LSU EE 4720 Homework 1 Solution Due: 4 February 2022

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” and “Problem 2.”

Those who want to start before getting to the lab can find the assembler for the entire assignment at `https://www.ece.lsu.edu/ee4720/2022/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`.

This Assignment

In class as MIPS review we wrote a routine, `strlen`, to find the length of a C string. In our completed routine (shown below) the main loop consisted of three instructions, and would load one character per iteration. Therefore at best it could run at the rate of $\frac{1}{3}$ characters per instructions.

```
strlen:
        # Register Usage
        # $a0: Address of first character of string.
        # $v0: Return value, the length of the string.
        addi $v0, $a0, 1        # Set aside a copy of the string start + 1.
LOOP:
        lbu $t0, 0($a0)         # Load next character in string into $t0
        bne $t0, $0, LOOP       # If it's not zero, continue
        addi $a0, $a0, 1        # Increment address. (Note: Delay slot insn.)

        jr $ra
        sub $v0, $a0, $v0
```

Can we do better? Since the main loop only consists of three instructions there is little that can be done to make it shorter, at least using MIPS I instructions. Notice that a character is loaded using `lbu` (load byte unsigned). Suppose instead a `lw` (load word) were used. Then four characters would be loaded. If our loop body contained 12 instructions (including the `lw`) then it would execute at the same rate as our original `strlen` because it would operate on 4 characters per 12 instructions or at the rate of $\frac{1}{3}$ characters per instruction. If we could somehow check for a null with fewer than 12 instructions our new code would be faster.

In Problem 1 such a string length routine is to be completed. It is assumed that most students' MIPS skills are rusty so the starting point is code using a `lhu` instruction. In the solution to Problem 1 I attained a rate of 0.392 char/insn, not much better than .329 attained by our original routine.

In Problem 2 the `strlen` routine is to be written using additional non MIPS-I instructions. These include `orc.b` from a RISC-V extension, and `clz` and `clo` from MIPS32 (based on their r6 versions). Using these instructions my solution achieves 0.942 chars per insn.

Test Routine

The code for this assignment includes a test routine that runs three string length routines: the routines to be written for Problems 1 and 2, and the string length routine written in class (called `strlen_ref` here). Each routine is run on several strings, including all lengths from 0 to 5, plus strings of length 23 and 196. The shorter-length strings are there to make sure that the routines are correct and to check how fast they are on short strings. The longest string is there to test performance. The performance numbers from the previous section are based on the longest string.

Here is the output from the unmodified assignment:

```
** Starting Test of Routine "strlen_p1  (Problem 1 - Bit Ops)" **
String  1: Length   1 is correct. Took  10 insn or 0.100 char/insn
String  2: Length   2 is correct. Took  13 insn or 0.154 char/insn
String  3: Length   3 is correct. Took  16 insn or 0.188 char/insn
String  4: Length   4 is correct. Took  19 insn or 0.211 char/insn
String  5: Length   5 is correct. Took  22 insn or 0.227 char/insn
String  6: Length   0 is correct. Took   7 insn or 0.000 char/insn
String  7: Length  23 is correct. Took  76 insn or 0.303 char/insn
String  8: Length 196 is correct. Took 595 insn or 0.329 char/insn

** Starting Test of Routine "strlen_p2  (Problem 2 - RISC V orc insn)" **
String  1: Length   1 is correct. Took  11 insn or 0.091 char/insn
String  2: Length   2 is correct. Took  15 insn or 0.133 char/insn
String  3: Length   3 is correct. Took  19 insn or 0.158 char/insn
String  4: Length   4 is correct. Took  23 insn or 0.174 char/insn
String  5: Length   5 is correct. Took  27 insn or 0.185 char/insn
String  6: Length   0 is correct. Took   7 insn or 0.000 char/insn
String  7: Length  23 is correct. Took  99 insn or 0.232 char/insn
String  8: Length 196 is correct. Took 791 insn or 0.248 char/insn

** Starting Test of Routine "strlen_ref (Simple strlen routine.)" **
String  1: Length   1 is correct. Took   9 insn or 0.111 char/insn
String  2: Length   2 is correct. Took  12 insn or 0.167 char/insn
String  3: Length   3 is correct. Took  15 insn or 0.200 char/insn
String  4: Length   4 is correct. Took  18 insn or 0.222 char/insn
String  5: Length   5 is correct. Took  21 insn or 0.238 char/insn
String  6: Length   0 is correct. Took   6 insn or 0.000 char/insn
String  7: Length  23 is correct. Took  75 insn or 0.307 char/insn
String  8: Length 196 is correct. Took 594 insn or 0.330 char/insn
```

To see all of this output when running graphically it might be necessary to make the pop-up window larger. It is possible to scroll the text in the pop-up window by focusing the window and using the arrow keys.

Each line shows the result from one string. The length of the string is shown, as well as the number of instructions executed in the string length routine, and the execution rate. If the returned length had been wrong both the returned and correct length would be shown but the instruction count would be omitted.

The strings themselves can be found in the test code after the `str` label. The testbench does not print out the strings, just their lengths. Feel free to modify the strings if it helps with debugging, but please restore them before the deadline.

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.)

Two changes were made for this assignment: implementation of the RISC-V-like `orb.c` instruction, and implementation of the MIPS32 r6 (revision 6) `clo` (count leading ones) instruction. Also new is the ability to start and stop tracing.

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 trace will mostly include the three string length routines, but it will also include a few testbench instructions. The trace includes line numbers so that there should be no confusion about where an instruction is from.

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)
```

At the prompt enter `step 100` to run the next 100 instructions. The instructions in the string length routines will be traced, but the count of 100 instructions also includes the test routine (as of this writing). For example:

```
(spim) step 100
[0x00400064]    0x4080b000  mtc0 $0, $22                    ; 218: mtc0 $0, $22

** Starting Test of Routine "strlen_p1  (Problem 1 - Bit Ops)" **
[0x004000d0]    0x0100f809  jalr $31, $8                    ; 251: jalr $t0
# Change in $31 ($ra)    0x4000bc -> 0x4000d8    Decimal: 4194492 -> 4194520
[0x004000d4]    0x40154800  mfc0 $21, $9                    ; 252: mfc0 $s5, $9
# Change in $21 ($s5)           0 -> 0x23        Decimal: 0 -> 35
[0x00400000]    0x20820000  addi $2, $4, 0                  ; 84: addi $v0, $a0, 0█
# Change in $2  ($v0)   0xffffffff -> 0x10010000  Decimal: -1 -> 268500992
[0x00400004]    0x94880000  lhu $8, 0($4)                   ; 87: lhu $t0, 0($a0)
# Change in $8  ($t0)    0x400000 -> 0x3100      Decimal: 4194304 -> 12544
[0x00400008]    0x3109ff00  andi $9, $8, -256               ; 88: andi $t1, $t0, 0xff00█
# Change in $9  ($t1)   0x100101f0 -> 0x3100     Decimal: 268501488 -> 12544
[0x0040000c]    0x11200006  beq $9, $0, 24 [DONE0-0x0040000c]; 89: beq $t1, $0, DONE0█
[0x00400010]    0x310900ff  andi $9, $8, 255                ; 90: andi $t1, $t0, 0xff█
```

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.

Problem 1: Routine `strlen_p1` in `hw01.s` computes the length of a string using a loop that loads two characters at a time. It achieves a rate of .329 char/insn. Modify it so that it uses a `lw` instead of `lhu`. (Note that there is no such thing as `lwu` in MIPS I. Such an instruction only makes sense if registers are larger than 32 bits.) It is possible to achieve .393 chars /insn, or maybe even faster.

The string starting address will be in register `a0`. That address will be a multiple of 4. Strings end with a null (a zero). The byte after the null is not part of the string and can be of any value. Don't assume it is a particular value.

Your solution should use MIPS-I instructions and should not use pseudo instructions except for `nop`. See the check-box comments (such as `[ ] Code should be efficient.`) for additional restrictions, requirements, and reminders.

The solution appears below. The complete solution file is at `https://www.ece.lsu.edu/ee4720/2022/hw01-sol.s.html`. The easy part is changing `lhu` to a `lw` and changing `addi a0, a0, 2` to `addi a0, a0, 4`. Next we need to modify the code so that it looks at the two most significant bytes. Because MIPS immediates are 16 bits we can't simply use an instruction like `andi t1, t1, 0xff000000`. But we can use an instructions like `lui` to load the constant in to a register, `t6` in the solution, then mask using `and t1, t1, t6`. Something is similar for mask `0xff0000`. Finally we need to adjust and add `DONE` targets for the different cases.

Since performance is important the masks are prepared before the loop is entered. The solution appears below.

```
strlen_p1:
        # CALL VALUE
        #  $a0: Address of first character of string.
        # RETURN
        #  $v0: The length of the string (not including the null).

        addi $v0, $a0, 0

        # SOLUTION: Prepare masks for two high bytes.
        lui $t6, 0xff00    # t6 ->  0xff000000
        lui $t7, 0xff      # t7 ->    0xff0000

LOOP:                          # SOLUTION described by comments below.
        lw $t0, 0($a0)         # Change lhu to lw
        and $t1, $t0, $t6      # Mask off most-significant byte ..
        beq $t1, $0, DONE0     # .. if it is zero we are done ..
        and $t1, $t0, $t7      # .. otherwise mask off next byte ..
        beq $t1, $0, DONE1     # .. and if that one is zero we're done ..
        andi $t1, $t0, 0xff00  # .. otherwise mask the next one ......
        beq $t1, $0, DONE2
        andi $t1, $t0, 0xff
        bne $t1, $0, LOOP
        addi $a0, $a0, 4
        sub $v0, $a0, $v0
        jr $ra
        addi $v0, $v0, -1

DONE2:  # SOLUTION: Modify the case for byte 2
        sub $v0, $a0, $v0
        jr $ra
        addi $v0, $v0, 2

DONE1:  # SOLUTION: Add a case for byte 1
        sub $v0, $a0, $v0
        jr $ra
        addi $v0, $v0, 1

DONE0:
        jr $ra
        sub $v0, $a0, $v0
```

Problem 2: Complete `strlen_p2` so that it determines the string length by loading four characters (using a `lw`) and checks for the null using the RISC-V-like `orc.b` (Bitwise OR-Combine, byte granule) instruction. Also helpful will be the MIPS32 r6 `clz` and `clo` instructions.

The `orc.b` instruction is part of the RISC-V bit manipulation ISA extensions. See the documentation for this instruction for details on what it does. The documentation is linked to the course references page and of course can be found on the RISC-V site. The `orc.b` is in the `strlen_p2` routine, but it doesn't do anything useful. Of course, that should be changed as part of the solution.

The MIPS32 `clz` and `clo` might also come in handy. Look for the MIPS32 r6 (not the older versions) Volume 2 manuals on the course references page.

It is possible to complete this so that it runs at 0.947 char /insn or faster.

The solution appears below. The complete solution file is at `https://www.ece.lsu.edu/ee4720/2022/hw01-sol.s.html`.

The `orc.b r1, r2` instruction operates on each byte of the `r2` value. If the byte is non-zero it is replaced by `0xff` otherwise it remains zero. The transformed value is returned. So for `r2 = 0x1100aa00` the value assigned to `r1` would be `0xff00ff00`. If `r2` holds four bytes of a string, then a value of `r1=0xffffffff` indicates that no null has been found.

The number of bytes before a null can be determined by counting the number of leading ones (the number of consecutive 1's starting at the most-significant bit position). As suggested in the assignment a `clo` instruction does just that.

Since performance is a goal, the main loop must be written using as few instructions as possible. For that reason the code in the main loop checks only that there are no nulls, and leaves the task of determining how many characters preceded the null to the code after the loop exit. In the solution appearing below the main loop consists of just four instructions: `lw`, `orc.b`, `bne`, and `addi`.

After the exit a `clo` counts the most-significant 1's and a `srl` is used to divide that value by 8, converting bits to chars. See the comments in the solution for details.

Grading note: Many seemingly did not take into account that when performance is important nothing should be done in a main loop that could be done either before or after the main loop. For example, in the solution below the main loop uses a value of `t3` that was prepared before the loop was entered.

```
strlen_p2:
        # CALL VALUE
        #  $a0: Address of first character of string.
        # RETURN
        #  $v0: The length of the string (not including the null).

        addi $v0, $a0, 4
        addi $t3, $0, -1  # SOLUTION: Prepare value: 0xffffffff

LOOPB:
        lw $t0, 0($a0)
        orc.b $t1, $t0

        # SOLUTION:  Example execution of orc.b:
        #       String at address in $a0: "AB"
        # t0:   A B  X             <-- Value in ASCII.  is null, X unknown.
        # t0: 0x41420099            <-- Value in hex.
        # t0: 0x41 0x42 0x00 0x99  <-- Value separated into four bytes
        # t1: 0xff 0xff 0x00 0xff  <-- Result. Non-zero bytes changed to 0xff
        # t1: 0xffff00ff            <-- Result. Non-zero bytes changed to 0xff

        beq $t1, $t3, LOOPB        # If t1 is 0xffffffff then no null found ..
        addi $a0, $a0, 4           # .. so increment address for next word.

        # A null has been found. Count number  of leading 1's to find out.
        # t1: 0xffff00ff            <-- Continuing with example.
        clo $t2, $t1
        # t2: 16                    <-- Found 16 consecutive 1's starting at MSB.
        sra $t2, $t2, 3            # Effectively divide by 8.
        # t2:  2                    <-- 16 1's means 2 bytes

        sub $v0, $a0, $v0          # Add on number of bytes before null.

        jr $ra
        add $v0, $v0, $t2
```