CS 252 (Fall 23): Computer Organization

Assembly Project #2

Loops, Arrays, and Strings

due at 5pm, Wed 20 Sep 2023

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 Required Filenames to Turn in

Name your assembly language file `asm2.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`
- `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.

1.3 Standard Wrapper

Your code will need to define a single function, named `studentMain()`. Use the same Standard Wrapper for this function as you used in Asm 1 (see the project spec there for details).

2 Tasks

Your code (inside the `studentMain()` function) will read three control variables.[1] These are `printInts, printWords, bubbleSort`, and each represents one high level operation. Each will be either 1 or 0.

In addition, `printInts` has a secondary control value; the variable `printInts_howToFindLen` can take three possible values: 0,1,2. These represent three different ways that you might determine the length of the `intsArray[]`:

- 0 - length variable. Read the variable `intsArray_len` to find the length.
- 1 - pointer subtraction. We have another variable, named `intsArray_END`, which is immediately **after** the last element in the array. Subtract the two pointers (start and end of the array) to figure out the length.
- 2 - null terminator. It's very unusual to have a null-terminated array of integers, but it happens occasionally.

2.1 Types of the Variables

The variables will have the following types. You may assume that all testcases use the same types - but **do not** assume anything about the order in which variables are declared (except that you may assume that the "END" pointer for `intsArray` is at the end of that array).

The following variables are bytes: `printInts, printWords, bubbleSort`.

The following variables are halfwords: `printInts_howToFindLen`.

The following variables are words: `intsArray_len`, and all of the elements of the `intsArray[]` array. The `intsArray_END` label points just past the end of the array; you can treat it like it's a pointer to `int`.

(spec continues on the next page)

[1] It's OK to read them ahead of time, if you want. But in this program, I had a little more register-pressure, and so I chose not to read these variables until they were required.

2.2 Task 1: Print Ints

This task prints out several integers from `intsArray[]`, one on each line. There are 3 different "modes" - which control how much you will print out from the array.

Implement the following C code:

```
if (printInts != 0)
{
    if (printInts_howToFindLen != 2)
    {
        int count;

        if (printInts_howToFindLen == 0)
            count = intsArray_len;
        else
            count = intsArray_END - intsArray;  // remember to divide by 4!

        printf("printInts: About to print %d elements.\n", count);

        for (int i=0; i<count; i++)
           printf("%d\n", intsArray[i];
    }
    else
    {
        /* searches for a null terminator */
        int *cur = intsArray;   // same as &intsArray[0];

        printf("printInts: About to print an unknown number of elements.  "
               "Will stop at a zero element.\n");   // all one line!

        while (*cur != 0)
        {
            printf("%d\n", *cur);
            cur++;
        }
    }
}
```

(spec continues on the next page)

2.3 Task 2: Print Words

This task scans through a string, converts spaces (just spaces, not other whitespace) into null terminators, counting how many times it does this as it goes. When it hits the end of the string, it then moves **backwards**, and prints out each word; since the spaces have been turned to null terminators, each will be just a simple word, not the whole line.

Implement the following C code:

```
if (printWords != 0)
{
    char *start = theString;
    char *cur   = start;
    int   count = 1;

    while (*cur != '\0')   // null terminator.  ASCII value is 0x00
    {
        if (*cur == ' ')
        {
            *cur = '\0';
            count++;
        }
        cur++;
    }

    printf("printWords: There were %d words.\n", count);

    while (cur >= start)
    {
        if (cur == start || cur[-1] == '\0')
            print("%s\n", cur);
        cur--;
    }
}
```

(spec continues on the next page)

2.4 Task 3: Bubble Sort

Bubble Sort is notoriously slow, but it's the easiest sort to implement.

In this code, we'll use the same `intsArray` as in the `printInts` task - but in this case, you don't need to worry about the various modes. If I ask you to do Bubble Sort, then your Bubble Sort code should **always** use `intsArray_len` to find the length.

Implement the following C code: **Make sure that you actually modify the integers in memory, or this won't work!**

```
if (bubbleSort != 0)
{
    for (int i=0; i<intsArray_len; i++)
        for (int j=0; j<intsArray_len-1; j++)
            if (intsArray[j] > intsArray[j+1])
            {
                print("Swap at: %d\n", j);

                int tmp        = intsArray[j];
                intsArray[j]   = intsArray[j+1];
                intsArray[j+1] = tmp;
            }
}
```

2.5 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.

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**[2] 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.

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).[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
```

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).

[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.

[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!

- (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 `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.

4.2 Testcases

You can find a set of testcases for this project in the zipfile that I've posted - both on D2L, and 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.**

4.3 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!)

4.4 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!

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.