LSU EE 4720 **Homework 1** **Due: 4 February 2025**

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/ee4720/2025/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 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

When completed MIPS assembly language routine `justify` will justify a string of text. The `justify` routine can be found in `hw01.s`. Also in that file is a test routine that calls `justify` and prints out the formatted text. To make the problem less tedious to solve the string of text provided to `justify` will not contain any line feeds (or carriage returns or the equivalent). In the version used as of this writing the text is 1028 characters long which might be possible to read on one of those ridiculously wide curved monitors. When solved correctly `justify` will add spaces and line feeds to make the text more readable. The input text starts:

```
We introduce our first-generation reasoning models, DeepSeek-R1-Zero and
```

That text is to be justified using left margins and text lengths provided to `justify` (to be explained below). Unlike conventional boring justification where the left margin and text length is the same for every line (except maybe for an initial indentation), `justify` can use a different left margin and text length for each line. The correctly justified text for a run of the homework code is:

```
Formatted text appears below.
                              We introduce
                         our first-generation
                    reasoning models, DeepSeek-R1-Zero
               and DeepSeek-R1. DeepSeek-R1-Zero, a model
          trained via large-scale reinforcement learning (RL)
     without supervised fine-tuning (SFT) as a preliminary step,
demonstrated remarkable performance on reasoning. With RL, DeepSeek-R1-Zero
     naturally emerged with numerous powerful and interesting reasoning
          behaviors. However, DeepSeek-R1-Zero encounters challenges
               such as endless repetition, poor readability,
                    and language mixing. To address
                         these issues and further
                              enhance reasoning
                              performance,
                         we introduce DeepSeek-R1,
                    which incorporates cold-start
               data before RL. DeepSeek-R1 achieves performance
          comparable to OpenAI-o1 across math, code, and reasoning
     tasks. To support the research community, we have open-sourced
DeepSeek-R1-Zero, DeepSeek-R1, and six dense models distilled from DeepSeek-R1
     based on Llama and Qwen. DeepSeek-R1-Distill-Qwen-32B outperforms
          OpenAI-o1-mini across various benchmarks, achieving
               new state-of-the-art results for dense models.
Formatted text appears above.
Input  string length 1028 characters.
Output string length 1425 characters.
Executed 7336 instructions at rate of 0.140 char/insn.
```

In addition to the justified text, the output above includes messages printed by the testbench. (The text is from the readme file in the DeepSeek-R1-Zero repo containing parameters distilled for a smaller model. See `https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-8B`.)

Routine `justify` is called with three arguments. Register `a0` is set to the address of the string to justify. Call it the *input string*. Register `a1` is set to the address where the justified string is to be written. Call it the *output string*. Register `a2` holds the address of the *line shape table*.

Each entry of the line shape table holds two bytes. The first byte indicates the left margin size. The second byte indicates the minimum length of the text on the line (not including the left margin).

The code below reads the first entry in the line shape table:

```
lb $t1, 0($a2)  # Left margin of first line.
lb $t2, 1($a2)  # Length of text of first line (not including margin)
```

Suppose `t1` is 30. (Which is what the testbench sets it to AoTW.) Then the left margin must be 30, meaning there should be 30 spaces before the text. The `justify` routine must start out writing 30 spaces beginning at the address in `a1` and then start copying the text from `a0`, which in the example above starts `We introduce`, to `a1+30`.

The second item in a shape entry is the text length, in register `t2` above. This is the *minimum* length of text after the left margin. For the example data `t1+t2 = 30 + 10 = 40`. At character position 40 on the first line is the letter `c` in `introduce`. The `justify` routine is to start the next word, `our`, on a new line. Character 40 is within the word `introduce` and so `justify` should continue copying text from `a0` to `a1` until a space is reached. It should then start a new line. That new line will start with `our` (after the new left margin.)

Let L denote the left margin (`t1=30` above) and W the margin (`t2=10` above) for a line. That line should start with L spaces. After the spaces, the line should have characters copied from `a0`. Copying continues until the line length is $L + W$ *and* a new word starts. *In real-world formatting routines* $L + W$ *would be the maximum length, not the minimum length as it is here.*

Each time a line is completed the next entry in the shape table should be read. There might be fewer entries in the shape table than there are lines of formatted text. When there are no more entries in the shape table the `justify` routine should start from the beginning of the shape table. The end of the shape table is marked by $L = 255$ and $W = 255$.

The text in `a0` has intentionally be kept simple. It does not contain line feeds or similar characters. The only whitespace is a space, and there is never (or shouldn't be) more than one consecutive space.

The testbench **does not** check for correctness. To verify correct line start positions when running non-graphically put the cursor of the first character in a line. The line and column (character) number is shown in the text editor status bar at the bottom.

Unsolved `justify` Routine Getting-Started Code

In the unsolved assignment the `justify` routine will copy two characters of the input string (the unformatted text) to the output string. When run it prints the word `We`. It also loads the first two entries in the Line Shape Table, but does not do anything with them. Here is an excerpt from that code, with many of the comments omitted:

```
        .text
justify:
        ## Register Usage
        #
        # CALL VALUES
        #  $a0: Address of start of text to justify.
        #  $a1: Starting address where justified text is to be written.
        #  $a2: Line Shape Table.
        #       Two bytes per entry.
        #          First byte is left margin.
        #          Second byte is length of text.

        # Load the first entry of the Line Shape Table.
        #
        lb $t1, 0($a2)  # Left margin of first line.
        lb $t2, 1($a2)  # Length of text of first line (not including margin)
        #
        # Load the second entry of the Line Shape Table.
        #
        lb $t3, 2($a2)  # Left margin of second line.
        lb $t4, 3($a2)  # Length of text of second line (not including margin)

        # Copy first two characters. (Ignoring left margin.)
        #
        lb $t0, 0($a0)
        sb $t0, 0($a1)
        lb $t0, 1($a0)
        sb $t0, 1($a1)

        jr $ra
        nop
```

One way to get started on the solution would be to copy more than two characters by using a loop. Then, try inserting a line feed every 64 characters, perhaps by using a loop nest with the inner loop iterating 64 times. Next, try inserting a bunch of spaces at the beginning of each line. Keep adding functionality until the problem is solved.

Testbench Output

The test program prints information that might be helpful in getting the code working and improving performance. The last three lines of output (if the code ran to completion) will be something like:

```
Input  string length 1028 characters.
Output string length 1425 characters.
```

```
Executed 7336 instructions at rate of 0.140 char/insn.
```

The input string length reported above should ordinarily not change. It is the length of the input to `justify` provided by the testbench. Search for `tb_text_start` to find the string. If it helps, one can temporarily change the string at `tb_text_start` to facilitate the solution. The length of the output string should be longer than the input string due to the left margin.

The last line shows the number of instructions executed and the execution rate. A goal of this assignment is to minimize the number of instructions executed, so the lower both numbers the better. The execution rate is the number of input characters divided by the number of instructions. In the example above that works out to about 7 instructions for each input character.

Helpful Examples

For your convenience three sample MIPS programs are included write in the assignment directory, `strlen.s`, `2022-hw01.s` and `2022-hw01-sol.s`. The `strlen.s` contains the string length we did in class. Look at it if you are rusty. In 2022 Homework 1 a fast string length routine was to be written. This might help with writing the left margin (but not copying the text). 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. If you've solved this problem but then have gone on to the following problems you can delete or comment out the code.* Modify `justify` so that it copies the string at `a0` to `a1` breaking lines so that they are 64 characters long, even if that means breaking a line in the middle of a word.

Problem 2: Complete `justify` so that it justifies text as described above, following the restrictions given in the routine, such as which registers to use. In the unmodified file `justify` copies two characters and loads the first entry in the shape table. Be sure to remove this getting-started code.

The first challenge in this problem is to get the solution to work. The second one is to make if fast. For full credit the code writing the left margin should use `sw` instructions where possible, reducing the number of instructions needed to write the left margin.

Alignment restrictions will make it difficult (but not impossible) to use `lw` and `sw` for copying text, and so full credit will be awarded to solutions that use `lb` and `sb` for copying text.