Programming Assignment 2

Shreya Bajpai

Rutgers, The State University of New Jersey

Computer Architecture (01:198:211) - Summer 2017

Section F6 - Professor Brian M. Russell

Overview

In the mystery directory, you will find (1) an x86 assembly code, mystery.s, along with comments on what each line does (2) mystery.c, the translated mystery.s and (3) mystery.h, the header file for mystery.c (4) and mystery.unoptimized.s.

Mystery.s & Converting it to Mystery.c

Mystery.s was tricky to decode if you were to go off on your memory of how the stack frame works. I found a helpful source by CMU's computer science department, which has a very cohesive outline of x86 assembly. I went line by line in mystery.s and decoded what each execution achieved.

It was difficult to understand what **dothething (**referred to as **fibb()** in mystery.c) was doing, until I referred back to the instructions for assembly.pdf on Sakai. It was then I realized that the example input into mystery (41) outputted its fibonacci sequence. As I went line by line in dothething, I was looking for clues to see when and where the recursive fibbonacci call was and I found it on lines 58-70, where we see a variable being set to add(fibb(input - 2), fibb(input - 1))and this was what helped me unravel the rest of dothething. It was difficult to realize what .comm was doing. I looked up Oracle's assembly guide and it noted that .comm has three parameters: name_of_var, size, alignment. It was surprising that we would have an int array of 1600 items, but then I realized that 32 was the alignment so I divided 32 by four since we are working with ints, and then I divided 1600/8 and got 200 and assigned int num[200].Another indication of the size was in our **main()** where we have a check to see if the number is less than or equal to 199. In this loop, we assign all the values in the array to -1 and then calculate the fibonacci number and print the value.

Note:

Mystery.c only outputs the correct values until fibb(46). After fibb(46), the bits spill over and the overflow flag is activated and our values are incorrect and negative.

Add was easy to figure out once I realized two inputs were being passed.

Optimization Flag

In the instructions for assembly, it asks us to do the following for mystery.c:

```
$ gcc -S mystery.c
$ mv mystery.s mystery.unoptimized.s
$ gcc -S -O mystery.c
$ diff --side-by-side mystery.unoptimized.s mystery.s
```

The last line apparently displays the differences between the optimized and unoptimized .s files.

I noticed that the optimized assembly file utilizes the **scaling factors** and **operand arithmetic** to avoid calling unnecessary add, mul, div ops. This is possible since the compiler has knowledge of the space that will be required to run each function in mystery, thereby rendering some local variables useless. As a result, calculations can occur in single lines within the other operations.

Due to this optimization, time and space complexity of the program is reduced.