CS 211: Computer Architecture, Fall 2011
Programming Assignment 2: Assembly Language Programming
Due: Oct 26, 2011. 11:55pm.

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 write some here to make sure you understand the computing model underlying your fancy C and Java programs. 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. Having some experience of writing assembly codes will help improve your reading skills too.

In this assignment, you will write a small function in Assembly and make changes to existing x86 code.

Important: Many of the iLab machines have 64-bit processors and so run x86-64. You do NOT want to work on these machines for this assignment. We believe that all of the *Cereal* machines, that is, alphabits, trix, frootloops, etc. (machines in Hill 248), have 32-bit processors and are suitable for this assignment.

2 Writing procedures in x86 Assembly Language

You will implement a program `fibonacci` - computing the n-th fibonacci number.

The n-th fibonacci number is defined as follows:
$f(n) = f(n-1) + f(n-2), n \geq 2$. By definition, $f(0) = 0, f(1) = 1$. In particular, your program `fibonacci` should support the following usage interface:

```
fibonacci <n>
```

where the argument `<n>` should be a non-negative (i.e. ≥ 0) integer . Your program should print out the n-th fibonacci number.

For example:

```
$ ./fibonacci 7
$ 13
```

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

```
$ ./fibonacci -h
$ Usage: fibonacci <non-negative integer >
```

You can write all your code in `c` *except for the function that calculates the fibonacci number . These have to be written in assembly* . In particular, you need to implement a functions in Assembly that can be called from your c code:

`int fib(int n):` This function computes the n-th fibonacci number.

To help you get started, we are providing 2 files:

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

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

Important: As n becomes large, you will not be able to compute the final results. Your program *must detect overflow conditions and print out* `overflow` *to indicate that an error has been encountered*. For example, the 100-th Fibonacci number 354224848179261915075 should cause an overflow. In these cases your output should look like:

```
$ ./fibonacci 100
$ overflow
```

Please note that the output is case sensetive, so don't modify it in anyway.

Another important requirement is that in your assembly code, the main loop where you compute the fibonacci number should be implemented using the `loop` instruction.

3 Modifying x86 Assembly Code

You have been given an file `prime.s`. This program is meant to compute the nth prime number. However, several mistakes have been made in writing the assembly code. Your task is to modify the code, changing *at most 6 lines*, so that the errors are corrected and the program works as desired. The program `prime` should support the following usage interface:

```
prime <n>
```

where the argument `<n>` should be a positive (i.e. ≥ 1) integer . The program should print out the n-th prime number. For example:

```
$ ./prime 10
$ Prime 10 is 29
```

In addition to the `prime.s` file, you have been provided with a `makefile` which contains two rules:

`prime` build the `prime` executable.

`clean` prepare for rebuilding from scratch.

Note: It may be helpful to run `prime` with many different types of input as you make changes to the code to get an understanding of what the program is doing and how you can change the assembly code to make it work correctly.

Important: For this part of the assignment you will only be modifying lines of code that already exist in the `prime.s` file. *You should not add or remove any of the lines.* You may only modify *at most 6 lines of code.*

4 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa2.tar` that can be extracted using the command:

```
tar -xf pa2.tar
```

Your tar file must contain a directory `pa2a` that contains:

- `readme.pdf`: this file should describe your design and implementation of the `fibonacci` 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`:

  `fibonacci` build your `fibonacci` executable.

  `clean` prepare for rebuilding from scratch.

- source code: all source code files necessary for building `fibonacci`. At a minimum, this should include three files, `fibonacci.c`, `fib.s`, and `fib.h`.

Your tar file must also contain a directory `pa2b` that contains:

- `readme.pdf`: this file should list the changes you made to the `prime.s` program. In particular, it should list the line numbers of `prime.s` that you modified and the changes you made to those lines.
- `prime.s`: Your `prime.s` file which includes the changes you made.

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 Design and Analysis

Important: We are highlighting this because it may not have been emphasized in your last classes. *The writeups that you will be providing in the* `readme.pdf` *files are important and will be worth a non-trivial amount of your grade!* So, don't spend all of your time coding. Spending time to think about your design and implementation, understand what it is really doing, and describing it to someone else is a very important skill for you to learn.

5.2 Functionality

This is a large class so that necessarily a 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.*
- Make sure that the outputs are in the correct format (as shown in the examples)

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

5.3 Coding Style

Having said the above about functionality, it is also important that you write "good" code. Thus, *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.

5.4 Performance

Finally, part of your grade will depend on your design. That is, we expect you to write reasonably efficient code based on reasonably performing algorithms. For each assignment, you will need to analyze the performance of your code and justify it as part of discussing your design.