1 Overview

Originally, this project was going to require that you implement Bubble Sort over strings. However, it turned out that that project was too hard, so I simplified it. Instead of performing Bubble Sort, you will implement string compare (comparable to C's standard function \texttt{strcmp}()) between strings that are on an array.

As with previous programs, this program has multiple modes; in one mode, you will print out all of the strings from an array; in another, you will print a single character from one of the strings in the array; in the last, you will compare two strings - and swap them in the array if they are out of order.

1.1 Required Filename

Name your file \texttt{proj03.s}

1.2 New Feature - Program Splitting

Some students have lost points on previous projects because they turned in code that included the testcase variables. This caused them to fail all testcases, since the grading script combines your code with each testcase in turn.

We have offered partial credit for this in the past, but \textbf{no longer}. However, we have improved the grading script to work around this problem. You are encouraged (but not required) to include the following line at the top of your .s file:

\begin{verbatim}
# ----- STUDENT CODE BELOW -----
\end{verbatim}

Our grading script will look for this line; if it finds it, it will remove any code above this line when it runs the testcases. So if you add this, your code should work even if you forget, and leave the testcase code in place. But if you don’t add this line, it’s up to you to get things right when you turn in!

1.3 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- \texttt{add, addi, sub}
- \texttt{sll, srl, sra}
• and, andi, or, ori, nor, xor, xori
• beq, bne, j
• slt, slti
• lw, lh, lb, sw, sh, sb
• la
• syscall

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

2 Modes

As before, I will provide variables which tell you what sort of operation to perform. However, instead of using boolean flags, this time we will use a single variable (a byte named mode).

At the beginning of your program, read mode. If it is one of these values I’ve provided, then perform that single operation. If not, then print out the error message

Invalid mode.

(with no leading or trailing spaces, but with a newline at the end) and then terminate your program.

You must implement the following modes:

• (There used to be a Mode 1, but it was removed. If mode=1, print Invalid mode.)

• Mode 2 - Print Strings

I will provide an array of strings (stored as an array of addresses) and a length of the array (a byte). You will print out all of the strings, one per line.

• Mode 3 - Double-Index

I will provide an array of strings and a length (as in Mode 2), as well as two indices. You will use the first index (s1) to choose a string from the array, and the second index (c) to choose a character from that string. You will print that character, followed by a newline.

Note: You are not required to verify that the indices are valid; trust that they are within the range of the array, and the string.
• Mode 4 - Compare Two Strings, and Swap
I will provide an array of strings and a length (as in Mode 2). I will also provide the indices of two strings (s1,s2). You will compare the two strings. If they are out of order (that is, the string at s1 ought to come after the string at s2), then swap the two strings in the array. (Don’t move the characters, just move the pointers!)
After the comparison and (possible) swap, print out all of the strings in the array, just like Mode 2. Since the last step in Mode 4 is identical to Mode 2, one option is to simply jump to Mode 2. If you do this, you may break the “keep all blocks in order” rule from the Style Guide - but just for this one jump.
Note: You are not required to verify that the indices are valid; trust that they are within the range of the array. You should also assume (not verify) that str1 < str2.

3 Testcase Variables
Every testcase will provide the following variables. Do not assume anything about their order in memory; however, pay attention to the types - since the variables will always have the types given here.
Note that all variables will be in all testcases - even if they are not being used.
• mode (byte)
  This variable tells you what operation to perform during this run of the program.
• strings (array of addresses)
  This is the array of addresses to various strings.
• numStrs (byte)
  This is the length of the strings array.
• s1, s2 (byte)
  These two variables are string indices; they give the index of a string from the strings array. Mode 3 will use only s1; mode 4 will use both s1,s2.
• c (half word) This variable is the index of a character. It is used only in Mode 3.
4 \texttt{strcmp}

I have provided a very limited implementation of the C standard library function \texttt{strcmp()}. I encourage you to use it for Mode 4 - but be aware that you will have to adapt it a bit; \texttt{strcmp()} returns a value at the end; you will have to swap (or not swap) two string pointers.

Note that this function has some simplifications which would not be present in real C code - for instance, real \texttt{strcmp()} implementations will routinely return values other than just -1, 0, 1.

\begin{verbatim}
int strcmp(char *str1, char *str2)
{
    int i = 0;

    while (str1[i] != \'\0\' && str2[i] != \'\0\')
    {
        // do we have a mismatched character?
        if (str1[i] < str2[i])
            return -1;
        if (str1[i] > str2[i])
            return 1;
        i++;
    }

    // str1 or str2 (or both) have hit the end of their
    // strings; they were identical up to that point.

    if (str2[i] != \'\0\')
        return -1;  // str1 is shorter than str2
    if (str1[i] != \'\0\')
        return 1;  // str2 is shorter than str1

    // exact match!
    return 0;
}
\end{verbatim}

5 \textbf{A Note About Grading}

Your code will be tested automatically. We will include all of the testcases that we have provided to you - and also some number of additional testcases, which we will not give to you. Since your code is being tested automatically, you must match the required output \textbf{exactly}! Any extra spaces, blank lines (or missing lines), misspelled words, etc. will cause your testcase to fail.
Thus, be careful to test your code thoroughly, and to read the spec carefully.
Test your code with every testcase that we provide! While it’s not a perfect
 guarantee, if your code passes all of the testcases we provide, there’s a good
 chance that it will also pass the ones that we have not given to you.

5.1 Testcases and Grading script

You can find a set of testcases - and the grading script - for this project at
http://www.cs.arizona.edu/classes/cs252/fall16/projects/proj03/
Each testcase has two parts: the .s file that contains the code we provide, and
an .out text file which shows exactly what the output should be.

For information about running - and checking - your code with the testcases
and/or grading script, see the Project 1 spec.

6 Turning in Your Solution

You must turn in your program using turnin on lectura. Use the assignment
name cs252_f16_proj03. Turn in only your program (for example: proj03.s);
do not turn in any testcases, grading scripts, or other files.

If you need to use a late day, simply add .late to the end of the assignment
name.