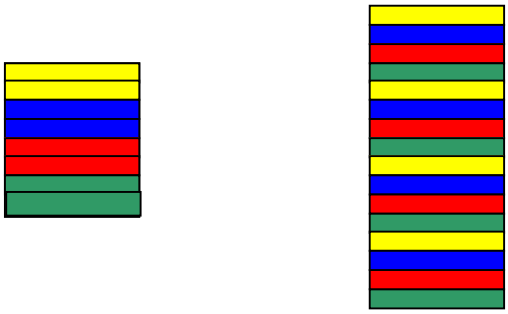
Address translation

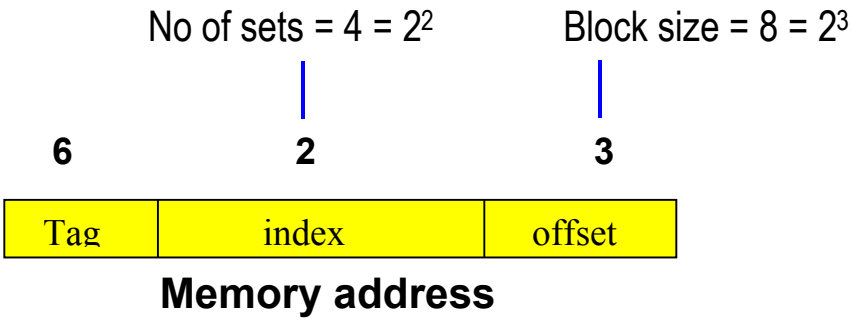


Main memory size = 2 KB = 2^{11}

Block size = 8 bytes = 2^3 (So, total # of M-blocks = 2^8)

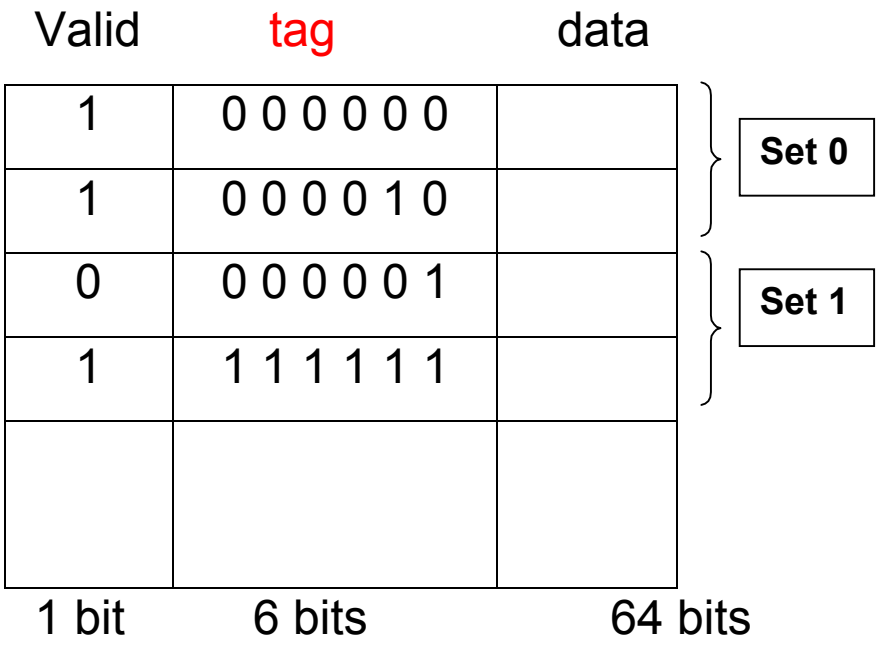
Cache size = 64 bytes = 2^6 (So, total # of C-blocks = 2^3)

Set size = 2, so the number of sets in cache = 4



To locate an M-block in cache, check the tags in the set S = (M-block) mod (number of sets) i.e. the index field.

Sample Cache Organization



Use **index** to choose the set.

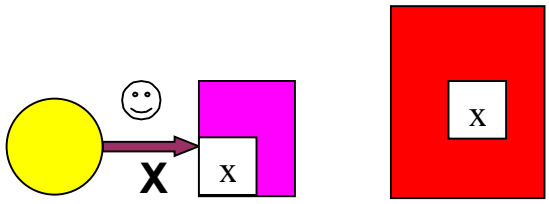
Check the **valid** bit (for invalid data or bad initialization) and then look for a match with the tag.

Direct mapped cache	Set size =1
Fully associative cache	Set size = total number of blocks in the cache

Tag search is limited within a set.

What about writing ?

Case 1. Write hit



(store X: X is in C)

Write through	Write back
Write into C as well as into M	Write into C only. Update M only when discarding the block containing x

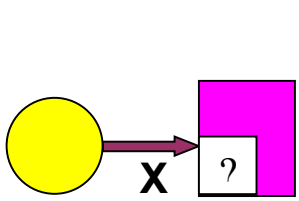
Q1. Isn't write-through inefficient?

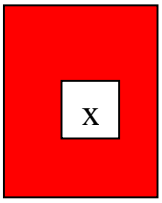
Not all cache accesses are for write.

Q2. What about data consistency in write-back cache?

If M is not shared, then who cares?

Case 2. Write miss





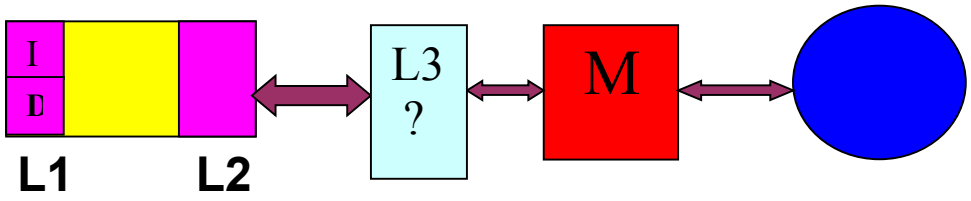
(Store X, X is NOT in C)

Write allocate	**Write around**
Allocate a C-block to X.	Write directly into
Load the block containing X from M to C.	X bypassing C
Then write into X in C.	Usually goes with
Usually goes with write back	write through.
.	

Question.

In write-allocate, it is important to read the entire block from the memory into the cache. Why?

A state-of-the-art memory hierarchy



0.5 ns*	**0.75 ns***	**50 ns***	**1 ms**
16kB+16KB	512KB-2 MB	64-256MB	40 GB

(* per 32-bit word)

Reading Operation

- Hit in L1.
- Miss in L1, hit in L2, copy from L2.
- Miss in L1, miss in L2, copy from M.

Write Hit

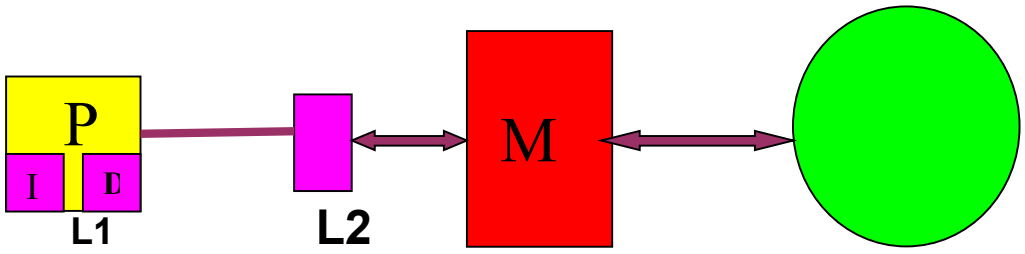
- Write through: Write in L1, L2, M.
- Write back

  Write in L1 only. Update L2 when discarding an L1 block. Update M when discarding a L2 block.

Write Miss

Write-allocate or write-around

Inclusion Property



In a consistent state,

- Every valid L1 block can also be found in L2.
- Every valid L2 block can also be found in M.

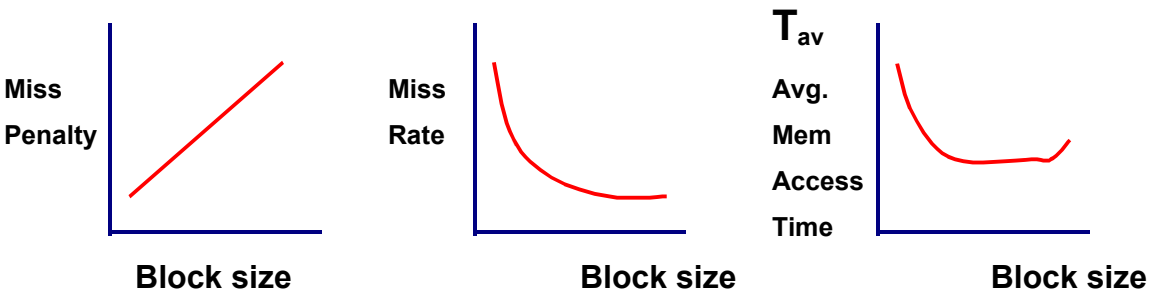
Average memory access time =

$(\text{Hit time})_{L1} + (\text{Miss rate})_{L1} \times (\text{Miss penalty})_{L1}$

$(\text{Miss penalty})_{L1} = (\text{Hit time})_{L2} + (\text{Miss rate})_{L2} \times (\text{Miss penalty})_{L2}$

Performance improves with additional level(s) of cache if we can afford the cost.

Optimal Size of Cache Blocks



Large block size supports program locality and reduces the miss rate.

But the miss penalty grows linearly, since more bytes are copied from M to C after a miss.

$$T_{av} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}.$$

The optimal block size is 8-64 bytes. Usually, I-cache has a higher hit ratio than D-cache. Why?

Improving Cache Performance

- Reduce miss rate
- Reduce miss penalty
- Reduce hit time

Reducing miss rate

Three reasons for cache miss (3 C's)

- Compulsory Cold Start
- Capacity Cache size < Program size
- Conflict Mapping restrictions

Method 1. ***Use larger blocks*****.**

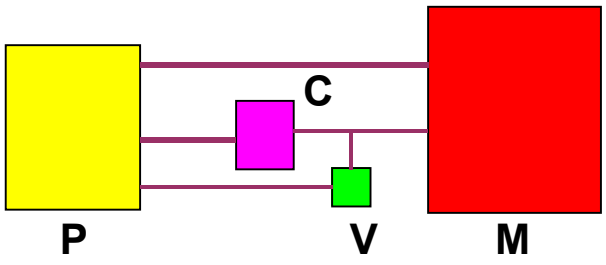
But it is counterproductive beyond a limit.

Method 2. ***Increase the associativity*****.**

But the cost goes up, and the hit time may increase due to increased overhead of associative search.

Method 3. ***Victim Cache.***

A fully associative cache that can hold 2-4 blocks, and works like a waste basket.



A 4-block victim cache reduced the conflict misses of a 4 KB direct-mapped caches by 20-95% without affecting the clock rate.

Method 4. ***Instruction and Data Prefetching***

Fetch *one or more additional blocks* during a cache miss, and store the prefetched blocks in the *instruction stream buffer.*

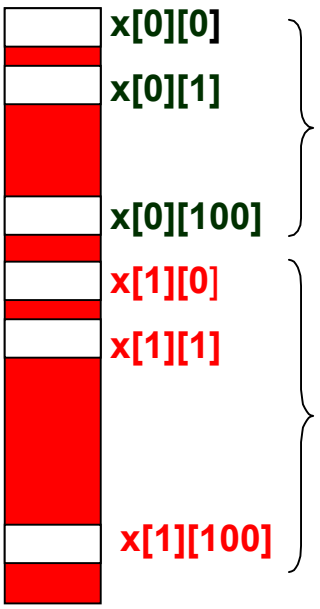
Method 5. *Compiler Optimizations*

Example 1: Loop Interchange

```
for (j=0;  j<100;  j=j+1)
        for (i=0; i <100; i= i+1)
                x[i][j] = 2* x[i][j]
```

loop interchange improves spatial locality

```
for (i=0; i<100; i= i+1)
        for (j=0;  j <100;  j=j+1)
                x[i][j] = 2* x[i][j]
```

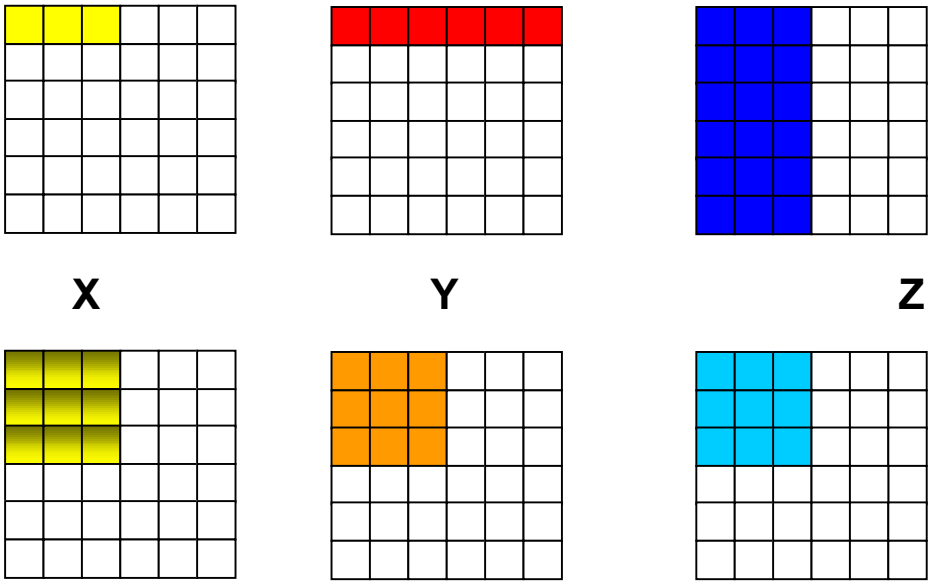


Note. In this example, we assumed that the elements of the matrix have been stored in **row-major form.**

Example 2. ***Blocking reduces capacity misses***

Maximize the use of existing cache blocks before replacing them.

Consider X = Y * Z (each matrix is N x N)



Instead of multiplying the elements of a row of Y by the elements in different columns of Z, divide them into sub-operations, and make the best use of the data elements already in the existing cache blocks