Simulation Project Report

Computer Architecture – 16:332:563

(Electrical and Computer Engineering - Fall 2021)

Lohitanvita Rompicharla

lr701@rutgers.edu

Rutgers University – New Brunswick

Dept: Electrical & Computer Engineering

Abstract **— The Simulation Project is performed using SimpleScalar simulator which helps to leverage faster and more flexible software development cycle. The SimpleScalar architecture consists of a tool set including compiler, assembler, linker, simulation, and visualization tools. With this tool set, we have performed various simulations in different ISA configurations to obtain the CPU performance status using benchmarks. The main objective of this simulation project is to understand and analyze instruction set architecture, quantitative performance analysis, memory hierarchy, and instruction-level parallelism.**

Index Terms— ISA, benchmarks, cache hit, cache miss, miss ratio, branch prediction, CPI

I. Introduction

SimpleScalar is a simulator used for performance analysis and system verification at the microarchitectural level. It supports various ISAs, including Alpha, PISA, ARM, and x86. We can simulate real programs on a range of modern processors and systems, using fast execution-driven simulation. Simulators ranging from a fast functional simulator to a detailed, out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction are present. The tool set is partly derived from the GNU software development tools. It provides an easily extensible, portable, high-performance test bed for systems design.

Below are the basic configurations, tools, benchmarks, and functions used for performing the simulations:

A. The two ISA configurations used are ALPHA and PISA. ALPHA ISA is a Reduced Instruction Set Architecture that provides careful balance for encoding common operations. PISA ISA is a simple MIPS-like instruction set maintained primarily for instructional use. It also consists of a GNU-based cross-compiler and pre-built libraries.

B. Benchmarks are standard tests used to compare similar products to analyze performance statistics. Benchmarks used in this simulation project are: a) **Anagram**: a program for finding anagrams for a phrase, based on a dictionary, b) **Compress95**: (SPEC) compresses and decompresses a file in memory, c) **Go** : Artificial Intelligence, plays the game of go against itself, d) **GCC**: (SPEC) limited version of gcc, e) **Test-math**: perform various math computations mostly in integer and display their result, f) **Test-fmath**: perform various math functions mostly in float, g) **Test-llong**: perform computations in long format. h) **Test-printf**: display various print statements.

C. The major four simulators used are: a) **sim-profile:** dynamic instruction interpreter and profiler, b) **sim-cache:** memory system simulator, c) **sim-bpred:** branch predictor simulator, d) **sim-outorder:** detailed micro-architectural simulator.

II. Environment Setting and Setup Installation

Linux OS supports best for running SimpleScalar on the system. For installing SimpleScalar, we must initially install and run Linux OS in VMware machine on our system (windows or Mac).

Note: My system has windows 10 OS and I have tried installing SimpleScalar 3.0 but could not run the 'make' command even after installing and upgrading MinGW manager for GCC compiler.

Followed the below mentioned procedure for Environment setting:

A. Installed VMware workstation 16 player and then installed Ubuntu 1604 Linux OS in VMware. Note: Used Ubuntu-s001.vmdk zip file present in the Ubuntu 1604 folder to install OS in the VMware. It is a virtual machine disk file containing the OS settings. After Ubuntu started working, downloaded and installed SimpleScalar, benchmarks zipped files onto Ubuntu.

B. For downloading the simplesim-3.0 and benchmarks folder, used http://www.simplescalar.com/ website's tool section. Then extracted the tar.zip folders using '$ tar xvzf simplesim-3v0e.tgz' command in the Ubuntu terminal.

C. Instead of extracting benchmarks folder in simplesim-3.0 folder, I have extracted all the files of benchmark folder into simplesim-3.0 to avoid the path confusion while running any command. That is that is instead of $ ./sim-cache benchmarks/cc1.alpha -0, I can directly write $ ./sim-cache cc1.alpha -0

D. Procedure to install and run a test to verify SimpleScalar: open the terminal in ubuntu and run the following commands

- $ cd simplesim-3.0
- $ make
- $ ./sim-cache tests-alpha/bin/test-math

Got "-1e-17 == -1e-17 Worked!" as terminal output, it shows that everything is ready for performing the simulation labs.

III. Lab 1: Introduction to ISA using SimpleScalar:

The main objective of Lab 1 is to make ourselves familiar with the ALPHA and PISA Instruction set Architectures by executing various benchmarks using sim-profile simulator. This simulator provides with the whole profile information of the ALPHA and PISA ISA based on benchmarks like anagram,

compress, go, gcc, test-math, test-fmath, test-llong, test-printf. From this information we are able to conclude the performance of both the ISA's.

1.1. Part 1: Initially, I have run the sim-profile simulator in ALPHA ISA configurations. To configure SimpleScalar in Alpha configurations, the following commands are ran:

- $ make clean
- $ make config-alpha
- $ make

And got an output as "my work is done here...", this shows that ALPHA configuration is set for simulation.

Later type "./sim-profile" to get the usage of the simulator, i.e.,

Usage: ./sim-profile {-options} executable {arguments}

This gives the format of the command and description of what to use in the command to work it our way.

1.2. Evaluation and Results

Refer: Lab 1_part 1_alpha-config_simulations_results.txt in the output folder for detailed result.

Simulation Commands:

{-options} = -iclass for class profile

- $ ./sim-profile -iclass anagram.alpha
- $ ./sim-profile -iclass go.alpha
- $ ./sim-profile -iclass compress95.alpha -O
- $ ./sim-profile -iclass cc1.alpha -O 1stmt.i

Tabulated Results:

Benchmark	Total # of Instructions	Load %	Store %	Unconditional Branch %	Conditional Branch %	Integer Computation %	Floating point Computation %
Anagram.alpha	4958	17.83	19.35	6.03	11.88	44.54	0.12
Go.alpha	709706	14.79	12.93	2.66	14.87	54.41	0.33
Compress.alpha	4908	17.14	12.90	5.18	13.96	50.56	0.00
Gcc.alpha	337341118	24.67	11.47	4.12	13.33	46.30	0.11

Table 1: Benchmarks vs Memory/Computational Instructions

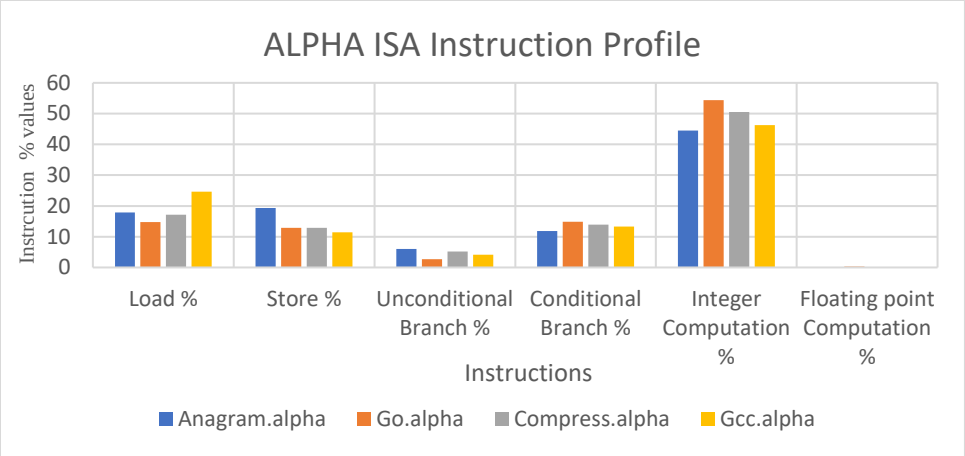

Fig 1. ALPHA ISA instruction Profile

Comments:

Refer "Lab 1_part 1_alpha-config_simulations_results.txt" for the detailed results.
Now answer the following questions for each individual benchmark executed above

1. Is the benchmark memory intensive or computation intensive?
2. Is the benchmark mainly using integer or floating point computations?
3. What % of the instructions executed are conditional branches? Given this %, how many instructions on average does the processor execute between each pair of conditional branch instructions (do not include the conditional branch instructions)

Solutions:

Anagram.alpha:

1. Memory instructions includes all load, store, unconditional branch, and conditional branch instructions that uses memory that constitutes of 55.06 % whereas computation instructions consist of integer computation and floating computation constituting 44.63 %. Therefore, anagram.alpha is Memory intensive.
2. Anagram.alpha is mainly using integer computation (44.54 %).
3. 11.88 % of total instructions are conditional branch instructions. This averages to execution of one pair of branch instructions per 8.41 instructions [Total number of instructions/ Total number of conditional branch instructions]

Go.alpha:

1. Memory instructions includes all load, store, unconditional branch, and conditional branch instructions that uses memory that constitutes of 45.25 % whereas computation instructions consist of integer computation and floating computation constituting 54.74 %. Therefore, it is Computation intensive.
2. Go.alpha is mainly using integer computation (54.41%)
3. 14.87 % of total instructions are conditional branch instructions. This averages to execution of one pair of branch instructions per 6.72 instructions [Total number of instructions/ Total number of conditional branch instructions]

Compress95.alpha:

1. Memory instructions includes all load, store, unconditional branch, and conditional branch instructions that uses memory that constitutes of 49.18 % whereas computation

instructions consist of integer computation and floating computation constituting 50.56 %. Therefore, it is Computation intensive.
2. Compress95.alpha is mainly using integer computation (50.56%).
3. 13.96 % of total instructions are conditional branch instructions. This averages to execution of one pair of branch instructions per 7.16 instructions [Total number of instructions/ Total number of conditional branch instructions]

GCC.alpha:

1. Memory instructions includes all load, store, unconditional branch, and conditional branch instructions that uses memory that constitutes of 53.53 % whereas computation instructions consist of integer computation and floating computation constituting 46.41 %. Therefore, it is Memory intensive.
2. GCC.alpha is mainly using integer computation (46.30 %).
3. 13.33 % of total instructions are conditional branch instructions. This averages to execution of one pair of branch instructions per 7.5 instructions [Total number of instructions/ Total number of conditional branch instructions]

1.3. Part 2: For this part of the lab, we will test ALPHA and PISA ISA based on mathematical computations benchmark. First, we will test ALPHA ISA configuration.

Refer: Lab 1_part 2_alpha-config_simulation_results.txt and Lab 1_part 2_pisa-config_simulation_results.txt files in the output folder for detailed output.

Simulation Commands:

- $ ./sim-profile -iclass ./tests-alpha/bin/test-math
- $ ./sim-profile -iclass ./tests-alpha/bin/test-fmath
- $ ./sim-profile -iclass ./tests-alpha/bin/test-llong
- $ ./sim-profile -iclass ./tests-alpha/bin/test-printf

Tabulated Results:

Alpha Benchmark	Total # of Instructions	Load %	Store %	Unconditional Branch %	Conditional Branch %	Integer Computation %	Floating point Computation %
Test-math	49409	17.18	10.42	3.94	11.07	55.36	1.88
Test-fmath	19498	17.72	12.52	4.70	11.28	53.19	0.43
Test-llong	10626	17.81	14.60	5.42	12.40	49.48	0.10
Test-printf	983472	17.99	10.73	4.82	11.39	54.84	0.09

Table2: Benchmark vs Instructions for ALPHA ISA

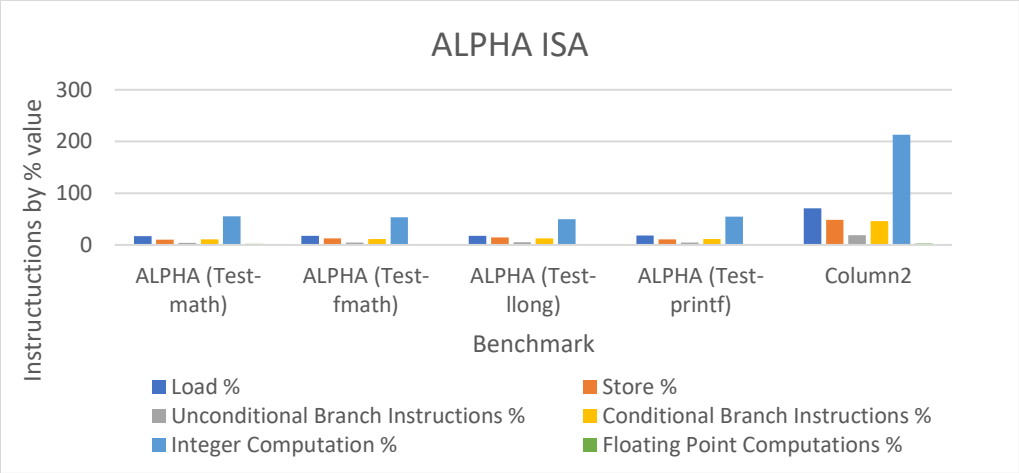

Fig 2. Instruction Profile for ALPHA ISA

Now configure PISA ISA by using following commands in the terminal:

- $ make clean
- $ make config-pisa
- $ make

Simulation Commands:

- $ ./sim-profile -iclass ./tests-pisa/bin.little/test-math
- $ ./sim-profile -iclass ./tests-pisa/bin.little/test-fmath
- $ ./sim-profile -iclass ./tests-pisa/bin.little/test-llong
- $ ./sim-profile -iclass ./tests-pisa/bin.little/test-printf

Tabulated Results:

Pisa Benchmark	Total # of Instructions	Load %	Store %	Unconditional Branch %	Conditional Branch %	Integer Computation %	Floating point Computation %
Test-math	213745	15.96	10.66	4.22	13.85	54.42	0.88
Test-fmath	53504	16.14	14.41	4.24	15.11	49.95	0.11
Test-llong	29687	16.33	17.99	4.37	15.45	45.82	0.00
Test-printf	1813937	19.22	9.28	5.13	17.01	49.33	0.01

Table 2: Benchmark vs Instructions for PISA ISA

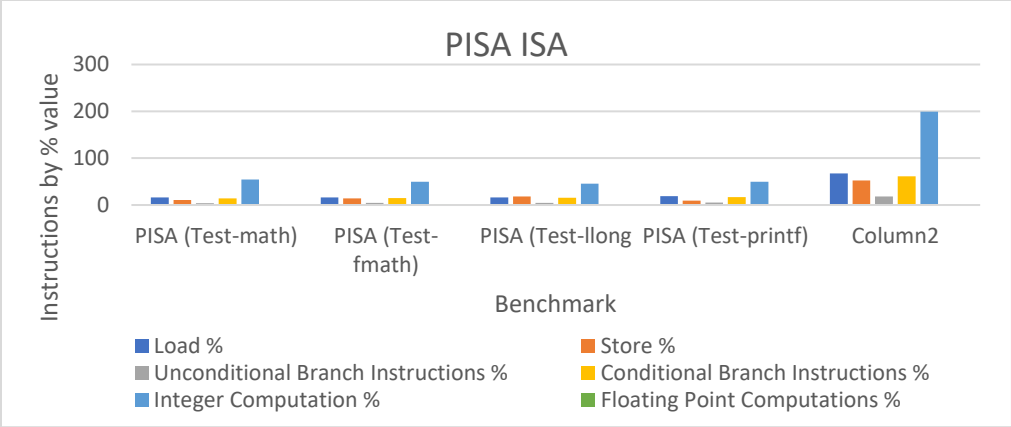

Fig 3. Instruction Profile for PISA ISA

Comments:

1) Now compare the two ISAs using a plot (a Histogram is preferred). Use MATLAB or EXCEL to plot the histogram. What can you conclude about the two ISAs from the Histogram.

Solution: Below are the histograms plotted for all the benchmarks with respect to instructions in percentage for both ALPHA and PISA ISAs. Now from the below histogram, by comparing the memory instructions with computational instructions, we can observe that ALPHA ISA has total memory usage upto 183.99 %, computational usage upto 215.37 %. So, ALPHA ISA is more of a computationally intensive architecture. Now observing PISA ISA, it has memory usage of total 199.37 % and computation usage of 200.52 %. That is, PISA ISA is also a computationally intensive architecture. But to distinguish between ALPHA and PISA ISAs, we can conclude that ALPHA is computation intensive and PISA is a memory intensive Instruction Set Architecture.

Histogram Plots:

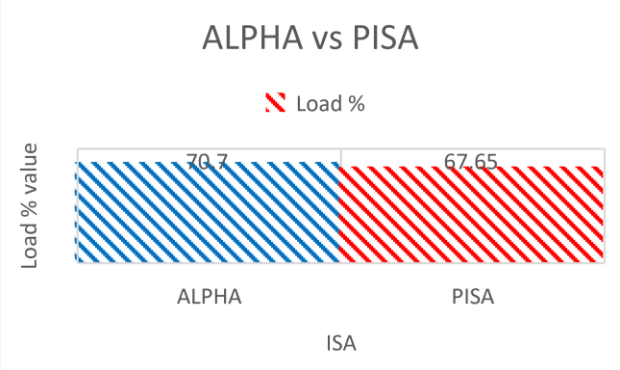

Fig 4. Load Instruction %

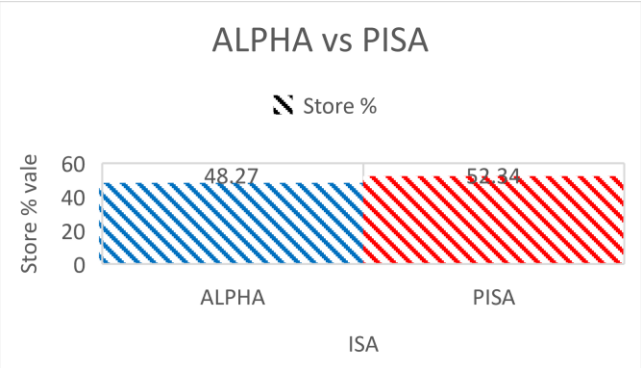

Fig 5: Store Instruction %

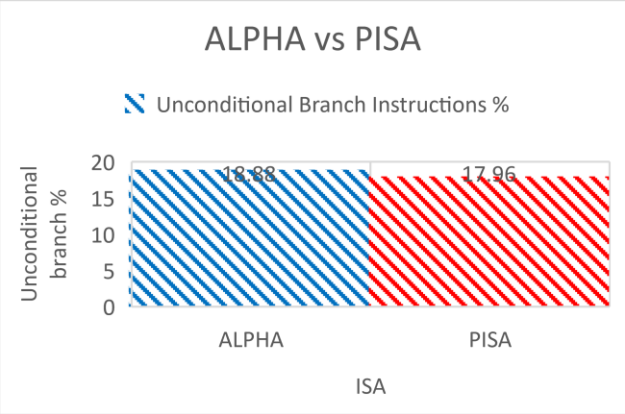

Fig 6: Unconditional Branch Instruction %

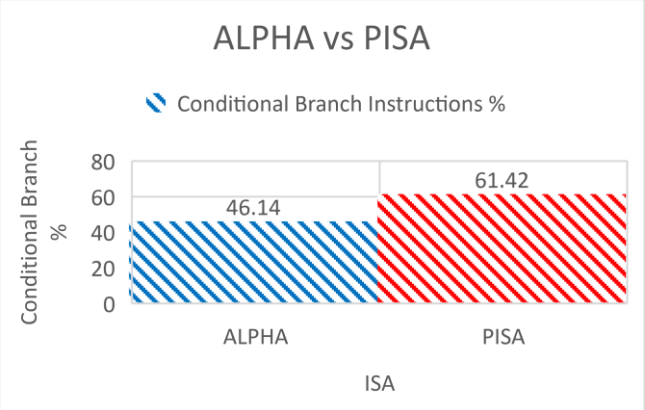

Fig 7: Conditional Branch Instruction %

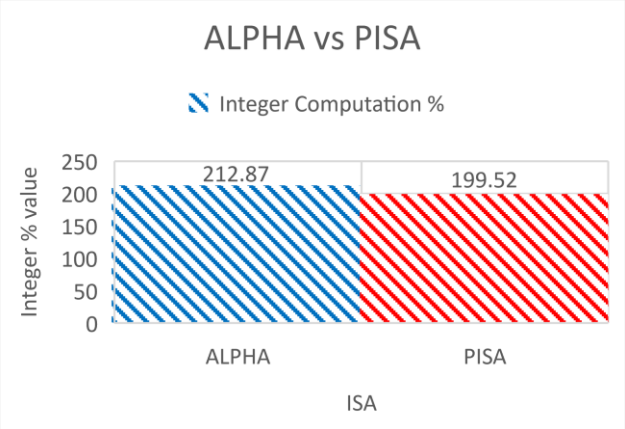

Fig 8: Integer Computation %

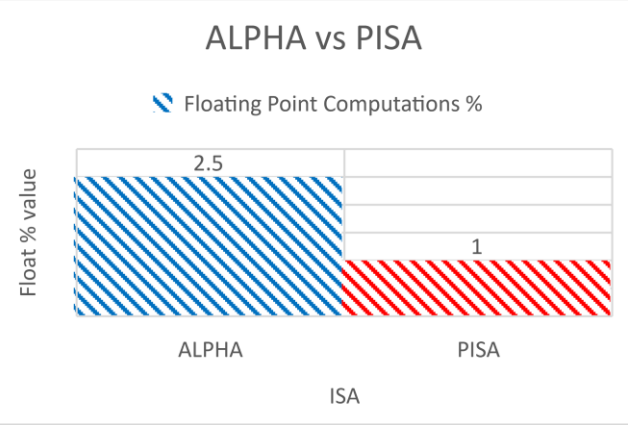

Fig 9: Floating Point Computation %

IV. Lab 2: Cache Organization and Associativity:

Cache organization and Associativity helps in improving the speed of the CPU by retrieving the required data from the cache rather than main memory. While requesting data from the cache, a cache hit or cache miss may occur, i.e., the data requested may be already present in the cache (cache hit) or data not available in the cache (cache miss).

1.1. Procedure: In this lab we are using PISA architecture and sim-cache simulator for finding out the cache miss ratio of Set Associative cache, i.e., for 1-way, 2-way, 3-way, 4-way cache associativity.

The cache miss ratio is calculated for both Instruction-only and Data-only cache in the following conditions:

- least-recently-used (LRU) replacement policy
- 32 to 512 sets
- 1-way to 8-way associativity
- 16-byte cache lines (block size)

And test-math benchmark is used to get mathematical results.

1.2. Evaluation and Results:

Refer: Lab 2_Datacache(32-512 sets)_pisa-config.txt and Lab 2_Instructioncache(32-512 sets)_pisa-config.txt for detailed output

The format of the sim-cache is as follows:

- ./sim-cache {-options} executable {arguments}
- **./sim-cache -cache:<name> <name>:<nsets>:<bsize>:<assoc>:<repl> executable**

Simulation Commands:

- $ ./sim-cache -cache:il1 il1:32:16:1:l ./tests-pisa/bin.little/test-math
- $ ./sim-cache -cache:dl1 dl1:32:16:2:l ./tests-pisa/bin.little/test-math

Changed the set and associativity value for both the commands accordingly.

Tabulated Results:

Miss Ratio (I-cache)	1-way	2-way	4-way	8-way
32 sets	0.4275	0.3734	0.2882	0.1982
64 sets	0.3763	0.2920	0.2255	0.1613
128 sets	0.3030	0.2275	0.1618	0.1028
256 sets	0.2503	0.1720	0.0994	0.0176
512 sets	0.1833	0.1010	0.0348	0.0142

Table 3: Miss Ratio for Instruction-only cache

Miss Ratio (D-cache)	1-way	2-way	4-way	8-way
32 sets	0.1372	0.0503	0.0236	0.0185
64 sets	0.0654	0.0280	0.0187	0.0184
128 sets	0.0408	0.0223	0.0184	0.0177
256 sets	0.0273	0.0194	0.0177	0.0176
512 sets	0.0209	0.0179	0.0176	0.0176

Table 4: Miss ratio for Data-only cache

Comments:

1) Now use MATLAB or EXCEL to plot the results of the simulations. For each of the simulations (data, instruction), plot the miss ratio versus associativity for each number of sets. Using markers, show the points on the curves which correspond to total cache sizes of 1 Kbytes, 2 Kbytes, 4 Kbytes and 8 K bytes (total cache size = sets * block size * associativity).

Solution:

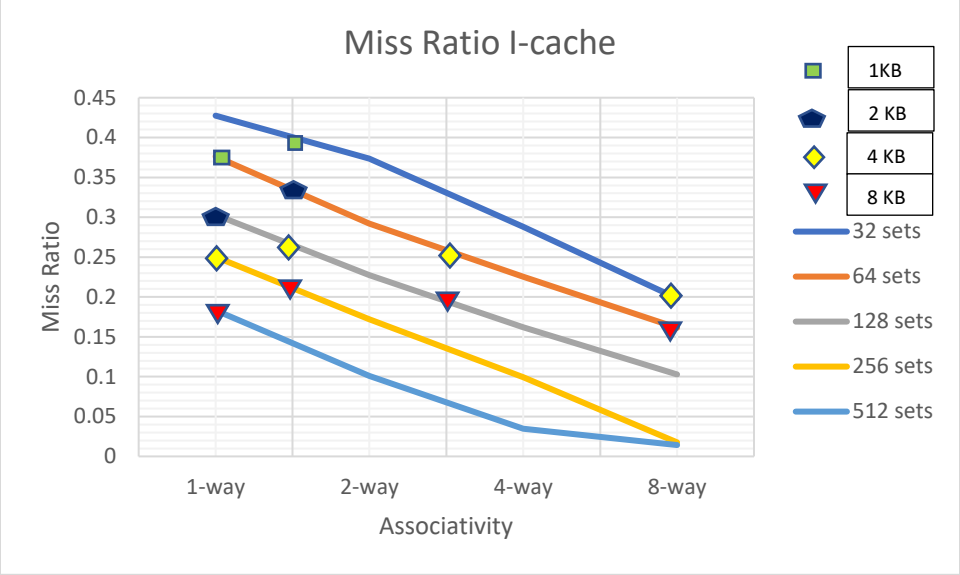

Fig 10: Miss Ratio vs Associativity I-cache

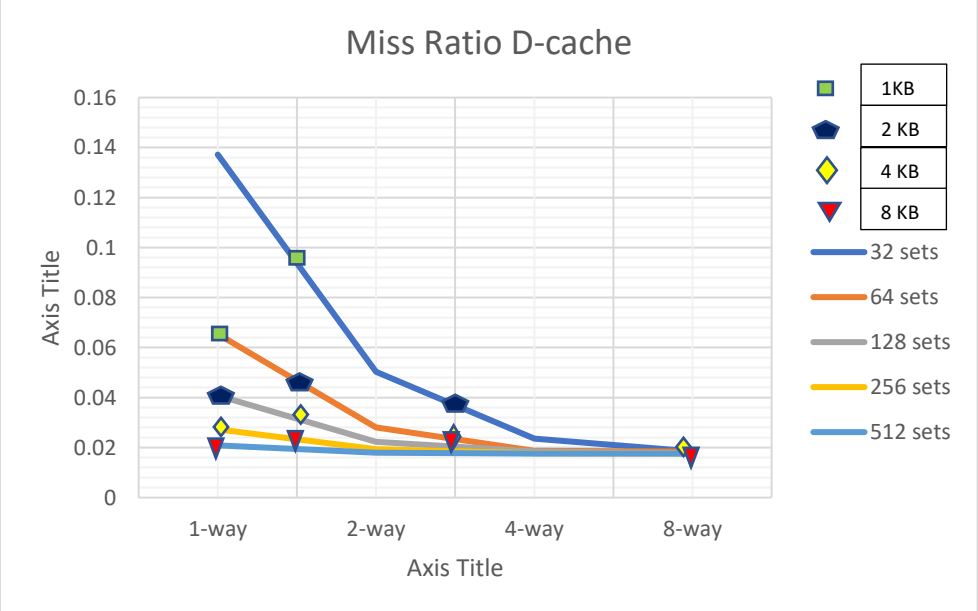

Fig 11: Miss Ratio vs Associativity D-cache

Total cache size Table: Total cache size = sets * block size * associativity

Total Cache Size (KB)	1- way	2-way	4-way	8-way
32 sets	0	1	2	4
64 sets	1	2	4	8
128 sets	2	4	8	16
256 sets	4	8	16	32
512 sets	8	16	32	64

2) Now answer the following questions based on the above results.

1. For a given number of sets, what effect does increasing associativity have on the miss ratio?
   Sol: We can observe from the tabulated results and line graph above that as the associativity increases, miss ratio is decreasing and nearing to zero. This means cache miss is decreasing and cache hits are increasing with the increase in set associativity.

2. For a given associativity, what is the effect of increasing the number of sets?
   Sol: Similar to above results, for a given associativity, for increase in number of sets, miss ratio decreases. This means cache misses are decreased and cache hits are increasing, leading to efficient cache results.

3. For a given cache size, how does the miss ratio change when going from an associativity of one to two to four? Explain.
   Sol: Considering all the cache sizes 1KB, 2KB, 4KB, 8KB for both I-cache and D-cache. When going from associativity 1-way to 4-way, there is always an increase in the miss ratio. But when we reach 8-way associativity, miss ratio decreases. This is because when we move from 1-way to 4-way for particular cache size we are moving from higher set to lower set, i.e 512 to 32 sets. Lowering the sets in cache gives more cache miss as flexibility of allocating data to blocks decreases due to less number of blocks.

4. If you were to design an Instruction cache, limited to a total cache size of 4 Kbytes, which cache organization would you choose, based solely on performance?
   Sol: From the line graph of Miss ratio of I-cache, we can observe that for a cache size of 4KB, the lowest miss ratio is found at 32 sets 8-way. Therefore, based solely on performance I would choose 32 sets 8-way associativity for Instruction cache.

5. If you were to design a data cache, limited to a total cache size of 4 Kbytes, which cache organization would you choose, based solely on performance?
   Sol: Similarly considering Miss Ratio line graph for Data cache and limiting cache size to 4KB, I would choose 32sets 8-way associativity because it gives the lease miss ratio of all.

V. Lab 2 Bonus:

1.1. Procedure: In this lab we are using PISA architecture and sim-cache simulator for finding out the cache miss ratio of Set Associative cache, i.e., for 1-way, 2-way, 3-way, 4-way cache associativity.

The cache miss ratio is calculated for Instruction-only cache in the following conditions:

- Random (R) replacement policy
- 32 to 512 sets
- 1-way to 8-way associativity
- 16-byte cache lines (block size)

And test-math benchmark is used to get mathematical results.

1.2. Evaluation and Results:

Refer: Lab 2_BonusPart_InstructionCache_random-replacement (32-512 sets).txt in simulation ouputs folder for detailed result.

The format of the sim-cache is as follows:

- ./sim-cache {-options} executable {arguments}
- **./sim-cache -cache:<name> <name>:<nsets>:<bsize>:<assoc>:<repl> executable**

Simulation Commands:

- $ ./sim-cache -cache:il1 il1:32:16:1:r ./tests-pisa/bin.little/test-math

Changed the set and associativity value for the command accordingly.

Tabulated Results:

Miss Ratio (I-cache)	1-way	2-way	4-way	8-way
32 sets	0.4269	0.3608	0.2855	0.2185
64 sets	0.3764	0.2936	0.2292	0.1662
128 sets	0.3030	0.2321	0.1658	0.0854
256 sets	0.2503	0.1726	0.0896	0.0290
512 sets	0.1833	0.0996	0.0379	0.0182

Table 5: Miss Ratio for I-cache with random replacement strategy

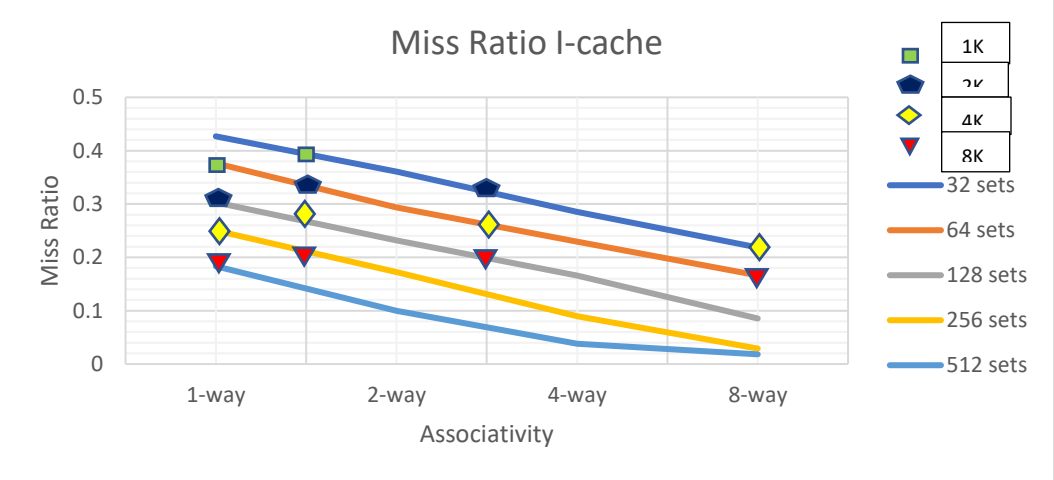

Fig 12: Miss Ratio vs Associativity for Instruction Cache

VI. Lab 3: Branch Prediction:

A branch predictor is a digital circuit that tries to guess which way a branch (e.g., an if–then–else structure) will go before this is known definitively. The purpose of the branch predictor is to improve the flow in the instruction pipeline. There are two types of Branch predictions 1. Static branch predictor 2. Dynamic branch predictor.

The objective of this lab is to predict the branches of the architecture. Whether the branches are Taken, Not Taken, Bimod, i.e., if the predicted branch is correct or not.

1.1. Procedure:

This lab is simulated in ALPHA ISA configuration. Therefore, we have to run following commands again in the terminal to set the ISA as ALPHA:

- $ make clean
- $ make config-alpha
- $ make

Then we use sim-bpred and sim-outorder simulators for calculating branch address prediction rate and CPI (Cycles Per Instruction) for all Taken, Not Taken, Bimod branches.

1.2. Evaluation and Results:

Refer: Lab3_Bpred_anagram.alpha.txt,
Lab3_Bpred_compress95.alpha.txt,
Lab3_Bpred_gcc.alpha.txt,
Lab3_Bpred_go.alpha.txt,
Lab3_sim-outorder_anagram.alpha.txt,
Lab3_sim-outorder_compress.alpha.txt files in the simulation outputs folder for detailed results.

Usage: ./sim-bpred {-options} executable {arguments}
{-options} = -bpred (branch prediction profiling)

Simulation Commands:
For Branch Prediction:
$ ./sim-bpred -bpred taken anagram.alpha
$ ./sim-bpred -bpred taken compress95.alpha -O
$ ./sim-bpred -bpred taken cc1.alpha -O 1stmt.i
$ ./sim-bpred -bpred taken go.alpha

For CPI:
$ ./sim-outorder -bpred taken anagram.alpha
$ ./sim-outorder -bpred bimod compress95.alpha -O

Note: In place of taken, change to nottaken or bimod accordingly.

Tabulated Results:

Benchmark	Taken	Nottaken	Bimod
Anagram.alpha	0.744	0.5923	0.5968
Go.alpha	0.6798	0.4721	0.8955
Compress.alpha	0.8040	0.4665	0.8253
Gcc.alpha	0.6005	0.6357	0.8682

Table 6: Benchmark vs Branch address Prediction

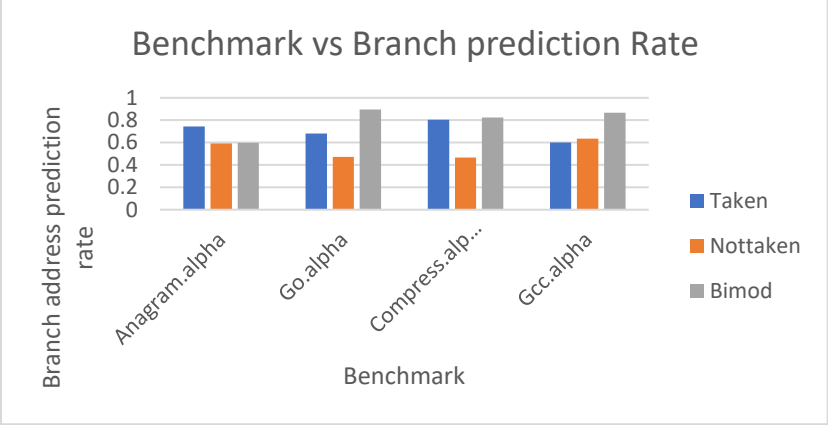

Fig 13: Benchmark vs Branch Address Prediction Rate

Benchmark	Taken(CPI)	Nottaken(CPI)	Bimod(CPI)
Anagram.alpha	2.7043	2.7305	2.6341
Compress.alpha	2.3152	2.3260	1.7060

Table 7: Benchmark vs CPI

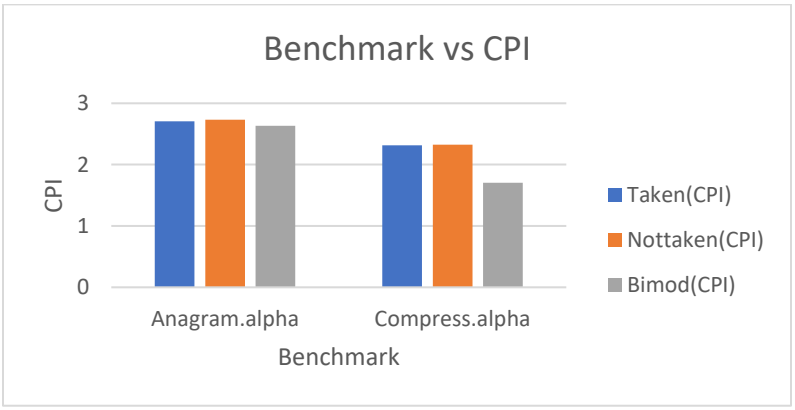

Fig 13: Benchmark vs Branch Address Prediction Rate

Histogram Plots:

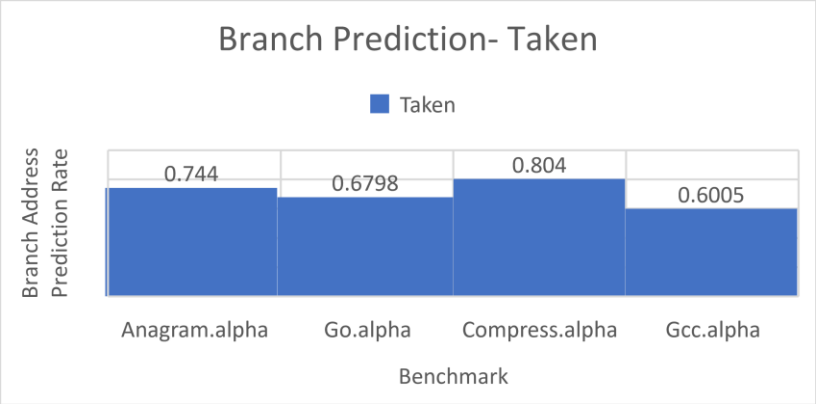

Fig 14: Branch Prediction vs Benchmark for Taken

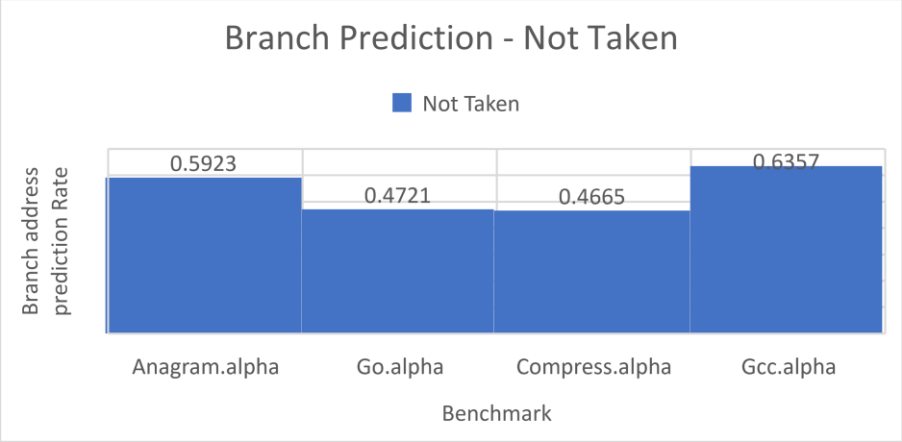

Fig 15: Branch Prediction vs Benchmark for Not Taken

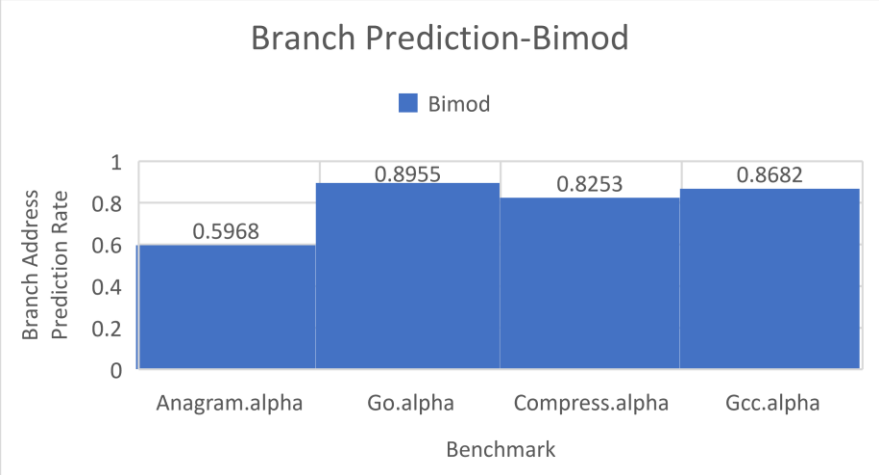

Fig 16: Branch Prediction vs Benchmark for Bimod

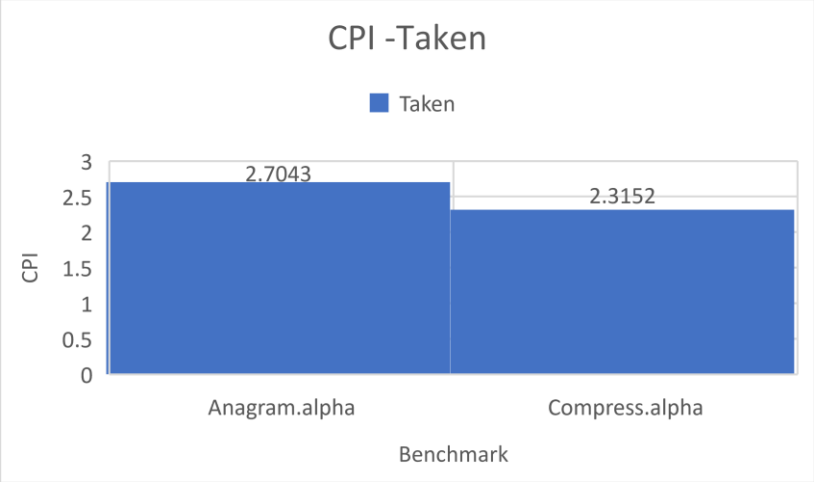

Fig 17: CPI vs Benchmark for Taken

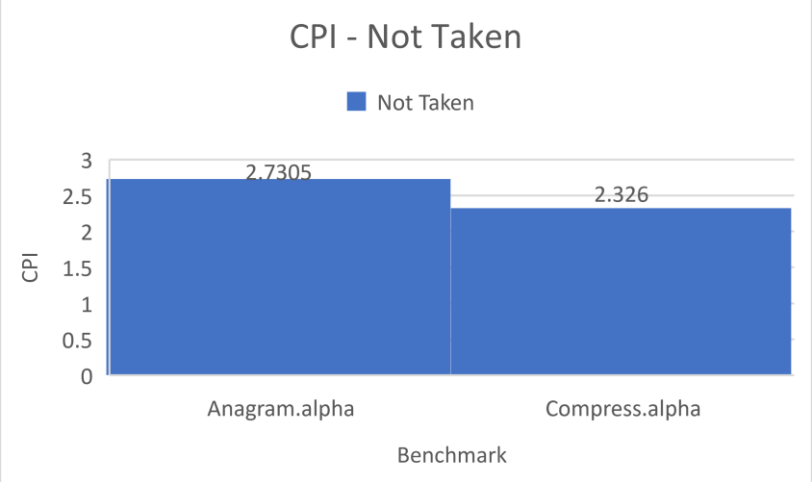

Fig 18: CPI vs Benchmark for Not Taken

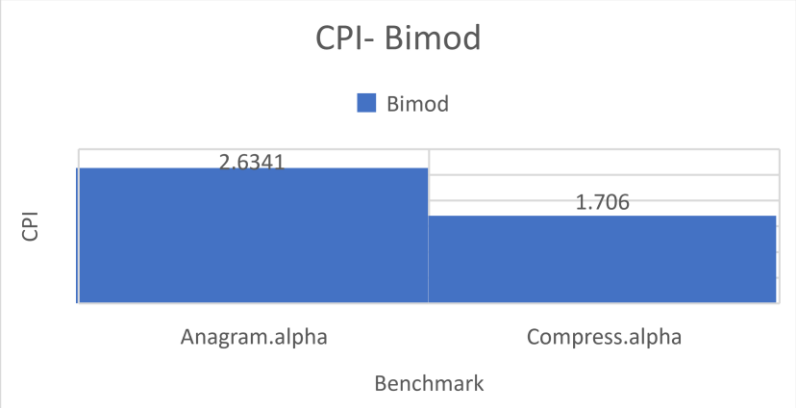

Fig 19: CPI vs Benchmark for Bimod

1.3. Conclusion: From the above graphs and tabulated results we can observe that branch address prediction rate is high for Taken mode compared to Not Taken. Also, Bimod mode makes **a prediction based on the direction the branch** went the last few times it was executed. Therefore, Bimod has better accuracy because it predicts using branch history. The result obtained is as expected only.