Assignment 5: MIPS and SPIM!

Due by: Friday, April 6, 2018 10 PM
Collaboration: None
Grading: Packaging 10%, Style 10% (where applicable), Design 10% (where applicable), Correctness/Functionality 70% (where applicable)

Overview

The fifth assignment is **all** about hacking MIPS assembly code using the excellent SPIM simulator.

`http://spimsimulator.sourceforge.net/`

For two problems we give you a bunch of starter code, so for those all you have to do is fill in one or two subroutines in the way described below (and in the code itself, be sure to read it for all the gory details).

Problem 1: Long Arithmetic (20%)

The MIPS processer we play with using SPIM is a 32-bit processor so it can only do arithmetic on 32-bit quantities. For this problem you need to write two subroutines that can perform **64-bit** addition and subtraction.

You may want to think back to the similar problem we did for the 6502 a few weeks ago. Of course the MIPS processor doesn't have an instruction that takes a "carry bit" into account, so you'll have to figure out how to handle the carry (or borrow) some other way. **Include a *detailed* discussion of how you solved this problem, particularly the carry/borrow part, in your `README` file!**

Your job? Add the necessary subroutines to the file `sixtyfour.s` of course!

Problem 2: Unhexing Numbers (20%)

The starter code for Problem 1 includes a subroutine for converting a 32-bit unsigned integer to its hexadecimal (base 16) string representation. Write a function (in either C, C++, Java, or Python) that does the **exact** same thing using the **exact** same algorithm. (So you're basically translating back from MIPS assembly to a high-level programming language of your choice.) In other words, your function should take an integer as an argument and return (or fill in a passed buffer in the case of C) the corresponding string.

Your job? Put the function described above into a file `unhex.c` or `unhex.cc` or `Unhex.java` or `unhex.py`. Make sure you add a main program that uses your function to print out the hexadecimal representations of the decimal numbers 0, -1, 65535, and 65536 (each on a separate line).

Be sure to include instructions for how to build your program on a Linux system in your README file!

Problem 3: Bubbly Sorts (40%)

Sorting is always a popular programming problem, and now it's time to hack a sorting algorithm in MIPS assembly. Sadly, this is going to be a little less exciting than it was for the 6502 because our MIPS simulator doesn't have any cool graphics features.

Note that you **don't** have a choice of sorting algorithm, you **must** implement **bubble sort**! It is, however, up to you whether you implement a smart bubble sort that stops as soon as a single pass over the array didn't result in any swaps.

Note that swapping two adjacent array elements should also be a subroutine, and that subroutine should be called from within the sorting subroutine! Also please make sure that you access the array only through the subroutine's parameters and not directly as a global!

Your job? Add the necessary subroutines to the file `bubble.s` of course!

Problem 4: How many bits in a word? (20%)

There is no starter code for this problem. Write a MIPS program `popcount.s` that accepts a single unsigned integer in hexadecimal notation as a command line argument. Your program is supposed to print out the number of set bits in that integer in decimal. Here are a few examples:

```
$ spim -file popcount.s 0
0
$ spim -file popcount.s 1
1
$ spim -file popcount.s 2
1
$ spim -file popcount.s 3
2
$ spim -file popcount.s 10
1
$ spim -file popcount.s ff
8
$ spim -file popcount.s ffffffff
32
```

You're on your own for all the details, everything from how to access the command line arguments to figuring out the number of set bits in a 32-bit word. We will, however, take note of the algorithm you used. There are **many** algorithms (quite a few more than

you may realize) and you should pick one that is (a) as efficient as possible but (b) simple enough for you to completely understand. So just googling for "fast population count" is not enough, you **must explain** how your algorithm works **in your own words** in the `README` file. **And make sure you break your program into sensible subroutines!**

Deliverables

Please turn in to Gradescope a `gzip` compressed tarball of your assignment; the filename should be `HW1_<your_name>.tar.gz` with *name* replaced by the first part of the email address you used to register on Piazza (so I would use `HW1_kmemon1.tar.gz`).

Include a `README` file that briefly explains what your programs do and contains any other notes you want us to check out before grading.

Grading

For reference, here is a short explanation of the grading criteria; some of the criteria don't apply to all problems, and not all of the criteria are used on all assignments.

Packaging refers to the proper organization of the stuff you hand in, following the guidelines for Deliverables above.

Style refers to either to programming style if we are talking about a programming problem, or to the clarity and readability of your solution for a written problem.

If your programs cannot be built you will get no points whatsoever for programming problems. If your programs fail miserably even once, i.e. terminate with an exception of any kind or dump core, we will take off 10%; do proper error checking and handling! Finally, make sure to include your name and email address in *every* file you turn in (well, if there *is* a way to include that information, otherwise you can leave it out)!

Acknowledgement

This assignment was originally designed by Peter Fröhlich.