CS 252 (Summer 19): Computer Organization

Assembly Project #3

Introduction to Functions

due at 5pm, Thu 18 Jul 2019

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`

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 Project 2 - 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 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!)

3.1 Task 1: Collatz, Part 1

The Collatz Conjecture (`https://en.wikipedia.org/wiki/Collatz_conjecture`) is a fun little property of integers; it has been shown to be true for **many** positive integers - but nobody yet knows if it is true for **all** positive integers. In this function, you'll model part of it - the part where an even number is divided down, many times, until you find an odd number underneath.

Declare a function named `collatz_line()`. It must take a single `int`[2] parameter, and also return an `int`. It must print out a sequence of integers, all on one line, with a newline at the end. The integers must be separated by single spaces - but there must **not** be any leading or trailing spaces.

The first integer that you print must be the parameter. If the parameter is even, then you will divide it by two and print the new value; you continue until you have reached an odd number. (If the parameter is odd, then print it, by itself on the line.)

Return the odd number that you find at the end of the loop.

[2] That is, a MIPS word.

3.2 Task 2: Collatz, Part 2

This will model the Collatz Conjecture for a certain number, all the way down until it hits 1.

Convert the following C function to MIPS assembly:

```
void collatz(int val)
{
    int cur   = val;
    int calls = 0;

    cur = collatz_line(cur);

    while (cur != 1)
    {
        cur = 3*cur+1;
        cur = collatz_line(cur);
        calls++;
    }

    printf("collatz(%d) completed after %d calls to collatz_line().\n", val, calls);
    printf("\n");
}
```

Remember that `printf()` is not available in MIPS; you'll have to do syscalls.

3.3 Task 3: Find the % Character

For this task, declare a function named `percentSearch()`. This function takes a single parameter, which is a string[3]. It returns an `int`.

Search the string for the first percent sign (`'%'`). If you find one, return the index of that character. If not, then return -1.

[3] That is, the parameter is a MIPS word which gives the address of the first element of a null-terminated array of characters

3.4 Task 4: A Tree of Letters

Convert the following C function to MIPS assembly:

```
int letterTree(int step)
{
    int count = 0;
    int pos   = 0;

    while (1)    // we'll break out manually, when required
    {
        char c = getNextLetter(pos);
        if (c == '\0')     // this is literally *ZERO*
            break;

        for (int i=0; i<=count; i++)
            printf("%c", c);          // use syscall 11
        printf("\n");

        count++;
        pos += step;
    }

    return pos;
}
```

The testcase will provide the implementation for the `getNextLetter()` function; it will return different values in different testcases. (Some testcases may even print out things when you call the function.)

The prototype for `getNextLetter()` is as follows:

```
char getNextLetter(int);
```

3.5 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.

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**[4] 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)!

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).[5] 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] 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.

[5] 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!

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.

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 testcaes 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.