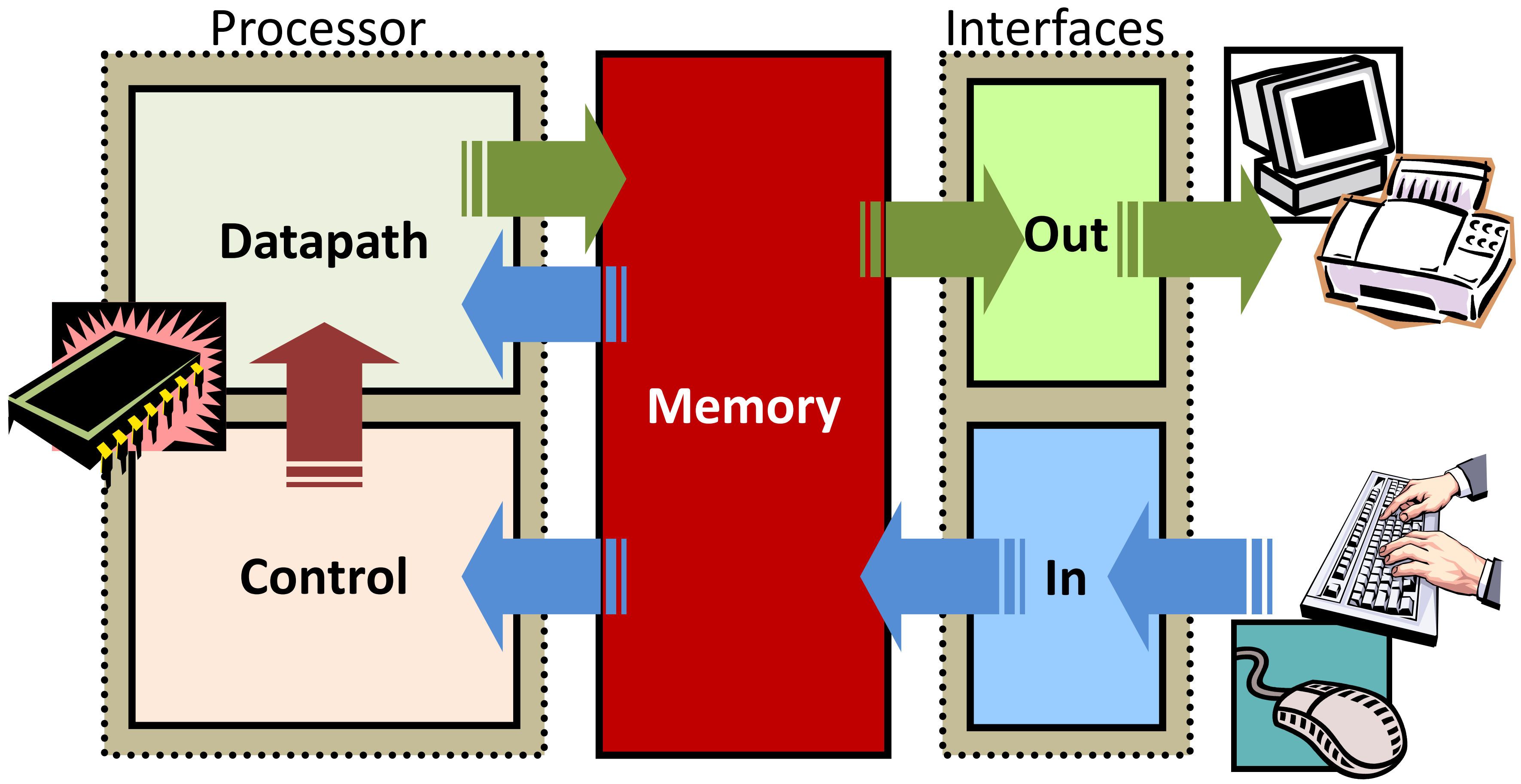
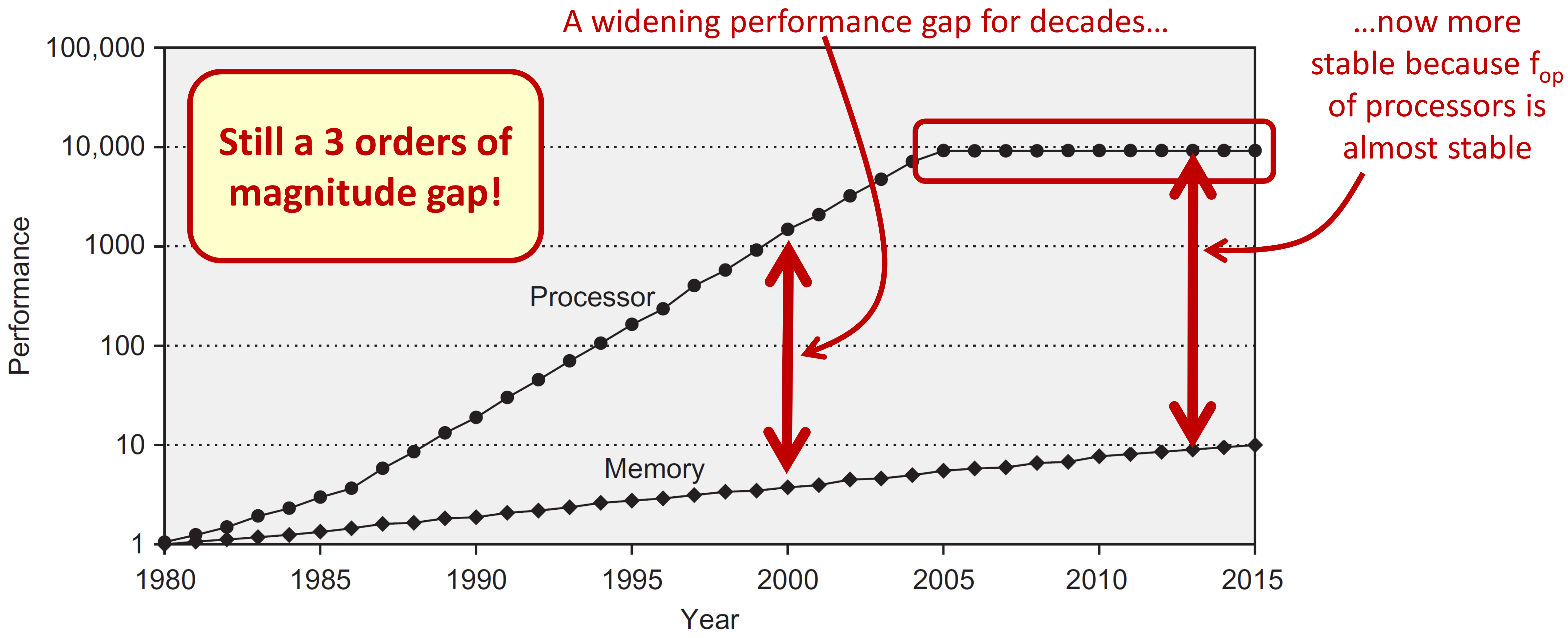
CS-200
Computer Architecture
—
Part 3a. Memory Hierarchy
Caches

Paolo Ienne
<paolo.ienne@epfl.ch>

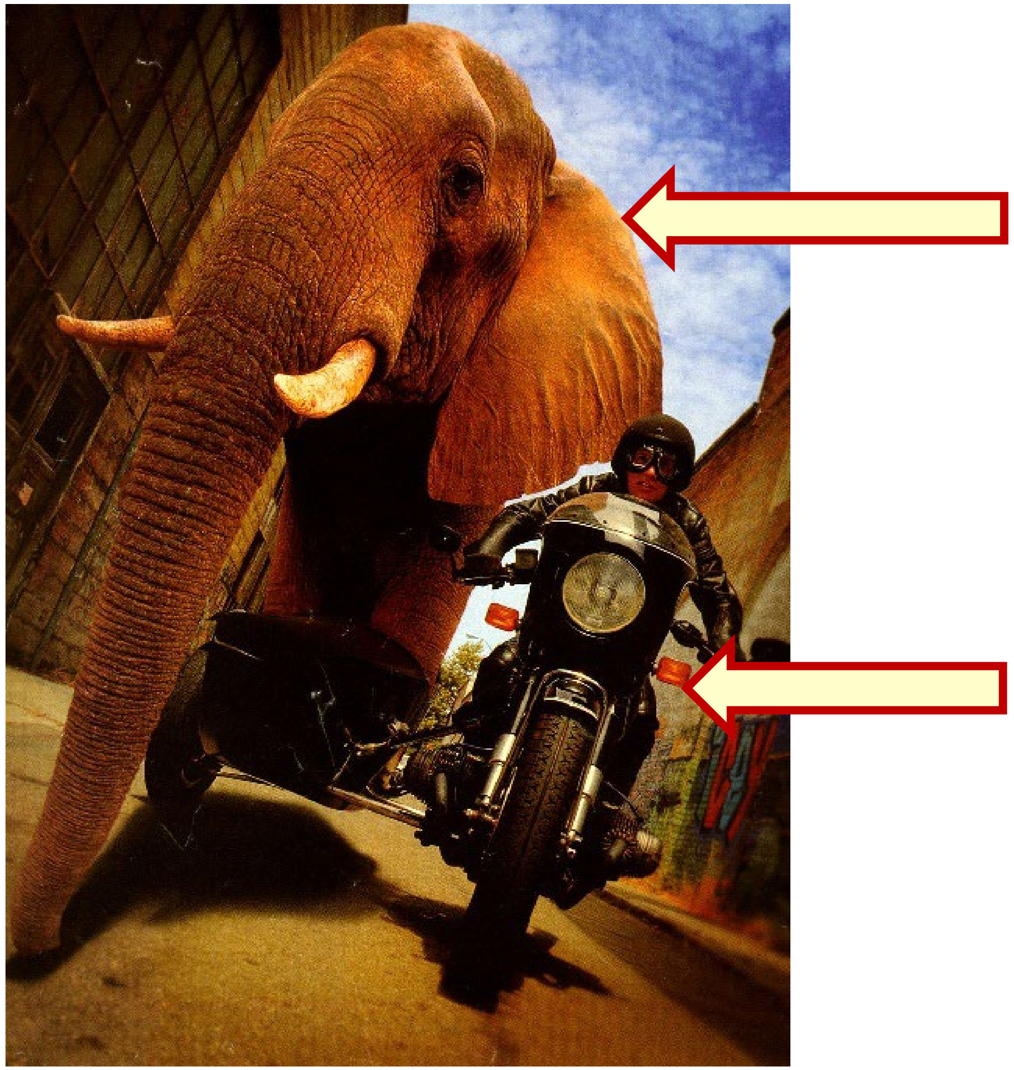
The Five Classic Components of a Computer



Memory Performance over Time



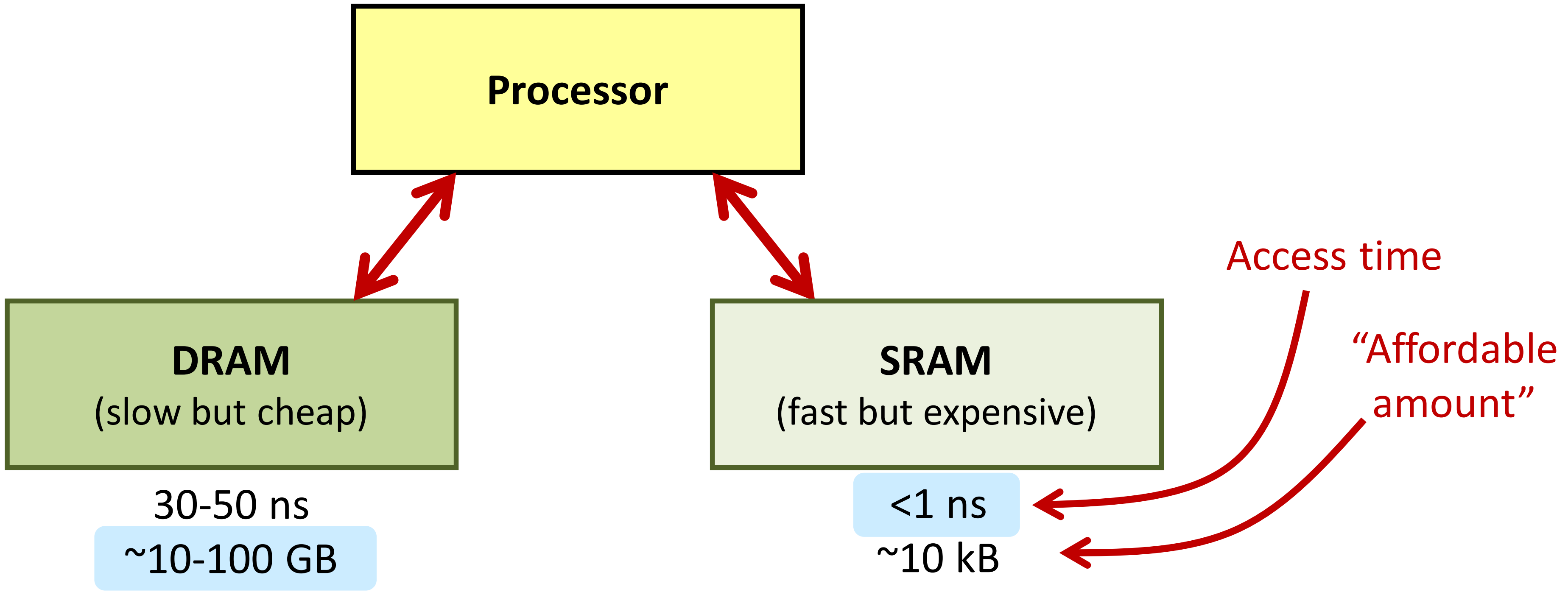
The Tension between Big and Fast!



This is your memory subsystem
You want it big!

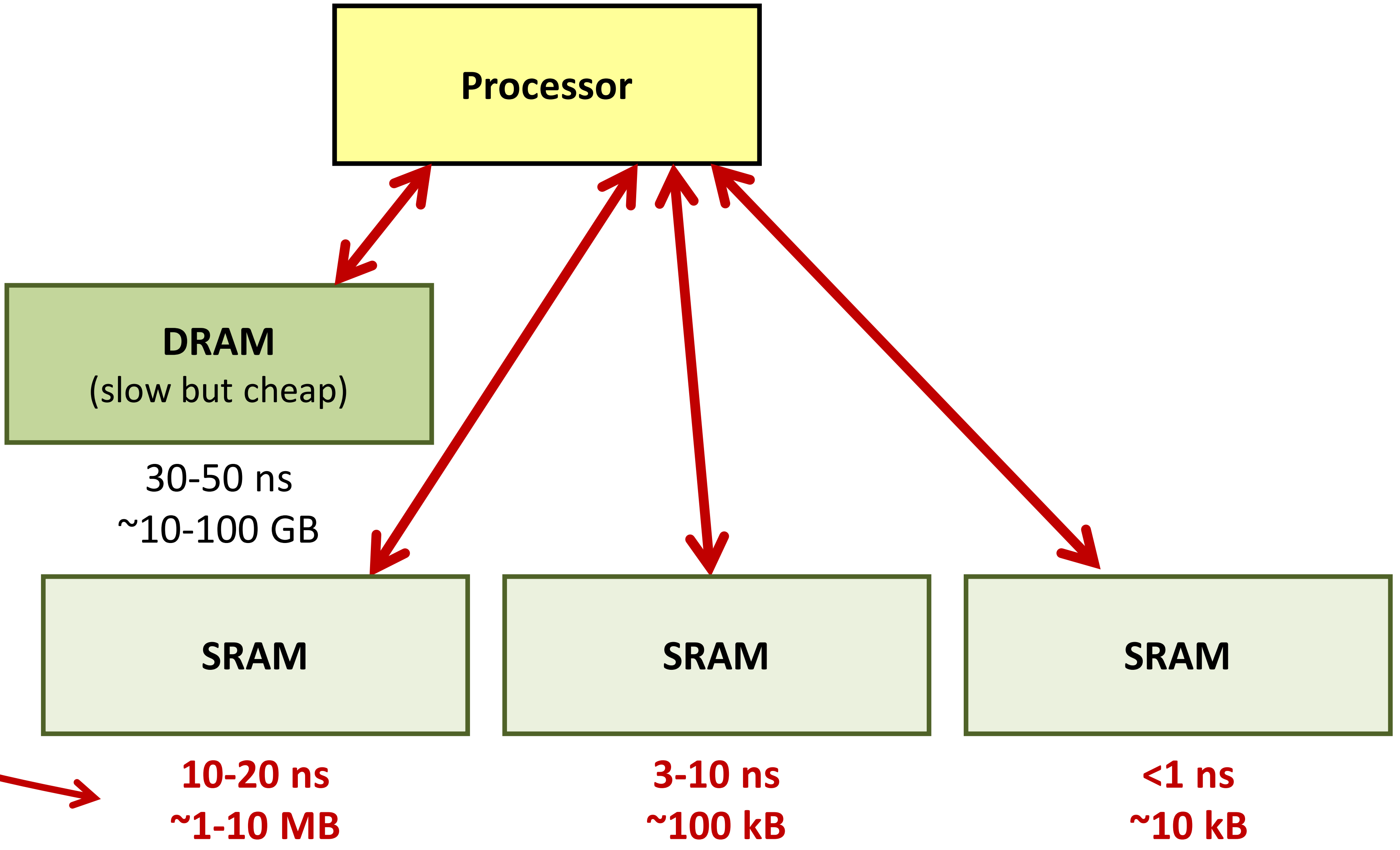
This is your CPU
You want it fast!

Our Goal Today: Use Different Memories



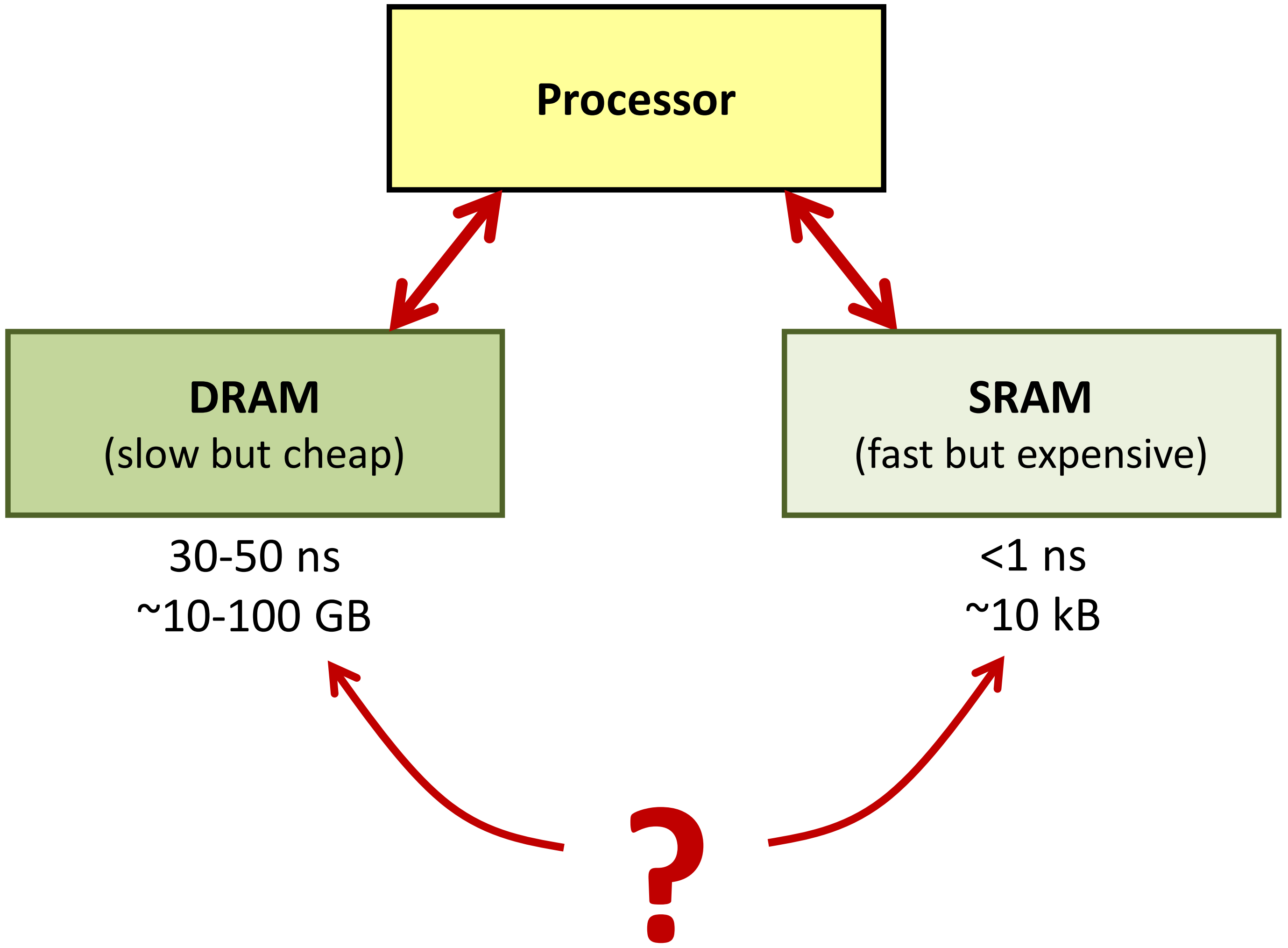
Can we get the best of both worlds?!

Our Goal Today: Use Different Memories



Using the same technology, **smaller is faster**

Where to Put What—and How?



What Memory to Use?

- Instructions corresponding to lines 3-5 are **read over and over**: should be in fast memory
- If variables **`i`** and **`sum`** are stored in memory, they are also **used often** and should be stored in fast memory
- One would like to anticipate the future and load **following** instructions and vector elements

```
1:    i = 0;
2:    sum = 0;
3:    while (i < 1024) {
4:      sum = sum + a[i];
5:      i = i + 1;
6:    }
```

Spatial and Temporal Locality

Two important criteria to decide on placement:

- **Temporal Locality**
  - Data that have been **used recently**, have high likelihood of being used again
    - Code: loops, functions,…
    - Data: local variables and data structures

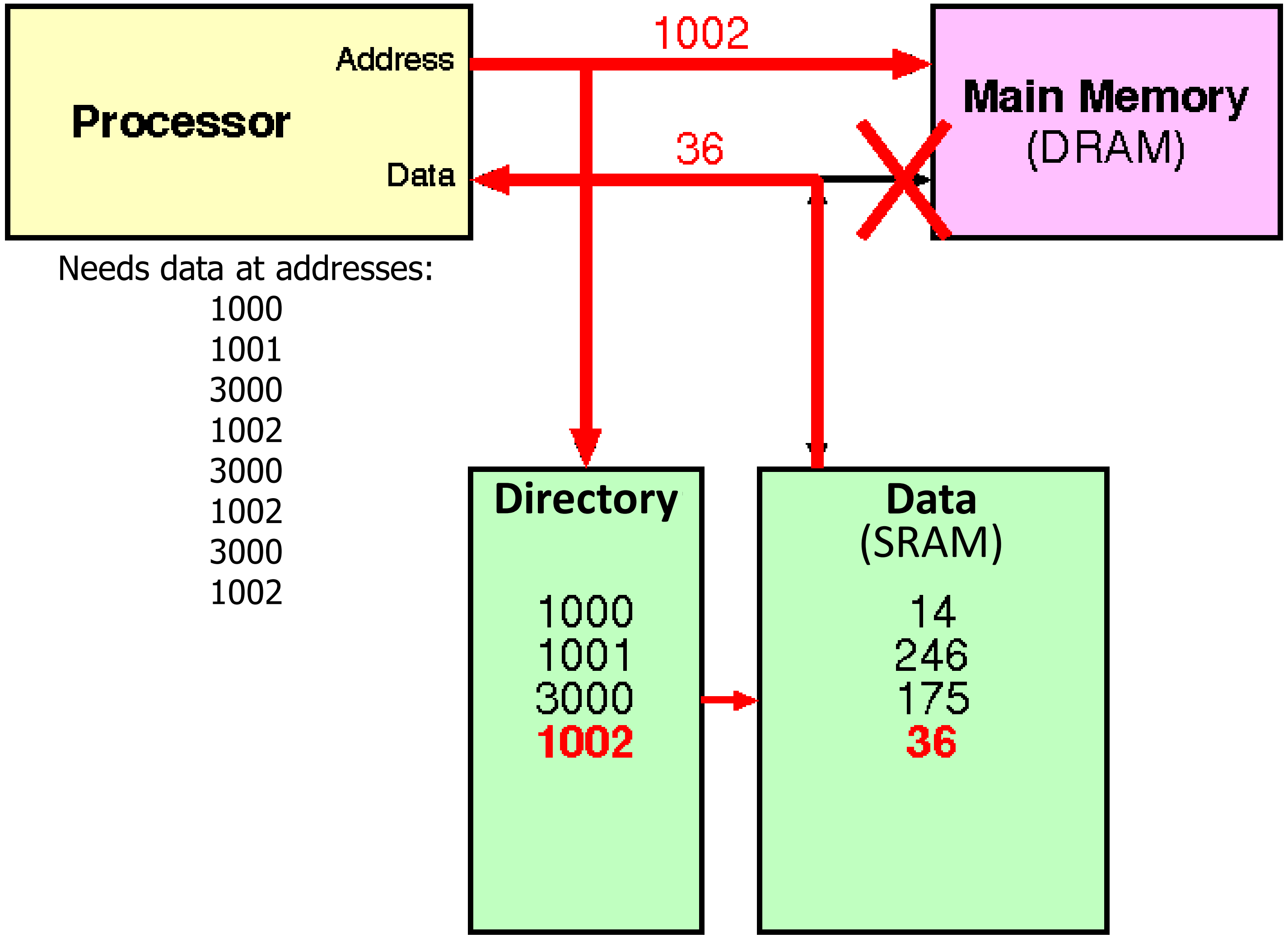
- **Spatial Locality**
  - Data which **follow in the memory other data** that are currently being used are likely to be used in the future
    - Code: usually read sequentially
    - Data: arrays

Our Placement Policy Must Be…

- **Invisible to the programmer**
  - One could analyse data structures and program semantics to detect heavily used variables/arrays and thus decide placement → Ok in some contexts (embedded) but we want to have the programmers not go through this hassle
  - We will add **hardware** to help

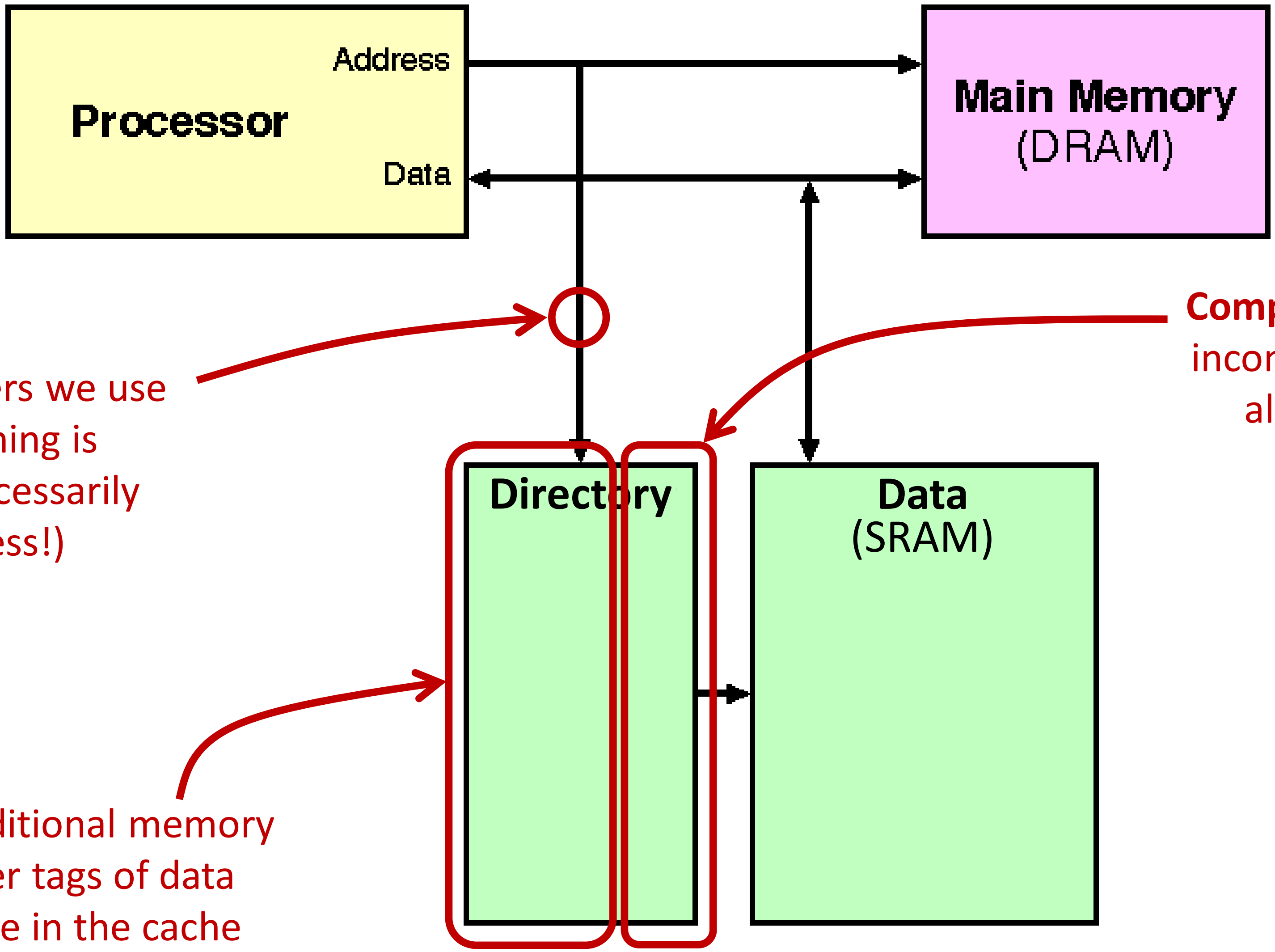
- **Extremely simple and fast**
  - If decisions are to be made in hardware, they need to be simple
  - Goal is to access fast memory in the order of a ns or less: **not much time** to make complex decisions…

Cache: The Idea



Needs data at addresses:
1000
1001
3000
1002
3000
1002
3000
1002

Not Just Fast Memory: Directory and Tags

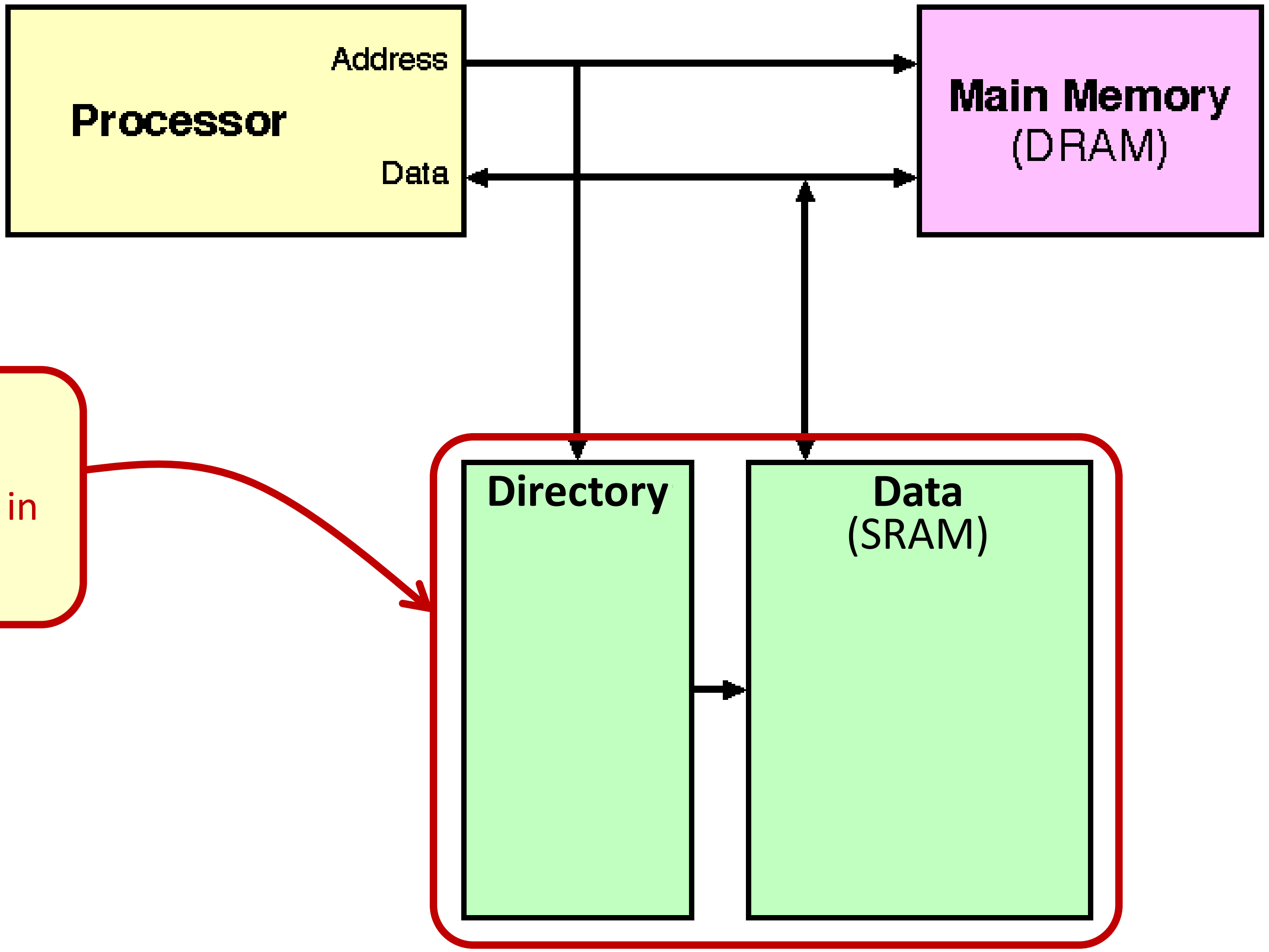


Tags: Minimal identifiers we use to check if something is in the cache (not necessarily the whole address!)

Comparators to check incoming tags against all those stored

Directory: Additional memory to remember tags of data that we have in the cache

A Cache!



Cache:
"a collection of items in an **hidden** place"

Cache Hits and Misses

- A **cache** is any form of storage which takes **automatically** advantage of locality of accesses
  - The idea works so well that now they are **not only in processors**!
  - Web browsers have caches, network routers have routing information and even data caches, DNSs cache frequent names, databases cache queries…

- When we find the data required in the cache, we call it a **Hit**; otherwise it is a **Miss**

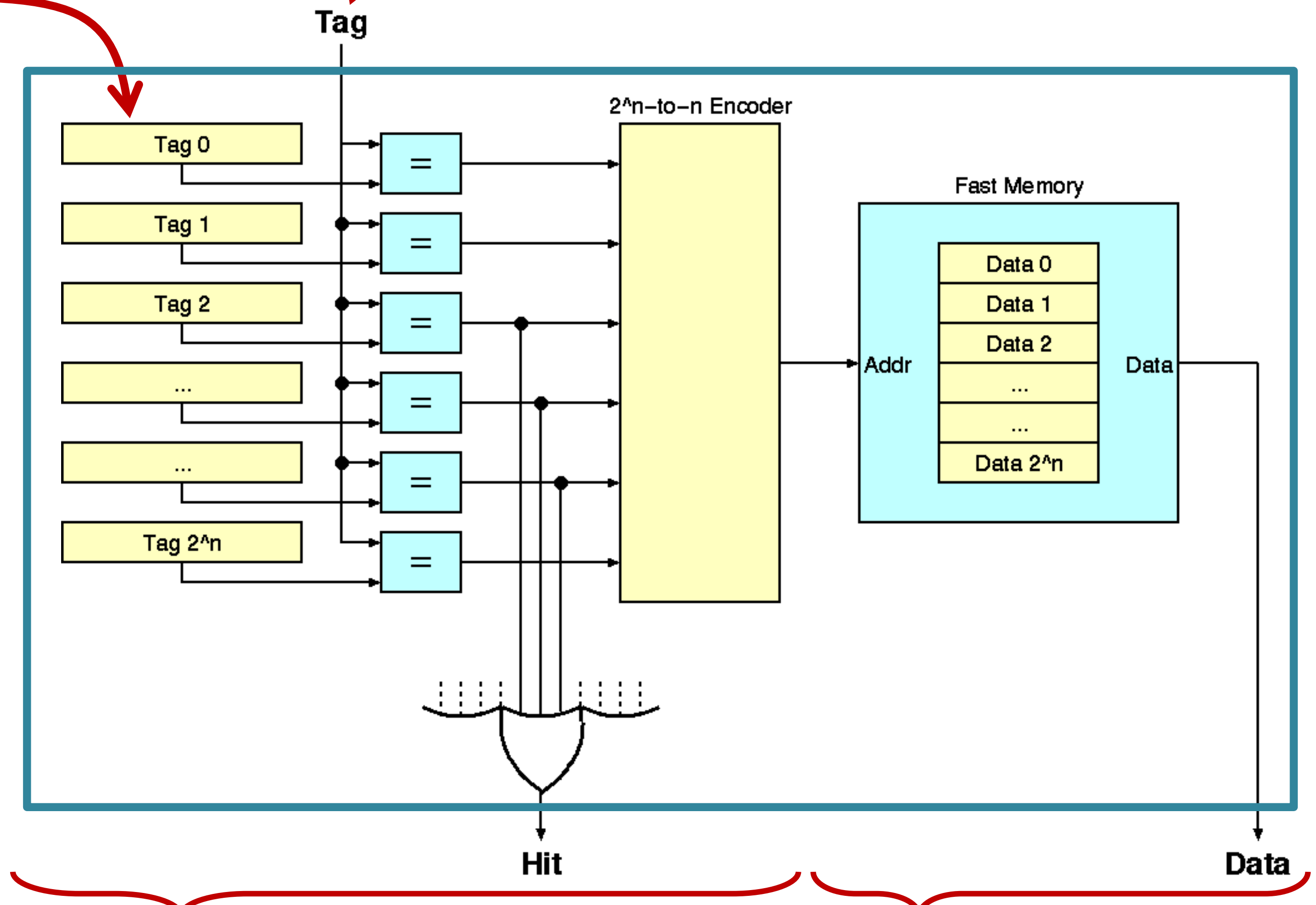
- **Hit (or Miss) Rate** is the number of hits (or misses) over the total number of accesses

Fully-Associative Cache

Registers

Request is not an "index" but "data" we search → "content addressable"

CAM
Content
Addressable
Memory

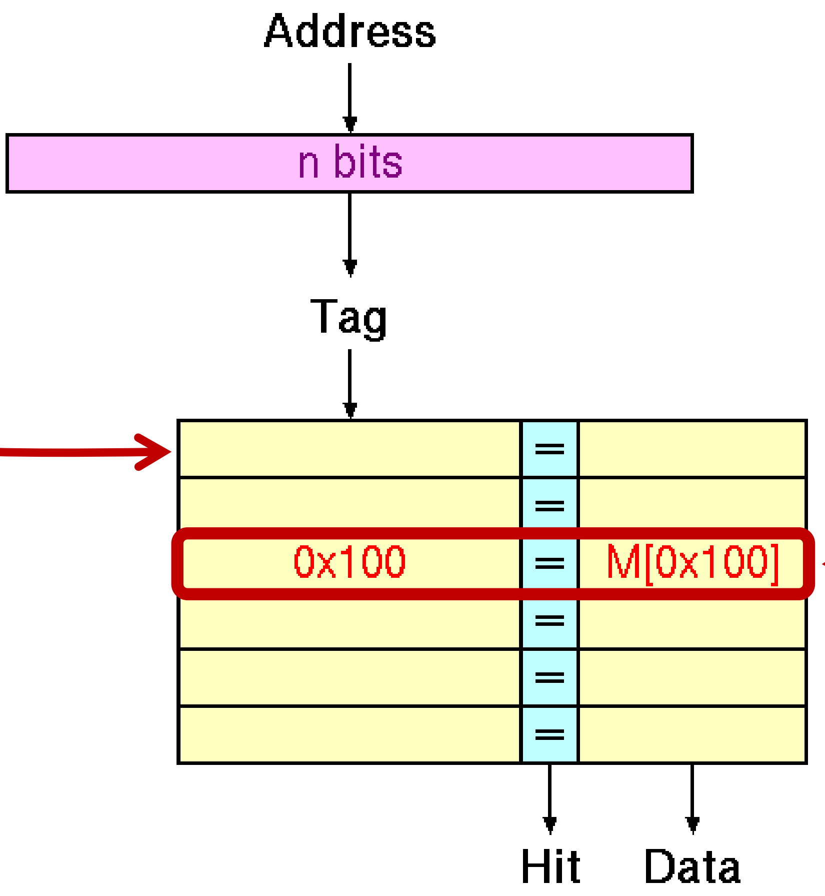


Directory

Memory (SRAM)

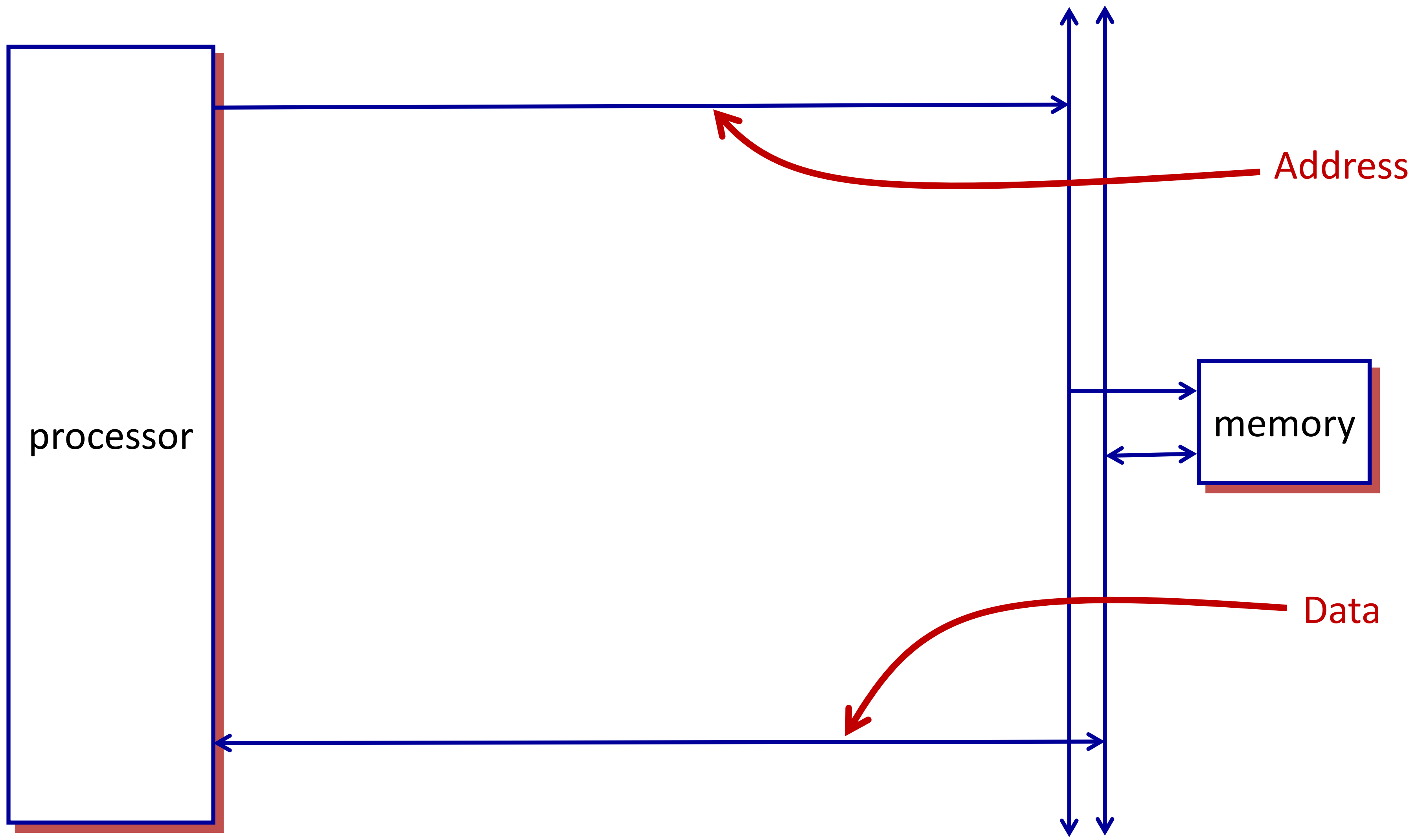
Fully-Associative Cache

The representation we will use

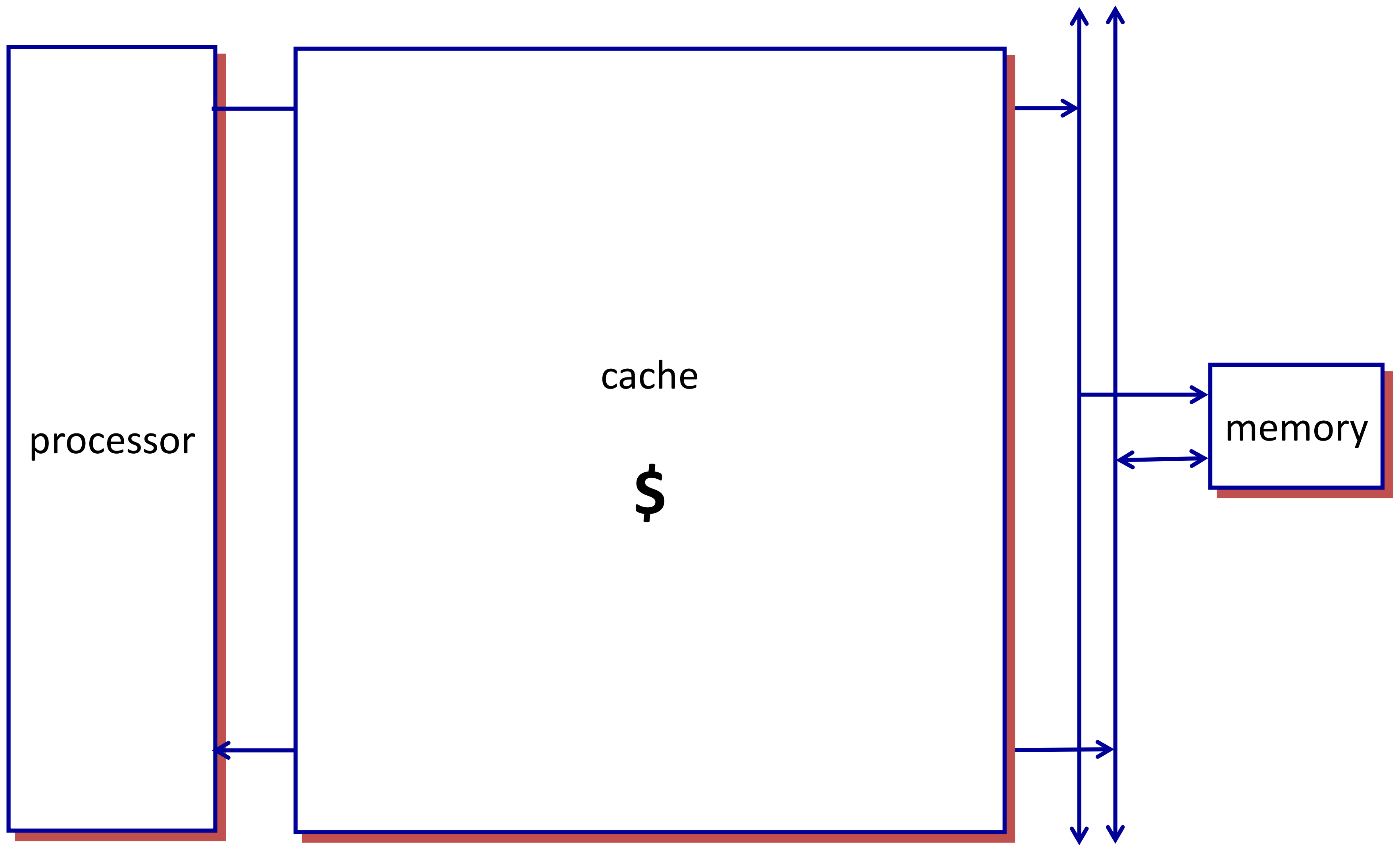


A **line** or a **block** of the cache

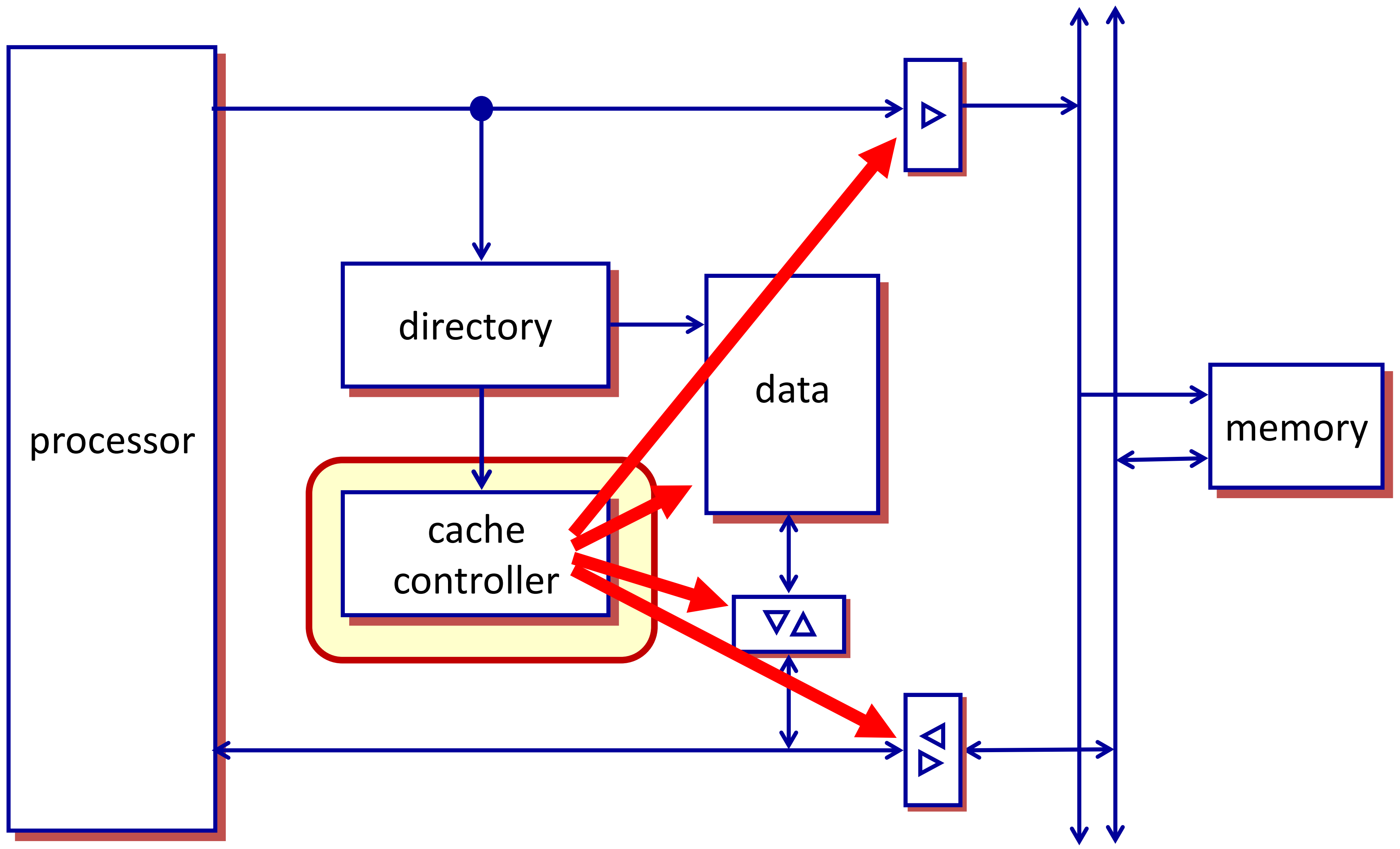
Cache and Cache Controller



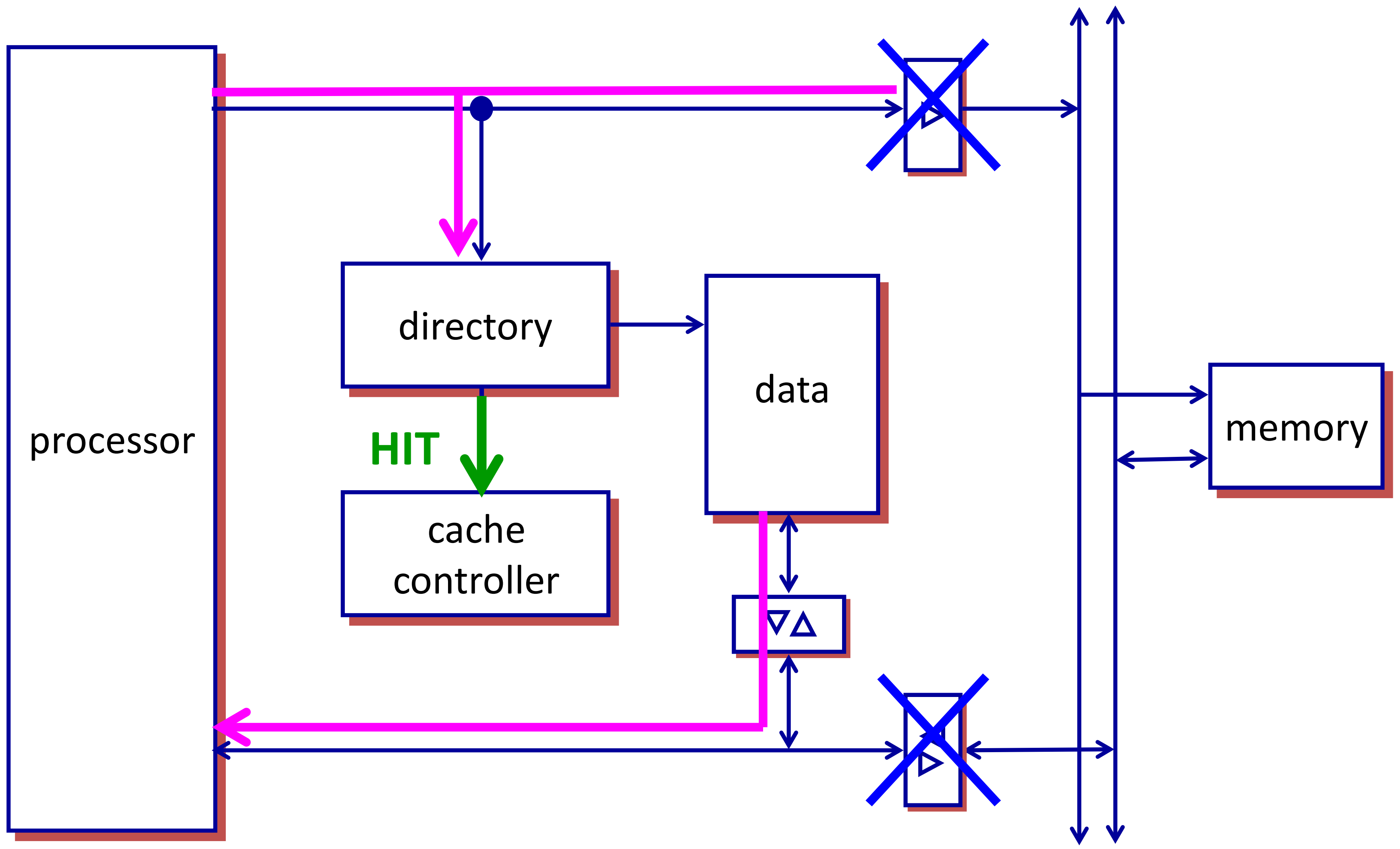
Cache and Cache Controller



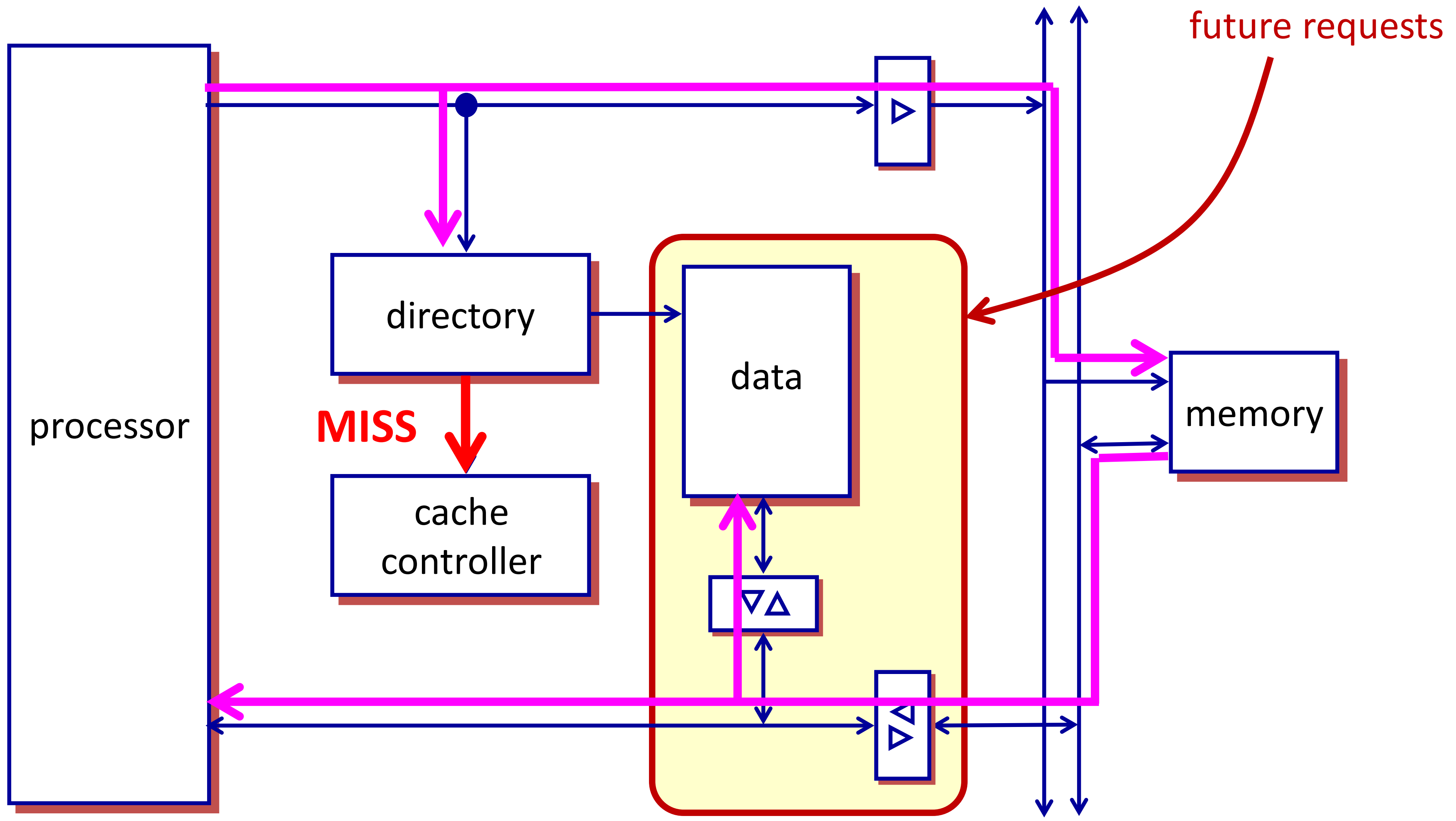
Cache and Cache Controller



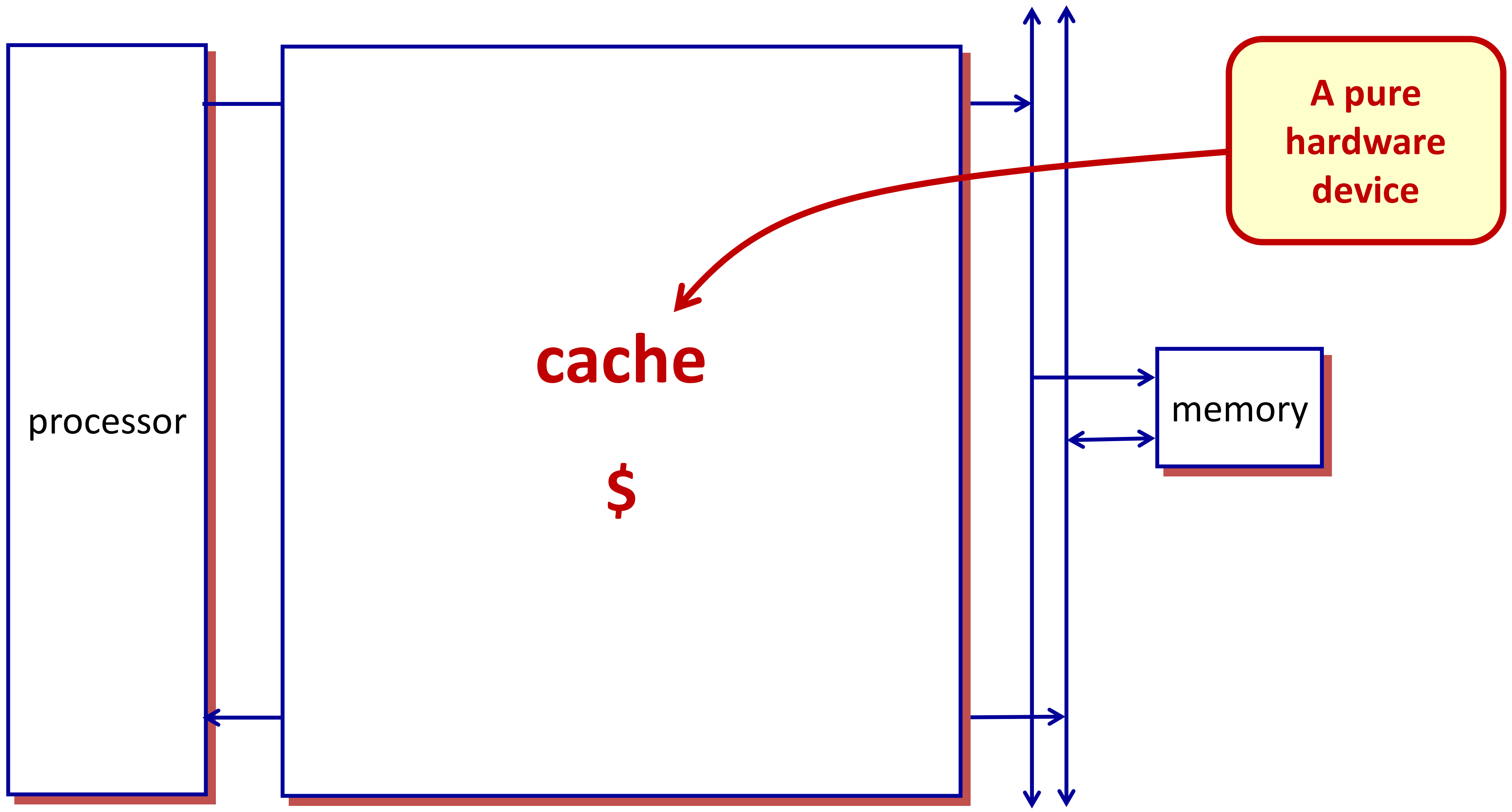
Cache Hit



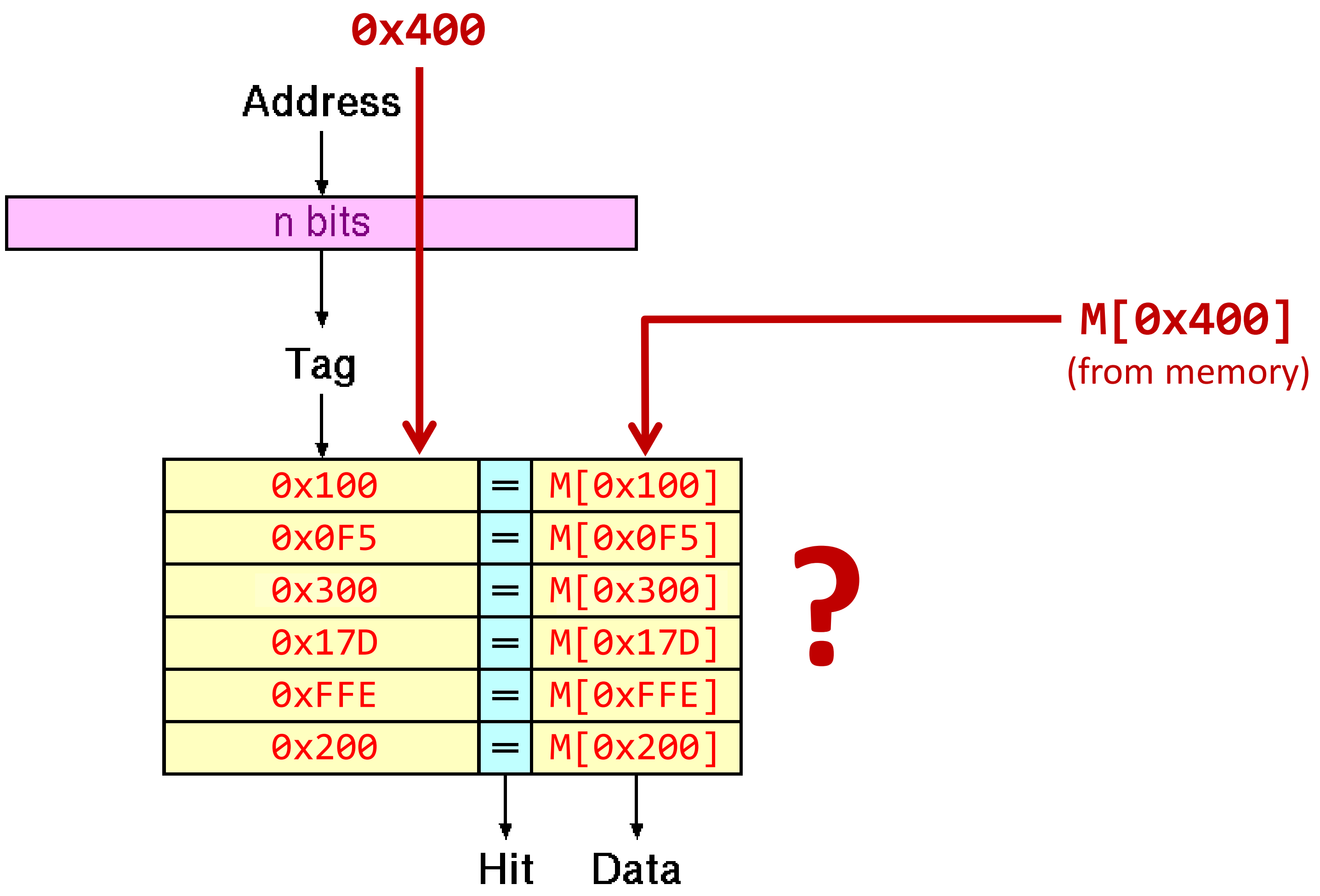
Cache Miss


Take note
of the data for
future requests
processor
directory
MISS
cache
controller
data
memory

Cache and Cache Controller



What If the Cache Is Full?

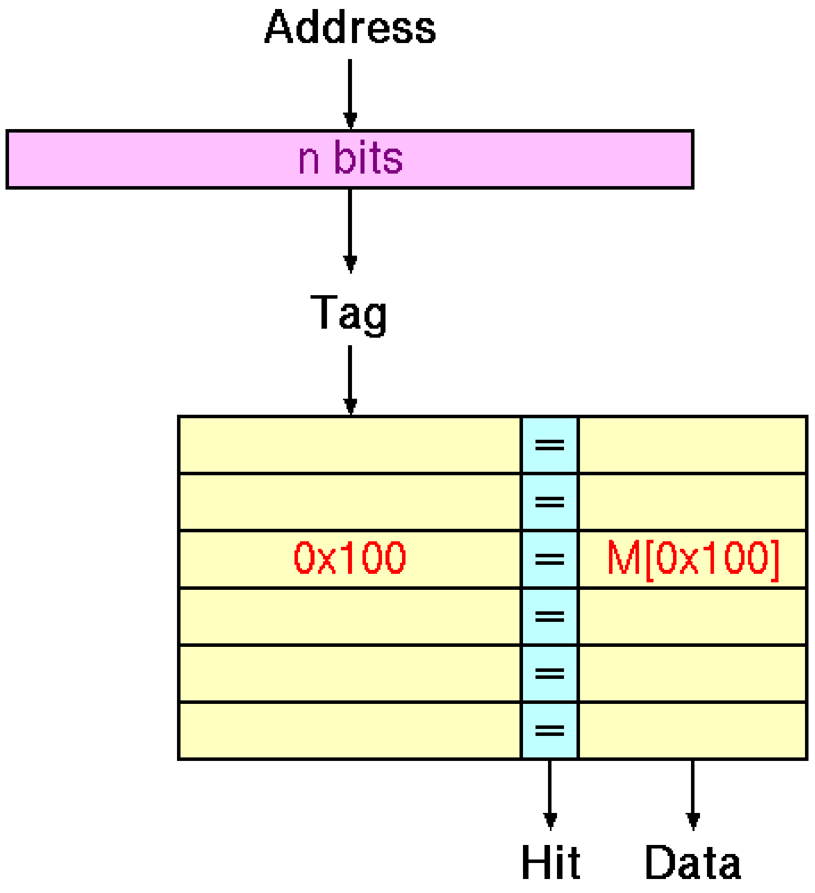


Eviction Policies

- When there is no appropriate space for a new piece of data, we must overwrite one of the existing lines (**eviction** or **replacement**)

- Several policies to decide what to evict:
  - **Least Recently Used** (LRU)
    - Replace the data that have been unused for the longest period of time
  - **First-In First-Out** (FIFO)
    - Replace the data that came in earliest
  - **Random**
    - Pick one, any one, and throw it away…
  - Approximate schemes, etc…

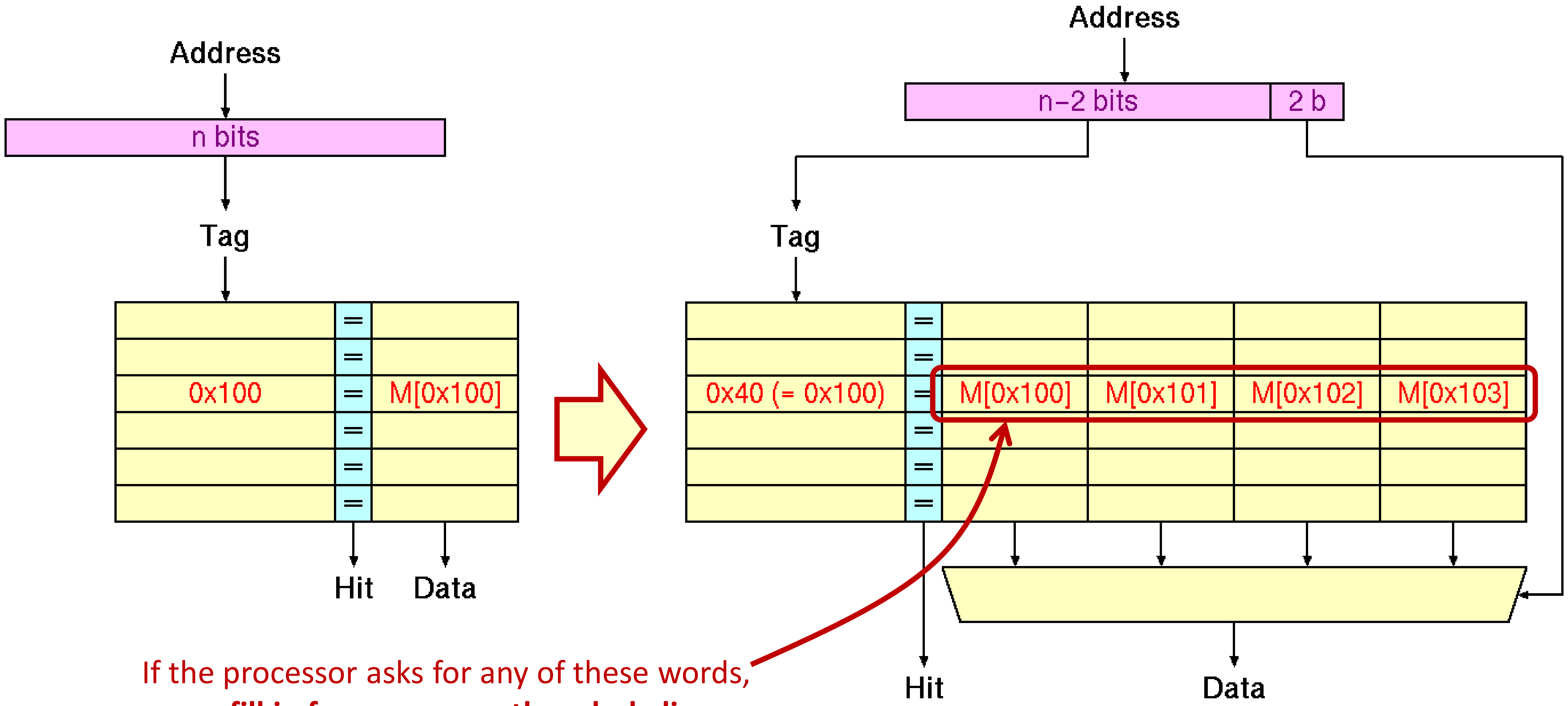
Only Exploiting Temporal Locality



We only bring from main memory to the cache **exclusively the data required and when required**, hoping to use them again

If the processor asks for neighbouring items (e.g., M[0x101]), we will not have it; we are **not able to exploit any spatial locality**

Exploiting Spatial Locality

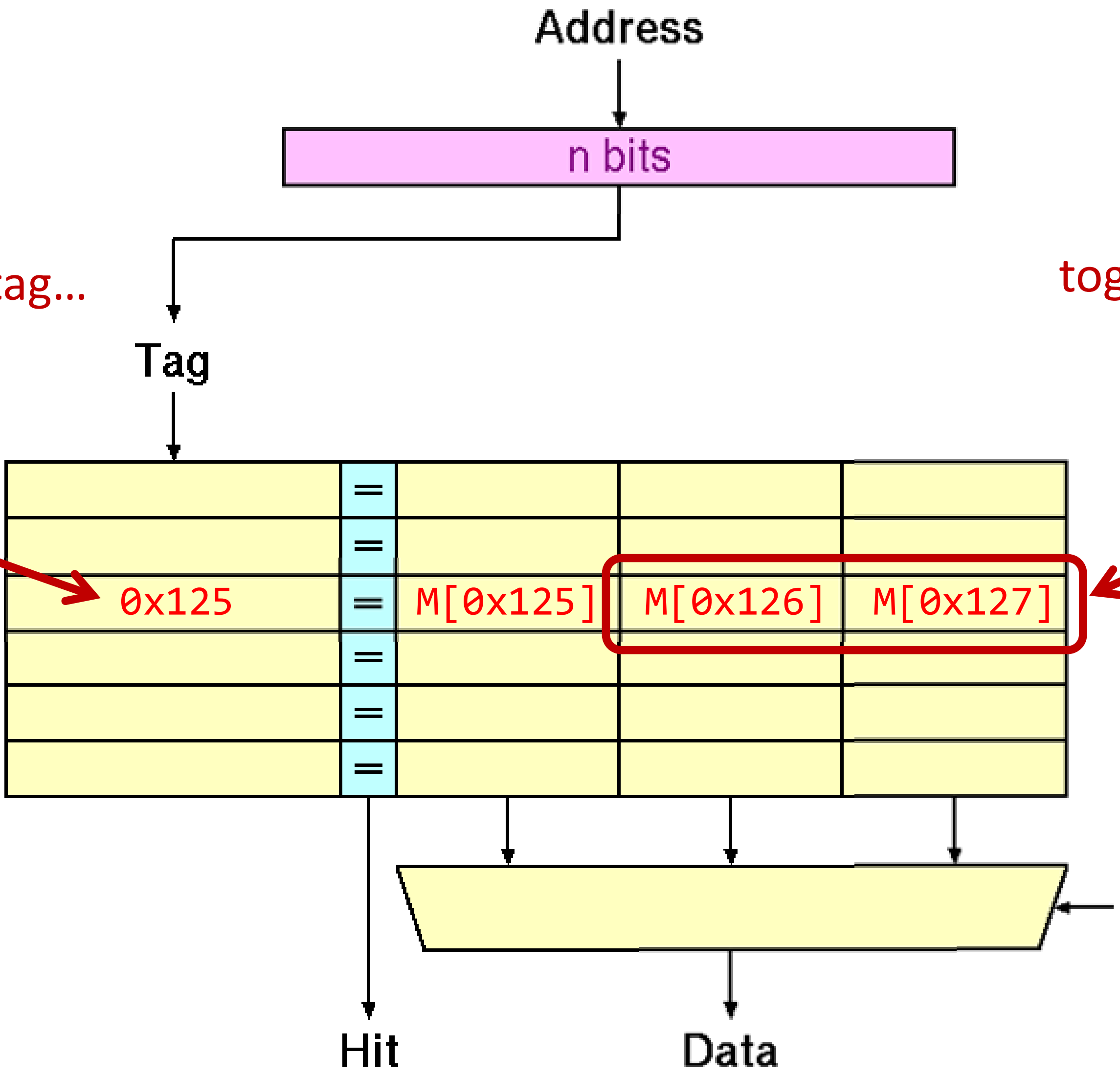


If the processor asks for any of these words,
we fill in from memory the whole line

Why Not This?!

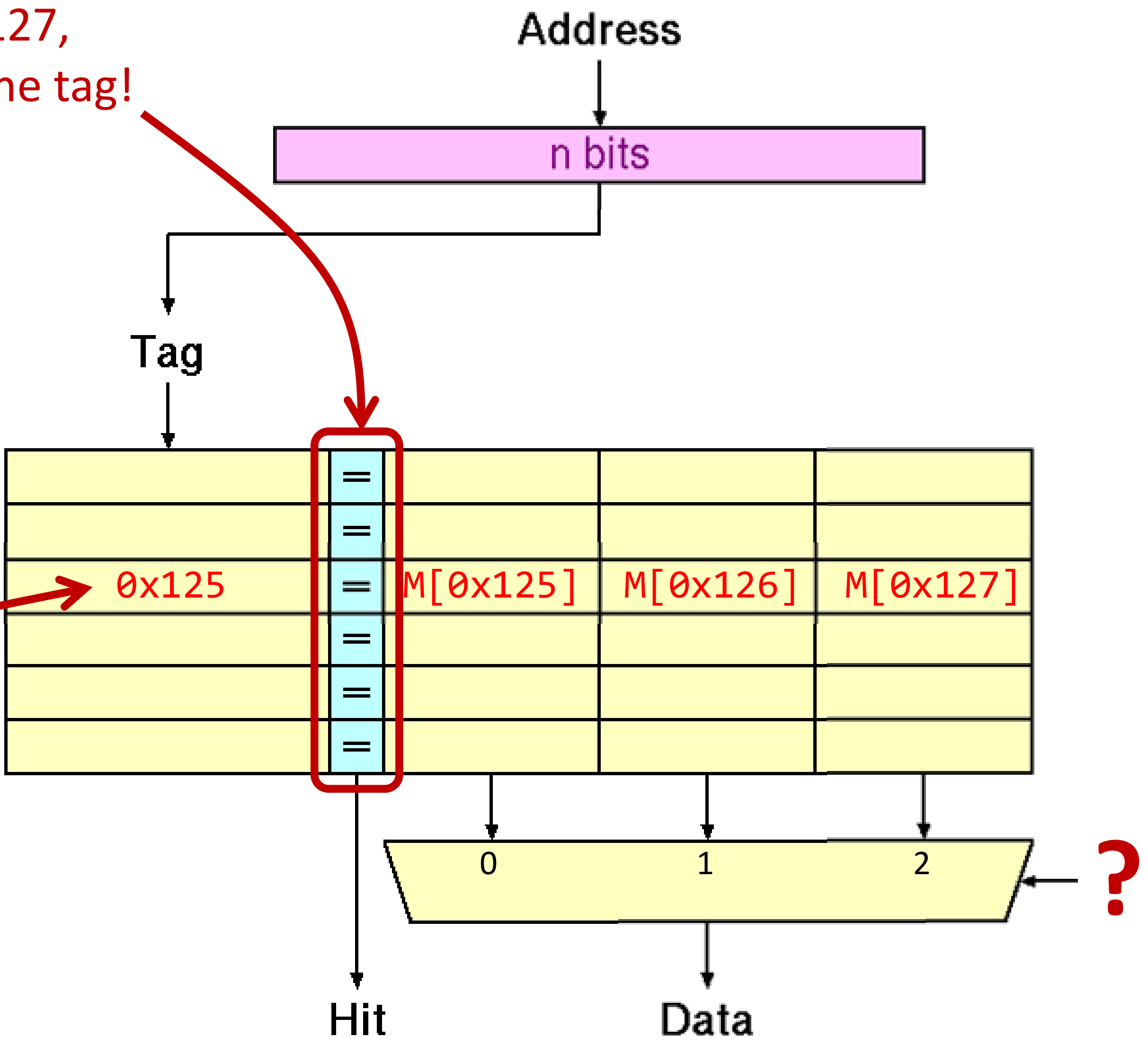
If we are asked 0x125,
we store the address as a tag…

…and bring the value in
together with the next 1, 2, 3, 4,…



Why Not This?!

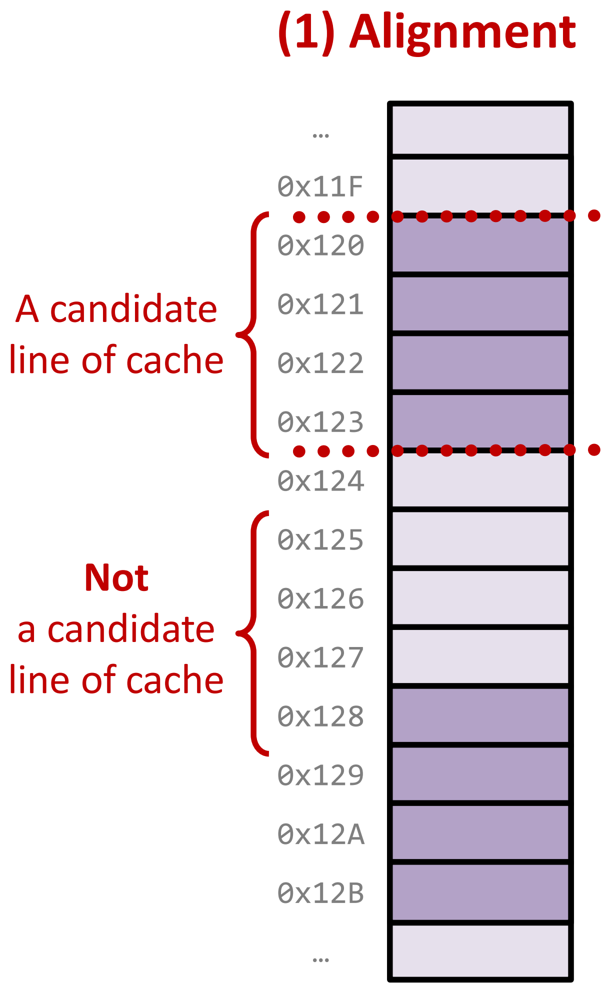
If now we are asked 0x127,
we cannot compare with the tag!

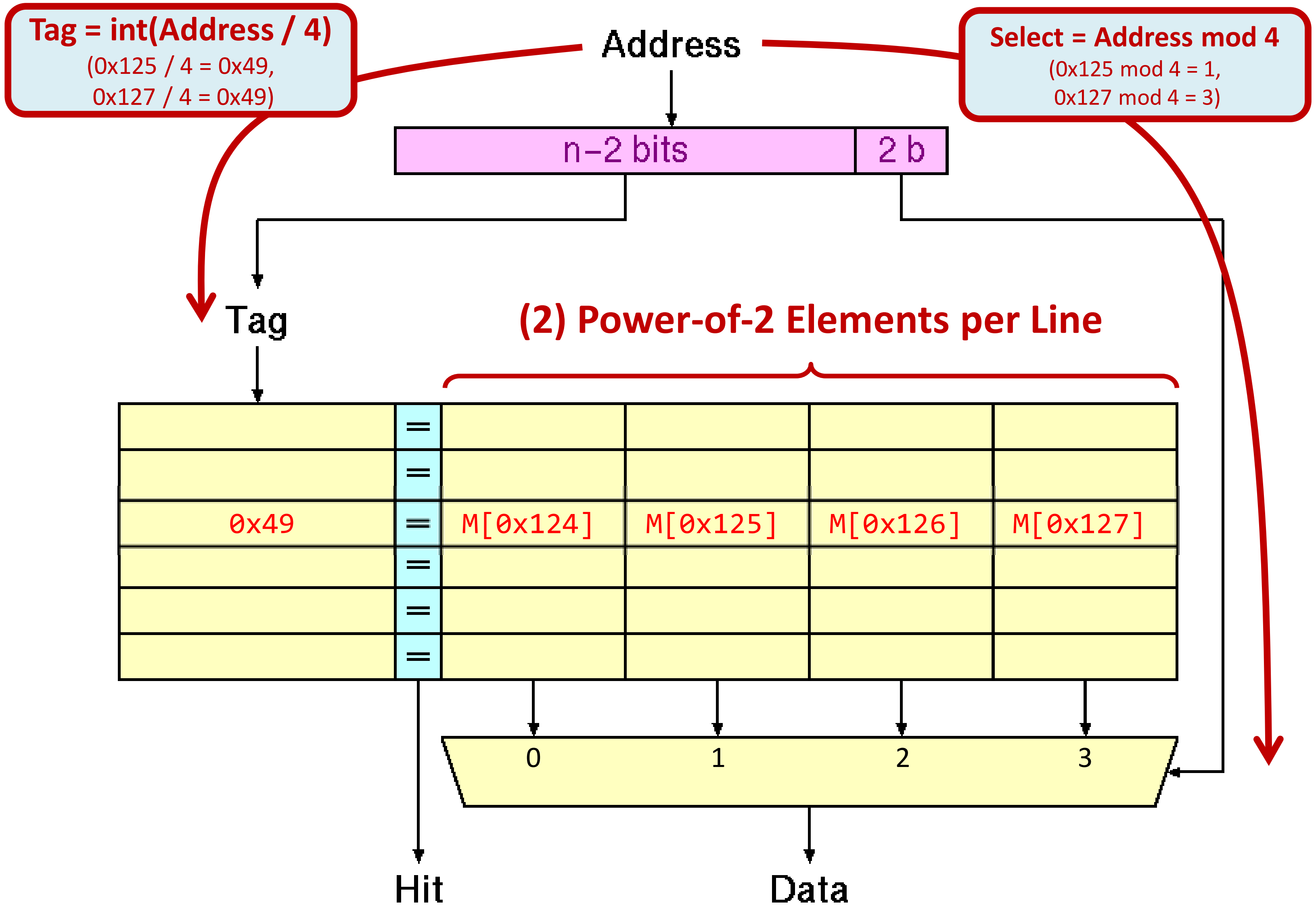


We select the right one with **Address – Tag**
(0x127 – 0x125 = 2)

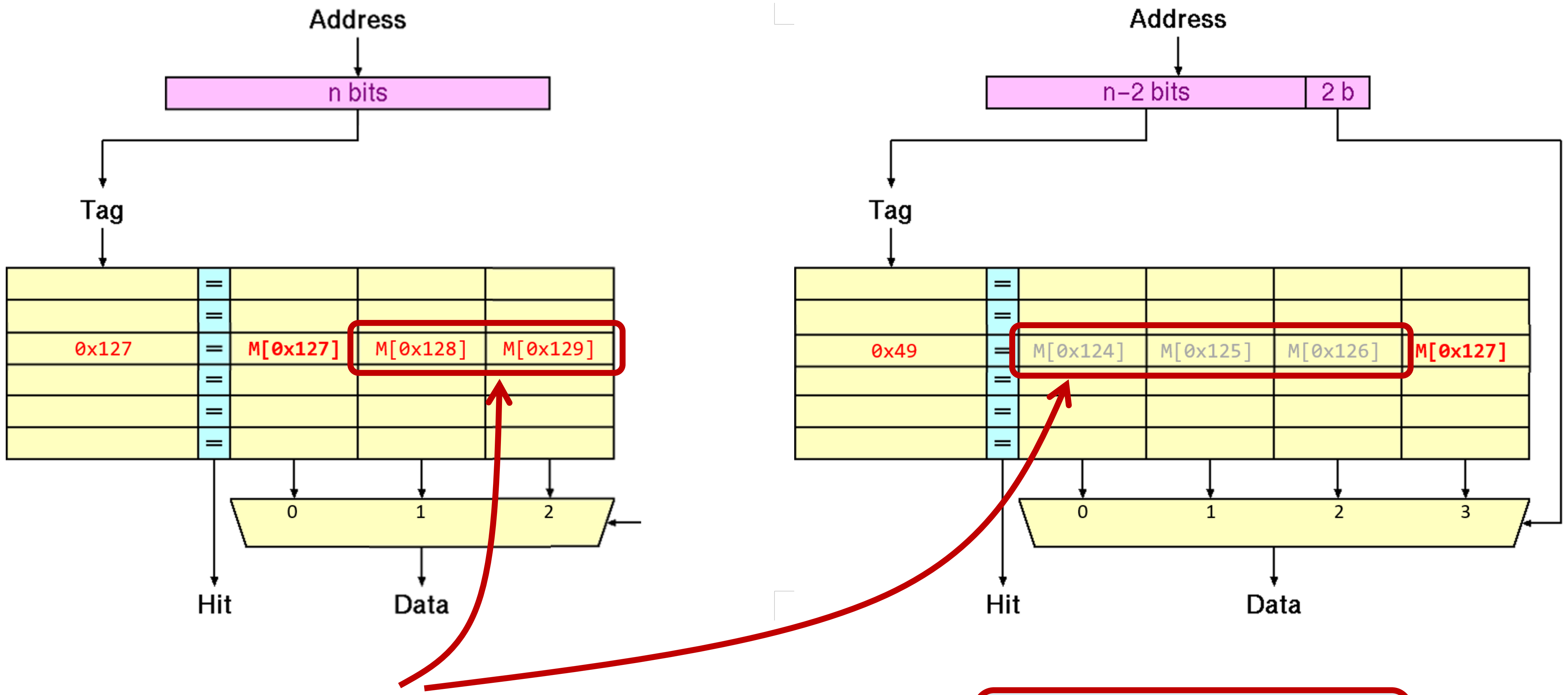
We need to check if
Tag ≤ Address < Tag + 3
(0x125 ≤ 0x127 < 0x128)

This Is Much More Hardware Friendly!





Hardware Friendliness Is Essential



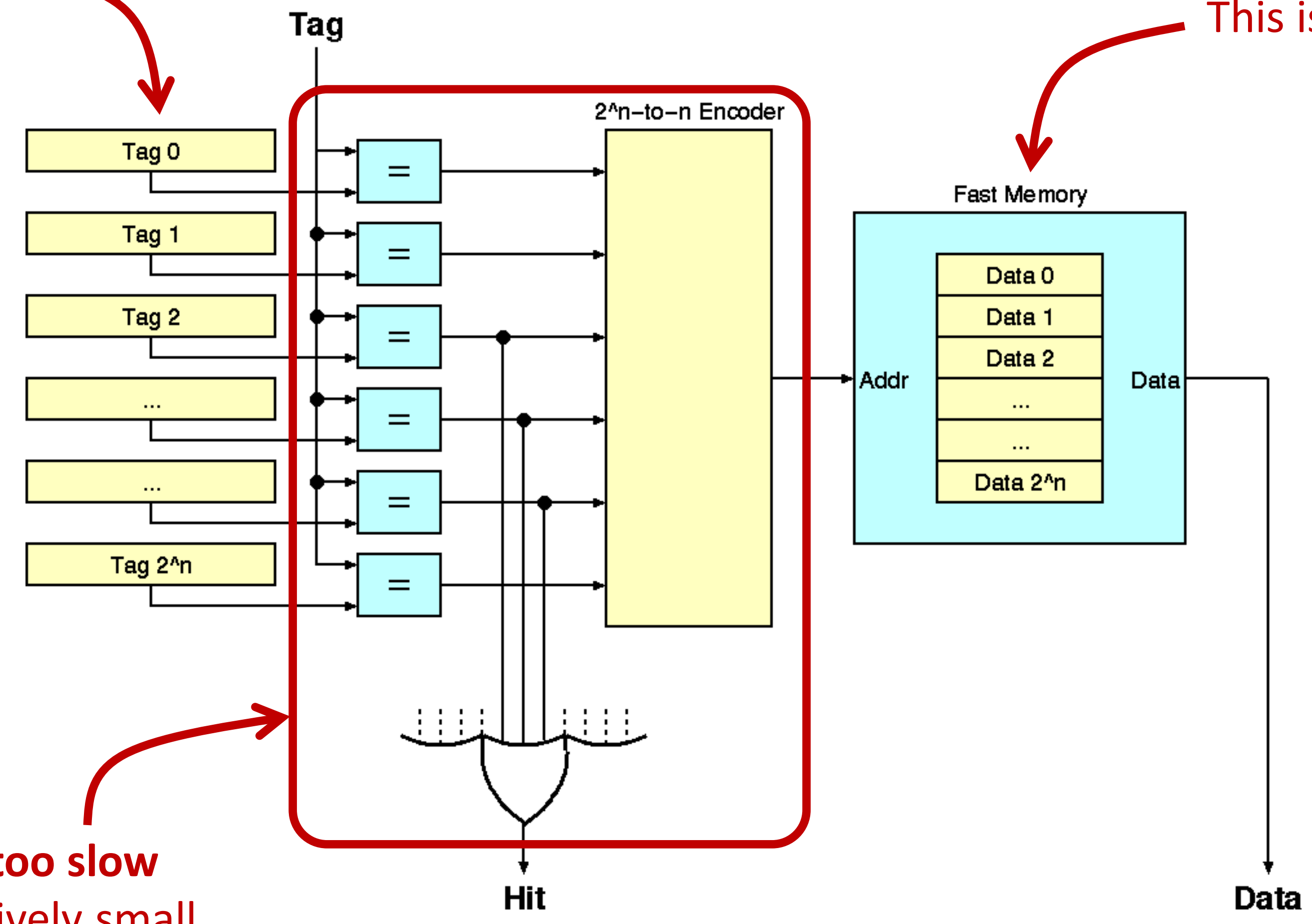
This cache may be better, because if the processor asks 0x127 we bring in potentially **useful** stuff instead of **stale** stuff…

…yet, this cache is **FEASIBLE!**

Fully-Associative Cache

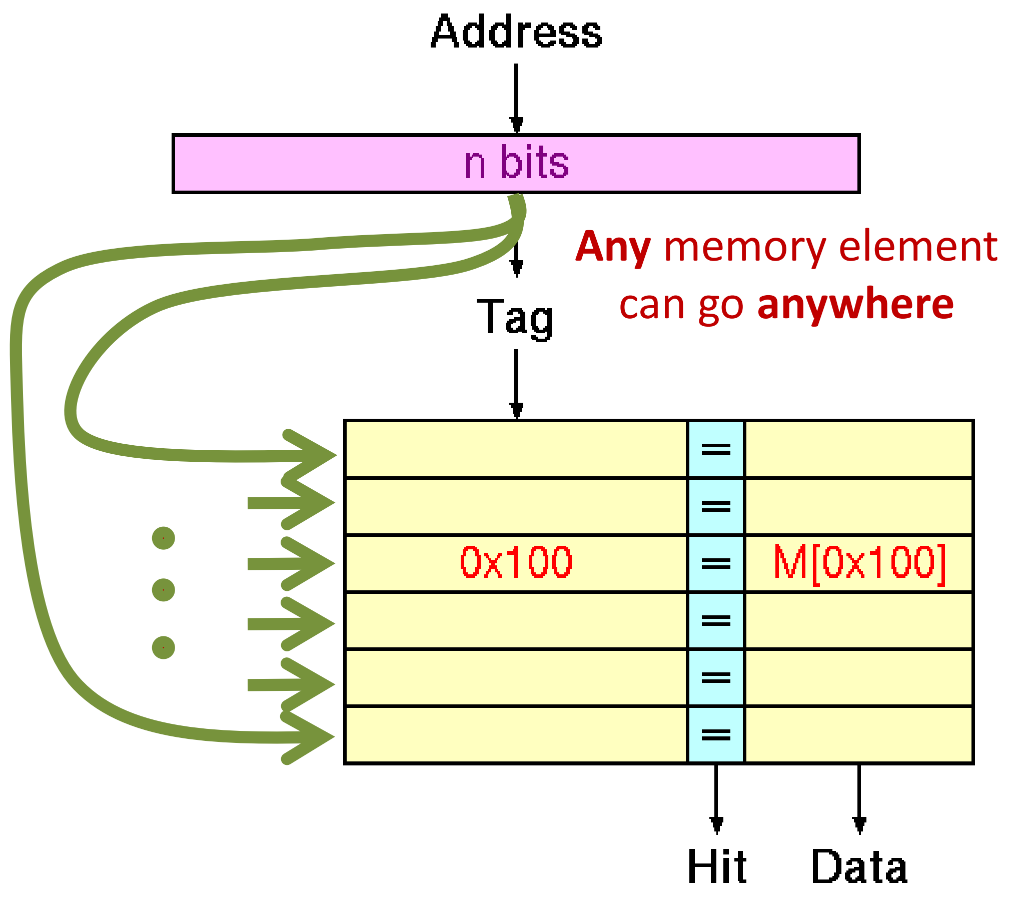
These are registers and relatively cheap

This is a RAM and is cheap



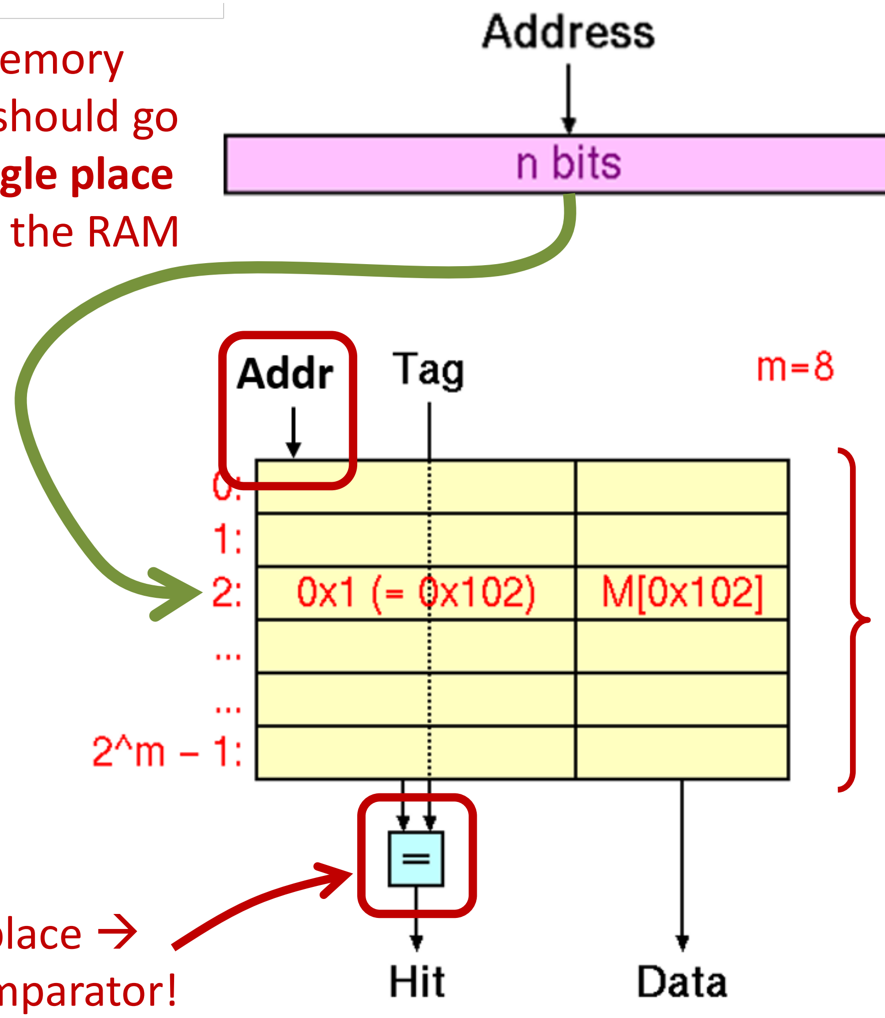
This is **too costly** and **too slow** unless the cache is relatively small

How Can We Make It Simpler?



Any memory element can go **anywhere**

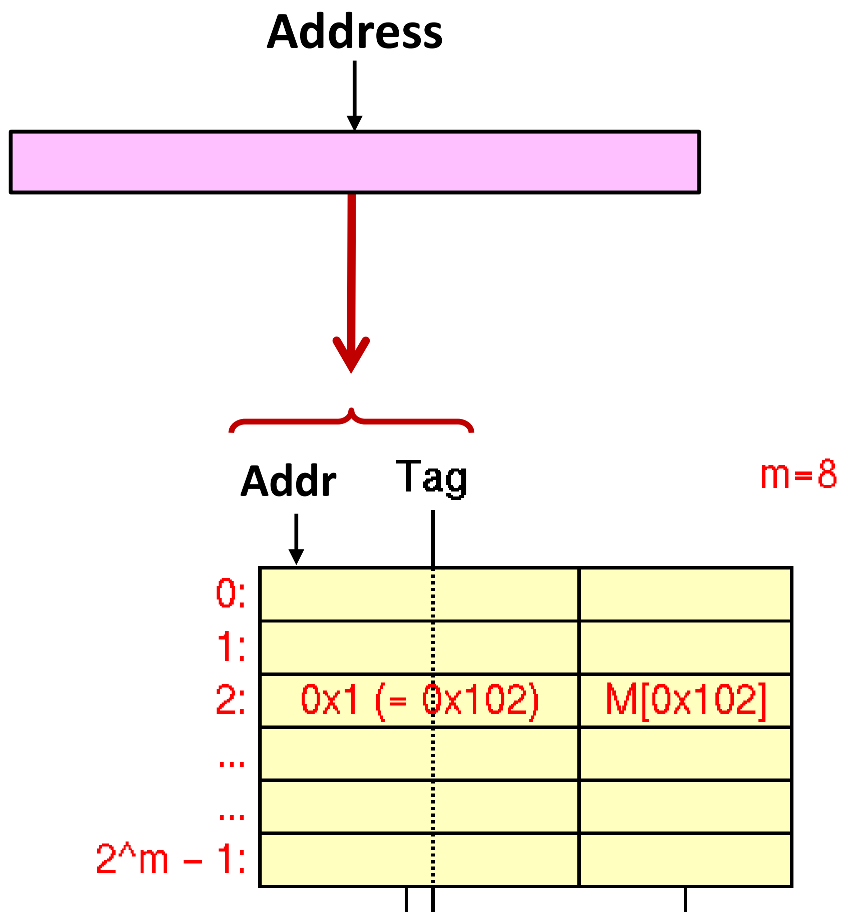
Any memory element should go into a **single place** = **Addr** of the RAM

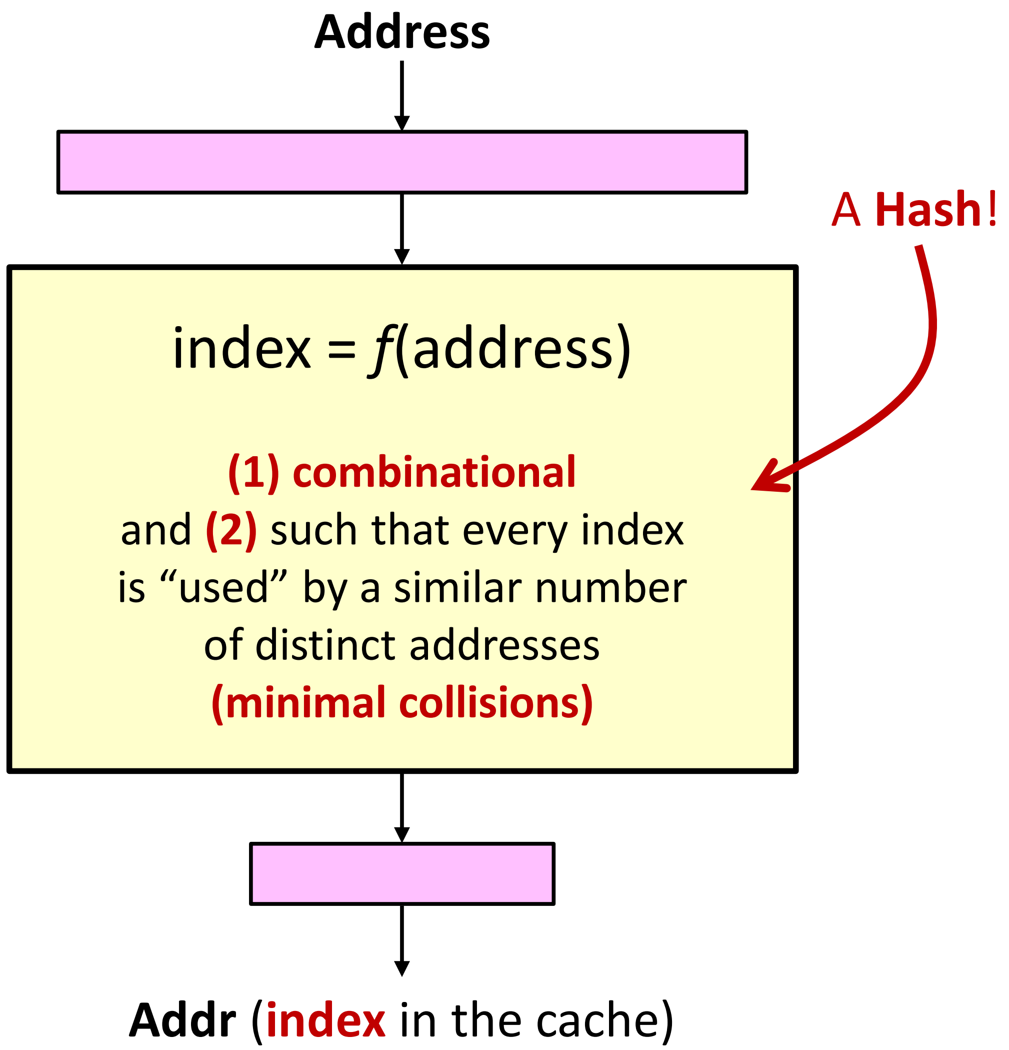


Very cheap **RAM**

A **single** place → a **single** comparator!

How to Generate Addr and Tag?



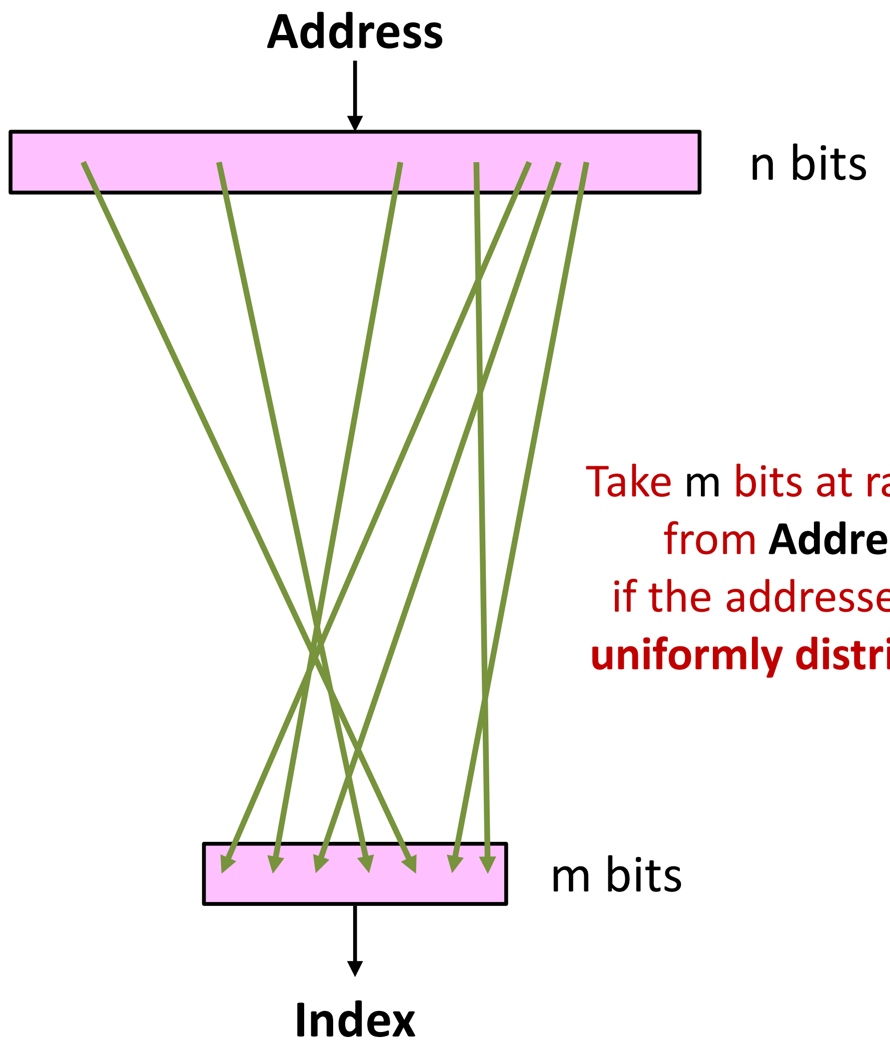


index = *f*(address)

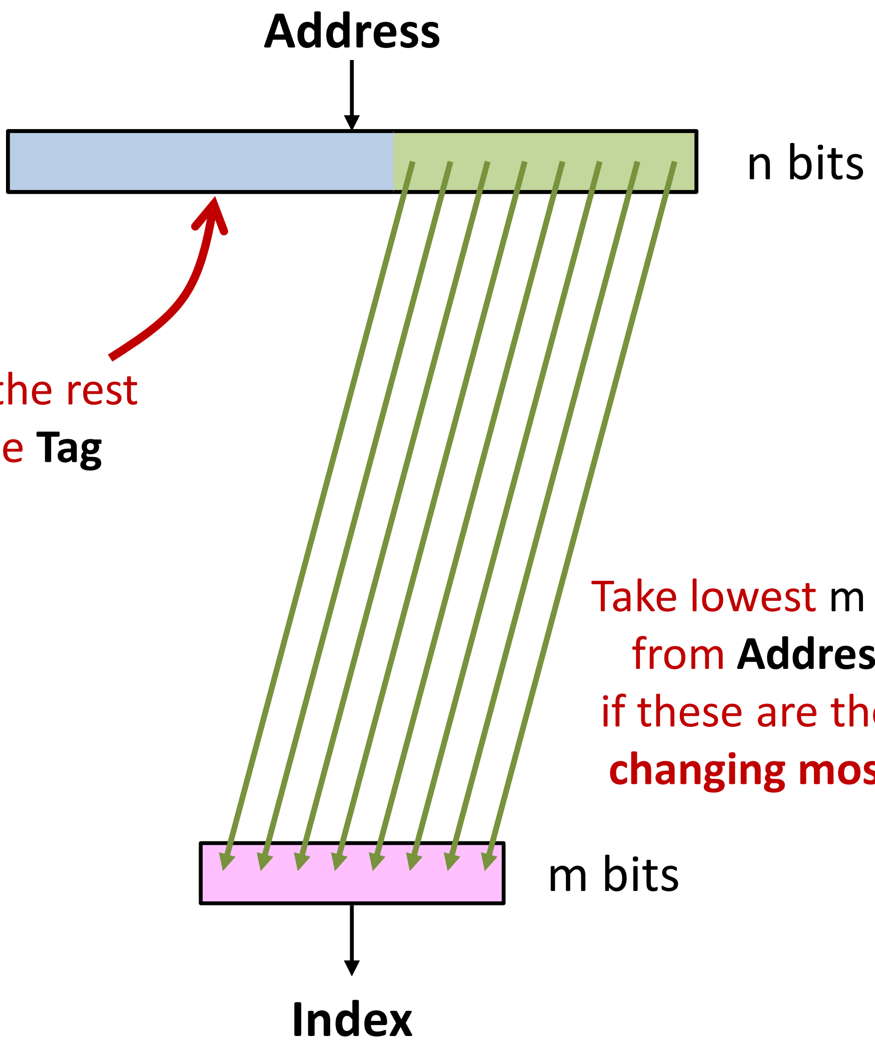
(1) combinational and **(2)** such that every index is “used” by a similar number of distinct addresses **(minimal collisions)**

Addr (**index** in the cache)

The Simplest Hashes



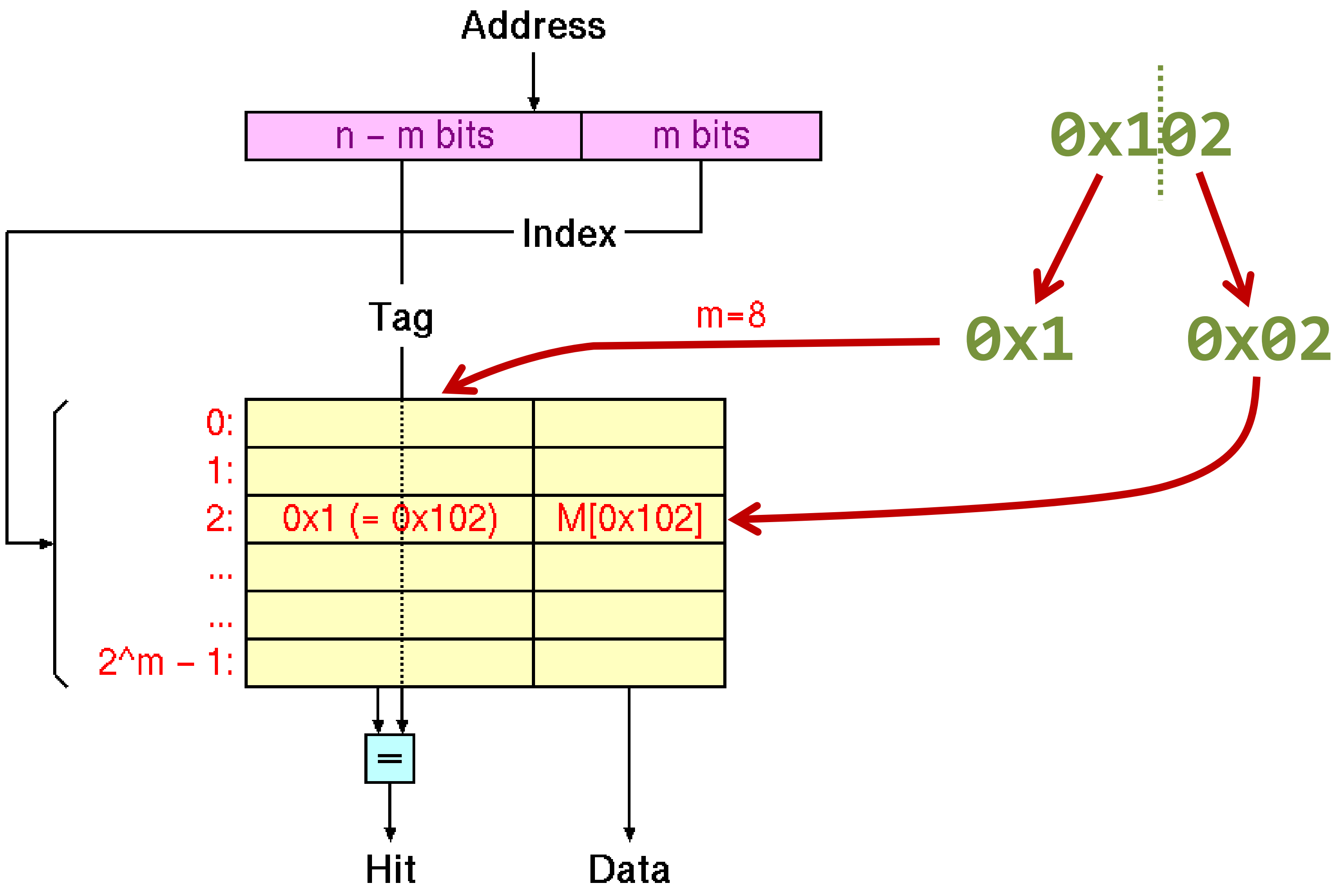
Take m bits at random from **Address**, if the addresses are **uniformly distributed**



…and the rest is the **Tag**

Take lowest m bits from **Address**, if these are those **changing most**…

Direct-Mapped Cache

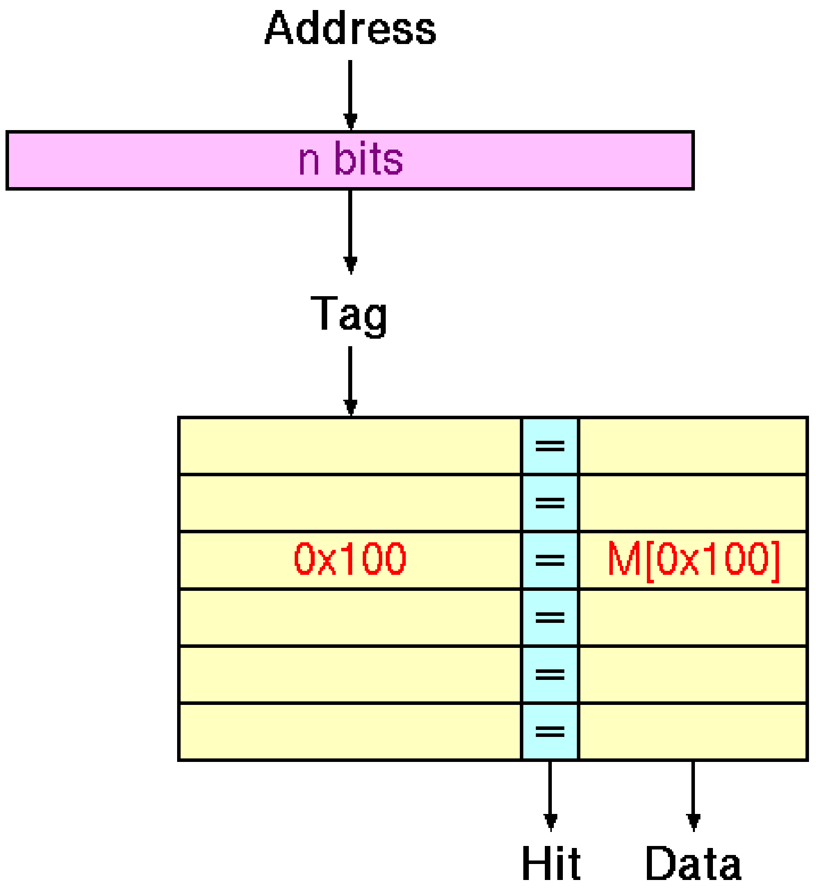


Much cheaper!

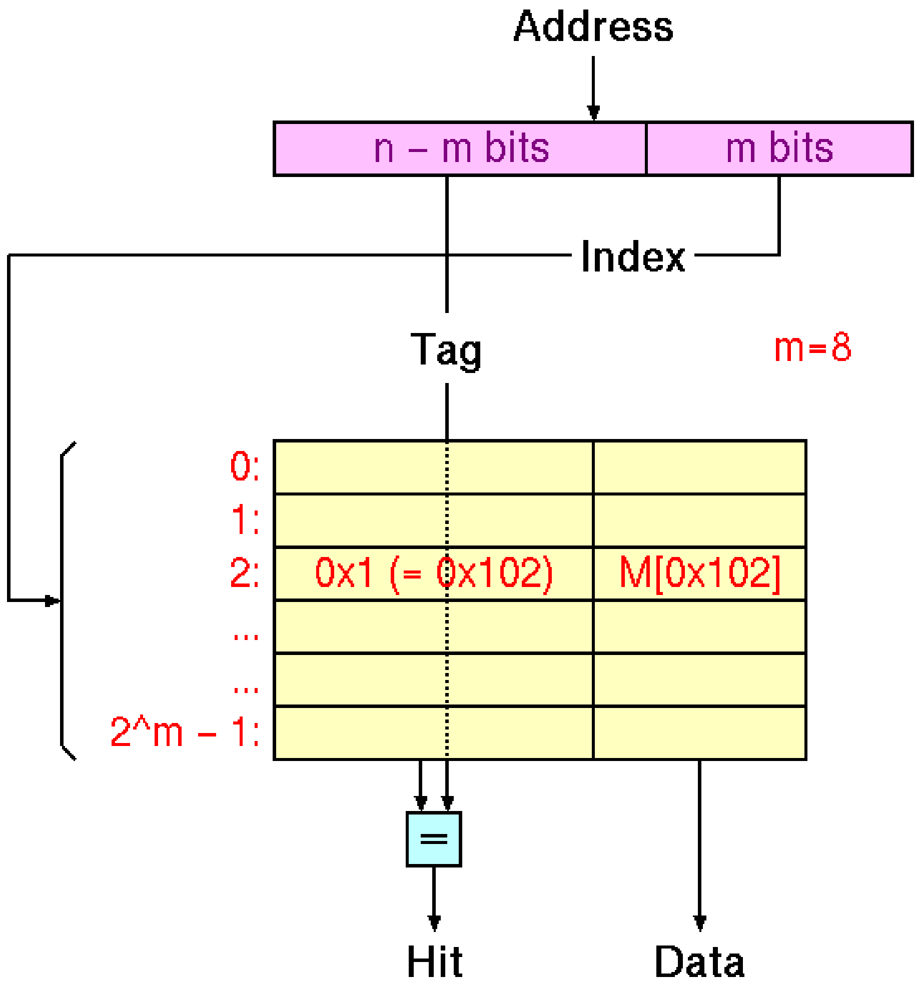
…but does it have any drawbacks?!

Which One Is the Best Cache?

Fully Associative Cache

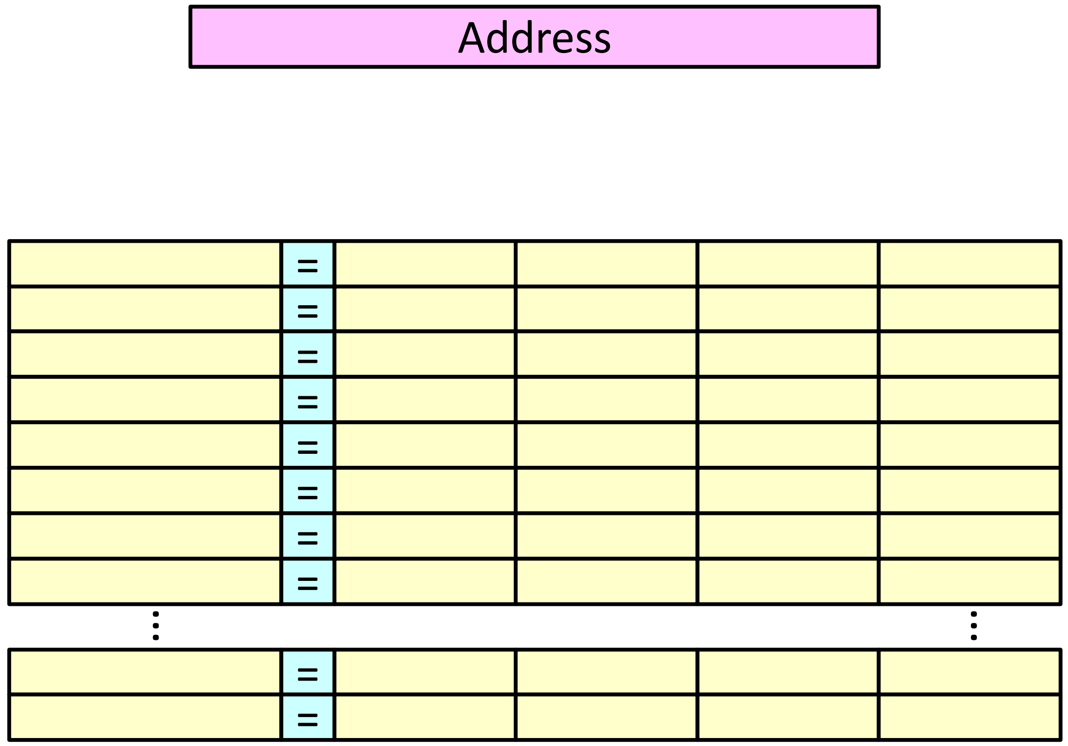


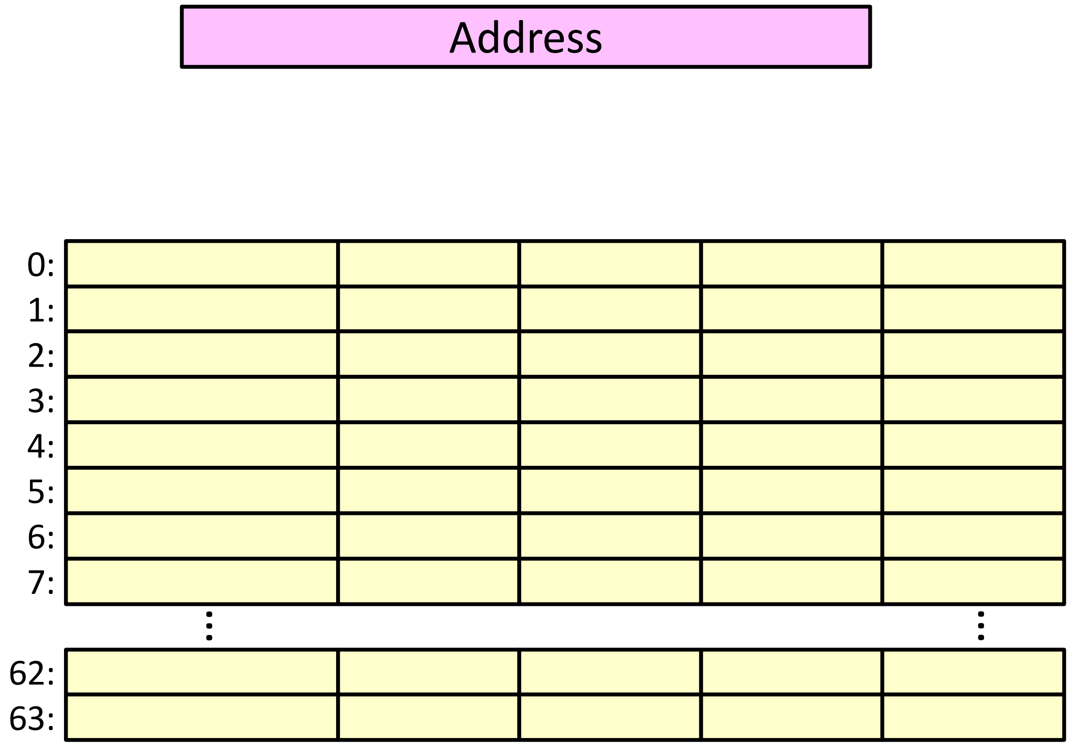
Direct-Mapped Cache



Which One Is the Best Cache?

- Consider a **fully associative** and **direct-mapped** cache, both with **64 lines** with **four words per line** (→ 256 words per cache)
- Suppose accesses at 0x100, 0x101, 0x200, 0x102, 0x300, 0x103, 0x201, 0x102, 0x301, 0x103,…
- What is the **Hit Rate** of each of these two caches?





	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103
Fully Ass.										
Direct Mapp.										

Cache Pollution, or Conflict Misses

	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103
Fully Ass.	**M**	H	**M**	H	**M**	H	H	H	H	H
Direct Mapp.	**M**	H	**M**	**M**	**M**	**M**	**M**	**M**	**M**	**M**

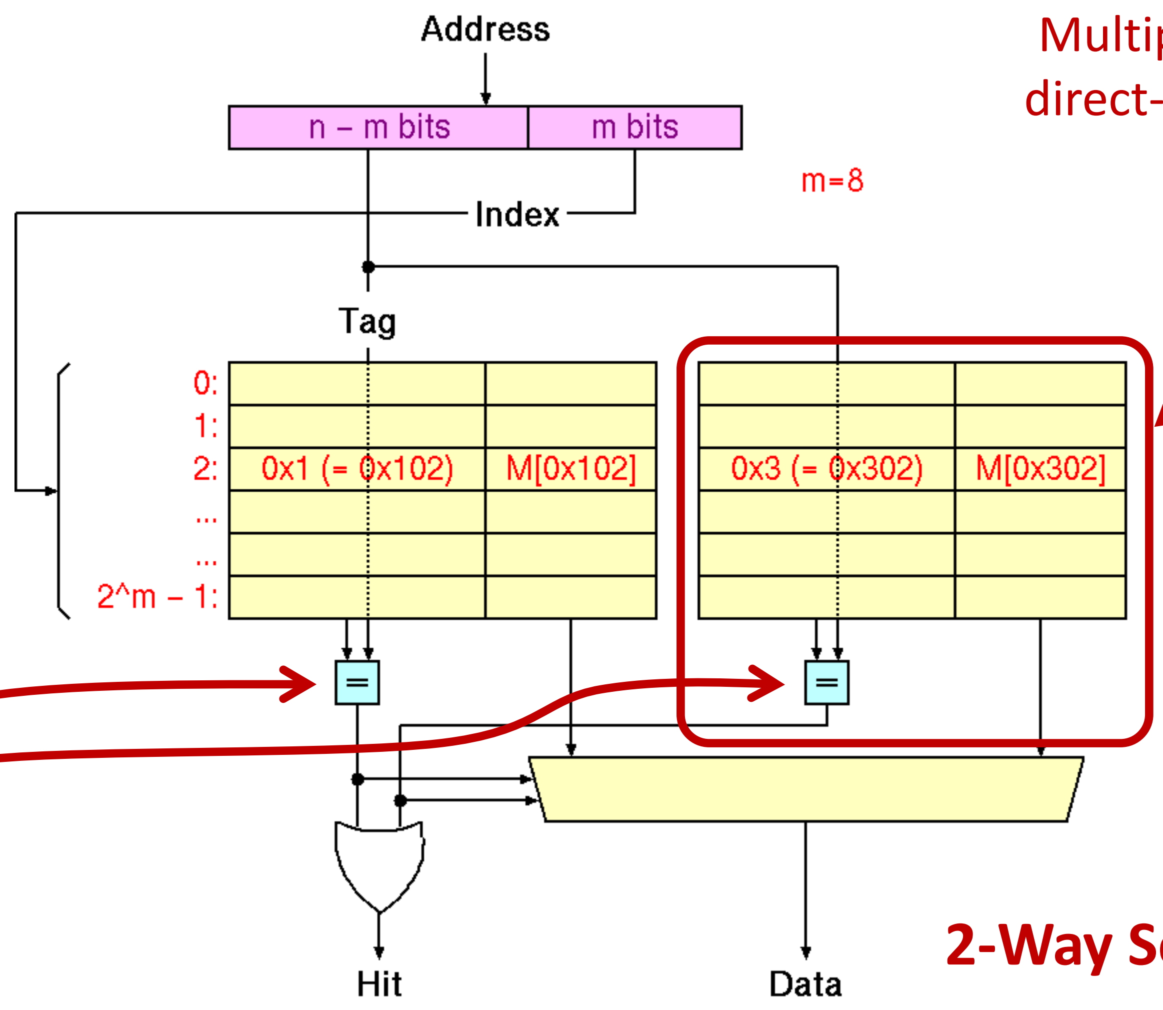
- Addresses 0x1…, 0x2…, and 0x3… use the same line of the direct-cache (they are said to **alias**) $\rightarrow$ **Cache pollution** or **conflict misses**
- Fully associative cache is **much more robust** than direct mapped

Associativity

- The probability of aliasing is related to the associativity of caches
- **Associativity** indicates the number of different positions in a cache where one element of data can be placed:
  - **Fully-associative**: every word can go in every line of the cache (hence the "full") → associativity is the number of lines in the cache
  - **Direct-mapped**: every word is "mapped" to a single line of the cache → associativity is 1

Can we think
of any **intermediate** possibility?

Set-Associative Cache

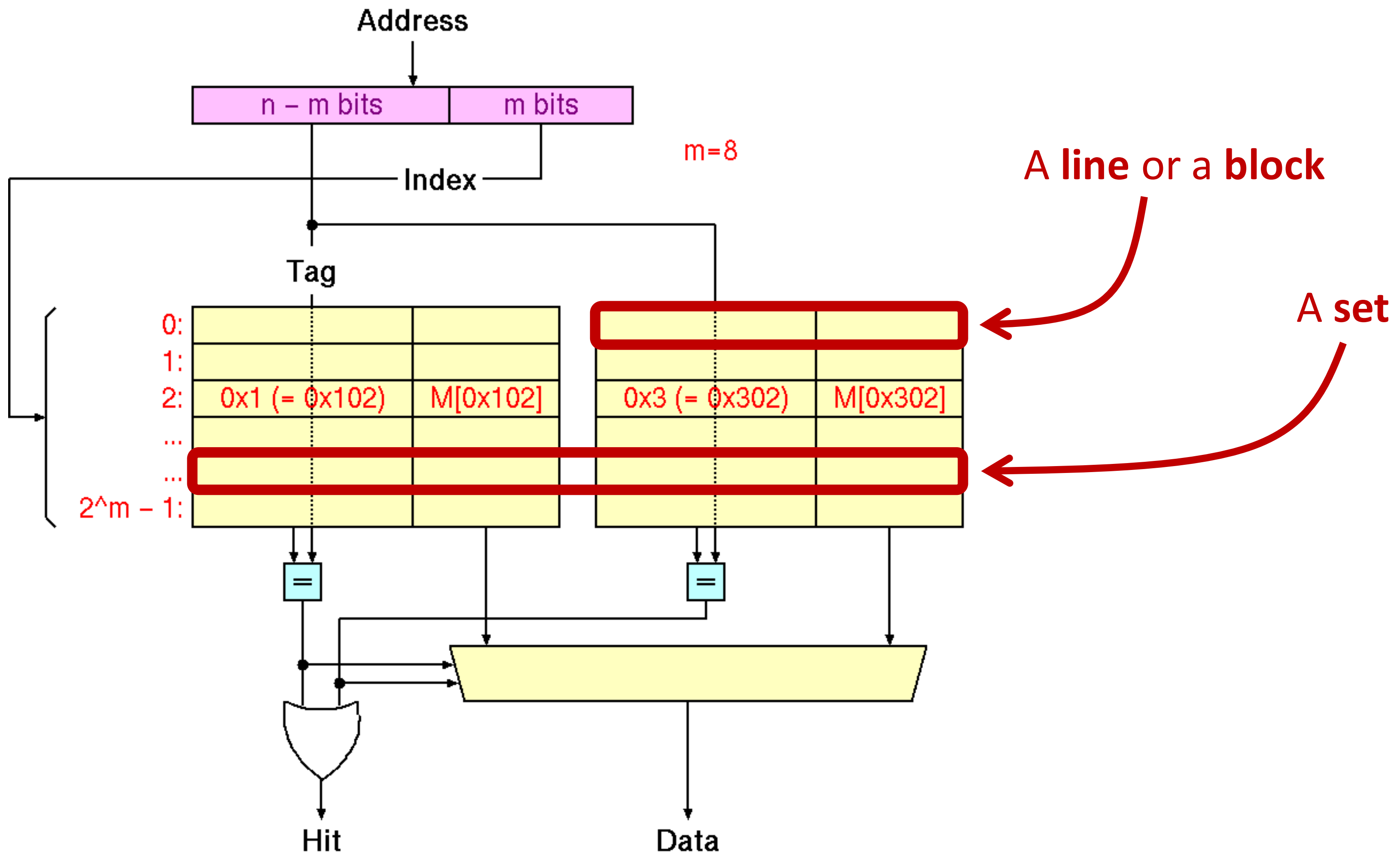


Multiple replicas of the direct-mapped structure → a **Way**

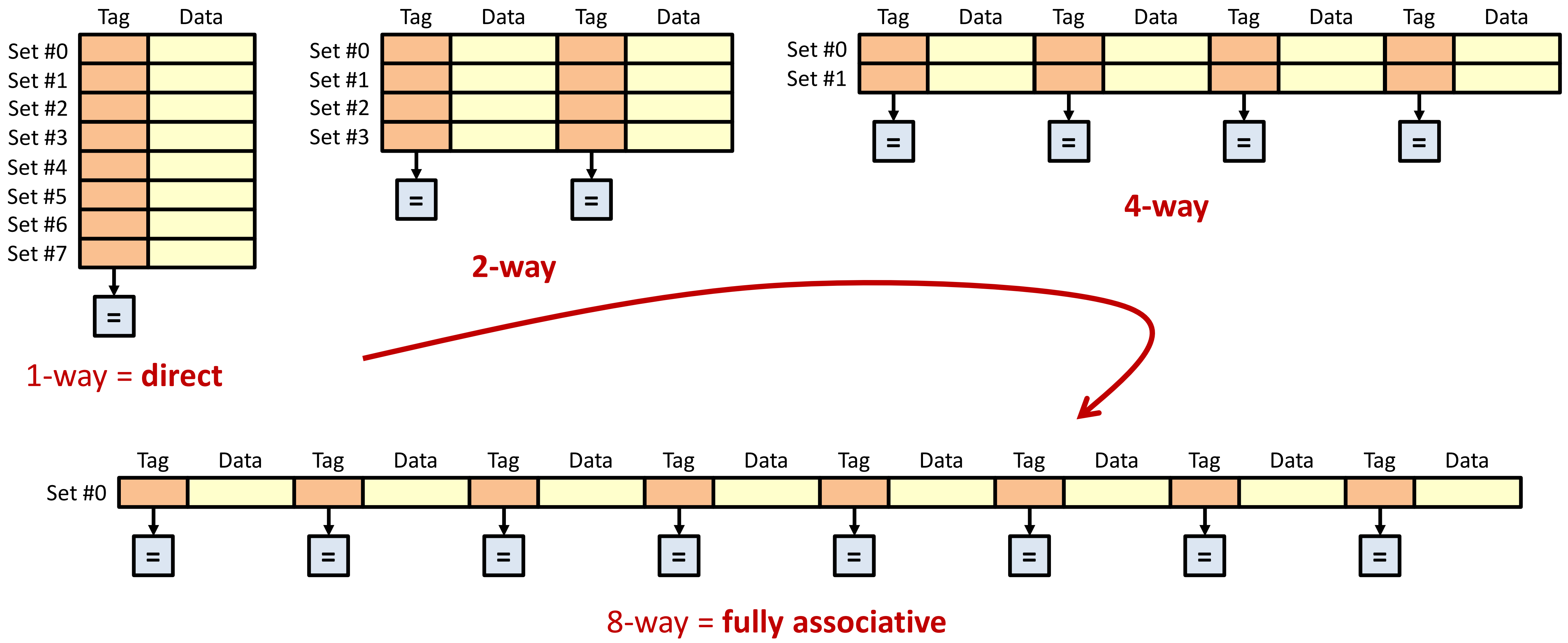
As many comparators as the ways → **# comparators = associativity**

2-Way Set-Associative Cache

Set-Associative Cache

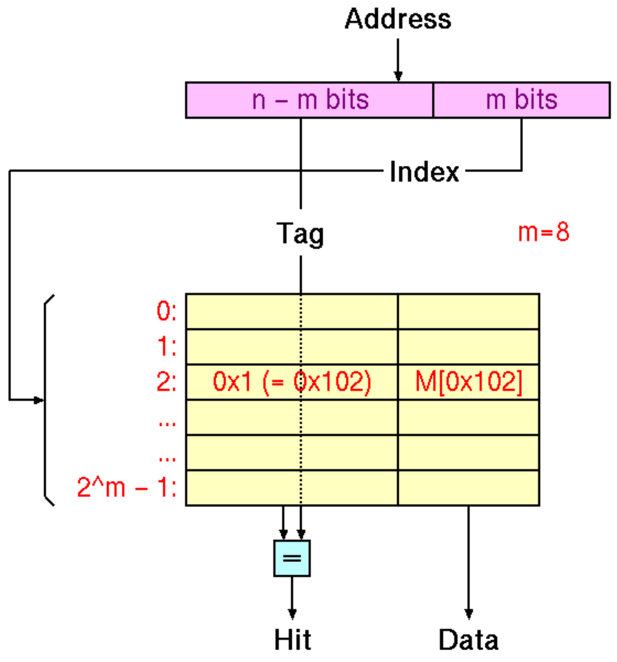


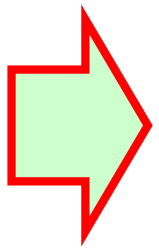
A Continuum of Possibilities

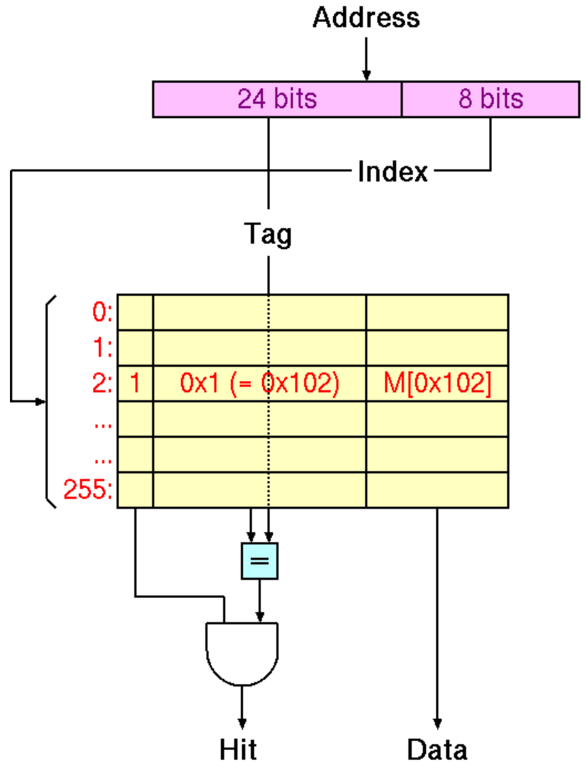


Validity

- Initial cache content is garbage
- All caches need a special bit (**Valid Bit**) in each cache line to indicate whether something meaningful is in the specific cache line (‘0’ at reset)

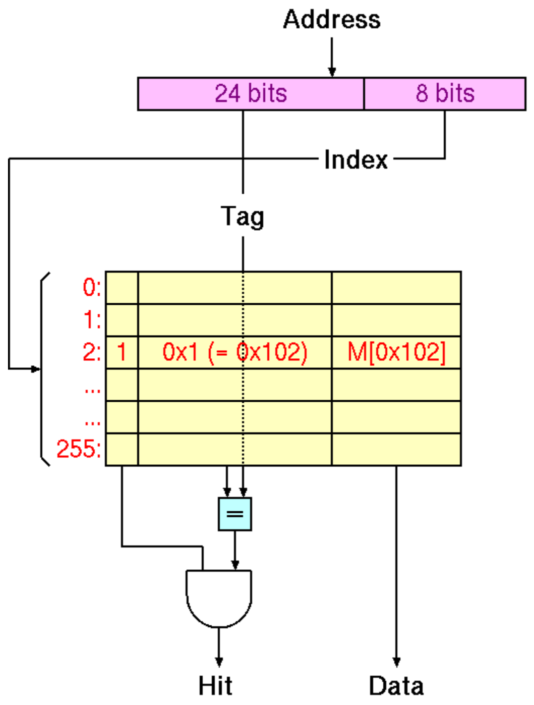




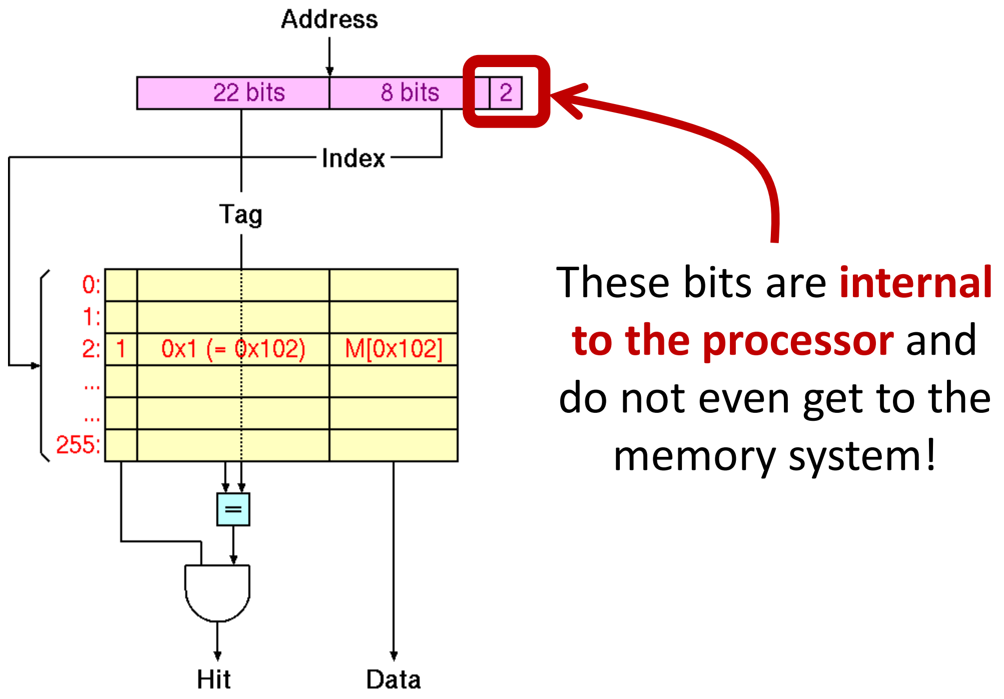


Addressing by Byte

- If addressing is **by byte** and **word size is 2^n bytes**, the ***n* least-significant bits** of the address represent the byte offset and are thus **irrelevant**



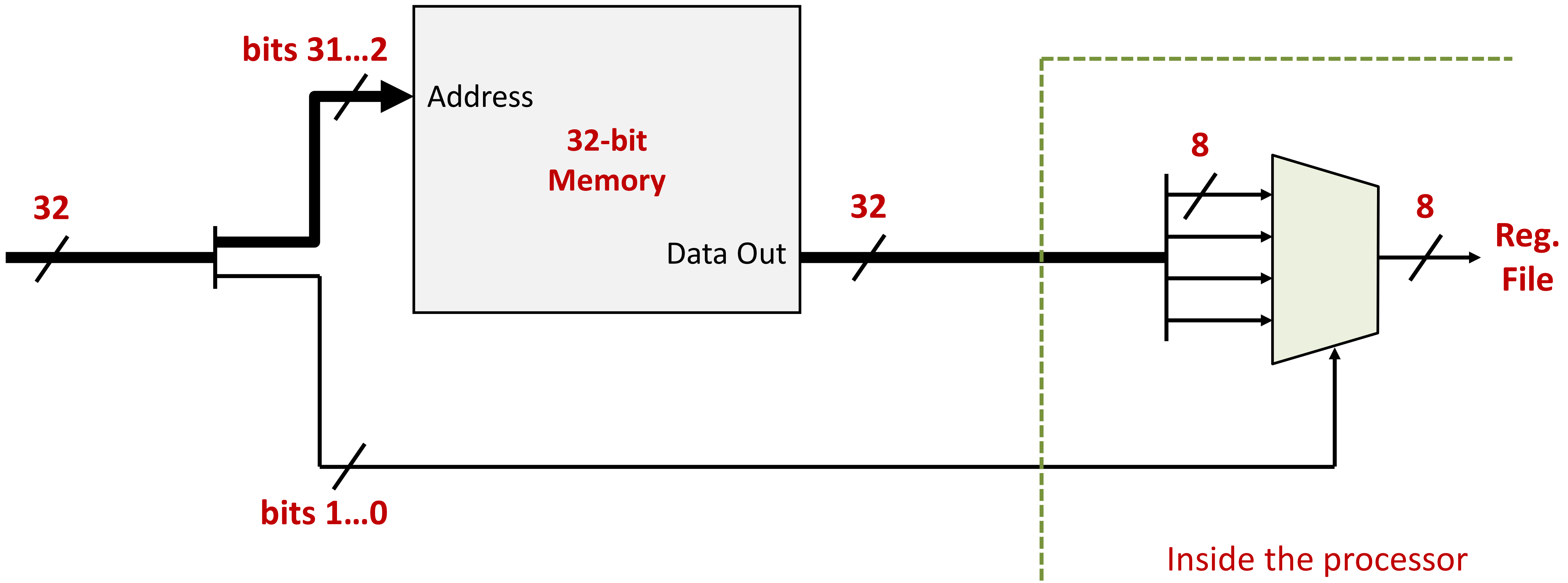
Addressing **by word**



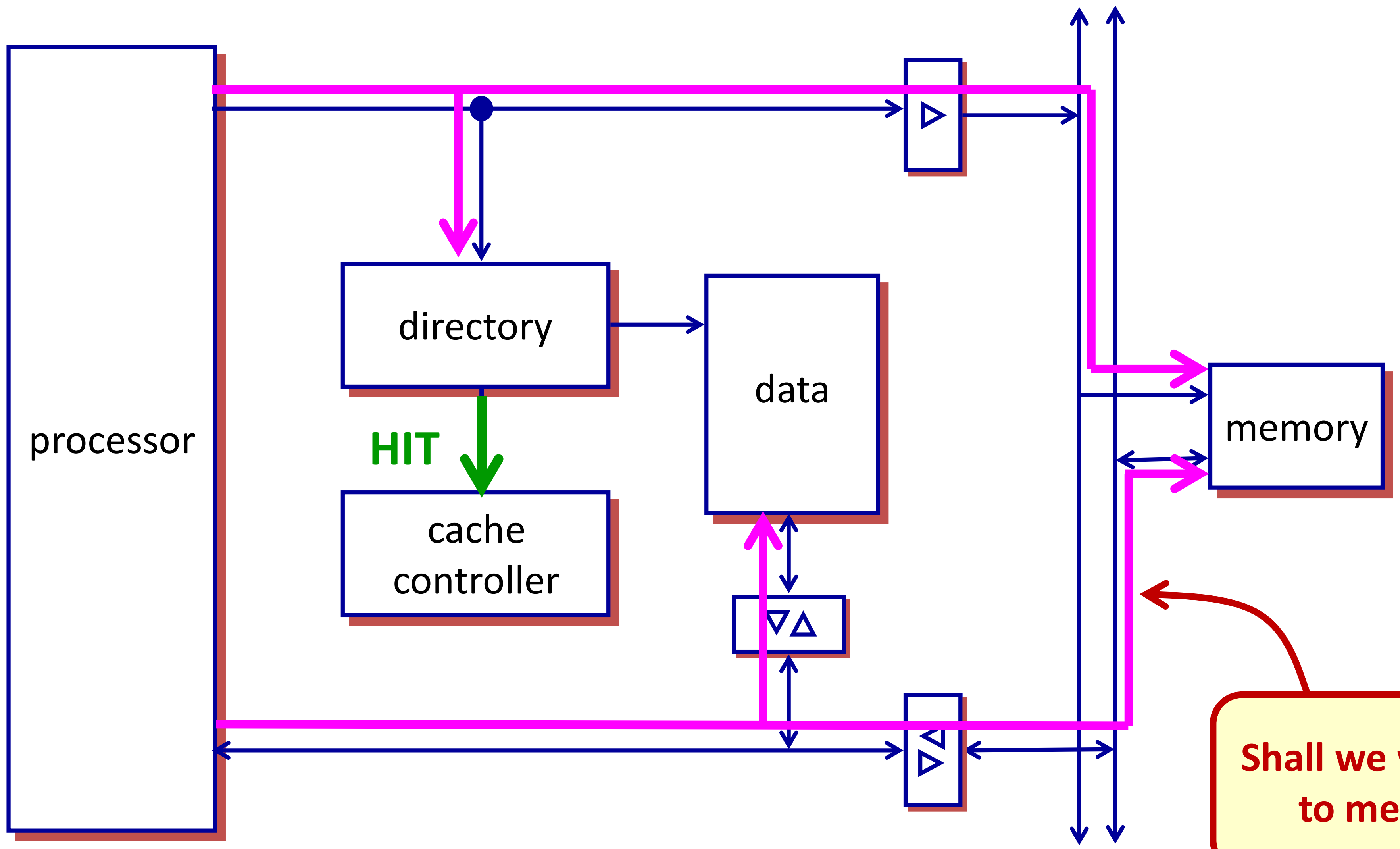
Address
22 bits
8 bits
2
Index
Tag
0:
1:
2:
1
0x1 (= 0x102)
M[0x102]
...
...
255:
=
Hit
Data
These bits are internal to the processor and do not even get to the memory system!

Addressing **by byte**

Loading Bytes (`lb`)



Write Hit



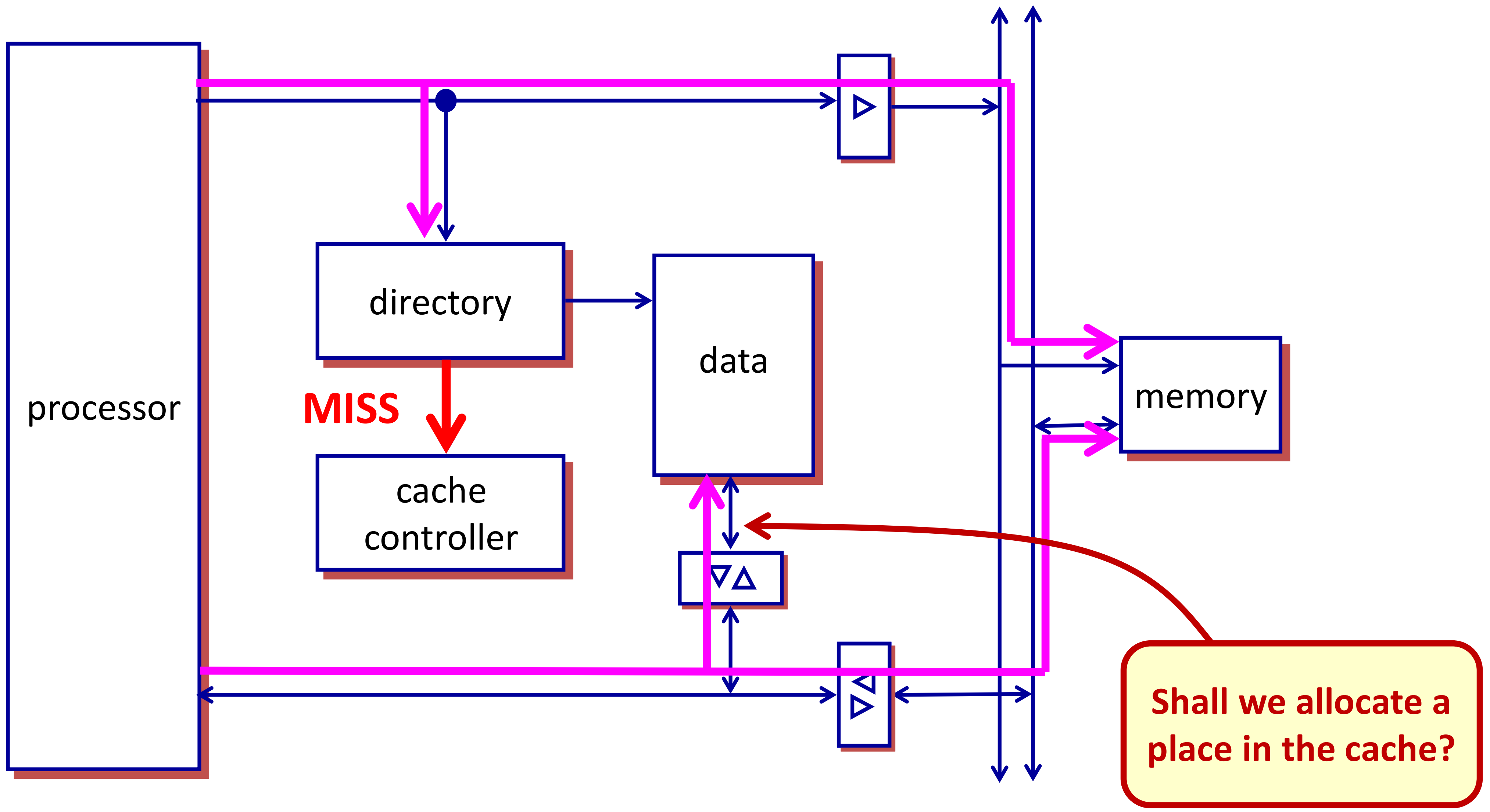
Shall we write also to memory?

Write Policies

- **Write-through**: on a write, data are always immediately written into main memory
  - Simpler policy
  - May keep the memory/buses busy for nothing

- **Write-back** or **Copy-back**: on a write, data are only updated in the cache (hence, main memory data will become wrong/obsolete)
  - Needs a **Dirty Bit** to remember that cache **data are incoherent with memory**
  - When a dirty line is **evicted**, first it must be **copied back** to main memory

Write Miss



Shall we allocate a place in the cache?

Allocation Policies

- **Write-allocate**: on a write miss, data are also placed in the cache
  - Simple and straightforward
  - Need to **fetch the block of data** from memory **first**
  - If the processor writes a lot of data that it will never read back, it may **unnecessarily pollute the cache**

- **Write-around** or **Write-no-allocate**: on a write miss, data are only written to memory
  - If the processor will load from the same address, it will be a **Read Miss**

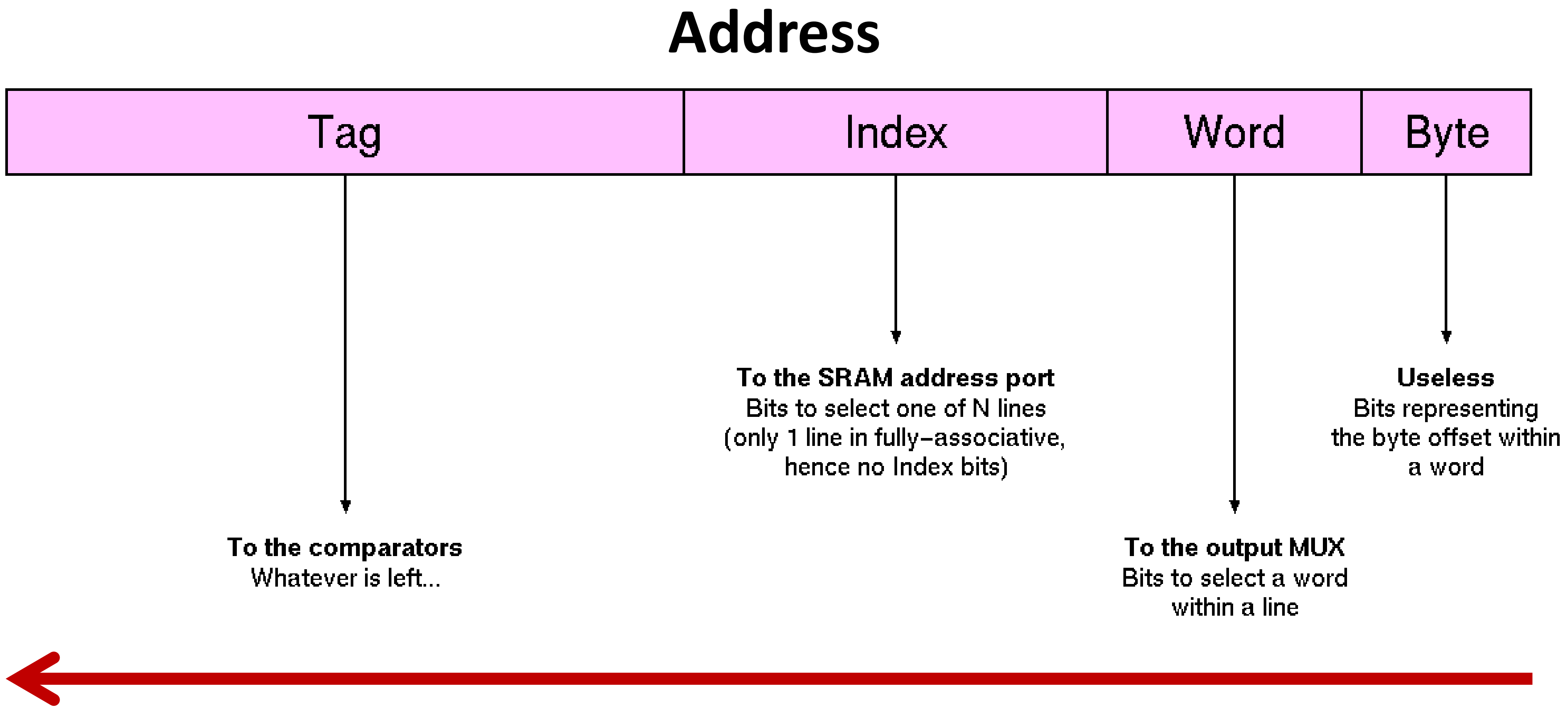
The “3 Cs” of Caches

- Three types of **cache miss**
  - **Compulsory** → Misses that would happen in an infinitely large fully-associative cache with the same blocks (also called **cold-start** misses or **first-reference** misses
  - **Capacity** → Additional misses that occur because the corresponding block has been evicted due to the **limited capacity of the real cache**
  - **Conflict** → Further misses that occur because the corresponding set is full and the corresponding block has been evicted due to the **limited associativity of the cache**
- Useful to understand the **source of the limited performance**

Summary of Cache Features

- **Cache size**: total data storage (usually excluding tags, valid bits, dirty bits, etc.)
- **Addressing**: by byte or word
- **Line** or **block size**: bytes or words per line
- **Associativity**: fully-associative, *k*-way set-associative, direct-mapped
- **Replacement policy** (except for direct mapped): LRU, FIFO, random, etc.
- **Write policy**: write-through or write-back
- **Allocation policy**: write-allocate or write-around

Summary of Cache Addressing


Address
Tag
Index
Word
Byte
To the comparators
Whatever is left...
To the SRAM address port
Bits to select one of N lines
(only 1 line in fully-associative, hence no Index bits)
To the output MUX
Bits to select a word within a line
Useless
Bits representing the byte offset within a word

Allocate bits as required, from LSB to MSB

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Sections 5.3** and **5.4**
  - **Sections 5.9** and **5.15** for more info