HPCA ASSIGNMENT 1 GROUP D

Siddhant Agarwal[*] **Yash Butala** [*] **Udit Desai** [*] **Rohan Ekka** [*] **Yash Gupta** [*]

Veldi Kumar [*] **Yogesh Kumar** [*] **Venktesh Lagaskar** [*] **Faraaz Mallick** [*]

Aadarsh Sahoo [*] **Amatya Sharma** [*] **Shrey Shrivastava** [*] **Raghu Vamshi**[*]

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
West Bengal, India

CONTENTS

1 Introduction **2**

2 Implementation **2**

2.1 System Code 2
2.2 Scripts 2
2.3 Outputs 3

3 Results **3**

4 Analysis **7**

4.1 Understanding the Benchmark 7
4.2 Reasoning for Top Configs 7
4.3 Reasoning the outputs 8

5 Run Instructions **8**

5.1 Installing/Building Gem5 8
5.2 Testing the default gem5 build 8
5.3 How to Modify 8
5.4 First test run on Modified Code 8
5.5 Running the main Simulation 9
5.6 Checking the progress of section 5.5 9
5.7 Running the top10 script 9
5.8 Setting the standard 9

[*]Author names are sorted w.r.t to the last name, and are *not in anyway related to the level of contribution by authors.*

1 Introduction

Gem5 can simulate CPUs with different configurations (e.g. number of cores, pipeline complexity, cache size, etc.) based on configuration scripts. We write a sample configuration script and configure it to an Out-Of-Order CPU with a list of the micro-architectural parameters (provided in the assignment problem statement) that reflects the characteristic of a single-core x86 processor and run the provided benchmark program. We analyse the output statistics of each of these config combinations to select top 10 combinations and finally share our observations along with explanations. The code along with relevant files including output files of top ten configuration will be made available at the following link https://github.com/aaysharma/High-Performance-Computer-Architecture-CS60003-Assignment-1 w.e.f. 8^{th} April 2021 i.e. after the submission deadline.

2 Implementation

The overall code can be found at this link.

2.1 System Code

Files Added : se_modified.py, Options_modified.py.

We briefly discuss the overall implementation for the project. Among gem5 source files, we modify **se.py** and **Options.py** files as follows:

1. **Options_modified.py** : Added arguments corresponding to LQ, SQ, IQ, ROB entries, changed default values of parameters as specified in the assignment.
2. **se_modified.py** : Imported the modified **Options_modified.py** as Options. Inherited new arguments corresponding to LQ, SQ, IQ, ROB entries from Options.

2.2 Scripts

Files Added : run.py, final_select_top10.py.

This completes the modifications required to build our required system. But in order to run the system with all 11664 different possible combinations of various parameters such as l1 cache size/associativity or l2 cache size/associativity, we wrote a python script **run.py** which enumerates and runs over the specified range of given arguments. We then partitioned the 11664 combinations to be run separately by group members . Note that we use *multi-core processing* to run commands in **run.py** on 8 different CPU cores to parallelize the process and reduce the time to run simulations on the benchmark. Also note that the final output files corresponding to different systems with different arguments is generated in sub-folder *Stats/Assgn1/* in the GitHub repository with the name as : **out_{l1d_size}_{l1i_size}_{l2_size}_{l1_assoc}_{l2_assoc}_{bp}_{lqent}_{sqent}_{robent}_{numiq}.txt**

Furthermore in order to read the output files and extract top 10 w.r.t CPI values, we wrote a script, **final_select_top10.py**, which first extracts out the top ten system configurations and then produces executable commands corresponding to those configurations, so that we can run the top 10 configurations by running the output file of **final_select_top10.py**. However, we observed a lot of ambiguity when the same system is simulated on the same benchmark on different machines. The SimTick values corresponding to the runs, varied from machine to machine, to remedy this, following measures were taken :

- **Find top 110 configurations or 10 from each machine** : We obtained top 10 configurations from the outputs of simulations of each machine. After obtaining the top 110 configurations, we ran all these on a single machine, thereby removing any kind of machine dependent ambiguity (in fact we ran on 3 different machines and observed the exact same output). The outputs of these 110 system configurations were then used to extract the top 10 configurations using the same script used before i.e. **final_select_top10.py**

- **Normalization** : To highlight the error incurred due to use of multiple machines, we ran a specific system configuration on all the systems and compared the obtained Sim-Ticks values from various machines. We observed two sets of SimTick values across all the machines, as depicted in Table 5.Another plausible explanation to this might be because the benchmark program uses random generation of data which might vary from machine to machine, depending on the seed and various other performance parameters.

As per the assignment, we were assigned **qsort5.c** as a benchmark to test our program. We compiled the c file using standard gcc compiler along with the $lpthread$ library as pthreads are being deployed in the c code.

2.3 OUTPUTS

Files Added : 10 txt files, New_final_top10.txt.

Output contains the following files:

- **10 txt files :** Ten files corresponding to the outputs and system statistics when run on given benchmark (such as CPI, mispredicts, misses, latency etc) for final top 10 system configurations w.r.t CPI.
- **final_top10.sh** : The file containing the commands (and therby the arguments) corresponding to that of top 10 system configurations.

The required output parameters (like CPI, branch mispredicts etc) for top 10 system configurations are depicted in Table 2, Table 3, Table 4.

3 RESULTS

#	l1d(l1i) size	l2 size	l1d(l1i) assoc	l2 assoc	bp type	LQ entries	SQ Entries	ROB Entries	IQ Entries
1	64 kB	256 kB	8	8	TournamentBP	64	64	192	64
2	64 kB	512 kB	8	8	TournamentBP	64	64	192	64
3	64 kB	128 kB	8	4	TournamentBP	64	64	192	64
4	64 kB	256 kB	8	4	TournamentBP	64	64	192	64
5	64 kB	512 kB	8	4	TournamentBP	64	64	192	64
6	64 kB	128 kB	8	8	TournamentBP	64	64	192	64
7	64 kB	512 kB	8	8	TournamentBP	32	64	192	64
8	64 kB	256 kB	8	8	TournamentBP	32	64	192	64
9	64 kB	128 kB	8	8	TournamentBP	32	64	192	64
10	64 kB	512 kB	8	4	TournamentBP	32	64	192	64

Table 1: Table lists the input arguments for top 10 systems.

#	cpi	branchMispreds	predNotTakenInc	predTakenInc	ipc
1	1.271791	7641	5301	2340	0.786293
2	1.271791	7641	5301	2340	0.786293
3	1.271791	7641	5301	2340	0.786293
4	1.271791	7641	5301	2340	0.786293
5	1.271791	7641	5301	2340	0.786293
6	1.271791	7641	5301	2340	0.786293
7	1.273201	7592	5253	2339	0.785422
8	1.273201	7592	5253	2339	0.785422
9	1.273201	7592	5253	2339	0.785422
10	1.273201	7592	5253	2339	0.785422

Table 2: Table lists the output parameters such for top 10 systems (for better typesetting the table has been divided into 3 parts with this table being the first part)

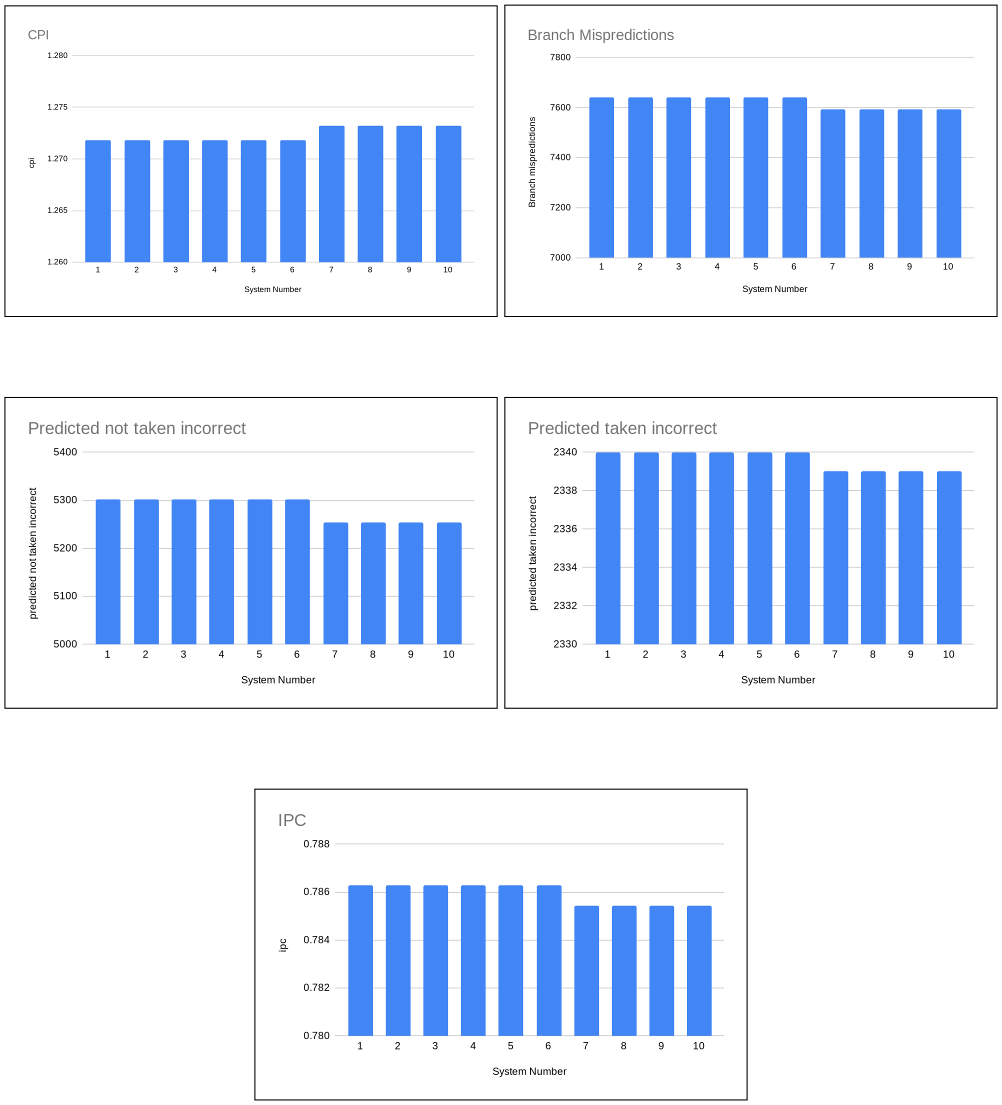

Figure 1: The set of plots correspond to the columns of Table 2 from left to right. Each of the above plots visualises the values in the corresponding column against the system number.

#	BTBHitRatio	ovrallMis(d)	ovrallMis(i)	dmdAvgMisLtncy(d)	dmdAvgMisLtncy(i)
1	0.960967	15559	1754	43852.14345	52259.12201
2	0.960967	15559	1754	43852.14345	52259.12201
3	0.960967	15559	1754	43852.14345	52259.12201
4	0.960967	15559	1754	43852.14345	52259.12201
5	0.960967	15559	1754	43852.14345	52259.12201
6	0.960967	15559	1754	43852.14345	52259.12201
7	0.961017	15493	1762	44195.31401	51982.40636
8	0.961017	15493	1762	44195.31401	51982.40636
9	0.961017	15493	1762	44195.31401	51982.40636
10	0.961017	15493	1762	44195.31401	51982.40636

Table 3: Table lists the output parameters such for top 10 systems (for better typesetting the table has been divided into 3 parts with this table being the second part)

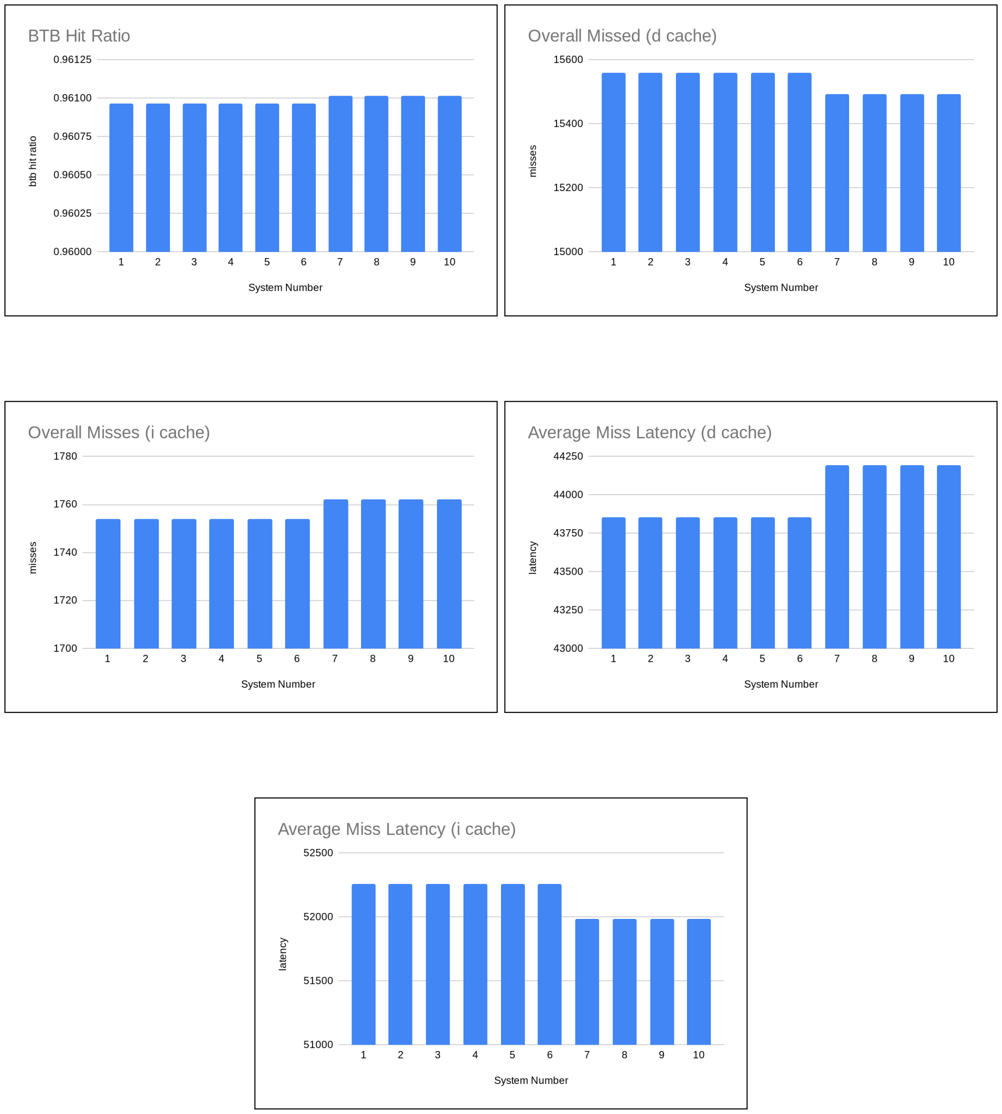

Figure 2: The set of plots correspond to the columns of Table 3 from left to right. Each of the above plots visualises the values in the corresponding column against the system number.

#	ovrallmisrate(d)	ovrallmisrate(i)	ROB (rds+wrts)	lsqFulEvnts	forwLds	blckdByCache
1	0.091281	0.019989	3686714	3955	20577	276
2	0.091281	0.019989	3686714	3955	20577	276
3	0.091281	0.019989	3686714	3955	20577	276
4	0.091281	0.019989	3686714	3955	20577	276
5	0.091281	0.019989	3686714	3955	20577	276
6	0.091281	0.019989	3686714	3955	20577	276
7	0.090925	0.020088	3685685	3901	20575	282
8	0.090925	0.020088	3685685	3901	20575	282
9	0.090925	0.020088	3685685	3901	20575	282
10	0.090925	0.020088	3685685	3901	20575	282

Table 4: Table lists the output parameters such for top 10 systems (for better typesetting the table has been divided into 3 parts with this table being the third part)

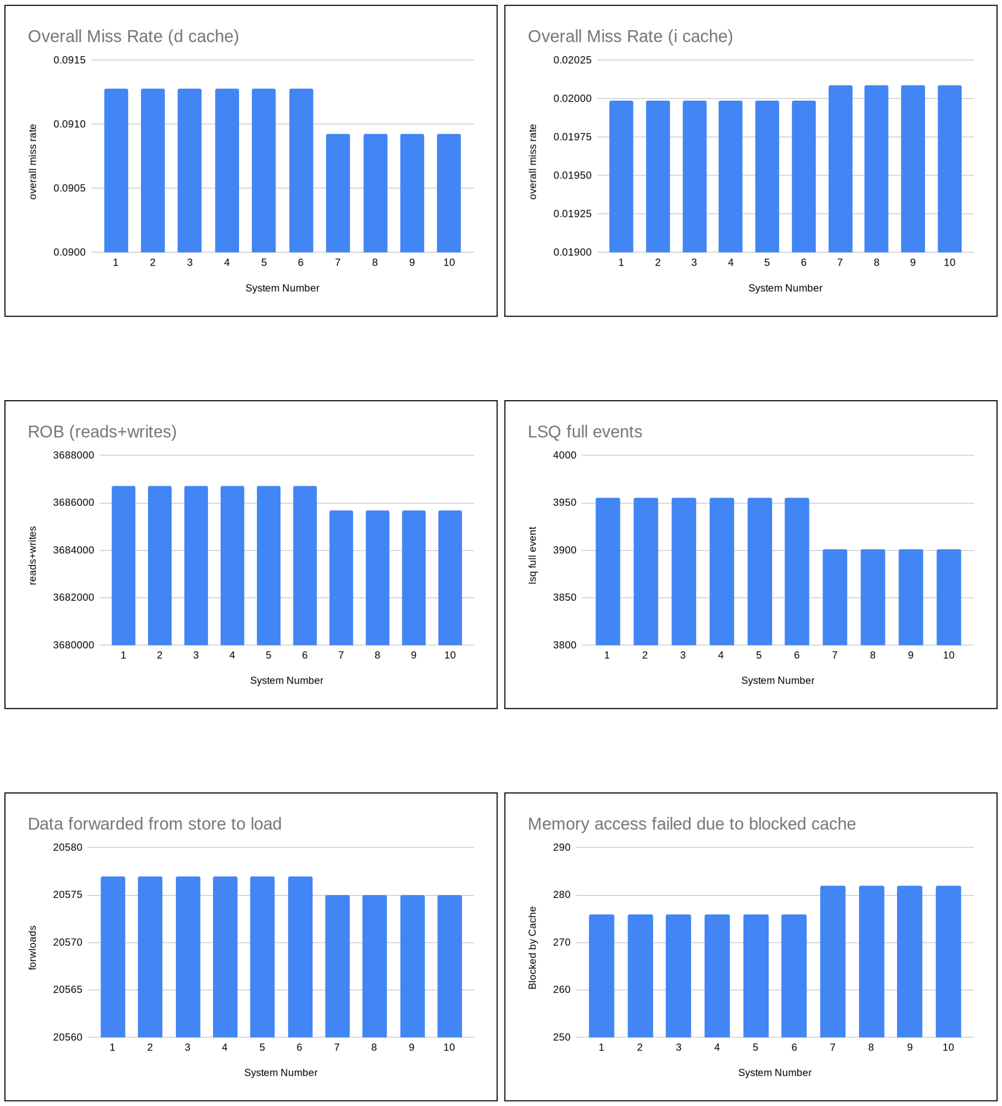

Figure 3: The set of plots correspond to the columns of Table 4 from left to right. Each of the above plots visualises the values in the corresponding column against the system number.

Machine Name	SimSeconds
Siddhant	0.001793
Yash	0.001793
Aadarsh	0.001793
Faraaz	0.000289
Udit	0.001794
Rohan	0.001794
Yogesh	0.000286
Amatya	0.001794

Table 5: Results of a standard run (from outputs obtained by running the command from Section 5.8) across all machines

4 Analysis

4.1 Understanding the Benchmark

We employ qsort5.c benchmark for testing our system on gem5.

Quicksort is in a way an inplace sorting algorithm. Being a recursive routine, it does use additional non constant memory space but the array updates are inplace. This means that there is a lot of spatial and temporal locality that is exploited so, cache choices will be an important factor. Also, there will be a lot of scope of store to load transfers as well, so store-queue will also play an important role. There is only one major loop in quicksort algorithm (for partitioning), but it will be running in every recursive call.

The given benchmark is a multithreaded implementation of quicksort. The basic idea of this program is that at most THREADS threads will be running. Once a recursive call gets a piece of the array of which the size is below THRESHOLD (which means the data will fit in the cache of one of the cores), a separate thread will be used for this if one is available. The count of free threads is kept in the variable available for which a mutex is used when updating it. **But**, on a deeper analysis of code, we observe that for the given value of N, **threading is never used since N $<$ threshold**.

In a nutshell the code employs function calls, standard library calls, other library calls, recursion, includes preprocessor, defines input/output threads, break/continue, and arrays.

4.2 Reasoning for Top Configs

In this section, we will provide theoretical justifications to the observed trends. We will explain why the specific choice of parameters led to the best performance and will also relate the observed statistics.

- *Choice of Branch Predictor:* Out of the three branch predictors, *Local*, *BiModal* and *Tournament*, clearly Tournament branch predictors are the best. They use both Local and Global predictors and choose between them. They make multiple predictions and choose the right prediction based on the context of the particular branch. Therefore the Tournament predictors, by adding global information, can further improve the performance. Clearly, Tournament predictors will be better than both Local and BiModal predictors and we can clearly see this as they enhance the performance of the cpu.
- *Choice of L1 cache:* Quick-sort is in-place w.r.t. the array elements that means it uses the same array space for manipulating the elements. So, it will have a high spatial locality. Hence a large cache will help in the performance. Moreover, increasing the associativity will reduce conflict misses and also add to in the large temporal locality of the algorithm. Hence a *64kB* L1 cache with set associativity of *8* will be optimal out of the given choices.
- *Ambiguity in L2 cache :* Considering the optimal size of L1 (data) cache, which is the largest of choices possible, we can infer that increasing L2 cache has diminishing marginal return with increasing size.
- *ROB entries:* Larger ROB table will mean more number of instructions can be issued before getting committed. So, if some instructions take some time to get committed, it will not restrict the other instructions from getting issued. So, a larger ROB table will in general lead to better performance.
- *IQ entries:* Again, a large instruction queue means that more instructions can be kept in the queue and issued quickly. So best performance is seen with *64* entry queue.
- *SQ entries:* A large store queue will first of all mean that more number of store instructions can be issued. Since, the instructions are fetched from the instruction queue in order, it would also mean that it will be less likely that instructions will be stalled because a store instruction could not be fetched to the store queue. Finally, larger store queue means more store to load transfers are possible leading to lesser memory overheads. So, larger store queues will increase the performance.
- *LQ entries:* For reasons similar to the store queue, larger load queues will help. But store to load transfer will not really depend on the size of the load queue so size of store queue will be more important.

4.3 Reasoning the outputs

We see that the change in the number of LQ entries brings about several changes in the corresponding values in the outputs. We now explain the statistics obtained in the runs and justifying their dependence on input parameters with theoretical reasoning, as follows:

- First of all, larger Load Queue means more instructions will be fetched from the instruction queue and so more number of instructions will be executed in general. So, more branch instructions will be executed and hence there will be more number of mispredictions.
- Former point can be extended to the observation that number of predict not taken incorrect predictions. Looking at the code, we observe that based on history most branches will be predict not taken. This in turn implies that with increasing LQ size, mispredicts will be more (which in turn justifies the plots in Figure 1).
- More number of instructions executed and more mispredictions means that there will more ROB reads and writes.
- This will also mean that there are more load events so more cache misses.
- The drop in LSQ full events can be reasoned with the decreasing LQ size.
- When comparing to the outputs from rest of the cases, top 10 configurations have lesser miss rates and conflicts. This can be supported with the fact that higher offering higher hit rates for the same cache size.

5 Run Instructions

5.1 Installing/Building Gem5

Follow this : http://learning.gem5.org/book/part1/building.html

5.2 Testing the default gem5 build

Run this by opening terminal in gem5 folder:

```
$ build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/
hello    /bin/x86/linux/hello --cpu-type=TimingSimpleCPU --l1d_size
=64kB --l1i_size=16kB    --caches
```

5.3 How to Modify

1. Download the following files from Drive: Options_modified.py, se_modified.py, run.py, qsort5
2. Copy Options_modified.py to ./gem5/configs/common/
3. Copy se_modified.py to ./gem5/configs/example/
4. Copy qsort5 to ./gem5/tests/
5. Copy run.py to ./gem5/

5.4 First test run on Modified Code

1. Open terminal in ./gem5
2. For a test run the following command :

```
$ python3 run.py
```

5.5 Running the main Simulation

1. First delete the gem5/Stats folder (to clear outputs from previous runs)
2. Open run.py
3. run.py : Comment out lines 27-28

```
# if i > 10:
#    break
```

4. Compile python script making file :

```
$ python run.py --numiqentries X --sqentries Y --lqentries Z
--verbose 2 -n=8
```

5. Fill X , Y, Z from values against ur name in the sheet... -n=8 refers to the number of parallel processes you want to run (set according to ur laptop configs)
6. Use python3 if there's an error

e.g. Aadarsh will run this :

```
$ python run.py --numiqentries 64 --sqentries 64 --lqentries 16
--verbose 2 -n=8
```

If lqentries is "all", then just use this :

```
$ python run.py --numiqentries X --sqentries Y --verbose 2 -n=8
```

Fill X, Y from spreadsheet

e.g. Yogesh will run :

```
$ python run.py --numiqentries 16 --sqentries 16 --verbose 2 -n=8
```

5.6 Checking the progress of section 5.5

Check the number of files in gem5/Stats/Assign1. Final count of files is given in the spreadsheet

Note : Save the gem5/Stats folder to

5.7 Running the top10 script

1. Go to folder where the run output txts are saved (by default it is gem5/Stats, but you may have saved them somwhere else)
2. Copy final_select_top10.py to this folder
3. Open terminal from the folder
4. Run

```
        $ python3 final_select_top10.py > AMATYA_top10.txt
(Write your own name)
```

5. Commit the file to Assignment repo/Stats/

5.8 Setting the standard

Run the following command from gem5 folder:

```
$ build/X86/gem5.opt -d "Stats/Assign1/." --stats-file=out_64kB_64kB
_512kB_8_4_LocalBP_16_64_192_64.txt configs/example/se_modified.py
-c ./tests/qsort5    --l1d_size=64kB --l1i_size=64kB --l2_size=512kB
--l1d_assoc=8 --l1i_assoc=8 --l2_assoc=4 --bp-type=LocalBP
--lqentries=16 --sqentries=64 --robentries=192
--numiqentries=64 --caches
```

Ouputs in the form of sim seconds across all the devices used for our experimentation can be seen in Table 5.

References

[1] https://github.com/aaysharma/High-Performance-Computer-Architecture-CS60003-Assignment-1
[2] http://learning.gem5.org/book/part1/index.html