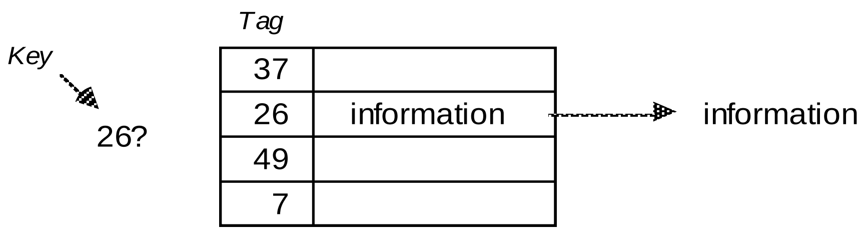
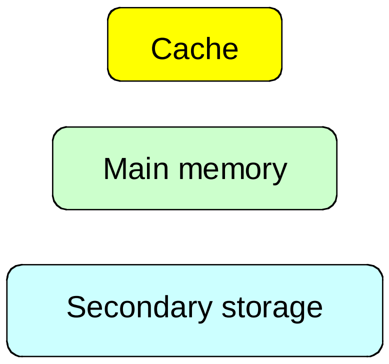
Cache memories

[§6.1] A *cache* is a small, fast memory which is *transparent* to the processor.

- The cache duplicates information that is in main memory.
- With each data block in the cache, there is associated an *identifier* or *tag.* This allows the cache to be *content addressable*.



- Caches are smaller and faster than main memory.
- *Secondary storage*, on the other hand, is larger and slower.



- A *cache miss* is the term analogous to a page fault. It occurs when a referenced word is not in the cache.
  - Cache misses must be handled much more quickly than page faults. Thus, they are handled in hardware.
- Caches can be *organized* according to four different strategies:
  - Direct
  - Fully associative
  - Set associative
  - Sectored

- A cache implements several different *policies* for retrieving and storing information, one in each of the following categories:
    - *Placement policy*—determines where a block is placed when it is brought into the cache.
    - *Replacement policy*—determines what information is purged when space is needed for a new entry.
    - *Write policy*—determines how soon information in the cache is written to lower levels in the memory hierarchy.

Cache memory organization

[§6.2] Information is moved into and out of the cache in *blocks.* When a block is in the cache, it occupies a cache *line*. Blocks are usually larger than one byte,

- to take advantage of locality in programs, and
- because memory may be organized so that it can overlap transfers of several bytes at a time.

The block size is the same as the line size of the cache.

A *placement policy* determines where a particular block can be placed when it goes into the cache. E.g., is a block of memory eligible to be placed in any line in the cache, or is it restricted to a single line?

In our examples, we assume—

- The cache contains 2048 bytes,
  with 16 bytes per line
  Thus it has lines.

- Main memory is made up of 256K bytes, or 16384 blocks.
  Thus an address consists of

We want to structure the cache to achieve a high *hit ratio.*

- *Hit*—the referenced information is in the cache.
- *Miss*—referenced information is not in cache, must be read in from main memory.

$$\text{Hit ratio} \equiv \frac{\text{Number of hits}}{\text{Total number of references}}$$

We will study caches that have three different placement policies (direct, fully associative, set associative).

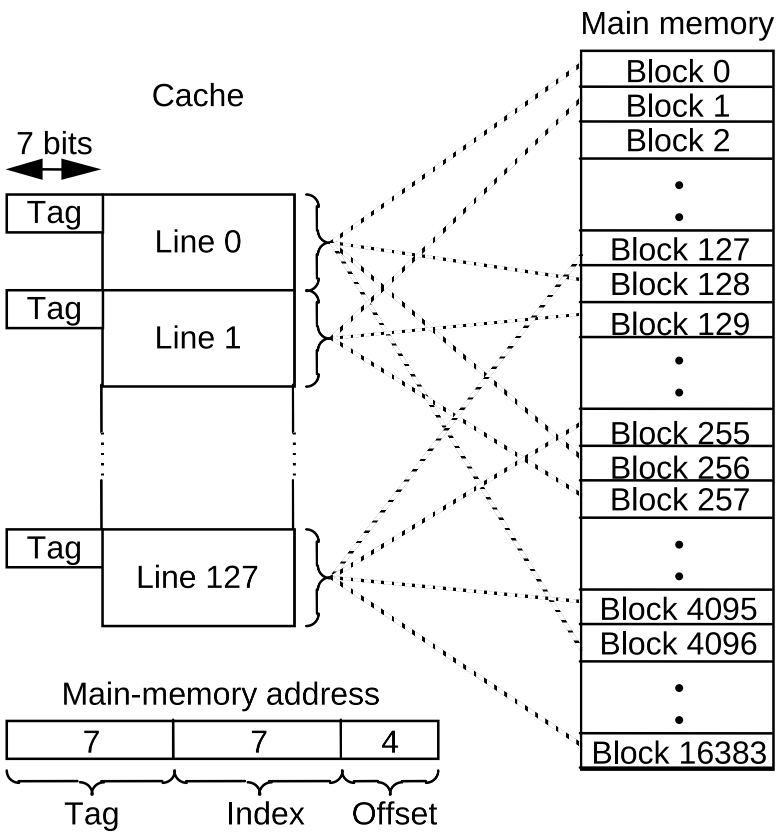
Direct

Only 1 choice of where to place a block.

$$\text{block } i \rightarrow \text{ line } i \bmod 128$$

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order seven bits of the main-memory address of the block.



To search for a word in the cache,

1. Determine what line to look in (easy; just select bits 10–4 of the address).

2. Compare the leading seven bits (bits 17–11) of the address with the tag of the line. If it matches, the block is in the cache.

3. Select the desired bytes from the line.

Advantages:

Fast lookup (only one comparison needed).

Cheap hardware (only one tag needs to be checked).

Easy to decide where to place a block

Disadvantage: Contention for cache lines.

Exercise: What would the size of the tag, index, and offset fields be if—

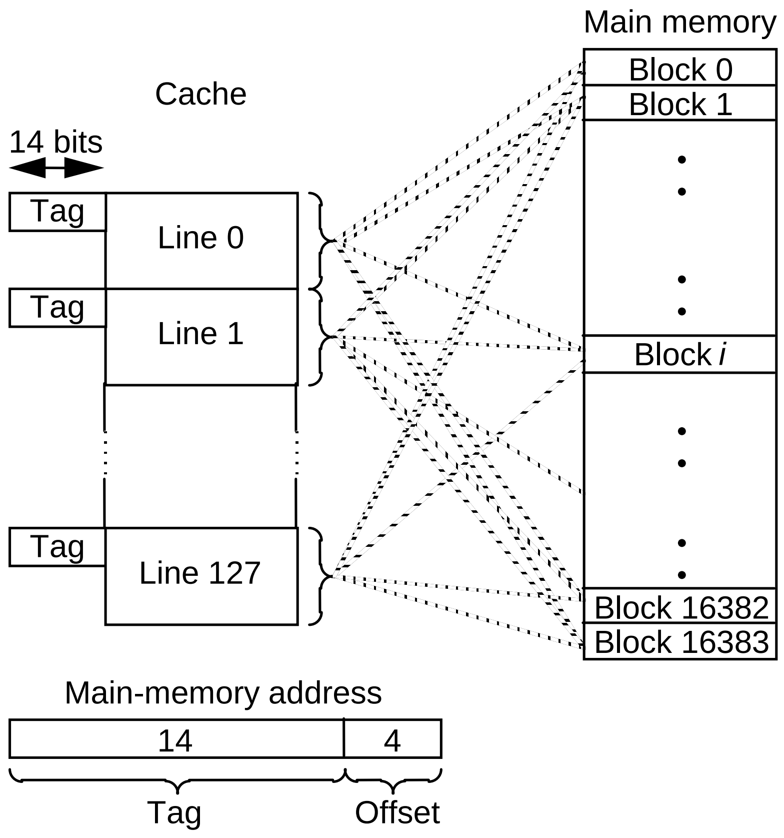
- the line size from our example were doubled, without changing the size of the cache?
- the cache size from our example were doubled, without changing the size of the line?
- an address were 32 bits long, but the cache size and line size were the same as in the example?

Fully associative

Any block can be placed in *any* line in the cache.

This means that we have 128 choices of where to place a block.

block *i* $\rightarrow$ any free (or purgeable) cache location



Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *fourteen* bits of the main-memory address of the block.

To search for a word in the cache,

1. Simultaneously compare the leading 14 bits (bits 17–4) of the address with the tag of all lines. If it matches any one, the block is in the cache.
2. Select the desired bytes from the line.

Advantages:

Minimal contention for lines.

Wide variety of replacement algorithms feasible.

Exercise: What would the size of the tag and offset fields be if—

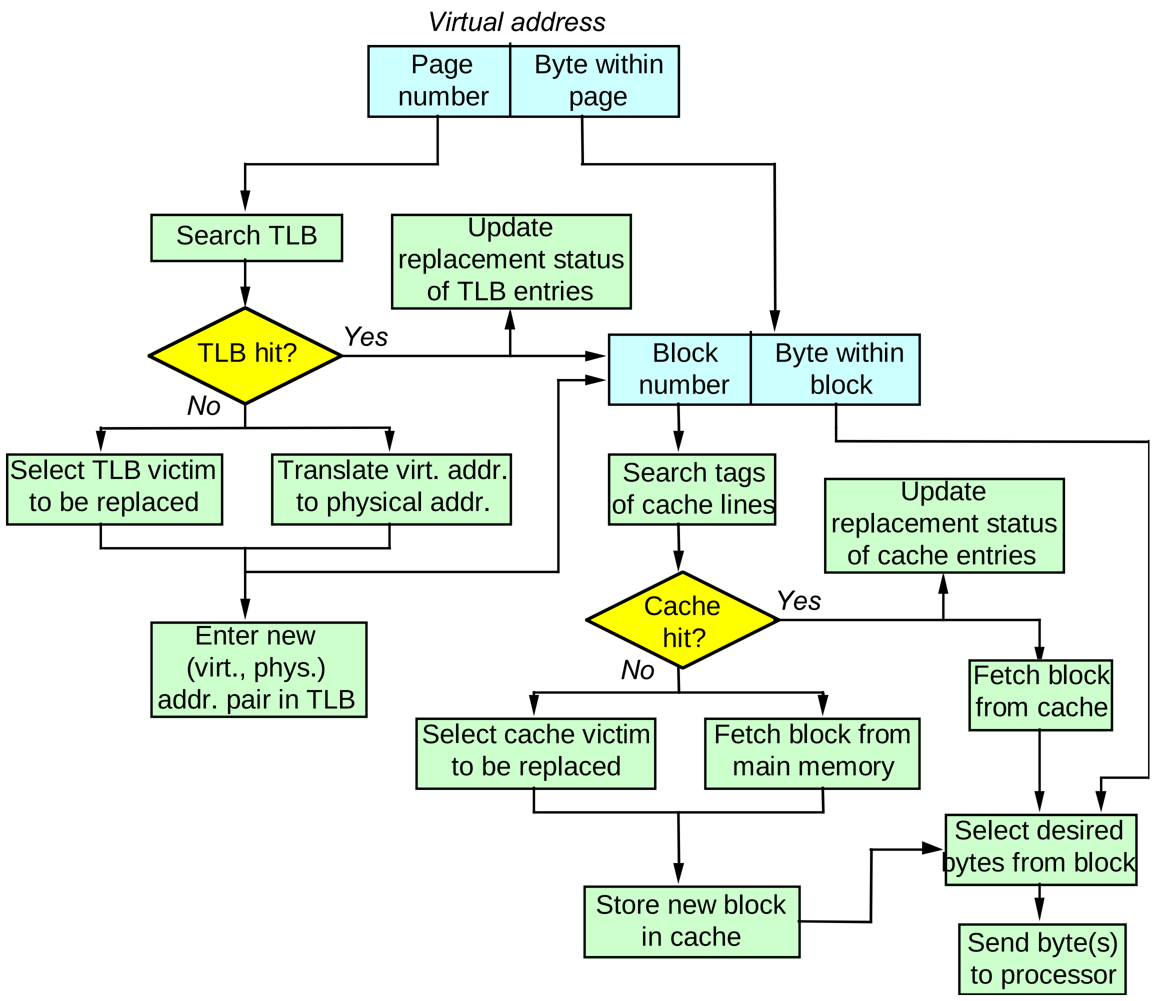
- the line size from our example were doubled, without changing the size of the cache?

- the cache size from our example were doubled, without changing the size of the line?
- an address were 32 bits long, but the cache size and line size were the same as in the example?

Disadvantage:

The most expensive of all organizations, due to the high cost of associative-comparison hardware.

A flowchart of cache operation: The process of searching a fully associative cache is very similar to using a directly mapped cache. Let us consider them in detail.


Virtual address
Page number
Byte within page
Search TLB
Update replacement status of TLB entries
TLB hit?
Yes
No
Block number
Byte within block
Select TLB victim to be replaced
Translate virt. addr. to physical addr.
Search tags of cache lines
Update replacement status of cache entries
Enter new (virt., phys.) addr. pair in TLB
Cache hit?
Yes
No
Fetch block from cache
Select cache victim to be replaced
Fetch block from main memory
Select desired bytes from block
Store new block in cache
Send byte(s) to processor

Note that this diagram assumes a *separate* TLB.

Which steps would be different if the cache were directly mapped?

Set associative

1 < *n* < 128 choices of where to place a block.

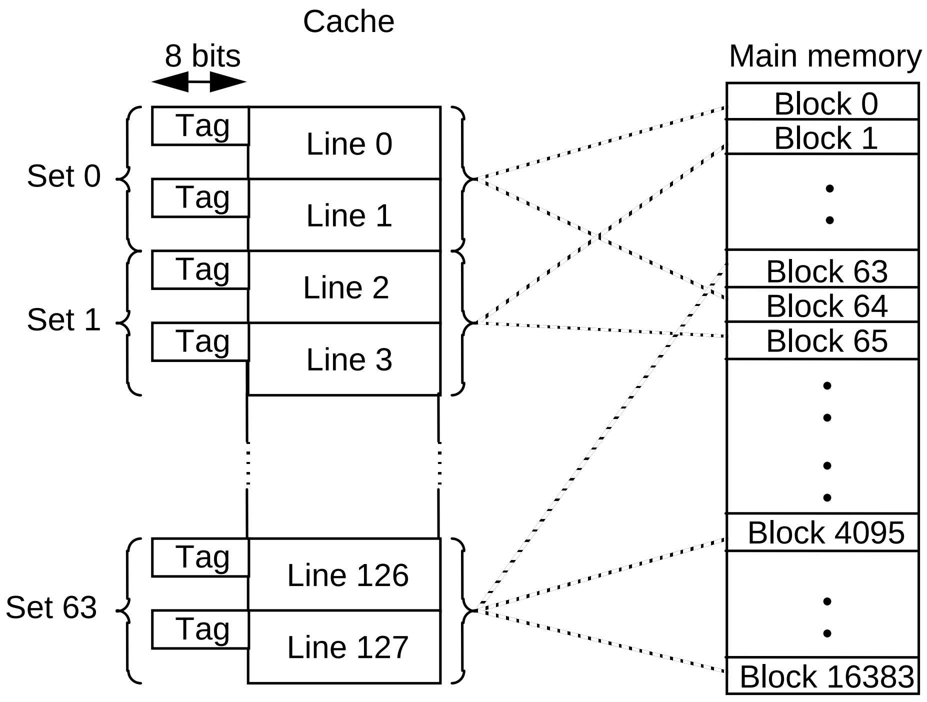
A compromise between direct and fully associative strategies.

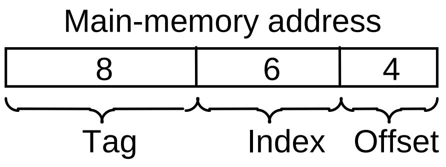
The cache is divided into *s* sets, where *s* is a power of 2.

$$\text{block } i \rightarrow \text{ any line in set } i \bmod s$$

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *eight* bits of the main-memory address of the block. (The next six bits can be derived from the set number.)





Exercise: What would the size of the tag, index, and offset fields be if—

- the line size from our example were doubled, without changing the size of the cache?
- the set size from our example were doubled, without changing the size of a line or the cache?
- the cache size from our example were doubled, without changing the size of the line or a set?
- an address were 32 bits long, but the cache size and line size was the same as in the example?

To search for a word in the cache,

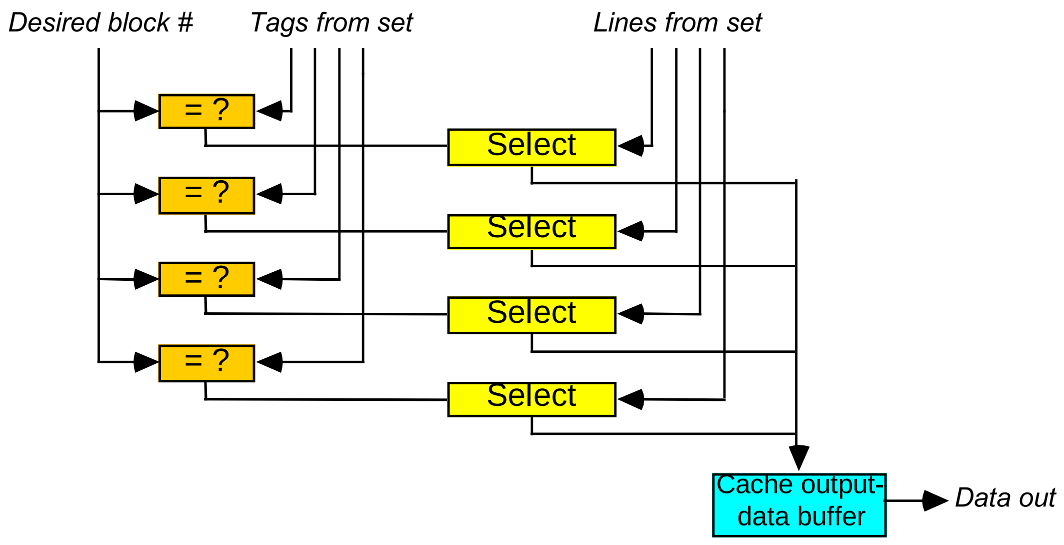
1. Select the proper set (*i* mod *s*).

2. Simultaneously compare the leading 8 bits (bits 17–10) of the address with the tag of all lines in the set. If it matches any one, the block is in the cache.

   At the same time, the (first bytes of) the lines are also being read out so they will be accessible at the end of the cycle.

3. If a match is found, gate the data from the proper block to the cache-output buffer.

4. Select the desired bytes from the line.



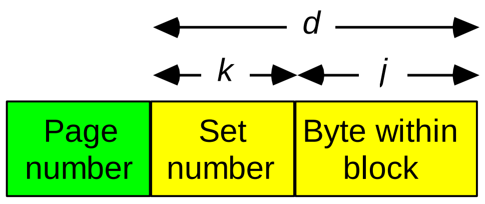
- All reads from the cache occur as early as possible, to allow maximum time for the comparison to take place.
- Which line to use is decided late, after the data have reached high-speed registers, so the processor can receive the data fast.

[§6.2.6] To attain maximum speed in accessing data, we would like to start searching the cache *at the same time* we are looking up the page number in the TLB.

When the bit-selection method is used, both can be done at once if the page number *is disjoint from* the set number.

This means that

- the number of bits k in the set number
- + the number of bits j which determine the byte within a line
- must be ≤ the number of bits d in the displacement field.



We want $k + j \leq d$.

(If the page size is 2^d, then there will be d bits in the displacement field.)

Factors influencing line lengths:

- Long lines ⇒ higher hit ratios.
- Long lines ⇒ less memory devoted to tags.
- Long lines ⇒ longer memory transactions (undesirable in a multiprocessor).
- Long lines ⇒ more write-backs (explained below).

For most machines, line sizes between 32 and 128 bytes perform best.

If there are *b* lines per set, the cache is said to be *b-way* set associative. How many way associative was the example above?

The logic to compare 2, 4, or 8 tags simultaneously can be made quite fast.

But as *b* increases beyond that, cycle time starts to climb, and the higher cycle time begins to offset the increased associativity.

Almost all L1 caches are less than 8-way set-associative. L2 caches often have higher associativity.

Two-level caches

Write policy

[§6.2.3] Answer these questions, based on the text.

What are the two write policies mentioned in the text?

Which one is typically used when a block is to be written to main memory, and why?

Which one can be used when a block is to be written to a lower level of the cache, and why?

Can you explain what error correction has to do with the choice of write policy?

Explain what a parity bit has to do with this.

Principle of inclusion

[§6.2.4] To analyze a second-level cache, we use the *principle of inclusion*—a large second-level cache includes everything in the first-level cache.

We can then do the analysis by assuming the first-level cache did not exist, and measuring the hit ratio of the second-level cache alone.

How should the line length in the second-level cache relate to the line length in the first-level cache?

When we measure a two-level cache system, two miss ratios are of interest:

- The *local miss rate* for a cache is the

  $$\frac{\text{\# misses experienced by the cache}}{\text{number of incoming references}}$$

  To compute this ratio for the L2 cache, we need to know the number of misses in

- The *global miss rate* of the cache is

$$\frac{\text{\# L2 misses}}{\text{\# of references made by processor}}$$

  This is the primary measure of the L2 cache.

What conditions need to be satisfied in order for inclusion to hold?

- L2 associativity must be ≥ L1 associativity, irrespective of the number of sets.

  Otherwise, more entries in a particular set could fit into the L1 cache than the L2 cache, which means the L2 cache couldn't hold everything in the L1 cache.

- The number of L2 sets has to be ≥ the number of L1 sets, irrespective of L2 associativity.

  (Assume that the L2 line size is ≥ L1 line size.)

  If this were not true, multiple L1 sets would depend on a single L2 set for backing store. So references to one L1 set could affect the backing store for another L1 set.

- All reference information from L1 is passed to L2 so that it can update its replacement bits.

Even if all of these conditions hold, we still won't have logical inclusion if L1 is write-back. (However, we will still have *statistical inclusion*—L2 *usually* contains L1 data.)