Lab 7

Objective:

Continuing on from our Lab 6 …

- The purpose of this lab is to introduce us to the idea of *microbenchmarking*, i.e., testing the performance of very small units of computation. To do so, we shall use more precise tools than `getrusage()` (used in Lab 6).

Lab 7 Submission – Participation Activity 10

- As part of Lab 7, you will need to submit your Participation Activity 10 (the last two pages of Lab 7):
  - Complete Lab 7 during your lab session.
  - Write your name, student number and circle your section (D100 or D200).
  - Save your Participation Activity 10 (the last two pages of Lab 7) as a pdf and upload it onto Crowdmark (the assessment is called *Lab 7 – Participation Activity 10*)

  Participation Activity 10 Deadline: 12:30pm on Tuesday April 4.

- You also need to submit your **completed** Lab 7 - Participation Activity 10 (the last two pages of Lab 7) as part of our Assignment 9. When marking Assignment 9, the TAs will be looking at the correctness of your data sets and your graph. Therefore, make sure you follow all the steps of Lab 7.

Part 1: Benchmarking a Loop

- Login to a CSIL machine in the Linux environment, download `Lab7-Files.zip` and unpack it into `sfuhome/CMPT295`. Change directory into `Lab7` where you will find the usual `makefile` and a collection of source files.
- When you open `main.c`, you see something similar to the benchmarking code in Lab 6: an array of N integers randomly initialized, and a function `sumPlus()` which is benchmarked 20 times using `getrusage()`.
- The function `sumPlus()` is written in assembly. It computes the sum of all of the positive integers within an array of integers `A[n]`. Have a look at the code.
- Build the executable, execute it a few times and have a close look at the results.

  Woa! What is happening? Some of these runs conclude: "`Average of 0 clock cycles`". Our testing does not seem to be working.

  The reason for these strange results is that we are measuring something very small: the time (number of clock cycles) it takes to compute the sum of the elements contained in a

very small array, which is in the order of nanoseconds, with a tool (`getrusage()`) that is meant to measure something much larger, i.e., computation ranging from microseconds to seconds.

Part 2: Inline Assembly

So, let's change our measuring tool! Let's include microbenchmarking code within our C code.

- First, let's remove the "old" benchmarking tool in `main.c` by commenting out the two instances of `getrusage()` and replacing the line that currently computes the number of clock cycles per call to `sumPlus()` at the end of the loop with this one:

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

To complete the change of tools (from benchmarking tool to microbenchmarking tool), we need to add code that will capture the `start_time` as well as the `end_time` so that their difference can be computed properly.

- Still in `main.c`, insert the following code before the call to `sumPlus()`:

```
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 we are going to write some *inline* assembly code in our C program. The `volatile` keyword is sometimes optional: it suggests to the optimizing compiler that it would be a bad idea to move this code elsewhere, i.e., that it would be a bad idea to optimize using the *code motion* technique.
- The parentheses after "`asm volatile`" contain four arguments, each separated by colons ("`:`").
  - The first argument:

```
"cpuid\n\t"
"rdtscp\n\t"
"movl %%eax, %0\n\t"
```

    is a string containing assembly instructions. Let's go over each of them in this order: first "`rdtscp`", then "`movl %%eax, %0`" and finally "`cpuid`":

- Let's start with `rdtscp`, the second instruction in this first argument as it is the core instruction of the microbenchmarking process. We learn from the web page https://www.felixcloutier.com/x86/RDTSCP.html that this instruction reads the current value of the microprocessor's time-stamp counter (i.e., 64-bit value of the clock cycle counter) into the `%edx:%eax` registers, i.e., the higher 32 bits of this counter value are copied into `%edx` and the lower 32 bits are copied into `%eax` registers, clearing the high-order 32 bits of each of `%rax` and `%rdx`. This time-stamp counter is incremented every clock cycle by the microprocessor.
- The next assembly instruction we shall look at in this first argument is

  ```
  movl %%eax, %0
  ```

  This instruction copies the value stored in register `%eax` into the global variable `start_time`, represented here by `%0`.
- These two instructions seem to be doing what is needed in order to time the call to `sumPlus()`, so why would we require the `cpuid` instruction? The reason why we need this instruction is that when `rdtscp` is executed, it is placed in the microprocessor pipeline along with all the other instructions, which means that it could be run before or after some of the work we are timing. Therefore, the serializing [1] instruction `cpuid` (https://www.felixcloutier.com/x86/CPUID.html) is issued to clear the pipeline and allow the timing instructions (the instructions contained in this first argument as well as the call to `sumPlus()`, i.e., all of the instructions of `sumPlus()`) to execute one after the other. This way, we can truly time the execution of `sumPlus()`.

- The second argument

  ```
  "=r" (start_time)
  ```

  is a comma-separated list of output variables. Here, we only have one output variable, namely `start_time`, which is seen as the 0th output variable of the list and is referred to by "`%0`" found in the instruction "`movl %%eax, %0`", which was explained above. This is how the `movl` instruction ends up copying the value stored in register `%eax` (i.e., the lower 32 bits of the starting time-stamp counter) into the global variable `start_time`.

[1] Definition: arranging in a sequence.

  - The third argument is a comma-separated list of input variables, empty in this case.
  - The last argument

    ```
    "rax", "rbx", "rcx", "rdx"
    ```

    is a comma-separated list of registers that should be treated as temporary registers by these assembly instructions. Because `cpuid` writes to registers `%rax`, `%rbx`, `%rcx` and `%rdx`, this argument warns the compiler not to leave anything important inside these registers, in case their values get overwritten. (The effect of this argument is akin to `pushq`'ing and `popq`'ing these registers onto the stack in order to preserve their content.)

- Place another round of inline assembly code ("`asm volatile`") after the call to `sumPlus(…)` in `main.c` to measure the `end_time`. Don't forget to replace `start_time` with `end_time`.
- Rebuild and run the code multiple times to get a sense of how long `sumPlus()` takes for `N` = 100.
- It may be the case that some samples are rather large compared to the rest of the samples. This may be due to context-switching: to achieve the appearance of parallelism, the operating system switches rapidly between computing tasks (called "processes"). The large number of clock cycles may include the execution time of several processes currently run by the microprocessor, not just the execution time of our program. Also, if the first few samples are larger than the rest, this may be due to cache misses, which we shall discuss in class during the coming lectures. In order to ignore these large samples (which would throw off our timing calculation), add the following code in `main.c`, after the computation of `cycles[i]` in the main loop:

```
if (cycles[i] >= 3000) {
   printf("Sample %d completed in %d clock cycles
           - DISCARDED.\n", i+1, cycles[i]);
   i--;
   continue;
}
```

- Note: Depending on the speed of the computer you are using, you may have to tweak this limit of 3000 either upward or downward. This limit of 3000 is the limit I used when I was remotely logged onto our target machine.
- Let's rebuild our code once again and run it 11 times, producing 11 average times.
- Discard the slowest 3 and fastest 3 average times. Then record the middle 5 average times in the table found on the penultimate page of this lab. Repeat for N = {150, 200, 250} and record the data in the table.

- Compute the averages of each data set.

Part 3: Optimization

The `sumPlus()` uses the following instructions

```
testl   %ecx, %ecx
jle     endif
```

to test if `A[i]` is greater than 0. When the microprocessor fails to predict the branch, perhaps close to half the time (50-50 chance that an element of A is negative), microprocessor clock cycles are wasted.

In this part of the lab, we shall optimize the code and see just how much of a penalty such misprediction costs.

- Open `sumPlus.s` and replace the line `jle endif` with `cmovl %r8d, %ecx`. This is a *conditional move* instruction which will move the value of `%r8d` into `%ecx` only if the content of `%ecx` is less than 0. The idea is:

  ```
  if A[i] < 0   (i.e., %ecx < 0)
     %ecx = %r8d (i.e., %ecx = 0)
  sum += %ecx
  ```

  This means that we need to store `0` into `%r8d`.

- Add a line that zeroes `%r8d` before the loop. Let's use the most efficient instruction to do so. If you can't remember which instruction that is, let's have a peek at our Assignment 5.
- Re-build and microbenchmark the code for the same values of `N` as in Part 2, i.e., 11 times for each value of `N`, discarding the slowest 3 and fastest 3 average times, then recording the middle 5 average times in the table.
- Complete the table and then plot both datasets on the same graph. You can use the graph paper at the end of this lab and create the graph by hand or you can use a software application to create the graph. Draw a straight line through each dataset, and compute its slope. The slope should have units of ***clock cycles per element***, i.e., **CPE**.
- Looking at your graph, what conclusion do you reach? Add your conclusion to the end of your Participation Activity 10.
- This is the last step of Lab 7.

Thank you to Brad Bart for having inspired this lab.

Last (Family) name:

First name:

Student number:

Circle your section: D100 D200

Lab 7 - Participation Activity 10

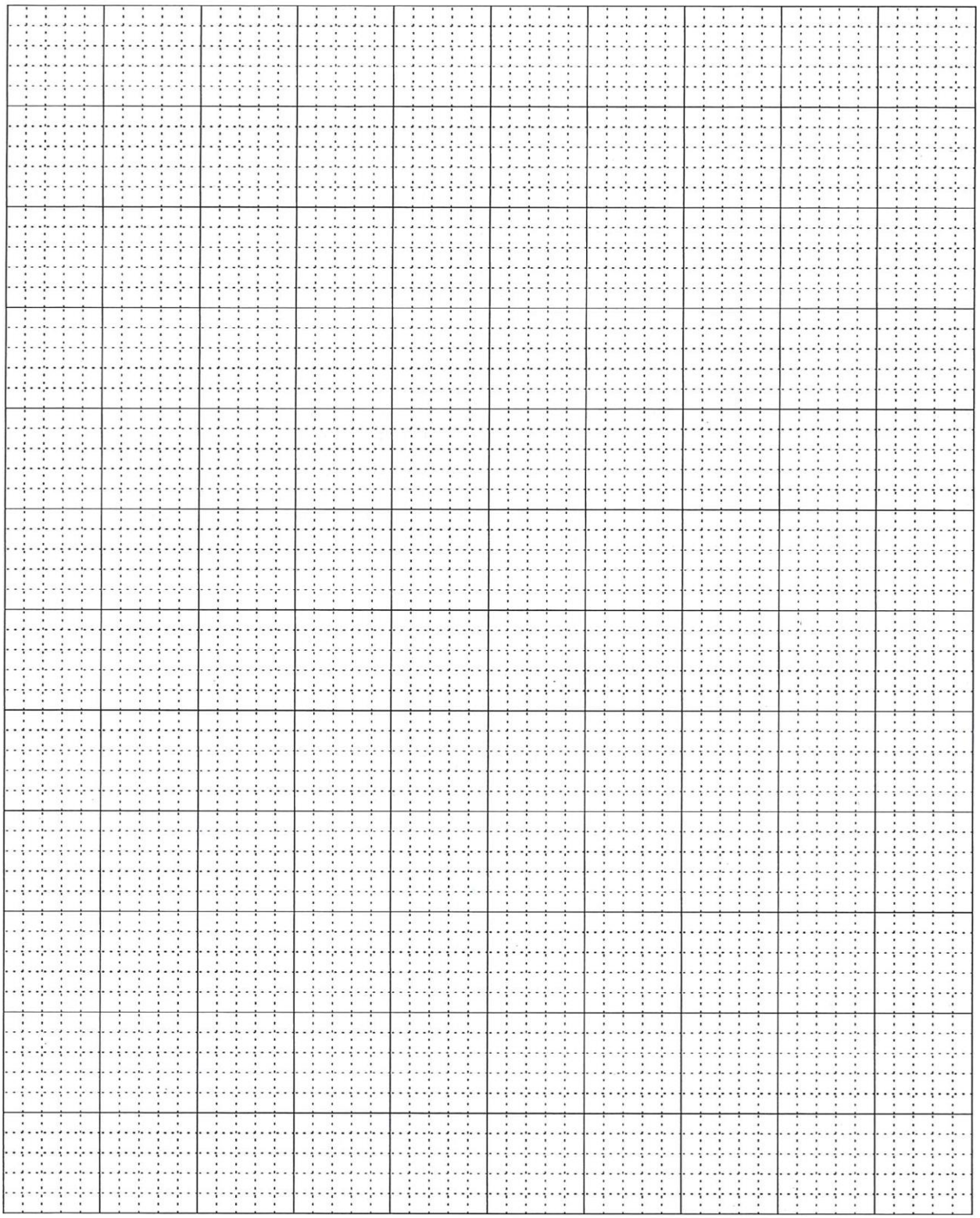
Data Sheet

Data set: *Version 1 - Branch*

N	t1	t2	t3	t4	t5	μ
100						
150						
200						
250						

Data set: *Version 2 – Conditional move*

N	t1	t2	t3	t4	t5	μ
100						
150						
200						
250						



Your conclusion: