CMSC 216 Project #5 Spring 2016

Assembly Language Programming **Wed Apr 13, 8:00PM**

1 Overview

In this project you will be writing assembly language programs using the Y86 assembly language. Your assignment is to take the C programs we give you and write equivalent assembly programs for each one. You should also aim to have your programs run in as few instructions as possible.

- Fri Apr 8, 8:00PM - Your code must pass the first three public tests (public01, public02, public03). That is the only requirement for this deadline. We will not grade the code for style. This first part is worth .5% of your course grade (NOT .5% of this project grade). Notice you can still submit late for this part.
- Wed Apr 13, 8:00PM - Final deadline for the project. Notice you can still submit late (as usual).

2 Project Files

Copy the project5 folder to your 216 directory. Remember that you need this folder as it contains the .submit file that allows you to submit.

3 Specifications

For each C program, you must create a Y86 assembly language program that functions in a similar manner (meaning it produces the same output as the C program if both are given identical input). Your primary goal should be to produce a working version of each program. As a secondary goal, you should also attempt to have your assembly programs execute on the simulator with as few steps as possible, as measured by the simulator's output. A small portion of your grade will depend on the efficiency of your programs.

Your Y86 source files should be named the same as their corresponding C source file, but with a `.ys` extension. For example, `reverse_prefix_sum.ys` is the Y86 assembly code for `reverse_prefix_sum.c`. The three C programs you will translate are:

1. `palindrome.c` - a program that reads in a string and prints Y if the string is a palindrome and N otherwise.
2. `fibonacci.c` - a program that prints out the *n*'th Fibonacci number.
3. `reverse_prefix_sum.c` - A program that computes the reverse prefix sum of numbers in an array.

All your programs are expected to terminate via a `halt` instruction; abnormal termination should never occur (except in the case of an I/O error, which you are not required to handle). **Also note that each program prints a newline at the end of execution; this is to guarantee that the output is distinguishable from the simulator's normal output. As such, it is imperative that you do not leave it out.**

You may, in all cases, assume that legitimate integer numbers or strings are read in. However, you must make sure your programs work for the ranges of values specified for each program; please pay attention to those ranges.

3.1 Palindrome program

1. Our Y86 variant has no byte-oriented instructions. You will need to store each character in an integer-sized slot, i.e. four bytes.
2. The newline is not part of the string.
3. You can use registers to store local variables (you do not need to use the stack).

3.2 Fibonacci program

This program begins execution by reading a single integer (called n) from standard input. It then prints out the nth Fibonacci number (F_n), followed by a newline, via a recursive algorithm. For the purposes of this assignment, we define $F_0 = 0$, $F_1 = 1$; for all other values of n, $F_n = F_{n-1} + F_{n-2}$.

Notice that you do not need need to define a main function and you can use registers to store local variables. Parameters must be kept in the stack.

You may assume that the value n is in the interval $[0, 14]$ (so that it doesn't take too long to simulate). Note that you **must** implement this program in a recursive manner, and not iteratively.

3.3 Reverse Prefix Sum Program

This program reads a set of values and computes the reverse prefix sum. For example, if the user enters 2 3 6 -1 the program generates 11 9 6 .

You must implement all the functions defined in the provided program, including a main function. The `reverse_prefix_sum` must be implemented using recursion, otherwise you will lose most of the credit associated with this Assembly program.

3.4 Rules

1. You may not change the C programs provided and then implement Assembly for the modified programs.
2. Local variables in C must not become globals in y86. Each local variable must be kept in a designated location on the stack.
3. Parameters must be passed on the stack so that they appear in left to right order. That is, push the last parameter first.
4. Labels for functions must be verbatim as in the assembly. Do not alter case or underscore formatting.
5. **Functions calls must be implemented using call and not jmp.**
6. Any function that takes parmeters must use the stack to store them.
7. If the function is recursive in reverse_prefix_sum.c it must be recursive in reverse_prefix_sum.ys.
8. You don't have to worry about the maximum length of 80 for a line of code.
9. Your comments should follow a style similar to the one illustrated by lecture examples.
10. Each local variable specified in reverse_prefix_sum.c must be defined and used in your assembly. That is, we expect that you:

    a. Reserve space on the stack for the variable.
    b. Have at least one instruction that copies a register content to the local variable memory in the stack.
    c. Have at least one instruction that reads a local variable content that is in stack memory into a register.

4 Common Mistakes

1. Each program prints a newline at the end of execution; this is to guarantee that the output is distinguishable from the simulator's normal output. As such, it is **imperative** that you do not leave it out. If you do you will get an error similar to the following: **runtest:66: bad info line (RuntimeError)**
2. Confusing irmovl with mrmovl or vice versa.
3. Using jmp instead of call.

5 Grading Criteria

Your project grade will be determined by the following:

Results of public tests	20%
Results of secret tests	65%
Code style grading	10%
Code efficiency	5%

Instructions on how to run the public tests will be provided along with the tests, in a file named `README`.

5.1 Style grading

For this project, some style guidelines are obviously different, as you are writing in assembly language, not C. Please pay close attention to these guidelines:

1. Reasonable and consistent indentation is still required.
2. Label names should be descriptive and meaningful.
3. Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).
4. Your code must be thoroughly commented. Assembly language can easily become unreadable without proper documentation, so it is absolutely necessary that you comment your code.
5. Use appropriate whitespace, especially between blocks of instructions that are performing different tasks.
6. Global variables can significantly reduce the complexity of assembly code. While they should be avoided when possible, this is not as necessary as in C.
7. Use the frame pointer `%ebp` if parameters are passed to functions on the stack.
8. You do not have to obey the callee-save (`%ebx`, `%edi`, and `%esi`) and caller-save (`%eax`, `%ecx`, and `%edx`) register saving conventions discussed in the lecture. However, they can be a big help in writing correct code.

5.2 Debugging

Debugging is challenging in assembly language. You can not step through in a source-level debugger, or even use `printf()`.

- However, crude versions of the `printf()` approach can be useful. For example, printing a specific letter each time you enter a given function can be useful.
- The interpreter will print out changed registers and memory locations. The latter can be identified by looking at where global variables are placed in the .yo file.
- The interpreter can accept a second argument that limits the number of steps that it executes. You can also use the "halt" instruction to see the state of the machine.
- Try to write, and test, small pieces of code at a time. This is even more important with assembly than with C.
- The `yis` simulator has a limit of 100000 steps per execution. While your programs should not take anywhere near that many steps to run, keep it in mind when designing your programs, so that they all stay within this boundary for all valid inputs.

5.3 Code efficiency

Running the `yis` simulator on your assembled code will print the number of steps the simulator takes to execute your program. You are expected to, once your code is working, attempt to revise it to lower this count, so that your code is more efficient.

Since the programs have ranges of valid inputs, we will usually be more concerned with your code's efficiency toward the high ends of those ranges rather than at the lower ends.

6 Submission

6.1 Deliverables

The only files we will grade are your three Y86 source files (`palindrome.ys`, `fibonacci.ys`, and `reverse_prefix_sum.ys`). Any changes made to the C files will not affect your submission or grade at all.

6.2 Procedure

The submission procedure is the same as for previous projects (execute "`submit`" in the project directory).

6.3 Possible problem with submit command

If you try to submit your project in grace, and you get the error:

"Exception in thread 'main' java.lang.OutOfMemoryError: unable to create new native thread"

then close all terminals windows except one, and try to submit again.

7 Academic Integrity

Please see the syllabus for project rules and academic integrity information. All programming assignments in this course are to be written individually (unless explicitly indicated otherwise in a written project handout). Cooperation between students is a violation of the Code of Academic Integrity.