1 Purpose

In this Project, we’ll be breaking some of the rules. Well, not really; you’re still going to follow all of the rules of MIPS - including the calling conventions. However, you will be doing some things which are pretty low-level, and which you can’t do in a structured language like C.

Each of these is a useful trick, which is used in the Real World. Sometimes, they are used by compilers to build more efficient code; other times, they are things which you can do with a library in C. (The library includes code that was written in assembly, which very carefully does the unusual thing for you.)

1.1 Required Filenames to Turn in

Name your assembly language file \texttt{asm4.s}.

1.2 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- \texttt{add, addi, sub, addu, addiu}
- \texttt{and, andi, or, ori, xor, xori, nor}
- \texttt{beq, bne, j}
- \texttt{jal, jr}
- \texttt{slt, slti}
- \texttt{sll, sra, srl}
- \texttt{lw, lh, lb, sw, sh, sb}
- \texttt{la}
- \texttt{syscall}

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), \textbf{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 define the functions listed below:

2.1 Task 1: bubbleSortString()

In this function, you will be given a string (as a parameter). You will perform Bubble Sort on the characters in the string - but you must not modify the original array. Instead, you will copy the array into a temporary array on the stack, and sort that.

Each time that you swap two items, you will call a utility function provided by the testcase, debugBubbleSort(). You will pass it three parameters: the indices of the two characters that you swapped, and the pointer to the duplicate string (where you're doing the copying).

Since your duplicate string is on the stack, you will need to add space to your stack frame. However, you must not simply hard-code a certain length; instead, you must find the length of the input string and allocate an appropriate size. You are allowed (and encouraged) to write any number of helper functions; for instance, strlen() would be handy.

(For details about how to allocate an array of arbitrary size on the stack, see the discussion below.)

2.2 Task 2: jumpTable()

In this function, you will implement a fairly simple function. You already know how to do this with conditional branches - but for this function, I have banned all conditional branches.

Instead, you will use a jump table. A jump table is an array, where the values in the array are addresses of instructions. This is easy to do in MIPS - because the .word assembler directive allows you to use label names as the init constant:

```
.data
MY_ARRAY: .word LABEL1
           .word LABEL2
```

Your function will use the ‘switch’ variable as the index into an array; you will read the address from the array, and then jump to the address given. In a real program, it would be very wise to implement bounds checking (making sure that the switch variable is inside a reasonable range), but for this task, we'll simplify and simply assume that it is OK.

Using a jump table, you must implement the following function:
void jumpTable(int indx, int param)
{
    switch(indx)
    {
    case 0:
        print("Case Zero: \%d\n", param);
        // intentional fall-through!

    case 1:
        print("Case One: ");
        printHex(param);
        printf(" ");
        printHex(param*3);
        printf("\n");
        break;

    case 2:
        print("Case Two: \%d\n", param);
        jumpTable(1, param*5);
        break;

    case 3:
        print("Case Three: \%d\n", param);
        break;

    case 4:
        print("Case Four: \%d\n", -param);
        break;
    }
}

NOTE: The function printHex() is provided by the testcase. It takes a single integer parameter, and prints it out as 8 hex characters. Basically, it’s equivalent to printf("\x08x", val);

3 Flexible-Length Arrays on the Stack

When you declare an array on the stack in C, you typically need to use a constant size for the array[1]. The simple reason for this is that the compiler needs to know

[1]There are exceptions. Some compilers will allow you to declare an array which uses a variable as the length if this array is the last local variable for that function. In effect, the compiler is implementing exactly what we’re doing in this project! However, I don’t believe that all compilers allow this.

Additionally, there is a C library call, alloca(), which allows you to perform these sort of allocations. However, its use is discouraged, and it is not necessarily available in all environments.
the size of the stack frame, so that it can allocate the proper size (and so that it can load/store local variables).

However, assembly can break this rule. Since we have direct control of the stack pointer, we can decrement the stack pointer as much as we want, in order to expand the stack frame. But there are a couple of problems with this:

- We need some way to save the length of the buffer for later - so that we can shrink the stack frame again. There are a number of ways to do this - but a simple one is to simply store the length in a register.
- The stack pointer must always be word aligned. (Can you figure out why this is true?)

So, your code at the beginning of the function must do three things:

- Calculate the size of the array you’ll need (remember to account for the null terminator)
- Round that length up to a multiple of four
- Allocate an array of that size on the stack

(Of course, at the end of the function, you’ll need to reclaim this stack space: but if you are careful, you won’t have to calculate the amount again. Instead, you can simply use a register to tell you how much to reclaim.)

## 4 Rounding Up

Here’s a trick to remember - it comes up a lot in programming. To round any number up to the next multiple of some factor \( x \), you first add a small value, then divide by \( x \), and then multiply by \( x \).

But how much should you add? To examine that, consider what would happen if you did not add anything. So (for example) give yourself a bunch of different integers, and divide each by 4, then multiply them by 4. (Remember that integer division rounds down.) Which give the correct answer? Which are wrong? What would you add to make them all work correctly?

Second, you’ll notice that I don’t (yet) allow you to perform multiply and divide. So, can you find a sequence of operations (using only the currently-allowed instructions) which do the same as divide by 4, followed by multiply by 4?

## 5 Good Exam Questions

This project has a couple of interesting things that you should study and understand - I think that they would be excellent Short Answer questions for a future exam.
First, can you round the length up to a multiple of 4? Could you generalize this to rounding up to a different power of 2? How would you modify this so that you could round up to a value which is not a power of 2? And how could you round down?

Similarly, could you adapt the “round up” or “round down” code to handle rounding a fraction after division? For instance, imagine that you had count items, and you needed to spread them across bins which could only carry 6 items. How many bins would you need?

Second, can you explain why the stack pointer always needs to be word-aligned? What would happen if it wasn’t?

6 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.

6.1 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 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)!

6.1.1 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.

---

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.
6.2 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). 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
```

7 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!

7.1 mips_checker.pl

In addition to downloading `grade_asm4`, you should also download `mips_checker.pl`, and put it in the same directory. The grading script will call the checker script.

7.2 Testcases

You can find a set of testcases for this project at [http://lecturer-russ.appspot.com/classes/cs252/spring18/asm/asm4/](http://lecturer-russ.appspot.com/classes/cs252/spring18/asm/asm4/)

You can also find them on Lectura at [http://lecturer-russ.appspot.com/classes/cs252/spring18/asm/asm4/](http://lecturer-russ.appspot.com/classes/cs252/spring18/asm/asm4/)

---

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!
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.

7.3 Automatic Testing

We have provided a testing script (in the same directory), named grade_asm4, 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!)

7.4 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 your 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!

8 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.

8.1 Late Work

I will set up a separate folder in D2L for Late work. Please do not put any files in that folder unless you intend to take a late day. (If you put files in both folders, we’ll ignore the regular folder and only use your files from the Late Day folder.)