LSU EE 4720 Homework 1 Due: ~~XX X~~2 14 February 2024

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of MIPS or assembler syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out assembly language resources for help on MIPS, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for MIPS instructions and the SPIM simulator, and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) Students are expected to experiment to learn how MIPS instructions work, and how to code assembly language sequences. Experimentation might be done on old homework assignments or the simple code samples provided in `/home/faculty/koppel/pub/ee4720/hw/practice`. Students are also expected to learn what error messages mean by consulting documentation and by asking others (including Dr. Koppelman), and also to develop debugging skills. It is the students' responsibility to resolve frustrations and roadblocks quickly. (Just ask for help!)

This assignment cannot be solved by blindly pasting together code fragments found in class notes or past assignments. Solving the assignment is a multi-step learning processes that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

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 area 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/ee4720/2024/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: 2024-02-04`. Make sure that the date is there and is no earlier than 2 February 2024. (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: 2024-02-04
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.
```

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 assignment routine (when running this homework assignment). To illustrate stepping consider the `lookup` routine from 2023 Homework 1. 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`.

Homework Background

MIPS routine `aadd` is to add two numbers. This would be trivial if the numbers were 32-bit integers, but the numbers to add are decimal integers encoded as ASCII strings. Routine `aadd` is called with four arguments, in registers `a0-a3`, and has one return value to be put in `v0`. Register `a1` and `a2` are set to the address of strings. Each string consists of only digits (ASCII 48-58) and a null (zero) termination. There are no `+` signs, `-` signs, no decimal point, and no whitespace. Call the two strings *operand 1* and *operand 2*. For example, the string for operand 1 might be `"4720"`, and operand 2 might be `"3002"`. The sum should be string `"7722"`.

Register `a0` holds the address of the *output buffer*, an area of memory where the sum is to be written. The sum must be a C-style ASCII string, and must be written into the output buffer. Register `v0` must be set to the address where the sum is written. The value in `v0` must be at least `a0` (its value when `aadd` is called) but less than `a0+20`. Those who are following this well may wonder why `v0` can't be set to exactly `a0` (the start of the output buffer). It can, but to do so one would need to know how many digits there were in the sum, which won't be known until the sum is computed and written to memory in which case the sum would have to be moved so that it started at the beginning of the output buffer. So to avoid the hassle of moving the sum, just put the actual starting address in `v0`.

For your solving convenience the length of the two operands, in characters, are in register `a3`. The length of operand 1 is in the upper 16 bits and the length of operand 2 is in the lower 16 bits. For your further convenience, those two lengths are extracted in the unmodified assignment file. Here is an abridged version of the start of `aadd`:

`aadd`:

```
        ## Register Usage
        #
        # CALL VALUES
        #  $a0: Address of the output buffer in which to write the result.
        #  $a1: Address of operand 1, an ASCII string.
        #  $a2: Address of operand 2, an ASCII string.
        #  $a3: Length of operands:
        #       Operand 1 length: bits 31:16
        #       Operand 2 length: bits 15:0
        #
        # RETURN VALUE
        #  $v0: Address of start of the result. Must be within output buffer.

        srl $t8, $a3, 16         # Length of operand 1
        andi $t9, $a3, 0xffff    # Length of operand 2

        # Put solution here.

        jr $ra
        nop
```

If you're one of those people that understand fully what the code is supposed to do and are eager to get started, great! For the rest, keep reading.

When the code in `hw01.s` is run is starts with a testbench that calls `aadd` multiple times. The call arguments start out easy and get more difficult. For example, for the first call $12321 + 0$ is to be computed.

The testbench will print one line for each call, and at the end will print a tally of results. For example, on a correctly solved assignment:

```
SPIM Version 6.3.1 lsu of   9 November 2001, 17:34:35 CST
LSU Version Date: 2024-02-04
[snip]
(spim) run
Num Insn:    65  Correct:  12321 + 0 = 12321
Num Insn:    29  Correct:  1 + 1 = 2
Num Insn:    56  Correct:  9007 + 2 = 9009
Num Insn:    57  Correct:  5107 + 8 = 5115
Num Insn:    59  Correct:  3 + 9002 = 9005
Num Insn:    55  Correct:  789 + 67 = 856
Num Insn:    58  Correct:  67 + 789 = 856
Num Insn:    83  Correct:  999999 + 1 = 1000000
Num Insn:   131  Correct:  765432 + 12345678 = 13111110
Num Insn:   260  Correct:  184737252196092 + 8383352872579977 = 8568090124776069
TOTALS:   Correct:  10    Wrong:  0
```

The value labeled `Num Insn:` is the number of instructions executed by your routine. Less is better, but for this assignment that's not a priority. The values above are my first correct solution without much effort put into tuning.

The operands are chosen so that they start out easier and get harder. In the first set one just adds zero. In the second set both operands are the same size, just one character, and there is no carry. There is no carry in the third set either, and the sum is palendromic, so it is correct even if it's backward.

The value to the left of the `=` is the output of `aadd` if register `v0>=a0` and `v0<=a0+20`, where `a0` is the value of register `a0` when `aadd` first called. Above the value of `v0` is fine and the values are correct.

On an unmodified assignment, where `aadd` does not modify `v0`, the output will look like:

```
(spim) run
Num Insn:     5  Wrong  :  12321 + 0 = ** $v0 too low **
Num Insn:     5  Wrong  :  1 + 1 = ** $v0 too low **
Num Insn:     5  Wrong  :  9007 + 2 = ** $v0 too low **
Num Insn:     5  Wrong  :  5107 + 8 = ** $v0 too low **
Num Insn:     5  Wrong  :  3 + 9002 = ** $v0 too low **
Num Insn:     5  Wrong  :  789 + 67 = ** $v0 too low **
Num Insn:     5  Wrong  :  67 + 789 = ** $v0 too low **
Num Insn:     5  Wrong  :  999999 + 1 = ** $v0 too low **
Num Insn:     5  Wrong  :  765432 + 12345678 = ** $v0 too low **
Num Insn:     5  Wrong  :  184737252196092 + 8383352872579977 = ** $v0 too low **
TOTALS:   Correct:   0    Wrong: 10
(spim)
```

For the output above `v0` is out of range (since `aadd` did not try to set it). If all `aadd` does is copy the first operand to the output buffer (the storage at `a0`) then the testbench output would be:

```
(spim) run
Num Insn:    36  Correct:  12321 + 0 = 12321
Num Insn:    16  Wrong  :  1 + 1 = 1
Num Insn:    31  Wrong  :  9007 + 2 = 9007
```

```
Num Insn:    31  Wrong  :  5107 + 8 = 5107
Num Insn:    16  Wrong  :  3 + 9002 = 3
Num Insn:    26  Wrong  :  789 + 67 = 789
Num Insn:    21  Wrong  :  67 + 789 = 67
Num Insn:    41  Wrong  :  999999 + 1 = 999999
Num Insn:    41  Wrong  :  765432 + 12345678 = 765432
Num Insn:    86  Wrong  :  184737252196092 + 8383352872579977 = 184737252196092
TOTALS:   Correct:   1    Wrong:  9
```

There are no `v0` errors, but the output is only correct when operand 2 is zero. But at least we know we can copy a string!

Helpful Examples

For your convenience two sample MIPS programs are included in the assignment directory, `strlen.s` and `hex-string.s`. The `strlen.s` contains the string length we did in class. Look at it if you are rusty. Then look at `hex-string.s`, which contains a routine that writes an ASCII string corresponding to the hexadecimal representation of a value in the call register. Like this assignment, in hex-string an ASCII string is written. Of course, there are many differences between this assignment and hex string. For more examples look in the practice directory and at Homework 1 assignments from earlier semesters.

Problem 1: *This problem is optional. It's here to help people get started.* Modify `aadd` so that operand 1 is copied to the output buffer. That is, the string at `a1` should be copied to the memory at `a0`, and `v0` should be set to the starting address of the output buffer (the original value of `a0`). If this is solved correctly the first call will be correct (because the second operand is zero).

Problem 2: Complete `aadd` so that it writes the sum to the output buffer and sets `v0` to the starting address of the sum. If Problem 1 has been solved remove, comment out, or jump over the Problem 1 solution since a correct solution to Problem 2 makes Problem 1 unnecessary.

To solve this problem one must start at the least significant digit of each operand (the end of the string), and then move backward. To make that easier the length of the operands are provided. A good way to start is to only add the least significant digits. Then use a loop to iterate toward the more significant digits. Don't forget to propagate carries and that the strings can be different sizes.

Feel free to insert new numbers to add, perhaps as to help debugging. Search for `table_numbers` and insert new sets. Below, $9 + 10$ is inserted. The third number is the sum, which the testbench trusts to be correct.

```
        .data
table_numbers:
        .asciiz "9", "10", "19"  # My test numbers.
        .asciiz "12321", "0", "12321"
        .asciiz "1", "1", "2"
        .asciiz "9007", "2", "9009"
        .asciiz "5107", "8", "5115"
```

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.