18. Block Size and Writes

EECS 370 – Introduction to Computer Organization – Fall 2020

Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

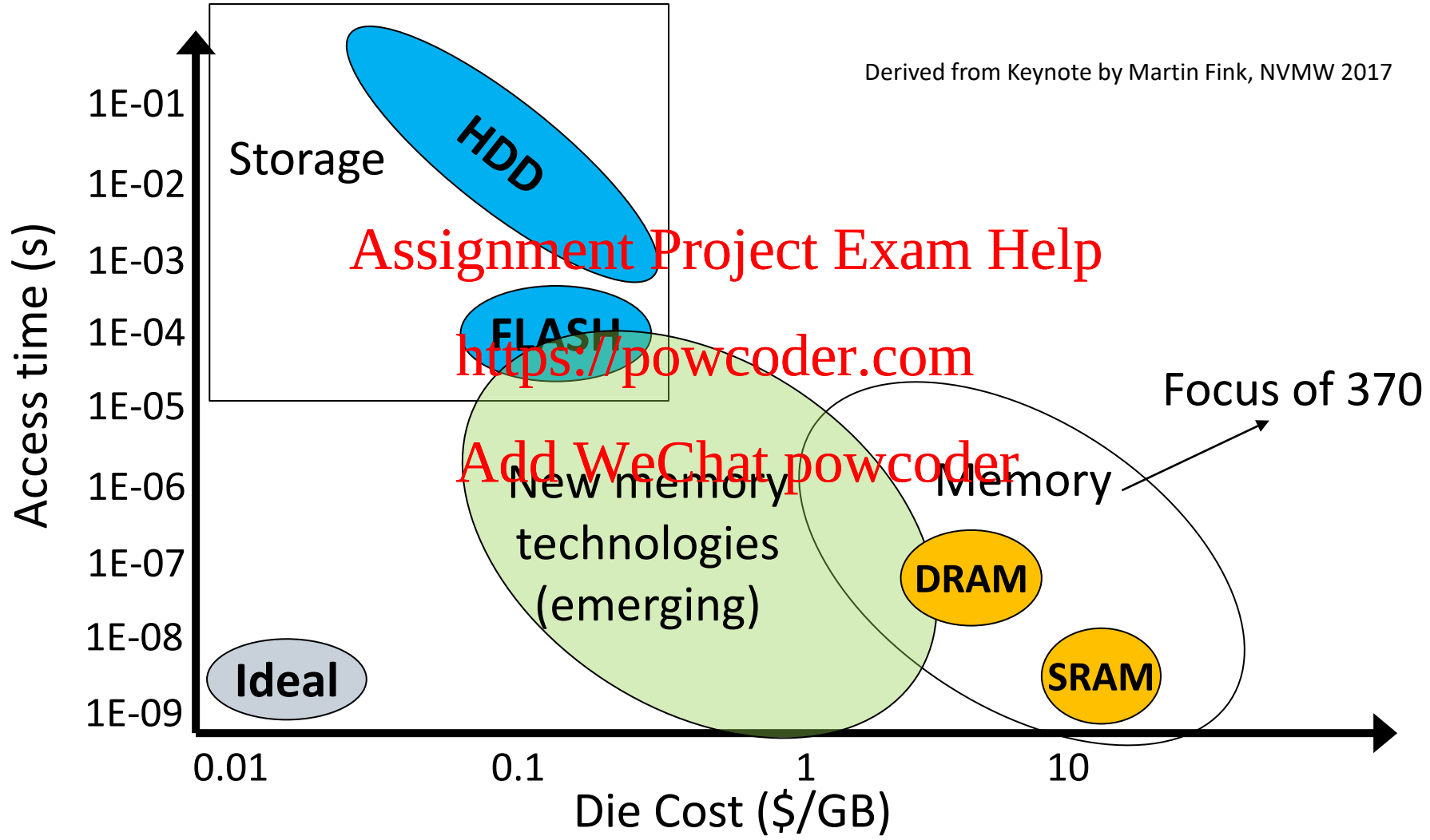
Upcoming deadlines:

HW4	due Nov 10th
Project 3	due Nov. 12th

Midterm regrade issues are being worked out. Should be resolved this week.

Recap: Memory Hierarchy and Cache: The Basics

Memory Technology Design Space

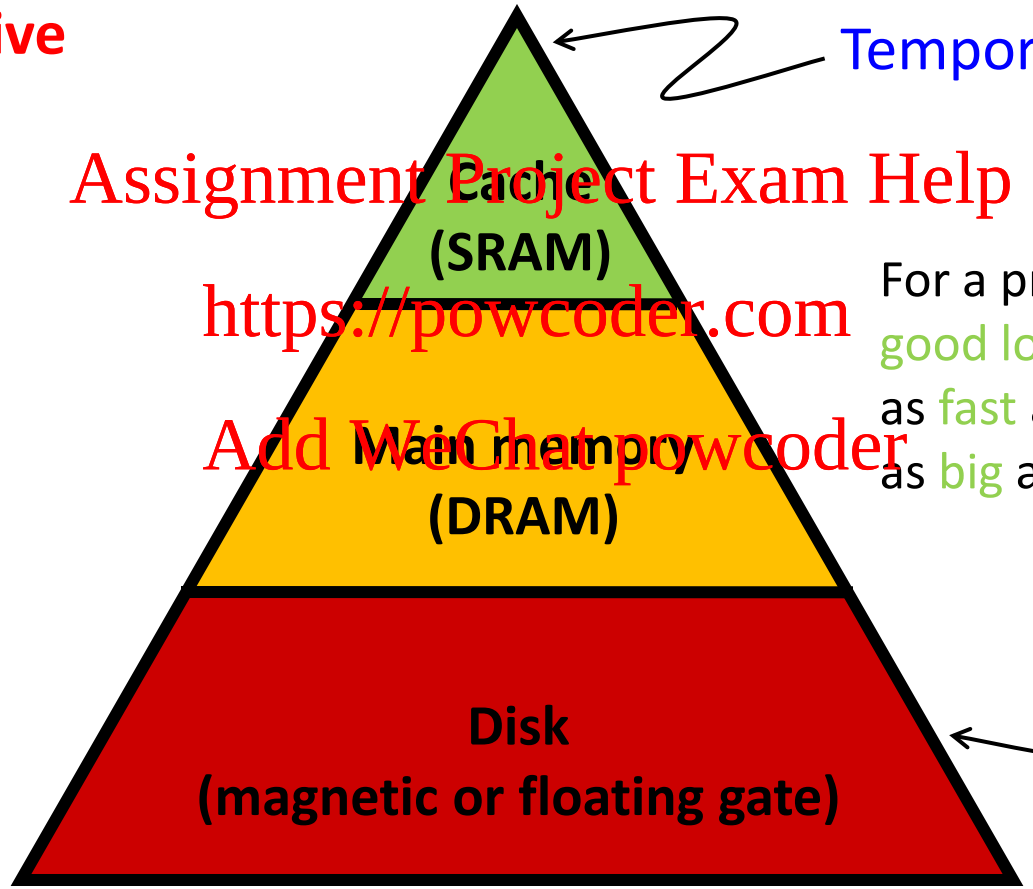


Memory hierarchy: Leveraging locality of reference

Fast

Cost Expensive

Size Small



Temporarily move what you use here

For a program with good locality of reference, memory *appears* as fast as cache and as big as disk

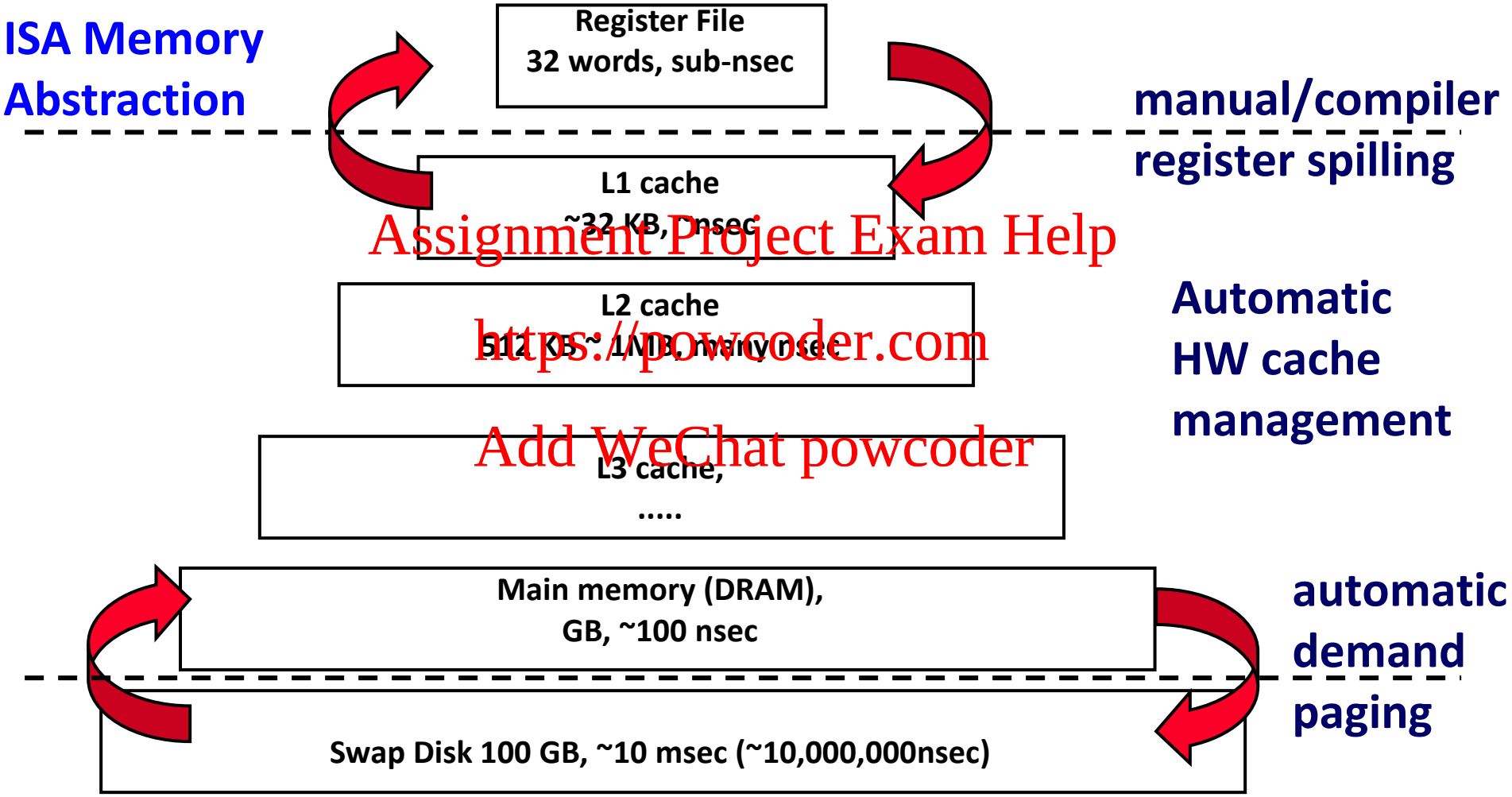
Slow

Cost Cheap

Size Big

Have a copy of everything here

Memory in a Modern System



Intel Optane

[Intel Optane web page]

Theme Article

Language Support for Memory Persistency

Aasheesh Kolli
Pennsylvania State University and VMware Research

Vaibhav Gogte
University of Michigan

Ali Saidi
Amazon Web Services

Stephan Diestelhorst
ARM Research

William Wang
University of Michigan

Peter M. Chen
ARM Research

Satish Narayanasamy
University of Michigan

Thomas F. Wenisch
University of Michigan

***Abstract*—Memory persistency models enable maintaining recoverable data structures in persistent memories and prior work has proposed ISA-level persistency models. In addition to these models, we argue for extending language-level memory models to provide persistence semantics. We present a taxonomy of guarantees a language-level persistency model could provide and characterize their programmability and performance.**

Persistent memories (PMs), such as Intel's upcoming 3D XPoint memory,[1] offer the durability of disk, better density than DRAM, and DRAM-like performance. These properties have spawned myriad efforts to adopt PM in computer systems. A particularly disruptive potential PM use case is to host in-memory recoverable data structures. PMs blur the traditional divide between a byte-addressable, volatile main memory, and a block-addressable, persistent storage. This memory allows programmers to directly manipulate recoverable data structures using processor loads and stores, rather than relying on performance-sapping software intermediaries like the operating system and file system.[2]

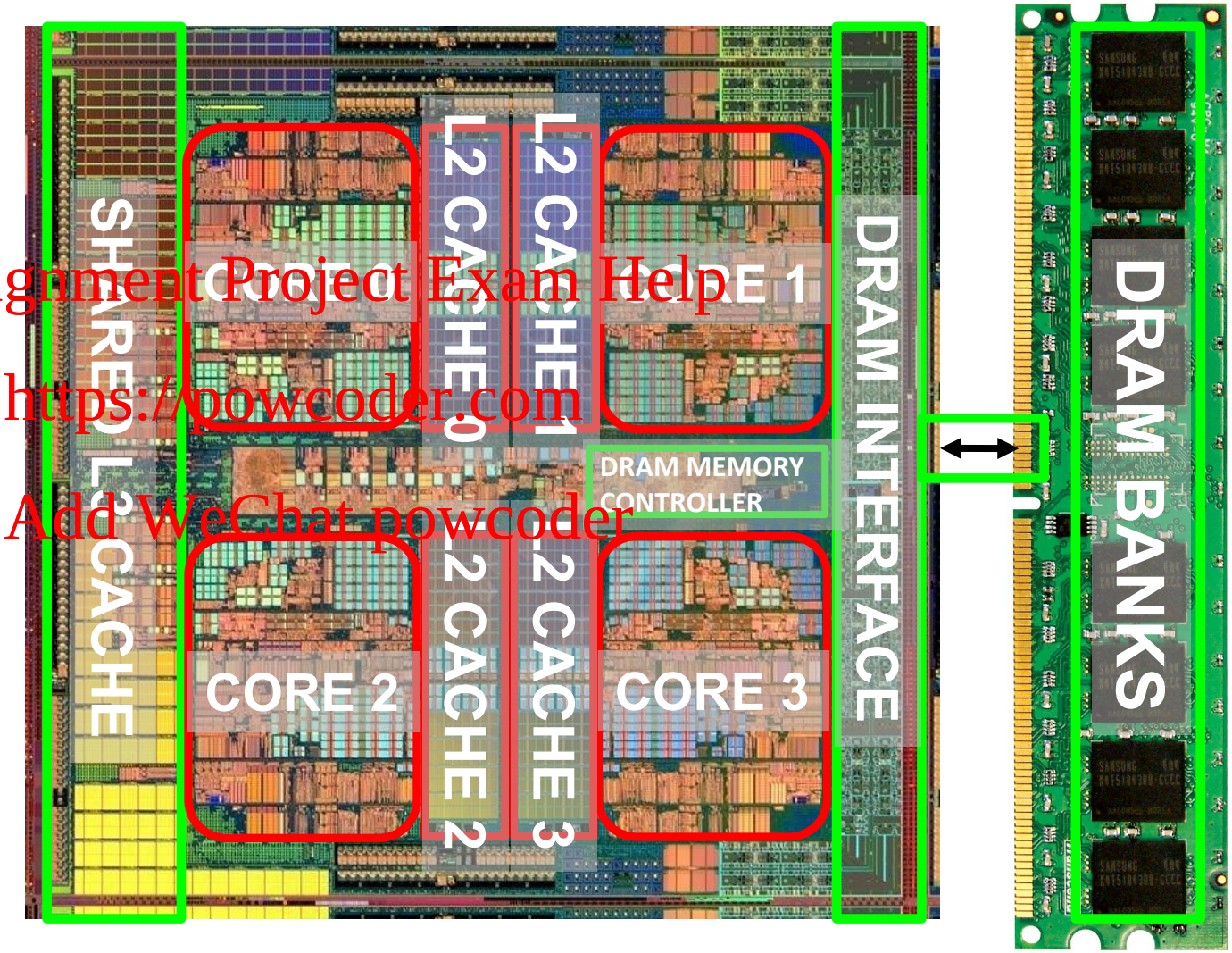
Ensuring the recoverability of data structures requires programmers to be able to control the order stores reach PM. With out-of-order processing and write-back caching, stores may reach PM out of order, compromising data structure recoverability. Existing systems do not provide efficient mechanisms to enforce the order in which stores are written back. Recent work has proposed *persistency models* to provide programmers an interface to control the order persistent stores write

Digital Object Identifier 10.1109/MM.2019.2910821
Date of publication 16 April 2019; date of current version 8 May 2019.

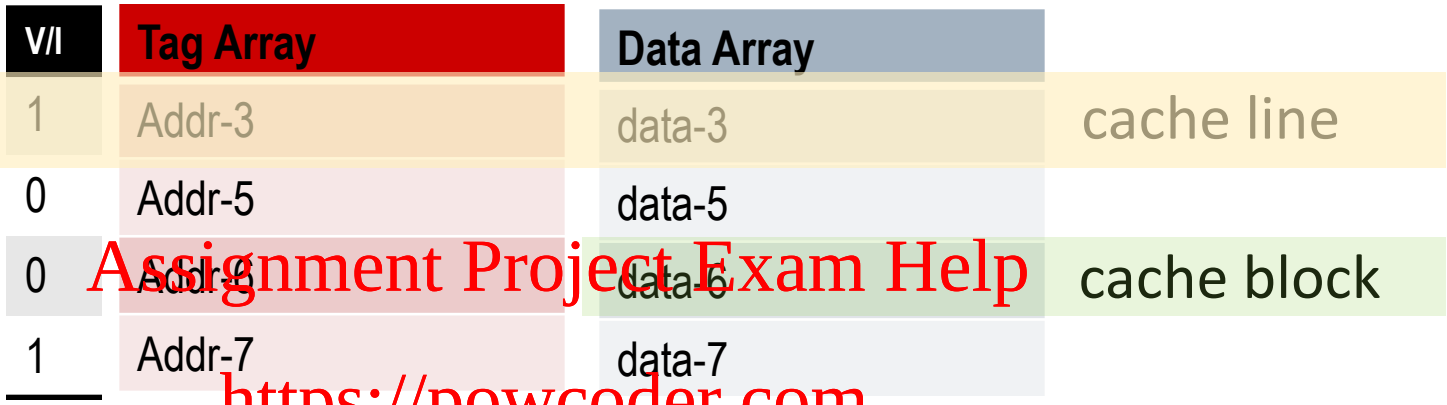
Published by the IEEE Computer Society

Cache: Importance

Caches consume
most of a processor's die area.



Review: A simple cache architecture



A cache memory consists of multiple tag/block pairs (called cache lines)

Address is searched across all tags in parallel.

- At most one tag will match
  - If there is a tag match, it is a cache HIT
  - If there is no tag match, it is a cache MISS

Cache data that is likely to be used in future – exploit temporal locality

Temporal Locality

The principle of temporal locality in program references says that if you access a memory location (e.g., 0x1000) you will be more likely to re-access that location (e.g., 0x1000) than you will be to reference some other random location

Temporal locality says any miss location should be placed into the cache

It is the most recent reference location

Temporal locality says that data in least recently referenced (or least recently used – LRU) cache line should be evicted to make room for the new line

Because the re-access probability falls over time as a cache line isn't referenced, the LRU line is least likely to be re-referenced

Tracking LRU

Naïve method

- Maintain LRU_rank per cache line
- Set the LRU_rank of newly accessed block to 0. Increment others by 1.
- Replace cache line with highest LRU_rank
- Area overhead: log(number of cache lines) per cache line

LRU with least area overhead

- # permutations for N cache lines is $n!$
- need just log($n!$) bits for N cache lines

Problem — AMAT

Assume main memory access time does not include cache access time.

Suppose that accessing a cache takes 10ns while accessing main memory in case of cache-miss takes 100ns. What is the average memory access time if the cache hit rate is 97%?

To improve performance, the cache size is increased. It is determined that this will increase the hit rate by 1%, but it will also increase the time for accessing the cache by 2ns. Will this improve the overall average memory access time?

Problem —AMAT

Suppose that accessing a cache takes 10ns while accessing main memory in case of cache-miss takes 100ns. What is the average memory access time if the cache hit rate is 97%?

AMAT = 10 + (1 - 0.97)*100 = 13 ns

To improve performance, the cache size is increased. It is determined that this will increase the hit rate by 1%, but it will also increase the time for accessing the cache by 2ns. Will this improve the overall average memory access time?

AMAT = 12 + (1 - 0.98)*100 = 14 ns

This lecture

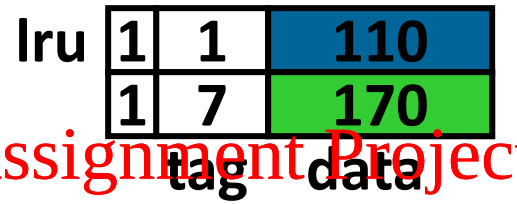
Cache blocks

Spatial locality

Problems

Stores: Write-through vs Write-back

How can we reduce the overhead for each cache line?



Cache bigger units than bytes

Each block has a single tag, and blocks can be whatever size we choose.

Increasing cache block size reduces (tag and other) overhead

Case 1:
Block size: **1** byte

V	tag	data (block)
1	0	74
1	6	160

Bits per tag
= $\log_2$(number of blocks in memory)
= $\log_2(16)$ = 4 bits

Case 2:
Block size: **2** bytes

V	tag	data (block)	
1	0	74	110
1	6	160	170

Bits per tag
= $\log_2$(number of blocks in memory)
= $\log_2(8)$ = 3 bits

	Memory	Block Case1	Block Case2
0	74	0	0
1	110	1	0
2	120	2	1
3	130	3	1
4	140	4	2
5	150	5	2
6	160	6	3
7	170	7	3
8	180	8	4
9	190	9	4
10	200	10	5
11	210	11	5
12	220	12	6
13	230	13	6
14	240	14	7
15	250	15	7

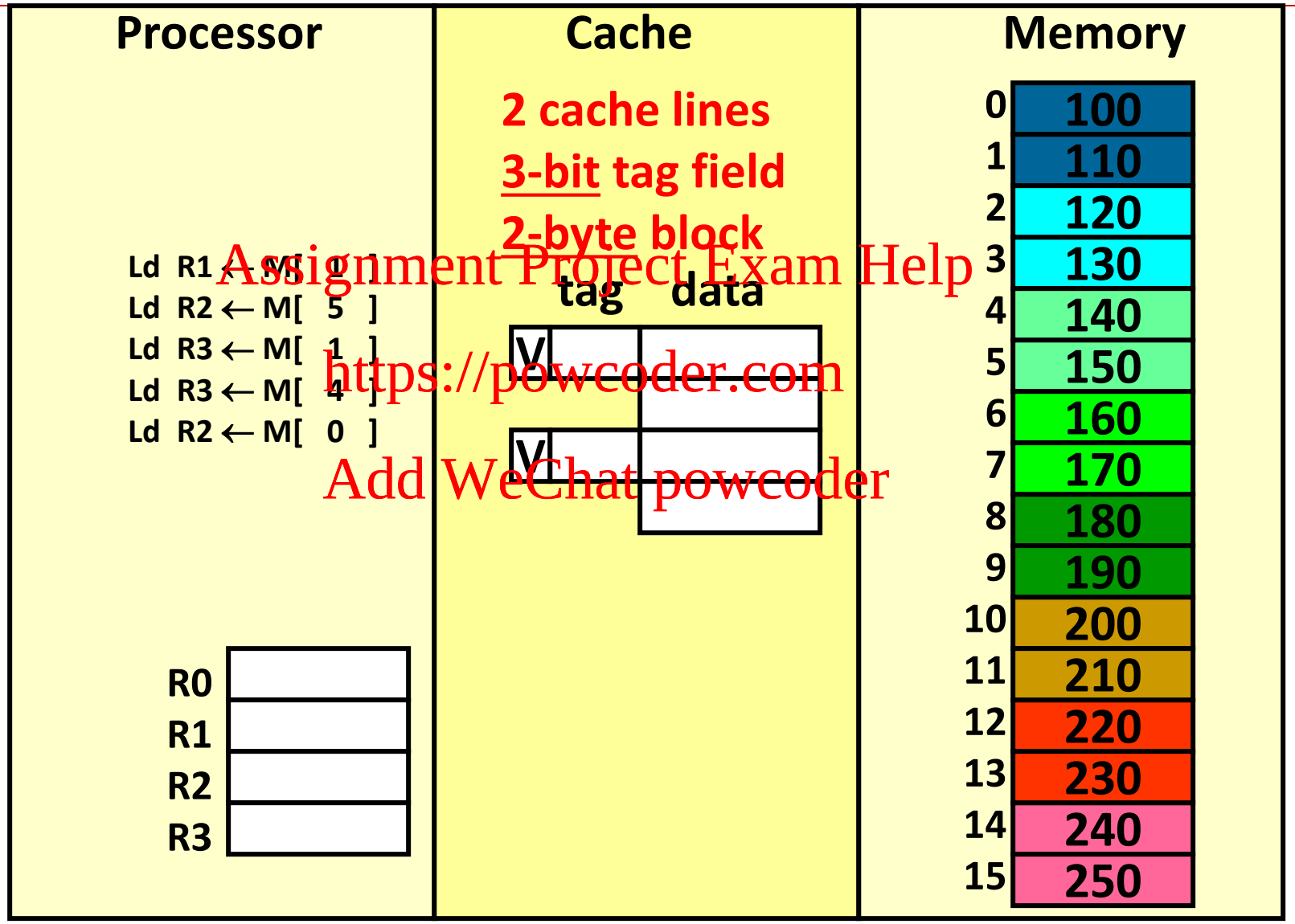
Block size: Idea

If you increase cache block size

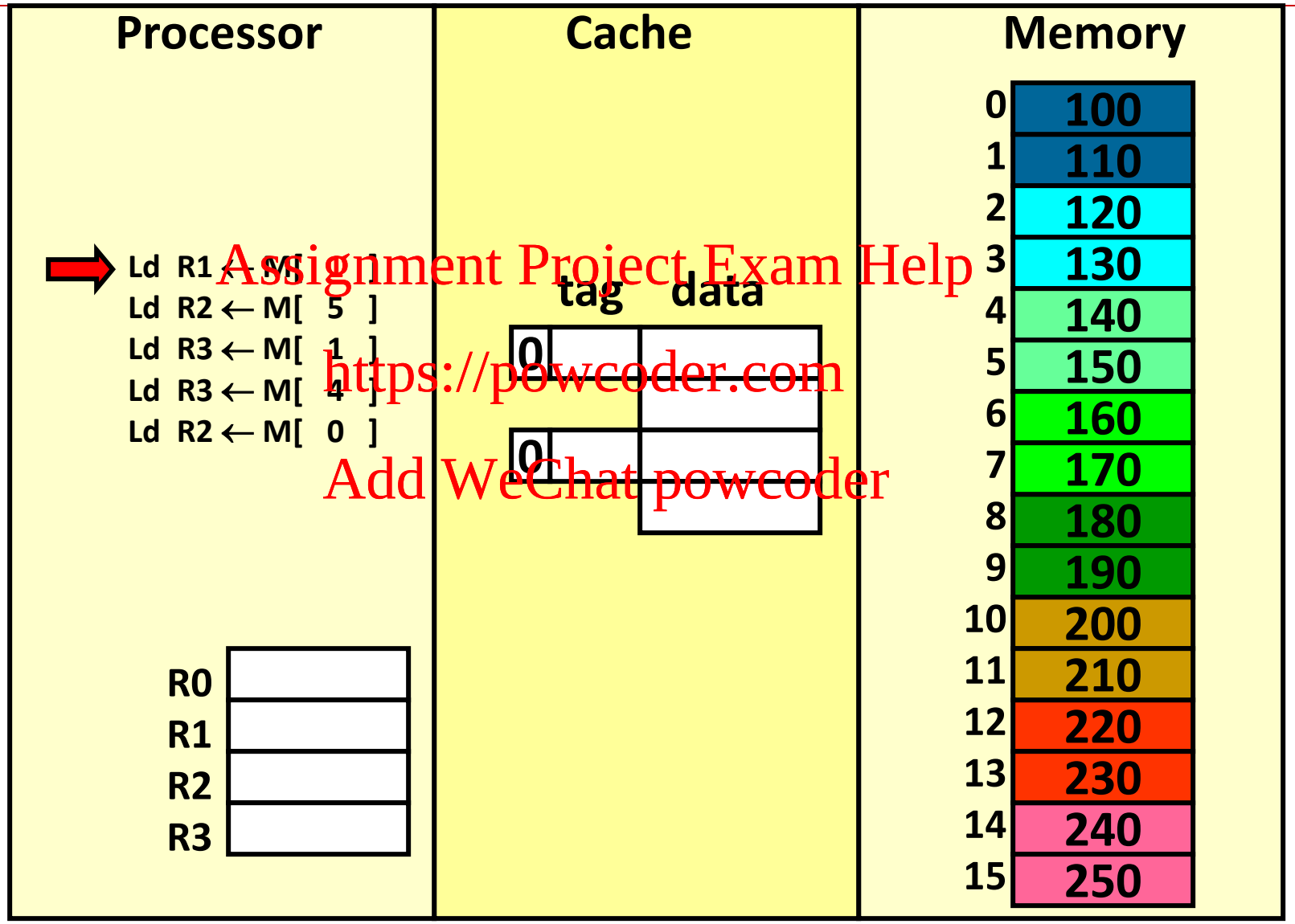
=> Increases data per cache line

=> Increases data per tag + reduces tag size

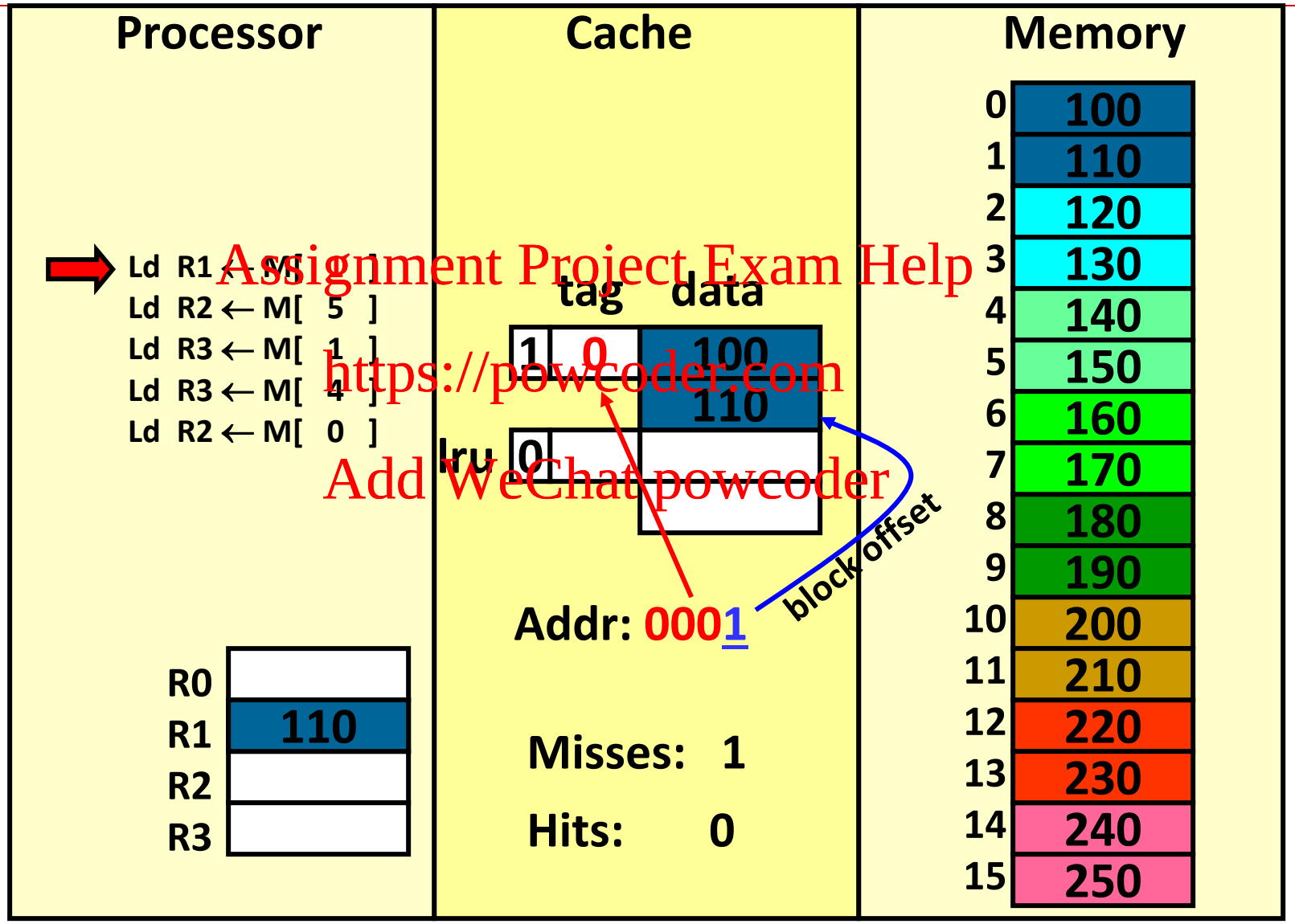
Block size for caches



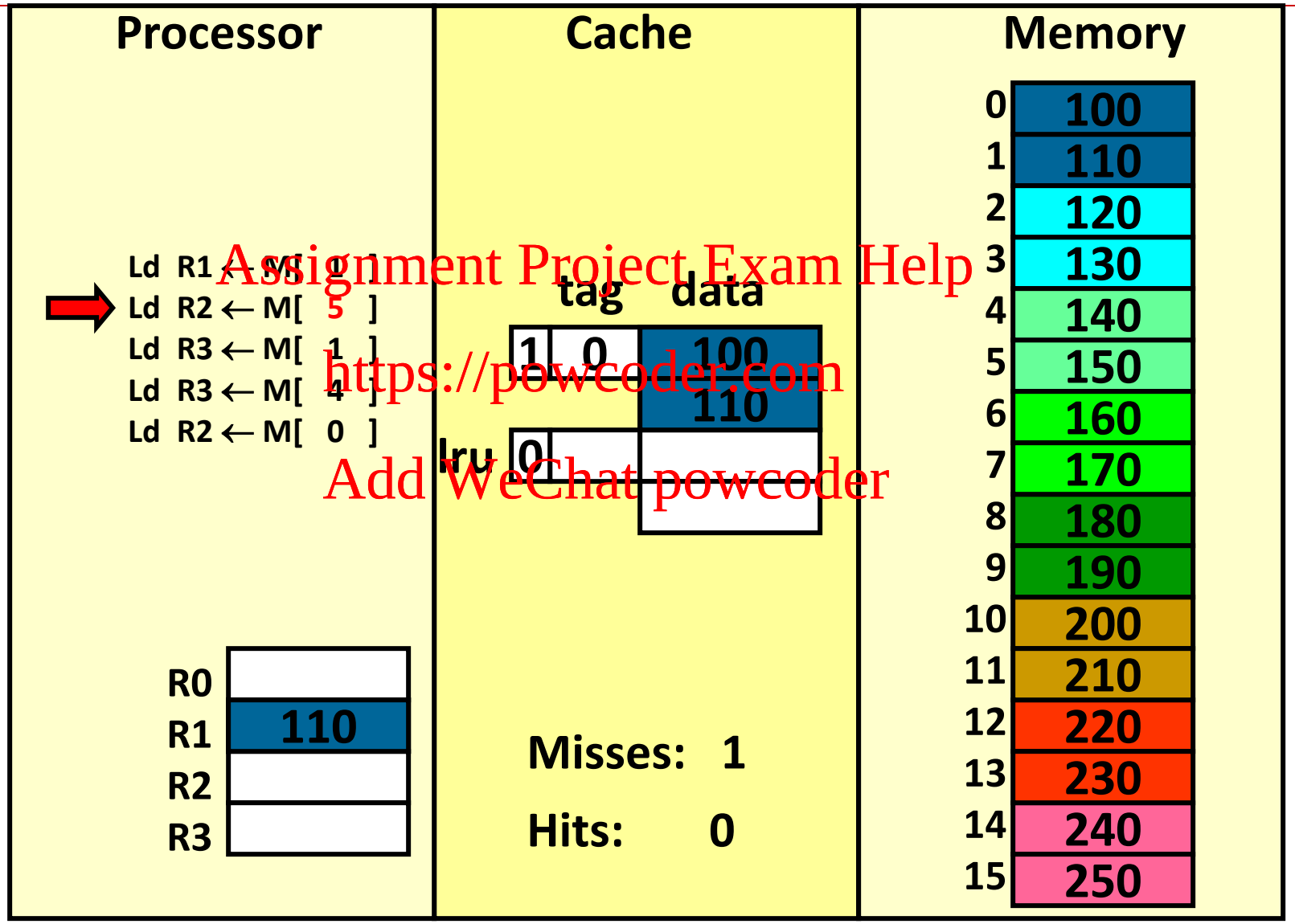
Block size for caches



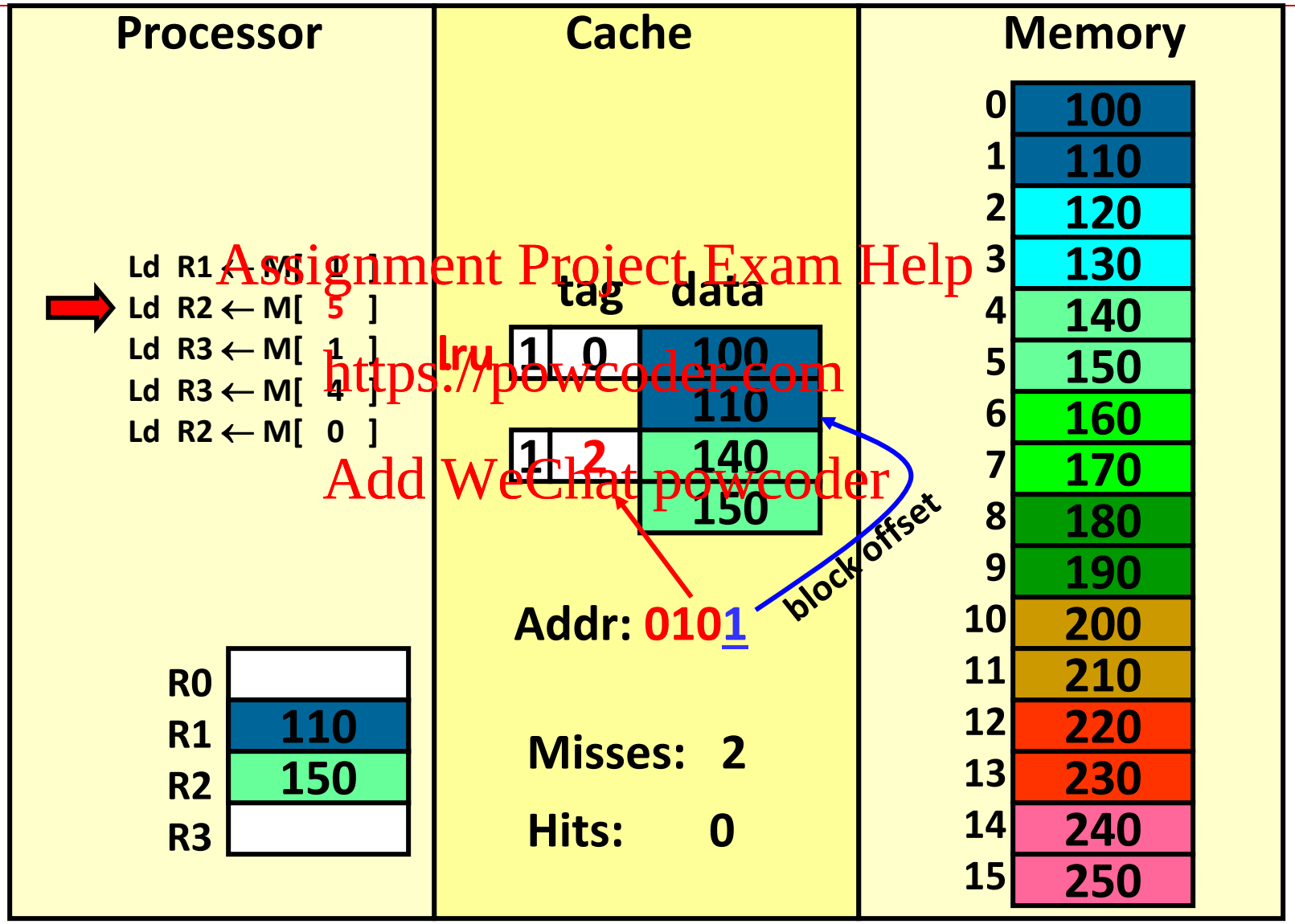
Block size for caches



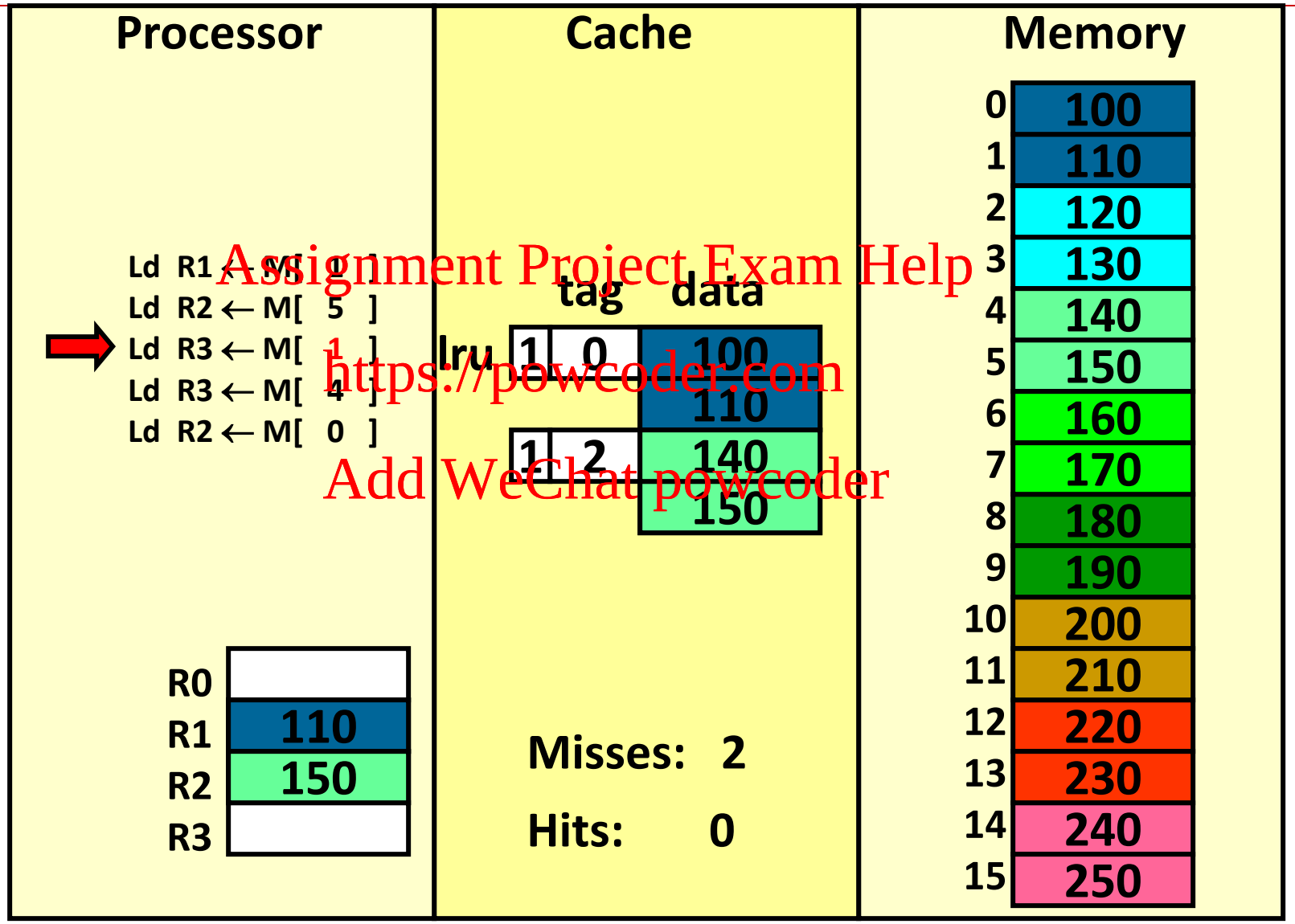
Block size for caches



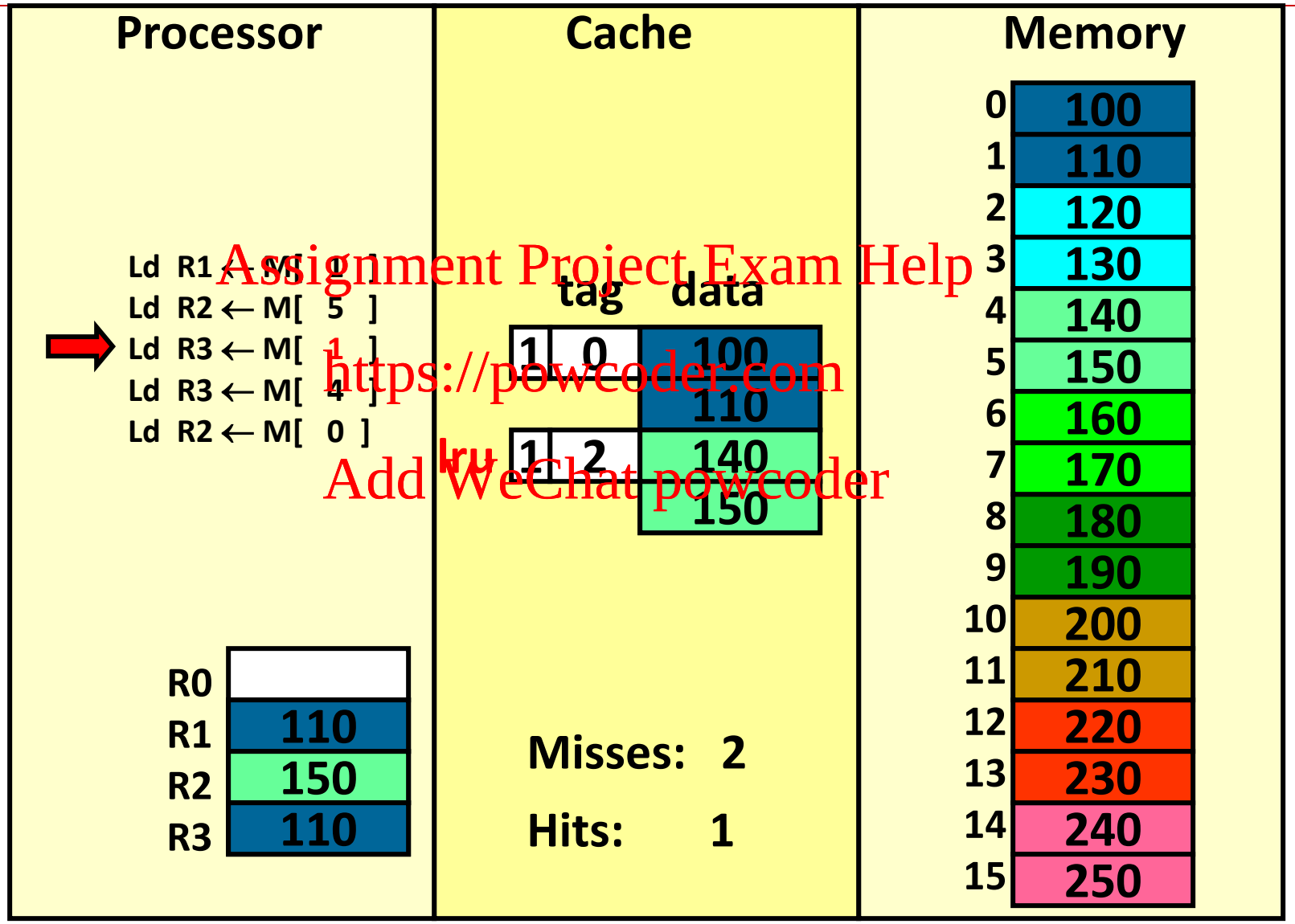
Block size for caches



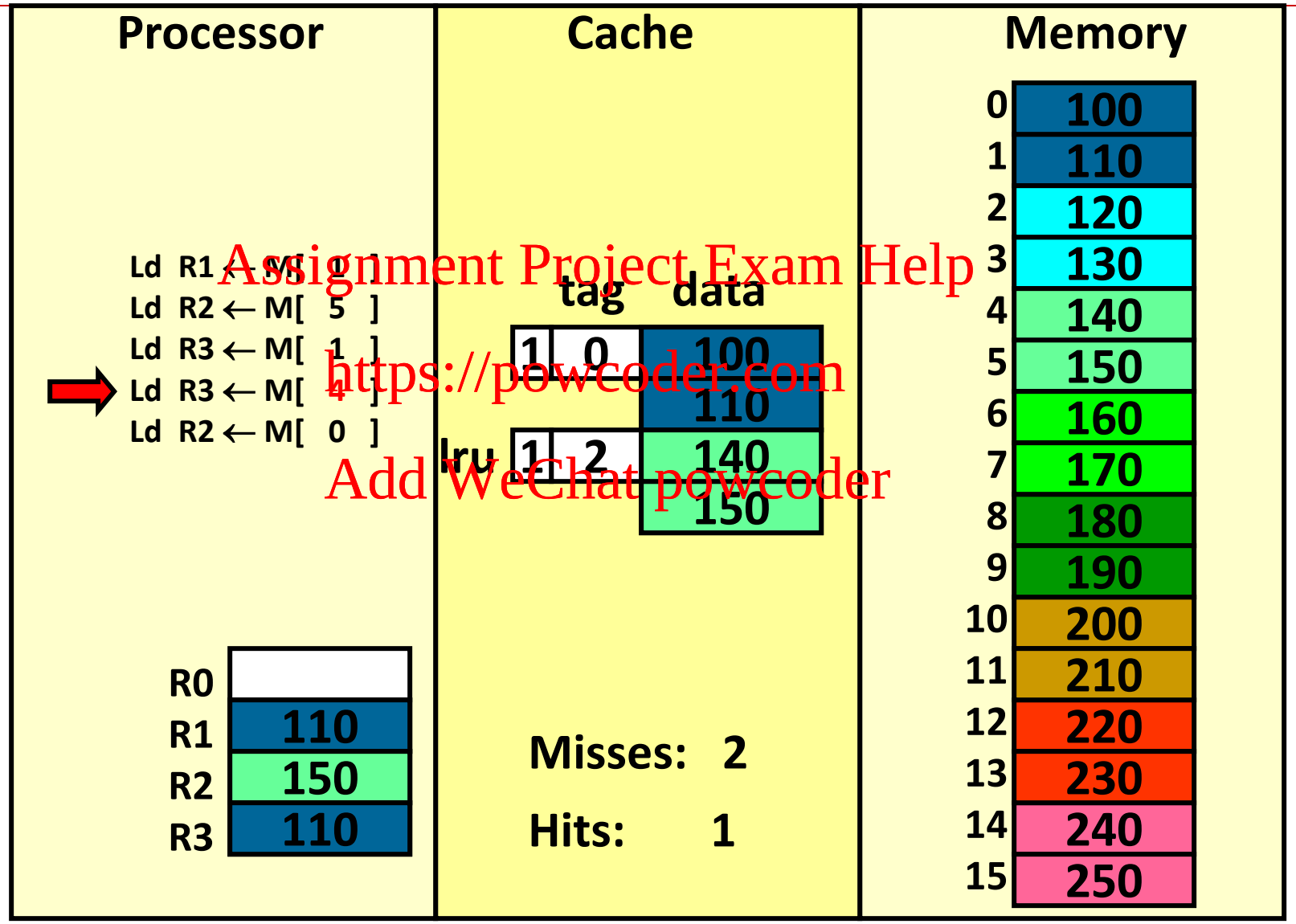
Block size for caches



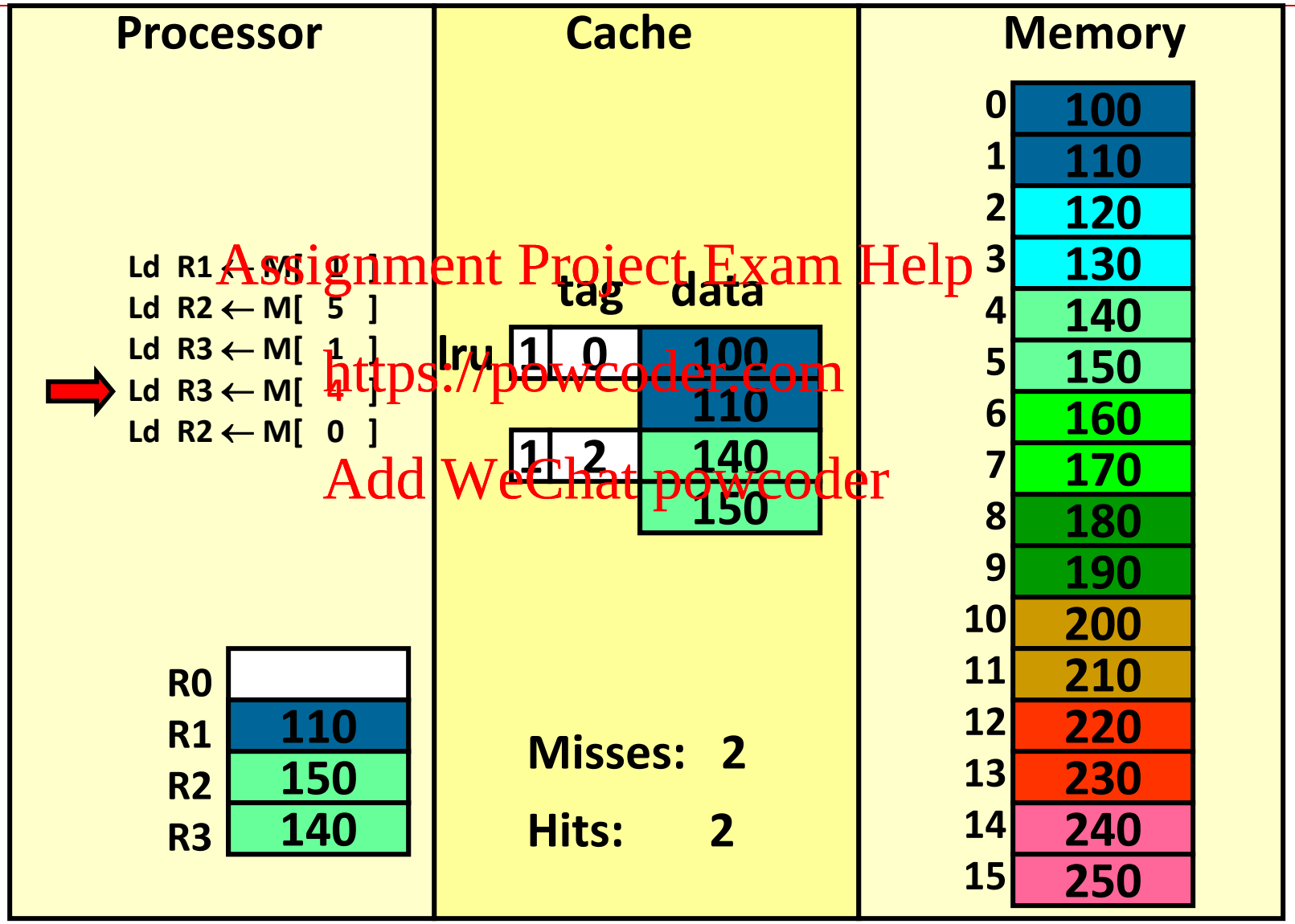
Block size for caches



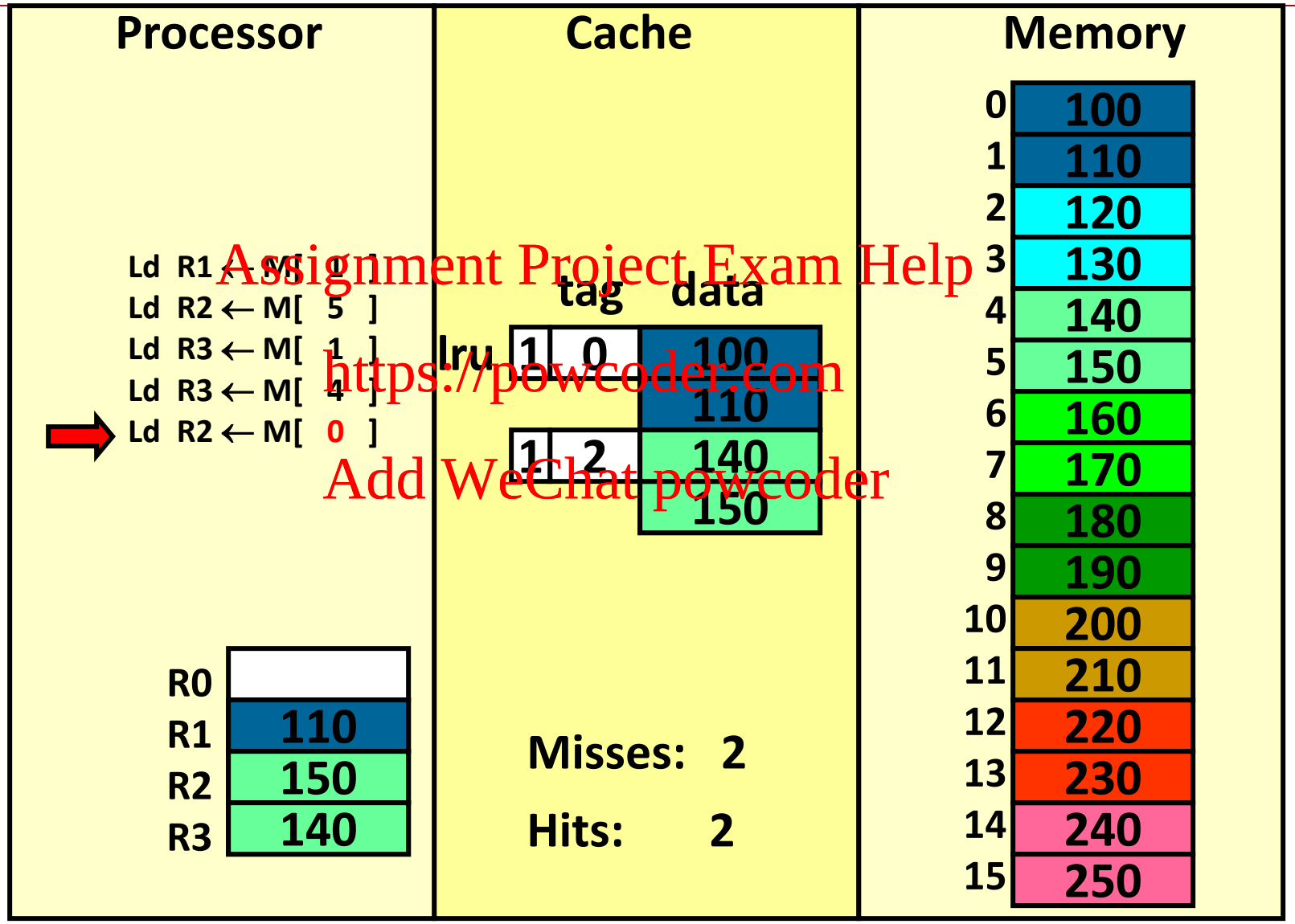
Block size for caches



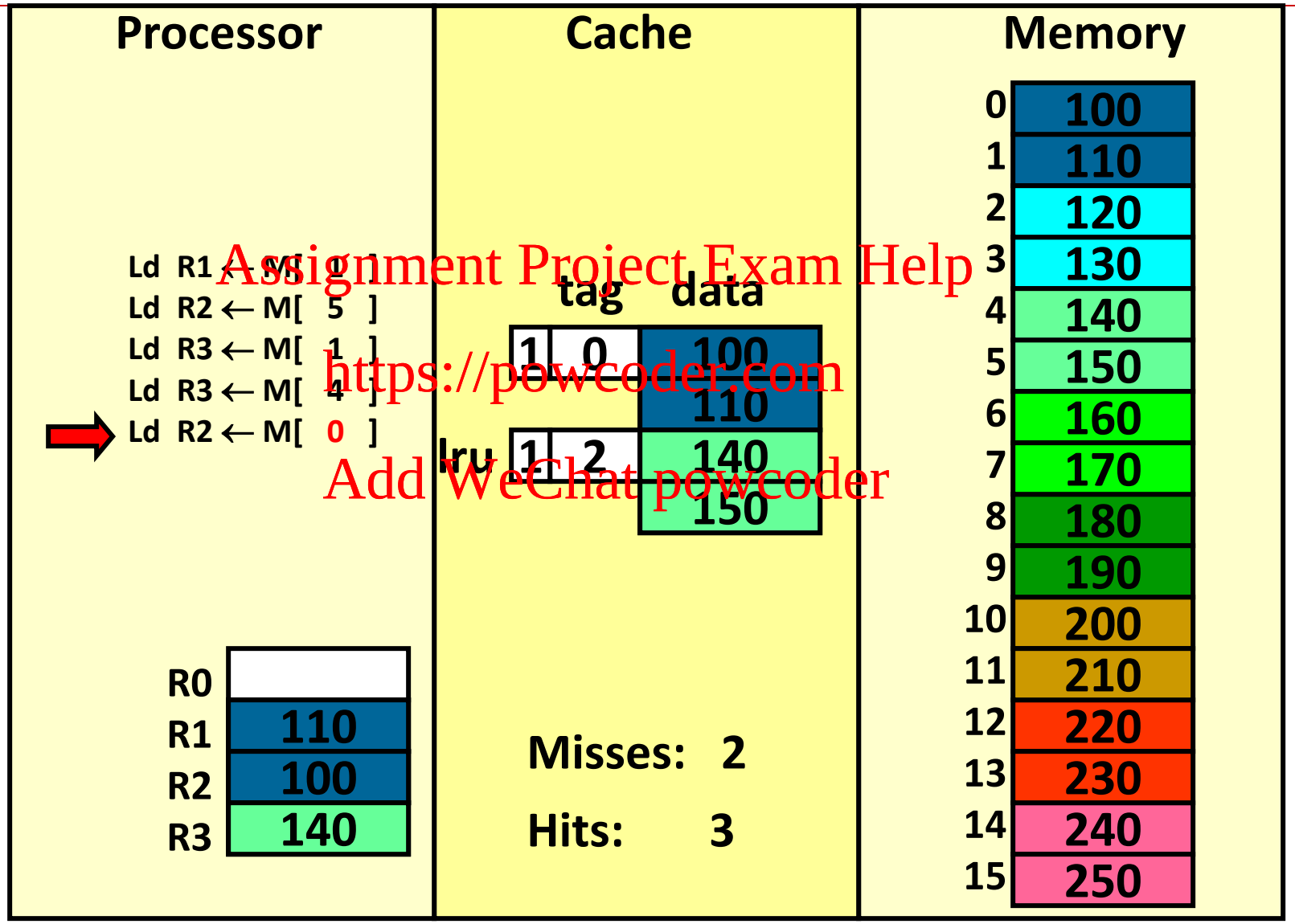
Block size for caches



Block size for caches



Block size for caches



Spatial Locality

When we accessed address 1, we also brought in address 0.

This turned out to be a good thing since we later referenced address 0 and found it in the cache.

Spatial locality in a program says that if we reference a memory location (e.g., 1000), we are more likely to reference a location near it (e.g. 1001, 999) than some random location.

Storing arrays in memory: Row major vs Column major

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

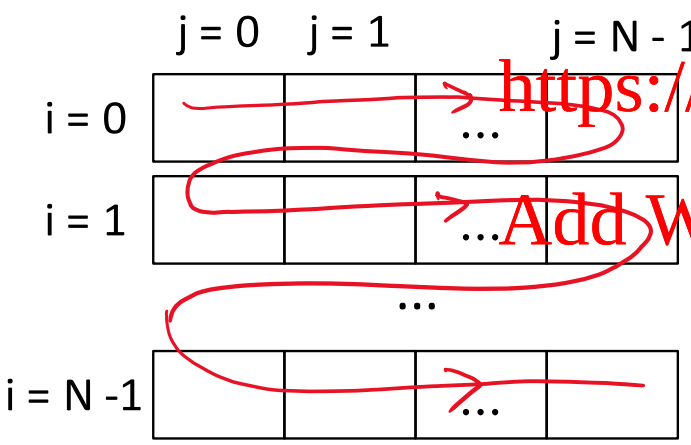
could be stored in two possible ways:

Address	Row-major order	Column-major order
0	a_{11}	a_{11}
1	a_{12}	a_{21}
2	a_{13}	a_{12}
3	a_{21}	a_{22}
4	a_{22}	a_{13}
5	a_{23}	a_{23}

Row major is
a common choice

Spatial Locality

Observation: Applications access data near to what they just accessed



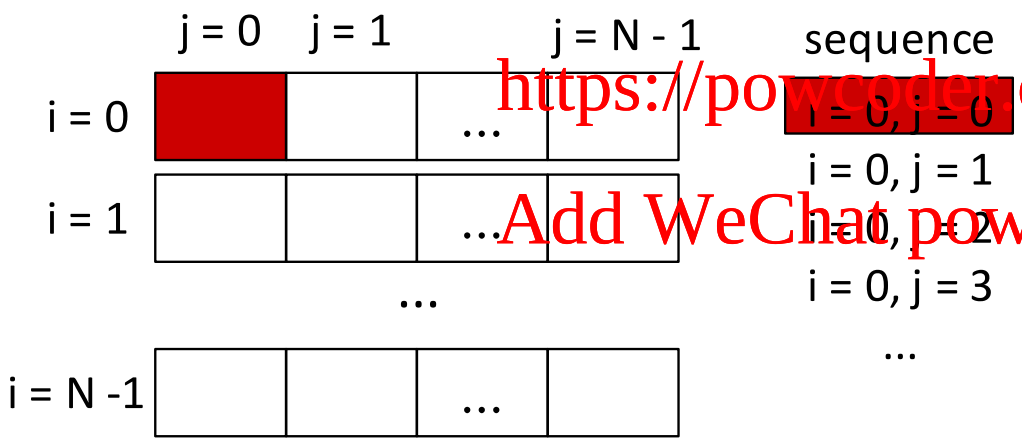
Access sequence

i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 0, j = 3
...

```
for(i=0; i< N; i++)
for(j = 0; j < N; j++ )
{
    count++;
    arrayInt[i][j] = 10;
}

printf("Count :%d\n", cou
}
```

Spatial Locality

Observation: Applications access data near to what they just accessed



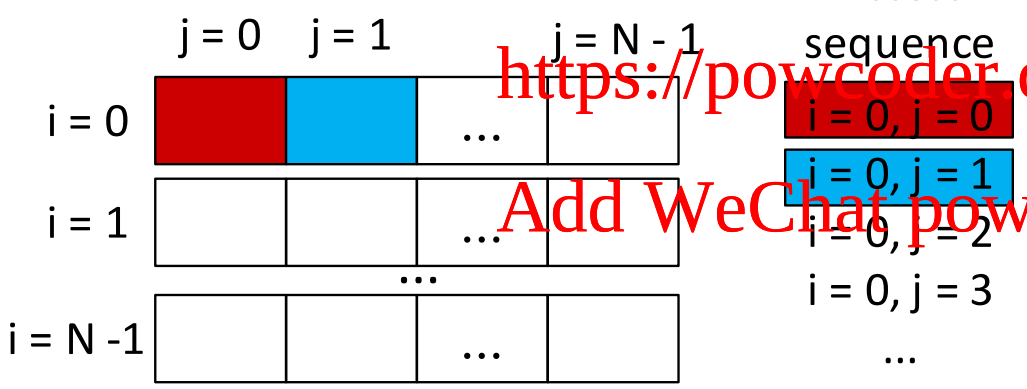
Access sequence

i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 0, j = 3
...

```
for(i=0; i< N; i++)
 for(j = 0; j < N; j++ )
 {
     count++;
     arrayInt[i][j] = 10;
 }

 printf("Count :%d\n", cou
}
```

Spatial Locality

Observation: Applications access data near to what they just accessed



Access sequence

i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 0, j = 3
...


```
for(i=0; i< N; i++)
for(j = 0; j < N; j++ )
{
    count++;
    arrayInt[i][j] = 10;
}

printf("Count :%d\n", cou
}
```

Applications have ***SPATIA***

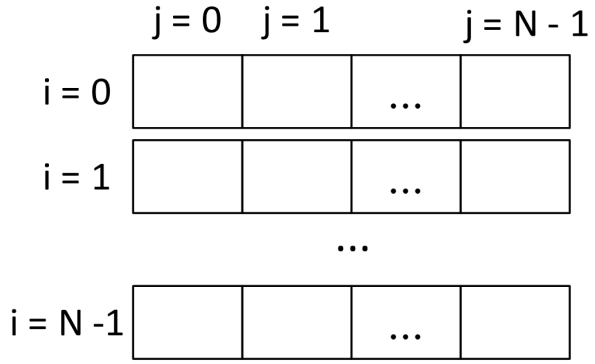
Importance of Cache on Performance:

Cache Aware code can be several times faster than non-Aware code

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

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
  int i, j;
  int count = 0;

  for(i=0; i< N; i++)
    for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[i][j] = 10;
    }

    printf("Count :%d\n", count);
}
```

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

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
  int i, j;
  int count = 0;

  for(i=0; i< N; i++)
    for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[j][i] = 10;
    }

    printf("Count :%d\n", count);
}
```

Live demo:
See L1_3_370_Course_Overview
Video at minute 18:00



Basic Cache organization

Decide on the block size

How? Simulate lots of different block sizes and see which one gives the best performance

Common cache block sizes: 32, 64, or 128 bytes

Larger block sizes reduce cache area overhead by:

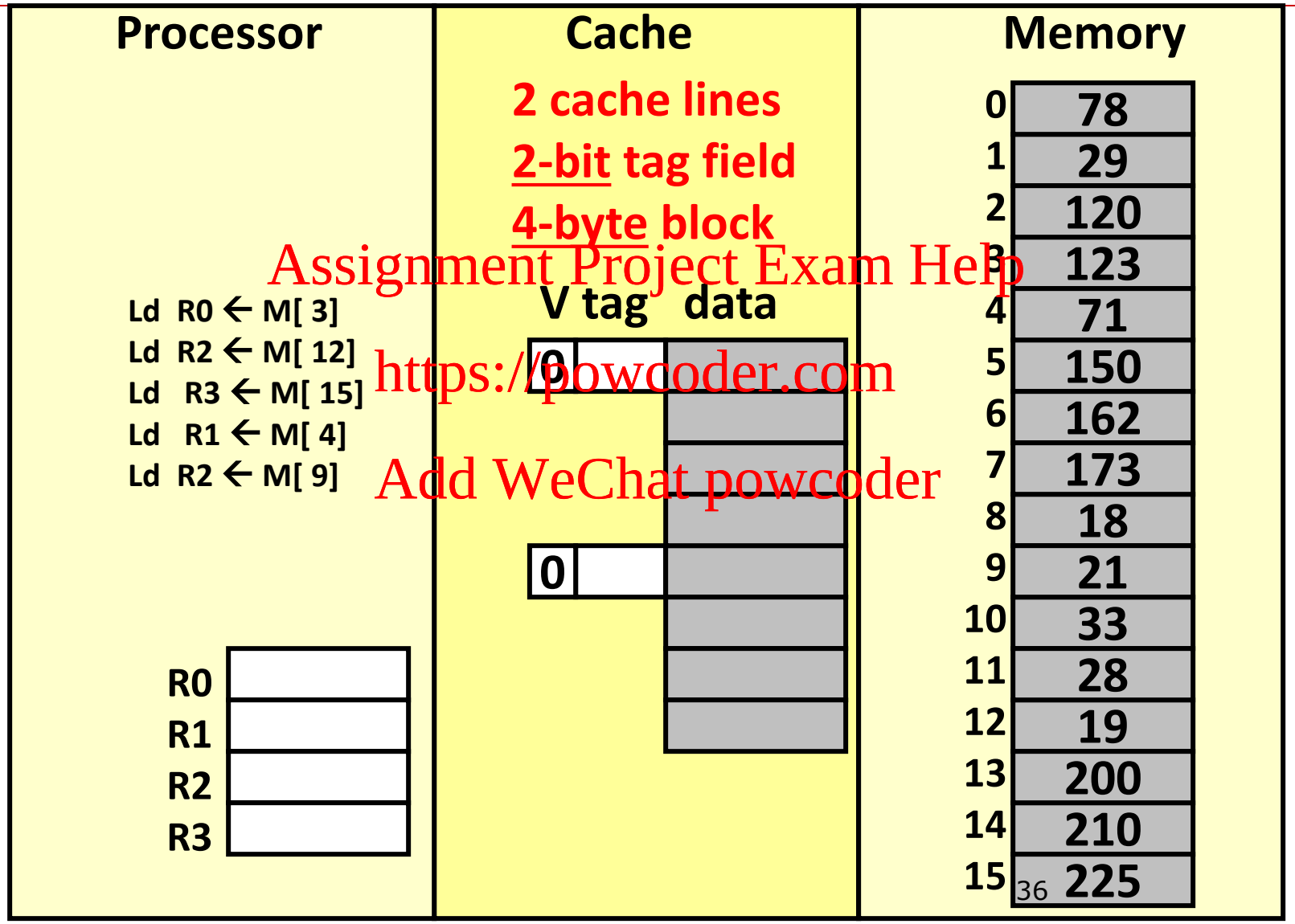
Reducing number of cache lines (therefore, number of tags and other meta-data)

Reducing each tag size

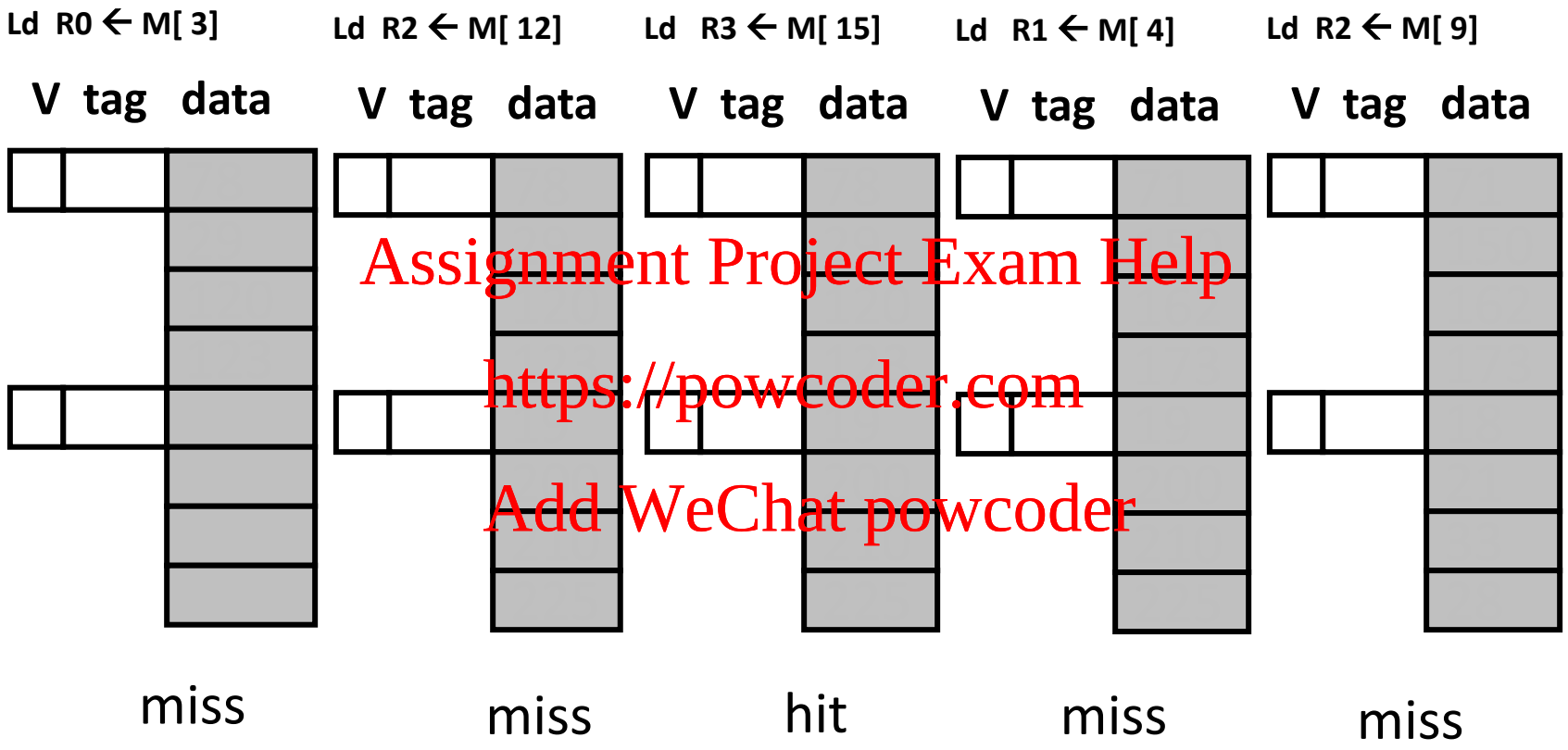
Block size = 2 ^ (block_offset) sizeof(block_offset) = log2(block_size)

Tag size = address_size – block_offset_size

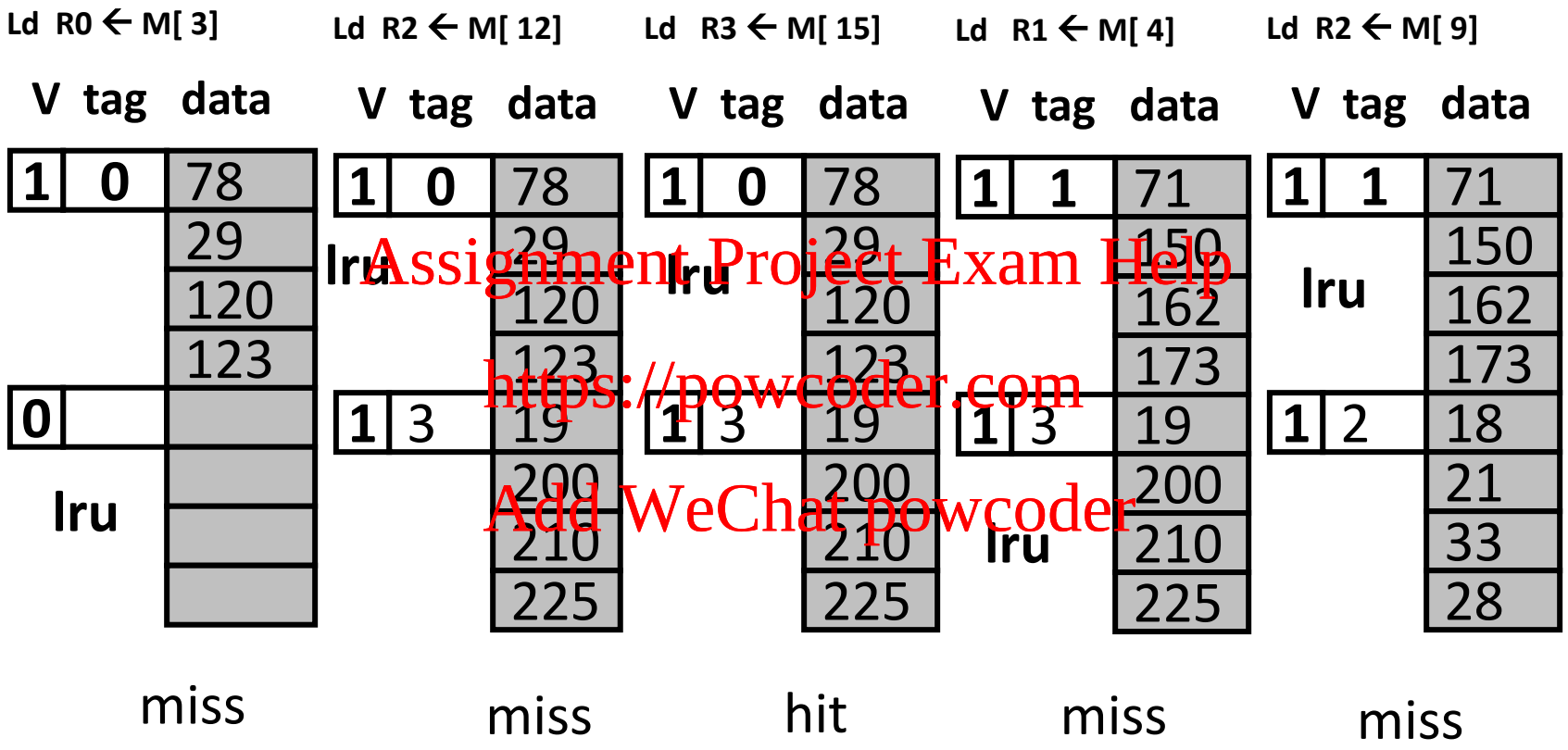
Practice Problem– Compute the register and cache state after executing the instruction sequence



Solution to Practice Problem



Solution to Practice Problem



Class Problem 1

Given a cache with the following configuration: cache size is 8 bytes, block size is 2 bytes*, fully associative*, LRU replacement. The memory address size is 16 bits and is byte addressable.

1. How many bits are for each tag? How many blocks in the cache?

2. For the following reference stream, indicate whether each reference is a hit or miss: 0, 1, 3, 5, 12, 1, 2, 9, 4

3. What is the hit rate?

4. How many bits are needed for storage overhead for each block?

Class Problem 1

Given a cache with the following configuration: cache size is 8 bytes, block size is 2 bytes, *fully associative*, LRU replacement. The memory address size is 16 bits and is byte addressable. Assume 2 bits per cache line to implement LRU.

1. How many bits are for each tag? How many blocks in the cache?

   Tag = 16 – 1 (block offset) = 15 bits,

   2 byte blocks, 8 bytes total = 4 blocks.

2. For the following reference stream, indicate whether each reference is a hit or miss: 0, 1, 3, 5, 12, 1, 2, 9, 4

   M,H,M,M,M,H,H,M,M

3. What is the hit rate? 3/9 = 33 %

4. How many bits are needed for storage overhead for each block?

   Overhead = 15 (Tag) + 1 (V) + 2 (LRU) = 18 bits

   We are assuming 2 bits per cache line to store the LRU rank.

   More efficient solutions for LRU exists: log2(#cache_lines!) = log2(4!) = 5 bits

Class Problem 2—Storage overhead

Consider the following cache:

32-bit byte addressable ISA

Cache size : 64KB (kilo-bytes) = 64 * 8 kilo-bits = 512 Kilo-bits

Cache block size : 64 Bytes

Write-allocate, write-back, *fully associative*

Recall: 1 kilobyte = 1024 bytes (NOT 1000 bytes!)

What is the cache area overhead (tags, valid, dirty, LRU)?

Class Problem 2—Storage overhead

Consider the following cache:

32-bit byte addressable ISA

Cache size : 64KB (kilo-bytes) = 64 * 8 kilo-bits = 512 Kilo-bits

Cache block size : 64 Bytes

Write-allocate, write-back, *fully associative*

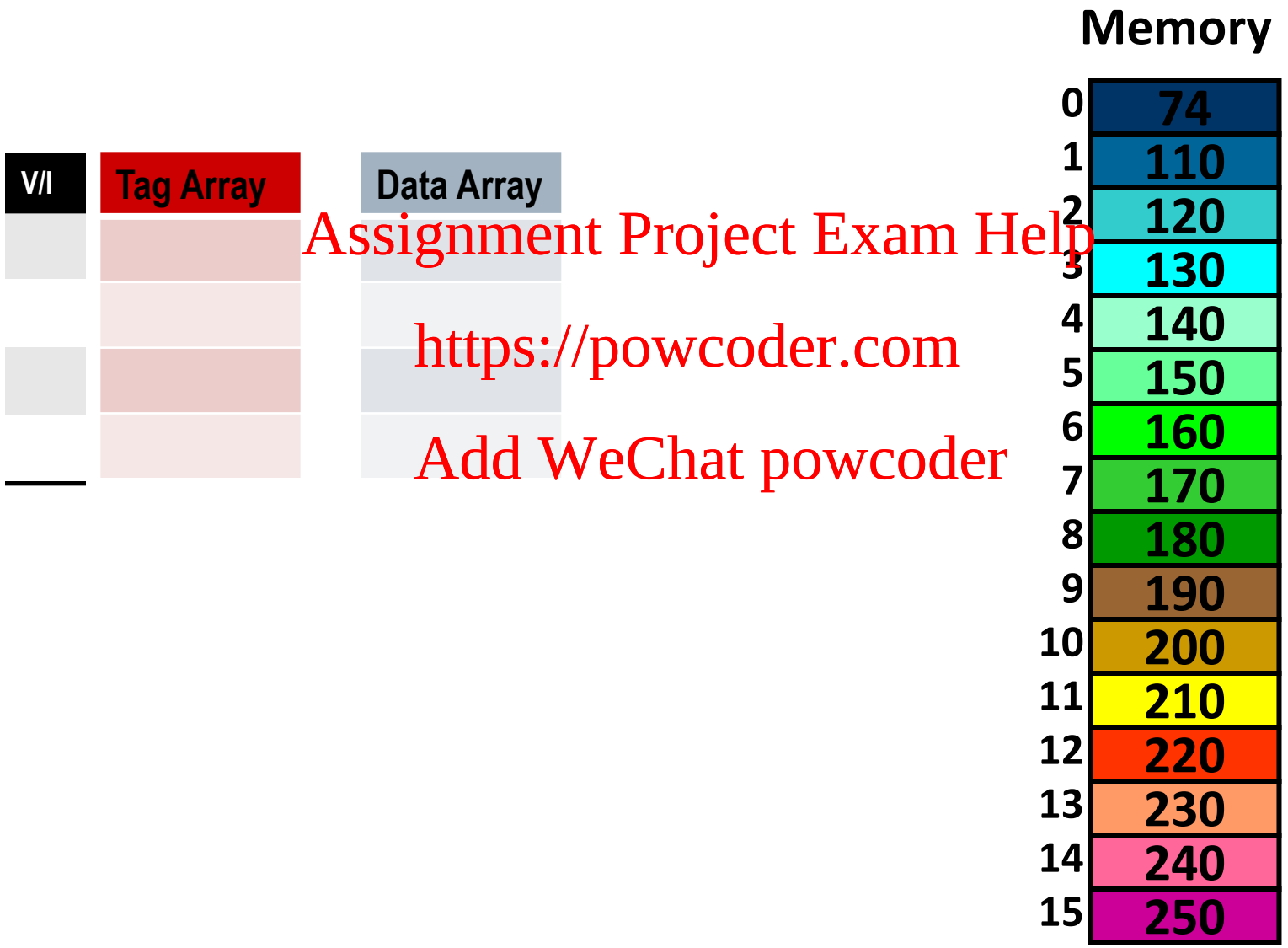
Recall: 1 kilobyte = 1024 bytes (NOT 1000 bytes!)

What is the cache area overhead (tags, valid, dirty, LRU)? Assume log(#blocks) bits for LRU.

Tag	= 32 (Address) – 6 (block offset)	= 26 bits
#blocks	= 64K / 64	= 1024
LRU bits	= log(#blocks)	= 10 bits
Overhead per block	= 26(Tag) + 1(V) + 1(D) + 10 (LRU)	= 38 bits
Total overhead for cache	= 38 bits * #blocks = 38 * 1024	= 38912 bits

Handling Stores:

write-through vs write-back caches



What about stores?

Where should you write the result of a store to an address X?

If address X is in the cache

Write to the cache.

Should we also write to memory?

(yes - write-through policy)

(no - write-back policy – write only when modified cache block is evicted)

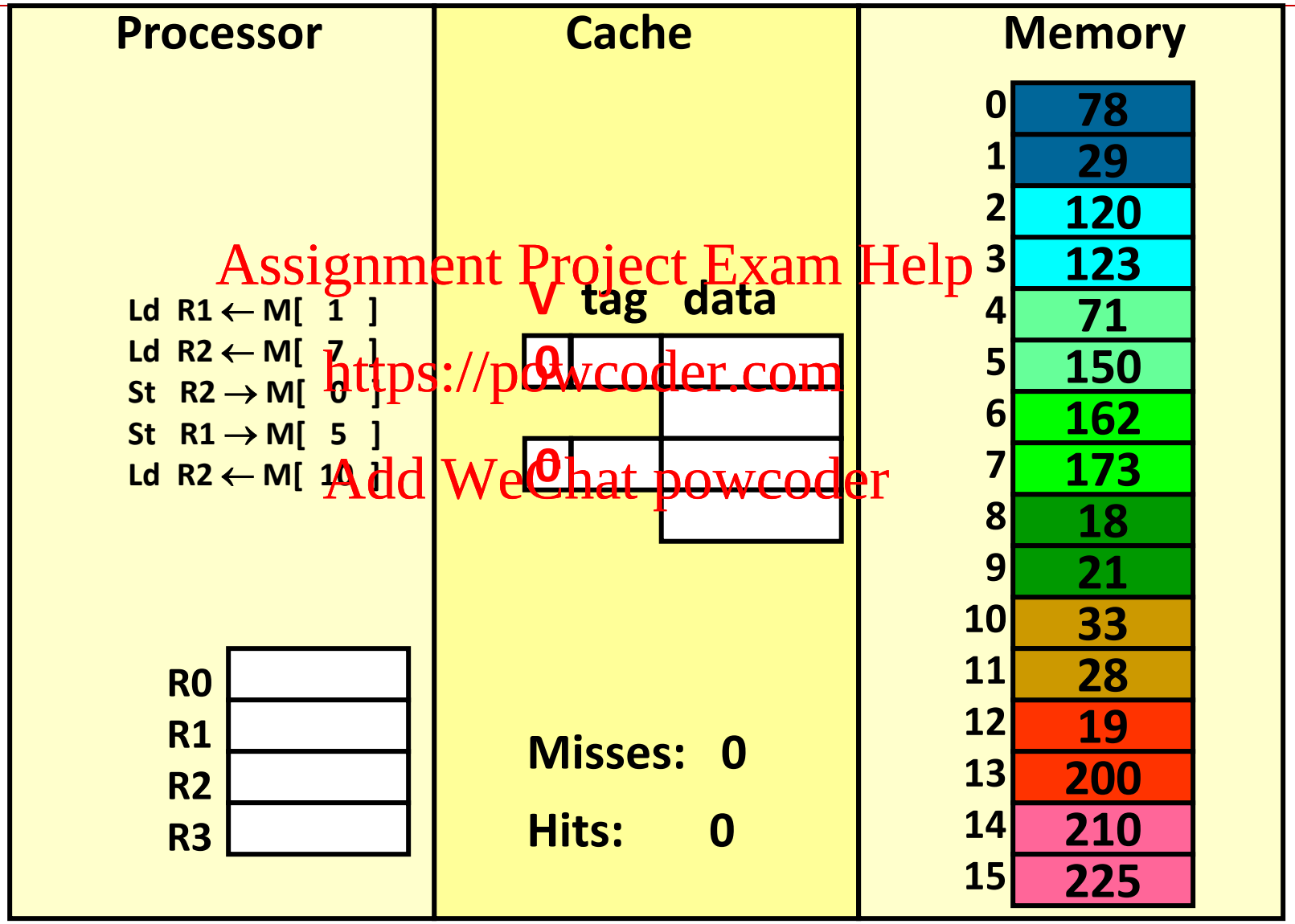
If address X is not in the cache

Allocate address in the cache?

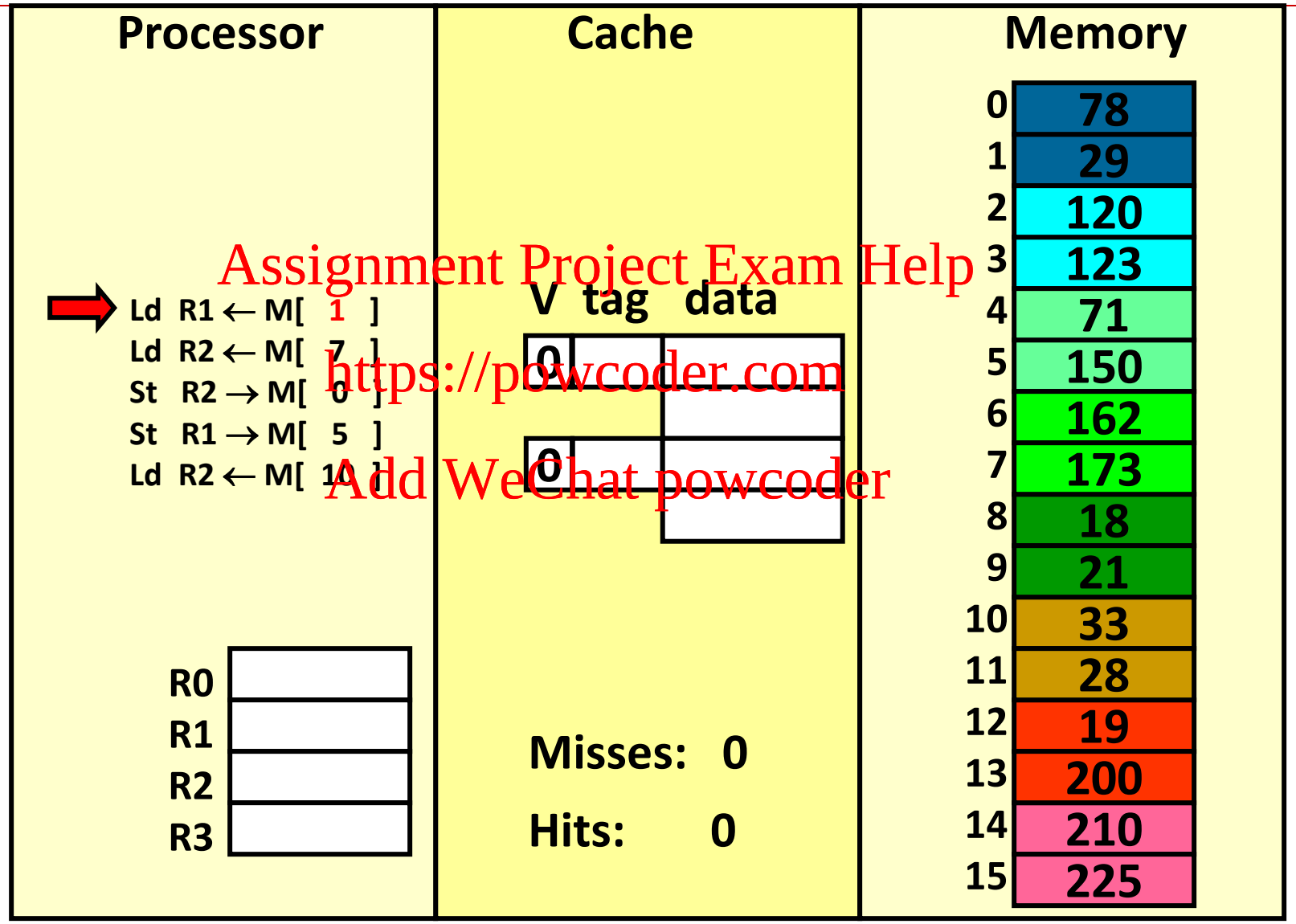
yes - allocate-on-write policy

no - directly write to memory, no allocate-on-write policy

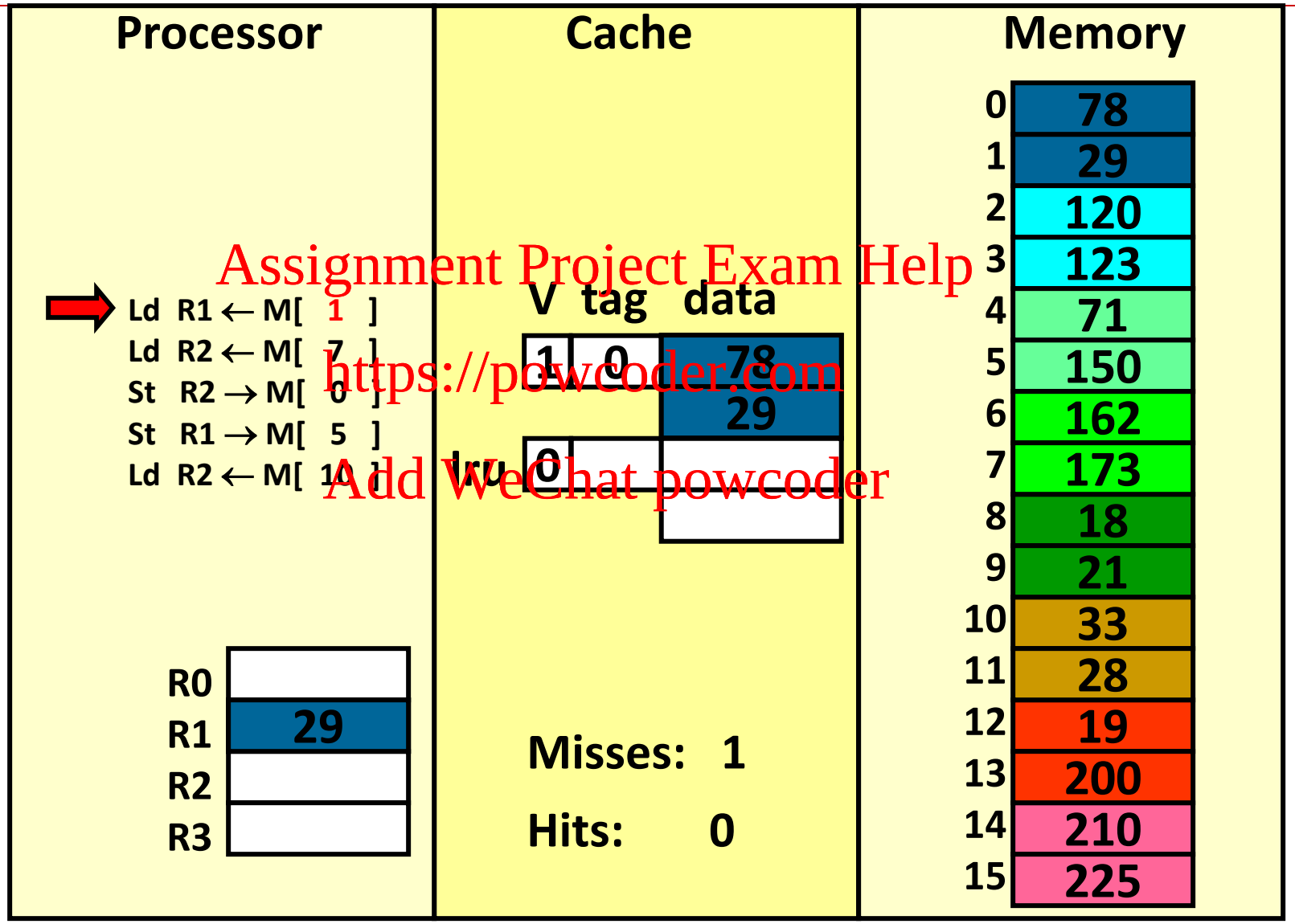
Handling stores (write-through)



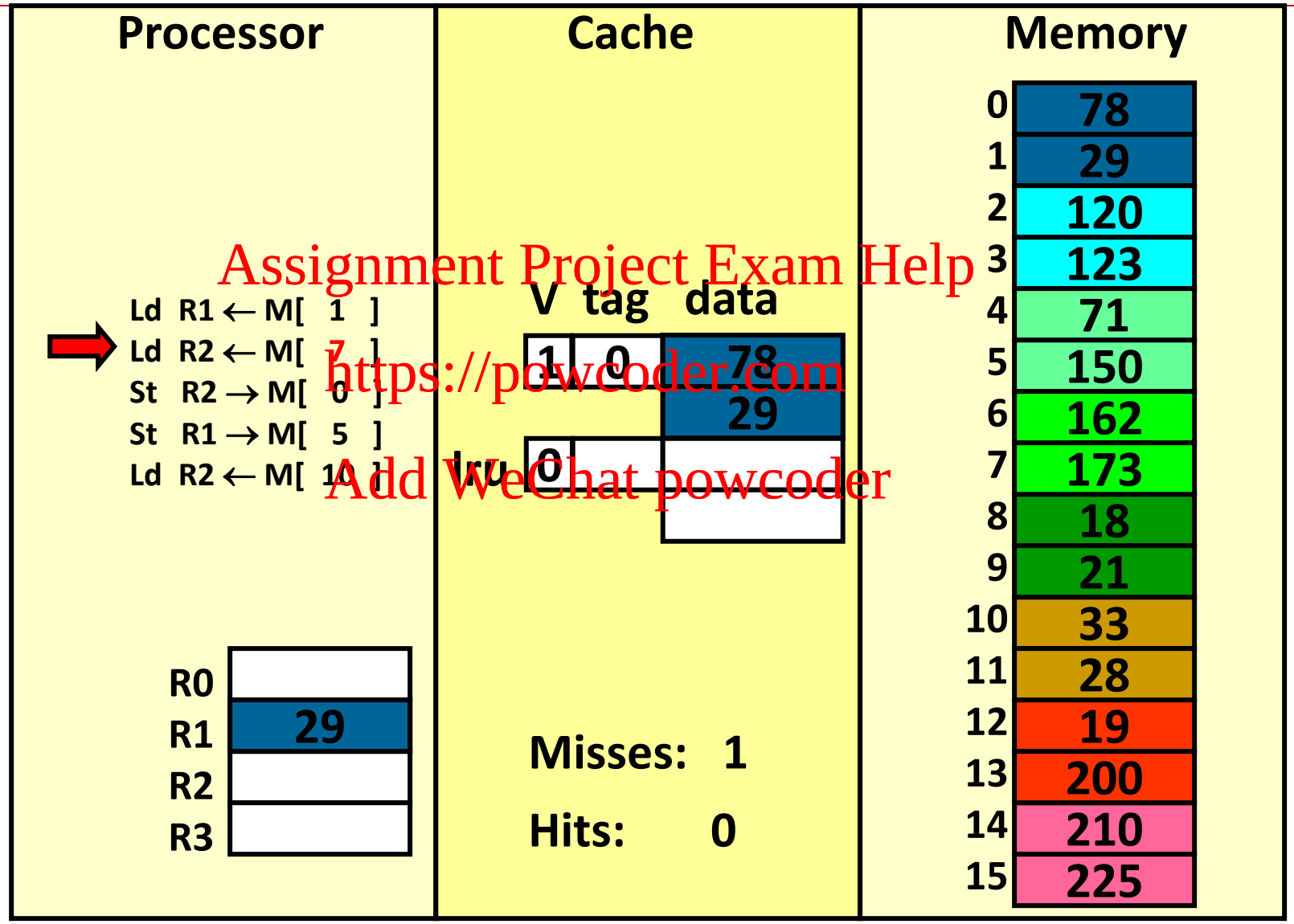
write-through (REF 1)



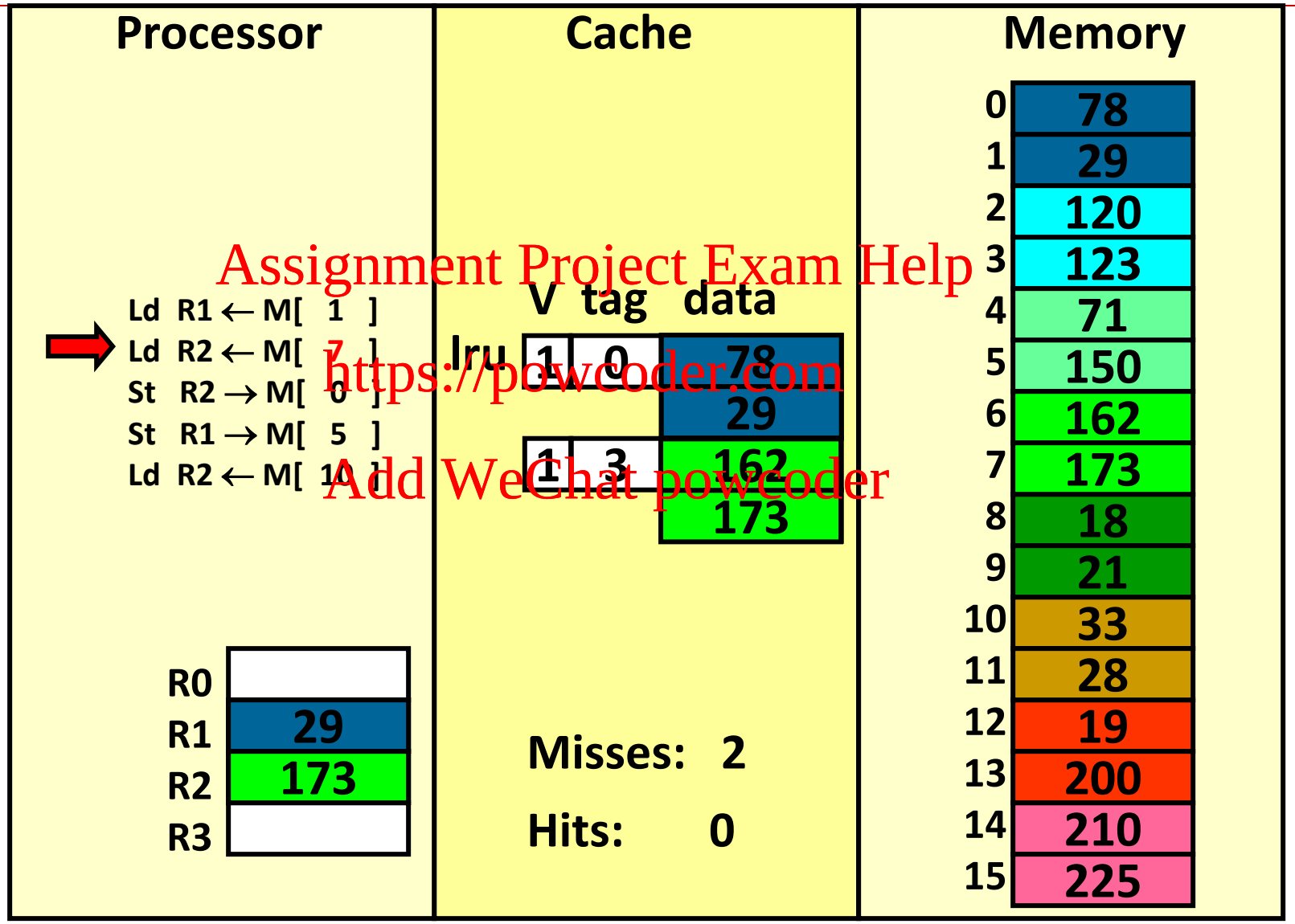
write-through (REF 1)



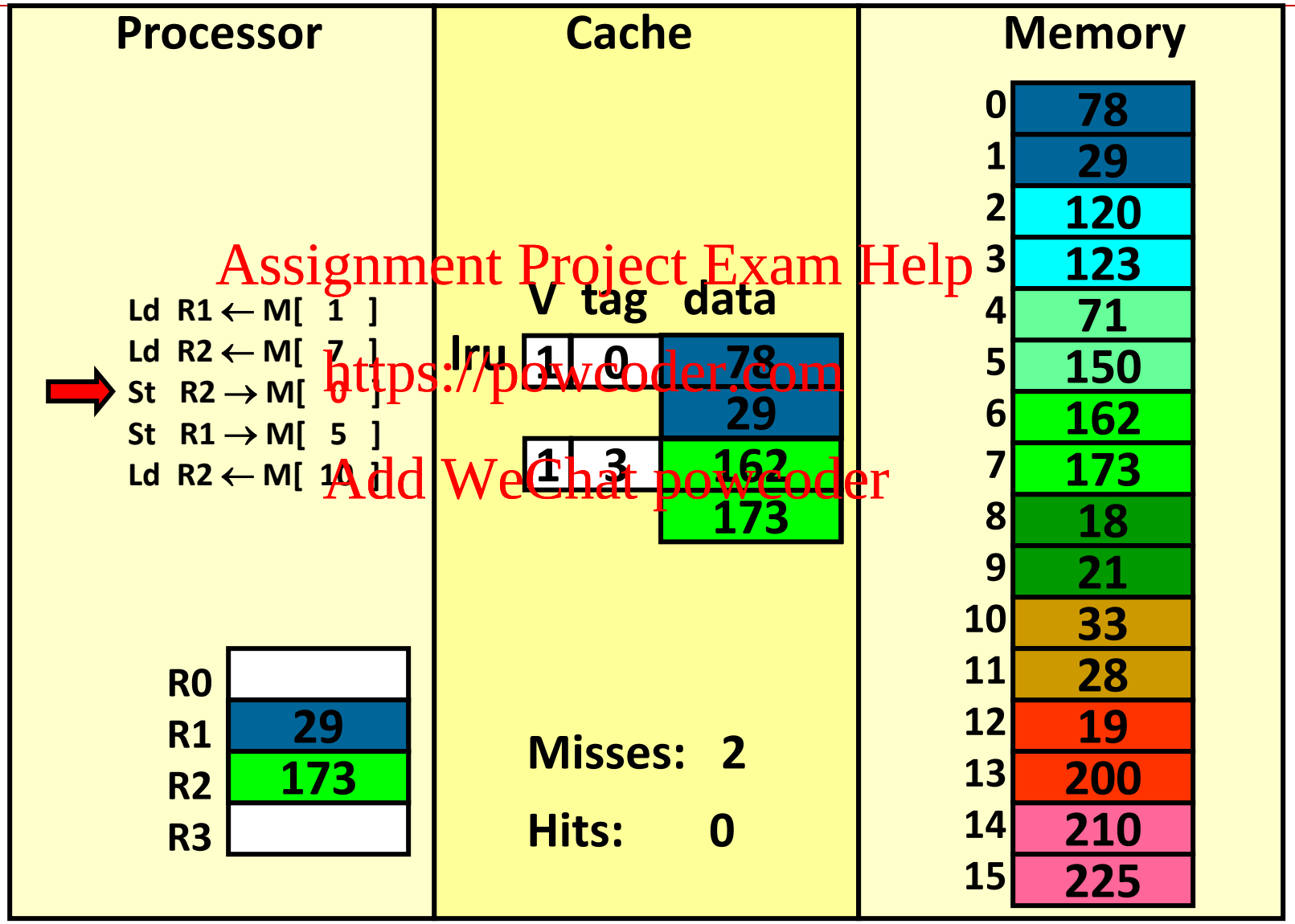
write-through (REF 2)



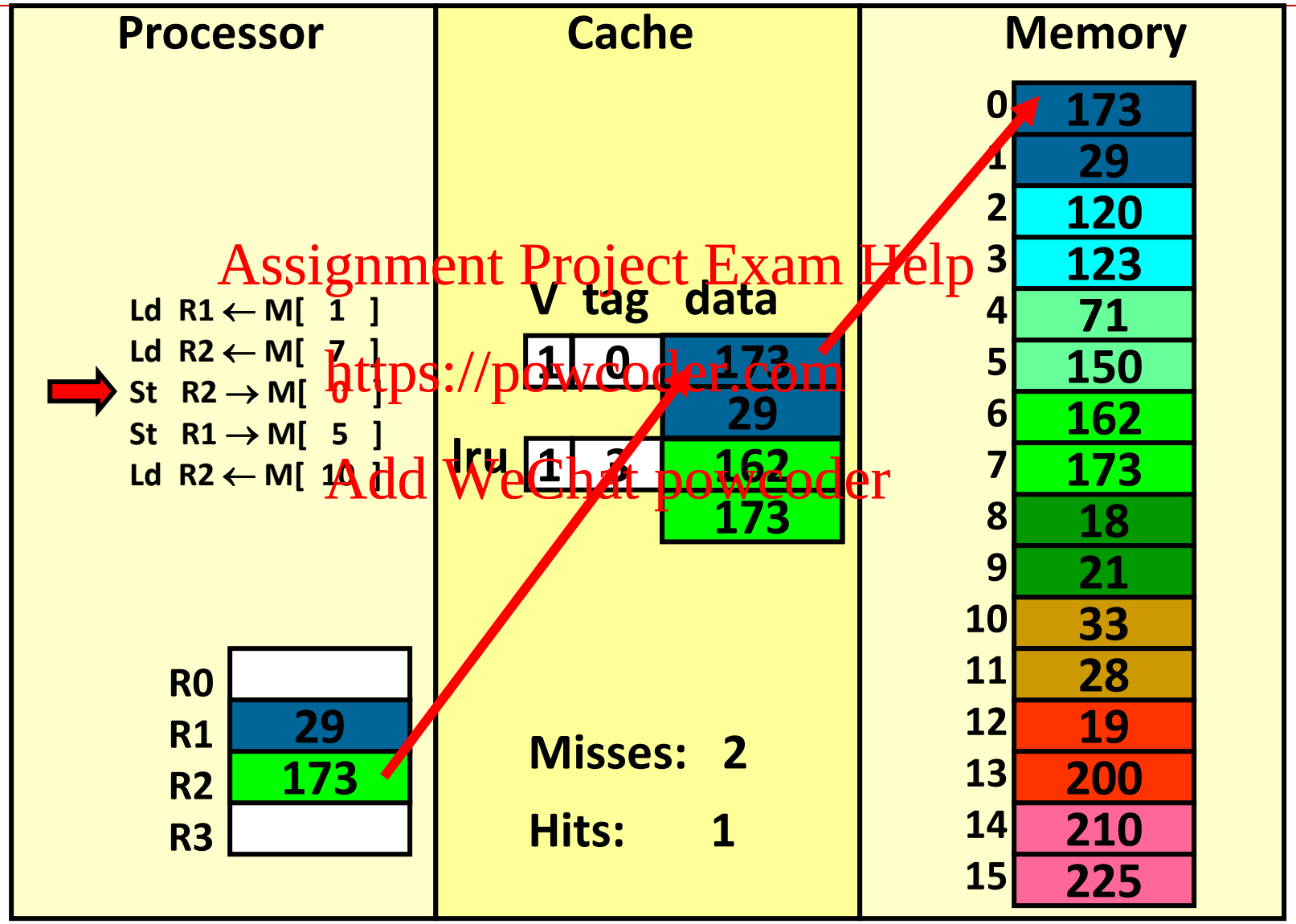
write-through (REF 2)



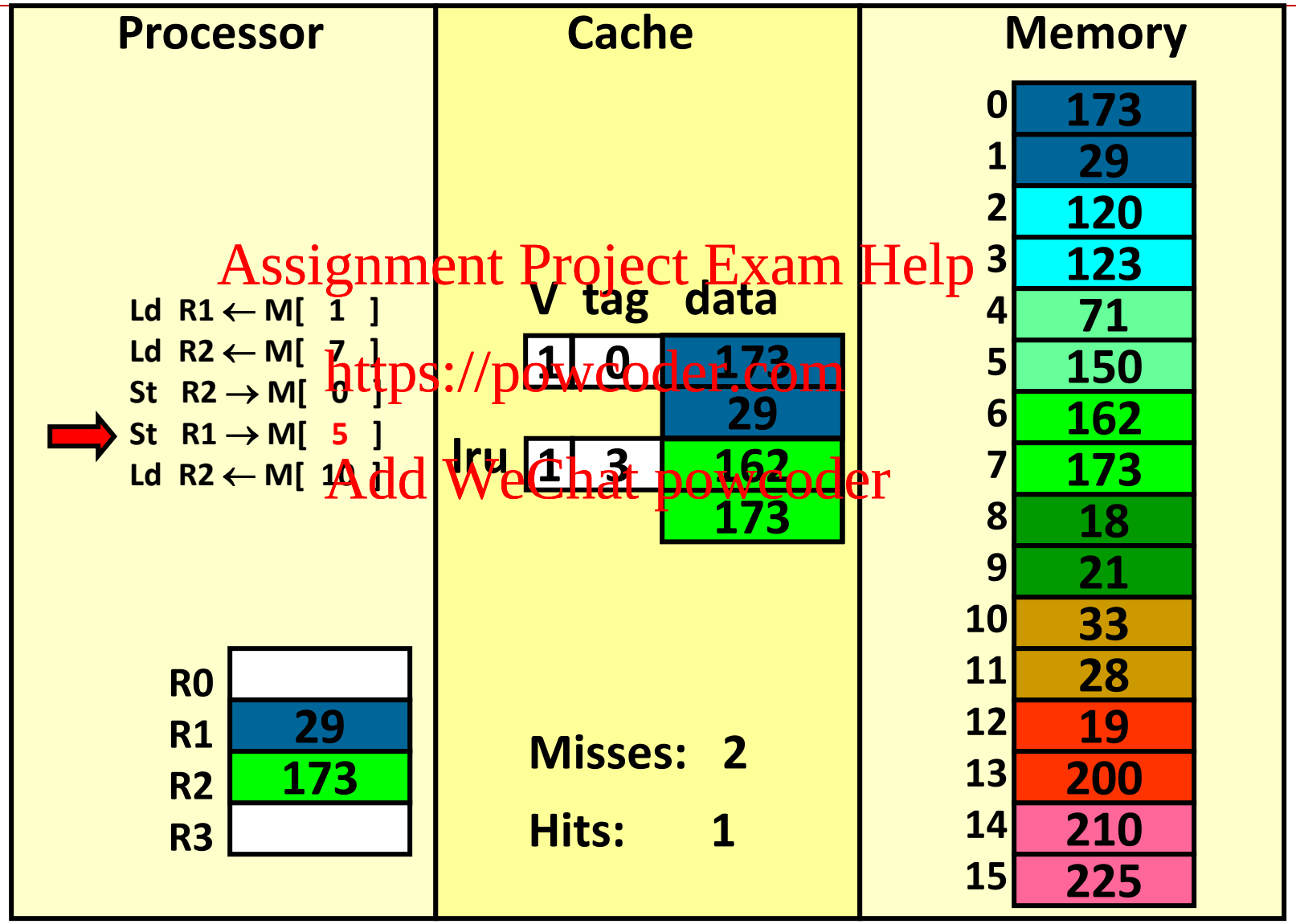
write-through (REF 3)



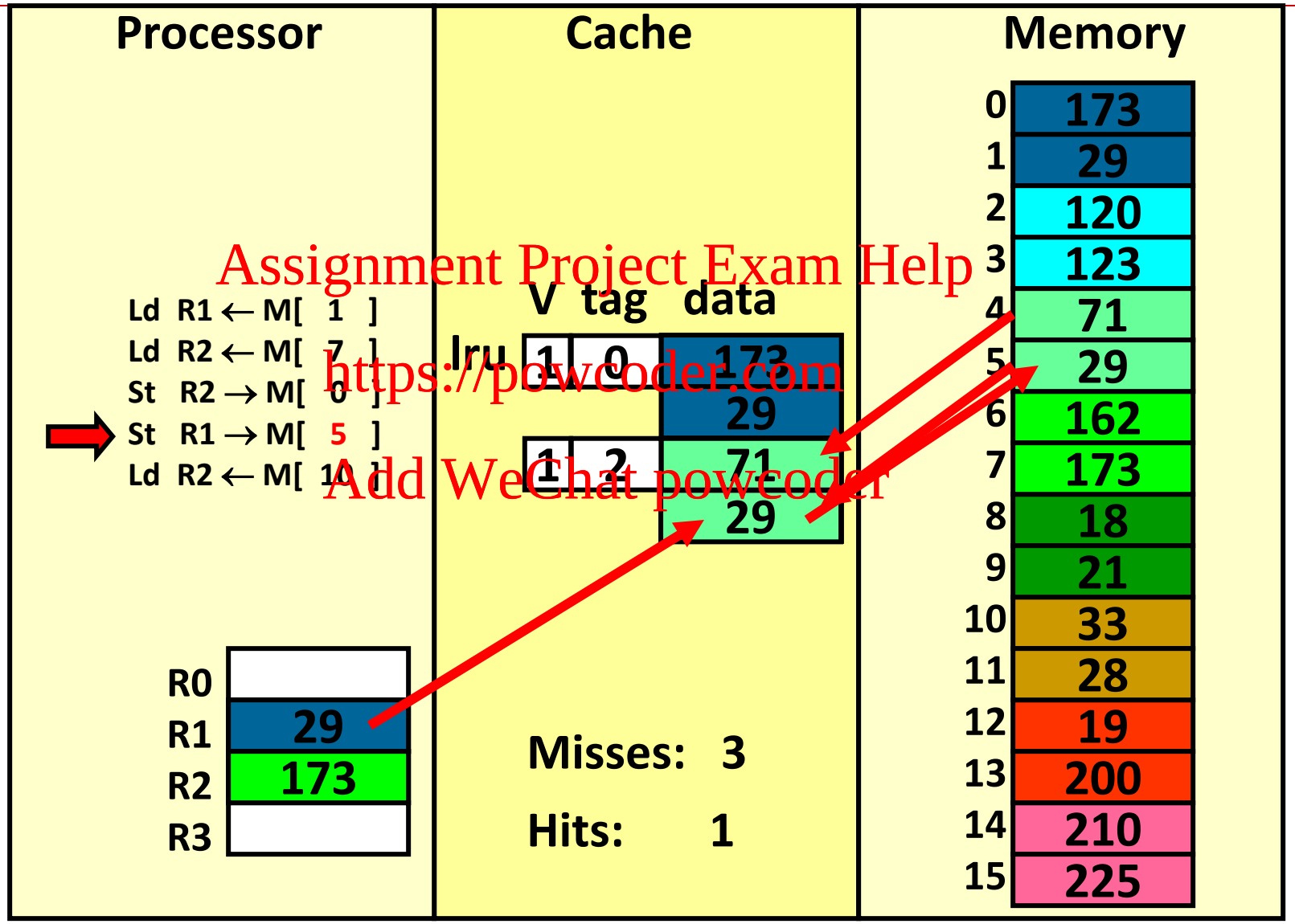
write-through (REF 3)



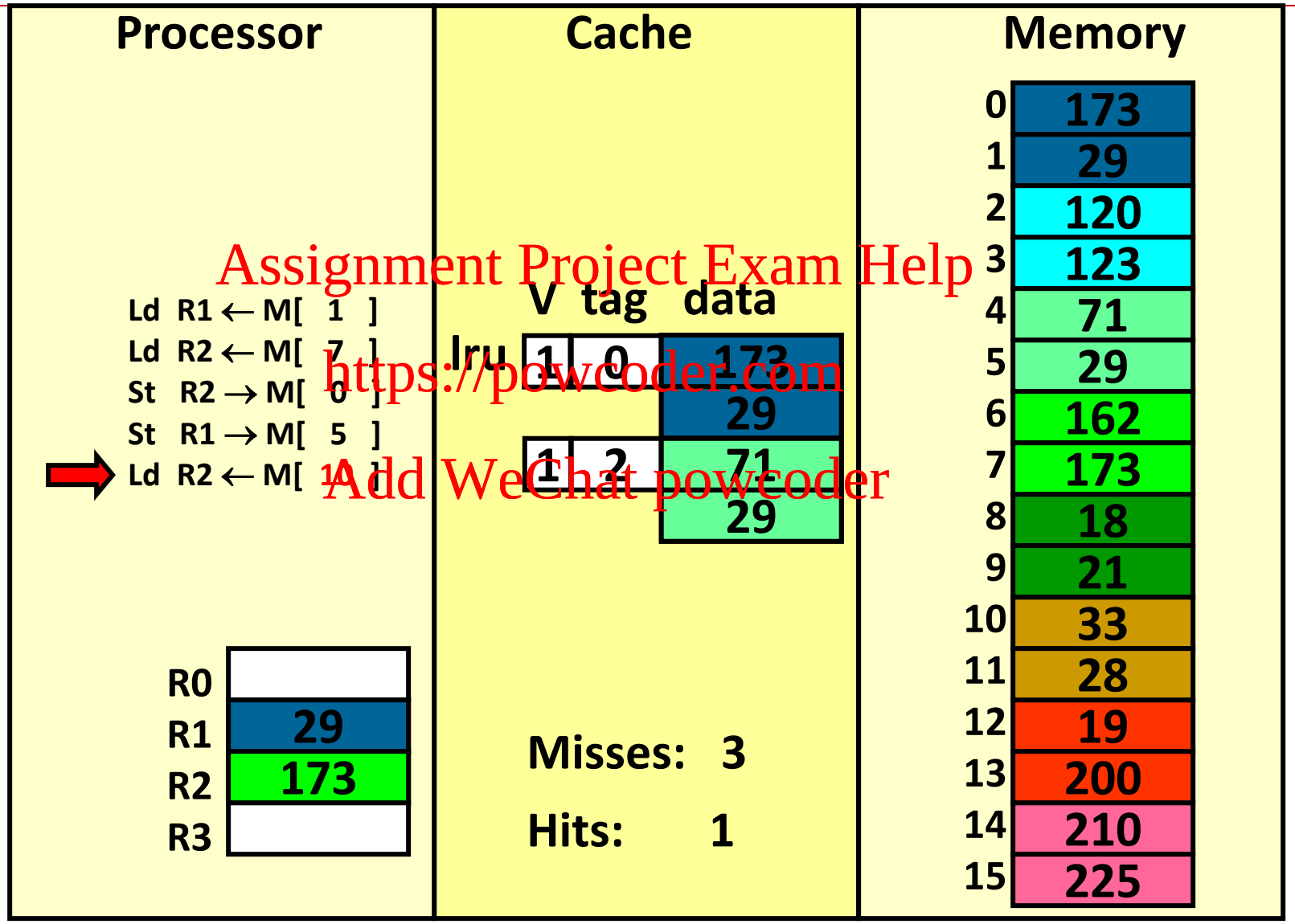
write-through (REF 4)



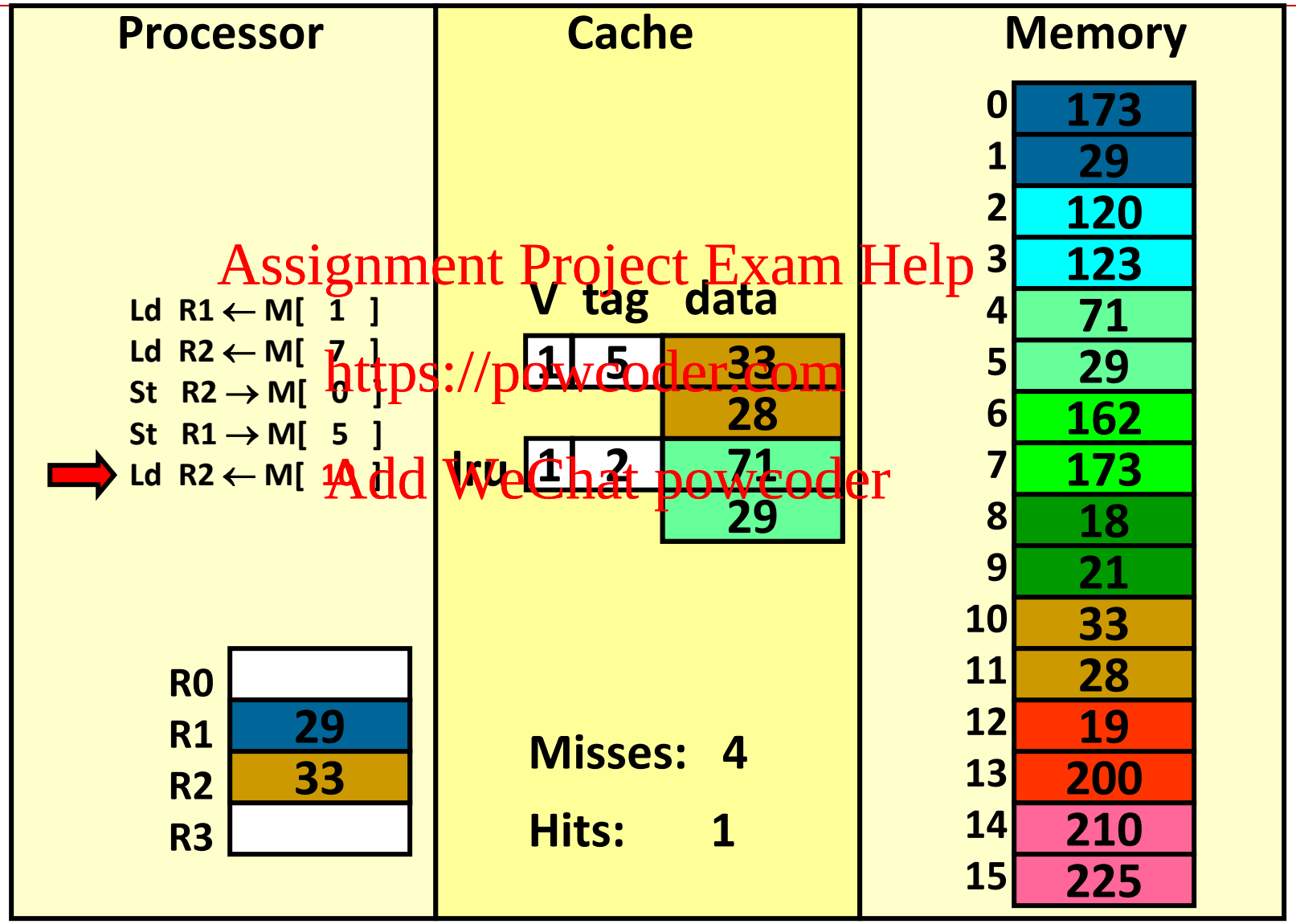
write-through (REF 4)



write-through (REF 6)



write-through (REF 6)



How many reads/writes to main memory in a write-through cache?

Each cache miss reads a block (2 bytes in our example) from main memory

Total reads from main memory: 8 bytes

Each store writes a byte to main memory

Total writes to main memory: 2 bytes

But, cache miss rate < 20%.

total main memory reads due to cache misses << main memory writes in a write-through cache

Write-through vs. write-back

Can we also design the cache to NOT write all stores to memory immediately?

We can keep the most recent copy in the cache and update the memory **only when** that data is evicted from the cache (a write-back policy).

Do we need to write-back all evicted cache blocks?

No, only blocks that have been stored into

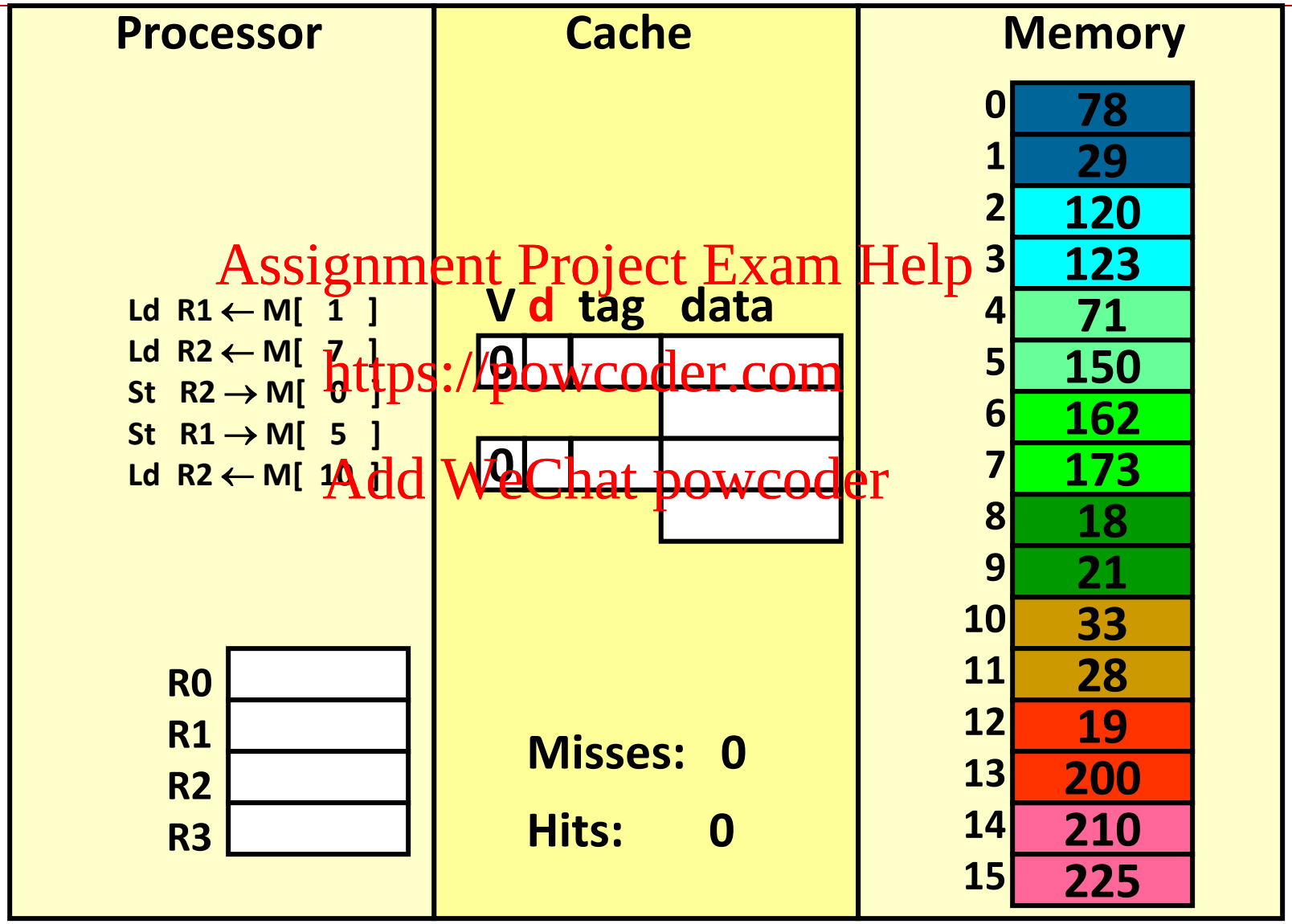
Keep a "dirty bit":

reset when the line is allocated

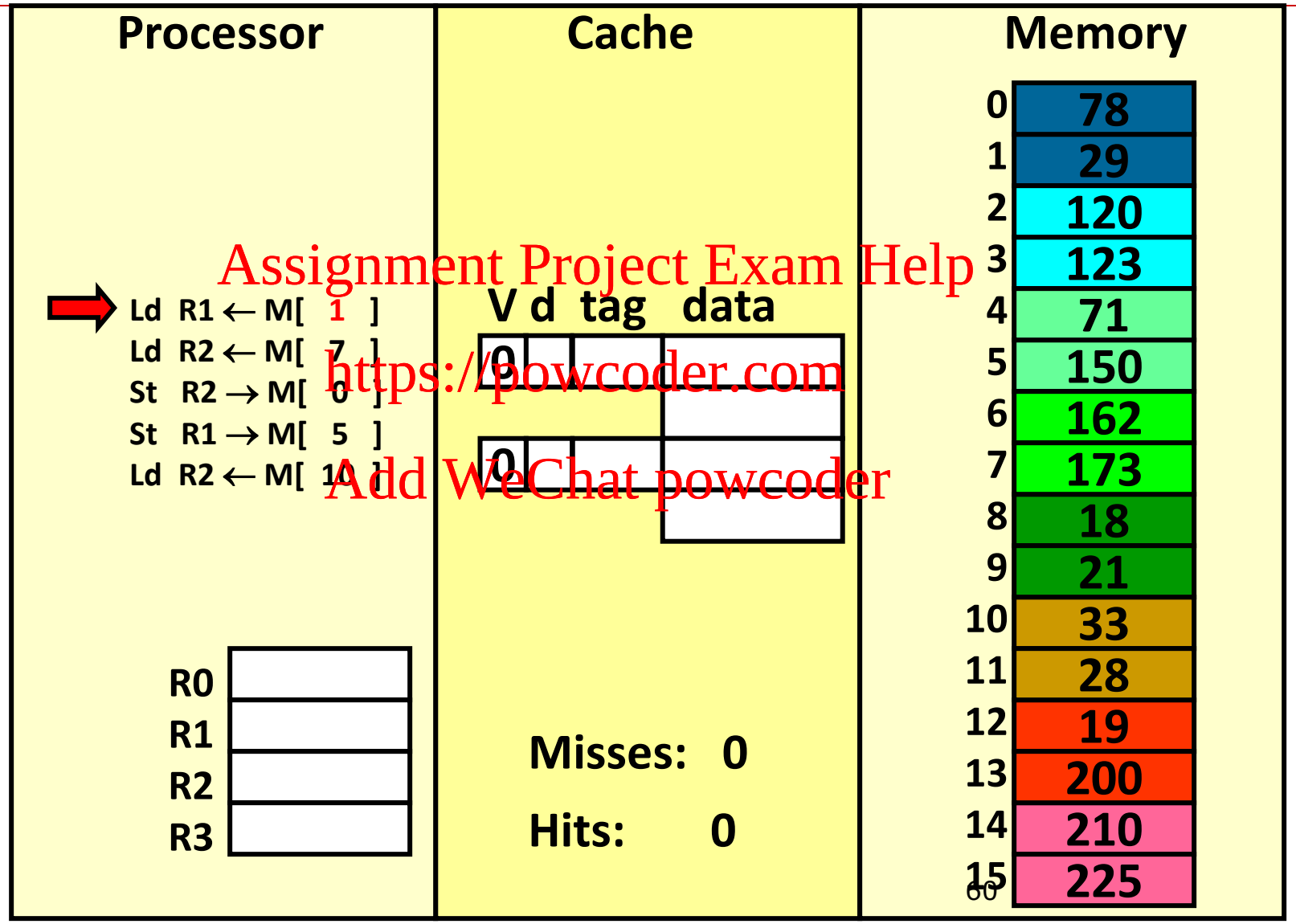
set when the block is stored into

If a block is "dirty" when evicted, write its data back into memory.

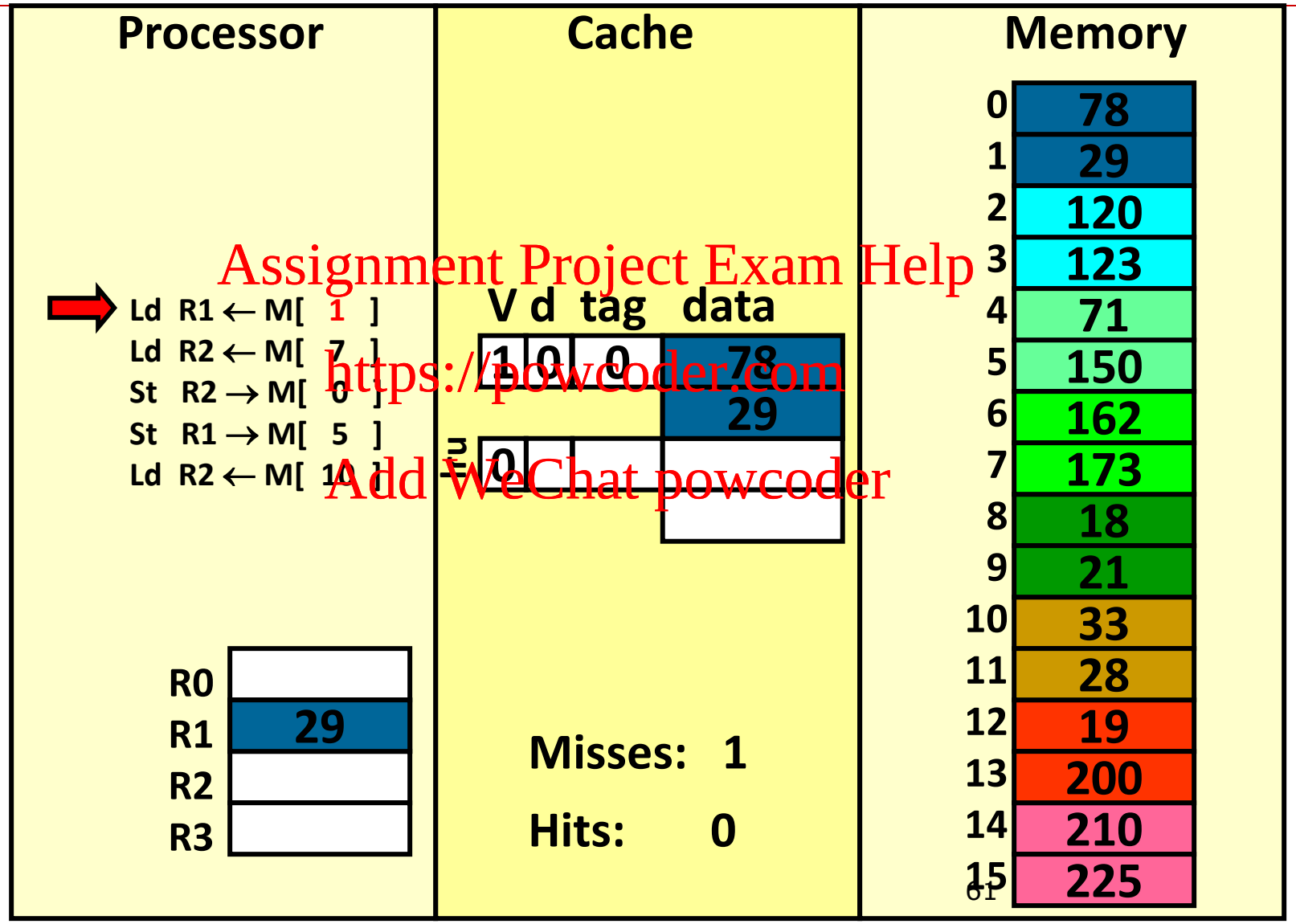
Handling stores (write-back)



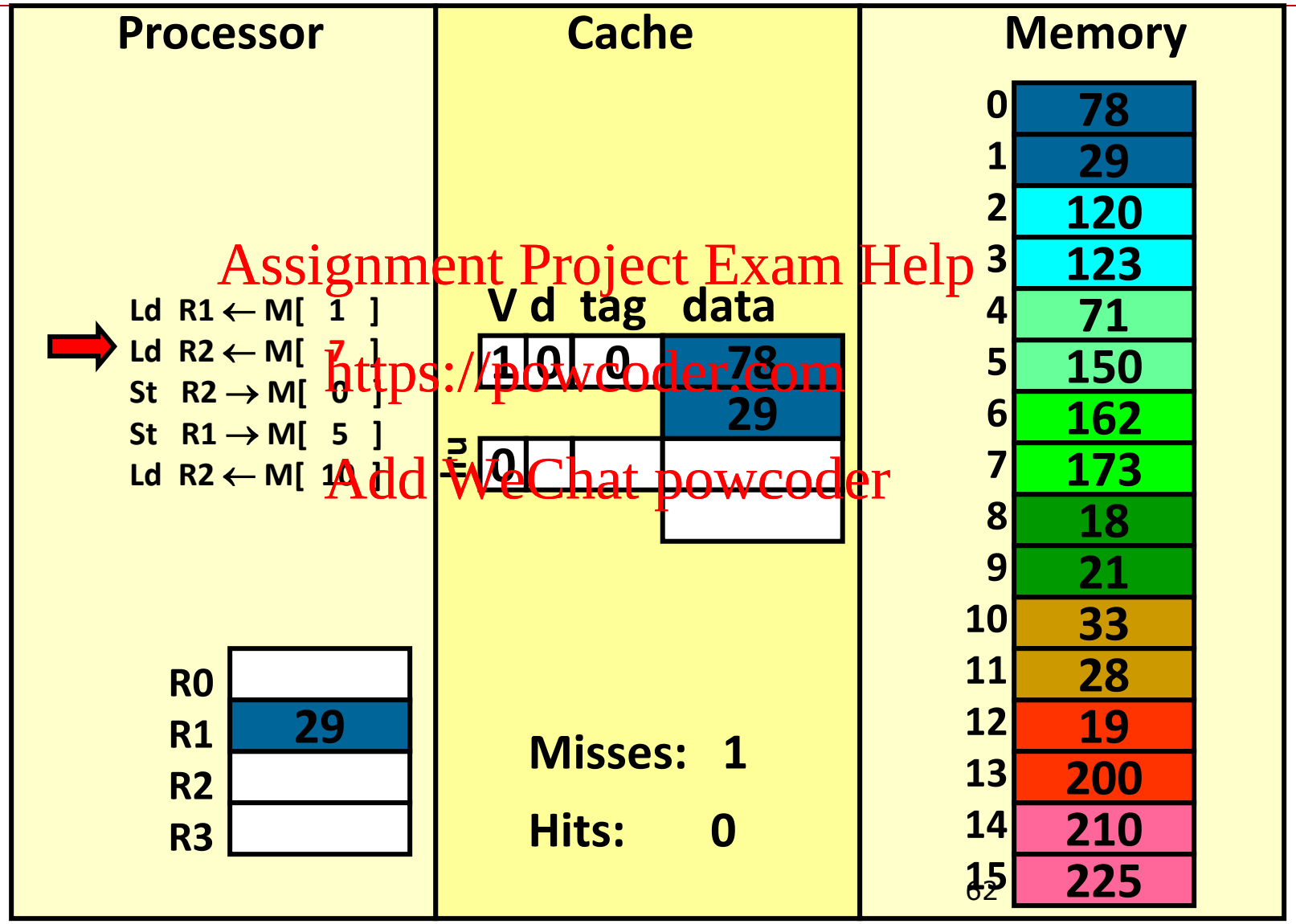
write-back (REF 1)



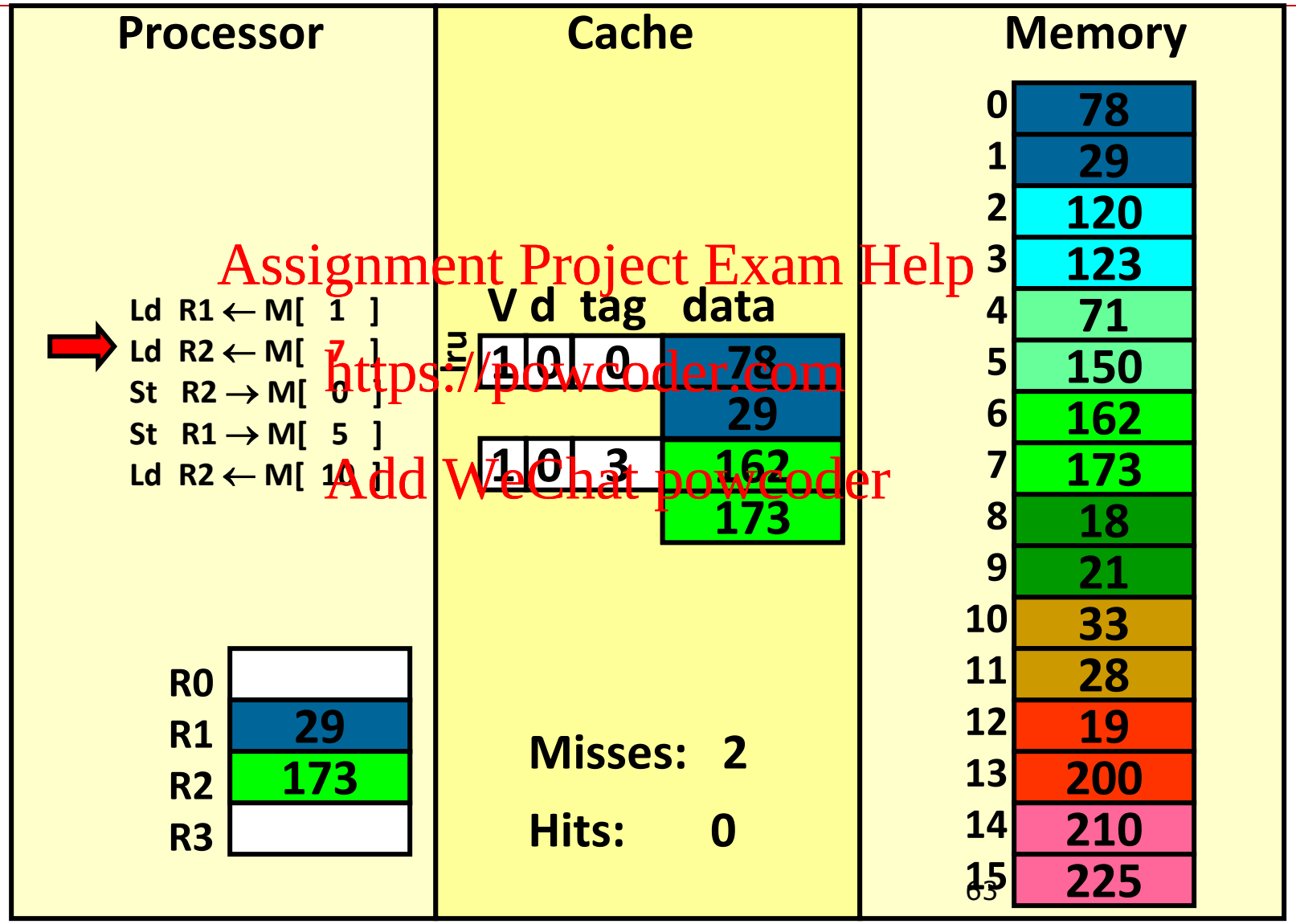
write-back (REF 1)



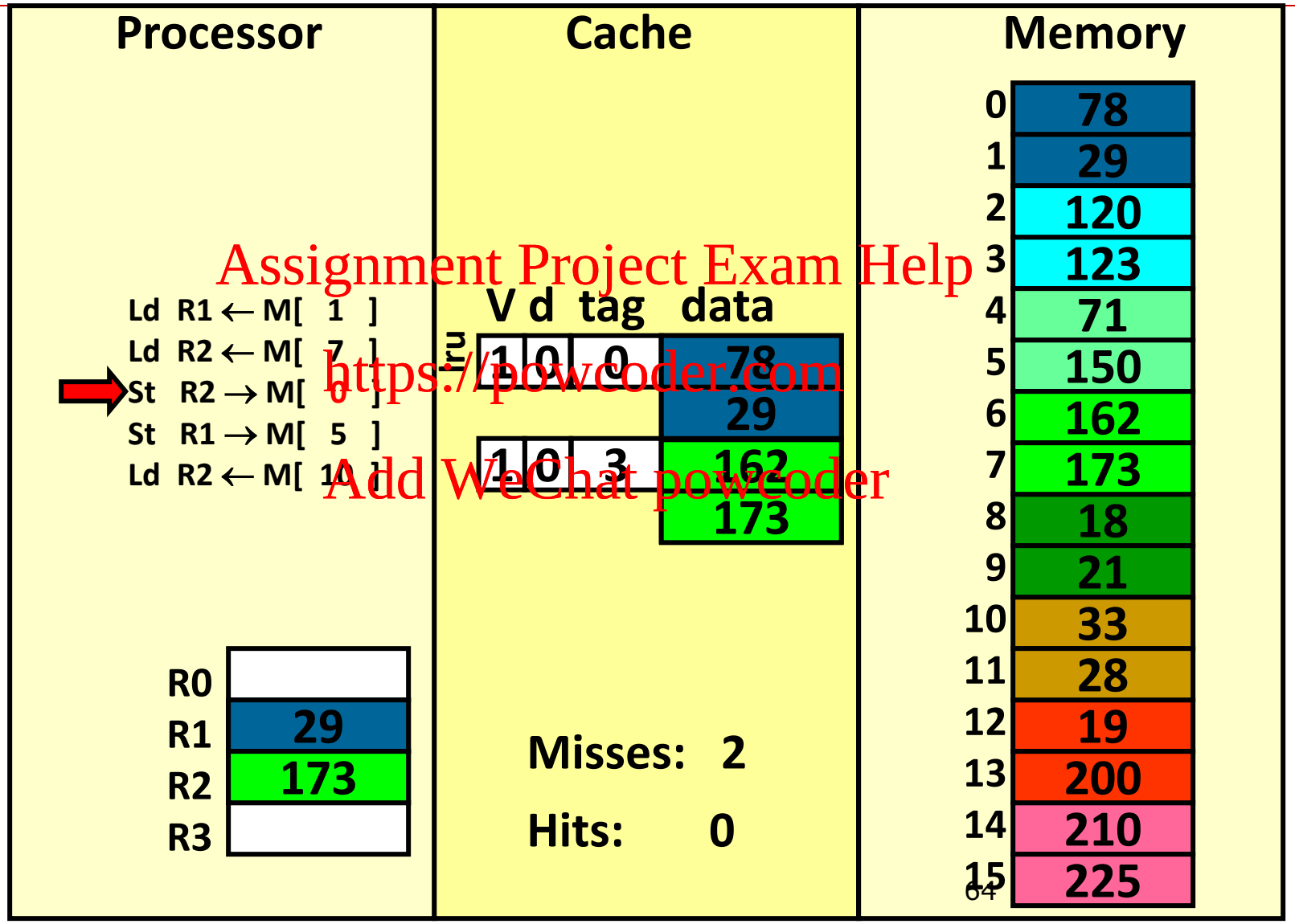
write-back (REF 2)



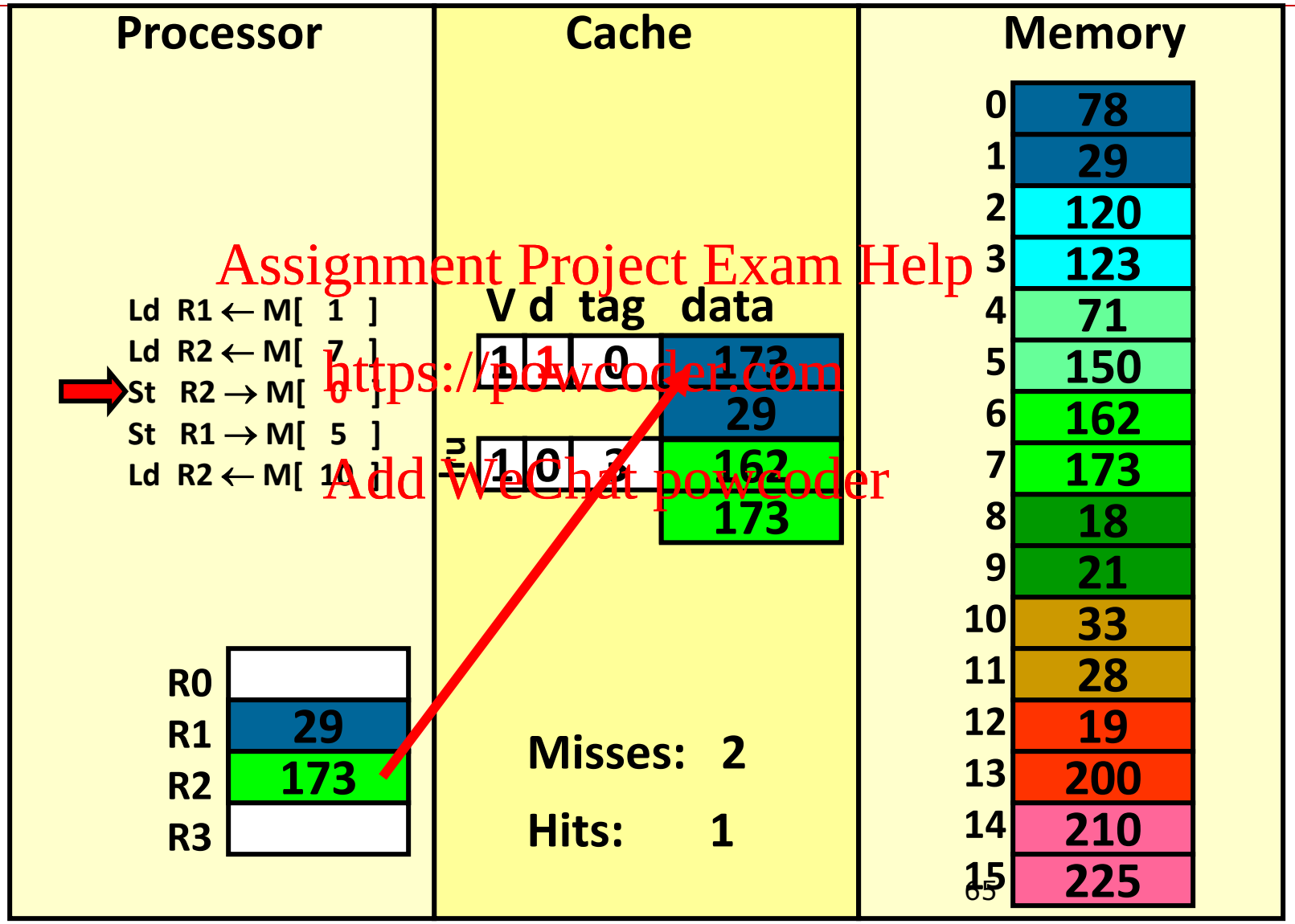
write-back (REF 2)



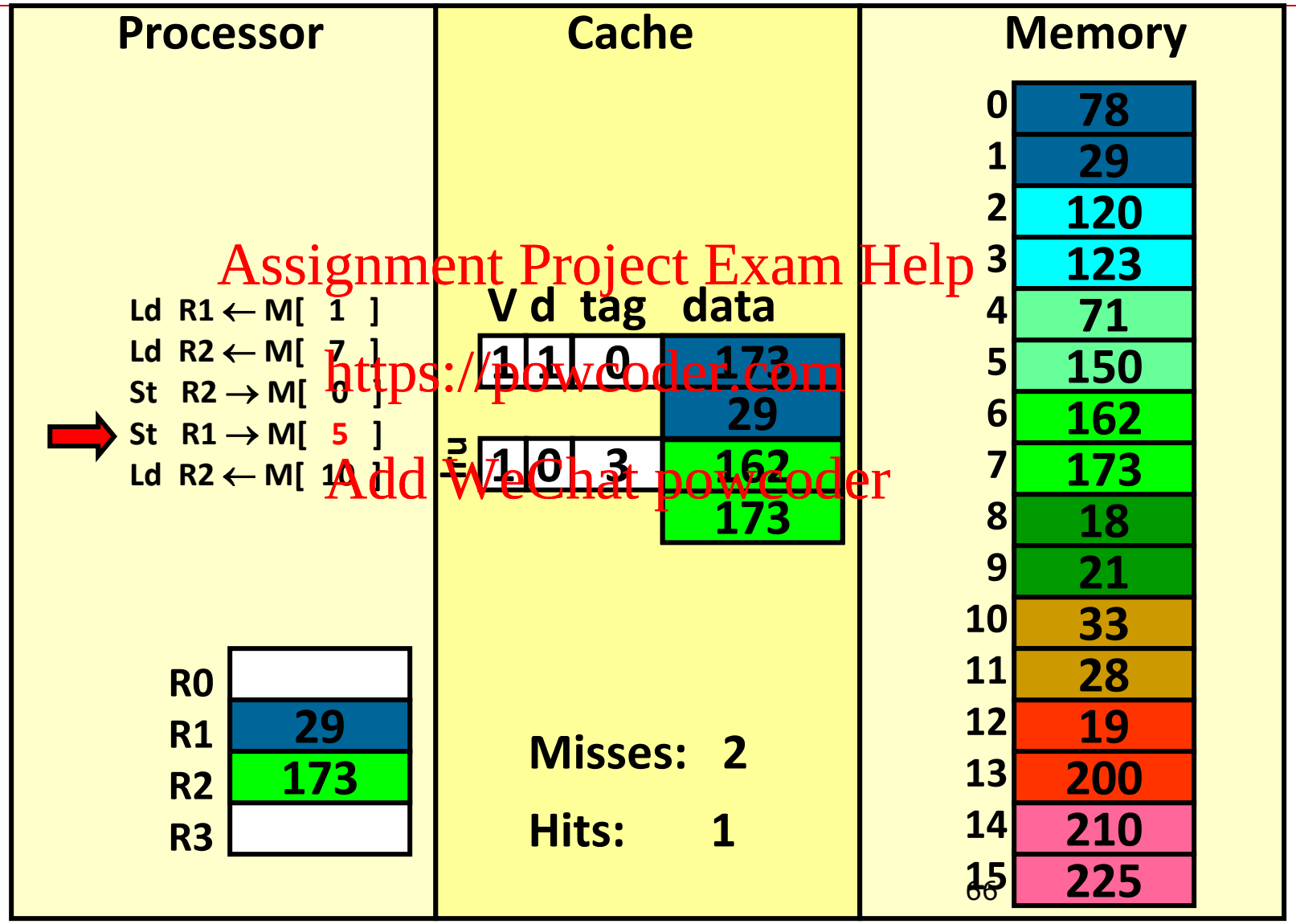
write-back (REF 3)



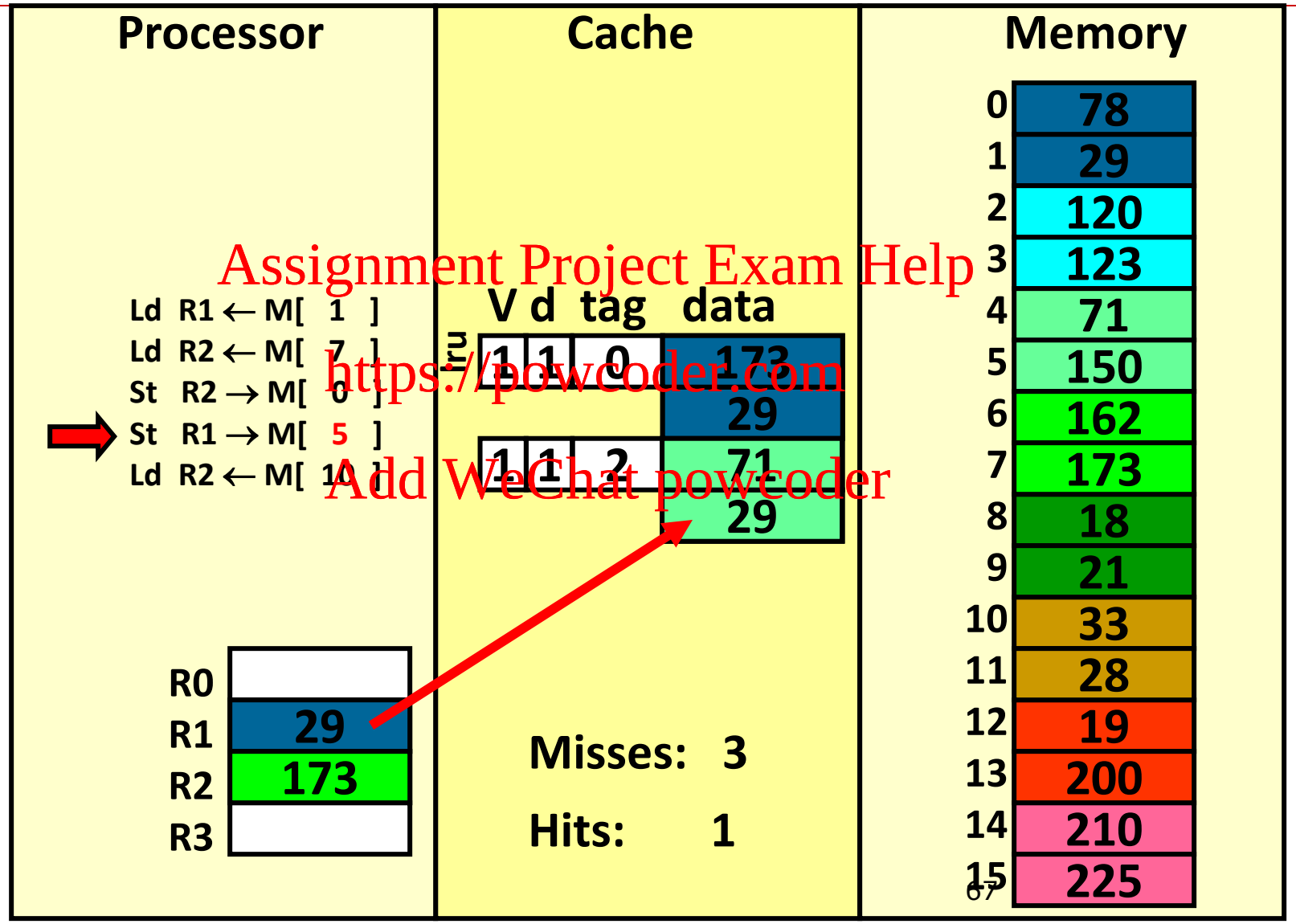
write-back (REF 3)



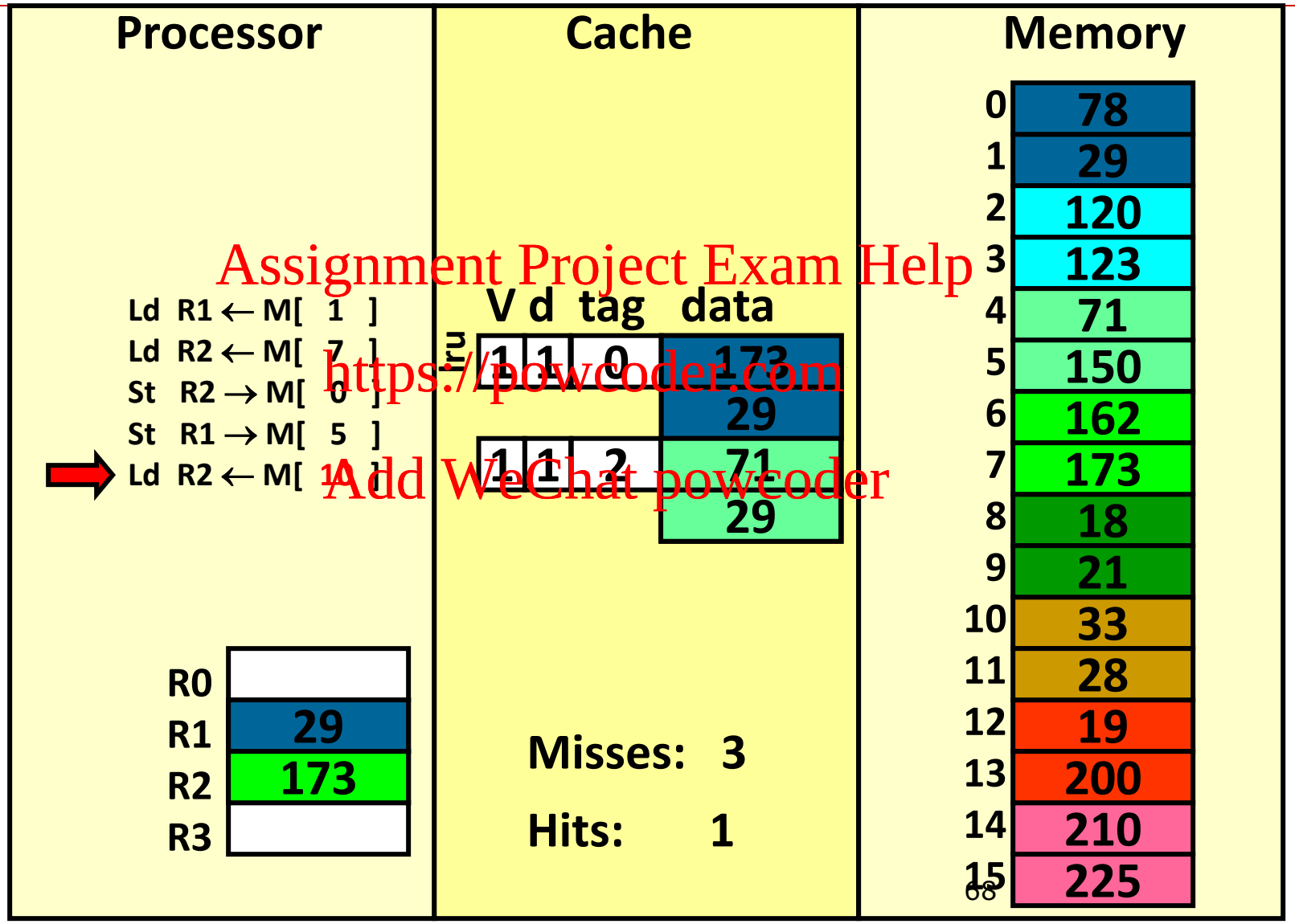
write-back (REF 4)



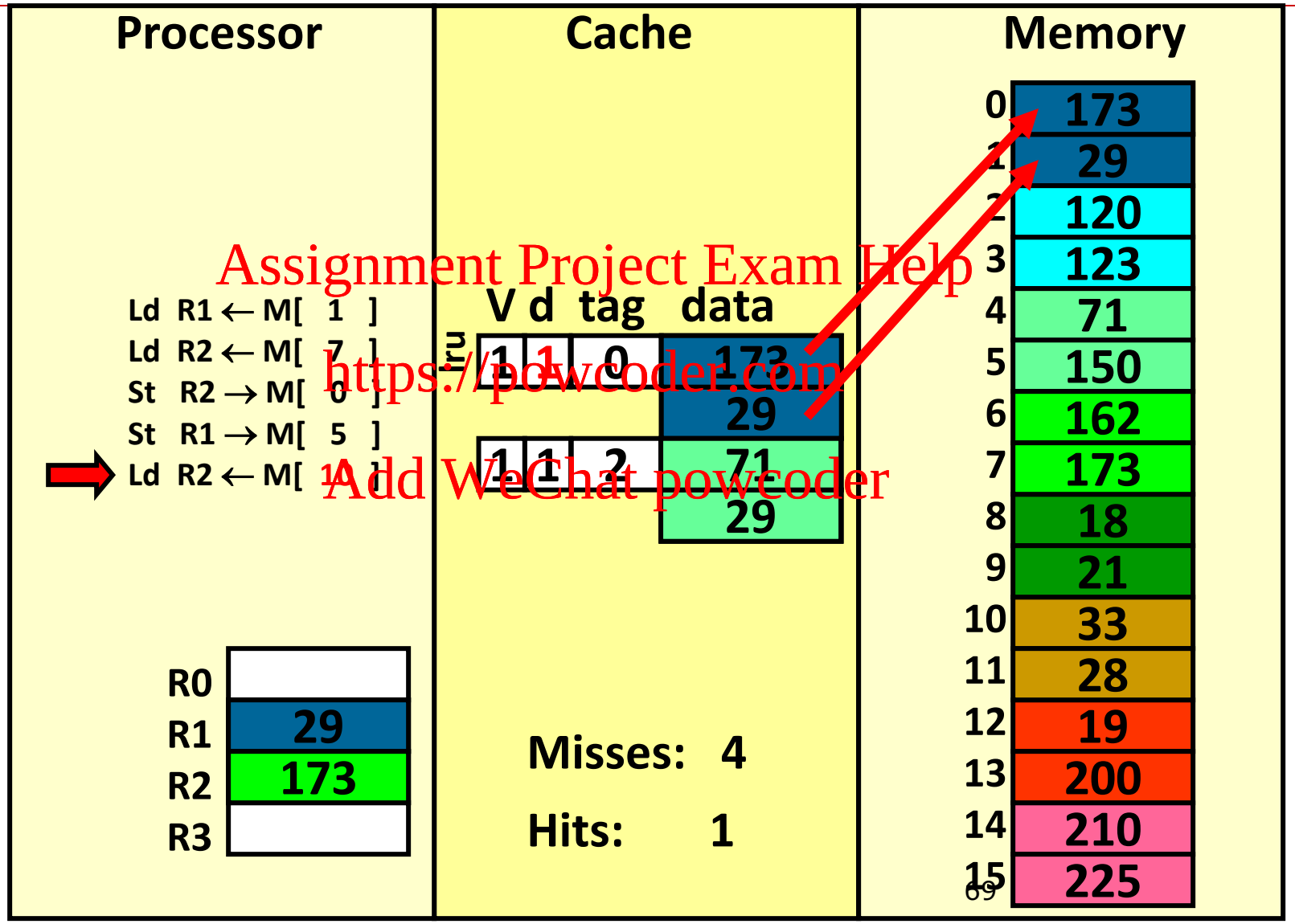
write-back (REF 4)



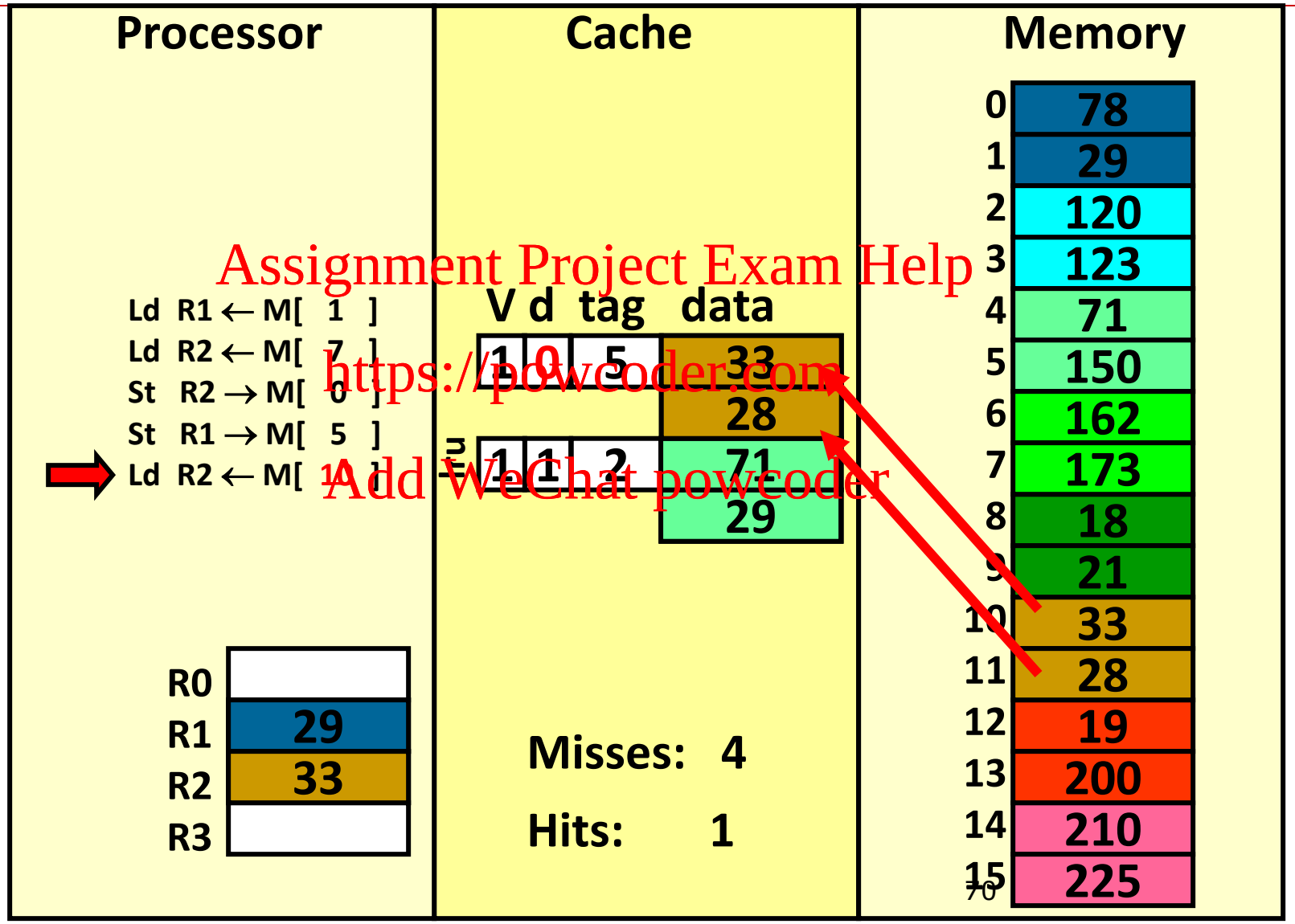
write-back (REF 5)



write-back (REF 5)



write-back (REF 5)



How many reads/writes to main memory in a write-through cache?

Each cache miss reads a block (2 bytes in our example) from main memory

Total reads from main memory: 8 bytes

Each store writes a byte to main memory

Total writes to main memory: 4 bytes

For this example, would you choose write-back or write-through?

Summary: Questions to ask about a cache

Cache design (so far):

How many bytes of data storage?		
What is the block size?		(spatial locality)
How many lines?	(= cache size / cache block size)	
What is the replacement policy?	(LRU? LFU? FIFO? Random?)	(temporal locality)

Performance

- What is the hit rate?
- What is the latency of an access?

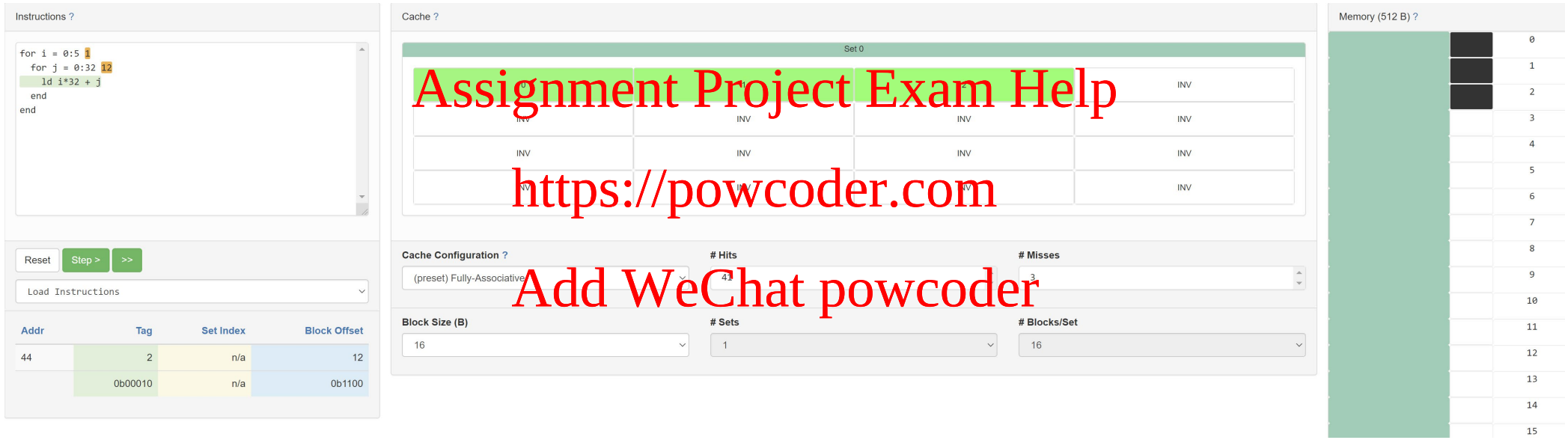
Area overhead

- How much overhead storage? (tags, valid, dirty, LRU)

Interactive tools

Interactive tool: caches

URL: Cache interactive simulator



Interactive tool: pipeline

URL: [Pipeline interactive simulator]