**CPSC 261: Lab 1**

**Goal**

The goals of this lab are to:

* set up the environment that you will use for the rest of the labs in this course,
* start to become become familiar with the basic tools that we'll be using,
* transfer your previous experience with assembly language programming to a new architecture, and
* explore the correspondence between expressions and statements in a high-level language (HLL) like C and instructions in assembly language.

**Tools**

The primary tools that you'll be using in this course are:

* An editor for programming.

My personal favourite is Gnu emacs, probably because I've been using it and its ancestors for 30 years. It is powerful and programmable, and has some aspects of Integrated Programming Environments such as eclipse and Visual Studio, while retaining the simplicity of mostly editing simple text files.

There are a lot of other editors available to you, including vi (or vim), and gedit. You can use any editor that is comfortable for you.

You can also use an integrated development environment. The most popular one is eclipse. It is very good but has a rather steep learning curve and some things that we will want to do in this course are rather awkward (generating assembly source from C source, for example).

* The C compiler - gcc.

Use:

gcc --help

to get a description of the arguments that control gcc's behaviour.

* The source code control system, Subversion - svn.

Use:

svn --help

to get a description of the sub-commands and arguments that control svn's behaviour. One of the sub-commands is

help

which can provide detail on the other sub-commands, as in

svn help ci

or

svn help help

We'll be describing the most used svn sub-commands in a little bit.

* The compilation driver - make. Make is a tool that orchestrates the steps necessary to compile a program. This functionality is built into eclipse and VisualC++. We will be using the make tool to do this. Use:

man make

or

make --help

to see the arguments that control make's behaviour. We will be providing a control file (known as a Makefile) for this initial lab.

* The debugger - gdb. Gdb is a powerful debugger that can be used to debug programs written in a variety of languages, including assembly language and C, which are of particular interest to us in this course.

Use:

gdb --help

to get a description of the arguments that control gdb behaviour. From within gdb, you can use the

help

command to get more information about the commands that gdb understands.

**What do you need to do?**

* First, if necessary, activate your Computer Science account. If you were in a Computer Science course last term your account may still be active. Use

getacct

to activate your account. The documentation for getacct is at: <https://www.cs.ubc.ca/getacct>.

* Connect to the Linux servers. The machines that you are using in lab can connect to either Linux or Windows. For this course you want Linux. Your TA will describe this process.
* Use either the command line tools (in a Terminal window) or the FileManager to create a directory for your work in CPSC 261. If you are using the command line tools, then after starting a Terminal window type:
* mkdir cpsc261

This creates a directory in your home directory named cpsc261. You should do all of your work for the course in this directory.

Then type:

cd cpsc261

This changes your working directory to the cpsc261 directory that you just created. Further commands that you type will operate in this directory.

* Use subversion to checkout the files that we are providing for your first lab. You do that by executing a command rather like this one:

svn checkout https://kunghit.ugrad.cs.ubc.ca:8261/repos/2013w2/users/L2X/YYYY/lab1

Except that you need to replace "L2X" with your actual lab section, which will be one of L2A, L2B, L2C, or L2D, and replace YYYY with your undergraduate login id, which has the form: letter-digit-letter-digit. If I were in lab L2A and my login id was "r2d2", then the command that I'd type would be:

svn checkout https://kunghit.ugrad.cs.ubc.ca:8261/repos/2013w2/users/L2A/r2d2/lab1

When you type this command, you are likely to see an error like this:

Error validating server certificate for 'https://kunghit.ugrad.cs.ubc.ca:8261':

...

(R)eject, accept (t)emporarily or accept (p)ermanently?

You want to accept this certificate permanently (or it will continue to ask you this question every time you use svn), so type "p" followed by "return".

Subversion will then ask you to authenticate yourself to the svn server, something like this:

Authentication realm: UBC CPSC 261

Password for 'r2d2':

In response you type your password.

Svn then creates a directory named "lab1" that contains the files that we have provided for you.

You should do another cd command to change your working directory to the lab1 subdirectory that svn has just created for you:

cd lab1

* Find an editor that you can use. The TA will describe some of the editors that you can select. If you are familiar with eclipse, you can use it as an editor as well.
* Examine the file: "add.s" in the lab1 directory that svn created for you. Make sure that you understand how the function add works.

Also look at the testadd.c file which contains a program that calls the add function defined in the add.s file and ensures that it works correctly.

You are required to add comments (comments in assembly go from the # character to the end of the line) to each line explaining what the line does. One of the main things that you'll need to figure out is which registers are used for what: which registers contain input arguments, which is the stack pointer, which register is the frame pointer, and which register contains the result of the function.

One good way to make sure that you understand what instructions do is to execute them one at a time in the debugger, looking at the registers as you do so. Before you can run the program, you will need to compile it: make can do that for you. Run make like:

make testadd

You should see something like this:

> make testadd

cc -g -c -o testadd.o testadd.c

gcc -g -c -o add.o add.s

cc testadd.o add.o -o testadd

You can run the resulting program by typing:

./testadd

The gdb debugger will let you run the program and examine how it executes; run gdb as:

gdb testadd

Gdb's prompt is

(gdb)

Gdb allows the use of unambiguous prefixes of any of its commands or arguments, so you can often type much shorter (but much less intuitive) commands once you get used to it. The shorter versions are shown in parentheses below.

Consider using the following sequence of gdb commands:

break add (or b add)

run

info registers (or i r)

up

down

next (or n)

info registers

next

info registers

next

next

continue (or c)

After typing each of these commands read what gdb displays and make sure that you know what is going on and what gdb is telling you about the execution of your program. Remember that within gdb, the help command can be used to explain what a command does. You will probably want to ask about what

up

and

down

do, as they are not completely intuitive.

* Now that you are an experienced x86\_64 assembly language programmer, take a look at sum.s. Again there is a test C program in testsum.c

If you

make testsum

and then run the resulting program

./testsum

you will see that it doesn't work all of the time. It does, however, work some of the time. Look at testsum.c to see how many tests it contains and look at the result of running testsum to see which tests it passes.

Now:

* + document each line of the function sum
  + figure out what the bug is, and
  + fix it.
* Write a couple of assembly language routines on your own. There are two more test programs: testmax and testfib that test functions max and fib, which don't exist.
  + max is supposed to return the maximum of its two 64 bit integer (long) arguments
  + fib is supposed to return the nth fibonacci number (given that n is the argument to the function). The fibonacci numbers are defined by
  + fib(1) == 1
  + fib(2) == 1

fib(n) == fib(n-1) + fib(n-2) for n > 2

* These functions are to be written in assembly language. You are not to write then in C and then compile them to assembly language, because that defeats the purpose of getting familiar with the x86\_64 instruction set.
* Write your max function in a file named max.s and then compile the test program with
* make testmax
* and run it and make sure it works. Or debug it until it does.
* Write your fib function in a file named fib.s and then compile the test program with
* make testfib
* and run it and make sure it works. Or debug it until it does.

**Provided Materials**

Your lab1 directory should contain the following files:

* add.s
* testadd.c
* sum.s
* testsum.c
* testmax.c
* testfib.c
* Makefile