17. Cache and memory hierarchy: The basics

EECS 370 – Introduction to Computer Organization – Fall 2020

Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

Instructor switch:

Professor Satish Narayanasamy

Taking over from Professor Bill Arthur

Covering caches and virtual memory (asynchronous lectures #17 - #25)

Upcoming deadlines:

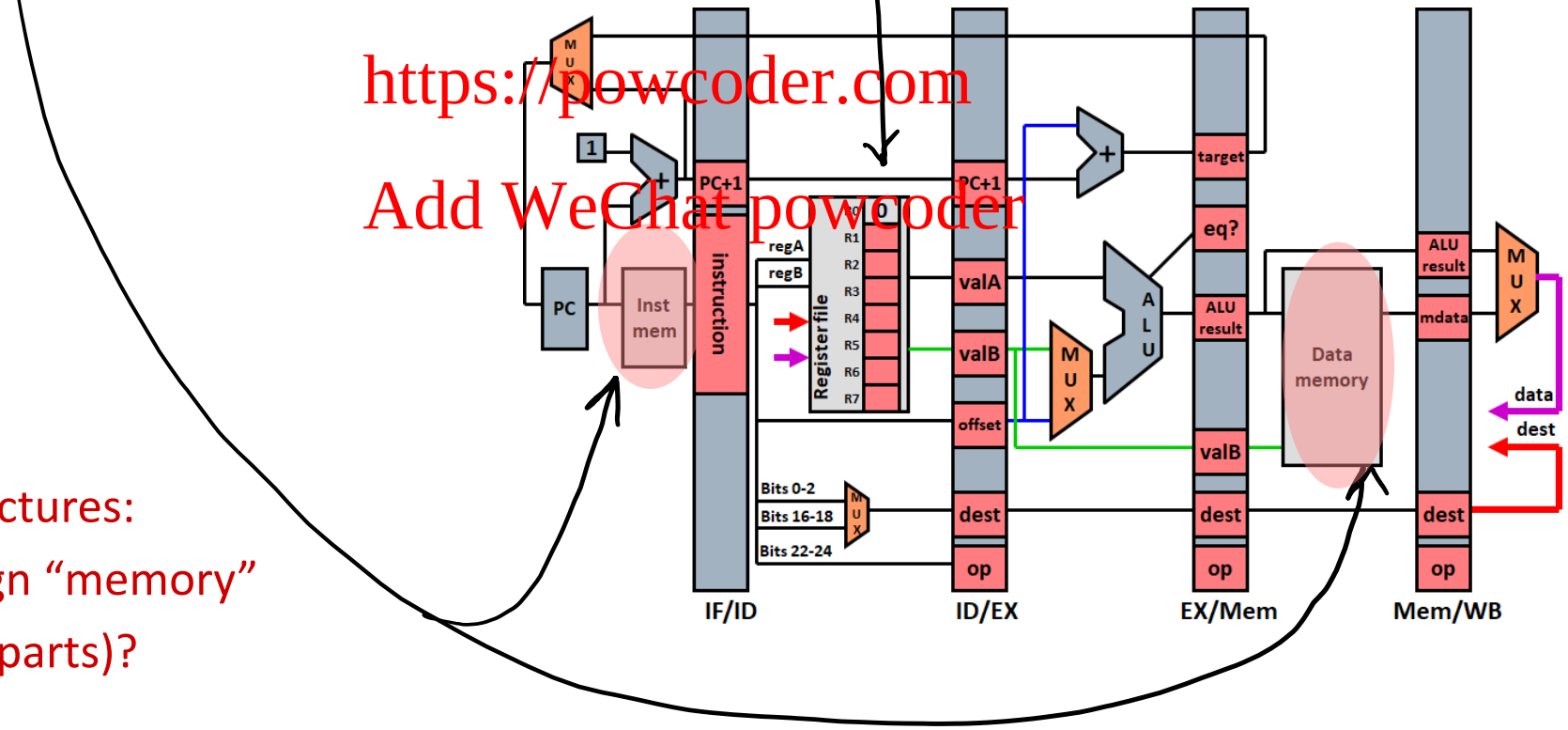
HW4	due Nov 10th
Project 3	due Nov. 12th

Part 1: Memory Hierarchy and Caches: Introduction

Memory seen in previous lectures

LC2K data-paths have these structures that hold data and instructions:

- Register file (little array of words)
- Memory (bigger array of words)



Upcoming lectures:

How to design “memory” (highlighted parts)?

Memory System: Learning objective

LC2k program	can access	2^{18} bytes of memory
MIPS program	can access	2^{32} bytes of memory
ARM64 or x86-64 program	can access	2^{64} bytes of memory (18 billion billion bytes!)

Problem: No one memory technology is both fast and big to store all of program's data

Goal: Design a fast, big, and cheap memory system to store a program's data.

Memory System: Desirable Properties

Big memory

Fast memory

A load instruction would stall a data-path, if memory access takes longer than a cycle

Cheap memory

Measured as cost per byte of data. A critical component that determines cost of computers.

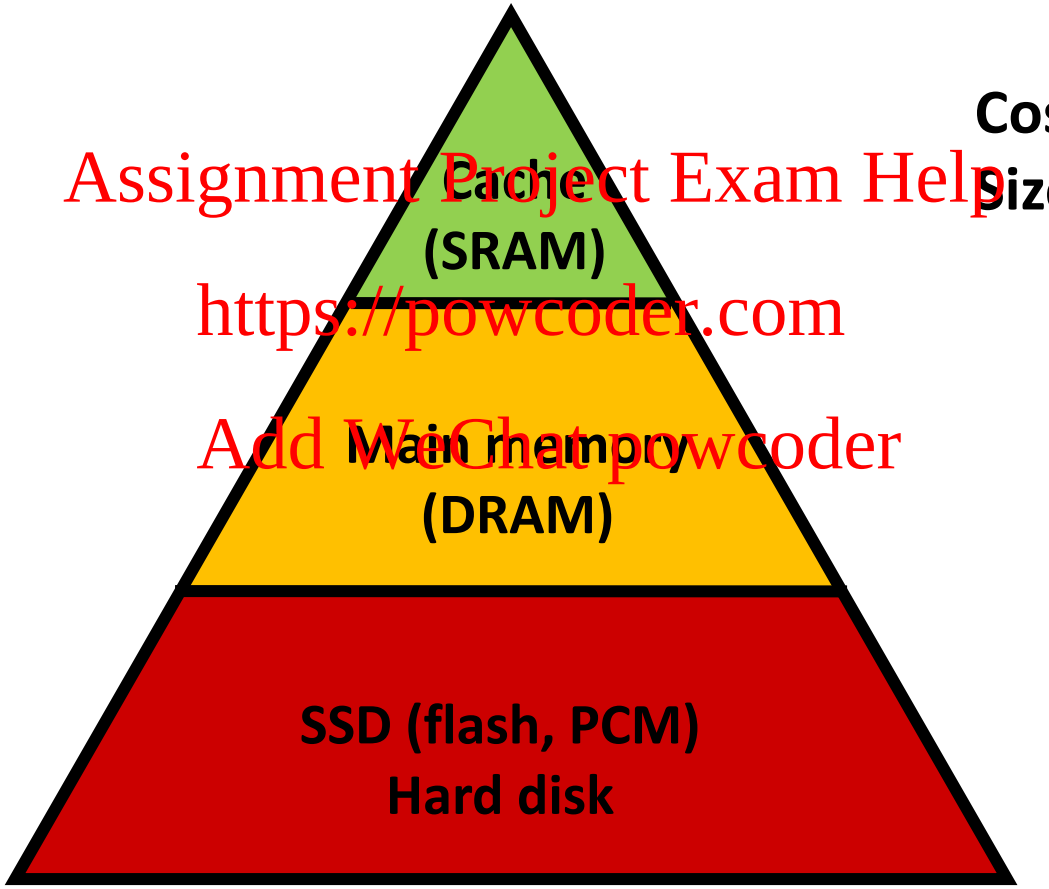
Volatile or not?

Does the data vanish or persist when power is turned off?

Memory Pyramid

Fast

Slow



Cost **Expensive**
Size **Small**

Cost **Cheap**
Size **Big**

SRAM (Static RAM)

Area:	6T: 6 transistors per bit (used on-chip within processor)
Fast:	~2ns access time, if size is small (few KBs)
	Larger the size, longer the access time
Typical Size:	Tens of KBs to a few MBs
Cost:	Expensive
	~$5.0 per megabyte

$0.13	for 2^18 bytes of memory	(LC2K ISA)
$20,000	for 2^32 bytes of memory	(MIPS ISA)
$88 trillion	for 2^64 bytes of memory	(ARM64 ISA)

Volatile

DRAM (Dynamic RAM)

Area:	A tiny capacitor and a transistor per bit
Slower:	~60ns access time (for few GBs of size)
Typical Size:	Tens of GBs
Cost:	Less expensive than SRAM ~$0.004 per megabyte

$0.00	for LC2K
$16	for MIPS
$70,000,000,000	for ARM64

Volatile

Disks

Obnoxiously slow: 3,000,000ns access time

Typical size: tens of TBs

Cost: Cheaper than SSDs (flash, PCM)

$0.000043 per megabyte

$0.00 LC2

$0.18 for MIPS

$760,000,000 for ARM64

Non-volatile

Flash

Floating-gate transistors. SSDs have replaced hard disks in mobile phones and laptops.

Slower still: ~250ns access time

Typical size: hundreds of GBs to a few TBs

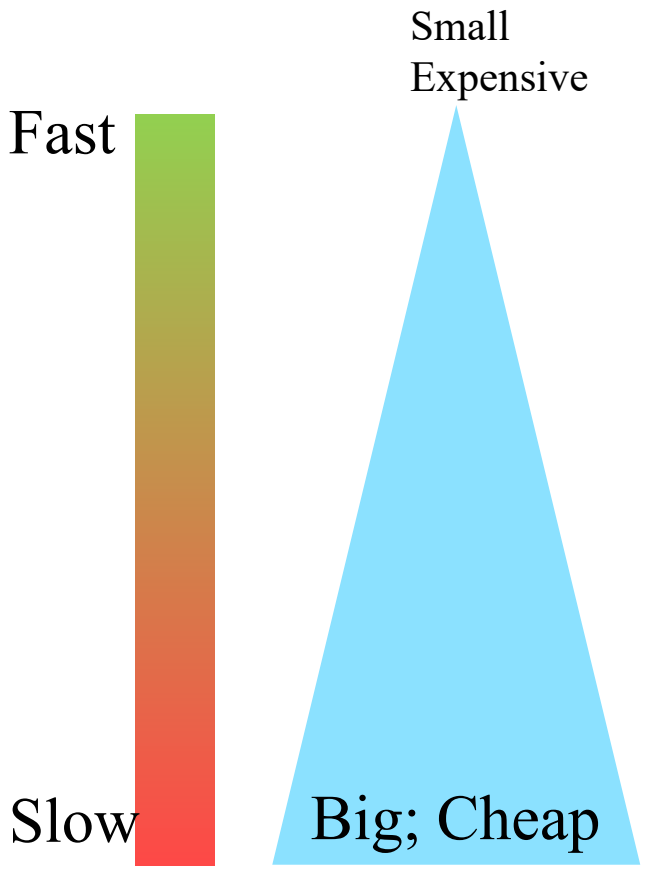
Cost: Less expensive than DRAM

$0.0012 per megabyte

$0.00	for LC2
$4.9	for MIPS
$21,000,000,000	for ARM64

Non-volatile

Memory Technologies: Summary



Static RAM	(SRAM)	Used inside processors
Dynamic RAM	(Dynamic memory)	"Main" memory

volatile

Non-volatile

Solid state disks

- Phase-change memory (PCM) — Emerging (e.g., Intel Optane)
- Flash — Common in mobile devices

Magnetic Disk (hard disk)
DVD
Magnetic tape

In research: DNA storage

Memory Hierarchy Goal

How to get best properties of different memory technologies?

A memory system that is as fast as SRAM, but as big and cheap as a hard-disk?

Fast:

Ideally run at processor's clock speed

1 ns access time

Big and Cheap:

Sufficiently large to hold a program's data

Memory Hierarchy Analogy: Storing and retrieving a book

Option 1: Library stores all the books. Every time you switch to another book, return current book to library and get a new book.

Latency = few hours

Option 2: Borrow 20 frequently-used books and keep them at home book-shelf

Latency = few minutes (mostly, go to library once a week or so)

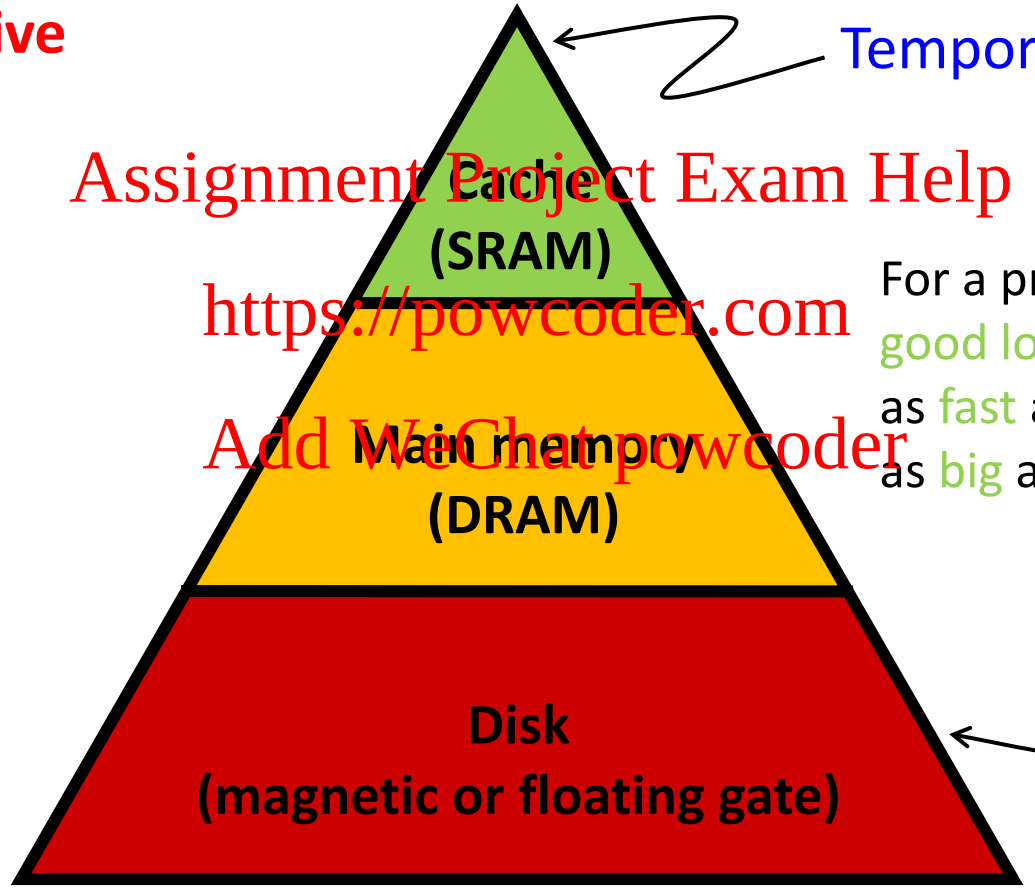
Option 3: Keep 3 books in backpack

Latency = few seconds (mostly, go to book-shelf once a day or so)

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

A Realistic Memory Hierarchy

Cache

Few KBs to MBs of SRAM (within processor – on-chip cache)

- Fast
- Small, so cheap
- Services most loads and stores, provided program has good locality

Main Memory

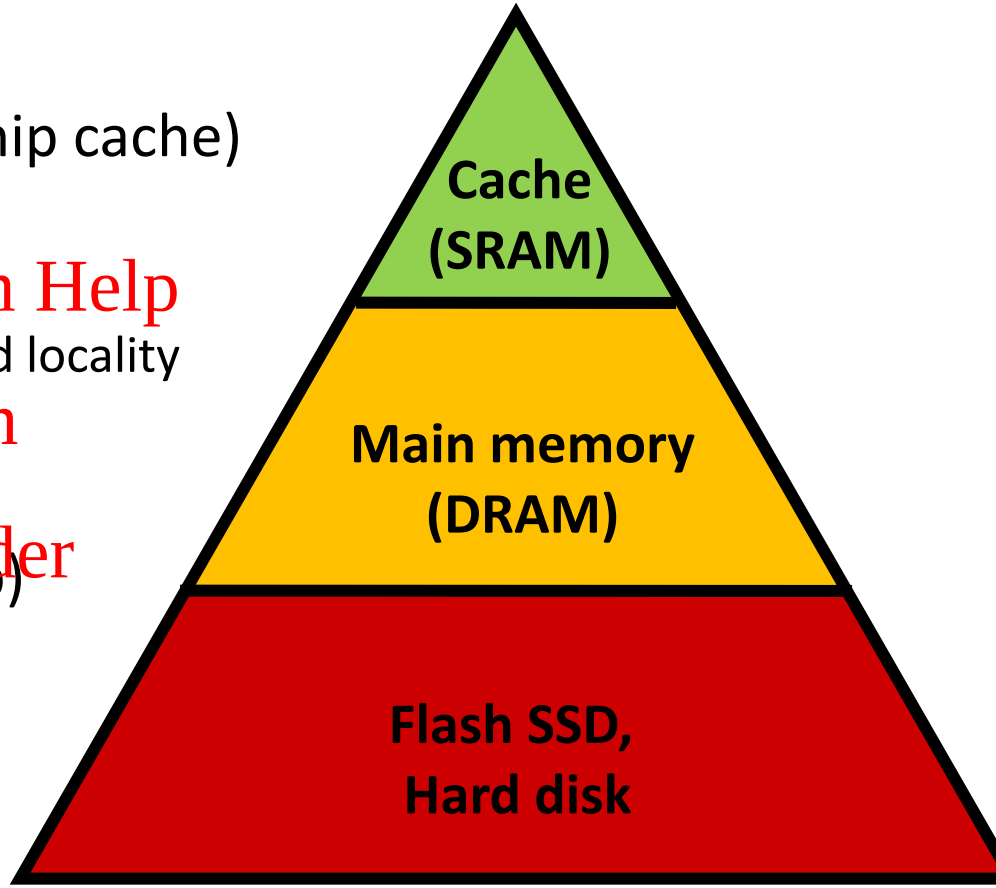
Tens of GBs of DRAM (outside processor – off-chip)

- Cheaper than SRAM, faster than flash/disk

"Swap space"

Few TBs OF flash and/or disk

- Cheap, Big, Non-volatile.



No memory is enough for a 64-bit ISA (ARM64) program

Hard disk cost for storing all addresses accessible to a ARM64 program

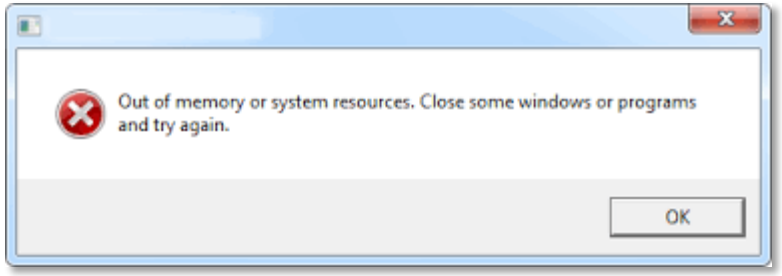
$760 million for 2^64 bytes

Don't provision 2^{64} bytes of storage (even a hard disk is too expensive!)

Fake it. Use "virtual memory" to provide an illusion that ISA's entire address space is available.

A few TB is enough for most desktop machines today, or a smartphone in a few years

Computer "crashes" if your program exceeds machine's available swap space on disk



ISA abstraction hides memory hierarchy from programmers

The **architectural** view of memory is

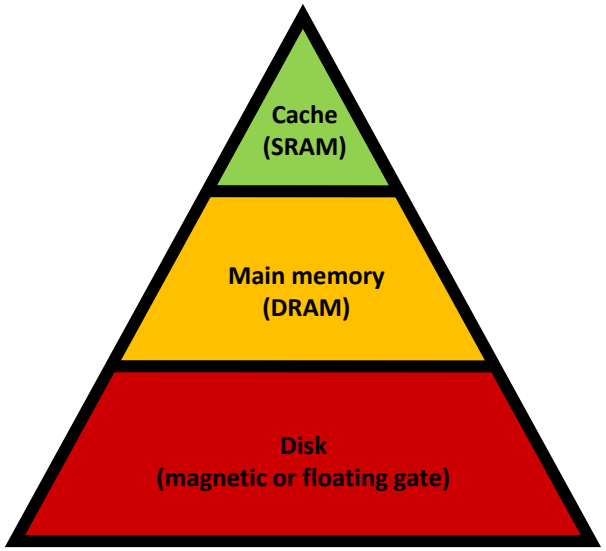
- What the machine language (or programmer) sees above ISA
- Just a big array

Breaking up the memory system into different pieces – cache (SRAM), main memory (DRAM) and disk – is **not architectural**

- ISA does not expose these details to the programmer
- A new system implementation may break it up in a different way

Programmer
can load/store to
2^64 memory locations;
Can't see memory hierarchy

ARM-64 ISA



What is a cache?

Cache commonly refers to SRAM used on-chip within the processor.

However, even DRAM (main memory) is a “cache” in that

it temporarily stores data fetched from hard disk

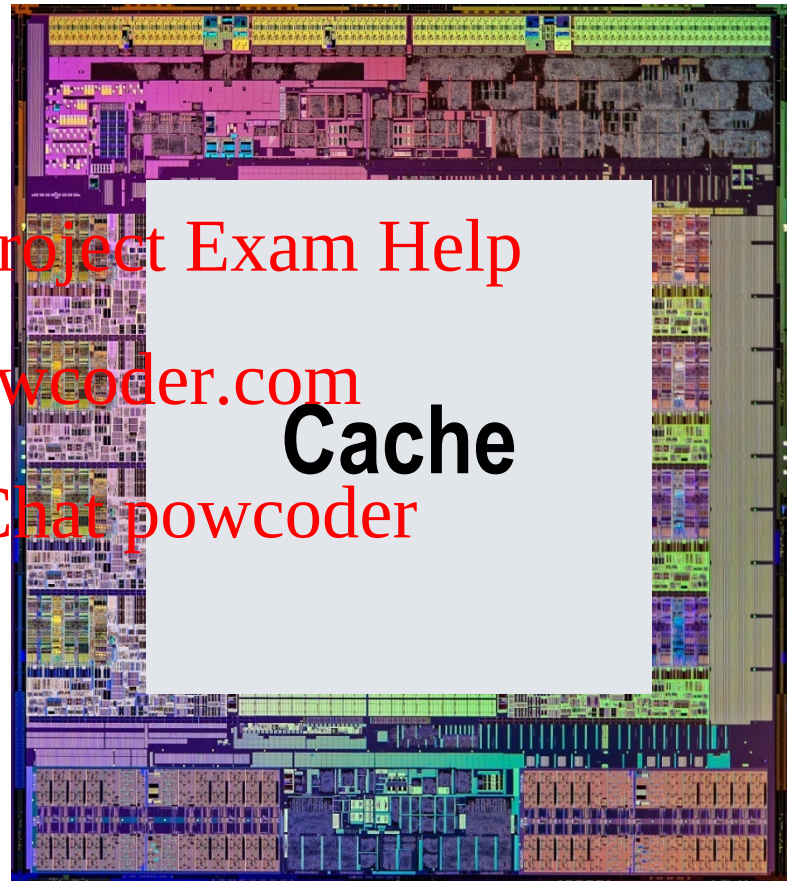
A cache is used to store data that is ***most likely*** to be **referenced** by a program

Try to maximize the number of references (loads/stores) that are serviced by the cache (avoid going slow, off-chip, main memory; or even worse, disk).

Thereby, minimize the **average memory access time (AMAT)** of a load/store

Cache: Importance

Caches consume most of a processor's die area



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

Cache Design: This lecture

Basic Cache Architecture

How to select data to store in cache?

Principle of "Temporal locality"

Illustration

Performance metric: average memory access time

Part 2: Basic Cache Architecture

Basic Cache Design

Cache memory can copy data from any part of main memory. It has 2 parts:

- The **TAG (CAM)** holds the memory address
- The **BLOCK (SRAM)** holds the memory data

Accessing the cache: **compare reference address and tag**

- Match? Get the data from the cache block
- No Match? Get the data from main memory
- How to implement this functionality? Solution: CAM for storing tags

CAMs: content addressable memories

Instead of thinking of memory as an array of data indexed by a memory address

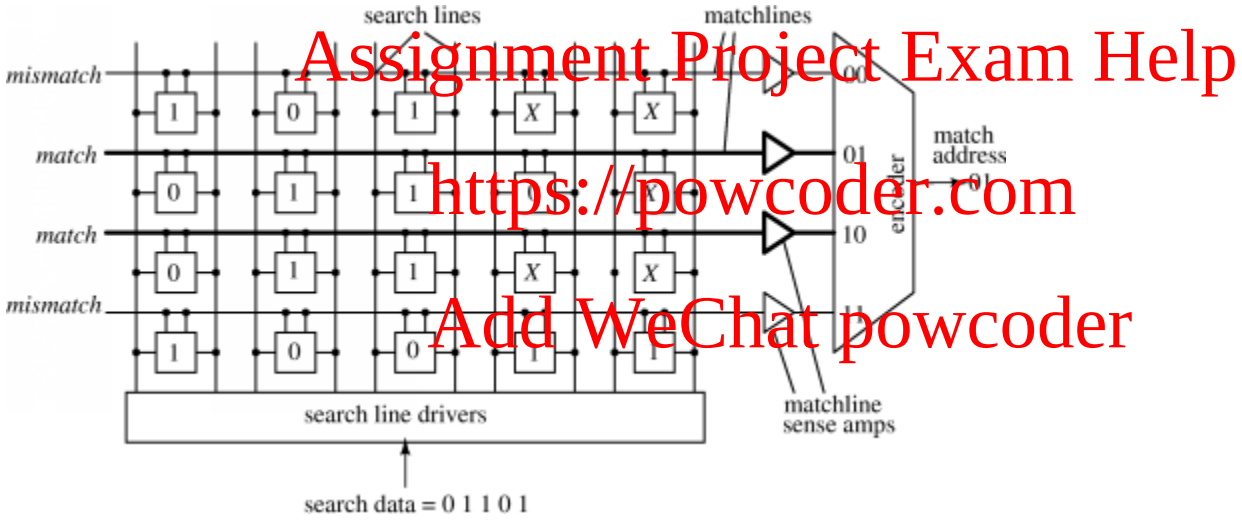
Think of memory as a set of data, that can search for a queried key

Operations on CAMs

- **Search**: the primary way to access a CAM
  - Send data to CAM memory
  - Return "found" or "not found"; "hit" or "miss"
  - If found, return location of where it was found or associated value

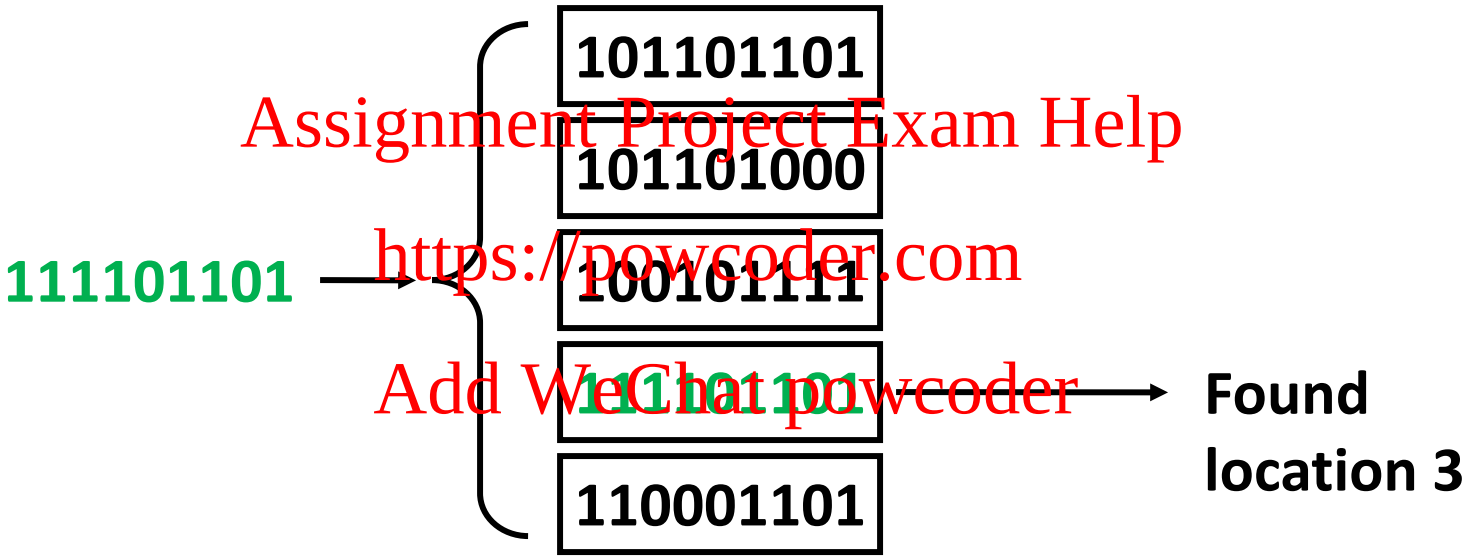
- **Write**:
  - Send data for CAM to remember
    - Where should it be stored if CAM is full?
      - Replacement policy
        - Replace oldest data in the CAM
        - Replace least recently searched data

CAM = content addressable memory



When used in caches, all tags are fully specified (no X – no don't cares)

CAM example

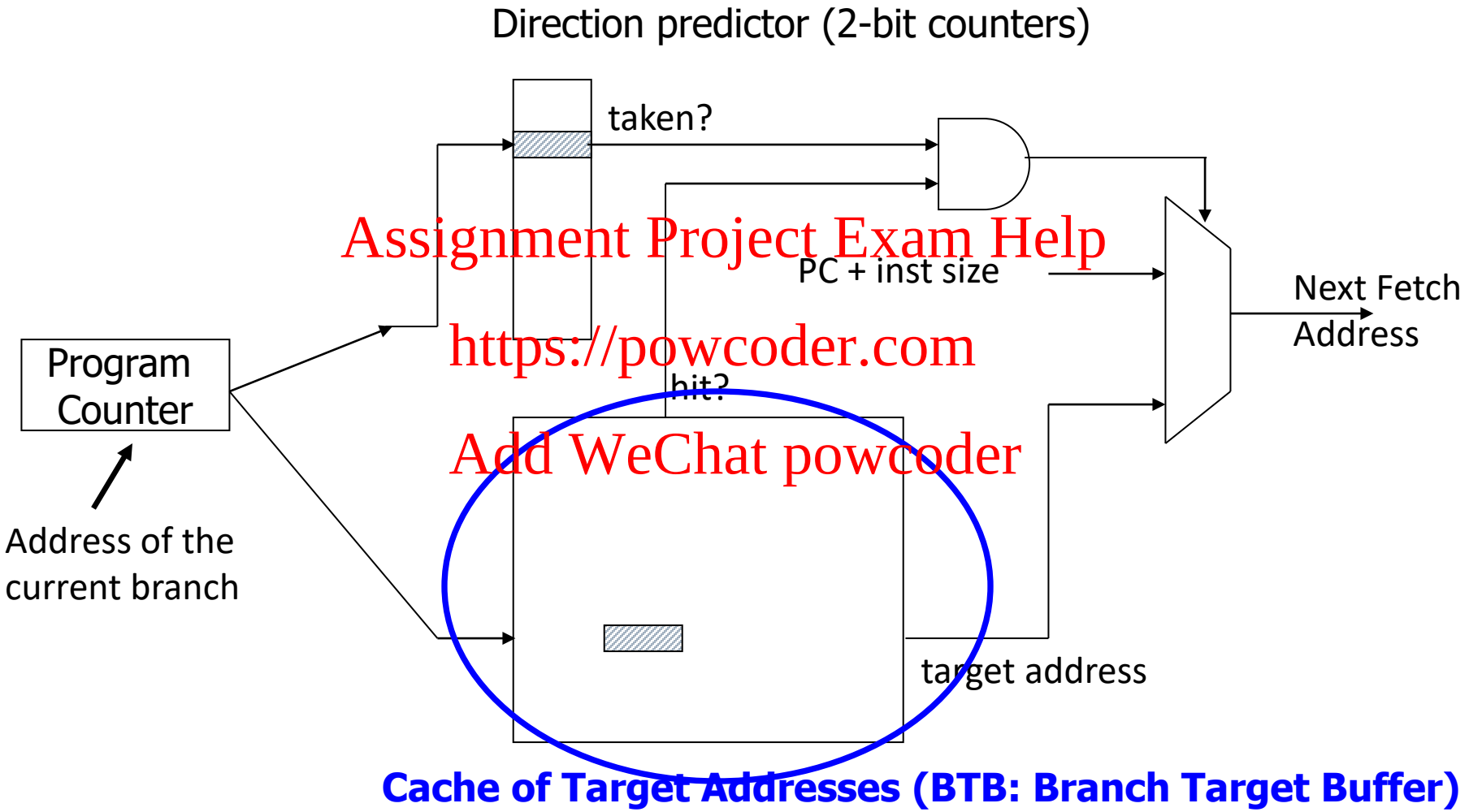


5 storage element CAM array of 9 bits each

Previous use of CAMs

- You have seen a simple CAM used before. When?

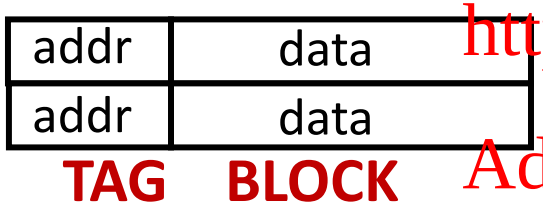
Fetch Stage with Branch Prediction



Cache Organization

Cache memory can copy data from any part of main memory.
It has 2 parts:

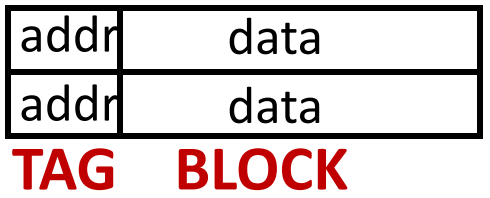
- The **TAG (CAM)** holds the memory address
- The **BLOCK (SRAM)** holds the memory data



Cache Organization

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

- Searches can be done in parallel (within reason)
- At most one tag will match



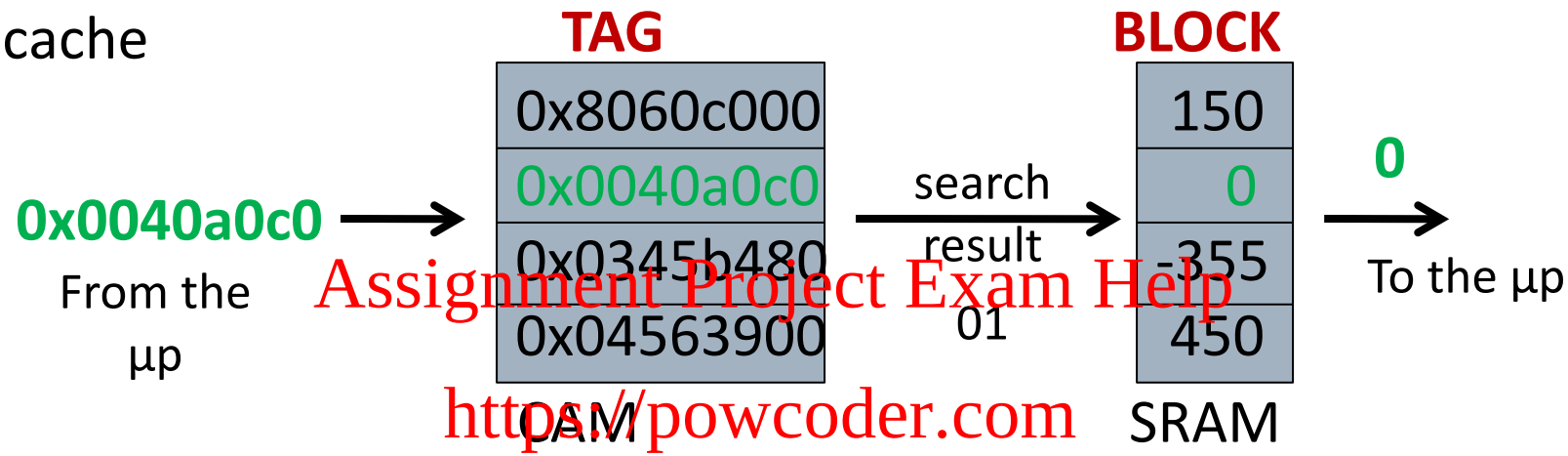
If there is a tag match, it is a cache HIT

If there is no tag match, it is a cache MISS

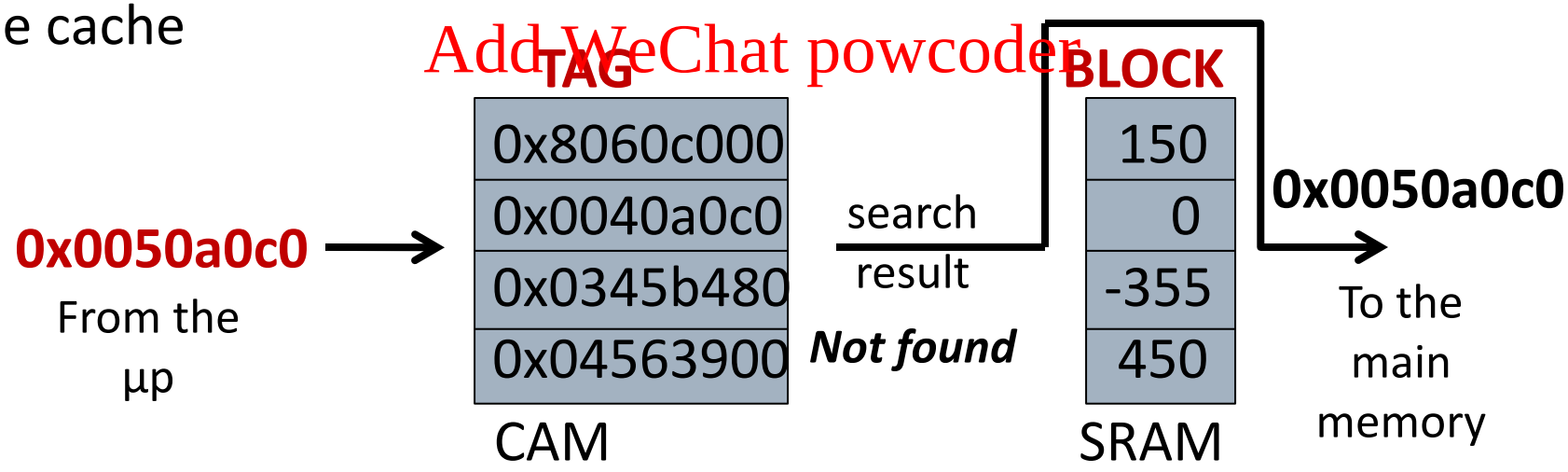
Goal: Cache data likely to be accessed in the future

Caches: the hardware view

A **hit** in the cache



A **miss** in the cache



Part 3: Temporal locality

Cache Operation

On a cache miss:

Fetch data from main memory and

Allocate a cache line and store fetched data in it

Which cache line should be allocated?

If all cache lines are allocated, how to pick the victim for data replacement?

Something To Think About

Does an optimal replacement policy exist?

That is, given a choice of cache lines to replace, which one will result in the fewest total misses during program execution

Why would we care?

Picking the Most Likely Addresses

What is the probability of accessing a random memory location?

With no information, it is just as likely as any other address

But programs are not random

They tend to use the same memory location over and over

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

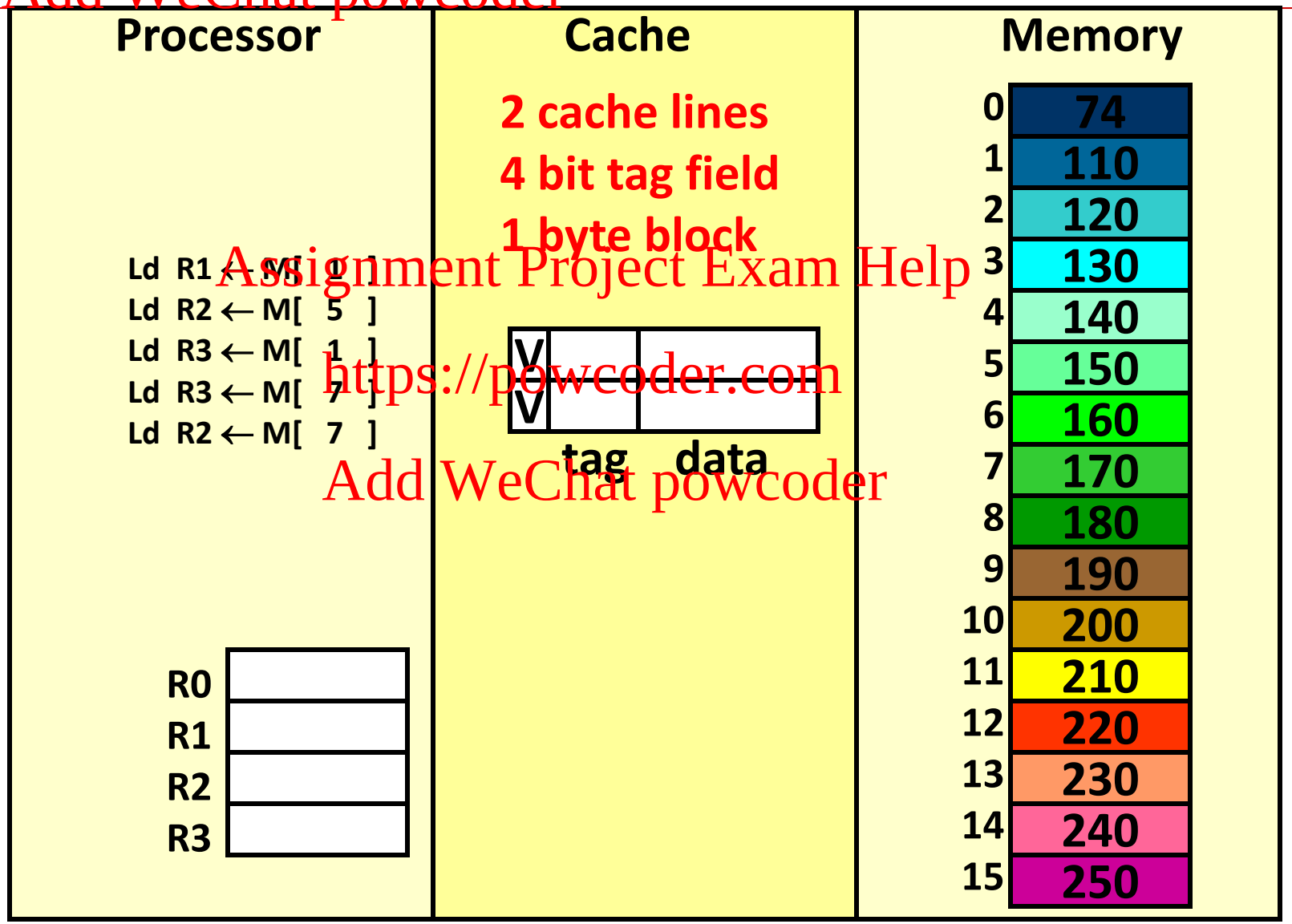
- It is the most recent reference location

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

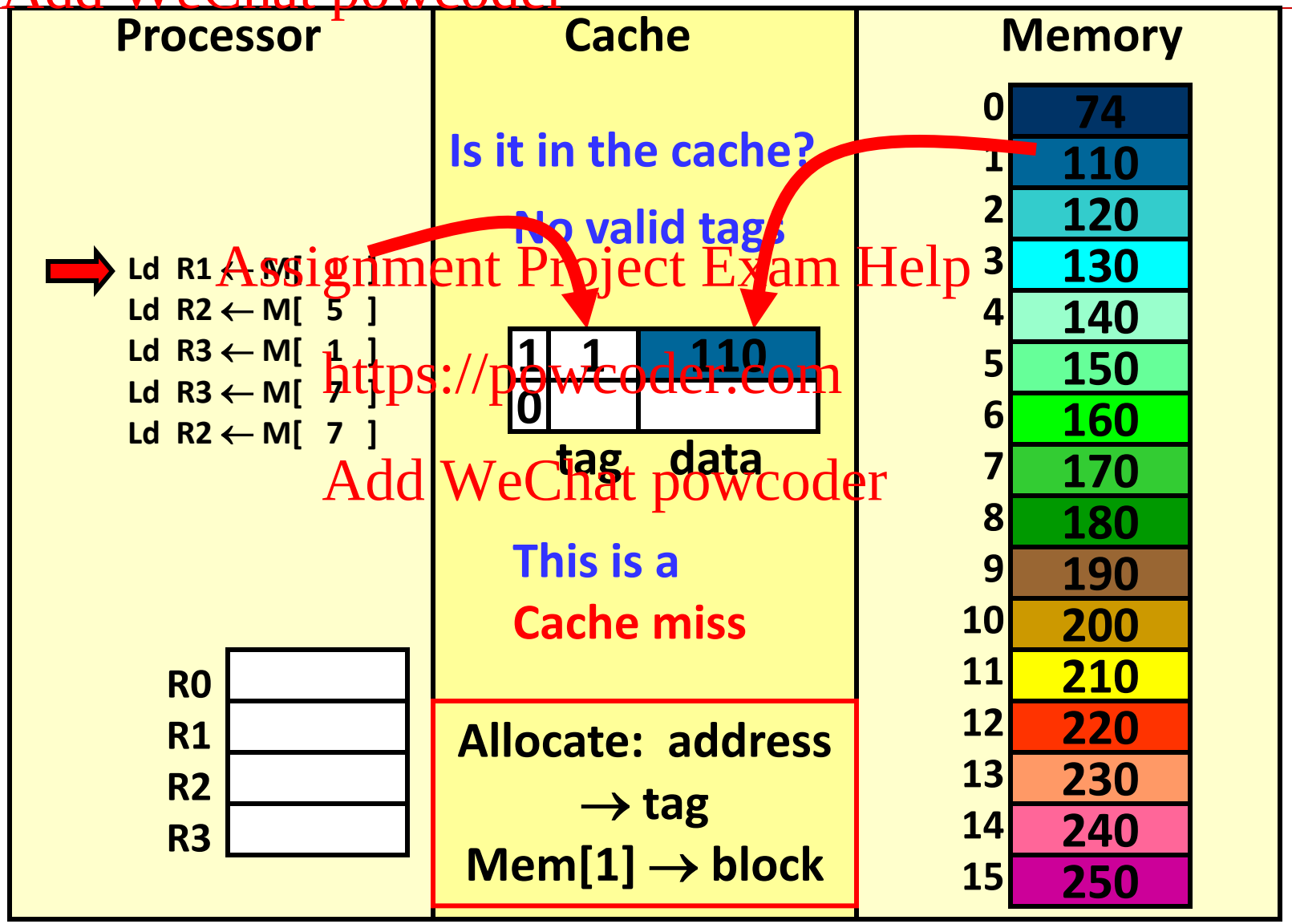
- Because the re-access probability falls over time as a cache line isn't referenced, the LRU line is least likely to be re-referenced

Part 4: Cache Illustration

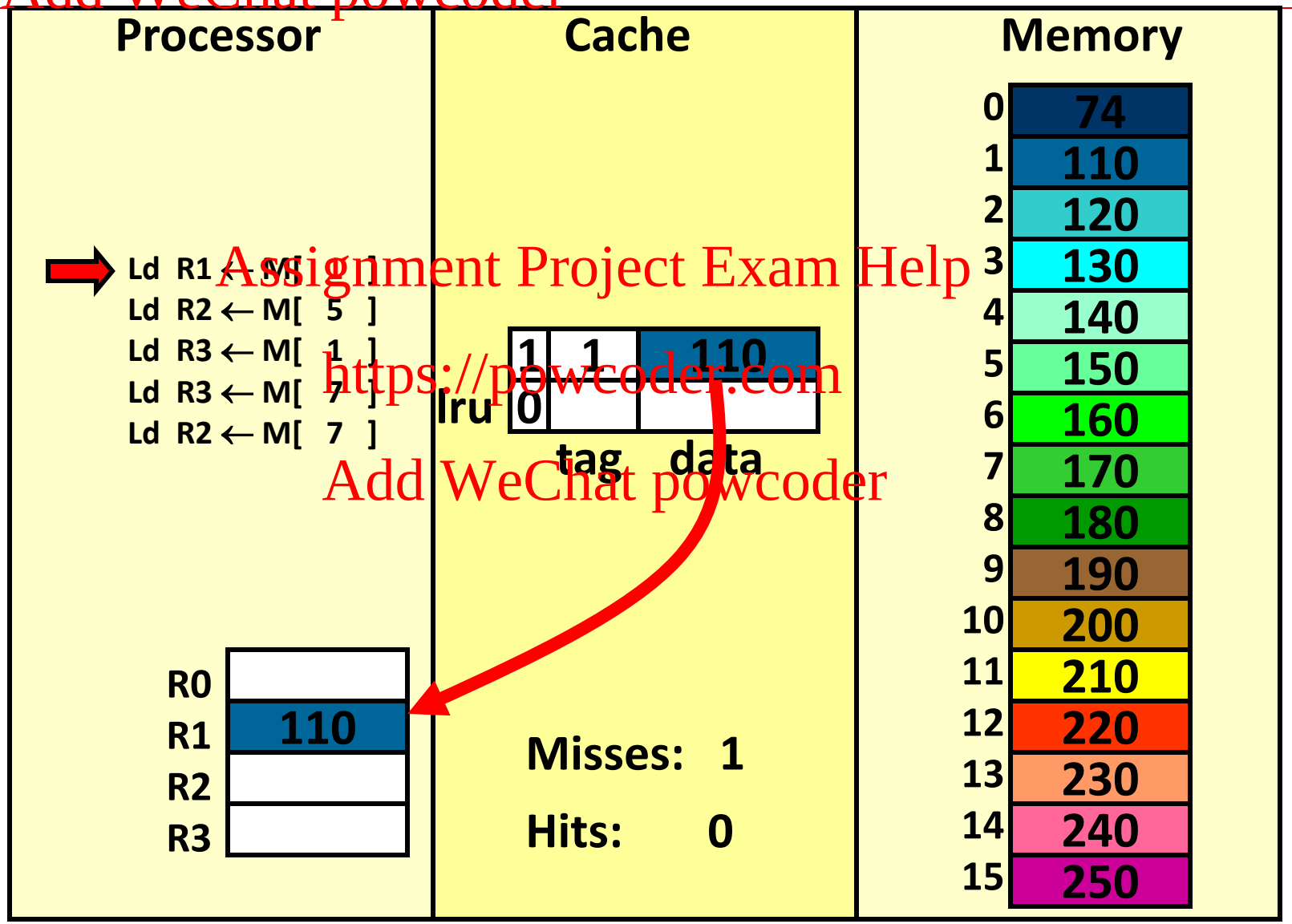
A Very Simple Memory System



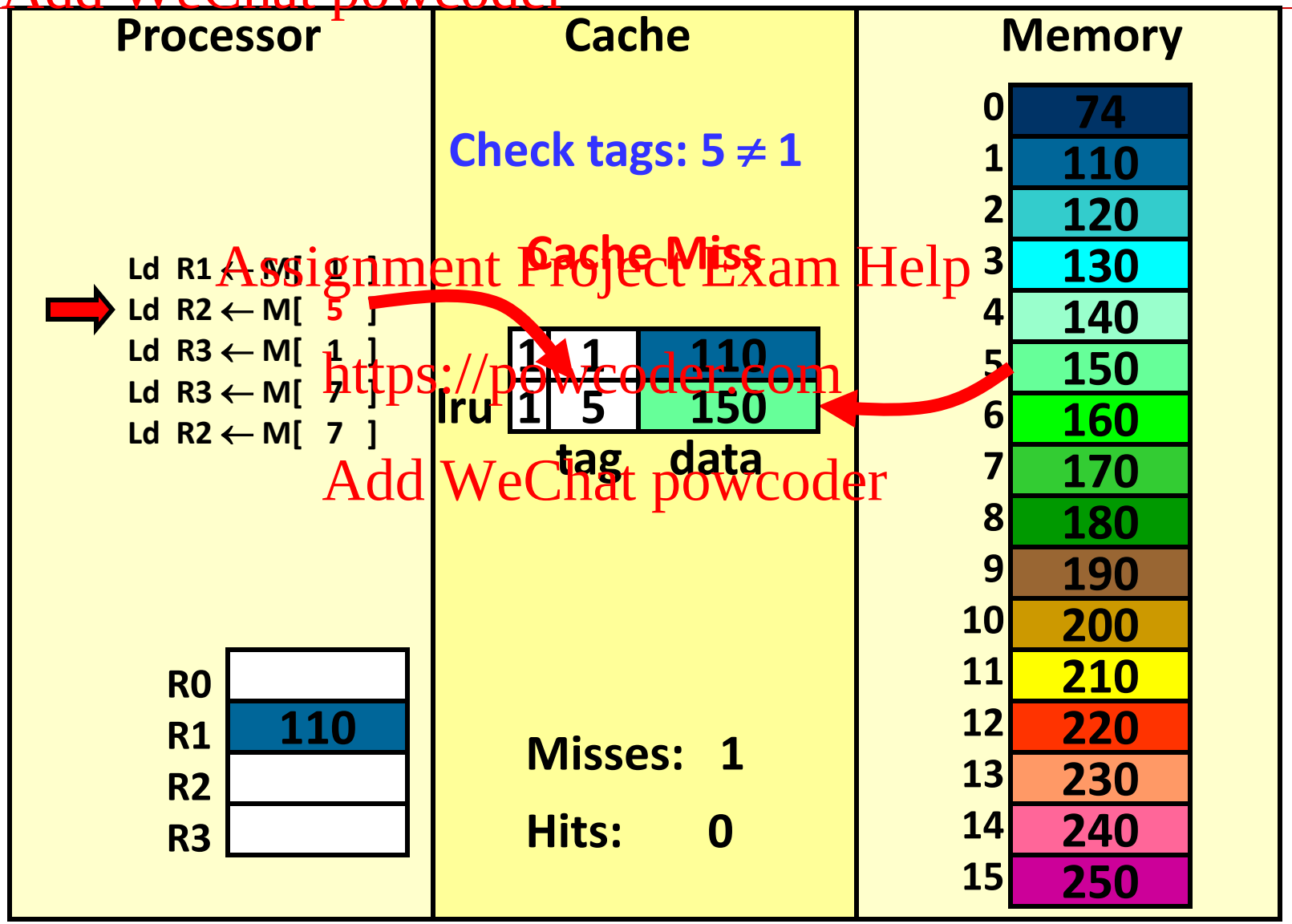
A Very Simple Memory System



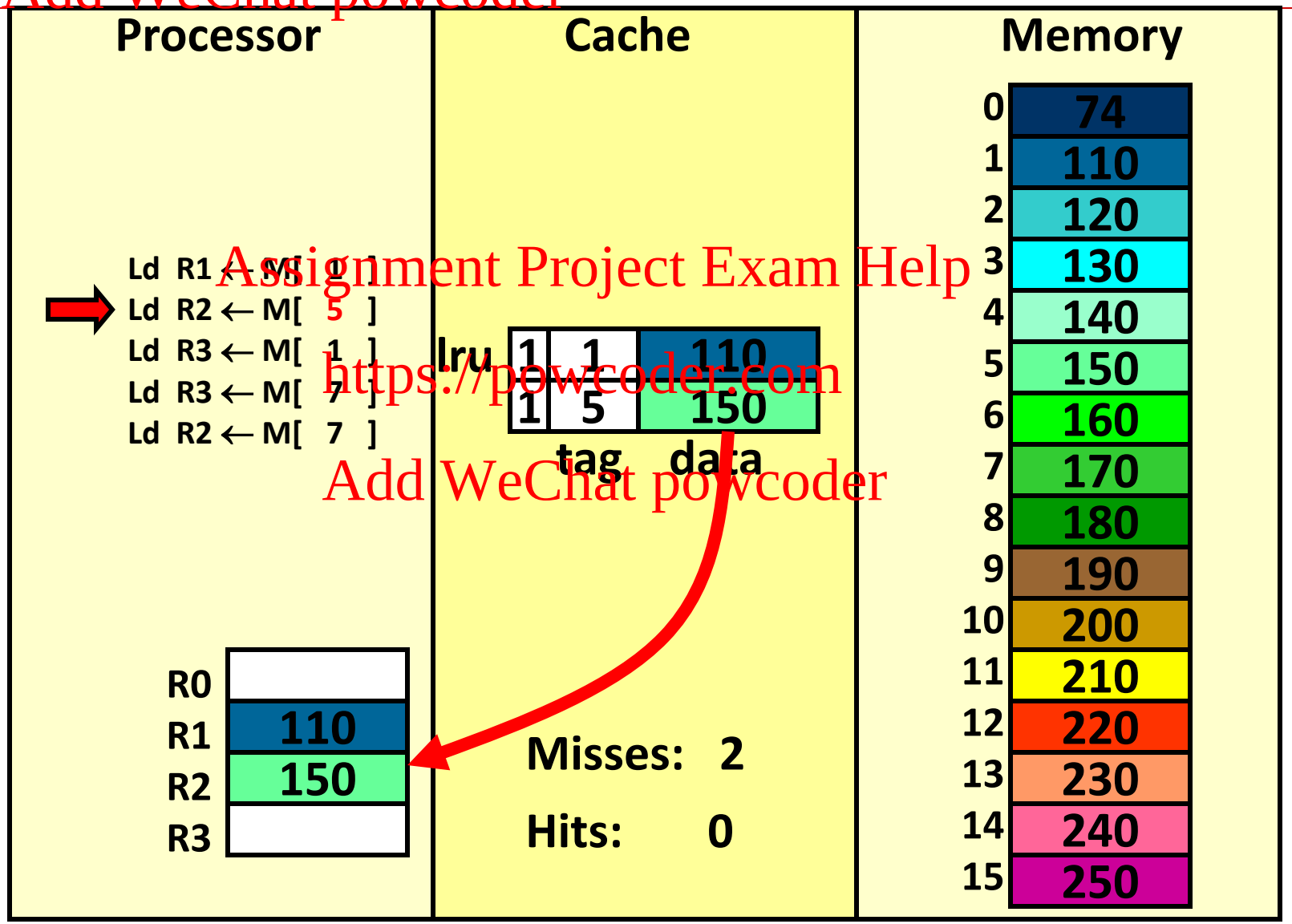
A Very Simple Memory System



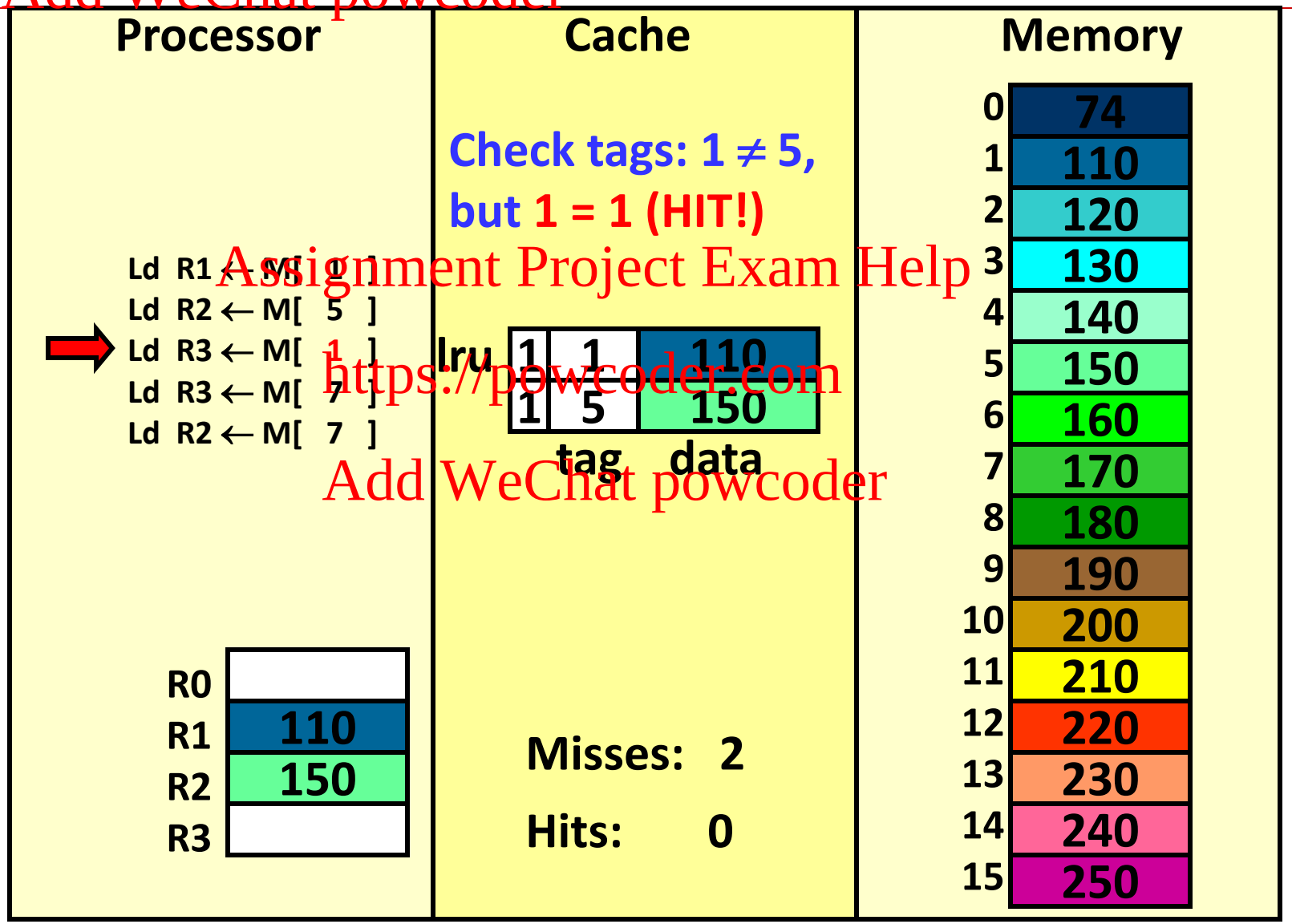
A Very Simple Memory System



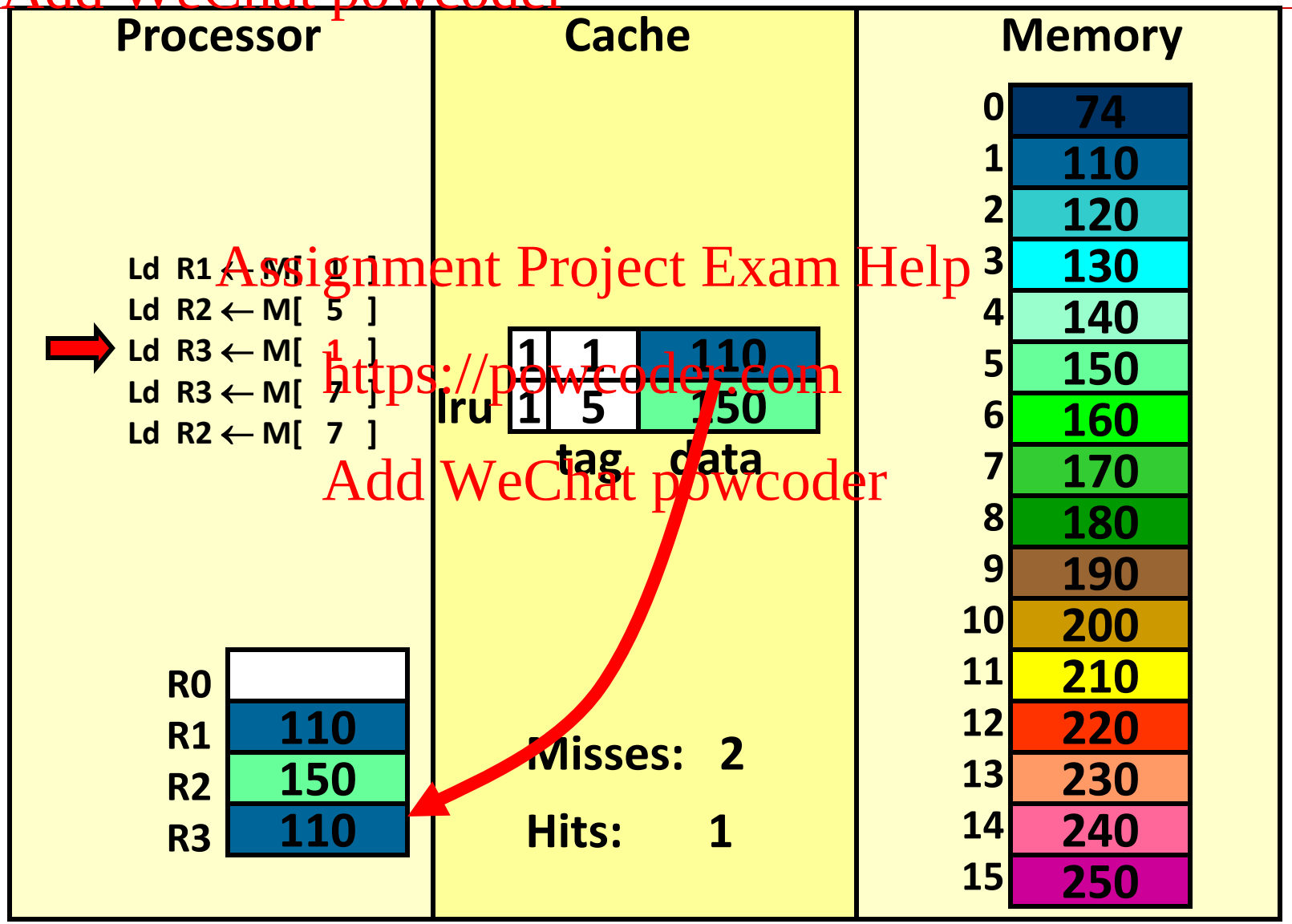
A Very Simple Memory System



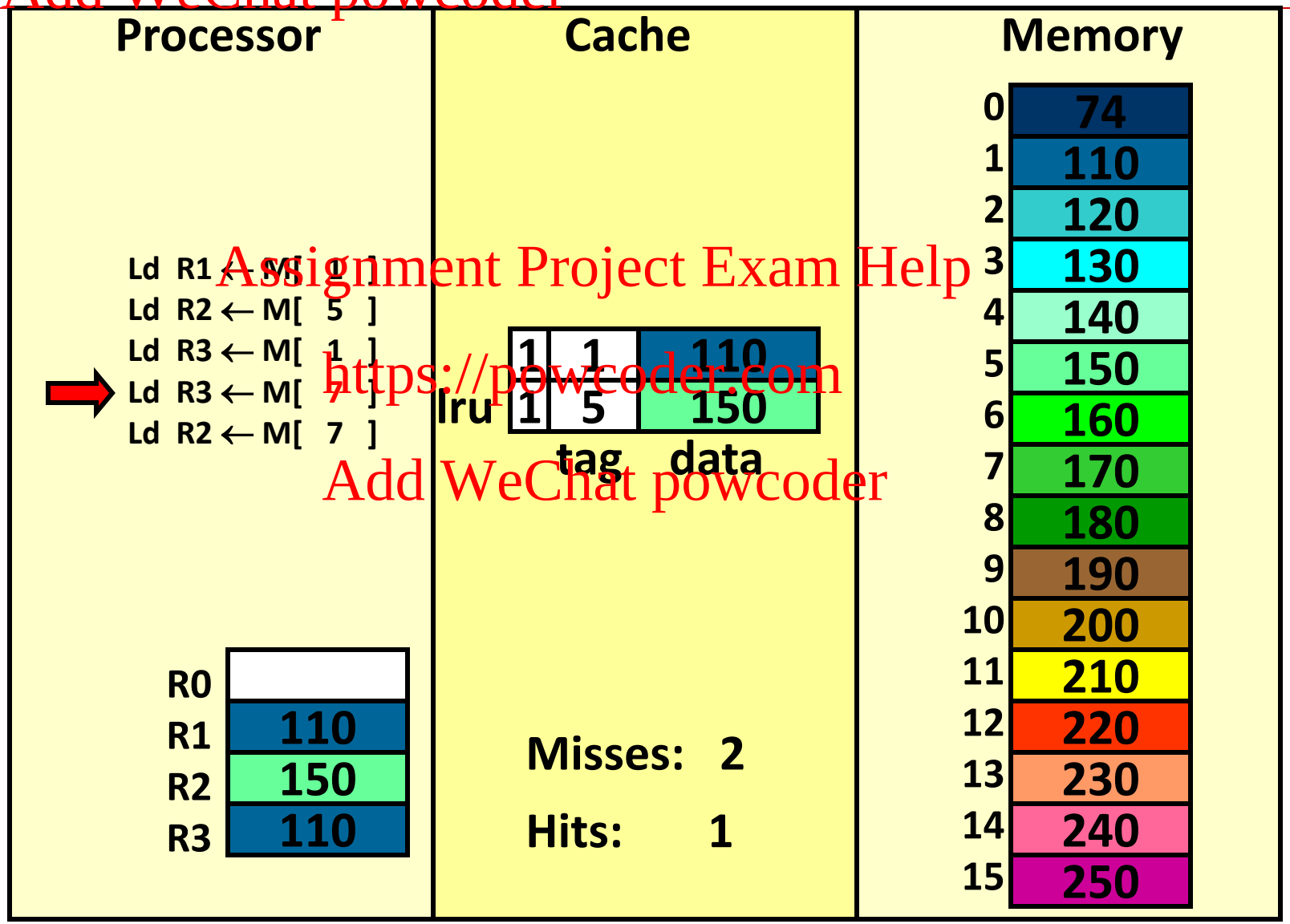
A Very Simple Memory System



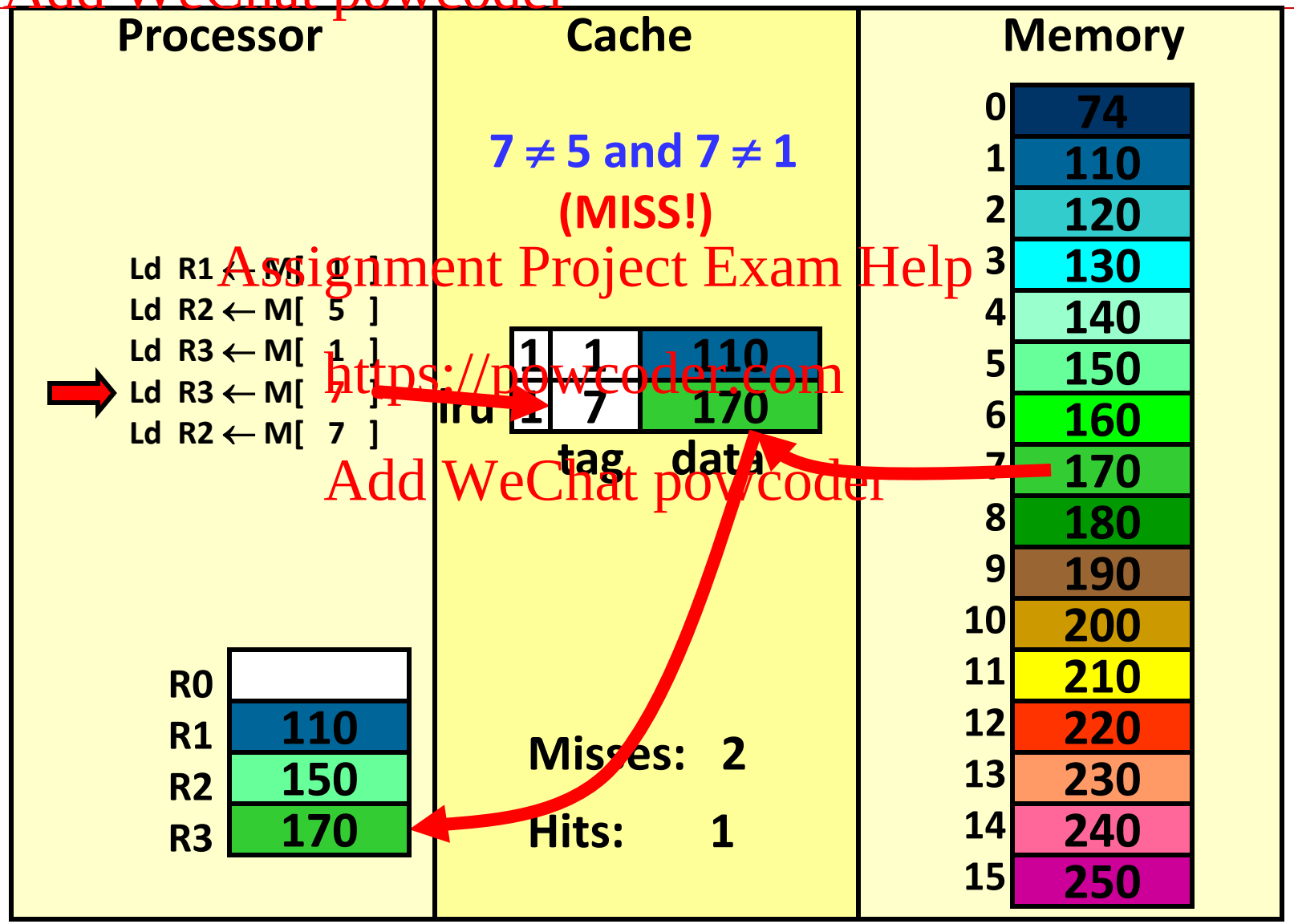
A Very Simple Memory System



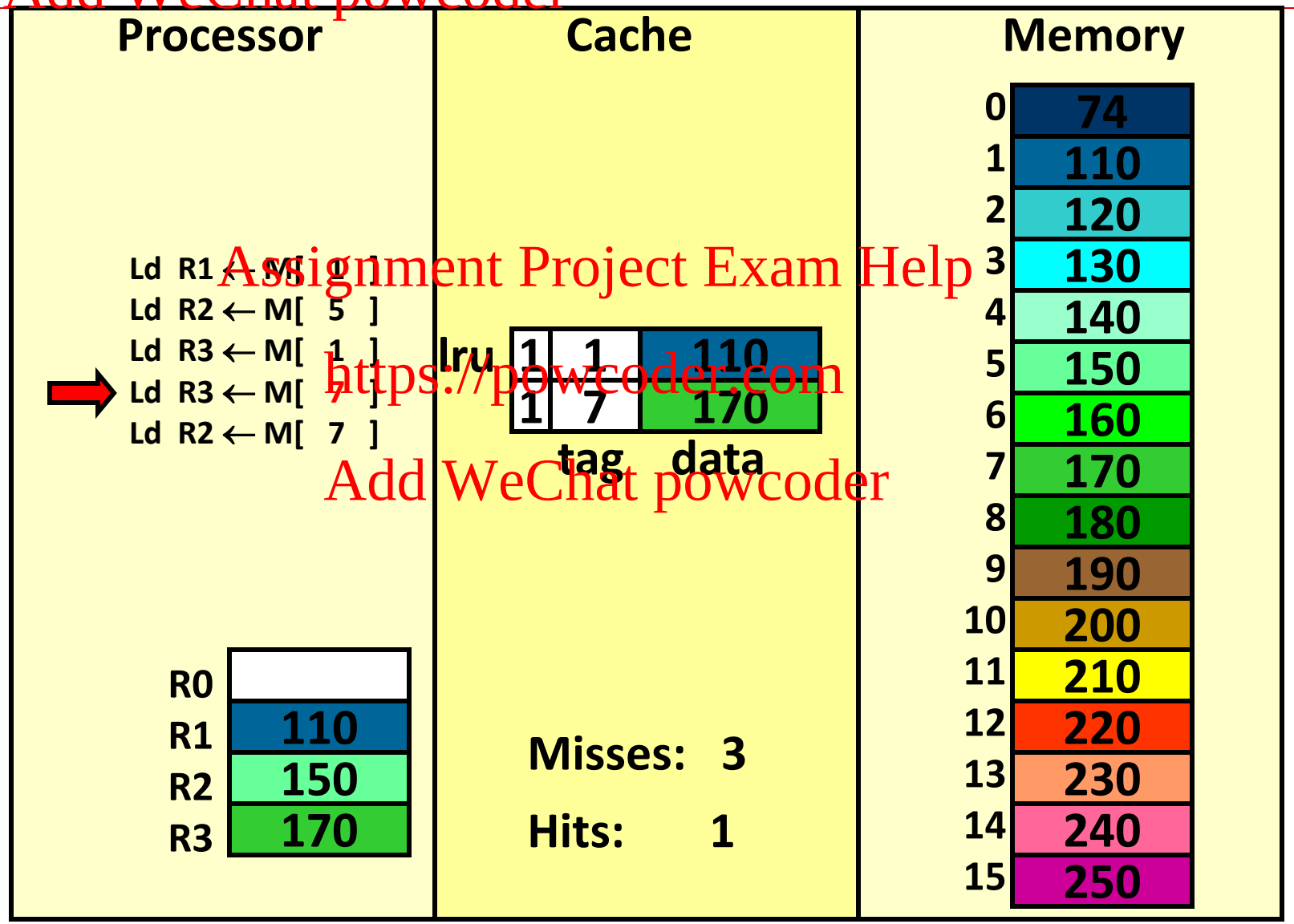
A Very Simple Memory System



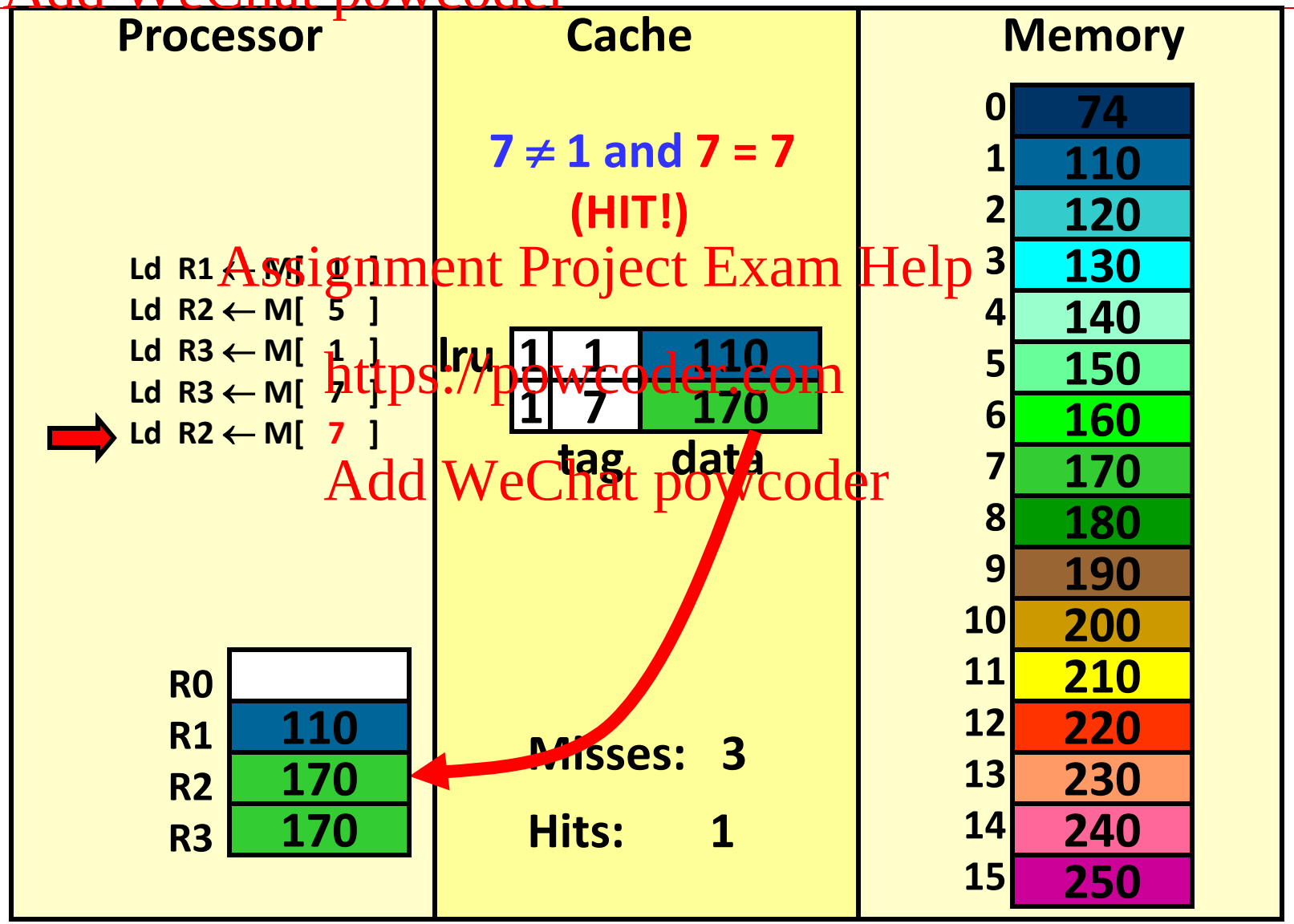
A Very Simple Memory System



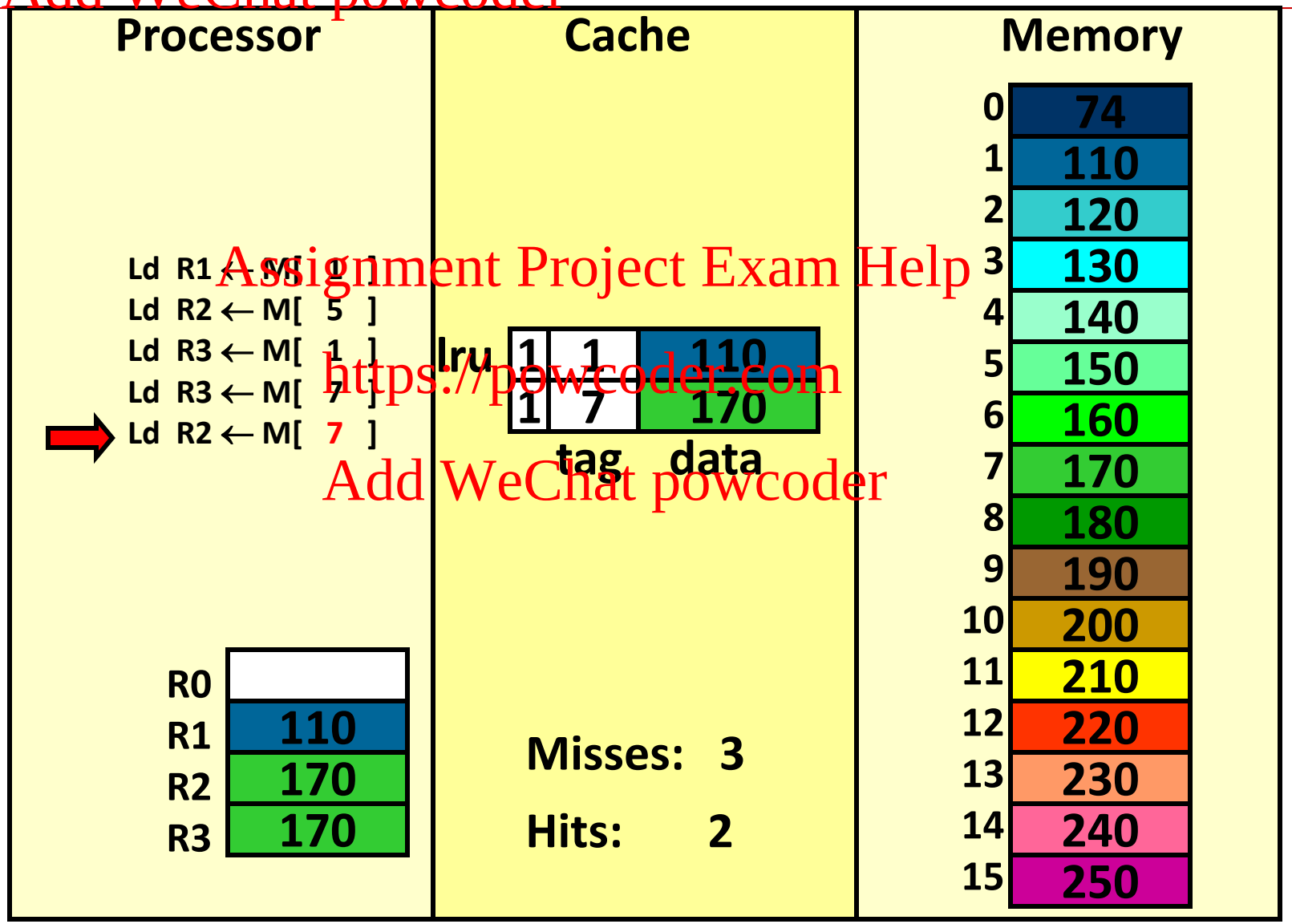
A Very Simple Memory System



A Very Simple Memory System



A Very Simple Memory System



Part 5: Cache Performance and Area Overhead

Calculating Average Memory Access Time (AMAT)

AMAT = cache latency × hit rate + memory latency × miss rate

Simple cache example: 3 misses, 2 hits

Assume following latencies:

Cache: 1 cycle

Memory: 15 cycles (assume it includes time to determine cache hit/miss)

AMAT for our example cache

= 1 cycle × (2/5) + 15 × (3/5)= 9.4 cycles per reference

AMAT: Example Problem

Assume the following latencies:

Cache	1	cycle
Main memory	100	cycles
Disk	10,000	cycles

Assume main memory latency (100 cycles) includes time to determine hit/miss in cache.

Assume main memory is accessed on all cache misses, and that disk latency does **not** include time to determine hit/miss in cache.

Assume a program with these characteristics:

100 memory references

90% of the cache accesses are hits

80% of the accesses to main memory are hits

What is the average memory access time (AMAT)?

$$0.9 * 1 + 0.1 * (100 + 0.2 * 10000) = 210.9$$

Reducing Average Memory Access Time

Reduce latency of cache, main memory, disk and/or

Increase hit rate of cache and main memory

Calculating Area Cost

How much does our example cache cost (in bits)?

- Calculate storage requirements
  - 2 bytes of SRAM
- Calculate overhead to support access (tags)
  - 2 4-bit tags
  - The cost of the tags is often forgotten for caches, but this cost drives the design of real caches
  - 2 valid bits

What is the area cost if a 32-bit address is used?

Next lecture: How can we reduce the area cost?

Have a small address.

- Impractical, and caches are supposed to be micro-architectural

Solution: Cache bigger units of data larger than bytes

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

To Be Continued…