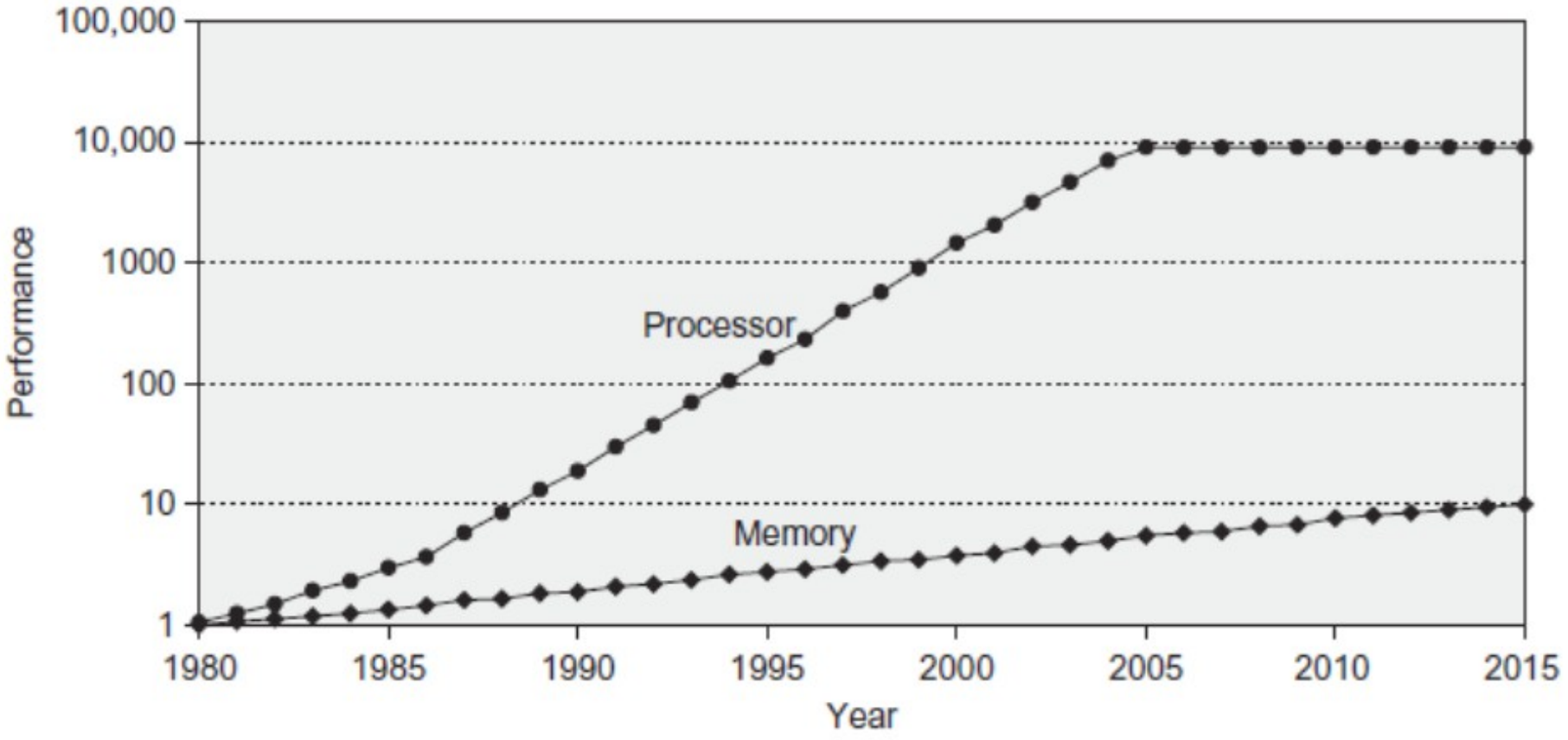
Cache memory

Raymond Namyst

Pierre-André Wacrenier

Elisabeth Brunet

Why caches ?



Why caches ?

- To achieve promised performance of brand new processors,
- data need to be right here right now near it.
  - The faster the memory, the more expensive and the smallest it is

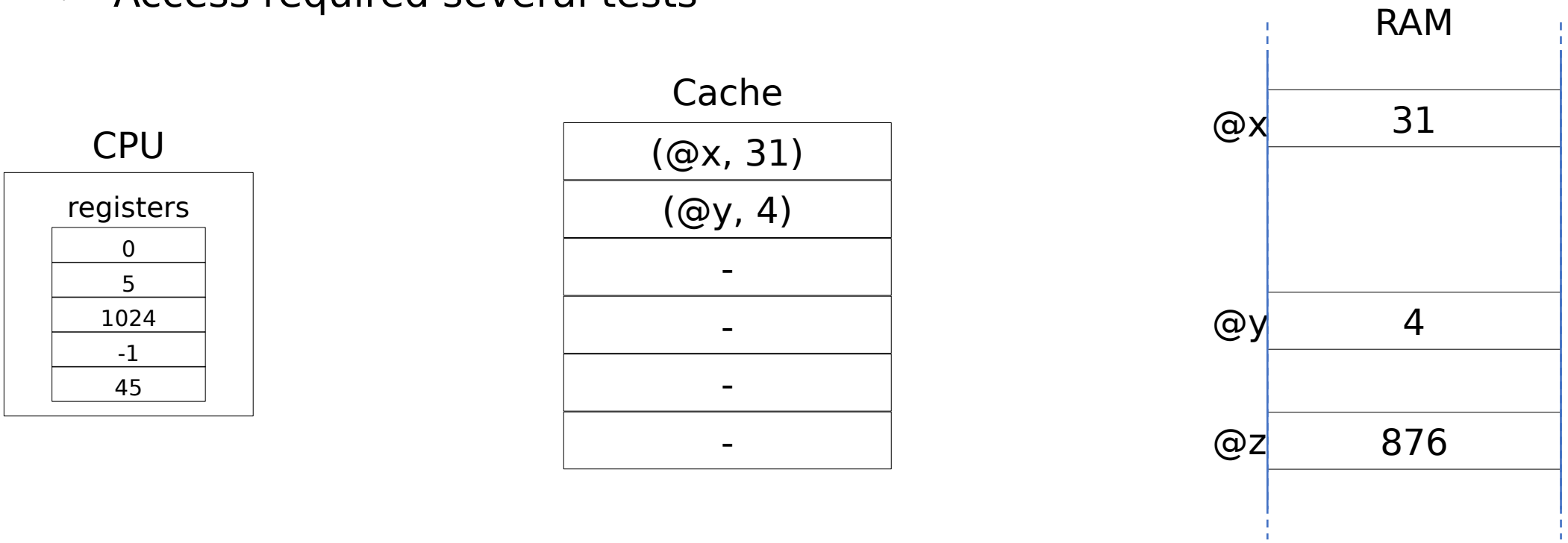
- Temporal locality
  - Many repeated accesses to the same variables, e.g in loops

- Spatial locality
  - Concentration of accesses on nearby data
    - Structures, arrays, local variables

- 90/10 rule ‘90% of execution takes place in 10% of the code’.

As first approximation…

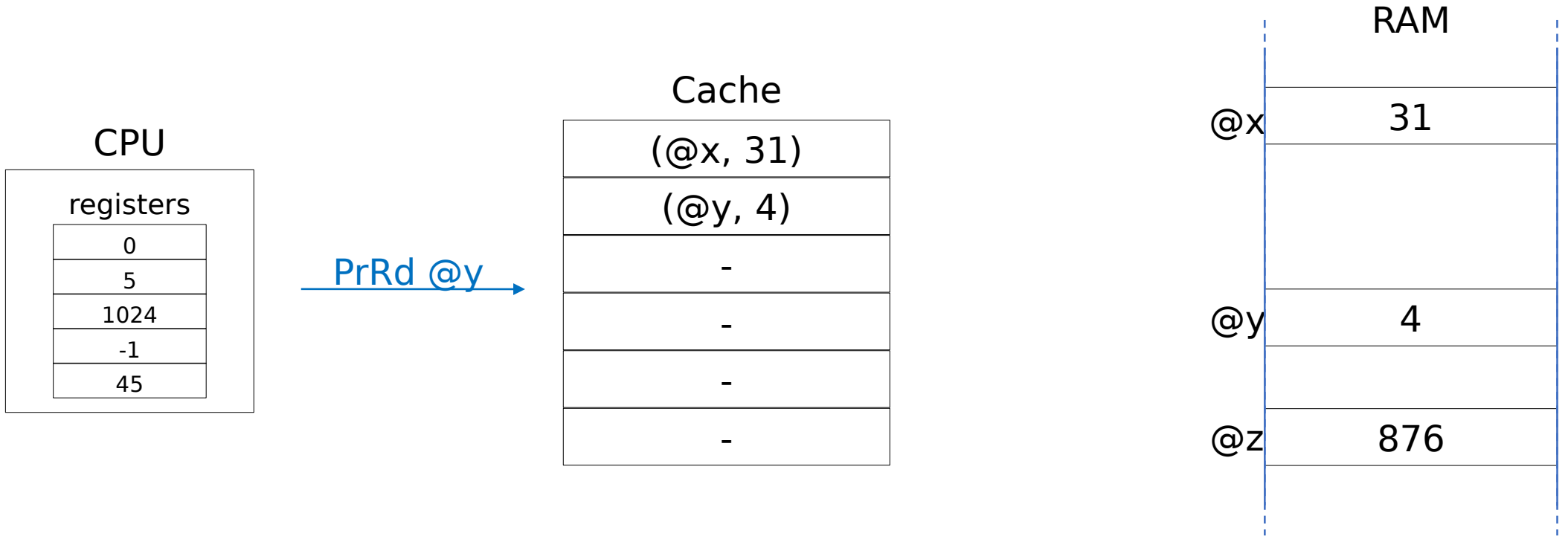
- The cache stores the most ‘useful’ data
  - Pairs storage(address, content) like in page table for RAM
  - Access required several tests



As first approximation…

When CPU emits a reading request, cache intercept it

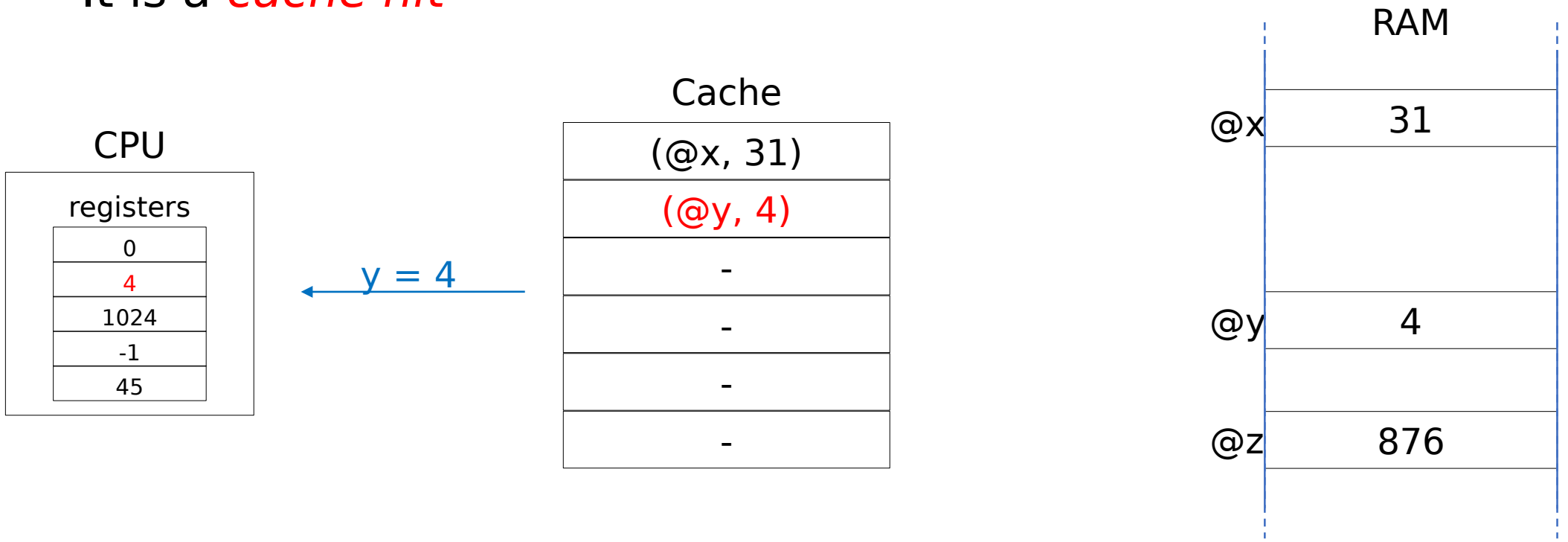
- if data is owned, it answers



As first approximation…

When CPU emits a reading request, cache intercept it

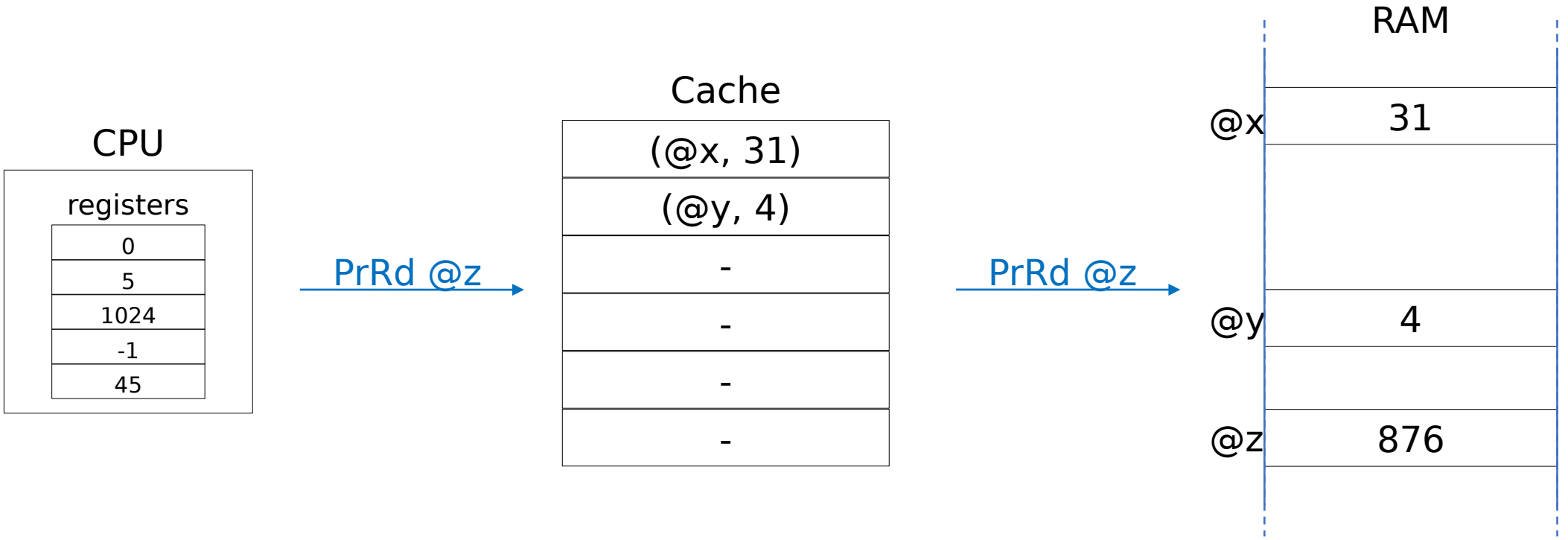
- if data is owned, it answers
  - It is a *cache hit*



As first approximation...

When CPU emits a reading request, cache intercept it

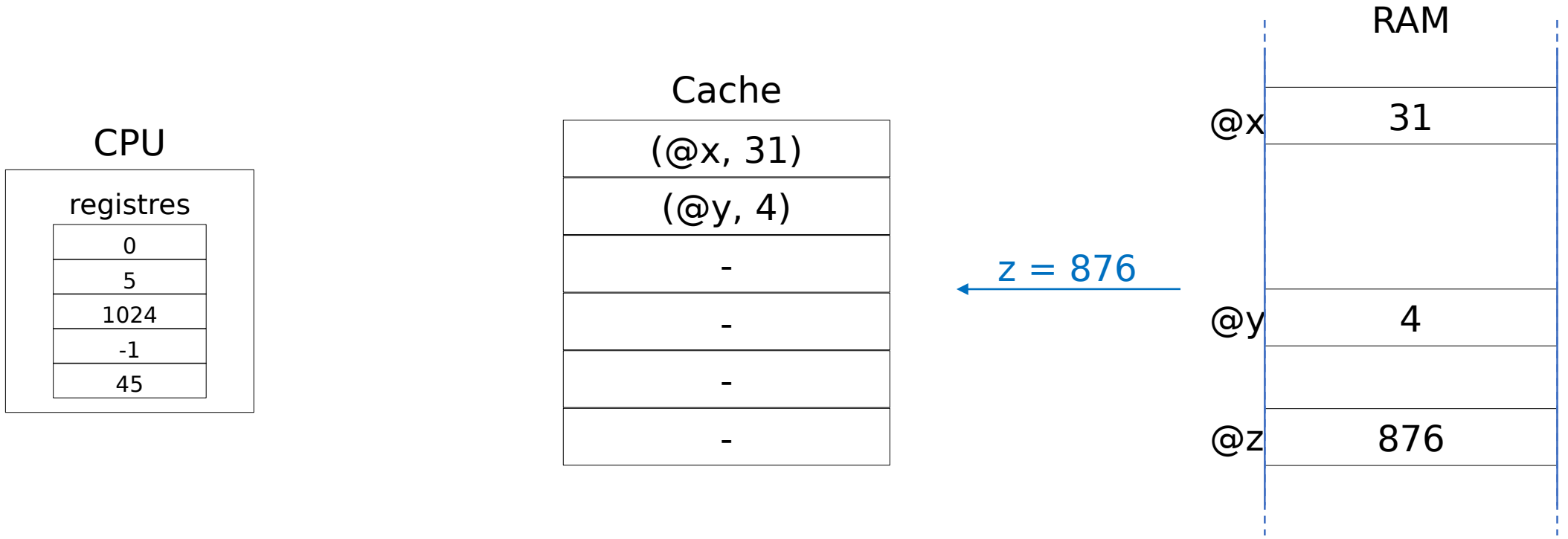
- If data is not owned, the request is forwarded to the RAM
- It is a *cache miss*



As first approximation...

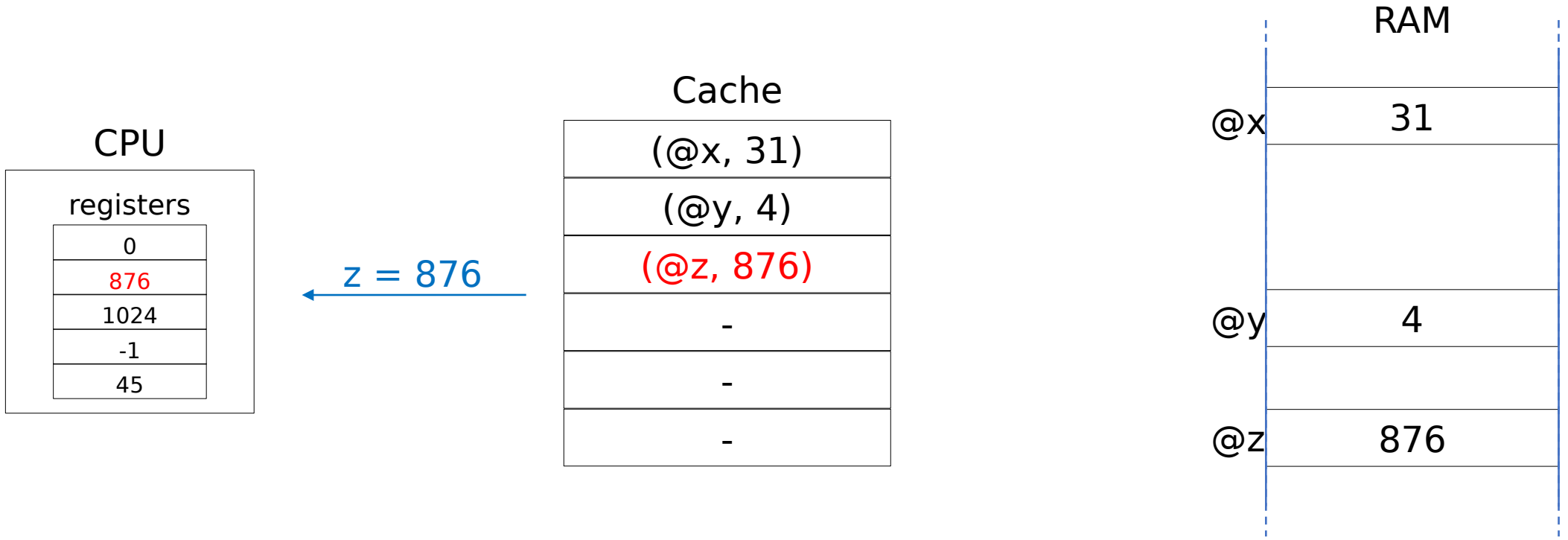
When CPU emits a reading request, cache intercept it

- If data is not owned, the request is forwarded to the RAM
- It is a *cache miss*



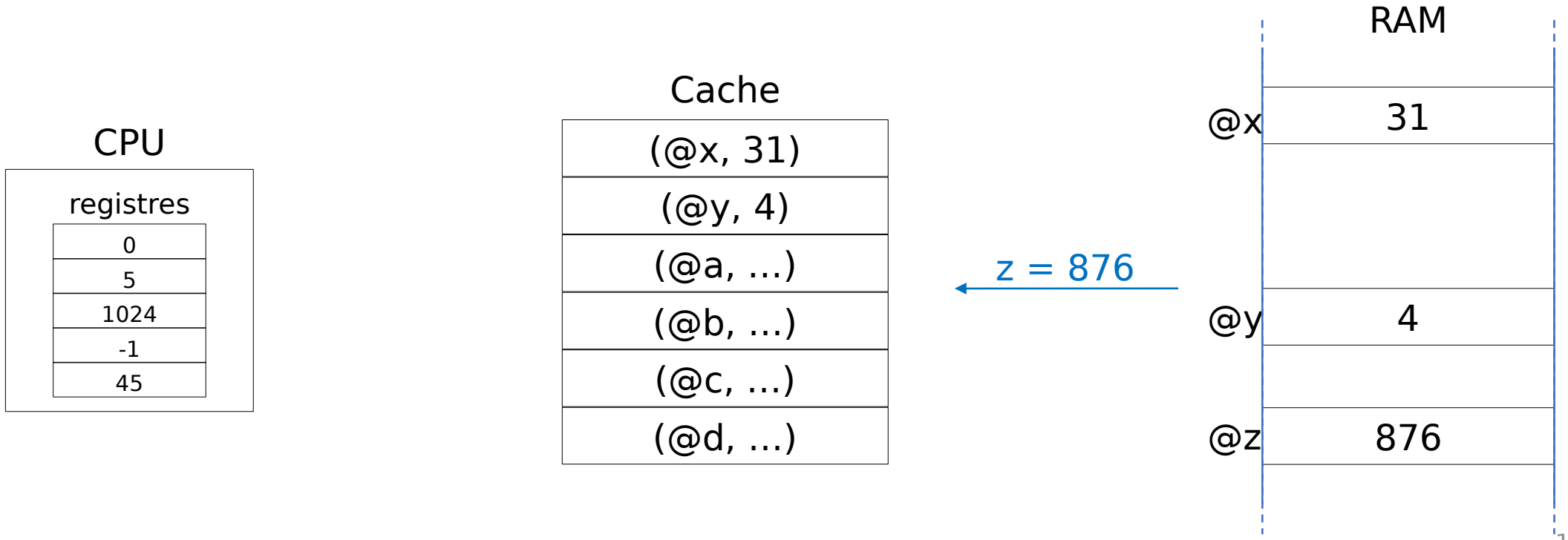
As first approximation...

- When RAM answers
  - Cache keeps a copy and forwards it to the CPU



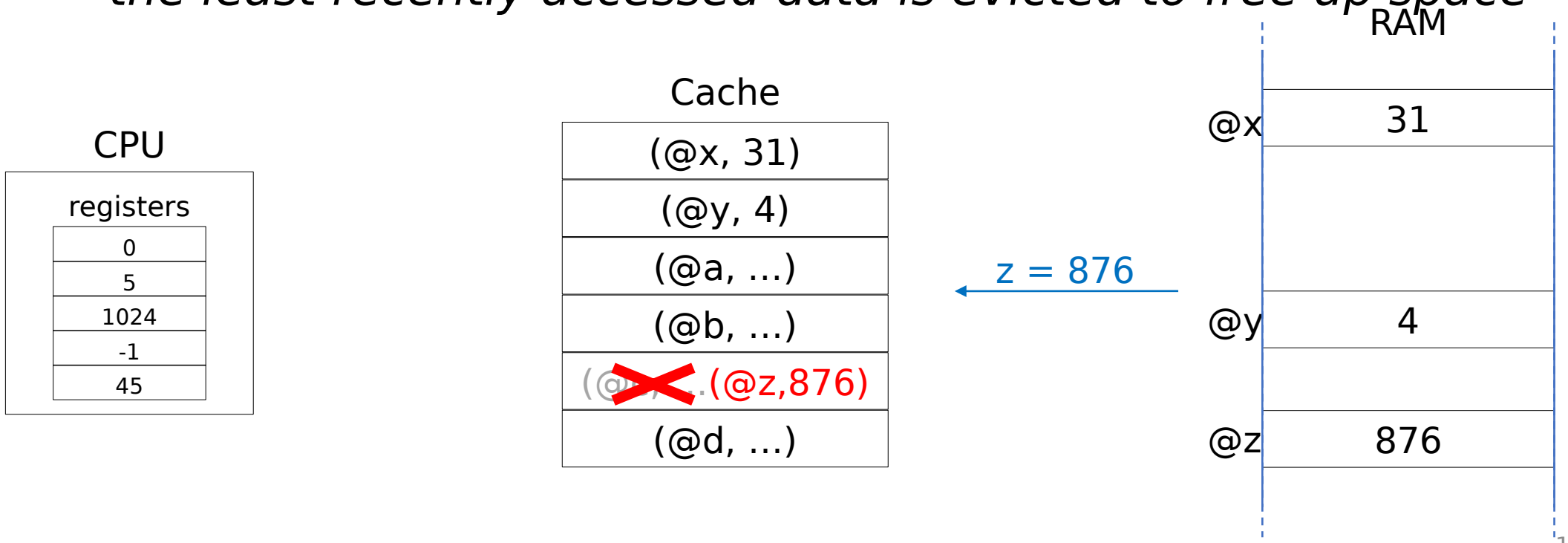
As first approximation…

- What happens when cache is full ?



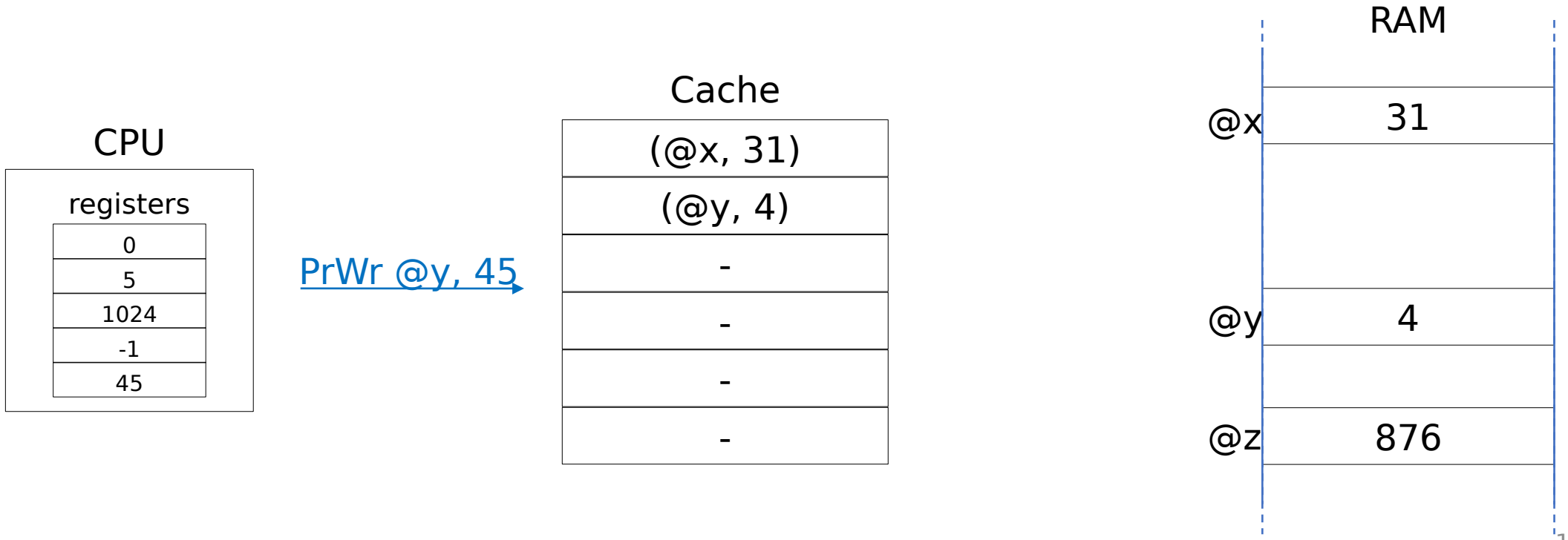
As first approximation...

- Temporal locality applied
  - Strategy LRU (*Least Recently Used)*
  - *the least recently accessed data is evicted to free up space*



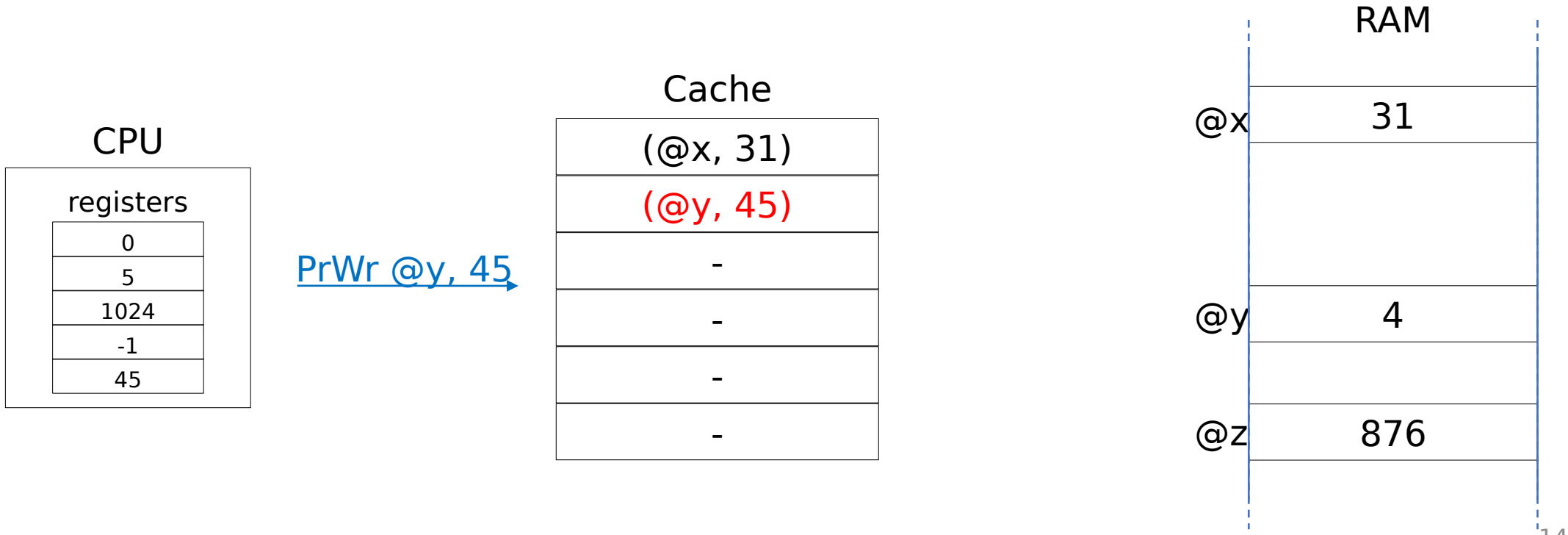
As first approximation…

- When CPU emits a writing request,
  - If cache owns the data…



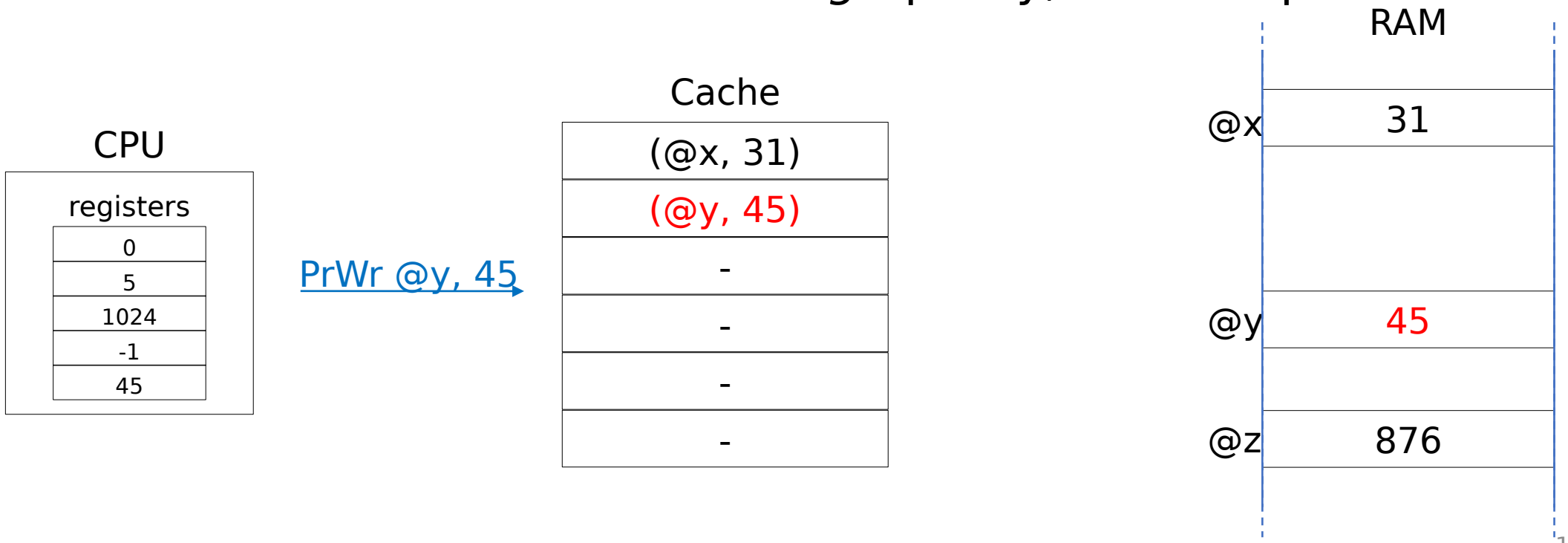
As first approximation...

- When CPU emits a writing request,
  - If cache owns the data ... cache is updated



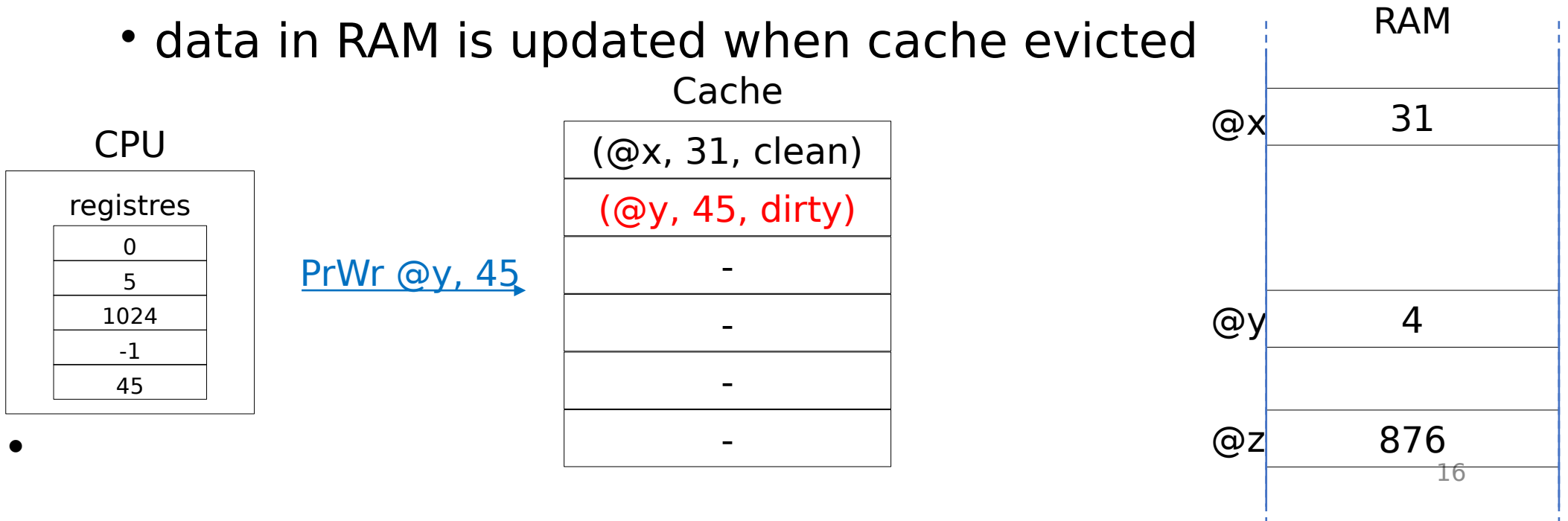
As first approximation…

- When CPU emits a writing request,
  - If cache owns the data … cache is updated
    - If RAM follows a *write through* policy, data is updated



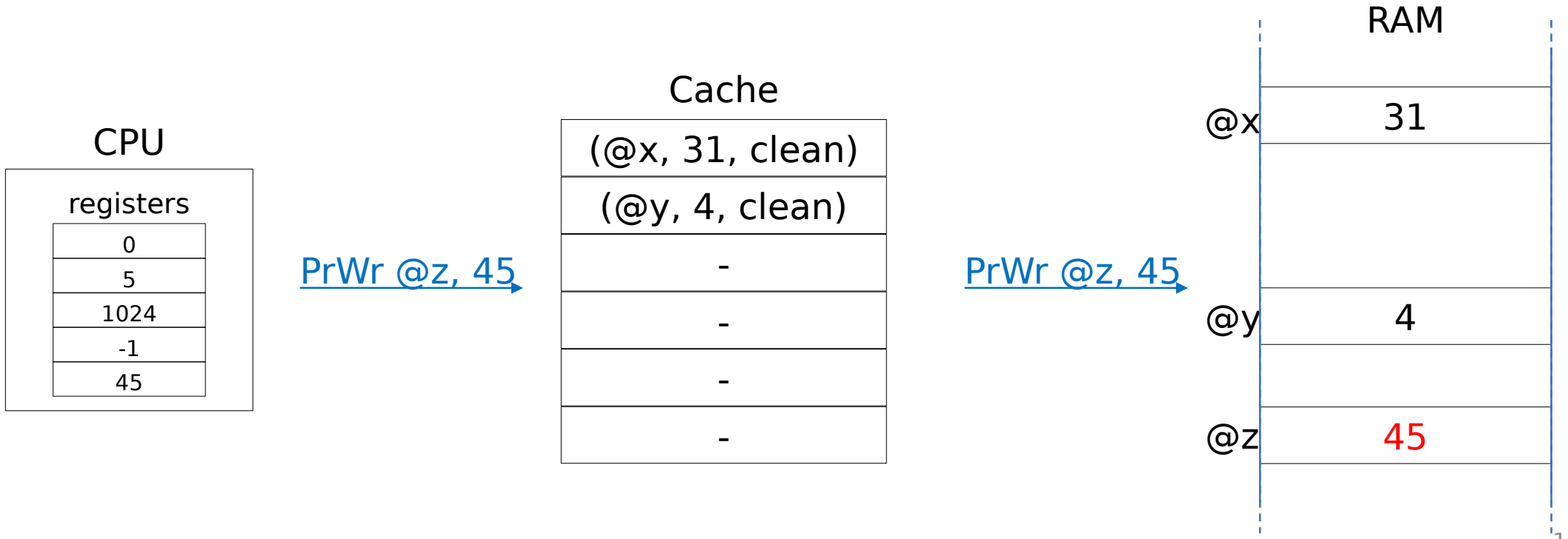
As first approximation…

- When CPU emits a writing request,
  - If cache owns the data … cache is updated
    - If RAM follows a *write back* policy
      - data in cache becomes obsolete : clean/dirty
      - data in RAM is updated when cache evicted



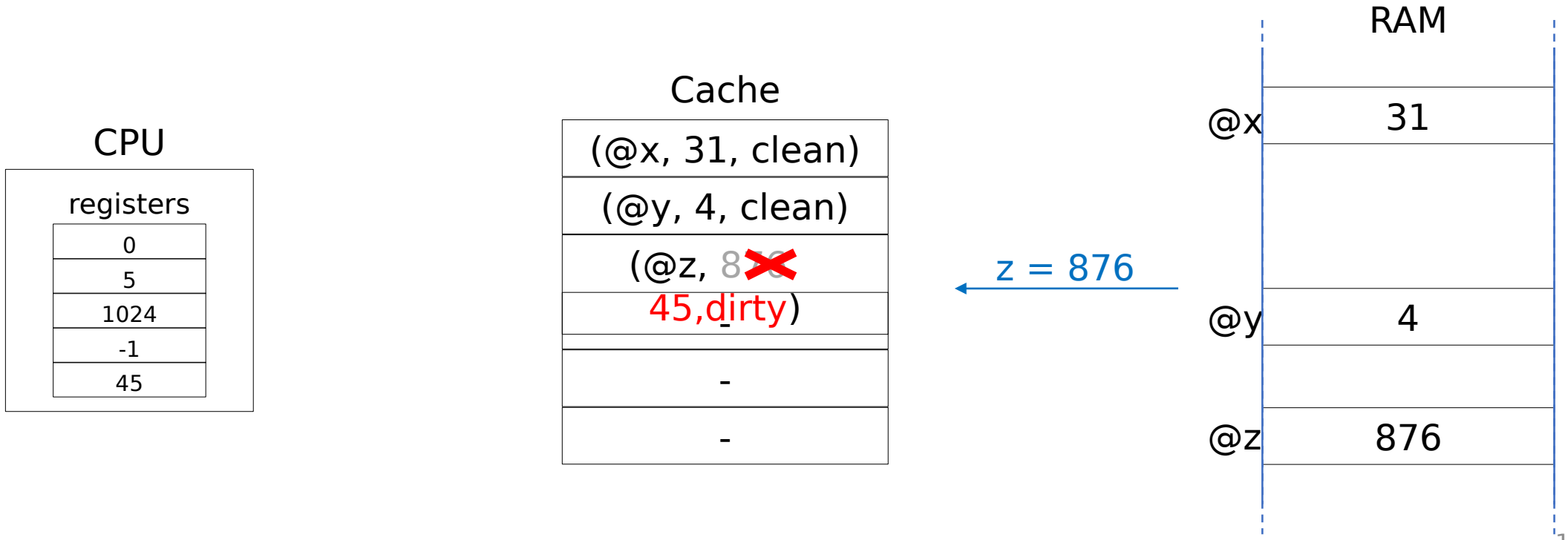
As first approximation…

- When CPU emits a writing request,
  - If cache does not own the data ….
    - Cache forwards it to RAM

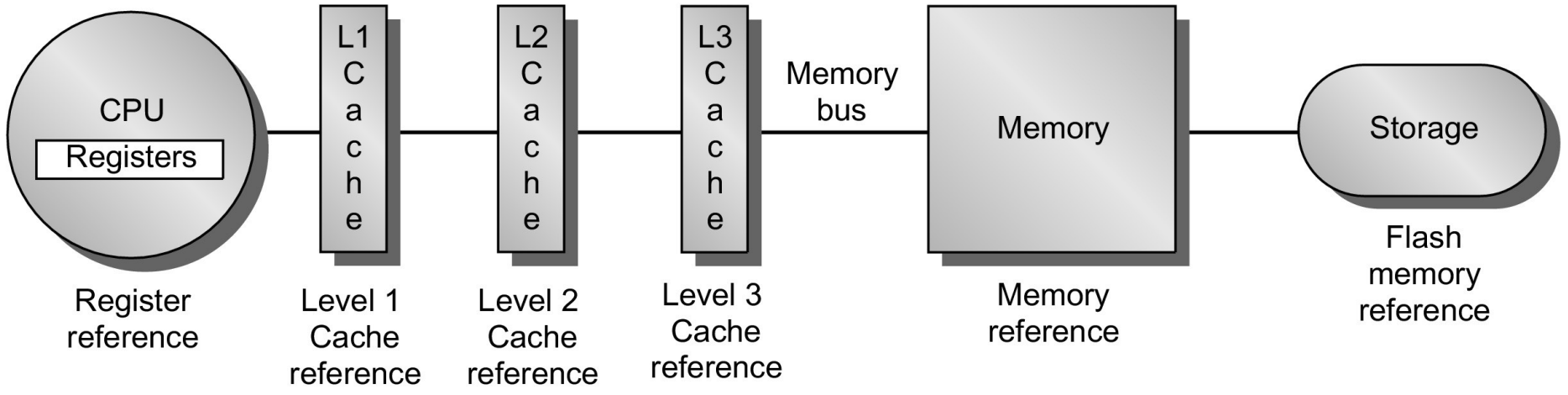


As first approximation...

- When CPU emits a writing request,
  - If cache does not own the data...
    - Cache forwards it to RAM and loads it from RAM in cache



Memory hierachy



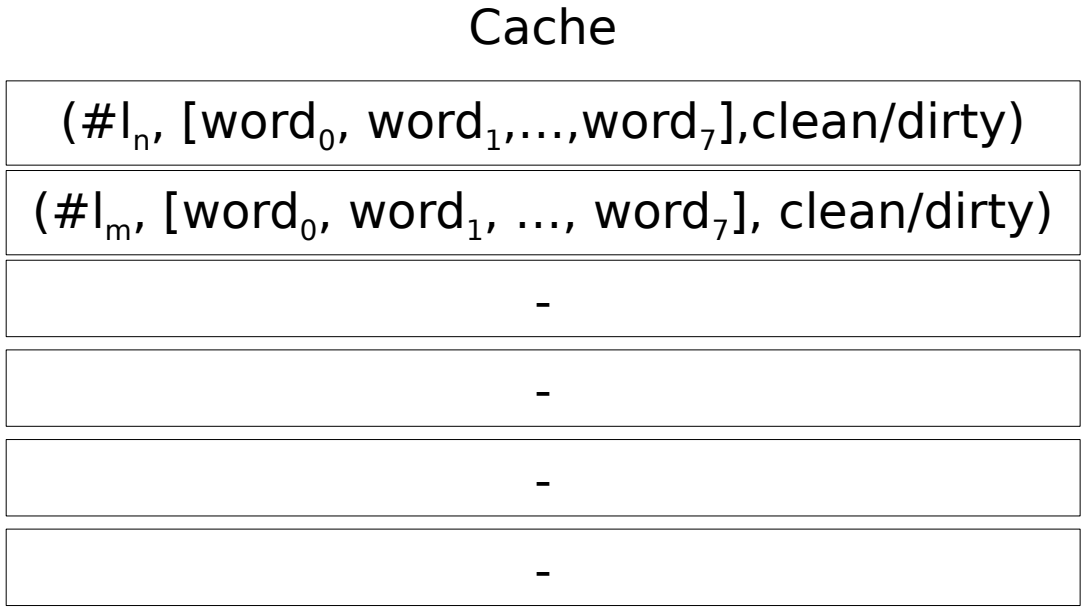
Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4–16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS

(B) Memory hierarchy for a laptop or a desktop

Caches are usually inclusives (i.e. L1 cache is included in L2 cache)

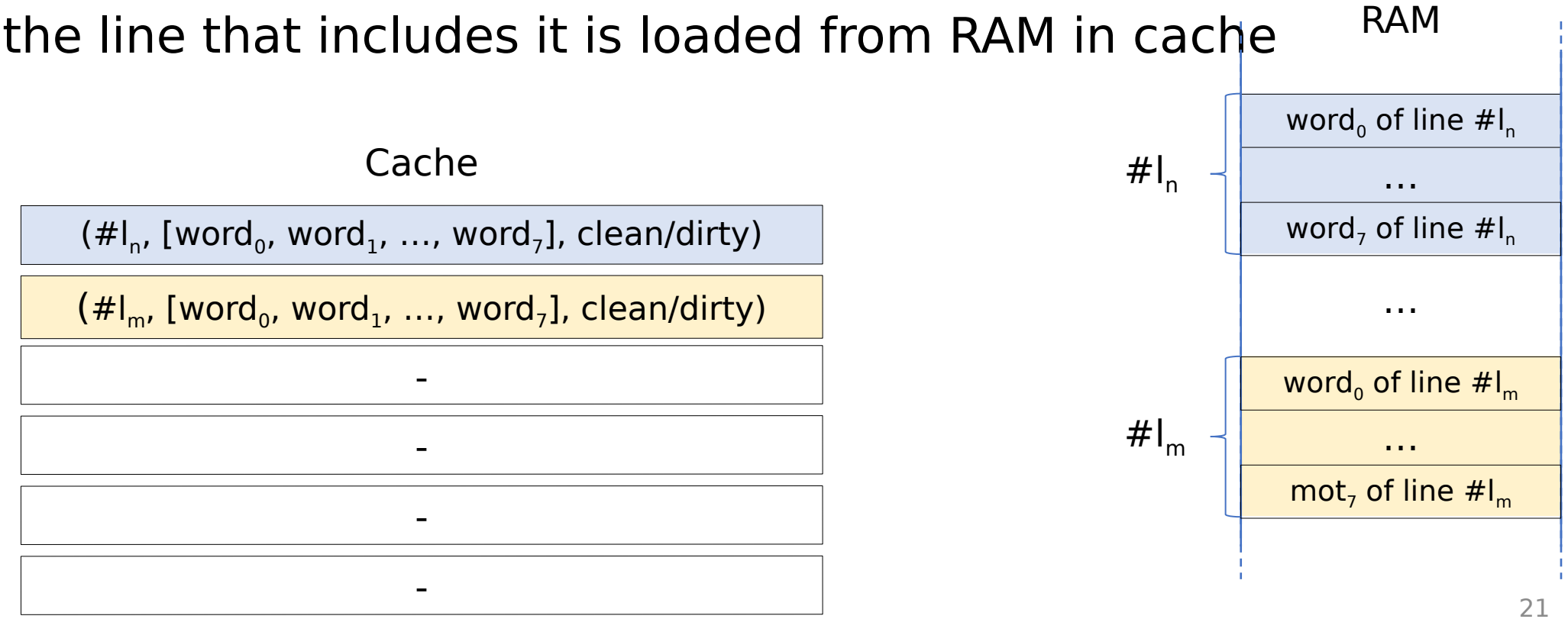
Spatial locality

- Caches store lines of typically 64 bytes
  - On a 64bits architecture,
    a cache line of 64 bytes contains
    - 8 double / long
    - 16 float / int



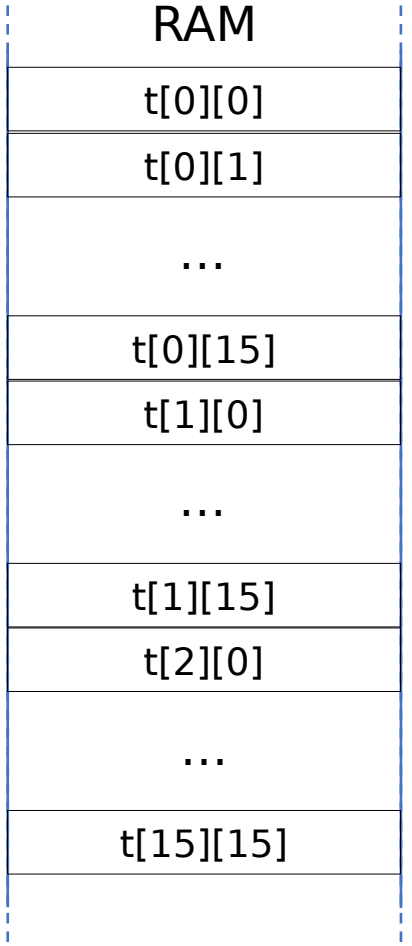
Memory is a sequence of lines

- Cache works only with lines
  - When CPU requests a missing data,
  - the line that includes it is loaded from RAM in cache



Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];
- t sizes 16 x 16 x 4 bytes (1 KB)



Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int i = 0; i < 16; i++)
   for (int j = 0; j < 16; j++)
      sum += t[i][j];
```

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int i = 0; i < 16; i++)
   for (int j = 0; j < 16; j++)
      sum += t[i][j];
```

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int i = 0; i < 16; i++)
   for (int j = 0; j < 16; j++)
      sum += t[i][j];
```

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int i = 0; i < 16; i++)
   for (int j = 0; j < 16; j++)
      sum += t[i][j];
```

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int i = 0; i < 16; i++)
   for (int j = 0; j < 16; j++)
      sum += t[i][j];
```

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

  ```
  for (int i = 0; i < 16; i++)
     for (int j = 0; j < 16; j++)
        sum += t[i][j];
  ```

- By loop i, one cache miss to access t[i][0] and cache hits next

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

  ```
  for (int j = 0; j < 16; j++)
     for (int i = 0; i < 16; i++)
        sum += t[i][j];
  ```

- With a cache of 8 lines,
  - j=0, i=0..7 → cache miss
  - and cache is full !

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

  ```
  for (int j = 0; j < 16; j++)
     for (int i = 0; i < 16; i++)
        sum += t[i][j];
  ```

- With a cache of 8 lines,
  - j=0, i=0..7 → cache miss
  - and cache is full !

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int j = 0; j < 16; j++)
   for (int i = 0; i < 16; i++)
      sum += t[i][j];
```

- With a cache of 8 lines,
  - j=0, i=8 : first eviction !

Cache

#l_{n+8}:	t[8][0], t[8][1], ..., t[8][15]
#l_{n+1}:	t[1][0], t[1][1], ..., t[1][15]
#l_{n+2}:	t[2][0], t[2][1], ..., t[2][15]
#l_{n+3}:	t[3][0], t[3][1], ..., t[3][15]
#l_{n+4}:	t[4][0], t[4][1], ..., t[4][15]
#l_{n+5}:	t[5][0], t[5][1], ..., t[5][15]
#l_{n+6}:	t[6][0], t[6][1], ..., t[6][15]
#l_{n+7}:	t[7][0], t[7][1], ..., t[7][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int j = 0; j < 16; j++)
   for (int i = 0; i < 16; i++)
      sum += t[i][j];
```

- With a cache of 8 lines,
  - j=0, i=9 : second eviction

Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+9}:	t[9][0], t[9][1], …, t[9][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

Importance of data layout

- Arrays in C are linearized in *row-major* indexing
- Example :
  float t [16][16];

```
for (int j = 0; j < 16; j++)
   for (int i = 0; i < 16; i++)
      sum += t[i][j];
```

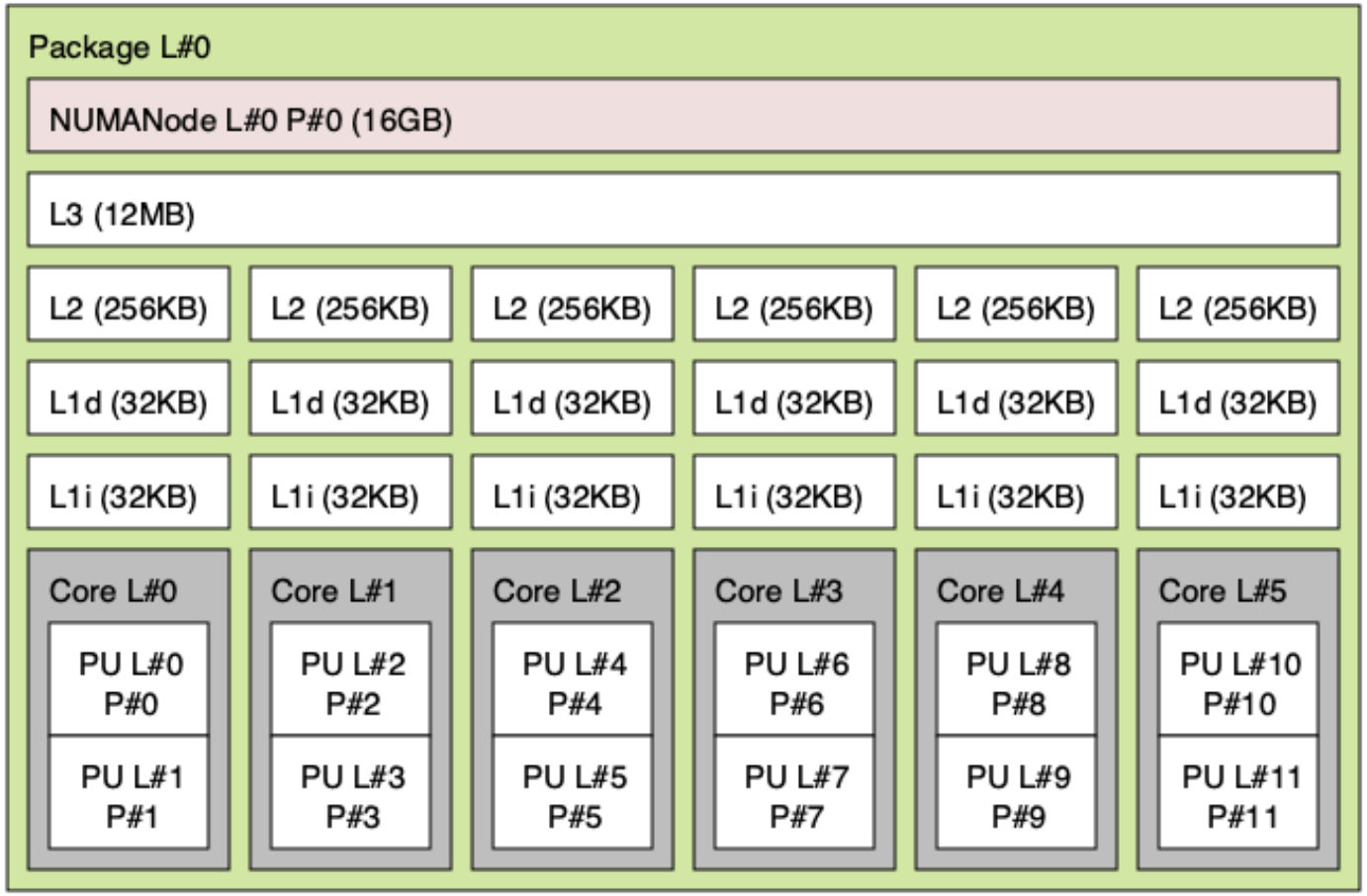
Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+9}:	t[9][0], t[9][1], …, t[9][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

- Each next access leads to an eviction

  → Better with a row-major path

Caches and multicores

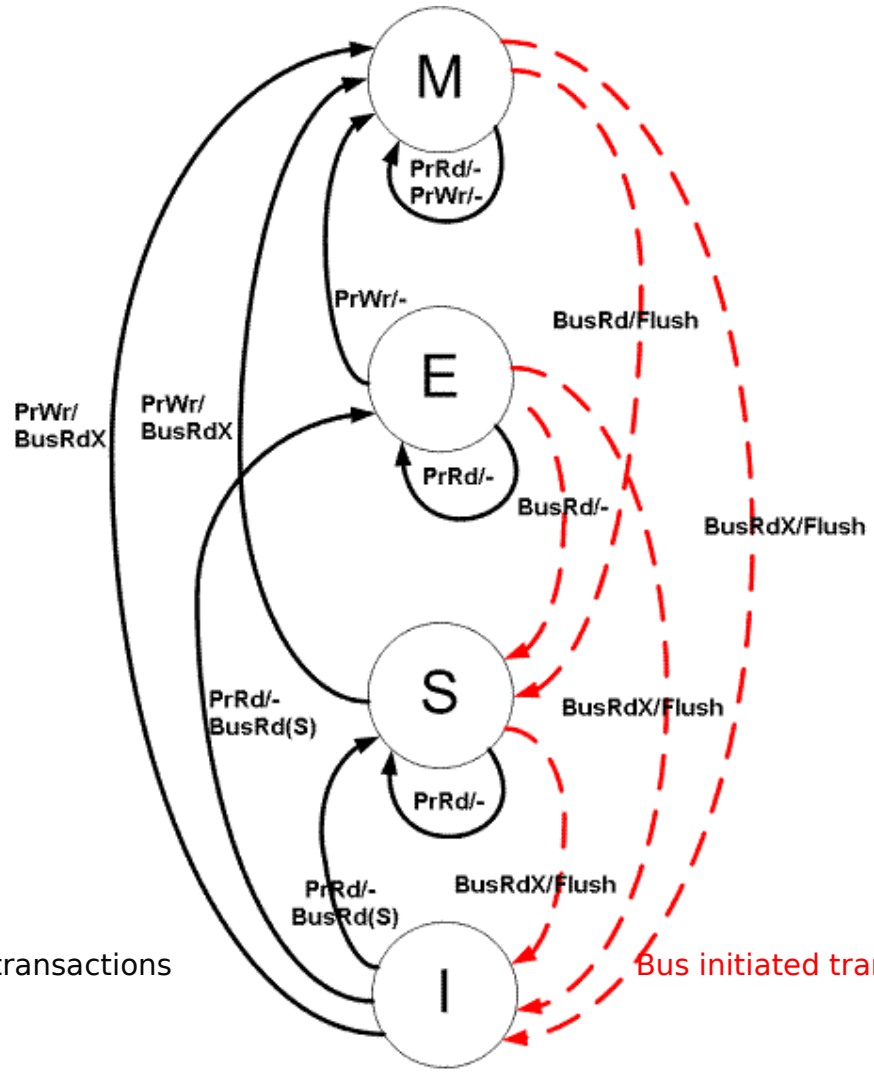


Caches and multicores

- Memory hierarchy are different from an architecture to another
- Some caches are private, some are shared between a subset of CPU (or all)
- Data coherency has to be maintained
  - Several protocols have been designed
    - MSI, MESI, MOESI, MESIF, etc.

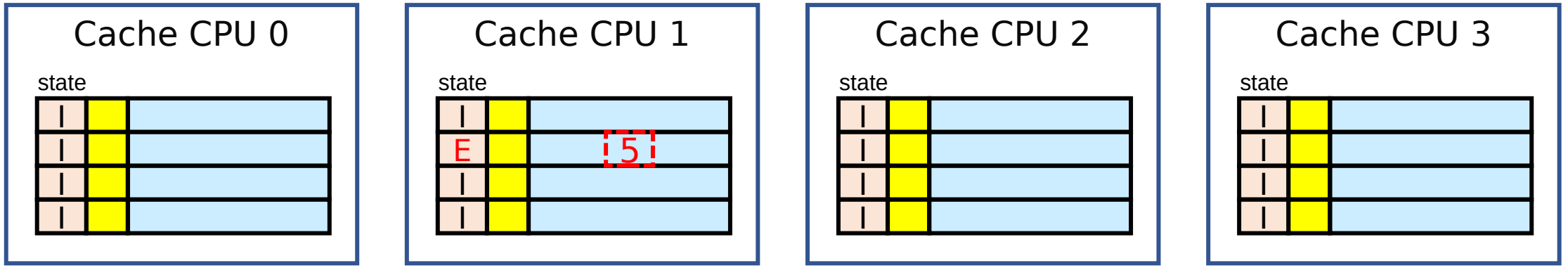
Caches and multicores

- Protocol MESI
  - Each cache line can be in one of those 4 states :
    - M (modified)
      - Line is in this cache and is *dirty*
    - E (exclusive)
      - Line is in this cache and is *clean*
    - S (shared)
      - Line is in another cache and is c*lean*
    - I (invalid)
      - Line is empty



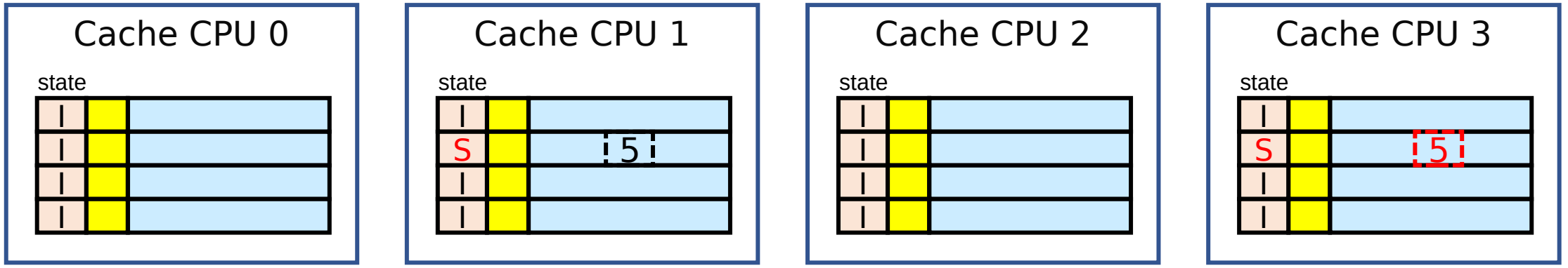
Caches and multicores

- CPU_1: PrRd (x) (whose value is 5 in RAM)
  - Signal BusRd is sent on the bus
  - If others caches answer they do not own a copy
    - Cache line is loaded from RAM and its state is *Exclusive*



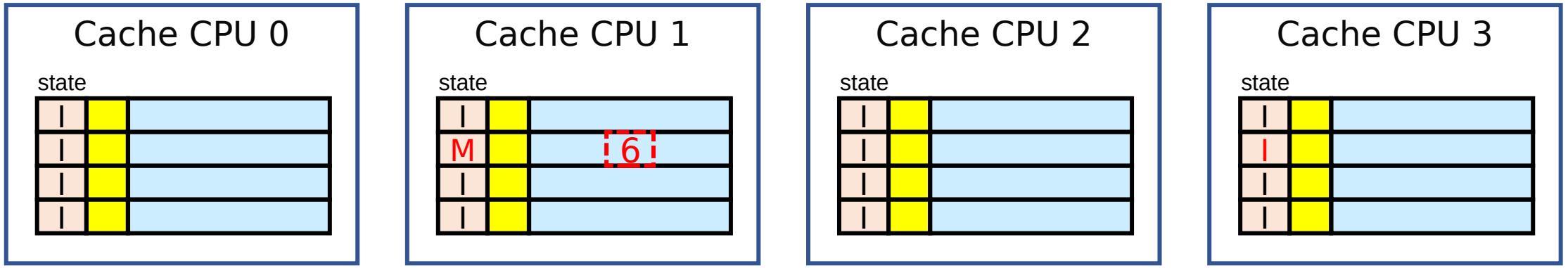
Caches and multicores

- CPU_3: PrRd (x)
  - Signal BusRd is sent on the bus
  - CPU_1 sent the cache line to CPU_3
  - Cache line state becomes *Shared* for CPU_1 and CPU_3



Caches and multicores

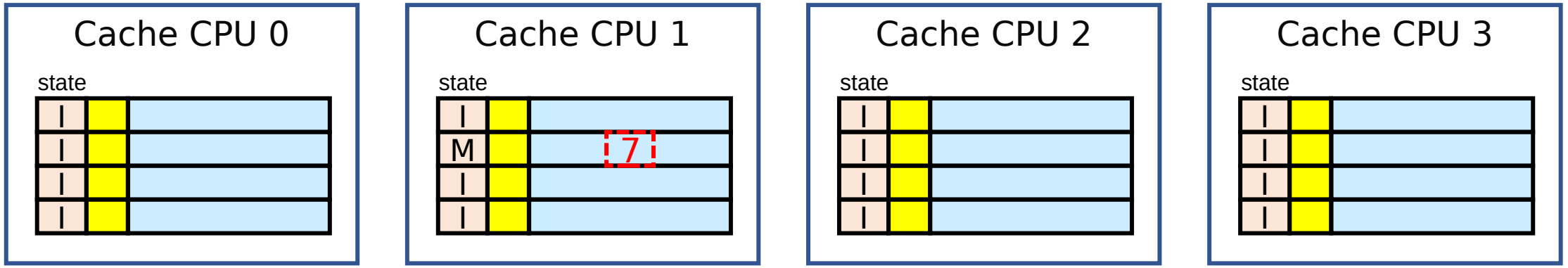
- CPU_1: PrWr (x, 6)
  - Signal BusRdX is sent on the bus
  - CPU_3 invalidate its copy
  - Cache line on CPU_1 becomes *Modified*



Caches and multicores

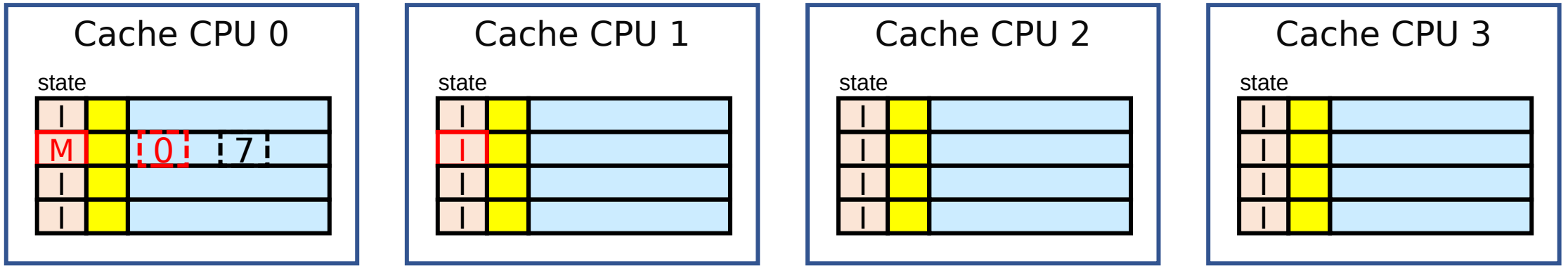
CPU_1: PrWr (x, 7)

- No signal sent on the bus
- Cache line remains *Modified*



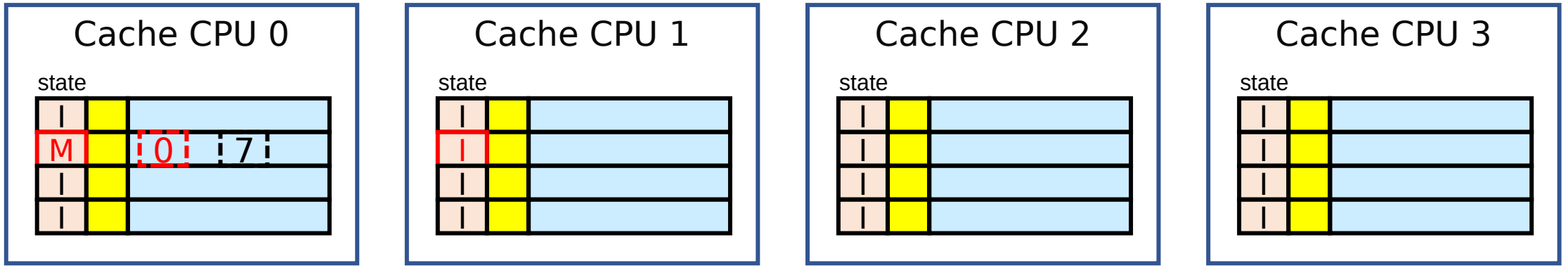
Caches and multicores

- CPU_0: PrWr (y, 0), y and x being in the same cache line
  - Signal BusRdX is sent on the bus
  - CPU_1 sends cache line to CPU_0 and invalides its copy
  - Cache line becomes *Modified* on CPU_0



Caches and multicores

- If CPU_1 modifies x, and then CPU_0 modifies y, etc.
  - Ping-pong effect between caches
  - This is false-sharing !
    - Introduction of padding needed



Caches and multicores

- Knowing that, the effectiveness of a code such as is debatable :

```
float tab [N];

#pragma omp parallel for schedule(static, 1)
      for (int i = 0; i < N; i++)
            tab [i] = f (i);
```