Assembly Programming
Naeem Hossain
November 6, 2016
Computer Architecture

This assignment is designed to give you additional practice in reading and writing Assembly Language programs. As discussed in lecture, unless you are working in increasingly rare areas such as low-level OS development, you are unlikely to be reading and/or writing Assembly Language programs in the remainder of your career. However, we are still requiring you to read and write some here to make sure you understand the computing model underlying your C and Java programs. In addition, being able to read Assembly Language is particularly important because there are times when you need to understand what the compiler is doing to your code.

There are two parts. In the first part, you will write two small functions in Assembly. In the second part, you will be deciphering (that is, write equivalent C code) an Assembly Language program. You will also be asked to compare unoptimized and optimized versions of the code and explained what the compiler did when it optimized the code.

***Important:** On most of the iLab machines that we checked, e.g., several of the Design Pattern machines, including adapter.cs.rutgers.edu, command.cs.rutgers.edu, and factory.cs.rutgers.edu, gcc by default will generate x86 Assembly, which is what we want. Do be careful and check the generated assembly, though, in case some of the machines are 64-bit processors, with gcc configured to generate x86-64 code. If you see Assembly code with registers beginning with %r (e.g., %rbx or %r9), then it is x86-64 and not what you want. Move to another iLab machine.*

Part 1: Writing x86 Assembly

In this part, you will implement a program formula that will print the formula for (1 + x) n . In particular, your program formula should support the following usage interface:

formula <power>

where the argument should be a non-negative integer. Your program should print out the "long" form of (1 + x) n , where n is equal to the argument . Your program should also print out its execution time (in microseconds).

Analysis:

The most challenging part of this assignment involved configurations and proper syntax more than actual logic in coding. This is because I'm used to working in very high-level languages and compilers like Swift and Java. After I understood Assembly and began coding, it was frustrating to work from basically a blank slate with absolutely no

libraries and functions. However, Assembly offered me much more freedom to do what I wanted with memory allocations and stuff. The C code was rather straightforward after I built the nCr and factor functions in Assembly. It was just a matter of checking for errors, calling the functions, and making sure that the formatting as correct. I did have some difficulty trying to get the time correct, as I wrongly implemented the struc gettimeofday().

Big-O:

Space:

The nature of the assignment only requires us to use up to 32-bit processing, restricting the input to $x < 13$.

Time:

Assembly makes the average time range from around 53-80 microseconds, with a few outliers in between. This is very fast because Assembly cuts out a lot of wasted time/space management otherwise lost in translation between a higher level language. Since I had full control over malloc, I was able to optimize the program to be very time efficient, even with space constraints to 32-bits.

```
bash-4.2$ ./formula 13
(1 + x)^13 = ERROR: input size is too large. enter an input <= 12
bash-4.2$ ./formula 12
(1 + x)^12 = 1 + 12*x^1 + 66*x^2 + 220*x^3 + 495*x^4 + 792*x^5 + 924*x^6 + 792*x^7 +
495*x^8 + 220*x^9 + 66*x^10 + 12*x^11 + 1*x^12
 Time Required 66 microseconds
bash-4.2$ ./formula 11
(1 + x)^11 = 1 + 11*x^1 + 55*x^2 + 165*x^3 + 330*x^4 + 462*x^5 + 462*x^6 + 330*x^7 +
165*x^8 + 55*x^9 + 11*x^10 + 1*x^11
 Time Required 70 microseconds
bash-4.2$ ./formula 10
(1 + x)^10 = 1 + 10*x^1 + 45*x^2 + 120*x^3 + 210*x^4 + 252*x^5 + 210*x^6 + 120*x^7 +
45*x^8 + 10*x^9 + 1*x^10
 Time Required 79 microseconds
bash-4.2$ ./formula 9
(1 + x)^9 = 1 + 9*x^1 + 36*x^2 + 84*x^3 + 126*x^4 + 126*x^5 + 84*x^6 + 36*x^7 + 9*x^8 +
1*x^9
 Time Required 62 microseconds
bash-4.2$ ./formula 8
(1 + x)^8 = 1 + 8*x^1 + 28*x^2 + 56*x^3 + 70*x^4 + 56*x^5 + 28*x^6 + 8*x^7 + 1*x^8
 Time Required 61 microseconds
bash-4.2$ ./formula
Segmentation fault
bash-4.2$ ./formula 7
```

(1 + x)^7 = 1 + 7*x^1 + 21*x^2 + 35*x^3 + 35*x^4 + 21*x^5 + 7*x^6 + 1*x^7
Time Required 21 microseconds
bash-4.2$ ./formula 6
(1 + x)^6 = 1 + 6*x^1 + 15*x^2 + 20*x^3 + 15*x^4 + 6*x^5 + 1*x^6
Time Required 59 microseconds
bash-4.2$ ./formula 5
(1 + x)^5 = 1 + 5*x^1 + 10*x^2 + 10*x^3 + 5*x^4 + 1*x^5
Time Required 57 microseconds
bash-4.2$ ./formula 4
(1 + x)^4 = 1 + 4*x^1 + 6*x^2 + 4*x^3 + 1*x^4
Time Required 56 microseconds
bash-4.2$ ./formula 3
(1 + x)^3 = 1 + 3*x^1 + 3*x^2 + 1*x^3
Time Required 57 microseconds
bash-4.2$ ./formula 2
(1 + x)^2 = 1 + 2*x^1 + 1*x^2
Time Required 53 microseconds
bash-4.2$ ./formula 1
(1 + x)^1 = 1 + 1*x^1
Time Required 53 microseconds

USAGE:

./formula -h : displays usage instructions
./formula <positive int> runs the program