ECE 3210 Laboratory 1: Develop an Assembly Program

Fall 2017

# Objective

To become familiar with the development system’s software: screen editor, assembler, linker, and debugger. After finishing this experiment, you should be able to do the following:

1. Able to remote login your Raspberry Pi as the workstation.
2. Use a text editor to create an assembly source code (.s) for a future executable program in Linux.
3. Use an assembler to create an object code file (.o)
4. Use a linker to generate an executable file.
5. Understand the general procedure to develop and debug an assembly language program.

# Background

## Assembler

The GNU Assembler, commonly known as **gas** or simply **as**, is the assembler developed by the GNU Project and sponsored by the Free Software Foundation. It is the default back-end of gcc and it will be the assembler used in this class.

To assembly a program (assuming the file name is lab1.s), one should type the command line:

$ as *−*o lab 1 . o lab 1 . s

where the “.s” is the source file and the “.o” is the output of the assembly, aka the object file. An object file is a file containing the machine code generate by the assembler and includes the name of variables, their locations and other pertinent information for the linker.

We need to generate the debugging information for each assembler source line to enable the debugger functionality. To do so, we type the command line:

$ as *−*g *−*o lab 1 . o lab 1 . s

## Linker

The GNU linker, or GNU **ld**, is the GNU Project’s implementation of the Unix command **ld**. The ld command creates an executable file (or a library) from one or more object files – from the compilation (gcc) or the assembly (as) of one or more source files.

To link a single object file into an executable, one should type the command line:

$ ld *−*o lab 1 lab 1 . o

At this point, the linker will take as input the “.o” file and generate an executable given by the “-o” option. Next, to execute the program, we type the command line:

$ . / lab 1

The entry point of an assembly source program is usually referred to as “ start”. Often, we will need to link object files from gcc with object files from the assembler. In that case, we will need to change the entry point to “main”. This can be accomplished by the following command line:

$ ld *−*e main *−*o lab 1 lab 1 . o

As before, the linker will generate a executable file named “lab1”.

## Debugger

The GNU Project debugger (**gdb**) allows you to execute, trace, inspect variables, and many other things that happen inside your program while it executes. This is very useful to understand, for example, what the program was doing at the very moment that it crashed. The main things that *gdb* can do to help you catch bugs in the act are:

* + 1. Start your program, specifying anything that might affect its behavior.
    2. Make your program stop on specified conditions.
    3. Examine what has happened when your program stops.
    4. Change things in your program, so you can experiment with correcting the effects of an identified bug so you can go on to learn about other bugs.

## GDB Graphical Interface

GNU DDD is a graphical front-end for command-line **gdb** debuggers.

# Prelab

No pre-lab report is required for Lab 1

# Laboratory

## Part 1: “Hello World” Example Breakdown

We first start a “Hello World!” example. In this section you will create an assembly program that calls the “printf” function from the C library in order to print. However, doing so in assembly allows you to see more lower level information compared to C or C++.

1. Create a .s file under the working directory. You can use Geany (graphical text editor) or vim, whichever you are comfortable with. You can find an example program in Appendix **lab1p1.s**.
2. Once you have created the file and saved it in the working directory, we will use **gcc** to assemble and link the files. Type the following into the command line:

$ gcc *−*o lab 1 p 1 lab 1 p 1 . s

1. Execute the program by typing ./lab1p1 into the command line. You should see the output “Hello World!” on your screen.

Next, we will break down the “printf” function from the C library to understand how it is imple- mented in Assembly language.

1. Create a second .s file under the working directory following the example program in Appendix

### lab1p2.s

1. Assemble and link the file with the instructions from Background section (**as** and **ld**) to generate the executable.
2. Execute the program. You should see similar output as the first example.

## Part 2: Debug an Assembly Program

In C programming, you usually print out the value of each variable to make sure your program is functioning properly. In assembly, the **registers** take the position as “variables” and share the same importance. However, from Part 1 you can tell it is quite tedious if you want to perform the “printf” function each time for just checking register values. This is where the debugger comes in handy.

First, we will start debugging the executable file **lab1p2** with the command line tool **gdb**.

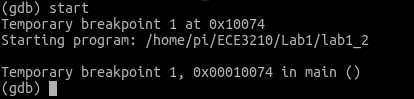
1. To start the debugger, in command line:

$ gdb lab 1 p 2

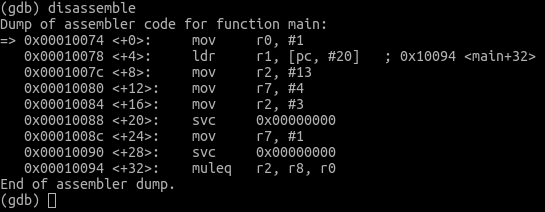
Once the debugger is running properly, you should see the following message on your screen.



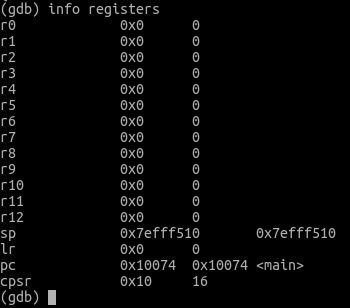
1. Start the debug procedure by the command “**start**”.



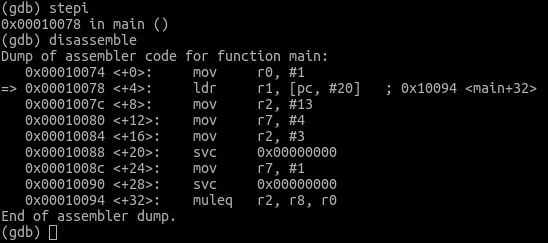
1. Next, we use the command “**disassemble**”. There is an arrow **=¿** pointing the instruction we are going to run (it has not been run yet).



1. Before running the program , let’s inspect some register values by using the “**info**” command.



1. To run the instruction one by one, we use the “**stepi**” command. Then we can follow with another “**disassemble**” to see what has happened.



1. Repeat Steps 3˜5 and fill the table with register values after each instruction has been run.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | R0 | R1 | R2 | R7 |
| MOV R0,#1 |  |  |  |  |
| LDR R1, =message |  |  |  |  |
| MOV R2, =length |  |  |  |  |
| MOV R7, #4 |  |  |  |  |
| MOV R2, #3 |  |  |  |  |
| SWI 0 |  |  |  |  |
| MOV R7, #1 |  |  |  |  |
| SWI 0 |  |  |  |  |

Table 1: Instruction trace table.

### Lab deliverable 1

Include the above Table 1 above in your lab report results section.

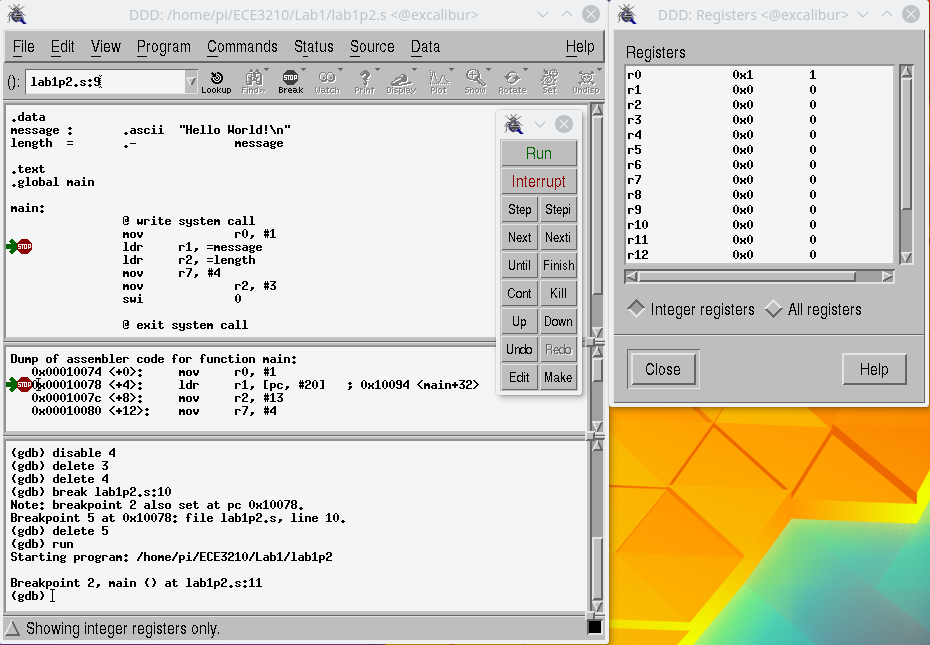


Figure 1: The proper setting for the DDD debugger.

## Part 3: Use the graphical interface DDD to debug the program

A graphical interface can provide a lot convenience for the process of debugging. **Part 2** is just an introduction of what the debugger actually is doing. In this section, we will use the DDD to debug and fix the **lab1p2.s** program. First, inside a terminal, you type the command:

$ ddd lab 1 p 2

You should see a similar interface as Figure **??**. Next, under the “**View**” tab open the “**Machine Code Table**” . And under the “**Status**” tab, click “**Registers**” . Now your interface should look similar to Figure 1. You can set a break point by left clicking the blank area (left side as shown in Figure 1) next to each instructions.

Once the break point has been set, you can click the “**Run**” button to start debugging and the “**stepi**” button to trace each instructions as well. The value of each register should be displayed in the Registers status window on the right side of Figure 1.

### Lab deliverable 2

After fixing the bug inside **lab1p2.s**, make changes of the program so it can print the following message. Submit the program with your lab report.

ECE 3210

*<*Your Name*>* He l lo World !

# Report

Please follow the template provided from the syllabus. Include the answer for the following questions in your Conclusion section.

1. What does **.data**, **.text**, and **.global** mean?
2. What does the instruction **LDR** do?

# Appendix

## lab1p1.s

. data

message : . a s c i i ” He l lo World ! *\* n”

. te xt

. g lo b a l main main :

push *{* ip , l r *}*

ld r r0 , =message @ Load the s t a r t i n g address o f the message bl p r i n t f @ Call the p r i n t f f u n c t io n

mov r0 , #0 @ Return 0 . pop *{* ip , pc *}*

## lab1p2.s

. data

message : . a s c i i ” He l lo World ! *\* n” le ngth = .*−* message

. te xt

. g lo b a l main main :

@ w r i te s y s c a l l

mov r0 , #1 @ For stdout

ld r r1 , =message @ b u f f e r i s loaded with message

ld r r2 , =le ngth @ count i s the le ngth o f message mov r7 , #4 @ w r i te i s s y s c a l l 4

mov r2 , #3

swi 0 @ in t e r r u p t @ e x i t s y s c a l l

mov r7 , #1 swi 0