**LSU EE 4720** **Homework 1** Solution **Due: 12 February 2016**

**Problem 1:** 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.)

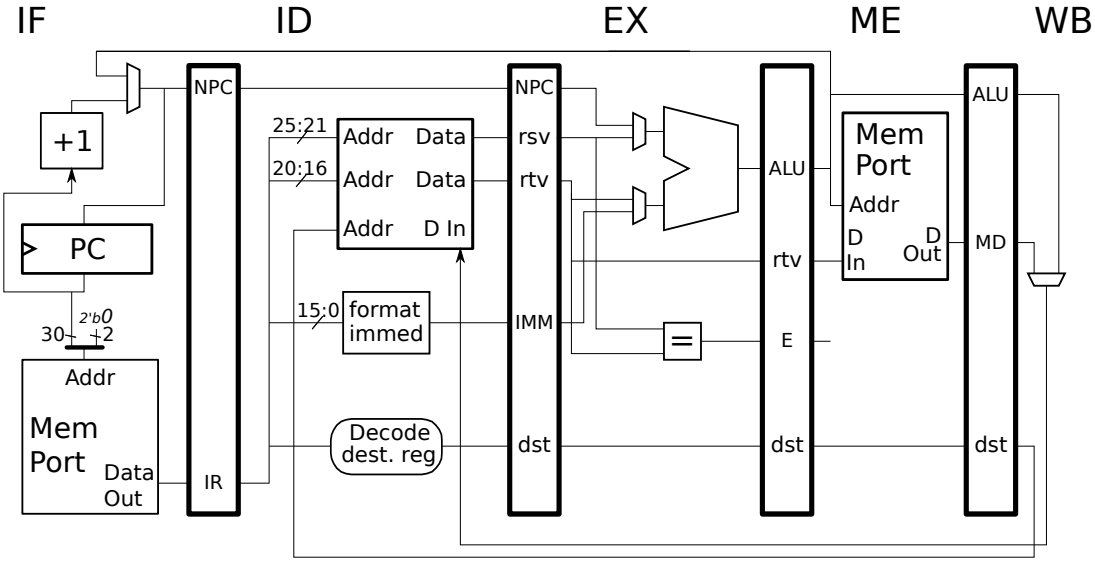
Register `s0` holds an address that the programmer decided to call `myarray`, so lets think of the data starting at that address as an array. Normally, to access element `i` of an array that starts at address `a`, you load data at address `a + i * s`, where `s` is the size of an array element. In the 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 2:** *Note: The following problem was assigned last year and two years ago 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.

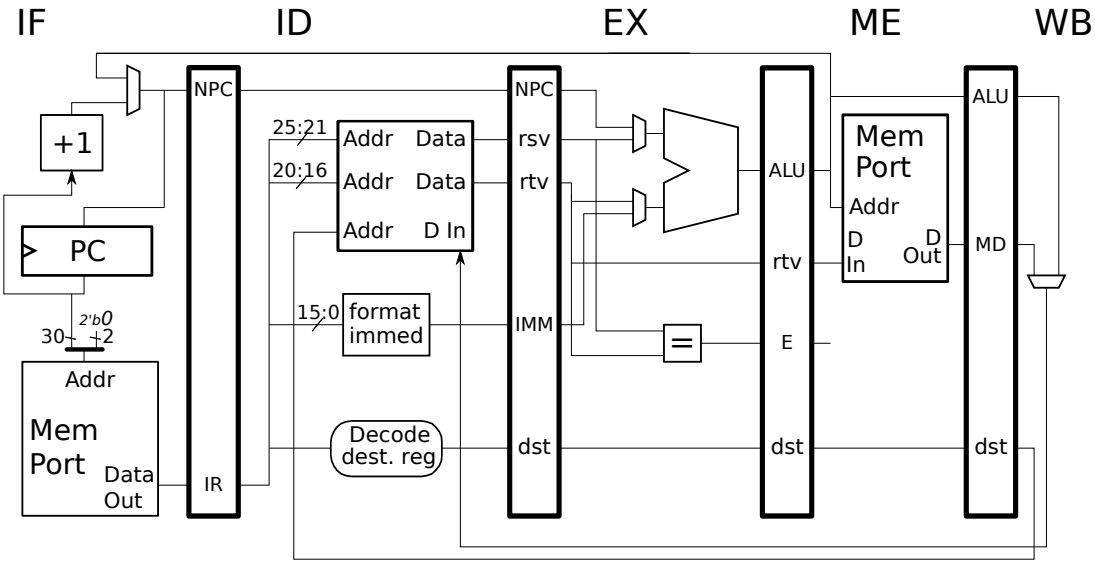
```
LOOP: # Cycles     0  1  2  3  4  5  6  7
 add r1, r2, r3    IF ID EX ME WB
 xor r4, r1, r5       IF ----> ID EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

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 3:** Show the execution of the MIPS code below on the illustrated implementation. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be value being written. (In class we called such a register file *internally bypassed*.)

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




The solution appears below. Note that because MIPS branches are delayed, the `lw` instruction is allowed to complete execution even though the branch is taken. The `xor`, in contrast is on the wrong path and so is squashed after the branch resolves. Also note the timing of the fetch of the branch target based on this particular implementation: the branch target, `ori`, enters `IF` in the cycle after the branch leaves `ME`.

```
 # SOLUTION
 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 add r1, r2, r3   IF ID EX ME WB
 sub r4, r1, r5      IF ID ----> EX ME WB
 beq r1, r1, SKIP       IF ----> ID EX ME WB
 lw r6, 0(r4)                    IF ID -> EX ME WB
 xor r7, r8, r9                     IF -> x
SKIP:
 ori r9, r10, 11                          IF ID EX ME WB

 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```