CS 252 (Summer 20): Computer Organization

Assembly Project #2

Loops, Arrays, and Strings

due at 5pm, Fri 10 Jul 2020

1 Purpose

In this project, you will be using loops, iterating over arrays of integers and strings. You will be implementing both `for()` and `while()` loops.

1.1 Reminders

Remember: if you're needing help with MARS, you can watch the "Using the MARS Simulator" video on the class Panopto site.

Also, pay attention to the Asm Style Guide, which is available on the class website.

1.2 Required Filenames to Turn in

Name your assembly language file `asm2.s`.

1.3 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`
- `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.

1.4 Standard Wrapper

Use the same Standard Wrapper as Asm1; read the spec from that project to find information about it.

2 Task Overview

As with Asm 1, you will read a number of different control variables (each of which are words). They will be either 0 or 1; you will do that particular task if the control variable is 1.

In addition, there will be a number of other variables, used by the various tasks. These will be detailed in the appropriate sections below.

2.1 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.**

3 Task 1: countNonzero

In the `countNonzero` task, you must iterate through the array (of words) named `intArray[]`; its length is `intArray_len` (also a word).

As you iterate through the array, count how many non-zero values there are in the array (along with how many of them are positive). Then print out the following message, followed by a blank line:

```
The intArray[] had 13 nonzero values, 7 of which were positive.
```

(Of course, replace the two numbers with the proper values.)

NOTE: `intArray_len` will never be negative in this project - although it might be zero.

4 Task 2: skipPrint

In the `skipPrint` task, you must print out every other character of the string `str`. That is, you must print elements `[0]`,`[2]`,`[4]`, etc. (Remember, the string is null-terminated, and the null terminator might be at either an odd or even position.)

After printing out all of these characters, print a newline at the end, followed by a blank line.

NOTE: You must not modify the string `str` - and you must support arbitrarily long strings. Thus, you need to print out the characters, one at a time, using syscall 11.

5 Task 3: groupCount

In this task, you will implement the following C code:

```
if (groupCount != 0)
{
    for (int i=0; i<intArray_len; i++)
        groups[intArray[i] & 0x3]++;

    printf("The group sizes are: %d,%d,%d,%d\n",
           groups[0], groups[1], groups[2], groups[3]);
    printf("\n");
}
```

The testcase will provide the array

```
int groups[4];
```

You may assume that it will be initialized to all zeroes; you do not need to initialize it.
Note that your code must actually update that array (don't just keep the values in registers) because the testcase will print out the contents of that array at the end.

6 Task 4: indirect

In this task, you will read the words from `intArray[]`; for each one, you will use it as an index into the string `str`. Print out those characters, followed by a newline at the end (and then a blank line).

You may assume that the values in the `intArray[]` are all non-negative (and also within the length of `str`) for this task. (If this task is **not** selected, then the values in `intArray[]` can take on any value, and might be negative or very large.)

7 Task 5: toUpper

In this task, you will **modify** the string `str`, converting any lowercase characters to uppercase. Print the string at the end; add a trailing newline after it, followed by a blank line.

NOTE: Unlike the `skipPrint` task, this task **must** modify the string.

8 Task 6: Factors

For this task, implement the following C code:

```
if (factors != 0)
{
    printf("Factors of %d:", num);

    int curVal = num;
    int f = 2;
    while (curVal > 1)
    {
        if (curVal % f == 0)
        {
            printf(" %d", f);
            curVal = curVal / f;
        }
        else
            f++;
    }

    printf("\n");
    printf("\n");
}
```

(You should assume that the testcase declares the variable `num`, and that it is a word.)

8.1 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.)

9 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.

10 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.

10.1 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)!

10.1.1 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

[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.

- `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.

10.2 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
```

11 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!

11.1 `mips_checker.pl`

In addition to downloading `grade_asm2`, you should also download `mips_checker.pl`, and put it in the same directory. The grading script will call the checker script.

[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!

11.2 Testcases

You can find a set of testcases for this project at
`http://lecturer-russ.appspot.com/classes/cs252/summer20/asm/asm2/`
You can also find them on Lectura at
`/home/russelll/cs252m20_website/asm/asm2/`
.

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.**

11.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm2`, 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!)

11.4 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!

12 Turning in Your Solution

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

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.)