CS 252 (Fall 24): Computer Organization

Assembly Project #5

due at 7pm, **Mon** 25 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 `asm5.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, subu`
- `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: `merge()`

Write a function which implements the `merge()` function from Merge Sort, over arrays of integers. That is, you will be given three arrays of integers - two inputs, and one output. I will give you the length of the two input arrays, and you can assume that the third one has a length equal to their sum. Iterate through the two arrays in parallel, always copying the smaller value into the output.

In order to check that you are doing this correctly (and also, to give you practice with calling functions), you will call the function `merge_debug()` after each new value is added to the output array. (I will provide this function in the testcase.)

Here is the C code for this function:

```
void merge(int *a, int aLen, int *b, int bLen, int *out) {
    int posA=0, posB=0;

    while (posA < aLen || posB < bLen) {
        if (posB == bLen || posA < aLen && a[posA] <= b[posB]) {
            out[posA+posB] = a[posA];
            posA++;
        } else {
            out[posA+posB] = b[posB];
            posB++;
        }
        merge_debug(out, posA+posB);
    }
}
```

(spec continues on the next page)

2.2 Task 2: `quicksort()`

The previous task had you implement only part of a sorting algorithm. For this one, you'll write the whole thing!

Here is the C code for this function:

```
void quicksort(int *data, int n) {
     if (n < 2)
         return;

     int left  = 1;       // first unsorted index.  Note that [0] is the pivot.
     int right = n-1;     // last unsorted index

     while (left <= right) {
         quicksort_debug(data,n, left,right);

         while (left <= right && data[left]  <= data[0])
             left++;
         while (left <= right && data[right] >  data[0])
             right--;

         if (left < right) {
             quicksort_debug(data, n, left,right);

             int tmp = data[left];
             data[left] = data[right];
             data[right] = tmp;

             left++;
             right--;
         }
     }

     quicksort_debug(data, n, left,right);

     int tmp = data[0];
     data[0] = data[left-1];
     data[left-1] = tmp;

     quicksort_debug(data, n, -1,-1);

     quicksort(data,      left-1);
     quicksort(data+left, n-left);

     quicksort_debug(data, n, -1,-1);
 }
```

2.3 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 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.

[1] 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.

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.**

[2] 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!

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 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!

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.