CSC 350: Computer Architecture
Spring 2019

Assignment #2

Due: Monday, March 11, 2019 at 12:50 pm by submission to conneX

Overview

This assignment involves assembly-language coding with the use of a combination of assembler and MIPS simulator. The one to be used for this assignment is the `MARS` (MIPS Assembler Runtime Simulator) made at Missouri State University (`https://bit.ly/2vcJ5zi`).

Students may be requested to explain to the instructor, in person, some of their submitted answers, after which their assignment marks will be released.

The assignment is in three parts where the first two can be considered something like "warm-up" exercise for the much harder third part (although the results of these two parts will be needed for the third part).

Setup

As mentioned above you will use the MARS application for not only writing your solution in MIPS assembly but also running, testing, debugging. Although there are other MIPS emulators (e.g., SPIM and its relatives) these are much harder to use. As your code will be tested using MARS, you must use this system.

MARS is written in Java and therefore will work on macOS, Windows, and Linux systems. (There is a slight "gotcha" with macOS which will be described in class.) Follow the instructions at the link above regarding installation and execution.

Part A: `FUNCTION_STRCMP`

In this part you will write an implementation of a string-comparison function that rhymes with `strcmp()`in a way that it behaves as you might expect in a C program. That is, in C a call of:

```
    strcmp("abc", "def")
```

returns `-1` as "abc" comes before "def" in lexicographic order; similarly:

```
    strcmp("def", "abc")
```

returns `1` as “def” appears after “abc” in lexicographic order; and lastly:

```
    strcmp("ghi", "ghi")
```

returns `0` as both strings passed to `strcmp()` are the same as each other (i.e., they would appear in the same position in lexicographic order). However, you will not be writing your code in C, nor will it be called from a C program. And, of course, the string values that will passed as parameters to your assembly-language function will differ from the examples just given.

You have been provided with a starter file named `part_a.asm`. Note that most of the file currently consists of some assembler directives, allocations of memory for global variables (in the .data) section, and some driver code that interacts with the user, calls `FUNCTION_STRCMP`, and outputs the value returned by that function. All of your solution for the first part of the assignment must appear in this function.

There is also a large comment just before `FUNCTION_STRCMP`. It reads:

```
###########################################################################
#
# YOUR SOLUTION MAY NOT ADD MORE GLOBAL DATA OR CONSTANTS.
# ALL OF THE CODE FOR YOUR FUNCTION(S) MUST APPEAR AFTER
# THIS POINT. SUBMISSIONS THAT IGNORE THIS REQUIREMENT
# MAY NOT BE ACCEPTED FOR EVALUATION.
#
###########################################################################
```

This comment indicates a strict requirement for your solution. You are *not permitted* to modify any of the code before this comment, or add any additional `.data` sections after the comment **unless given express written permission by the course instructor**. Your function must operate correctly as it is called by the driver code, and also work when during evaluation some of the constants (such as `MAX_WORD_LEN`,`MAX_WORD_LEN_SHIFT` or `MAX_NUM_WORDS`) are changed. During marking your assignment will always be called with properly-formed input (i.e., we will not be testing for error handling). Any changes to constants will be coherent, e.g., if `MAX_WORD_LEN` is changed to 64, then `MAX_WORD_LEN_SHIFT` will be changed to 6 (which means `MAX_WORD_LEN` will always be a power of 2).

Part B: `FUNCTION_SWAP`

The code in part A does not modify the parameters passed to it. The code in this part, however, must swap the contents of two strings. That is:

- The function receives the starting address of each string.
- It copies the characters corresponding to the first string into a temporary area.
- It then copies the characters corresponding to the second string into the memory of the first string.
- Finally it copies the characters in the temporary area into the second string.

The copying will involve using loops and the `lbu` and `sb` MIPS instructions. Further the space for the temporary area must be obtained from the stack, which will involve increasing the size as the start of the function and decreasing it at the end of the function.

You have been provided with a starter file named `part_b.asm`. Note that most of the file currently consists of some assembler directives, allocations of memory for global variables (in the `.data`) section, and some driver code that interacts with the user and calls `FUNCTION_SWAP`. All of your solution for the second part of the assignment must appear in this function.

The same strict requirements as described for part A also apply for part B with respect to modifying the provided code.

Part C: `FUNCTION_PARTITION` and `FUNCTION_HOARE_QUICKSORT`

You will need your work from parts A and B to complete part C. In fact, you will need copy-and-paste your functions into the starter file provided to you (`part_c.asm`). (And, yes, the strict requirements are repeated for this part.) In this part your code will sort an array of strings where the strings are input by the user; this input and output code is already provided to you.

The quicksort implementation you are to write must be the one described in this particular section of the Wikipedia article for *quicksort*:

```
https://en.wikipedia.org/wiki/Quicksort#Hoare_partition_scheme
```

For operations such as:

```
A[i] < pivot
```

you will use your `FUNCTION_STRCMP`, and actions such as:

```
swap A[i] with A[j]
```

you will, of course, call upon your own `FUNCTION_SWAP` implementation.

The hardest task in this part will be the implementation of `FUNCTION_PARTITION`, **and you must write this as a separate function**. (In fact, once this function is completed, calling it from `FUNCTION_QUICKSORT` in a way that works is almost laughingly easy.)

Please make use of `FUNCTION_PRINT_WORDS` provided to you in part_c.asm as a debugging aid. You will find this will be an very helpful aid when debugging FUNCTION_PARTITION.

And two final cautions:

- For all of the parts of this assignment, your implementations of the functions must properly save registers (at function start) and properly restore registers (at function end). If this is not properly done, you will run into infuriating bugs as registers in some function have their values changed in ca called function in a way that you do not intend or desire. Put all of the register-save code at the start of functions, and all register-restore code at the end of functions. Do not intersperse save and restore code amongst the rest of the function as that style of coding infuriatingly difficult to keep correct.
- Beware of what you read on *Stack Overflow* and other places. You will find lots of hits for "MIPS" and "quicksort", but the slough of despond into which you descend trying to figure out the postings will not be worth the visit. Many of those postings are aimed at new programmers or those unfamiliar with many aspects of computer science. If you not only trust what you have learned so far in CSC and SENG courses, but also use that knowledge as you attempt a solution from first principles, then you will find coding easier and more successful.

What you are to submit to conneX

Your completed solutions to the three parts of the assignment in appropriately modified versions of:

- `part_a.asm`
- `part_b.asm`
- `part_c.asm`

Note that the last file will contain duplication of some of the work in the first two files, but we do require three separate files.

Evaluation

Our grading scheme is relatively simple.

- “A” grade: An exceptional submission demonstrating creativity and initiative. The code for all three parts runs without any problems. The solution to the assignment is clearly written.
- “B” grade: A submission completing the requirements of the assignment. The code for all three parts runs without any problems.
- “C” grade: A submission completing some of the requirements of the assignment. Parts A and B are mostly correct; the code for part C runs with some problems.
- “D” grade: A serious attempt at completing requirements for the assignment. Parts A and B run with serious problems; part C is not complete.
- “F” grade: Submission either represents very little work or cannot be assembled for testing.