# Caching

# Caching

**cache** /kæʃ/: noun.

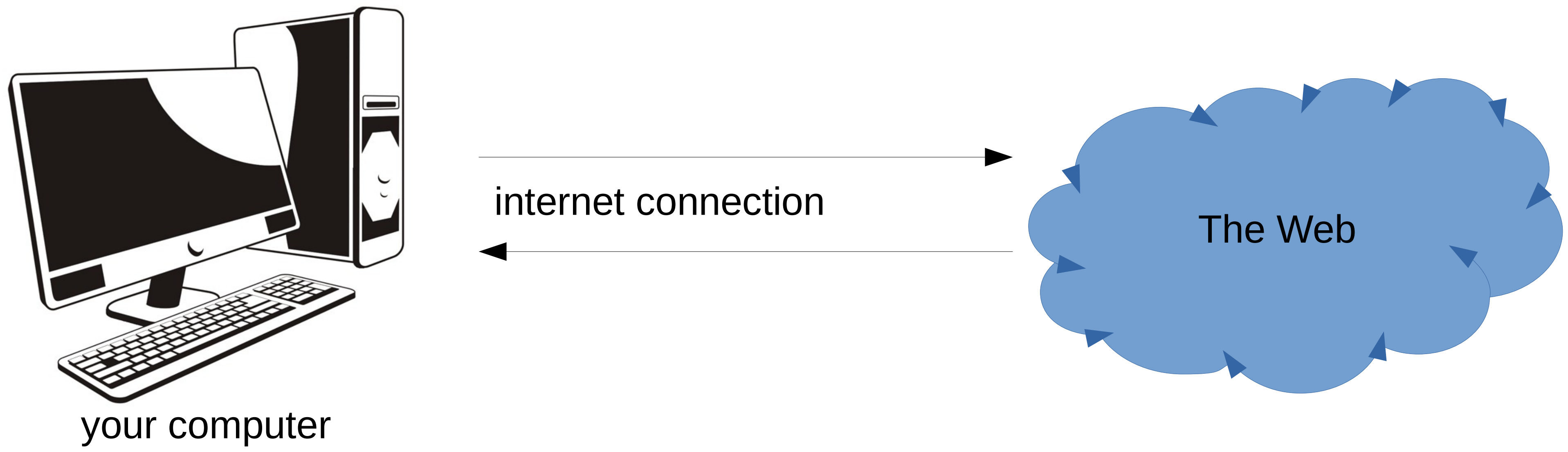
1. a collection of items of the same type stored in a hidden or inaccessible place. "an arms cache".

2. an auxiliary memory from which high-speed retrieval is possible.

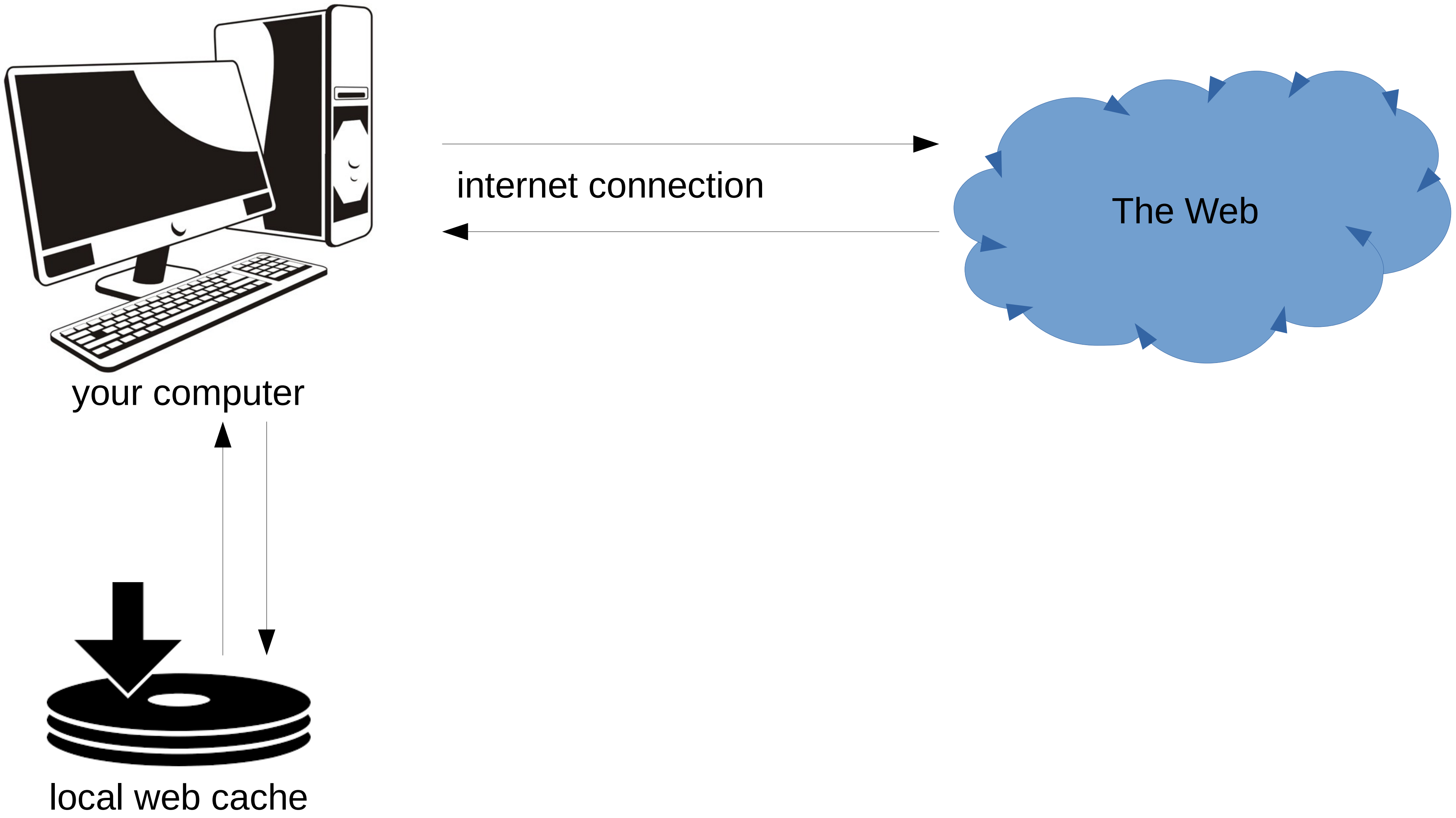
From French Canadian slang meaning "hiding place"; from French *cacher* "to hide, conceal."

Not to be confused with **cash** /kæʃ/: noun, from French *caisse* "money box";
from Italian *cassa* "box"

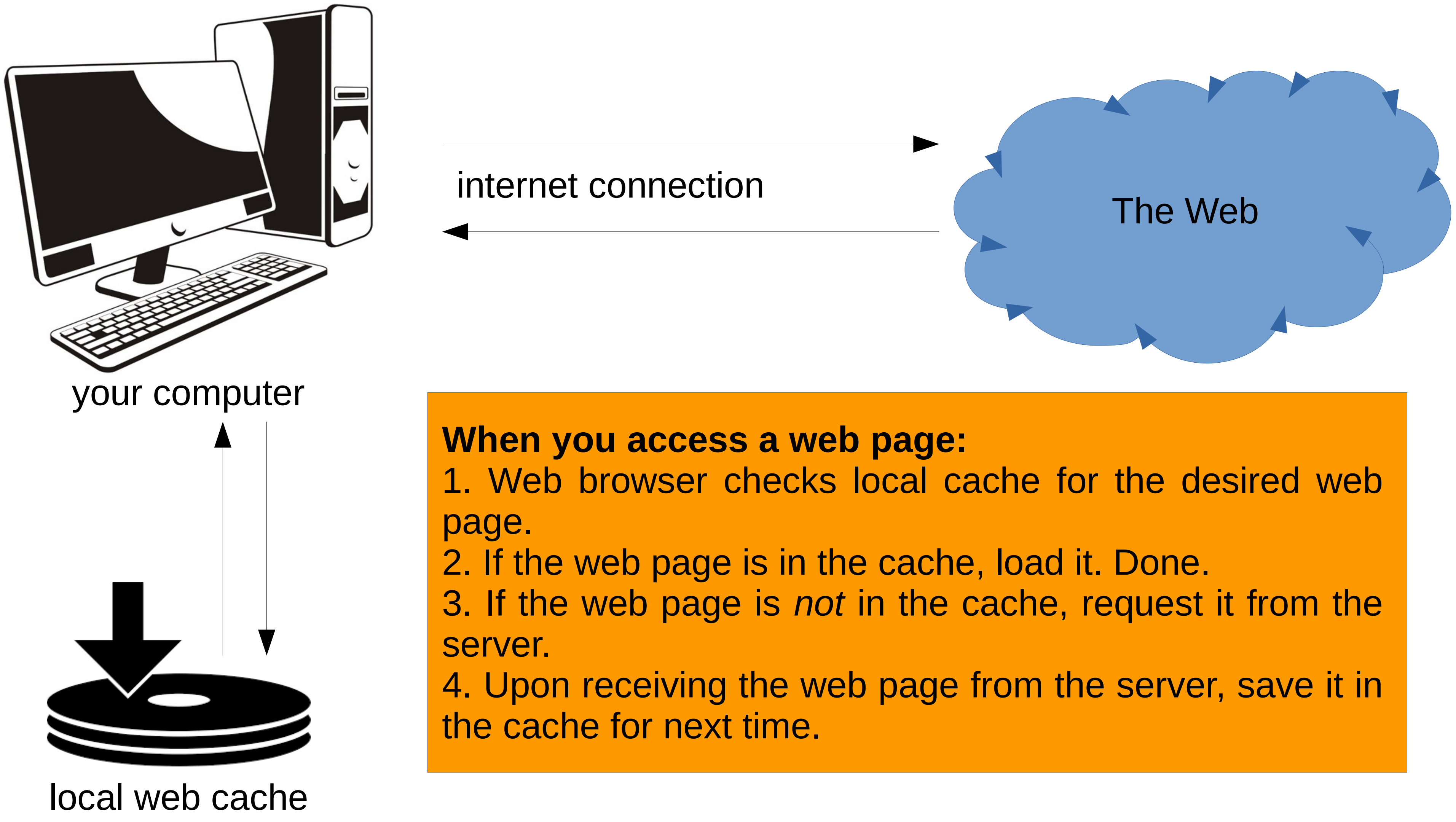
One kind of cache: a web browser cache



One kind of cache: a web browser cache

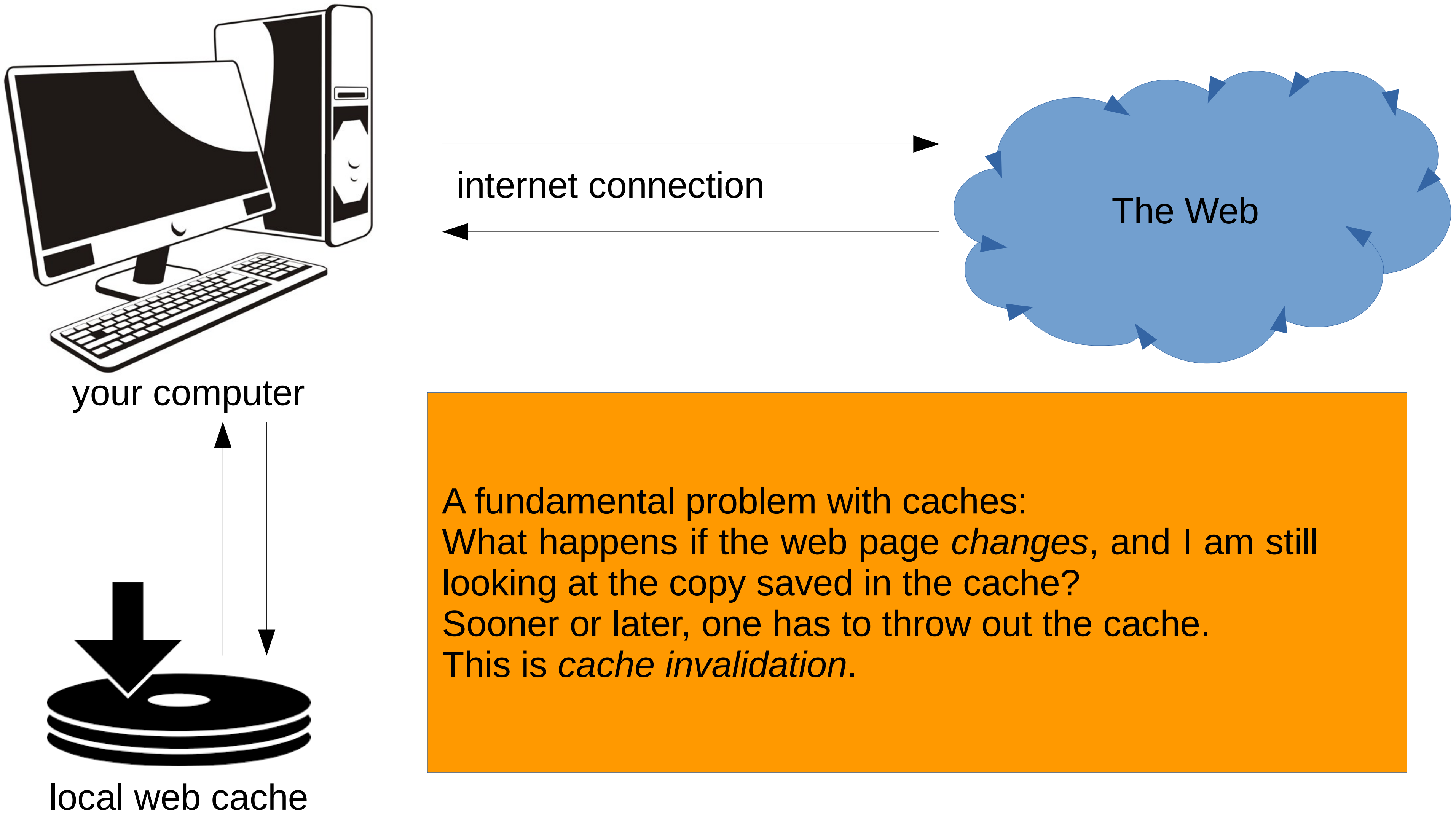


## One kind of cache: a web browser cache



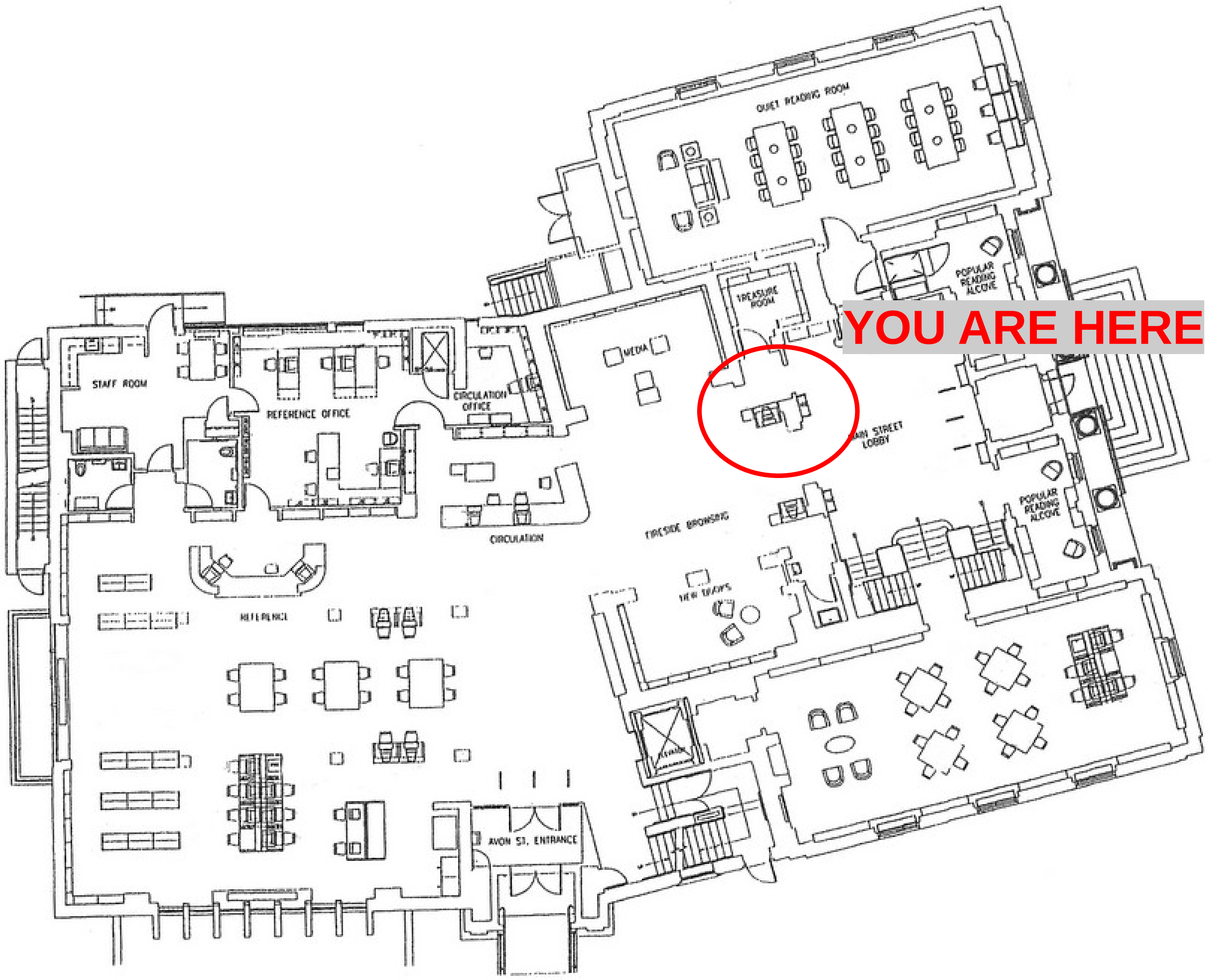
**When you access a web page:**
1. Web browser checks local cache for the desired web page.
2. If the web page is in the cache, load it. Done.
3. If the web page is *not* in the cache, request it from the server.
4. Upon receiving the web page from the server, save it in the cache for next time.

## One kind of cache: a web browser cache



A fundamental problem with caches:
What happens if the web page *changes*, and I am still looking at the copy saved in the cache?
Sooner or later, one has to throw out the cache.
This is *cache invalidation*.

# The Library Analogy



## Are these programs equivalent?

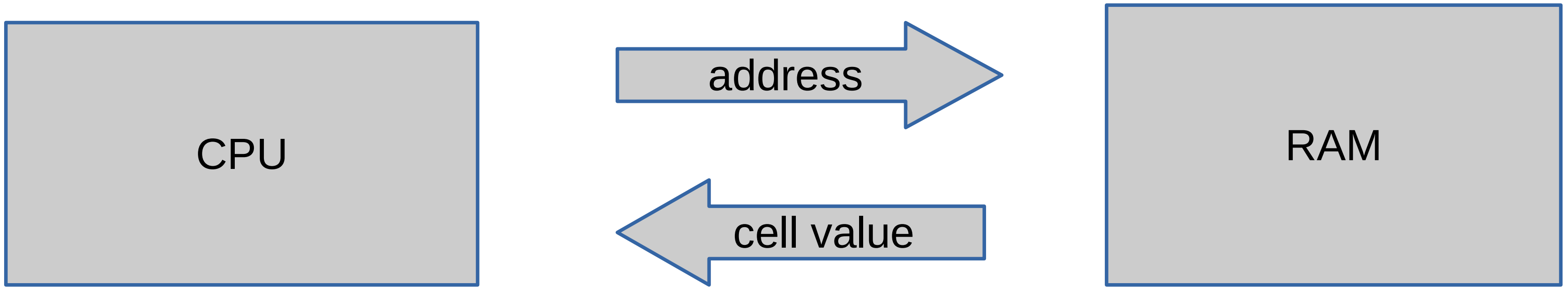
```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 8000

int main () {
  int i,j;
  static int x[SIZE][SIZE];
  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      x[j][i] = i + j; }
  }
  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      x[j][i] += 1; }
  }
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 8000

int main () {
  int i,j;
  static int x[SIZE][SIZE];
  for (j = 0; j < SIZE; j++) {
     for (i = 0; i < SIZE; i++) {
       x[j][i] = i + j; }
   }
  for (j = 0; j < SIZE; j++) {
     for (i = 0; i < SIZE; i++) {
       x[j][i] += 1; }
   }
  return 0;
}
```

## Where should important data be?

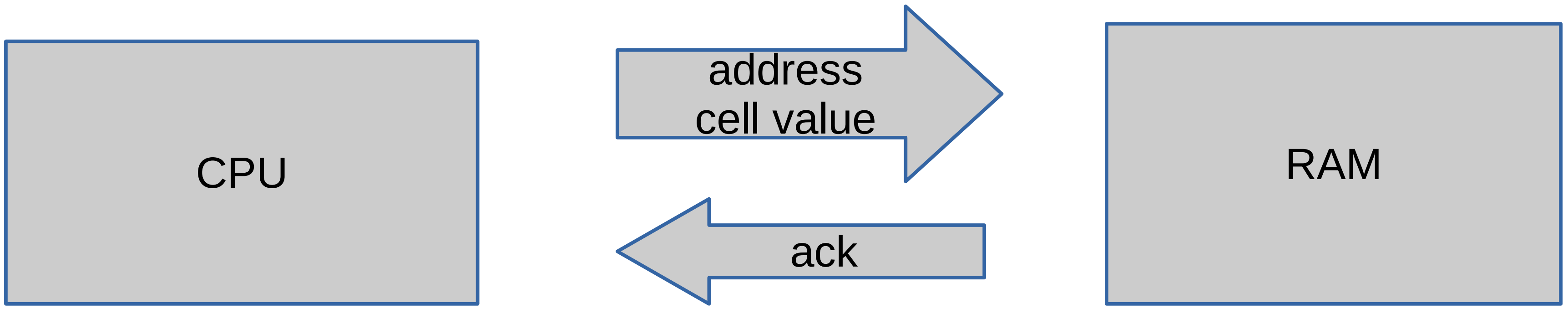
- **Registers are fast**
  - Accessing a register is "free," built into the time to execute the instruction
  - On E20, every instruction has an ID stage
  - If there are no hazards, accessing a register is very fast
- **Memory is slow**
  - On E20, `lw` and `sw` may cause a stall of 4 cycles
  - On a real computer (including Intel instruction set), the penalty can be more, sometimes 50 cycles
  - The penalty is not based just on the speed of the memory, but on the latency of transmission between the CPU and the memory

## How to read memory



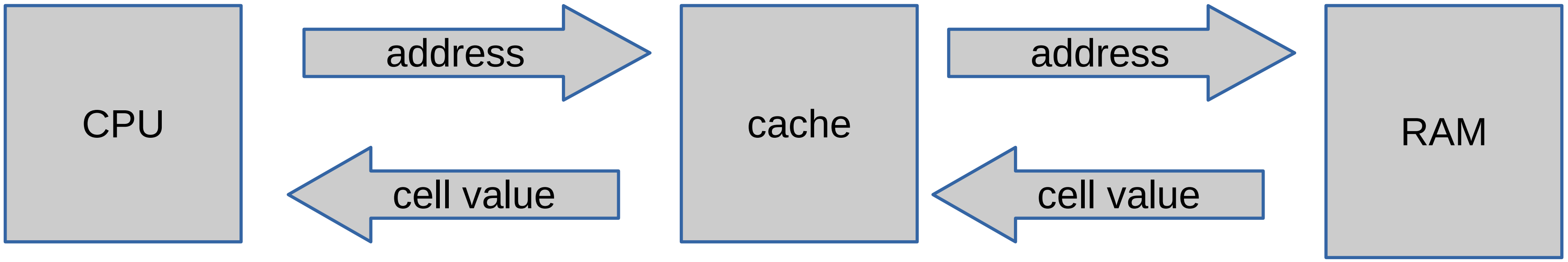
```
lw $1, 5($0)
```

## How to write memory



```
movi $1, 53
sw $1, 5($0)
```

# How to read memory with cache



```
lw $1, 5($0)
```

# Assume we have an arbitrarily large cache

```
$1 = 0
$2 = 5
$3 = 9
$4 = 43
$5 = 2
$6 = 0
$7 = 1
```

CPU

```
cache[0] = ?
cache[1] = ?
cache[2] = ?
cache[3] = ?
cache[4] = ?
cache[5] = ?
cache[6] = ?
cache[7] = ?
....
```

cache

```
ram[0] = 99
ram[1] = 34
ram[2] = 45
ram[3] = 0
ram[4] = 12
ram[5] = 42
ram[6] = 0
ram[7] = 8
....
```

RAM

# Assume we have an arbitrarily large cache

lw $1, 5($0)

$1 = 0
$2 = 5
$3 = 9
$4 = 43
$5 = 2
$6 = 0
$7 = 1

CPU

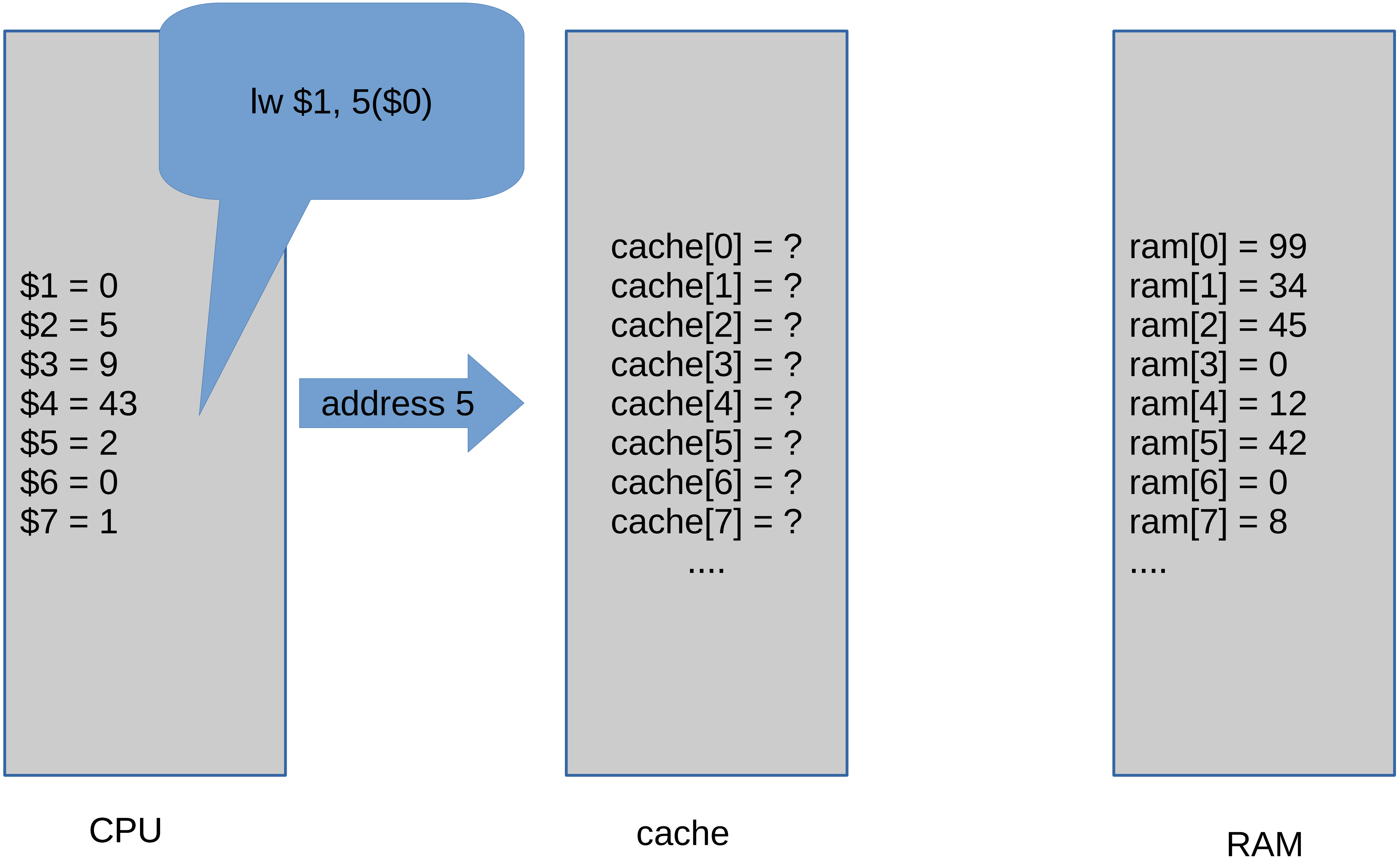
cache[0] = ?
cache[1] = ?
cache[2] = ?
cache[3] = ?
cache[4] = ?
cache[5] = ?
cache[6] = ?
cache[7] = ?
....

cache

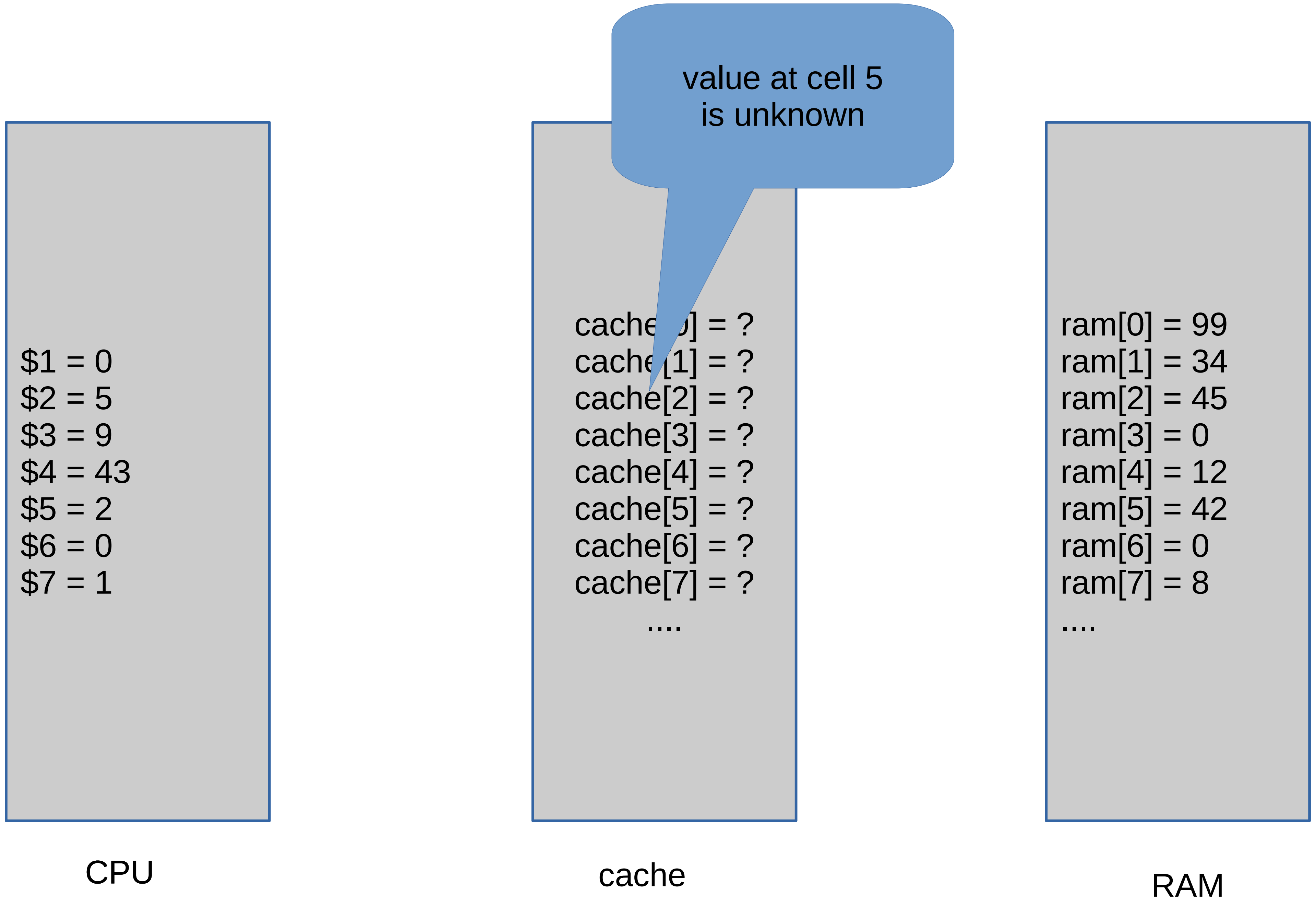
ram[0] = 99
ram[1] = 34
ram[2] = 45
ram[3] = 0
ram[4] = 12
ram[5] = 42
ram[6] = 0
ram[7] = 8
....

RAM

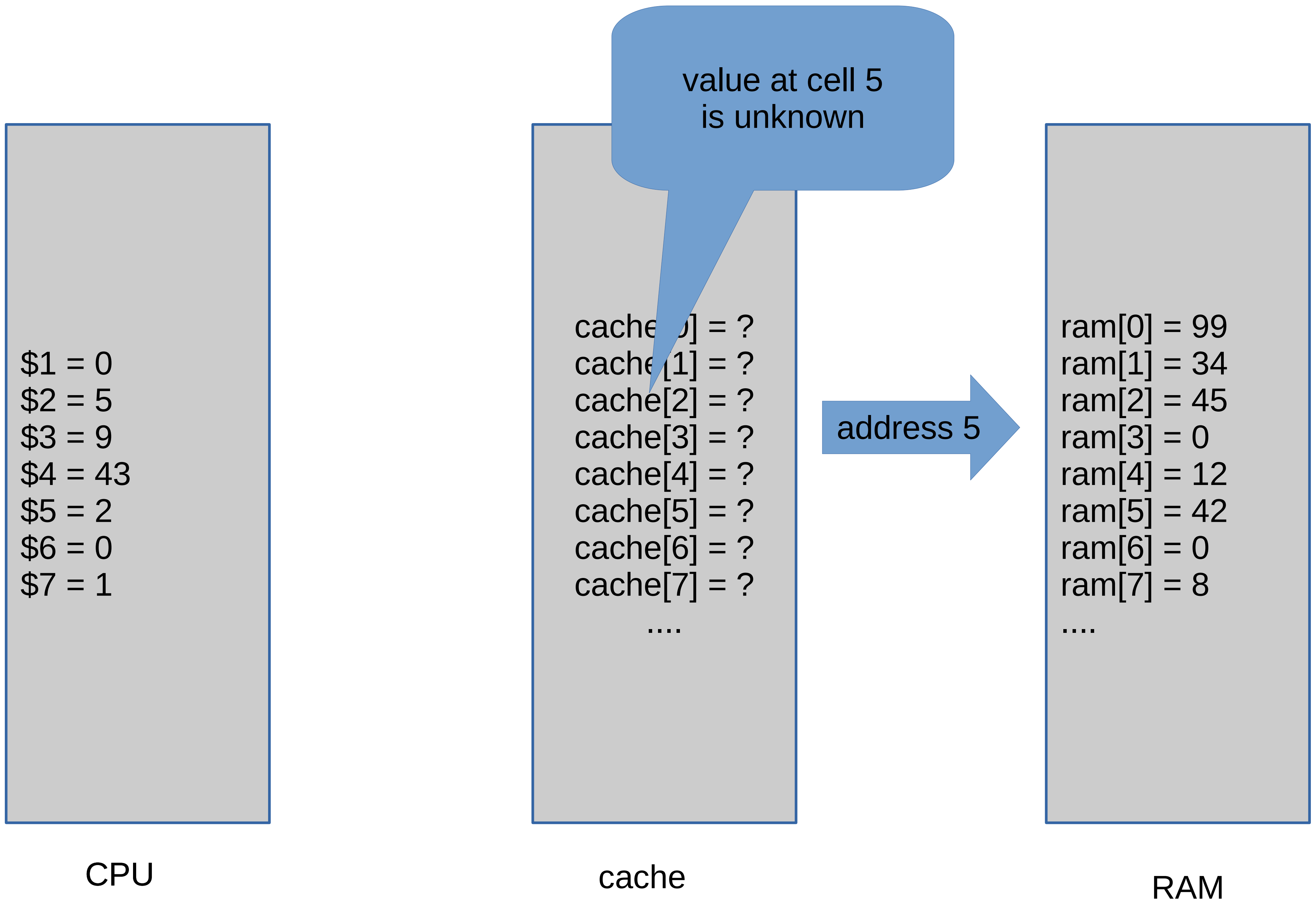
# Assume we have an arbitrarily large cache



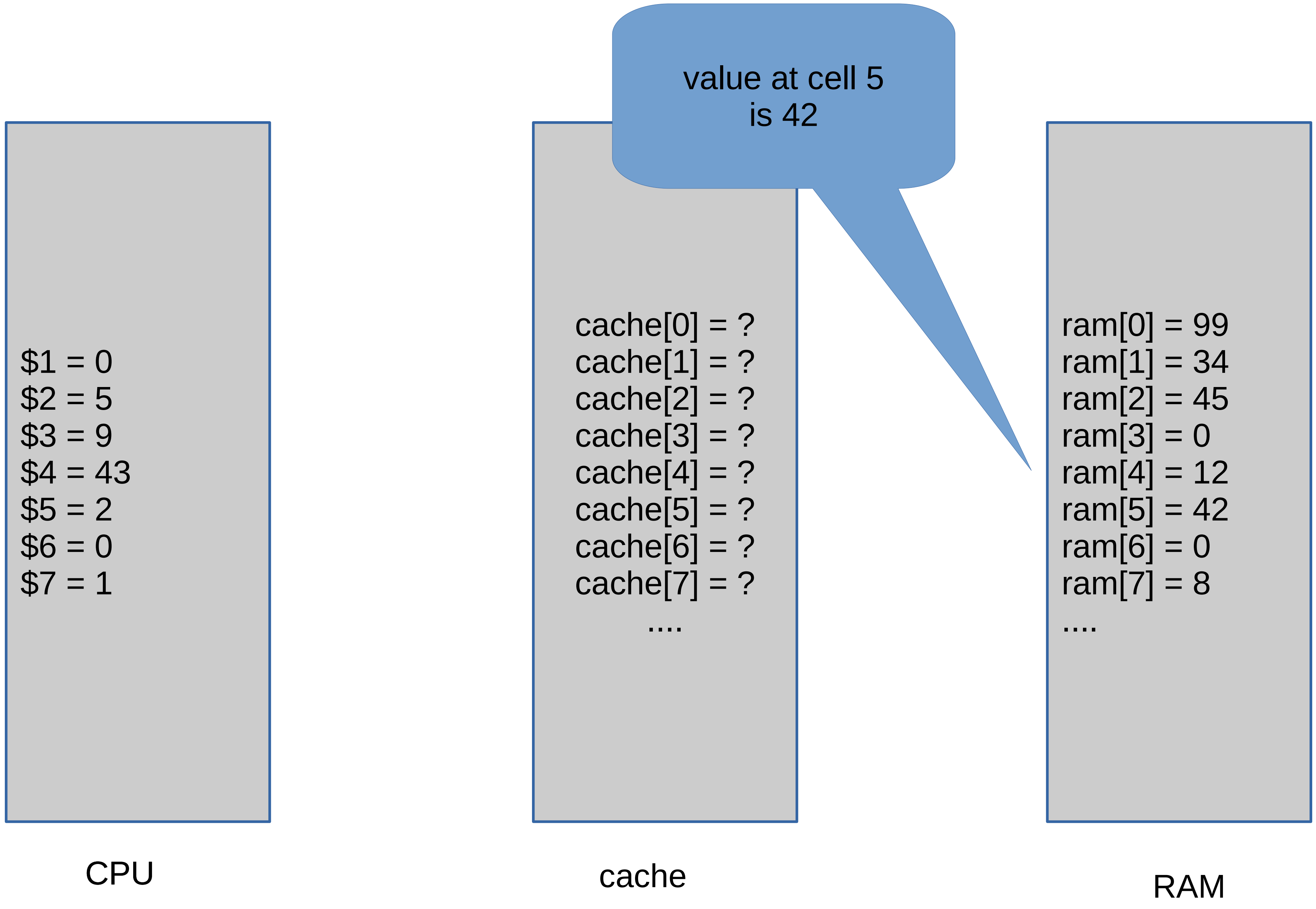
# Assume we have an arbitrarily large cache



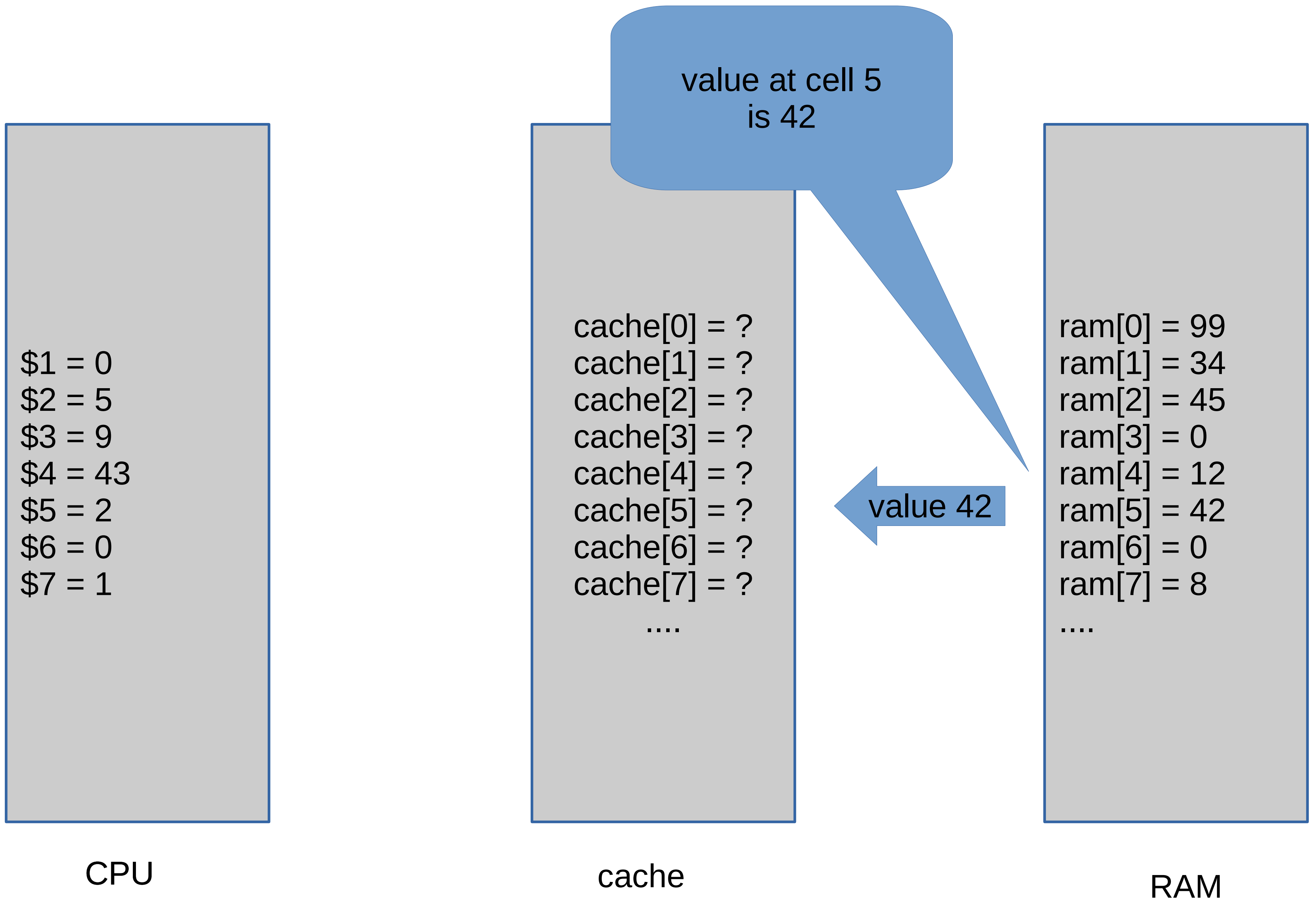
# Assume we have an arbitrarily large cache



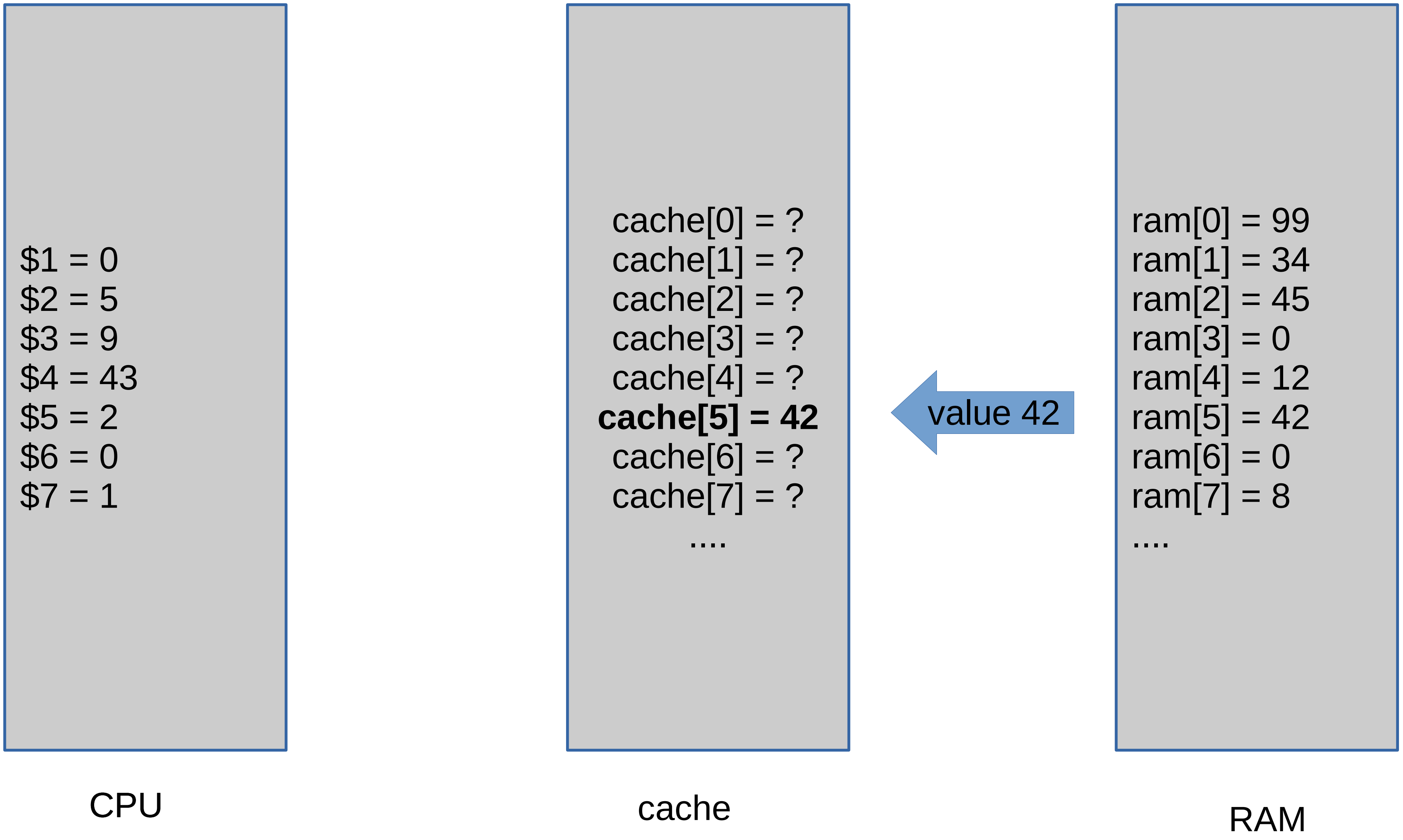
# Assume we have an arbitrarily large cache



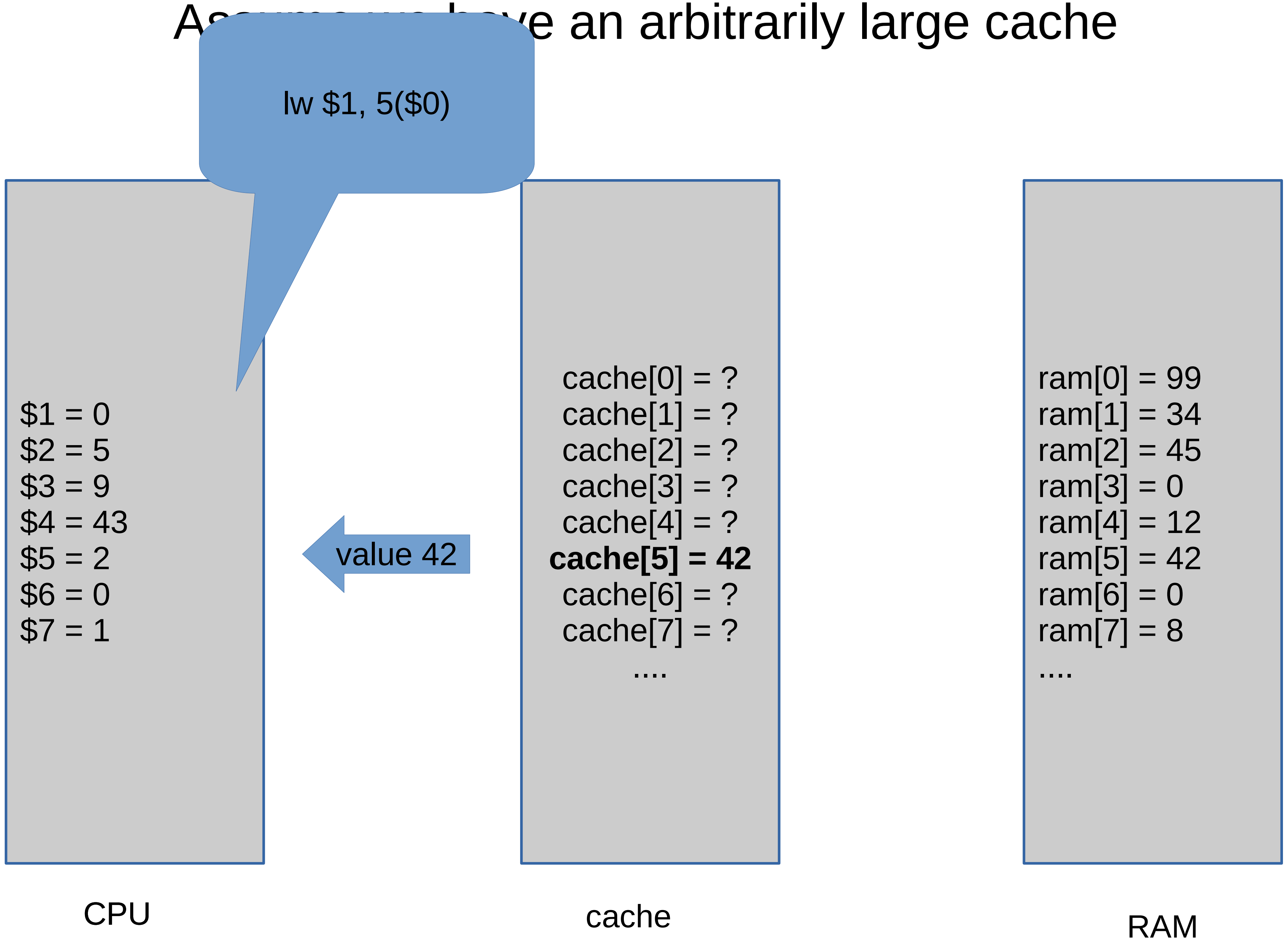
# Assume we have an arbitrarily large cache



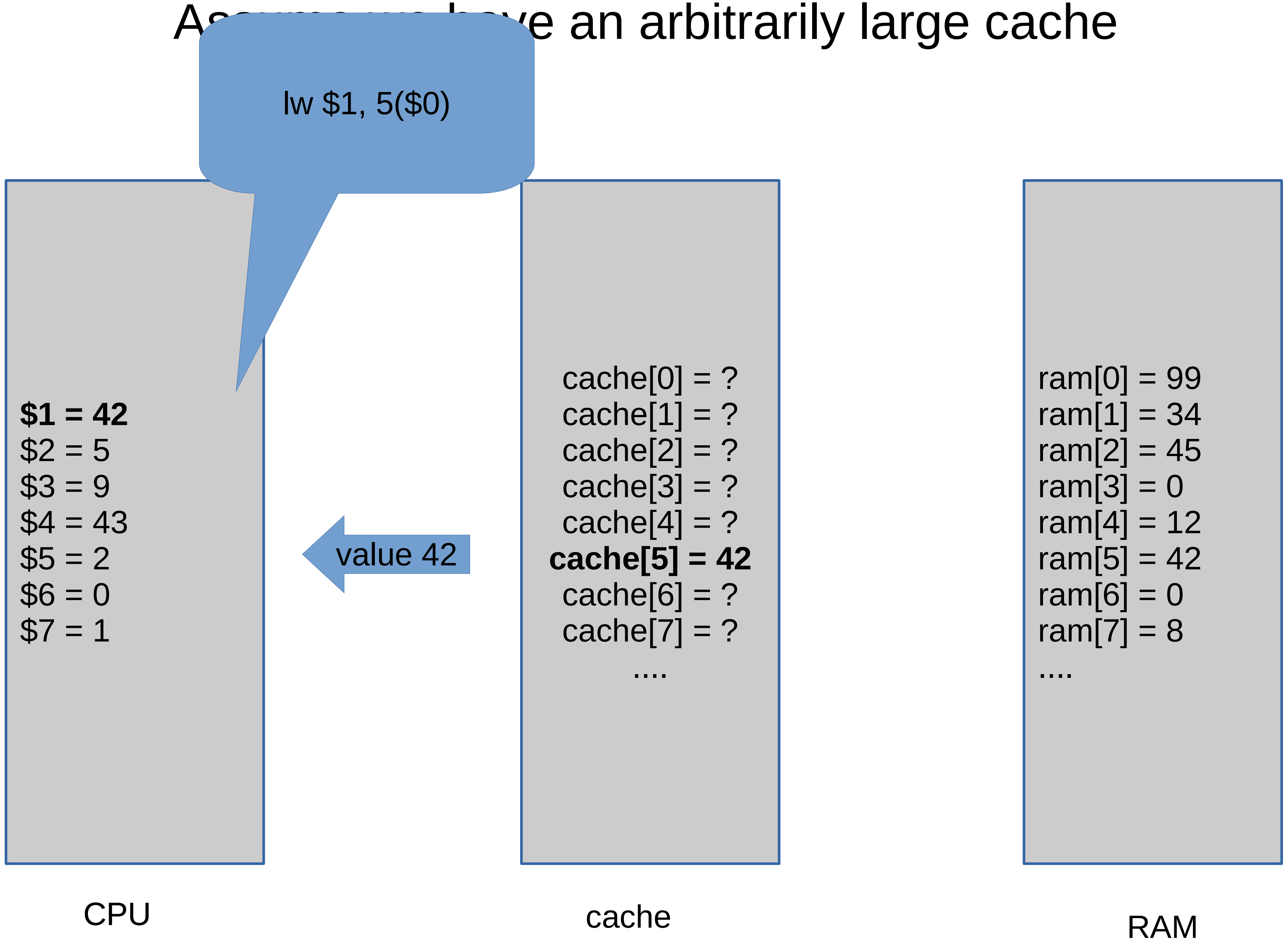
## Assume we have an arbitrarily large cache



## Assume we have an arbitrarily large cache



# Assume we have an arbitrarily large cache



# Assume we have an arbitrarily large cache

Now let's read the same address again

lw $2, 5($0)

$1 = 42
$2 = 5
$3 = 9
$4 = 43
$5 = 2
$6 = 0
$7 = 1

CPU

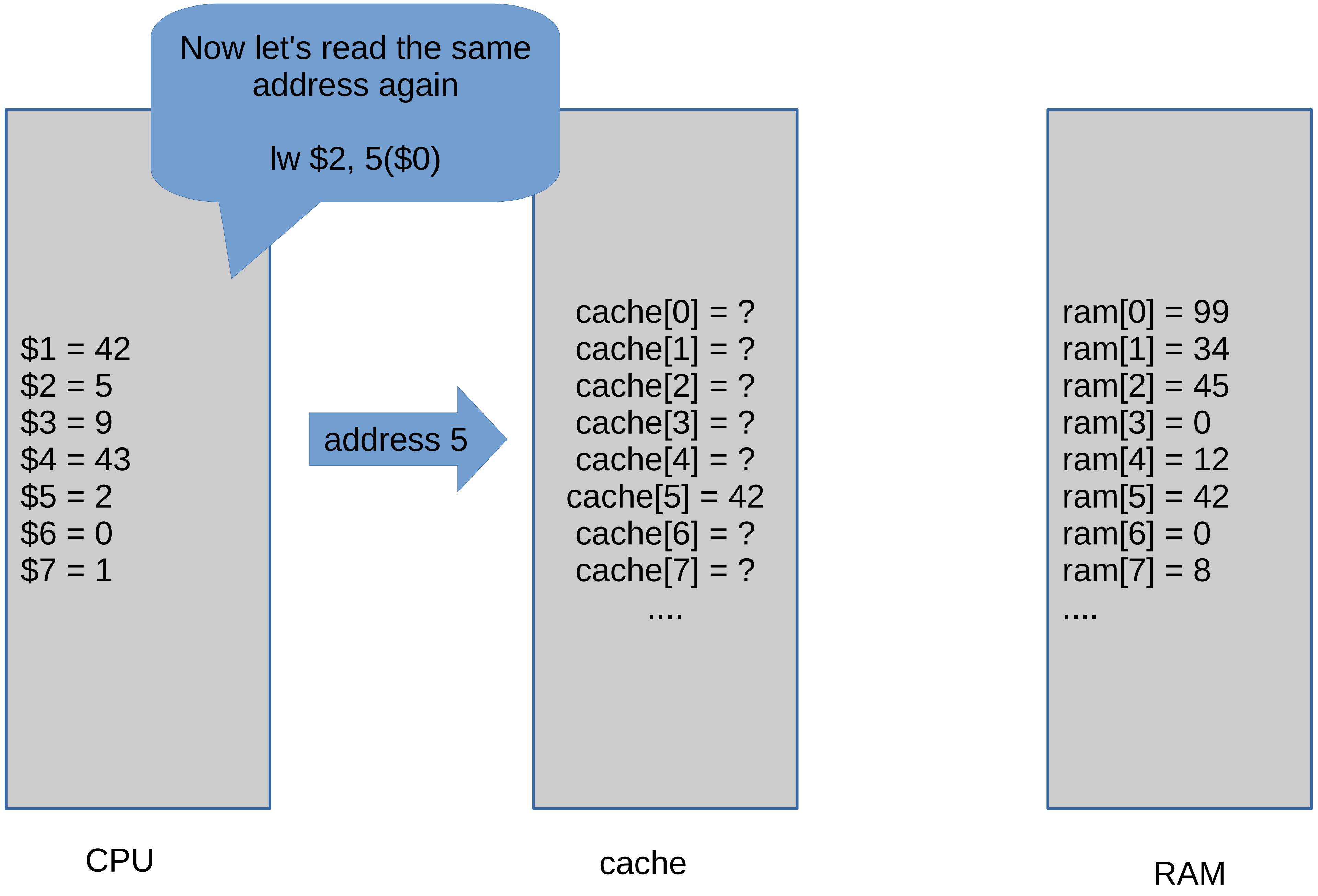
cache[0] = ?
cache[1] = ?
cache[2] = ?
cache[3] = ?
cache[4] = ?
cache[5] = 42
cache[6] = ?
cache[7] = ?
....

cache

ram[0] = 99
ram[1] = 34
ram[2] = 45
ram[3] = 0
ram[4] = 12
ram[5] = 42
ram[6] = 0
ram[7] = 8
....

RAM

# Assume we have an arbitrarily large cache


Now let's read the same address again

```
lw $2, 5($0)
```

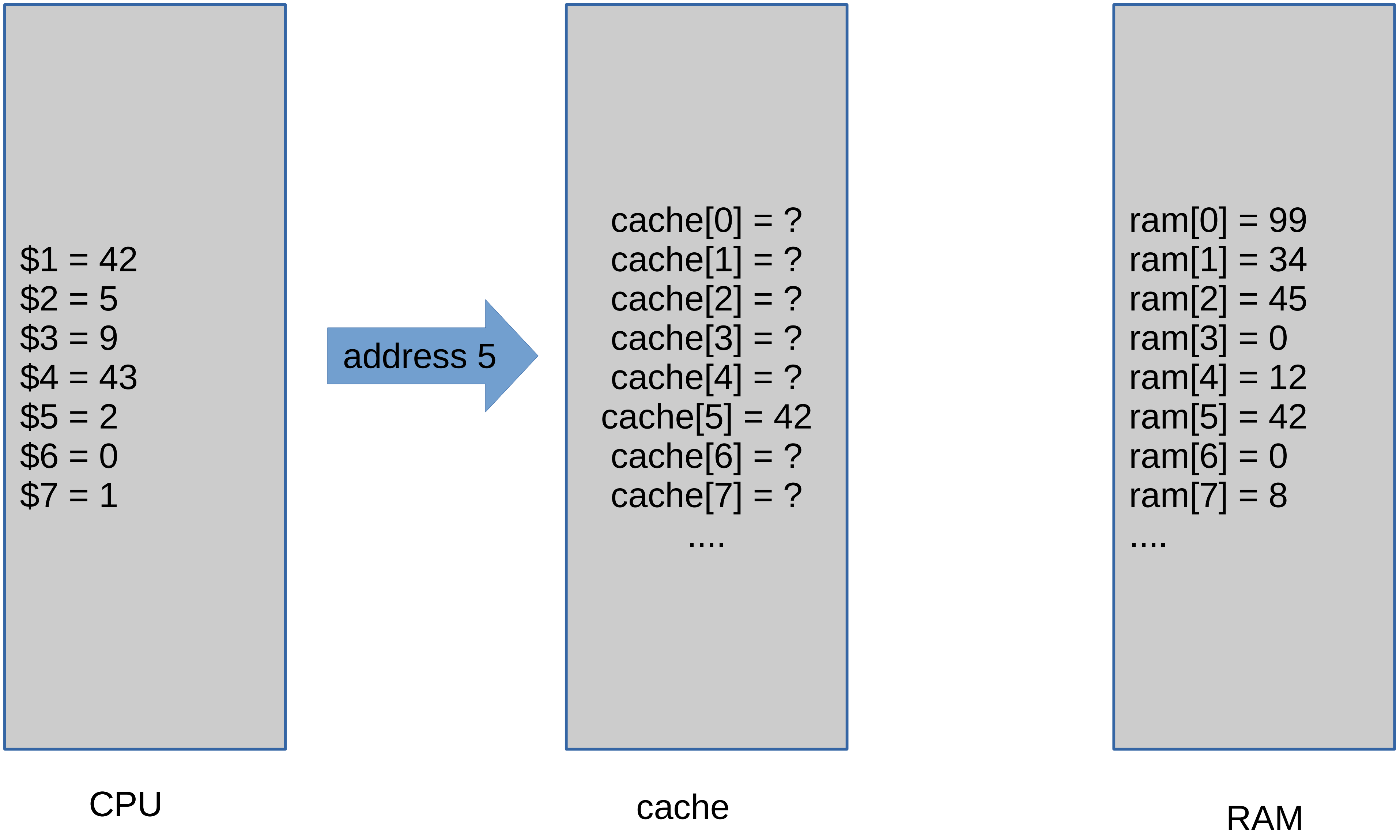
$1 = 42
$2 = 5
$3 = 9
$4 = 43
$5 = 2
$6 = 0
$7 = 1

address 5

cache[0] = ?
cache[1] = ?
cache[2] = ?
cache[3] = ?
cache[4] = ?
cache[5] = 42
cache[6] = ?
cache[7] = ?
....

ram[0] = 99
ram[1] = 34
ram[2] = 45
ram[3] = 0
ram[4] = 12
ram[5] = 42
ram[6] = 0
ram[7] = 8
....

# Assume we have an arbitrarily large cache

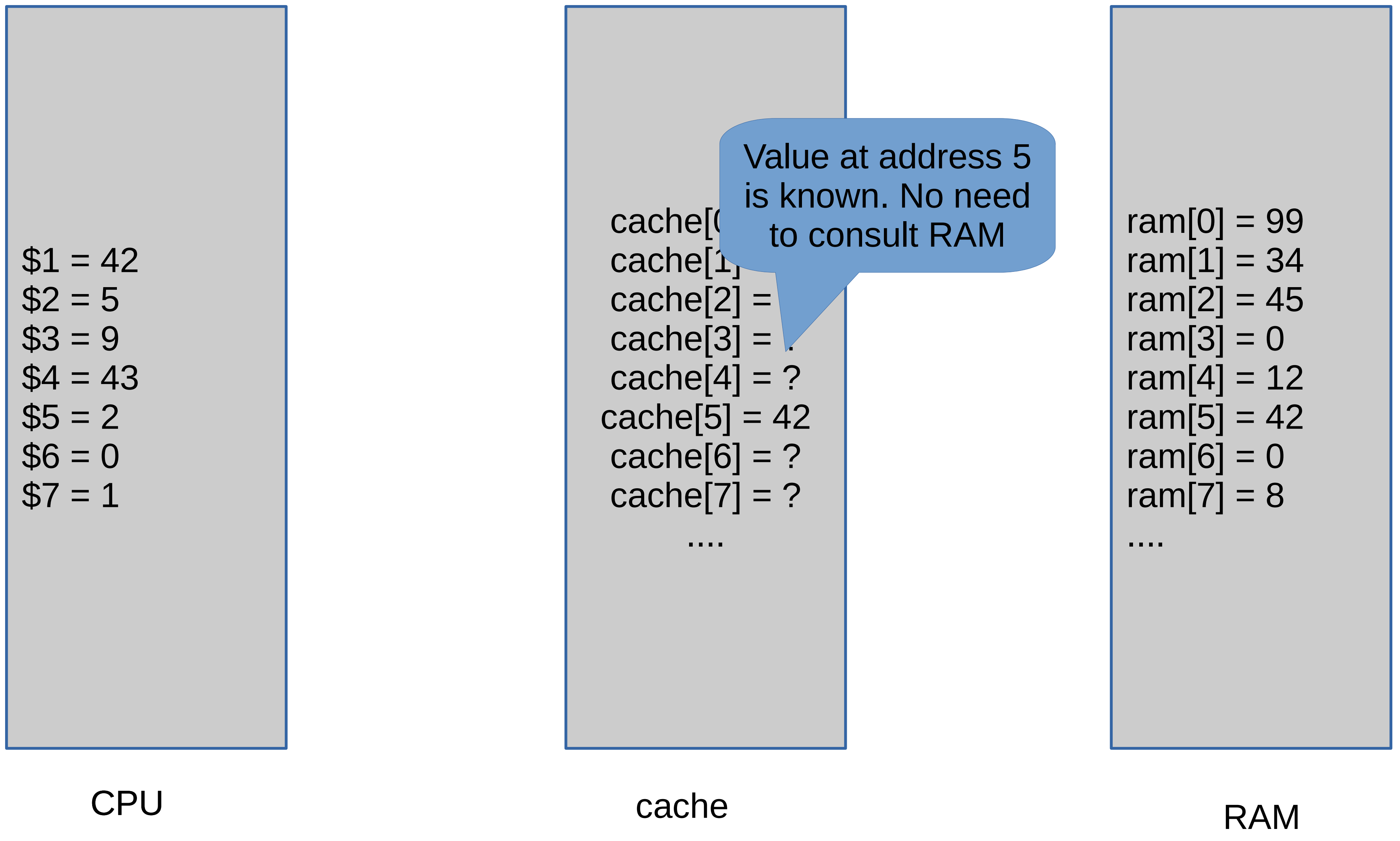


$1 = 42
$2 = 5
$3 = 9
$4 = 43
$5 = 2
$6 = 0
$7 = 1

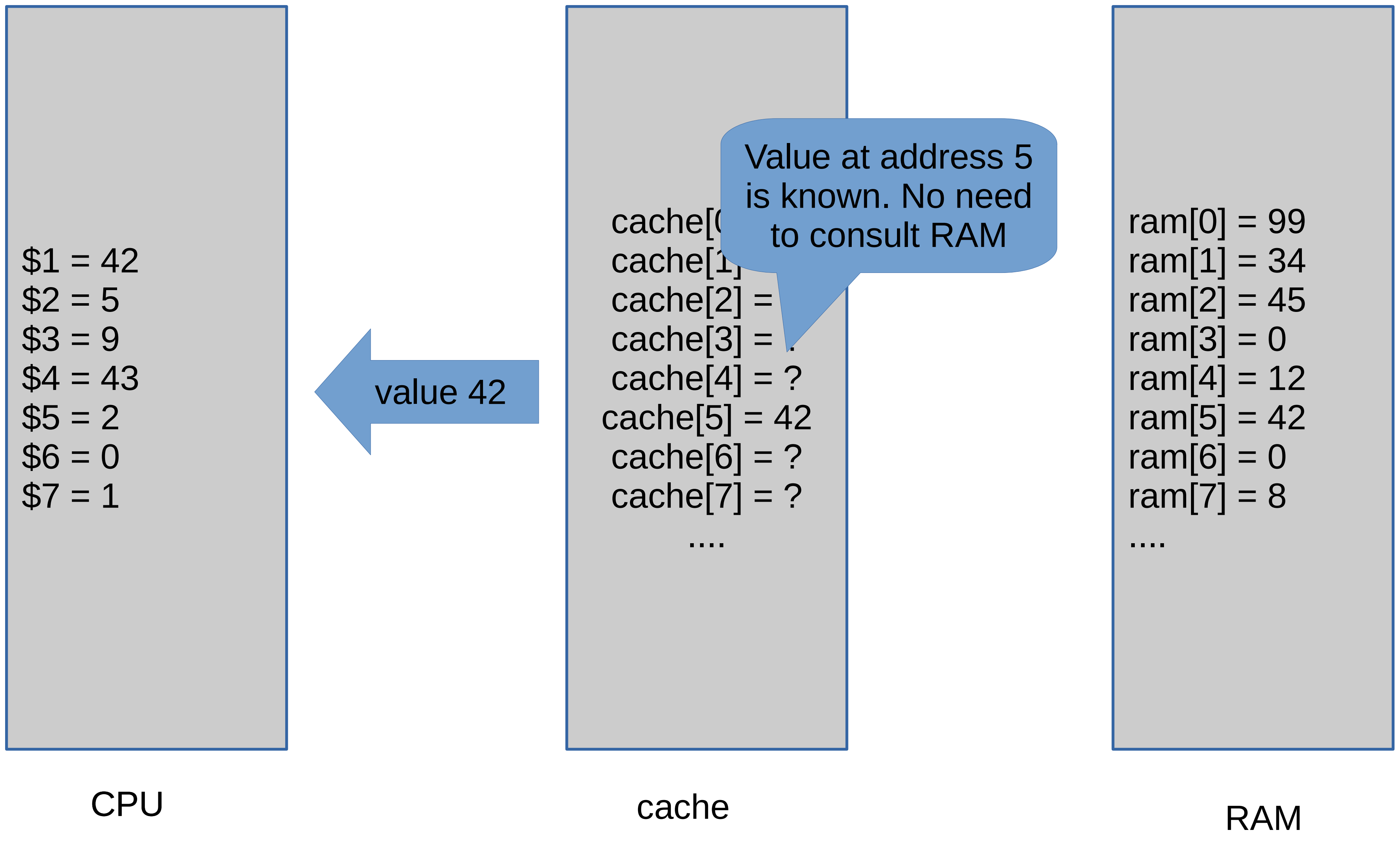
cache[0] = ?
cache[1] = ?
cache[2] = ?
cache[3] = ?
cache[4] = ?
cache[5] = 42
cache[6] = ?
cache[7] = ?
....

ram[0] = 99
ram[1] = 34
ram[2] = 45
ram[3] = 0
ram[4] = 12
ram[5] = 42
ram[6] = 0
ram[7] = 8
....

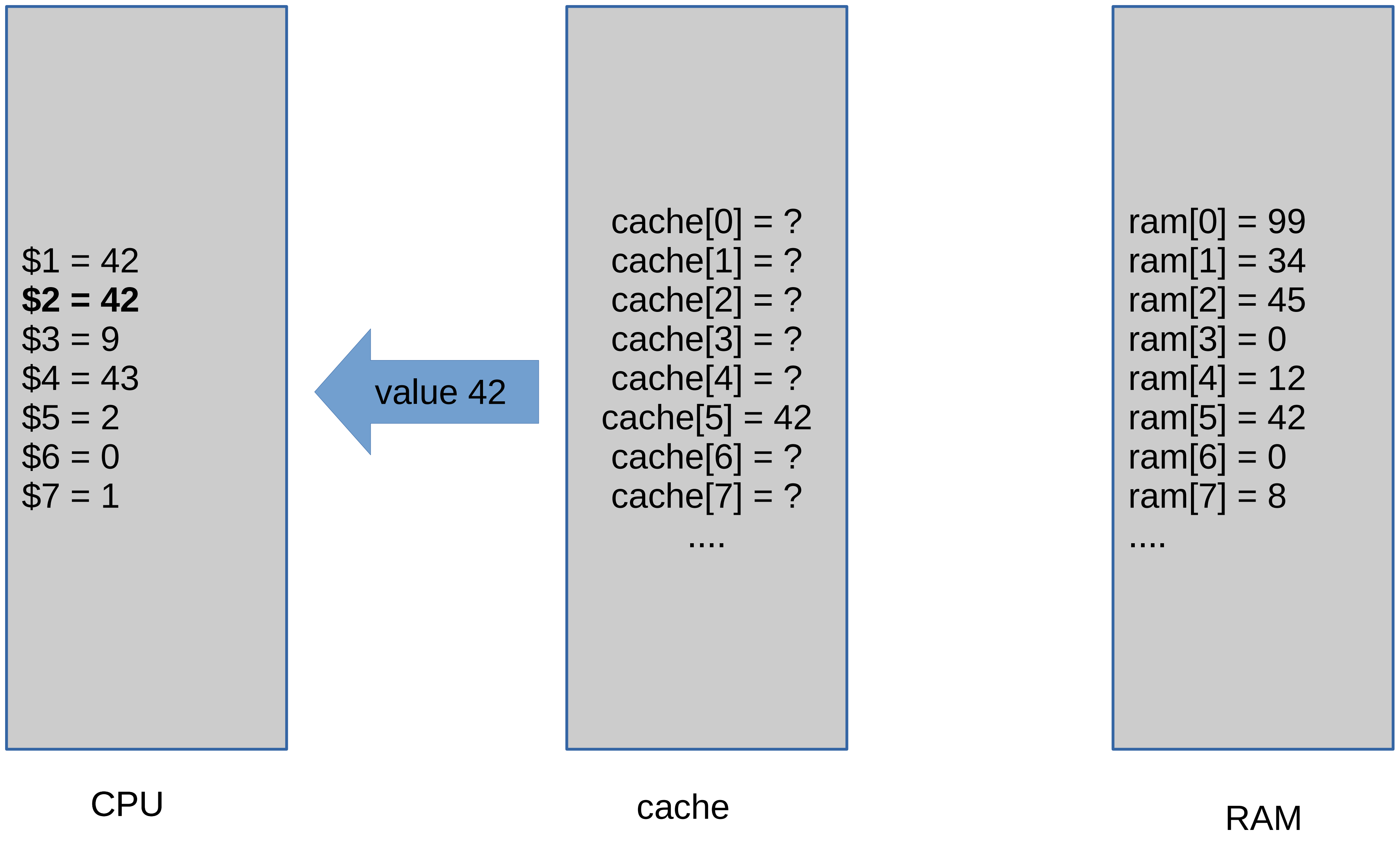
# Assume we have an arbitrarily large cache



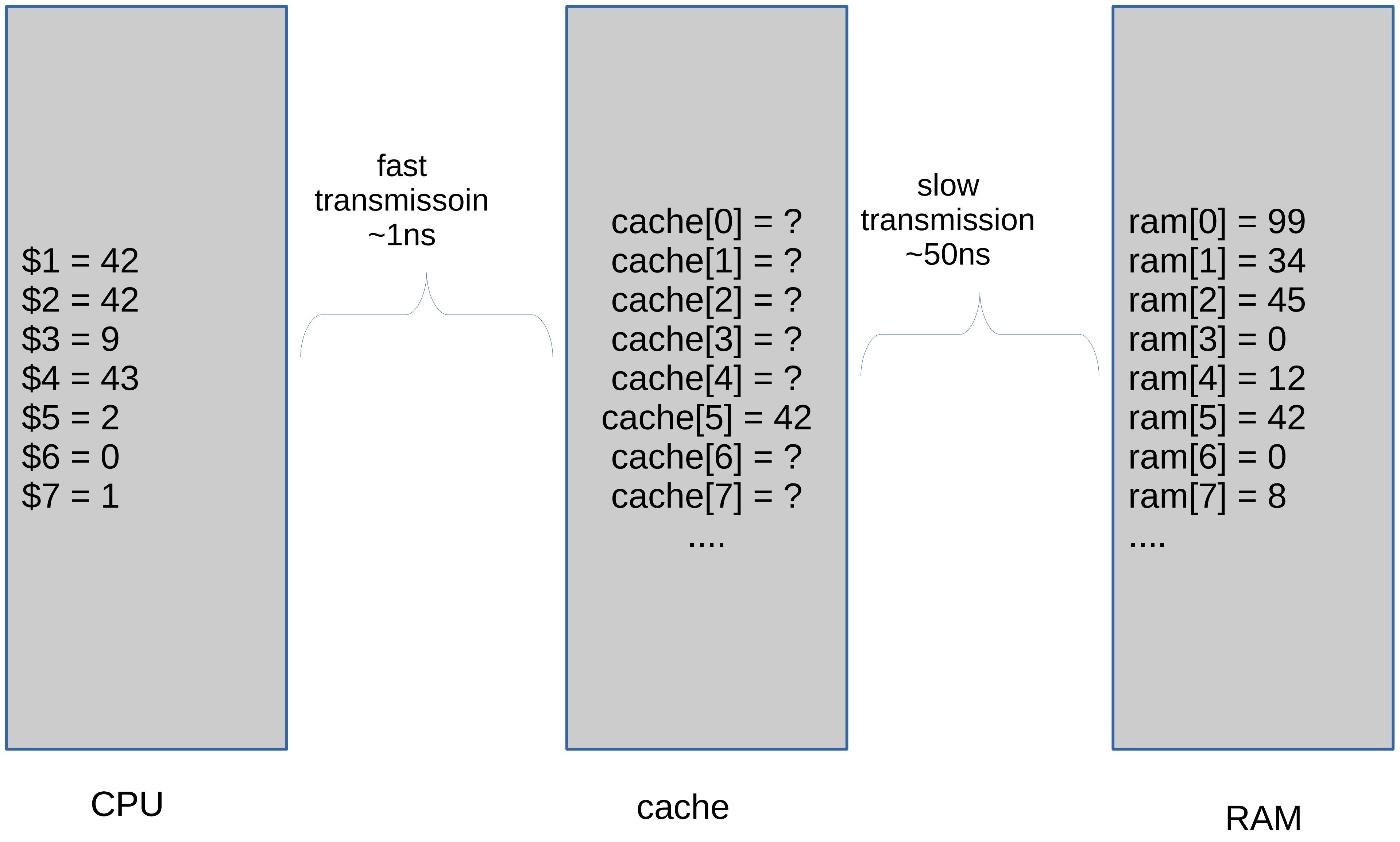
# Assume we have an arbitrarily large cache



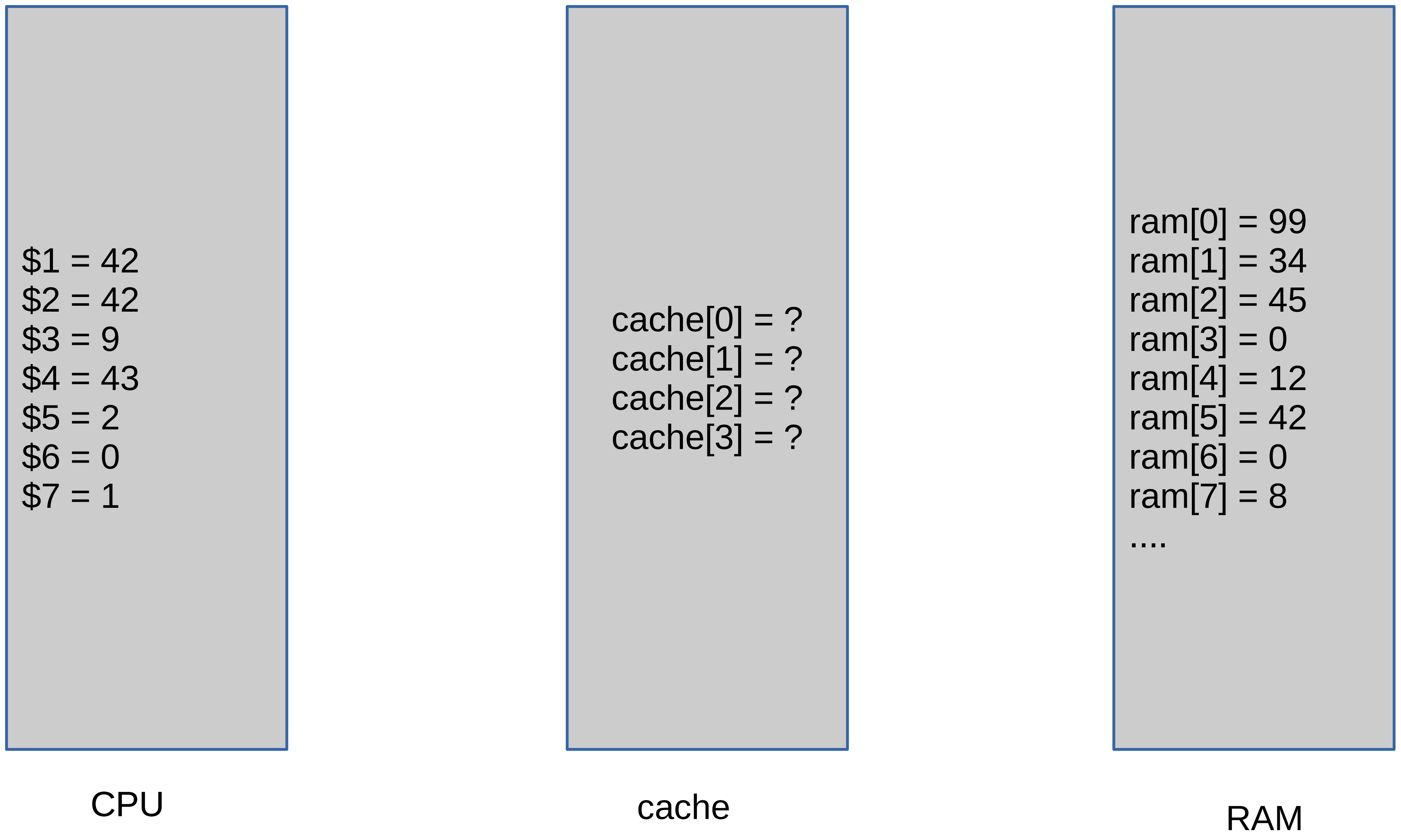
## Assume we have an arbitrarily large cache



# Assume we have an arbitrarily large cache



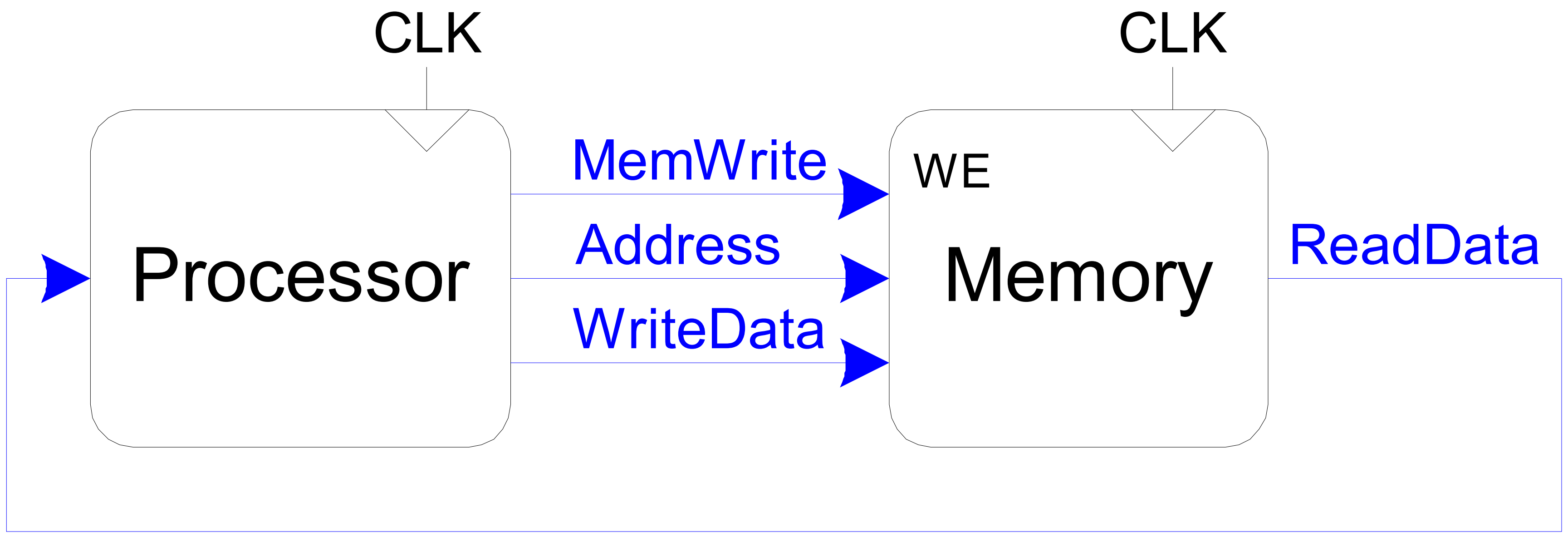
# In reality, cache is much smaller than RAM



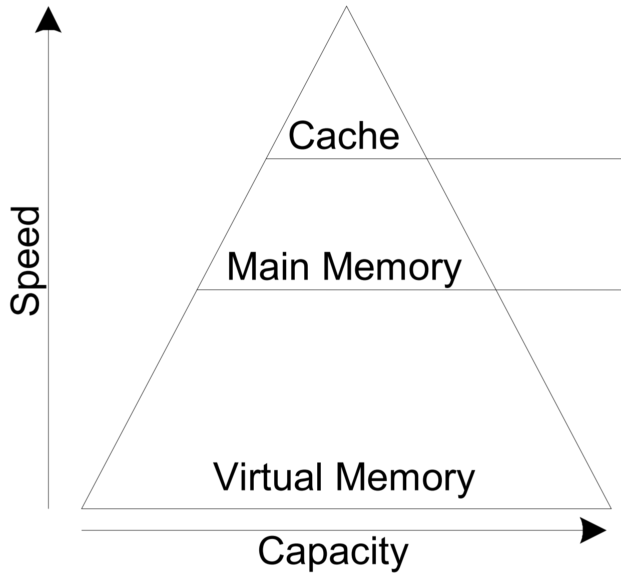
## Introduction

- Computer performance depends on:
  - Processor performance
  - Memory system performance

**Memory Interface**



## Memory Hierarchy



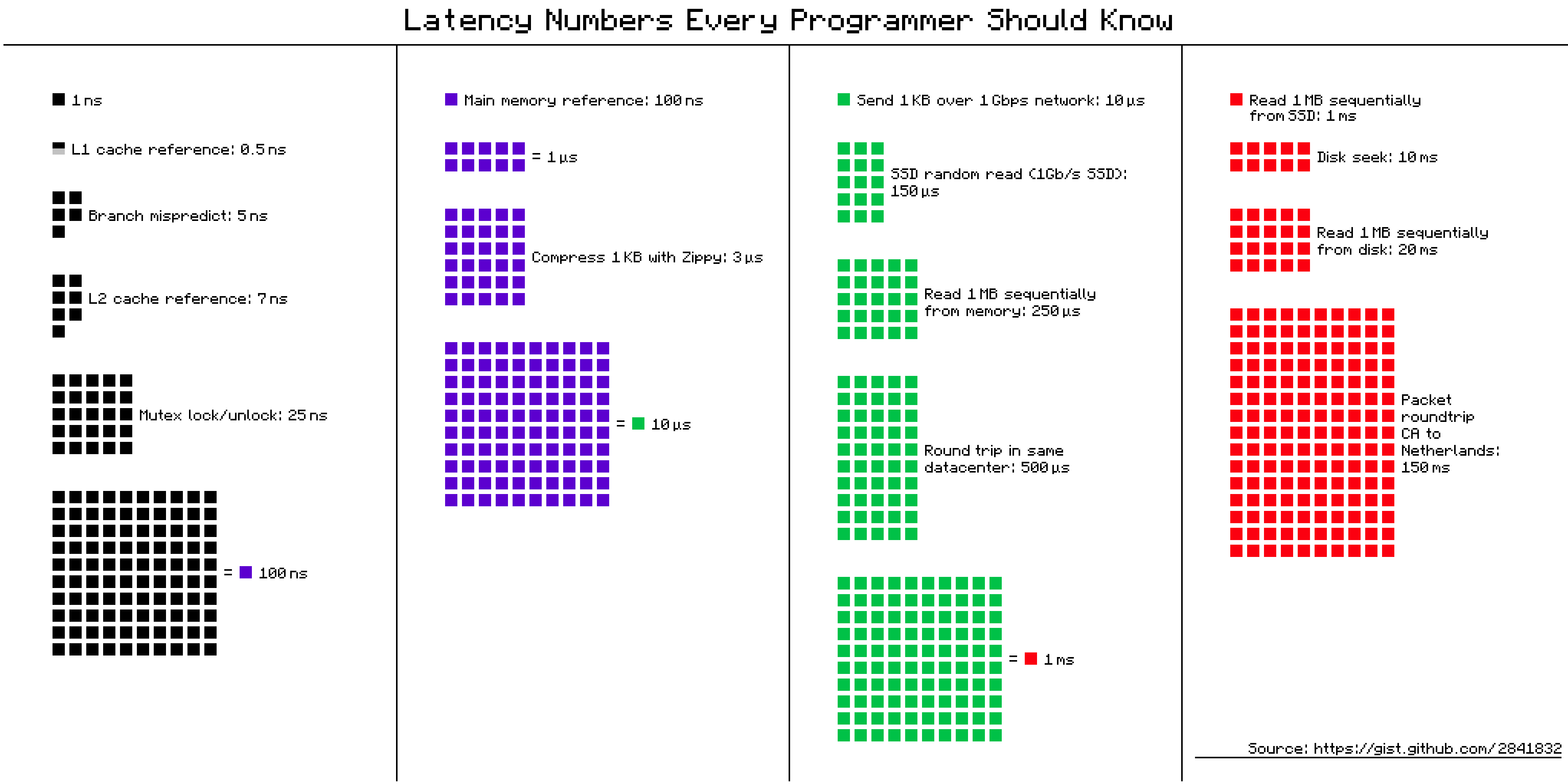
| Technology | Price / GB | Access Time (ns) | Bandwidth (GB/s) |
|---|---|---|---|
| SRAM | \$10,000 | 1 | 25+ |
| DRAM | \$10 | 10 - 50 | 10 |
| SSD | \$1 | 100,000 | 0.5 |
| HDD | \$0.1 | 10,000,000 | 0.1 |

SRAM, DRAM – volatile
SSD, HDD – non-volatile

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

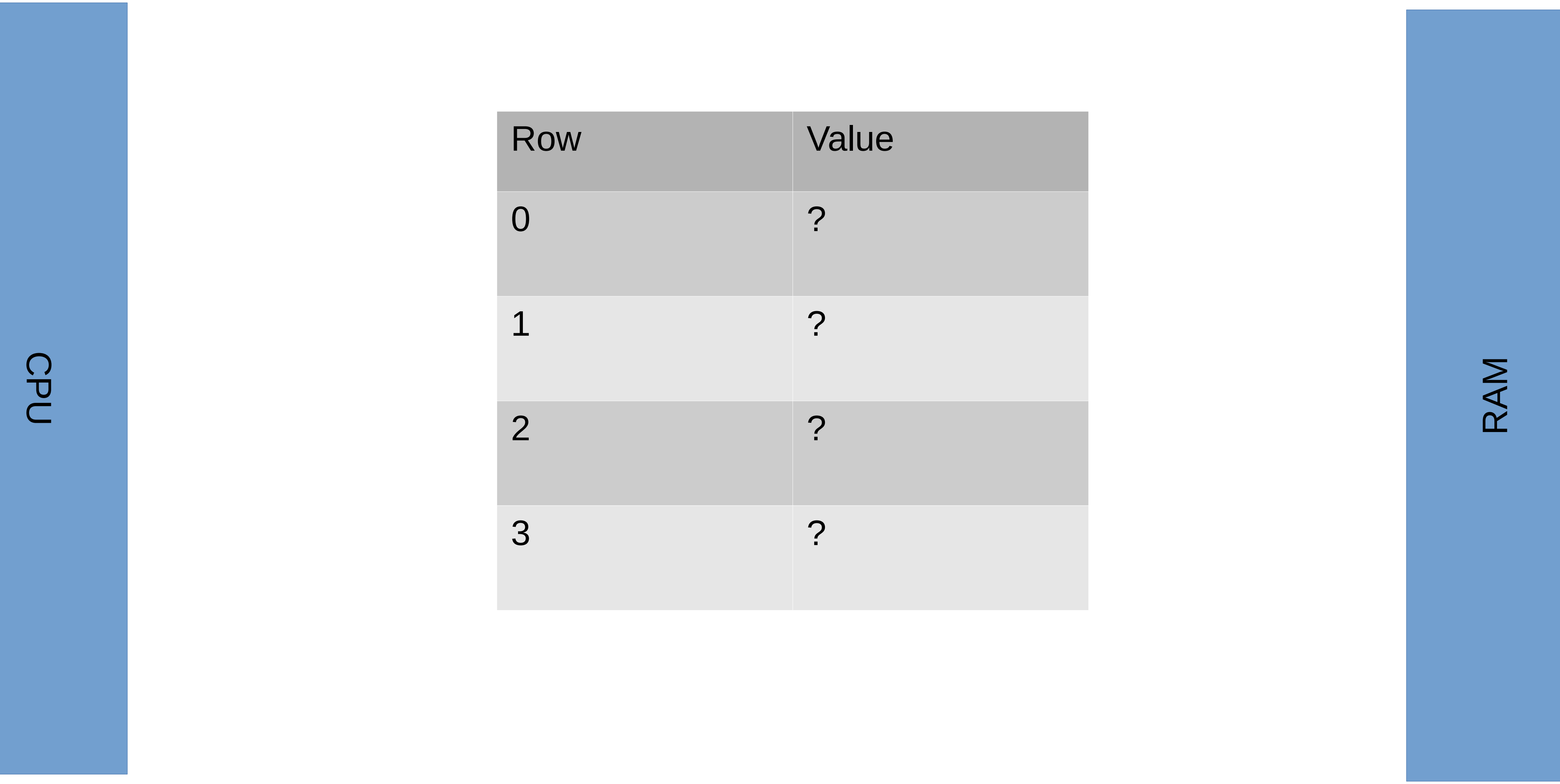
```
Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns
```

```
Credit
------
By Jeff Dean:               http://research.google.com/people/jeff/
Originally by Peter Norvig: http://norvig.com/21-days.html#answers
Contributions
-------------
'Humanized' comparison:  https://gist.github.com/hellerbarde/2843375
Visual comparison chart: http://i.imgur.com/k0t1e.png
```


Latency Numbers Every Programmer Should Know
1 ns
L1 cache reference: 0.5 ns
Branch mispredict: 5 ns
L2 cache reference: 7 ns
Mutex lock/unlock: 25 ns
= 100 ns
Main memory reference: 100 ns
= 1 µs
Compress 1 KB with Zippy: 3 µs
= 10 µs
Send 1 KB over 1 Gbps network: 10 µs
SSD random read (1Gb/s SSD): 150 µs
Read 1 MB sequentially from memory: 250 µs
Round trip in same datacenter: 500 µs
= 1 ms
Read 1 MB sequentially from SSD: 1 ms
Disk seek: 10 ms
Read 1 MB sequentially from disk: 20 ms
Packet roundtrip CA to Netherlands: 150 ms
Source: https://gist.github.com/2841832

```
Lets multiply all these durations by a billion:
Magnitudes:
### Minute:
    L1 cache reference                  0.5 s          One heart beat (0.5 s)
    Branch mispredict                   5 s            Yawn
    L2 cache reference                  7 s            Long yawn
    Mutex lock/unlock                   25 s           Making a coffee
### Hour:
    Main memory reference               100 s          Brushing your teeth
    Compress 1K bytes with Zippy        50 min         One episode of a TV show
### Day:
    Send 2K bytes over 1 Gbps network   5.5 hr         From lunch to end of work day
### Week
    SSD random read                     1.7 days       A normal weekend
    Read 1 MB sequentially from memory  2.9 days       A long weekend
    Round trip within same datacenter   5.8 days       A medium vacation
    Read 1 MB sequentially from SSD    11.6 days       Waiting for almost 2 weeks for a delivery
### Year
    Disk seek                           16.5 weeks     A semester in university
    Read 1 MB sequentially from disk    7.8 months     Almost producing a new human being
    The above 2 together                1 year
### Decade
    Send packet CA->Netherlands->CA     4.8 years      Average time it takes to complete a bachelor's
```

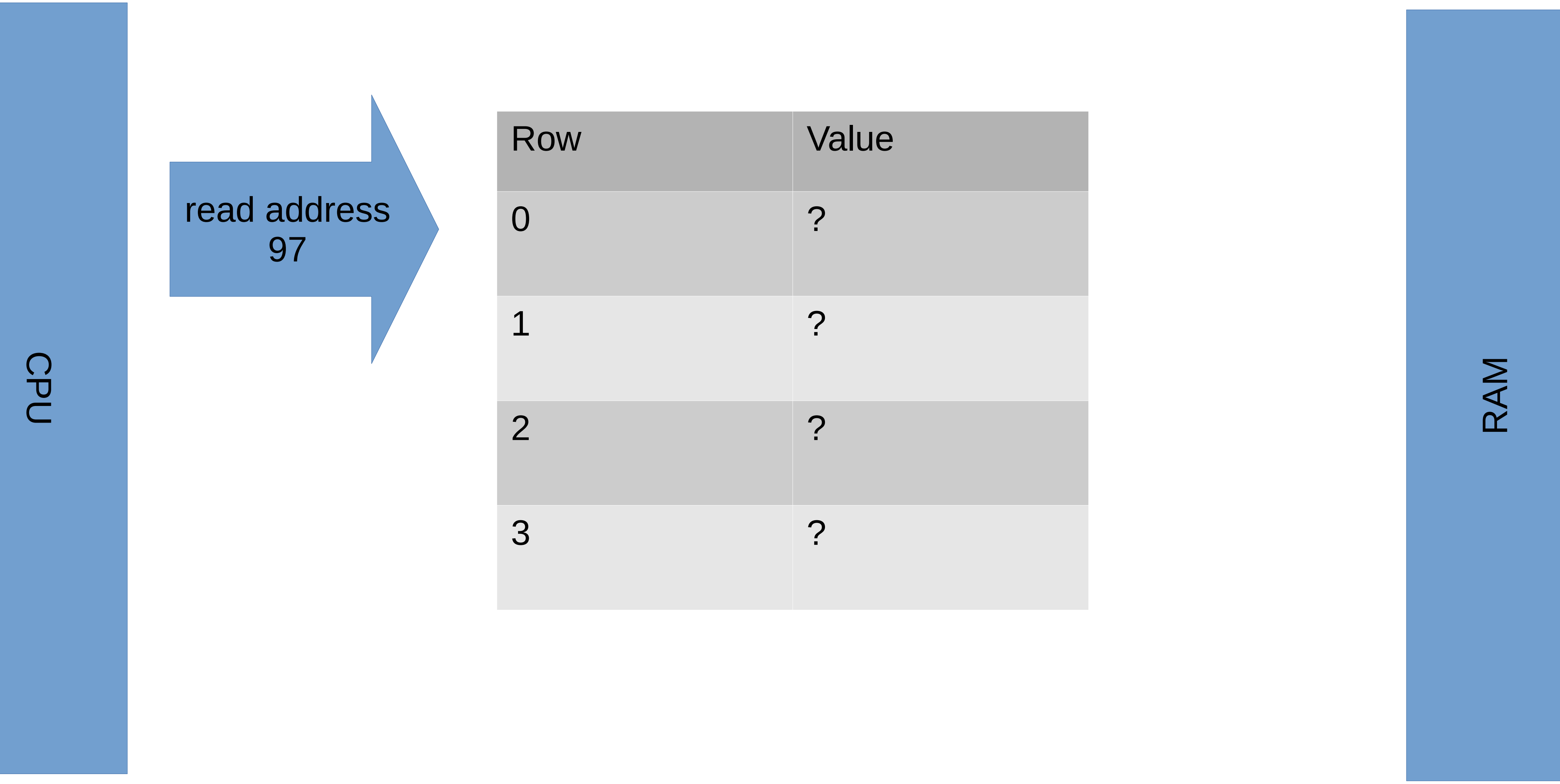
## Allocating memory cells to cache rows: first attempt



| Row | Value |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |

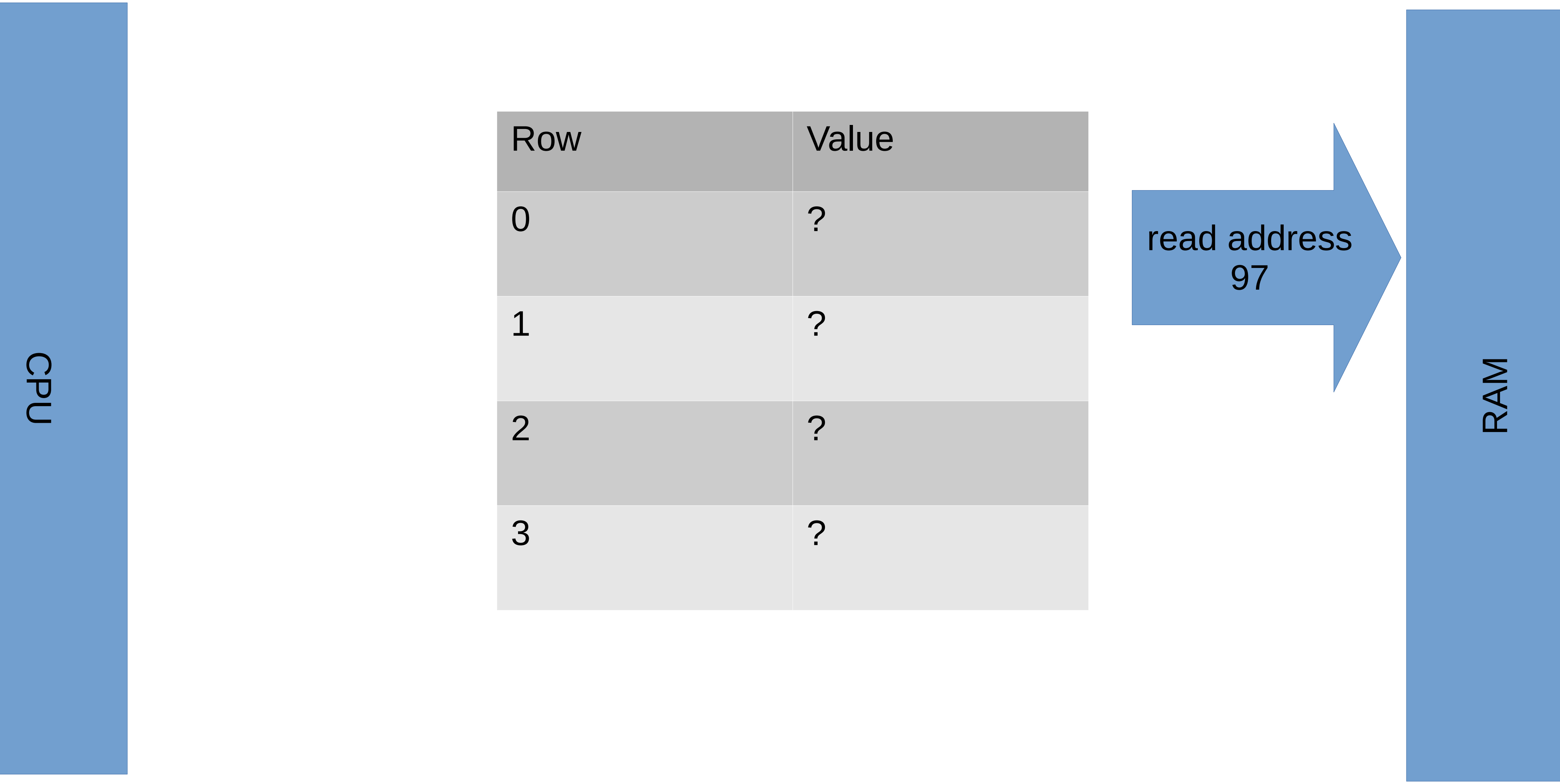
Our cache has four *rows*, or storage units for data received from memory.

## Allocating memory cells to cache rows: first attempt



| Row | Value |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |

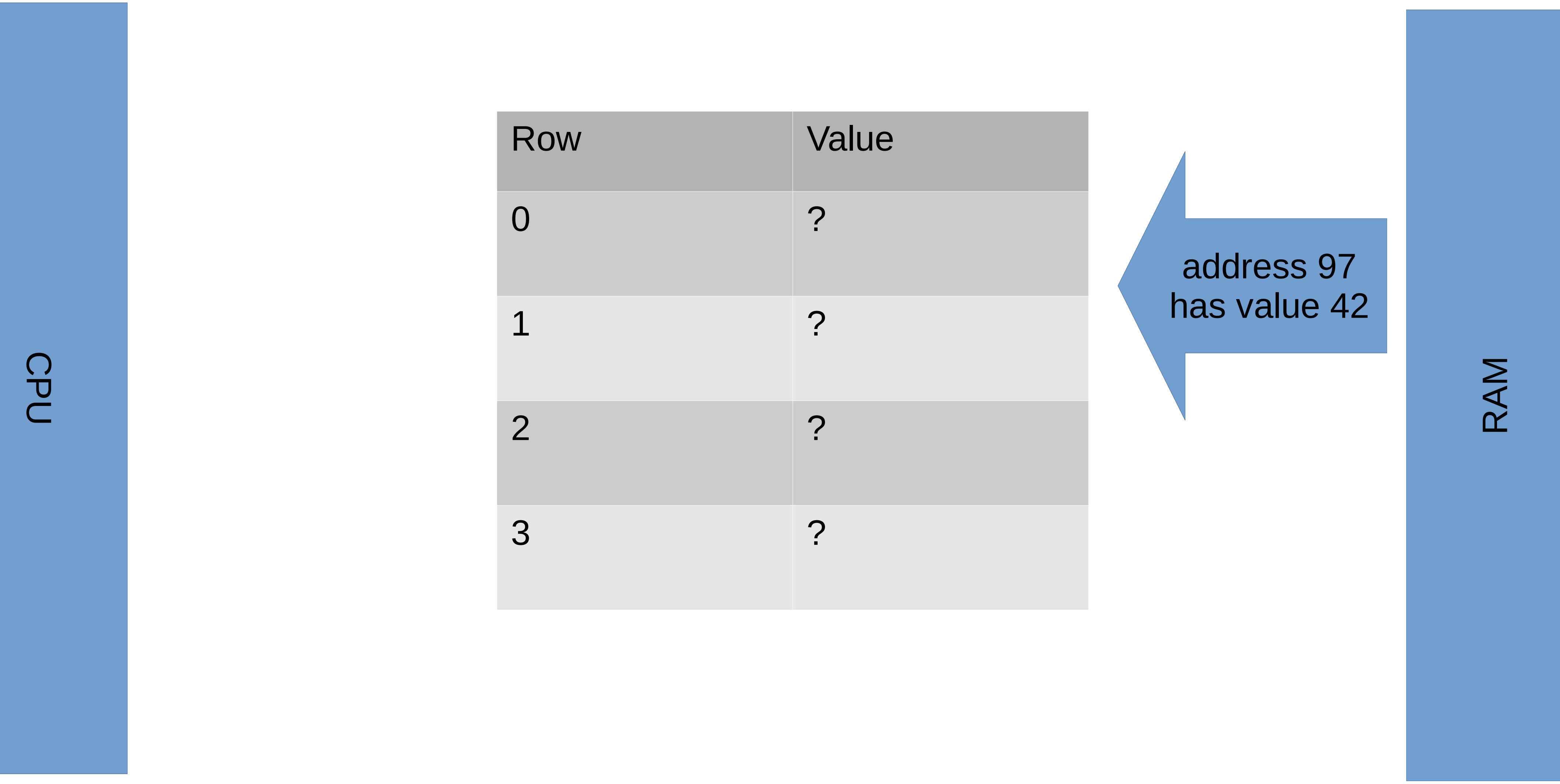
## Allocating memory cells to cache rows: first attempt



| Row | Value |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |

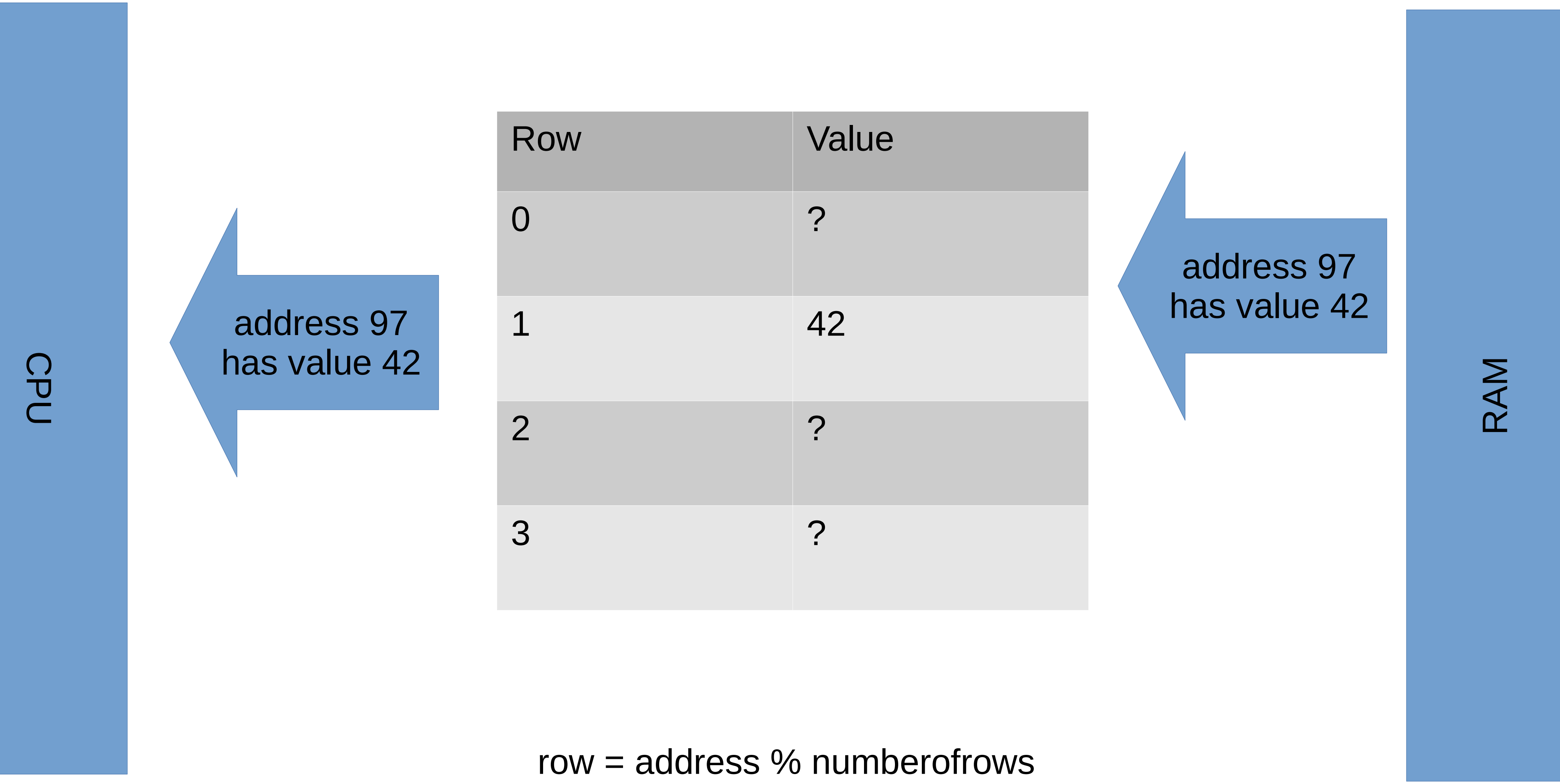
This is a *cache miss*: requested data is not in the cache, so we refer to RAM.

## Allocating memory cells to cache rows: first attempt



| Row | Value |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |

# Allocating memory cells to cache rows: first attempt

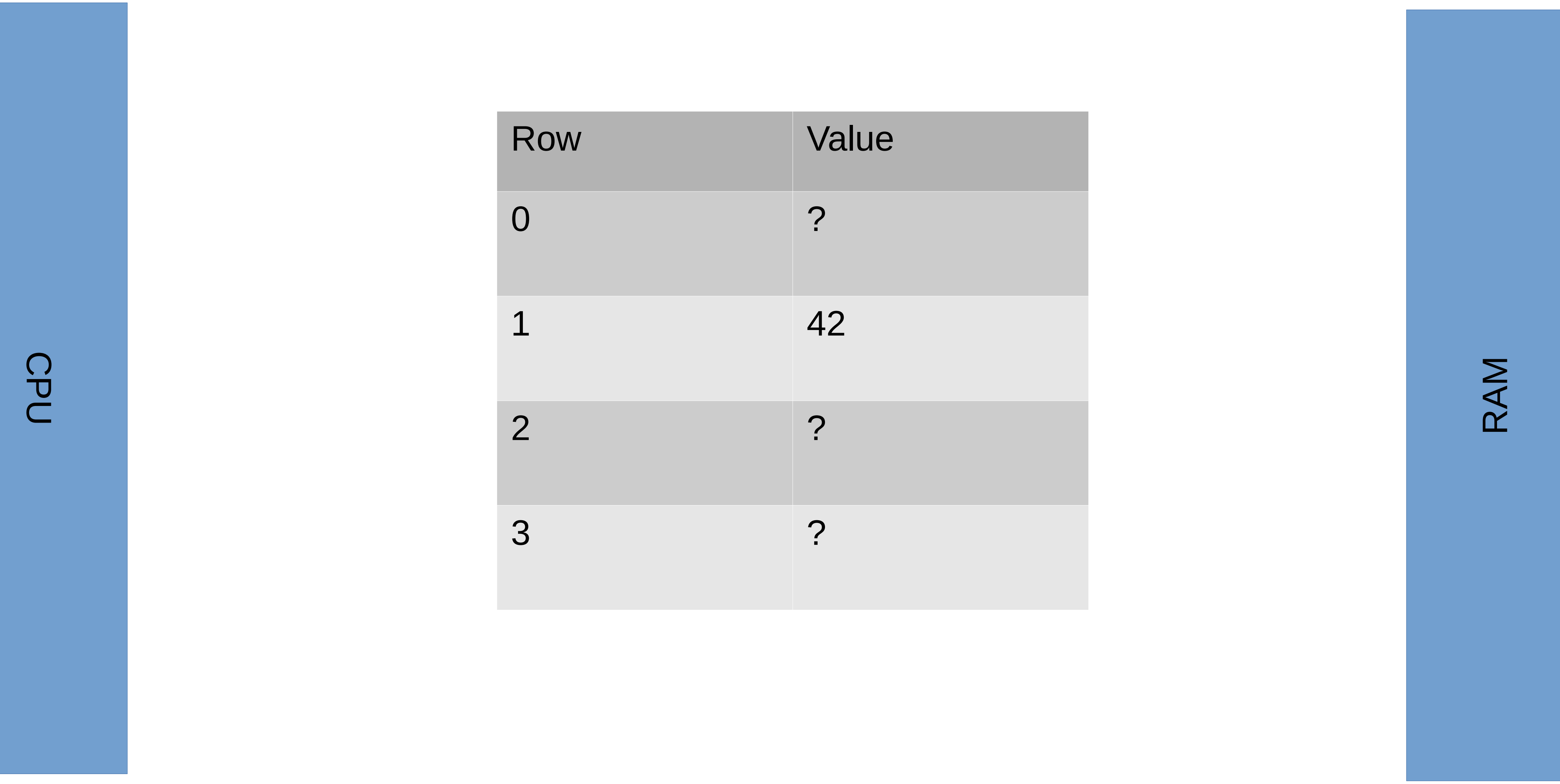


| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

row = address % numberofrows
1 = 97 % 4

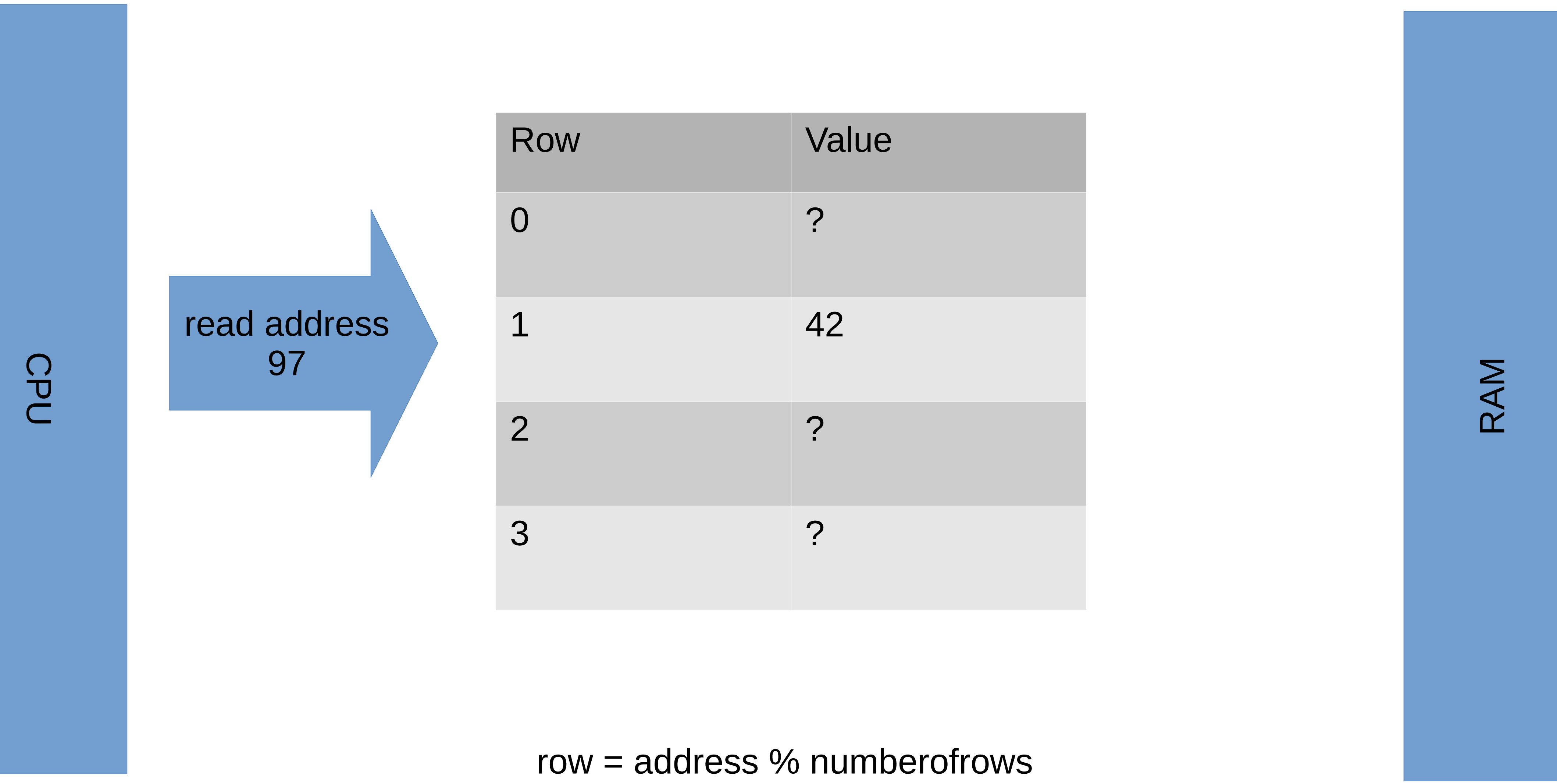
After a cache miss, the correct value is copied into the cache, then returned to the CPU.

## Allocating memory cells to cache rows: first attempt



| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

# Allocating memory cells to cache rows: first attempt

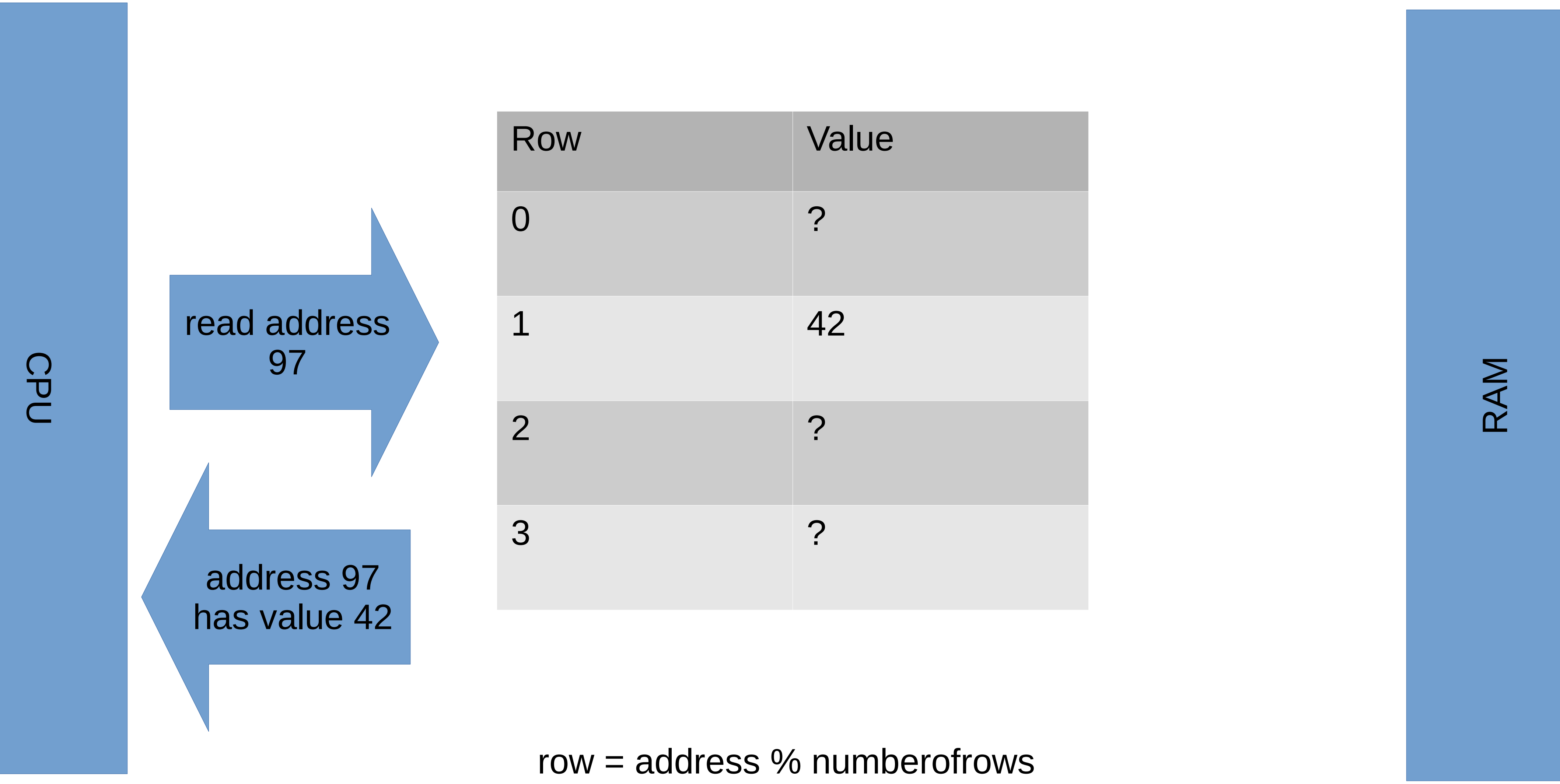


| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

row = address % numberofrows
1 = 97 % 4

If the CPU later requests the same address, it can be serviced from the cache. This is a *cache hit*.

# Allocating memory cells to cache rows: first attempt



| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

row = address % numberofrows
1 = 97 % 4

If the CPU later requests the same address, it can be serviced from the cache. This is a *cache hit*.

## Allocating memory cells to cache rows: first attempt

**What's wrong with this?**

CPU

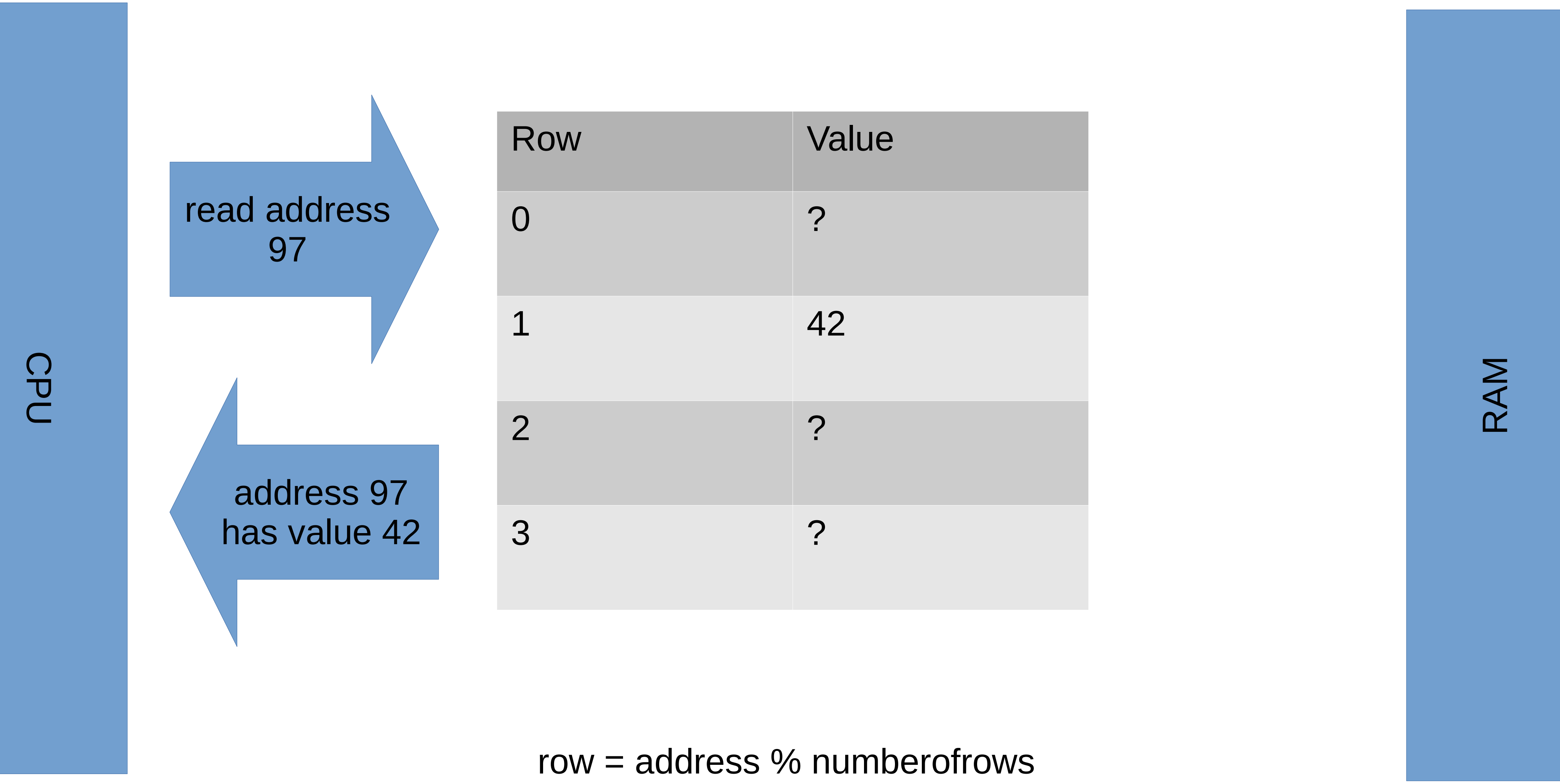
| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

RAM

row = address % numberofrows
1 = 97 % 4

## Allocating memory cells to cache rows: first attempt

**What's wrong with this?**

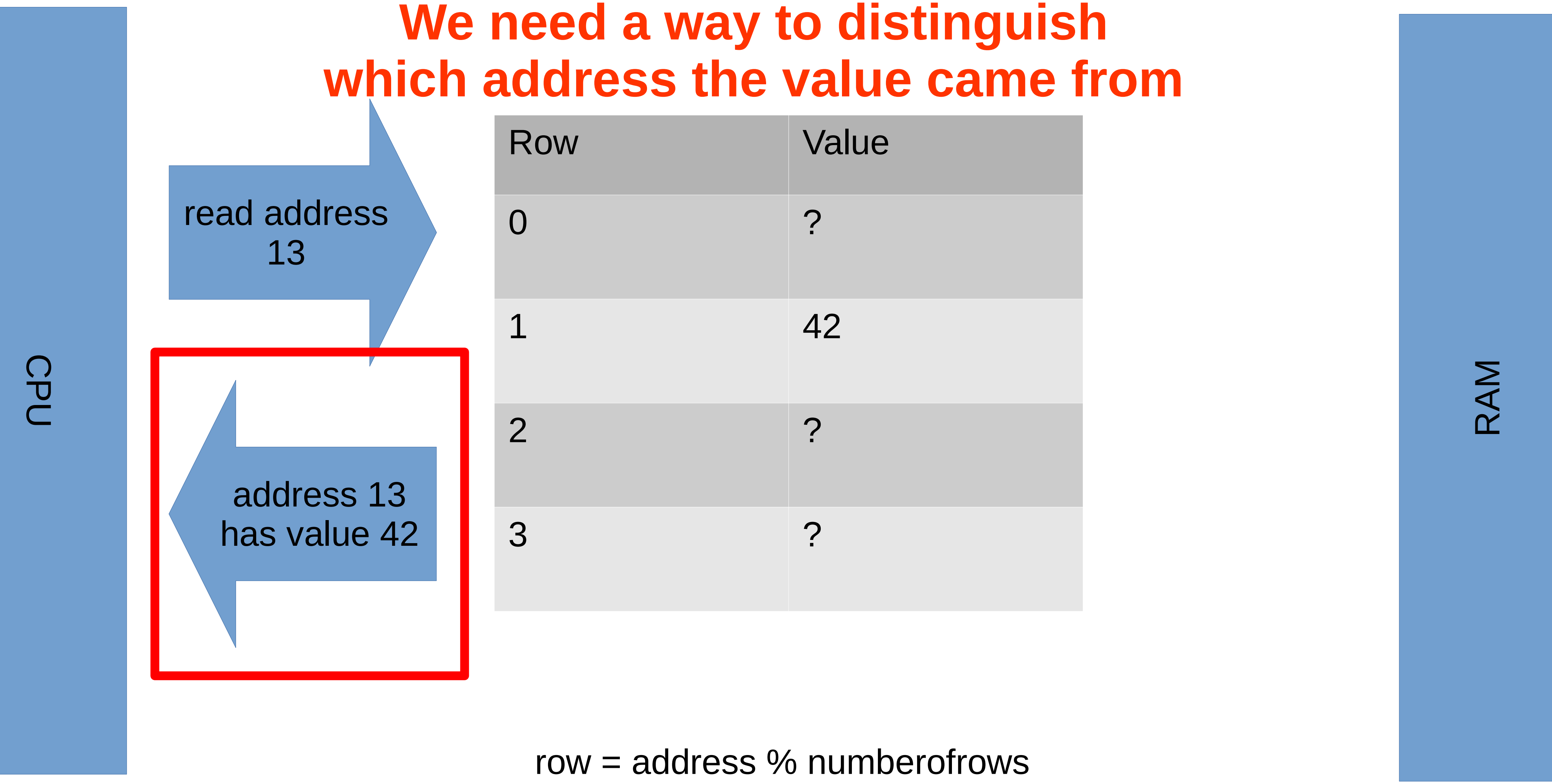


| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

row = address % numberofrows
1 = 97 % 4

## Allocating memory cells to cache rows: first attempt

**What's wrong with this?**
**We need a way to distinguish which address the value came from**

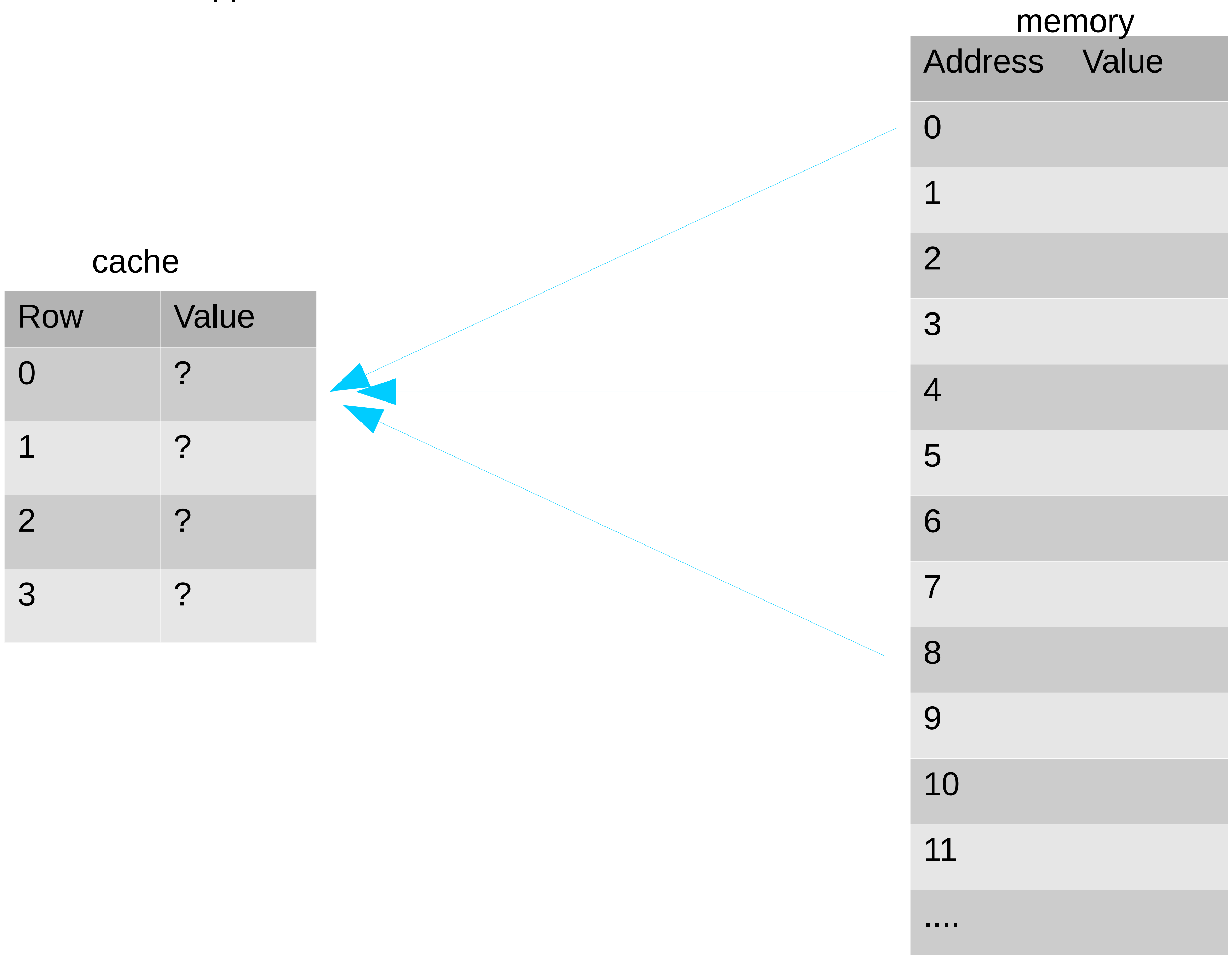


| Row | Value |
|---|---|
| 0 | ? |
| 1 | 42 |
| 2 | ? |
| 3 | ? |

row = address % numberofrows
1 = 13 % 4

We just returned the wrong value. We gave the value of address 97, when the CPU asked for 13.

Using this approach, in a cache with four rows, one quarter of addresses will be mapped to the same cache row.



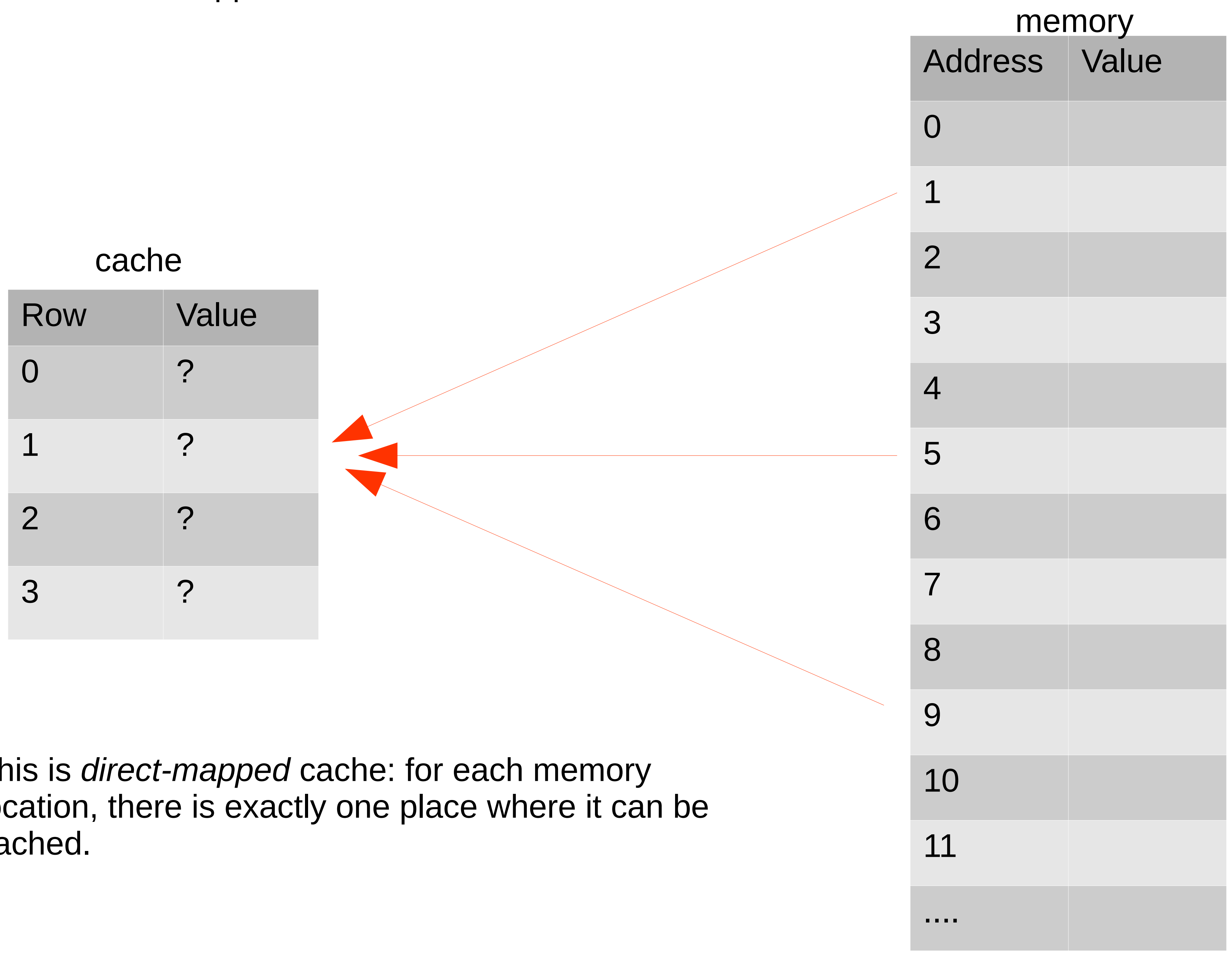
cache

| Row | Value |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |

memory

| Address | Value |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| .... | |

Using this approach, in a cache with four rows, one quarter of addresses will be mapped to the same cache row.



cache

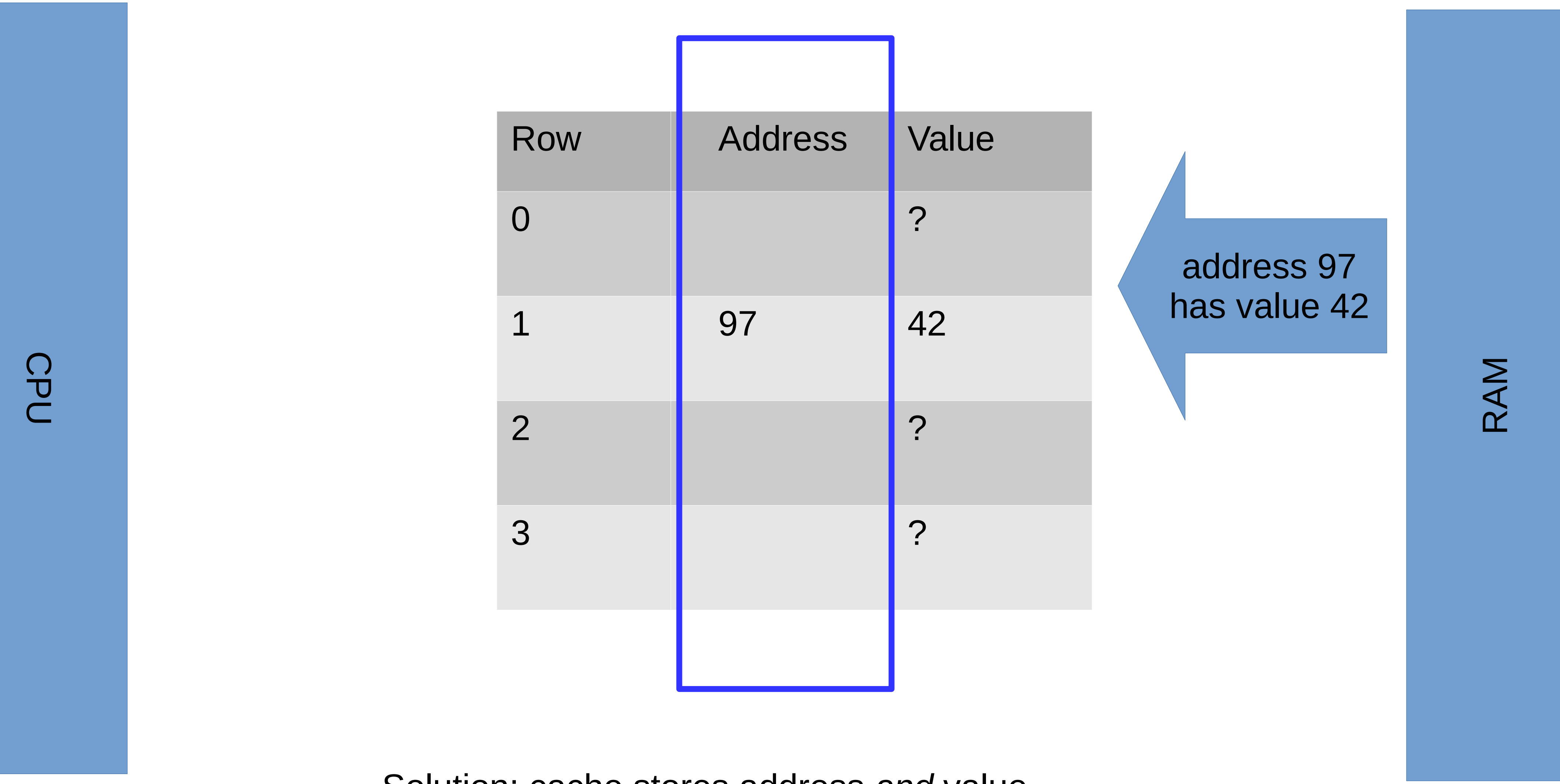
| Row | Value |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |

memory

| Address | Value |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| .... | |

This is *direct-mapped* cache: for each memory location, there is exactly one place where it can be cached.

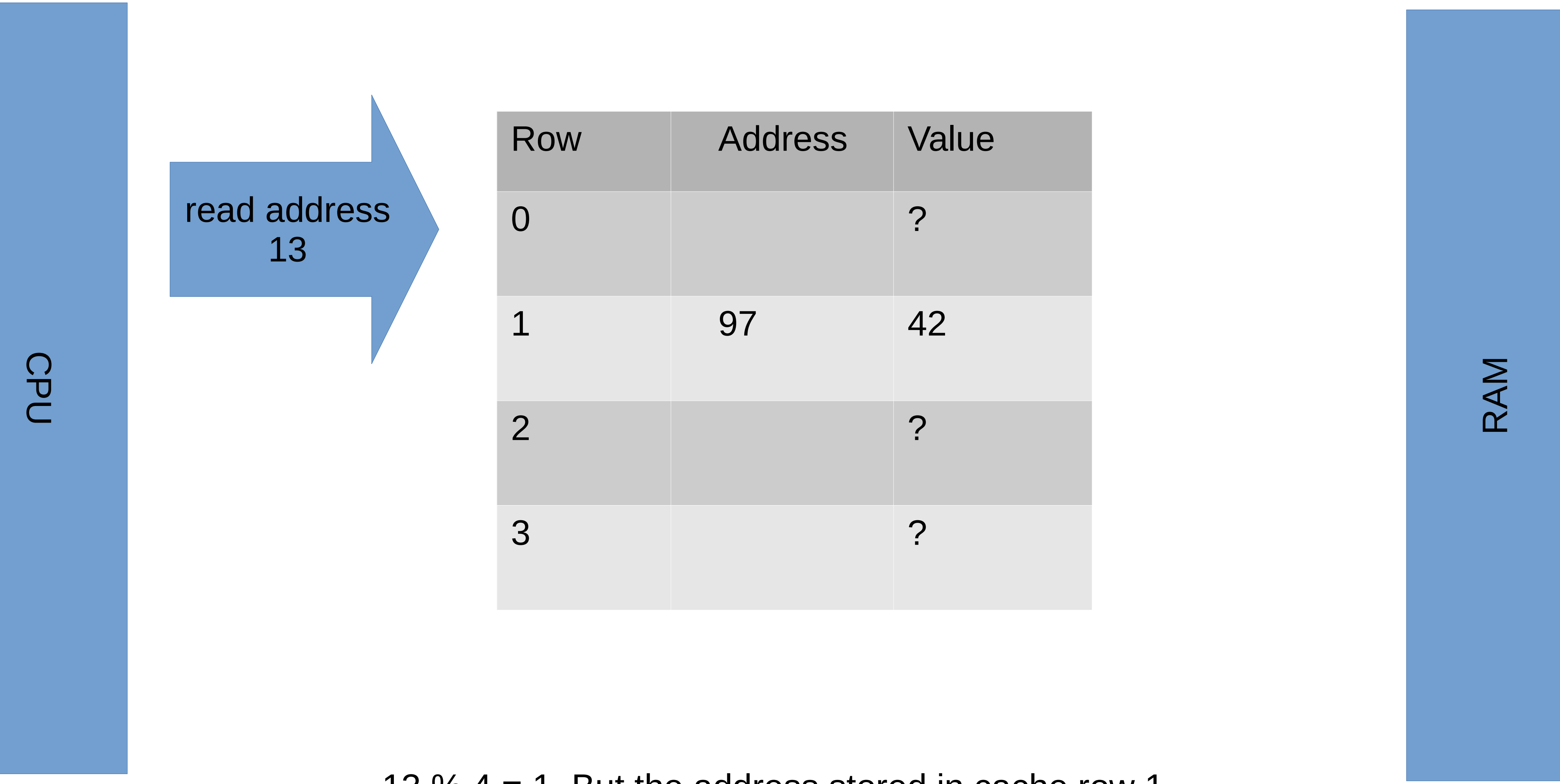
## Allocating memory cells to cache rows: second attempt



| Row | Address | Value |
|---|---|---|
| 0 | | ? |
| 1 | 97 | 42 |
| 2 | | ? |
| 3 | | ? |

Solution: cache stores address *and* value

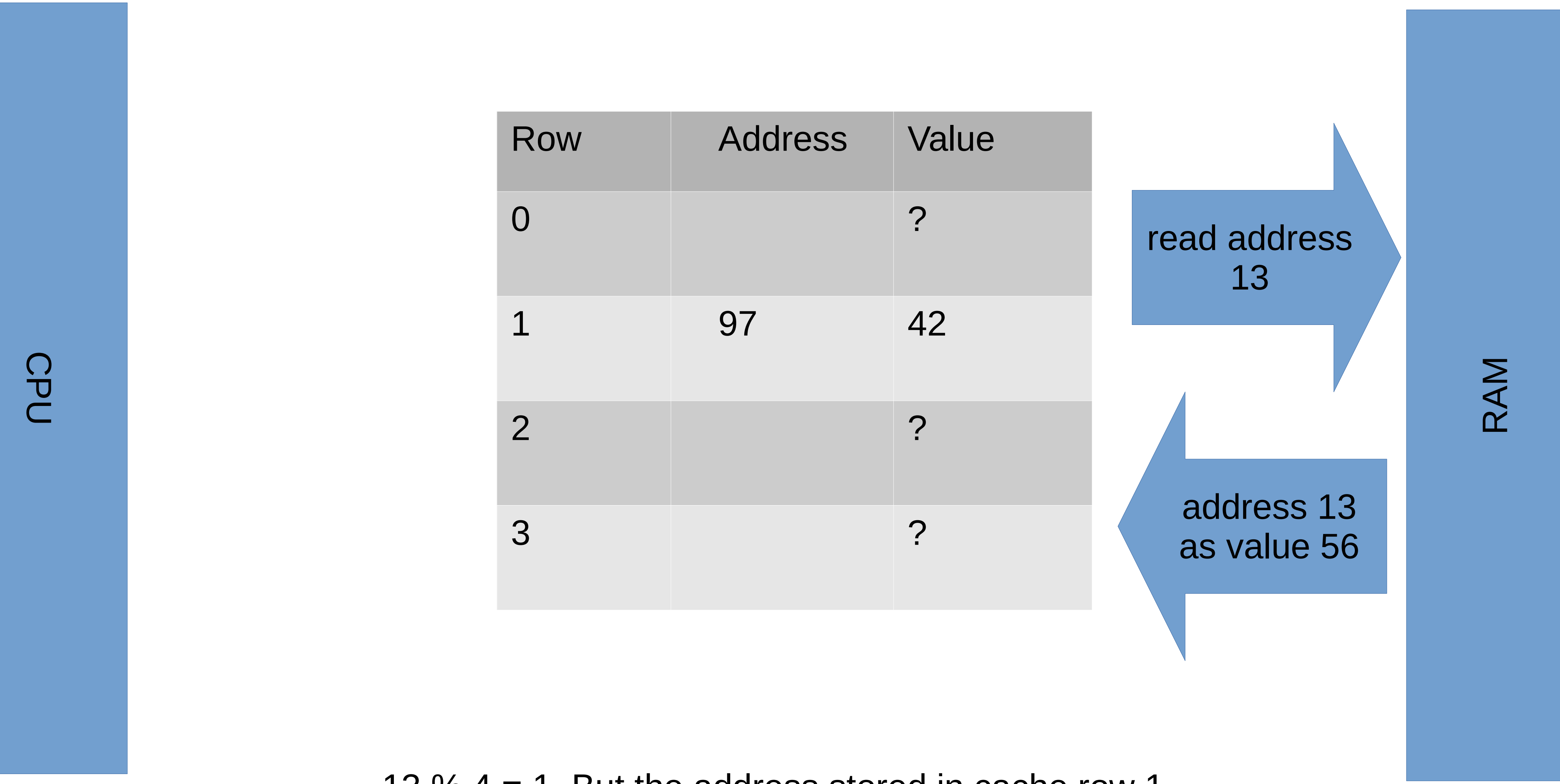
# Allocating memory cells to cache rows: second attempt



| Row | Address | Value |
|---|---|---|
| 0 | | ? |
| 1 | 97 | 42 |
| 2 | | ? |
| 3 | | ? |

13 % 4 = 1. But the address stored in cache row 1 doesn't match, so it's a cache miss. We consult memory and overwrite cache row 1. But at least we don't return the wrong value!

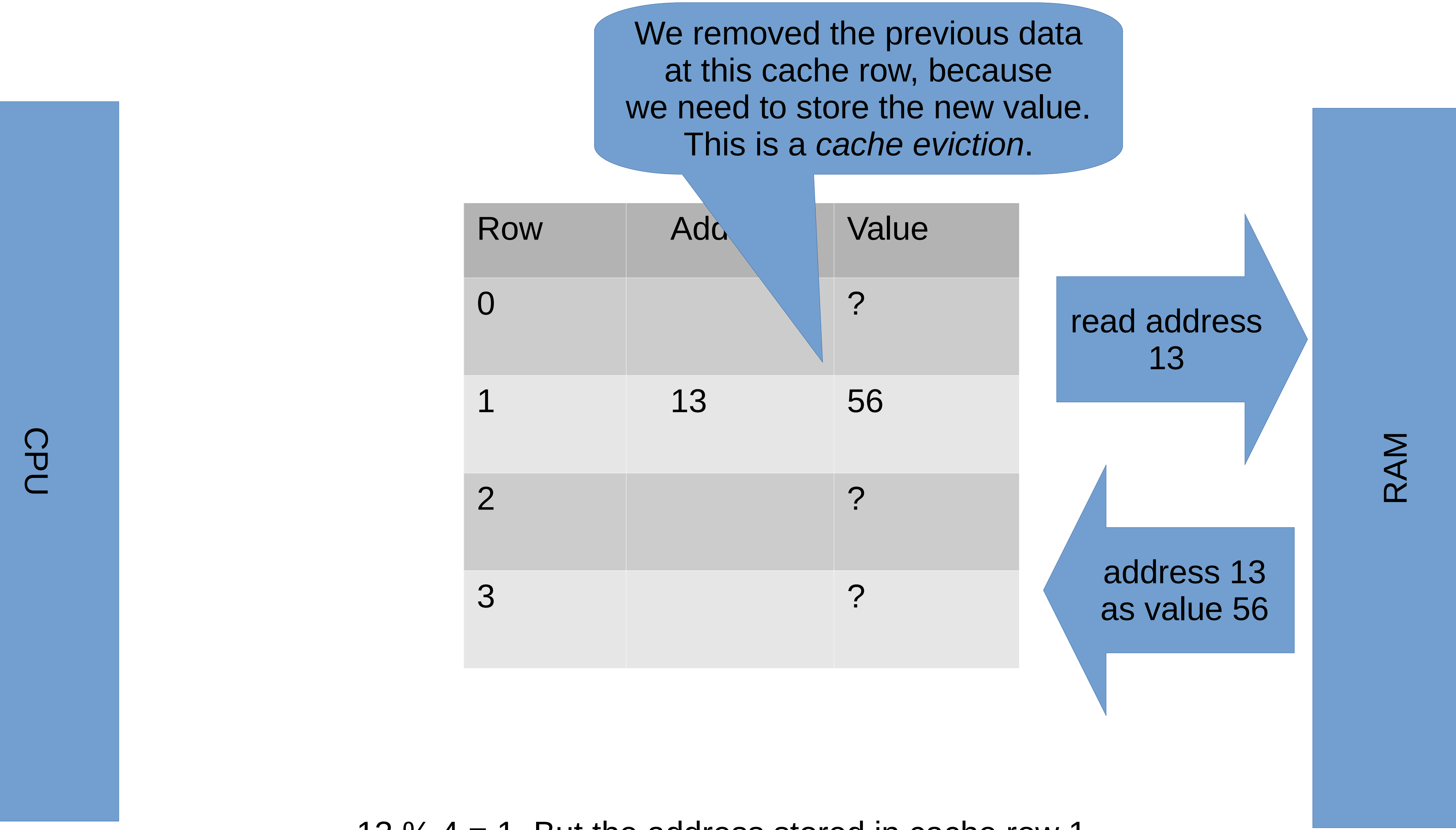
## Allocating memory cells to cache rows: second attempt



| Row | Address | Value |
|---|---|---|
| 0 | | ? |
| 1 | 97 | 42 |
| 2 | | ? |
| 3 | | ? |

13 % 4 = 1. But the address stored in cache row 1 doesn't match, so it's a cache miss. We consult memory and overwrite cache row 1. But at least we don't return the wrong value!

# Allocating memory cells to cache rows: second attempt



We removed the previous data at this cache row, because we need to store the new value. This is a *cache eviction*.

| Row | Add | Value |
|---|---|---|
| 0 | | ? |
| 1 | 13 | 56 |
| 2 | | ? |
| 3 | | ? |

13 % 4 = 1. But the address stored in cache row 1 doesn't match, so it's a cache miss. We consult memory and overwrite cache row 1. But at least we don't return the wrong value!

## Allocating memory cells to cache rows: third attempt

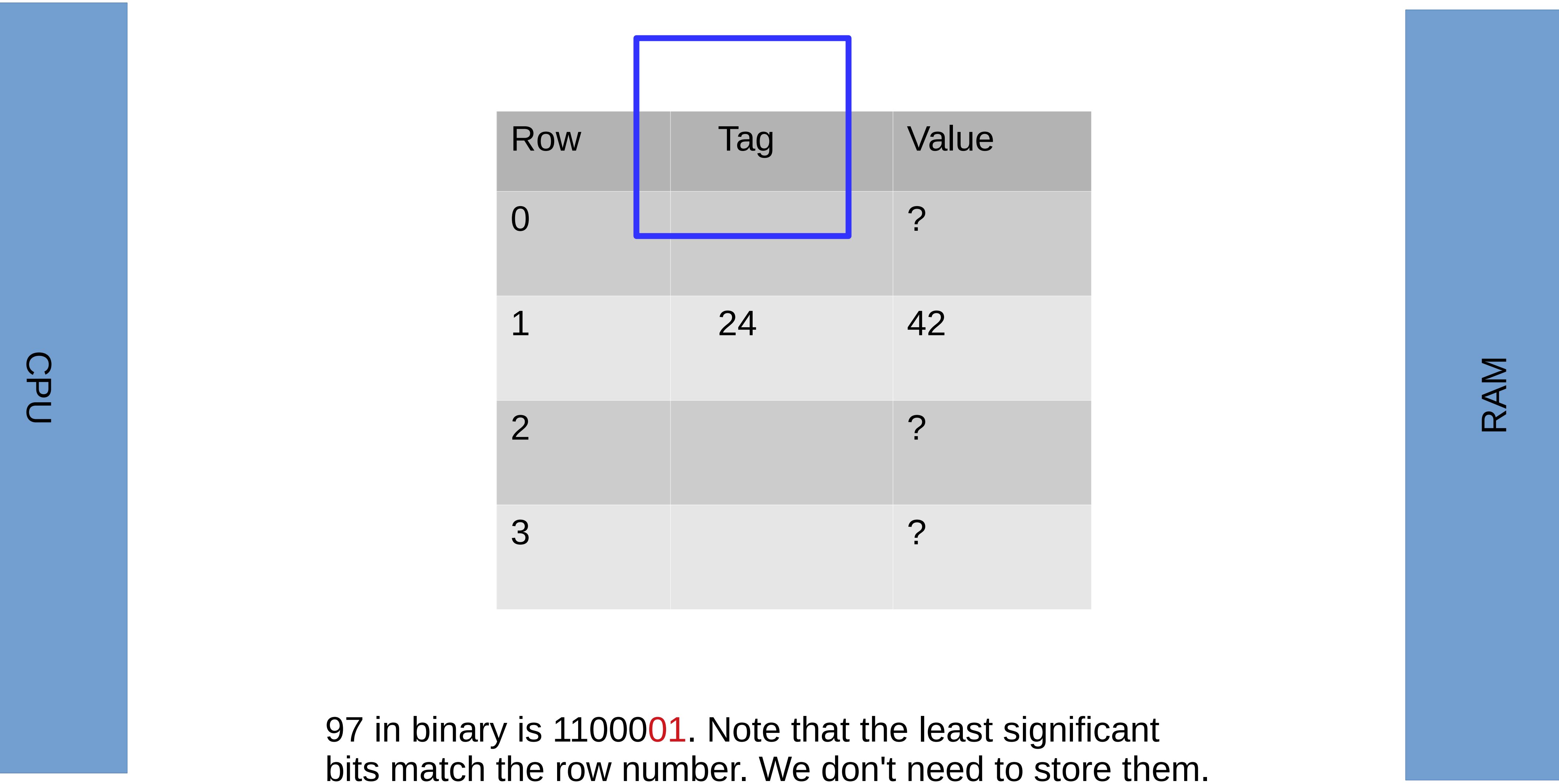
CPU

| Row | Address | Value |
|---|---|---|
| 0 | | ? |
| 1 | 97 | 42 |
| 2 | | ? |
| 3 | | ? |

RAM

97 in binary is 1100001. Note that the least significant bits match the row number. We don't need to store them.

# Allocating memory cells to cache rows: third attempt



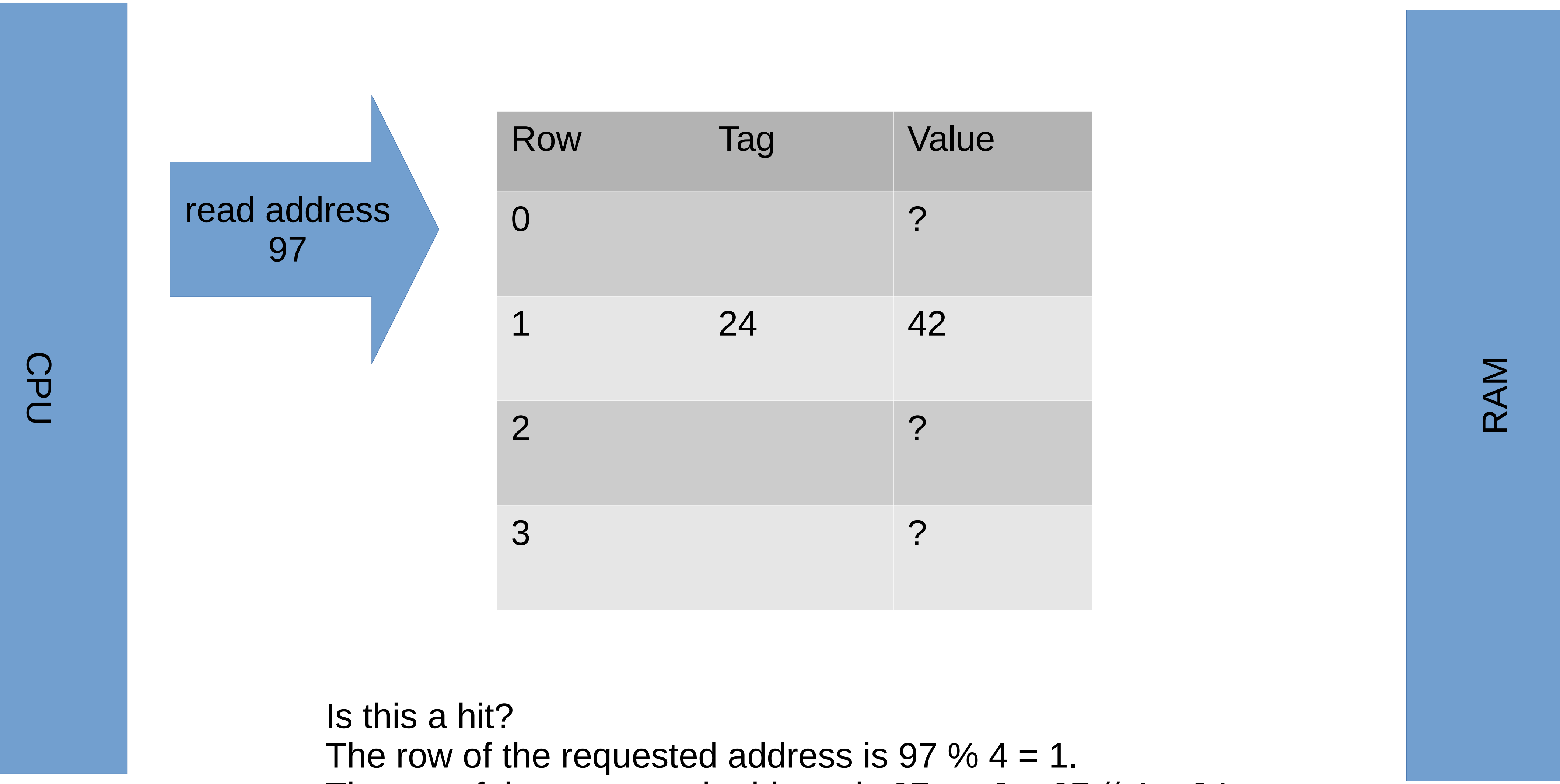
| Row | Tag | Value |
|---|---|---|
| 0 | | ? |
| 1 | 24 | 42 |
| 2 | | ? |
| 3 | | ? |

97 in binary is 1100001. Note that the least significant bits match the row number. We don't need to store them.

24 = 97 // 4 = 97 >> 2 This is the *tag*.

We shift by 2 because the row number is a 2-bit value. 2**2 == 4.

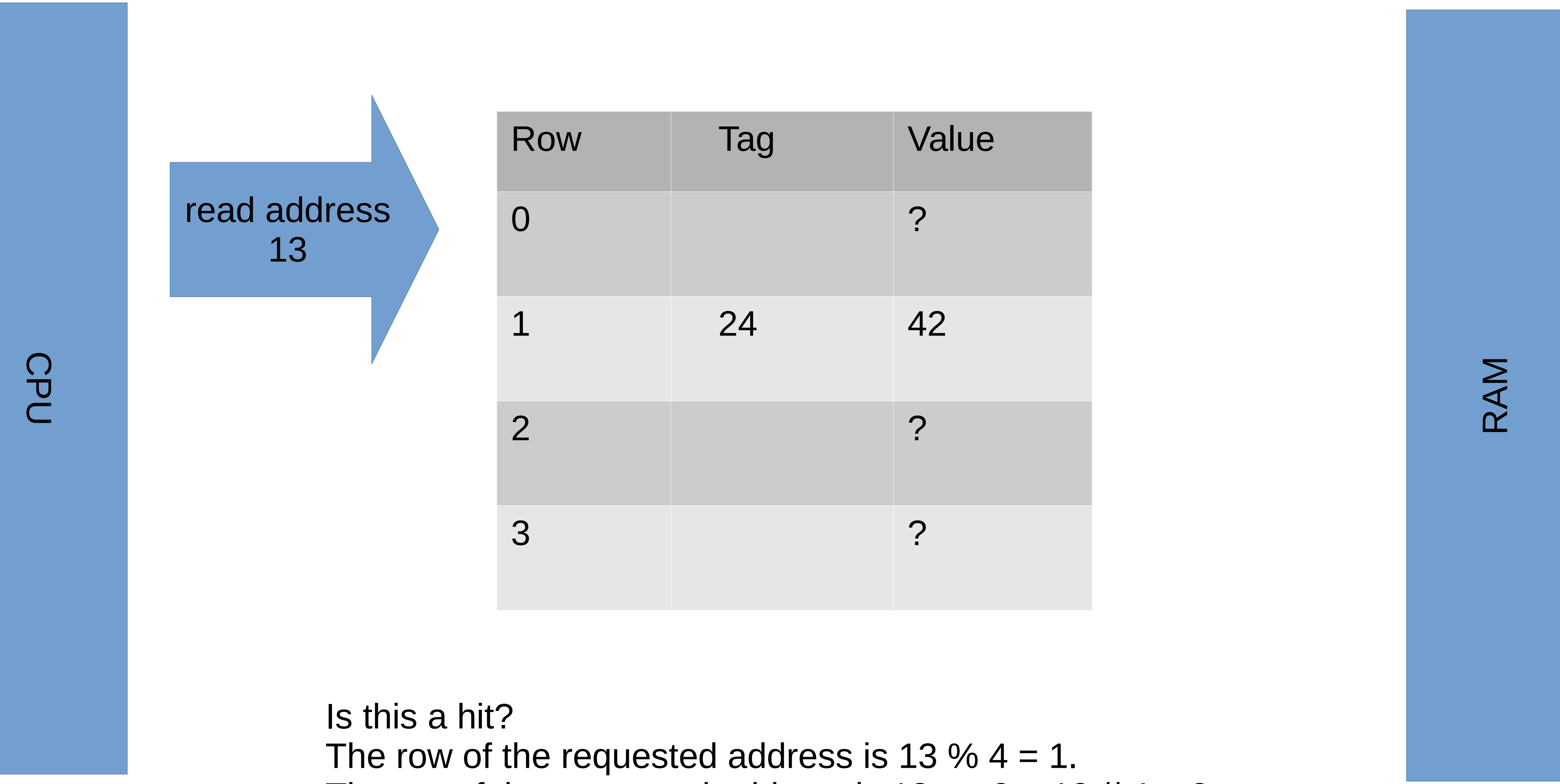
# Allocating memory cells to cache rows: third attempt



| Row | Tag | Value |
|---|---|---|
| 0 | | ? |
| 1 | 24 | 42 |
| 2 | | ? |
| 3 | | ? |

Is this a hit?
The row of the requested address is 97 % 4 = 1.
The tag of the requested address is 97 >> 2 = 97 // 4 = 24.
The cache will examine the tag stored at at row 1. It matches, so this is a HIT.

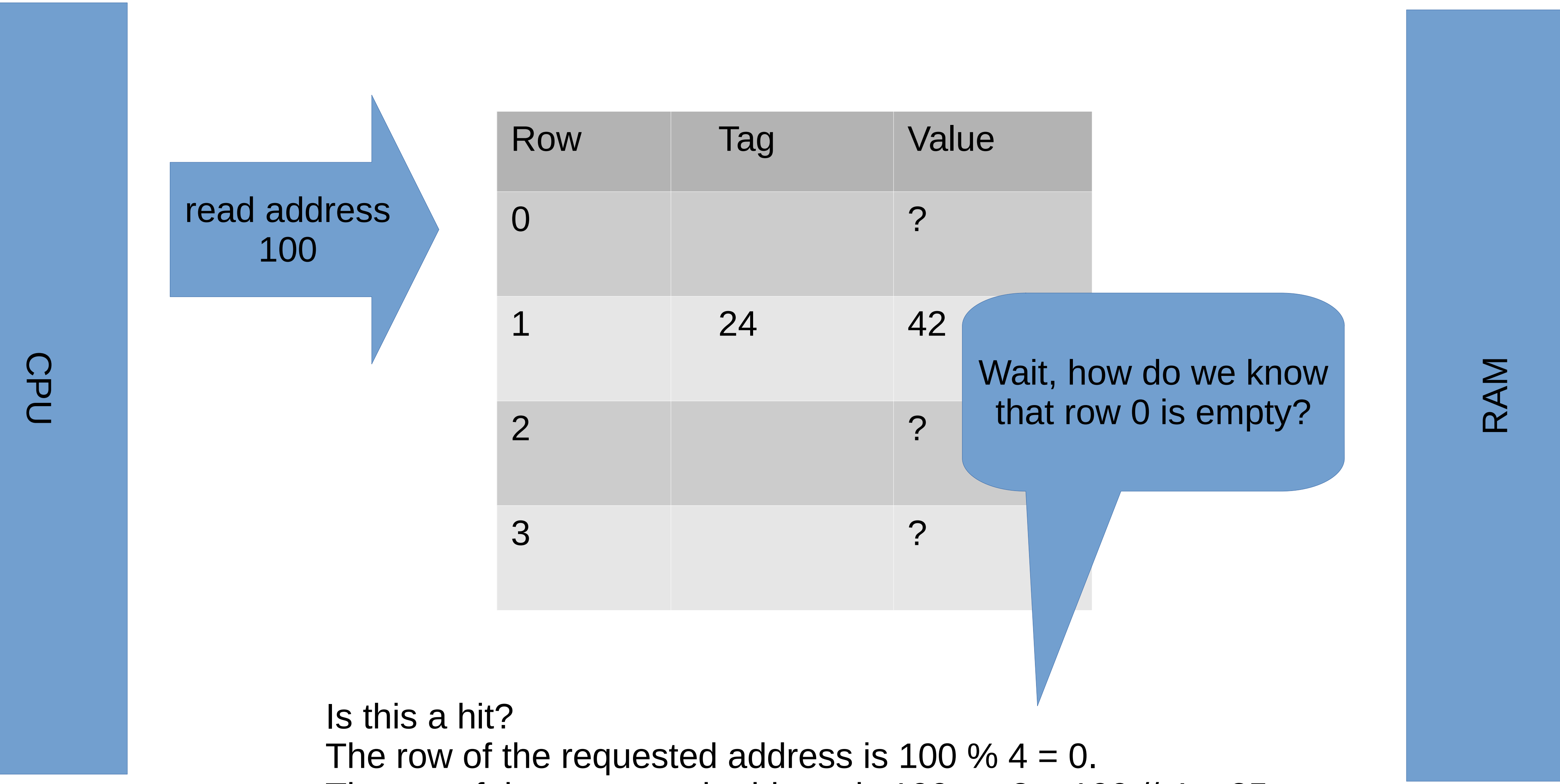
# Allocating memory cells to cache rows: third attempt



| Row | Tag | Value |
|---|---|---|
| 0 |  | ? |
| 1 | 24 | 42 |
| 2 |  | ? |
| 3 |  | ? |

Is this a hit?
The row of the requested address is 13 % 4 = 1.
The tag of the requested address is 13 >> 2 = 13 // 4 = 3.
The cache will examine the tag stored tag at row 1. It doesn't match, so this is a MISS.
We should *evict* the current block in row 1 and replace it.

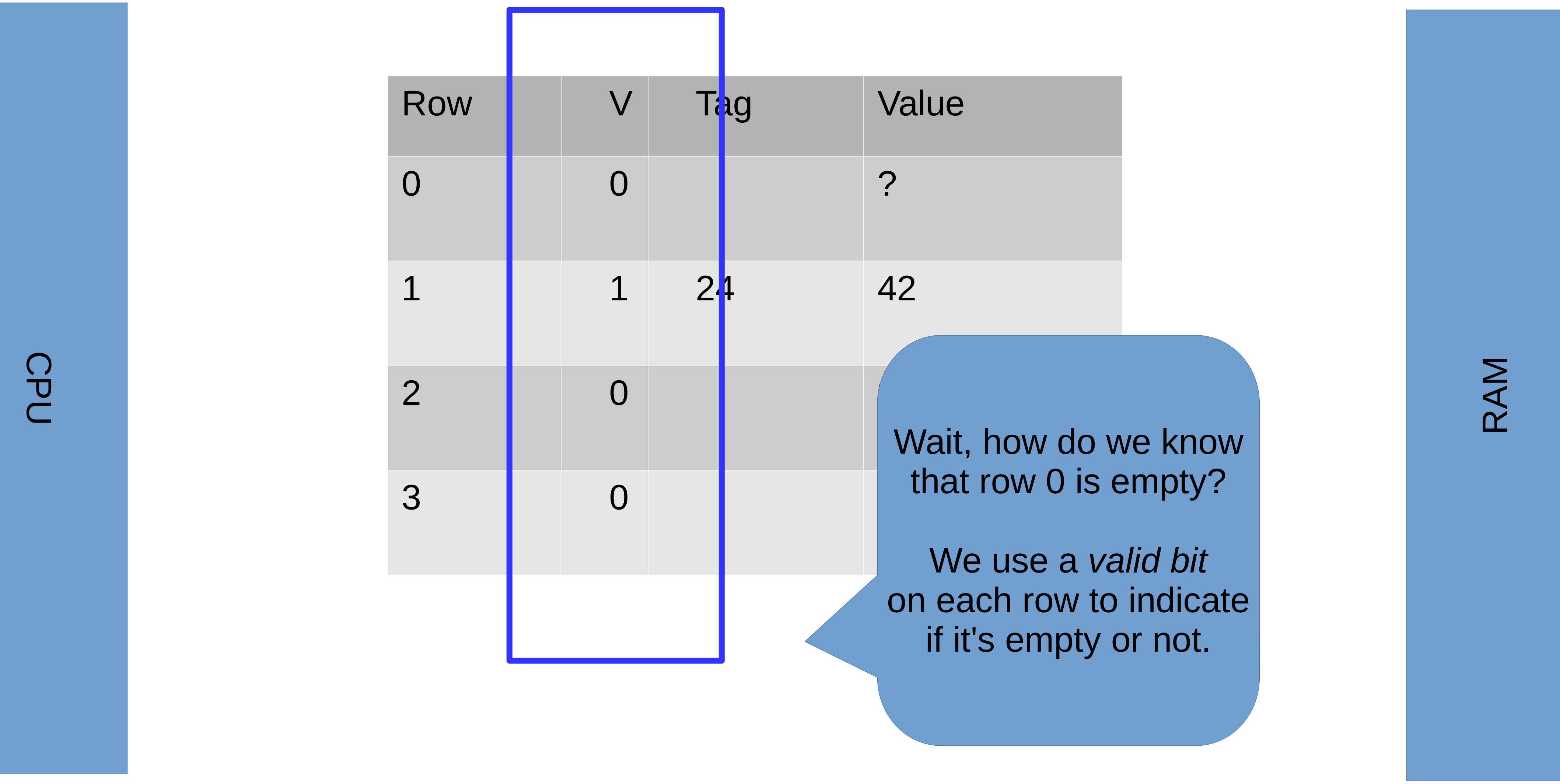
# Allocating memory cells to cache rows: third attempt



| Row | Tag | Value |
|---|---|---|
| 0 | | ? |
| 1 | 24 | 42 |
| 2 | | ? |
| 3 | | ? |

Is this a hit?
The row of the requested address is 100 % 4 = 0.
The tag of the requested address is 100 >> 2 = 100 // 4 = 25.
Row 0 is empty, so this is a MISS.

# Allocating memory cells to cache rows: fourth attempt

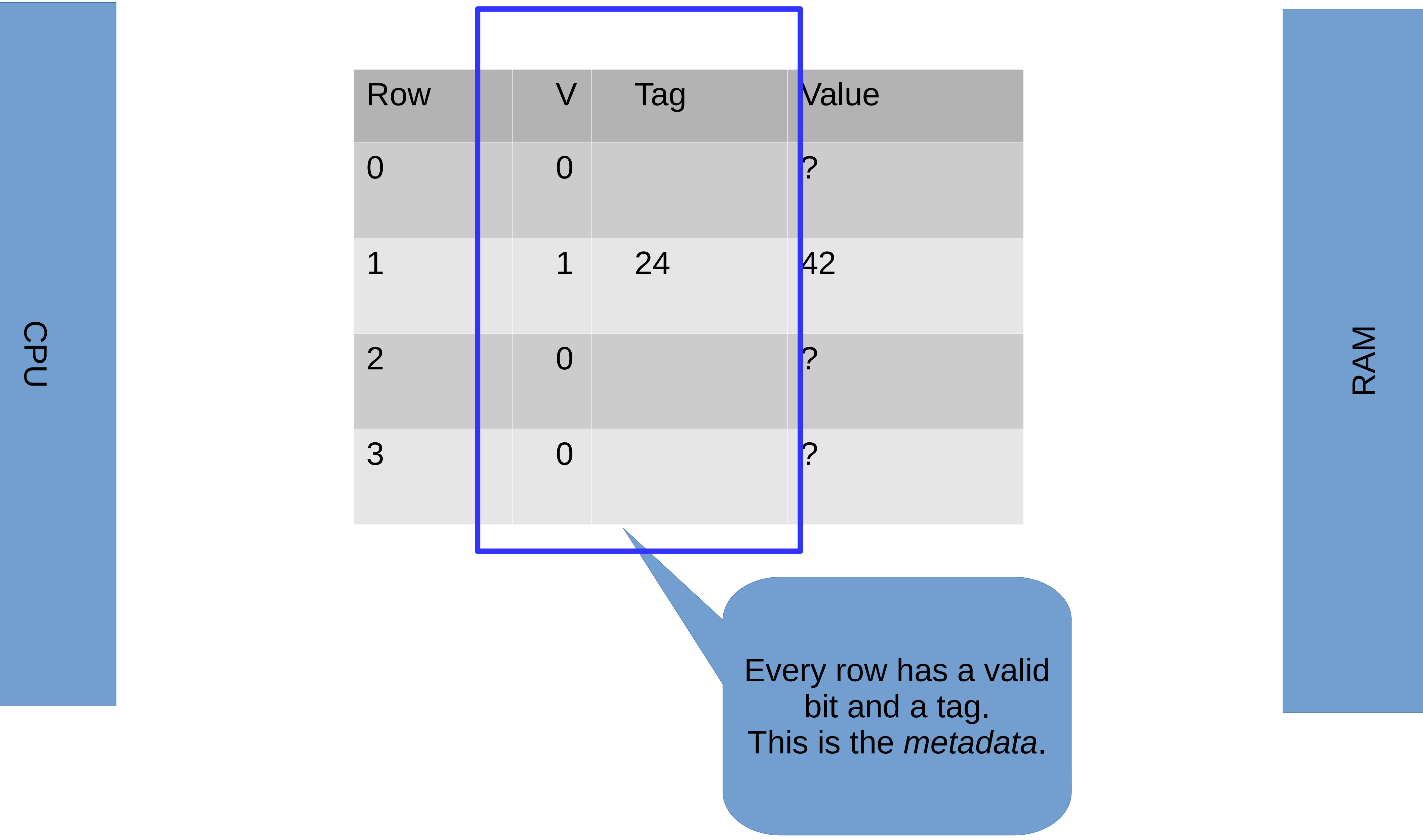


| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | ? |
| 1 | 1 | 24 | 42 |
| 2 | 0 | | |
| 3 | 0 | | |

Wait, how do we know that row 0 is empty?

We use a *valid bit* on each row to indicate if it's empty or not.

# Allocating memory cells to cache rows: fourth attempt



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | ? |
| 1 | 1 | 24 | 42 |
| 2 | 0 | | ? |
| 3 | 0 | | ? |

Every row has a valid bit and a tag.
This is the *metadata*.

A goal of our cache design is to increase our *hit ratio*.

$$hitratio = \frac{hits}{hits + misses}$$

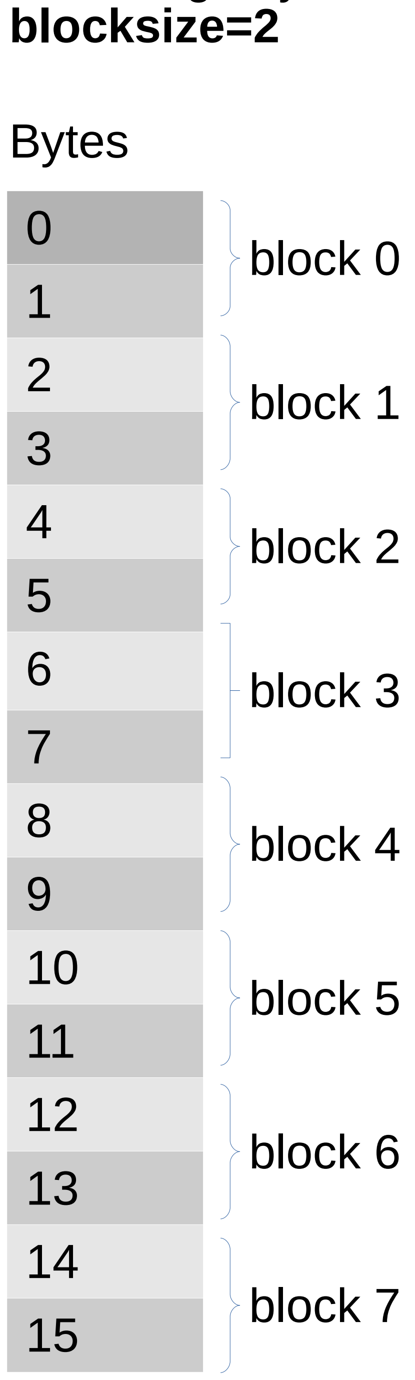
```
int my_array[1024] = {....};

int main() {
    int sum = 0;
    for (int i=0; i < 1024; i++)
        sum += my_array[i];
    return sum;
}
```

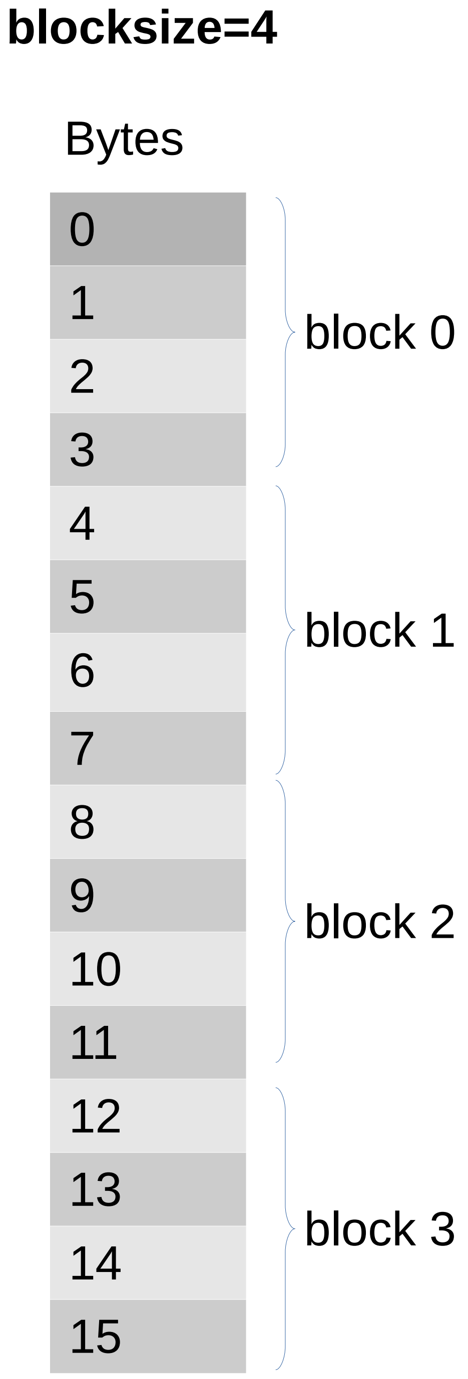
Considering code like this, it's clear that memory accesses are not arbitrary: we frequently access memory cells adjacent to previously-accessed cells.

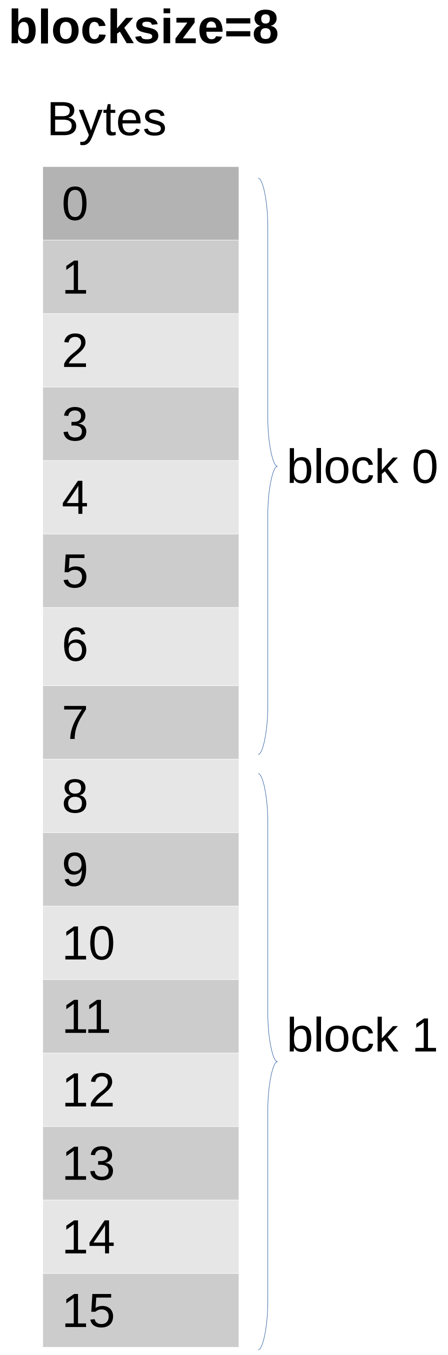
This principle is called *spatial locality*.

We can improve hit ratio if our cache contains data that has **not yet been requested by the CPU, but is likely to be requested soon**.

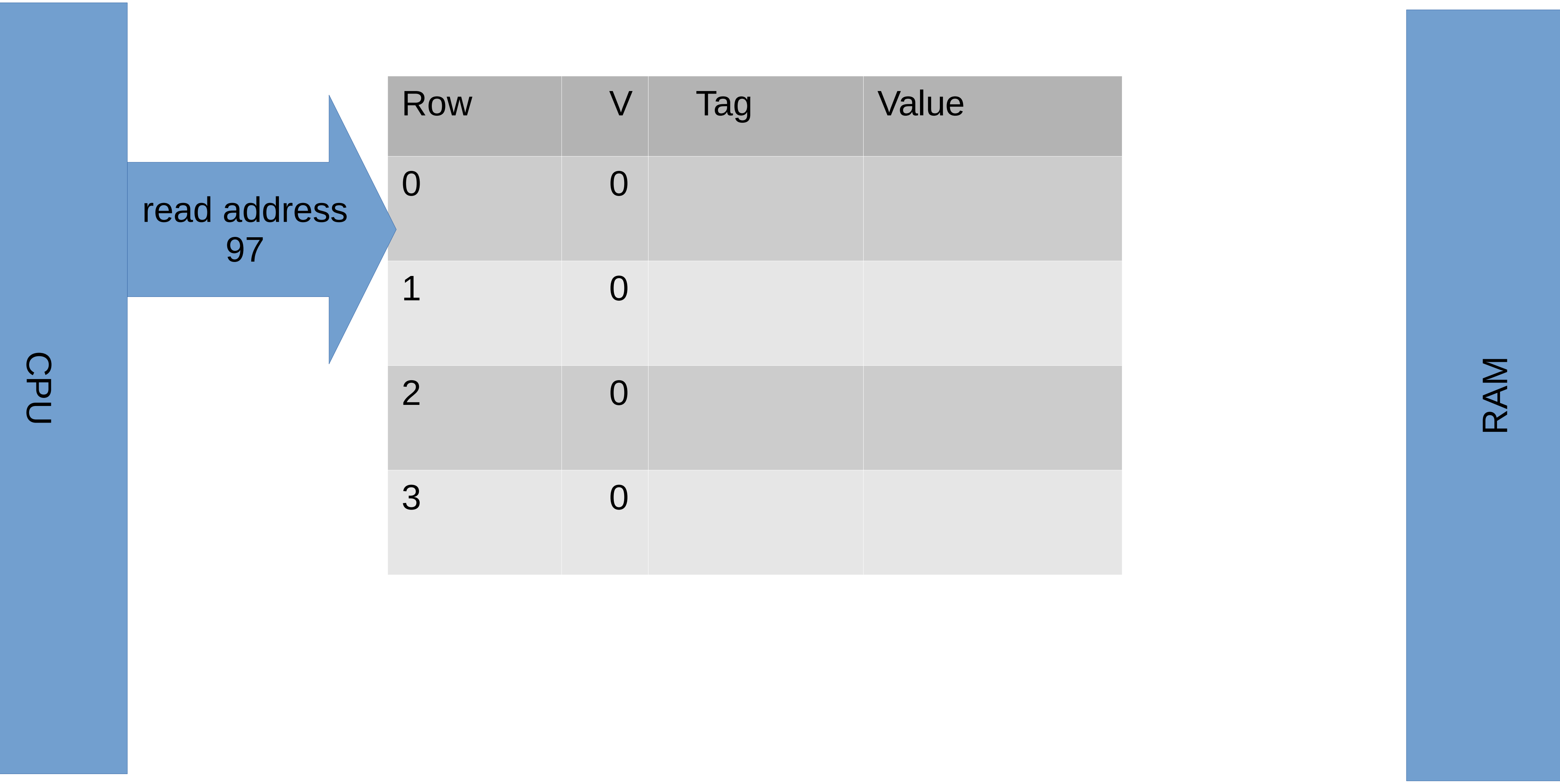
Proposal: let's divide memory into *blocks*, consisting of *n* cells (typically a power of 2). Accessing any cell in a block will cause the entire block to be cached.





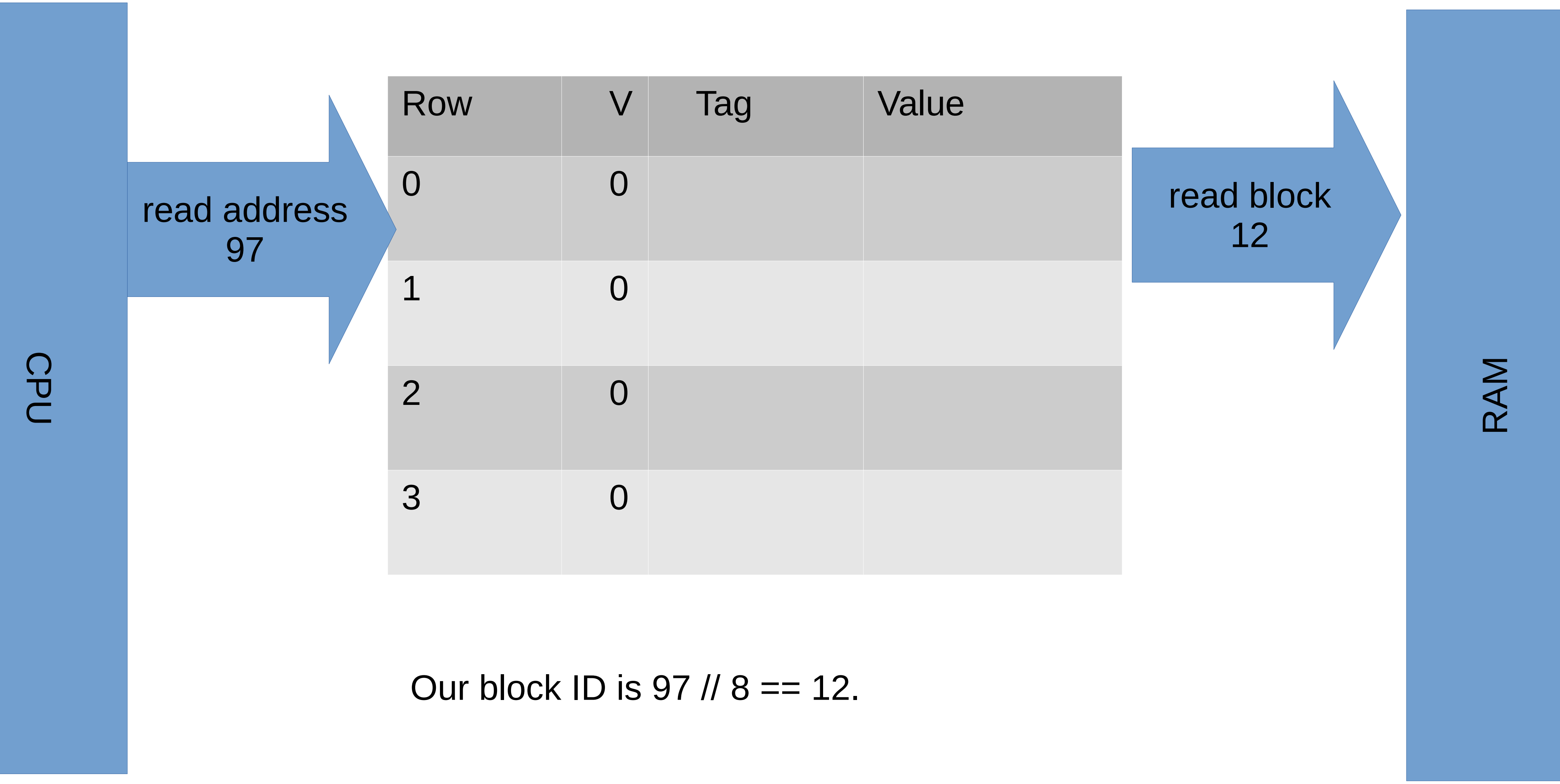


## Our cache has four rows and a block size of 8



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 0 | | |
| 2 | 0 | | |
| 3 | 0 | | |

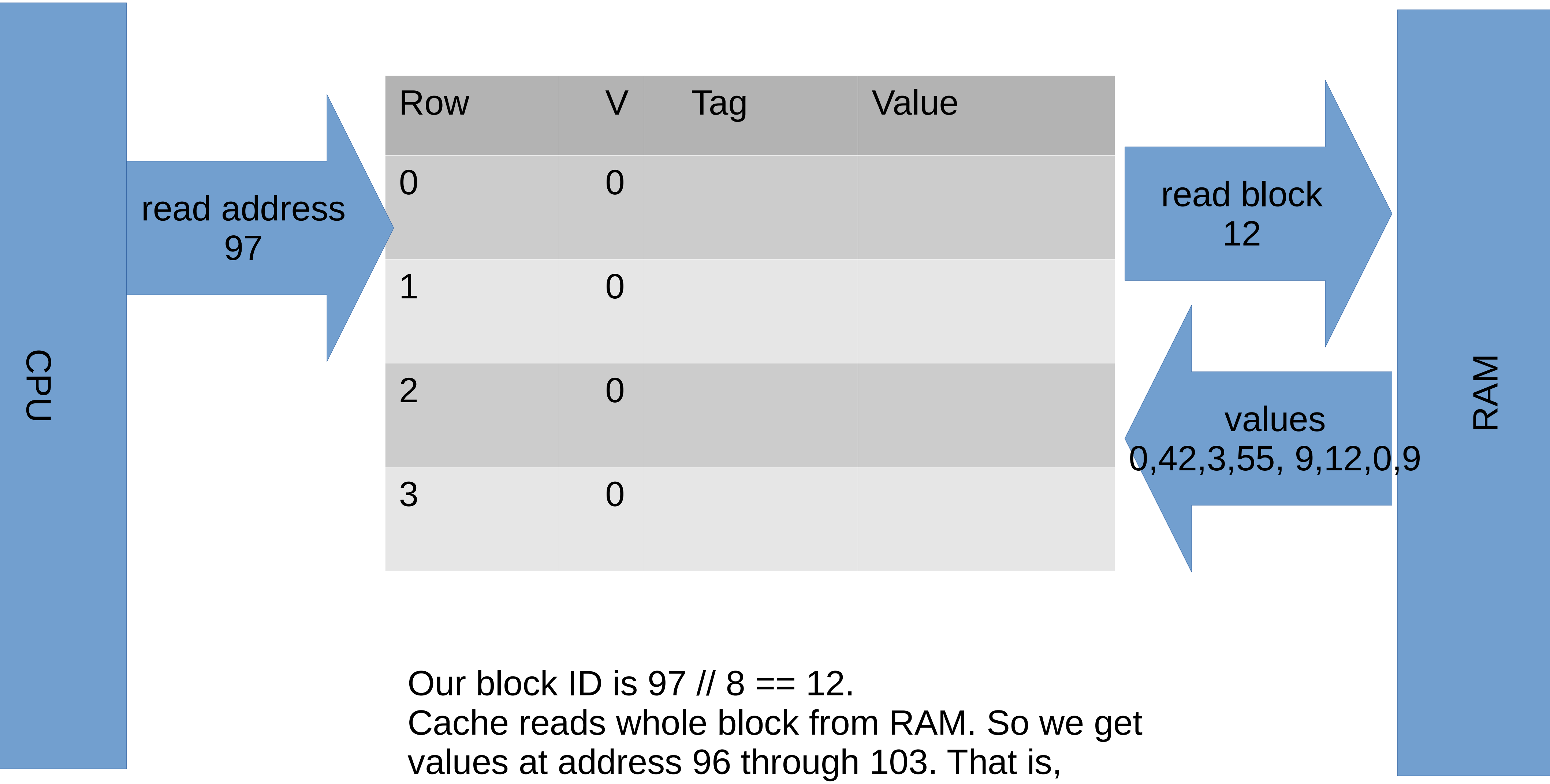
## Our cache has four rows and a block size of 8



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 0 | | |
| 2 | 0 | | |
| 3 | 0 | | |

Our block ID is 97 // 8 == 12.

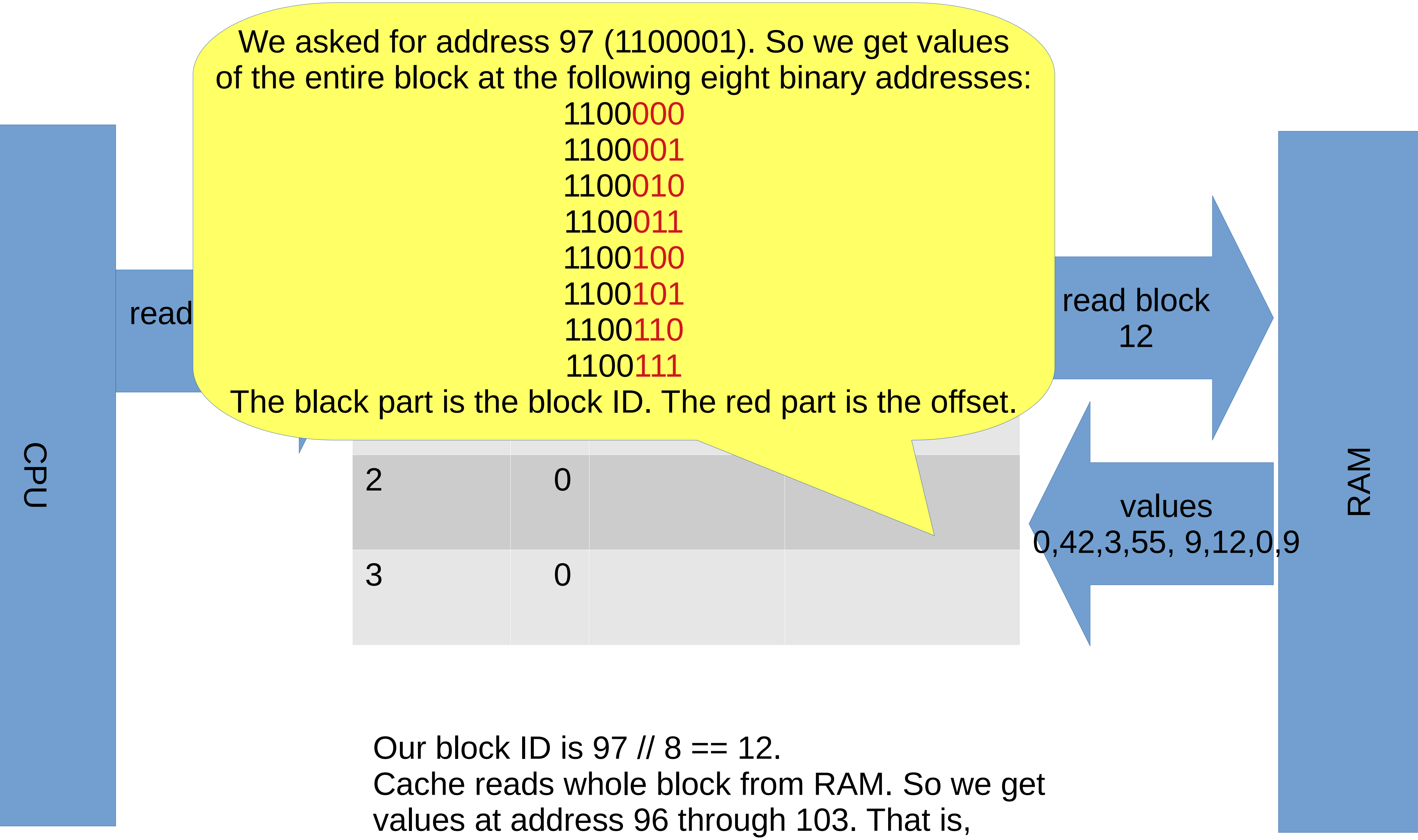
# Our cache has four rows and a block size of 8



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 0 | | |
| 2 | 0 | | |
| 3 | 0 | | |

Our block ID is 97 // 8 == 12.
Cache reads whole block from RAM. So we get values at address 96 through 103. That is,
97//8*8 through 97//8*8 + (8-1)

## Our cache has four rows and a block size of 8



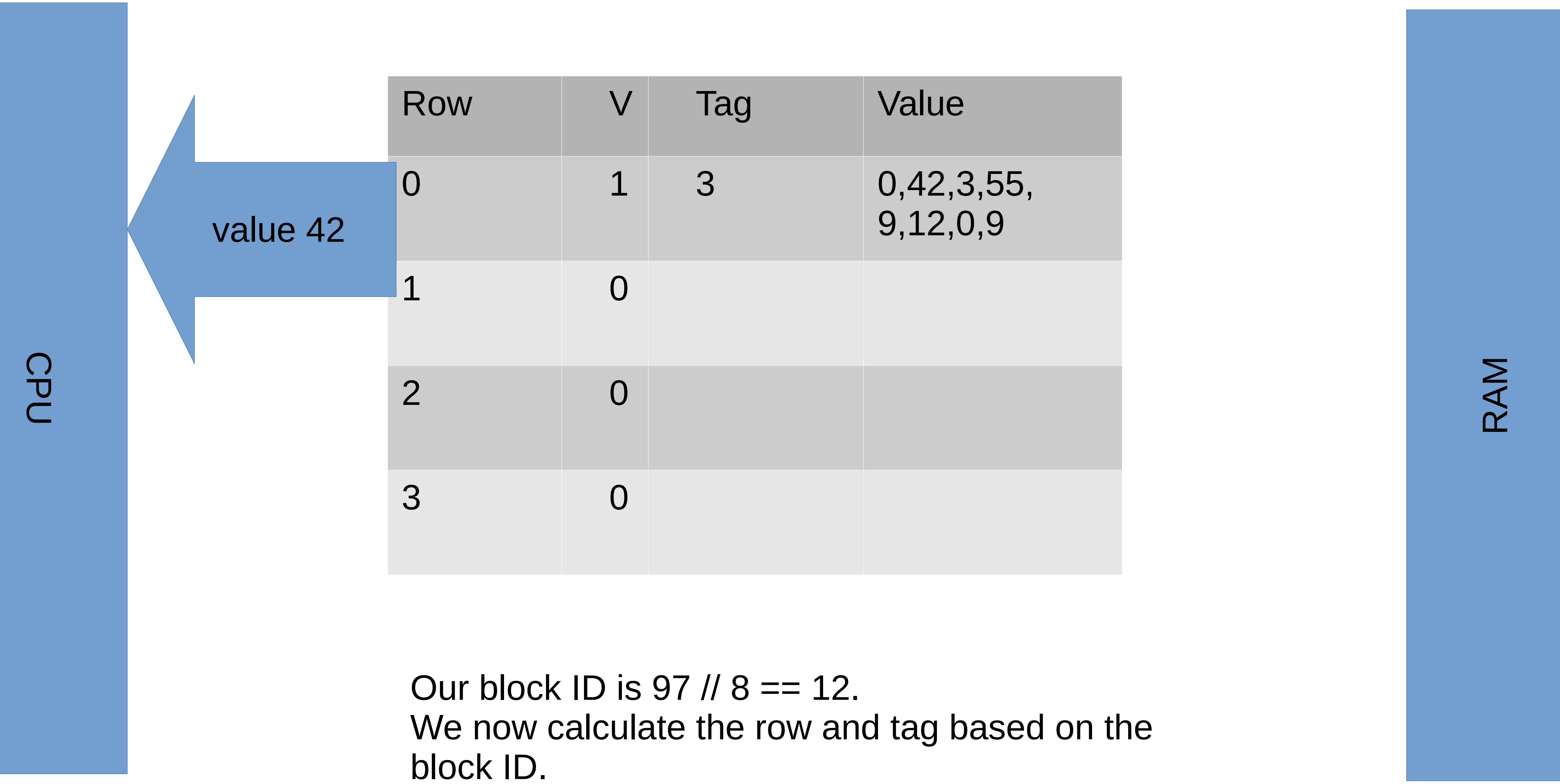
We asked for address 97 (1100001). So we get values of the entire block at the following eight binary addresses:

1100000
1100001
1100010
1100011
1100100
1100101
1100110
1100111

The black part is the block ID. The red part is the offset.

Our block ID is 97 // 8 == 12.
Cache reads whole block from RAM. So we get values at address 96 through 103. That is,
97//8*8 through 97//8*8 + (8-1)

# Our cache has four rows and a block size of 8



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 3 | 0,42,3,55,<br>9,12,0,9 |
| 1 | 0 | | |
| 2 | 0 | | |
| 3 | 0 | | |

Our block ID is 97 // 8 == 12.
We now calculate the row and tag based on the block ID.

- row = blockid % 4 = 0
- tag = blockid >> 2 = blockid // 4 = 3

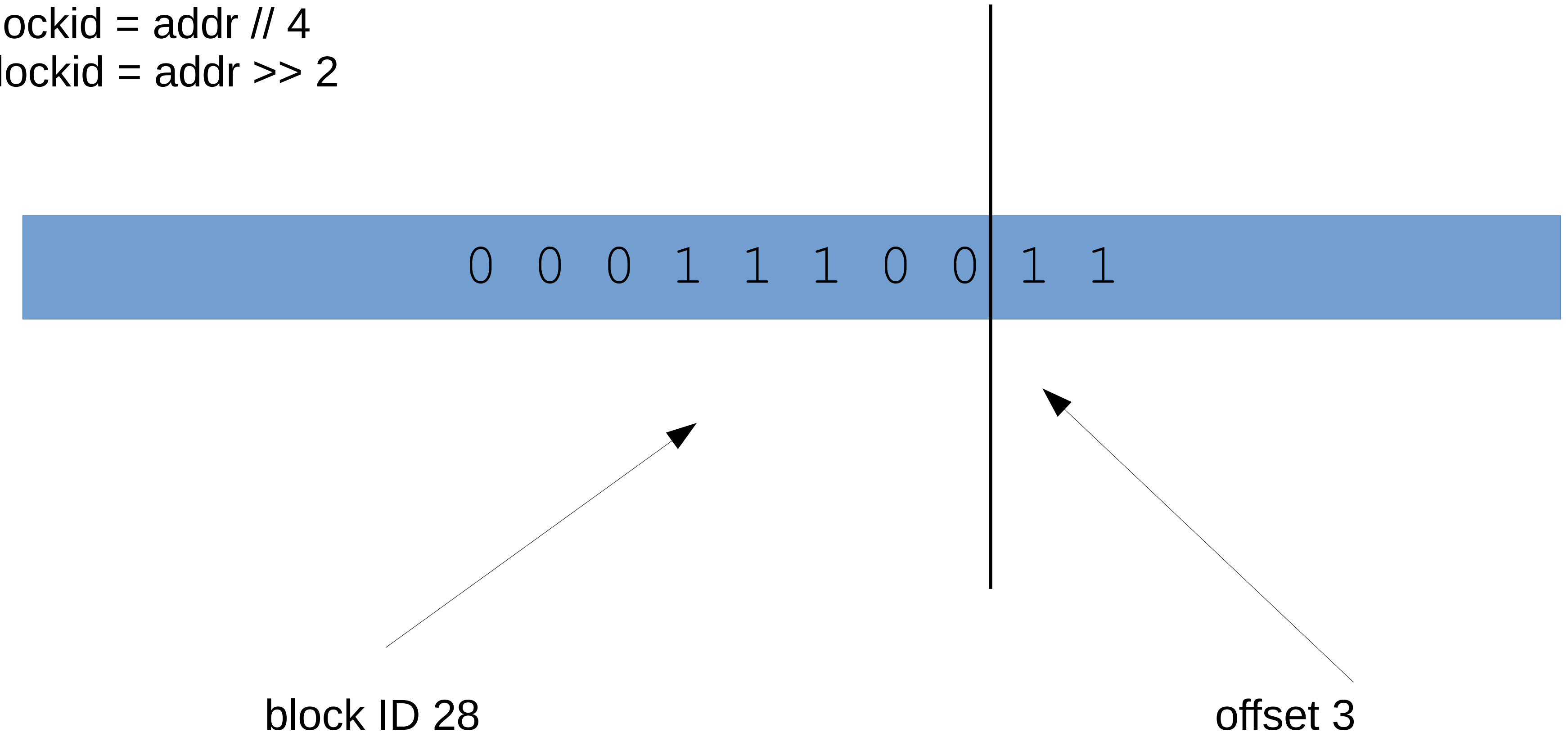
On a system with 10-bit addresses and blocksize of 4....

A blocksize of 4 means we need 2 bits to express an address within a block, and 8 bits to express the block ID.

Here, address 115 (binary 0001110011) is in block 28
(along with bytes 112, 113, and 114).

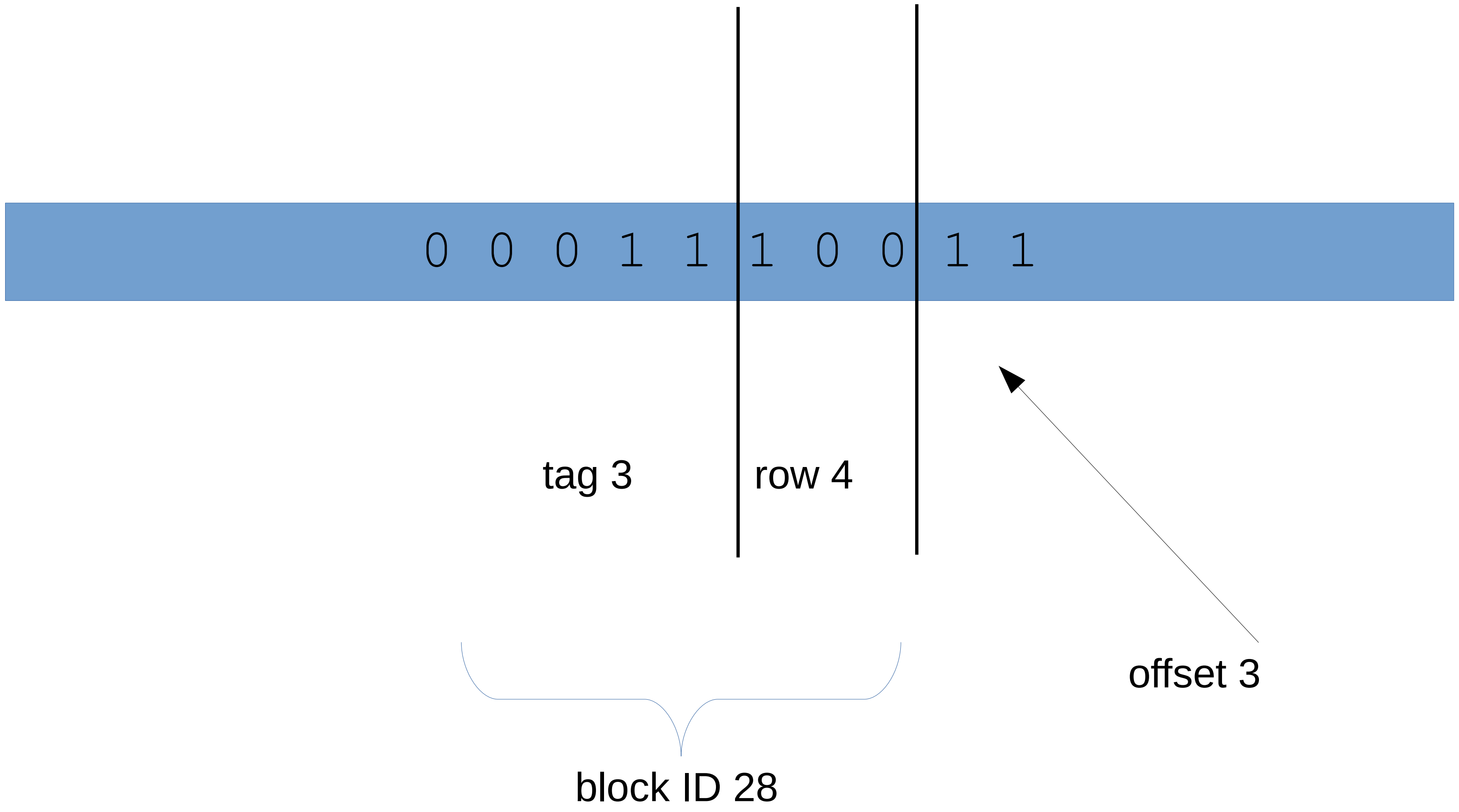
We can calculate the block ID with division or shifting.

```
blockid = addr // 4
blockid = addr >> 2
```



On a system with 10-bit addresses and blocksize of 4....

Furthermore, if there are 8 cache rows, then we know that the block ID will be further divided as follows:



## An exercise

Consider a cache (as previously) with 4 rows and a blocksize of 8. You are given below a sequence of memory address. For each memory address, calculate its cache row and tag.

```
[89, 106, 161, 85, 88, 124, 159, 104, 76, 90]
```

(That is, some program is executing a `lw` instruction on each of these addresses. Presumably the program is doing other stuff, as well, but at the moment we are concerned only with its memory accesses.)

Then, mentally simulate a cache system accessing those addresses in sequence. Determine which of them will be HITS and which MISSES. Assume the cache is initially empty.

## An exercise

Consider a cache (as previously) with 4 rows and a blocksize of 8. You are given below a sequence of memory address. For each memory address, calculate its row and tag.

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | | | |
| 106 | | | |
| 161 | | | |
| 85 | | | |
| 88 | | | |
| 124 | | | |
| 159 | | | |
| 104 | | | |
| 76 | | | |
| 90 | | | |

Then, mentally simulate a cache system accessing those addresses in sequence. Determine which of them will be HITS and which MISSES. Assume the cache is initially empty.

## An exercise

Consider a cache (as previously) with 4 rows and a blocksize of 8. You are given below a sequence of memory address. For each memory address, calculate its row and tag.

block = addr // 8
row = block % 4
tag = block // 4

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | |
| 106 | 1 | 3 | |
| 161 | 0 | 5 | |
| 85 | 2 | 2 | |
| 88 | 3 | 2 | |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Then, mentally simulate a cache system accessing those addresses in sequence. Determine which of them will be HITS and which MISSES. Assume the cache is initially empty.

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | |
| 106 | 1 | 3 | |
| 161 | 0 | 5 | |
| 85 | 2 | 2 | |
| 88 | 3 | 2 | |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 0 | | |
| 2 | 0 | | |
| 3 | 0 | | |

Cache is initially empty

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | |
| 161 | 0 | 5 | |
| 85 | 2 | 2 | |
| 88 | 3 | 2 | |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 0 | | |
| 2 | 0 | | |
| 3 | 1 | 2 | [block 11] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | |
| 85 | 2 | 2 | |
| 88 | 3 | 2 | |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 1 | 3 | [block 13] |
| 2 | 0 | | |
| 3 | 1 | 2 | [block 11] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | |
| 88 | 3 | 2 | |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 3 | [block 13] |
| 2 | 0 | | |
| 3 | 1 | 2 | [block 11] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 3 | [block 13] |
| 2 | 1 | 2 | [block 10] |
| 3 | 1 | 2 | [block 11] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 3 | [block 13] |
| 2 | 1 | 2 | [block 10] |
| 3 | 1 | 2 | [block 11] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | MISS |
| 159 | 3 | 4 | |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 3 | [block 13] |
| 2 | 1 | 2 | [block 10] |
| 3 | 1 | 3 EVICTION | [block 15] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | MISS |
| 159 | 3 | 4 | MISS |
| 104 | 1 | 3 | |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 3 | [block 13] |
| 2 | 1 | 2 | [block 10] |
| 3 | 1 | 4 EVICTION | [block 19] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | MISS |
| 159 | 3 | 4 | MISS |
| 104 | 1 | 3 | HIT |
| 76 | 1 | 2 | |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 3 | [block 13] |
| 2 | 1 | 2 | [block 10] |
| 3 | 1 | 4 | [block 19] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | MISS |
| 159 | 3 | 4 | MISS |
| 104 | 1 | 3 | HIT |
| 76 | 1 | 2 | MISS |
| 90 | 3 | 2 | |

Our cache

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 1 | 5 | [block 20] |
| 1 | 1 | 2 EVICTION | [block 9] |
| 2 | 1 | 2 | [block 10] |
| 3 | 1 | 4 | [block 19] |

Our sequence of acceses

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | MISS |
| 159 | 3 | 4 | MISS |
| 104 | 1 | 3 | HIT |
| 76 | 1 | 2 | MISS |
| 90 | 3 | 2 | MISS |

Our cache

| Row | V | Tag | | Value |
|---|---|---|---|---|
| 0 | 1 | 5 | | [block 20] |
| 1 | 1 | 2 | | [block 9] |
| 2 | 1 | 2 | | [block 10] |
| 3 | 1 | 2 | EVICTION | [block 8] |

## An exercise

Consider a cache (as previously) with 4 rows and a blocksize of 8. You are given below a sequence of memory address. For each memory address, calculate its row and tag.

hitratio = 2/10 = 1/5

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 106 | 1 | 3 | MISS |
| 161 | 0 | 5 | MISS |
| 85 | 2 | 2 | MISS |
| 88 | 3 | 2 | HIT |
| 124 | 3 | 3 | MISS |
| 159 | 3 | 4 | MISS |
| 104 | 1 | 3 | HIT |
| 76 | 1 | 2 | MISS |
| 90 | 3 | 2 | MISS |

Then, mentally simulate a cache system accessing those addresses in sequence. Determine which of them will be HITS and which MISSES. Assume the cache is initially empty.

## An exercise

Based on your result from the previous exercise, what results would you expect from the following sequence of addresses, given the same cache configuration? Justify your position.

```
[89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
 99, 100, 101, 102, 103, 104, 105, 106, 107]
```

## An exercise

block = addr // 8
row = block % 4
tag = block // 4

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | | | |
| 90 | | | |
| 91 | | | |
| 92 | | | |
| 93 | | | |
| 94 | | | |
| 95 | | | |
| 96 | | | |
| 97 | | | |
| 98 | | | |
| 99 | | | |
| 100 | | | |
| 101 | | | |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

## An exercise

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | |
| 90 | 3 | 2 | |
| 91 | 3 | 2 | |
| 92 | 3 | 2 | |
| 93 | 3 | 2 | |
| 94 | 3 | 2 | |
| 95 | 3 | 2 | |
| 96 | 0 | 3 | |
| 97 | 0 | 3 | |
| 98 | 0 | 3 | |
| 99 | 0 | 3 | |
| 100 | 0 | 3 | |
| 101 | 0 | 3 | |
| 102 | 0 | 3 | |
| 103 | 0 | 3 | |
| 104 | 1 | 3 | |
| 105 | 1 | 3 | |
| 106 | 1 | 3 | |
| 107 | 1 | 3 | |

## An exercise

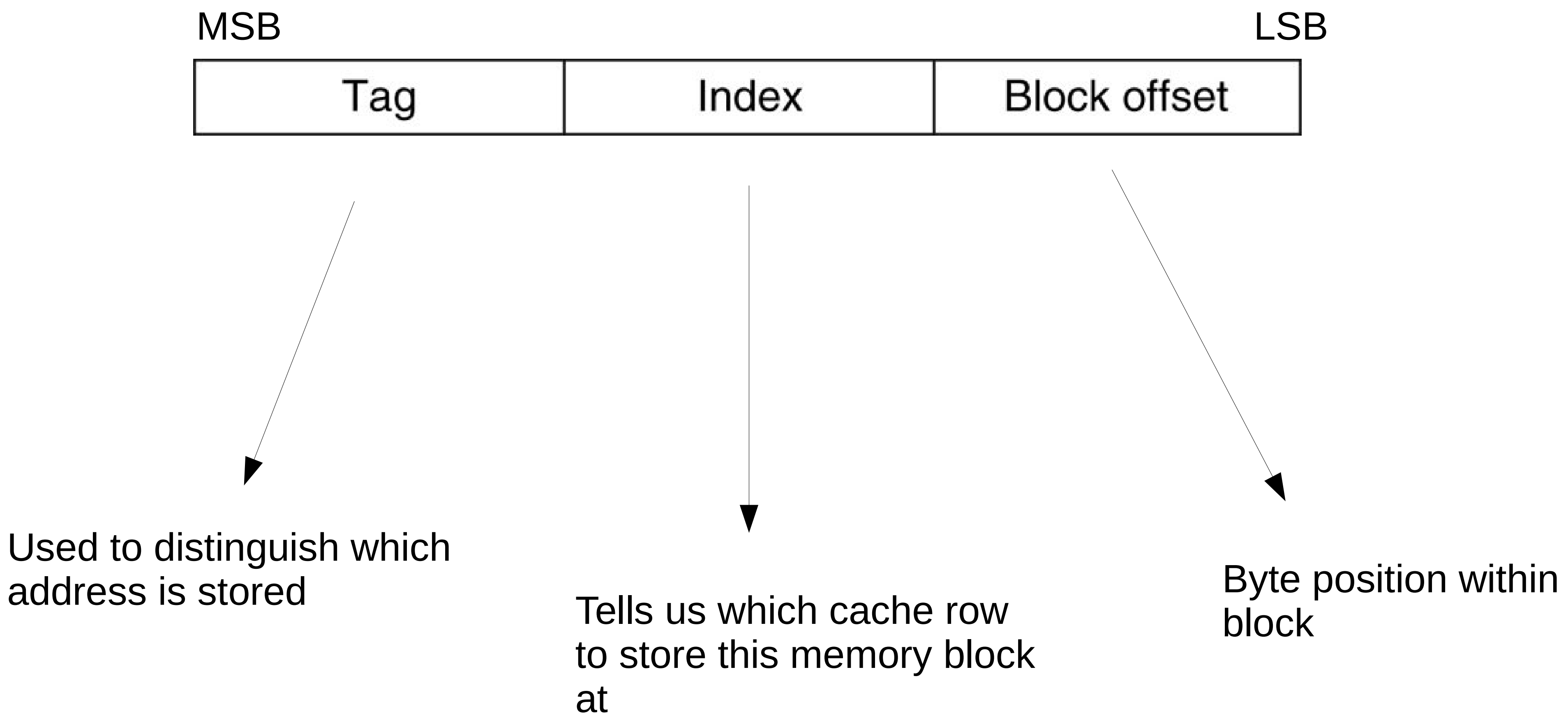
hitratio = 16/19

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 89 | 3 | 2 | MISS |
| 90 | 3 | 2 | HIT |
| 91 | 3 | 2 | HIT |
| 92 | 3 | 2 | HIT |
| 93 | 3 | 2 | HIT |
| 94 | 3 | 2 | HIT |
| 95 | 3 | 2 | HIT |
| 96 | 0 | 3 | MISS |
| 97 | 0 | 3 | HIT |
| 98 | 0 | 3 | HIT |
| 99 | 0 | 3 | HIT |
| 100 | 0 | 3 | HIT |
| 101 | 0 | 3 | HIT |
| 102 | 0 | 3 | HIT |
| 103 | 0 | 3 | HIT |
| 104 | 1 | 3 | MISS |
| 105 | 1 | 3 | HIT |
| 106 | 1 | 3 | HIT |
| 107 | 1 | 3 | HIT |

```
int main() {
    int someArray[10];
    initArray(someArray);
    int sum = 0;
    for (int i=0; i<10; i++)
        sum += someArray[i];
    return 0;
}
```

| variable | someArray[0] | someArray[1] | someArray[2] | someArray[3] | someArray[4] | someArray[5] | etc... |
|---|---|---|---|---|---|---|---|
| address | 89 | 90 | 91 | 92 | 93 | 94 | etc.. |

## Mapping a memory address to a direct-mapped cache row


MSB
LSB
Tag
Index
Block offset
Used to distinguish which address is stored
Tells us which cache row to store this memory block at
Byte position within block

Therefore: (memoryaddress / blocksize) % numberofcacheindices = cacheindex
memoryaddress % blocksize = blockoffset
memoryaddress / blocksize = blockaddress
(memoryaddress / blocksize) / numberofcacheindices = tag

## Calculating the total size (including metadata) of a direct-mapped cache:

valid bit

Number_of_rows *

  size_of_tag_in_bits
+ 1
+ block_size_in_bits

Calculating the total size (including metadata) of a direct-mapped cache:

Number_of_rows * (size_of_tag_in_bits + 1 + block_size_in_bits)

valid bit

Example: assume a processor with 16-bit addresses. The cache has 32 rows, with a blocksize of 64 cells. Each cell is one byte.

## Calculating the total size (including metadata) of a direct-mapped cache:

valid bit

Number_of_rows *
size_of_tag_in_bits
+ 1
+ block_size_in_bits

Example: assume a processor with 16-bit addresses. The cache has 32 rows, with a blocksize of 64 bytes.

Six bits of the address are for the offset (2**6 == 64). Five bits of the address select the row (2**5 == 32). The remaining 5 bits are for the tag.

The size of each block in bits is 64*8 = 512.

Thus, the size of the cache including metadata is:
32 * (5 + 1 + 512) = 16576 bits, or 2072 bytes.

## Associative caches

## A problem

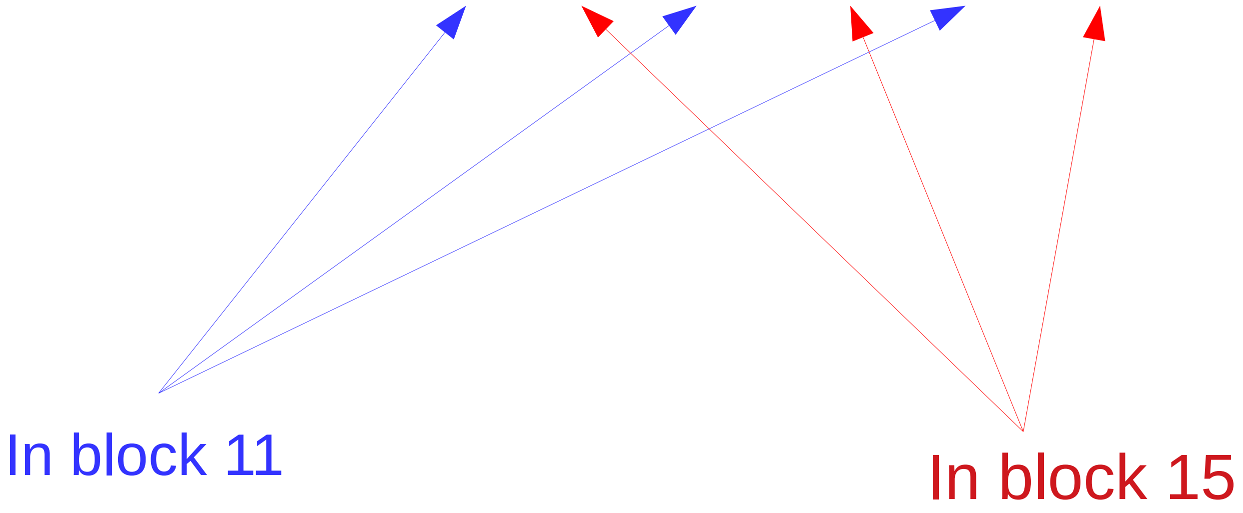
In a cache with four rows and a blocksize of 8, what would you expect of the following sequence of addresses?

```
[88, 120, 89, 121, 90, 123, 157]
```

## A problem

In a cache with four rows and a blocksize of 8, what would you expect of the following sequence of addresses?

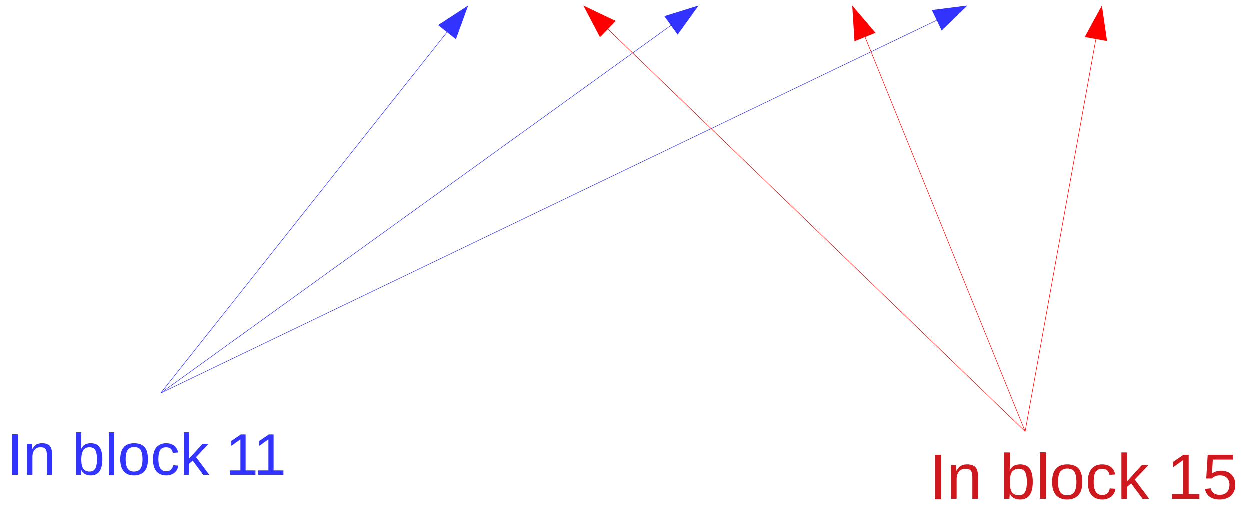
```
[88, 120, 89, 121, 90, 123, 157]
```



# A problem

In a cache with four rows and a blocksize of 8, what would you expect of the following sequence of addresses?

```
[88, 120, 89, 121, 90, 123, 157]
```

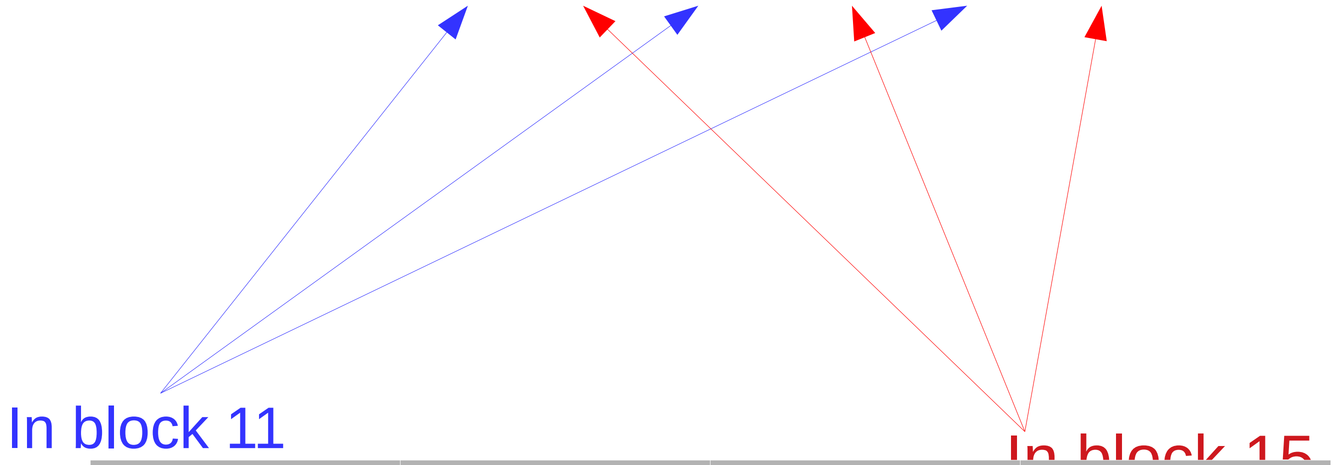


Both blocks map to row 3. So these two blocks are "fighting" over one row.

# A problem

In a cache with four rows and a blocksize of 8, what would you expect of the following sequence of addresses?

```
[88, 120, 89, 121, 90, 123, 157]
```

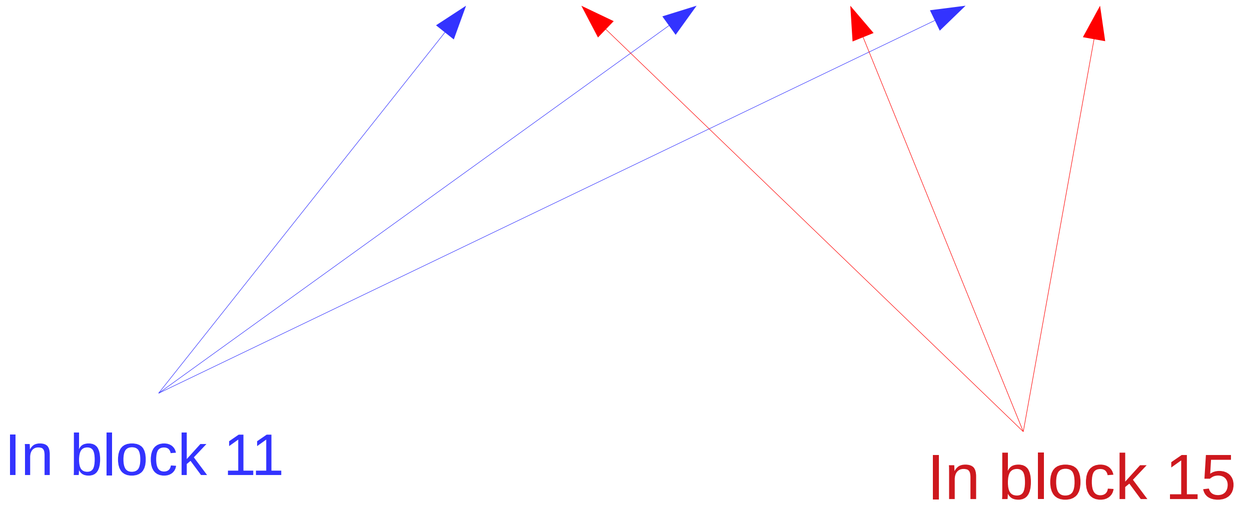


| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 88 | | | |
| 120 | | | |
| 89 | | | |
| 121 | | | |
| 90 | | | |
| 123 | | | |
| 157 | | | |

# A problem

In a cache with four rows and a blocksize of 8, what would you expect of the following sequence of addresses?

```
[88, 120, 89, 121, 90, 123, 157]
```



hitratio = 0

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 88 | 3 | 2 | MISS |
| 120 | 3 | 3 | MISS |
| 89 | 3 | 2 | MISS |
| 121 | 3 | 3 | MISS |
| 90 | 3 | 2 | MISS |
| 123 | 3 | 3 | MISS |
| 157 | 3 | 4 | MISS |

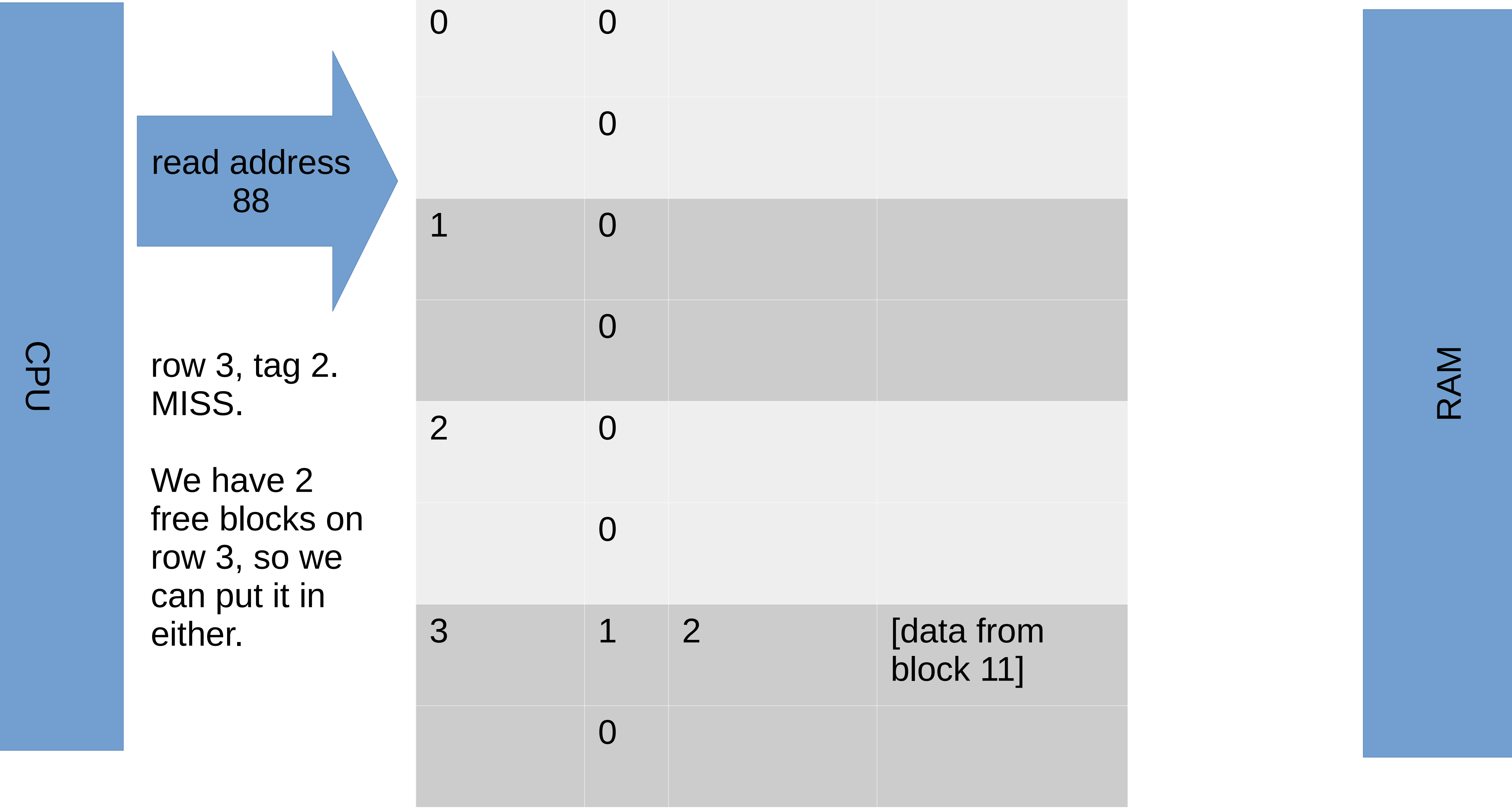
## Proposal: multiple blocks per row

CPU

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 0 | | |
| | 0 | | |

RAM

## Proposal: multiple blocks per row



row 3, tag 2.
MISS.

We have 2 free blocks on row 3, so we can put it in either.

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 0 | | |

# Proposal: multiple blocks per row

CPU

read address 120

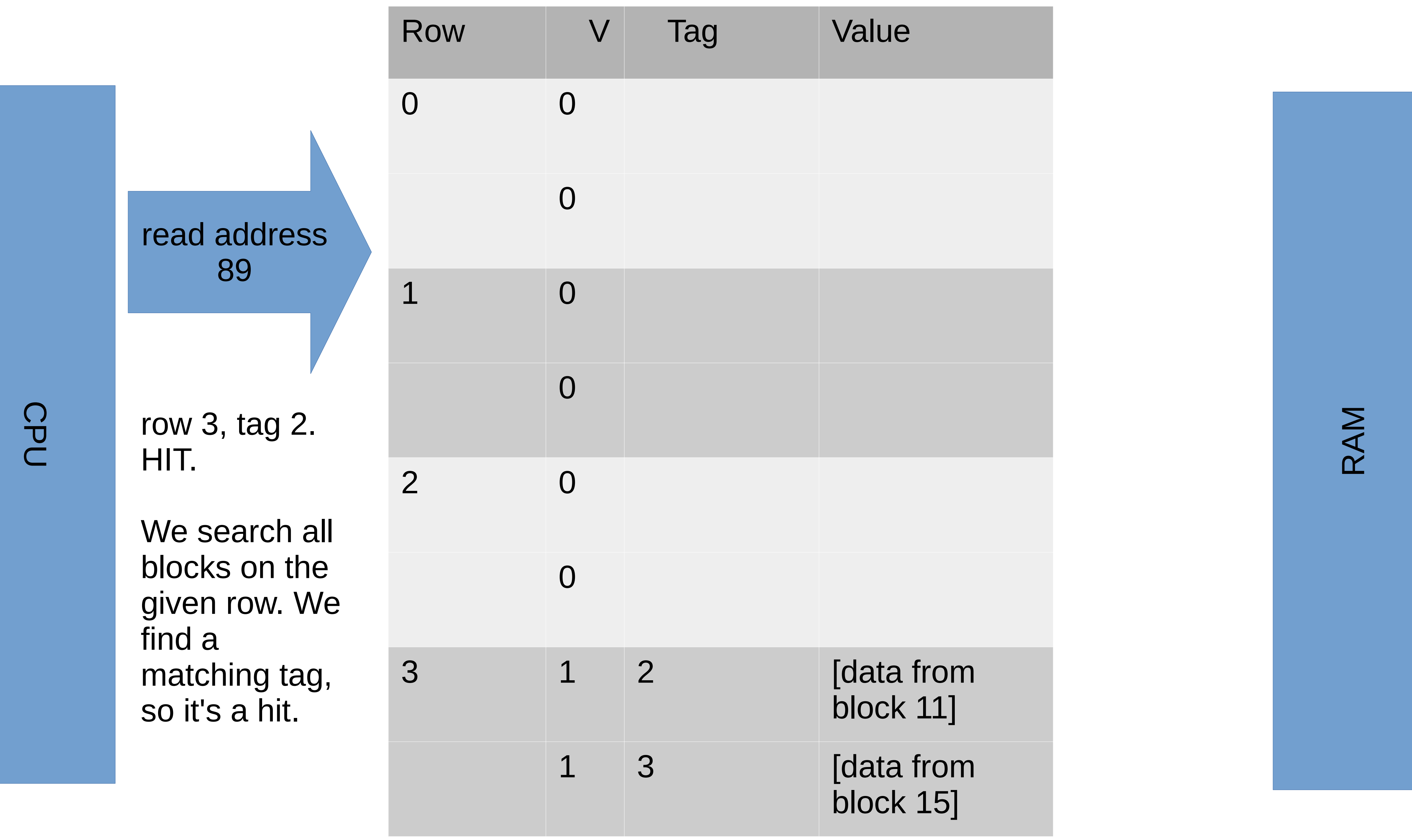
| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

row 3, tag 3.
MISS.

We still have one block available on row 3, so we use that. We do *not* evict the other block.

RAM

## Proposal: multiple blocks per row

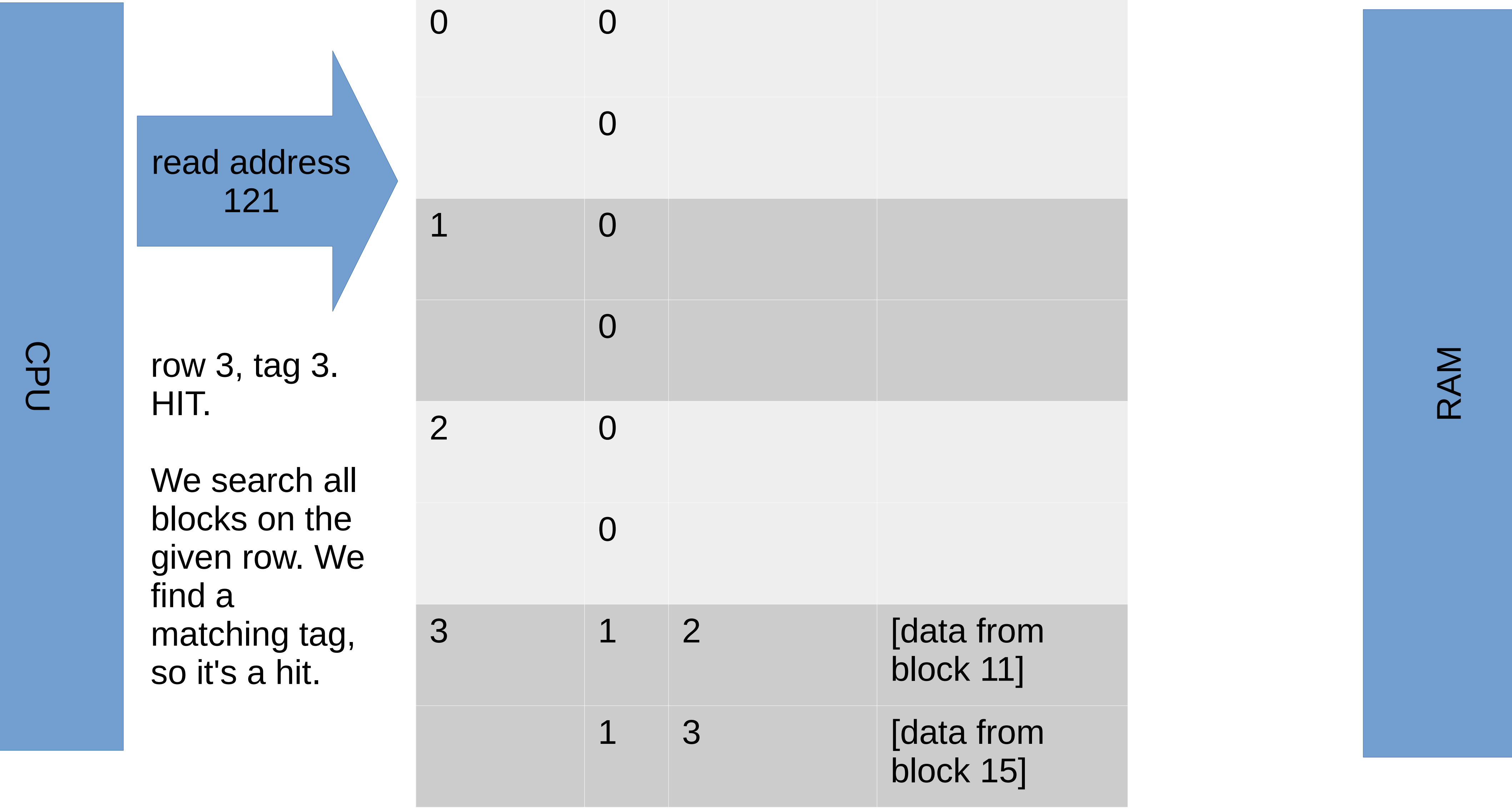


row 3, tag 2.
HIT.

We search all blocks on the given row. We find a matching tag, so it's a hit.

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

## Proposal: multiple blocks per row



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

row 3, tag 3.
HIT.

We search all blocks on the given row. We find a matching tag, so it's a hit.

## Proposal: multiple blocks per row

CPU

read address 90

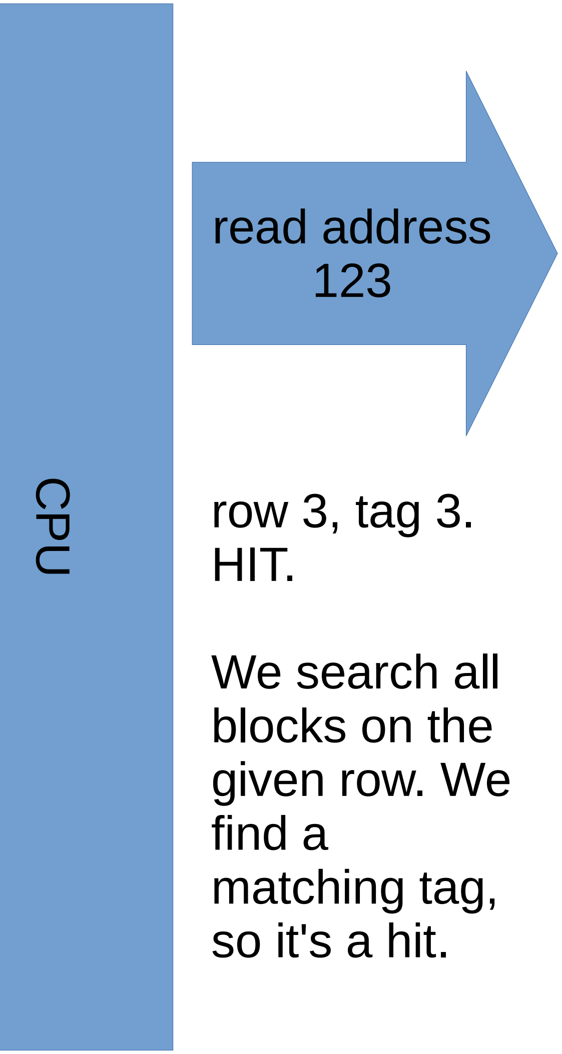
row 3, tag 2.
HIT.

We search all blocks on the given row. We find a matching tag, so it's a hit.

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

RAM

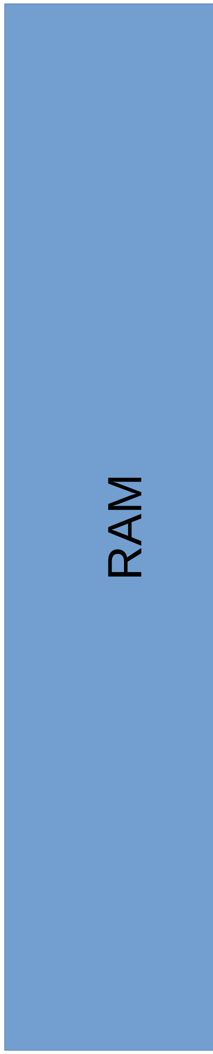
## Proposal: multiple blocks per row



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

row 3, tag 3.
HIT.

We search all blocks on the given row. We find a matching tag, so it's a hit.



## Proposal: multiple blocks per row

CPU

read address 157

row 3, tag 4. MISS.

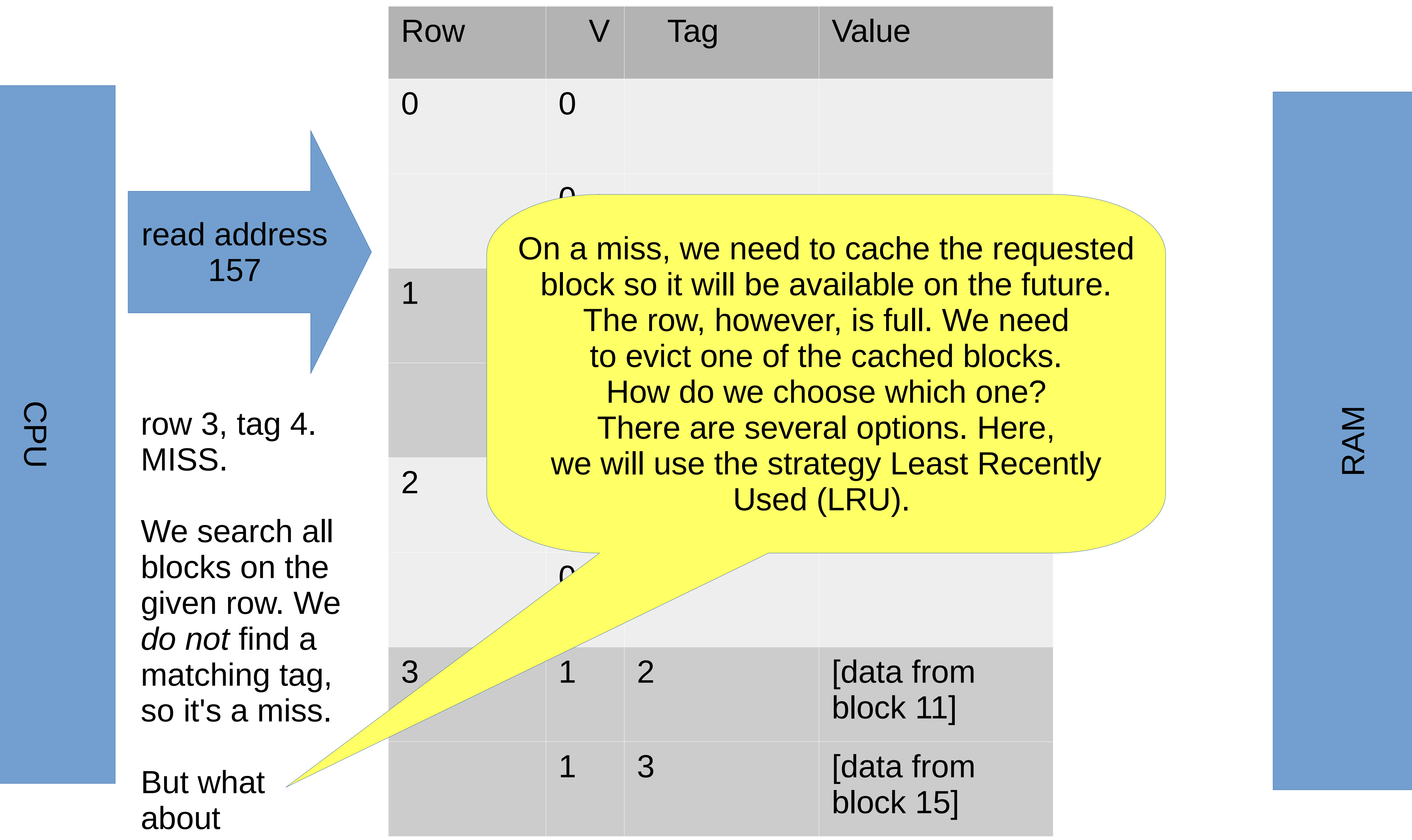
We search all blocks on the given row. We *do not* find a matching tag, so it's a miss.

But what about eviction?

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

RAM

## Proposal: multiple blocks per row



| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | | | |
| | | | |
| 2 | | | |
| | 0 | | |
| 3 | 1 | 2 | [data from block 11] |
| | 1 | 3 | [data from block 15] |

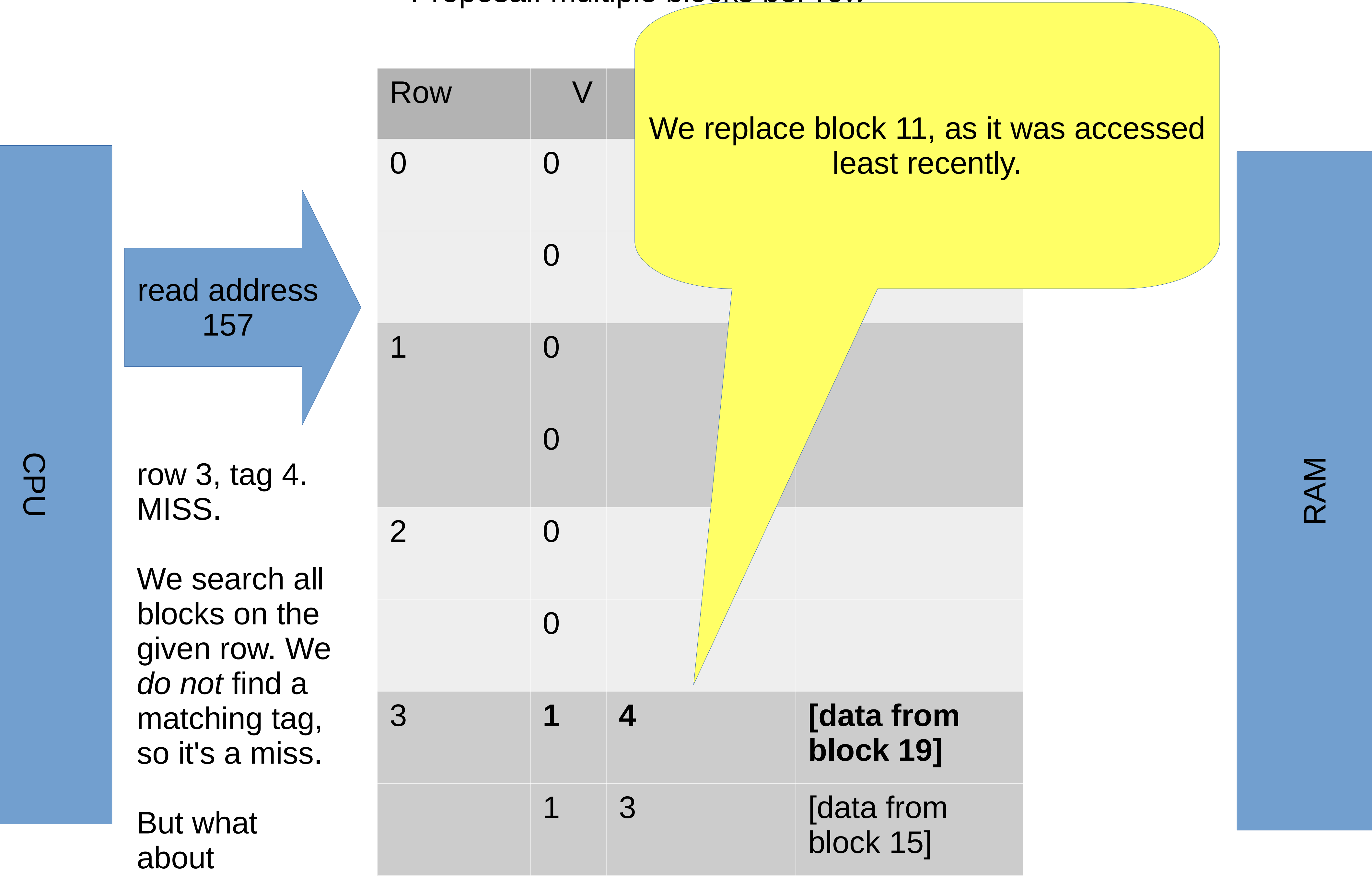
On a miss, we need to cache the requested block so it will be available on the future. The row, however, is full. We need to evict one of the cached blocks. How do we choose which one? There are several options. Here, we will use the strategy Least Recently Used (LRU).

row 3, tag 4.
MISS.

We search all blocks on the given row. We *do not* find a matching tag, so it's a miss.

But what about eviction?

## Proposal: multiple blocks per row



| Row | V | | |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| 2 | 0 | | |
| | 0 | | |
| 3 | **1** | **4** | **[data from block 19]** |
| | 1 | 3 | [data from block 15] |

row 3, tag 4. MISS.

We search all blocks on the given row. We *do not* find a matching tag, so it's a miss.

But what about eviction?

## Types of caches

- Direct-mapped cache
  - Each memory block can be cached at exactly one cache block
  - Simple and fast, but more misses
- *n*-way set-associative cache
  - Each memory block can be cached at *n* different cache blocks (*n*>1)
  - Slower, but fewer misses
  - Requires an *eviction policy* (also known as *replacement policy*)

## 4-way set associative cache, with two rows

CPU

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 0 | | |
| | 0 | | |
| | 0 | | |
| 1 | 0 | | |
| | 0 | | |
| | 0 | | |
| | 0 | | |

RAM

## Eviction policies (AKA replacement policies)

- Random replacement (RR)
  - We randomly choose one of the blocks to evict
- Least-recently used (LRU)
  - We evict the block that was accessed (read or written) least recently
  - Good strategy, but complicated to implement
- Not most-recently used (NMRU)
  - We randomly choose one of the blocks to evict, as long as it isn't the most recently used (read or written) block
- First in, first out (FIFO)
  - We evict the block that was written least recently

## An exercise

Consider a 4-way set associative cache. The cache has 4 rows. The blocksize is 2.
Cache uses LRU eviction policy.
Complete the table.

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 100 | | | |
| 125 | | | |
| 101 | | | |
| 109 | | | |
| 152 | | | |
| 140 | | | |
| 165 | | | |

What is the final state of the cache?

## An exercise

Consider a 4-way set associative cache. The cache has 4 rows. The blocksize is 2.
Cache uses LRU eviction policy.
Complete the table.

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 100 | 2 | 12 | |
| 125 | 2 | 15 | |
| 101 | 2 | 12 | |
| 109 | 2 | 13 | |
| 152 | 0 | 19 | |
| 140 | 2 | 17 | |
| 165 | 2 | 20 | |

What is the final state of the cache?

# An exercise

Consider a 4-way set associative cache. The cache has 4 rows. The blocksize is 2.
Cache uses LRU eviction policy.
Complete the table.

| Address | Row | Tag | Hit/Miss |
|---|---|---|---|
| 100 | 2 | 12 | MISS |
| 125 | 2 | 15 | MISS |
| 101 | 2 | 12 | HIT |
| 109 | 2 | 13 | MISS |
| 152 | 0 | 19 | MISS |
| 140 | 2 | 17 | MISS |
| 165 | 2 | 20 | MISS |

What is the final state of the cache?

In row 0, we have just tag 19 (including address 152).
In row 2, we have tags 20, 17, 13, 12. Tag 15 was evicted by LRU when we cached tag 20.

# Fully associative caches

Consider a 4-way associative cache with only one row. Block size is 8.

| Row | V | Tag | Value |
|---|---|---|---|
| 0 | 0 | | |
| | 1 | 3 | [data of block 3] |
| | 1 | 6 | [data of block 6] |
| | 0 | | |

# Fully associative caches

Consider a 4-way associative cache with only one row. Block size is 8.

All blocks can be placed anywhere in the cache. So there's no point contemplating the row.

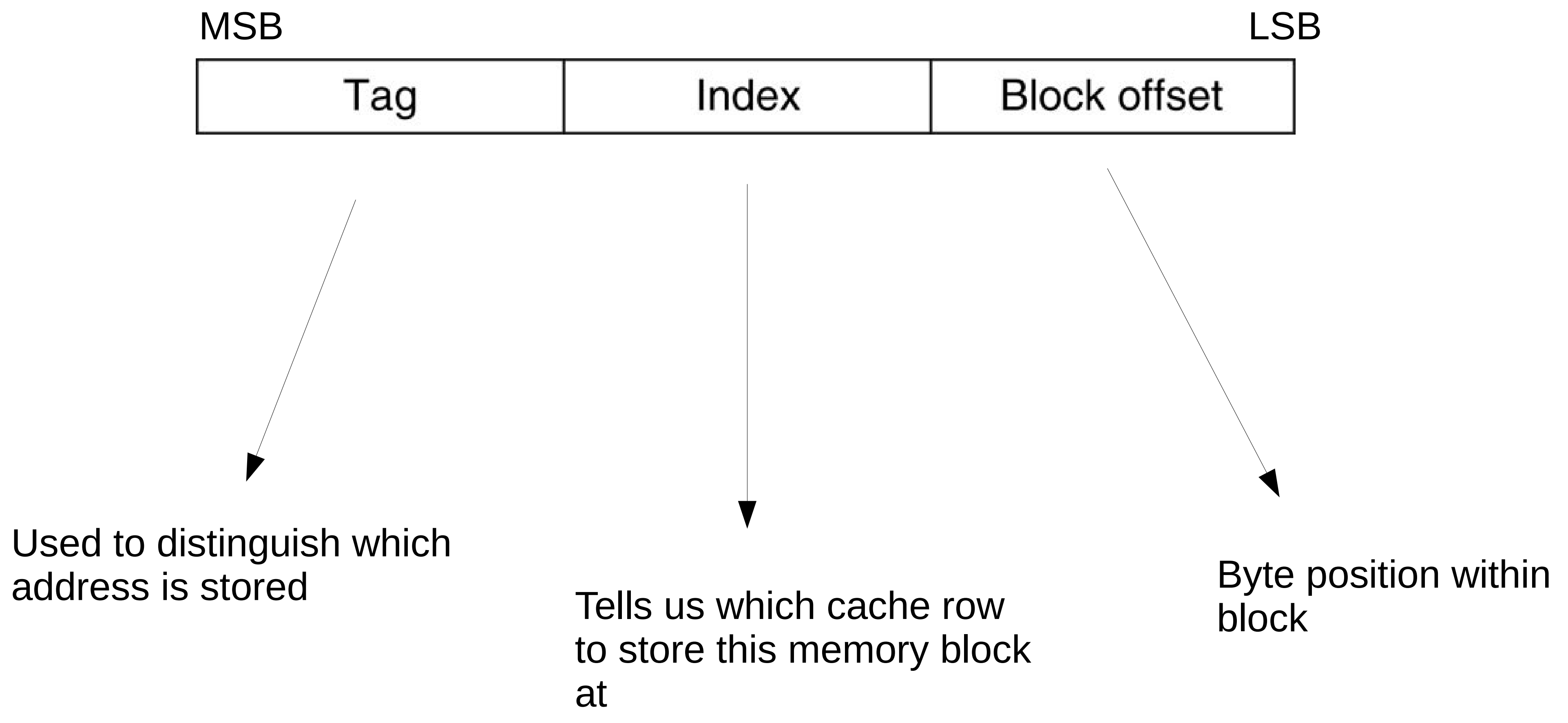
Thus, a 4-way associative cache with one row is called a *fully-associative cache*.

| ~~Row~~ | V | Tag | Value |
|---|---|---|---|
| ~~0~~ | 0 | | |
| | 1 | 3 | [data of block 3] |
| | 1 | 6 | [data of block 6] |
| | 0 | | |

We no longer calculate the row, so the address is divided only into the offset and tag. The tag is equal to the block ID.

## Mapping a memory address to a cache row

In a fully-associative cache, the number of cache rows is 1, so zero bits are given to the index.


MSB
LSB
Tag
Index
Block offset
Used to distinguish which address is stored
Tells us which cache row to store this memory block at
Byte position within block

Therefore: (memoryaddress / blocksize) % numberofcacheindices = cacheindex
memoryaddress % blocksize = blockoffset
memoryaddress / blocksize = blockaddress
(memoryaddress / blocksize) / numberofcacheindices = tag

# Fully associative caches

Consider a fully-associative cache with 4 blocks. Block size is 8.

```
[20, 90, 40, 93, 16, 20, 100, 200, 300, 400]
```

| V | Tag | Value |
|---|---|---|
| 0 | | |
| 0 | | |
| 0 | | |
| 0 | | |

# Fully associative caches

Consider a fully-associative cache with 4 blocks. Block size is 8.

```
[20, 90, 40, 93, 16, 20, 100, 200, 300, 400]
  M   M   M   H   H   H
```

| V | Tag | Value |
|---|---|---|
| 1 | 2 | [block 2] |
| 1 | 11 | [block 11] |
| 1 | 5 | [block 5] |
| 0 | | |

# Fully associative caches

Consider a fully-associative cache with 4 blocks. Block size is 8.

```
[20, 90, 40, 93, 16, 20, 100, 200, 300, 400]
```

M M M H H H M M

| V | Tag | Value |
|---|---|---|
| 1 | 2 | [block 2] |
| 1 | 11 | [block 11] |
| 1 | 25 | [block 25] |
| 1 | 12 | [block 12] |

# Fully associative caches

Consider a fully-associative cache with 4 blocks. Block size is 8.

```
[20, 90, 40, 93, 16, 20, 100, 200, 300, 400]
  M   M   M   H   H   H   M    M    M    M
```

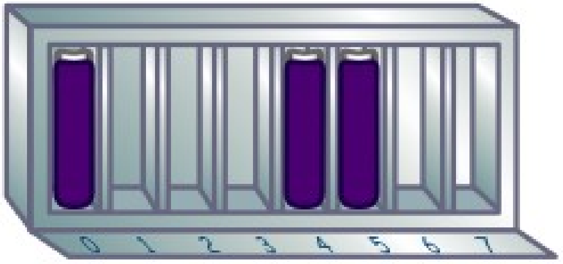
| V | Tag | Value |
|---|---|---|
| 1 | 50 | [block 50] |
| 1 | 37 | [block 37] |
| 1 | 25 | [block 25] |
| 1 | 12 | [block 12] |

## A note on terminology

| **Term** | **Meaning** |
|---|---|
| *n*-way set associative cache | Each cache row contains *n* blocks. When the cache row is full, they are evicted according to some strategy (often least-recently-used). |
| Direct-mapped cache | Each cache row contains exactly 1 block. |
| Fully associative cache | A block's placement in the cache does not depend on the block's memory address. In other words, this is an *n*-way set associative cache, where *n* is the size of the cache. |
| | |

Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.

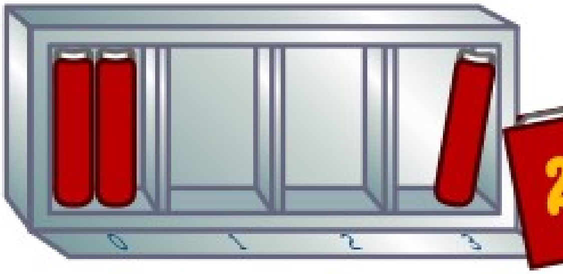
## Direct Mapped



| | | |
|---|---|---|
| Tag | Index | Offset |

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

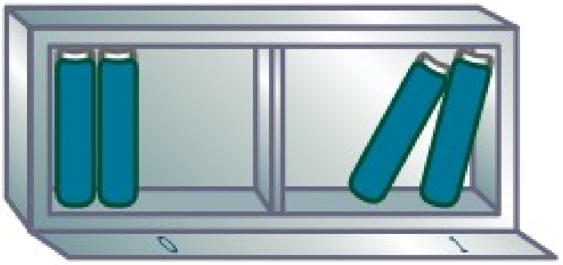
## 2-Way Set Associative



| | | |
|---|---|---|
| Tag | Index | Offset |

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

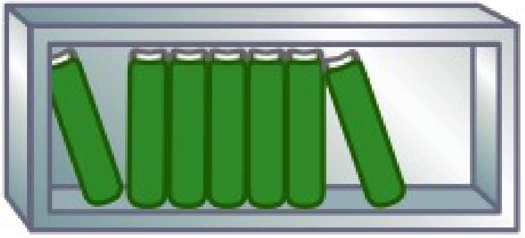
## 4-Way Set Associative



| | | |
|---|---|---|
| Tag | Index | Offset |

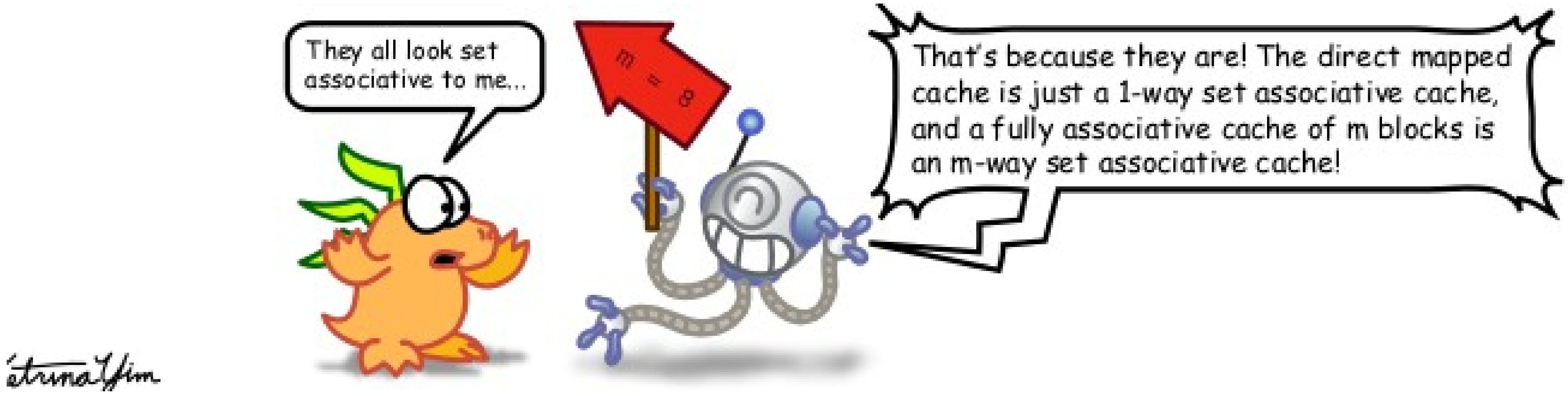
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

## Fully Associative



| | |
|---|---|
| Tag | Offset |

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!


They all look set associative to me...
m = 8
That's because they are! The direct mapped cache is just a 1-way set associative cache, and a fully associative cache of m blocks is an m-way set associative cache!

## Handling writes with cache

## Handling writes

- Goals
  - Speed
  - Coherence: we should never return to the CPU an old (*stale*) value

## Handling writes: a wrong solution

- A wrong solution
  - When executing `sw`, write changes only to cache
  - When that data is read with `lw`, the updated value will be read
  - But what if the cache block is evicted?

## Handling writes: a wrong solution

Consider a 4 row direct-mapped cache with blocksize 1. Assume that writes go to cache only.

```
movi $1, 5
movi $7, 20
lw $5, 0($7)


sw $1, 0($7)



lw $2, 0($7)



lw $3, 4($7)




lw $2, 0($7)
```

## Handling writes: a wrong solution

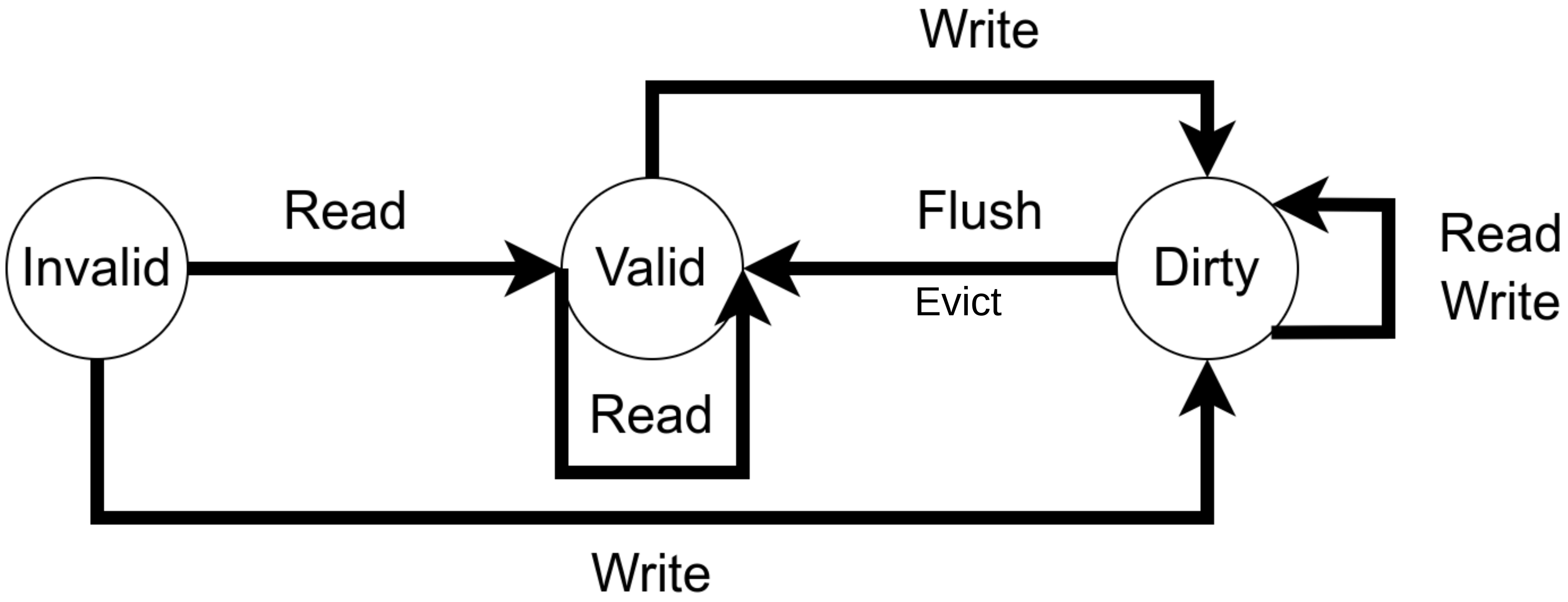
Consider a 4 row direct-mapped cache with blocksize 1. Assume that writes go to cache only.

```
movi $1, 5
movi $7, 20
lw $5, 0($7)
# write 5 to addr 20
# Value stored in cache row 0, not written to RAM
sw $1, 0($7)

# read from addr 20
# Cache HIT on row 0: ok!
lw $2, 0($7)

# Read some value from addr 24
# Cache MISS on row 0.
# Previous block in row 0 is evicted
lw $3, 4($7)

# read from addr 20
# Cache MISS on row 0.
# We read a STALE value, because our write was
# never stored to memory.
lw $2, 0($7)
```

## Handling writes

- Write-through
  - Writes will go to the cache *and* to RAM
  - Writes are slow, but logic is simpler
- Write-back
  - Writes will go only to the cache, and that block will be marked "*dirty*"
  - When a dirty block is evicted from the cache, it is written to RAM
  - Writes are faster, but implementation is complex

State machine for a row of write-back cache



## Handling writes

Consider a 4 row direct-mapped cache with blocksize 1. Assume write-back policy.

```
movi $1, 5
movi $7, 20
lw $5, 0($7)
# write 5 to addr 20
# Value stored in cache row 0, marked dirty
sw $1, 0($7)

# read from addr 20
# Cache HIT on row 0: ok!
lw $2, 0($7)

# Read some value from addr 24
# Cache MISS on row 0.
# Previous block is evicted and flushed to RAM
lw $3, 4($7)

# read from addr 20
# Cache MISS on row 0.
# Block with current value is fetched from RAM
lw $2, 0($7)
```

## Handling writes

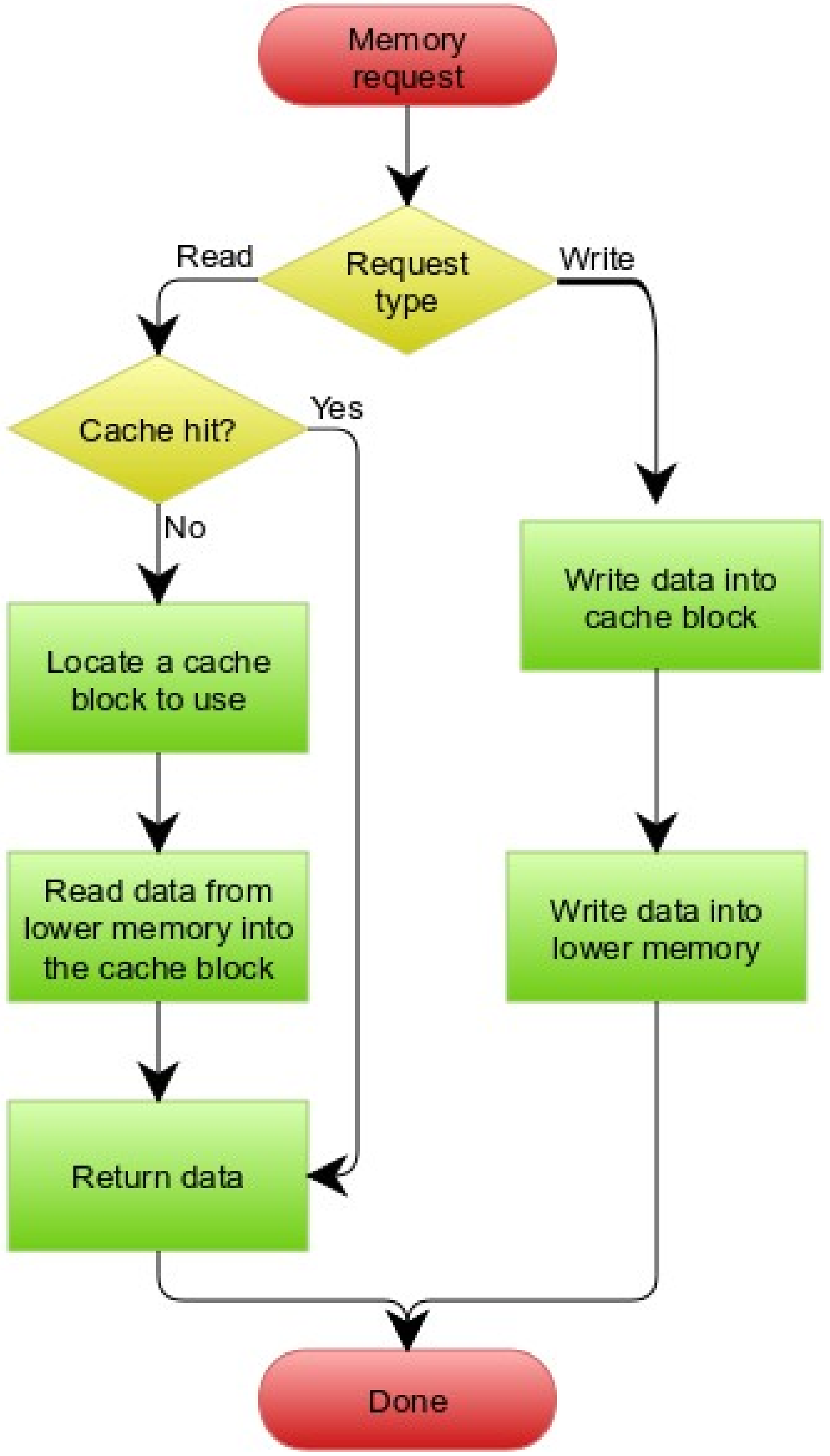
Consider a 4 row direct-mapped cache with blocksize 1. Assume write-through policy.

```
movi $1, 5
movi $7, 20
lw $5, 0($7)
# write 5 to addr 20
# Value stored in cache row 0 and in RAM
sw $1, 0($7)

# read from addr 20
# Cache HIT on row 0: ok!
lw $2, 0($7)

# Read some value from addr 24
# Cache MISS on row 0.
# Previous block is evicted
lw $3, 4($7)

# read from addr 20
# Cache MISS on row 0.
# Block with current value is fetched from RAM
lw $2, 0($7)
```

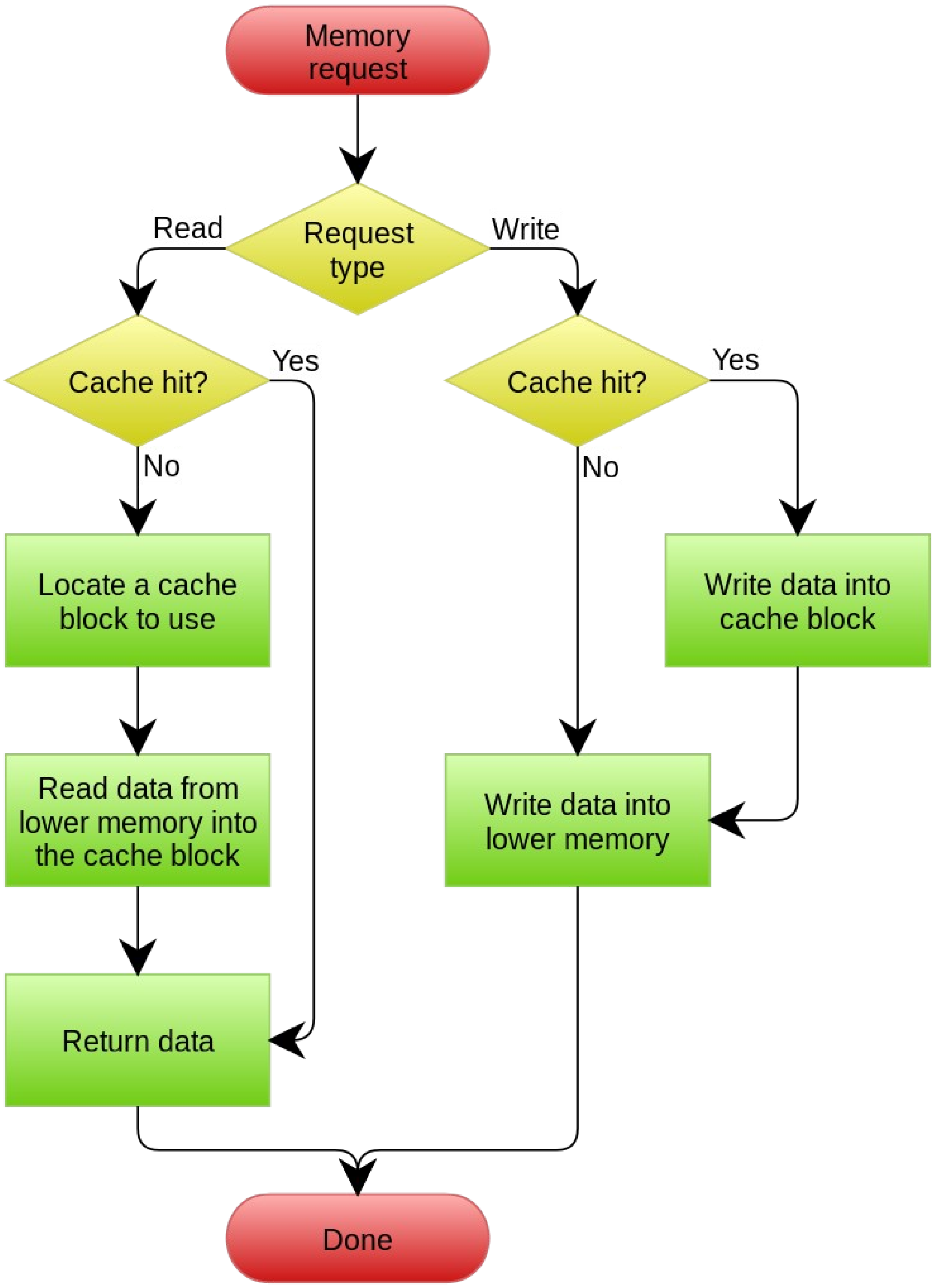
## Handling writes

- What should we do when we write to a location that is not in the cache?
  - We could write only to memory, bypassing the cache. This is *write-around*.
  - Or we could write to cache, as if it were a miss. This is *write-allocate*.
    - With **write-allocate and write-through**, we write to cache *and* memory.
    - Write **write-allocate and write-back**, we write only to cache.
- In either case, the policy for handling write misses is orthogonal to the policy for handling write hits.

Write-through for write hits, and write-allocate for write misses.

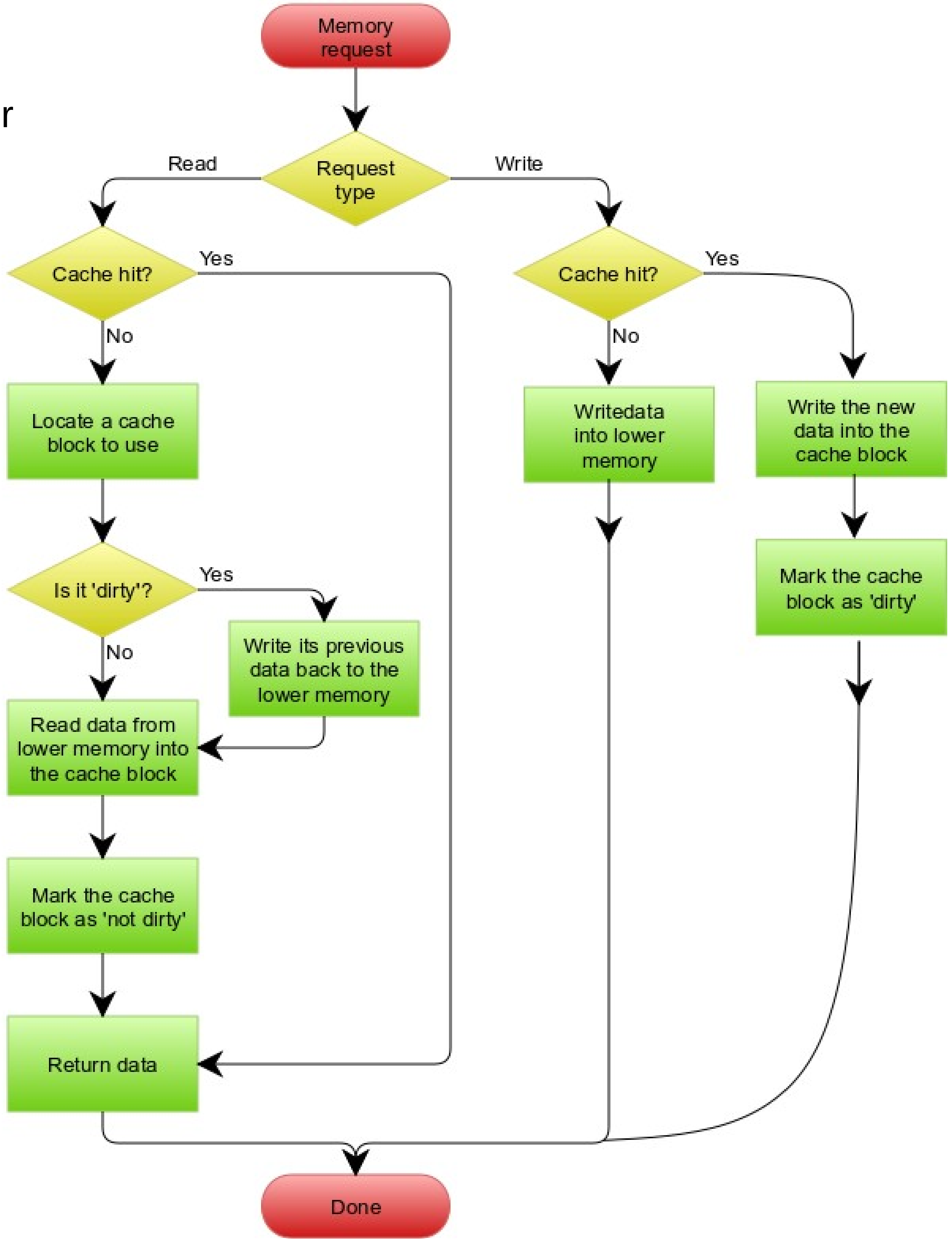

Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Read data from lower memory into the cache block
Return data
Write data into cache block
Write data into lower memory
Done

Write-through for write hits, and write-around for write misses.

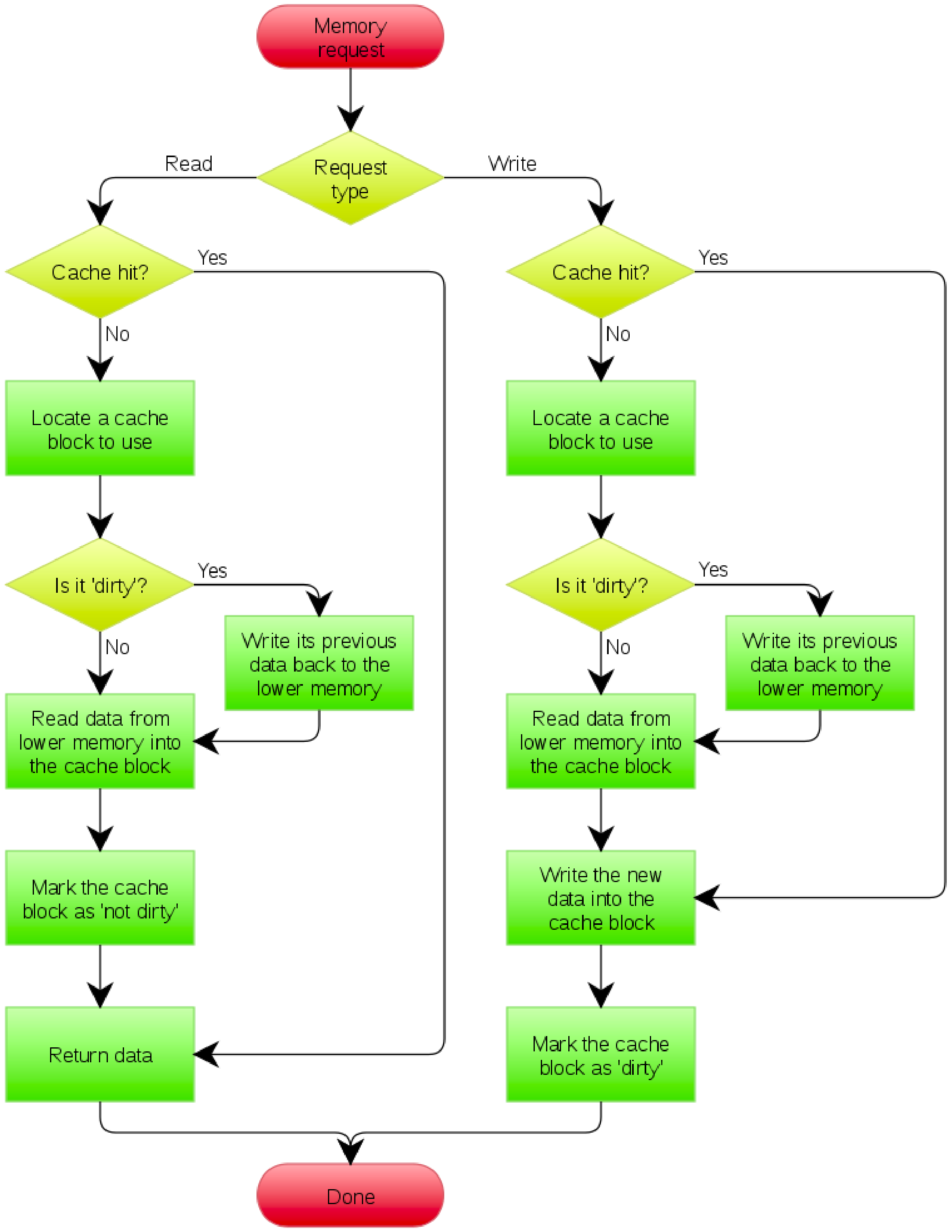


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Write data into cache block
Read data from lower memory into the cache block
Write data into lower memory
Return data
Done

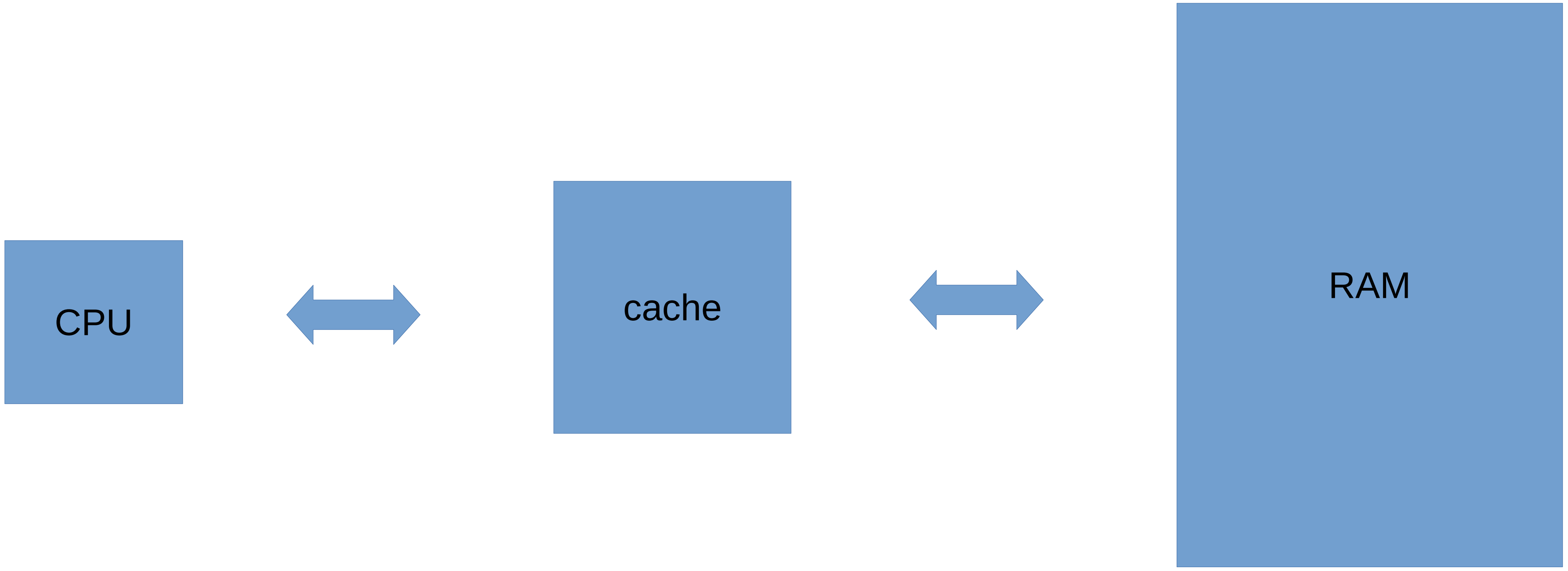
Write-back for write hits, and write-around for write misses.

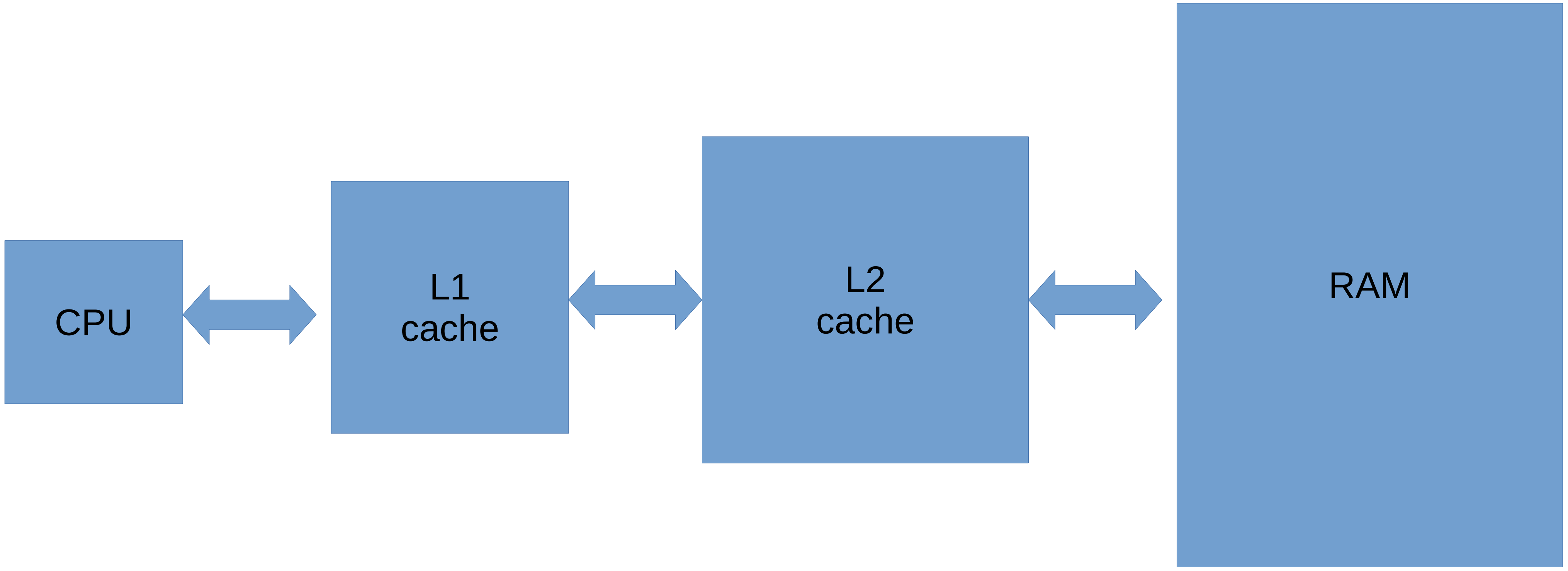

Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Writedata into lower memory
Write the new data into the cache block
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Mark the cache block as 'dirty'
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Done

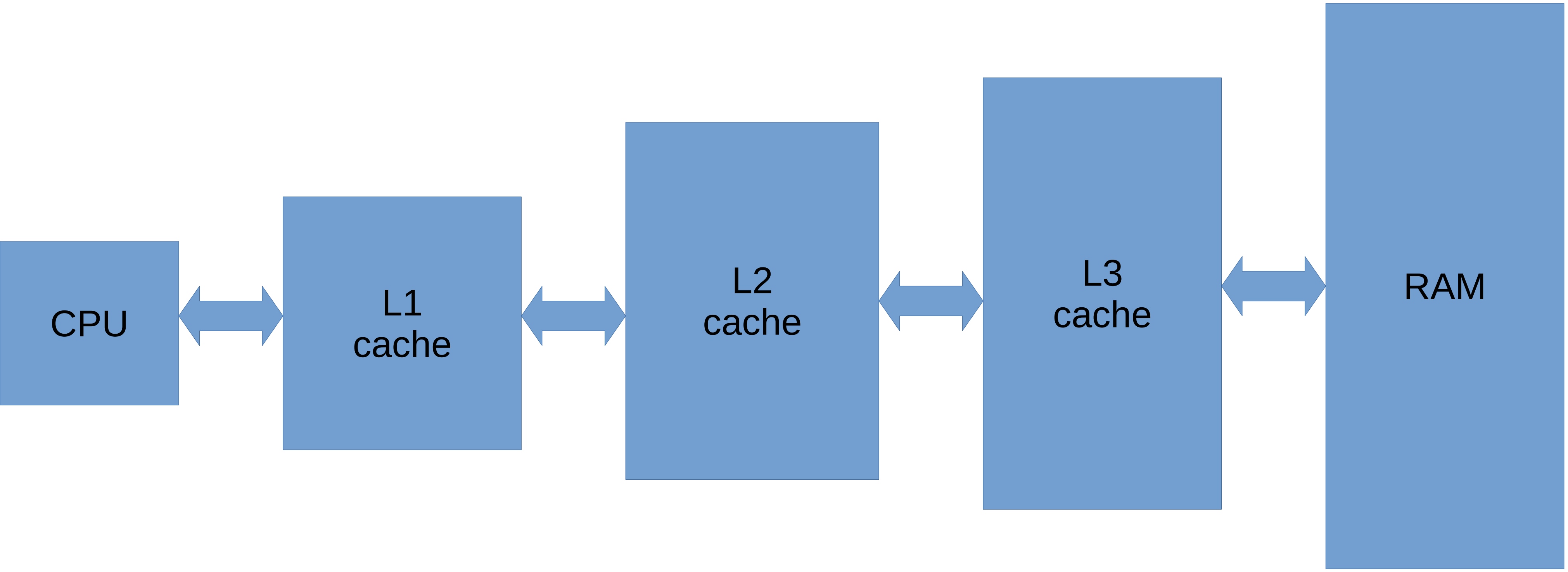
Write-back for write hits, and write-allocate for write misses.

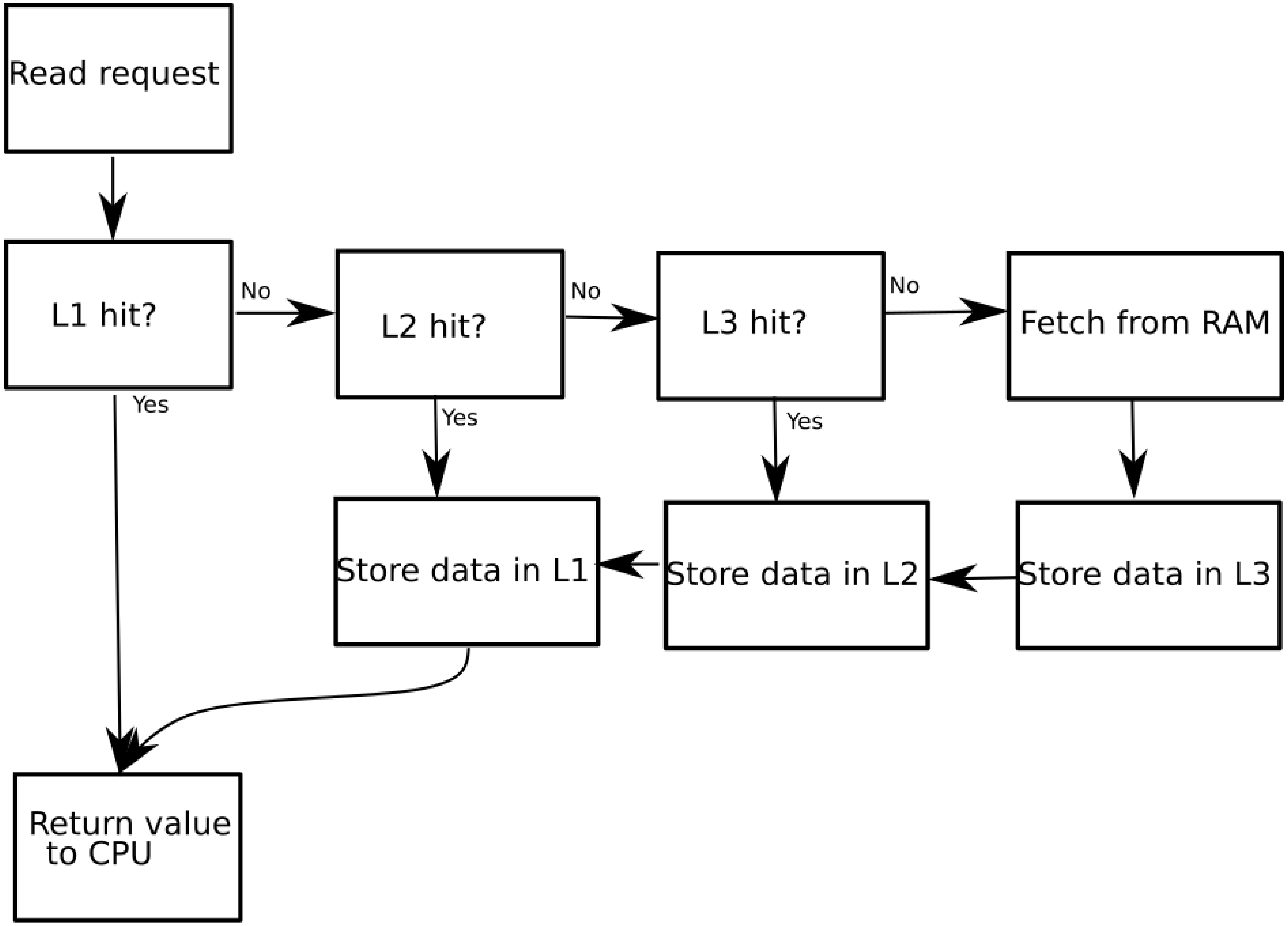

Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as 'dirty'
Done

## Cache hierarchy

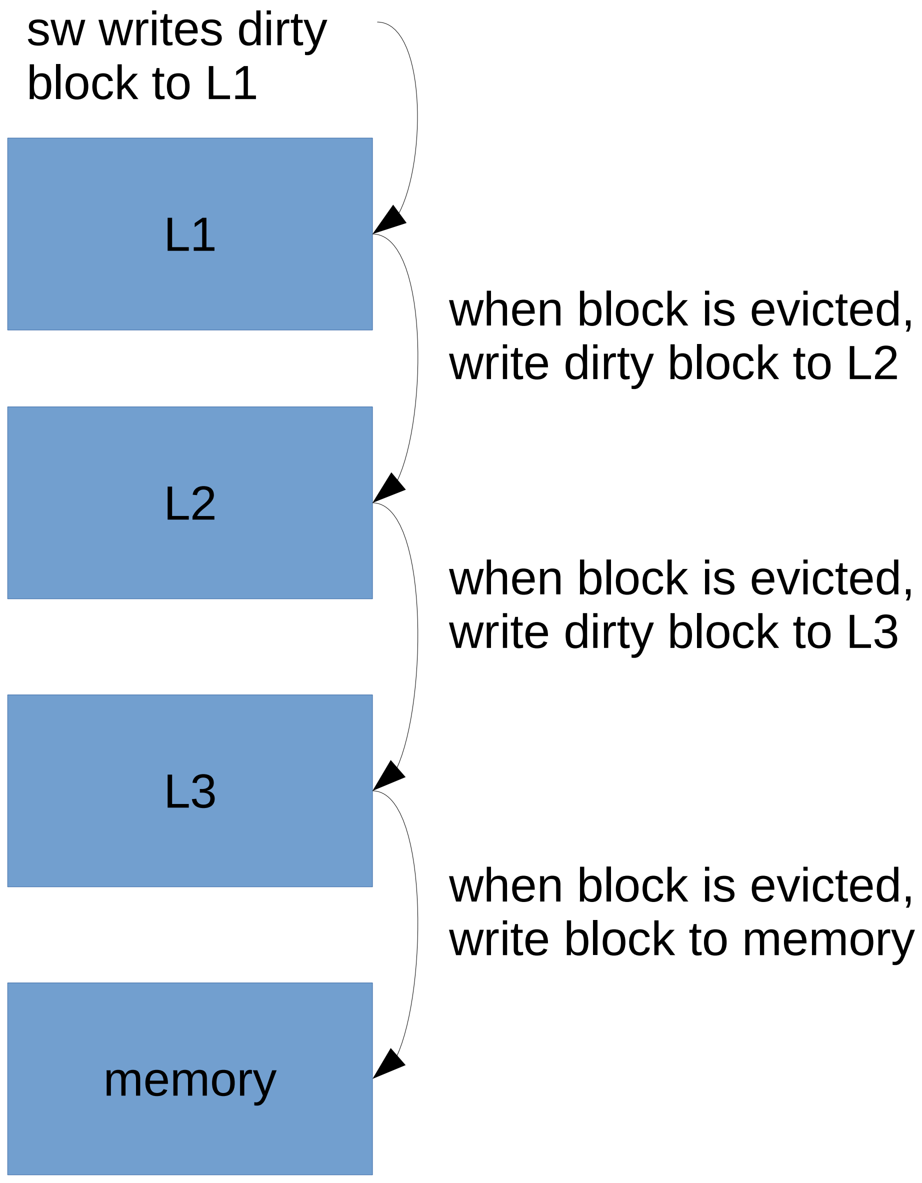

CPU
cache
RAM


CPU
L1 cache
L2 cache
RAM

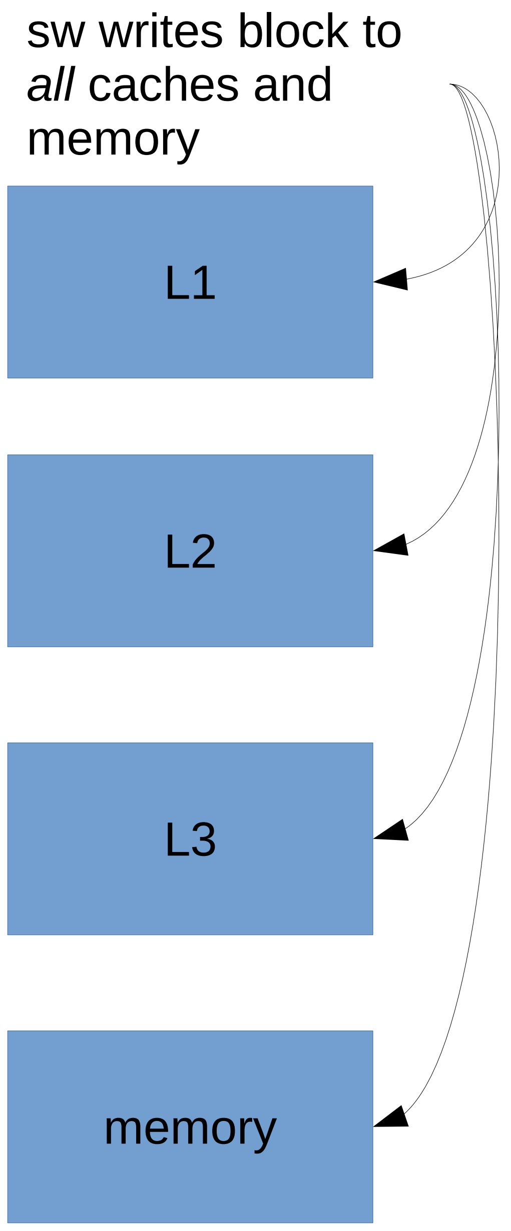

CPU
L1 cache
L2 cache
L3 cache
RAM


Read request
L1 hit?
No
L2 hit?
No
L3 hit?
No
Fetch from RAM
Yes
Yes
Yes
Store data in L1
Store data in L2
Store data in L3
Return value to CPU

**Write-back** in case of multiple caches.


sw writes dirty block to L1
L1
when block is evicted, write dirty block to L2
L2
when block is evicted, write dirty block to L3
L3
when block is evicted, write block to memory
memory

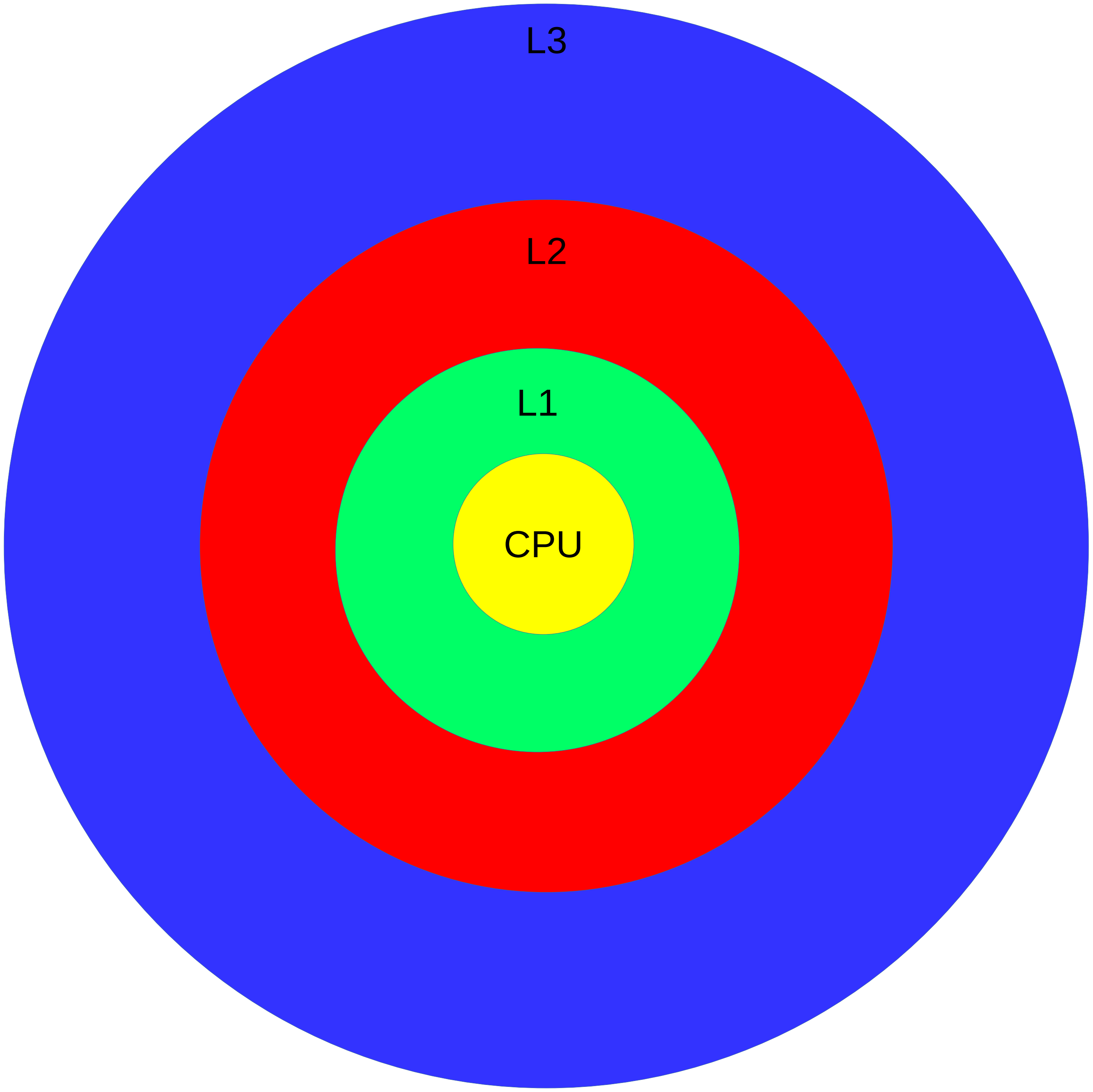
**Write-through** in case of multiple caches.



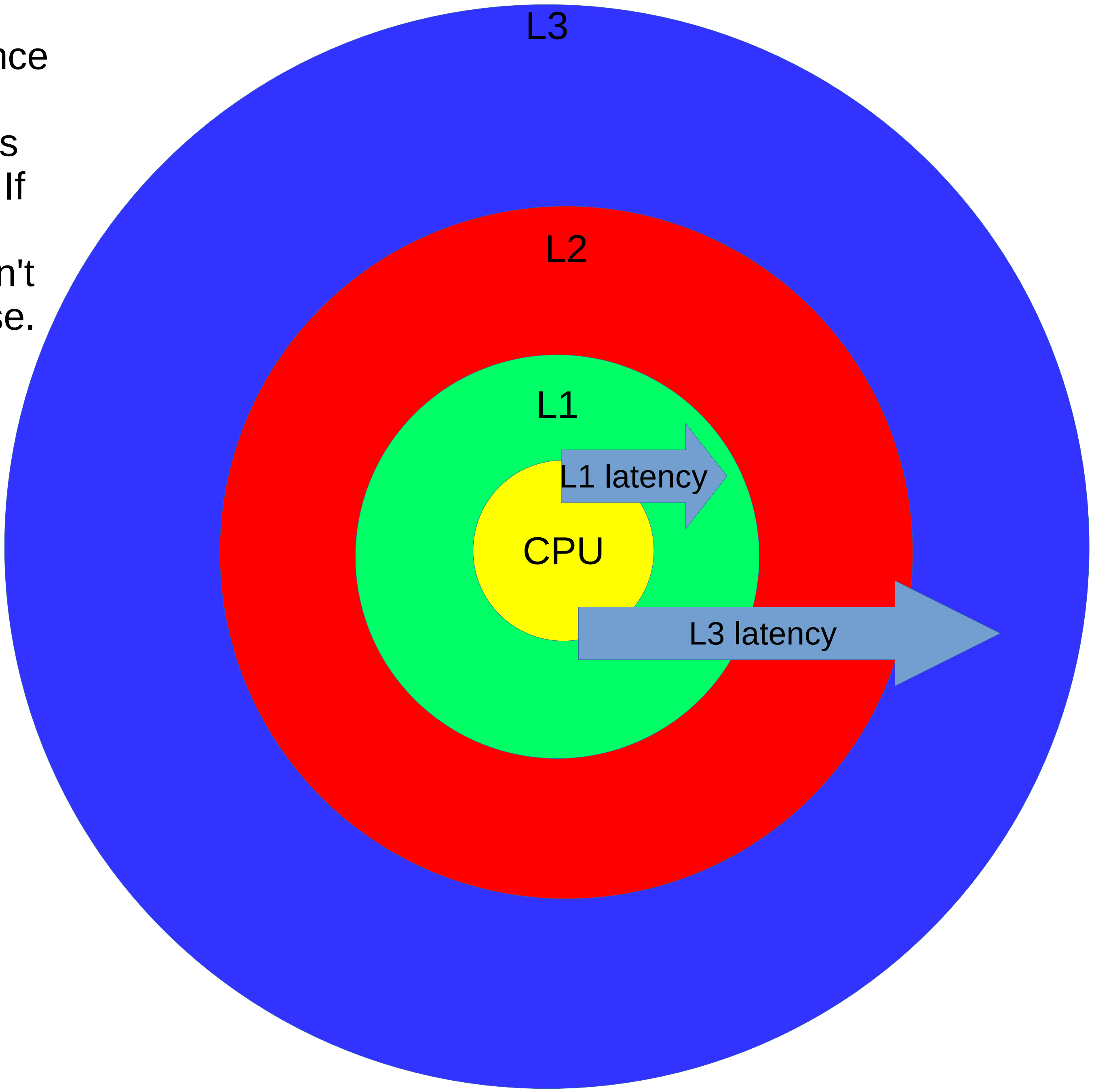
## Why not make it bigger?

- Students ask:
  - Since L1 cache is faster, why not make make it bigger?
- Price is a limiting factor. Faster caches need more read/write ports, which increases routing concerns and manufacturing cost. Less frequently accessed caches don't.
- On x86, L1 is limited in size by the page size of main memory (times the number of ways) in order to limit reliance on the TLB.
- There's a trade-off between speed and size. You can adjust your priorities, but you can't arbitrarily increase both.

https://stackoverflow.com/questions/4666728/why-is-the-size-of-l1-cache-smaller-than-that-of-the-l2-cache-in-most-of-the-pro


L3
L2
L1
CPU

A limiting factor in speedy access to cache is the physical distance to the CPU. Smaller caches can be closer. If we made L1 larger, it couldn't (all) be as close.

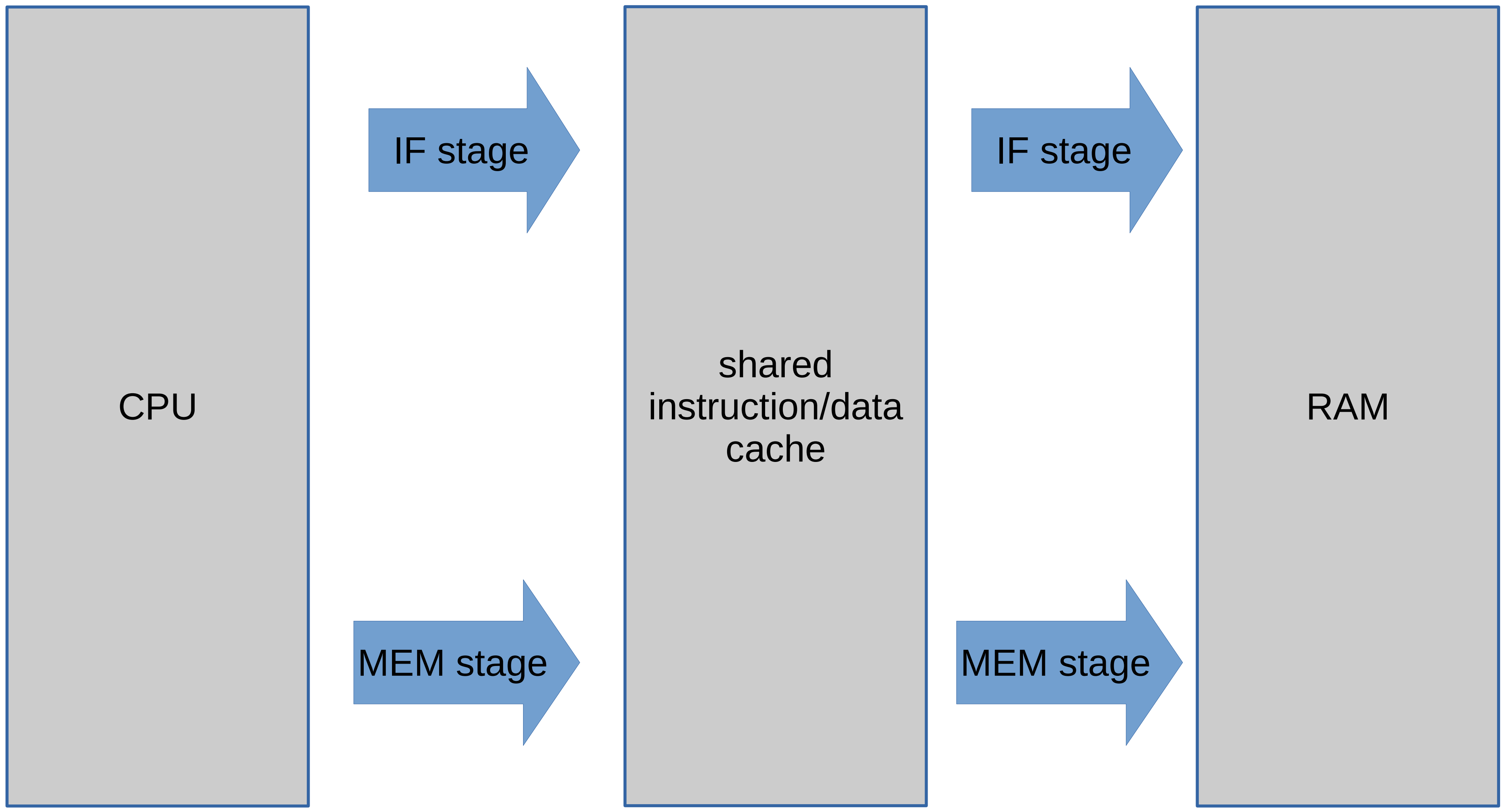


## Instruction cache

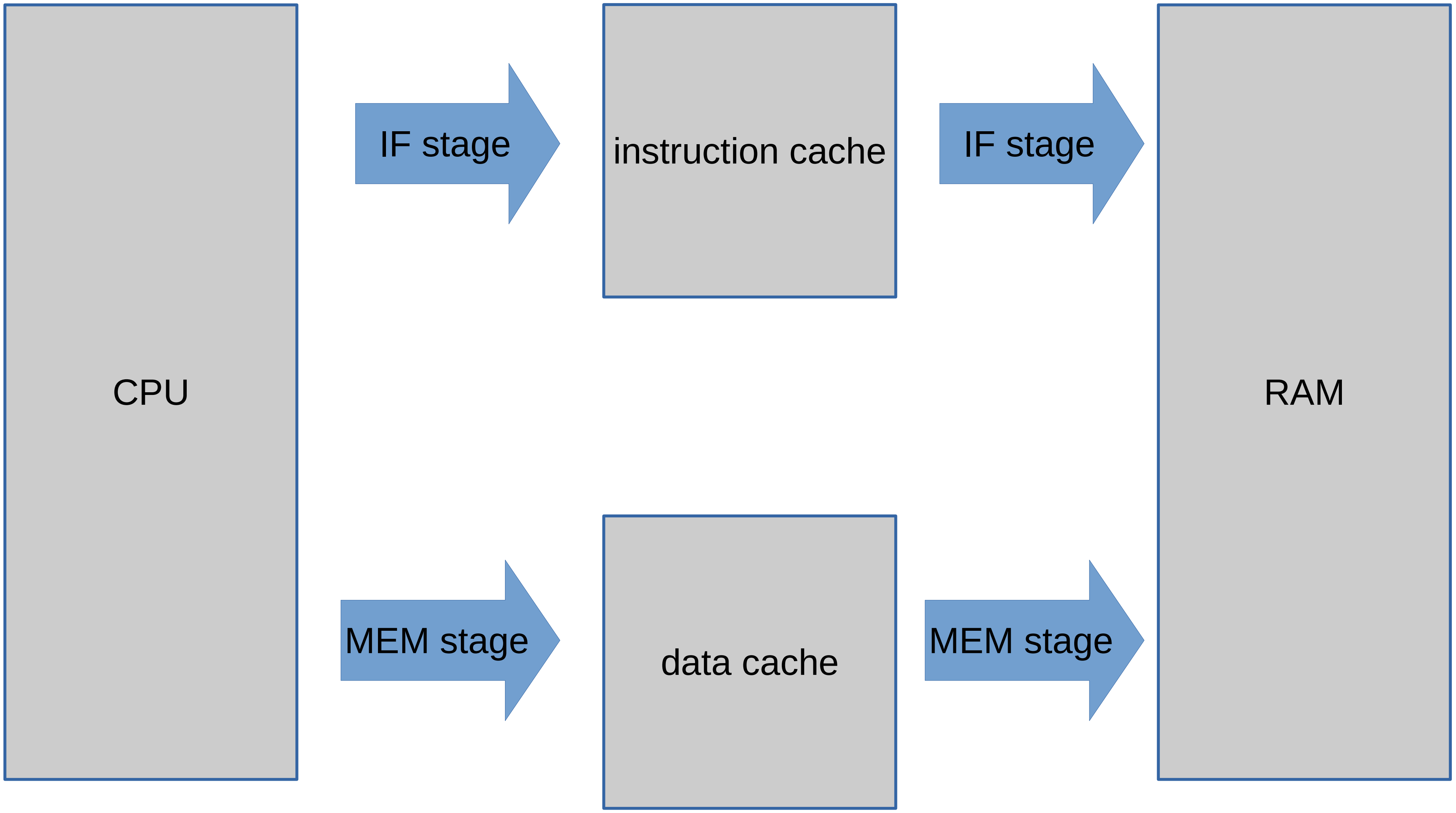
## Instruction cache

- **lw** and **sw** are not the only cases when memory is accessed
- *Every instruction* must be read from memory in the fetch stage
- Because instructions are accessed frequently, caching leads to significant performance improvement

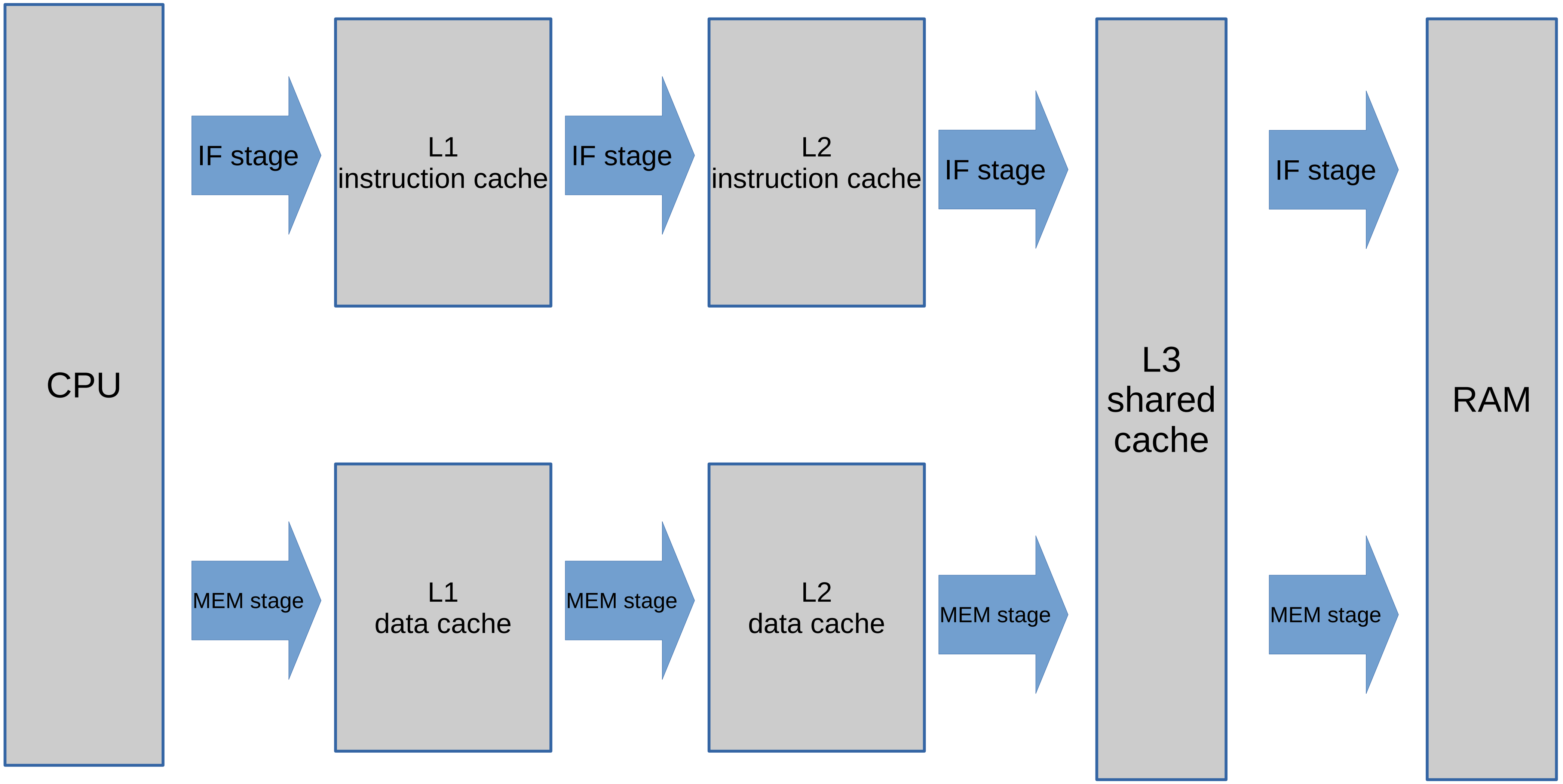
## Shared instruction/data cache



## Discrete instruction/data caches



# Hierarchical discrete instruction/data caches

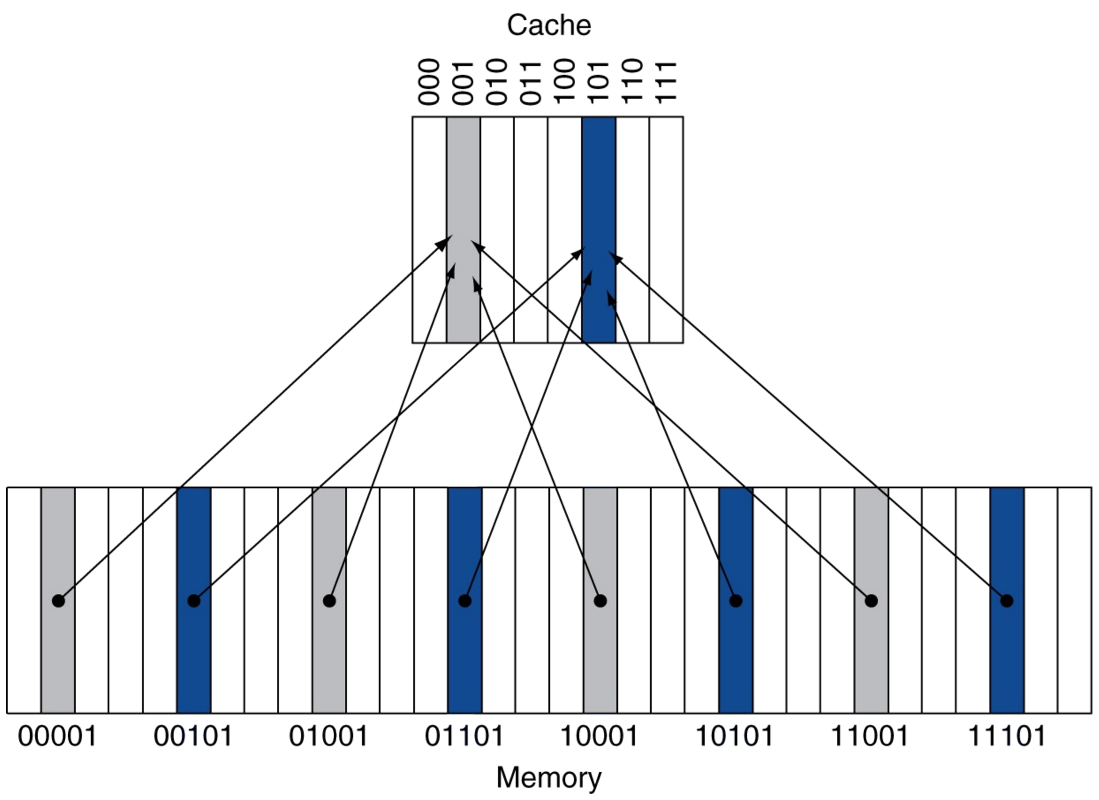


# Instruction cache

- Shared or discrete instruction cache?
- Arguments for discrete:
  - IF and MEM will rarely access the same address
- Arguments for shared:
  -

# Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

## An example

Address size: 32 bits
Block size: 256 bytes
Cache size: 1024 blocks

Therefore:

8 bits from the address to store the block offset
10 bits for cache index
The rest (14 bits) is the tag

The overhead per block is then 14+1=15 bits (for valid bit)
Total cache size in bits is 1024*(256*8 + 15)

# Cache Example

- 8 blocks, 1 byte per block, direct mapped
- Main memory address is 5 bits
- Initial state:

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | N | | |
| 111 | N | | |

# Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| **110** | **Y** | **10** | **Mem[10110]** |
| 111 | N | | |

# Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 26 | 11 010 | Miss | 010 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| **010** | **Y** | **11** | **Mem[11010]** |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Hit | 110 |
| 26 | 11 010 | Hit | 010 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 16 | 10 000 | Miss | 000 |
| 3 | 00 011 | Miss | 011 |
| 16 | 10 000 | Hit | 000 |

| Index | V | Tag | Data |
|---|---|---|---|
| **000** | **Y** | **10** | **Mem[10000]** |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| **011** | **Y** | **00** | **Mem[00011]** |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 18 | 10 010 | Miss | 010 |
| 26 | 11 010 | Miss | 010 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | Y | 10 | Mem[10000] |
| 001 | N | | |
| **010** | **Y** | **10** | **Mem[10010]** |
| 011 | Y | 00 | Mem[00011] |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

## What cache do I have on my computer?

```
$ lscpu
... (omitted for brevity)
L1d cache:                  32K
L1i cache:                  32K
L2 cache:                   256K
L3 cache:                   3072K
...
```

## What cache do I have on my computer?

```
$ sudo lshw -C memory
...
  *-cache:0
       description: L1 cache
       physical id: 7
       slot: L1 Cache
       size: 256KiB
       capacity: 256KiB
       capabilities: synchronous internal write-back unified
       configuration: level=1
  *-cache:1
       description: L2 cache
       physical id: 8
       slot: L2 Cache
       size: 1MiB
       capacity: 1MiB
       capabilities: synchronous internal write-back unified
       configuration: level=2
  *-cache:2
       description: L3 cache
       physical id: 9
       slot: L3 Cache
       size: 8MiB
       capacity: 8MiB
       capabilities: synchronous internal write-back unified
       configuration: level=3
```

## What cache do I have on my computer?

```
$ sudo dmidecode -t cache
# dmidecode 3.0
Getting SMBIOS data from sysfs.
SMBIOS 2.8 present.

Handle 0x0003, DMI type 7, 19 bytes
Cache Information
        Socket Designation: L1 Cache
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 64 kB
        Maximum Size: 64 kB
        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Parity
        System Type: Data
        Associativity: 8-way Set-associative
..
(etc etc etc)
```

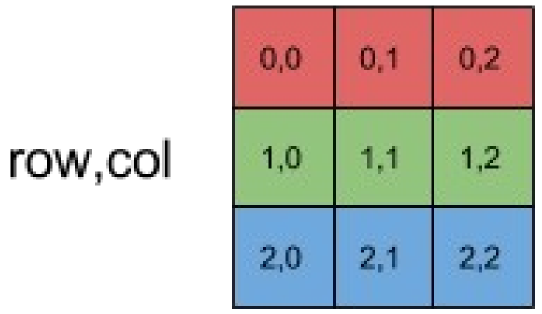
## What cache do I have on my computer?

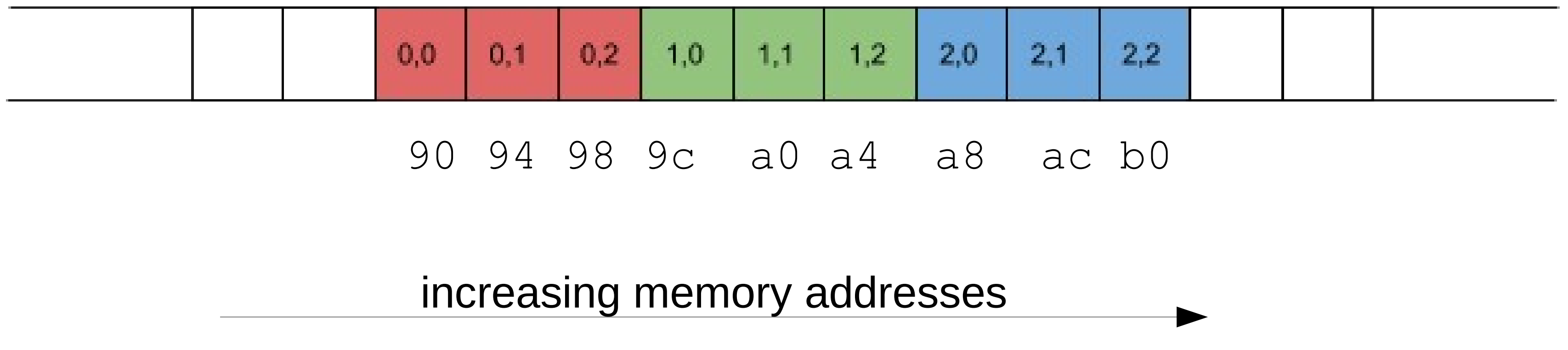
```
$ cat /proc/cpuinfo | grep cache
cache size   : 3072 KB
cache_alignment  : 64
cache size   : 3072 KB
cache_alignment  : 64
cache size   : 3072 KB
cache_alignment  : 64
cache size   : 3072 KB
cache_alignment  : 64
```

Cache block size (in bytes) for each of my CPU's four virtual cores

Multidimensional array layout

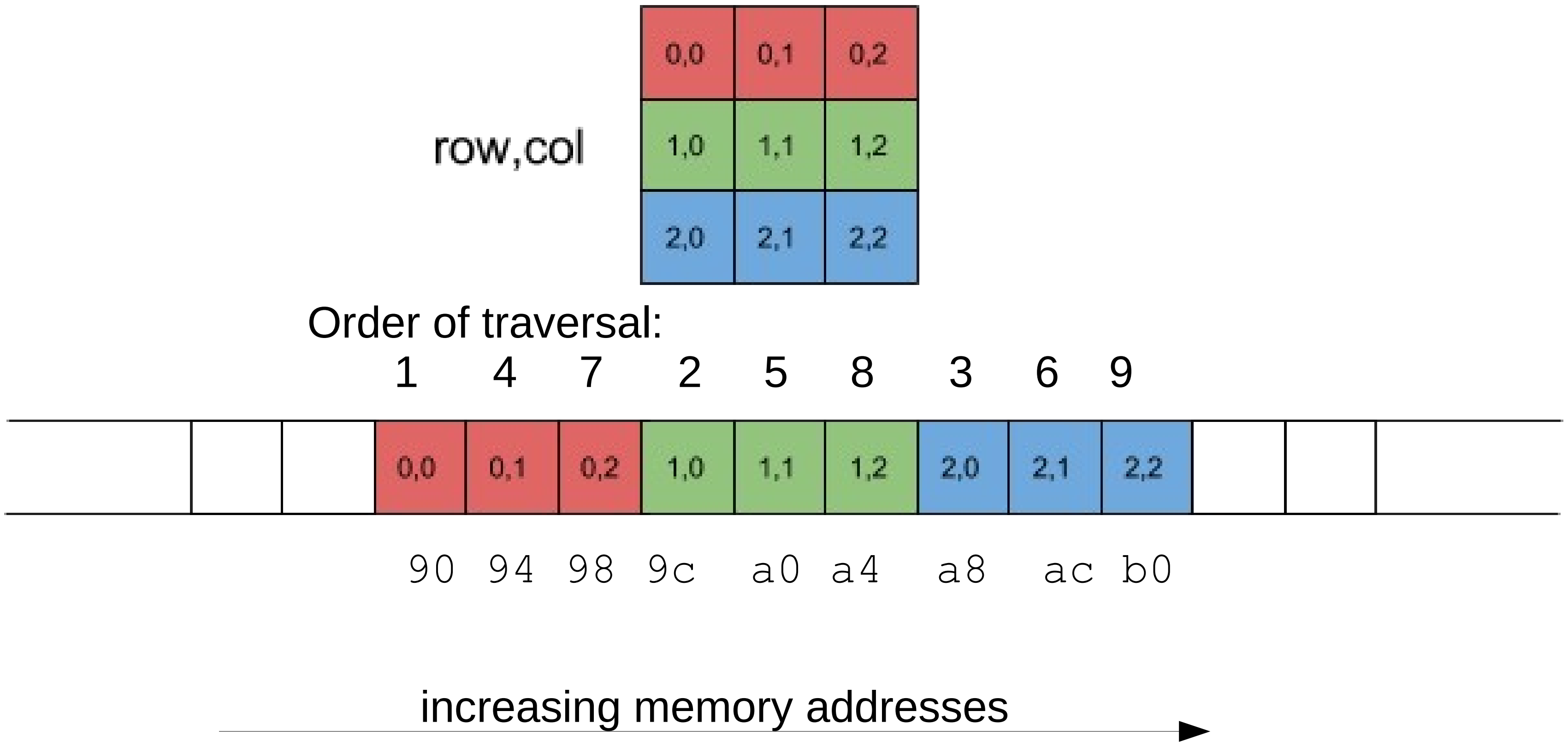
```
int main() {
    int my_array[3][3];
    /*
    There are nine elements in the array:
        my_array[0][0]
        my_array[0][1]
        my_array[0][2]
        my_array[1][0]
        etc.
    */
}
```





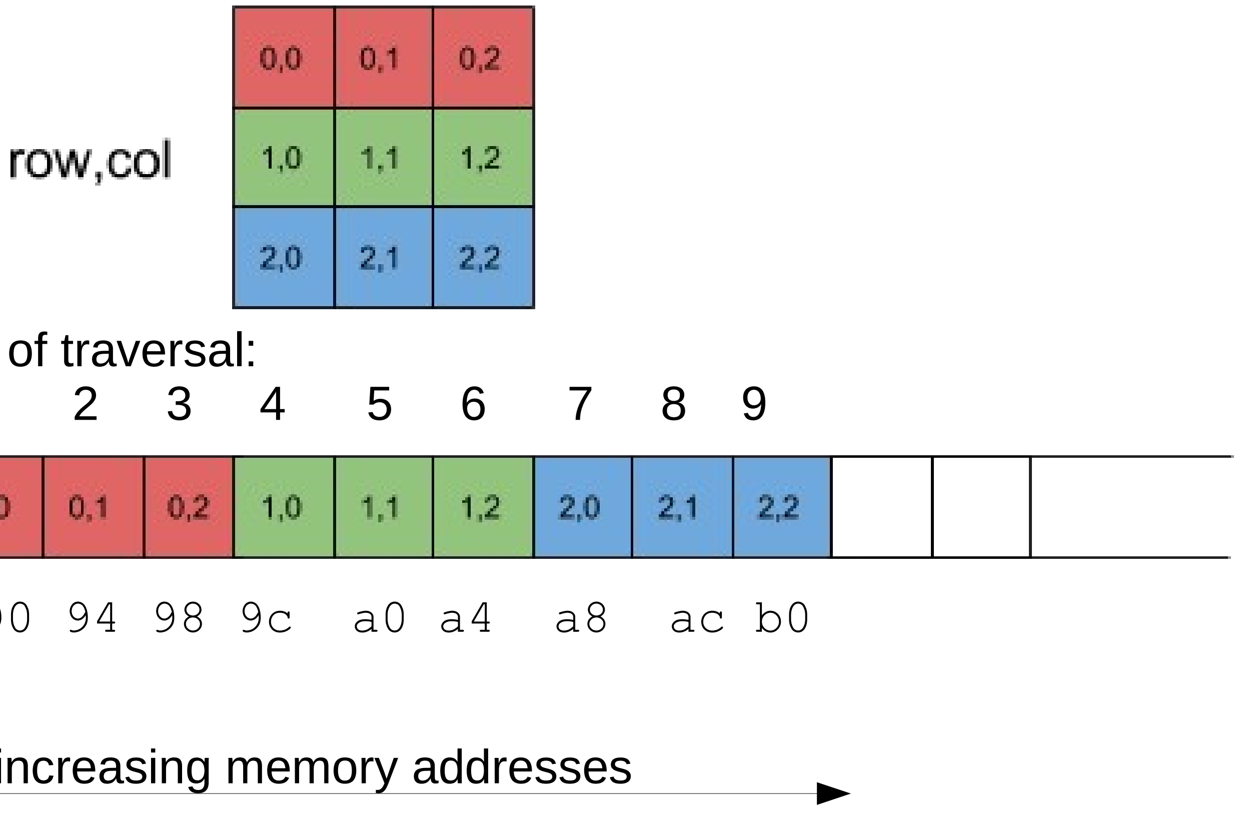
Multidimensional array layout: column-first traversal

```
int main() {
    int my_array[3][3];
    for (int col=0; col<3; col++)
        for (int row=0; row<3; row++)
            my_array[row][col]=0;
}
```



## Multidimensional array layout: row-first traversal

```
int main() {
    int my_array[3][3];
    for (int row=0; row<3; row++)
        for (int col=0; col<3; col++)
            my_array[row][col]=0;
}
```



v1

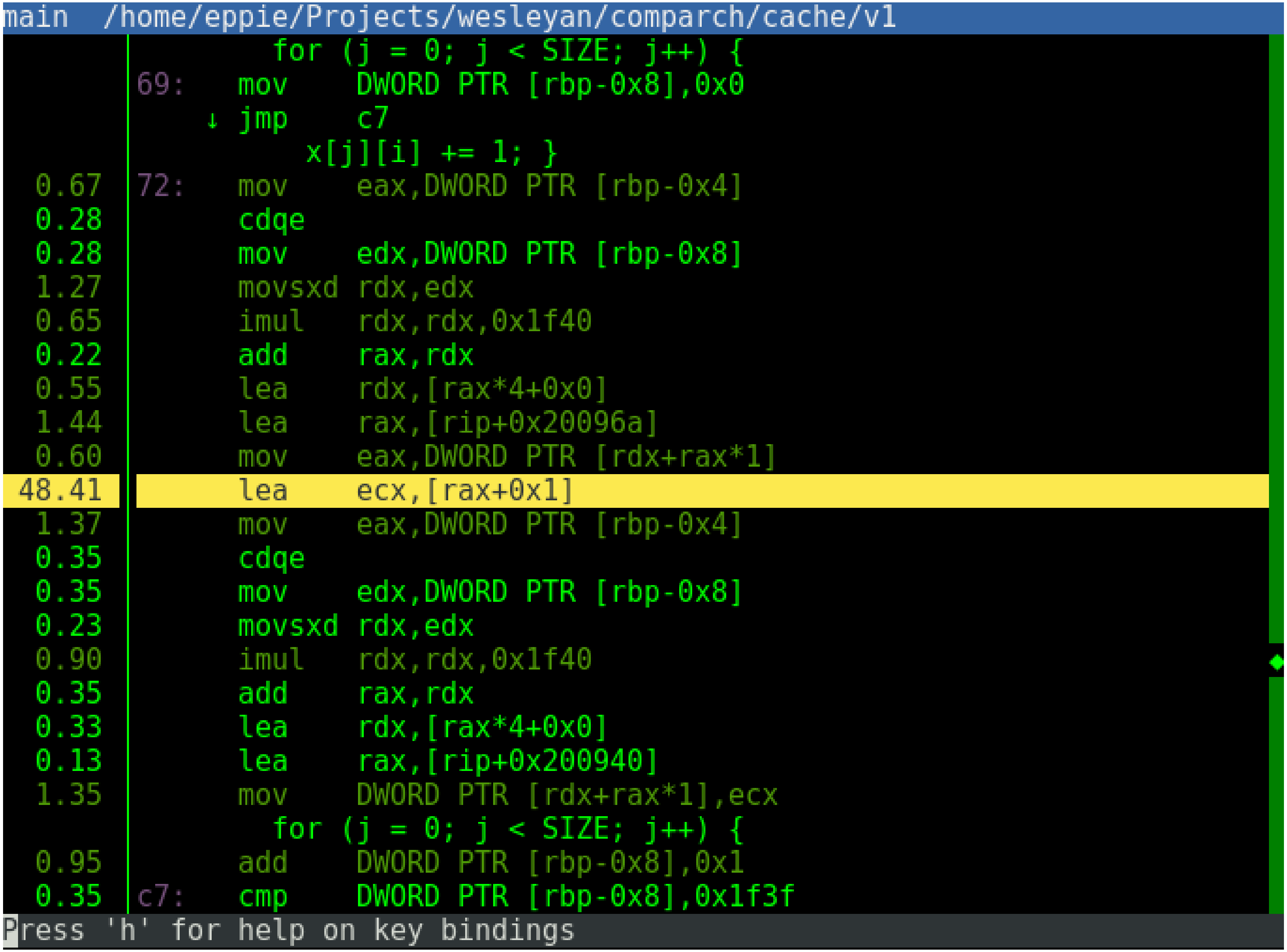
```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 8000

int main () {
  int i,j;
  static int x[SIZE][SIZE];
  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      x[j][i] = i + j; }
  }
  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      x[j][i] += 1; }
  }
  return 0;
}
```

v2

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 8000

int main () {
  int i,j;
  static int x[SIZE][SIZE];
  for (j = 0; j < SIZE; j++) {
     for (i = 0; i < SIZE; i++) {
       x[j][i] = i + j; }
   }
  for (j = 0; j < SIZE; j++) {
     for (i = 0; i < SIZE; i++) {
       x[j][i] += 1; }
   }
  return 0;
}
```

```
$ sudo perf stat -e task-clock,cycles,instructions,L1-dcache-loads,L1-dcache-misses ./v1

 Performance counter stats for './v1':

       1498.077528      task-clock (msec)         #    0.999 CPUs utilized
     3,575,871,154      cycles                    #    2.387 GHz
     2,536,022,768      instructions              #    0.71  insn per cycle
       877,617,534      L1-dcache-loads           #  585.829 M/sec
       271,229,779      L1-dcache-misses          #   30.91% of all L1-dcache hits

       1.498949860 seconds time elapsed

$ sudo perf stat -e task-clock,cycles,instructions,L1-dcache-loads,L1-dcache-misses ./v2

 Performance counter stats for './v2':

        450.457602      task-clock (msec)         #    0.997 CPUs utilized
     1,249,410,086      cycles                    #    2.774 GHz
     2,535,257,152      instructions              #    2.03  insn per cycle
       877,377,093      L1-dcache-loads           # 1947.746 M/sec
         8,898,434      L1-dcache-misses          #    1.01% of all L1-dcache hits

       0.451741141 seconds time elapsed
```

```
sudo perf record -e L1-dcache-misses ./v1 && sudo perf annotate -
Mintel --source
```



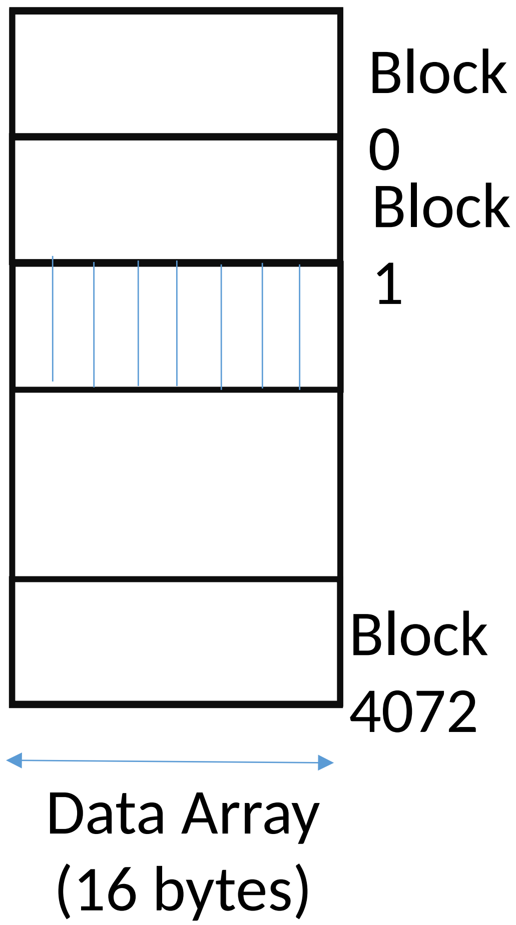
```
main  /home/eppie/Projects/wesleyan/comparch/cache/v1
              for (j = 0; j < SIZE; j++) {
       69:    mov     DWORD PTR [rbp-0x8],0x0
            ↓ jmp     c7
                  x[j][i] += 1; }
  0.67 72:    mov     eax,DWORD PTR [rbp-0x4]
  0.28        cdqe
  0.28        mov     edx,DWORD PTR [rbp-0x8]
  1.27        movsxd rdx,edx
  0.65        imul    rdx,rdx,0x1f40
  0.22        add     rax,rdx
  0.55        lea     rdx,[rax*4+0x0]
  1.44        lea     rax,[rip+0x20096a]
  0.60        mov     eax,DWORD PTR [rdx+rax*1]
 48.41        lea     ecx,[rax+0x1]
  1.37        mov     eax,DWORD PTR [rbp-0x4]
  0.35        cdqe
  0.35        mov     edx,DWORD PTR [rbp-0x8]
  0.23        movsxd rdx,edx
  0.90        imul    rdx,rdx,0x1f40
  0.35        add     rax,rdx
  0.33        lea     rdx,[rax*4+0x0]
  0.13        lea     rax,[rip+0x200940]
  1.35        mov     DWORD PTR [rdx+rax*1],ecx
              for (j = 0; j < SIZE; j++) {
  0.95        add     DWORD PTR [rbp-0x8],0x1
  0.35 c7:    cmp     DWORD PTR [rbp-0x8],0x1f3f
Press 'h' for help on key bindings
```

# Loop Interchange

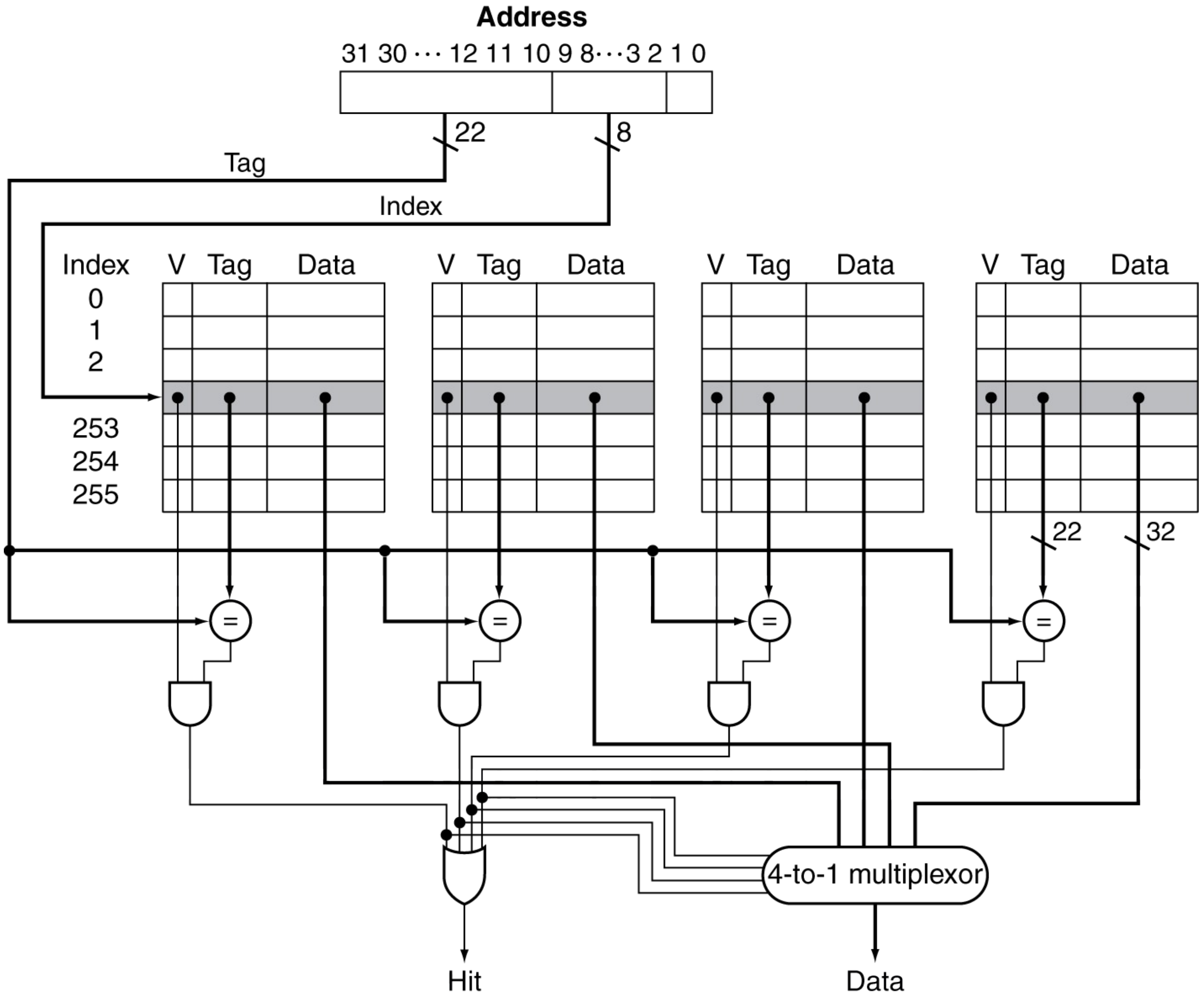
```
for (i=0; i<2048; i++)
    for (j=0; i<2048; i++)
        sum+= a[j][i];
```

```
//Arrays stored in column major order
//a[0][0]
//a[0][1]….
//a[1][0] …
```

```
    for (j=0; i<2048; i++)
        for (i=0; i<2048; i++)
            sum+= a[j][i];
```

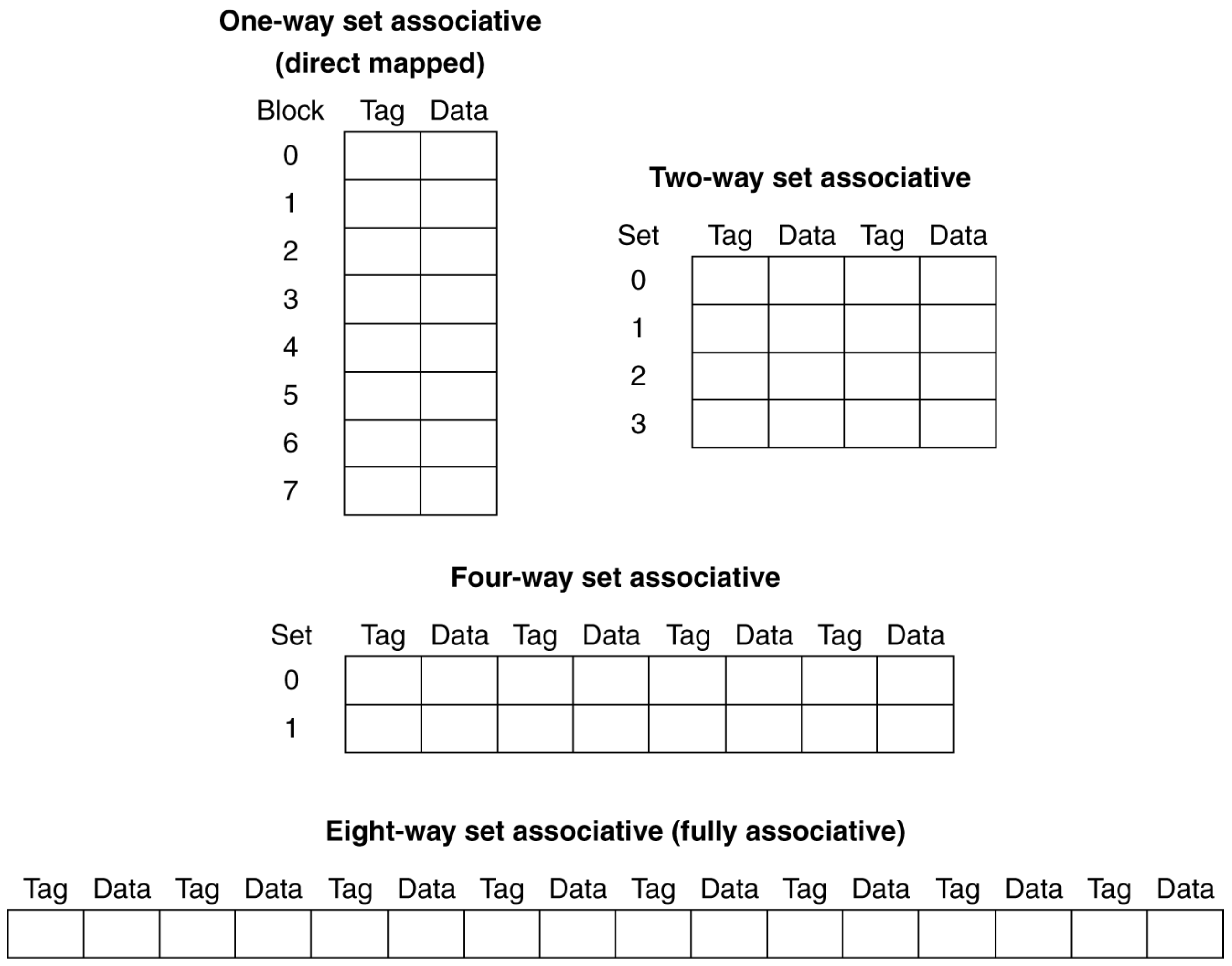


# Set Associative Cache Organization



## Spectrum of Associativity

- For a cache with 8 entries 4 possibilities:


One-way set associative
(direct mapped)
Block Tag Data
0
1
2
3
4
5
6
7
Two-way set associative
Set Tag Data Tag Data
0
1
2
3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0
1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

# Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - 4 bytes per block
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 |
| 0 | 0 | miss | **Mem[0]** | | | |
| 8 | 0 | miss | **Mem[8]** | | | |
| 0 | 0 | miss | **Mem[0]** | | | |
| 6 | 2 | miss | Mem[0] | | **Mem[6]** | |
| 8 | 0 | miss | **Mem[8]** | | Mem[6] | |

# Associativity Example

- 2-way set associative

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | Set 0 | | Set 1 | |
| 0 | 0 | miss | **Mem[0]** | | | |
| 8 | 0 | miss | Mem[0] | **Mem[8]** | | |
| 0 | 0 | hit | **Mem[0]** | Mem[8] | | |
| 6 | 0 | miss | Mem[0] | **Mem[6]** | | |
| 8 | 0 | miss | **Mem[8]** | Mem[6] | | |

- Fully associative

| Block address | | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| 0 | | miss | **Mem[0]** | | | |
| 8 | | miss | Mem[0] | **Mem[8]** | | |
| 0 | | hit | **Mem[0]** | Mem[8] | | |
| 6 | | miss | Mem[0] | Mem[8] | **Mem[6]** | |
| 8 | | hit | Mem[0] | **Mem[8]** | Mem[6] | |

## Memory Performance

- **Hit:** data found in that level of memory hierarchy
- **Miss:** data not found (must go to next level)

**Hit Rate** = # hits / # memory accesses
= 1 – Miss Rate

**Miss Rate** = # misses / # memory accesses
= 1 – Hit Rate

- **Average memory access time (AMAT):** average time for processor to access data

**AMAT** = $t_{\text{cache}} + MR_{\text{cache}} * t_{MM}$

## Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values found in cache
- Rest supplied by higher levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

**Hit Rate** = 1250/2000 **= 0.625**

**Miss Rate** = 750/2000 = **0.375** = 1 – Hit Rate

## Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}}$ = 1 cycle
- $t_{MM}$ = 100 cycles
- **What is the AMAT of the program from Example 1?**

$$\begin{aligned}\textbf{AMAT} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= \mathbf{38.5\ cycles}\end{aligned}$$

# Improving Cache Performance

- Three basic approaches to improving cache performance:
  - Reduce the miss penalty
  - Reduce the miss rate
  - Reduce the hit time
- Different techniques attack different aspects
- As usual, there are tradeoffs, i.e., improving one comes at the cost of another
- Main considerations: size of cache, size of blocks, amount of associativity

## Multilevel Caches

- Primary cache attached to CPU
  - Small, very fast (1 cycle hit time)
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory (10 cycles hit time)
- Main memory services L-2 cache misses
- L-3 cache (and higher) typical on high performance chips

# Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

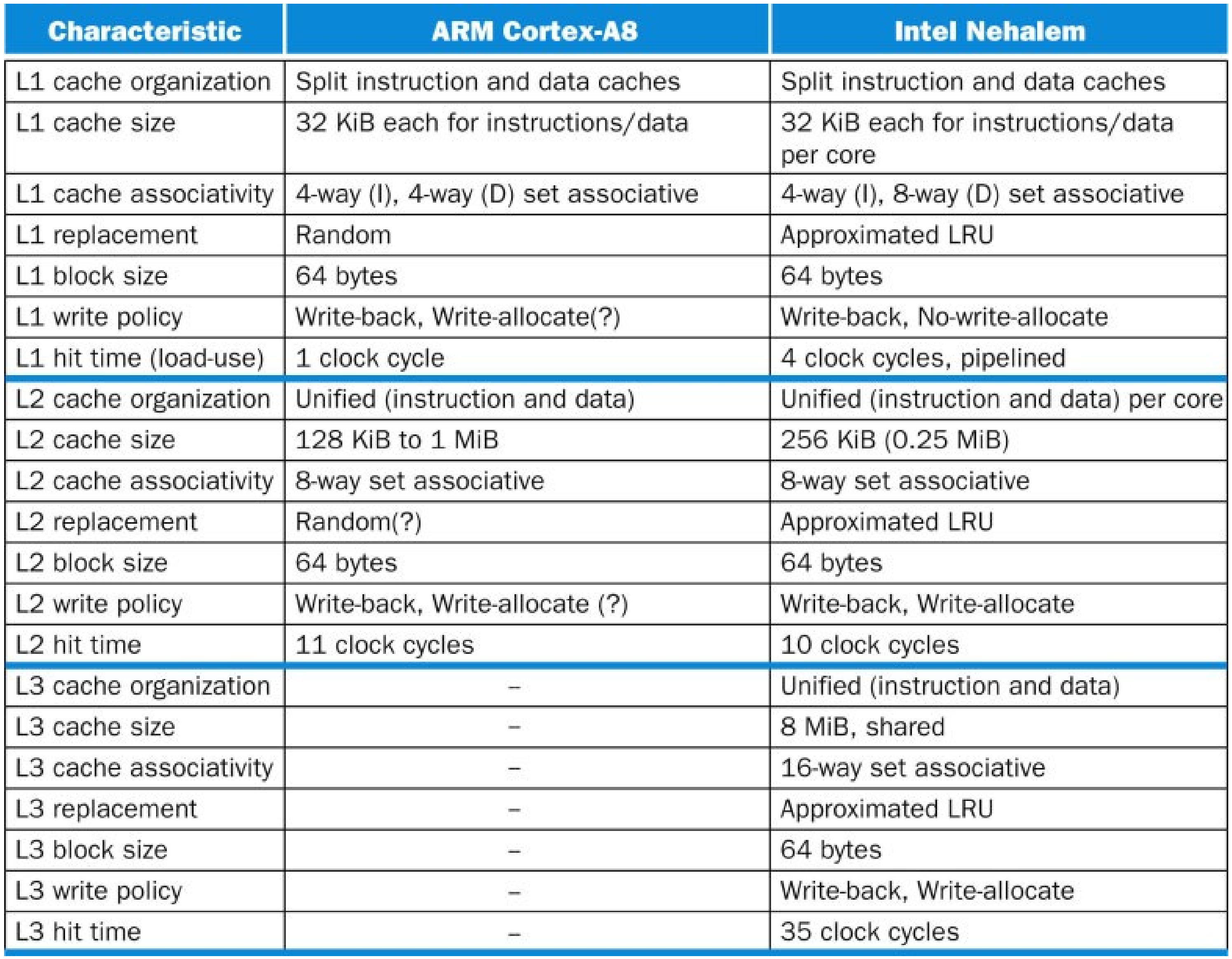
# Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

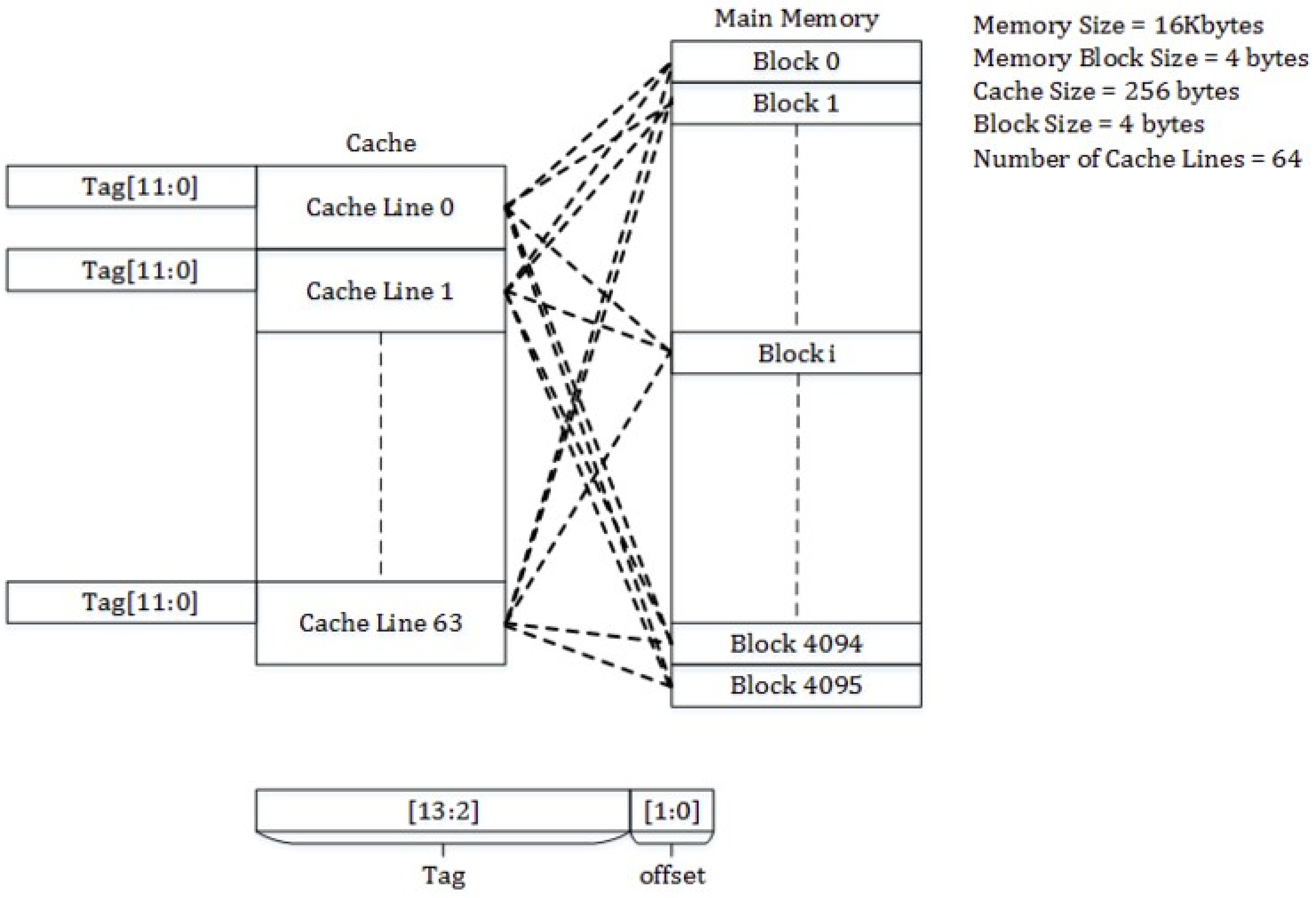
# Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than if using a single cache
  - L-1 block size often smaller than L-2 block size

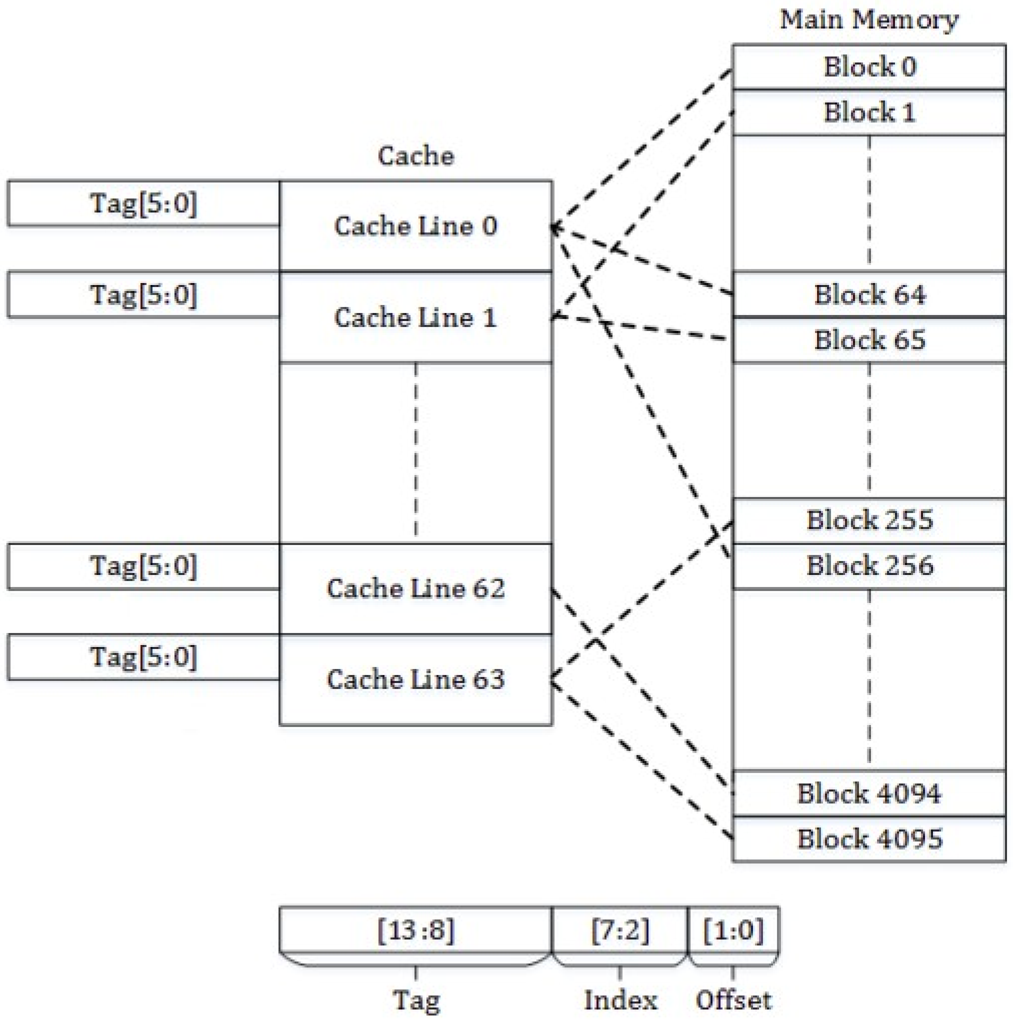
# Some Real Caches



| Characteristic | ARM Cortex-A8 | Intel Nehalem |
|---|---|---|
| L1 cache organization | Split instruction and data caches | Split instruction and data caches |
| L1 cache size | 32 KiB each for instructions/data | 32 KiB each for instructions/data per core |
| L1 cache associativity | 4-way (I), 4-way (D) set associative | 4-way (I), 8-way (D) set associative |
| L1 replacement | Random | Approximated LRU |
| L1 block size | 64 bytes | 64 bytes |
| L1 write policy | Write-back, Write-allocate(?) | Write-back, No-write-allocate |
| L1 hit time (load-use) | 1 clock cycle | 4 clock cycles, pipelined |
| L2 cache organization | Unified (instruction and data) | Unified (instruction and data) per core |
| L2 cache size | 128 KiB to 1 MiB | 256 KiB (0.25 MiB) |
| L2 cache associativity | 8-way set associative | 8-way set associative |
| L2 replacement | Random(?) | Approximated LRU |
| L2 block size | 64 bytes | 64 bytes |
| L2 write policy | Write-back, Write-allocate (?) | Write-back, Write-allocate |
| L2 hit time | 11 clock cycles | 10 clock cycles |
| L3 cache organization | – | Unified (instruction and data) |
| L3 cache size | – | 8 MiB, shared |
| L3 cache associativity | – | 16-way set associative |
| L3 replacement | – | Approximated LRU |
| L3 block size | – | 64 bytes |
| L3 write policy | – | Write-back, Write-allocate |
| L3 hit time | – | 35 clock cycles |

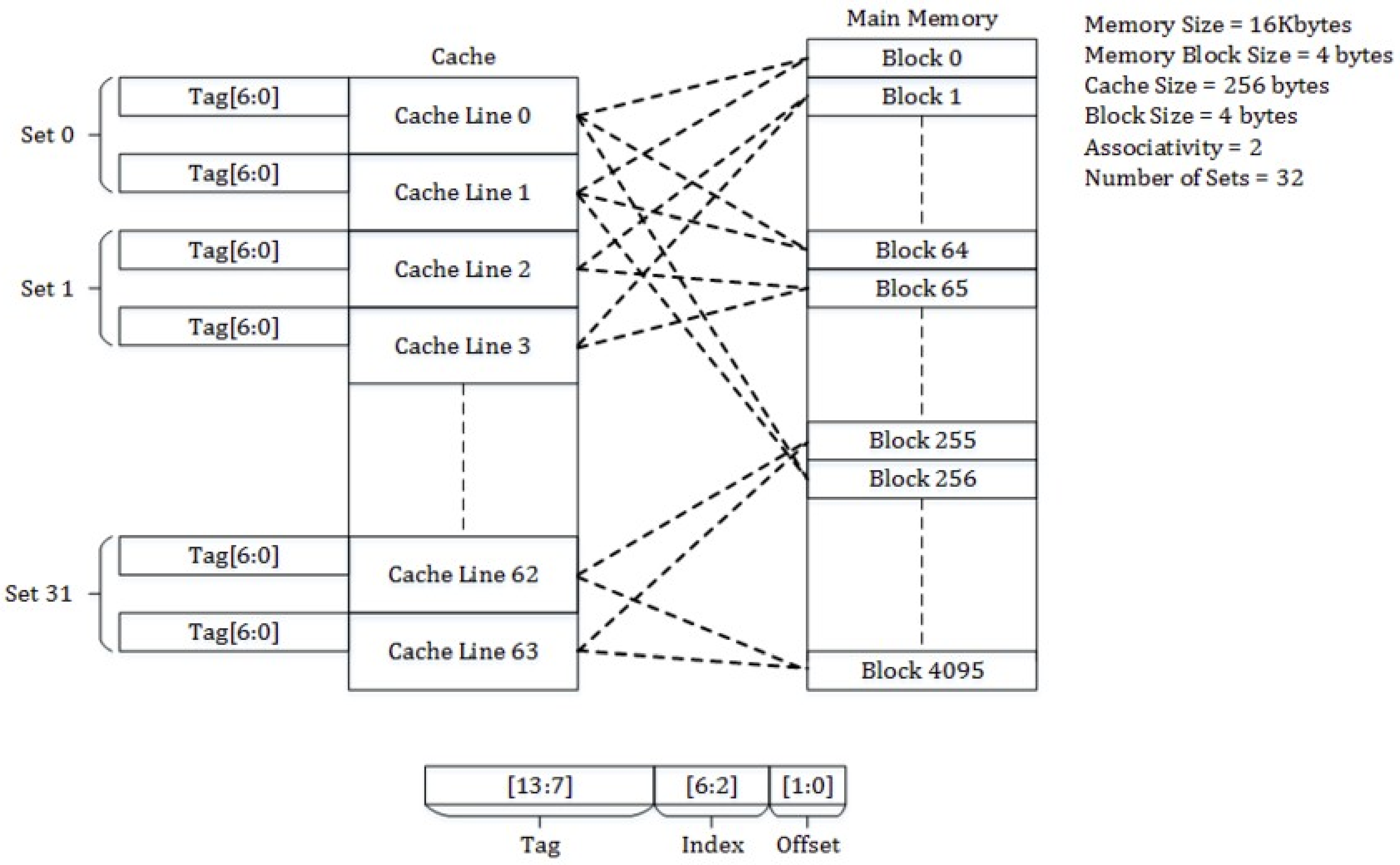


Fully associative



Memory Size = 16Kbytes
Memory Block Size = 4 bytes
Cache Size = 256 bytes
Block Size = 4 bytes
Associativity = 1
Number of Sets = 64

direct-mapped



set-associative