Computer Architecture. Week 13

Caching: Main Principles and Their Application to a Memory System Design

Alexander Tormasov

Innopolis University

a.tormasov@innopolis.ru

November 18, 2021

Content of the class

- The recap of a computer memory hierarchy
- General principles of caching
- The principles of spacial and temporal locality
- A simple cache design

Memory

- Any element that holds its state in a digital system can be thought of as a form of memory.
- Today's computers depend upon large and fast storage systems.
- Large storage capacities are needed for many database applications, scientific computations with large data sets, video and music, and so forth.

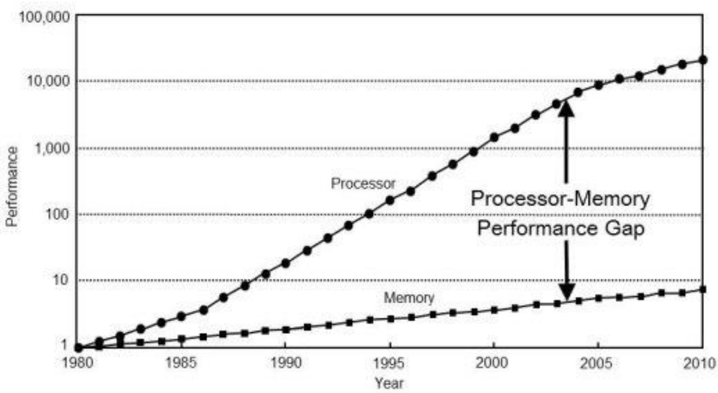
Trade-off between Speed, Cost and Capacity

- Unfortunately there is a tradeoff between speed, cost and capacity.

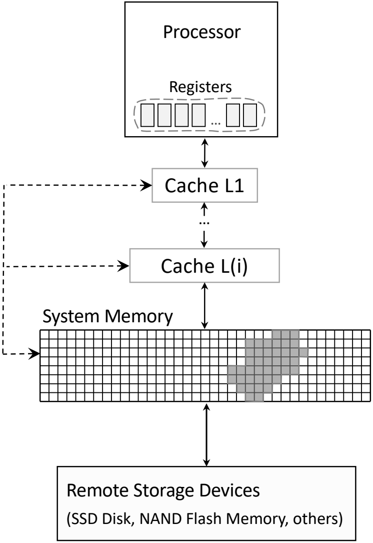
Storage	**Speed**	**Cost**	**Capacity**
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
NVRAM	Fast	Expensive	Small
Solid-State Drive (SSD)	Slow	Cheap	Largest
Hard Disks	Slowest	Cheapest	Largest

- Fast memory is too expensive
- Dynamic memory has a much longer delay than other functional units in a datapath.
- If every `lw` or `sw` accessed dynamic memory, we would have to either increase the cycle time or stall frequently.

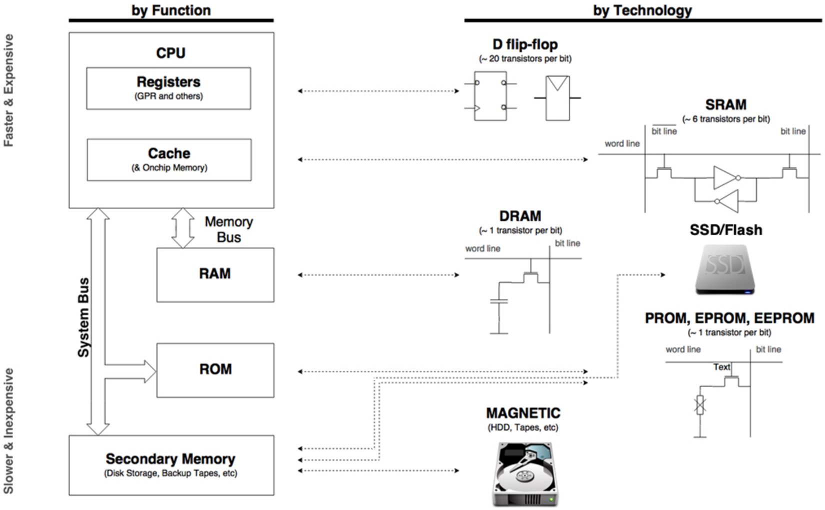
Processor and Memory Performance Gap



Solution: Memory Hierarchy



Memory Hierarchy

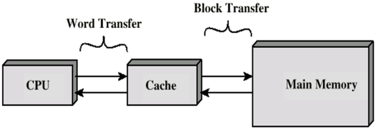


A General Principle of Caching

- Caching can be either physical or logical;
- Caching in general refers to the mechanism of matching a big address space into small address space, and not necessarily to a hardware cache memory unit
- Caching is widely used in Operating Systems, Computer Networks, and other areas

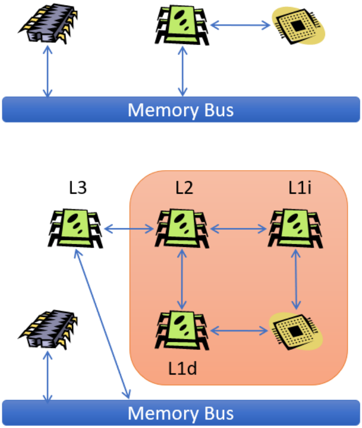
Physical Cache Memory (1/3)

- Cache is an intermediate buffer with fast access containing information that can be most likely requested.
- Access to data in the cache is faster than fetching source data from a slower memory or from a remote source.
- However, its volume is significantly limited compared to the source data storage.
- Cache can be located on CPU chip.



Physical Cache Memory (2/3)

- Modern systems use **several types** of cache (for data, instructions, etc.)
- Many cache units are **physically located** by a CPU side
- Many cache systems are organized **hierarchically**
- Usually hyper threads of the same CPU share **L1 cache**
- Usually different CPU cores share **L2 cache** (cores are grouped in pairs)
- **L3** is generally shared by all cores



Physical Cache Memory (3/3)

- Cache is a small amount of fast, expensive memory.
  - The cache goes between the processor and the slower, dynamic main memory.
  - It keeps a copy of the most frequently used data from the main memory.
- Note: Cache is different from a communication bus!
- Memory access speed increases overall, because we have made the common case faster.
  - Reads and writes to the most frequently used addresses will be serviced by the cache.
  - We only need to access the slower main memory for less frequently used data.

Motivation for a Virtual Memory

- A large program or many active programs may not be entirely resident in the main memory
- Use secondary storage to hold portions exceeding memory capacity
- Needed portions are *automatically* loaded into the memory, replacing other portions Programmers need not be aware of actions; **virtual memory** hides capacity limitations

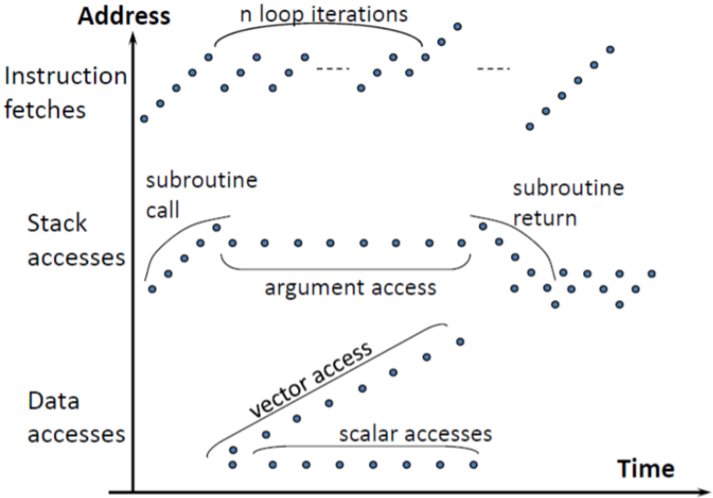
Note: More details in OS course

The Principle of Locality (1/2)

- Programs access a small proportion of their address space at any time
- **Temporal locality (in time)**
  - Weather forecasting for tomorrow from today's weather conditions with probability of 70% will be correct.
  - Items accessed recently are likely to be accessed again soon
  - For Example: Instructions in a loop, *induction variables
- **Spatial locality (in space)**
  - Items near those accessed recently are likely to be accessed soon
  - For Example: A sequential instruction access

NOTE: An induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop

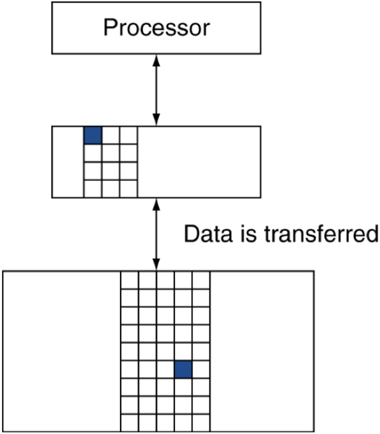
The Principle of Locality (2/2)



Taking Advantage of Locality

- Memory hierarchy
  - Copy more recently accessed (and nearby) items from DRAM to smaller SRAM
    - Cache memory attached to CPU
    - Hardware implementation of a cache
- Store everything on disk
  - Copy recently accessed (and nearby) items from disk to DRAM
    - Main memory
    - Software implementation of a cache
- Remark on caching efficiency:
  - Caching efficiency might be low, if a program does not comply well to the principles of spacial and temporal localities

Memory Hierarchy Levels)



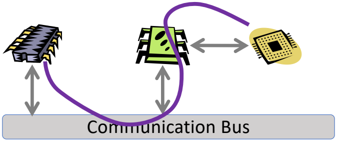
Cache Hit/Miss

- When CPU initiates a **request to memory**, the request first goes to its cache

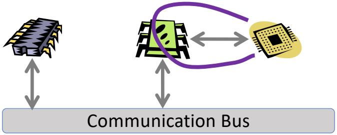
- **Cache miss** (data with a required address is not present in cache). The steps:
  1. access to cache;
  2. access to main memory;
  3. freeing up memory space in cache;
  4. loading into cache (!);
  5. finally passing data from cache to processor

- **Cache hit** (data with a required address is present in cache). The steps:
  - access to cache;
  - data transfer from cache to processor

Cache miss rate drastically affects the performance (e.g. the miss rate of 6% leads to 6 times higher runtime)



Cache miss scenario



Cache hit scenario

The Memory Hierarchy: Terminology (1/2)

- **Hit Rate:** The fraction of memory accesses found in a level of the memory hierarchy
- **Hit Time:** Time to access that level which consists of:
  `Time to access the block + Time to determine hit/miss`
- **Miss Rate:** The fraction of memory accesses not found in a level of the memory hierarchy that is 1 - hit rate

The Memory Hierarchy: Terminology (2/2)

- **Miss Penalty:** Time to replace a block in that level with the corresponding block from a lower level which consists of:

  `Time to access the block in the lower level + Time to transmit that block to the level that experienced the miss + Time to insert the block in that level + Time to pass the block to the requestor`

  `Hit Time << Miss Penalty`

Cache Characteristics

- Granularity and Capacity
  - Data in cache is stored by means of a cache line, with a typical size of 32 or 64 bytes
  - Cache storage capacity is limited; impossible to store all data. Stored data should be chosen effectively
- Cache hit/miss rate
  - The measure of effectiveness
  - Depends on locality
- Cache types:
  - Fully associative cache
  - Directly mapped cache

Cache Characteristics

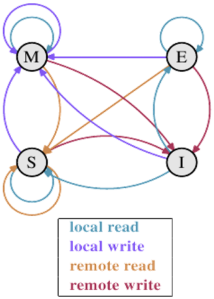
- **Cache Operations**
  - Cache read/write:
    - To read data from cache, we read an entire cache line
    - Drastically increases the number of transactions to the main memory
  - To write data into cache, we use `Write Buffer`:
    - If not all data in a cache line is up-to-date, then write operation into cache is performed only after the entire cache line is updated from the main memory

Multiple Caches Coherence: MESI protocol

- Cache coherence – the uniformity of a shared data stored in multiple (local) caches;
- Several mechanisms (protocols) for cache coherence are available, and one of them is MESI

- MESI principle:
  - Each cache line has a flag taking one of these values:
    - **Modifed (M)** – line is present only in this cache; its value is modified, but main memory is not updated;
    - **Exclusive (E)** – line is present only in this cache; corresponds to data in main memory
    - **Shared (S)** – line can be present in other caches, but corresponds to data in main memory
    - **Invalid (I)** – line is invalid; data should be updated
  - Read/write operations are performed based on this flag:
    - Read is permitted for all states except "Invalid"
    - Write is permitted for states "Modified" or "Exclusive"
  - Operation in a local cache affects the flags of the respective lines in other caches (e.g. write to a line in a given cache leads to invalid flags set for its copies in other caches)

Transition of states for MESI (within some cache):



Concept of Dirty Bit/Modified Bit

- A dirty bit or modified bit is a bit that is associated with a block of computer memory and indicates whether or not the corresponding block of memory has been modified.
- The dirty bit is set when the processor writes to (modifies) this memory.
- The bit indicates that its associated block of memory has been modified and has not been saved to storage yet.
- When a block of memory is to be replaced, its corresponding dirty bit is checked to see if the block needs to be written back to secondary memory before being replaced or if it can simply be removed.
- Dirty bits are used by the CPU cache and in the page replacement algorithms of an operating system.

A Simple Cache Design (1/2)

- Caches are divided into blocks, which may be of various sizes.
  - The number of blocks in a cache is usually a **power of 2.**
  - Here is an example cache with eight blocks, each holding one byte.

Block index	**8 bit data**
000	
001	
010	
011	
100	
101	
110	
111	

A Simple Cache Design (2/2)

Four important questions

1. When we copy a block of data from main memory to the cache, where exactly should we put it?

2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we would have to replace one of the existing blocks in the cache... which one?

4. How can write operations be handled by the memory system?

Where Should We Put Data in the Cache?

- Fully associative cache
- Direct-mapped cache

Fully Associative Cache

- A fully associative cache is the simplest approach that permits data to be stored in any cache block.
- When data is fetched from memory, it can be placed in any unused block of the cache.
- If all the blocks are already in use, it's usually best to replace the least recently used (LRU) one, assuming that if it hasn't used it in a while, it won't be needed again anytime soon.
- However, a fully associative cache is expensive to implement.

Least Recently Used (LRU) Algorithm

- Idea of LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache.

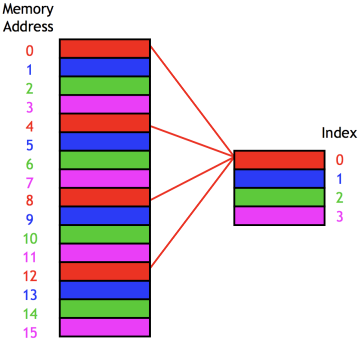
- A good approximation to the optimal algorithm is based on the observation that data that have been heavily used in the last few instructions will probably be heavily used again in the next few.

- Conversely, data that have not been used for ages will probably remain unused for a long time

- LRU cache is implemented by using a doubly linked list and a hash with page number as key and address of the corresponding queue node as value

Direct-Mapped Cache (1/3)

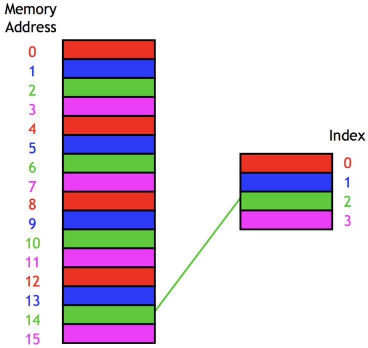
- A direct-mapped cache is another approach: each **main memory address** maps to exactly one **cache block.**
- For example: A 16-byte main memory and a 4-byte cache (four 1-byte blocks).
- Memory locations 0, 4, 8 and 12 all map to cache block 0.
- Addresses 1, 5, 9 and 13 map to cache block 1, etc.
- How can we compute this mapping?



Direct-Mapped Cache (2/3)

- If the cache contains 2^k blocks, then the data at memory address i would go to cache block index
  - Formula:

    `(Memory address) modulo (Number of blocks in cache)`

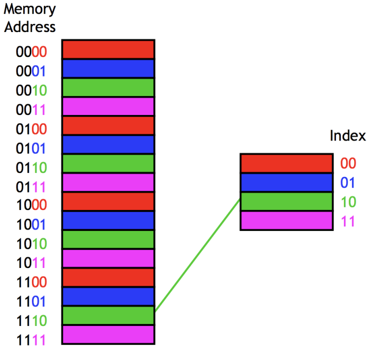


- For example: `14 mod 4 = 2`

Direct-Mapped Cache (3/3)

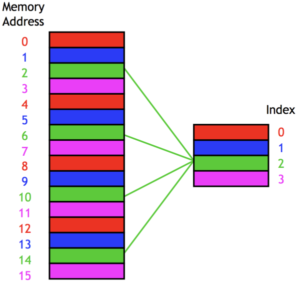
- An equivalent way to find the placement of a memory address in the cache is to look at least significant **k** bits of the address.

- With our four-byte cache we would inspect the two least significant bits of our memory addresses.
- Again, you can see that address 14 (1110 in binary) maps to cache block 2 (10 in binary).
- Taking the least k bits of a binary value is the same as computing that value mod 2^k



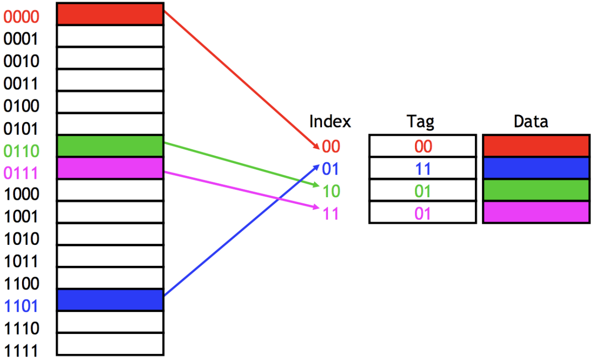
How Can We Find Data in the Cache?

- If we want to read memory address **i**, we can use the mod trick to determine which cache block would contain **i**.
- But other addresses might also map to the same cache block. How can we distinguish between them?
- For Example: The cache block 2 could contain data from addresses 2, 6, 10 or 14.



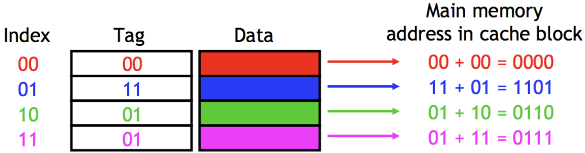
Adding Tags (1/2)

- We need to add **tags** to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.



Adding Tags (2/2)

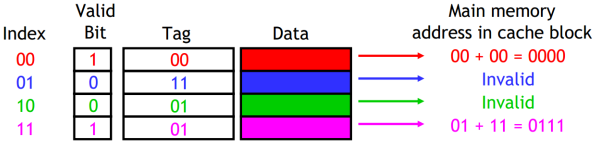
- Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices.



Index	Tag	Data	Main memory address in cache block
00	00	🟥	00 + 00 = 0000
01	11	🟦	11 + 01 = 1101
10	01	🟩	01 + 10 = 0110
11	01	🟪	01 + 11 = 0111

One More Detail: Valid Bit

- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.

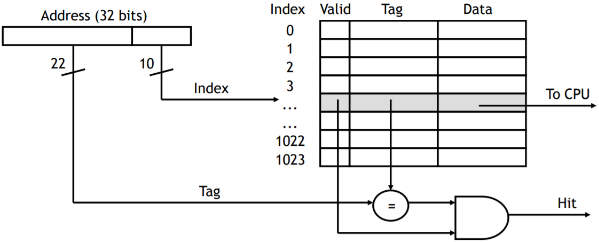


Index	Valid Bit	Tag	Data	Main memory address in cache block
00	1	00	🟥	00 + 00 = 0000
01	0	11	🟦	Invalid
10	0	01	🟩	Invalid
11	1	01	🟪	01 + 11 = 0111

- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

What Happens on a Cache Hit?

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest k bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper (m - k) bits of the m-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10} -byte cache.



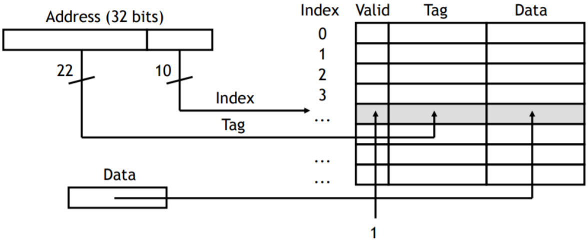
What Happens on a Cache Miss?

- The delays that we have been assuming for memories (e.g., 2ns) are really assuming cache hits.
  - If our CPU implementations accessed main memory directly, their cycle times would have to be much larger.
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time.

- However, a much slower main memory access is needed on a cache miss. The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache).

Loading a Block into the Cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest k bits of the address specify a cache block.
  - The upper (m - k) address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.



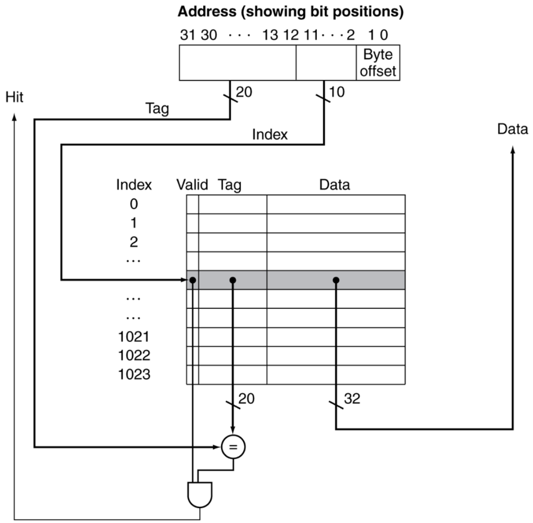
What If the Cache Fills Up?

- Our third question was what to do if we run out of space in our cache, or if we need to reuse a cache line for a different memory address.

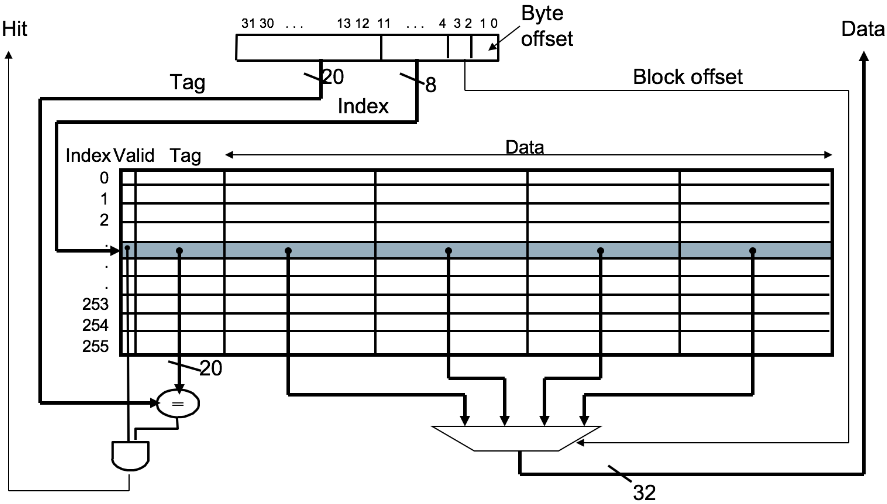
- A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data.

- This is a least recently used replacement policy, which assumes that older data is less likely to be requested than newer data.

Accessing a Cache



Multiword Block Direct-Mapped Cache



Summary

- The recap of a computer memory hierarchy
- General principles of caching
- The principles of spacial and temporal locality
- A simple cache design