Hyun Suk Ryoo

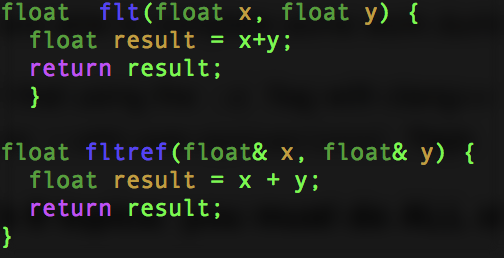
CS2150

10/31/17

Inlab8.pdf

**Parameter Testing**

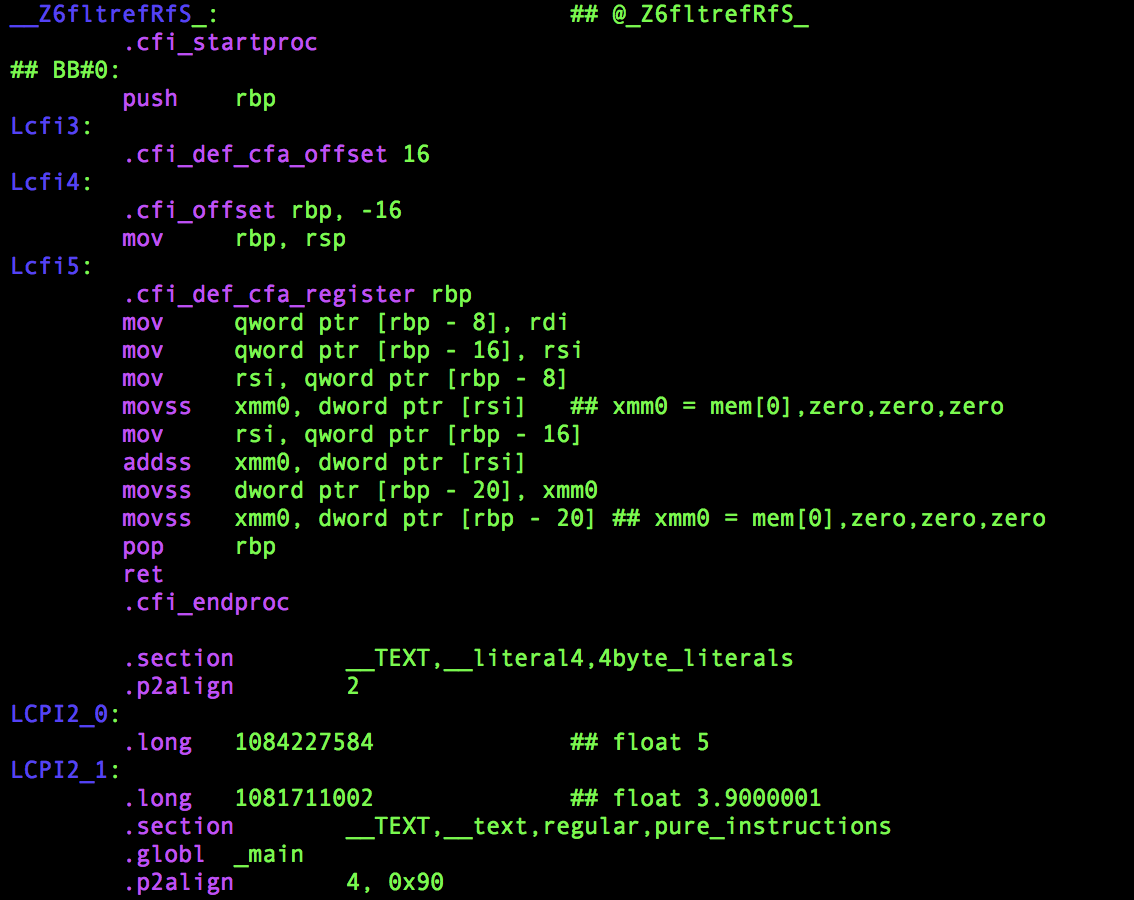
The purpose of this in lab was to explain the assembly code generated by the compiler for a simple function with variety of parameters. The major parameters that were looked at was ints, chars, pointers, floats, and objects and the difference of the assembly when the parameters were passed by value or if it was passed by reference.

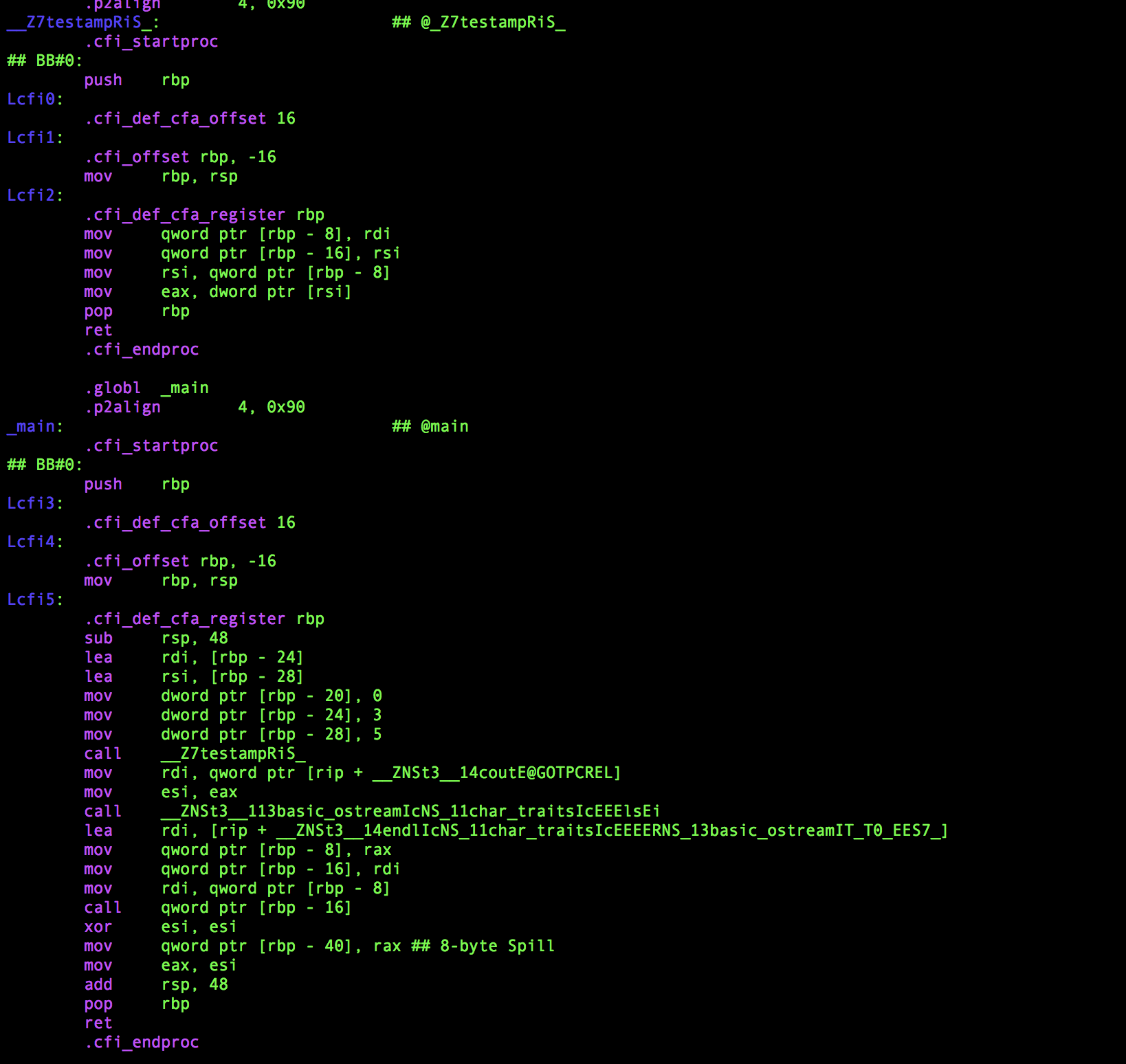
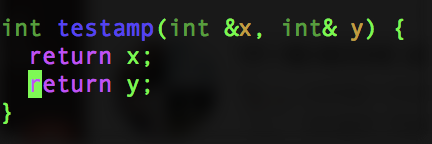
In order to find out the differences in the code many small functions were made in a cpp file. Each function would do simple maneuvers like add or maybe sometimes simply return to see how passing by value is different than passing by reference.

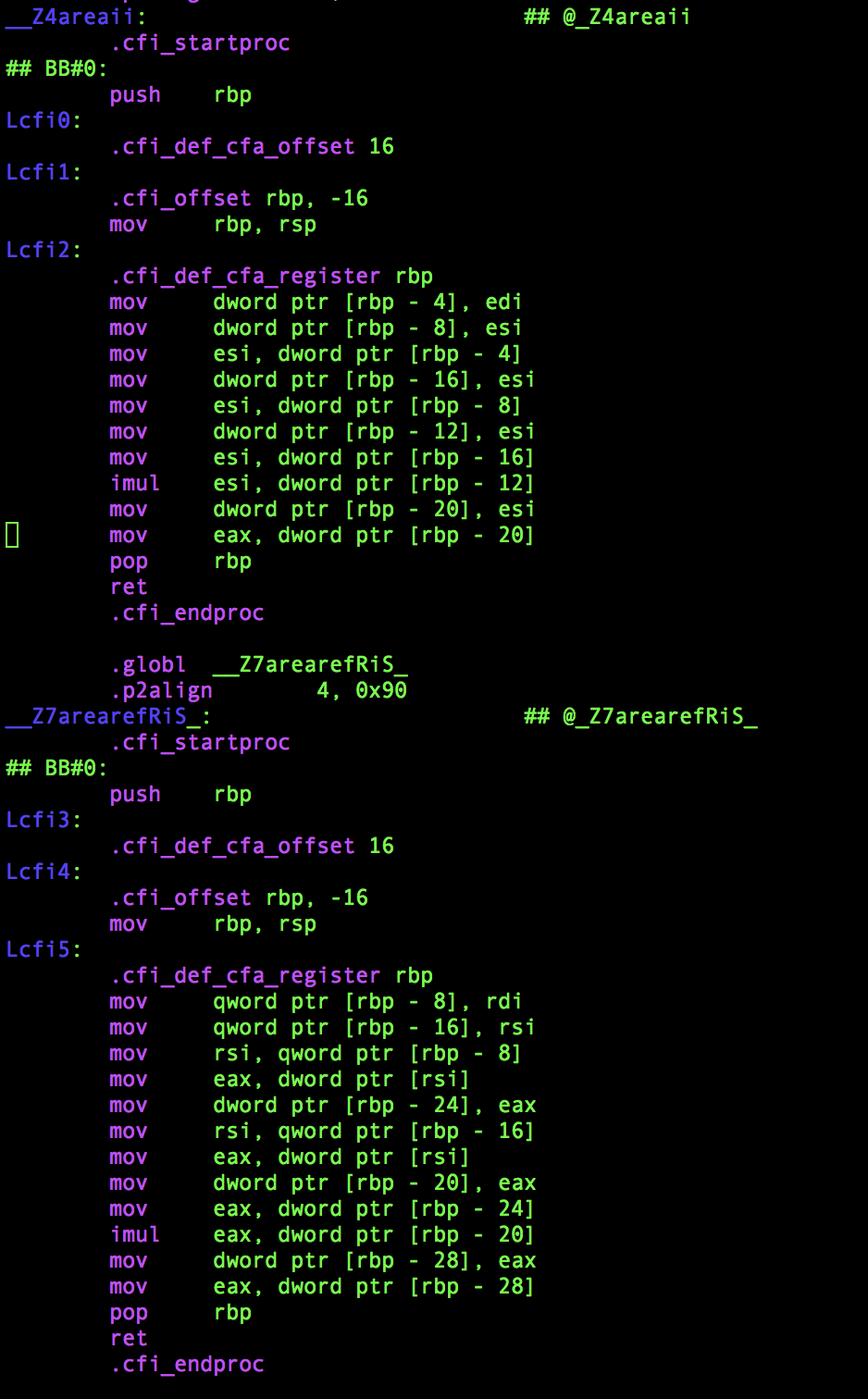
These methods would be called in the main method with differing values as parameters to execute. The assembly code for all the results and findings found are from the compiler assembly code given by the –S -mllvm --x86-asm-syntax=intel on a 2017 MacBook Pro 2.3Ghz i5 laptop.

When looking at ints, floats, chars the assembly code for the caller looked fairly similar. It would save the stack pointer before pushing the data into the stack and then call the callee functions. However, when the parameters were passed by reference the code looked a little different. When looking at the example for the char type example with both passing by value and reference, the difference was fairly clear.

Looking at figure directly below, there was 1 more mov command in the callee. This is necessary for accessing the value of the address of the parameter that was passed in.

This is logical and makes sense because the caller would need to work with the value that is being passed in not the address. Because the dereferencing happens in the callee portion it makes sense to have another mov command to retrieve the value of the address that was being passed in.

This extra command isn’t needed in the callee with the value being passed in because the caller could just access the value that is being passed in directly rather than having to dereference. This is why in the assembly code that has to deal with parameters passed by reference has the lea command which loads the effective address of the second register into the first register. Samples of the lea command are shown in the figure below along with the corresponding cpp method.

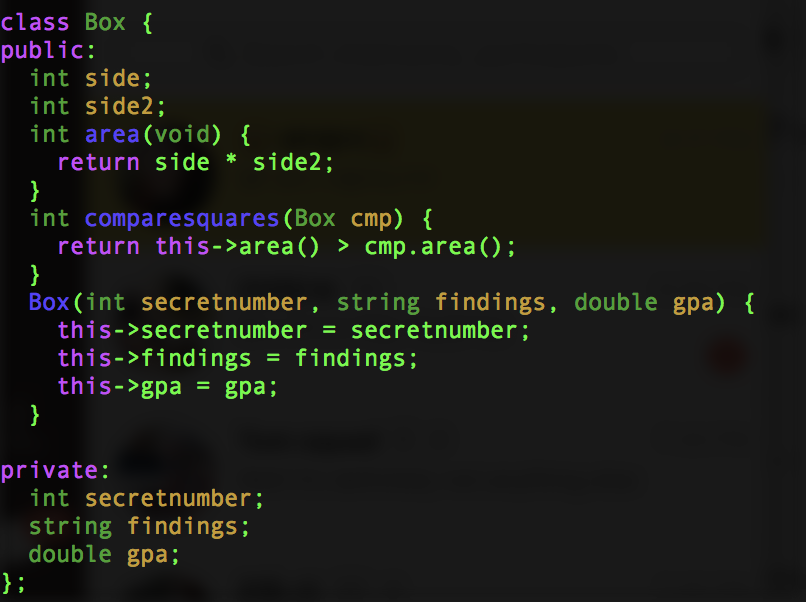


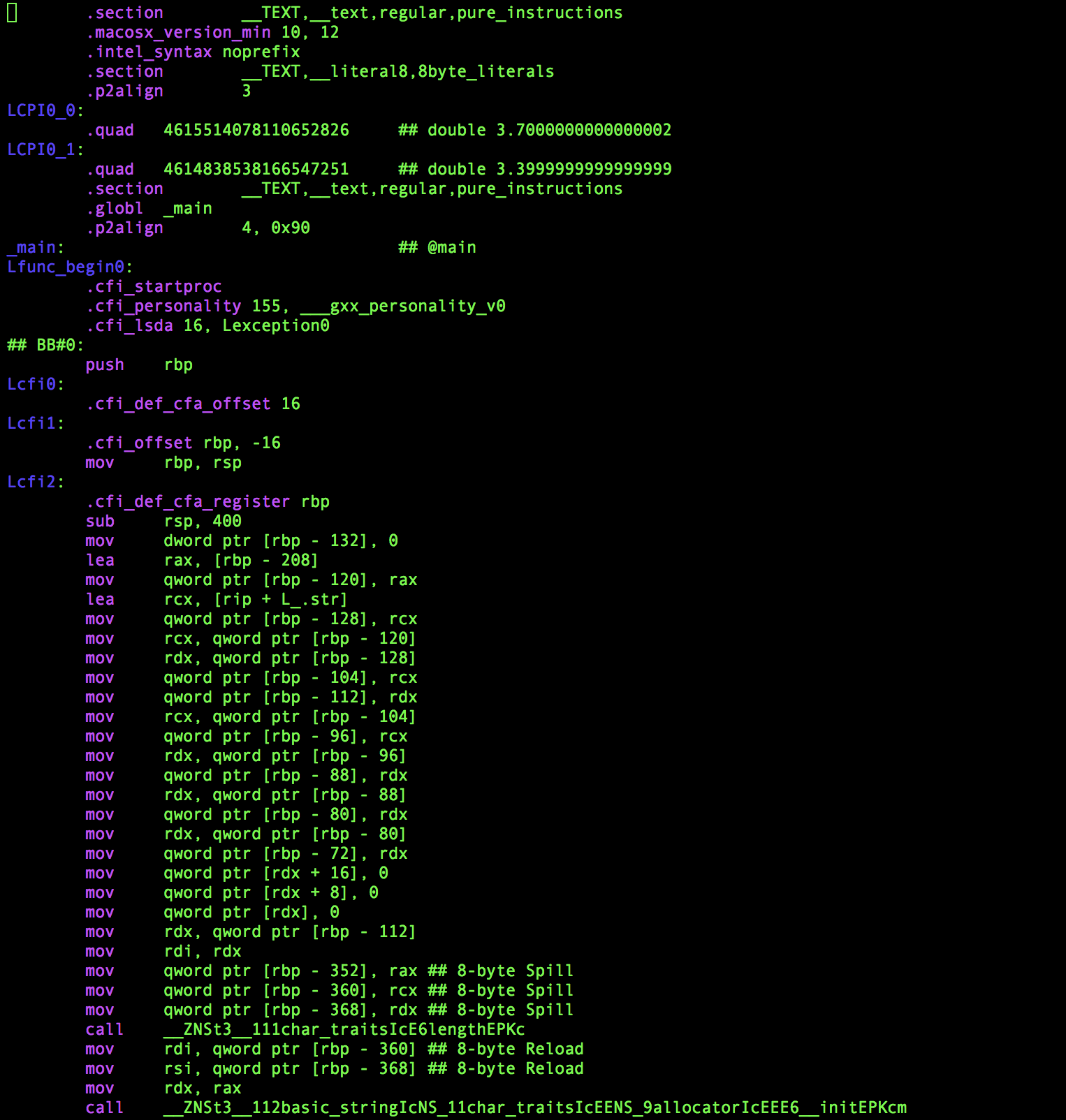
Looking at the box class, a class where there existed a value for the side length of the box and a value for the other side length of the box, I made to test the difference between passing by reference and values, it is also clear there is a difference in the dword ptr. Dword ptr is command that is used for size directives and for all the cases including ints, floats, chars, and objects when passing by referencing it has a higher dword ptr [rbp-#] than it does when it is passed by the value. The following figure shows how the number following rbp is different in the assembly code.

The dword ptr is used in this case to store the values of the object that was created to the stack and registers to further carry out the function such as finding the area of the object square that was created.

It seems like the process for both the pass by reference and pass by values are pretty similar except for more commands in the callee and more dword ptr in the caller method. There was definitely increase in the stack pointer for the passing by references as shown by the difference as explained by the figure directly above. It just seemed that it took extra steps to work with parameters passed by references because it takes more time and code to access the value inside the address that was provided while pass by value can directly work with the value.

**Objects**

 To determine how objects are dealt with in assembly code I generated my own object class called Box. It has 2 public fields, 2 public functions, 3 private fields, and a constructor for the Box object that initializes the private fields.

There wasn’t really a big change in how private and public variables were declared in the assembly code generated. However, there was a slight change in accessing the private field. It had to call a different function in order to deal with the value for the private field while for the public field such action wasn’t needed. In terms of declaring the class I thought all the declarations would happen before the main method assembly code, but in actually the assembly code started straight in the main method. This makes sense in a way because nothing in the Box declaration would taken up immediate memory. The following assembly code is generated when the previously shown c++ code is clang++ -S –mllvm --x86-asm-syntax=intel.

In this code the first line of code is decreasing the rsp by 400, which is to make room for the variables in the Box class. Every subsequent dword command is used for storing the object and for storing the temporary parameters that are needed in the class, which explains the 132, 120, 128 ..

When looking at the following code it is clear to see that the “this” pointer creates an offset of bytes to access the Box class in the memory stored. The “this” pointer isn’t actually directly stored on the stack but rather is used as a memory address. The offset is for leaving space for storing the memory address of the “this pointer”, which was shown in the comparesquares assembly code.

Sources:

<http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>

<http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>