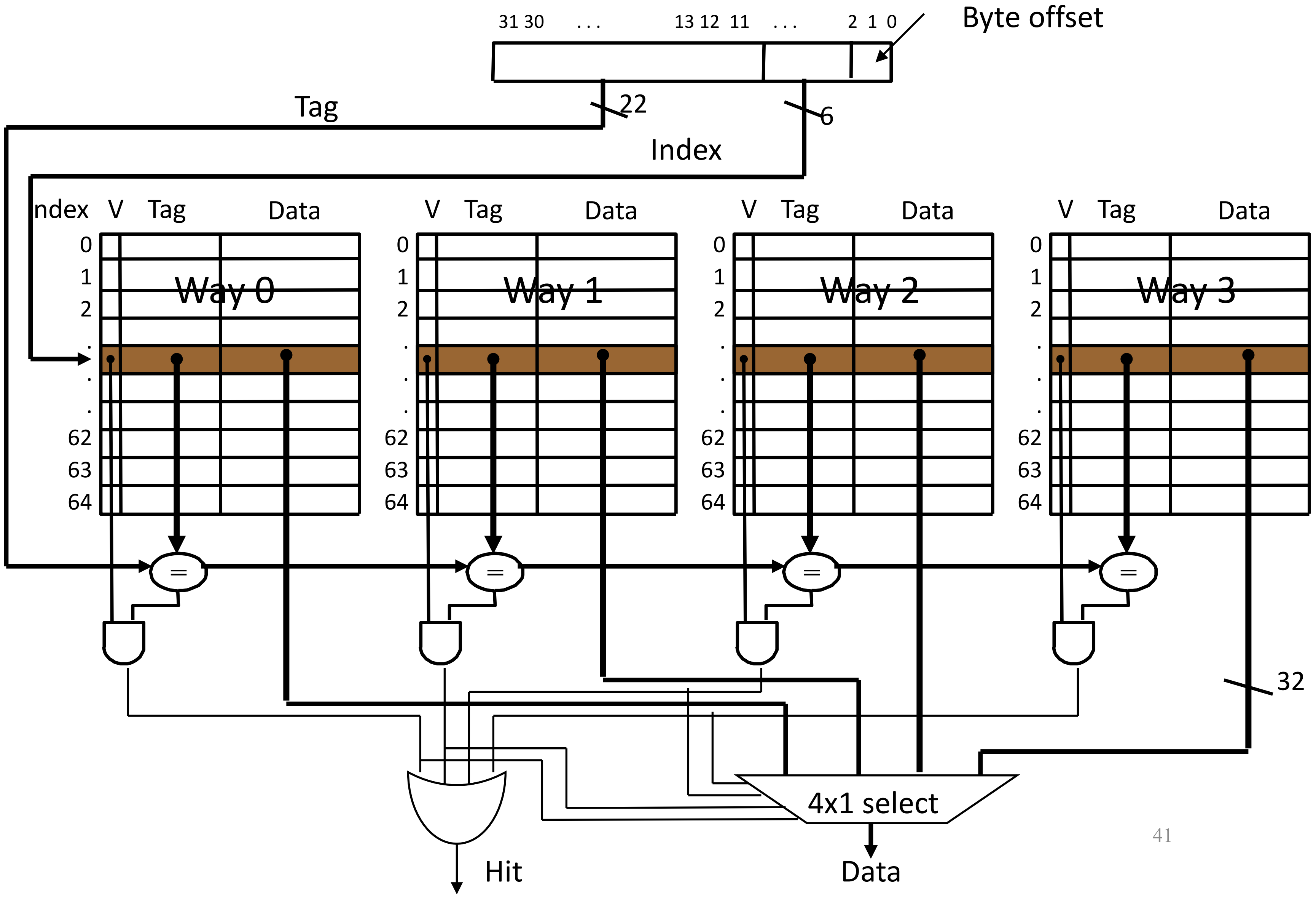
- Avoids the conflict misses due to ping pong effect in a DM cache, since now two memory locations (0 and 4) that map into the same cache set can co-exist!

Recall: DM cache: Example 3

- 32-bit address space. Tag: 20 bits; Index: 8 bits; Offset: 4 bits
- Index 8 bits → 2^8=256 cache blocks
- Tag 20 bits → 2^20=1 million memory addresses mapped to each cache index
- Offset 4 bits → 2^4=16 Bytes (4 words) /block
- Cache size = 256 blocks * 16 Bytes/block = 4K Bytes (1K words)



SA cache: Example 2

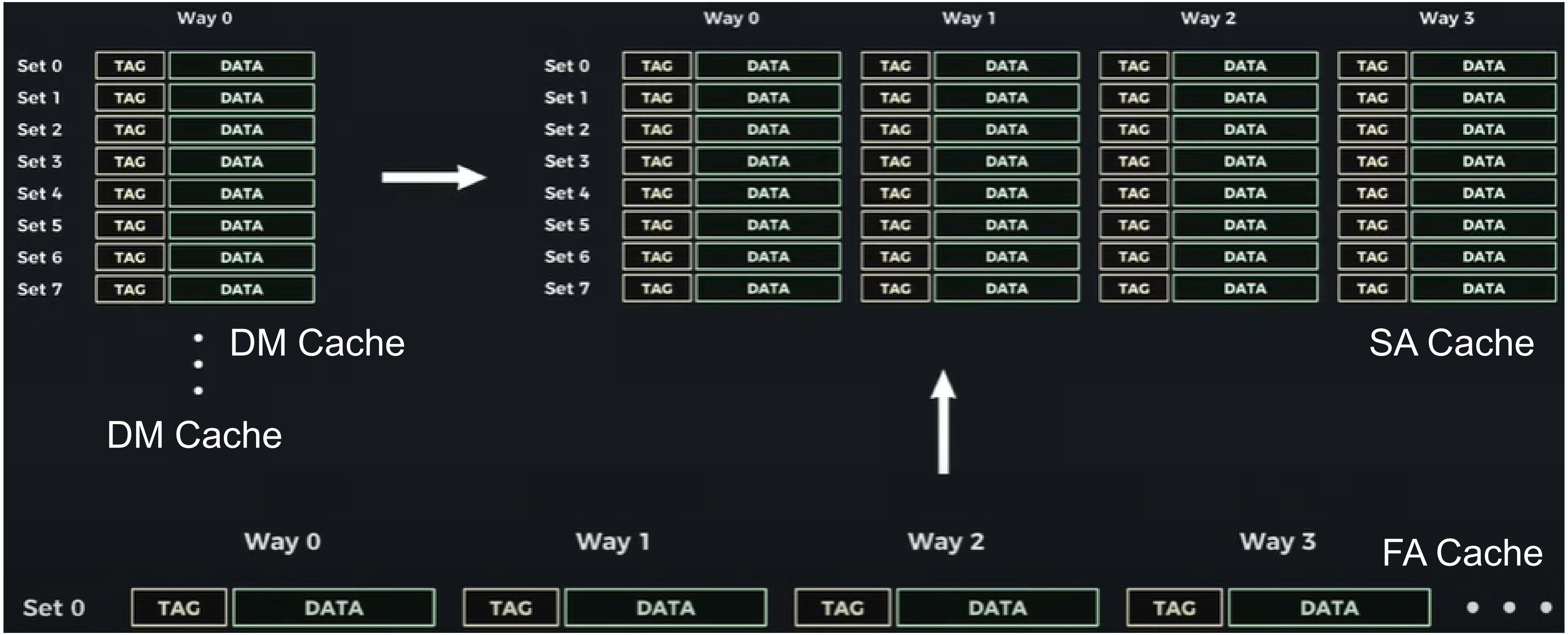


- 32-bit address space. Tag: 22 bits; Index: 6 bits; Offset: 4 bits
- Index 6 bits → 2^6=64 cache sets, 4 blocks per set
- Tag 22 bits → 2^22=4 million memory addresses mapped to each cache set
- Offset 4 bits → 2^4=16 Bytes (4 words) /block
- Cache size = 256 blocks * 16 Bytes/block = 4K Bytes (1K words)

Explanations

- This is called a 4-way set associative cache because there are 4 cache blocks for each cache index.
  - The cache index selects a set from the cache.
  - The 4 tags in the set are compared in parallel with the upper 4 bits (Tag) of the memory address.
  - If no tags match the memory address tag, we have a cache miss.
  - Otherwise, we have a cache hit and we will select the data from the way where the tag match occurs.
- SA cache
  - Pro: miss rate is lower.
  - Con: hit time is larger. A N-way SA cache needs N parallel comparators for tag comparisons followed by a multiplexer instead of just one comparator for DM cache, and it is slower than DM cache due to this extra multiplexer delay.

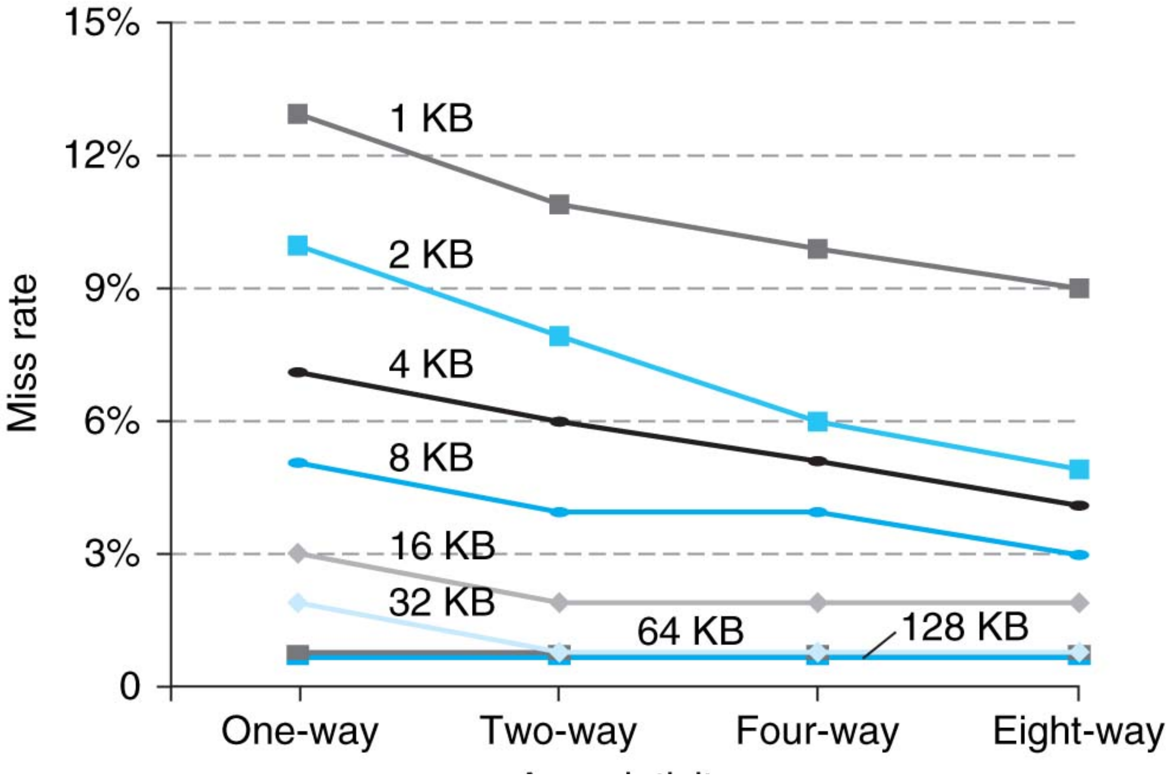
YouTube: How Cache Works Inside a CPU



How Cache Works Inside a CPU
https://www.youtube.com/watch?v=zF4VMombo7U&list=PL38NNHQLqJqYnNrTenxBvGJSPCkV9EOWk&index=1

Miss rate vs. Associativity

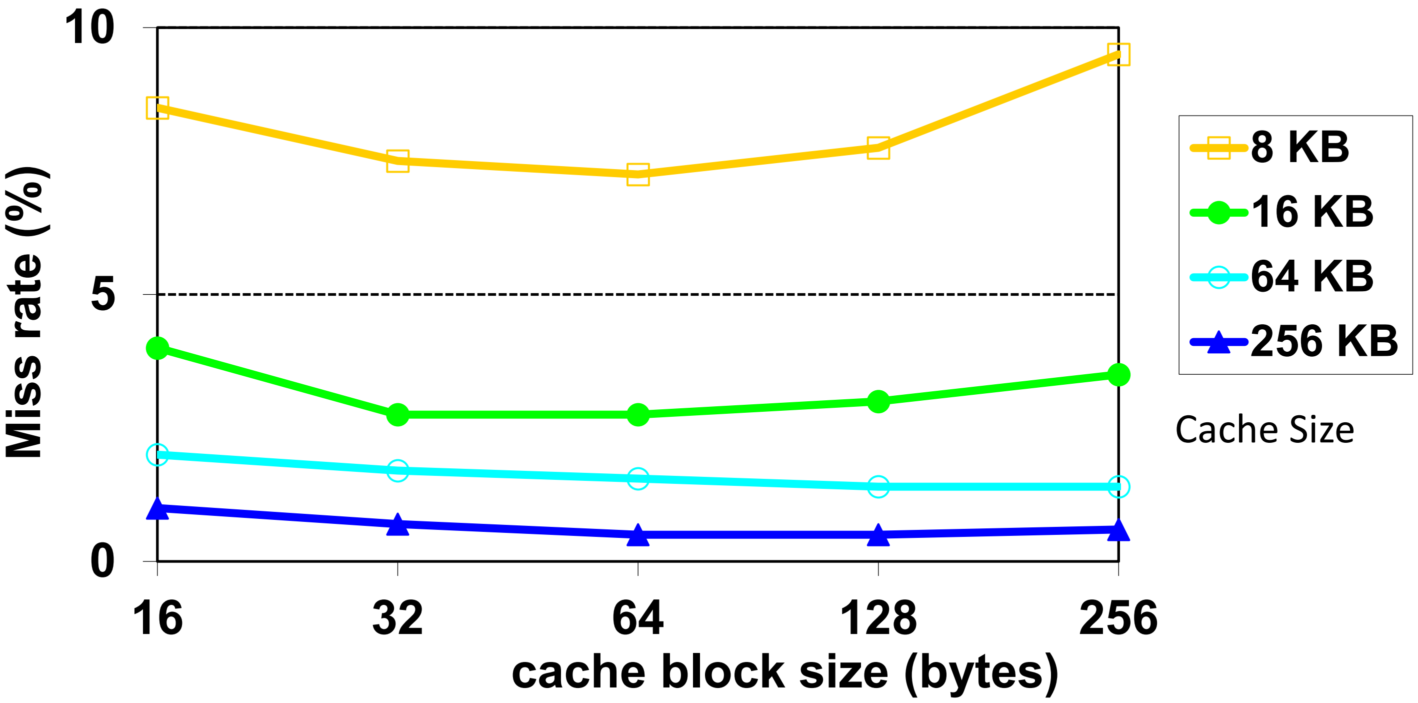
- Choice of DM or SA depends on the tradeoff between hit time and miss rate.
- As cache size increases, relative improvement from associativity increases only slightly; since the overall miss rate of a larger cache is lower, the opportunity for reducing the miss rate decreases and the absolute improvement in miss rate from associativity shrinks significantly.



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

Miss rate vs. cache block size and cache size

- Miss rate always decreases with increasing cache size (for any associativity)
- As cache block size increases from a single byte, the miss rate will decrease at first.
  - It is likely that bytes near a needed byte will be accessed at about the same time.
  - But as cache block size increases the number of lines decreases.
- Miss rate goes up if cache 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)
  - Useful cache blocks may be kicked out prematurely.
- Performance depends on the application workload, associativity, and replacement algorithm.



Cache Size

Caching Policy: Write-Back + Write-Allocate

- Write-Back: Data is written to the cache first and marked as "dirty“. Main memory is updated only when a dirty cache block is evicted. This reduces the number of writes to slower main memory, improving performance.
- Write-Allocate: On a write miss (when the data is not in the cache), a cache block is allocated, and the data is written to the cache. This makes future writes or reads to the same location faster.
- Example: CPU writes to memory address A, which is not in the cache:
  - A write miss occurs. The write-allocate policy loads the memory block containing A into the cache. The CPU writes data to the cache block and marks it as "dirty“. Later, when the dirty cache block containing A is evicted, the updated data is written back to main memory.
- Benefit: This approach minimizes memory writes and improves performance for workloads with frequent writes to the same location, which are mostly absorbed by the cache and do not go to main memory.

Caching Policy: Write-Through + No-Write-Allocate

- Write-Through: Data is written simultaneously to both the cache (if present) and main memory. This ensures that main memory always has the most up-to-date data.
- No-Write-Allocate: On a write miss, no cache block is allocated. Instead, data is written directly to main memory without involving the cache.
- Example: CPU writes to memory address A, which is not in the cache:
  - A write miss occurs. The no-write-allocate policy skips loading A into the cache. The CPU writes directly to main memory, bypassing the cache.
- Benefit: This approach avoids polluting the cache with data that may not be reused, which can be advantageous for workloads with infrequent writes or streaming data.

Caching Policy Tradeoffs

- Performance vs Consistency:
  - Write-back improves performance but risks losing data if the system crashes before eviction.
  - Write-through ensures consistency but incurs higher latency due to immediate writes.
- Cache Utilization:
  - Write-allocate optimizes for future accesses but may evict useful blocks.
  - No-write-allocate avoids unnecessary evictions but doesn’t benefit from caching for future writes.

Policy	Cache Behavior on Write Miss	Memory Update Timing	Best For
Write-Back + Write-Allocate	Allocates block; writes to cache	On eviction	Write-intensive workloads
Write-Through + No-Write-Allocate	Writes directly to memory; no allocation	Immediate	Read-heavy or streaming workloads

CPU Cache Write Policies Explained
https://www.youtube.com/watch?v=wfVy85Dqiyc&list=PL38NNHQLqJqYnNrTenxBvGJSPCkV9EOWk&index=3

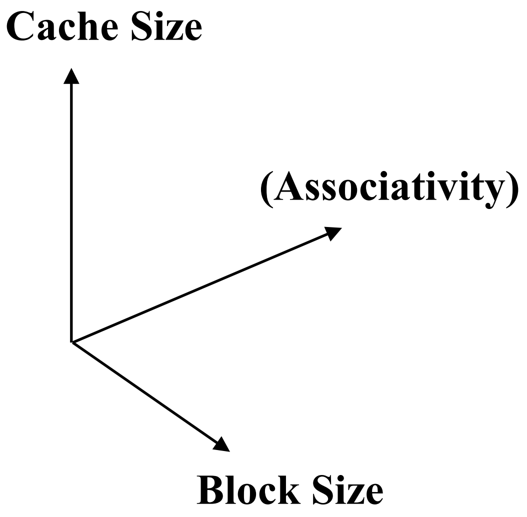
Cache Hierarchy: How Modern CPU Caches Are Organized
https://www.youtube.com/watch?v=7yrK_9PderQ&list=PL38NNHQLqJqYnNrTenxBvGJSPCkV9EOWk&index=4

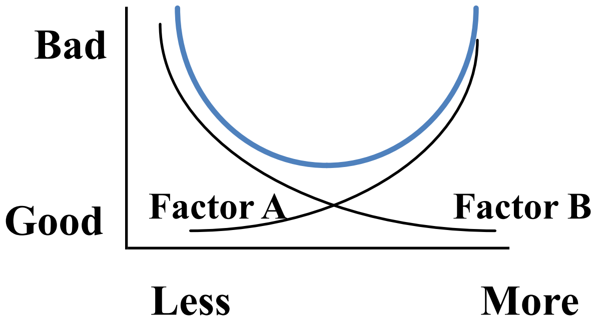
Cache Block Replacement Policy

- If a cache block in a SA or FA cache needs to be replaced due to a cache miss, typical replacement policy is Least Recently Used
  - Hardware keeps track of access history
  - Replace the block that has not been used for the longest time, since it is less likely to be reused due to temporal locality
- For DM cache, each memory block is mapped to a unique cache slot, hence no need for replacement policy

The Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Write-through vs. write-back
  - Write-allocate vs write-no-allocate
  - Replacement policy (for SA and FA caches)
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost



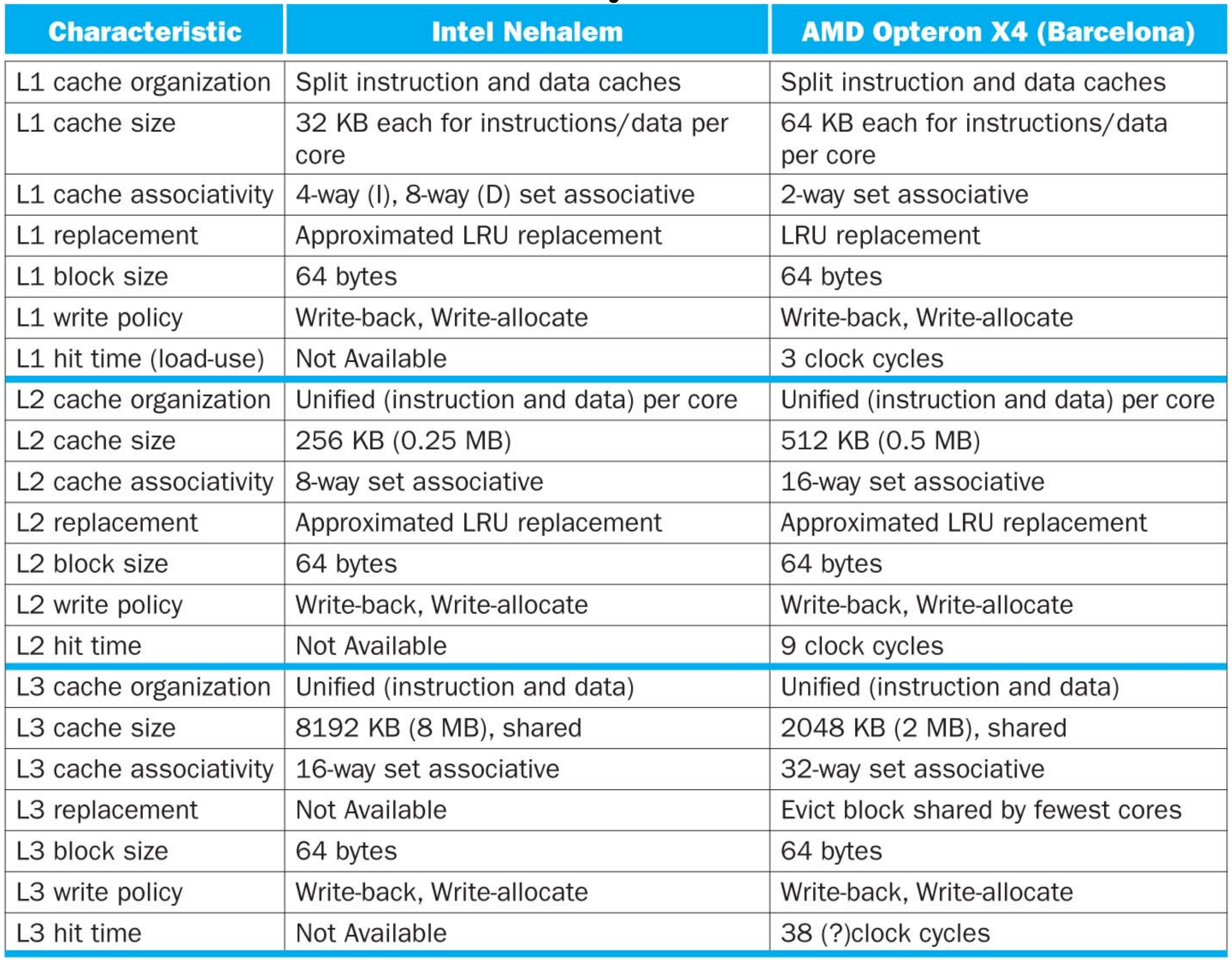


Improving Cache Performance: Summary

1. Reduce hit time
   - Smaller cache
   - 1 word blocks (no multiplexor/selector to pick word)
2. Reduce miss rate
   - Bigger cache size
   - Larger blocks (16 to 64 bytes typical)
   - More flexible placement by increasing associativity
3. Reduce miss penalty
   - Smaller blocks
   - Use multiple cache levels
   - Write-back instead of write-through

Intel Nehalem and AMD Opteron

- L1→L2→L3
  - Size increasing
  - Associativity increasing
  - Both cause hit time to increase, and miss rate to decrease



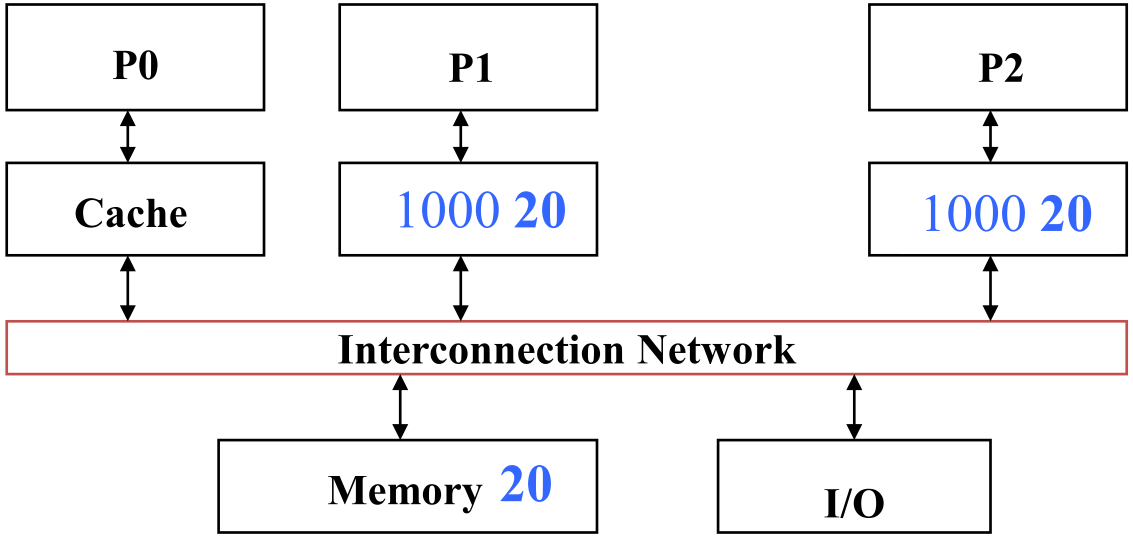
Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

Keeping Multiple Caches Coherent

- HW architect’s job is to keep cache values coherent across multiple cores with shared memory.
- One approach: When any processor has cache miss or writes, notify other processors via bus or interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies, and write through to memory

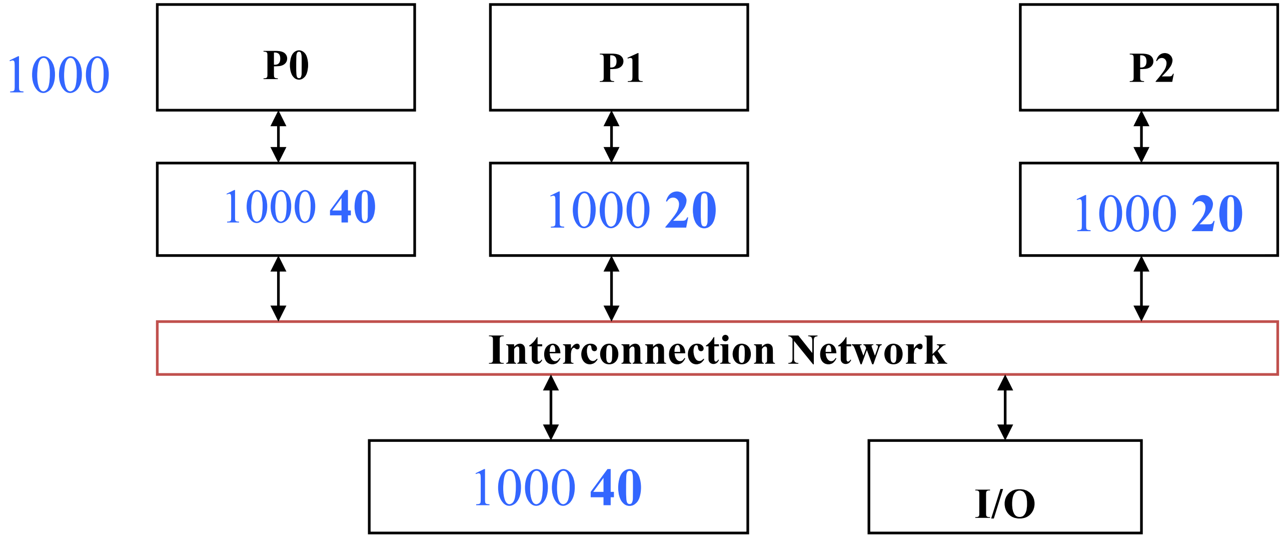
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Processor 0 *write-invalidates* other copies

False Sharing

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose X is at address 4000, Y in 4012
  - Two variables are in the same cache block, even though they have different addresses
- Effect called *false sharing*

Summary #1/2: Cache

- 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:
    - e.g., cold start misses.
  - Capacity misses:
    - Can be reduced by increasing cache size
  - Conflict misses
    - Can be reduced by increasing cache size and/or associativity

Summary #2/2: Cache

- Cache organizations:
  - Direct Mapped (DM): single block per set
  - Set Associative (SA): more than one block per set
  - Fully Associative (FA): all blocks in one set
- Set associativity - reduce cache miss rate
  - Memory block maps into more than 1 cache block
  - N-way: N possible locations in a cache set to hold a memory block
- Lots of cache parameters!
  - Block size, cache size, associativity, write-back vs. write-through, write-allocate, etc.
- Multi-level caches - reduce cache miss penalty
  - Optimize first level to be fast
  - Optimize 2nd and 3rd levels to minimize the memory access penalty

Quiz

- Q: Consider 32-bit address space; a DM cache with size 32 KB; each cache block is 8 words. What is the TIO breakdown?
- ANS: T=17, I=10, O=5
  - 8-word per block => 32 bytes / block => O = 5
  - 32 KB / (32 bytes / block) = 2^10 blocks total => I = 10
  - T = 32 – 10 – 5 = 17
- Q: Consider 32-bit address space; a 4-way SA cache with size 32 KB; each cache block is 8 words. What is the TIO breakdown?
- ANS: T=19, I=8, O=5 (Tag 19b, Index 8b, Offset 5b)
  - 8-word per block => 32 bytes / block => O = 5
  - 32 KB / (32 bytes / block) = 2^10 blocks total
  - 2^10 blocks / (4 blocks / set) = 2^8 sets total => I = 8
  - T = 32 – 8 – 5 = 19
- Q: Consider 32-bit address space; an FA cache with size 32 KB; each cache block is 8 words. What is the TIO breakdown?
  - ANS: T=27, I=0, O=5
  - 8-word per block => 32 bytes / block => O = 5
  - FA cache => I = 0
  - T = 32 – 0 – 5 = 27

Quiz con’t

- Q: Consider 32-bit address space; a DM cache with size 16 KB; each cache block is 4 words. What is the TIO breakdown?
- ANS: T=18, I=10, O=4
  - 4-word per block => 16 bytes / block => O = 4
  - 16 KB / (16 bytes / block) = 2^10 blocks total => I = 10
  - T = 32 – 10 – 4 = 18
- Q: Consider 32-bit address space; a 2-way SA cache with size 16KB; each cache block is 4 words. What is the TIO breakdown?
- ANS: T=19, I=9, O=4
  - 4-word per block => 16 bytes / block => O = 4
  - 16 KB / (16 bytes / block) = 2^10 blocks total
  - 2^10 blocks / (2 blocks / set) = 2^9 sets total => I = 9
  - T = 32 – 9 – 4 = 19
- Q: Consider 32-bit address space; an FA cache with size 16KB; each cache block is 4 words. What is the TIO breakdown?
  - ANS: T=27, I=0, O=5
  - 4-word per block => 16 bytes / block => O = 4
  - FA cache => I = 0
  - T = 32 – 0 – 4 = 28

Quiz con’t

- Q: How many 32-bit integers can be stored in a DM cache with 15 tag bits, 15 index bits, and 2 offset bits?
- A: Each cache block is 2^2=4 Bytes and can store one 32-bit integer. The cache has a total number of 2^15=32K blocks, hence it can store 32K integers. (The tag bits are irrelevant here since it is related to memory size, not cache size)

Quiz con’t

- For a given cache size, a larger block size can cause a lower hit rate than a smaller one.
  - True. A large block size leads to fewer cache blocks.
- If you know your computer’s cache size, you can often make your code run faster.
  - True. By tuning your code to be cache-aware.
- Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.
  - False. This is called temporal locality.