Assembly Project #5

Variables on the Stack due at 7 pm, Wed 9 Apr 2025

Purpose

In this Project, you'll write functions that use variables that must be stored on the stack. You must not use global variables; you must use extra-large stack frames.

Required Filenames to Turn in

Name your assembly language file `asm5.s`.

Allowable Instructions

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

- `add, addi, sub, addu, addiu, subu`
- `and, andi, or, ori, xor, xori, nor`
- `beq, bne, j`
- `jal, jr`
- `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.

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, how did we 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?

Tasks

Your file must define the functions listed below:

Task 1: `countLetters()`

Implement the following function. You must use stack space to store the array. (Notice that I've carefully designed this function so that there are more `int`s to store than you have available registers!)

You must not use any global variables! (If you use `.data` for anything other than string constants, you probably are breaking this rule.)

```
  void countLetters(char *str)
   {
        int letters[26];    // this function must fill these with zeroes
        int other    = 0;

        printf("----------------\n%s\n----------------\n", str);

        char *cur = str;
        while (*cur != '\0')
        {
            if (*cur >= 'a' && *cur <= 'z')
                letters[*cur-'a']++;
            else if (*cur >= 'A' && *cur <= 'Z')
                letters[*cur-'A']++;
            else
                other++;

            cur++;
        }

        for (int i=0; i<26; i++)
            printf("%c: %d\n", 'a'+i, letters[i]);
        printf("<other>: %d\n", other);
    }
```

Task 2: `subsCipher()`

Implement the following function, which is a "substitution cipher." A substitution cipher is an ancient style of encryption, where each letter or symbol is replaced with another; in order to encode or decode the message, you simply need a table which "maps" from the cleartext to the ciphertext[^1].

In this function, I will pass you a string (which is null-terminated), and a "map," which is a table of characters[^2]. The table will have 128 entries (since the "normal" ASCII characters only use up the first 128 possible values in a byte); your program will look up each character in the message, and encode it, using the map[^3]. However, you must not modify the original string - and so you must store the encoded message in a buffer that is allocated on the stack.

This function calls `strlen()` - you will have to provide the implementation for this function.

Most importantly, this function uses an array variable which has a length which is not known until the function runs. This is not something which is not typically allowed in C - but which is relatively straightforward to implement in assembly. See the discussion below to see how it works.

```
  void subsCipher(char [] str, char [] map)
   {
       // NOTE: len is one more than the length of the string; it includes
       //       an extra character for the null terminator.
       int len = strlen(str)+1;

       int len_roundUp = (len+3) & ~0x3;
       char dup[len_roundUp];    // not legal in C, typically.  See spec.

       for (int i=0; i<len-1; i++)
           dup[i] = map[str[i]];
       dup[len-1] = '\0';

       printSubstitutedString(dup);
   }
```

NOTE: The bitmask constant above is the bitwise negation of `0x3`. In other words, it is "every bit set except for the two lowest." If you accidentally use negative three, you will get an incorrect result, because the negative three (in two's complement) is `1111...1101`.

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[^4]. The simple reason for this is that the compiler needs to know 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.
  For this reason, I've saved the length in a C variable. You don't have to use this variable, but if you do, it will be easy to clean up.
- The stack pointer must always be word-aligned. (Can you figure out why this is true?)

So, our code at the beginning of the function does three things:

- Calculates the size of the array we'll need (remembering to account for the null terminator)
- Rounds that length up to a multiple of four
- Allocates an array of that size on the stack

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.

The Report

For this project, in addition to writing some assembly, you also need to write a short report detailing some features of your code. To find the required information, load up your solution (**`asm5.s`**) and a test that calls the function subsCipher. The easiest way to do this is to put both files in the same directory (with no other **`.s`** files) and then turn on the "Compile all files in the directory" option of MARS.

Assemble these two files, and do a quick test run to ensure your code works.

For this report, you need to examine the arrays `str` and `map` in memory.

Set a breakpoint at the **`JAL subsCipher`** instruction and step into the function. Fill out the values on registers $a0 and $a1 and write their meaning for this function.

Register	Value when calling **`subsCipher`**	What is this value?
$a0		
$a1		

Using the function arguments, examine the memory on MARS (.data) and take a (full screen)[1] screenshot of the data on arrays `str` and `map`. Include these screenshots in your report.
Complete the following table:

	Address of element	Value of element
The first element of the array `str`		
The last element of the array `str`		
The first element of the array `map`		
The last element of the array `map`		

Grading Rubric

[1] We will check that no two students have the same screenshot.

By now, you should already know that you cannot use pseudo-instructions, so starting with this project, the autograder will assign you a **zero or a negative score** if you use any pseudo-instructions.

You can verify if you are using pseudo-instructions by disabling pseudo-instructions in MARS and reviewing your Gradescope submission. If the "`mips_checker.pl Results`" shows up as a test case on Gradescope, you used pseudoinstructions.

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.)

To find exactly the correct spelling, spacing, and other details, always look at the `.out` file for each example testcase! 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 included testcase, then use the spec as the authoritative source.)

Turning in Your Solution

You must turn in your code and your report to GradeScope. Turn in only your program; do not turn in any testcases.

Name your assembly language file `asm5.s`.

Name your report `asm5_report.pdf`

[^1]: Substitution ciphers are notoriously easy to crack. They can often be cracked by hand - and so are never used for any important secrets!

[^2]: Since the length of the map is fixed, it is not required to have a null terminator. However, my testcase adds one, just so that it's possible to print out the map. This null terminator should never be used by your code.

[^3]: Since the map is only 128 characters long, there would be a serious problem if I sent you any input string which used the upper 128 values - you'd be accessing undefined memory. For this reason, I'll promise to send you only strings which use the ordinary 128 ASCII characters.

[^4]: 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.
```

[^5]: 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]: 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!