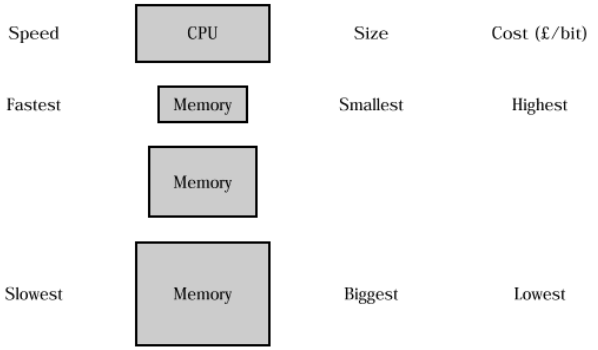
Lecture 10 - Cache Memory

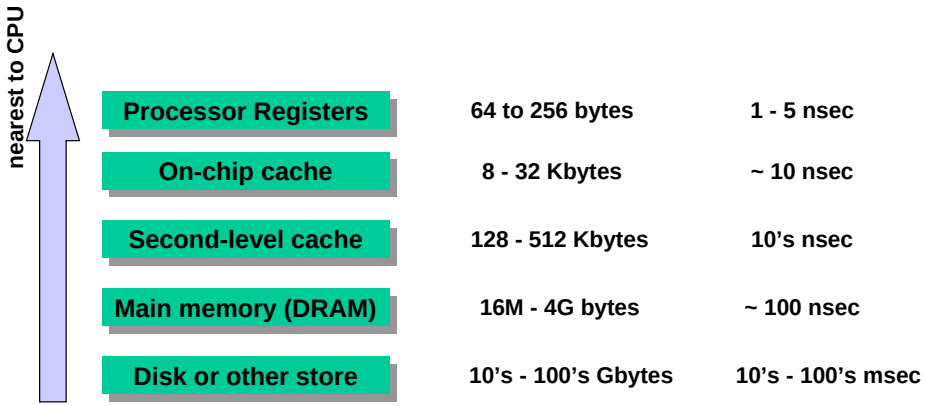
- **SRAM**:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space (4 to 6 transistors per bit) than DRAM
- **DRAM**:
  - value is stored as charge on a capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)
- **Memory** vs **Logic Performance**
  - memory speed improves much slower than logic speed
  - consequently, very fast memory is expensive
  - applications eat up more and more memory (without necessarily providing better functionality)
- **Users want large and fast memories**!
  - SRAM access times are 2 - 25ns at cost of $100 to $250 per Mbyte
  - DRAM access times are 60-120ns at cost of $2 to $5 per Mbyte
  - Disk access times are 10 to 20 million ns at cost < $0.05 to $.10 per Mbyte

Exploiting Memory Hierarchy

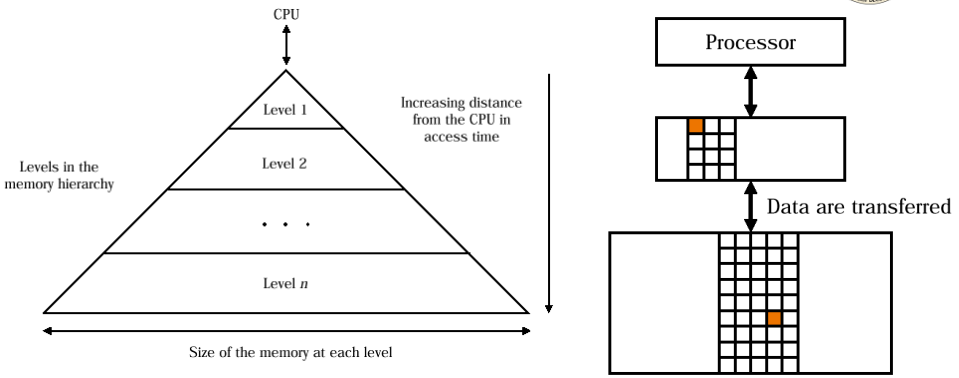
- Question:
  - how to organize memory to improve performance without the cost?
- Answer:
  - build a memory hierarchy



Exploiting Memory Hierarchy



Memory transfer between levels of hierarchy



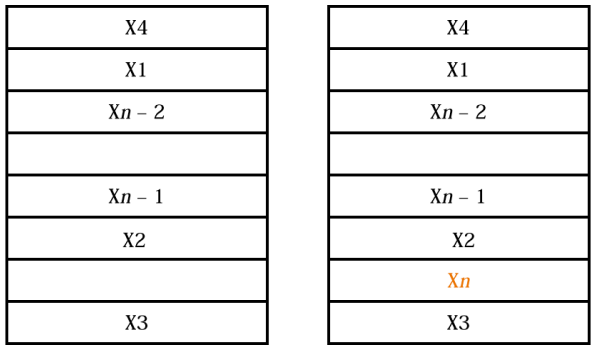
- Every pair of levels in memory of hierarchy can be thought of as having an upper and lower level
- Within each level, the unit of information that is present or not is called a block.
- Usually an entire block is transferred between levels

Principle of Locality

- The principle of locality makes having a memory hierarchy a good idea
- If an item is referenced,
  - **temporal locality**: it will tend to be referenced again soon
  - **spatial locality**: nearby items will tend to be referenced soon.
- **Why does code have locality?**
- Memory hierarchy can be multiple levels
- Data is copied between two adjacent levels at a time
- We will focus on two levels:
  - Upper level (closer to the processor, smaller but faster)
  - Lower level (further from the processor, larger but slower)
- Some terms used in describing memory hierarchy:
  - **block**: minimum unit of data to transfer - also called a **cache line**
  - **hit**: data requested is in the upper level
  - **miss**: data requested is not in the upper level

The Basics of Caches

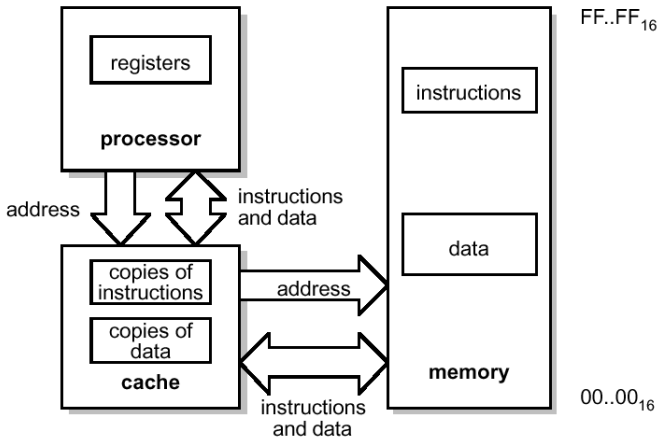
- Assuming that we try to reference a data item **Xn** from cache and it is not there. This reference causes a miss that forces the cache to fetch **Xn** from memory lower in the hierarchy and insert into the cache:



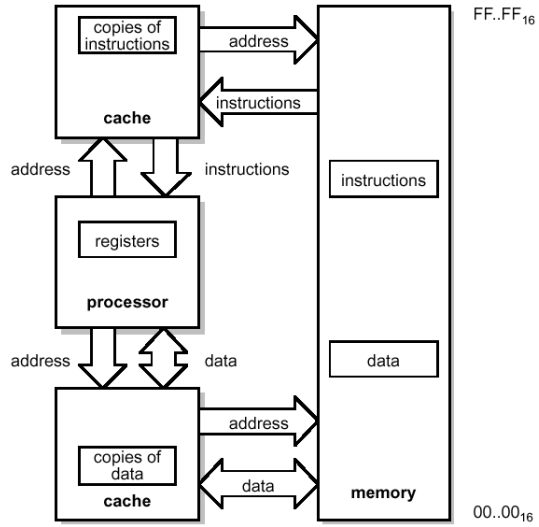
a. Before the reference to Xn b. After the reference to Xn

Unified instruction and data cache

- Single cache shared between instruction and data:



Separate data and instruction caches



The Basics of Caches

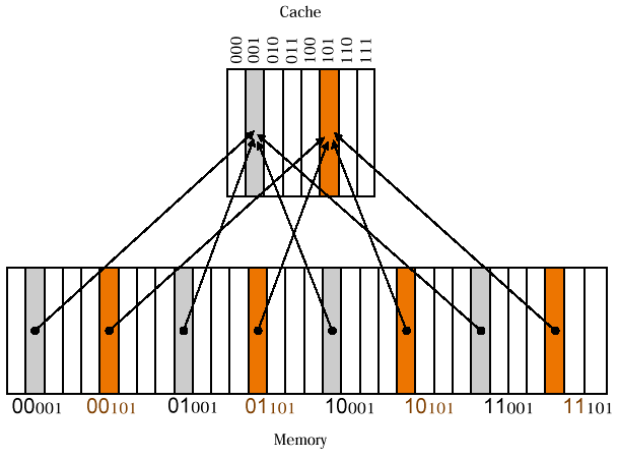
- Two issues:
  - How do we know if a data item is in the cache? (Is it a **hit** or a **miss**?)
  - If it is there (a hit), **how do we find it**?
- Our first example:
  - block size is one word of data
  - we will consider the "**direct mapped**" approach

For each item of data at the lower level,
there is exactly one location in the cache where it might be.

Lots of items at the lower level share locations in the upper level

Direct Mapped Cache

- Mapping: address is modulo the number of blocks in the cache

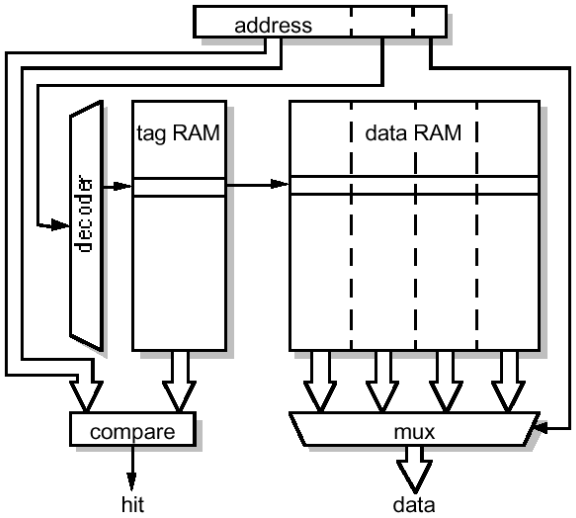


Cache Contents - A walk-through

Index	Valid bit (V)	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

- Initial state on power-ON
- After handling miss at address 10110
- After handling miss at address 11010
- After handling miss at address 10000
- After handling miss at address 00011
- After handling miss at address 10010

Organization of Direct-mapped cache



Direct Mapped Cache

- A particular memory item is stored in a unique location in the cache.
- To check if a particular memory item is in cache, the relevant address bits are used to access the cache entry.
- The top address bits are then compared with the stored tag. If they are equal, we have got a hit.
- Two items with the same cache address field will contend for use of that location.
- Only those bits of the address which are not used to select within the line or to address the cache RAM need be stored in the tag field.
- When a miss occurs, data cannot be read from the cache. A slower read from the next level of memory must take place, incurring a miss penalty.
- A cache line is typically more than one word. It shows 4 words in the diagram here. A large cache line exploits principle of spatial locality - more hit for sequential access. It also incurs higher miss penalty.

Hits vs. Misses

- Read hits
  - this is what we want!
- Read misses
  - stall the CPU, fetch block from memory, deliver to cache, restart
- Write hits:
  - can replace data in cache and memory (write-through)
  - write the data only into the cache (write-back to the cache later)
- Write misses:
  - read the entire block into the cache, then write the word

Write Strategies in Caches

- Cache write is more complicated than cache read because even if you have a hit, you have to decide if and when you write it back to main memory. This is of utmost importance in multiprocessor systems.
- Three strategies are used:
  - 1. Write-through
    - All write are passed to main memory immediately
    - If there is a hit, the cache is updated to hold new value
    - Processor slow down to main memory speed during write
  - 2. Write-through with buffered write
    - Use a buffer to hold data to write back to main memory
    - Processor only slowed down to write buffer speed (which is fast)
    - Write buffer transfers data to main memory (slowly), processor continues its tasks
  - 3. Copy-back
    - Write operation updates the cache, but not main memory
    - Cache remember that it is different from main memory via a **dirty bit**
    - It is **copied back** to main memory only when the cache line is used by new data