CMSC 216 Project #4 Spring 2013

Date due: Thursday, April 11, 10:00:00 p.m.

1 Administrative

The TAs' "extra" office hours for this project will run from Monday, April 8, through Friday, April 12. During that time they will hold the higher–level office hours shown on the office hours handout. Note that April 12 is the one–day–late deadline. Of course the project may be submitted two days late, but the second late day is Saturday, April 13, on which there are no office hours. It is recommended that you not wait until Saturday to finish the project, in case you find you need to get help with something in office hours.

As before, you will have the opportunity to earn extra credit on this project, but as before you may also lose points for having an excessive number of submissions. Appendix C has the full details. On the other hand, if you make too many submissions for this project, you may again lose credit.

Be sure to frequently check the Projects page on ELMS, where this project is posted, to see if any clarifications or corrections to this assignment have been made during the time that it is assigned.

The public tests will be provided during the time that the project is assigned. The project will be set up on the submit server after the public tests are provided. In order that you can start working on the project right away, the project tarfile, with the required files, is available now. When the public tests are available they will be added to the project tarfile (a note will be put on the Projects page on ELMS to let you know when this is), and you can copy it and extract the files from it again to get the public tests.

2 Introduction

In this project you will be writing assembly language programs, using the Y86 assembly language described in the Bryant and O'Hallaron textbook (with slight modifications). Your assignment is to take the C programs we give you and write equivalent assembly programs for each one.

3 Development procedure

3.1 Obtain the project files

We are supplying you with both a tarfile containing the files you will be working with, and a Y86 assembler and simulator. Both the `yas` and `yis` programs discussed in lecture are available in the `~/216public/bin` directory, so they should already be in your path. The `yas` and `yis` assembler and simulator support all of the Y86 instructions discussed in Project #2 except for the conditional move instructions (`cmovele`, `cmovl`, etc.) The tarfile contains a `.submit` file and four C programs (`prog1.c`, `prog2.c`, `prog3.c`, and `prog4.c`) that you must translate to Y86 assembly programs. Extract the files from the tarfile into your `~/216/project4` directory, using a procedure similar to what you used for prior projects.

3.2 Create the assembly source files

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 were given the same input). Your primary goal should be to produce a working version of each program. Your Y86 source files **must** be named the same as their corresponding C source file, but with a `.ys` extension; for example, `prog1.ys` is the Y86 assembly code for `prog1.c`.

4 Specifications

The four C programs you will translate are:

1. `prog1.c`, a program that performs exponentiation via repeated multiplication;

2. `prog2.c`, a program that reads in n numbers and performs insertion sort on them;
3. `prog3.c`, a program that searches for an occurrence of one string inside another one (like what strchr does), and
4. `prog4.c`, a program that reads in n numbers, stores them in an array (it is assumed that they are in increasing sorted order), reads in an additional number, and searches in the array for that last number that was read, using binary search.

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 your assembly programs do the same, otherwise the output of your programs will be on the same line as the output of the simulator, so your output will appear to be incorrect on the submit server, and you will fail all of the tests.

You may assume, in all programs that read integer input, that legitimate integer numbers will be input, and that the numbers will be within the range of an integer. Note the assumptions for some of the programs regarding what input values they will be run on; you do not have to worry about inputs that do not satisfy those assumptions.

4.1 Exponentiation program

This program reads in two integers from the standard input and then computes and prints out the first one raised to the power of the second one, followed by a newline.

There are no assumptions for this program, other than what is mentioned above (that you may assume that valid integers will be input). The Y86 only has integer and character data types and instructions (no real numbers), but the if statement in the program ensures that 1 is printed if the second number is negative.

4.2 Sorting program

This program reads a list of integers from the standard input, stopping when a 0 is read or when 100 integers have been read, storing the numbers into an array as they are read. It then sorts the integers using the insertion sort algorithm, and prints them out with a single space after each number– including the final one–, followed by a newline.

You may assume that a 0 will appear somewhere in the program's input; it will be preceded by zero or more integers (any of which may be positive or negative). Even if there are no numbers at all to be sorted (no integers appear before a 0) a newline is still printed. Note that the integers to be sorted may be positive or negative, but can never be 0, because an 0 in the input signifies the end of the list of input numbers.

4.3 Program to find one string in another

This program first reads in a string of characters from the standard input, by reading all the characters in the input up to the first newline. It then reads another string in the same fashion, and searches for the first occurrence of the entire second input string in the first input string. The index of the beginning (first character) of the occurrence of the second string in the first one is printed, followed by a newline, but -1 (followed by a newline) is printed if the second string is not found as a substring anywhere in the first string's contents.

You may assume that a newline character will always appear in the input in the first 101 characters; the newline will be preceded by zero or more characters. You may assume that another newline will appear in the next 101 characters, also preceded by zero or more characters. In other words, the two input strings will each have up to 101 characters, including their terminating newlines.

Note that our Y86 variant has no byte–oriented instructions. You will need to store each character in an integer–sized slot, i.e., four bytes.

4.4 Binary search program

This program also reads a list of integers from the standard input, stopping when a 0 is read or when 100 integers have been read, storing the numbers in an array as they are read. It then reads one additional number, and performs binary search on the array to search for that value. If it is found, the program prints the subscript of the array element where it was found where the first element has subscript 0, otherwise -1 is printed. Either the subscript or the -1 is followed by a newline.

You may assume that the elements read are in increasing sorted order, and they are unique.

Note that you **must** implement the function called by this program in a recursive manner, and not iteratively. If your function is not recursive all credit that you get for all public and secret tests of this part of the project will be deducted from your score during grading.

5 Writing a Makefile

You must write and submit a makefile with your project. Your makefile will be used to assemble your program on all of the public tests, so if you don't submit a makefile that satisfies the requirements below your programs will not assemble for any of the tests, and your score will be zero. When the tests are provided you will know how many of them there are and what their names are. Since you will not know before the project due date how many secret tests there will be, or what their names are, we will use our own makefile to assemble those.

Your makefile **must** be in a file named `Makefile`. There are no options that should be used in assembling the programs. Your `Makefile` **must** contain the following targets:

all: This target must create `.yo` "object" files for all four of your programs.

a `.yo` object file for each of your programs: Since Y86 code is simulated by the `yis` program (which is an interpreter), no Linux executables will be created by your makefile, and of course separate compilation will not be used.

clean: This target must delete the `.yo` object files for all of your programs from the current directory.

Your makefile may contain additional targets if you desire, so long as it has the targets described. The "clean" target may remove additional files if desired, but it must remove all `.yo` object files for your four programs.

It is strongly recommended that before submitting you run "make clean" then "make all", to ensure that your makefile builds all the public tests correctly.

A Development procedure review

A.1 Running your programs and checking your results

As you have seen (or will see) in lecture, the `yis` simulator prints out a considerable amount of information about changed memory and registers. Because your implementation need not manipulate the same memory locations and registers as ours, we cannot simply do a straight diff of the simulator output to test your programs. Therefore we have created a program named `runtest` that runs your assembly programs on the simulator and filters out the extraneous output produced by `yis`. The `runtest` program is located in `~/216public/bin`, along with the `yas` assembler and `yis` simulator.

For this project, the public tests are data files that each contain three pieces of information:

- the assembled program to be executed
- the input to that program when simulated
- the expected output of that program when simulated

These pieces of information are contained in a file with a `.yml` filename extension (for those interested, it is in a format called YAML, which is a markup language). If the test is testing a program named `prog1.yo`, which is supposed to print the first number in its input raised to the power of the second number in its input, a hypothetical file `public01.yml` to run it might look something like this:


```
input: 3 6
output: 729
program: prog1.yo
```

Note that the order of the lines is not important. The public tests will all be files named in the format `publicXX.yml`. To check if your program passes a test, simply run the `runtest` program with the name of the testfile you want to check as the argument to runtest; for example:

```
grace4:~/216/project4: runtest public01.yml
Testing prog1.yo with input 3 6: PASSED in 56 steps.
```

If your program does not yield correct output, or if it stops execution via an exception, `runtest` will note that and declare the test failed. If the test is passed, runtest reports how many steps were taken to complete execution.

The third program is a slight exception to the above, in that it must read more than one line of input (two lines). The syntax in YAML to express a multiline input is to place a vertical bar (pipe symbol) after the word `input:`, and to begin each line of the input after that with a single blank space, as illustrated below (blank spaces in the file are indicated using ␣):

```
input:␣|
␣hippopotamus
␣pot
output:␣5
program:␣prog3.yo
```

The blank space at the beginning of the two lines is not read (the two strings read in this input will be `hippopotamus` and `pot`), but indicate that they are all part of the input. For the third program, which must read two input lines, two lines will begin with a blank space, as above. Be sure to add a blank space before the input strings in writing your own tests of the third program, since they will also have to contain blank spaces in the same way a our tests do in order to work right.

The `runtest` program is not written in C, and in any case you do not need to read it or understand it.

A.2 Submitting your program

As before, the command `submit` will submit your project. **Before** you submit, however, you must first check to make sure you have passed all the public tests, by running them yourself. And **after** you submit, you **must** then log onto the submit server and check whether your program worked right on the public tests there.

Do **not** submit programming assignments using the submit server's mechanism for uploading a jarfile or zipfile or individual files. Do not wait until the last minute to submit your program, because the submission server enforces deadlines to the exact second.

Project extensions will not be given to individual students as a result of hardware problems, network problems, power outages, etc., so you are urged to finish **early** so any such situations will not affect you even if they do arise.

A.3 Grading criteria

Your project grade will be determined according to the following weights:

Results of public tests	35%
Results of secret tests	50%
Code style grading	15%

The public tests will not be comprehensive, and you will need to do testing on your own to ensure the correctness of your functions.

A.3.1 Style grading

For this project, some style guidelines are different, as you are writing in assembly language, not C. However, assembly language is still a programming language, and good style is important to use in any language, so that you or others can read and understand your code. Please pay close attention to these guidelines:

1. All your Y86 source code files should begin with a comment containing your name, University ID number, and Grace login/Directory ID.
2. Lines should be no longer than 80 columns.
3. Reasonable and consistent indentation is required, although Emacs will likely not be of much use to you. As assembly language is straightforward (syntactically) compared to higher–level languages, it should not be too difficult to manually maintain indentation.
4. Label names should be descriptive and meaningful.
5. 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.
6. Use appropriate whitespace, especially between blocks of instructions that are performing different tasks.
7. "Global" variables (using a label, and `.long` or `.pos` directives) can significantly reduce the complexity of assembly code. While they should be avoided when possible, this is not as necessary as in C. For example, the string or array storage in some of the programs could be in global variables for simplicity (although a real compiler would actually place them in the runtime stack, since they are local variables). Note, however, that values that would be locals in a recursive C function cannot be placed in global variables, as recursive calls will overwrite data needed by prior calls.
8. Use the frame pointer `%ebp` if parameters are passed to functions on the stack, as will be explained in class.
9. You do not have to obey the callee–save (`%ebx`, `%edi`, and `%esi`) and caller-save (`%eax`, `%ecx`, and `%edx`) register saving conventions to be discussed in lecture. In fact, it is often easier to write code where the callee saves all modified registers, in which case calling a function will be completely transparent to the calling context.

B Project–specific requirements, notes, and hints

- Try to write, and test, small pieces of code at a time. This is even more important with assembly than with C code.
- 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.
- The last line of each of your Y86 source files **must** end with a newline character, because the assembler will not work properly unless the final line of an assembly program is terminated by a newline. Furthermore, the assembler will not print an error message if the last line of an assembly program does not end with a newline; it will just produce strange and inconsistent results. You have been warned. Always press "return" or "enter" after the last line of any assembly program.
- Your code must be developed in and submitted from your extra course disk space.
- In all cases each of your assembly programs must perform identically and produce the same results as the corresponding C program.

- There is no requirement that your programs be efficient. However, your programs **must** do the same thing that the C programs do, **and use the same algorithms**. For example, if your sorting program uses selection sort instead of insertion sort, you will not receive any credit for any public or tests of that program. You should strive to implement the programs in assembly as closely to what the C programs are doing as possible. Do not add any extra conditions or checks, or remove any of the existing conditions or checks.
- To reiterate again, you **must** use recursion for the last program, passing parameters via the runtime stack (of course this is a consequence of the prior item).
- Be sure to carefully reread the requirements for your makefile in Section 5 before submitting.

  Important: do not wait until the end of coding to write your makefile. Write it first, and use it every time you compile your programs. It will be quicker than typing the compilation commands by hand, and every time you test your code you will be debugging your makefile, allowing you to find any problems in it before submitting.
- Recall that the course syllabus says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details. (The compilation penalties mentioned below only apply to submissions made before the end of the project late submission period.)
- For this project you will **lose one point** from your final project score for every submission that you make in excess of five submissions. You will also **lose one point** for **any** submission that does not assemble. Check your makefile carefully, as described in Section 5 above.

C Extra credit

If you make **only one submission** for this project, and that submission passes all of the **public and secret tests**, we will give you **5 extra–credit bonus points** when your project is graded.

As mentioned in the prior section, credit may be lost for making an excessive number of submissions, or having submissions that fail to assemble correctly.

D Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.

