The first goal of today's lab is to feel comfortable navigating the Gradescope submission and to get comfortable with debugging tools by fixing buggy programs that reinforce concepts from class.

Getting Started Start by getting the files from the course page in your development environment (`wget https://cs.pomona.edu/classes/cs181ca/labs/03.zip; unzip lab03`).

1 Computing Functional Simulator Scaling Overhead

The goal of today's lab is to develop a tool that analyzes a binary file and estimates how long it will run on a processor with a particular data path. From this, it measures how long it actually took the program to run and produces a guess as to how accurate our model is for various programs. When you're done, try it out for yourself and see what kinds of programs seem to be predicted more or less accurately for different configurations! If you can do so consistently, you can reverse engineered your computer's processor components!

Repository Overview In the repository, you will find a base file called `buggy.cc` that is the main file we will be working with in this lab. In addition, the repository comes with a `Makefile` that will help with the compilation of the program. We can build the program by calling `make buggy`. There are also three subdirectories in the repository: ① `bin` is the directory where the `buggy` executable will be written to after compilation. ② `configs` is a directory of sample configurations of a processor data path; configurations are CSV (comma-separated variable) files where components are listed alongside their latency to execute separated by a comma. ③ `test-progs` is a subdirectory where there are a series of tests that we can use to test our emulator. For each test, there is a simple program and a `Makefile`. Upon compiling `make all` from the subdirectory for that test, a binary executable and assembly program are produced (notice the `-S` flag).

1.1 `gdb` Warm Up

Let's start by trying to compile and run our `buggy` program. Use the command `./bin/buggy` to do so (do you think it will work?). Ah shoot, it didn't run. Without closely examining the source, we could easily find ourselves stuck, so it's natural to want to use the debugger to help figure out what's going on!

To launch `gdb`, we can run `gdb --args ./bin/buggy`[1] to launch the debugging tool. If we want to run the command through the debugger, then we will start running the program by calling `run` (or `r` for short). Fortunately, our debugger stops on the crash point! Let's examine the state.

If we look at the output, you'll notice that our debugger is currently stopped in a C++ system file. Not that helpful! If we want to see the current call stack, we can run `backtrace` or `bt` in our debug console to see the whole call stack. Each frame in the stack (i.e., function call) is numbered, and we can move to the appropriate calling function by calling `frame <n>` or `f <n>`. As a rule of thumb, the error is 99.999% likely to be in something that you've written (we all do it!) so let's go to the lowest frame number that refers to our source.

Once we've gotten to this point in the debugging process, let's examine the state. We have all sorts of variables that we can check, and fortunately we can see what is likely impacting the

[1] If you are on a Mac, you will need to run `lldb` instead (see the course debugging resource for equivalent commands).

function that we're calling. What are in the contents of these variables? You can examine their state by calling `print <var name>` or `p <var name>` for whichever variable you would like to look at.

Do you see the bug?? Take a few minutes to try to inspect the relevant state to figure out what went wrong... (scroll to the next page to find out what went wrong).

We didn't pass the appropriate input to our `buggy` executable! As a result, our file never opened and we could not parse the inputs to get the cycle time. Let's go ahead and fix this by running `./bin/buggy configs/config1 test-progs/test1/test1`. (Did it work??) Of course it didn't work... it isn't called `buggy.cc` for nothing! What is the error that you see this time?

1.2 Stepping Through Programs

It seems that the error happens in a function that is called by our main function this time around. This means that our debugger is jumping all over the place, which can be difficult to keep track of! To make it easier to see, you may want to use a helper `gdb` tool called `layout`. If you run `layout split` from inside the debugger, you will see that pane gets split. The top pane will now show the code source and where the debugger is in its execution!

Rather than running from the beginning, let's just start our program at the beginning. Launch the debugger (`gdb --args ./bin/buggy configs/config1 test-progs/test1/test1`). Rather than saying `run`, we will instead use `start`. This sets a temporary breakpoint in the first line of `main`, calls run, and stops at the temporary breakpoint. From here, let's walk through the execution. We can go to the next line of code by running `next` or `n`. If we want to step into a function rather than step over it, we can call `step` or `s`.

When you get to the `clockSpeed` function, `step` into it. We can continue the execution from here. You may find it helpful to continue printing out state as you step to see what else is going on in the program. Eventually, we will get to the point where the error happens. Ugh... What happened? Can you see what the error was at this point in the execution? (go to the next page to see the error).

It looks like we were trying to parse the header line of the CSV file! We don't really want to change all possible input CSV files, so let's fix our source to account for this format. Do you have any ideas of what to do? Once you come up with a fix, recompile your code (`make buggy`) and repeat this process until you are confident that you can produce the correct calculation of the timing!

There's more! Keep going on your own (this document will be updated when Prof. Thomas gets time to finish it).