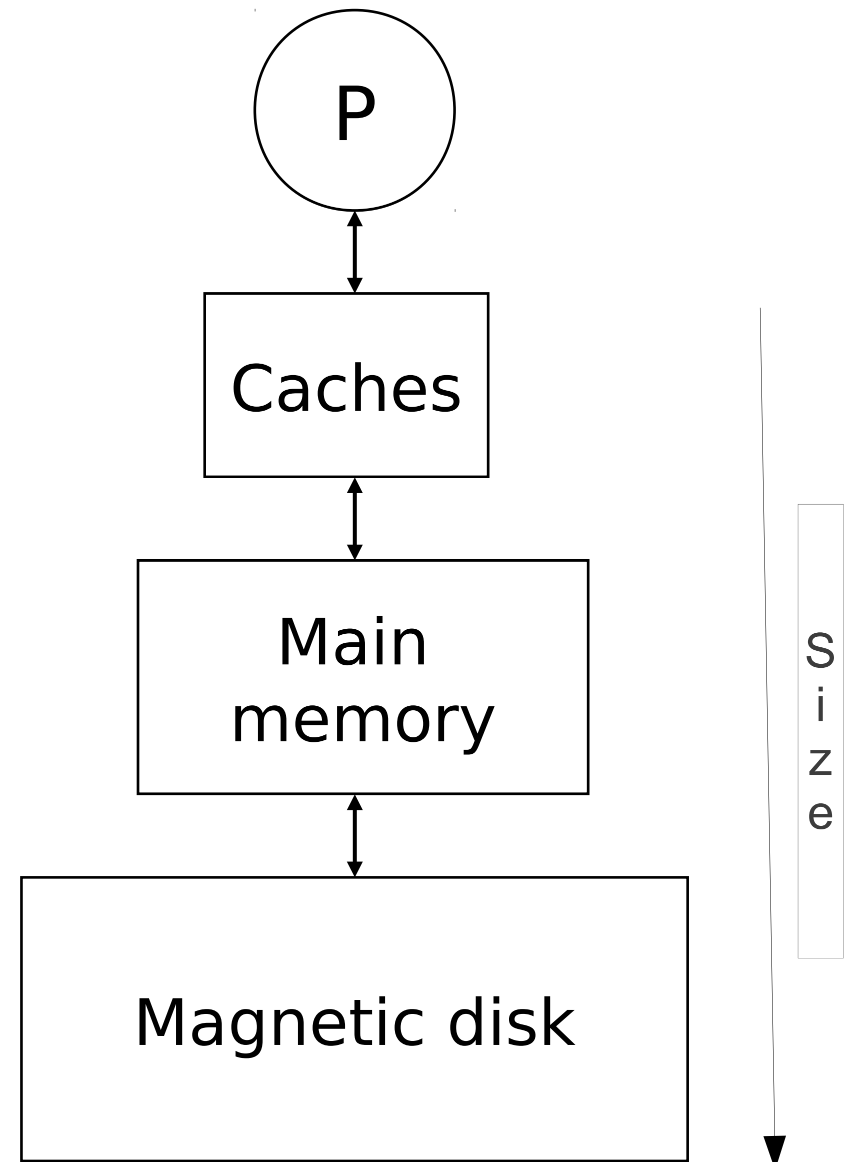
Tutorial 8

Caches

Memory hierarchy

- Consists of multiple levels of memory with different speeds and sizes.
- A level close to the processor is a subset of any level further away.
- Gives the illusion of a memory that is as large as the lowest level, but as fast as the highest level.



© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

Caches

- Memory is often divided in a hierarchy of memories of different sizes/speeds for cost and efficiency reasons.
  - Without fast caches, CPU speed would be limited by the speed of its memory.
  - The faster a memory, the costlier it is. You therefore normally only have small quantities of very small memory.
- A cache corresponds to a small and very fast memory used to store a subset of commonly accessed items.

Common definitions

- A cache is divided into fixed-size blocks, containing multiple words of data.
- The principles of temporal and spatial locality tell us that recently accessed data, and data close to it, are likely to be reused in the near future.
- When you want to access data and it is in the cache, there is a cache hit, otherwise miss.
- A cache can be unified or split, depending on whether it contains both data and/or instructions

Cache Organization

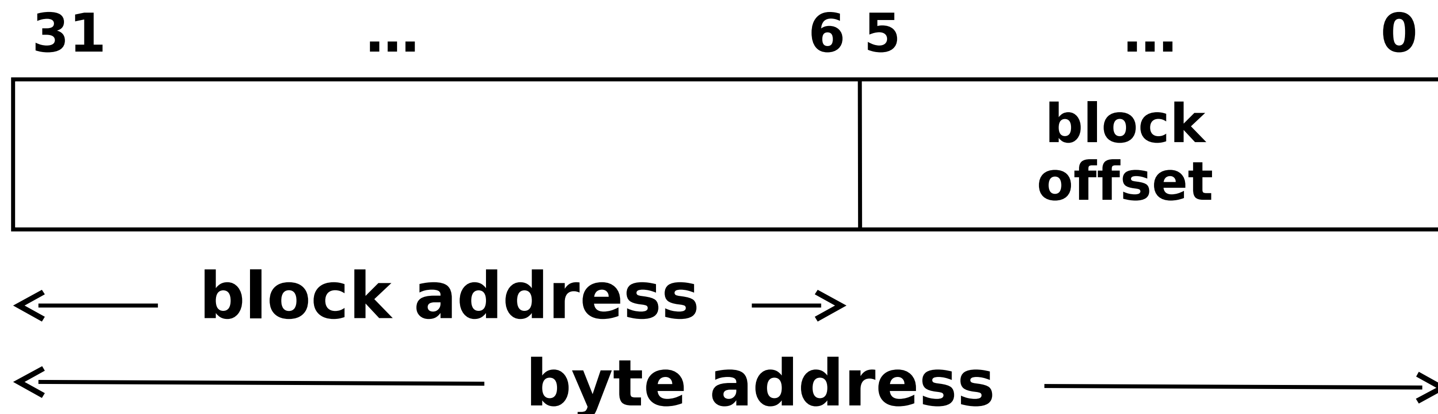
- Data is stored in **blocks**
- Blocks contain multiple words of data
- Each block is selected by an **index**
- **Valid** bit indicates whether data is valid
- **Tag** is the "remaining" part of the address

Index	Valid	Tag	Data
0			
1			
2			
3			
…			

© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

Side note: byte address, block address

- Suppose that memory addresses are 32 bits, and each block of data is 8 words of 8 bytes (64 bytes / block).
- Each block contains 64 (2^6) bytes of data. Since memory is byte-addressable, we need 6 bits as an offset to the block. In theory, we are thus working with 64M (2^26) block addresses.



© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

Side note: byte address, block address

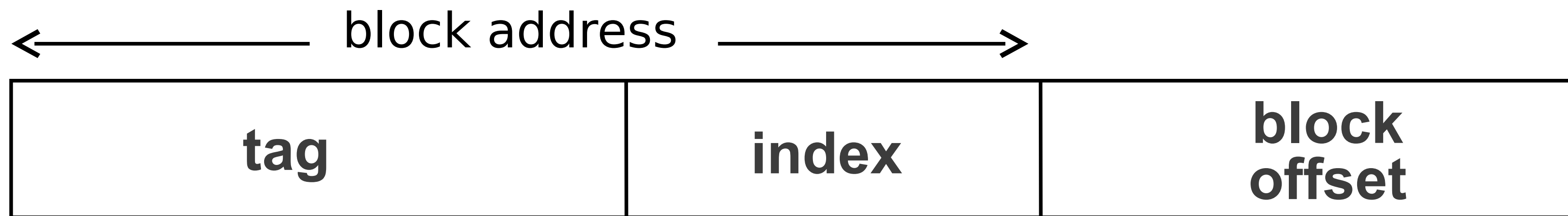
- Since we cannot have 64M blocks in the cache, some locations are re-used by multiple block addresses. When a block of data is loaded from main memory into the cache, its block address is divided into 2 fields:

  - **Index**: the lower bits. Used to determine where to put the data in the cache.
  - **Tag**: the upper bits. This is entered into the tag field of the cache entry.
- Many different block addresses map to the *same* index in the cache, so we need the tag entry to verify which block is currently in the cache.

© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

Side note: byte address, block address

The address issued by the processor is divided into 3 fields



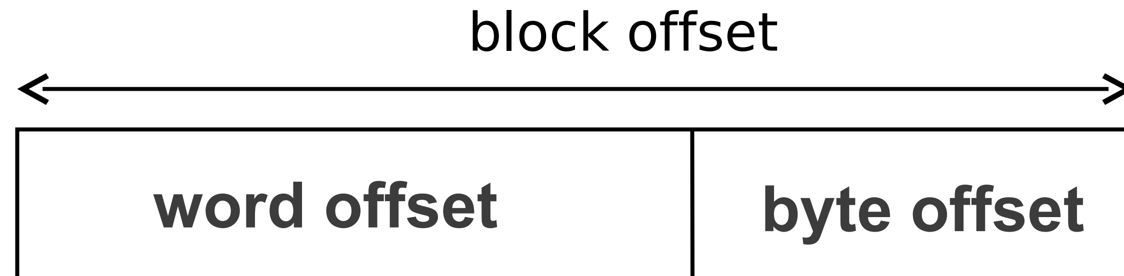
Then:

1. The **index** field is used to select one block from the cache
2. The **tag** is compared with the tag field of the selected block
   - If they match, then this is the data we want = *cache hit*
   - Otherwise, it is a *cache miss* and the block will need to be loaded from main memory
3. The **block offset** selects the requested part of the block, and sends it to the processor

© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

Side note: byte address, block address

Depending on the processor word size, the block offset can be further divided, if a block contains more than one word.



If the CPU can only address 8-byte words, and a block contains 64 bytes (8 words), the byte offset will be 3 bits long (to address the 8 bytes of a word) and the word offset will also be 3 bits long (to address the 8 words of a block). The total is what we expect,6 bits (to address the 64 bytes of the block).

© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

Where can a block be placed?

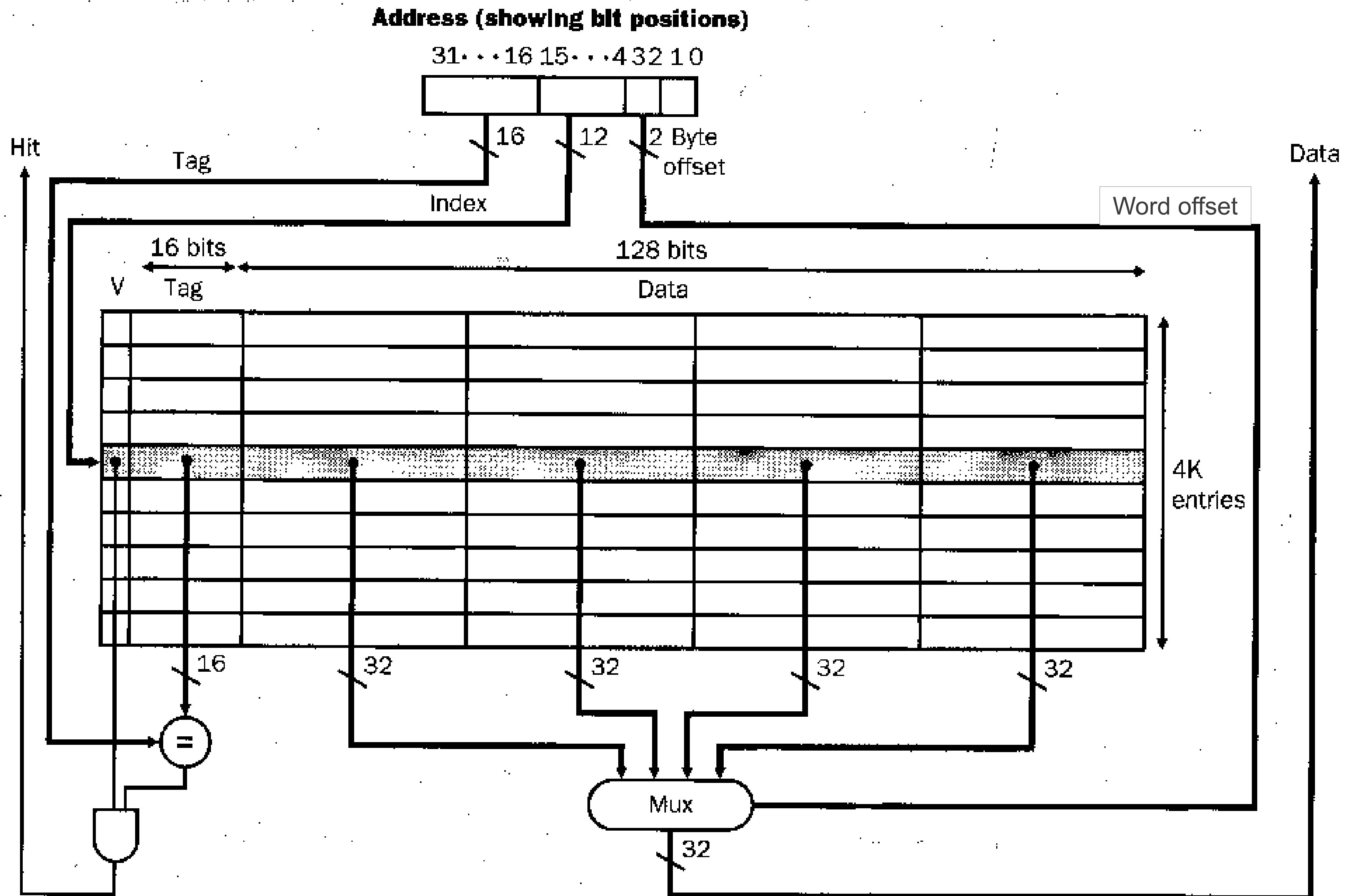
- Direct mapped: a block with a given address can only be placed in a single location in the cache
- Fully associative: a block can be placed anywhere in the cache
- Set associative: a block can be placed anywhere within a set of locations in the cache

Where can a block be placed?

Assuming a cache with $N=2^n$ blocks, and a block with block address A,

- Direct mapped: the index is A mod N (the last n bits tell you the block location)
- Fully associative: The block can go anywhere in the cache
- Set associative: with K sets ($K=2^k$), block A can only go in set A mod K (the last k bits tell you the set nb.)

Simple direct-mapped cache



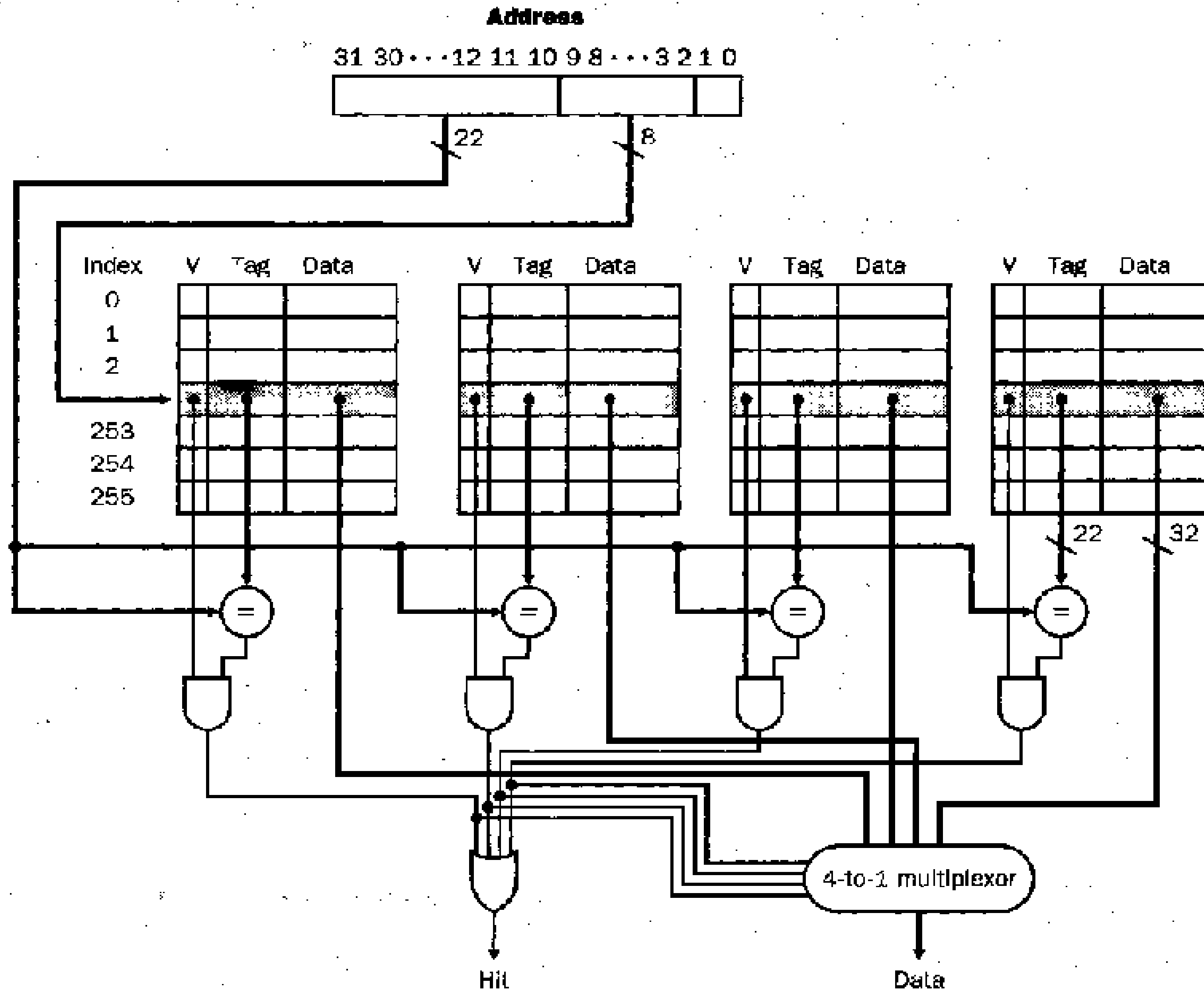
Example (direct-mapped)

- A cache is direct-mapped and has 64 KB data. Each block contains 32 bytes. The address is 32 bits wide. What are the sizes of the tag, index, and block offset fields?

- # bits in block offset = 5 (since each block contains 2^5 bytes)
- # blocks in cache = 64×1024 / 32 = 2048 blocks
  - So # bits in index field = 11 (since there are 2^11 blocks)
- # bits in tag field = 32 - 5 - 11 = 16 (the rest!)

© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

4-way set-associative cache



Example (set-associative)

- A cache is 4-way set-associative and has 64 KB data. Each block contains 32 bytes. The address is 32 bits wide. What are the sizes of the tag, index, and block offset fields?

- # bits in block offset = 5 (since each block contains 2^5 bytes)
- # blocks in cache = 64×1024 / 32 = 2048 (2^11)
- # sets in cache = 2048 / 4 = 512 (2^9) sets (a set is 4 blocks kept in the cache for each index)
  - So # bits in index field = 9
- # bits in tag field = 32 – 5 - 9 = 18

© 2005 Linda Wang, Dept. of Elec. & Comp. Eng., McGill University

What about cache misses?

When you have a n-way associative cache, you can replace a block in the cache with a newer one

- At random: replace a random block in the set
- Least-recently used (LRU): replace the block which was the least recently used
- First in, first out (FIFO): replace the block which was put first into the cache

Why would you choose one over the others?

- Efficiency vs ease of implementation

What do you do when you write?

When you do a write, you have to ask yourself if you want the written data to live only in the cache, or to be written all the way to main memory.

- Write through: write your change in the cache and in main memory (ensures consistency)
- Write back: write your change only to the cache (and it will be reflected in main memory when the block is replaced in the cache)

Why choose one over the other? Speed vs implementation complexity vs data consistency

- Write back sometimes uses a dirty bit to avoid writing back a block to main memory if it hasn't been modified

What do you do when you write?

When you write to the cache, you can have a write miss. Two options here...

- Write allocate: when there is a write miss, the block is loaded in the cache and the write is re-attempted
- No-write allocate: when there is a write miss, only main memory is changed (block not loaded in the cache)