LSU EE 4720 **Homework 1** **Due: 28 January 2026**

Solution *Formatted 16:10, 10 February 2026*

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 each student's duty to him or herself 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/ee4720/2026/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: 2025-02-04`. Make sure that the date is there and is no earlier than 4 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 mostly 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`.

Problem 1: When completed MIPS assembly language routine `reverse_lists` will copy the C-style string starting at `a0`, call it the *input string*, to `a1`, the *output string*, with items in lists (described below) reversed but all other characters copied unchanged. Register `a2` points to storage (1024 bytes) that can be used to solve the problem. There are no values to return in registers. See the solution checklist in `reverse_lists` for additional requirements, including registers that can be modified.

Consider the following excerpt from a sample run:

```
In : Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Out: Total ice between a (glaze,lite), (amounts,higher,locally) possible.
```

The line starting `In :` shows the input string which contains two lists. A list always starts with an open parenthesis, '(', and ends with a close parenthesis, ')'. It can contain zero or more items, items are separated by commas. An item is a string of characters except parentheses, a comma, and a null. This means lists **are not** nested, so don't worry about `Never an input (a,b,(c,d,e),f,g).` Also assume that each open parenthesis is followed by a close parenthesis.

In the example above the first list has two items and the second list has three. The output string is shown on the line starting with `Out:`. Notice that the lists in the output string have the same items as the input string, but their orders are reversed.

The testbench will run `reverse_lists` on several input strings. For each one it will show the input and output strings, as shown above, and it will also show string length, x's at error positions, and computation rate.

Here is sample output from a correct solution:

```
---- Input number 2, length 68 characters.
In : Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Out: Total ice between a (glaze,lite), (amounts,higher,locally) possible.
Err:
---- Computed with 629 instructions at 0.108 char/insn or 9.2 insn/char.
```

For this case `reverse_lists` executed 629 instructions, which works out to 9.2 instructions for each character of the input. The fewer instructions executed the better.

The unmodified routine has some getting-started code. Registers `t3`, `t4`, and `t5` are set to the ASCII values for the open parenthesis, close parenthesis, and comma. Also, the first character of the input string is copied to the output string. After that, the routine returns. Here is the routine with some comments removed:

```
reverse_lists:
        #  CALL VALUES
        #   $a0:  Address of string to read.
        #   $a1:  Address of string to write.
        #   $a2:  Address of scratch storage.

        # Helpful Starter Code
        ori $t3, $0, 40    # Open paren.
        ori $t4, $0, 41    # Close paren.
        ori $t5, $0, 44    # Comma

        lb $t0, 0($a0)     # Read first character of input string.
        sb $t0, 0($a1)     # Write first character of output string.
```

```
all_done:
        jal $ra
        nop
```

The output for an unmodified assignment is:

```
---- Input number 2, length 68 characters.
In : Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Out: T
Err:  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
---- Computed with 7 instructions at 9.714 char/insn or 0.1 insn/char.
```

Notice that the output, except for the first character, is empty which is why there are lots of x's. If the code just copies the input string without modification the output would be:

```
---- Input number 2, length 68 characters.
In : Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Out: Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Err:                       xxxxxx xx     xxxxxxx        xxxxxxx
---- Computed with 350 instructions at 0.194 char/insn or 5.1 insn/char.
```

Notice that `higher` in the second list above is correct because the list has three items and the first and last are the same length.

Four test strings are used (as of this writing). Input 1 does not have any lists. That will help judge how fast your code scans for parentheses. Input 2 is the example shown above, mixing lists and text. Input 3 includs a zero-length list, and zero-length items. Finally, Input 4 consists of one large list, so you can gauge the performance of your item reversing code. Their outputs are:

```
---- Input number 1, length 36 characters.
In : This string has no lists to reverse.
Out: This string has no lists to reverse.
Err:
---- Computed with 226 instructions at 0.159 char/insn or 6.3 insn/char.

---- Input number 2, length 68 characters.
In : Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Out: Total ice between a (glaze,lite), (amounts,higher,locally) possible.
Err:
---- Computed with 629 instructions at 0.108 char/insn or 9.2 insn/char.

---- Input number 3, length 49 characters.
In : Corner cases: (), (one), (a,,see), (,b,) (,d,e,).
Out: Corner cases: (), (one), (see,,a), (,b,) (,e,d,).
Err:
---- Computed with 513 instructions at 0.096 char/insn or 10.5 insn/char.

---- Input number 4, length 48 characters.
In : (1,two,three,four,five,six,seven,eight,nine,ten)
Out: (ten,nine,eight,seven,six,five,four,three,two,1)
Err:
---- Computed with 606 instructions at 0.079 char/insn or 12.6 insn/char.
```

Those who are rusty on MIPS should first practice some sample problems. The assignment directory includes three sample problems, `strlen`, `hex-string`, and `histo`. The files with names containing `-live` are unsolved, such as `strlen-live.s`, use those to practice. Those without `-live` are complete.

Those who are not sure where to start might consider these intermediate goals, which might get some partial credit:

Intermediate goal 1: Copy the string unchanged. That is, copy the string at `a0` to `a1` without making any changes.

Intermediate goal 2: Reverse the string between each pair of parentheses (so that items would be backward). For example, for input `between a (lite,glaze)` generate output `between a (ezalg,etil)`.

Intermediate goal 3: Copy the string unchanged, except within a list only show the commas. For example, for input `between a (lite,glaze)` generate output `between a (  ,  )`.

There are many ways that this problem can be solved. In some of those ways a table of information about the items is constructed and then used. That table might contain the length of each item, it might contain the address of the start of each item, or it might contain the distance (offset) of each item from either the start of the list or the end of the list. The address of storage you can use for such a table (or tables) is in register `a2`. There are 1024 bytes of storage. The storage can be read or written like any other memory, using `sb`, `lb`, `sh`, `lh`, `lw`, etc. See the array access lecture slides, `https://www.ece.lsu.edu/ee4720/2026/larray.s.html`, for examples of how to read and write memory used for arrays (which can be thought of as tables of numbers). The `histo.s` program, one of the examples included in directory `hw01`, reads and writes a table (the histogram). The storage used for the histogram table is in register `a1`.

The solution has been placed in the directory where the assignment was found. An HTMLized version is on the Web at `https://www.ece.lsu.edu/ee4720/2026/hw01-sol.s.html` and is included in this assignment.

This solution is better than the one used to generate the sample output above. Unlike the code used to generate the output above, the solution below assumes that every open parenthesis is followed by a closing parenthesis.

The code was written to scan the input list just twice (making two *passes*). The first time it writes item start addresses to the table in scratch memory. The second time it copies items from the input string to the output string.

Care was taken so that loops that scan or copy text used few instructions. This was done by filling delay slots and avoiding useful-sounding but unnecessary code, such as code for counting the number of characters copied. (Addresses could be used instead, see the solution.)

Most submissions were slower than they needed to be by leaving delay slots unfilled and using item-length counters. I'd prefer that the effort spent creatively renaming branch labels and re-writing comments instead be spent writing code from scratch after looking at a possible solution.

When grading I focused on the speed of the first string, which did not have a list, and the last string, which was one long list. The fastest took 6.2 instruction per character for the first string and 11.0 for the last. My original solution took 6.3 and 12.6 insn and the version that assumes matched parenthesis took 6.3 and 10.1, so the best solution beats me on the no-list case and does very well on the all-list case. Also, it's better than mine on the other two test cases. The second-best solution uses 6.3 and 13.3 insn/char and the median solution (counting only those that are mostly correct) uses 10.8 and 19.7 insn/char.

The solution assembler and output appear below.

```
reverse_lists:
        ## Register Usage
        #
```

```
#  CALL VALUES
#   $a0:  Address of string to read.
#   $a1:  Address of string to write.
#   $a2:  Address of scratch storage.
#
#  RETURN VALUES
#   No return registers.
#   Instead write memory starting at $a1 with string at $a0
#   with its lists reversed.
#
ori $t3, $0, 40    # Open paren.
ori $t4, $0, 41    # Close paren.
ori $t5, $0, 44    # Comma

## SOLUTION
#
# Plan:
#
#  Use an approach that scans the input list just twice, once
#  to find item start addresses, and once to copy the items.
#
#  Use the table to store the start address of each item. The
#  last element of the table is the address of the character
#  after the last parenthesis.
#
#  Identify where the code spends most of its time and take
#  care to reduce the number of instructions in those places.
#  Search for "important" to find these places.
#
#  Achieve performance by filling delay slots, creatively if
#  necessary. Also avoid unnecessary code such as maintaining
#  an item length counter, when addresses can be used for the
#  same purpose.
#
#
# Code Outline:
#
#  First, copy input to output until a open paren found.
#  -- See label find_next_open_paren.
#
#  Scan until a close paren is found. While scanning store
#  the address of the start of each item in the table.
#  -- See label find_item_end.
#
#  When the closing paren is found, write the address of the
#  character after the close paren in the table.
#
#  Iterate over the table backward:
#
#    Retrieve two adjacent table elements, the first is an
#    item start address, the second (the start of the *next*
```

```
        #    item) is used to compute the item's end address.
        #    -- See label copy_item_start.
        #
        #    Using these values copy the element from the input to the
        #    output.
        #    -- See label copy_item.
        #

find_next_open_paren_prep:
        # Copy start address of scratch storage to t6.
        ori $t6, $a2, 0    # Scratch storage.

find_next_open_paren:
        # Copy input to output until an open parenthesis is found.
        #
        # This loop is important, so it uses as few instructions as is
        # reasonably possible.
        #
        lb $t0, 0($a0)
        sb $t0, 0($a1)
        beq $t0, $t3, found_next_open_paren
        addi $a0, $a0, 1
        bne $t0, $0, find_next_open_paren
        addi $a1, $a1, 1
        #
        # Time: 6 insn / char.

        # The input string has been completely copied. Time
        # to return.

        jr $ra
        nop

found_next_open_paren:
        addi $a1, $a1, 1   # Increment output string address.
found_comma:
        sw $a0, 0($t6)     # Store element start address in table.
        addi $t6, $t6, 4

find_item_end_start:
        lb $t0, 0($a0)

find_item_end:
        # Find the end of the current item.
        #
        # This loop is also important. It's reduced to four
        # instructions by putting the lb at the end, rather than as
        # the first instruction of the loop. (That's way there's a lb
        # instruction, at find_item_end_start, *before* the loop is
        # entered.)
        #
```

```
        beq $t0, $t5, found_comma
        addi $a0, $a0, 1
        bne $t0, $t4, find_item_end
        lb $t0, 0($a0)
        #
        # Time: 4 insn / char

        # Found closing parenthesis. Write the table with the address
        # of character after the parenthesis. Note that all other
        # entries in the table are the start address of items.
        #
        sw $a0, 0($t6)

copy_item_start:
        # Prepare for copying an item by retrieving its start address
        # and the next item's start address (or the end-of-list address).
        #
        lw $t1, -4($t6)    # This item's start address.
        lw $t2, 0($t6)     # Start address of next item or end of list.

        addi $t2, $t2, -1  # Compute the address at which to stop copying.
        beq $t1, $t2, item_copied  # Check whether there's nothing to do.
        addi $t6, $t6, -4  # Get ready for the next item.

copy_item:
        # Copy item from input to output.
        #
        # This loop is also important.
        #
        lb $t0, 0($t1)
        sb $t0, 0($a1)
        addi $t1, $t1, 1
        bne $t1, $t2, copy_item
        addi $a1, $a1, 1
        #
        # Time: 5 insn / char

item_copied:
        sb $t5, 0($a1)     # Provisionally write comma. Needed if bne taken.
        bne $t6, $a2 copy_item_start
        addi $a1, $a1, 1

        j find_next_open_paren_prep
        sb $t4, -1($a1)    # Write closing parenthesis.
```

The output of this code appears below:

```
---- Input number 1, length 36 characters.
In : This string has no lists to reverse.
Out: This string has no lists to reverse.
```

```
Err:
---- Computed with 228 instructions at 0.158 char/insn or 6.3 insn/char.

---- Input number 2, length 68 characters.
In : Total ice between a (lite,glaze), (locally,higher,amounts) possible.
Out: Total ice between a (glaze,lite), (amounts,higher,locally) possible.
Err:
---- Computed with 552 instructions at 0.123 char/insn or 8.1 insn/char.

---- Input number 3, length 49 characters.
In : Corner cases: (), (one), (a,,see), (,b,) (,d,e,).
Out: Corner cases: (), (one), (see,,a), (,b,) (,e,d,).
Err:
---- Computed with 445 instructions at 0.110 char/insn or 9.1 insn/char.

---- Input number 4, length 48 characters.
In : (1,two,three,four,five,six,seven,eight,nine,ten)
Out: (ten,nine,eight,seven,six,five,four,three,two,1)
Err:
---- Computed with 486 instructions at 0.099 char/insn or 10.1 insn/char.
```