CS204 - Computer Architecture
Practice Lab - 1.1
Date: 17th Jan 2025. Not graded!

1. Write a C program to find out if your machine is Big-Endian or Little-Endian.

2. Write a C program to convert a Big-Endian representation to Little-Endian, and vice-versa. Clearly print the value stored along with its address.

3. Write a C program to perform Bubble-sort on 1024 random numbers. Compile it using different optimization levels (0, 1, 2, and 3). Run the binaries created with these different optimization levels, one at a time and note down the time taken by the program. Do you see any impact?

4. Comparing performance of C, Java and Python.
   (You can use any code available online. Onus here is more on understanding the performance than testing your coding expertise.)
   Write a simple matrix multiplication program using the three languages. To get the time taken by each of the program, you can either use
   (i) 'time' command at the terminal. When you simply use 'time' command, it gives three different times, namely, *real*, *user* and *sys*. Read about them to understand what they are.
   (ii) Use inbuilt functions in C (like *clock_gettime*), Java (like *System.nanoTime*()), Python (like *time.time*()). This method is more accurate.
   For C based implementation, you can use different compiler optimziation (- O) and check the timings.
   Write your observations on which of these programming languages is fast/slow. Clearly mention by "how much" an implementation is faster or slower when compared to others.

5. To get the exact Dynamic Instruction Counts of each of these implementations, we need specialized tools. To get started, we will use **Valgrind**
   (*https* : *//valgrind.org/info/about.html*). Download the latest version.
   Next follow the steps given below
   Step 1. To profile, you run your program under Valgrind and explicitly request the *callgrind* tool (if unspecified, the tool defaults to *memcheck*).
   valgrind –tool=callgrind <program-to-run program-arguments>
   example: valgrind –tool=callgrind **./a.out**

   Step 2. The above command starts up valgrind and runs the program inside of it. The program will run normally, albeit a bit more slowly, due to Valgrind's instrumentation (read interference). When finished, it reports the total number of collected events:

```
==22417== Events :  Ir
==22417== Collected :  7247606
==22417==
==22417== I refs:  7,247,606
```

Valgrind has written the information about the above 7 million collected events to an output file named callgrind.out.pid. (pid is replaced with the process id, which is 22417 in the above run, id shown in leftmost column).

Step 3. The callgrind output file is a text file, but its contents are not intended for you to read yourself. Instead, you run the annotator callgrind_annotate on this output file to display the information in an useful way (replace pid with your process id):
callgrind_annotate –auto=yes callgrind.out.pid

The output from the annotator will be given in Ir counts which are "instruction read" events. The output shows total number of events occurring within each function (i.e., number of instructions executed) and displays the list of functions sorted in order of decreasing count. Your high-traffic functions will be listed at the top.

For our current experiment, you can treat the number given under "Ir PROGRAM TOTALS" as dynamic instructions executed.

In the report you submit, write the number of dynamic instructions you observed for each of C, Python and Java based implementations.

Submit code for each question in a separate file and place them in single folder. You are required to submit a single compressed file named "your_entry_no.gzip" and upload it on the Google classroom.

Other information:
(1) For your "Under the hood" assignment, you can take help of an application "CPU-Z" from $https://www.cpuid.com/softwares/cpu-z.html$ Its runs quite smoothly on Windows and Linux systems; and on most Android phones. In case you are using Apple products, you need to explore other options like $https://alternativeto.net/software/cpu-z/?platform=mac$
Most of the time many of the processor insider information is not easily revealed by the manufacturers. Reverse engineering would reveal a few of them easily. You can check websites like:
1. $https://www.anandtech.com/$,
2. $https://hothardware.com/$,
3. $https://www.realworldtech.com/$,
4. $https://bit-tech.net/$,
5. $https://www.nextplatform.com/$,
6. $https://www.tomshardware.com/$
7. $https://en.wikichip.org/wiki/WikiChip$
To name a few.