Family Name:_____

Given Names:_____

ID#:_____

CMPT 295 Lab 6 (0.5%)

This is a continuation of Lab 5. The purpose will be to benchmark some more code, but by using more precise tools than `getrusage()`. This is known as *microbenchmarking*.

Part 1: Benchmarking a Loop

- Login to the CSIL machines in the Linux environment, download and unpack the care package into `sfuhome`, and change directories into `lab6`.
- When you open `main.c`, you will see something similar to the benchmarking code in Lab 5: an array of `N` integers will be initialized, and a subroutine `sum_plus()` will be benchmarked 20 times using `getrusage()`.
- The subroutine `sum_plus()` is written in assembly. It computes the sum of all of the positive integers within an array of integers `A[N]`. You may wish to read the code.
- Build the executable and run it. The strange results are because the operating system doesn't update its usage statistics every clock cycle, but every *clock tick*. Therefore, you should always take the results of `getrusage()` with a grain of salt: each measurement consistently contains an experimental error of up to one clock tick, a millisecond scale.
- Comment out the two instances of `getrusage()` and the line that does the cycles computation. At the end of the loop add the line:

```
cycles[i] = end_time - start_time;
```

- Rebuild and run the code and it will tell you another absurd cycle time. To add the time to measure the code correctly, you will have to edit the `main.s` assembly source. Open it in your editor and find the subroutine `call` to `sum_plus`. Before this line, add the following:

```
pushq   %rax
pushq   %rbx
pushq   %rcx
pushq   %rdx
cpuid
rdtscp
movl    %eax, start_time(%rip)
popq    %rdx
popq    %rcx
popq    %rbx
popq    %rax
```

The measurement occurs on the `rdtscp` command. It places the value of the cycle counter (64 bits) into `%edx`:`%eax` — a notation that means the higher 32 bits of the value are in `%edx` and the lower 32 bits are in `%eax`. The next line stores the value in variable `start_time`.

That seems simple enough, so why all the other commands? The problem with simply running `rdtscp` is that the command is placed in the CPU pipeline along with all the other instructions, i.e., it could be run before or after some of the work in the subroutine. Therefore, the *serializing command* `cpuid` is issued to clear the pipeline.

But since `cpuid` has the unfortunate side-effect of writing to registers `%rax`, `%rbx`, `%rcx` and `%rdx`, those registers must be saved and restored. Hence 4× pushes and 4× pops.

- To measure the cycle time after `sum_plus()`, place the same 11 lines of code after the function call, but replace `start_time` with `end_time`. Rebuild and run the code. You can re-run and re-re-run it to get a sense of how long `sum_plus()` takes for `N` = 100.

- You may have noticed some samples are rather large compared to the rest. It's probably due to a *context-switch*: to achieve the appearance of parallelism, the operating system switches rapidly between computing tasks. The large number includes *all* of the cycles run on the processor, not just for your program.

  In `main.c`, adjust your sampling to automatically exclude cases that take longer than 4000 cycles.

Part 2: Inline Assembly

It would be a better approach to include the microbenchmarking code directly within your C source.

- Open `main.c` and just before the call to `sum_plus()`, insert the following:

```
asm volatile (
    "cpuid\n\t"
    "rdtscp\n\t"
    "movl %%eax, %0\n\t"
    : "=r" (start_time)
    :
    : "rax", "rbx", "rcx", "rdx"
);
```

  What does it all mean?

  - "`asm volatile`" tells the compiler you are going to write some inline assembly. The `volatile` keyword is sometimes optional: it suggests to the optimizer that it would be a bad idea to move this code elsewhere.
  - There are four arguments, each separated by colons.
  - The first argument is a string of assembly text. It looks a lot like the commands you issued earlier except for the "`%0`".
  - The second argument is a comma-separated list of output variables. That's where the "`%0`" comes in: it refers to argument 0 in this list, i.e., the integer `start_time`.
  - The third argument is a comma-separated list of input variables, empty in this case.
  - The last argument is a comma-separated list of registers that should be treated as scratch by these commands: the *clobber list*. It's an alternative to pushing them and popping them: instead the compiler is warned not to leave anything important inside them, lest their values get clobbered.

- Place another round of inline assembly after the call to `sum_plus.s` to measure the `end_time`.
- Rebuild your code and run it. Run it 9 times and discard the 2 fastest and 2 slowest average times. Record the middle 5 average times in the table. Repeat for `N` $\in \{150, 200, 250\}$ and record your data in the table. Compute the averages of each data set.

Part 3: Optimization

The `sum_plus()` source uses the branch instruction `jle endif` to test if `A[i]` is greater than 0. When the CPU fails to predict the branch, perhaps close to half the time, CPU cycles are wasted. In Part 3, you will optimize the code and see just how much of a penalty is paid.

- Replace the line `jle endif` with `cmovle %r8d, %ecx`. This is a *conditional move* instruction which will move the value of `%r8d` into `%ecx` only if less than or equal to. The idea is this: if `A[i]` is positive, then add it, otherwise add 0.
- Add a line before the loop that zeroes `%r8d`.
- Re-build and benchmark the code for the same values of `N` as in Part 2. Complete the table and then plot both datasets on the graph paper at the end of this lab. Using a ruler, draw a straight line through each dataset, and compute the slope of each. The slope should have units of cycles per element (CPE).
- This is the last step of Lab 6. Bring your work to your TA at the front of the lab. After verifying your work, the TA will then collect this sheet.

Data Sheet

Version 1: Branch

N	t_1	t_2	t_3	t_4	t_5	μ
100						
150						
200						
250						

Version 2: Conditional Move

N	t_1	t_2	t_3	t_4	t_5	μ
100						
150						
200						
250