Binary Lab

Description

The main aim of this lab is to understand provided assembly code and generate C code that corresponds to that assembly. We are giving you 5 binary files (executable): f1, f2, f3, f4, and f5. Each one of those files represents the binary code of a main function and a C function distributed among those files. That is, f1, for example, corresponds to main function and f1() function. We also provide you with 5 C files: f1.c, f2.c, f3.c, f4.c and f5.c. Each one contains the empty functions f1(), f2(), f3(), f4(), and f5(). Your job is to read the object files (more on this shortly), understand the assembly of each function, and write the corresponding C code of each of the 5 functions in the 5 C files.
Note: The fact that you have one function in each of the provided empty C files does not mean that this function does not call another function. For example, when you examine the code for f1() you may find that f1() calls another function b1(). In this case, you need to include b1() in the file f1.c too. If you find that f1() does not call any other functions, then your file f1.c will contain f1() code only.

Steps:

1. Download the file **lab2.tar**
2. Open a terminal on the virtual machine
3. Go to the directory that contains the file you downloaded. This can be *Download* directory.
4. Type: **tar -xvf lab2.tar**
5. Type: **cd lab2**
6. Inside that directory, lab2, you will find several things:
   - Several *.o files: you do not need to do anything with them.
   - A file called Makefile: This will be used in compiling your code as we will show you shortly.
   - f1.c, f2.c, …, f5.c: These files contain empty function. You need to write those functions. This is what you have to submit at the end.
   - A sub-subdirectory called executables: This directory contains the executable files f1, f2, f3, f4, and f5. Each one of these files serves two purposes:
     a) By executing each file, you get to know what are the inputs and outputs of the programs. So, you can compare your implementation with this them for correctness. To execute f1, for example, just type: **./f1**
     b) You will disassemble each executable, as we will show you in this report, to figure out what each function does.

How to work on this lab?

Your main source of help is a tool called objdump. Suppose you are in the directory that contains the executable files. When you type for example:

objdump - d f5

After executing the above command, objdump will print on the screen the assembly code of all the executable f5. This includes the assembly of the main function, f5() function, statically linked libraries, jump table (if any), any other functions called by main or f5 (if any), … .
You do not need to read all that. You can look for the label “f5:” and read from that point.

(Note: The output of objdump may be big and the screen can scroll down a lot. There is a small trick to save the output on a file by typing: objdump -d f5 > somefile.txt
The output will be written into that file somefile.txt, or any name you pick, and you can read it with any editor as it is a text file).

After you read the code and understand it, you are ready now to open the file f5.c and start writing the C code for the function f5, including the arguments of the function, the return value, as well as any functions called by f5.

Once done editing f5.c and saving it, you generate the new executable of f5 by typing:

make f5

If there are no compilation or linking errors, you will get an output file called f5 that you can execute by typing ./f5
Compare this new executable with the original executable to ensure you have implemented it correctly.

You need to read the assembly code, understand it, and write the code that does the same thing. It does not need to generate exactly the same assembly when compiled, because this is almost impossible, but do the same function. That is, if I compile and link your submitted files, I must get an executable that does exactly the same actions (inputs and outputs) as the original.

We will not try to break your code by testing with corner cases or wrong inputs.

Grading

There are 5 functions to implement. Each one is worth 20 points, for a total of 100 points.

Submission

Once done with everything, do the following:

1. Go to the directory lab2 (if you are not already there).
2. Type: tar -cvf lastname.firstname.tar f*.c (where lastname is your last name and firstname is your first name).
3. A file called: lastname.firstname.tar is generated. Upload that file to NYU classes.

Important Notes

Some instructions that we did not cover in class but you may find in the object files (not included in the exam though):

- **repz**: This is used due to some strange behavior with old AMD K8 regarding its branch prediction. To make long story short, neglect it! So if you see, for example, repz retq assume it is retq only.
- **nopl**: this is a no-operation instruction. That is, do nothing instruction. It can take argument but does nothing with it. It is mainly used as a delay instruction waiting for an event to happen, such as incrementing rip register (more in-depth explanation will take us a bit deeper into hardware). Also, it is used as "padding" in the code to make following instruction start at specific address.

Do not delete the files main*.o because if you do you will not be able to compile your code. If you accidently deleted them, you may want to re-download the file lab2.tar again from the web. but save any work you may have done in a different directory first.

Enjoy!