HPCA Group-9 Assignment-1

Suryam Arnav Kalra Hritaban Ghosh Surkanti Akshitha Suhas Jain
Srijan Das Srijanak De Ishan Goel Aditya Vallakatla

Contents

1 Introduction

Gem5 can simulate CPUs with different configurations (e.g. number of cores, pipeline complexity, cache size, etc.) based on configuration scripts. We create sample configuration scripts and configure it to an Out-Of-Order CPU with a combination of the micro-architectural parameters (as provided in the assignment problem statement) that reflects the characteristic of a singlecore x86 processor and run the provided benchmark program (in our case the sieve of eratosthenes). We analyse the output statistics of each of these combinations to select the top 10 configurations based on the CPI value and finally share our observations along with explanations

2 Implementation

2.1 How does our code work?

Our submission directory tree is as follows:

```
Group_9_HPCA_Assignment_1
├── source_code
│   ├── config.py
│   ├── exe.py
│   ├── get_stats.py
│   ├── options.py
│   ├── sieve
│   └── sieve.c
└── ...
```

config.py contains the configuration specifications of the system we want to simulate. It takes input from **options.py** via argparse which is parser for command-line options and arguments.

options.py is used to specify various options that can be used to modify the type of simulation we want to run. Additionally, we have modified it to take arguments from the command line via argparse.

exe.py is python script that is used to execute all the 256 possible configurations of the system. It uses the binary form of the integers from 0 to 255 to index into the lists containing the specific configurations (See Figure 22 or 23). Then, it uses the subprocess library to run the commands from the python script itself. In order to run only a specific configuration or a range of configurations, take a look at section 5.3.

sieve is the executable file obtained from **sieve.c** by running the following command.

```
$ gcc -o sieve sieve.c -lm
```

sieve.c is the C program code provided to us by the Teaching Assistants. It contains the code to run Eratosthenes Sieve to obtain prime numbers below 10000.

get_stats.py is the python script which will be run after all the outputs have been generated. It will create the **top_10.csv** file which contains the top 10 configuration sorted in CPI values and will also create the graphs required for the assignment by extracting the required values from the **stats.txt** file of the different output folders.

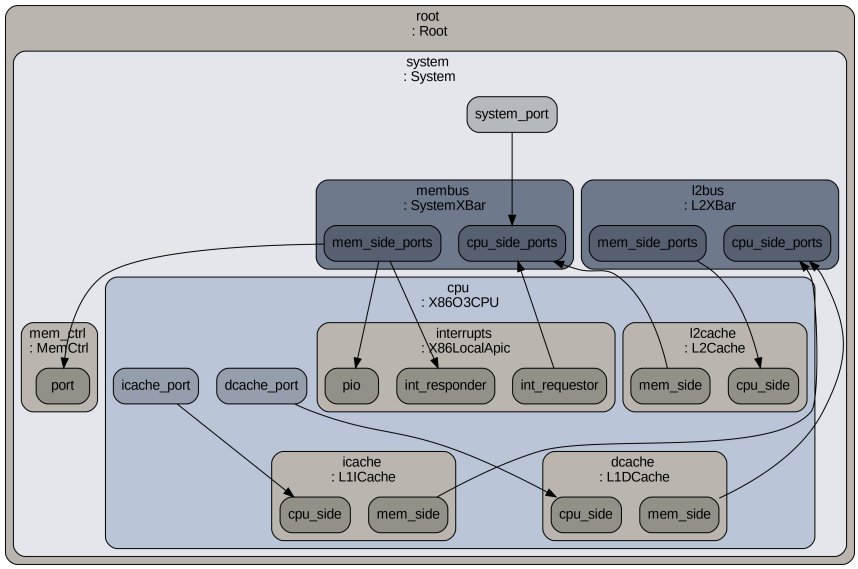

Figure 1: General Configuration of the System

2.2 Outputs

Files added: graphs folder, top_10.csv file

Output contains the following files.

- **graphs folder:** This folder contains the graphs required in the assignment.
- **top_10.csv:** This file contains the micro-architectural parameters for the top 10 configurations sorted on their CPI values.

3 Results

3.1 Top 10 configurations

The top 10 configurations along with their CPI values are summarized in the Table 1.

S.No.	CPI	LQ Entries	SQ Entries	l1d_size	l1i_size	l2_size	bp type	ROB Entries	IQ Entries
1	0.993225	32	64	64kB	16kB	512kB	TournamentBP	192	64
2	0.994351	32	64	64kB	16kB	256kB	TournamentBP	192	64
3	0.995125	64	64	64kB	16kB	512kB	TournamentBP	192	64
4	0.995382	32	64	64kB	16kB	512kB	BiModeBP	192	64
5	0.995868	64	64	64kB	16kB	512kB	BiModeBP	192	64
6	0.99591	64	64	64kB	16kB	256kB	TournamentBP	192	64
7	0.996465	32	64	64kB	16kB	256kB	BiModeBP	192	64
8	0.997072	32	32	64kB	16kB	512kB	BiModeBP	192	64
9	0.997425	32	32	64kB	16kB	512kB	TournamentBP	192	64
10	0.997721	64	32	64kB	16kB	512kB	TournamentBP	192	64

Table 1: Input arguments for top 10 configuration

3.2 Plots

Configuration numbers in the plots are according to Table 1.

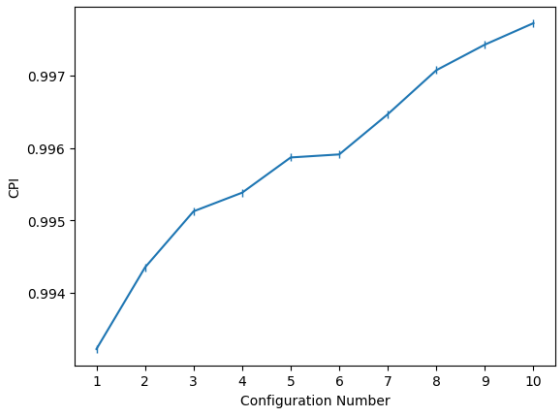

Figure 2: Cycles Per Instruction

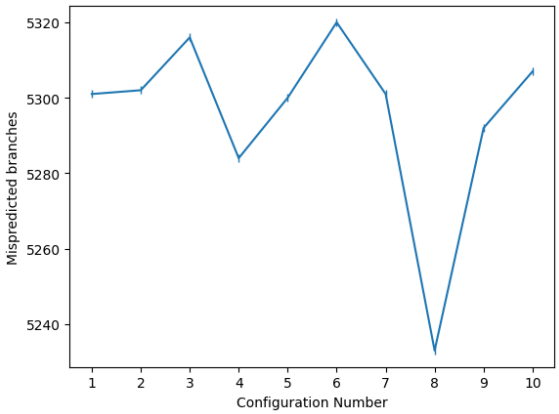

Figure 3: Mispredicted branches detected during execution

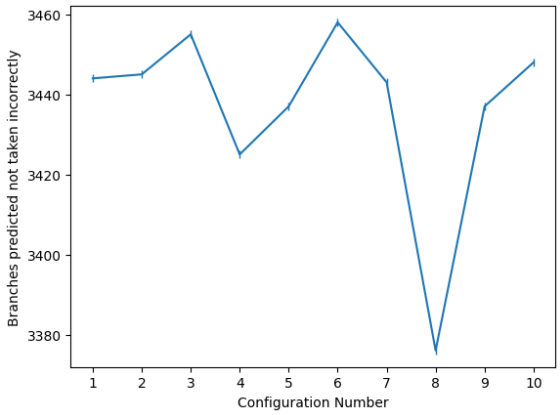

Figure 4: Number of branches that were predicted not taken incorrectly

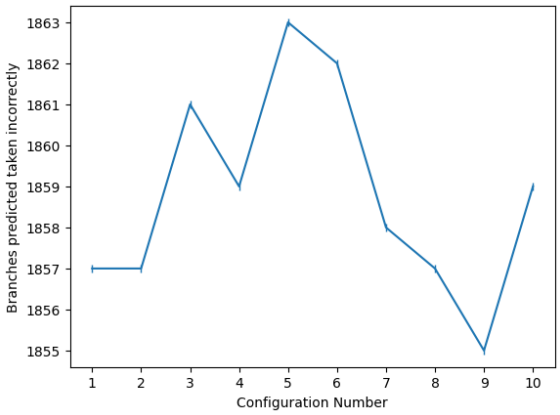

Figure 5: Number of branches that were predicted taken incorrectly

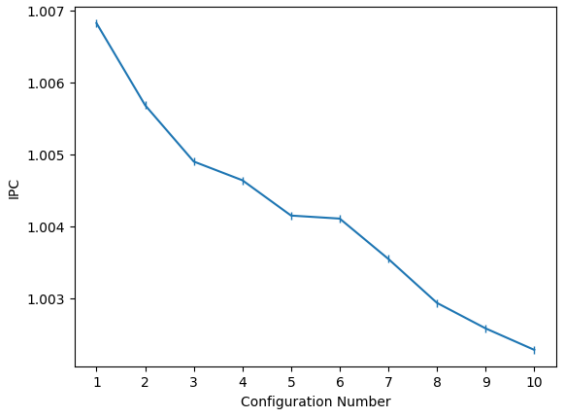

Figure 6: Instructions Per Cycle

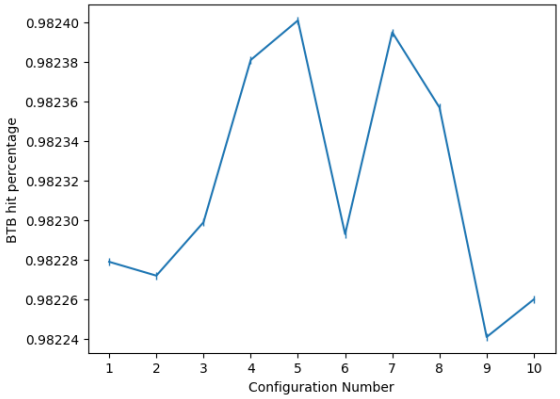

Figure 7: Number of BTB hit percentage

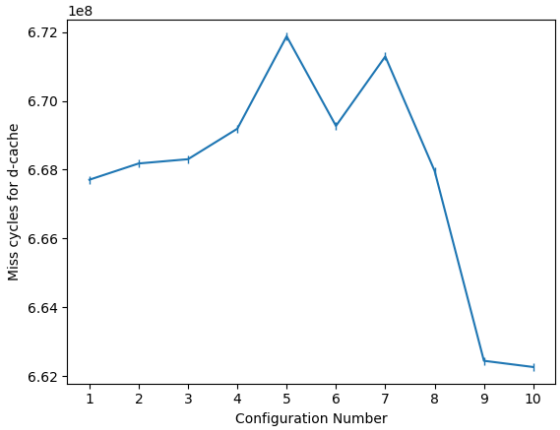

Figure 8: Number of overall miss cycles for d-cache

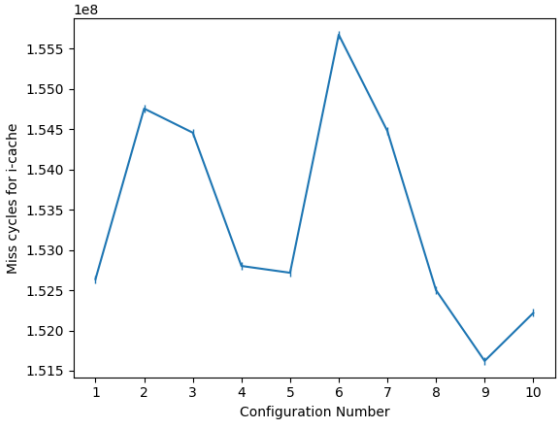

Figure 9: Number of overall miss cycles for i-cache

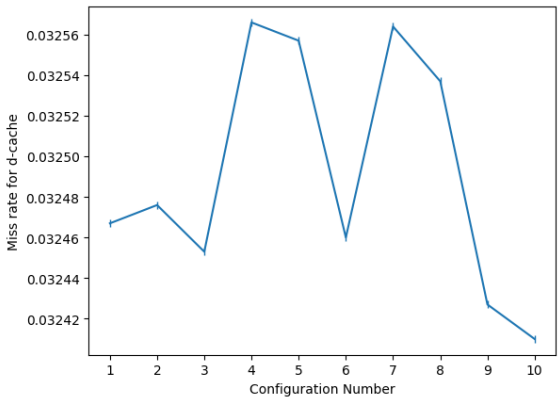

Figure 10: Number of overall miss rate for d-cache

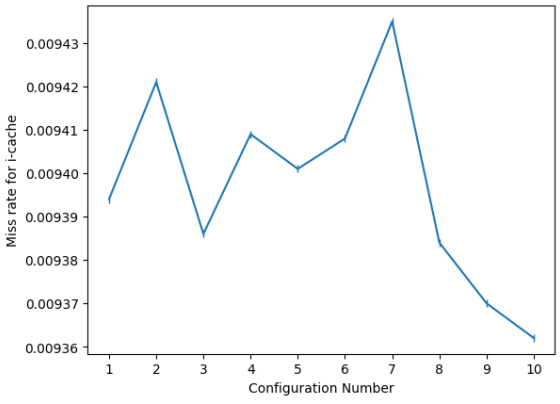

Figure 11: Number of overall miss rate for i-cache

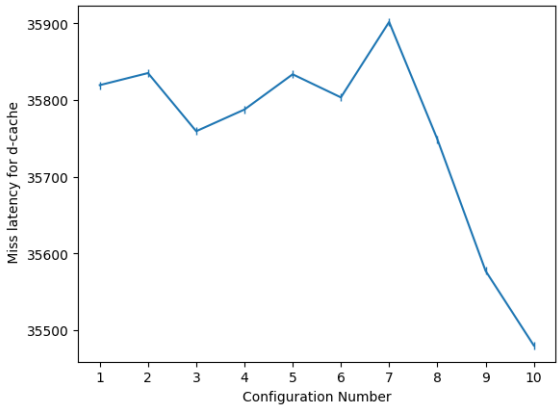

Figure 12: Average overall miss latency for d-cache

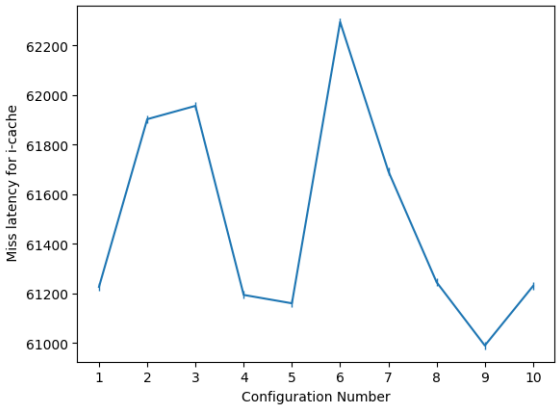

Figure 13: Average overall miss latency for i-cache

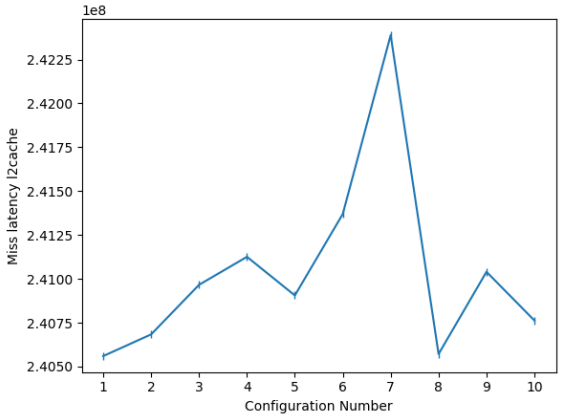

Figure 14: Number of overall miss cycles for l2-cache

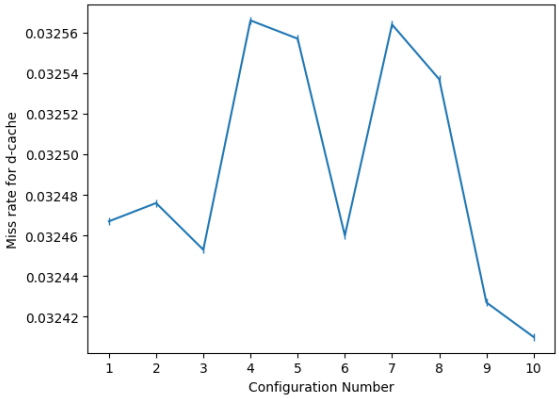

Figure 15: Number of overall miss rate for l2-cache

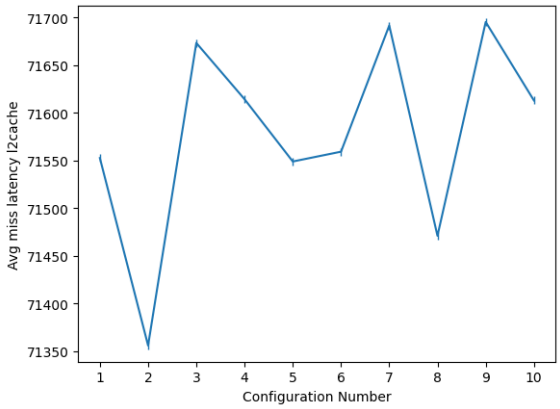

Figure 16: Average overall miss latency for l2-cache

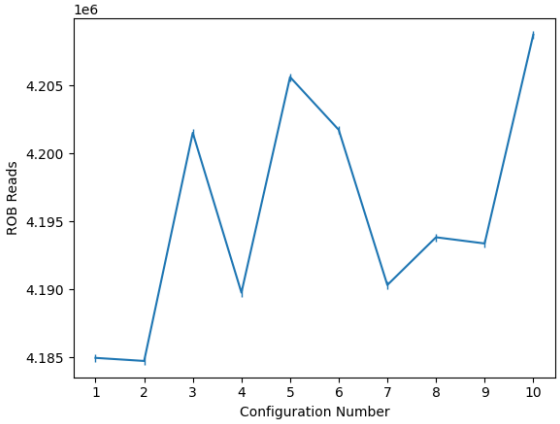

Figure 17: Number of ROB reads

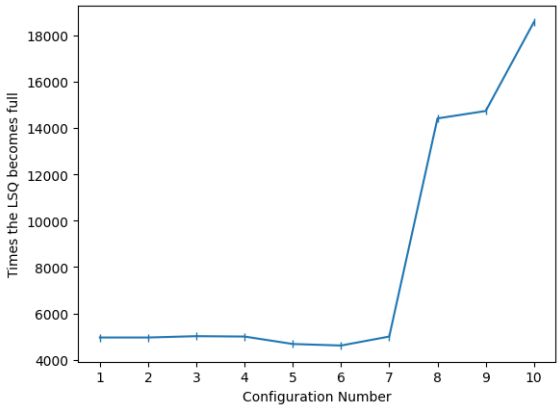

Figure 18: Number of times the LSQ has become full, causing a stall

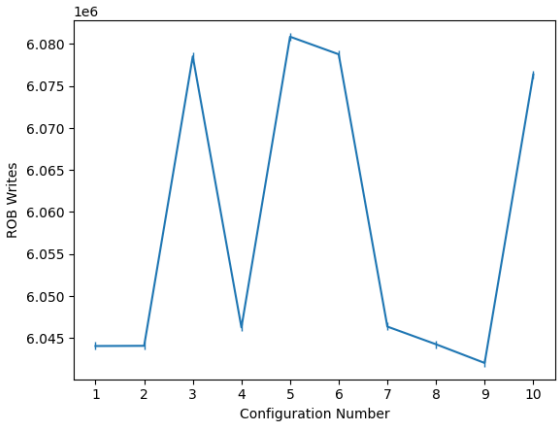

Figure 19: Number of ROB writes

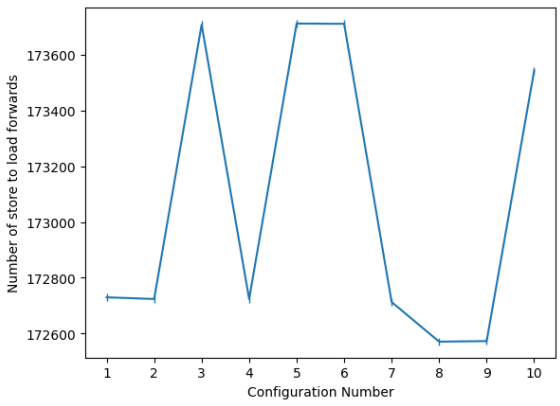

Figure 20: Number of loads that had data forwarded from stores

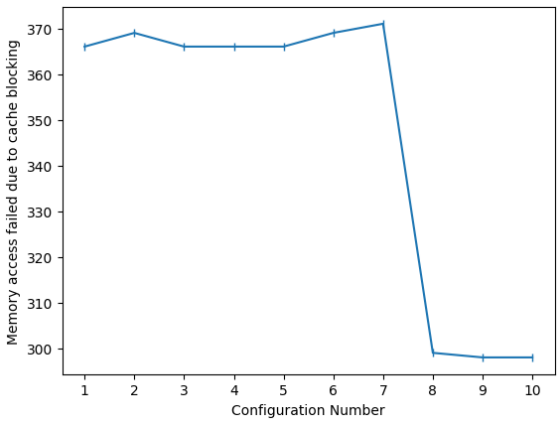

Figure 21: Number of times access to memory failed due to the cache being blocked

4 Analysis

4.1 Understanding the benchmark

Given benchmark is **sieve.c**. The **sieve.c** benchmark is a program that calculates all the prime numbers below a specified maximum value. The program is implemented using the Sieve of Eratosthenes algorithm, which is a simple and efficient algorithm for finding all prime numbers up to a given limit. The algorithm works by creating an array of numbers from 0 to the maximum value and marking each number as either prime or non-prime. The program iterates over the array and eliminates all multiples of each prime number, leaving only the prime numbers as true. It accomplishes this using a series of memory allocation instructions, loop instructions, and conditional statements, making it a good benchmark for evaluating the performance of lower-level computing systems. To run the benchmark we set the maximum value to 10,000 and measure the time taken to complete the computation.

4.2 Reasoning for top configurations

- **LQ entries**: We observe that the size of the load queue is not having much effect on CPI. This can happen as our program is not highly memory load instruction dependent; a program with a higher dependency on memory loads will take advantage of bigger LQ more effectively. It is also possible that the performance of your benchmark is being limited by other bottlenecks, such as cache misses, branch mispredictions, or other types of stalls. In these cases, increasing the size of the load queue may have little effect on performance.

- **SQ entries**: The impact of the increase in the size of SQ depends on multiple factors like the processor architecture, memory store dependency of the benchmark as well as memory bandwidth limitations. Here we observe that a higher number of SQ entries does have some advantage. With more SQ entries, the benchmark has been able to take advantage and, as a result, reduce the number of memory stalls and improve resource utilization.

- **l1d cache & l1i cache**: Increased L1 cache size reduces the number of times the processor must access main memory, which is much slower than the L1 cache. Bigger caches, on the other hand, have higher access latencies due to the additional levels of hierarchy involved, and they use more power. As a result, the optimal size of the L1 cache is determined by a variety of factors, such as program characteristics, processor architecture, and available hardware resources. In the top-10 configurations, we can observe that bigger L1 cache sizes give better results with our benchmark.

- **l2 cache**: L2 cache is a larger, slower memory cache that is typically larger in size than L1 cache and is located on the processor chip or on a separate chip. L2 cache, like L1 cache, is larger and can store more data and instructions, reducing the number of times the processor must access main memory. However, because L2 cache has a higher access latency than L1 cache, the impact of L2 cache size on CPI and the running time is usually less significant than that of L1 cache size. We clearly see this as top-10 configurations have a balance of both 256kB and 512kB L2 caches,

- **Choice of branch predictor**: For branch prediction, BiModal predictor and Tournament Predictor have significantly greater prediction accuracy compared to Local predictor. Therefore, these predictors result in lower CPI.

- **ROB**: A larger ROB can increase the number of instructions that can be executed in parallel, which leads to reducing CPI and improvement in overall performance. This is because a larger ROB can store more instructions, allowing the processor to take advantage of instruction-level parallelism more effectively. A larger ROB, on the other hand, necessitates more hardware resources and may result in higher power consumption.

- **Instruction Queue Length**: A higher instruction queue length means that there are more instructions waiting to be executed by the processor. More instructions in the instruction queue lead to the processor having a greater opportunity to find and execute instructions with fewer

dependencies or stalls, which can lead to a reduction in CPI. Hence, we can observe all the top-10 configurations have 64 IQ entries.

5 Run Instructions

5.1 Installing gem5

Before installing gem5, it is necessary to download a few dependencies.
Dependencies :

- **git** : gem5 uses git for version control.
- **SCons** : gem5 uses SCons as its build environment. SCons 3.0 or greater must be used.
- **Python 3.6+** : gem5 relies on Python development libraries. gem5 can be compiled and run in environments using Python 3.6+.
- **protobuf 2.1+** (Optional) : The protobuf library is used for trace generation and playback.

Use the following command to downaload the dependencies

```
$ sudo apt install build-essential git m4 scons zlib1g zlib1g-dev \
libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev \
python-dev python
```

To get the code for the gem5

```
$ git clone https://gem5.googlesource.com/public/gem5
```

gem5's build system is based on SCons, an open-source build system implemented in Python. The main scons file is called SConstruct and is found in the root of the source tree.Within the root of the gem5 directory, gem5 can be built with SCons using:

```
$ scons build/{ISA}/gem5.{variant} -j {cpus}
```

where ISA is the target Instruction Set Architecture, and variant specifies the compilation settings. For most intents and purposes opt is a good target for compilation. The -j flag is optional and allows for parallelization of compilation with cpus specifying the number of threads. In our case, we build gem5 on 9 threads with opt and targeting x86 architecture with following command:

```
$ scons build/X86/gem5.opt -j9
```

5.2 Testing the default gem5 build

Now that we have a gem5 binary, we can run your first simulation! gem5's interface is Python scripts. The gem5 binary reads in and executes the provided Python script which creates the system under test and executes the simulator. In this example, the script creates a simple system and executes a "hello world" binary.

```
$ build/X86/gem5.opt configs/learning_gem5/part1/simple.py
```

After running this command, you'll see gem5's output as well as Hello world, which comes from the hello world binary! Now, you can start digging into how to use and extend gem5.

5.3 Modify to run our code

Please follow the below steps in order to run our code using varying configurations of the architecture. Our submission directory tree is as follows:

```
Group_9_HPCA_Assignment_1
└── source_code
```

```
    ├── config.py
    ├── exe.py
    ├── options.py
    ├── get_stats.py
    ├── sieve
    └── sieve.c
└── ...
```

After installing, building and testing gem5 as per the instructions in sections 5.1 and 5.2, your gem5 directory will look similar to the following directory tree:

```
gem5
├── build
├── build_opts
├── build_tools
├── configs
├── ext
├── include
└── ...
```

1. Place the **source_code** folder of Group_9_HPCA_Assignment_1 directory in the **configs** folder of gem5 directory.

2. Modify the **exe.py** file in the source_code folder to run your choice of the configuration. The following modifications can be made:

   (a) **Modification Type 1 : Run a particular configuration** In order to run a particular configuration of your choice, follow the Table 2 below to obtain a binary number and edit the exe.py file as shown in Figure 22.

	0	1
LQEntries	32	64
SQEntries	32	64
l1d_size	32kB	64kB
l1i_size	8kB	16kB
l2_size	256kB	512kB
bp_type	TournamentBP	BiModeBP
ROBEntries	128	192
numIQEntries	16	64

Table 2: Reference Table

For example, if I want my system configuration to be the following,

LQEntries	32	0
SQEntries	64	1
l1d_size	32kB	0
l1i_size	8kB	0
l2_size	512kB	1
bp_type	TournamentBP	0
ROBEntries	192	1
numIQEntries	64	1

then, using the Reference Table 2, we get a binary number 01001011 which can then be put in **exe.py** as shown in Figure 22.

```python
import os
import subprocess

LQEntries_List = [32, 64]
SQEntries_List = [32, 64]
l1d_size_List = ["32kB", "64kB"]
l1i_size_List = ["8kB", "16kB"]
l2_size_List = ["256kB", "512kB"]
bp_type_List = ["TournamentBP", "BiModeBP"]
ROBEntries_List = [128, 192]
numIQEntries_List = [16, 64]

for i in [0b01001011]:

    bin_i = [int(j) for j in list(format(i, '08b'))]

    LQEntries = LQEntries_List[bin_i[0]]
    SQEntries = SQEntries_List[bin_i[1]]
    l1d_size = l1d_size_List[bin_i[2]]
    l1i_size = l1i_size_List[bin_i[3]]
    l2_size = l2_size_List[bin_i[4]]
    bp_type = bp_type_List[bin_i[5]]
    ROBEntries = ROBEntries_List[bin_i[6]]
    numIQEntries = numIQEntries_List[bin_i[7]]

```

Figure 22: Modification Type 1 to exe.py

(b) **Modification Type 2 : Run a range of configurations**
In order to run a range of configurations simply edit the **exe.py** file as shown in Figure 23. The example shown in the figure will run the variations of configurations from index 100 to 239 (both inclusive). It is to be noted that the starting index must be greater than or equal to 0 and the stopping index must be less than or equal to 256, because there are a total of 256 possible configurations.

```python
import os
import subprocess

LQEntries_List = [32, 64]
SQEntries_List = [32, 64]
l1d_size_List = ["32kB", "64kB"]
l1i_size_List = ["8kB", "16kB"]
l2_size_List = ["256kB", "512kB"]
bp_type_List = ["TournamentBP", "BiModeBP"]
ROBEntries_List = [128, 192]
numIQEntries_List = [16, 64]

for i in range(100, 240):

    bin_i = [int(j) for j in list(format(i, '08b'))]

    LQEntries = LQEntries_List[bin_i[0]]
    SQEntries = SQEntries_List[bin_i[1]]
    l1d_size = l1d_size_List[bin_i[2]]
    l1i_size = l1i_size_List[bin_i[3]]
    l2_size = l2_size_List[bin_i[4]]
    bp_type = bp_type_List[bin_i[5]]
    ROBEntries = ROBEntries_List[bin_i[6]]
    numIQEntries = numIQEntries_List[bin_i[7]]
```

Figure 23: Modification Type 2 to exe.py

3. Now that a modification of your choice has been made, running the code is a one-liner. Simply open the terminal and change directory to **source_code** and then execute **exe.py**.

```
$ cd ..path/to/source_code
$ python exe.py
```

4. Run the above code with the range (0, 256) to get the outputs for all the desired configurations in the **outputs** folder.

5. Create an empty folder named **graphs** in the same directory as **get_stats.py** file.

```
$ mkdir graphs
```

6. Execute the **get_stats.py** file and all the required graphs will be stored in the **graphs** folder. Along with it a file named **top_10.csv** will be created which stores the parameters for the top 10 configurations sorted based on their CPI values.

```
$ python3 get_stats.py
```

5.4 Running the Top 10 Configurations

The top 10 configurations obtained in our assignment are highlighted in Table 1. In order to run these configurations, a simple modification to the **exe.py** python file is all that is needed. It is similar to the Modification of Type 1 made in section 5.3.

In order to make it even more simple, we have provided the required binary numbers of the top 10 configurations as shown below using Reference Table 2.

Configurations	CONFIG 1		CONFIG 2		CONFIG 3		CONFIG 4	
LQEntries	32	0	32	0	64	1	32	0
SQEntries	64	1	64	1	64	1	64	1
l1d_size	64kB	1	64kB	1	64kB	1	64kB	1
l1i_size	16kB	1	16kB	1	16kB	1	16kB	1
l2_size	512kB	1	256kB	0	512kB	1	512kB	1
bp_type	TournamentBP	0	TournamentBP	0	TournamentBP	0	BiModeBP	1
ROBEntries	192	1	192	1	192	1	192	1
numIQEntries	64	1	64	1	64	1	64	1

Configurations	CONFIG 5		CONFIG 6		CONFIG 7		CONFIG 8	
LQEntries	64	1	64	1	32	0	32	0
SQEntries	64	1	64	1	64	1	32	0
l1d_size	64kB	1	64kB	1	64kB	1	64kB	1
l1i_size	16kB	1	16kB	1	16kB	1	16kB	1
l2_size	512kB	1	256kB	0	256kB	0	512kB	1
bp_type	BiModeBP	1	TournamentBP	0	BiModeBP	1	BiModeBP	1
ROBEntries	192	1	192	1	192	1	192	1
numIQEntries	64	1	64	1	64	1	64	1

Configurations	CONFIG 9		CONFIG 10	
LQEntries	32	0	64	1
SQEntries	32	0	32	0
l1d_size	64kB	1	64kB	1
l1i_size	16kB	1	16kB	1
l2_size	512kB	1	512kB	1
bp_type	TournamentBP	0	TournamentBP	0
ROBEntries	192	1	192	1
numIQEntries	64	1	64	1

In order to execute all of them at once simply make use of the following array construct and modify the **exe.py** as shown in Figure 24.

```
[0b01111011, 0b01110011, 0b11111011, 0b01111111, 0b11111111, 0b11110011,
    0b01110111, 0b00111111, 0b00111011, 0b10111011]
```

```python
import os
import subprocess

LQEntries_List = [32, 64]
SQEntries_List = [32, 64]
l1d_size_List = ["32kB", "64kB"]
l1i_size_List = ["8kB", "16kB"]
l2_size_List = ["256kB", "512kB"]
bp_type_List = ["TournamentBP", "BiModeBP"]
ROBEntries_List = [128, 192]
numIQEntries_List = [16, 64]

for i in [0b01111011, 0b01110011, 0b11111011, 0b01111111, 0b11111111, 0b11110011, 0b01110111, 0b00111111, 0b00111011, 0b10111011]:
    bin_i = [int(j) for j in list(format(i, '08b'))]

    LQEntries = LQEntries_List[bin_i[0]]
    SQEntries = SQEntries_List[bin_i[1]]
    l1d_size = l1d_size_List[bin_i[2]]
    l1i_size = l1i_size_List[bin_i[3]]
    l2_size = l2_size_List[bin_i[4]]
    bp_type = bp_type_List[bin_i[5]]
    ROBEntries = ROBEntries_List[bin_i[6]]
    numIQEntries = numIQEntries_List[bin_i[7]]

```

Figure 24: Modification for Top 10 Configurations to exe.py

6 Contribution

Name	Contribution
Suryam Arnav Kalra	• Ran iterations in the range (128, 160). • Wrote the python file **get_stats.py** in source_code folder. • Wrote the sections 3 (Results) and 2.2 (Outputs). • Contributed to sections 2.1 and 5.3 jointly with others.
Hritaban Ghosh	• Ran iterations in the range (224, 256). • Wrote the python files **exe.py**, **config.py**, and **options.py** in source_code folder. • Wrote the section 5.4 (Running the Top 10 Configurations). • Contributed to sections 2.1 and 5.3 jointly with others.
Suhas Jain	• Ran iterations in the range (32, 64). • Wrote the section 4.2 (Reasoning Top Configurations) jointly.
Surkanti Akshitha	• Ran iterations in the range (96, 128). • Wrote the sections 5.1 (Installing gem5) and 5.2 (Testing the default gem5 build).
Ishan Goel	• Ran iterations in the range (192, 224). • Wrote the section 4.2 (Reasoning Top Configurations) jointly.
Srijanak De	• Ran iterations in the range (64, 96).
Aditya Vallakatla	• Ran iterations in the range (160, 192). • Wrote the section 4.1 (Understanding the Benchmark)
Srijan Das	• Ran iterations in the range (0, 32).

Table 3: Contribution Table