CS 252 (Summer 20): Computer Organization

Assembly Project #3

Intro to Functions

due at 5pm, Fri 24 Jul 2020

1 Purpose

In this Project, we'll be using functions for the first time[1]. You will implement several different functions, which can be called by the testcases; some of those functions will have to call other functions.

Note: In this project you will **not** be writing a `studentMain()` function; instead, you will implement other functions, which the testcases may call.

1.1 Required Filenames to Turn in

Name your assembly language file `asm3.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`
- `mult, div, mfhi, mflo`

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.

[1] In truth, you've been defining `studentMain()` since Asm 1 - but this is the first time that you understand what you're doing.

1.3 Files

You can find the files for this project in the following locations:

- The GitHub Classroom repository
- The project directory on the class website: `http://lecturer-russ.appspot.com/classes/cs252/summer19/asm/asm3/`.
- The mirror of the class website, accessible on any department computer: `/home/russelll/cs252m19_website/`

2 No Standard Wrapper

There's no standard wrapper for this project; you will be writing several independent functions. But it might be interesting to go back to the Assembly Project 2 spec, and see what you've been doing. By now, you have the information necessary to understand what `studentMain()` was.

3 Task Overview

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!)

3.1 Properly Saving `sX` (and other) Registers

The testcase includes lots of features which are designed to verify that you are saving registers properly.

First of all, the MIPS calling convention requires that every function preserve the value of all `sX` registers, and also `$fp`. Likewise, the stack pointer must be restored before the function returns.

To test this, each testcase will initialize all of the `sX` registers - and `$fp` as well - to various 32-bit values. At the end of the testcase, we'll print out these values (along with a word we passed onto the stack); if you have saved those registers properly - and also restored `$sp` to the proper location - then your output will match the expected output.

3.2 Properly Saving `tX` (and other) Registers

Similarly, the MIPS calling convention says that **any** function is entitled to corrupt any `tX,aX,vX` register; it is not required to save and restore them.

Thus, if you call another function and have values that you care about in any one of these registers, it is the **caller's** responsibility to save the value.

To test this, every function in the testcases which you might call will **intentionally** corrupt every `tX` register, `aX` register, and `vX` register. (Many will return a value in `v0`; in that case, they will only corrupt `v1`.)

Thus, if your code depends on any value that you've stored in a `tX` register - and you don't save it properly - then your program will operate improperly.

3.3 Matching the Output

You must match the expected output **exactly,** byte for byte. Every task ends with a blank line (if it does anything at all); do not print the blank line if you do not perform the task. (Thus, if a testcase asks you to perform no tasks, your code will print nothing at all.)

To find exactly the correct spelling, spacing, and other details, always look at the `.out` file for each example testcase I've provided. **Any example (or stated requirement) in this spec is approximate; if it doesn't match the `.out` file, then trust the `.out` file.**

(Of course, if there are any cases in the spec which are not covered by any testcase that I've provided, then use the spec as the authoritative source.)

Task 1: `strlen()`

Implement a function which takes a single parameter: a pointer to a string. Count the number of characters in the string (that is, read up to the null terminator). Return the count.

NOTE: Since you are given a pointer to the string, you **must not** attempt to load its address from a label; you have **no idea** what the label might be. You've been given an address - use it!

Task 2: `int gcf(int a, int b)`

This function calculates the GCF (Greatest Common Factor) between two numbers. **Your implementation must be recursive.**

Implement the following code. (See below for information about how to do multiply and divide in MIPS.)

```
int gcf(int a, int b)
{
    if (a < b)
        swap(a,b);
    if (b == 1)
        return 1;

    if (a % b == 0)
        return b;
    else
        return gcf(b, a % b);
}
```

(You will notice that the function above assumes that both inputs are positive numbers; you are not required to enforce this assumption in any way.)

Multiply/Divide, and Move From Hi/Lo

MIPS provides divide and multiply instructions. However, both of them need more than one register to hold the answer: multiply produces a 64-bit result, and divide will **simultaneously** calculate both the quotient and the remainder. Thus, these two instructions have **two** destination registers.

Surprisingly, the way MIPS chose to do this was to have two special registers for this purpose, `$hi,$lo`. These cannot be accessed directly, but you can use the instructions `mfhi` ("move from hi") and `mflo` to copy them into other registers. For instance, if you want to divide `$s0` by `$t3`, and you want to put the quotient into `$s4` and the remainder into `$s5`, you would do:

```
div   $s0, $t3        # lo = (s0 / t3)
                      # hi = (s0 % t3)
```

```
mflo  $s4             # s4 = lo = (s0 / t3)
mfhi  $s5             # s5 = hi = (s0 % t3)
```

(Note that you aren't required to read both HI and LO; you are allowed to read only one of them, if that's what you need.)

Task 3: bottles(int)

Implement the following function:

```
void bottles(int count, char *thing)
{
    for (int i=count; i>0; i--)
    {
        printf("%d bottles of %d on the wall, %d bottles of %d!\n",
               i,thing, i,thing);
        printf("Take one down, pass it around, %d bottles of %s on the wall.\n",
               i-1,thing);
        printf("\n");
    }

    printf("No more bottles of %s on the wall!\n", thing);
    printf("\n");
}
```

NOTE 1: Remember that the format specifier `%d` prints out an integer, and `%s` prints out a string.

NOTE 2: You have been given the address of a string. You should not bother with reading any of its characters, since you don't care what they contain. Instead, simply print the string!

Task 4: `int longestSorted(int *array, int len)`

The first parameter is a pointer to an array of integers; the second is the length of the array. Scan through the array, and figure out the length of the longest sorted run - that is, the longest sequence of integers from the array which are sorted. Return the number that you find.

(Note that, unlike a previous assembly project, you should not check for both ascending and descending sequences - **only** check for ascending ones.)

If the array is empty, return 0. If it is not empty, then the smallest possible that you might return is 1. Of course, at all times, the maximum value that you might return is the length of the array.

Task 5: `int rotate(int count, int a, int b, int c, int d, int e, int f)`

```
int rotate(int count,
           int a, int b, int c, int d, int e, int f)
{
    int retval = 0;

    for (int i=0; i<count; i++)
    {
        retval += util(a,b,c,d,e,f);

        int tmp = a;
        a = b;
        b = c;
        c = d;
        d = e;
        e = f;
        f = tmp;
    }

    return retval;
}
```

NOTE: `util()` is a function that will be provided by the testcase. Each testcase will provide a **different** implementation.

4 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 often handy to run the code yourself.

4.1 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**[2] testcase, in some directory. Open your code in the Mars editor;

[2]Why 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.

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)!

4.1.1 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.

4.2 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).[3] 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
```

5 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.

[3] Mars 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!

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!

5.1 Testcases

You can find a set of testcases for this project in the GitHub Classroom; they are also present on the class website.

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.**

5.2 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 on Lectura, or a similar department machine. It **might** also work on your Mac or Linux box, but no promises!)

5.3 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 you 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!

6 Turning in Your Solution

You must turn in your code using GitHub Classroom. Turn in only your program; do not turn in any testcases or other files.

Make sure that your code is actually on GitHub before the deadline. This will require that you add the file **and** then push it to GitHub. (You can confirm that you have uploaded the files correctly by viewing your repo through the GitHub website.)