Hierarchical memory Technology

Presented By

Dr. Banchhanidhi Dash

School of Computer Engineering
KIIT University

Introduction

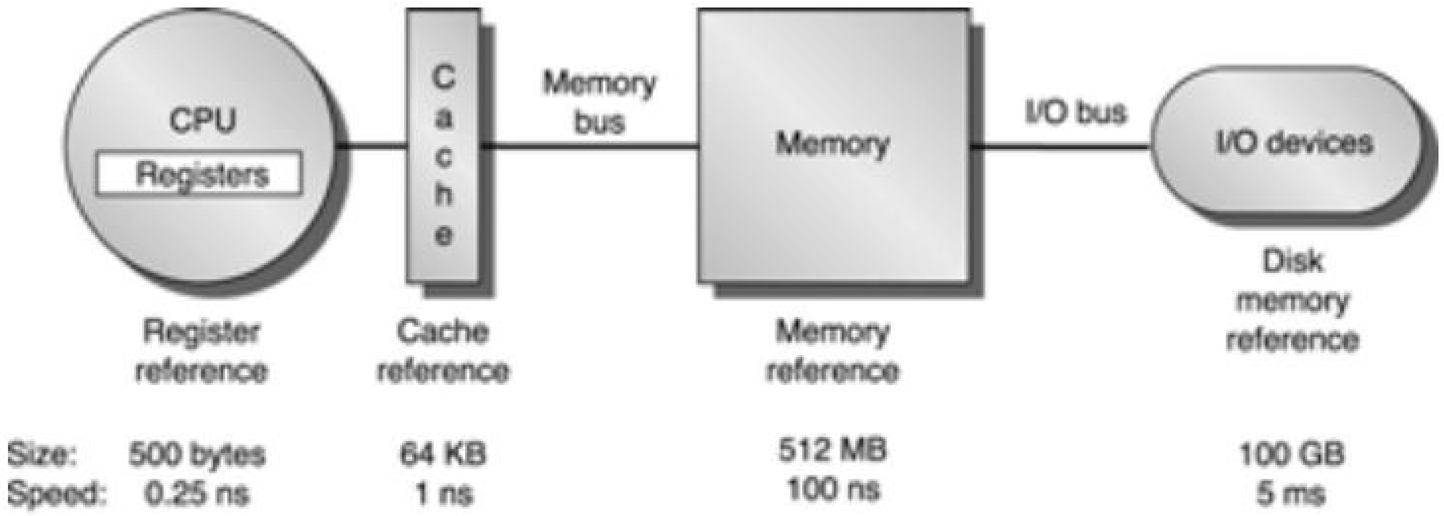
- Even a sophisticated processor may perform well below an ordinary processor:
  - Unless supported by matching performance by the memory system.
- The focus of this module:
  - Study how memory system performance has been enhanced through various innovations and optimizations.

The principle of locality…

- …says that most programs don't access all code or data uniformly
  - *e.g.* in a loop, small subset of instructions might be executed over and over again…
  - …& a block of memory addresses might be accessed sequentially…

- This has led to "memory hierarchies"
- Some important things to note:
  - Fast memory is expensive
  - Levels of memory usually smaller/faster than previous
  - Levels of memory usually "subset" one another
    - All the stuff in a higher level is in some level below it

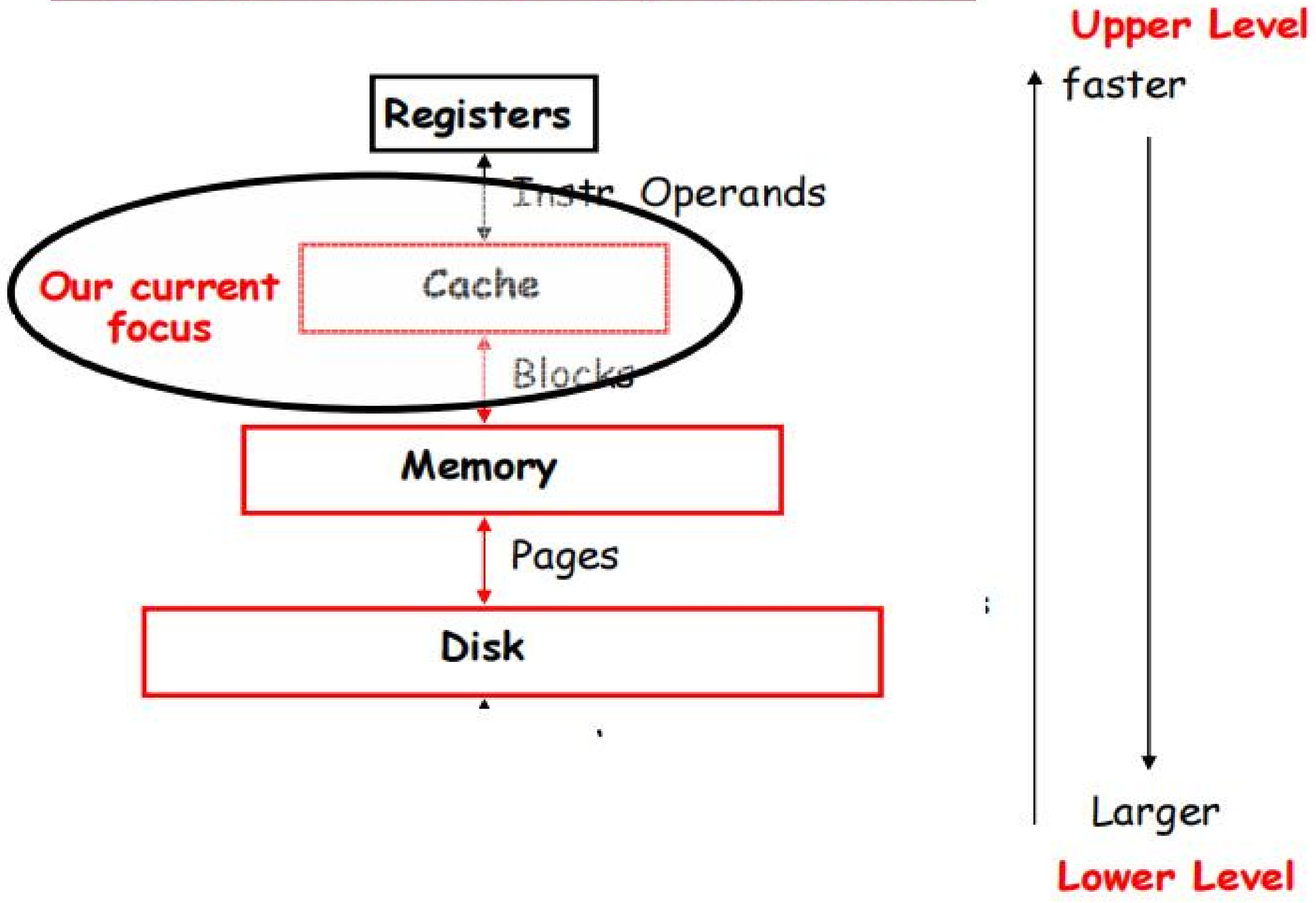
Levels in a typical memory hierarchy



Memory Hierarchies

- Key Principles
  - Locality – most programs do not access code or data uniformly
  - Smaller hardware is faster
- Goal
  - Design a memory hierarchy “with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level”
    - This implies that we be clever about keeping more likely used data as “close” to the CPU as possible
- Levels provide subsets
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory

The Full Memory Hierarchy "always reuse a good idea"

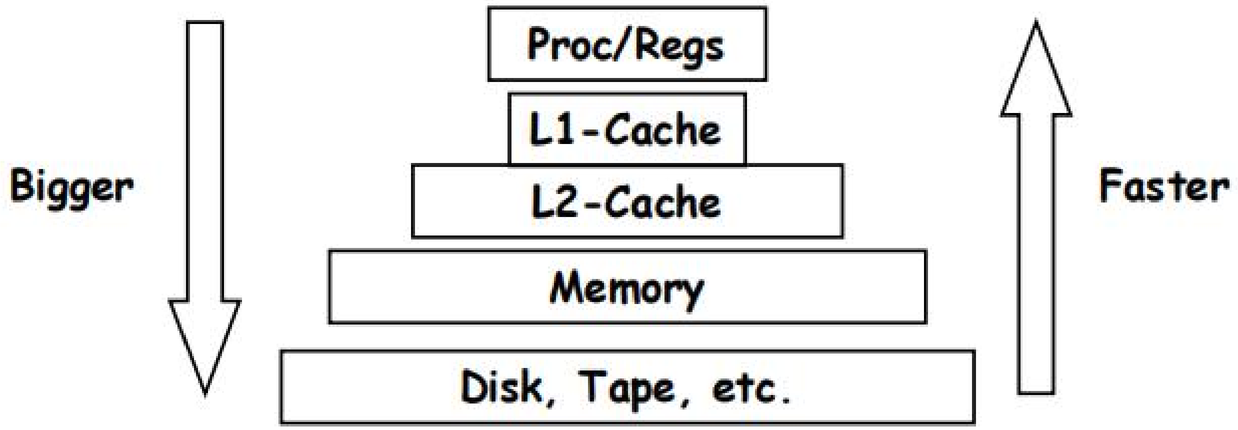


Cache: Terminology

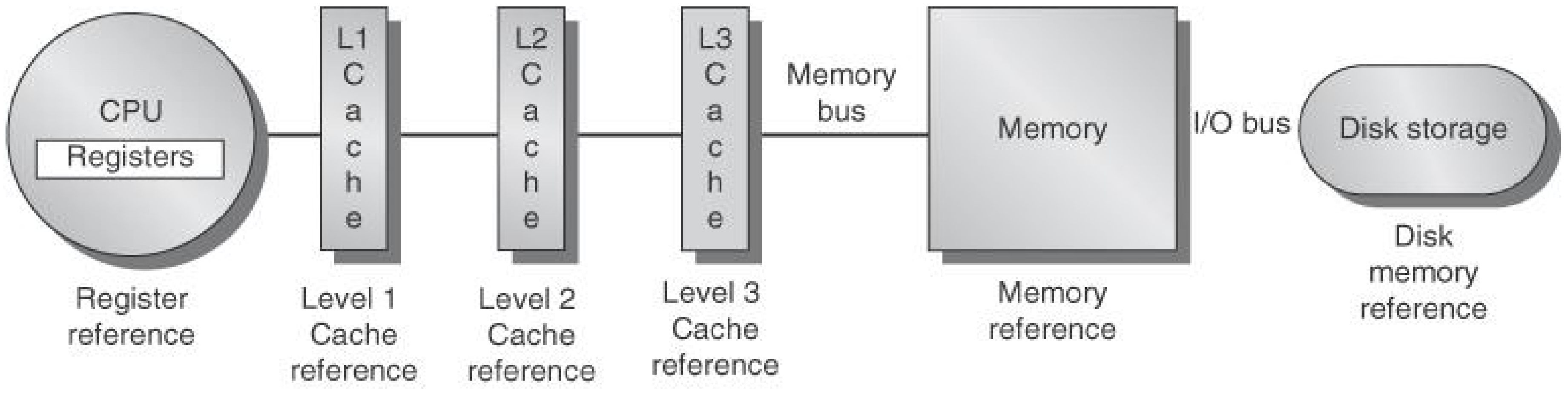
- Cache is name given to the first level of the memory hierarchy encountered once an address leaves the CPU
  - Takes advantage of the principle of locality
- The term cache is also now applied whenever buffering is employed to reuse items
- Cache controller
  - The HW that controls access to cache or generates request to memory

What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- Exploits spatial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers "a cache" on variables – software managed
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?

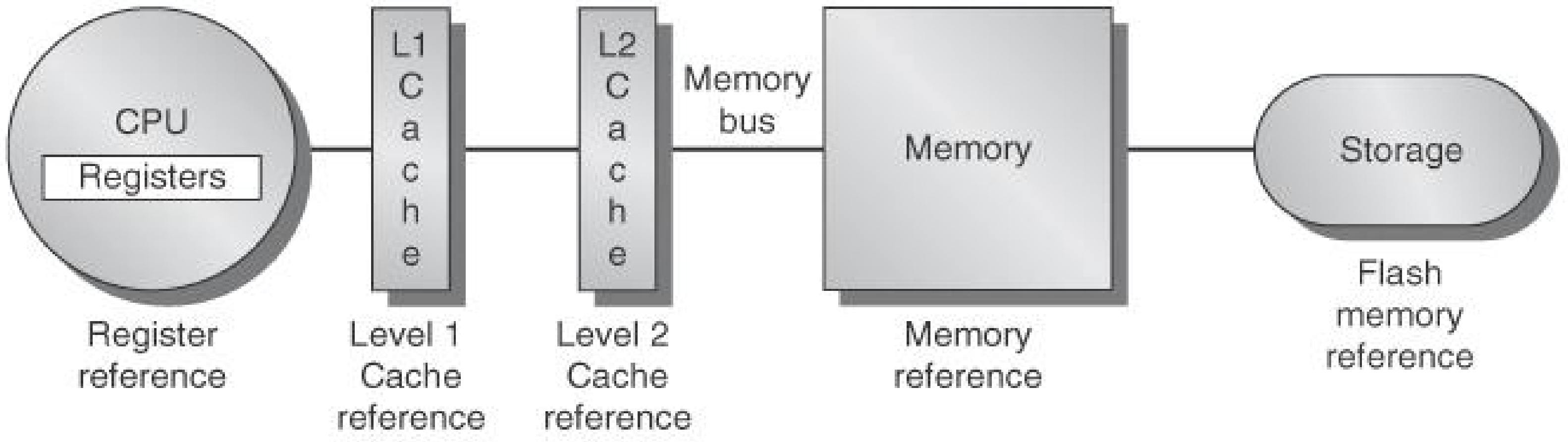


Multi-Level Cache



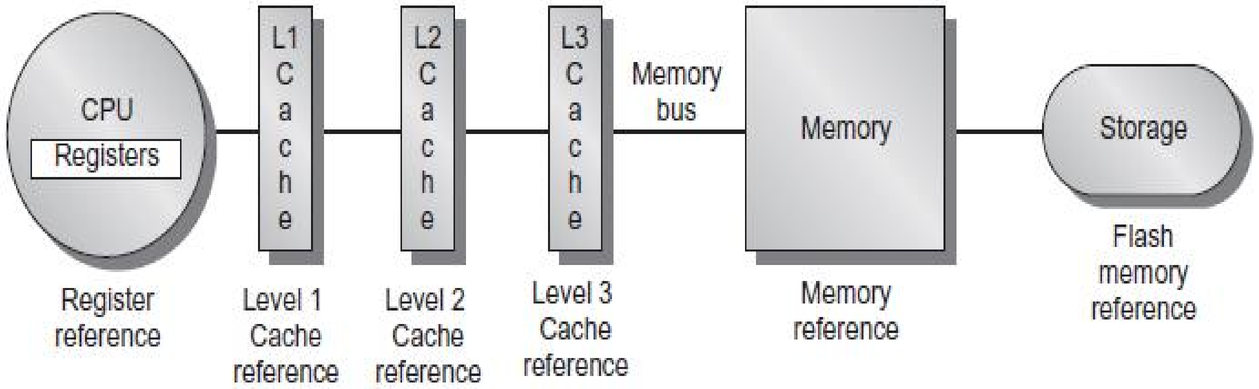
Size:	1000 bytes	64 KB	256 KB	2–4 MB	4–16 GB	4–16 TB
Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	5–10 ms

(a) Memory hierarchy for server



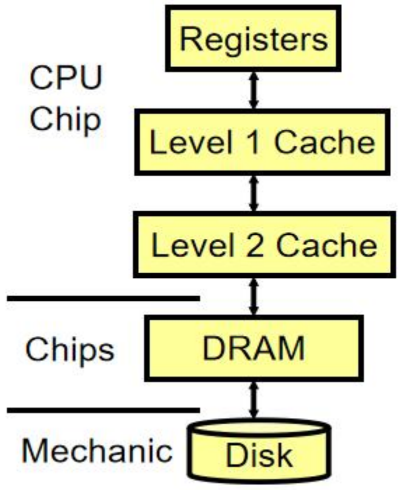
Size:	500 bytes	64 KB	256 KB	256–512 MB	4–8 GB
Speed:	500 ps	2 ns	10–20 ns	50–100 ns	25–50 us

(b) Memory hierarchy for a personal mobile device



		Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference	Memory reference	Flash memory reference
Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4–16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS

Memory hierarchy for a laptop or a desktop



	Cycle	Words/cycle	Management
Registers	1	3-10	Compiler
Level 1 Cache	1-3	1-2	Hardware
Level 2 Cache	5-10	1	Hardware
DRAM	30-100	0.5	OS
Disk	10^6-10^7	0.01	OS

What is the Role of a Cache?

- A small, fast storage used to improve average access time to a slow memory.
- Caches have no "inherent value", only try to close performance gap
- Improves memory system performance:
  - Exploits spatial and temporal locality
  - Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality (also termed data locality) refers to the use of data elements within relatively close storage locations.

A brief description of a cache

Cache = next level of memory hierarchy up from register file

- ✓ All values in register file should be in cache

- Cache entries usually referred to as "blocks"

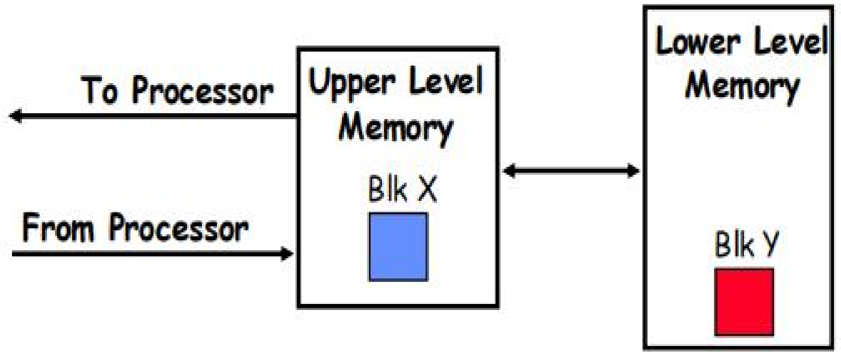
✓ Block is minimum amount of information that can be in cache

✓ fixed size collection of data, retrieved from memory and placed into the cache

- Processor generates request for data/inst, first look up the cache.If we're looking for item in a cache and find it, have a **cache hit**; if not then a **Cache miss**

- Hit: data appears in block in upper level (i.e. block X in cache)
  - Hit Rate: fraction of memory access found in upper level
  - Hit Time: time to access upper level which consists of
    - RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (i.e. block Y in memory)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Extra time to replace a block in the upper level +
    - Time to deliver the block the processor

Hit Time << Miss Penalty



Unified or Separate I-Cache and D-Cache

- **Two types of accesses:**
  - Instruction fetch
  - Data fetch (load/store instructions)
- **Unified Cache**
  - One large cache for both instructions and date
    - Pros: simpler to manage, less hardware complexity
    - Cons: how to divide cache between data and instructions? Confuses the standard harvard architecture model; optimizations difficult
- **Separate Instruction and Data cache**
  - Instruction fetch goes to I-Cache
  - Data access from Load/Stores goes to D-cache
    - Pros: easier to optimize each
    - Cons: more expensive; how to decide on sizes of each

Four Basic Questions

- Q1: Where can a block be placed in the cache? *(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the cache? *(Block identification)*
  - Tag/Block
- Q3: Which block should be replaced on a miss? *(Block replacement)*
  - Random, LRU
- Q4: What happens on a write? *(Write strategy)*
  - Write Back or Write Through (with Write Buffer)

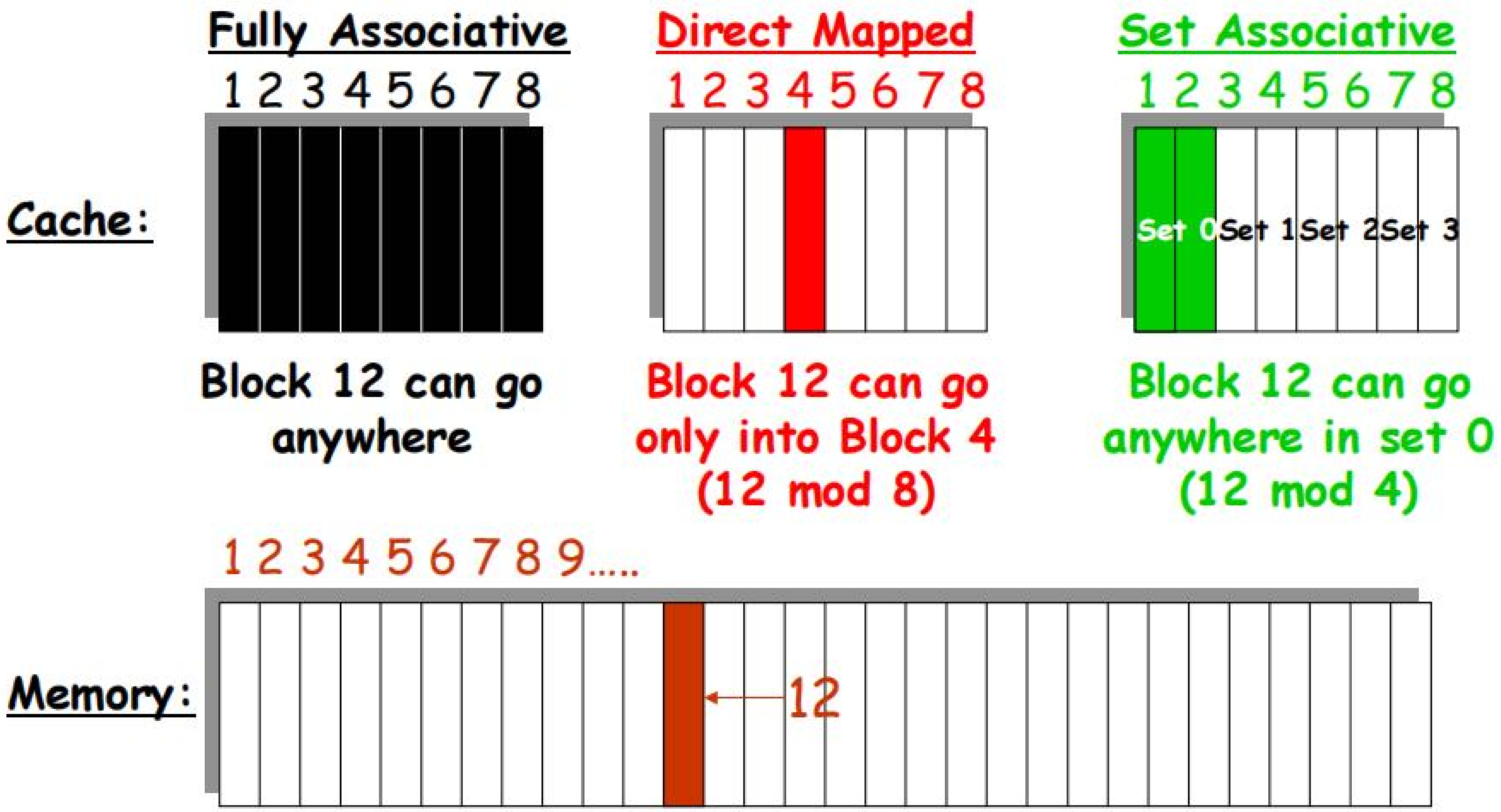
Block Placement

- If a block has only one possible place in the cache: direct mapped
- If a block can be placed anywhere: fully associative
- If a block can be placed in a restricted subset of the possible places: set associative
  - If there are n blocks in each subset: n-way set associative
- Note that direct-mapped = 1-way set associative

Where can a block be placed in a cache?

- 3 schemes for block placement in a cache:
  - Direct mapped cache:
    - Block (or data to be stored) can go to only 1 place in cache
    - Usually: (Block address) MOD (# of blocks in the cache)
  - Fully associative cache:
    - Block can be placed anywhere in cache
  - Set associative cache:
    - "Set" = a group of blocks in the cache
    - Block mapped onto a set & then block can be placed anywhere within that set
    - Usually: (Block address) MOD (# of sets in the cache)
    - If n blocks in a set, we call it n-way set associative

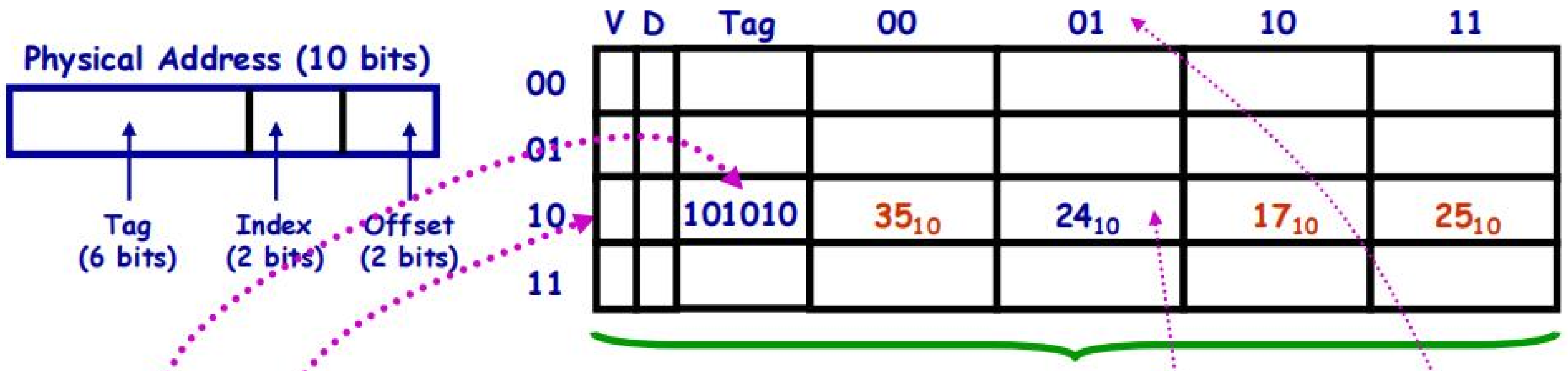
Where can a block be placed in a cache?



- **Direct Mapped vs Fully Associate**
  - Direct mapped is not flexible enough; if X(mod K)=Y(mod K) then X and Y cannot both be located in cache
  - Fully associative allows any mapping, implies all locations must be searched to find the right one –expensive hardware
- **Set Associative**
  - Compromise between direct mapped and fully associative
  - Allow many-to-few mappings
  - On lookup, subset of address bits used to generate an index
  - BUT index now corresponds to a set of entries which can be searched in parallel – more efficient hardware implementation than fully associative, but due to flexible mapping behaves more like fully associative

- **If total cache size is kept same, increasing the associativity increases number of blocks per set**
  - Number of simultaneous compares needed to perform the search in parallel = number of blocks per set
  - Increase by factor of 2 in associativity doubles number of blocks per set and halves number of sets

Analysis on MAPPING with a Example



A 4-entry direct mapped cache with 4 data words/block

1 Assume we want to read the following data words:

Tag	Index	Offset	Address Holds Data
101010	10	00	35_{10}
101010	10	01	24_{10}
101010	10	10	17_{10}
101010	10	11	25_{10}

All of these physical addresses would have the same tag

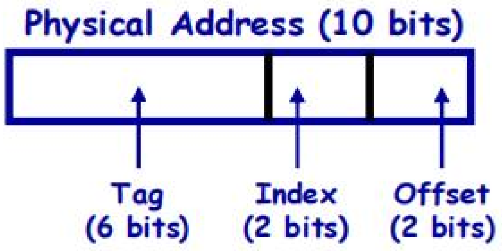
All of these physical addresses map to the same cache entry

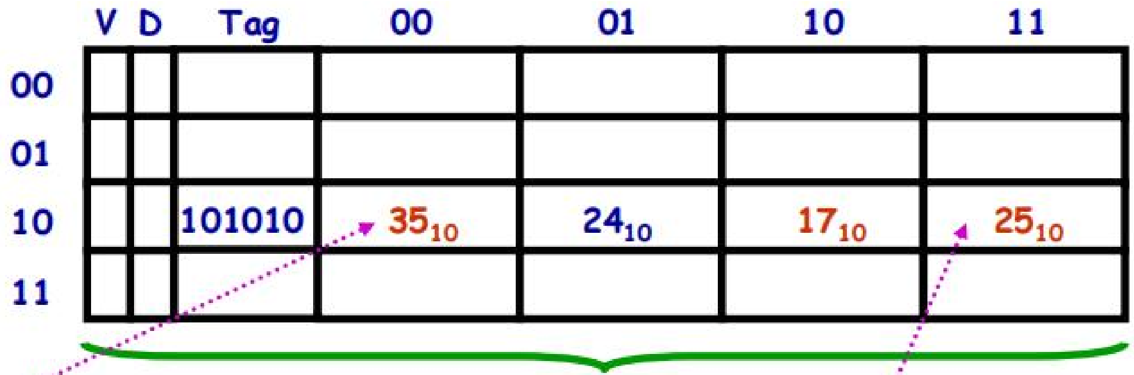
2 If we read 101010 10 01 we want to bring data word 24_{10} into the cache.

Where would this data go? Well, the index is 10. Therefore, the data word will go somewhere into the 3rd block of the cache. (make sure you understand terminology)

More specifically, the data word would go into the 2nd position within the block – because the offset is '01'

3 The principle of spatial locality says that if we use
one data word, we'll probably use some data words
that are close to it – that's why our block size is bigger than one data word. So we fill in the data word entries surrounding 101010 10 01 as well.





A 4-entry direct mapped cache with 4 data words/block

4 Therefore, if we get this pattern of accesses when we start a new program:

1.) 101010 10 00
2.) 101010 10 01
3.) 101010 10 10
4.) 101010 10 11

After we do the read for 101010 10 00 (word #1), we will automatically get the data for words #2, 3 and 4.

What does this mean? Accesses (2), (3), and (4) ARE NOT COMPULSORY MISSES

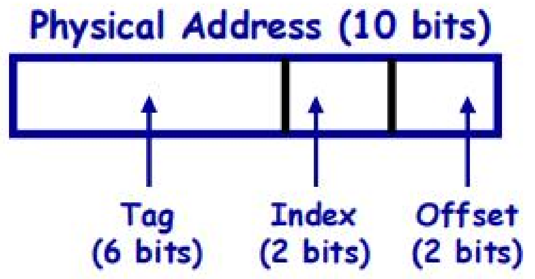
5 What happens if we get an access to location:
100011 | 10 | 11 (holding data: 12_{10})

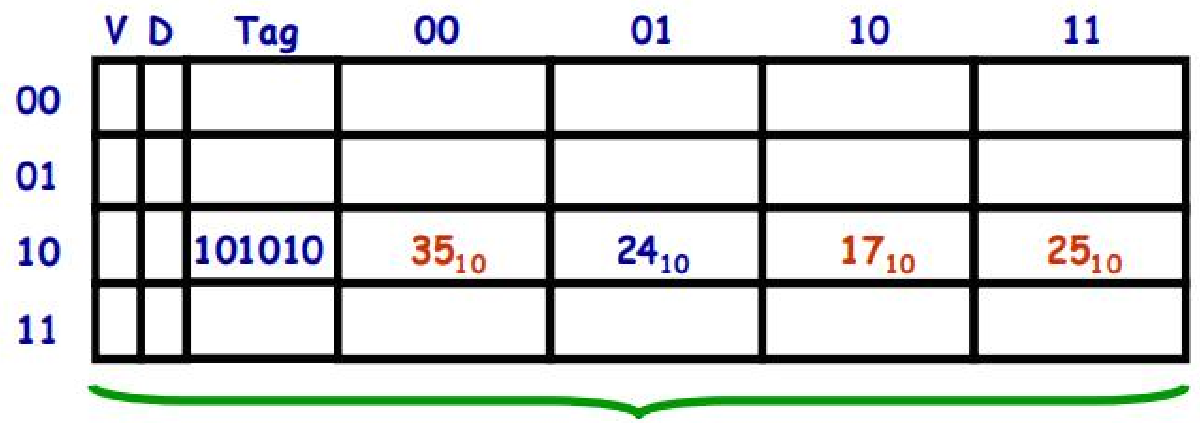
Index bits tell us we need to look at cache block 10.

So, we need to compare the tag of this address – 100011 – to the tag that associated with the current entry in the cache block – 101010

These DO NOT match. Therefore, the data associated with address 100011 10 11 IS NOT VALID. What we have here could be:

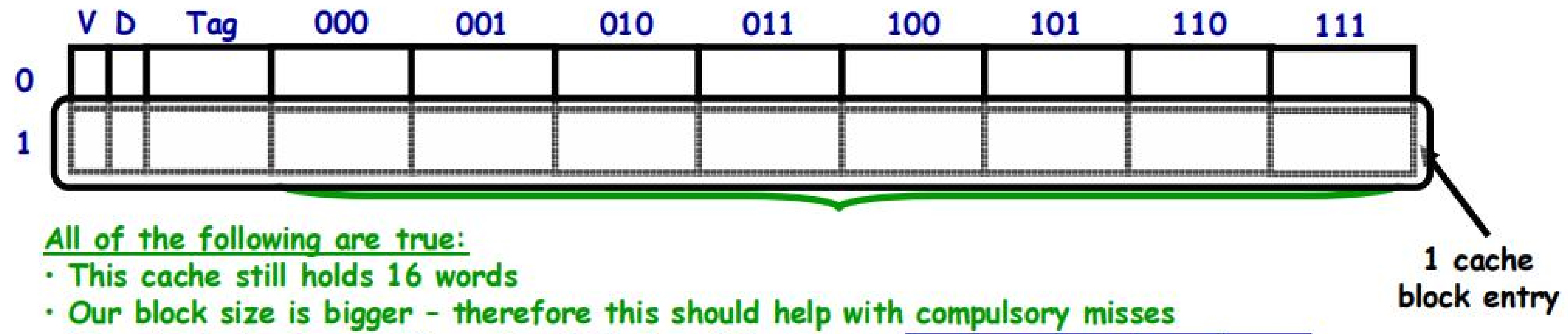
- A compulsory miss
  - (if this is the 1st time the data was accessed)
- A conflict miss:
  - (if the data for address 100011 10 11 was present, but kicked out by 101010 10 00 – for example)





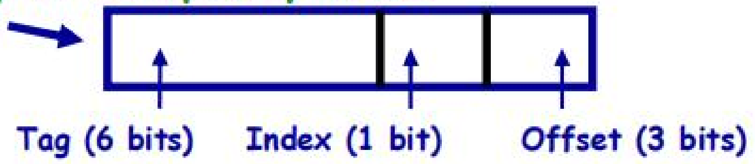
This cache can hold 16 data words…

6 What if we change the way our cache is laid out – but so that it still has 16 data words? One way we could do this would be as follows:



<u>All of the following are true:</u>

- This cache still holds 16 words
- Our block size is bigger – therefore this should help with compulsory misses
- Our physical address will now be divided as follows:
- The number of cache blocks has DECREASED
  - This will INCREASE the # of conflict misses



7 What if we get the same pattern of accesses we had before?

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1			101010	35_{10}	24_{10}	17_{10}	25_{10}	A_{10}	B_{10}	C_{10}	D_{10}

Note that there is now more data associated with a given cache block.

Pattern of accesses:
(note different # of bits for offset and index now)

1.) 101010 1 000
2.) 101010 1 001
3.) 101010 1 010
4.) 101010 1 011

However, now we have only 1 bit of index.
Therefore, any address that comes along that has a tag that is different than '101010' and has 1 in the index position is going to result in a conflict miss.

7 But, we could also make our cache look like this…

	V	D	Tag	0	1
000					
001					
010					
011					
100			101010	35_{10}	24_{10}
101			101010	17_{10}	25_{10}
110					
111					

There are now just 2 words associated with each cache block.

Again, let's assume we want to read the following data words:

	Tag	Index	Offset	Address Holds Data
1.)	101010	100	0	35_{10}
2.)	101010	100	1	24_{10}
3.)	101010	101	0	17_{10}
4.)	101010	101	1	25_{10}

Assuming that all of these accesses were occurring for the 1st time (and would occur sequentially), accesses (1) and (3) would result in compulsory misses, and accesses would result in hits because of spatial locality. (The final state of the cache is shown after all 4 memory accesses).

Note that by organizing a cache in this way, conflict misses will be reduced.
There are now more addresses in the cache that the 10-bit physical address can map too.

	V	D	Tag	0	1
000					
001					
010					
011					
100			101010	35_{10}	24_{10}
101			101010	17_{10}	25_{10}
110					
111					

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

All of these caches hold exactly the same amount of data – 16 different word entries

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1			101010	35_{10}	24_{10}	17_{10}	25_{10}	A_{10}	B_{10}	C_{10}	D_{10}

As a general rule of thumb, "long and skinny" caches help to reduce conflict misses, "short and fat" caches help to reduce compulsory misses, but a cross between the two is *probably* what will give you the best (i.e. lowest) overall miss rate.

But what about capacity misses?

8

What's a capacity miss?

- Can avoid capacity misses by making cache bigger

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

	V	D	Tag	00	01	10	11
000							
001							
010			10101	35_{10}	24_{10}	17_{10}	25_{10}
011							
100							
101							
110							
111							

Thus, to avoid capacity misses, we'd need to make our cache physically bigger – i.e. there are now 32 word entries for it instead of 16.

FYI, this will change the way the physical address is divided. Given our original pattern of accesses, we'd have:

Pattern of accesses:

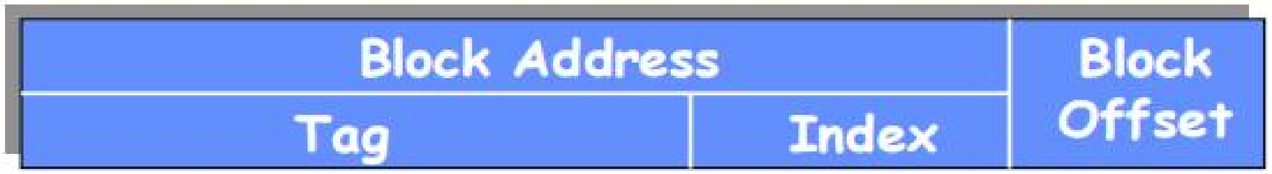
1.) 10101 | 010 | 00 = 35_{10}
2.) 10101 | 010 | 01 = 24_{10}
3.) 10101 | 010 | 10 = 17_{10}
4.) 10101 | 010 | 11 = 25_{10}

(note smaller tag, bigger index)

How is a block found in the cache?

- Cache's have address tag on each block frame that provides block address
  - Tag of every cache block that might have entry is examined against CPU address (in parallel! – why?)
- Each entry usually has a valid bit
  - Tells us if cache data is useful/not garbage
  - If bit is not set, there can't be a match...
- How does address provided to CPU relate to entry in cache?
  - Entry divided between block address & block offset...
  - ...and further divided between tag field & index field

How is a block found in the cache?



- Block offset field selects data from block
  - (i.e. address of desired data within block)
- Index field selects a specific set
  - Fully associative caches have no index field
- Tag field is compared against it for a hit

- Could we compare on more of address than the tag?
  - Not necessary; checking index is redundant
    - Used to select set to be checked
    - Ex.: Address stored in set 0 must have 0 in index field
  - Offset not necessary in comparison –entire block is present or not and all block offsets must match

Which block should be replaced on a cache miss?

- If we look something up in cache and entry not there, generally want to get data from memory and put it in cache
  - B/c principle of locality says we'll probably use it again

- Direct mapped caches have 1 choice of what block to replace
- Fully associative or set associative offer more choices
- Usually 2 strategies:
  - Random – pick any possible block and replace it
  - LRU – stands for "Least Recently Used"
    - Why not throw out the block not used for the longest time

Cache Write Policies

What happens on a write?

- Generically, there are 2 kinds of write policies:
  - Write through (or *store through*)
    - With write through, information written to block in cache *and* to block in lower-level memory
  - Write back (or *copy back*)
    - With write back, information written only to cache. It will be written back to lower-level memory when cache block is replaced

- The dirty bit:
  - Each cache entry usually has a bit that specifies if a write has occurred in that block or not…
  - Helps reduce frequency of writes to lower-level memory upon block replacement

What happens on a write?

- What if we want to write and block we want to write to isn't in cache?

- There are 2 common policies:
  - Write allocate: (or *fetch on write*)
    - The block is loaded on a write miss
    - The idea behind this is that subsequent writes will be captured by the cache (ideal for a write back cache)
  - No-write allocate: (or *write around*)
    - Block modified in lower-level and *not* loaded into cache
    - Usually used for write-through caches
      - (subsequent writes still have to go to memory)

Write Policies: Analysis

- **Write-back**
  - Implicit priority order to find most up to date copy
  - Require much less bandwidth
  - Careful about dropping updates due to losing track of dirty bits
- **What about multiple levels of cache on same chip?**
  - Use write through for on-chip levels and write back for off-chip
    - SUN UltraSparc, PowerPC
- **What about multi-processor caches ?**
  - Write back gives better performance but also leads to cache coherence problems

Write Policies: Analysis

- **Write through**
  - Simple
  - Correctness easily maintained and no ambiguity about which copy of a block is current
  - Drawback is bandwidth required; memory access time
  - Must also decide on decision to fetch and allocate space for block to be written
    - Write allocate: fetch such a block and put in cache
    - Write-no-allocate: avoid fetch, and install blocks only on read misses
      - Good for cases of streaming writes which overwrite data

Example: Assume a fully associative write-back cache with many cache entries that starts empty. Below is sequence of five memory operations (The address is in square brackets):

Write Mem[100];
Write Mem[100];
Read Mem[200];
Write Mem[200];
Write Mem[100].

What are the number of hits and misses (inclusive reads and writes) when using no-write allocate versus write allocate?

Answer

No-write Allocate:

Write Mem[100];	1 write miss
Write Mem[100];	1 write miss
Read Mem[200];	1 read miss
Write Mem[200];	1 write hit
Write Mem[100].	1 write miss

4 misses; 1 hit

Write allocate:

Write Mem[100];	1 write miss
Write Mem[100];	1 write hit
Read Mem[200];	1 read miss
Write Mem[200];	1 write hit
Write Mem[100];	1 write hit

2 misses; 3 hits