Computer Architecture

DATE: 27/4/22

MEMORY

- Spatial Locality → nearby access
- Temporal "

} Programs have both of these

```
m := a[i]
for i := 1 to 1000000
    if a[i] < m then
        m := a[i]
```

— spatial locality

temporal locality

IF this is in the memory, then it's a case of temporal locality

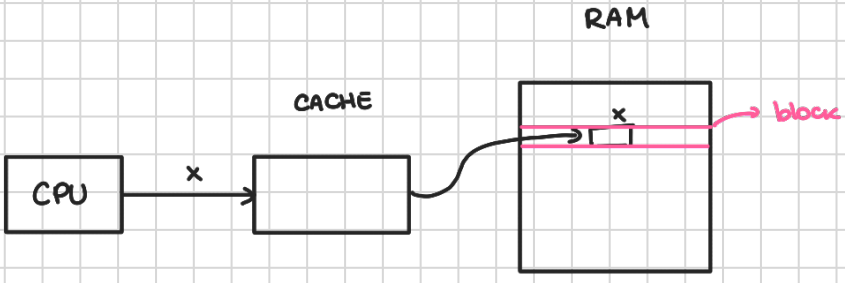
(Usually "i" is in the register)

Each cycle we load instructions but since it's in a loop now, it's temporal locality.

Linked lists, binary trees and hash tables don't have locality properties.

- We exploit locality properties to make memory instr. faster.
- We can build small memories → expensive but very fast

HIERARCHY OF MEMORY



READ

CPU asks the content of an address "x"

1. the cache may already have the content (HIT)
2. cache asks the RAM for the content, saves it in cache and gives to CPU (MISS)
   - when cache doesn't have the content asked from CPU

GOAL: to reduce "MISS" rates

We are exploiting temporal locality properties.

2.1 when we take the content from RAM, the cache takes a block (chunck) of data

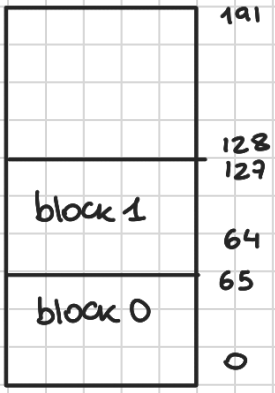
WRITE

CPU wants to write into the memory at the address "x".

1. write in cache only
2. = in memory too

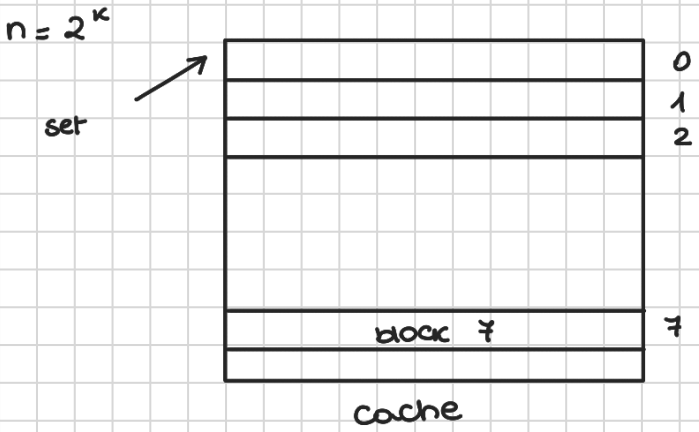
You would want to do the 2nd one, but typically we write only on cache. If its a lot to write and cache needs to save blocks, you write on memory (DEALLOCATE BLOCK)

- When you remove a block from the cache



Let's say block is 64 bytes

only fixed blocks



Let's say we want to load block 7 in cache

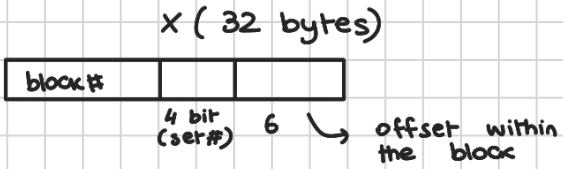
- we load it into set 7

STATICAL ALLOCATION

- we load block only if CPU requests it (this happens during MISS situation) and we load it in its according set

$n = 2^k$ # sets

block made of 64 bytes



HOW CAN I COMPUTE THE NUMBER OF BLOCK STARTING FROM "X"?

$$block\# = \frac{X}{64}$$ (easier in binary)

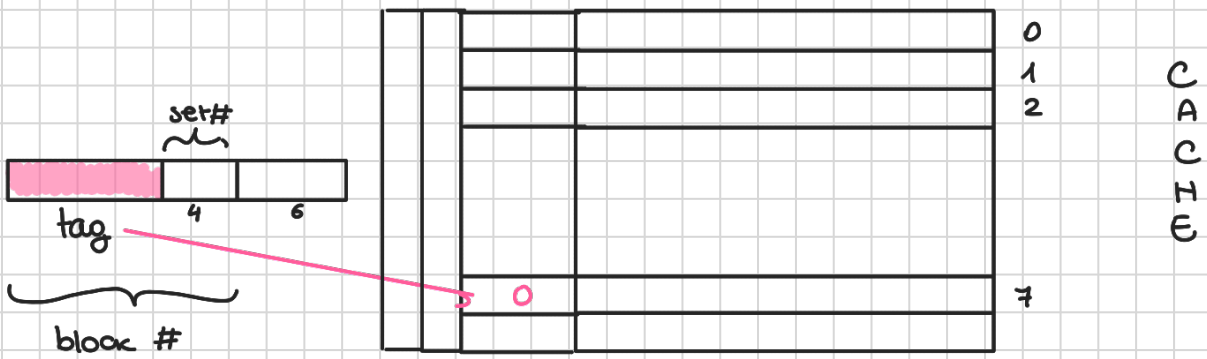
64 ← block bytes

HOW TO KNOW WHICH IS THE SET# OF BLOCK#?

- Let's say we have 16 sets.

set# = block # % n^2 sets

HOW TO BE SURE SET 7 CONTAINS BLOCK 7 ?
(you can't put the whole block number)



dirty | validity | tag (where you put the tag part of the block #)

example: 7 → 0..... 0101

tag = 0

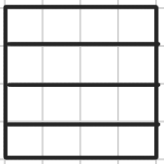
→ set n° (set 7)

- We need one bit to determine if the block content is meaningful or if its empty. This is the "validity" part of the cache.

- We need another bit to know whether the content written in cache has been written inside the memory as well

  (bit = 0 if RAM has been written as well)

EXERCISE TIME GUYS



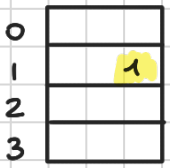
cache → 4 sets
block → 32 bytes

M = Miss

M H M M M H H M H
36, 48, 0, 100, 164, 172, 96, 92, 16

Suppose each one of those are addresses the CPU wants to access

- **36.** The first one is a Miss (M) 'cause cache empty



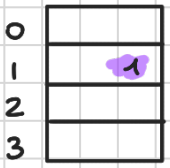
Block # = $\frac{36}{32}$ = 1 so block 1
(32 ↖ block bytes)

Since it's a M, we load "block 1" from memory to a set:

Set# = 1 % 4 = 1 so set 1
(4 ↑ n° sets)

- **48.** This is a HIT (H)

Block # = $\frac{48}{32}$ = 1 so block 1 (inside cache)



Set# = 1 % 4 = 1

- **0.** This is MISS (M)

block # = $\frac{0}{32}$ = 0 no block 0 in cache

0	0
1	1
2	
3	

Since it's a M, we load "block 0" from memory to a set:

Set# = 0 % 4 = 0 so set 0

- **100.** This is MISS (M)

block # = $\frac{100}{32}$ = 3 no block 3 in cache

0	0
1	1
2	
3	3

Since it's a M, we load "block 3" from memory to a set:

Set# = 3 % 4 = 3 so set 3

- **164.** This is MISS (M)

block # = $\frac{164}{32}$ = 5 no block 5 in cache

0	0
1	5 ~~1~~
2	
3	3

Since it's a M, we load "block 5" from memory to a set:

Set# = 5 % 4 = 1 so set 1

We have to overwrite set 1 and put block 5 now, no block 1

- **172.** This is a HIT (H)

Block # = $\frac{172}{32}$ = 5 so block 5 (inside cache)

0	0
1	5
2	
3	3

Set# = 5 % 4 = 1

- **96.** This is a HIT (H)

Block # = $\frac{96}{32}$ = 3 so block 3 (inside cache)

0	0
1	5
2	
3	3

Set# = 3 % 4 = 3

- 92. This is MISS (M)

block # = $\frac{92}{32}$ = 2 — no block 2 in cache

0	0
1	5
2	2
3	3

Since it's a M, we load "block 2" from memory to a set:

Set# = 2 % 4 = 2 so set 2

- 16. This is a HIT (H)

Block # = $\frac{16}{32}$ = 0 so block 0 (inside cache)

0	0
1	5
2	2
3	3

Set# = 0 % 4 = 0

Cache → 8 sets
Block → 64 bytes

0x00010000	0
	1
	2
	3
	4
	5
	6
	7

- data

```
x: .word 5, 6, 2, 0, -2, 5,
n: .word 1024
```

.text

```
li  t3, 0
lui t0, 0x10010
lw  t1, 0x01000 (t0)   # get 1024
```

0x00400000
0x00010000

```
ciclo: lw  t2, 0(t0)
       add  t3, t2, t2
       addi t0, t0, 4
       addi t1, t1, -1
       bne  t1, zero, ciclo
```

- loads 1st instruction
- looks for block number