CS 252: Computer Organization

Assembly Project #2

Loops, Arrays, and Strings

due at 11:59pm, Sat 2 Oct

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

Be sure to pay attention to the Asm Style Guide, which is available on Piazza.

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`

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. This is the function prologue and epilogue so that the tests can call your code.

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.**

(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.)

3 Task 1: Fibonacci

In this task, you will print out Fibonacci numbers. Use the iterative algorithm I've provided below. (You have certainly seen the recursive implementation, but that has $O(2^n)$ cost, whereas this one has $O(n)$ cost. Plus, you don't know how to do functions yet.)

Implement the following code:

```
if (fib != 0)
{
    printf("Fibonacci Numbers:\n");
    printf("  0: 1\n");
    printf("  1: 1\n");

    int prev = 1, beforeThat = 1;
    int n = 2;

    while (n <= fib)
    {
        int cur = prev+beforeThat;
        printf("  %d: %d\n", n, cur);

        n++;
        beforeThat = prev;
        prev = cur;
    }

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

NOTE: Unlike most other control variables in this project, `fib` can take more values than just 0 or 1. If `fib==0`, then do nothing; if it is positive, then use it as the limit for the loop. (You may assume that it will never be negative.)

4 Task 2: square

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

```
if (square != 0)
{
    // NOTE: square_fill is a byte
    // NOTE: square_size is a word

    for (int row=0; row < square_size; row++)
    {
        char lr, mid;
        if (row == 0 || row == square_size-1)
        {
            lr  = '+';
            mid = '-';
        }
        else
        {
            lr  = '|';
            mid = square_fill;
        }

        printf("%c", lr);
        for (int i=1; i<square_size-1; i++)
            printf("%c", mid);
        printf("%c\n", lr);
    }

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

5 Task 3: Run Check

A handy tool, used by some sorting algorithms, is to detect if a list of values is either an ascending list (that is, each value is `>=` the one before) or a descending list (`<=`).

If `runCheck==1`, then scan through the array of integers. You can find its location at `intArray`, and it contains `intArray_len` integers. (You may assume that the length I provide you is non-negative, and that the array has at least that many values. It might have more, but if so, you must ignore them.)

If it is an ascending list, then print out

```
Run Check: ASCENDING
```

and if it is a descending list, then print out

```
Run Check: DESCENDING
```

(It is possible, in a corner few cases, for both cases to be true. In that case, print out both lines, ascending first.)

If neither is true, then print out

```
Run Check: NEITHER
```

In any of these cases, print out a blank line afterwards, if you performed the task.

NOTE: Zero-length arrays, and arrays with only a single element, are both valid. Both of these special cases should be reported as **BOTH** ascending and descending lists.

6 Task 4: countWords

If `countWords==1`, then read the string `str`. Count the number of words in it; words will be separated by spaces or newlines (you do not have to check for any other types of whitespace). Print out the number of words, followed by a blank line, like this:

```
3
```

Ignore leading and trailing spaces, and multiple spaces in a row. For instance, the following string has 3 words:

```
str = "   foo bar    baz ";
```

Do not modify the string in memory.

7 Task 5: revString

If `revString==1`, then scan through the string `str` and reverse it in memory. You may implement your own code if you wish, but your algorithm **must** be able to handle any size string!

Here is one possible implementation - implement this, or your own design:

```
if (revString != 0)
{
    int head = 0;
    int tail = 0;

    // advance the tail until we find the correct location
    while (str[tail] != '\0')
        tail++;
    tail--;

    // when we get here, tail gives the index of the last non-NULL
    // character in the string.  If the string is empty, it actually
    // would be -1, which is weird but OK.

    while (head < tail)
    {
        swap(str[head], str[tail]);
        head++;
        tail--;
    }

    printf("String successfully swapped!\n");
    printf("\n");
}
```

8 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.

9 Running Your Code

You should always run your code on lectura using the grading script before you turn it in. However, while you are writing (or debugging) your code, it is often handy to run the code yourself.

9.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from Piazza. 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)!

9.1.1 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.

9.2 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:

[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.

[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!

```
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s
```

10 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!

10.1 `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.

10.2 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.**

10.3 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!)

10.4 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!

11 Turning in Your Solution

From Lectura, navigate to the folder that **contains** the folder asm2. Then run the command: turnin cs252f21-asm2 asm2. **Please turn in only your program; do not turn in any testcases.**

You must ensure that your folder is named asm2 and it contains files that exactly match filenames described above in this spec.