LSU EE 4720 Homework 2 Solution Due: 16 February 2018

The solution to several of the problems in this assignment requires material about to be covered in class, in particular, stalling instructions to avoid hazards. For coverage of this material see slide set six, `http://www.ece.lsu.edu/ee4720/2018/lsli06.pdf`*. For a solved problem see 2014 Homework 1 Problem 3. Feel free to look through old homework and exams for other similar problems, but when doing so make sure that the MIPS implementation matches the one in this problem: the muxen at the ALU inputs should each have just 2 inputs.*

Problem 1: Recall that in the `unsqw` program from Homework 1 there was a loop that had to copy the prior occurrence of a piece of text to the output buffer. That loop from the solution appears below, and again re-written to improve performance, at least that was the goal.

```
# Original Code --------------------------------------------------------------
        #  Copy the prior occurrence of text from some part of the
        #  output buffer to the end of the output buffer.

        #  At this point in code:
        #    Reference marker is in $t0.
        #    Length is in register $t3.
        #    Distance is in register $t4.
        #
        sub $t4, $a1, $t4  # Compute starting address of prior occurrence.
        add $t5, $t4, $t3  # Compute ending address of prior occurrence.
        addi $a0, $a0, 1

COPY_LOOP:
        lb $t0, 0($t4)     # Load character of prior occurrence ..
        sb $t0, 0($a1)     # .. and write it to the end of the output buffer.
        addi $t4, $t4, 1
        bne $t4, $t5, COPY_LOOP
        addi $a1, $a1, 1
        j LOOP
        nop
```

```
# Improved Code ---------------------------------------------------------------
        #  Copy the prior occurrence of text to the end of the output buffer.

        #  At this point in code:
        #    Reference marker is in $t0.
        #    Length is in register $t3.
        #    Distance is in register $t4.
        #
        sub $t4, $a1, $t4  # Compute starting address of prior occurrence.
        addi $a0, $a0, 1

        # Round length (L) up to a multiple of 4.
        #
        addi $t7, $t3, 3
        andi $t8, $t7, 0xfffc   # Note: this only works if L < 65536
        sub $t6, $t8, $t3
        #
        # At this point:
        #   $t8: L', rounded-up length.
        #   $t6: Amount added to L to round it up. That is, L' - L
        #        t6 can be 0, 1, 2, or 3.
        #        If t6 is 0, then L' = L;
        #        if t6 is 1, then L' = L + 1; etc.

        # Decrement prior-occurrence and output-buffer pointers.
        #
        sub $t4, $t4, $t6
        sub $a1, $a1, $t6

        # Jump to one of the four lb instructions in the copy loop.
        #
        la $t7, COPY_LOOPd4   # Get address of first lb instruction.
        sll $t6, $t6, 3       # Compute offset to lb that we want to start at.
        add $t7, $t7, $t6     # Compute address of starting lb ..
        jr $t7                # .. and jump to it.
        add $t5, $t4, $t8     # Don't forget to compute stop address.

COPY_LOOPd4:
        lb $t0, 0($t4)
        sb $t0, 0($a1)
        lb $t0, 1($t4)
        sb $t0, 1($a1)
        lb $t0, 2($t4)
        sb $t0, 2($a1)
        lb $t0, 3($t4)
        sb $t0, 3($a1)
        addi $t4, $t4, 4
        bne $t4, $t5, COPY_LOOPd4
        addi $a1, $a1, 4
        j LOOP
        nop
```

Let L denote the length of the prior occurrence of text to copy.

(a) Determine the number of instructions executed by the original code in terms of L. Include the copy loop and the instructions before it shown above. State any assumptions.

The number of instructions executed is: $3 + 5L + 2 = 5 + 5L$. This includes the `nop` since the time to execute the program includes the time to handle `nop`s.

(b) Determine the number of instructions executed by the improved code in terms of L. Include the copy loop and the instructions before it shown above. State any assumptions.

Short Answer: The number of instructions executed is: $13 + 2L + 3\lceil L/4\rceil + 2 = 15 + 2L + 3\lceil L/4\rceil \approx 15 + 2.75L$.

Explanation: The 15 term covers instructions outside of the loop. Each full iteration of the loop executes four `lb/sb` instruction pairs however the loop can be entered at any one of the four `lb` instructions, with the entry point chosen so that exactly L characters are copied. The $2L$ term covers the `lb/sb` pairs. The loop also includes two `addi` instructions and a branch. These are executed once per iteration and there are $\lceil L/4\rceil$ iterations. These three instructions are covered by the $3\lceil L/4\rceil$ term.

(c) What is the minimum value of L for the improved method to actually be faster?

An approximate solution can be found by solving $5 + 5L = 15 + 2.75L$ for L, yielding $L = \frac{40}{9} \approx 4.\overline{44}$. Because L must be an integer and because we ignored the ceiling function we need to investigate values of L around 4. For $L = 6$ the improved method is 2 instructions faster, for all $0 < L < 6$ the original method takes fewer cycles. So the minimum value is $\boxed{L = 6}$.

(d) What is it about the improved code that helps performance?

Short Answer: The two `addi`s and the `bne` are executed just once for each four characters copied.

Long Answer: In both the original and improved code one `lb` and one `sb` are executed for each character copied. That is captured in the $2L$ term in the expression for the improved code and part of the $5L$ term in the expression for the original code. So for these the two codes are comparable. But in the improved code the two `addi` instructions and the `bne` are executed once for every four characters copied while in the original code they are executed four times as often: one for each character copied. That is what is responsible for the improved performance.

Problem 2: *Note: The following problem is identical to 2016 Homework 1 Problem 1. Try to solve it without looking at the solution.* Answer each MIPS code question below. Try to answer these by hand (without running code).

(*a*) Show the values assigned to registers `t1` through `t8` (the lines with the tail comment `Val:`) in the code below. Refer to the MIPS review notes and MIPS documentation for details.

Solution appears below (to the right of `SOLUTION`, of course).

```
        .data
myarray:
        .byte 0x10, 0x11, 0x12, 0x13
        .byte 0x14, 0x15, 0x16, 0x17
        .byte 0x18, 0x19, 0x1a, 0x1b
        .byte 0x1c, 0x1d, 0x1e, 0x1f


        .text
        la $s0, myarray   # Load $s0 with the address of the first value above.
                          # Show value retrieved by each load below.
        lbu $t1, 0($s0)   # Val:    SOLUTION:  0x10
        lbu $t2, 1($s0)   # Val:    SOLUTION:  0x11
        lbu $t2, 5($s0)   # Val:    SOLUTION:  0x15
        lhu $t3, 0($s0)   # Val:    SOLUTION:  0x1011
        lhu $t4, 2($s0)   # Val:    SOLUTION:  0x1213

        addi $s1, $0, 3

        add $s3, $s0, $s1
        lbu $t5, 0($s3)   # Val:    SOLUTION:  0x13

        sll $s4, $s1, 1             SOLUTION:  Note: s4 <= 3<<1 = 6
        add $s3, $s0, $s4
        lhu $t6, 0($s3)   # Val:    SOLUTION:  0x1617

        sll $s4, $s1, 2             SOLUTION:  Note: s4 <= 3<<2 = 12
        add $s3, $s0, $s4
        lhu $t7, 0($s3)   # Val:    SOLUTION:  0x1c1d
        lw $t8, 0($s3)    # Val:    SOLUTION:  0x1c1d1e1f
```

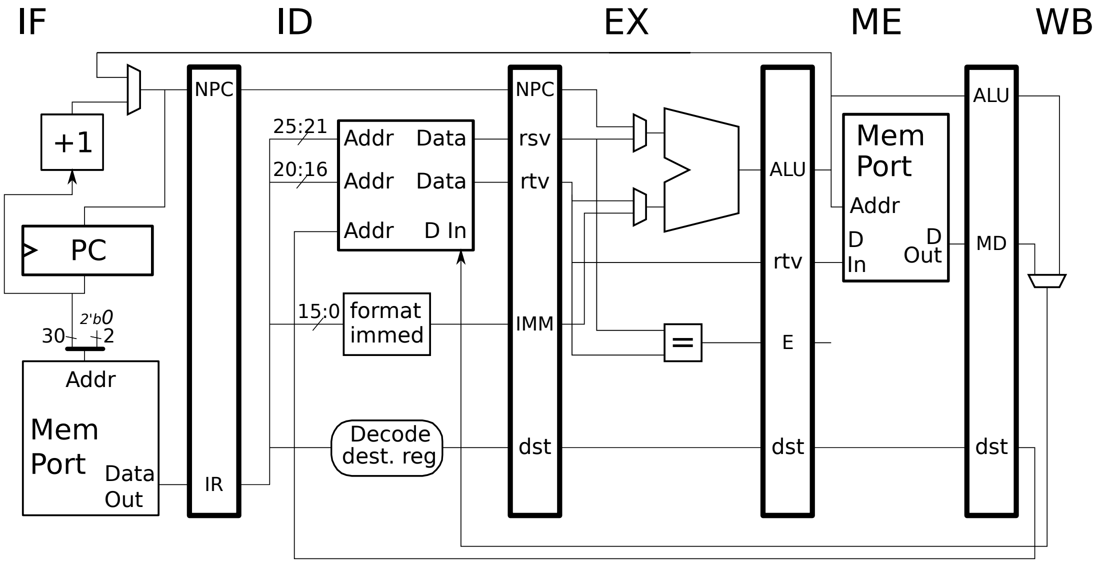
(*b*) The last two instructions in the code above load from the same address. Given the context, one of those instructions looks wrong. Identify the instruction and explain why it looks wrong. (Both instructions should execute correctly, but one looks like it's not what the programmer intended.)

Register `s0` holds an address that the programmer decided to call `myarray`, so lets think of the data starting at that address as an array. Normally, to access element `i` of an array that starts at address `a`, you load data at address `a + i * s`, where `s` is the size of an array element. In the code fragment above, register `s0` holds the starting address (`a` in the example). From the way the code is written it looks like register `s1` is holding the element index (`i` in the example). Because the `sll` in the last group of four instructions is effectively multiplying `s1` by 4, it looks like the load should be of the `s1`'th element of an array of elements of size 4. That's consistent with the `lw`, which loads a 4-byte element, and the last `lhu` looks out of place. The `lhu` that loads `t6` looks fine, because its address was computed from a value of `s1` multiplied by 2.

(*c*) Explain why the following answer to the question above is wrong for the MIPS 32 code above: *"The `lw` instruction should be a `lwu` to be consistent with the others."*

There is no **`lwu`**, because when loading a 32-bit quantity into a 32-bit register there is no need to distinguish between a signed and unsigned quantity. In contrast, the **`lhu`** and **`lh`** load a 16-bit quantity into a 32-bit register, the **`lhu`** sets the high 16 bits to zero, it *zero-pads*, while **`lh`** sets the high 16 bits to the value of the MSB of the loaded value, it *sign extends*.

Problem 3: *Note: The following problem was assigned in each of the last three years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse.* Appearing below are **incorrect** executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.



(*a*) Explain error and show correct execution.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7
 lw r2, 0(r4)      IF ID EX ME WB
 add r1, r2, r7       IF ID EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

The add depends on the lw through r2, and for the illustrated implementation the add has to stall in ID until the lw reaches WB.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7     SOLUTION
 lw r2, 0(r4)      IF ID EX ME WB
 add r1, r2, r7       IF ID ----> EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

(*b*) Explain error and show correct execution.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7
 add r1, r2, r3    IF ID EX ME WB
 lw r1, 0(r4)         IF ID -> EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

There is no need for a stall because the lw writes r1, it does not read r1.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7  SOLUTION
 add r1, r2, r3    IF ID EX ME WB
 lw r1, 0(r4)         IF ID EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

(*c*) Explain error and show correct execution.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7
 add r1, r2, r3    IF ID EX ME WB
 sw r1, 0(r4)         IF ID -> EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

A longer stall is needed here because the `sw` reads `r1` and it must wait until `add` reaches `WB`.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7  SOLUTION
 add r1, r2, r3    IF ID EX ME WB
 sw r1, 0(r4)         IF ID ----> EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

(*d*) Explain error and show correct execution.

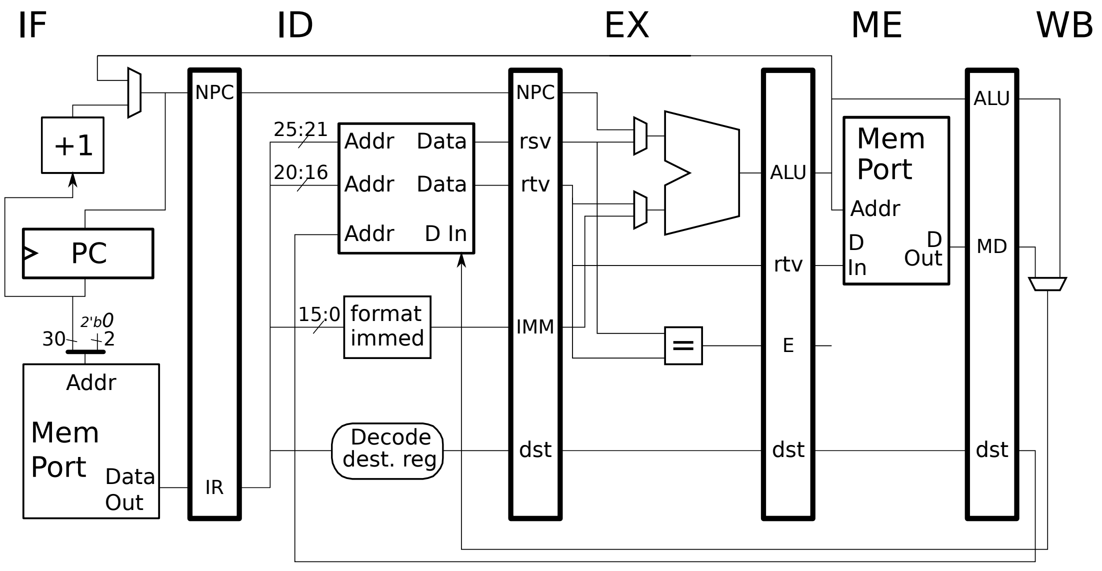
```
LOOP: # Cycles     0  1  2  3  4  5  6  7
 add r1, r2, r3    IF ID EX ME WB
 xor r4, r1, r5       IF ----> ID EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

The stall above allows the `xor`, when it is in `ID`, to get the value of `r1` written by the `add`; that part is correct. But, the stall starts in cycle 1 *before* the `xor` reaches `ID`, so how could the control logic know that the `xor` needed `r1`, or for that matter that it was an `xor`? The solution is to start the stall in cycle 2, when the `xor` is in `ID`.

```
LOOP: # Cycles     0  1  2  3  4  5  6  7   SOLUTION
 add r1, r2, r3    IF ID EX ME WB
 xor r4, r1, r5       IF ID ----> EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

Problem 4: The MIPS code below is taken from the solution to 2018 Homework 1. Show the execution of this MIPS code on the illustrated implementation for two iterations. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be the value being written. (In class we called such a register file *internally bypassed*.)

- Check carefully for dependencies.
- Focus on when the branch target is fetched and on when wrong-path instructions are squashed.
- Be sure to stall when necessary.



The solution appears below. Each iteration includes two stalls due to dependencies, and two squashes due to the taken branch. The first stall occurs in cycles 3 and 4 and is due to the dependence between the `lbu` and `sb` carried by `t0`. The second stall occurs in cycles 7 and 8 and is due to the dependence between the first `addi` and the `bne` carried by `t4`. Because branches are resolved in ME (look for the path from `EX/ME.ALU` back to `PC`) two wrong-path instructions will be fetched before the control logic can determine that they are indeed wrong-path instructions. During the cycle at which the branch is resolved the two wrong path instructions are squashed, this occurs below in cycle 10. The wrong-path instructions in this example are called `X1` and `X2`, these are whatever instructions appear in memory after the second `addi`.

```
CLOOP:  # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 lbu $t0, 0($t4) IF ID EX ME WB
 sb $t0, 0($a1)     IF ID ----> EX ME WB
 addi $t4, $t4, 1      IF ----> ID EX ME WB
 bne $t4, $t5, CLOOP            IF ID ----> EX ME WB
 addi $a1, $a1, 1                  IF ----> ID EX ME WB
 X1                                         IF IDx
 X2                                            IFx
CLOOP:  # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
 lbu $t0, 0($t4)                                  IF ID EX ME WB
 sb $t0, 0($a1)                                      IF ID ----> EX ME WB
 addi $t4, $t4, 1                                       IF ----> ID EX ME WB
 bne $t4, $t5, CLOOP                                             IF ID ----> EX ME WB
 addi $a1, $a1, 1                                                   IF ----> ID EX ME WB
 X1                                                                          IF IDx
 X2                                                                             IFx
CLOOP:  # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
 lbu $t0, 0($t4)                                                                   IF ID
```