The Dining Philosophers Problem

Cache Memory

The dining philosophers problem: definition

It is an artificial problem widely used to illustrate the problems linked to resource sharing in concurrent programming.

The problem is usually described as follows.

- A given number of philosopher are seated at a round table.
- Each of the philosophers shares his time between two activities: thinking and eating.
- To think, a philosopher does not need any resources; to eat he needs two pieces of silverware.

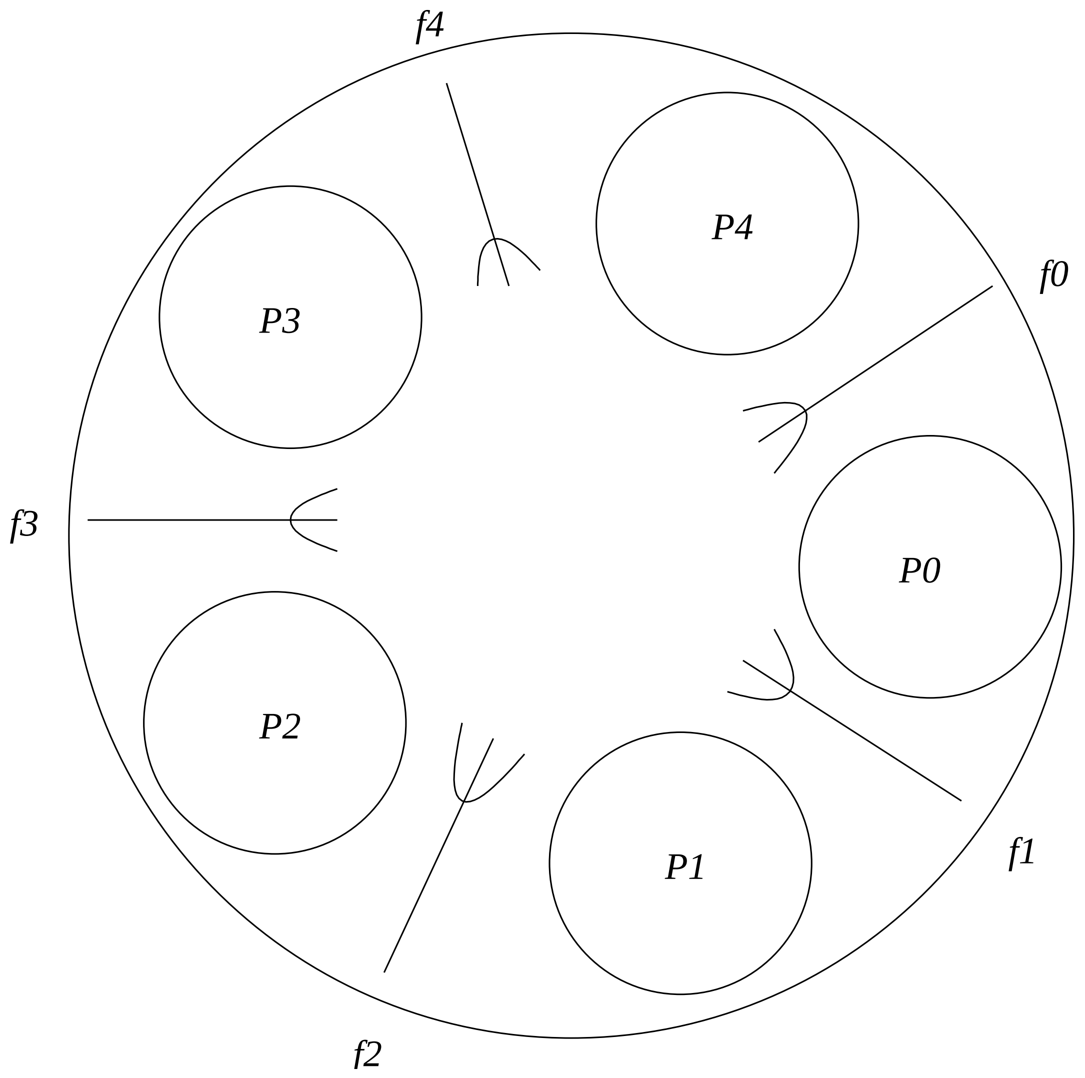
- However, the table is set in a very peculiar way: between every pair of adjacent plates, there is only one fork.

- A philosopher being clumsy, he needs two forks to eat: the one on his right and the one on his left.

- It is thus impossible for a philosopher to eat at the same time as one of his neighbors: the forks are a shared resource for which the philosophers are competing.

- The problem is to organize access to these shared resources in such a way that everything proceeds smoothly.

The dining philosophers problem: illustration



The dining philosophers problem: a first solution

- This first solution uses a semaphore to model each fork.
- Taking a fork is then done by executing a operation `wait` on the semaphore, which suspends the process if the fork is not available.
- Freeing a fork is naturally done with a `signal` operation.

```
/* Definitions and global initializations */
#define N = ? /* number of philosophers */
semaphore fork[N]; /* semaphores modeling
                      the forks */
int j; for (j=0, j < N, j++) fork[j]=1;
```

Each philosopher (`0` to `N-1`) corresponds to a process executing the following procedure, where `i` is the number of the philosopher.

```
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]); wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]); signal(fork[(i+1)%N]);

  }
}
```

The dining philosophers problem: a first solution - the deadlock

- With this first solution, a deadlock is possible.

- Indeed, if each philosopher executes `wait(fork[i])` before any philosopher has executed `wait(fork[(i+1)%N])`, each philosopher is then holding one fork and waiting for the second.

- The problem is that each philosopher must acquire two resources and does this

  1. in two steps,

  2. in an order that can lead to a deadlock, and

  3. without the possibility of the operation being canceled

To avoid deadlocks, one of these three items has to be eliminated.

The dining philosophers problem: a second solution

In this solution, the order in which the philosopher `N-1` picks up his forks is modified.

```
/* Philosophes 0 à N-2 */
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]);
    wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1)%N]);

  }
}
```

```
/* Philosophe N-1*/
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[(i+1)%N]);
    wait(fork[i]);
    eat();
    signal(fork[(i+1)%N]);
    signal(fork[i]);
  }
}
```

The dining philosophers problem: a second solution - deadlock ?

In the second solution, no deadlock is possible. The principle that has been applied is the following.

- The shared resources are ordered (`0` to `N-1` for the forks).
- A process that needs several resources must acquire them in increasing order with respect to the global order on resources.
- No deadlock is then possible.
  - Indeed, in a hypothetical deadlock situation, consider the process that holds the highest order resource.
  - This process has already acquired all the lower order resources that it might need and thus cannot be blocked.

Furthermore, assuming the the semaphores are fair, not process can be blocked forever.

- Indeed, if a process is waiting for a fork, semaphore fairness ensures that it will has access to it as soon as it is freed.

- For a fork never to be freed, the process holding it must be waiting for a higher-order fork.

- Since the chain of blocking waits takes us up in the fork order, it must stop when the highest-order fork is reached. Indeed, this one will necessarily be the last one acquired by the process using it and will thus eventually be freed.

The dining philosophers problem: a third solution

In this solution, resource acquisition is done in one step. For this we will use a monitor through which all the fork management will be done.

- This monitor uses a table `f[]`, where the number of forks available for each philosopher (0, 1 or 2) is held.

- If philosopher `i` does not find two available forks, he is suspended on his own wait queue `oktoeat[i]`. This queue is implemented by a simple semaphore since it never contains more than one process. The implementation is done directly with semaphores and the class `Waitqueue` is not used.

- A table of Booleans `waiting` is used to indicate for each philosopher whether of not he is waiting in his queue `oktoeat`. Note that a call to `semNbWait` on the semaphore used to implement the queue `oktoeat` would not provide the same information as `waiting`, since mutual exclusion is freed before the call to `SemWait`.

The dining philosophers problem: The fork monitor - constructors

```
public class ForkMonitor
 { private int nb;
   private int urcount, f[];
   private boolean waiting[];
   private Semaphore urgent, mutex, oktoeat[];

   public ForkMonitor(int N)
    { nb = N; urcount = 0;
      f = new int[nb];
      for (int i=0; i<nb ; i++) f[i] = 2;
      urgent = new SemaphoreFIFO(0);
      mutex = new SemaphoreFIFO(1);
      oktoeat = new Semaphore[nb];
      for (int i=0; i<nb ; i++)
        { oktoeat[i] = new Semaphore(0);
          waiting[i] = false;
        }
    }
```

The dining philosophers problem: the fork monitor - picking up forks

```
public void takeFork(int i)
  { mutex.semWait();
    if (f[i] != 2)
      { waiting[i] = true;
        if (urcount > 0) urgent.semSignal;
            else mutex.semSignal();
         oktoeat[i].semWait();
         waiting[i] = false;
      }
    f[(i+1) % nb]--;
    f[(i-1+nb) % nb]--;
    if (urcount > 0) urgent.semSignal();
      else mutex.semSignal();
  }
```

The dining philosophers problem: the fork monitor - releasing forks

```
  public void releaseFork(int i)
    { mutex.semWait();
      f[(i+1) % nb]++;
      f[(i-1+nb) % nb]++;
      if ((f[(i+1) % nb] == 2) && (waiting[(i+1) % nb])
        { urcount++;
          oktoeat[(i+1) % nb].semSignal();
          urgent.semWait(); urcount--;
        }
      if ((f[(i-1+nb) % nb] == 2) && (waiting[(i-1+nb) % nb]))
        { urcount++;
          oktoeat[(i-1+nb) % nb].semSignal();
          urgent.semWait(); urcount--;
        }
      if (urcount > 0) urgent.semSignal();
        else mutex.semSignal();
   }
}
```

The dining philosophers problem: Using the fork monitor

Once an object for the forks has been created,

```
ForkMonitor F = new ForkMonitor(nb);
```

each philosopher executes the following procedure, in which the argument is the philosopher number.

```
philosophe(int i)
  { while(true)
      { think();
        F.takeFork(i);
        eat();
        F.releaseFork(i);
       }

  }
```

In this solution no deadlock is possible. Indeed, if all processes are waiting, no fork is used, a contradiction. Nevertheless, it is possible for two philosophers to exclude a third.

Cache memory

Cache memory: principle

- In the machine ULg03, access to DRAM is done at the same speed as access to registers.

- This is only possible if the operation of the machine is very significantly slowed down.

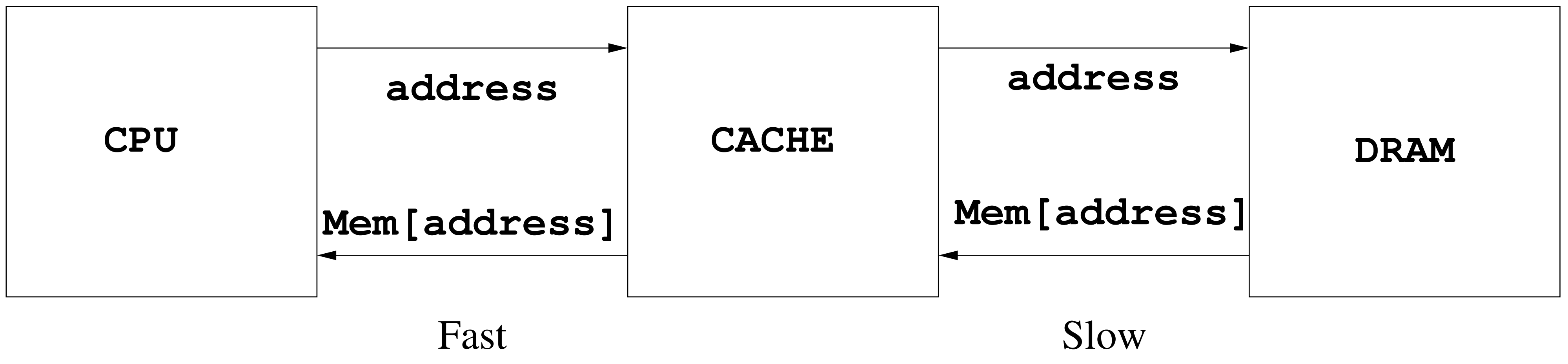
- There is however another possibility: to keep in a small fast memory a copy of part of the DRAM. Such a memory is called a*cache*.

- One can then hope that most accesses will be to the cache and will thus be fast. When the required data is not in the cache, it must be transferred from DRAM, which is slow and will force the processor to wait.

Cache memory: its associative character

- Main memory creates an association between values and addresses.
- Cache memory must provide the same function and thus associate data, not to its own addresses, but to those of the main memory.
- Cache memory must thus contain pairs (address, data) and make it possible to very quickly find the data part of a pair given its address part.
- A memory providing this function is called an *associative memory*.
- When searching for the data corresponding to an address in a cache memory, either it is present in the cache (*cache hit*), or it cannot be found there (*cache miss*).

Cache memory: the overall schema



When the required data is not in the cache, the processor is blocked until the data is transferred from the DRAM to the cache.

Why are cache memories effective?

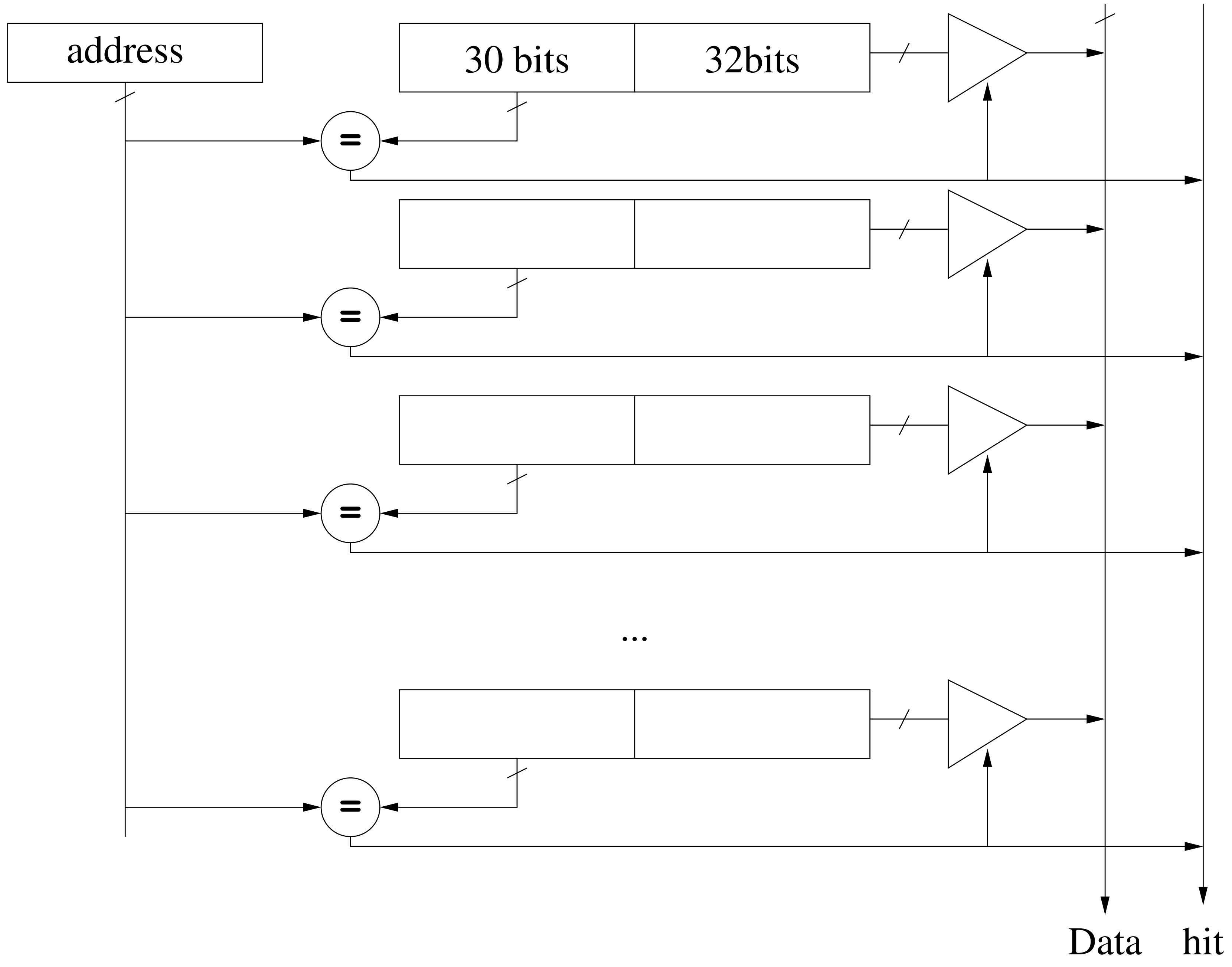
- If accesses to DRAM where perfectly random, a cache memory would not be very effective. Fortunately, these accesses are far from random.

- Indeed, at times that are close to each other, one often accesses the same addresses. This is called *temporal locality*.

- When accessing a given address, it is frequent that the next addresses to be accessed will be close. This is called *spatial locality*.

- Cache memories and their management are thus designed to exploit these localities.

Totally associative cache



Totally associative cache: characteristics

- An address is associated to each memory word stored in the cache. As much space is thus used for addresses as for the data.

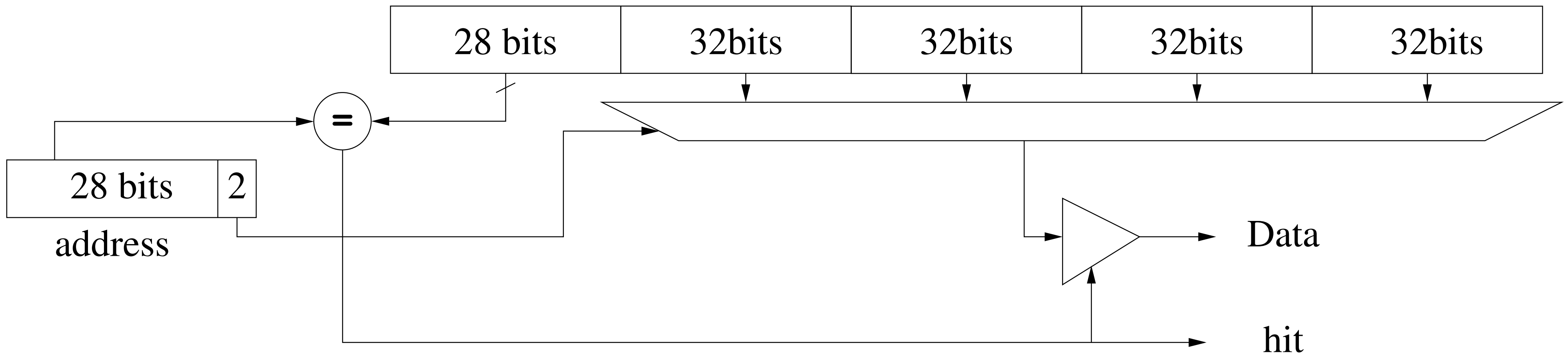
- The cache includes a comparator for each stored word, which limits its capacity.

- A good replacement policy is *LRU* (Least Recently Used), but it is costly to implement.

- It is also possible to associate an address to more than one word. This is called a *block cache*.

Block cache

The organization is the same as in an associative cache, but a data block of 2^B words is now associated with each $30 - B$ bit address.



When there is a "cache miss", an entire block must thus be transferred from memory. However, because of spatial locality, this is usually useful.

In practice, there is s block size that minimizes the number of cache misses.

Direct mapped cache

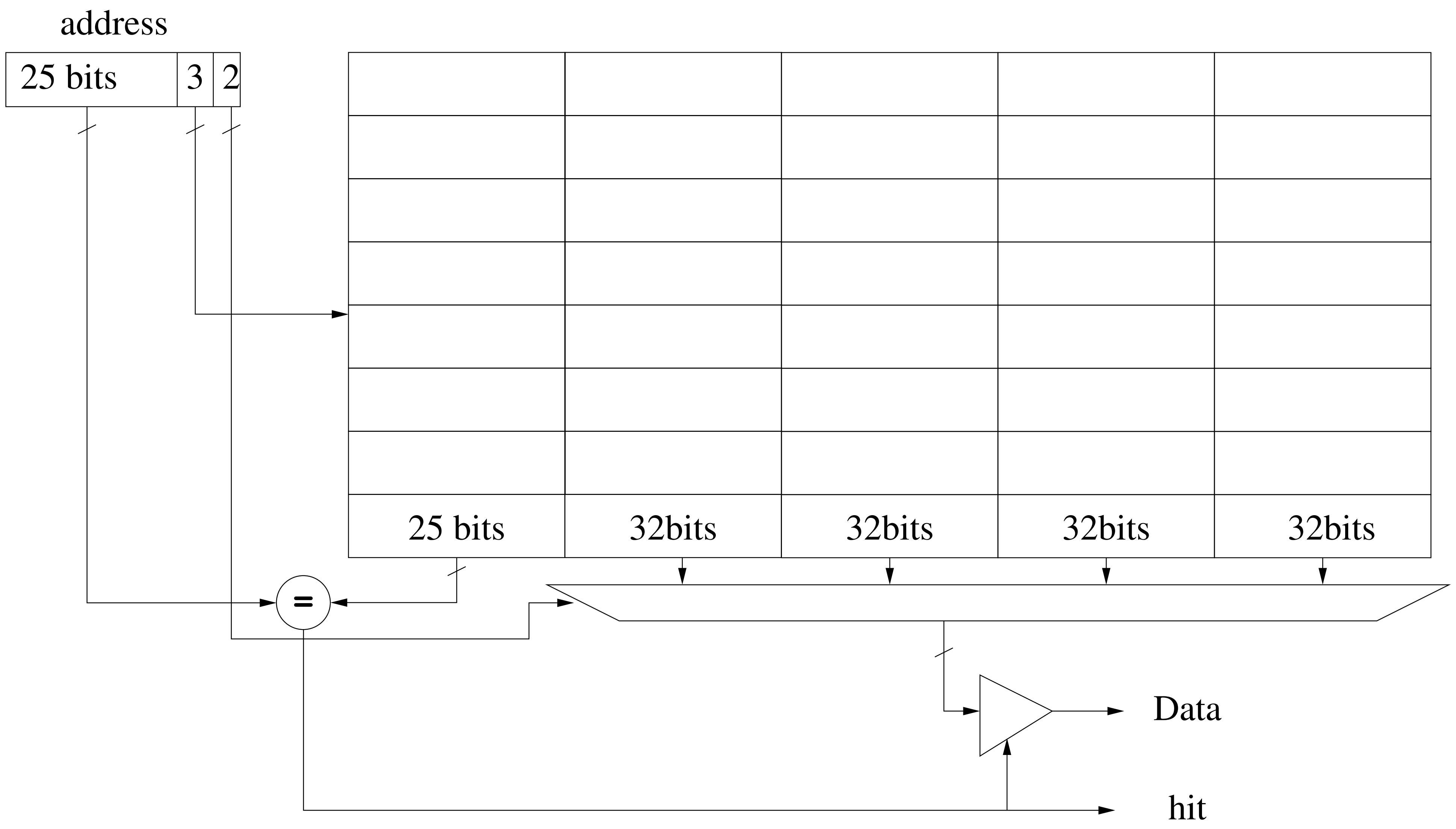
- A *direct mapped cache* uses part of the memory address as cache address. The rest of the address is then compared when doing a cache look up.

- There is only one possible cache address for each memory address. A replacement policy is thus not needed.

- On the other hand, it is not possible to have in cache the content of different addresses that correspond to the same cache address.

- In a direct mapped cache, only one comparator is needed. It can thus be built with ordinary memory.

A block direct mapped cache



An intermediate solution: set associative cache

- A compromise between totally associative cache and direct mapped cache is *set associative cache*, which combines the two techniques.

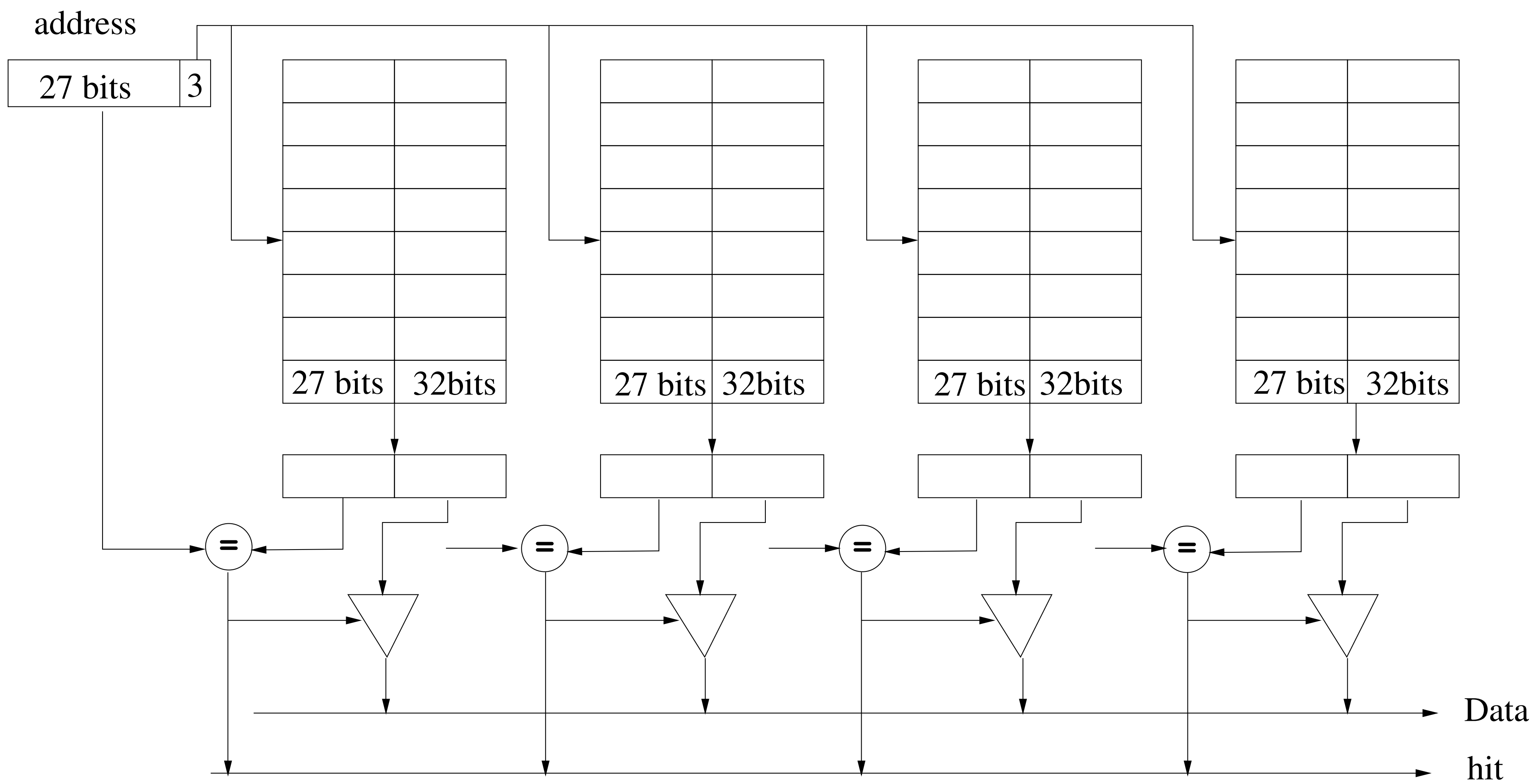
- In this organization there are N direct mapped caches, the selection between these being done as in an associative cache.

- A set associative cache, can also use blocks instead of words.

- From a practical point of view, $N = 8$ is usually sufficient.

- For a large enough cache, a random replacement strategy yields results that are almost as good as LRU.

A set associative cache



Cache and write operations

- Most memory operations are reads. What must be done for writes? Is it necessary to immediately modify the memory? There are several options.
  - *write-through :* immediate modification of the memory, the CPU waiting for the operation to be completed.
  - *write-back :* delayed modification.
- Delayed modification can improve performance, but it is necessary to know for each element stored in the cache if it needs to be written to memory or not.
- In the case of multiprocessors, one has to deal with the *cache coherence* problem.

Cache memory versus virtual memory

- In the context of virtual memory, the cache can use either physical or virtual addresses.

- A cache working with virtual addresses avoids address translation as long as one remains within the cache. However, the entire cache becomes non valid upon a change of context.

- A virtual cache is problematic if several virtual addresses can correspond to the same physical address (data sharing through aliasing).