Memory hierarchy

Since the early days, memory has been a hierarchy of components of various speeds and capacities.

Hierarchy has been driven by cost and performance.

In the early days, distinction between primary memory (core) and secondary memory (disks)

Nowadays, there is a hierarchy within the primary memory with:

- One or more levels of cache (fast SRAM, costly)
- Maybe another cache level (DRAM, less costly)
- Vast amount of main memory (DRAM, a little slower and less costly)

Goal of a memory hierarchy

Keep close to the ALU the information that will be needed now and in the very near future (memory closest to ALU is designed to be the fastest).

Keep close to the ALU, only the information that is required now and in the near future (can't keep everything because of cost)

Trend that makes things more difficult: processor speed increases and SRAM access time decreases much faster than DRAM access time decreases.

Technology	Typical access time	$per Mbyte in 1997
SRAM	4-20 ns.	about $100-$250
DRAM	40-120 ns	about $5 -$10
Disk	10^7 ns	about $0.1-$0.2

Locality

A memory hierarchy concept works because computer programs exhibit the *principle of locality*

- Temporal locality: data (code) used in the past is likely to be reused again in the future (e.g., code of loops, data in stacks)
- Spatial locality: data (code) close to the data that you are presently referencing will be referenced in the near future (e.g., traversing an array)

Caches

Registers are not sufficient to keep enough locality close to the ALU.

Main memory (DRAM) is "too far"; it takes many cycles to access it. Would completely destroy the pipeline.

Hence need for a fast memory between main memory and registers, namely a *cache*.

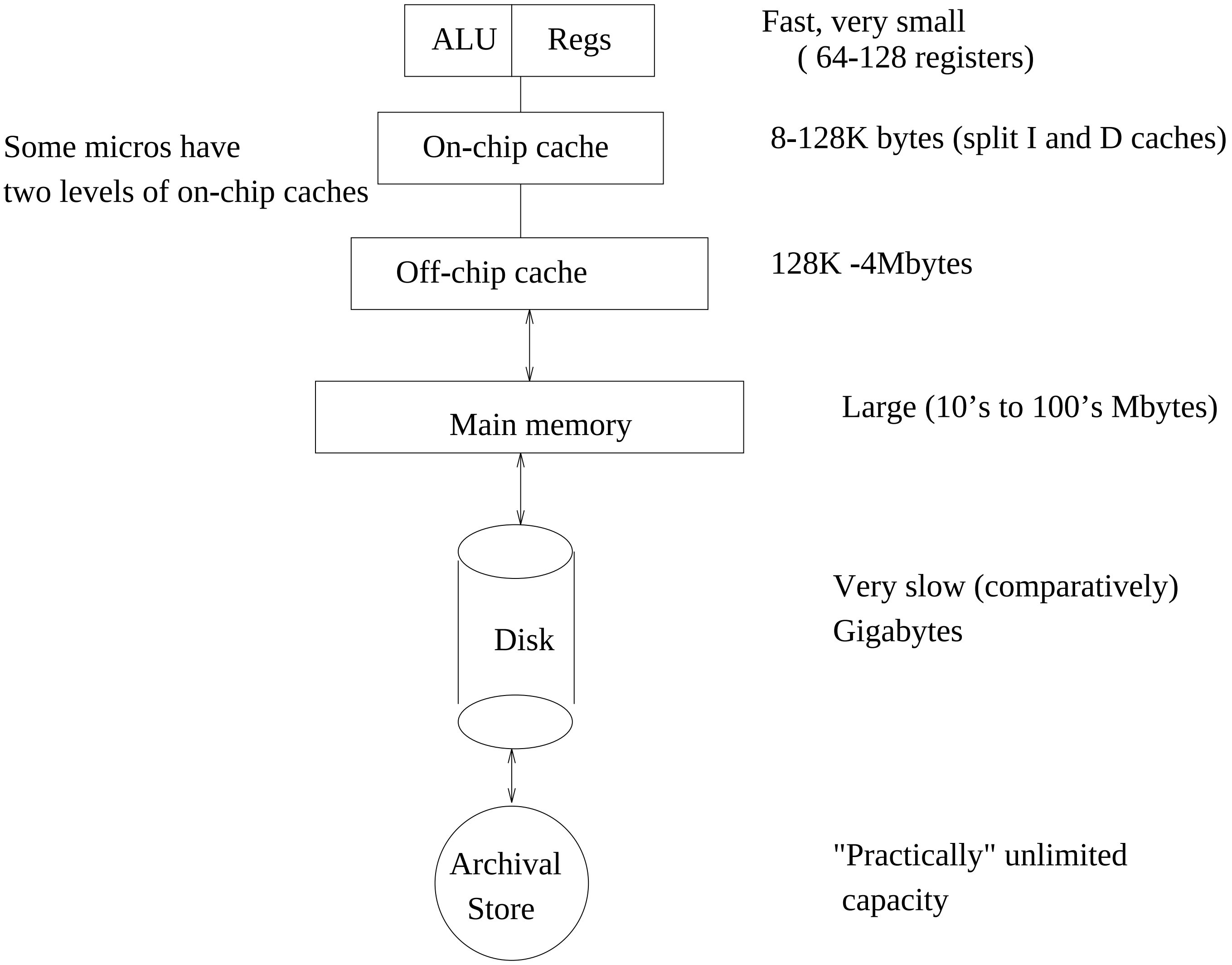
Keep in the cache what's most likely to be referenced in the near future.

When fetching an instruction, first check to see if it is in the cache before going to main memory.

When performing a load, first check to see if it is in the cache before going to main memory.

When performing a store, first write it in the cache before going to main memory (there are variations here).

Levels in the memory hierarchy



Caches

Not a new idea. First cache in late 60's in IBM 360/85.

Concept used in many other aspects of computer systems (hardware and software)

Works because programs exhibit locality.

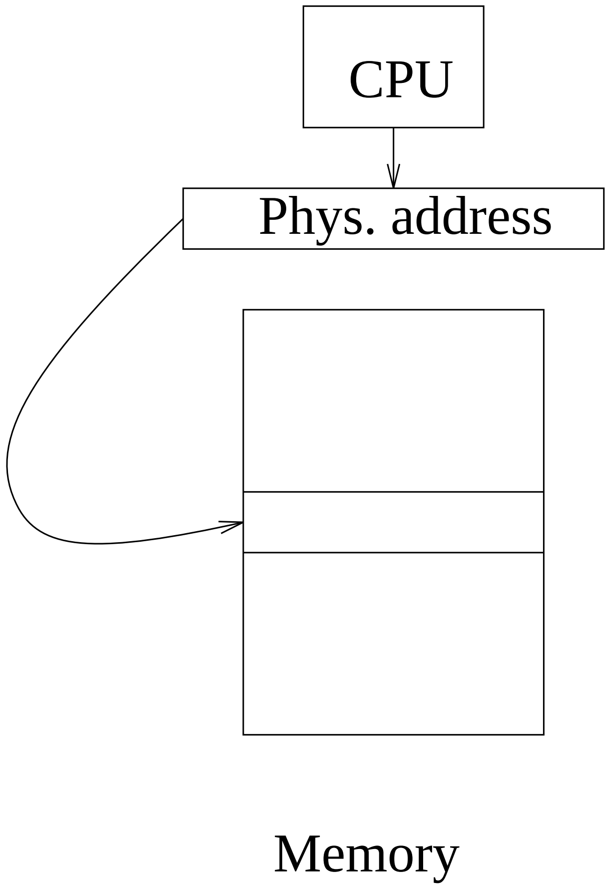
Lots of research on caches in the last 20 years because of the difference in processor speed and memory access time.

Every current microprocessor system has a cache hierarchy, with at least two levels of caches, one on-chip and one off-chip.

Main memory access (review)

In a Load (or Store) instruction, the address is an index in the memory array.

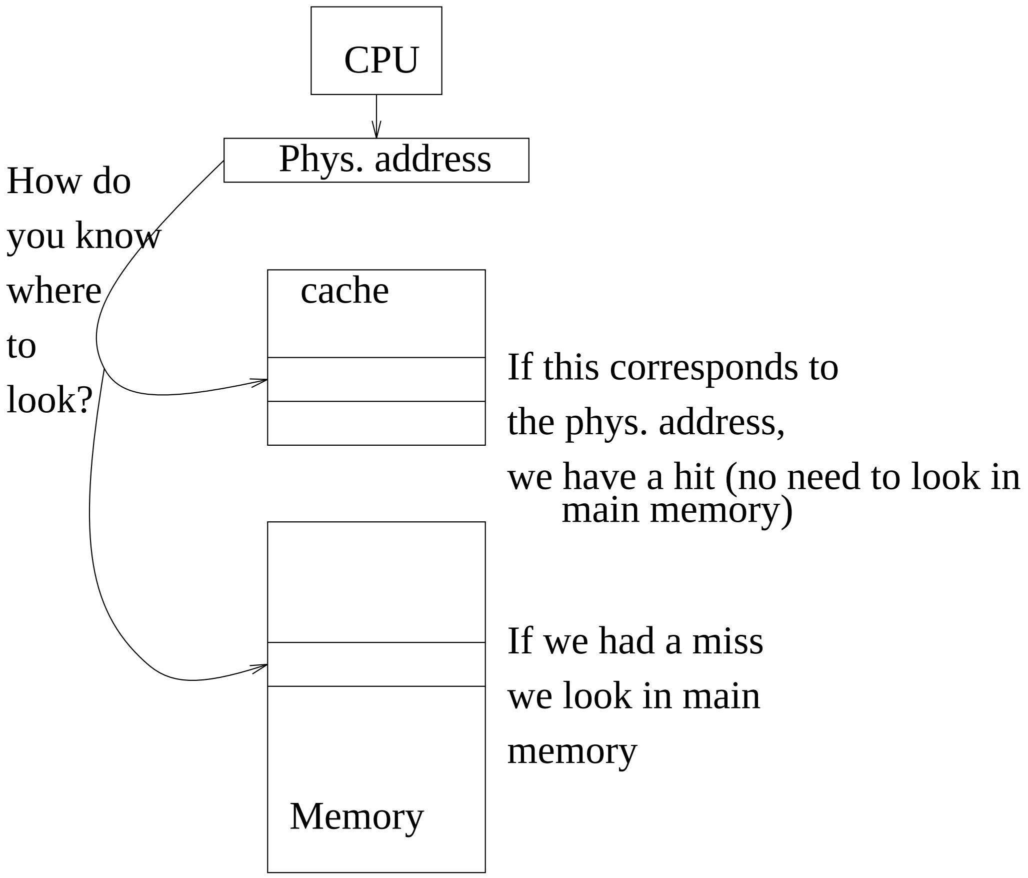
Each byte (word) of memory has a unique address, or, in other words, the mapping between memory address and memory location is unique.



Cache access

Cache is much smaller than memory. Not all memory locations have a corresponding entry in the cache at a given time.

When the CPU generates an address, it first looks up if the corresponding memory location is mapped in the cache. If so we have a *cache hit*. Otherwise we have a *cache miss*.



Some basic questions on cache design and some "top level" answers

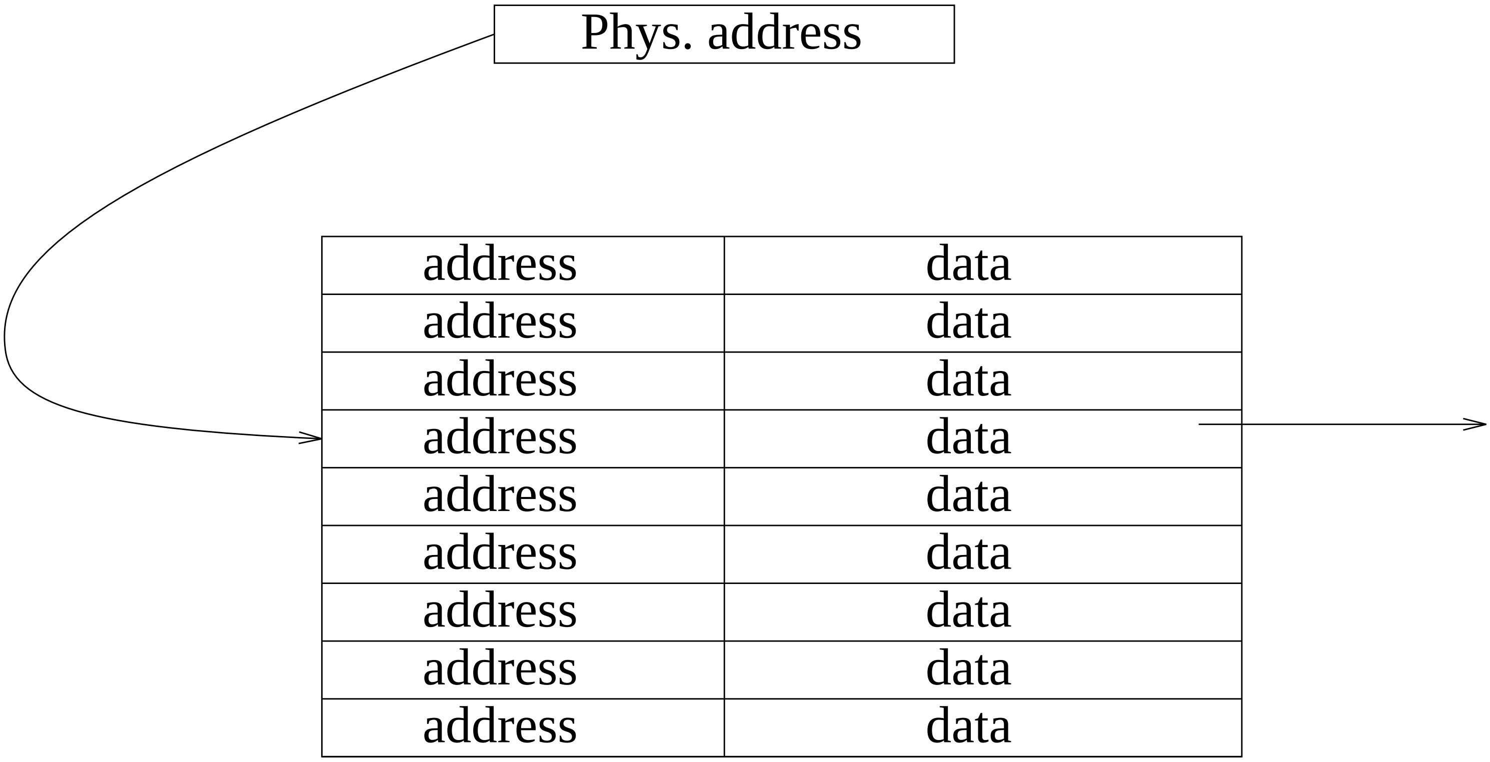
Questions:

1. When do we bring the contents of a memory location in the cache?
2. Where do we put it?
3. How do we know it's there?
4. What happens if the cache is full and we want to bring in something new?

Answers:

1. The first time there is a cache miss for that location
2. Depends on "cache organization" (see next slides)
3. Each entry in the cache carries its own "name" (or *tag*)
4. One entry will be kicked out (replaced) to make room for the new one.

Generic cache organization



if phys. address =
address (or tag)
we have a hit

data is good if
we have a hit

Cache organizations

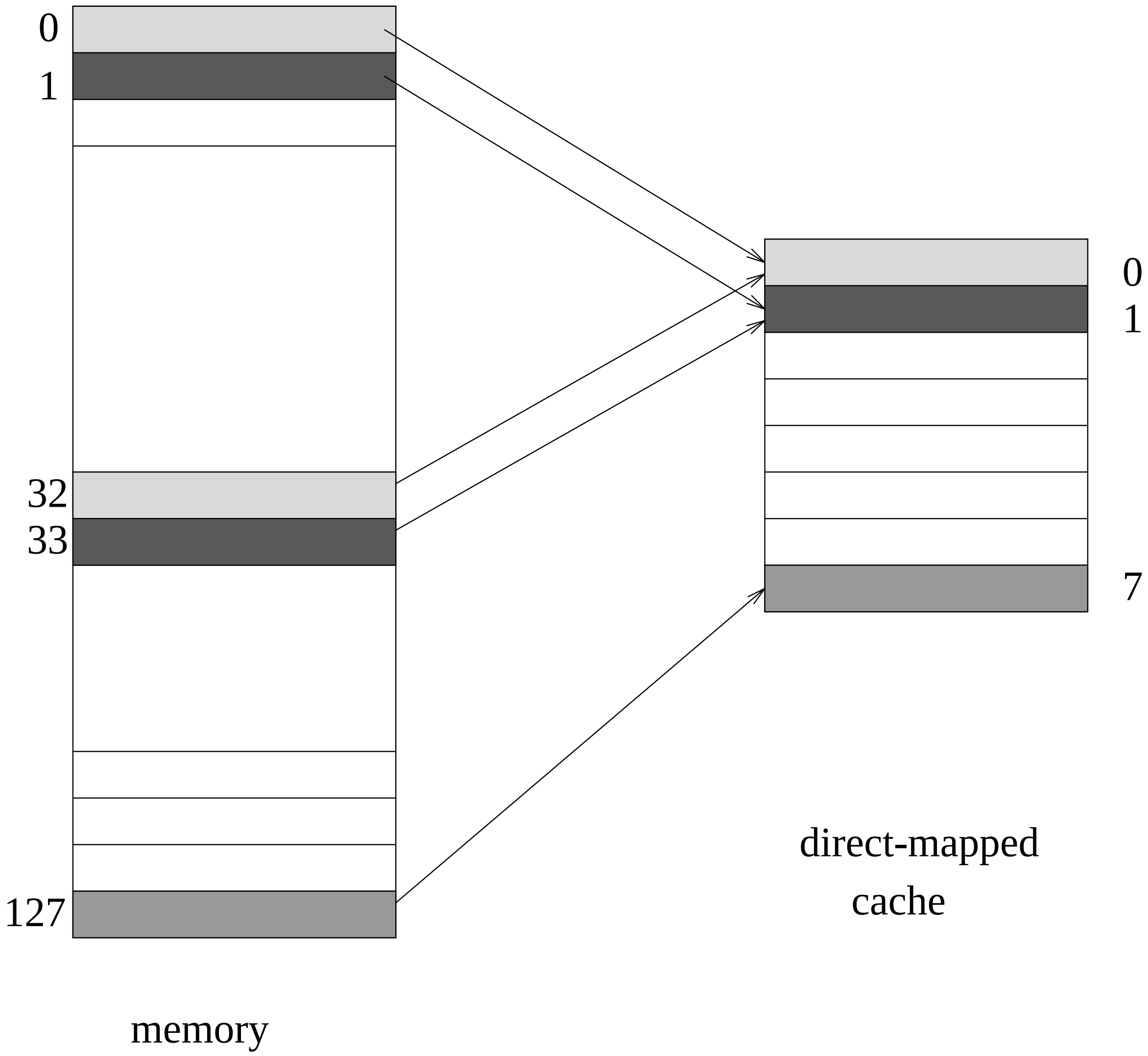
The mapping of a memory location to an entry in the cache ranges from full generality to very restrictive.

- If a memory location can be mapped anywhere in the cache (most general mapping) we have a *fully associative* cache
- If a memory location can be mapped at only one location in the cache (most restrictive mapping), we have a *direct-mapped* cache
- If a memory location can be mapped at one location among a set of locations, we have a *set-associative* cache

In order to see if a memory reference is a hit or a miss, the requirements are:

- For a fully associative cache: check all tag (address) fields in parallel for a match (very expensive)
- For a direct-mapped cache: check the tag (address) field of only one entry
- For a set-associative cache: check the tag (address) field of a few entries

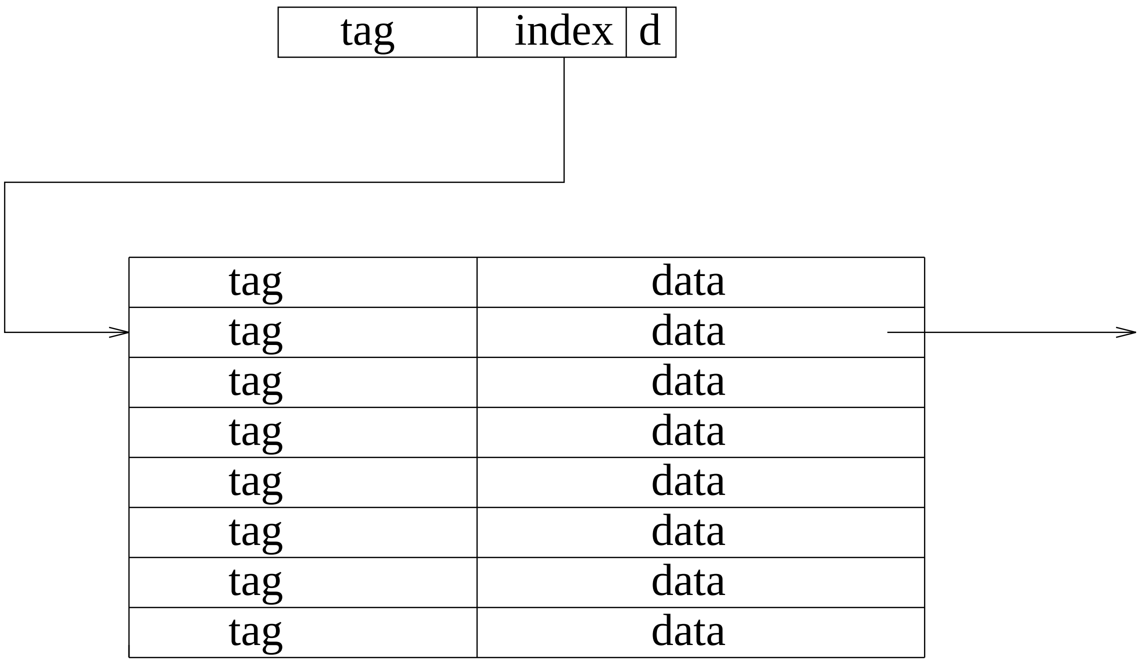
Direct-mapped Cache



Physical addressing fields (direct-mapped cache)

index = log(number of entries in the cache)

d = log(number of bytes in data entry)



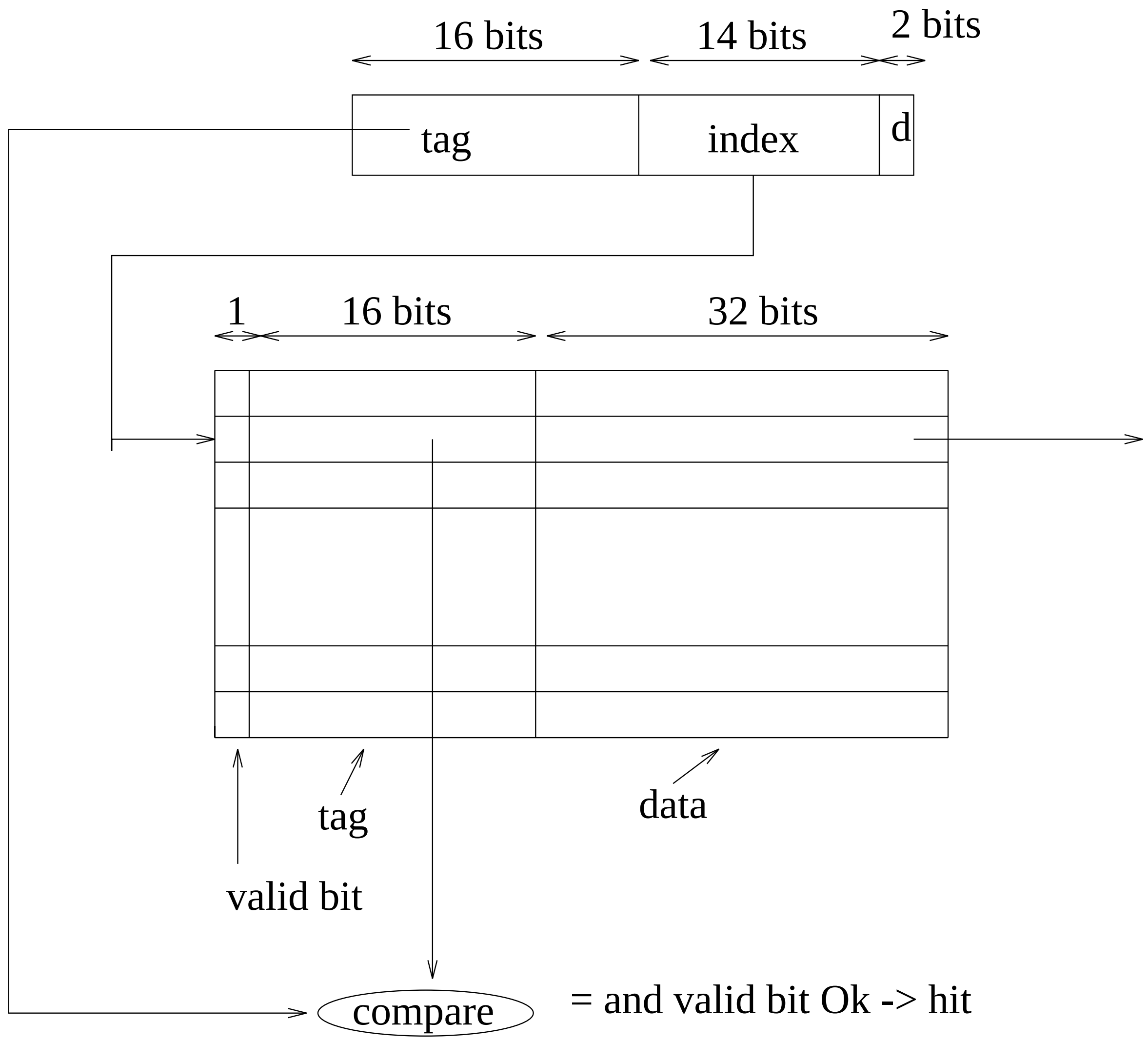
if Tag (phys.address) =
tag of cache(index) then
we have a hit

data is good if
we have a hit

Example of a direct-mapped cache.
DEC Station 3100

Direct-mapped. 16K blocks (lines, entries) of 1 word (4 bytes) each.

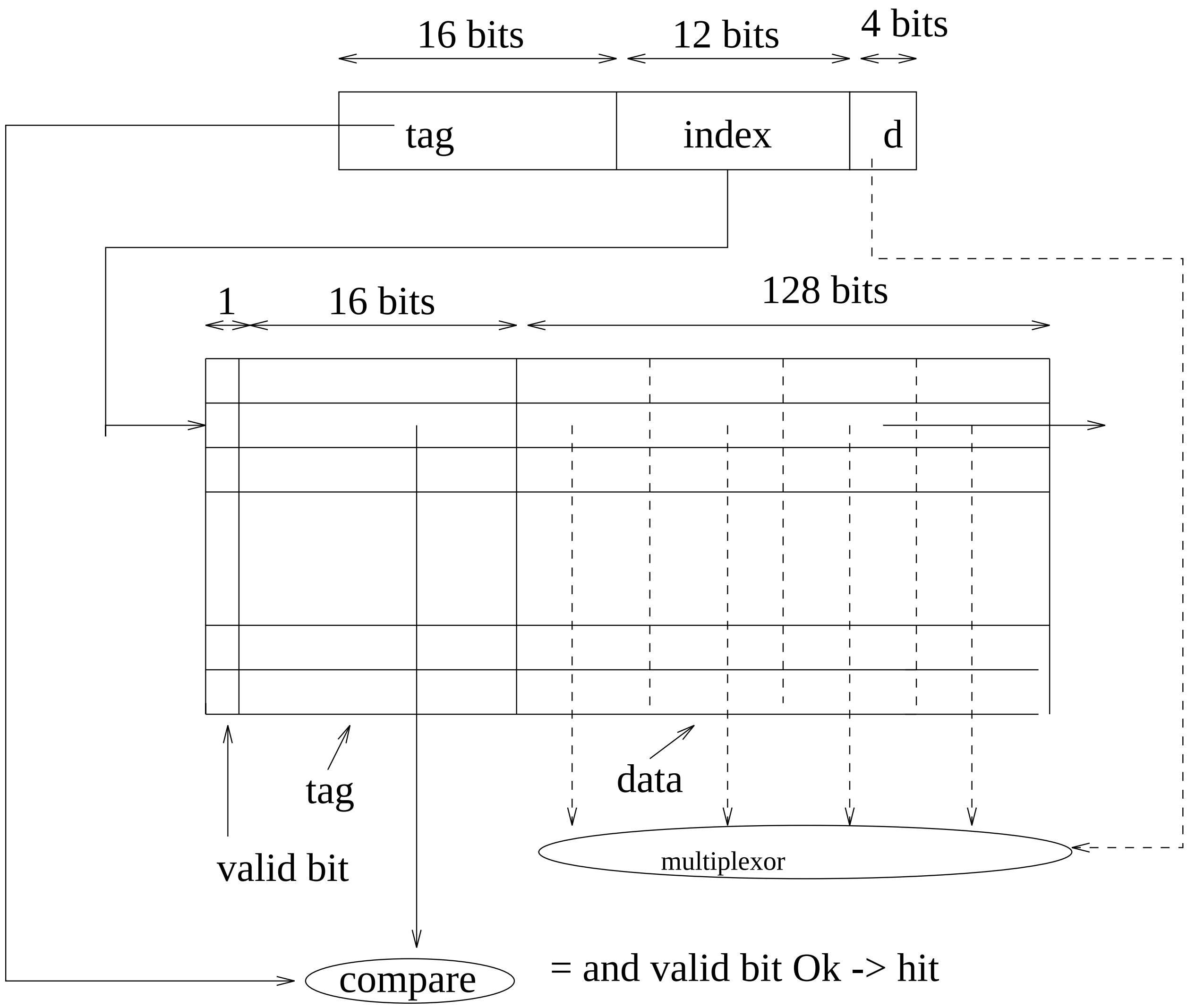
Total capacity: $16 \times 4 = 64$Kbytes



Example of a direct-mapped cache with longer blocks

Direct-mapped. 4K blocks (lines, entries) of 4 words (16 bytes) each.

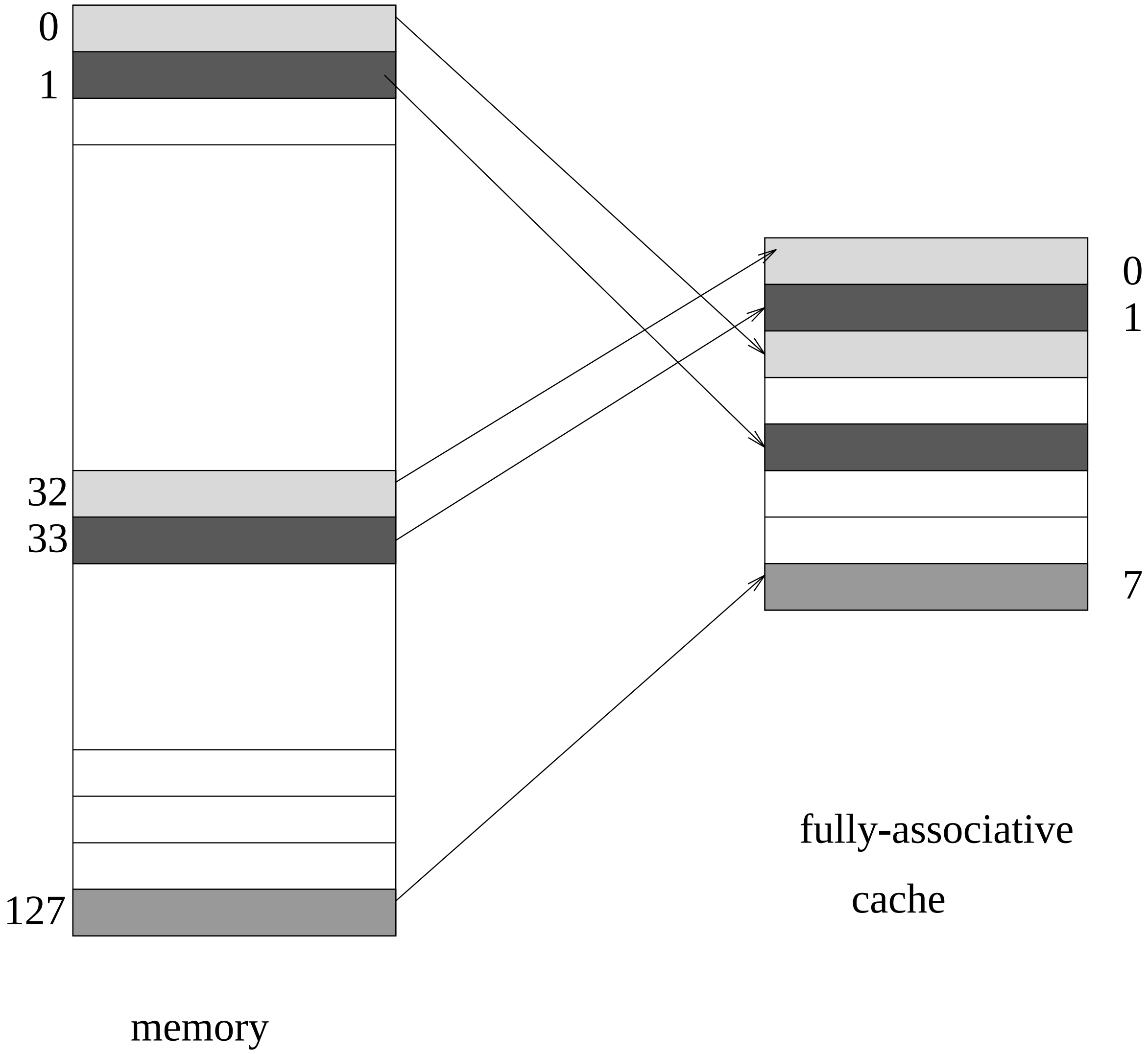
Total capacity: $4 \times 16 = 64$Kbytes



Fully-associative mapping

Used only for small special-purpose caches.

However, the general mapping will be used in paging systems (see later)



Set-associative mapping

Example: Two-way set associative

index = log(number of entries in the cache /associativity)

d = log(number of bytes in data entry)

tag	index	d

(note tag is one bit longer for two-way set-associative than for direct-mapped)

tag	data
tag	data
tag	data
tag	data
tag	data
tag	data
tag	data
tag	data

tag	data
tag	data
tag	data
tag	data
tag	data
tag	data
tag	data
tag	data

Needs one comparator for each bank and one multiplexor to see if we have a hit (of course we also keep valid bits)