CS 211: Computer Architecture, Summer 2015
Programming Assignment 3: Assembly Language Programming

1 Introduction

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, much like you have done in your homework and the Midterm Exam. 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.rutgers.edu, command.rutgers.edu, and factory.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.

2 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 `<power>` 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 `<power>`. Your program should also print out its execution time (in microseconds).

For example:

```
$ ./formula 5
(1 + x)^5  = 1  + 5*x^1 + 10*x^2 + 10*x^3 + 5*x^4 + 1*x^5
Time Required = 50 microsecond
```

```
$ ./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 = 55 microsecond
```

(Hint: You can use the system call `gettimeofday()` to measure the running time of a chunk of code.)

More generally, given the argument n, your code needs to generate:

```
(1 + x)^n = 1 + nC1*x + nC2*x^2+ ... + nCr*x^r + ... + nCn*x^n
```

Your program should also print a usage message if the user runs `formula` with the help flag (`-h`). For example:

```
$ ./formula -h
Usage: formula <positive integer>
```

2.1 nCr Calculation

Hopefully, you remember from one of your Math classes that each of the constant nCr above can be computed using the formula:

$$nCr = \frac{n!}{r!(n-r)!} \tag{1}$$

Your task is to implement this computation in Assembly. In particular, you need to implement two functions in Assembly:

`int nCr(int n, int r)`: This function computes the nCr constant according to Equation (1).

`int Factorial(int n)`: This function computes the factorial of the input (that is, $n!$).

To help you get started, we are providing two files:

`nCr.s`: contains the necessary GAS (Gnu ASsembler) directives so that your Assembly code can be compiled and linked in with your C code (in `format.c`).

`nCr.h`: contains the prototype for the function `nCr()` so that you can compile your C code which calls `nCr()`.

Important: As n becomes large, you will not be able to compute $n!$ and nCr. Both `nCr()` and `Factorial()` *must detect overflow conditions using the processor's condition codes and return 0 to indicate that an error has been encountered.*

3 Part 2: Reading x86 Assembly Code

In this part, you are asked to decipher the Assembly Language program in the attached `mystery.s` file. Specifically, you need to provide a concise description of what the program does and how it does it. You should also implement a C program `mystery` that performs the same task *in the same manner* that the code in the attached file `mystery.s` does.

The provided program takes a single integer as input.

```
$ gcc -o mystery mystery.s
$ ./mystery 41
Value: 165580141
```

Hint: This program performs a well known and easily recognizable computation. However, it includes an optimization to speedup the computation. You need to figure out both the basic functionality as well as the optimization, describe them, and replicate them in your C code.

Another Hint: You are not strictly required to go backward from the `mystery.s` file that we give you. That is, when you start writing your `mystery.c` program, you can compile it to Assembly (`gcc -S`), and compare the generated code against our `mystery.s`. Our `mystery.s` was generated on factory.rutgers.edu so you should be able to generate the exact same code.

Once you have implemented your C program, you should compile it with and without the -O option (optimization). You should then compare the two versions and explain the differences inside the mystery function.

For example:

```
$ 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 command in the above sequence will show you the differences between the two .s files side by side (use a large terminal window). You should look at the differences *inside* the mystery function and explain why the compiler made the changes that it did when optimizing the code.

Collaboration: Please do not share your solution with your classmates as this gives the problem away. You may help each other with understanding Assembly details but not the solution.

4 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa3.tar` that can be extracted using the command:

```
tar xf pa3.tar
```

Your tar file must contain:

- A sub-directory named `formula`. `formula` must contain:
  - `readme.pdf`: this file should describe your design and implementation of the `formula` program. In particular, it should detail your design, any design/implementation challenges that you ran into, and an analysis (e.g., big-O analysis) of the space and time performance of your program.
  - `Makefile`: there should be at least two rules in this `Makefile`:

    `formula` build your `formula` executable.

    `clean` prepare for rebuilding from scratch.
  - source code: all source code files necessary for building `formula`. At a minimum, this should include four files, `formula.h`, `formula.c`, `nCr.s`, and `nCr.h`.

- A sub-directory named `mystery`. `mystery` must contain:
  - `readme.pdf`: this file should describe how you went about figuring out what the mystery program does. It also should describe the changes that the compiler made when optimizing your C code and why you think that the compiler made those changes. (That is, why might the changes make your program run faster?)
  - `Makefile`: there should be at least two rules in your `Makefile`:

    `mystery` build your `mystery` executable.

    `clean` prepare for rebuilding from scratch.
  - source code: all source code files necessary for building `mystery`. At minimum, this should include two files, `mystery.h` and `mystery.c`.

We will compile and test your programs on the iLab machines so you should make sure that your programs compile and run correctly on these machines. You must compile all C code using the gcc compiler with the `-ansi -pedantic -Wall` flags.

5 Grading Guidelines

5.1 Functionality

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build a binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. *In particular, your code should be adept at handling exceptional cases.*

Be careful to follow all instructions. If something doesn't seem right, ask.

5.2 Design

Having said the above about functionality, design is a critical part of any programming exercise. In particular, we expect you to write reasonably efficient code based on reasonably performing algorithms and data structures. More importantly, you need to understand the performance (time & space) implications of the algorithms and data structures you chose to use. Thus, the explanation of your design and analyses in the `readme.pdf` will comprise a non-trivial part of your grade. *Give careful thoughts to your writing of this file, rather than writing whatever comes to your mind in the last few minutes before the assignment is due.*

5.3 Coding Style

Finally, it is important that you write "good" code. Unfortunately, we won't be able to look at your code as closely as we would like to give you good feedback. Nevertheless, *a part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.
- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `i`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.
- Error and warning messages should be printed to `stderr` using fprintf.