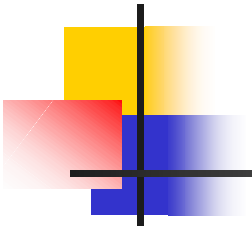


Computer Architecture
Lecture 13

Memory Hierarchy

Reference material:

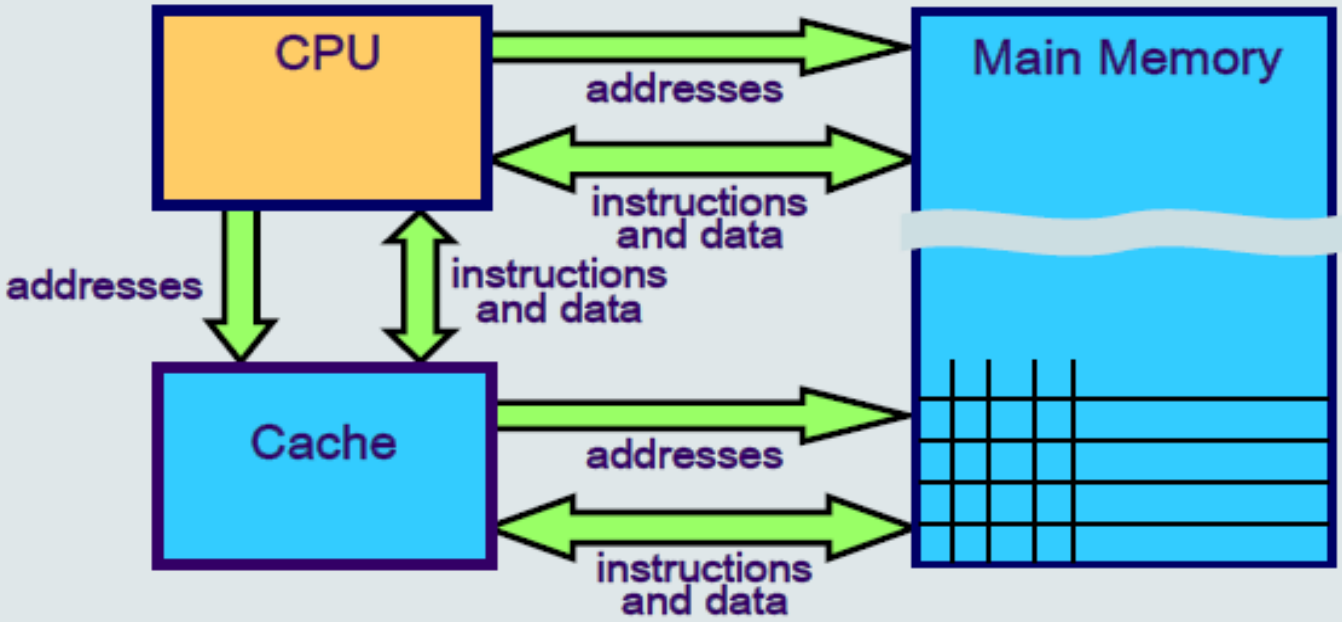
Computer Organization and Design: The Hardware and Software Interface by Patterson and Hennessy, 4th Edition, Morgan and Kauffman Publications, 2008 (chap-07)

Computer System Architecture by Morris Mano, 3rd ed. (chap-12)

Recommended YouTube links

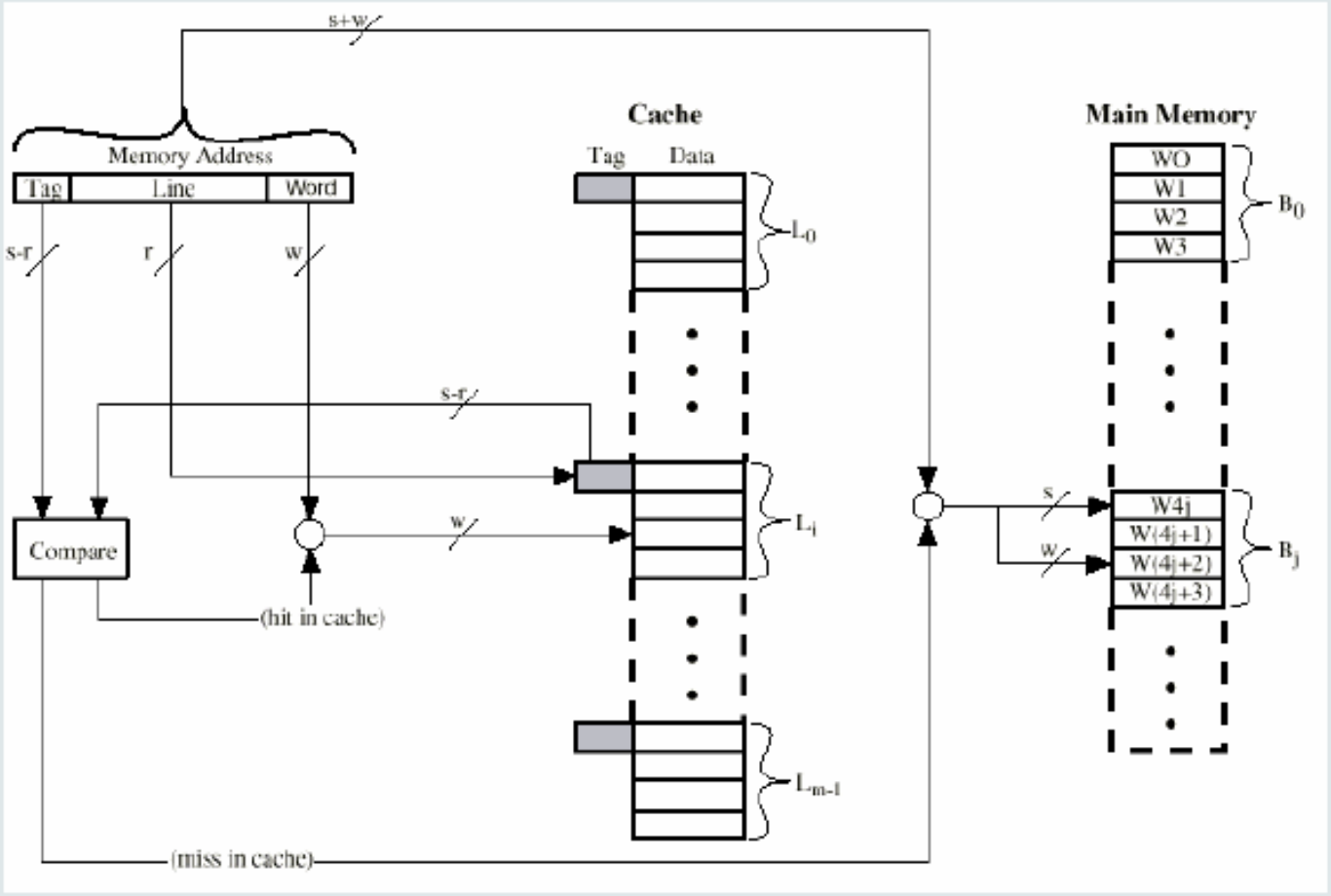
- https://www.youtube.com/watch?v=LzsiFYVMqi8
- https://www.youtube.com/watch?v=Ref231Eg9QE
- https://www.youtube.com/watch?v=7IVE6vXqEoc

Cache Memory



- A cache is a very fast memory which is put between the main memory and the CPU, and used to hold segments of program and data of the main memory.

Direct Mapping Cache (Cont'd)

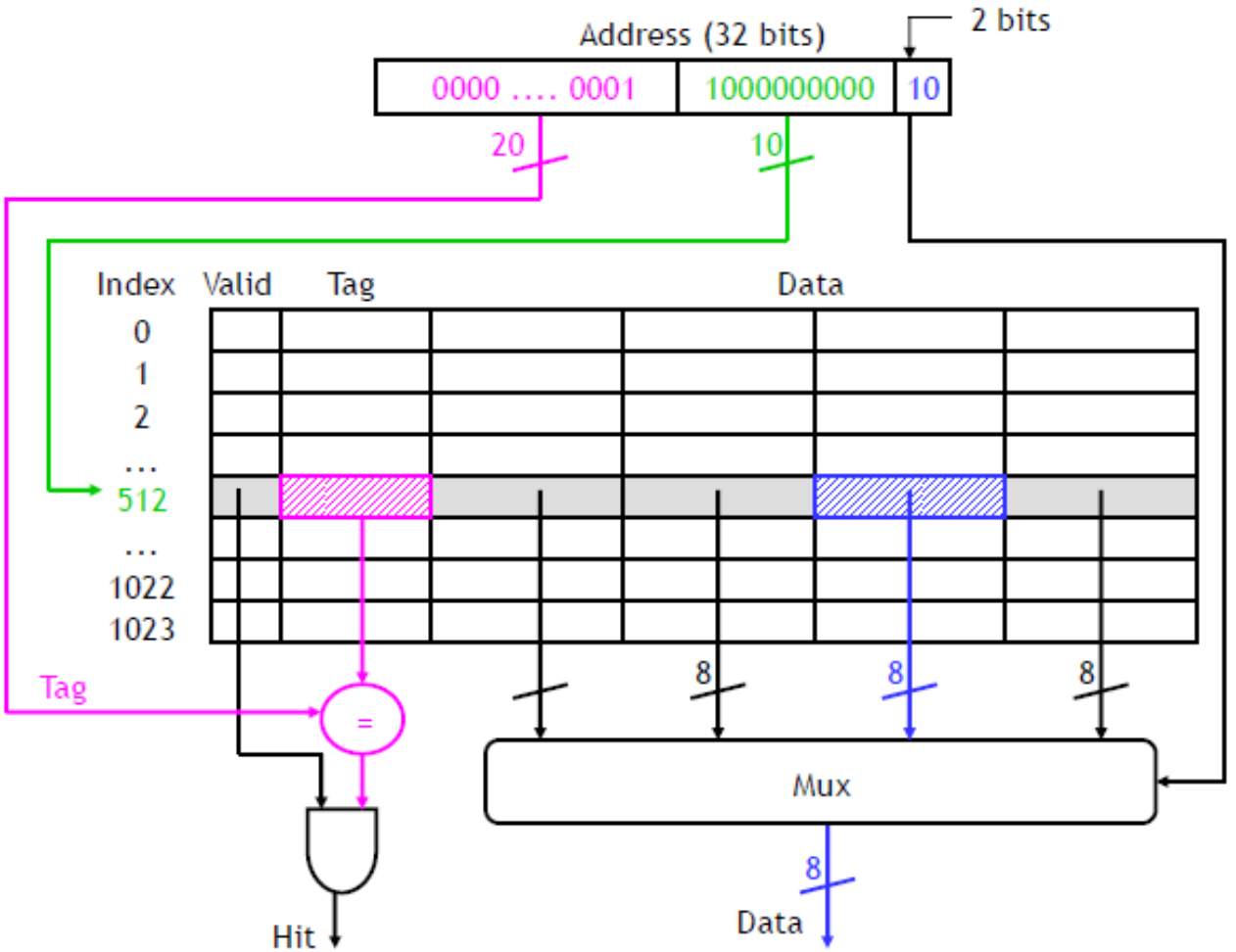


A larger example cache mapping

- Where would the byte from memory address 6146 be stored in this direct-mapped 2^{10}-block cache with 2^2-byte blocks?

- 6146 in binary is 00…01 1000 0000 00 10.

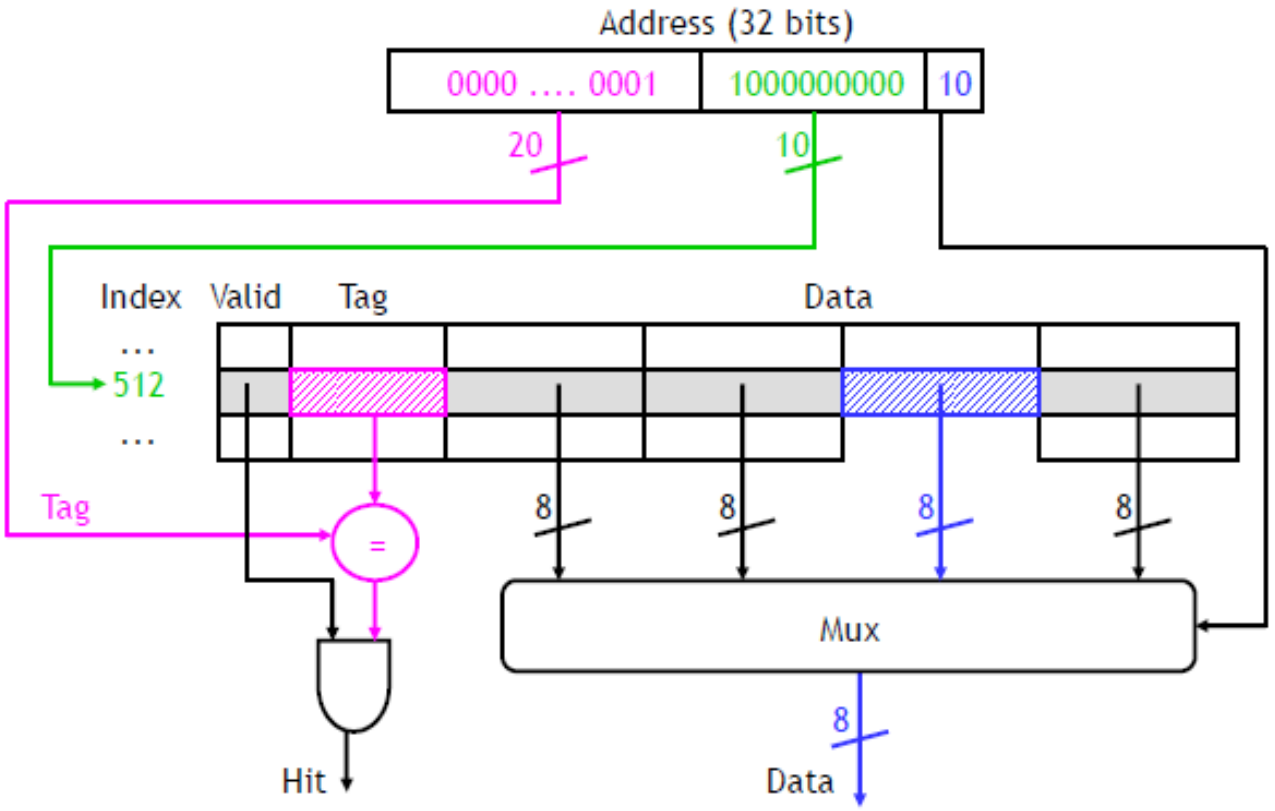
A larger diagram of a larger example cache mapping



5

What goes in the rest of that cache block?

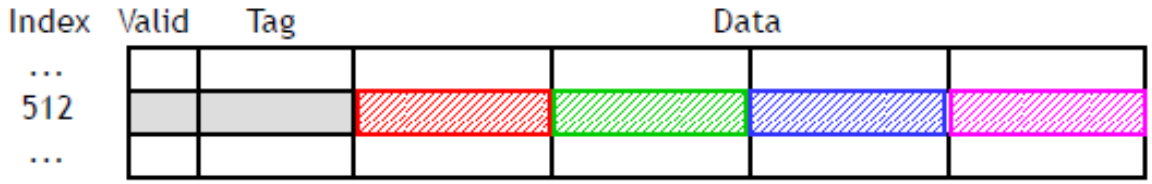
- The other three bytes of that cache block come from the same memory block, whose addresses must all have the same index (1000000000) and the same tag (00…01).



The rest of that cache block

- Again, byte *i* of a memory block is stored into byte *i* of the corresponding cache block.
  - In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively.
  - You can also look at the lowest 2 bits of the memory address to find the block offsets.

Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147

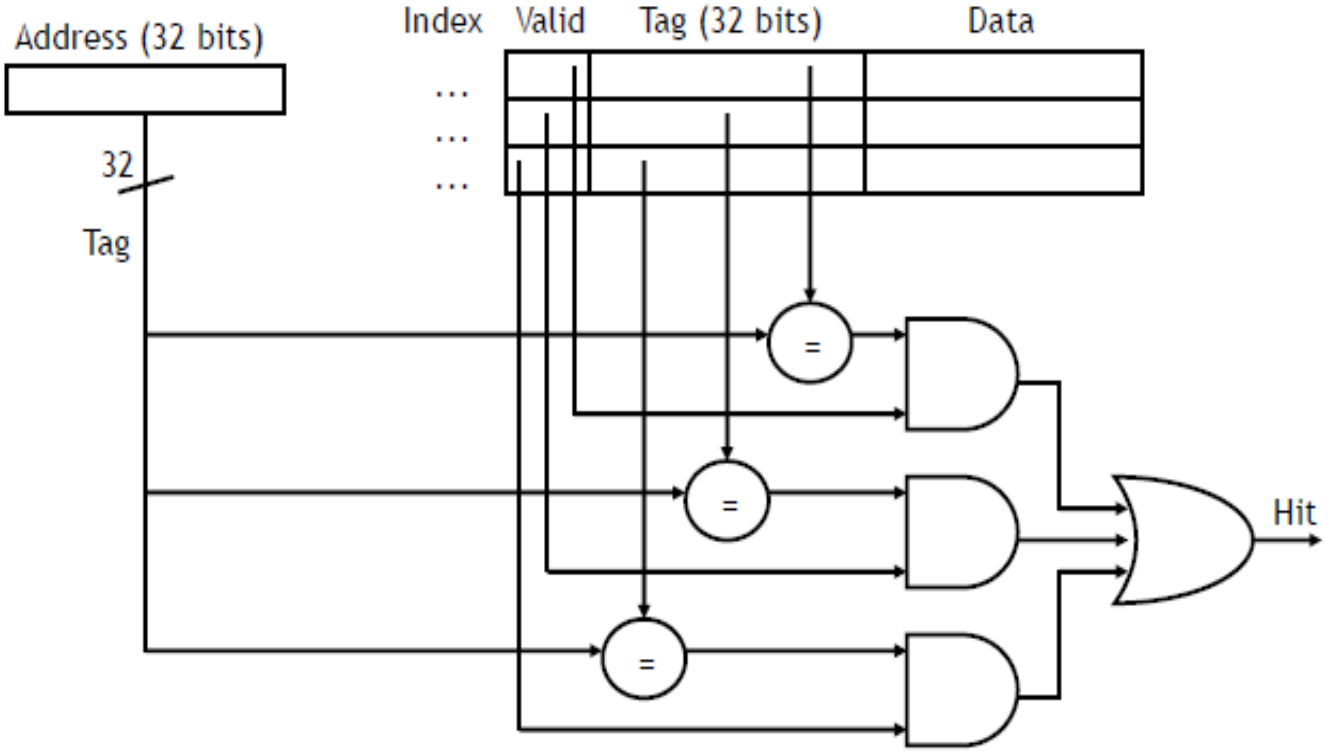


A fully associative cache

- A fully associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block.
  - When data is fetched from memory, it can be placed in *any* unused block of the cache.
  - This way we'll never have a conflict between two or more memory addresses which map to a single cache block.

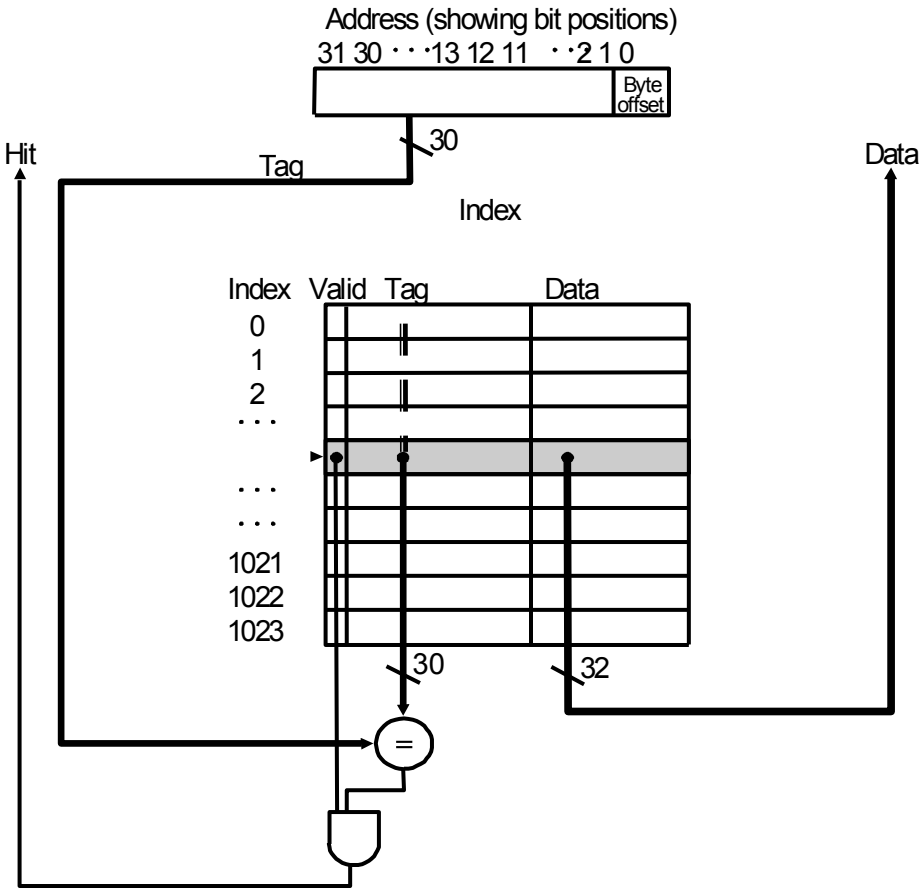
The price of full associativity

- However, a fully associative cache is expensive to implement.
  - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
  - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!

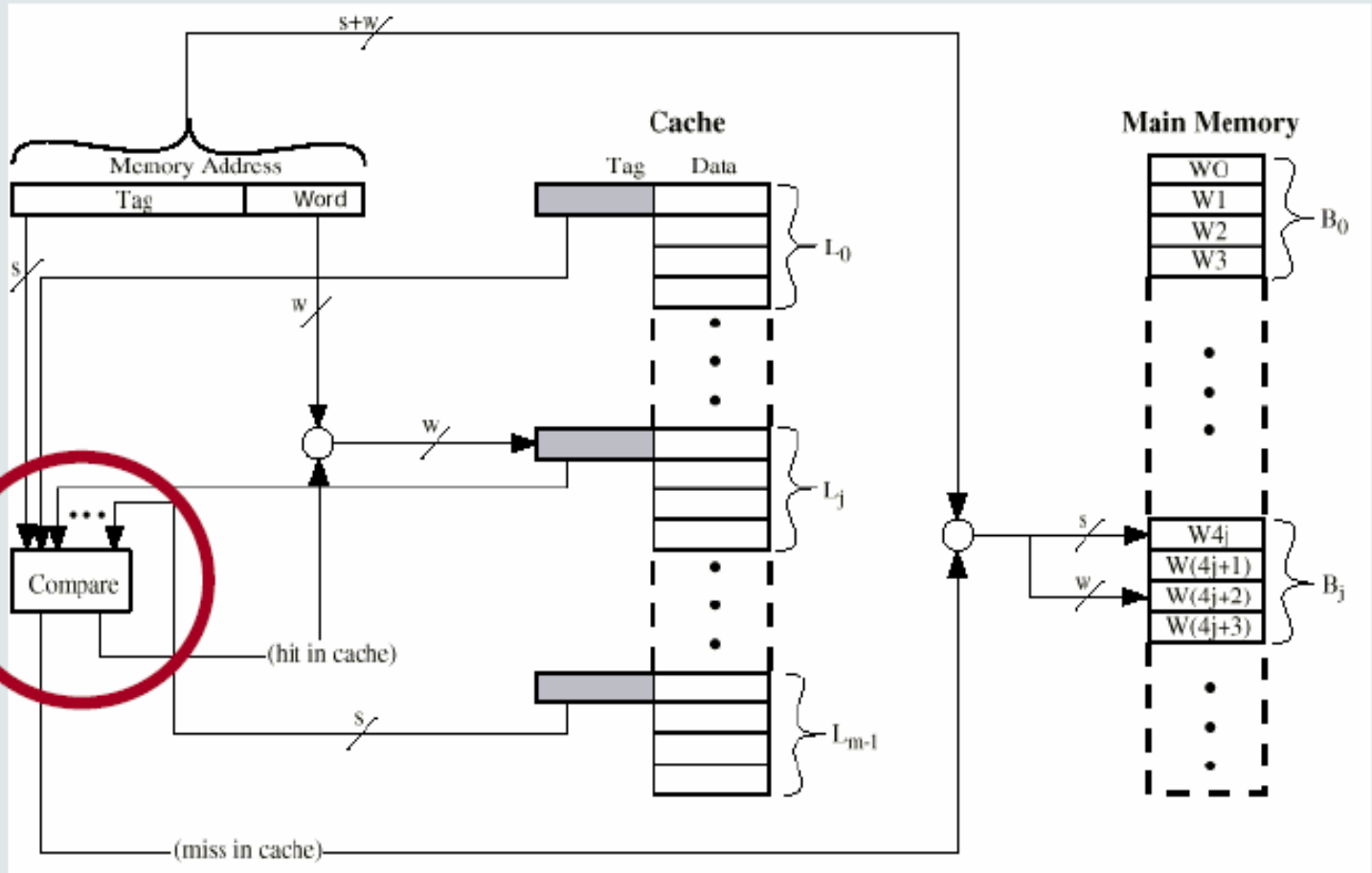


Fully Associative

- Associative memory stores address (Tag) and content (data) of the main memory word.
- CPU address is compared with all the Tags in parallel, **needs as much comparators as much Cache lines**.
- When a Tag matches with CPU address, valid bit is checked for valid data.
- If **Tag** bits are equal & **valid bit** is also set, data from the corresponding location is sent to the CPU.
- Requires more hardware for comparison (30 bit comparator)
- Any location from main memory can be placed anywhere in cache.
- Increases hit rate.

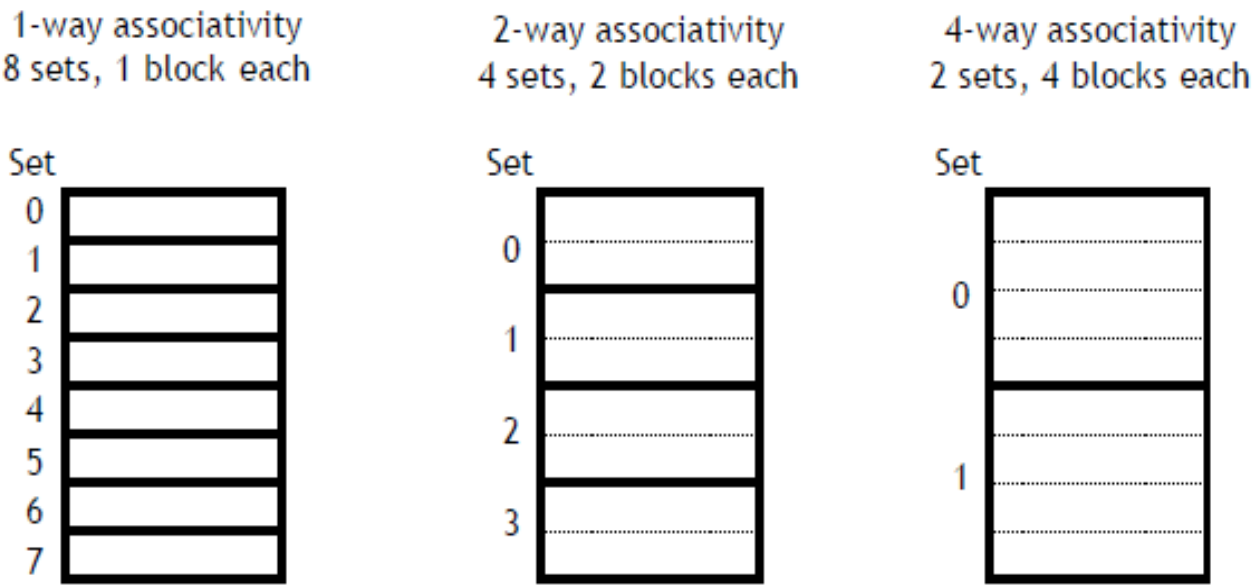


Fully Associative Organization



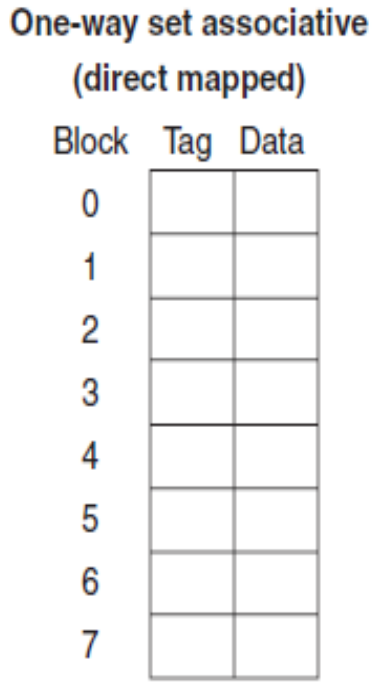
Set associativity

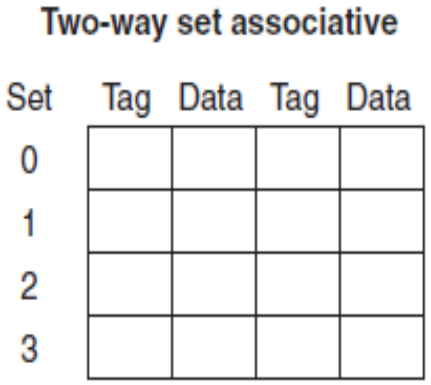
- An intermediate possibility is a set-associative cache.
  - The cache is divided into *groups* of blocks, called sets.
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.
- If each set has 2^x blocks, the cache is an 2^x-way associative cache.
- Here are several possible organizations of an eight-block cache.



One/Two Way Set Associative

- Two data items having same index with different Tags can be placed in Cache.
- Less bits are required for index whereas more bits are required for Tag field.
- Improves hit rate.





Four Way Set Associative

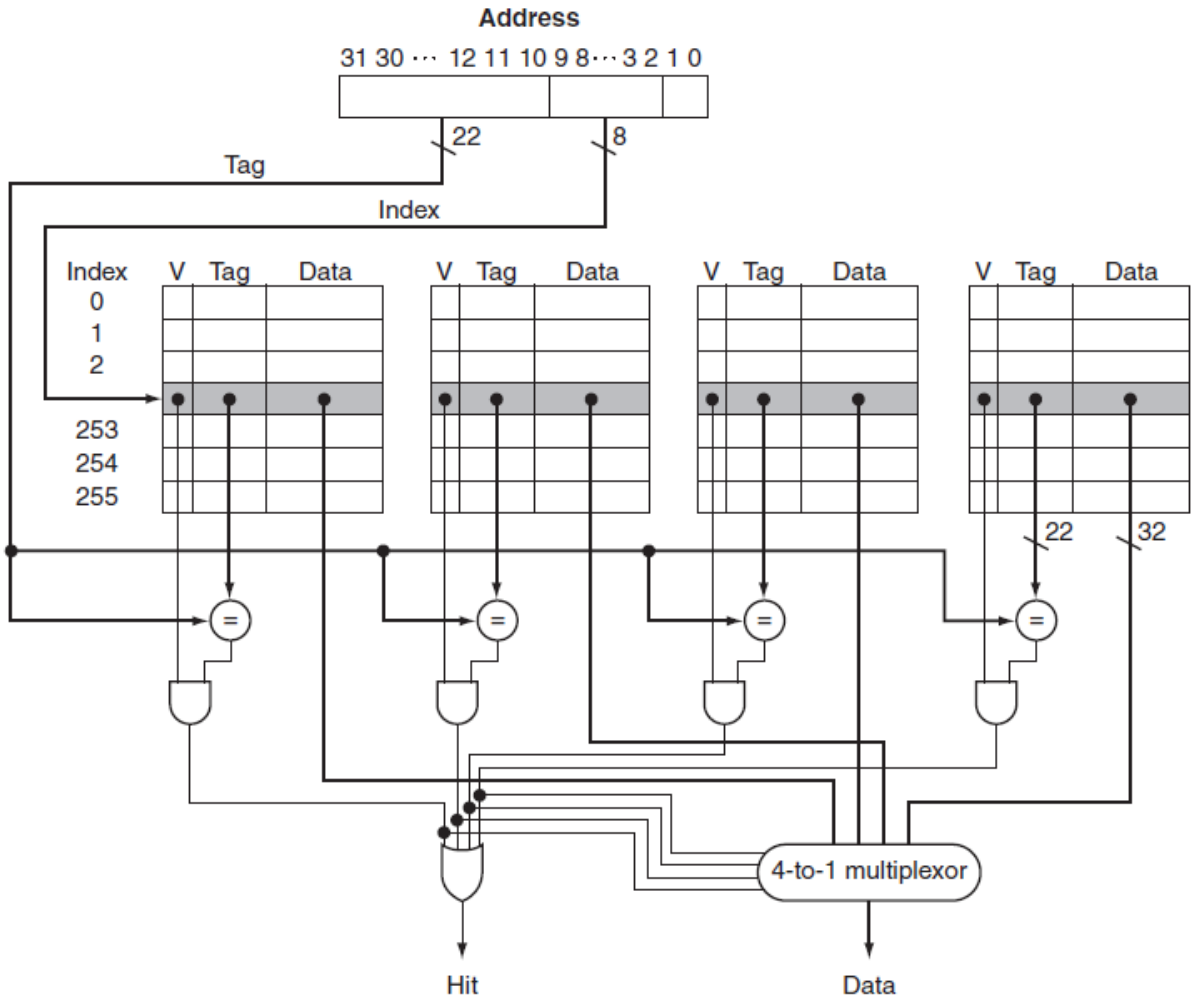
- Four data items having same index with different Tags can be placed in Cache (Four way set Associative)
- Eight data items having same index with different Tags can be placed in Eight way set Associative Cache.

Four-way set associative

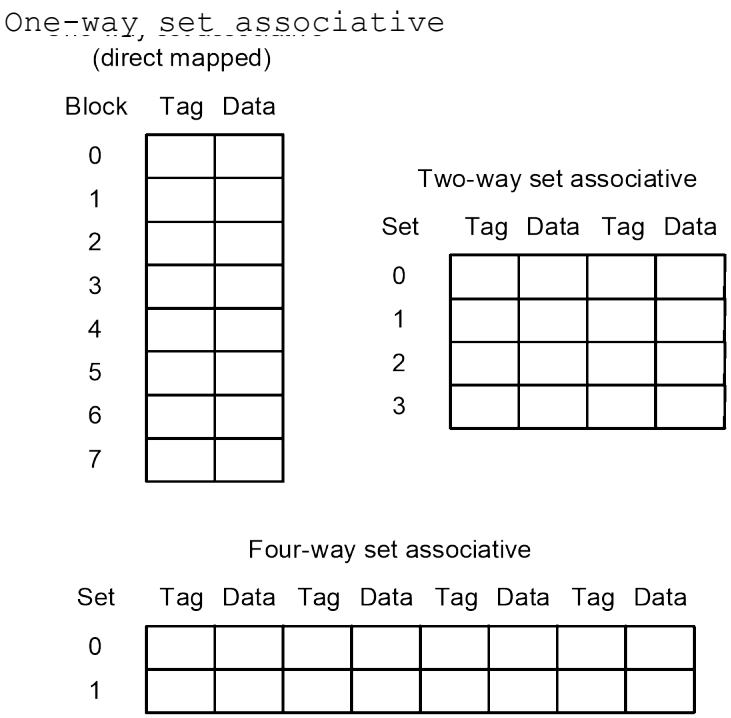
Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Four Way Set Associative

- Dividing 1024 line m into 4 blocks, 1024/ locations per block.
- 8 bits index field
- 22 bits Tag field.
- Improves Hit rate.
- Needs extra hardwar each Tag comparisor
- Costly solution



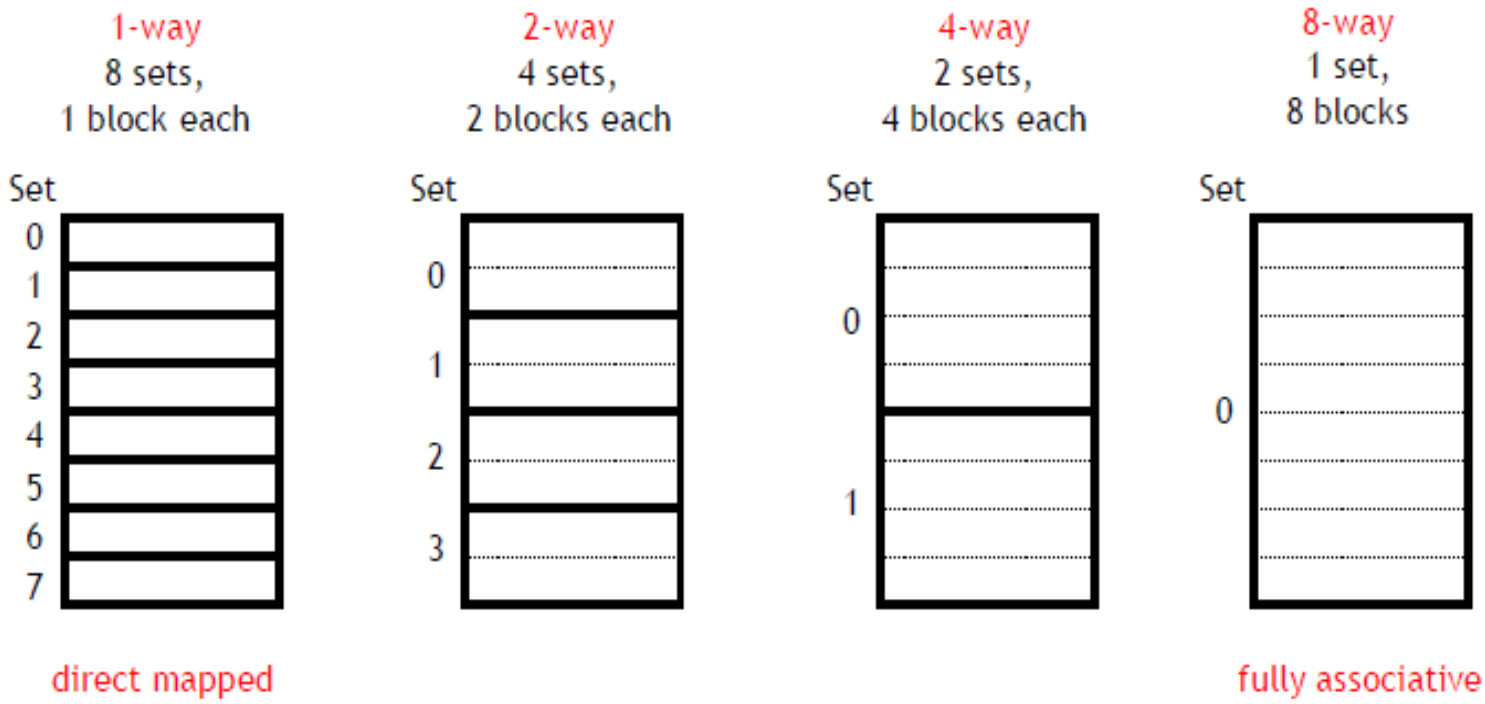
Decreasing Miss Rates with Associative Block Placment



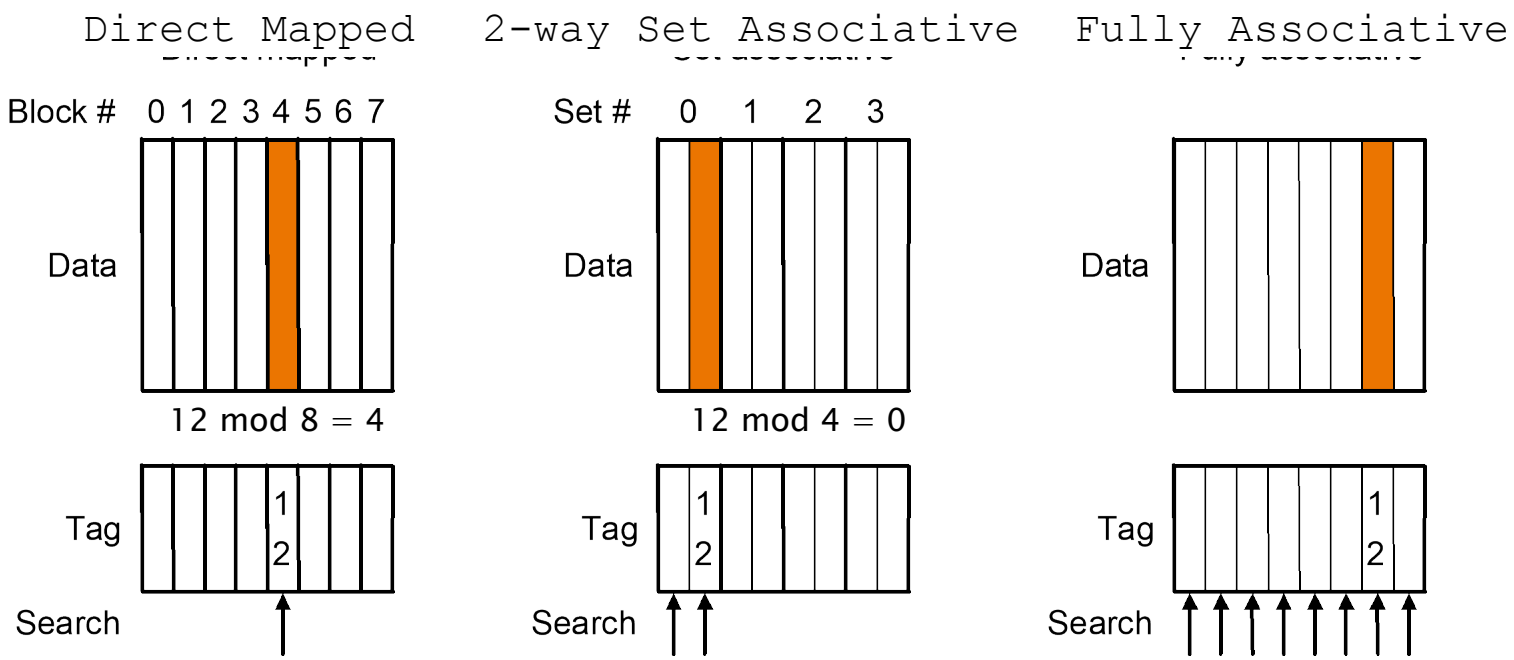
Configurations of an 8-block cache with different degrees of associativity

Set associative caches are a general idea

- By now you may have noticed the 1-way set associative cache is the same as a direct-mapped cache.
- Similarly, if a cache has 2^k blocks, a 2^k-way set associative cache would be the same as a fully-associative cache.



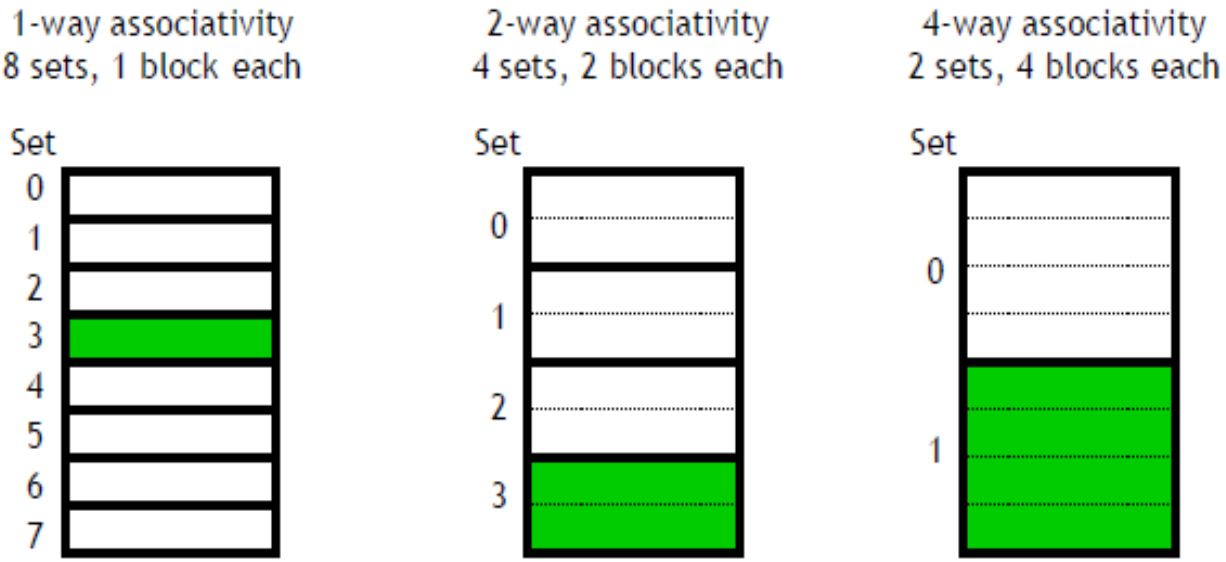
Decreasing Miss Rates with Associative Block Placment



Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity

Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00…0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in *any* block, shown in green, within the correct set.



Summary

- Direct mapping
  - A word from main memory can only reside in only one Cache location
  - Simple, smaller, cheaper mapping.
  - Decreases hit rate
- Fully Associative
  - A word from main memory can reside anywhere in Cache.
  - More complex, requires more hardware, more power dissipation and more costly.
  - Gives best hit rate.
- Set Associative
  - A word from main memory can reside in some restricted locations in Cache.
  - Complex, requires more hardware, more power, costly.
  - Improves hit rate as multiple blocks having same index with different Tags can reside in different Cache locations.

Replacement Algorithms

- With direct mapping, it is no need because each block has a pre-defined location where it can be placed and it replaces the old one.
- With associative and set-associative mapping, a replacement algorithm is needed in order to determine which block to replace:
  - First-in-first-out (FIFO).
  - Least-recently used (LRU) – replace the block that has been in the cache longest with not reference to it.
  - Lest-frequently used (LFU) – replace the block that has experienced the fewest references.
  - Random.

Write Policy

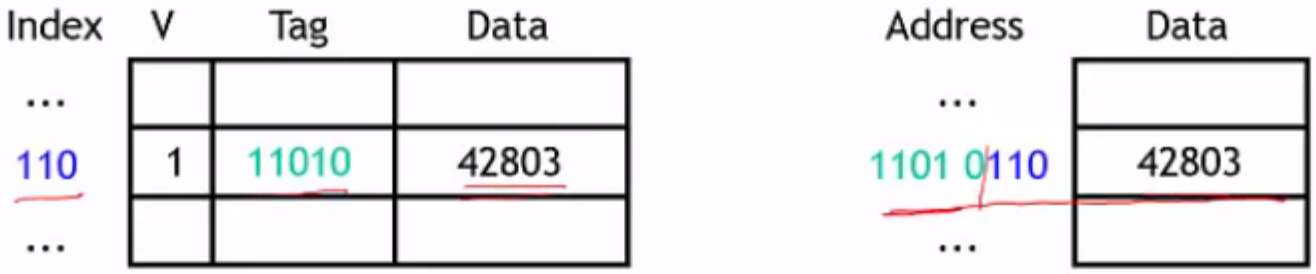
- **The problem:**
  - How to keep cache content and main memory content consistent without losing too much performance?
- **Write through:**
  - All write operations are passed to main memory: If the addressed location is currently in the cache, the cache is updated so that it is coherent with the main memory.
  - For writes, the processor always slows down to main memory speed.
  - Since the percentage of writes is small (ca. 15%), this scheme doesn't lead to large performance reduction.

Write Policy

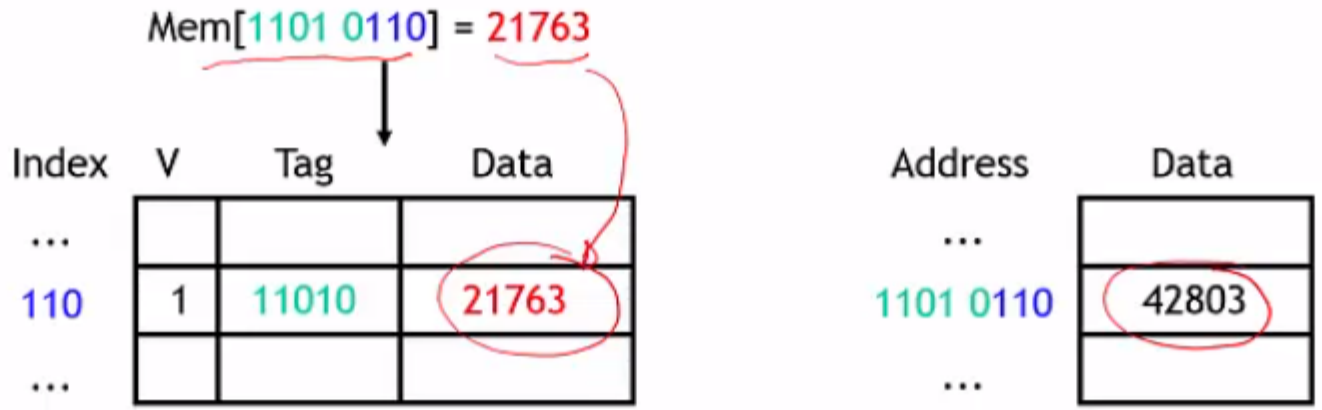
- **Write through with buffered write:**
  - The same as write-through, but instead of slowing the processor down by writing directly to main memory, the write address and data are stored in a high-speed write buffer; the write buffer transfers data to main memory while the processor continues its task.
  - Higher speed, but more complex hardware.
- **Write back:**
  - Write operations update only the cache memory which is not kept coherent with main memory. When the slot is replaced from the cache, its content has to be copied back to memory.
  - Write-back (or copy back) writes only to cache but sets a “dirty bit” in the block where write is performed.
  - When a block with dirty bit “on” is to be overwritten in the cache, it is first written to the memory.
  - Good performance (usually several writes are performed on a cache block before it is replaced), but more complex hardware is needed.

Writing to a cache

- Writing to a cache raises several additional issues
- First, let's assume that the address we want to write to is already loaded in the cache. We'll assume a simple direct-mapped cache:

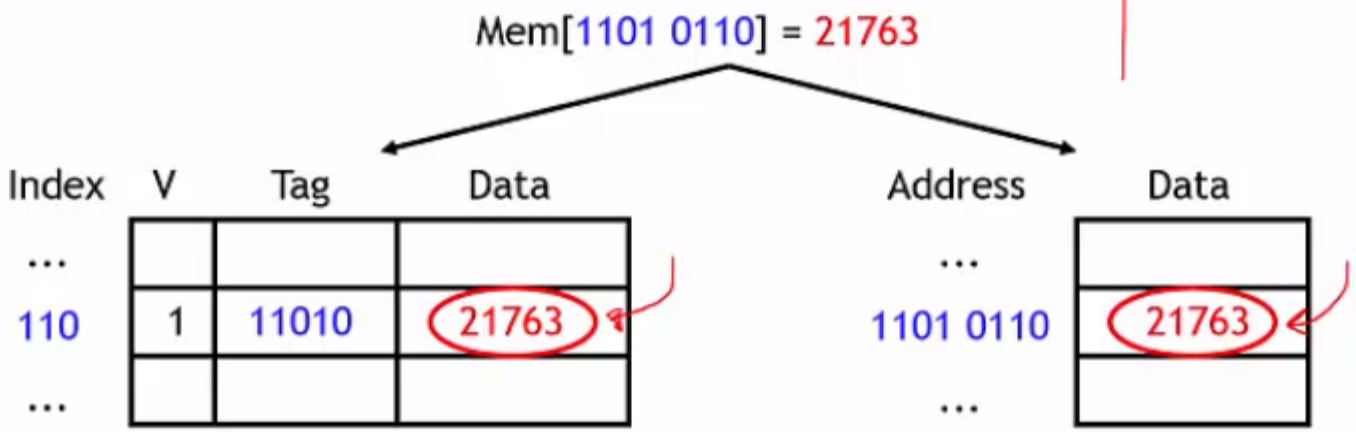


- If we write a new value to that address, we can store the new data in the cache, and avoid an expensive main memory access [but inconsistent]



Write-through caches

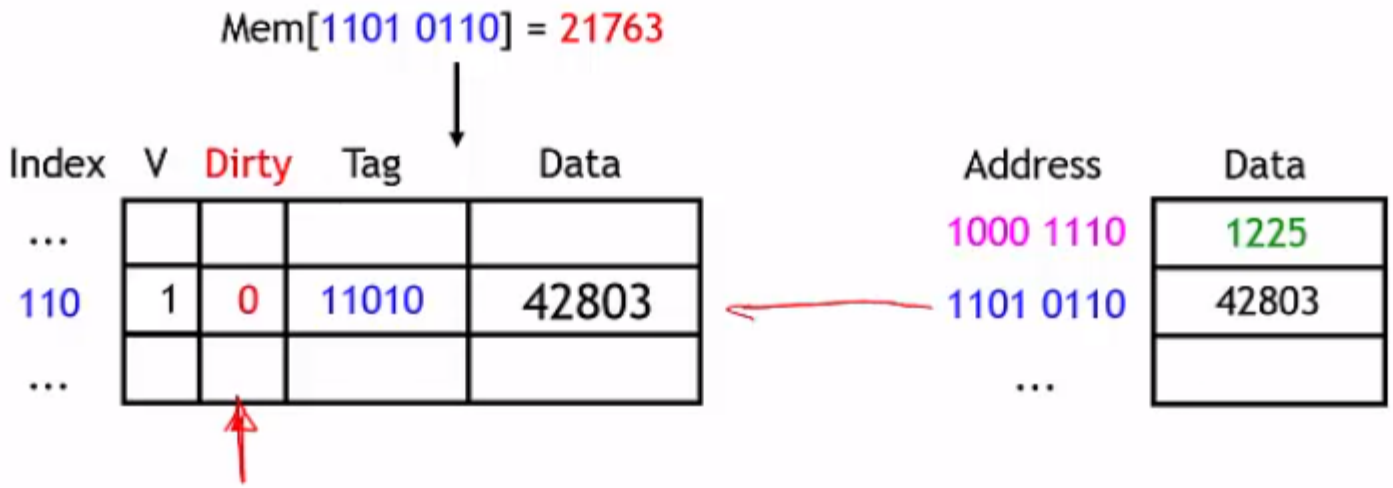
- A write-through cache solves the inconsistency problem by forcing all writes to update both the cache *and* the main memory



- This is simple to implement and keeps the cache and memory consistent
- Why is this not so good?

Write-back caches

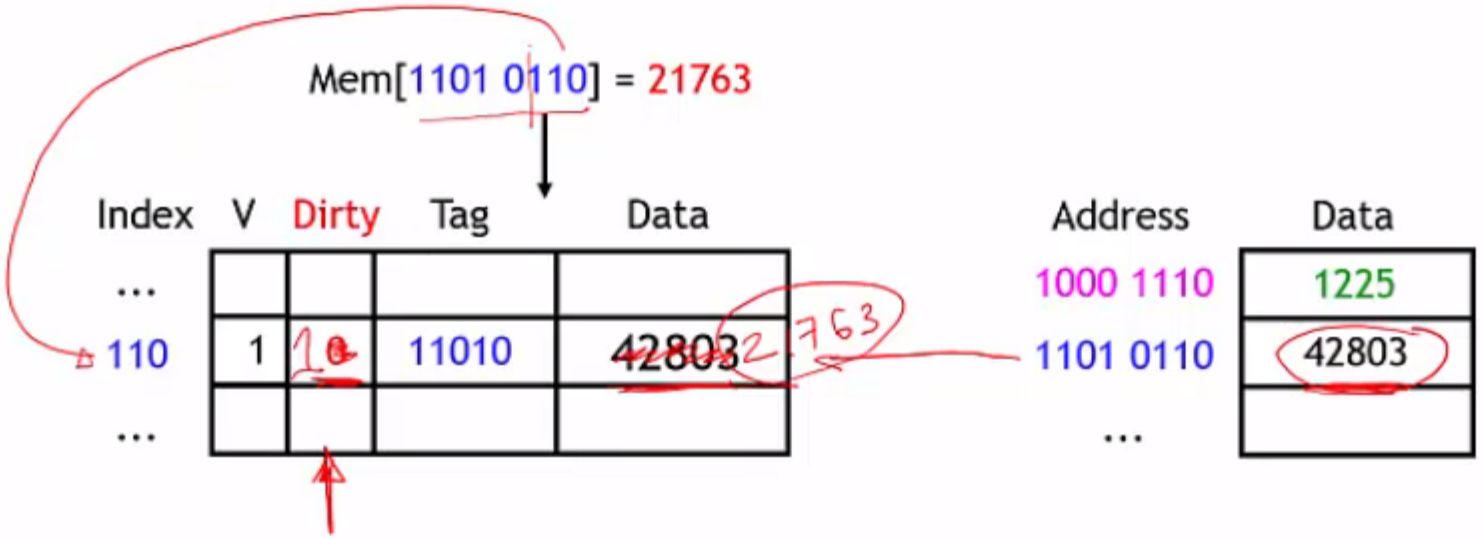
- In a write-back cache, the memory is not updated until the cache block needs to be replaced (*e.g.*, when loading data into a full cache set)
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before
  - The cache block is marked "dirty" to indicate this inconsistency



- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data

Write-back caches

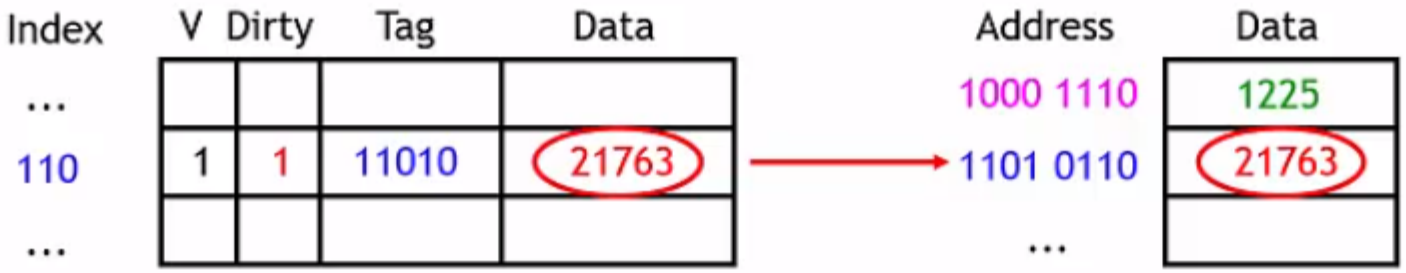
- In a write-back cache, the memory is not updated until the cache block needs to be replaced (*e.g.*, when loading data into a full cache set)
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before
  - The cache block is marked "dirty" to indicate this inconsistency



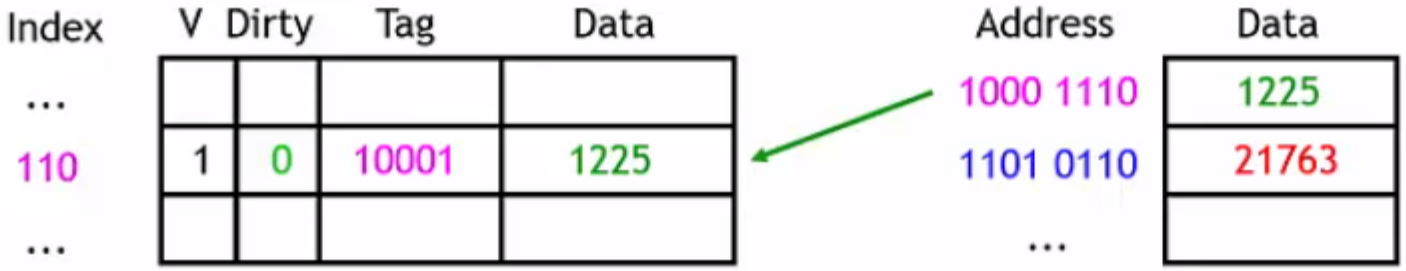
- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data

Finishing the write back

- We don't need to store the new value back to main memory unless the cache block gets replaced
- e.g. on a read from Mem[1000 1110], which maps to the same cache block, the modified cache contents will first be written to main memory



- Only then can the cache block be replaced with data from address 142



Numericals on cache mapping

2. Cache can hold 64KB of data.

Data is transferred between MM and the cache in blocks of 4 bytes each.

MM consists of 16MB.

Find the distribution of MM address in all 3 mapping techniques.

1. Direct Mapping

Since MM consists of 16MB.

$2^4 \times 2^{20} = 2^{24}$

i.e 2^{24} locations (addresses)

Hence no. of bits in MM address is 24.

No. of cache lines= $2^6 \times 2^{10} / 2^2 = 2^{14}$

To select one of the cache line 14 bits are required.

LINE = 14 bits, WORD = 2 (since block size = 4 bytes)

2. Associative Mapping

MM address

Word = 2 bits (since block size = 4 bytes)

TAG= 24-2 =22 bits

3. Set Associative Mapping

Assume 2 way set associative

i.e 2 cache lines make 1 set.

MM add.

Hence no. of sets $=2^{14}/2 = 2^{13}$

To select 1 of the 2^{13} sets we require 13 bits.

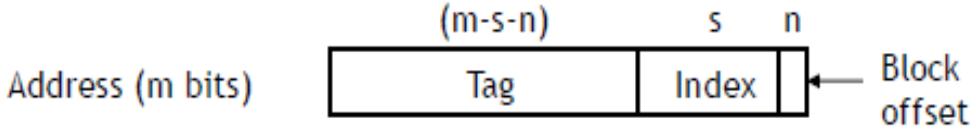
Hence no. of bits in SET field =13.

Hence no. of bits in WORD field =2.

Hence no. of bits in TAG field =24-(13+2) =9

Locating a set associative block

- We can determine where a memory address belongs in an associative cache in a similar way as before.
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

Block Offset = Memory Address mod 2^n

Block Address = Memory Address / 2^n

Set Index = Block Address mod 2^s