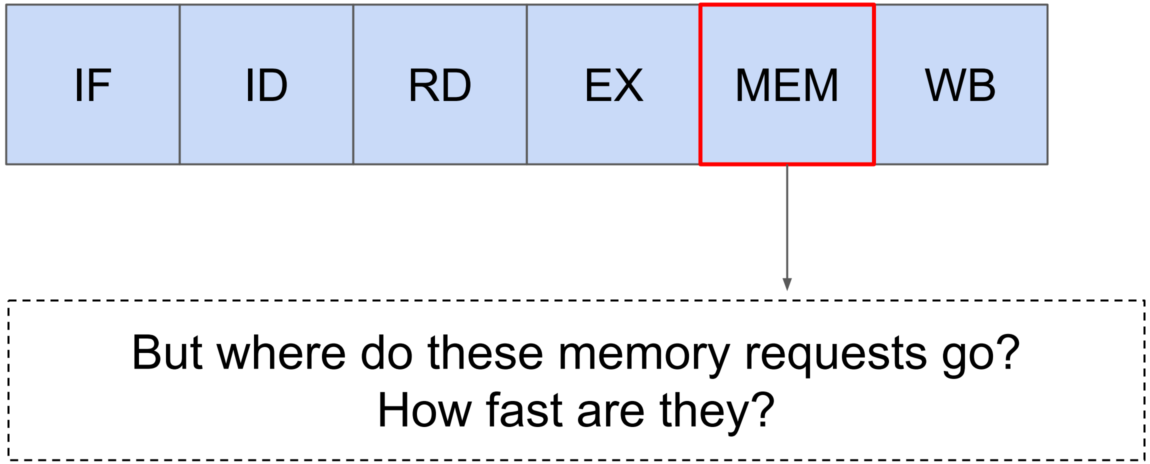
Memory Hierarchy

Feeding your CPU

The MEM stage

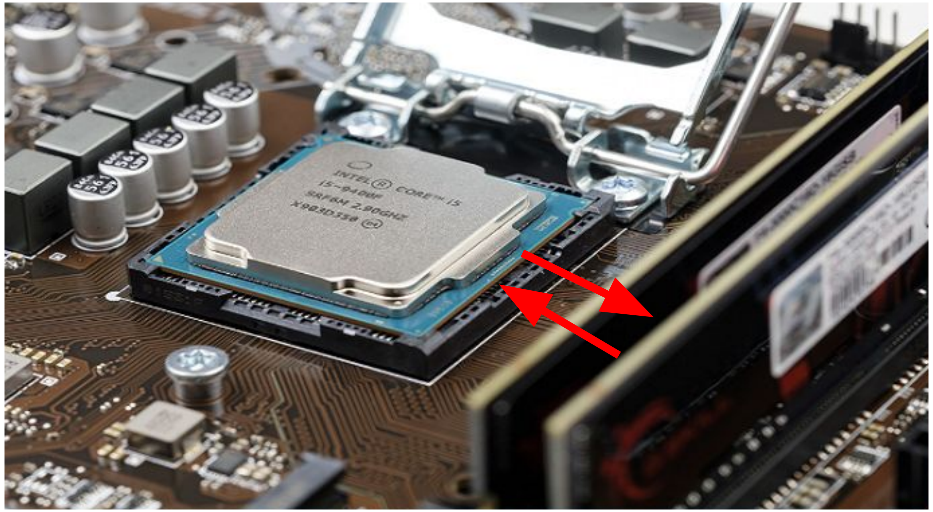
- We have been seeing this diagram of a Pipeline



But where do these memory requests go?
How fast are they?

Simplest design: Main Memory

- CPU's requests go to the external RAM (Random Access Memory)



The main memory is slow because it's:

- Big (hard to index)
- Far (Data must physically travel)

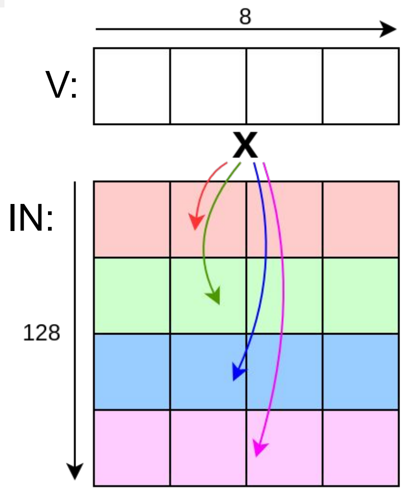
Many instructions depend on memory operations $\longrightarrow$ We need them to be as fast as possible!

… but we can't make the RAM smaller or move it closer…

Are memory accesses "Random"?

- Is using RAM the best option when accesses are not random?

```
int IN[128][8]
int OUT[128][8]
int V[8]
for (int i=0; i<128; ++i){
  for(int j=0; j<8; ++j){
     OUT[i][j] = IN[i][j] * V[j]
  }
}
```



Multiply all rows of "IN" by a vector "V"

Total iterations: 128*8 = 1024

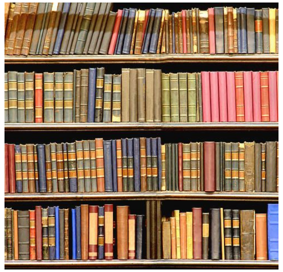
Accesses **OUT**:	1024 stores
Accesses **IN**:	1024 loads
Accesses **V**:	1024 loads

But V only has 8 elements! We are accessing them over and over…

Temporal Locality

- When something is accessed frequently, we say it has **“Temporal Locality”**

- And what do we do with frequent usage?

VS


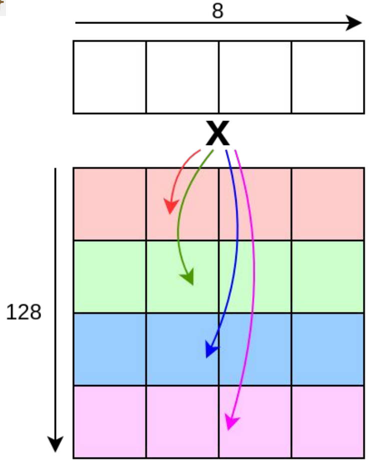
VS

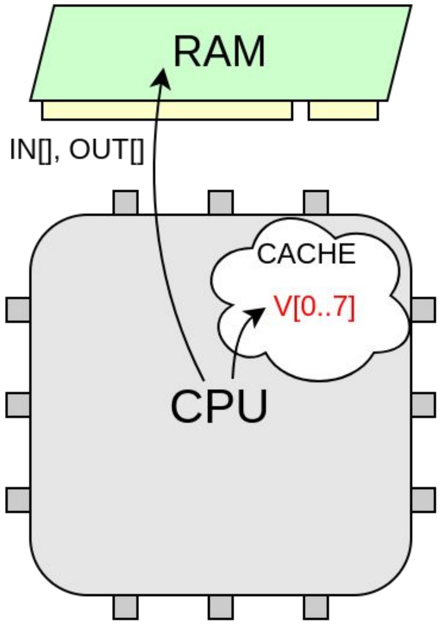
VS

We bring them closer, to smaller storage!

- In computer architecture, we call this a **Cache**
- With a small cache, we reduce the memory accesses

```
int IN[128][8]
int OUT[128][8]
int V[8]
for (int i=0; i<128; ++i){
  for(int j=0; j<8; ++j){
     OUT[i][j] = IN[i][j] * V[j]
  }
}
```





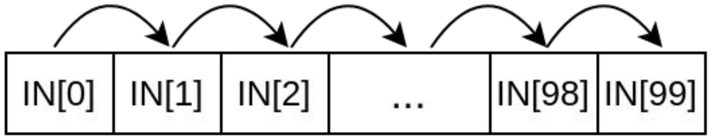
After the first 8 accesses to V, the CPU gets the data from the cache.

Memory Accesses = 2048 + 8
Cache Accesses = 1016

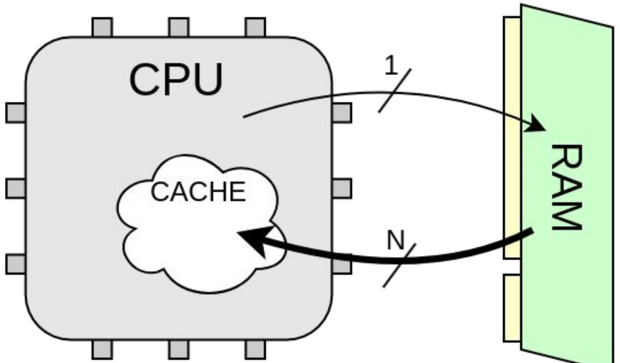
33% cache usage (hit ratio)...
Can we do better?

Spatial locality

- While IN[] and OUT[] are **not reused**, they are not random either.
- We are doing consecutive accesses:



- How do we take advantage of this?



Each time we access the RAM, we put N contiguous elements in the Cache.

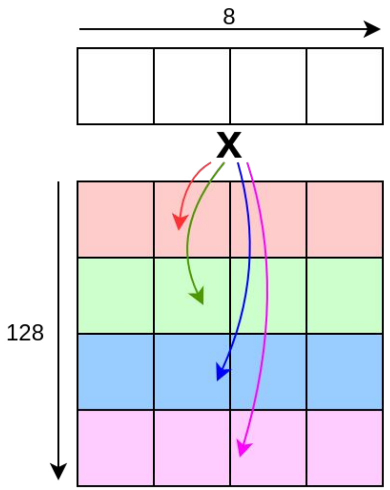
How big is N?

- Too big → More Silicon, More Risky
- Too small →Not useful

Cache Line

- This “N” is called Cache Line, and it’s usually 32, 64, or 128 Bytes wide (*int* = 4 Bytes)

```
int IN[128][8]
int OUT[128][8]
int V [8]
for (int i=0; i<128; ++i){
  for(int j=0; j<8; ++j){
     OUT[i][j] = IN[i][j] * V[j]
  }
}
```



With a Cache Line of 64B, only 1/16 accesses go to the main memory.

IN[] and OUT[]:

Memory accesses:	1/16 x 2048 = 128
Cache accesses:	15/16 x 2048 = 1920

V[]:

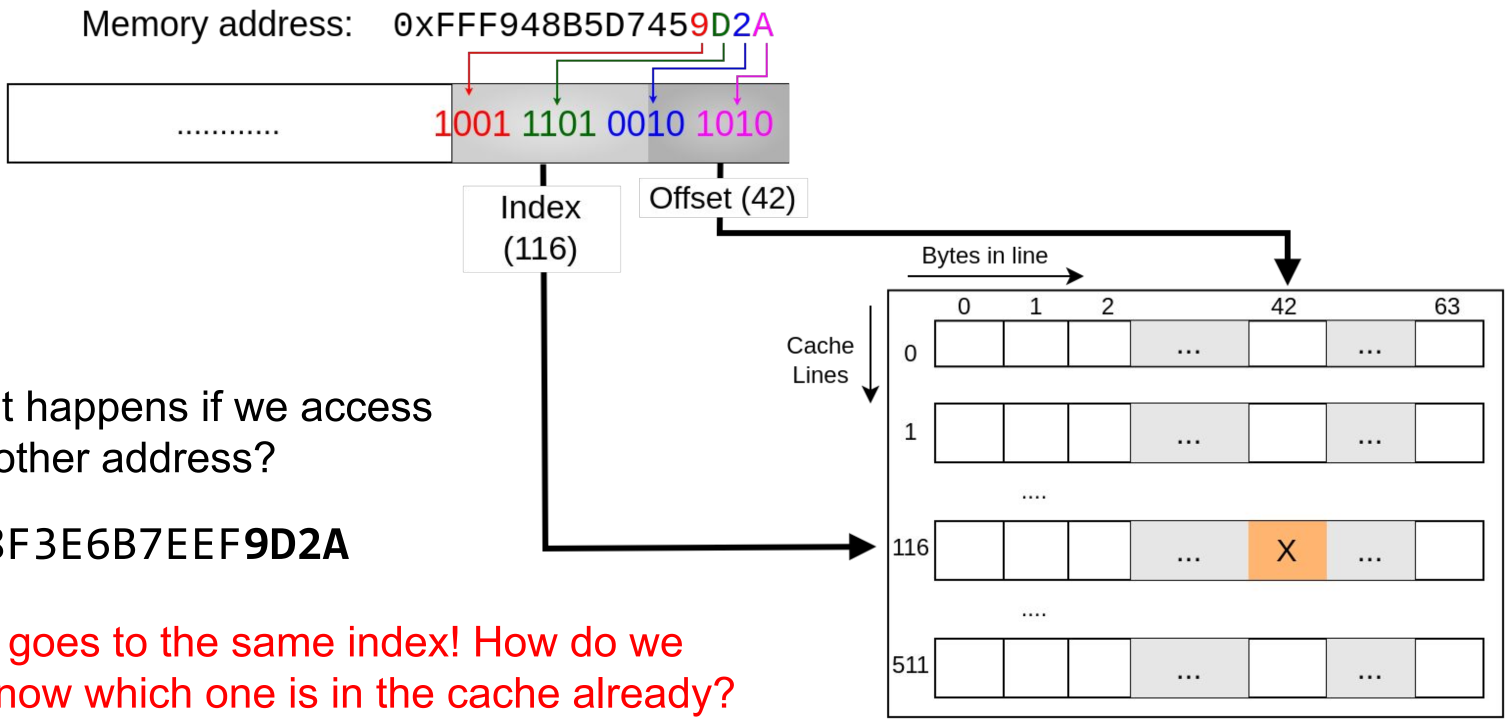
Memory accesses:	1
Cache accesses:	1023

Miss ratio:

$$\frac{129}{3072} = 4.2\%$$

How do we actually fill this cache?

- Easiest approach: **Direct mapping** (lower bits decide where to store data)



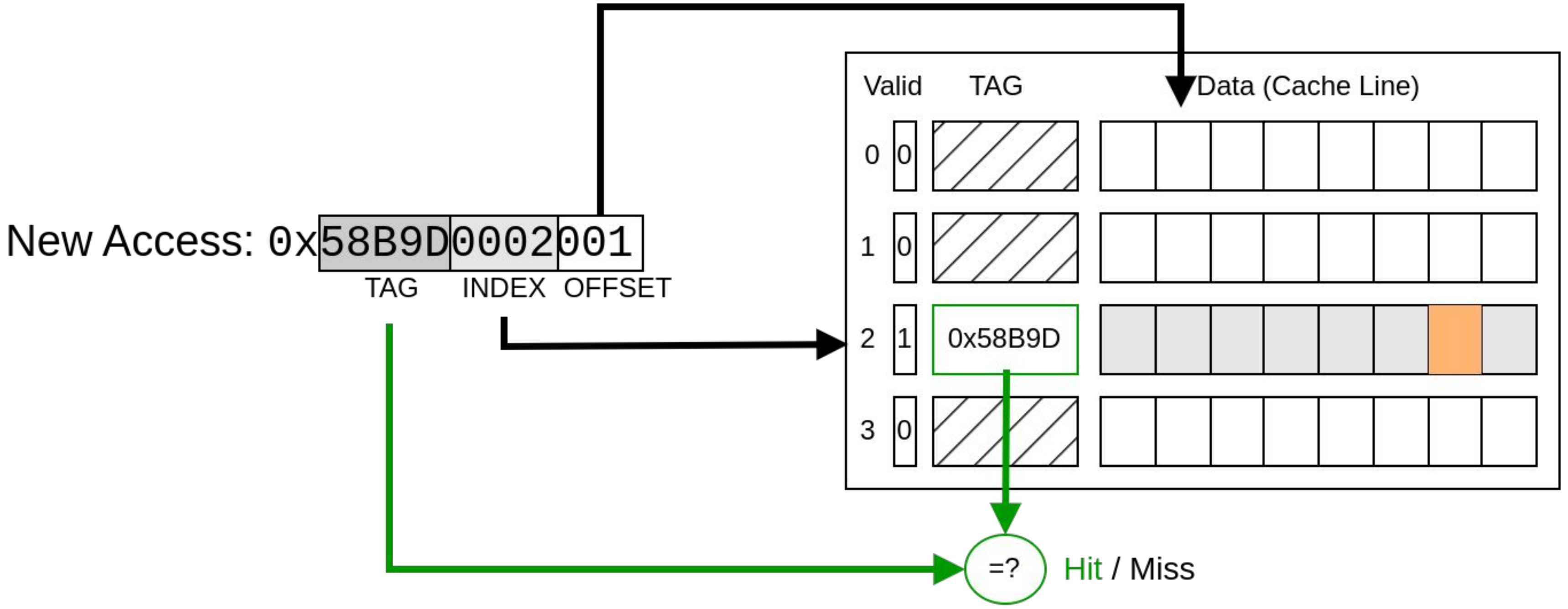
What happens if we access this other address?

0xC8F3E6B7EEF**9D2A**

It goes to the same index! How do we know which one is in the cache already?

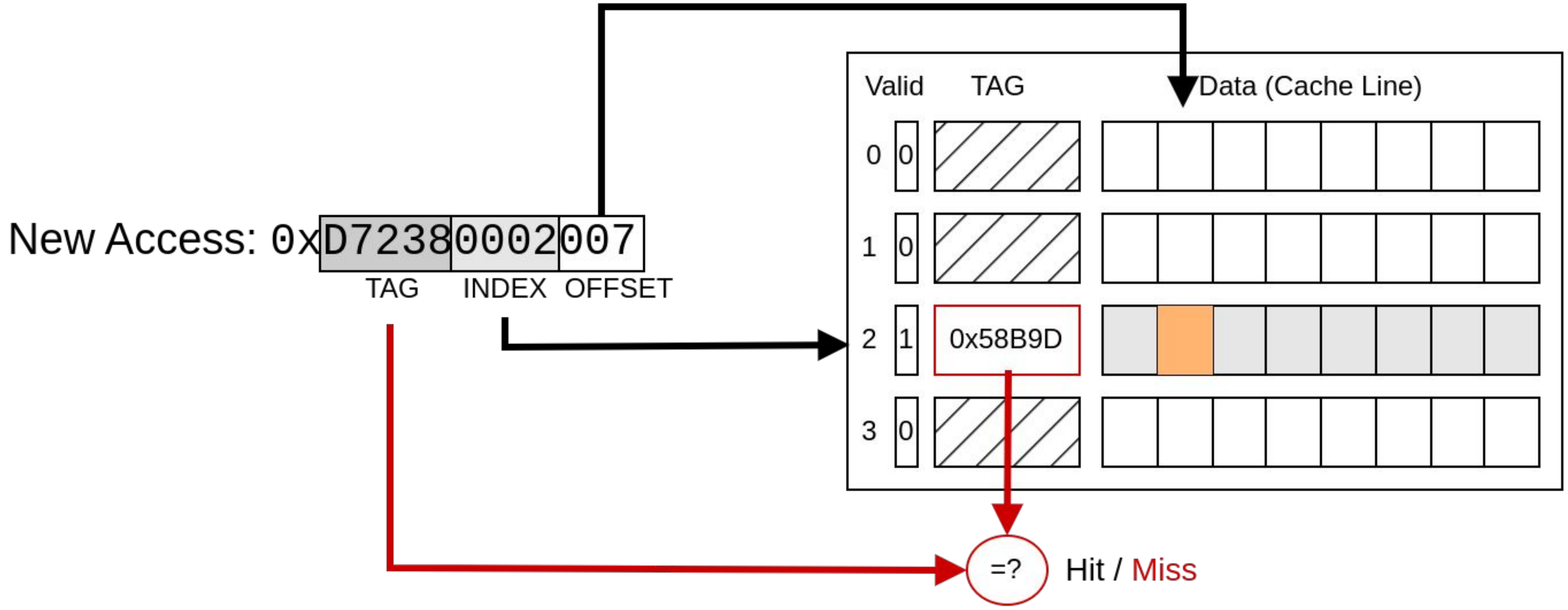
Collisions in a Cache

- We need to add a new field to distinguish addresses: the **TAG**



Collisions in a Cache

- We need to add a new field to distinguish addresses: the **TAG**



Verdict on direct mapping

- Benefits:
    - Easy to implement (extracting bits from an address to index a table is fast and cheap)
    - One single comparison per access
    - This allows to have big caches → Bigger problems can fit

- Drawbacks:
    - High collision chance:

```
int A[256][32]    →    0xF3B44A57D6010000
int B[256][32]    →    0xF3B44A57D6020000
```

Consecutive arrays usually collide on lower bits

```
for(int i=0; i<256; ++i){
   for(int j=0; j<32; ++j){
      A[i][j] = A[i][j] + B[i][j]
   }
}
```

No matter how big the Cache is, each access will evict the other array

↓

All accesses will be misses

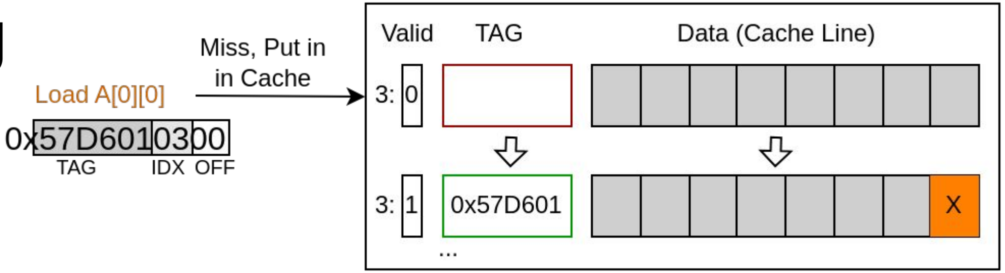
More on Cache Trashing

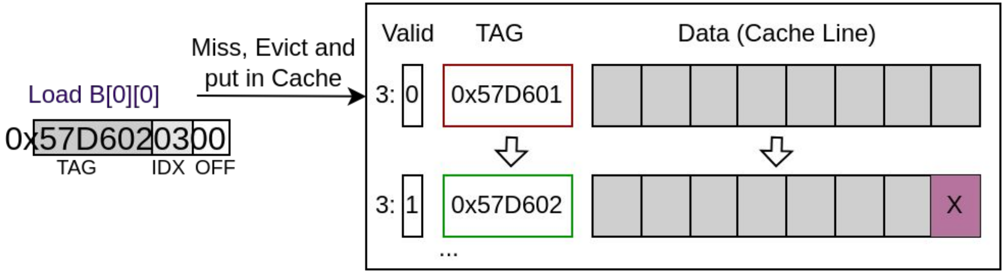
- A and B share a Cache Line:

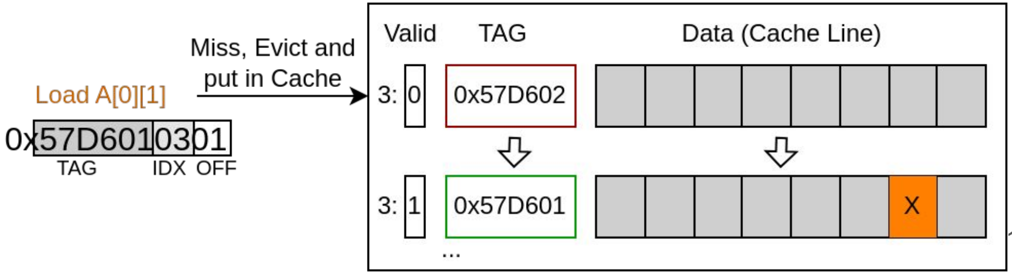
```
int A[256][32]
int B[256][32]
```

```
for(int i=0; i<256; ++i){
   for(int j=0; j<32; ++j){
         A[i][j] = A[i][j] + B[i][j]
   }
}
```

They will **never** hit the cache



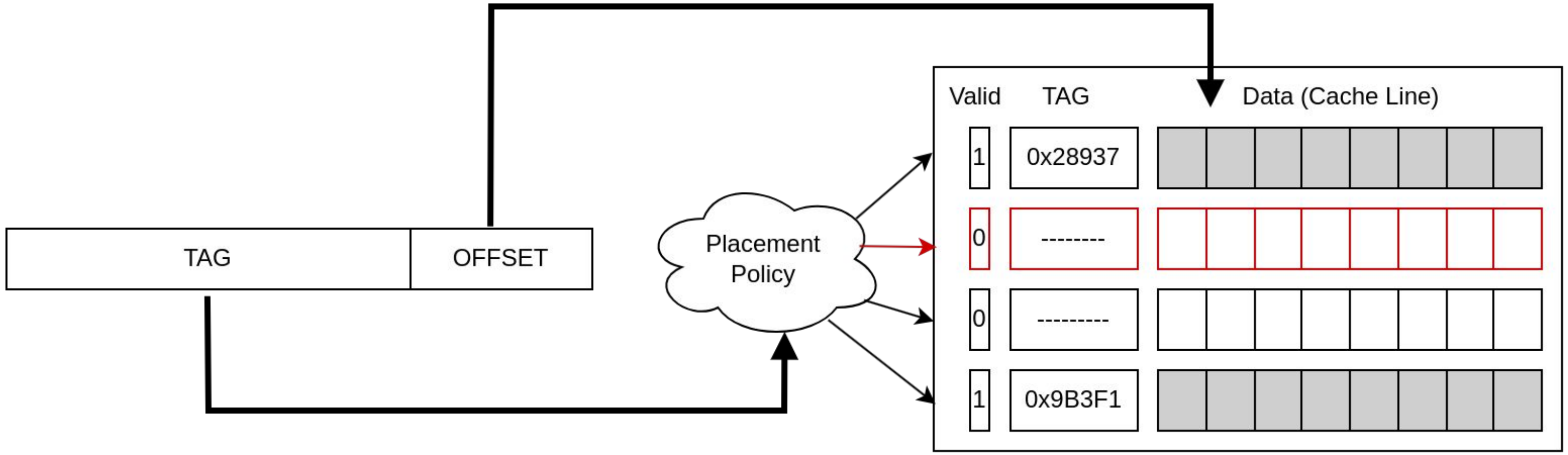




Another approach to caches

- The static placement of direct-map under-utilizes the cache.

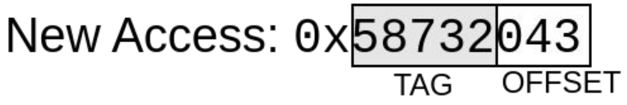
- Alternative: Place lines freely / dynamically on the cache → **Fully Associative Cache**
  - E.g. on the **next free line** (valid==0).

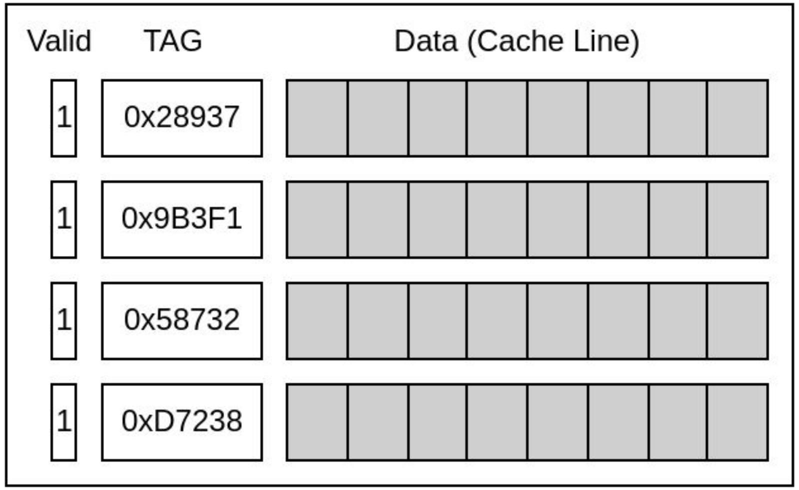


- More expensive, but ensures that the cache gets filled before evictions happen.

How do we find our data?

- Consider this **Fully Associative Cache** with all entries occupied:

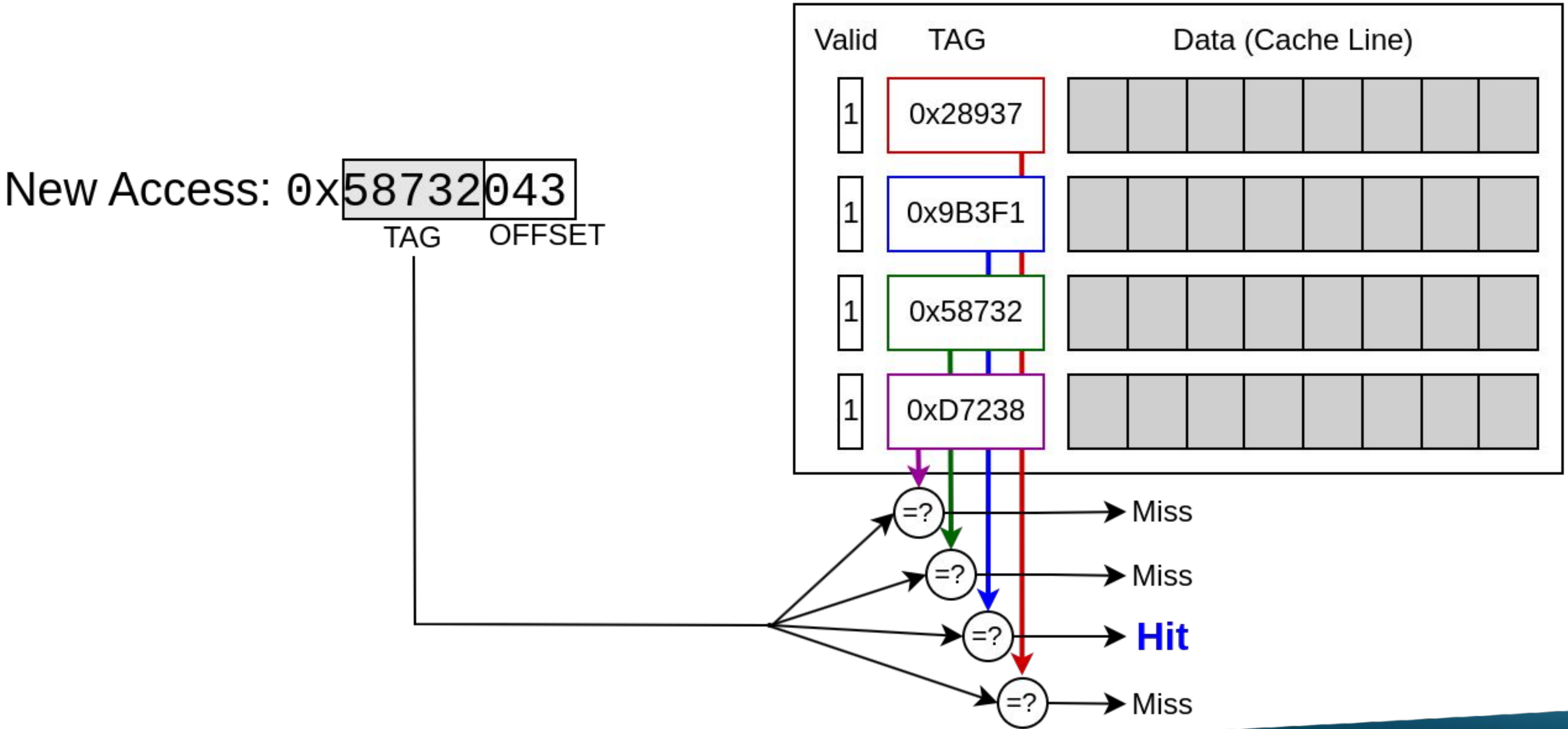




Can this access be served by the cache, or shall it go to memory?

We look at the tags!

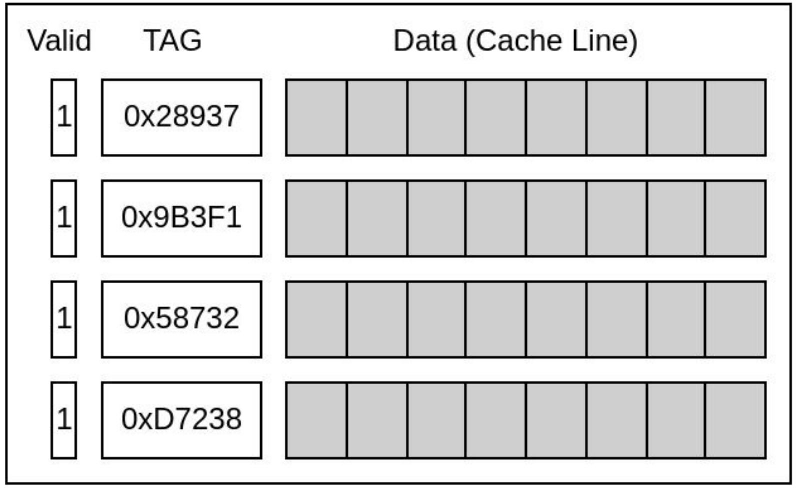
- We need to do as many comparisons as number of lines



Other considerations

- Assume this full cache again:

New Access: 0x$\underbrace{\boxed{\texttt{4B94A}}}_{\text{TAG}}\underbrace{\boxed{\texttt{022}}}_{\text{OFFSET}}$



Does this access hit or miss?

Where do we put it? The cache is full…

Replacement Policies

- With direct mapping, we did not need replacement policies, as each address could only go to one specific line.

- Now, we have the freedom to choose who to evict. Some common policies are:

Complexity ↓

  - **FIFO**: First in - First out → Replace the oldest cache line
  - **LFU**: Least Frequently Used → Replace the cache line that got accessed the least
  - **LRU:** Least Recently Used → Replace the cache line that got used further in time

- There are many replacement policies, but some require adding extra logic, counters, and comparisons to the cache, will comes at a cost

“Good enough” policies

- Sometimes approximations to LRU / LFU are used
- For example: **Hot Potato**
    - 1. The first line of the cache is given a “Hot Potato”
    - 2. Whenever it is accessed, it passes the Hot Potato to the next line
    - 3. If the cache is full and we need to evict a line, we choose the one with the Hot Potato

- Very simple mechanism (you only need a counter)

- Lines that get used the most are less likely to hold the Hot Potato (but still possible)

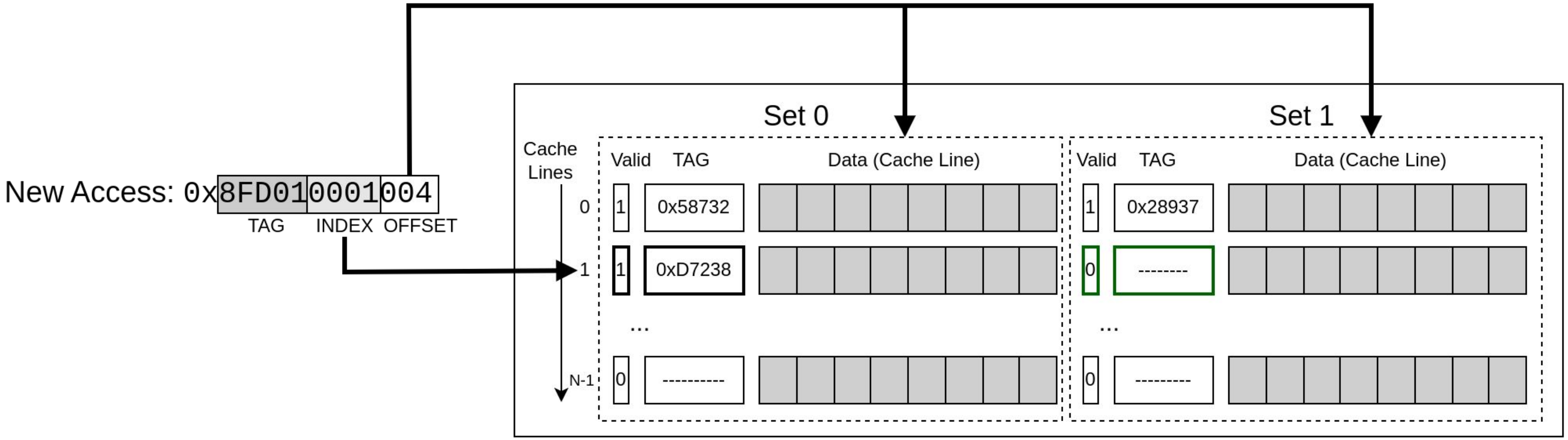
Verdict on Fully Associative Caches

- Benefits
  - Ensures that the cache gets full before evictions happen.
  - Allows to control who gets evicted.

- Drawbacks
  - As many comparisons as Cache Lines *(1MB cache with 64B Lines → 16384 comparisons/access)*
  - Expensive in hardware → Limit on how big they can be.

The best of both worlds

- What if we mix direct mapping with associativity? → **Set Associative Cache**
- Addresses are mapped to cache lines, but can go into S sets
  - We refer to S-way Associative Cache (e.g. 2-way)



- Placement and replacement between sets is handled like in an associative cache.
- #S comparisons per access.

Verdict on Set Associative Caches

- Benefits
  - Can avoid some collisions (e.g. This example from before)

```
int A[256][32]
int B[256][32]
for(int i=0; i<256; ++i){
   for(int j=0; j<32; ++j){
        A[i][j] = A[i][j] + B[i][j]
   }
}
```

0xF3B44A57D6010000

0xF3B44A57D6020000

Even if they share a Line, they can go to different sets

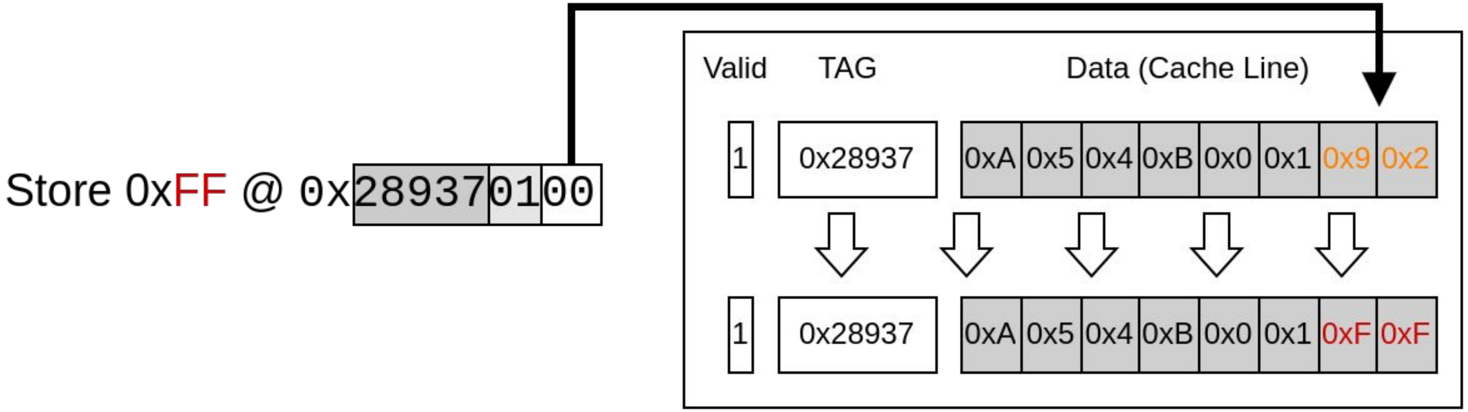
  - Cheaper than Full Associative Caches (fewer comparisons and logic)

- Drawbacks
  - *"Jack of all trades, master of none"* → More Costly than Direct Map, can't avoid all collisions.
  - No optimal configuration (i.e. 2-way? 4-way? … Depends on the workload!)

Modify data that is cached

- What do we do if we want to **store** new data to a cached address?

- **First approach** → Modify the cache line:

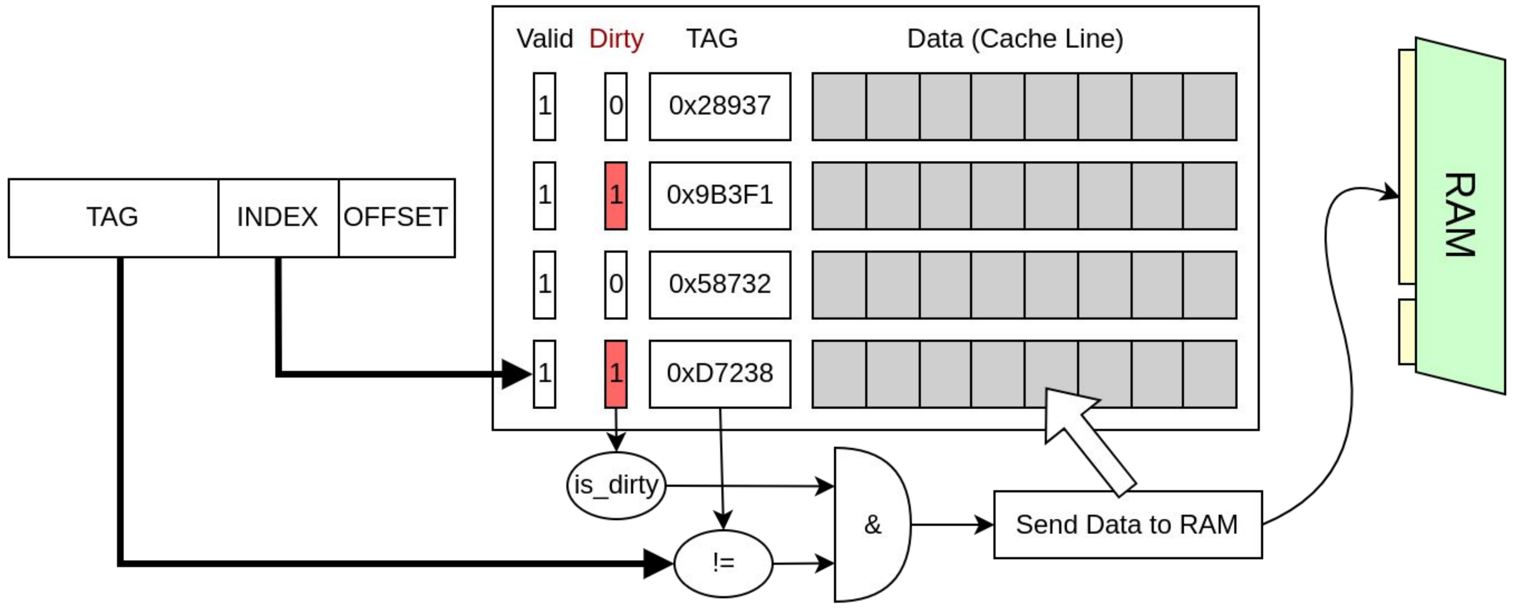


Store 0xFF @ 0x28937|01|00
Valid
TAG
Data (Cache Line)
1
0x28937
0xA 0x5 0x4 0xB 0x0 0x1 0x9 0x2
1
0x28937
0xA 0x5 0x4 0xB 0x0 0x1 0xF 0xF

What happens when this Line is evicted?

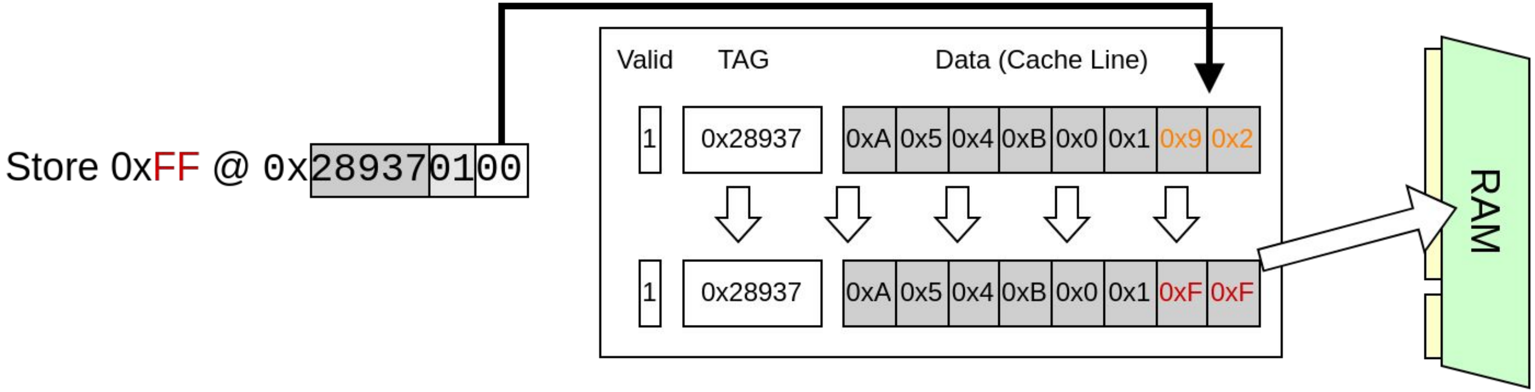
Propagate writes to the main memory

- We set a “**Dirty**” bit when we modify a Cache Line
- When we evict that line, we also **write-back** to the main memory



Another solution

- Some caches are "**write-through**" instead of "**write-back**".
- When a Cache Line gets updated, so does the main memory


Store 0xFF @ 0x28937|01|00
Valid
TAG
Data (Cache Line)
1
0x28937
0xA 0x5 0x4 0xB 0x0 0x1 0x9 0x2
1
0x28937
0xA 0x5 0x4 0xB 0x0 0x1 0xF 0xF
RAM

- More traffic to the main memory
- Higher latencies
- Defeats the purpose of the cache…
- … But helps keeping coherence, and avoids the dirty bit

What to do with missing stores

- “**write-no-allocate**” policy:
  - Common policy in “**write-through**” architectures.
  - Missing writes directly modify the main memory without using the cache
  - Simple approach, with fewer evictions,
  - But increases main memory traffic, defeating the purpose of a cache

- “**write-allocate**” policy:
  - Common policy in “**write-back**” architectures.
  - Missing writes do not modify the main memory, only the cache.
  - Leaves the modification for later, when line is evicted
  - Can save up on main memory traffic if line is modified multiple times.
  - May evict a dirty line, which in turn means writing to the main memory anyway.

Can we improve cache hits further?

- We exploit the regular nature of most codes with spatial locality, but we still miss:

Consecutive

- 0x032678f000 $\longrightarrow$ Miss
- 0x032678f004
- 0x032678f008
- 0x032678f00C
- 0x032678f00E $\longrightarrow$ Miss
- 0x032678f010
- 0x032678f014
- 0x032678f018
- 0x032678f01C $\longrightarrow$ Miss
- 0x032678f01E
- 0x032678f020
- 0x032678f024

What's the next access? $\longrightarrow$ Most likely **0x032678f024 !**

Prefetchers

- We can leverage these “guesses”.

- The Load / Store unit can include a **Prefetcher:**
  - Analyzes the past accesses
  - Predicts future accesses, puts them in cache before they happen

- Varying complexity:
  - How far into the past can they look?
  - What patterns to they recognize?

0, 1, 2, 3, 4 … 5?

0, 10, 20, 30, 40 … 50?

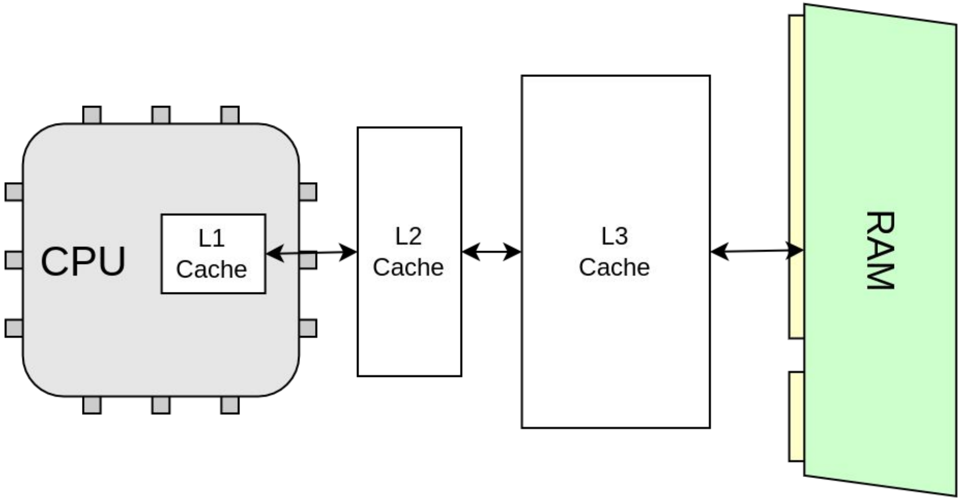
0, 10, 1, 11, 2, 12 … 3?

- Not always helpful:
  - Polluting the cache with speculative data can hurt the performance!

What about a second Cache?

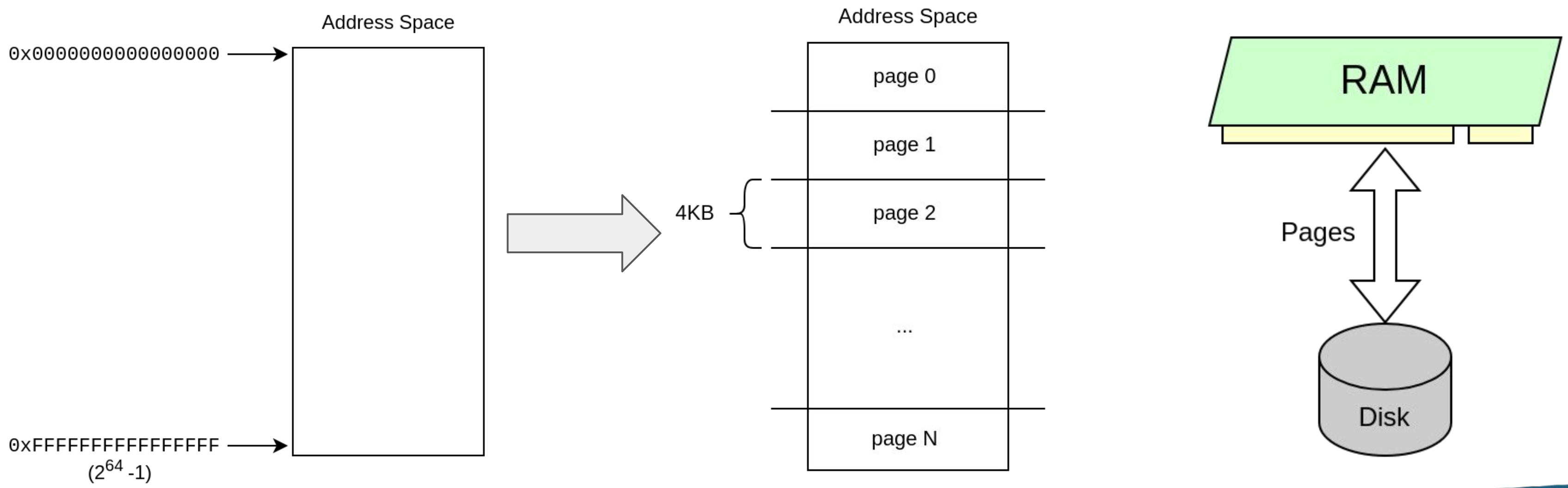
- Modern architectures have different “Levels” of caches:
    - A first level (**L1**) inside the core, relatively **small** (16/32/64 KB) but **fast**
    - More levels (**L2, L3**, …) outside the core, **bigger** (1MB - 100MB) but **slower**

- When an access misses on a cache level, it queries the next one.



Are caches only used to avoid accessing the RAM?

- No! But first, a 1-minute lecture on virtual memory.
- Loads/Stores can index addresses ranging from 0 to 2^{64} -1 → The **address space.**
- The O.S. breaks the address space into **pages** of 4KB.
- Pages are, for example, used to load data from Disk into RAM.



Pages and virtual memory

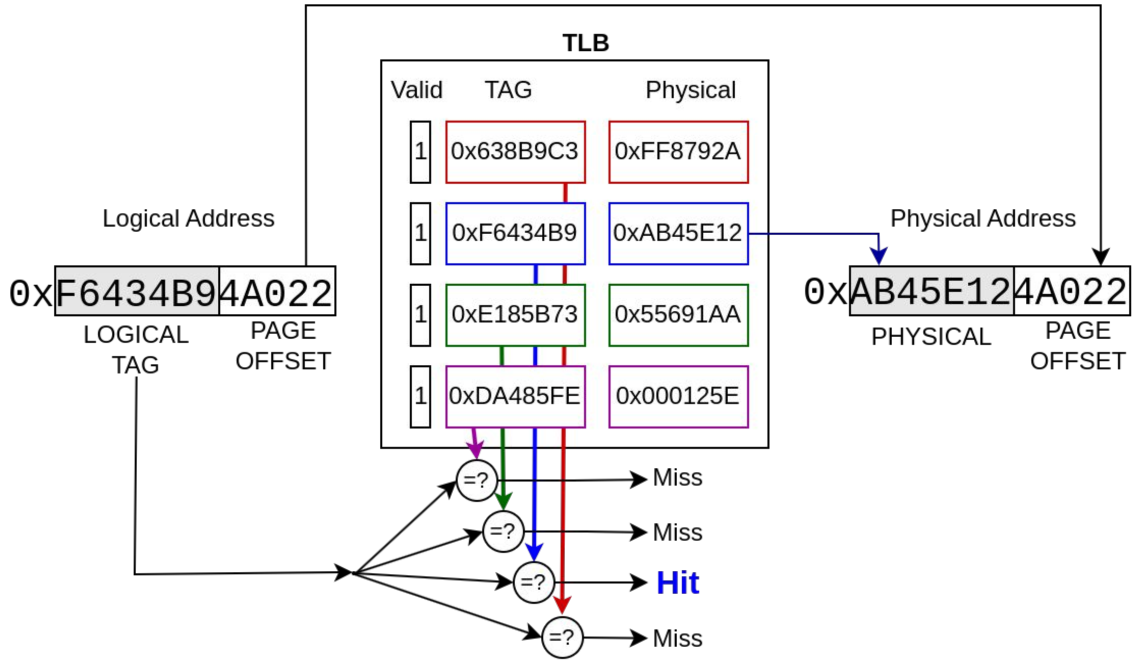
- Pages are also useful for **Virtual Memory:**
  - The O.S translates the addresses that users request to physical addresses (for security).
  - This is done in the **MMU** (Memory Management Unit) using a **Page Table**
  - Page table can be huge → And slow!

Page Table

Physical	**Virtual**
...	...
0x1A8	0x325
0x1A9	0xFF8
0x1AA	0x023
0x1AB	0x55E
0x1AC	0xFD1
0x1AD	0x123
...	...

Translation Lookaside Buffer (TLB)

- The hardware can help the O.S. by putting the page table in a cache → the **TLB**
- Whenever we need to translate an address from virtual to physical space, we use it:



- We want to avoid expensive misses, so we make this cache **Full-Associative** (and small!)

- On miss, access the page table.

Conclusions

- A fast memory access needs small and close storage (not RAM!)

- We can exploit access regularity:
  - Temporal locality → Reusing data
  - Spatial locality → Using data that is close together

- We create small memories (caches) to store frequently used data
  - Direct map, Associativity, write modes, replacement policies, …

- These caches can be stacked into multiple levels

- They can be used on other contexts (e.g. address translation)