LSU EE 4720 Homework 1 Due: 6 February 2023

Problem 0: Follow the instructions for class account setup and for homework workflow in `https://www.ece.lsu.edu/ee4720/proc.html`. Review the comments in `hw01.s` and look for the areas labeled "Problem 1".

Those who want to start before getting to the lab can find the assembler for the entire assignment at `https://www.ece.lsu.edu/ee4720/2023/hw01.s.html`. For MIPS references see the course references page,
`https://www.ece.lsu.edu/ee4720/reference.html`. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in
`https://www.ece.lsu.edu/ee4720/proc.html`.

Using LSU version of SPIM

This assignment requires a modified version of the SPIM simulator originally developed by James Larus. Instructions for using this simulator appear on the course procedures page. When running SPIM check the LSU version date, there should be a line reading `LSU Version Date: 2022-01-31`. Make sure that the date is there and is no earlier than 31 January 2022. (The date will appear on the console output near the top when run non-graphically, and in the lowermost window pane when run graphically.)

Debugging

To facilitate debugging the code can be run so that the simulator emits a trace of executed instructions, plus an indication of changed register values.

The best way to get a trace is to run the code non-graphically. To do so load the code into an Emacs buffer in a properly set up account. Press `Ctrl`-`F9` to start the simulator non-graphically. That should pop up a window showing a simulator banner followed by a prompt:

```
SPIM Version 6.3.1 lsu of   9 November 2001, 17:34:35 CST
LSU Version Date: 2022-01-31
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Includes LSU modifications.
File loaded.
  Type "run" to run normally.
  Type "step 100" to execute next 100 instructions with tracing.
  Type "help" for more help.
(spim)
```

To see a trace of instructions enter `step` followed by the number of instructions, say `step 100`. This will execute next 100 instructions but will only trace instructions in the `lookup` routine (as of this writing). Suppose that the `lookup` routine starts with the following code:

```
lookup:
        addi $v0, $0, -1
START_WORD:
        addi $t0, $a0, 0
        addi $v0, $v0, 1
```

Then a trace of execution would produce the following:

```
(spim) step 100
[0x004000cc]    0x4080b000  mtc0 $0, $22                    ; 278: mtc0 $0, $22
[0x00400118]    0x0c100000  jal 0x00400000 [lookup]         ; 299: jal lookup
# Change in $31 ($ra)          0 -> 0x400120    Decimal: 0 -> 4194592
[0x0040011c]    0x40154800  mfc0 $21, $9                    ; 300: mfc0 $s5, $9
# Change in $21 ($s5)          0 -> 0x14        Decimal: 0 -> 20
[0x00400000]    0x2002ffff  addi $2, $0, -1                 ; 16: addi $v0, $0, -1
[0x00400004]    0x20880000  addi $8, $4, 0                  ; 18: addi $t0, $a0, 0
# Change in $8  ($t0)          0 -> 0x1001024f  Decimal: 0 -> 268501583
[0x00400008]    0x20420001  addi $2, $2, 1                  ; 19: addi $v0, $v0, 1
```

Each line starting with square brackets shows the execution of an instruction. The address of the instruction is shown inside the square brackets. After the square brackets the instruction is shown in three different forms. First encoded, shown in hexadecimal. Then a disassembled form (which is based on the encoded instruction). Finally, after the semicolon the instruction is shown as it appears in the assembler file. Immediately after the semicolon is a line number.

The lines that start with a `#` show register values that change. The values are shown both in hexadecimal and decimal. In the example above the first three instructions are from the testbench, the fourth instruction shown, at address `0x400000`, is the first instruction of `lookup`.

Problem 1: MIPS routine `lookup` is called with three arguments: `$a0` holds the address of a string, called the *lookup word*, `$a1` holds the address of a word table, `$a2` holds the address of a word length table. Complete `lookup` so that when it returns `$v0` is set to the index of the lookup word in the word table (as explained below) or to -1 if the lookup word is not in the word table.

Register `$a1` holds the address of a word table. (Look for `table_words:` in the file.) The words are stored one after the other in alphabetical order, but without even a null separating them. The first three words are aah, aardvark, able. They are stored in memory as `aahaardvarkable`. Register `$a2` holds the address of a length table. The first entry is the length of the first word, the second entry is the length of the second word, and so on. The lengths are stored as 1-byte unsigned integers. The first three values stored in the length table are 3, 8, and 4. Note that without the length table it would not be possible to reliably identify the words in the word table.

The *index* of a word is its position in the word table, with the first word, `aah`, having index 0. (The second word, `aardvark`, has index 1, and so on.)

(The word table, word size table, and the lookup words can be found in `hw01.s`. Inspecting these might help in understanding the problem. Search for `table_words:` to find the word table. The other tables are below.)

If `lookup` is called with `$a0` pointing to `able` then when it returns `$v0` should hold a 2. If `$a0` points to `abacus` then `$v0` should be set to -1. (Since the word table is in alphabetical order `abacus` can't be in the word list even if all we know is that the first three words are aah, aardvark, able.)

The testbench will run `lookup` on several lookup words and report the number of instructions needed for each lookup word and whether it was correct. Here is the output for the instructor's solution:

```
a                   Num Insn:   227   Index:  -1 -- Correct
aah                 Num Insn:    43   Index:   0 -- Correct
able                Num Insn:    62   Index:   2 -- Correct
i                   Num Insn:   230   Index:  41 -- Correct
```

```
blank               Num Insn:   133   Index:  -1 -- Correct
counselor           Num Insn:   182   Index:  -1 -- Correct
county              Num Insn:   217   Index:  -1 -- Correct
fish                Num Insn:   328   Index:  -1 -- Correct
gram                Num Insn:   328   Index:  -1 -- Correct
palindromic         Num Insn:   359   Index:  -1 -- Correct
zymurgy             Num Insn:   361   Index:  -1 -- Correct
bibliographical     Num Insn:   216   Index:  13 -- Correct
bibliographically   Num Insn:   239   Index:  14 -- Correct
cross               Num Insn:   204   Index:  24 -- Correct
zydeco              Num Insn:   365   Index:  55 -- Correct
zygotes             Num Insn:   379   Index:  57 -- Correct
TOTALS:             Num Insn:  3873   Tests:  16    Errors:  0
```

First, complete `lookup` so that it is correct (shows zero errors). Do not use pseudoinstructions except for `nop`. Comments in the code show other rules and indicate which registers can be modified. Next, optimize it to reduce the number of executed instructions. There are many ways to do this.

To make the solution fast take advantage of the fact that the word table is in alphabetical order and that the length table provides the length of each word in the table.

Your solution **should not** rely on extra storage that's set up by an initialization call. For example, if one computes an offset table from the length table then a binary search is possible. That's interesting, but not allowed in this problem. Another possibility is to compute a hash table. (See 2019 Homework 1.) Using a hash table is interesting but not allowed in this assignment either.

Finally, write clear code and add comments appropriate for an expert MIPS programmer. For example, don't explain things that can easily be figured out.

For examples of MIPS program see past Homework 1 assignments in this class.