# Cache

M. Sonza Reorda

**Politecnico di Torino**
**Dipartimento di Automatica e Informatica**

## Introduction

**A cache (corresponding to a fast DRAM) is often inserted between the CPU and the slower (but larger) main memory (corresponding to a SRAM).**

**The cache is normally controlled by a *cache controller*.**

**Depending on the CPUs, either the cache controller, or both the cache controller and the cache are on-chip.**

## Cache controller

It orchestrates all the cache operations, including (in case of a miss) the *cache line fill* operation, corresponding to

- Possibly writing the cache line to be substituted to the memory
- Uploading the new cache line from the memory
- Generating the required READY signal to the CPU.

The cache line fill takes advantage of the efficiency of the burst transfer mode.

## Write operation

When a write operation is performed on a cache, two mechanisms can be adopted:

- Write-through
- Write-back.

## Write-through

**The write operation is performed both on the cache and on the main memory.**

**To avoid blocking the CPU during this operation, a *write buffer* can be exploited.**

**Write-through ensures coherence in multiprocessor systems.**

## Write-back

**Write operations are performed on the cache data, only.**

**Cache data are written back to memory when**

- **A cache miss arises, and the cache line is going to be substituted, or**
- **A `WBINVD` (write back and invalidate data cache) instruction is executed, or**
- **The FLUSH signal is activated.**

**Write back means (w.r.t. write-through) slower cache miss operations, but faster accesses to cache.**

## Write-allocate strategy

**What exactly happens when a write operation causes a cache miss?**

**Different solutions can be followed:**

- **The cache writes in memory and the CPU starts again, while the cache controller performs the cache line fill (*write-allocate*)**
- **The cache writes in memory, but the cache content is not updated. This results in a slightly lower cache hit rate, but simplifies the cache controller.**

## Cache invalidation

**If someone (e.g., another processor, or a DMA controller) can write in the memory beyond the processor, a mechanism is required, to tell the processor that the cache line that possibly stores the same block is no more coherent.**

**This is done associating a *valid bit* to each cache line, that can be reset from the cache controller.**

## Cache flush

Sometimes the processor asks the cache controller to immediately perform the write-back operation of a line to memory.

This happens for example when write-back is adopted, and a block of data in the cache must be transferred to a peripheral through DMA.

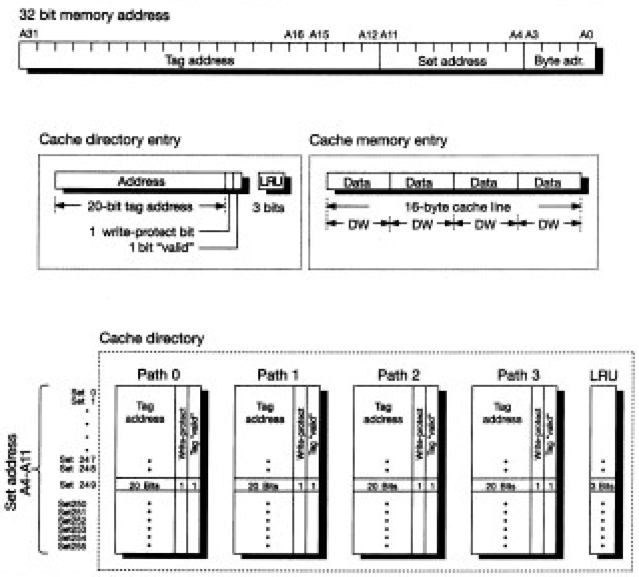
Suitable instructions often support this operation (named *cache flush*).

## Example

Let us consider a cache with the following characteristics

- Size: 16 Kb
- Cache line size: 16 bytes
- Allocation policy: 4-way set associative.

# Implementation



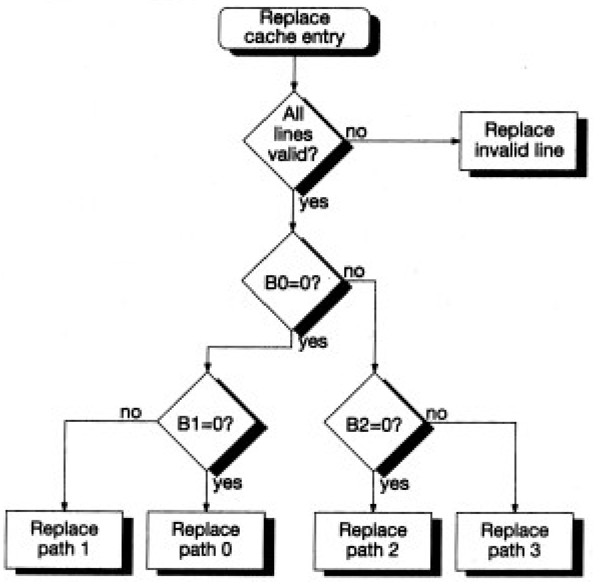
# Replacement strategy

**If a set-associative policy is adopted, a strategy is required to select the line in the set to be substituted when a miss occurs.**

**Possible solutions are**

- **LRU**
- **Pseudo LRU**
- **Random.**

## Pseudo LRU



## On-chip cache

The first Intel processor with an on-board cache was the 80486. The on-board cache size increased in the following processors:

- 80486: 8Kb (4-way set-associative with 128 sets, 16-byte cache lines)
- Pentium III: 16 Kb + 16 Kb
- Athlon: 128 Kb.

## L1 and L2

Very often, a second cache is hosted on the motherboard.

Originally, the on-line cache was named *L1* and the external cache was named *L2*.

In more recent microprocessors (e.g., AMD-K6 or Pentium4), even the L2 cache can be integrated on the CPU chip.

## L2 cache

When two levels of caches are available

- A miss in L1 causes an access to L2
- A miss in L2 causes an access to the memory.

Normally, L1 works at the CPU speed (one clock cycle for accessing to cache data), while L2 is slower (e.g., two clock cycles per access).

## Cache effectiveness

**Clearly, it depends on the kind of application.**

**Mathematical applications do normally access to data in a much more *local* way; therefore, they take grater advantage of caches.**

**Data base applications normally perform more *spread* memory accesses, and take less advantage from caches.**

## I/O accesses

**In the Intel processors, I/O accesses never involve any cache, and are directly performed on the corresponding I/O port.**

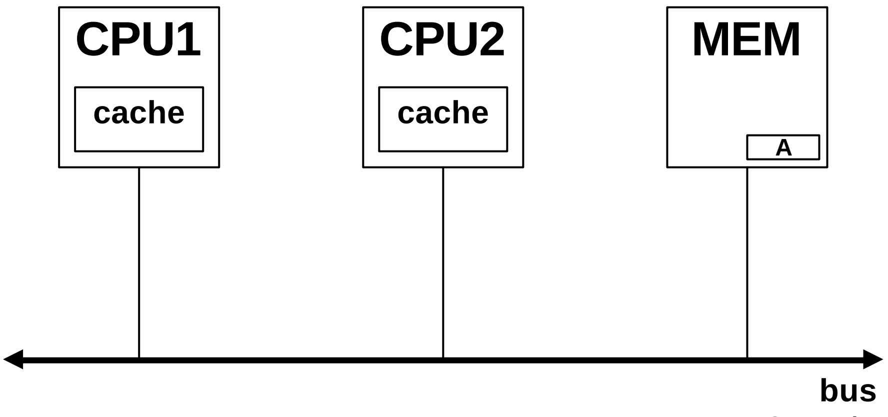
**If memory mapped is adopted, all I/O ports must be marked as *non-cachable*.**

## Cache consistency

Cache consistency problems can arise when more than one cache exists in the system. This can happen

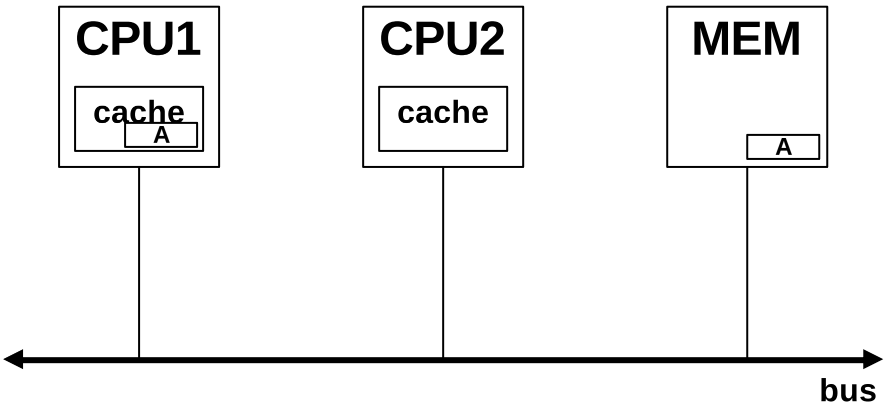
- When a L1 and a L2 cache are present
- When more than one processor with cache is present.

## Example



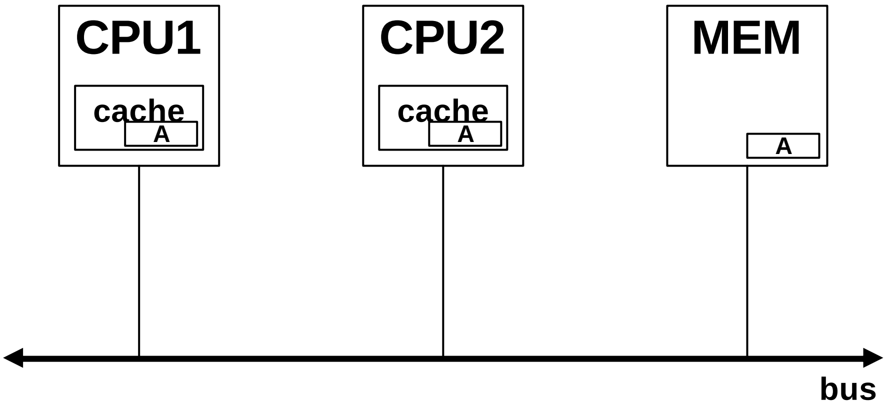
## Examp

CPU1 reads A



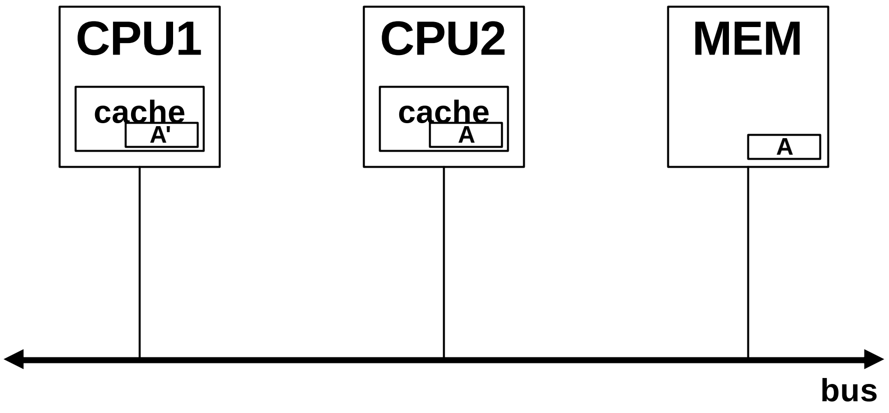
## Examp

CPU2 reads A



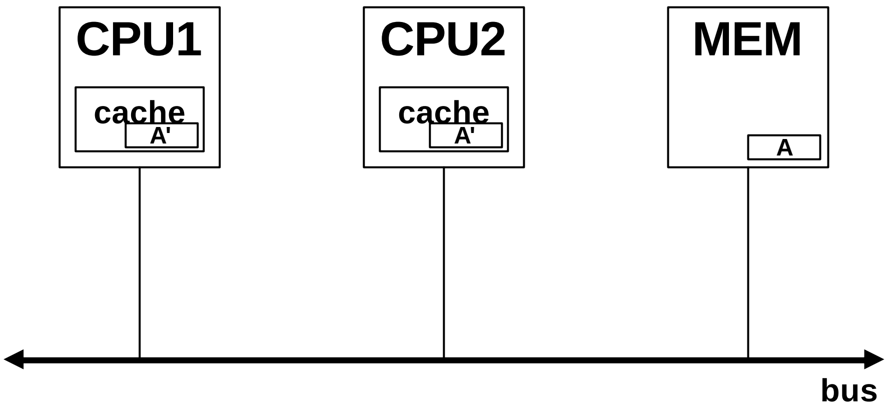
## Examp

CPU1 increments A
(CPU1 uses write-back)



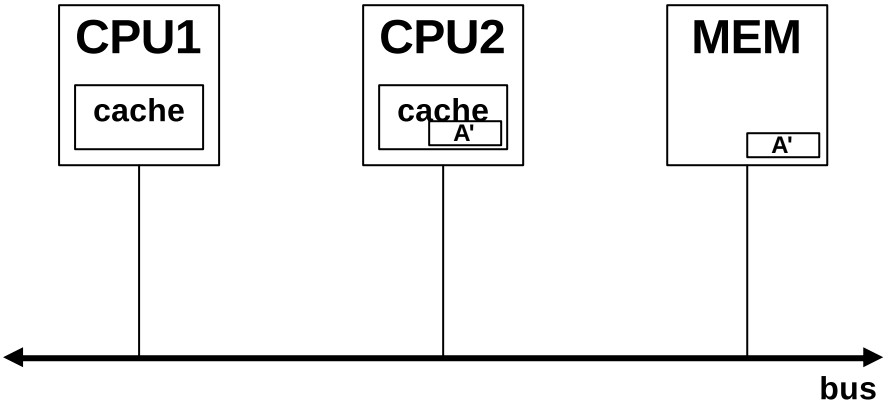
## Examp

CPU2 increments A
(CPU2 uses write-back)



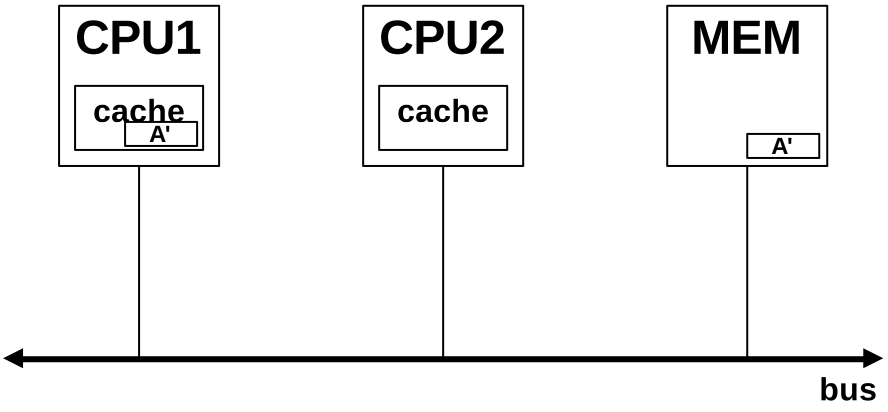
## Examp

CPU1 writes A' to memory

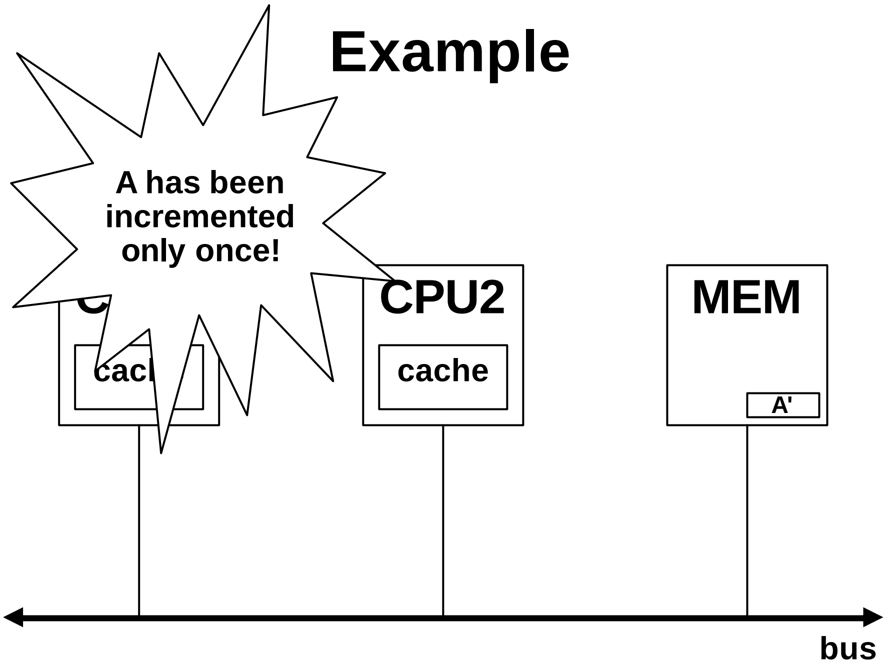


## Examp

CPU2 writes A' to memory



## Example



## Cache coherence protocol

To prevent consistency problems, a cache coherence protocol must be adopted.

One of these protocols is the MESI protocol. The MESI protocol was first implemented in the Pentium.

## The MESI protocol

**The MESI protocol is based on associating each cache line with a state that can assume 4 different values:**

- **Modified (M)**
- **Exclusive (E)**
- **Shared (S)**
- **Invalid (I).**

## Modified

**The data in the cache line in only available in a single cache in the entire system. Its value is different than that in the memory.**

**The line can be read and written to without the need for any access to other caches or to the memory.**

## Exclusive

**The data in the cache line is only available in a single cache in the entire system.**

**However, the data has not been modified up to now, so its value is identical to that in the memory.**

**If a write operation occurs on the data, the cache status will transform into *modified*.**

## Shared

**The data in the line could be stored in other caches in the system.**

**Any write operation on the data (independently on the adopted write-through or write-back strategy) must be also immediately performed on the memory, so that other caches storing the same data can invalidate the corresponding lines.**

## Invalid

**The line is logically not available in the cache. The cause for this could be either that the line is empty, or that the line contains an out-of-date data.**

**Every access to an invalid line causes a miss.**

**A write access must be performed as a write-through. Write-allocate is not supported.**

## Write-back and write-through

**The MESI protocol was designed for write-back caches without write-allocation.**

**It can be extended to the case of write-through caches: in this case the M and E states do not exist.**

## Snooping

Snooping is the operation performed by each cache controller, which monitors the external bus master, and may interact with other cache controllers.

One cache controller may ask another which is the state of a given line (*inquiring*).

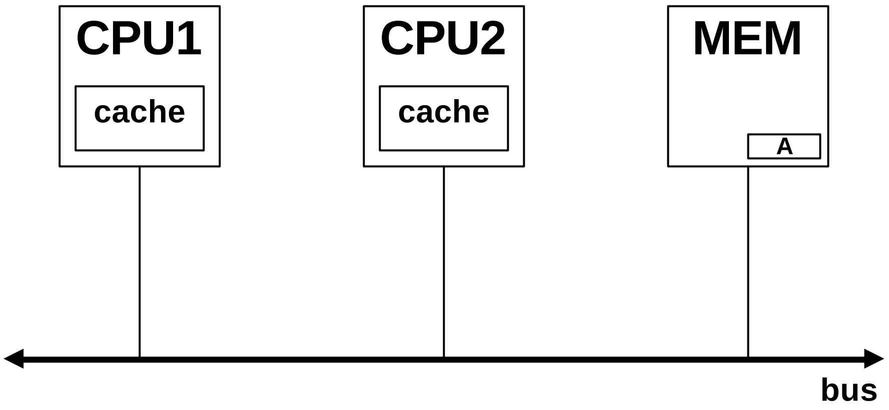
As a consequence of such an inquire, some action may be performed, and the state of the line can be changed.

## The MESI protocol

The protocol specifies which action should be taken by each cache controller when

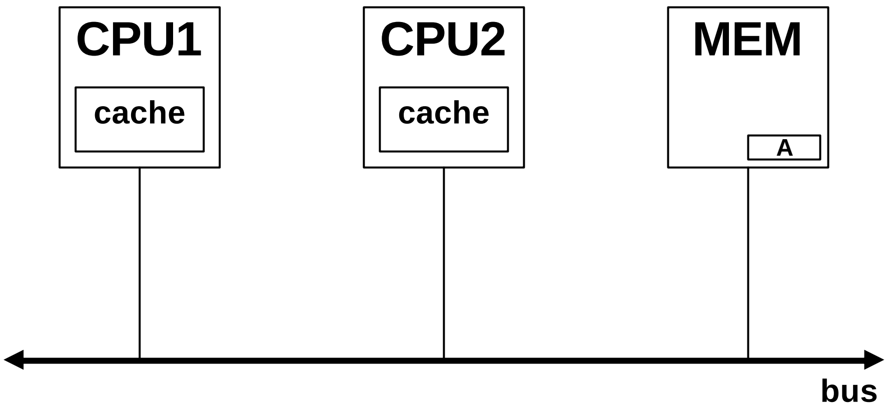
- An operation is performed on a local variable
- An operation is performed through the bus on a variable which is also stored locally
- An inquire operation is received through the bus.

## Example



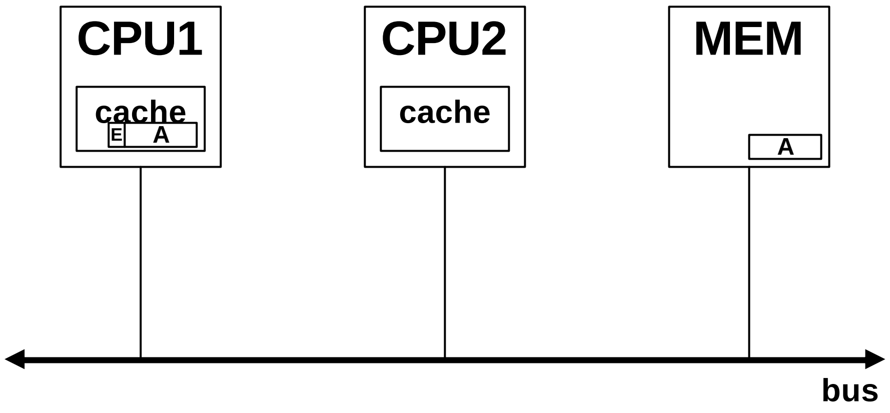
## Examp

At the beginning, all cache lines are marked Invalid.



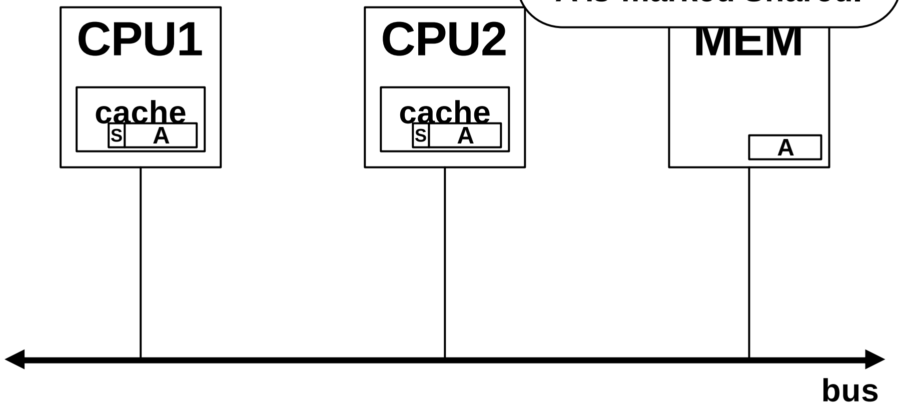
## Examp

CPU1 reads A.
A is marked
Exclusive.



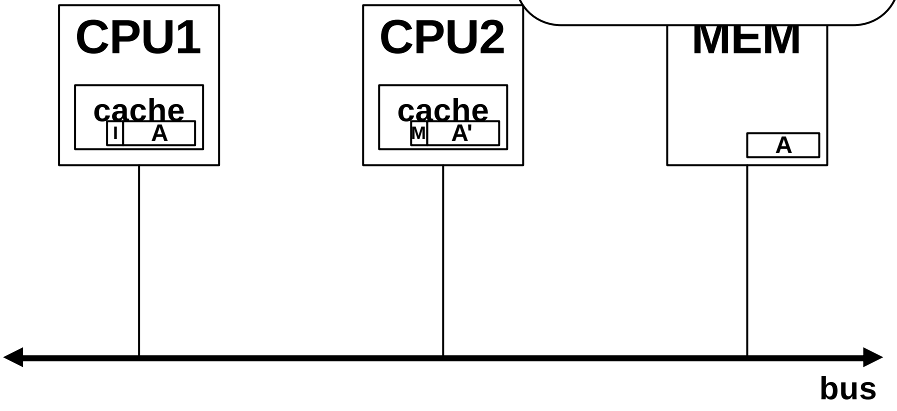
## Examp

CPU2 reads A.
Cache1 sees the
operation and, by
snooping, informs
Cache2 that another
copy of A exists.
A is marked Shared.



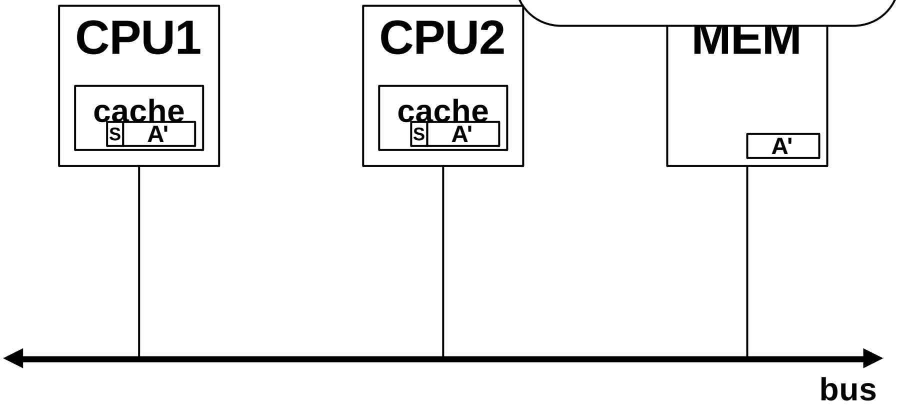
## Examp

CPU2 writes on A, which is now marked Modified.
Cache2 informs the other caches, that invalidate their copy.



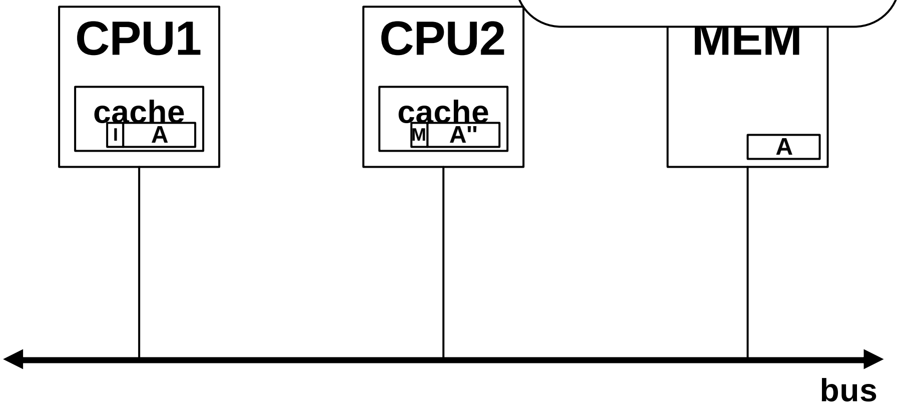
## Examp

CPU1 reads A.
Cache2 writes A to memory.
Cache1 reads A.
A is marked Shared everywhere.



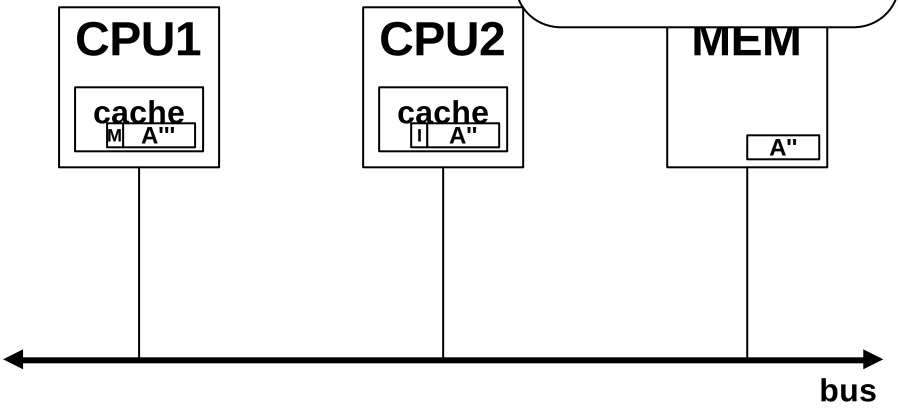
## Examp

CPU2 writes again on A, which is now marked Modified. Cache2 informs the other caches, that invalidate their copy.



## Examp

CPU1 writes on A. Cache2 writes A to memory. A is marked Invalid.
Cache1 reads A. A is marked Modified.



## Cache inclusion

**If the system includes L1 and L2 caches, the MESI protocol should also take into account that each processor could store a variable either in L1, or in L2.**

**Consistency between the two caches is supported first by assuming that L2 includes L1, i.e., all the data in L1 are always present also in L2.**

## MESI extension

**When two levels L1 and L2 are present, consistency among processors caches can be obtained in two ways**

- **By forcing L1 to adopt the writh-through strategy towards L2, and then adopt the standard MESI protocol among the L2 caches (IBM S/390 SP)**
- **By allowing L1 to adopt the write-back strategy: in this case the MESI protocol becomes more complex (Pentium II).**