CSEN 702

Microprocessors

Lectures 5/6

# Memory hierarchy design

## Parts 1 and 2

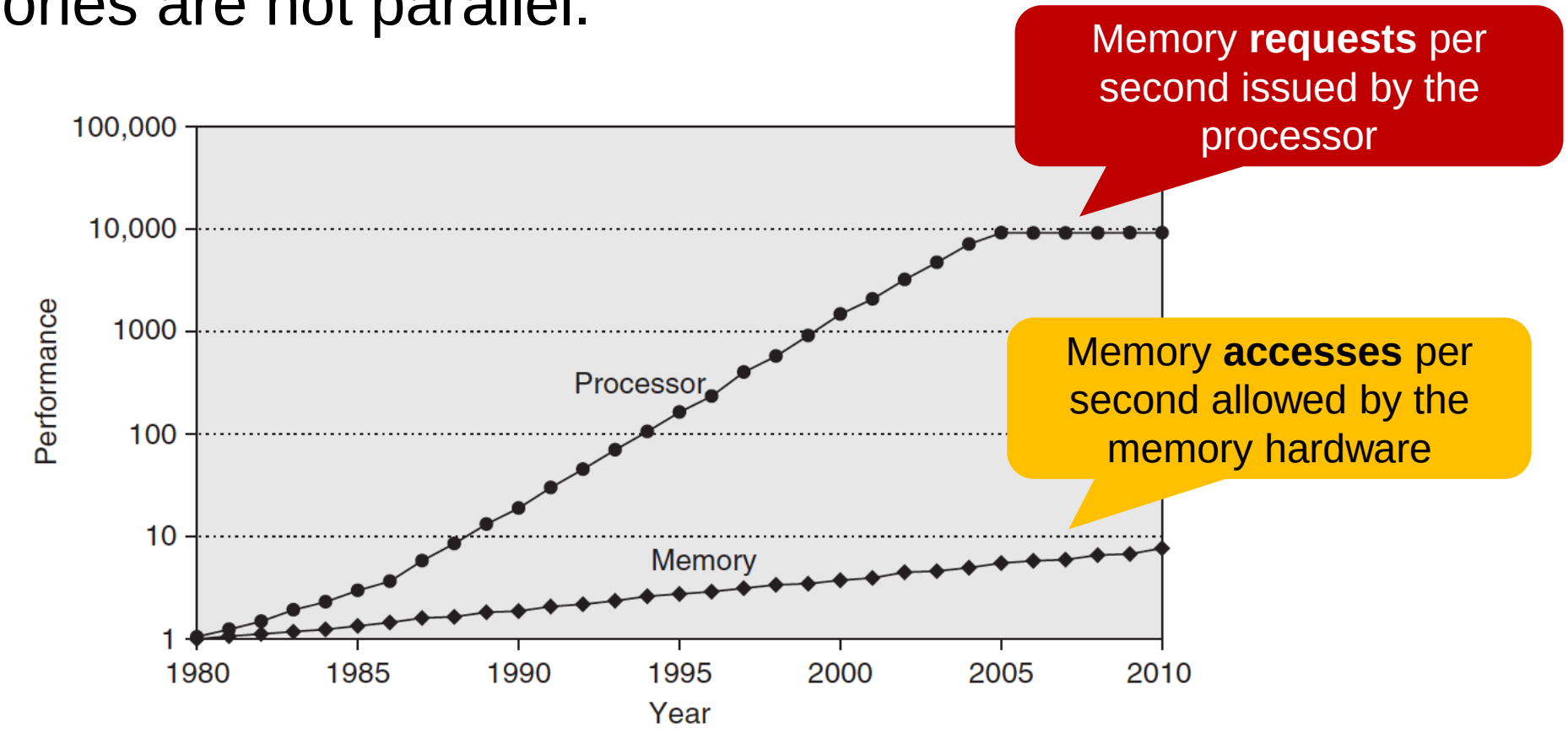
**Reading**

Textbook Chapter 2, Appendix B

# Part 1 Intro to memory hierarchy

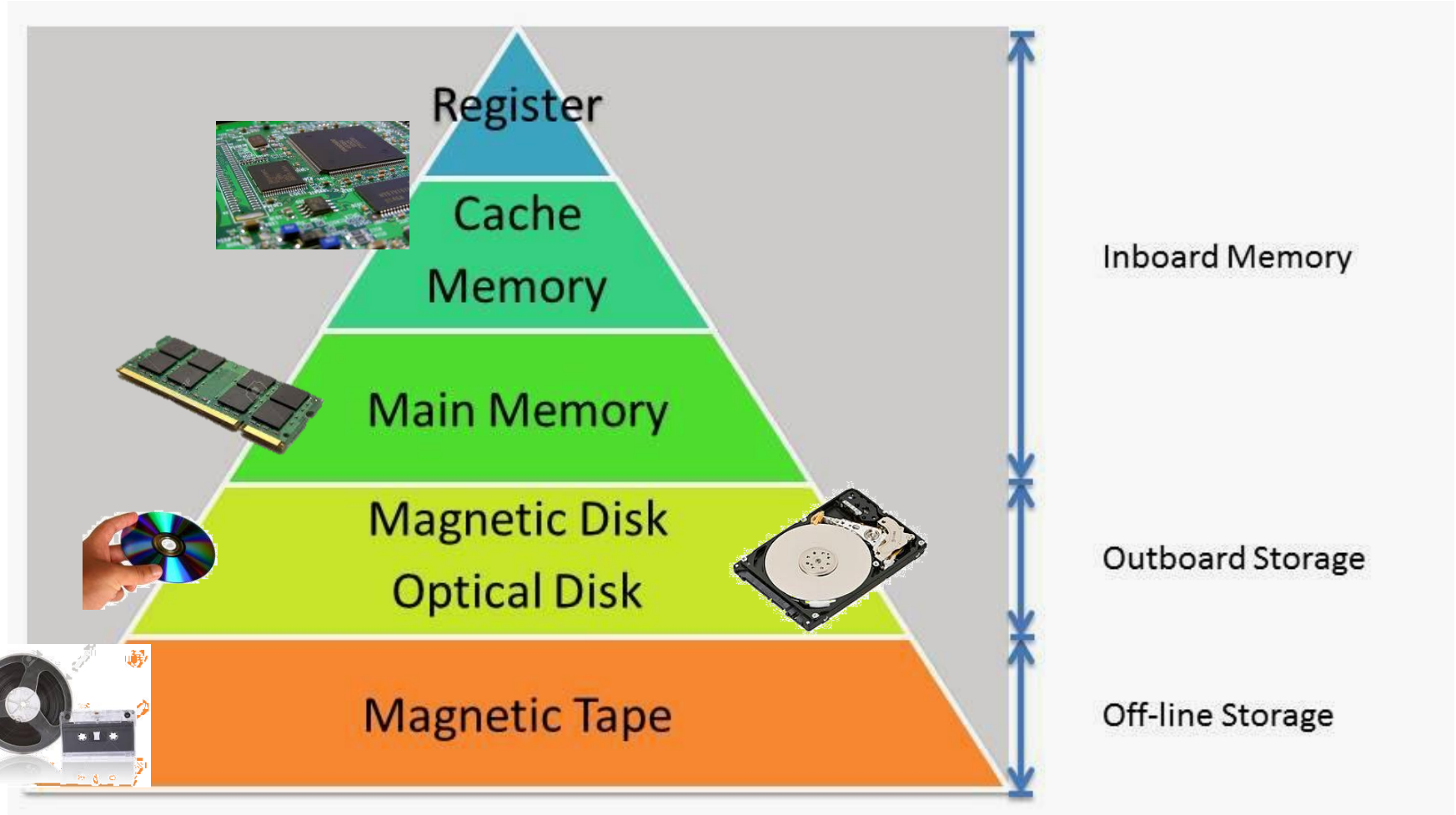
# Why do we care?

- Advancements in speeds/performance of processors and memories are not parallel.



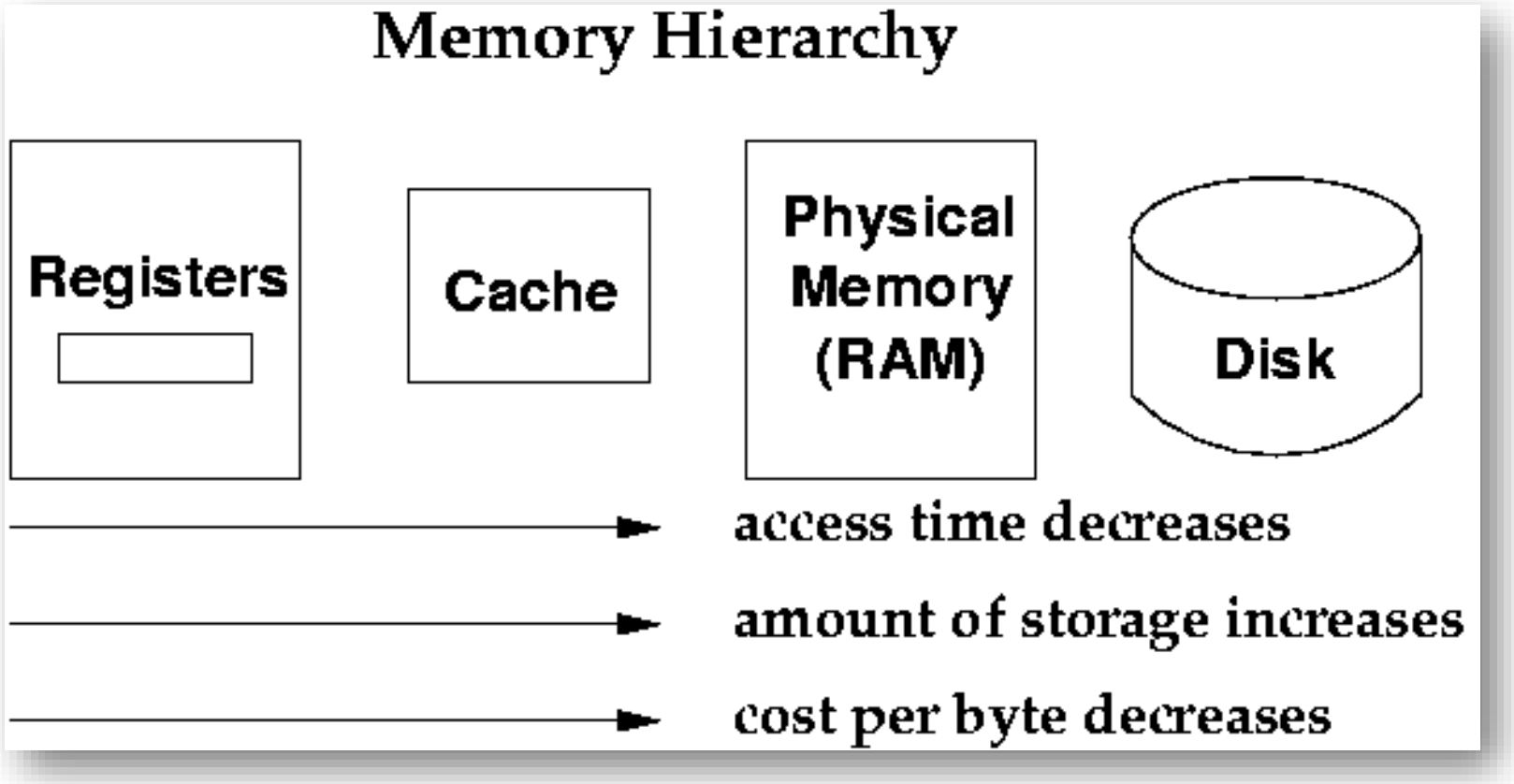
# Solutions

- Split the memory into a **hierarchy** of different types and sizes.

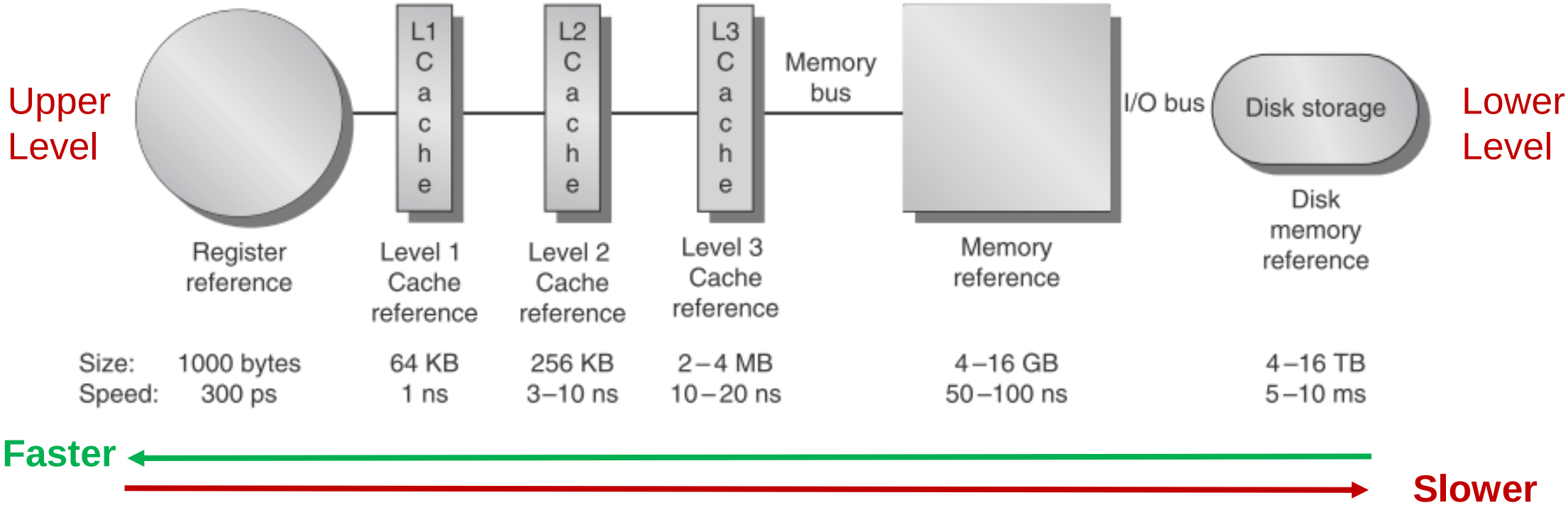


# What's the point?

- Smaller sizes close to the processor, fast access
- Bigger sizes further, slower access
- Biggest sizes even further, slowest access

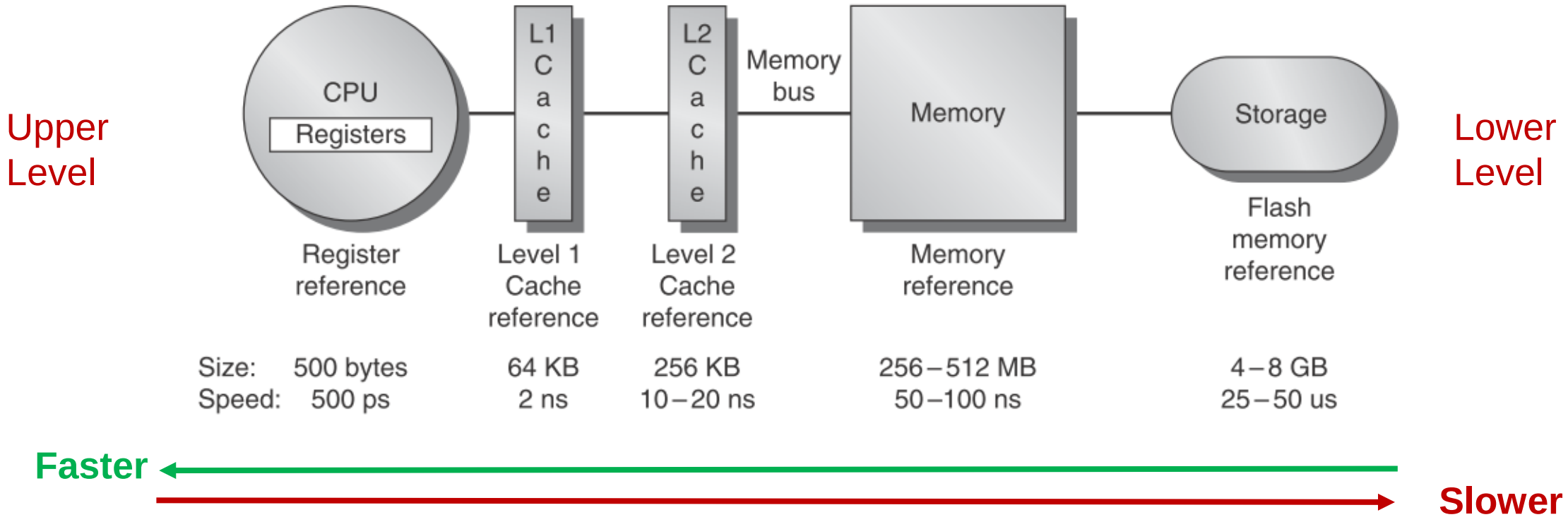


# Memory Hierarchy of a Server




- Notice that the time units change by a factor of $10^9$ (picoseconds to milliseconds) and the size units change by a factor of $10^{12}$ (bytes to TeraBytes)

# Memory Hierarchy of a PMD




- Personal Mobile Devices have slower clock rate, smaller caches and main memory
- Unlike servers that use disks as the lowest level in the hierarchy, PMDs use FLASH (built from EEPROM technology)

# Typical sizes and speeds of memories

| Level | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Name | Registers | Cache | Main memory | Disk storage |
| Typical size | <1 KB | 32 KB–8 MB | <512 GB | >1 TB |
| Implementation technology | Custom memory with multiple ports, CMOS | On-chip CMOS SRAM | CMOS DRAM | Magnetic disk |
| Access time (ns) | 0.15–0.30 | 0.5–15 | 30–200 | 5,000,000 |
| Bandwidth (MB/sec) | 100,000–1,000,000 | 10,000–40,000 | 5000–20,000 | 50–500 |
| Managed by | Compiler | Hardware | Operating system | Operating system/ operator |
| Backed by | Cache | Main memory | Disk | Other disks and DVD |

# Part 2 Cache categories

# Caches

- Cache is the name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor.

- **Quick examples**: instruction and data memory in the datapath are actually called instruction and data caches.

- The concept is when the processor needs data, it looks first in the cache to find it.

# Cache hits and misses

- When the processor finds a requested data item in the cache, it is called a **cache hit**.

- When the processor does not find a data item it needs in the cache, a **cache miss** occurs.

- A fixed-size collection of data containing the requested word, called a **block** or line run, is retrieved from the main memory and placed into the cache, after a cache miss.
  - Why not just the word needed?

# Principle of locality

## Temporal locality

We are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly

## Spatial locality

There is a high probability that the other data in the block will be needed soon.

# CPU execution, updated!

- If we take into account the stall cycles of the processor, waiting for memory access, the CPU execution time formula needs to be updated.

CPU execution time = (CPU clock cycles + Memory stall cycles) x Clock cycle time

Includes processor execution time including time of **cache hit**

Waiting time on a **cache miss**

- **But how are memory stall cycles computed?**

# Memory stall cycles

- The number of memory stall cycles depends on both:
  1. The number of misses
  2. The cost per miss, which is called the **miss penalty** (measured in # of cycles)

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$= IC \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$= IC \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}$$

**The component *miss rate* is simply the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by number of accesses**

# Cache categories

- Whenever a block is brought from main memory to the cache, it must be placed at a specific part of the cache.
- Main memory addresses are much more than caches so same address placement is not an option.

- Three famous cache approaches to place a block:
  1. **Direct mapped**
  2. **Fully associative**
  3. **Set associative**

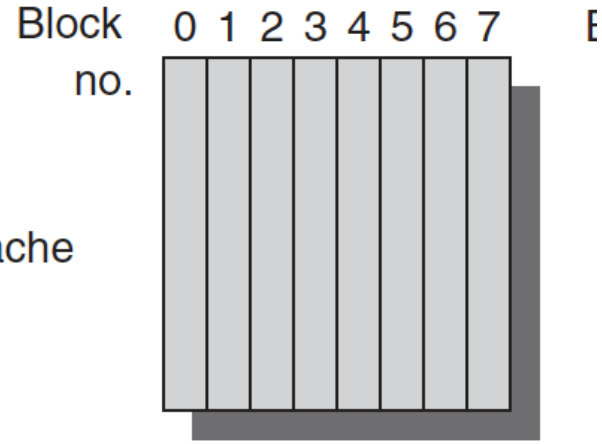
# Cache categories (cont.)

- Assume we want to place Block 12 from main memory into one of the three cache categories. Given that cache can hold 8 blocks.

*cks)*

Place block in any of the sets 0 to 7

1 Fully associative: block 12 can go anywhere

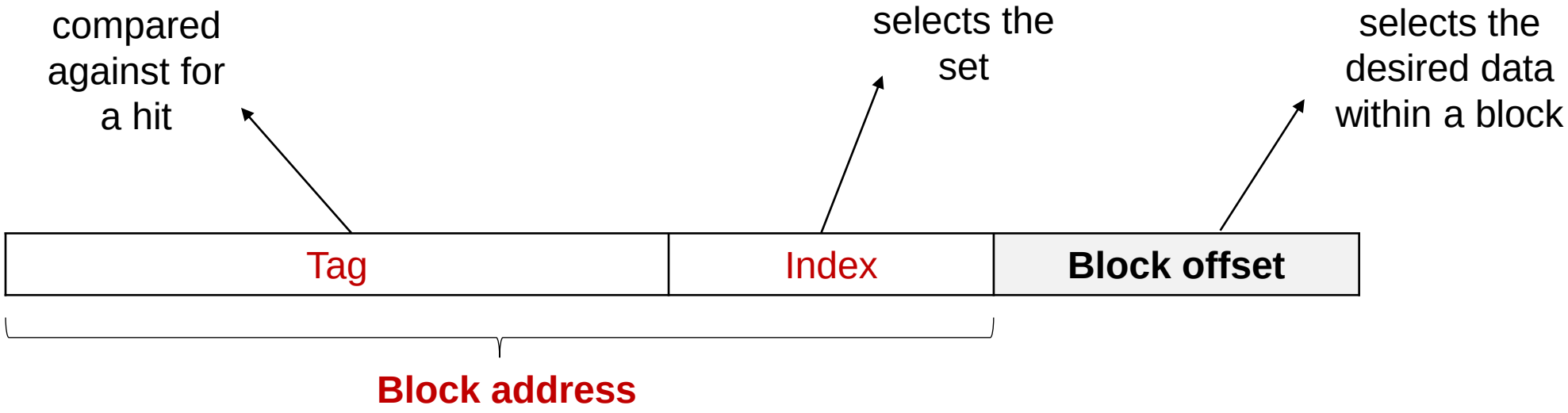



Place block in set: *(block #) MOD (# of sets)*

set can fit more than 1 block. Once set is chosen, place block anywhere inside

# How is a block found in the cache?

- The address of the desired data is divided into 3 fields: Tag, index, and block offset.




Additional bit: **Valid bit**, set to 1, whenever valid block is brought into the cache.

# How is a block found in the cache?

# Example 1: Direct mapped

- Consider 12-bit addresses
- 128 Bytes cache with 32 Bytes blocks

- **Block offset:** To find a byte in the block, we need log(32) = 5 bits
- **Index**: Number of sets in direct mapped is 128/32 = 4 sets → 2 bits
- **Tag**: Remaining number of bits 12-(5+2) = 5 bits.

| Tag : 5 bits | Index: 2 bits | Block offset: 5 bits |
|---|---|---|

# How is a block found in the cache?

# Example 1 (cont.)

| Address | Tag | Index (set) | Offset |
|---|---|---|---|
| **0x070** | **00000** | **11** | **10000** |
| 0x080 | 00001 | 00 | 00000 |
| 0x068 | 00000 | 11 | 01000 |
| 0x190 | 00011 | 00 | 10000 |

M

| Set | Valid bit | Tag |
|---|---|---|
| 00 | **0** | |
| 01 | **0** | |
| 10 | **0** | |
| 11 | ~~0~~ 1 | 00000 |

1) data at address 0x070 (or 000001110000) is requested. We go to the cache to set 11, valid bit is 0, means no valid data yet, so it's a miss.

→ **Therefore, We bring the block and store it and change the valid bit to 1.**

→ **then we use offset to read the desired byte within the block.**

# How is a block found in the cache?

# Example 1 (cont.)

| Address | Tag | Index (set) | Offset |
|---|---|---|---|
| 0x070 | 00000 | 11 | 10000 |
| **0x080** | **00001** | **00** | **00000** |
| 0x068 | 00000 | 11 | 01000 |
| 0x190 | 00011 | 00 | 10000 |

M

M

| Set | Valid bit | Tag |
|---|---|---|
| 00 | ~~0~~ 1 | 00001 |
| 01 | **0** | |
| 10 | **0** | |
| 11 | ~~0~~ 1 | 00000 |

2) data at address 0x080 (or 000010000000) is requested. We go to the cache to set 00, valid bit is 0, means no valid data yet, so it's a miss again.

→ **Therefore, We bring the block and store it and change the valid bit to 1.**

→ **then we use offset to read the desired byte within the block.**

# How is a block found in the cache?

# Example 1 (cont.)

| Address | Tag | Index (set) | Offset |
|---|---|---|---|
| 0x070 | 00000 | 11 | 10000 |
| 0x080 | 00001 | 00 | 00000 |
| **0x068** | **00000** | **11** | **01000** |
| 0x190 | 00011 | 00 | 10000 |

M
M
H

| Set | Valid bit | Tag |
|---|---|---|
| 00 | ~~0~~ 1 | 00001 |
| 01 | **0** | |
| 10 | **0** | |
| 11 | ~~0~~ 1 | 00000 |

3) data at address 0x068 (or 000001101000) is requested. We go to the cache to set 11, valid bit is 1, so now we compare the tags, also equal! so it's a hit**!**

**Now we use the offset to bring the byte needed.**

# How is a block found in the cache?

# Example 1 (cont.)

| Address | Tag | Index (set) | Offset | |
|---|---|---|---|---|
| 0x070 | 00000 | 11 | 10000 | M |
| 0x080 | 00001 | 00 | 00000 | M |
| 0x068 | 00000 | 11 | 01000 | H |
| **0x190** | **00011** | **00** | **10000** | M |

| Set | Valid bit | Tag |
|---|---|---|
| 00 | ~~0~~ X 1 | ~~00001~~ XXXXX 00011 |
| 01 | **0** | |
| 10 | **0** | |
| 11 | ~~0~~ X 1 | 00000 |

4) data at address 0x190 (or 000110010000) is requested. We go to the cache to set 00, valid bit is 1, so now we compare the tags, not equal! so it's a miss.

→ **remove the block from the cache to bring in the new one.**

# Which block should be replaced on a miss?

- When you search for a block# in the cache and you don’t find it, you must bring in this block from the main memory.
- One block in the cache must be swapped out to make room for the new block.
- In direct-mapped caches, there’s only one option
- In associative caches, several approaches can be employed.
  1. **Random**: choose any block
  2. **LRU**: Least recently used
  3. **FIFO**: First on First out.

## How is a block found in the cache?

# Example 2: two-way set associative

- total size of cache 128 Bytes
- Block size: 32 Bytes
- Two-way assoc. → Each set contains 2 blocks.
- Total number of sets = 128 / (2x32) = 2 sets

- **We need:**
  - 5 bits **block offset** (as before, to find a byte inside a block)
  - 1 bit for set **index** (since we have only 2 sets)
  - Remaining 12-6 = 6 bits for the **tag**

| Tag : 6 bits | Index: 1 bit | Block offset: 5 bits |
|---|---|---|

# How is a block found in the cache?

# Example 2: two-way set associative

| Address | Tag | Index (set) | Offset |
|---|---|---|---|
| **0x070** | **000001** | **1** | **10000** |
| 0x080 | 000010 | 0 | 00000 |
| 0x068 | 000001 | 1 | 01000 |
| 0x190 | 000110 | 0 | 10000 |

M

| | | Way 1 | | Way 0 | |
|---|---|---|---|---|---|
| | **LRU** | **V** | **Tag** | **V** | **Tag** |
| **Set 0** | | | | | |
| **Set 1** | way 1 | | | 1 | 000001 |

1) data at address 0x070 (or 000001110000) is requested. We go to the cache to set 1, valid bit is 0, means no valid data yet, so it's a miss.

→ **Therefore, We bring the block and store it in set 1 (in way 0 as it's empty now) and change the valid bit to 1.**

→ **Also set LRU of set 1 to way 1 to denote that way 1 is the Least recently used now**

# How is a block found in the cache?

# Example 2: two-way set associative

| Address | Tag | Index (set) | Offset | |
|---|---|---|---|---|
| 0x070 | 000001 | 1 | 10000 | M |
| **0x080** | **000010** | **0** | **00000** | M |
| 0x068 | 000001 | 1 | 01000 | |
| 0x190 | 000110 | 0 | 10000 | |

| | | Way 1 | | Way 0 | |
|---|---|---|---|---|---|
| | **LRU** | **V** | **Tag** | **V** | **Tag** |
| **Set 0** | way 1 | | | 1 | 000010 |
| **Set 1** | way 1 | | | 1 | 000001 |

2) data at address 0x080 (or 000010000000) is requested. We go to the cache to set 0, valid bit is 0, means no valid data yet, so it's a miss again.

→ **Therefore, We bring the block and store it in way 0 and change the valid bit to 1.**

→ **Also set LRU to way 1**

# How is a block found in the cache?

# Example 2: two-way set associative

| Address | Tag | Index (set) | Offset | |
|---|---|---|---|---|
| 0x070 | 000001 | 1 | 10000 | M |
| 0x080 | 000010 | 0 | 00000 | M |
| **0x068** | **000001** | **1** | **01000** | **H** |
| 0x190 | 000110 | 0 | 10000 | |

| | | Way 1 | | Way 0 | |
|---|---|---|---|---|---|
| | **LRU** | **V** | **Tag** | **V** | **Tag** |
| **Set 0** | way 1 | | | 1 | 000010 |
| **Set 1** | way 1 | | | 1 | 000001 |

3) data at address 0x068 (or 000001101000) is requested.
We go to the cache to set 1, check the tag and find it's equal and it's valid. so it's a hit.

# How is a block found in the cache?

# Example 2: two-way set associative

| Address | Tag | Index (set) | Offset | |
|---|---|---|---|---|
| 0x070 | 000001 | 1 | 10000 | M |
| 0x080 | 000010 | 0 | 00000 | M |
| 0x068 | 000001 | 1 | 01000 | H |
| **0x190** | **000110** | **0** | **10000** | M |

| | | Way 1 | | Way 0 | |
|---|---|---|---|---|---|
| | **LRU** | **V** | **Tag** | **V** | **Tag** |
| **Set 0** | **way 0** | 1 | 000110 | 1 | 000010 |
| **Set 1** | way 1 | | | 1 | 000001 |

4) data at address 0x0190 (or 000110010000) is requested.
We go to Set 0, compare the tags (00010 != 000110), so it's a miss and need to bring it.
Check the LRU to know where to bring the block, we find way 1, so we bring the block to way 1, and update the LRU to way 0

# Writing to the cache

- There are **two approaches** when writing data to the cache:
  - **Write-through**—The information is written to both the block in the cache **and** to the block in the lower-level memory (level 2 cache, RAM…)
  - **Write-back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced later.

- To reduce the frequency of writing back blocks on replacement, a **dirty bit** is used. It indicates whether the block is dirty (modified while in the cache) or clean (not modified).
- If it is clean, the block is not written back on a miss, since identical information to the cache is found in lower levels.

# Which one is better?

- **Write-back** is attractive for multi-processors since some writes don’t go back to main memory and hence less bandwidth

- **Write-through** is easier to implement and the next lower level memory has always fresh data which is also good for multiprocessor to keep data coherency.
- If the processor must wait for the write to be done, it’s called **write stall**. It can be solved using a *write buffer*.

# Write misses

- When we miss on a write to the cache, there are also two options:

1. **Write-allocate**: the block is allocated (brought into the cache) on a write miss, followed by the write hit actions mentioned before (write back and write through)

2. **No-Write-allocate**: Misses do not affect the cache. The block is only written to lower level memory.

# Example

- Assume a fully associative write-back cache with many cache entries, and cache starts empty.
- Below is a sequence of five memory operations.
- **Question**: What are the number of hits and misses when using no-write allocate **versus** write allocate?

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

# Solution

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

- **Option 1: No-write allocate**
  - the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses.
  - Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit.
  - The last write to 100 is still a miss.
  - **The result for no-write allocate is 4 misses and 1 hit.**

- **Option 2: Write allocate**
  - First accesses to 100 and 200 are misses, and the rest are hits since 100 and 200 are both found in the cache.
  - **Thus, the result for write allocate is 2 misses and 3 hits.**

# Part 2 Cache performance

# Average memory access time

- A better measure of memory hierarchy performance is the **average memory access time**:

$$\textit{Average memory access time} = \textit{Hit time} + \textit{Miss rate} \times \textit{Miss penalty}$$

* *hit time* is the time to hit in the cache

- This formula can help us decide between **split caches** and a **unified cache**.