

Assembly Project #4

due at 7pm, Wed 13 Nov 2024

1 Purpose

Let's practice some more! We're going to write a whole bunch of functions: practicing with functions themselves (with a variety of arguments), calling other functions (including recursion), as well as reading and writing various variables and arrays.

1.1 Required Filenames to Turn in

Name your assembly language file `asm4.s`.

1.2 Allowable Instructions

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

- `add`, `addi`, `sub`, `addu`, `addiu`
- `and`, `andi`, `or`, `ori`, `xor`, `xori`, `nor`
- `beq`, `bne`, `j`
- `jal`, `jr`
- `slt`, `slti`
- `sll`, `sra`, `srl`
- `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 Tasks

Your file must declare a set of functions, as detailed below. In this project, you won't have any global variables provided by the testcase; instead, everything that you need to know will be provided through parameters.

(In the descriptions below, I've described some of the functions by giving you C code; I've described others using words. Of course, you'll be writing MIPS assembly for all of them!)

2.1 Task 1: Array Duplicate Search

Write a function named `arrayDupSearchInts()` which searches through an array of `int`, looking for a specific value. If you find the value in the array **twice in a row**, then return the index of the first of the two values. (If you don't find this sort of duplicate, then return -1.)

You are searching for a specific value; don't simply return the location of **any** duplicate; look for a duplicate of the specific value you were given.

The function will take three parameters:

- `int*` - The array to search
- `int` - The length of the array
- `int` - The value to search for

EXAMPLES

If the array is

[3, 9, 15, 7, -5, -10, 5, 5, 12, -2, 19]

and you are looking for the value 5, then return 6. But if you were looking for -5, then return -1.

(spec continues on the next page)

2.2 Task 2: Substring Search 5

Write a function named `stringSearch5()` which searches a string for 5 specific characters (each given as an argument). If those characters are found in the string, all in order and sequential, then return the index of the first character. Otherwise, return -1.

The function will take 6 parameters. The first is a null-terminated string, and the other 5 are the characters to search for.

EXAMPLES

If the string is

```
"Hank and John are good guys"
```

and you are looking for the characters `'a','n','d',' ','J'`, then return 5. But if you were looking for `'h','a','n','k',' '`, then return -1 (because the comparisons are case-sensitive).

2.3 Task 3: Head to the Polls

Why not? Let's do an election themed one. You will write a set of functions that model vote counting. You must use **global variables** (declare them in your own file) to store the totals as we count.

In this election, there are two parties: the Purple and Yellow parties. Start both of their vote counts at 0.

Implement a function named `addVotes(purple,yellow)` which adds the specified number of votes to each party. You may assume that neither of the values are negative, and either (or both) might be zero. Normally, your function will simply update the values stored in the global variables, and print out nothing. However, if the candidates are tied, then print

```
TIED!!!
```

More interestingly, if the lead **changes** (that is, Yellow was leading and Purple pulls ahead, or vice-versa), then print out a message which announces the change (along with the current vote totals). This will be either:

```
Purple pulls ahead!  purple:5678  yellow:1234
```

or

```
Yellow pulls ahead!  purple:1234  yellow:5678
```

(There are two spaces before `purple:` and three before `yellow:`)

(spec continues on the next page)

2.4 Requirement: Don't Assume Memory Layout!

It may be tempting to assume that the variables are all laid out in a particular order. Do not assume that! Your code should check the variables in the order that we state in this spec - but you **must not** assume that they will actually be in that order in the testcase. Instead, you must use the `la` instruction for **every variable** that you load from memory.

To make sure that you don't make this mistake, we will include testcases that have the variables in many different orders.

3 Running Your Code

You should always run your code using the grading script before you turn it in. However, while you are writing (or debugging) your code, it is often handy to run the code yourself.

3.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from the Mars website. You may be able to just double-click it in your operating system; if not, then you can run it by typing the following command:

```
java -jar <marsJarFileName>
```

This will open a GUI, where you can edit and then run your code. Put your code, plus **one**¹ testcase, in some directory. Open your code in the Mars editor; you can edit it there. When it's ready to run, assemble it (F3), run it (F5), or step through it one instruction at a time (F7). You can even step **backwards** in time (F8)!

3.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the **Settings** menu, and set two options:

- **Assemble all files in directory** - so your code will find, and link with, the testcase
- **Initialize Program Counter to 'main' if defined** - so that the program will begin with `main()` (in the testcase) instead of the first line of code in your file.

¹Why can't you put multiple testcases in the directory at the same time? As far as I can tell (though I'm just learning Mars myself), the Mars GUI only runs in two modes: either (a) it runs only one file, or (b) it runs **all** of the files in the same directory. If you put multiple testcases in the directory, it will get duplicate-symbol errors.

3.2 Running Mars at the Command Line

You can also run Mars without a GUI. This will only print out the things that you explicitly print inside your program (and errors, of course).² However, it's an easy way to test simple fixes. (And of course, it's how the grading script works.) Perhaps the nicest part of it is that (unlike the GUI, as far as I can tell), you can tell Mars exactly what files you want to run - so multiple testcases in the directory is OK.

To run Mars at the command line, type the following command:

```
java -jar <marsJarFileName> sm <testCaseName>.s <yourSolution>.s
```

4 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

4.1 Testcases

For assembly language programs, the testcases will be named `test*.s`. For C programs, the testcases will be named `test*.c`. For Java programs, the testcases will be named `Test*.java`. (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have “secret testcases,” which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

²Mars has lots of additional options that allow you to dump more information, but I haven't investigated them. If you find something useful, be sure to share it with the class!

4.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm3`, along with a helper script, `mips_checker.pl`. Place both scripts, all of the testcase files (including their `.out` files), and your program files in the same directory. (I recommend that you do this inside the standard Docker container that I've provided. It **might** also work on your Mac or Linux box, but no promises!)

4.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception**. We encourage you to share your testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

5 Turning in Your Solution

You must turn in your code to GradeScope. Turn in only your program; do not turn in any testcases or other files.