MEMORY HIERARCHY

Development- Final Project

Computer Architecture Fall 2021 (16:332:563)

Lohitanvita Rompicharla
lr701@rutgers.edu
Rutgers University – NB
Dept: Electrical & Computer Engg

Avinash Suresh
as3516@rutgers.edu
Rutgers University– NB
Dept: Electrical & Computer Engg

Vrushali Dattatray Palande
vdp40@rutgers.edu
Rutgers University– NB
Dept: Electrical & Computer Engg

Abstract: In this project we have simulated Pipeline and Superscalar, Cache Organization and Associativity labs in simplescalar, developed a L1-L2 DRAM memory hierarchy system that handles cahce hit and cache miss for read and write. The main objective of this project is to analyse the memory hierarchy and its importance in accessing data faster and easier using cache organization and pipelining techniques.

Introduction:

Memory Hierarchy: In the Computer System Design, Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time. Since size of cache memory is less as compared to main memory. So, to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference. The two types of locality references are:

1. Temporal locality -- If a program accesses one memory address, it is likely that it will access the same address again.
2. Spatial locality -- If a program accesses one memory address, it is likely that it will also access other nearby addresses.

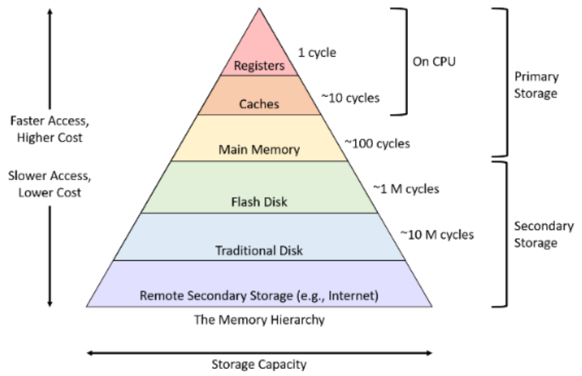


Figure 1: Memory Hierarchy Design

Problem Statement:

1. *Finish the regular lab (2/3/4/7) OR bonus labs (8/9) of SimpleScalar http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/*

2. *Develop a simple L1-L2-DRAM memory hierarchy system. L1 cache is direct mapping, and L2 cache is set associativity. Basic functions include handling cache hit and miss for read and write.*

- For Part 1 of the problem, we have chosen to simulate labs 2,3,4,7, that are about pipelining, superscalar, cache organization and associativity.
  These above mentioned topics are crucial part of memory hierarchy. The advantage of using **pipelining** is CPU works at higher frequencies than the RAM and overall performance of the CPU enhances, and **superscalar** helps achieve high performance by utilizing the hardware units to the maximum extent.
  More precisely, superscalar processors can start executing two or more instructions in the same cycle. Pipelined processors can execute more than one instruction at a time

- For Part 2 of the problem, we have developed a python code using NumPy and Pandas modules. In this part we have coded 2 python files l1cache.py and l2cache.py which depicts l1cache that uses direct mapping scenario and l2cache that displays set associativity. These program helps us analyze and understand the difference between direct mapping and set associativity. Also, how set associativity is better and flexible than direct mapping.

Technical Background and System Setup:

Technical Background: We have referred the class lectures and watched youtube videos on memory hierarchy, direct mapping, set associtivity for developing the code.

System Setup: **For Part 1**: Installed Ubuntu 1604 in VMware workstation 16 player on our system. Then downloaded and run simplescalar software tool for conduction simulation labs 2,3,4,7. These simulation labs are ran using **sim-outorder** and **sim-cache** simulators, where sim-outorder is a detailed microarchitectural simulator and sim-cache is a memory system simulator.

For part 2 of the project, we have downloaded Jupyter noteback and Pycharm community edition 2021.1.2 and installed packages numpy, pandas in Pycharm IDE for numerical computation and data frames for creating memory tables. The links and description for software is given in the last part.

Evaluation and Results:

NOTE: For reference we have the raw data files from SimpleScalar also converted them into text files for easy understanding.

1. PART I

1.1 Lab 2 /3 : *Pipelining and Superscalar using SimpleScalar (1)*

Lab Part 1: In pipelining and Superscalar using SimpleScalar (1), we copied the 'default.cfg' configuration from the simplesim-3.0/config folder, renamed it as 'config_a.cfg' and modified the values as follows:

- Decode: width = 1, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size =1, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 8, number of reservation stations.
- Issue: width= 1, maximum number of instructions that are issued in a cycle
- Memory ports= 1, read/write access to one memory address.
- All other ALU resources =1, number of integers, floating, multipliers/dividers ALUs
- Configuration: Inorder = true

Command: ./sim-outorder -config ./labs/config_a.cfg -ptrace ./config_a.trc 0:1024 -redir:sim sim_configa.out ./labs/test-math

Results and Comments:

Refer: "Part1_output files/simulation_output_textfiles/sim_configa_out.txt" for the detailed output. Also config_atrc.txt for tracing the pipeline.

1. Describe the Configuration?
   Ans: With this configuration as inorder = true, the processor fetches and runs multiple instructions from a thread of execution with the compiler generated order which has simpler implementation. But when there is a data dependency (RAW), it stalls the stage which has the dependency along with later stages irrespective of dependency. This slows the pipelining process.
2. What is the IPC of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for inorder configuration obtained is 0.5072
3. Is the forwarding implemented? To answer this question, consider the following two instructions. ag: sll r2, r16, 2 | ah: addu r3, r3, r2
   Ans: No, forwarding has not been implemented. If we see the above two instructions in the output window, we can see that Execution stage of instruction 'ah' has been waited till the Write-Back stage of instruction 'ag' is implemented.

1.2 Lab Part 2: In this part of lab 2, we copied the 'default.cfg' configuration from the simplesim-3.0/config folder, renamed it as 'config_b.cfg' and modified the configuration same as part 1) except Inorder = false, i.e.,

- Decode: width = 1, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size = 1, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 8, number of reservation stations.
- Issue: width = 1, maximum number of instructions that are issued in a cycle
- Memory ports = 1, read/write access to one memory address.
- All other ALU resources = 1, number of integers, floating, multipliers/dividers ALUs
- Inorder = False

Command: ./sim-outorder -config ./labs/config_b.cfg -ptrace ./config_b.trc 0:1024 -redir:sim sim_configb.out ./labs/test-math

Results and Comments:

Refer: “Part1_output files/simulation_output_textfiles/sim_configb_out.txt” for the detailed output. Also config_btrc.txt for tracing the pipeline.

1. Describe the Configuration?
   Ans: With this configuration as inorder = false, the processor executes outorder configuration. Out-of-order execution executes faster because it moves the dependent instructions out of way from independent instruction
2. What is the IPC of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for Outorder configuration obtained is 0.5437
3. Compared to part 1) or configuration A, what can you say about this configuration?
   Ans: In part 1) configuration is inorder with IPC 0.5072 and for part 2) it is Outorder configuration with IPC 0.5437. Therefore, the IPC for outorder is higher than inorder, it means more instructions are issued in one cycle for outorder improving the speed and performance of the system. Since there is no stalling for the independent stages in outorder unlike in inorder, this performance difference is observed.
4. The commit stage CT reorders the execution of instructions, explain why it is necessary.
   Ans: The results we get from the execution stage of the pipelining kept on hold at this commit stage. Depending on the exceptions, page fault or branches taken, this result might get discarded and whatever the data has been fetched can be bypass to the execution stage aligner by being a continuous feedback mechanism.

1.3 Lab 4 : *Pipelining and Superscalar using SimpleScalar (2):*

Lab 4 Part 1: In pipelining and Superscalar using Simplescalar (2), we copied the ‘default.cfg’ configuration from the simplesim-3.0/config folder, renamed it as ‘config_c.cfg’ and modified the values as follows:

- Decode: width = 1, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size =1, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 8, number of reservation stations.
- Issue: width= 1, maximum number of instructions that are issued in a cycle
- Memory ports= 1, read/write access to one memory address.
- All other ALU resources = 4, number of integers, floating, multipliers/dividers ALUs
- Configuration: Inorder = true

Command: ./sim-outorder -config ./labs/config_c.cfg -ptrace ./config_c.trc 0:1024 -redir:sim sim_configc.out ./labs/test-math

Results and Comments:

Refer: “Part1_output files/simulation_output_textfiles/sim_configc_out.txt” for the detailed output. Also config_ctrc.txt for tracing the pipeline.

1. What is the IPC (Instructions per cycle) of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for inorder configuration we obtained is 0.5072.
2. Compare this configuration and configuration in part 1 of previous lab. Does the added resources increase performance?
   Ans: Comparing the configurations of this lab and previous lab part 1, we see that there is change in the number of integer and floating ALU's, multipliers/dividers. There has been constant increase of resources from 1 to value 4. But this has brought no change in the output.

1.4 Lab 4 Part 2: In pipelining and Superscalar using Simplescalar (2), we copied the 'default.cfg' configuration from the simplesim-3.0/config folder, renamed it as 'config_d.cfg' and modified the values as follows:

- Decode: width = 1, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size =1, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 8, number of reservation stations.
- Issue: width= 1, maximum number of instructions that are issued in a cycle
- Memory ports= 1, read/write access to one memory address.
- All other ALU resources = 4, number of integers, floating, multipliers/dividers ALUs
- Configuration: Inorder = false

Command: ./sim-outorder -config ./labs/config_dcfg -ptrace ./config_d.trc 0:1024 -redir:sim sim_configd.out ./labs/test-math

Results and Comments:

Refer: "Part1_output files/simulation_output_textfiles/sim_configd_out.txt, sim_confige_out.txt, sim_configf_out.txt , sim_configg_out.txt, sim_configh_out.txt, sim_configi_out.txt" for the detailed output. Also config_dtrc.txt, config_etrc.txt, config_ftrc.txt, config_gtrc.txt, config_htrc.txt , config_itrc.txt for tracing the pipeline.

1. What is the IPC (Instructions per cycle) of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for outorder configuration we obtained is 0.5437.
2. Compare this configuration and configuration in part 1 of this lab. Does the out of order increase performance?
   Ans: In part 1) configuration is inorder with IPC 0.5072 and for part 2) it is Outorder configuration with IPC 0.5437. Therefore, the IPC for outorder is higher than inorder, it means more instructions are issued in one cycle for outorder improving the speed and performance of the system. Since there is no stalling for the independent stages in outorder unlike in inorder, this performance difference is observed.
3. As the system contains 4 Functional units in parallel, find the bottleneck of execution. This means changing the number of which parameters of these
   a. Memory port, b. Issue width, c. Decode width, will increase the performance most. You will have to run multiple simulations to see this. Change only one of these to 4 at a time and observe the effect on IPC.

Ans: After multiple runs by changing the above conditions, we can conclude that with issue width = 4, we are getting higher IPC, that is more number of instructions are issued in a cycle giving better performance.

4. Based on part e) what is the new IPC and what is your conclusion?
   Ans: The new IPC value is 0.5968, with the increase in issue width, there will be increase in multithreading. Since maximum number of instructions are executed in given cycle, execution speed increases which in-turn increase performance of the system.

1.5 Lab 4 Part 3: In pipelining and Superscalar using Simplescalar (2), we copied the 'default.cfg' configuration from the simplesim-3.0/config folder, renamed it as 'config_e.cfg' and modified the values as follows:

- Decode: width = 4, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size =4, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 8, number of reservation stations.
- Issue: width= 4, maximum number of instructions that are issued in a cycle
- Memory ports= 4, read/write access to one memory address.
- All other ALU resources = 1, number of integers, floating, multipliers/dividers ALUs
- Configuration: Inorder = false

Command: ./sim-outorder -config ./labs/config_e.cfg -ptrace ./config_j.trc 0:1024 -redir:sim sim_configj.out ./labs/test-math

Results and Comments:

Refer: "Part1_output files/simulation_output_textfiles/sim_configj_out.txt" for the detailed output. Also config_jtrc.txt for tracing the pipeline.

1. What is the IPC (Instructions per cycle) of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for outorder configuration we obtained is 0.6949.
2. Compare this configuration with the configuration in part 2 of the previous lab.
   Ans: Both the configurations are outorder, but in this case there is an increase in decode width, fetch queue size, issue width and memory ports. With increase in these values, we can observe that large number of instructions are fetched, decoded, and issued. Therefore, there is an increase in performance in this lab.

1.6 Lab 4 Part 4: In pipelining and Superscalar using Simplescalar (2), we copied the 'default.cfg' configuration from the simplesim-3.0/config folder, renamed it as 'config_f.cfg' and modified the values as follows:

- Decode: width = 4, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size =4, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 8, number of reservation stations.
- Issue: width= 4, maximum number of instructions that are issued in a cycle

- Memory ports= 4, read/write access to one memory address.
- All other ALU resources = 4, number of integers, floating, multipliers/dividers ALUs
- Configuration: Inorder = false

Command:./sim-outorder -config ./labs/confi_f.cfg -ptrace ./config_k.trc 0:1024 -redir:sim sim_configk.out ./labs/test-math

Results and Comments:

Refer: "Part1_output files/simulation_output_textfiles/sim_configk_out.txt" for the detailed output. Also config_ktrc.txt for tracing the pipeline.

1. What is the IPC (Instructions per cycle) of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for outorder configuration we obtained is 0.8692.

1.7 Lab 4 Part 5: In pipelining and Superscalar using Simplescalar (2), we copied the 'default.cfg' configuration from the simplesim-3.0/config folder, renamed it as 'config_g.cfg' and modified the values as follows:

- Decode: width = 4, which gives us maximum number of instructions decoded in a cycle.
- Fetch queue size =4, which defines number of instructions in the fetch cycle queue.
- Load store queue size remain same that is 8 for this configuration as well.
- Register update unit (ruu) size = 16, number of reservation stations.
- Issue: width= 4, maximum number of instructions that are issued in a cycle
- Memory ports= 4, read/write access to one memory address.
- All other ALU resources = 4, number of integers, floating, multipliers/dividers ALUs
- Configuration: Inorder = false

Command:./sim-outorder -config ./labs/confi_g.cfg -ptrace ./config_l.trc 0:1024 -redir:sim sim_configl.out ./labs/test-math

Results and Comments:

Refer: "Part1_output files/simulation_output_textfiles/sim_configl_out.txt" for the detailed output. Also config_ltrc.txt for tracing the pipeline.

1. What is the IPC (Instructions per cycle) of the test-math program using this configuration?
   Ans: The Instructions Per Cycle (IPC) for outorder configuration we obtained is 0. 9552.

1.8. Lab 7 : *Cache Organization with Unified/Separate Instruction and Data Cache:*

Lab Part 1: We have considered following conditions to simulate the outputs:

- Least-recently-used (LRU) replacement policy
- 16 -byte cache size (block-size)
- Set/ Associativity combination: 128/1, 128/2, 128/4, 2048/1, 2048/2, 2048/4
- Only L1 cache present (L2 cache assumed not present)

With the above set of combinations, we have obtained the performance for Unified, Separate Instruction and Data cache using **sim-cache** simulator.

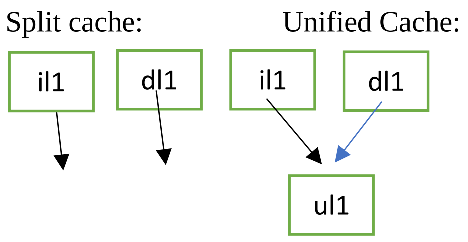
Since 128-set unified configuration is similar to 64-set Instruction and 64-set Data cache configuration, and 2048-set unified configuration is similar to 1024-set Instruction and 1024-set Data cache configuration, the following commands are used for tabulating the results.

Commands:

Sim-cache simulation for split cache configuration: ./sim-cache -cache:il1 il1:64:16:1:l -cache:il2 none -cache:dl1 dl1:64:16:1:l -cache:dl2 none ./tests-pisa/bin.little/test-math

Sim-cache simulation for Unified cache configuration: ./sim-cache -cache:il1 dl1 -cache:il2 none -cache:dl1 ul1:128:16:1:l -cache:dl2 none ./tests-pisa/bin.little/test-math

Where the format is **cache-name: sets: block-size: associativity: replacement strategy.**



Below table shows the miss ratios of Split cache (Instruction/data cache) and Unified cache. I-cache miss ratio = (Instruction cache misses/Instruction cache accesses)

D-cache miss ratio = (Data cache misses/Data cache accesses)

Effective combined cache miss ratio = (Instruction cache misses +Data cache misses /Instruction cache accesses + (Data cache accesses)

Evaluation and Results:

Refer: “Part1_output files/ Simulation_output_text files/ Lab7_128_1 to Lab7_2048_4” text files for outputs.

Table 1: Lab 7-Sets/associativity vs Miss Ratio

Sets/Associativity	**I-Cache Miss Rate (N/2 sets)**	**D-Cache Miss Rate (N/2 sets)**	**Effective Combined Cache Miss Rate**	**Unified Cache Miss Rate**
128/1	0.3764	0.0654	0.31048	0.297
128/2	0.292	0.028	0.236	0.206
128/4	0.2255	0.0187	0.1816	0.1438
2048/1	0.1171	0.018	0.0961	0.0554
2048/2	0.0414	0.0176	0.0363	0.0184

Graph of miss ratio vs. associativity:

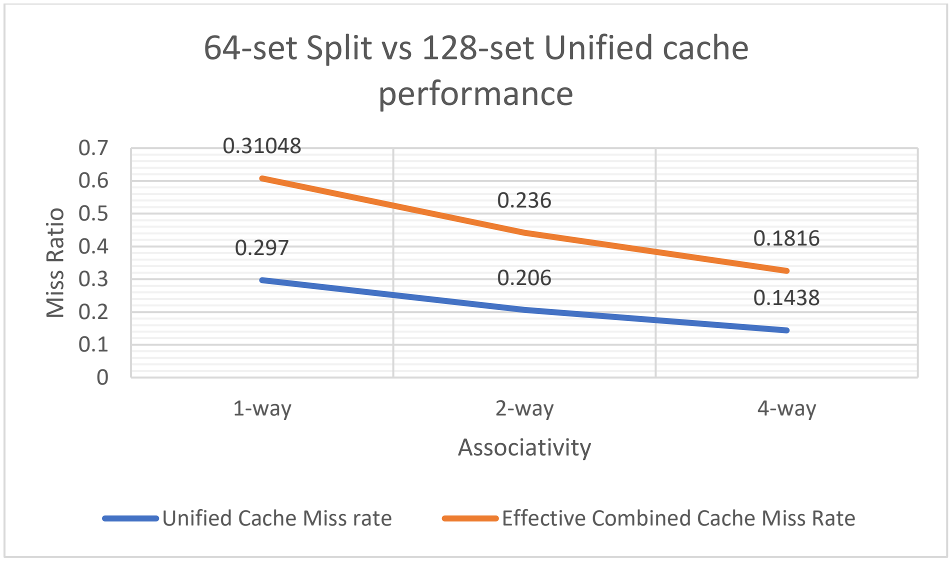


Figure 2: Lab7-Unified vs Split cache Miss ratio

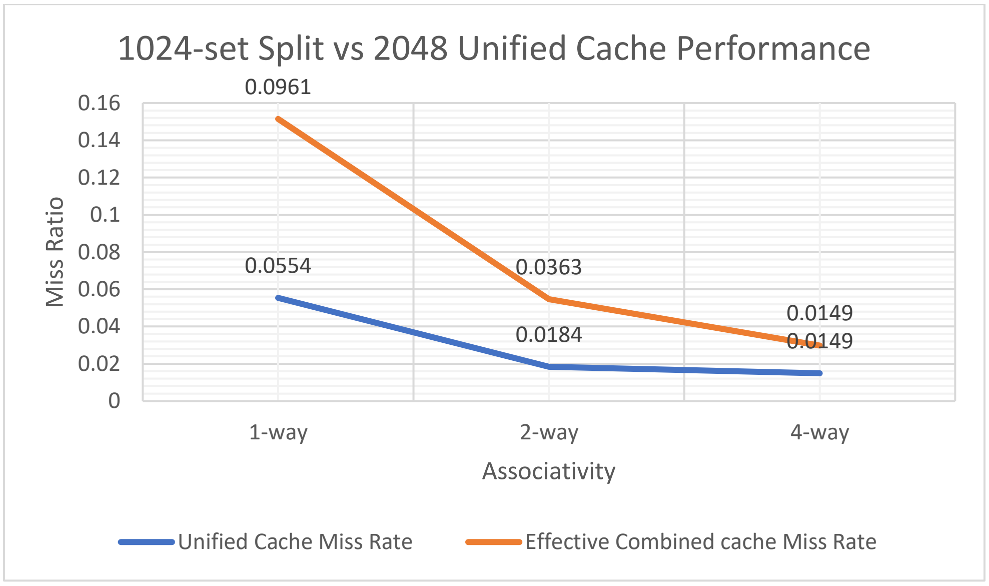


Figure 3: Lab7-Unified vs Split cache Miss ratio

<u>1.9 Lab 7 Bonus part:</u>

Commands:

Sim-outorder simulation for split cache configuration: ./sim-outorder -cache:il1 il1:64:16:1:l -cache:il2 none -cache:dl1 dl1:64:16:1:l -cache:dl2 none ./tests-pisa/bin.little/test-math

Sim-outorder simulation for Unified cache configuration: ./sim-outorder -cache:il1 dl1 -cache:il2 none -cache:dl1 ul1:128:16:1:l -cache:dl2 none ./tests-pisa/bin.little/test-math

Evaluation and Results:

Refer: "Part1_output files/ Simulation_output_text files/ Lab7_bonus_128_1 to Lab7_bonus_2048_4" text files for outputs.

Table 2: Lab7-Bonus Set/Associativity vs CPI

Sets/Associativity	CPI Unified Cache	CPI Split Cache
128/1	6.8025	7.7881
128/2	5.0633	6.1743
128/4	3.7546	4.8687
2048/1	1.707	2.812
2048/2	0.924	1.3624
2048/4	0.8521	0.853

Graph CPI vs Associativity:

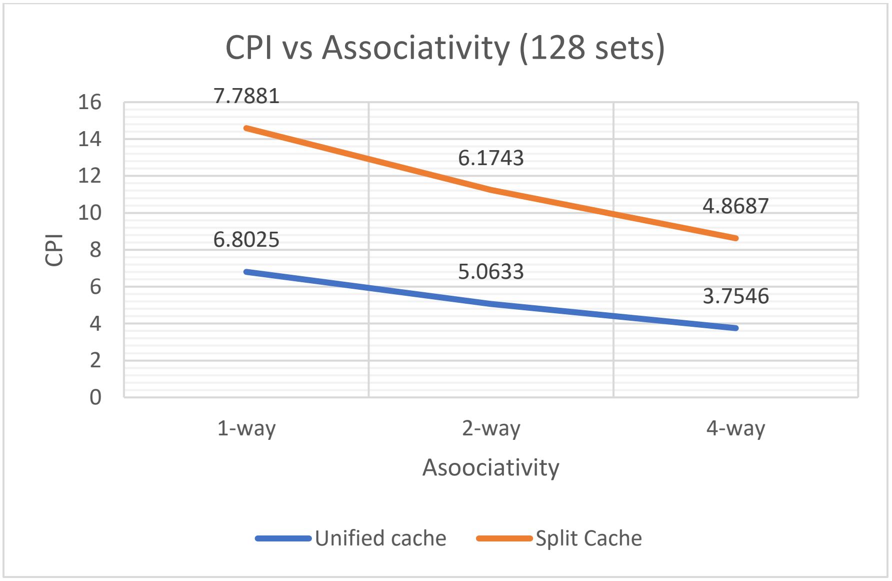


Figure 4: Lab7-CPI vs Associativity (128 sets)

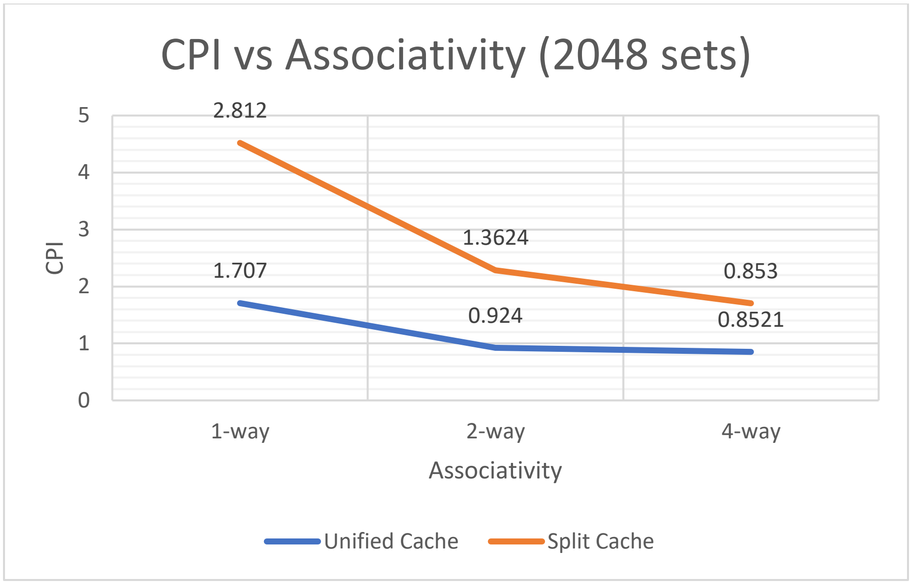


Figure 5: Lab7-CPI vs Associativity (2048 sets)

Conclusion: By observing the tabulated data and graphs of miss ratio and CPI vs sets/associativity of Unified and Split cache for Instrcution and Data Cache, we can conclude that even after combining the result of Instruction cache and Data cache together for miss ratio, Unified cache is performing better with low miss ratio values as well as low Clock cycles per Intruction. Therefore, performance of Unified cache is better than Split cache.

2. PART II:

1.10. Objective:

To develop a L1-L2 Dram memory hierarchy system, where L1 cache organization is direct mapping cache and L2 cache is set associativity. Also, to test the developed code using a problem statement by taking user inputs and obtain the results in output.

Introduction: In the computer organization 3 parts plays an important role. They are as follows:

- **Storage**: Involves hard disk in the storage mechanism and the operation of this device is mechanical. Hence, the data loading to the CPU will be slow and reduces the average memory access time.
- **Memory**: This device, stores data on a temporary basis and cannot hold the data permanently. However, its throughput is faster than storage devices. Some of the memory devices are RAM (SRAM, DRAM). Though having only RAM in the memory hierarchy system cannot cater to the speed of operation of CPU. The throughput of this architecture can further be improved by adding a cache. Cache organization are of three types:

1) *Direct mapped cache*: The most frequently used data from the memory is directly mapped to the cache and it is stored there. When the CPU requests for this information, cache can supply it quickly.
2) *Fully associative cache:* Permits data to be stored in any block but is expensive due to higher hardware units.
3) *Set associative cache*: Gives better hit rate and flexibility to store data than direct mapping cache and hence, this organization is better.

- **CPU**: It consists of 2 components: Arithmetic Logic Unit (ALU) and Control Unit. It performs all the computations required by the computer. For example, addition and subtraction, comparison, etc.

1.11. Implementation:

Here, our code for L1 cache demonstrates direct mapping cache. It takes the input from the user for number of bits in memory address and number of bits in cache address. Then, it forms a table for memory and cache by adding random integer data in memory table. Later, it takes a user input to find the data in particular memory index. However, the data fetching happens from cache table if it contains the data the code prints it as cache hit. And if the data, is not present in cache table it prints it as cache miss and updates the cache table with, tag bits, data, and valid bit.

Example: memory bits will be considered as 12

cache table contains 128 frames,

cache index contains 7 bits

Using this cache index, we can calculate number of tag bits in memory table and index bits in memory table.

Here, we assume capacity to store the size of data stored in each page of memory table and each frame of cache table is same. Outputs for 0th memory index is displayed in the L1 cache project.

Further, our code for L2 cache demonstrates set associativity cache. It takes the input from the user for cache length i.e., number of frames in cache memory, associativity type number of bits in memory address and number of bits in cache address. Then, it forms a table for memory by adding random integer data in memory table and cache containing equal number of cache frames in each set. Later, it takes a user input to find the data in particular memory index. However, the data fetching happens from cache table if it contains the data the code prints it as cache hit. And if the data, is not present in cache table it prints it as cache miss and updates the cache table with, tag bits, data. We can observe the flexibility to store the data is increased in the set associativity cache and cache hit rate also increases.

Example: memory bits will be considered as 12

cache table contains 128 frames

cache index contains 7 bits

Using this cache index, we can calculate number of tag bits in memory table and index bits in memory table.

Here, we assume capacity to store the size of data stored in each page of memory table and each frame of cache table is same. Outputs for 0th memory index is displayed in the L1 cache project.

1.12.Evaluation and Results:

Refer: Part2_L1_L2_DRAM_output files folder for l1cache.py, l2cache.py and Part_2_L1_L2_DRAM_Outputs for details code and outputs for direct mapping, set associativity.

Evaluation of results for L1 cache (direct mapping):

1) We are considering a memory address containing 12 bits.
2) We are assuming our cache is containing 128 frames that is 2**7.
3) Cache index contains 7 bits.

Output:

For the testing purpose we are considering 0th memory index**:**

Result of first iteration will be cache miss, as the cache table will be empty. Hence, CPU fetches the corresponding data from Memory table and data gets updated in cache table at the 0th index and the corresponding tag value from the memory table gets updated in the cache table. Also the valid bit is set.

Also, if we try to fetch the data from 64th memory index, the result will be a cache miss, and new data and tag value gets updated in the cache table. By this step we can conclude that the correct tag value is getting updated in the cache table.

If we try to fetch the data from 64th memory index, the CPU, verifies the tag and index values of the cache table and that matches with the memory address bits, and it fetches the data from cache table, and this will be a cache hit.

Evaluation of results for L2 cache (set associativity):

1) We are considering a memory address containing 12 bits.
2) We are assuming our cache is containing 128 frames that is 2**7.
3) Cache index contains 7 bits.
4) Set associativity = 2, 4

Output:

For the testing purpose we are considering 0th memory index:

Result of first iteration will be cache miss, as the cache table will be empty. Hence, CPU fetches the corresponding data from Memory table and data gets updated in cache table at the 0th index and the corresponding tag value from the memory table gets updated in the cache table.

Also, if we try to fetch the data from 64th memory index, the result will be a cache miss, and new data and tag value gets updated in the cache table. By this step we can conclude that the correct tag value is getting updated in the cache table.

If we try to fetch the data from 64th memory index, the CPU, verifies the tag and index values of the cache table and that matches with the memory address bits, and it fetches the data from cache table, and this will be a cache hit.

1.13. Source code description:

The below are the website link for pycharm and jupyter where you can install and run the code.
https://www.jetbrains.com/pycharm/download/#section=windows
https://jupyter.org/install
The code for the final project is in the zipped folder sent along with this report. The files are named as l1cache.py and l2cache.py and their location is Part2_L1_L2_DRAM_output_files/l1cache.py and Part2_L1_L2_DRAM_output_files/l2cache.py. To run the code you can use any of the platforms (Pycharm or Jupyter), we have tried on both the platforms and successfully achieved the expected output.

For l1cache.py you can enter:

```
Enter the num of bits address 12
Enter the cache size 8
```

or any cache size and search the index in which you need the data. The output will be cache miss or cache hit accordingly.

For l2cache.py you can enter:

```
Enter cache length 128
Enter associativity type 2
Enter membits 12
Enter cache bits 7
Enter memory location to search 1
```

or any other location that you want to retrieve data from. The output will be cache miss or cache hit accordingly.

1.14.Conclusion:

From the above two program executions, we can conclude that set associative cache is flexible and trades-off between latency and hardware cost unlike direct mapped cache which is easy but not flexible because it might not utilize all the blocks in the cache and creates more cache misses.