

Cache memory simulator

Chiorean Bianca Anamaria An3 gr3

Contents:

[1 Introduction 3](#_Toc149497442)

[1.1 Project proposal 3](#_Toc149497443)

[1.2 Project plan 5](#_Toc149497444)

[2 Project proposal/ Bibliographic study 8](#_Toc149497445)

[2.1 Locality of reference (Principle of locality): (very important) 8](#_Toc149497446)

[2.2 Fully associative addressing mode 9](#_Toc149497447)

[2.3 Some of the replacement algorithms 10](#_Toc149497448)

[2.4 Fully Associative Cache 11](#_Toc149497449)

[2.5 DIRECT MAPPED CACHES 12](#_Toc149497450)

[2.6 Set-Associative Caches (middle solution/ Balanced) 13](#_Toc149497451)

[2.7 Set of flags associated with cache 16](#_Toc149497452)

[**2.8** **Cache write policies:** 16](#_Toc149497453)

[3 Analysis 17](#_Toc149497454)

[4 Design 17](#_Toc149497455)

[4.1 Design functional part (how cache works) 18](#_Toc149497456)

[4.2 Design user interface 18](#_Toc149497457)

[5 Implementation 23](#_Toc149497458)

[6 Tests/Experiments 24](#_Toc149497459)

[7 Conclusions 34](#_Toc149497460)

[8 Reference list 35](#_Toc149497461)

# Introduction

## Project proposal

Project Overview

The project aims to develop a simulation application that provides a graphical representation of multiple cache memory types and demonstrates how data within the cache changes in response to CPU requests, both reads and writes. This tool is primarily designed for educational purposes, particularly for computer science and engineering courses. It serves as a bridge between theoretical knowledge and practical application, offering a hands-on, customizable, and visually engaging cache memory simulation. Users will be able to select different cache association map types using buttons. The chosen programming language for this project is Java, and development will be carried out within the IntelliJ integrated development environment (IDE).

Project Proposal

Project Overview

The project aims to develop a simulation application that provides a graphical representation of multiple cache memory types and demonstrates how data within the cache changes in response to CPU requests, both reads and writes. This tool is primarily designed for educational purposes, particularly for computer science and engineering courses. It serves as a bridge between theoretical knowledge and practical application, offering a hands-on, customizable, and visually engaging cache memory simulation. Users will be able to select different cache association map types using buttons. The chosen programming language for this project is Java, and development will be carried out within the IntelliJ integrated development environment (IDE).

Objectives

Cache Memory Simulation: Create a user-friendly simulation application that emulates the behavior of different cache memory types.

Graphical Representation: implement a graphical interface that visually represents cache structures and demonstrates how data is managed within the cache in response to CPU requests.

Educational Focus: The primary purpose of this tool is to facilitate learning in computer science and engineering courses. It will help students understand the fundamental concepts of cache memory and its impact on computer systems.

Customization: Enable users to configure cache parameters and choose between various cache association map types to explore different cache memory scenarios.

Intuitive User Interface: Design an intuitive and accessible user interface that allows users to interact with the simulation effortlessly.

Real-World Relevance: Emphasize the practical application of cache memory concepts, highlighting their significance in improving computer system performance.

Benefits

This simulation application will offer a valuable educational tool for students and educators to gain a deeper understanding of cache memory concepts.

It provides a practical means for students to experiment with different cache configurations and memory access patterns, reinforcing their learning.

By visualizing cache memory behavior and allowing for customization, the project will bridge the gap between theoretical knowledge and its real-world application.

The tool's versatility and compatibility with standard development environments make it accessible and widely applicable.

## Project plan

Phase 1: Initial Preparations (week 1)

Workload Planning: Develop a detailed project schedule, allocating time and resources efficiently.

Phase 2: Research and Learning (week 2)

Cache Memory Principles: Thoroughly study the fundamentals of cache memory, including its purpose and functionality.

Cache Memory Types: Investigate different cache memory architectures and understand their unique characteristics.

Phase 3: Design (week 3)

User Interface Design: Create a user-friendly and modifiable interface that accommodates user configuration.

Graphical Representation: Design an intuitive graphical interface that visually illustrates changes in both main memory and cache memory.

Data Structures: Choose appropriate data structures for representing main memory and cache memory efficiently.

Phase 4: Implementation (week 4 and week 5)

Application Development: Implement the user interface and the functional components of the cache memory simulation.

Functional Testing: Rigorously test the application's functionality to ensure accuracy and reliability.

Phase 5: Object-Oriented Modeling (week 4 and week 5)

Java Classes: Model Java classes for main memory, cache memory, and follow the Model-View-Controller (MVC) design pattern for clean architecture.

Phase 6: Documentation

System Diagrams: Create comprehensive system diagrams to illustrate the relationships and interactions between components.

Technical Documentation: Write detailed technical documentation, including code documentation and system specifications.

Source Documentation: Carefully document the sources used for reference and research throughout the project.

Phase 7: Problem Solving

Adaptation: Address and overcome any technical or conceptual challenges encountered during development.

Phase 8: Conclusion

Project Conclusion: Summarize the project's outcomes, lessons learned, and future recommendations.

Week1:

- plan the project development on weeks

- propose the project

- make refence list

Week2

? bibliographic study

* Grasp the principles of cache memory
* Establish the data types used for each cache memory

Week3

- analyze usescases

Week4 and week 5

- develop user interface

- model the classed according to modeled objects

- develop the back end application

Week6

- finalize debugging

- prepare presentation

Week7:

-present the final project

# Project proposal/ Bibliographic study

Cache memory is a buffer between processor and main memory (see memory hierarchy)

The memory of a computer is structured as a pyramid. The fastest and the smallest memory type is the register. Under the register comes the cache memory which will be discusses in detail, followed by the main memory and after that comes long term storage.

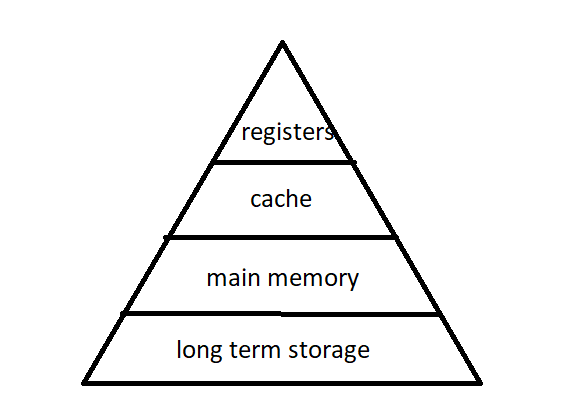


Figure :Memory hierarchy

Miss = element was not found at that memory level

Hit = element was found at that memory level

Hit Rate = percentage of time that element was found at that memory level

## Locality of reference (Principle of locality): (very important)

- once executed or accessed, an element from memory will be used again soon more than likely

- Temporal locality (example: loops)

- neighbors will also be used soon

-spacial locality ( body of a loop)

=> miss, go out to main memory, grab not only the used element with its neighbors

Executing one instruction after the other=> sequential locality

How cache works:

Due to the principle of locality, an instruction fetched from memory, it’s neighbors will PROBABLY be used to and that instruction probably will be reused in the close future.

To save up on resources, the whole block will be put into the cache memory’s desired space.

## Fully associative addressing mode

A block can be stored at any line

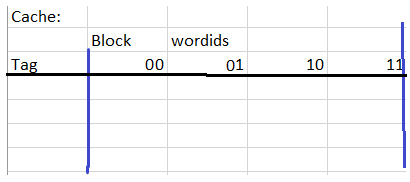
The Full Address is divided into Block id and Word id

Block id = which chunk of memory is

Word id =which element of the taken block is

Fully Associative = take block id and store as tag

There is exactly one line (which is block id) or no line => go to next level (Main Memori) to get data

Figure :Cache Fully Associative

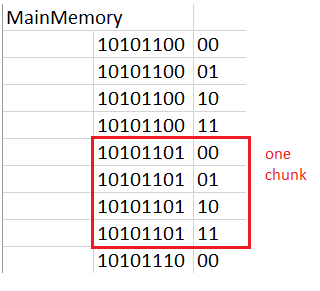


Figure :Main Memory

The remaining part of the FULLADDRESS without the wordID is stored in cache as the tag in the empty space or based on the replacement algorithm (in this project, the chosen algorithm is RANDOM).

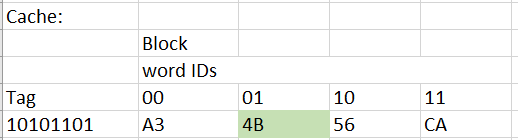


Figure :Cache Fully Associative

No more room in the cache?

Solution is to throw something out => Replacement Algorithms

In this project will be implemented the Random Replacement algorithm due to its cheaper implementation, it’s not heavy for the battery. Compared to the other replacement algorithms. In simulation performs as good as the others.

## Some of the replacement algorithms

1)FIFO (first in first out)

- replace existing block

- create a list of the existing cache entries, first one added, first one ejected

There is a problem: just because is loaded first, doesn’t mean we don’t need it (example: operating system’s runtime)

2) Least recently used

- replace block that’s been least recently used

- create a timer that resets to zero, increment all the others, remove max when cache is full)

- draw back: takes time to reset all the times, to search for the timer to reset and takes time to increment all the others, also memory is also used

3) LFU

- replace block least frequently used

-Problem: nor good in multitasking

- is good for embedded systems

3)Random

- just remove one random element

- memory used: variable to stored the random generated number

In this project will be implemented the Random Replacement algorithm due to its cheaper implementation, it’s not heavy for the battery. Compared to the other replacement algorithms. In simulation performs as good as the others.

PROBLEM

-expansive

-slow

## Fully Associative Cache

Benefits:

-least chase of trashing (throwing something, going back to main memory than bringing it back multiple times)

Drawbacks:

-expensive:

-compute Block id to all the Tags

-slow

## DIRECT MAPPED CACHES

* Specify exactly which line of the code to store block in
* There is no replacement algorithm

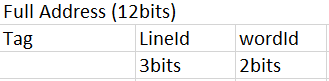


Figure :Full Address

The full address is divided into

* Word id
* Line id
* Tag

1. Writing in the cache
   1. Based on the Line id, the block of memory will added to the cache
      1. If there is no existing element in the cache at that line,
         1. Just put it at that line
      2. Else
         1. Just overwrite what is in the line
2. Reading from the cache
   1. Go to line number equal to line id (of the full address)
   2. Compare the tag that is already there (check if there is something written at tag in that line) with the tag from Full Address
      1. If it’s a match
         1. return the address corresponding with the word id
      2. Else
         1. miss

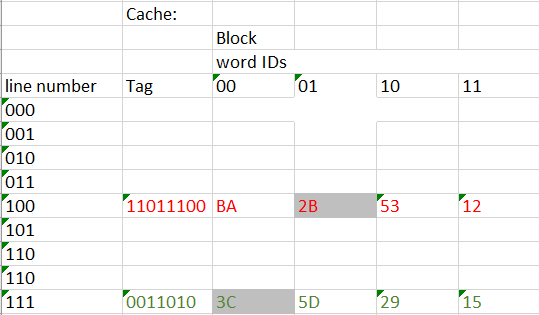


Figure : Direct Mapped Cache Memory

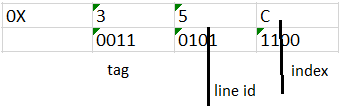


Figure : Address

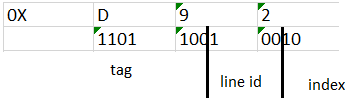


Figure :Address

Direct mapping is cheap and fast BUT has a problem with data crashing

PROBLEM:

* Trashing (throws out data, then brings it back and again and again)

## Set-Associative Caches (middle solution/ Balanced)

Now we have a mini associative cache (for each set)

Full Address broken down into: Tag, Set Id, word id

Example on 12 bit address

Based on set id:

If both slots from given set empty:

Pick first one

If same set id

1. Case1
   1. Compare if exists, one line from set taken🡪add to next one (load it)
2. Case 2 (Set is full, no match) => replacement algorithm
   1. Based on replacement algorithm, throw a row out and replace it with the wanted block

Only 2 lines per set => easy to use a replacement algorithm

Larger and large set, get closer to fully associative

k-way associative ( k = nr lines per set)

Too much trashing?

>make sets bigger

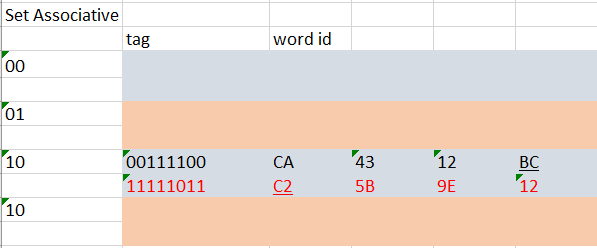


Figure : Set-Associative Mapping

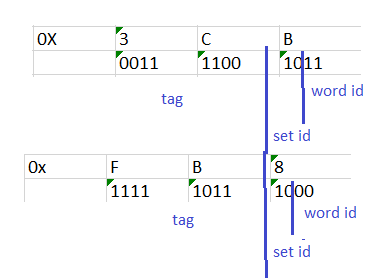


Figure :Address

Bigger size sets example:

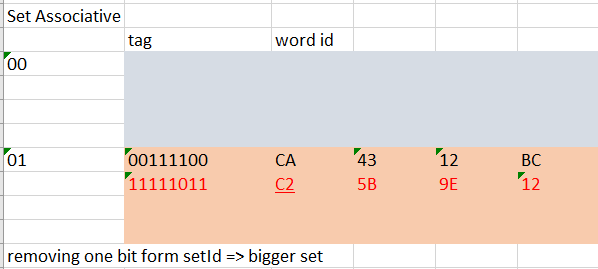


Figure :Set-Associative Cache Memory

## Set of flags associated with cache

* Type: data/ instruction
  + Instructions: decode them, never write to them
  + Data: does not get executed, can write to it
  + Block should contain only data or only instrutctions
* Valid
  + Whether the line contains valid elements of memory
    - Start up cache:
    - All lines should be invalid because the cache is empty
  + Easy way to start over the cache (delete everything ) is to set all the V flags to invalid
* Lock
  + Lock a line of the cache to prevent replacement (don’t replace operating system)
* Dirty bit
  + Identifies data that has been written to but not updated in main memory

## **Cache write policies:**

* Write Through
  + Don’t look at dirty bit
  + Every write to the cache also writes through to main memory
    - Increases bus trafic
  + Important to shared memory
* Write Back
  + Update main memory only when a dirty line is to be replaced
    - Reduced bus traffic

Unified cache (flexibility) or split data

# Analysis

In this project will be implemented direct mapped cache, fully associative cache and set-associative (chosen replacement algorithm is random)

Set associative mapping can be transformed into direct mapping (n-ways: every line is a set, (set id becomes line id) and 1-way: whole memory is a set)

## Functional specifications

User can:

1. Choose the type of cache memory.
2. Choose the size of the address
3. Choose the size of the cache
4. Choose the write policy
5. See the effect of different CPU requests
6. Track performance metrics such as cache hit rate, cache miss rate.

## Interaction between user and application

User chooses the desired addressing mode and configurations via the combo boxes.

# Design

Architecture:

The application will follow the Model, View, Controller (MVC) architecture.

The View package contains:

* The window that contains the start view
  + User has the ability to choose:
    - Addressing mode
    - Address size
    - Cashe size
    - Write policy
  + User can press Start button to begin the simulation
* The window that contains the simulation will display the simulation

The Controller package contains the logic behind the view, handling user’s input.

The Model package contains the models of the data, classes that represent the cache memories, the addresses, the main memory and the logic behind it.

* Cache memories (fullyAssociative and direct mapping ) inherit the CashMemory, adding their particular fields
  + direct mapping adds line id field
  + set-associative add set id field

## Design functional part (how cache works)

## Design user interface

User can choose from the combo boxes the desired configuration and press start to start the simulation.

The main memory will be preloaded with addresses. The CPU too will have preset requests whose effects will be observed during the simulation.

A screenshot of a computer

Description automatically generated

Figure :User Interface

Simulation view to be implemented.

A screenshot of a computer

Description automatically generated

Figure :SimulationView

Simulation view will present main memory next to the chosen cache memory to offer the ability to observe the changes.

The Cpu requests, Main memory and Cache memory are presented on one page, each area has a scroll bar.

How to break the address apart:

String input = "123456789";

int lengthS1 = 3;

int lengthS2 = 3;

int lengthS3 = 2;

if (input.length() == (lengthS1 + lengthS2 + lengthS3)) {

String s1 = input.substring(0, lengthS1);

String s2 = input.substring(lengthS1, lengthS1 + lengthS2);

String s3 = input.substring(lengthS1 + lengthS2);

System.out.println("s1: " + s1);

System.out.println("s2: " + s2);

System.out.println("s3: " + s3);

} else {

System.out.println("The sum of desired substring lengths does not match the length of the input string.");

Used Algorithms:

Set-Associative addressing mode:

1. CPU requests an address
2. Fetch that address and the whole block
3. Full Address broken down into: Tag, Set Id, word id
4. Add to Set-Associative cache memory
   1. Go to corresponding set:
      1. All set is empy
         1. Just add the block
      2. Exists free space (some ocupied)
         1. Case1
            1. Compare if exists, one line from set taken🡪add to next one (load it)
         2. Case 2 (Set is full, no match) => replacement algorithm
            1. Based on replacement algorithm, throw a row out and replace it with the wanted block

Adapt set-associative to be direct mapped cahe(each line is a set) and to be fully associative (the whole cache is set).

Class diagram:

A diagram of a network

Description automatically generated

Figure :CLass Diagram

A diagram of a computer

Description automatically generated

Figure : Complex Class Diagram

Use case diagram:

A diagram of a person with pink circles

Description automatically generated

Figure :Use Case diagram

# Implementation

Set-Associative Cache is the general cache type. The main difference between the caches is the constructor, mainly how the number of bits are allocated to the tag, setId, wordId.

The two main method of the cache is the writeData (in: address, nrHits, nrMisses, out: message)and the readData(in: address, nrHits, nrMisses, out: message). Both of these methods, in case of not finding the requested address in the cache, addToCache(in:address, out:message) method is called. Together they create a string that informs the user of the changes. In these two methods, writeData and readData, the writing policy is evaluated and executed accordingly. The number of hits and missed are counted.

The method replaceRandom(in: address, startSet, sizeSet, chunkAndLine) is a particular implementation of the replacement algorithms, in this case random. Based on the chosen cache, it creates an empty space in the cache, and it fills with the wanted address and it apples the write policy.

The method convertBinaryToHexa(in: binaryString, out: hexString) converts a binary string to hexString for better visibility and ease of understanding during the simulation.

The method prettyPrintCache(out: String) creates a sting with all the cache’s elements: universal cache line, set number, line of set number, tag and the words.

The Main Memory is represented not as whole but as contiguous chunks of addresses, snippets, it is an array of arrays.

The Simulator class orchestrates the simulation by creating processes. Creates the main memory, the random data, the cache and the processes. By printing the request, the information about the cpu request, cache memory, main memory, hits and miss will be printed in the corresponding text area and the request will be executed before the prints.

The Process class executes the requests of the processor on the cache and main memory. In order for a process to do the work, it needs to be provides with the address, request type (0 for read and 1 for write), the writeData (“none” in case of read and the actual data in case of write, the cache and the view where the messages will be printed).

# Tests/Experiments

Test were conducted during the development of simulator, on the simulationView via messages and hand checking. Later, automated testes were implemented and the type of data creation was modified (initially the data was generated random, now, for testing purposes , the data is the hex representation of the line number in the contiguous memory chunk). The order of the simulationView tests has no correlation to the coded tests.

Cpu Requests are the request set by cpu and the answer received after the operation.

Cache Memory is the whole cache memory, initially empty.

Configuration for test#1: A screenshot of a computer game

Description automatically generated

Figure :Configuration for test 1

Read address 433

A screenshot of a computer

Description automatically generated

Figure : T1 Read address 433

It can be observed that the data from the main memory is in the cache at words in the right order, a miss is counted correctly. The correct tag is added to the cache, at the right set.

A computer screen shot of a computer program

Description automatically generated

Figure : Read from 432

A computer screen shot of a computer

Description automatically generated

Figure :T1 Request3

A screenshot of a computer

Description automatically generated

Figure : T1 Request 4

A screenshot of a computer

Description automatically generated

Figure : T1 Request 5

Dirty bit is set for line zero of the cache, shortly before the next change suffered the changes will be transferred to main memory, now the main memory is identical to the start.

A screenshot of a computer

Description automatically generated

Figure :T1 Request 6

It was requested to modify line 0 cache, wordId 11. The value is modified in cache with the lates request, and the last value (99) is written in the main memory.

Test 2

A screenshot of a computer

Description automatically generated

Figure : Configuration for test 2

A computer screen shot of a computer

Description automatically generated

Figure : T2 Request1

Expected: data from corresponding main memory address to be transmitted to cache

Result: as expected.

A computer screen shot of a computer program

Description automatically generated

Figure : T2 Request2

Expected: read data form 432 address, word id = 10 from set 00

Result: as expected.

A computer screen shot of a computer

Description automatically generated

Figure : T3 Request 3

The modified data is written directly into main memory (before was ed, now is 77).

Result as expected.

A computer screen shot of a computer

Description automatically generated

Figure : T2 Request 4

Expected: add to cache requested address with the corresponding data.

Result: as expected

A screenshot of a computer

Description automatically generated

Figure :T2 Request 5

Expected: write data at main memory [1][51].

Result: as expected.

A computer screen shot of a computer

Description automatically generated

Figure : T2 Request 6

Expected: write data at main memory [1][51].

Result: as expected.

Manual Test 3:

A screenshot of a computer game

Description automatically generated

Figure : T3 Configuration

A computer screen shot of a memory

Description automatically generated

Figure : T3 Request1

Expected: based on provided address 433 data from main memory to be added to cache line 12 and data wordId 11 to be read.

Result: as expected.

A computer screen shot of a computer

Description automatically generated

Figure : T3 Request3

A screenshot of a computer

Description automatically generated

Figure : T3 Request 5

Result: as expected. The main memory will not be updated util the cache line 12, wordid 11 will be changed again.

A screenshot of a computer

Description automatically generated

Figure : T3 Request6

Results as expected.

The automatic tests contain different addresses, and different for all cache types. Test1 does not work on Direct Mapped (unable to populate the whole cache with the available data).

The automatic test 9 failed for Set-Associative 4-Way and fully associative. Exception was thown at a print line, presuming some problem with the conversion form binary to hex.

The automatic test for write polies fail due to an index out of bounds.

Pach note:

Modified the type of mainMemory it operates on (before were snippits on main memory, now is a contiguous main memory).

Added tests for checking the null data in the caches after being fully populated.

Addes tests for the correct tag in at the right tag line after caches were fully populated, plus, in the same manner added data tests, the cache should contain the right data.

All new tests have passed. The content of the caches and each instruction can be read in the terminal.

The data in the main memory is generated as consecutive numbers starting form zero.

The expected data that should be in caches (tag and words) are stored in separate variables.

Met Bugs:

Difficulties adapting set associative cache to direct mapped and fully associative due to null pointer exception: resolved by checking for null pointers.

Cache addresses were not compared correctly due use of == and not .equals().

Hard time implementing the writing policies: data would not be transmitted when the main memory before overwriting. Resolved by writing first to the main memory than writing to cache.

Manage to populate the whole cache addresses (for set associative, fully associative) but the data in some cases was not transmitted to the cache data. Resolved by adding one (for direct mapped cache was correct the original formula).

# Conclusions

In conclusion, the outlined project creates a simulation application with a focus on educating computer science and engineering students about cache memory concepts. By developing a user-friendly tool in Java within the IntelliJ IDE, the project aims to provide a visually engaging experience that bridges theoretical knowledge and practical application. Through graphical representation, customization options, and an intuitive interface, this cache memory simulation offers a valuable educational resource, allowing users to experiment with different configurations and access patterns. The project's emphasis on real-world relevance, compatibility, and versatility positions it as an impactful tool for enhancing understanding and application of cache memory principles in academic settings.

Possible improvements: adding more replacement algorithms, more address sizes.

# Reference list

* Wikipedia, Cache (computing) <https://en.wikipedia.org/wiki/Cache_(computing)>
* Intermation, Introduction to Cache Memory <https://www.youtube.com/watch?v=Bz49xnKBH_0>
* Intermation , Fully Associative Caches and Replacement Algorithms <https://www.youtube.com/watch?v=A0vR-ks3hsQ>
* Intermation, Direct Mapped Caches <https://www.youtube.com/watch?v=zocwH0g-qQM>
* Intermation, Set-Associative Caches <https://www.youtube.com/watch?v=gr5M9CULUZw>
* Cache simulator <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjylsG64uqBAxXggf0HHe4lBSsQFnoECA0QAQ&url=https%3A%2F%2Fcourses.cs.washington.edu%2Fcourses%2Fcse351%2Fcachesim%2F&usg=AOvVaw0gTxrkBJgfKjMkZdWGK9ND&opi=89978449>
* Make Diagrams: https://www.drawio.com