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`

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

To start with, your code (inside the studentMain() function) will perform a single string-printing task. You will always perform this task, in every testcase. Then, your code will read two control variables: countCars, choose, both of which are words. Each is either 1 or 0; do the operation if the control variable is nonzero.

In addition, there will be a number of other variables, used by the three operations. 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 0: Reverse Print

In the first task, you will read a string, byte by byte, and print it out; however, you will reverse it. The string is an array of bytes, and is null-terminated; it is named printString.

You may not make any assumptions about the length of the string, and I’ve not yet shown you how to dynamically allocate memory - so it’s not possible for you to copy the string, and then use syscall 4. (Also, it’s illegal to modify the string.) Instead, you will need to print out the string one character at a time, moving backwards through it.

After you have printed out the (reversed) string, print out two newline characters: one to end the line you’re on, and another to place a blank line after it. This task should not print anything other than this.

4 Task 1: Count Cars

In the countCars task, we imagine that there is some sort of mechanism which counts cars as they go by on the road; periodically, it records a count to an
array. However, occasionally the machine goes haywire, and so we have to filter out invalid entries.

There is a fixed amount of space available; sometimes, the machine fills up all of the slots in the array. Other times, we only get a partial set of records - the value -999 (if it exists) indicates the end of the records.

You must implement the following C code:

```c
if (countCars != 0)
{
    int total = 0;
    int count = 0;

    for (int i=0; i<carsArray_len; i++)
    {
        if (carsArray[i] == -999)
            break;
        if (carsArray[i] >= 0)
        {
            total += carsArray[i];
            count++;
        }
    }

    printf("Total number of cars: %d\n", total);
    printf("Valid entries: %d\n", count);
    printf("\n");
}
```

In this code, there are two local variables (total, count). You should use registers to store those (and also to store the various temporaries you will need). carsArray[] is an array of halfwords, which will be provided by the testcase; carsArray_len is a word, which gives the maximum amount of entries which might be in the array.
5 Task 2: Choose Your Own Adventure

This task simulates reading through Choose Your Own Adventure book (https://en.wikipedia.org/wiki/Choose_Your_Own_Adventure). I have provided an array of words; this array is arranged in pairs, with each pair representing a page on the book - and two choices that you can make on that page. (Equivalently, you can think of this as an array of structs, where each struct contains two words.)

For this task, I will give an English-language description of the code you are to implement. You must convert this to C, and place the C code in your block comment above this section of the assembly file. (I'll allow you to mix in a bit of pseudocode into your C - but make sure that the code you write is clear and very straightforward to convert to assembly.) Part of your grade will be based on the quality, clarity, and correctness of the C code you place in the comment!

5.1 The Input Array: int cyoaTable[]

So, what does this task do? First, of all, just like Task 1, it checks a control variable (choose), and will do nothing if this is zero. However, if the control variable is nonzero, it will do the following:

The “reader” of the book (your program) will always start on page 0 of the book (we’re using zero-based numbering on the pages!). Each page has two numbers (both single bytes), so page 0 of the book takes up slots [0],[1] in the array, and page 1 takes up slots [2],[3]. These two numbers represent two choices (that is, two page numbers) you might take from the current location; we’ll call them A and B.

Each choice is represented by a page number, or by -1. If you “choose” a certain page, then you go to the point in the array that represents that page, and then read the two numbers there, and continue. You loop until you find an end to the story.

End-of-story pages are any page which has -1 for both numbers. Pages which have only one option (that is, an unconditional branch) will be represented by a page number in slot A, and -1 in slot B. Choices (where you can go either of two places are represented by page numbers in both slots. (-1 in slot A is only valid in an end-of-story page: that is, if -1 is also in slot B.)

NOTE: To simplify your program, you may assume that any page which has -1 for A will have -1 for B as well. We will sometimes have invalid page numbers in the table - but only check a page number for validity after you have followed it. If there is an invalid page number, but you don’t choose it, then ignore it.

5.2 Required Output

On each page that you read, you will print out three numbers, all on the same line; see the .out files I provide for examples.
5.3 When to Terminate

Your program will loop, reading various pages in the book, until it reaches an end-of-story page (remember, that’s a page where A and B are both -1); print out the information for that page, and then terminate the loop. Do not print out any extra message, other than the information for that page.

If you hit any of the following error scenarios, print out an error message (see the testcases for examples) and terminate the loop. **Perform these checks in the order given here; that way, if a testcase has two of these errors, you know which to report.**

- **Jump to an invalid page number**
  
  If the program ever jumps to a page number which is negative, or that is $\geq \text{cyoaNumPages}$, report the error and terminate the program before you attempt to read the array at the (invalid) location.

  \text{cyoaNumPages} is a word.

- **Looping Too Many Times**
  
  If you ever read more than 20 pages, terminate the program before you read the 21st page. (This will help you detect infinite loops.)

5.4 How to Choose

Your program will simply alternate choices. On its very first choice, it will choose the first option (A); on the next it will choose B, and then alternate until the loop ends.

If a page has an unconditional branch, do **not** count this as making a choice; instead, remember the current state, and use it the next time that the choice comes up.

For example, suppose that page 0 has (10, 23). You program chooses 10, because this is the first choice. Page 10, however, has an unconditional branch: (43, -1). So your program jumps to page 43. On page 43, you find another choice: (17, 3). Your program chooses 3 (B), because it chose A the last time that it had the ability to choose.

6 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 \texttt{la} instruction for
\textbf{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.

7 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.

7.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you down-
loaded 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:

\begin{verbatim}
java -jar <marsJarFileName>
\end{verbatim}

This will open a GUI, where you can edit and then run your code. Put your
code, plus one \texttt{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 \textbf{backwards}
in time (F8)!

7.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the \texttt{Settings}
menu, and set two options:

\begin{itemize}
  \item \texttt{Assemble all files in directory} - so your code will find, and link
        with, the testcase
  \item \texttt{Initialize Program Counter to ‘main’ if defined} - so that the pro-
        gram will begin with \texttt{main()} (in the testcase) instead of the first line of
        code in your file.
\end{itemize}

7.2 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)\footnote{However, it’s
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!}
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
```

8 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!

8.1 mips_checker.pl

Like in Project 1, we’ll be checking your source code to see if it follows the
rules - in this case, we’ll be checking to make sure that you don’t use any
pseudoinstructions. In Project 1, we could do this with a simple `grep`
command; in Project 2, the check requires a Perl script.

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.

8.2 Testcases

You can find a set of testcases for this project at

```
http://lecturer-russ.appspot.com/classes/cs252/spring18/asm/asm2/
```

You can also find them on Lectura at

```
/home/russell11/cs252_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.

## 8.3 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!)

## 8.4 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!

## 9 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.

### 9.1 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.)