# University of Windsor

# Electrical and Computer Engineering

# 06-88-448 Digital Computer Architecture

# Instructor: Roberto Muscedere

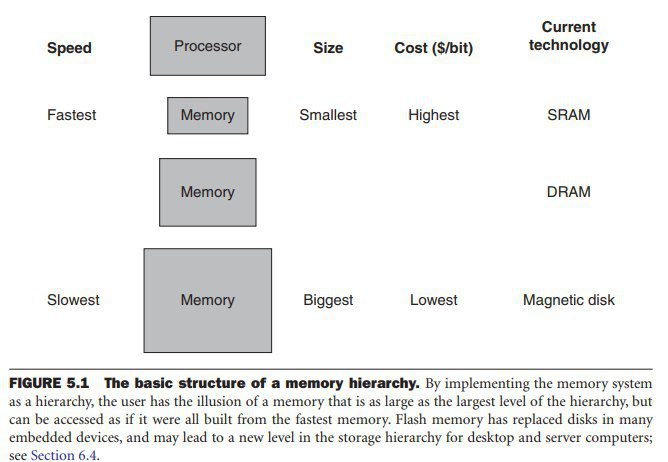
# Assignment #5: Simulate One Level of Cache Performance

# Daksh Patel           ID: 104 030 031

# Due Date: Mar 30th, 2018

# Introduction

The objective of this assignment is to simulate one level of cache performance to determine which configuration would lead to the least amount of CPU stalls. Sample set of memory traces (MATs) were given to analyze. These MATs were used to simulate the memory addressing caching deals with. The programming language used to implement the assignment was Python 3, which was chosen due the available documentation and community that allowed learning it an easier task than any other language would have allowed. Performing this assignment using C would have lead to very incomplete code, whereas Python allowed some leeway in terms of coding due to easier handling of variables. Opening the sample set showed that every value and letter indicating read or write were in its own line. This allowed the capability to read the values without too much changes made. Each line either begins with an ‘R’ for read or ‘W’ for write. These texts files can be opened with VSCode text editor due to familiarity. Cache is a communication method between the CPU and slower memory and acts as a way to quickly references the more used data. SRAM is one such fast memory. But caching methods can be utilized in various mediums to increase the responsivity of a task in modern computer, such as disk caching which uses the same method as CPU cache but on the slower disk medium.



# Procedure

1. To understand the caching mechanism various resources needed to be utilized. Instructional videos, course textbook, course notes and online websites. A basic understanding was created and one by one methods on how to implement into code were theorized.
2. Beginning with initially something as simple as reading the files themselves and separating the operation letter from the rest of the numbers. The numbers were converted to binary using the format function which was utilized in such a way that allowed padding with zero values. Otherwise the values would have not been full 32bit values.
3. A basic without any extra functions was developed. This implementation involved reading values and identifying hits and misses with direct mapping.
4. Extra functions were added such as selecting associativity and least recently use (LRU). Write Back was added after this but not completely.
5. A table with values such as show in the Figure below is provided to show the results.

# Results

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **MAT** | **rowbits** | **blockbits** | **associativity** | **total cycles** | **RAM size** | **hits** | **misses** | **write-backs** |
| quick4k | 6 | 0 | 1 | 1486950 | 3712 | 123933 | 64650 | 64650 |
| quick16k | 6 | 0 | 2 | 6777042 | 7424 | 575581 | 294654 | 294654 |
| quick64k | 6 | 0 | 4 | 32094522 | 14848 | 2704100 | 1395414 | 1395414 |
| quick256k | 6 | 0 | 8 | 138543536 | 29696 | 12018917 | 6023632 | 6023632 |
| random4k | 6 | 0 | 16 | 585465 | 59392 | 179345 | 25455 | 25455 |
| random16k | 6 | 0 | 2 | 2358512 | 7424 | 716656 | 102544 | 102544 |
| random64k | 6 | 0 | 2 | 9427723 | 7424 | 2866899 | 409901 | 409901 |
| random256k | 6 | 0 | 2 | 37718689 | 7424 | 11467257 | 1639943 | 1639943 |
| quick1M | 6 | 0 | 2 | 587077622 | 7424 | 52229141 | 25525114 | 25525114 |
| random1M | 6 | 0 | 2 | 150680406 | 7424 | 45877478 | 6551322 | 6551322 |

# Discussion

There are different types of memory that work at different speeds and have large differences in sizes due to cost. It is easier to develop code to communicate between smaller fast memory and slower large types of memory rather than using the fastest for everything. There are techniques to allow quick access of commonly used data. Such as caching. Caching can be used for all mediums of memory but when related to CPU is the closest and fastest available storage medium. The caching method used for this assignment is to store the most recently used data in cache for easy referencing and write back the data over to another location, which would simulate another level of memory in the hierarchy. SRAM and DRAM can be used as examples.

The basic caching systems require the use of tag bits, row/index bits and offset bits. Valid and dirty bits can be utilized. LRU means the least recently used value is kicked out of the cache. Write back uses a dirty bit as an indicator of value to be replaced, that block is copied over and then evicted and overwritten. Direct mapped cache maps to each address to a unique location and if its not there previous then the block that compares with the index bits is overwritten. In a set associative cache there is a fixed number of locations where each block can be placed. Fully set associative would be a one to one mapping of the cache.

For the purposes of this project the following code seems to perform the given tasks to a suitable degree. More testing can be completed for this project but due to the inherent limitations associated with something brand new the complete experience in python and theory was not available at this time. In the below Appendix section there are several definitions for different components in the cache. Along with diagrams with how this caching system is supposed to work.

# Conclusion

This project was theory intensive and required knowledge of a suitable programming language. The attached code should be capable of the minimum requirements of this assignment and provides results in suitable manner. It requires some variable adjustments such as file location and associativity. The current code does not properly show the write-backs but it is assumed if further testing were performed then it would be possible to correct. It is assumed the error is a location of the counter issue and a checking system. The equations for the various required values were given in the assignment documentation and included in the code to calculate the total cycles. The results show similar values as the professor provided values. It is not exact so there is room for improvement. Overall, much was learned for programming and cache theory. The understanding of both can be improved.

# Appendix

## Code

#//////////////////////////////////////////////////////////////

# DAKSH PATEL #

# 104030031 #

# 104030031\_Assign5.py #

# Simulate Cache #

#//////////////////////////////////////////////////////////////

import math

#Plenty of arrays to hold values and grab bits from

RW\_Indicator= []

validBits = []

dirtyBits = []

rowBits = []

dataArray = []

WBArray = []

LRU = []

tempVar = []

#Global Variables for some specific calculations and initializing to zero

blockSize = 64

tag = 24

row = 6

block = 0

ofst = 2

hits = 0

misses = 0

cycles = 0

WB = 0

associativity = 1 #Select 1,2,4,8,16

#init to zero

def initToZero():

for x in range(0, blockSize):

validBits.append(0)

dirtyBits.append(0)

LRU.insert(x, [])

dataArray.insert(x, [])

WBArray.insert(x, [])

for y in range(0, associativity):

LRU[x].append([])

dataArray[x].append([])

WBArray[x].append([])

y = y

# Open one file at a time and run sim

quickFile = open('quick/quick4k.txt', 'r')

# quickFile = open('quick/quick16k.txt', 'r')

# quickFile = open('quick/quick256k.txt', 'r')

# quickFile = open('quick/quick64k.txt', 'r')

# quickFile = open('quick-big/quick1M.txt', 'r')

# quickFile = open('random/random4k.txt', 'r')

# quickFile = open('random/random16k.txt', 'r')

# quickFile = open('random/random256k.txt', 'r')

# quickFile = open('random/random64k.txt', 'r')

# quickFile = open('random-big/random1M.txt', 'r')

initToZero() #Initialize values to zero, call function

for value in quickFile.readlines() :

RW\_Indicator = value[:2] #Specify the Read or Write letter location

valueInt = int(value[2:]) #Specify non RW letters and only numbers

data = f'{valueInt:032b}' #Format numbers to binary data

tagBit = data[0:24]

rowBit = data[24:30]

#block not used otherwise it would be here in order

ofstBit = data[30:32]

location = int(rowBit, 2)

if('R' in RW\_Indicator): #Read

hits += 1

if(data[0] in dataArray):

cycles += 1 + int(rowBit, 2)/2 + math.log2(associativity)

if(associativity == 1):

pass

else:

#LRU

col = LRU[int(rowBit, 2)].index(tagBit)

tempVar = ([LRU[int(rowBit, 2)][col]])

LRU[location][0].insert(0, tempVar)

LRU[location][0].pop(associativity)

#Cache - Dirtybit changes, line allocation/writing/evicting

dirtyBits[int(rowBit, 2)] = 0

validBits[int(rowBit, 2)] = 1

col = dataArray[int(rowBit, 2)].index(data)

dirtyBits[int(rowBit, 2)] = 1

dataArray[int(rowBit, 2)].insert(0, dataArray[int(rowBit, 2)])

dirtyBits[int(rowBit, 2)] = 0

validBits[int(rowBit, 2)] = 0

dataArray[int(rowBit, 2)].pop(col)

else:

misses += 1

cycles += 20 + 2\*\*block

validBits[int(rowBit, 2)] = 1

dirtyBits[int(rowBit, 2)] = 0

if(associativity == 1):

dataArray[location][0] = data

LRU[int(rowBit, 2)][0] = tagBit

# WB += 1

else:

dirtyBits[int(rowBit, 2)] = 0

LRU[location].insert(0, tagBit)

LRU[int(rowBit, 2)].pop()

dirtyBits[int(rowBit, 2)] = 1

WBArray[location][associativity-1] = dataArray[location][associativity-1] # copy to writeback array before overwriting cache location

# WB += 1

dataArray[location].insert(0, data)

dataArray[int(rowBit, 2)].pop()

dirtyBits[int(rowBit, 2)] = 0

validBits[int(rowBit, 2)] = 0

if('W' in RW\_Indicator): #Write

if(data in dataArray):

cycles += 1 + int(rowBit, 2)/2 + math.log2(associativity)

if(associativity == 1):

pass

else:

#LRU

col = LRU[int(rowBit, 2)].index(tagBit)

tempVar = ([LRU[int(rowBit, 2)][col]])

hits += 1 + int(rowBit, 2)/2 + math.log2(associativity) #Hits counter

LRU[location][0].insert(0, tempVar)

LRU[location][0].pop(associativity)

#Cache - Dirtybit changes, line allocation/writing/evicting

dirtyBits[int(rowBit, 2)] = 0

validBits[int(rowBit, 2)] = 1

col = dataArray[int(rowBit, 2)].index(data)

dirtyBits[int(rowBit, 2)] = 1

dataArray[int(rowBit, 2)].insert(0, dataArray[int(rowBit, 2)])

dirtyBits[int(rowBit, 2)] = 0

validBits[int(rowBit, 2)] = 0

dataArray[int(rowBit, 2)].pop(col)

else:

cycles += 1 + 2\*\*block

validBits[int(rowBit, 2)] = 1

dirtyBits[int(rowBit, 2)] = 0

if(associativity == 1):

dataArray[location][0] = data

LRU[int(rowBit, 2)][0] = tagBit

WB += 1

else:

LRU[location].insert(0, tagBit)

LRU[int(rowBit, 2)].pop()

dirtyBits[int(rowBit, 2)] = 1

WBArray[location][associativity-1] = dataArray[location][associativity-1] # copy to writeback array before overwriting cache location

WB += 1 #increment writebacks

dataArray[location].insert(0, data)

dataArray[int(rowBit, 2)].pop()

validBits[int(rowBit, 2)] = 0

dirtyBits[int(rowBit, 2)] = 0

print('Valid Bit Array:', validBits)

print('Dirty Bit Array:', dirtyBits)

print('Write Back Array:', WBArray)

print ('LRU Array:', LRU)

print('Data Array: ', dataArray )

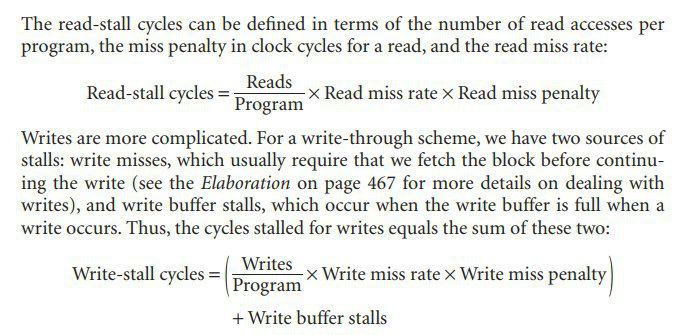
print( 'Tagbits: ', len(tagBit), ', rowBits: ', row, ', Blockbits: ,', block, ', Associativity Level: ', associativity)

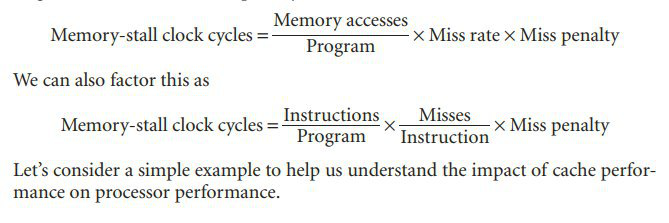
print('Total Cycles: ', cycles, ', SRAM Size:', associativity\*2\*\*6\*58)

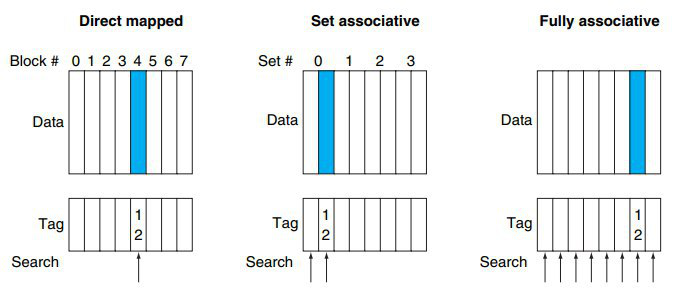
print('Hits:', hits, ', Misses: ', misses, ', Write Back Cost: ', WB)

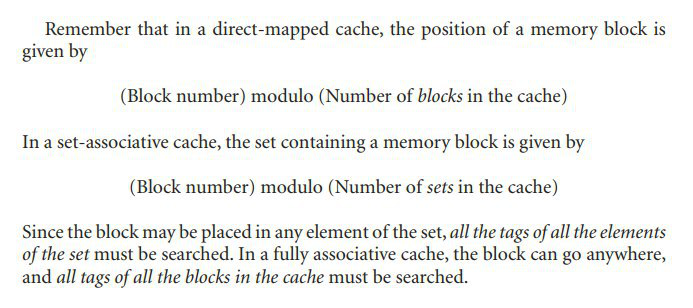
quickFile.close()

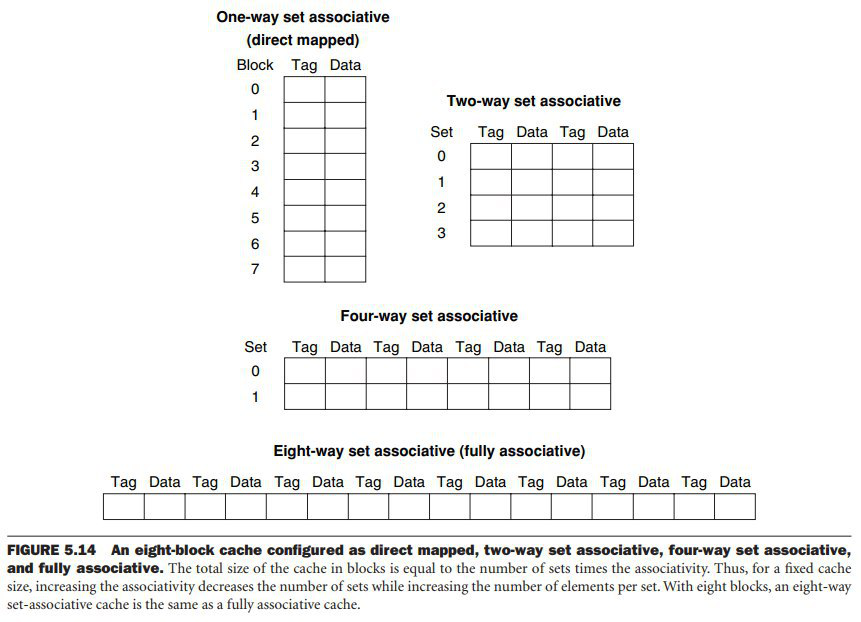
## Figures



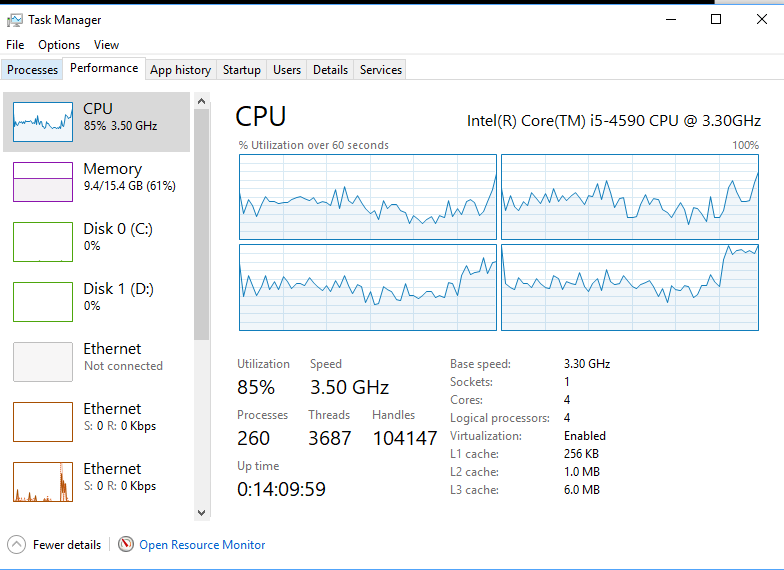




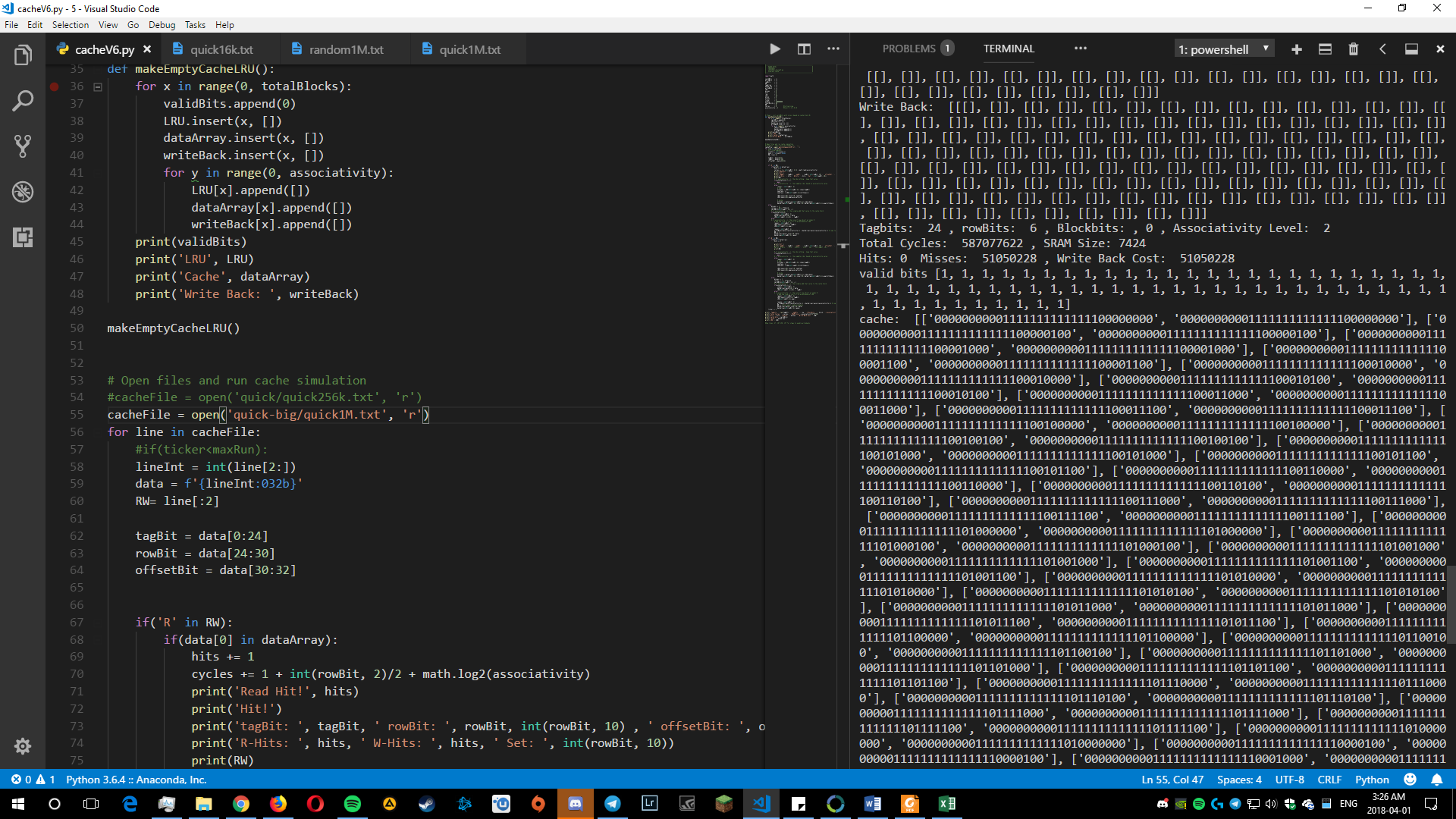




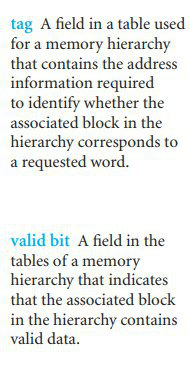
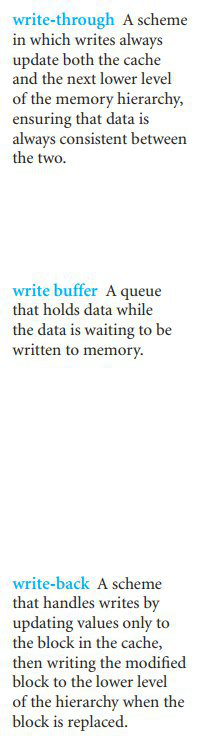
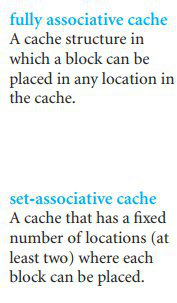
Running quick1M.txt

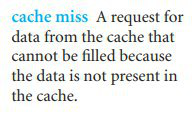


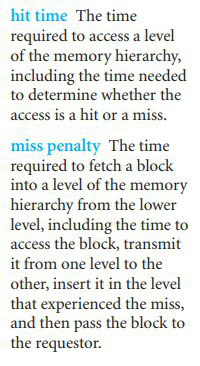


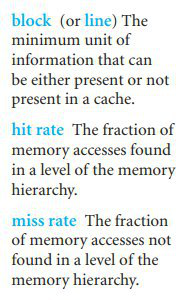


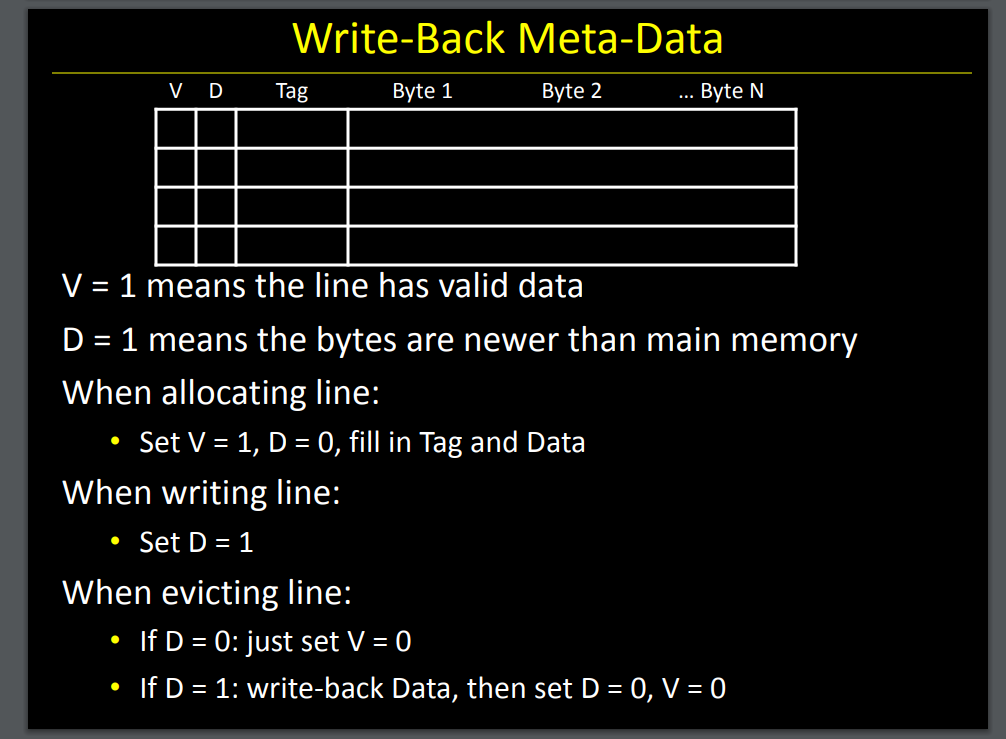
## Definitions











# References

Cs.cornell.edu. (2018). [online] Available at: http://www.cs.cornell.edu/courses/cs3410/2013sp/lecture/18-caches3-w.pdf [Accessed 29 Mar. 2018].

GeeksforGeeks. (2018). *LRU Cache Implementation - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/lru-cache-implementation/ [Accessed 29 Mar. 2018].

Patterson, D. and Hennessy, J. (2012). *Computer organization and design*. 4th ed. San Francisco: Morgan Kaufmann.

Pcguide.com. (2018). *Cache Write Policy and the Dirty Bit*. [online] Available at: http://www.pcguide.com/ref/mbsys/cache/funcWrite-c.html [Accessed 29 Mar. 2018].