LSU EE 4720 Homework 1 Solution Due: 20 February 2015

Problem 1: Answer each MIPS code question below. Try to answer these by hand (without running code).

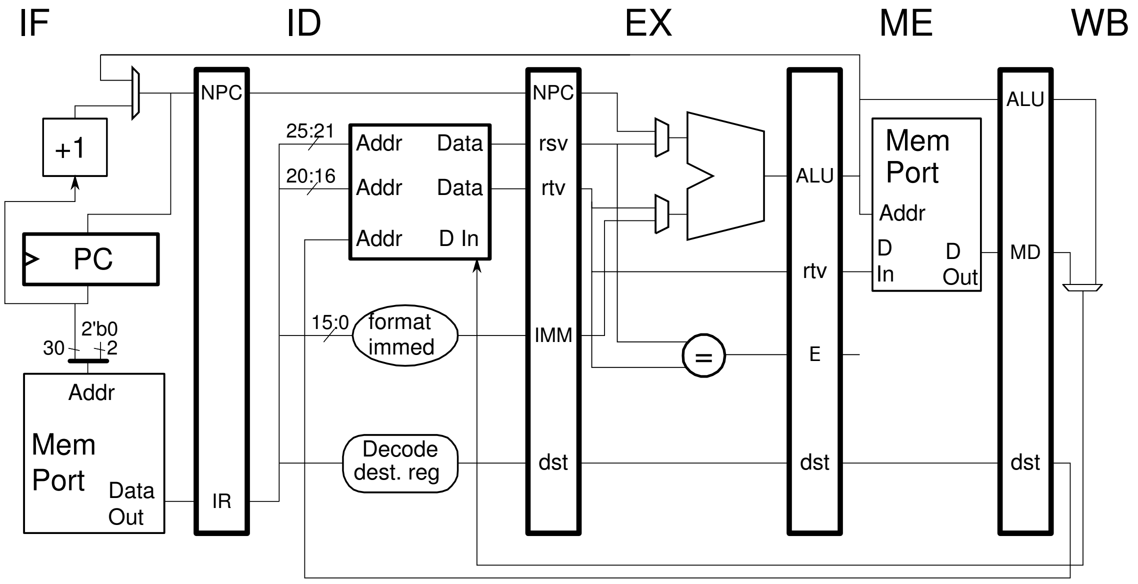
(*a*) Where indicated, show the changed register in the following simple code fragments:

```
# r1 = 10, r2 = 20, r3 = 30, etc.
#
add r1, r2, r3
#
# Changed register, new value:   SOLUTION:  r1 from 10 to 50.

# r1 = 10, r2 = 20, etc.
#
add r1, r1, r2
#
# Changed register, new value:   SOLUTION:  r1 from 10 to 30.
```

(*b*) Show the values assigned to registers `s1` through `s6` in the code below. Correctly answering this question requires an understanding of MIPS big-endian byte ordering and of the differences between `lw`, `lbu`, and `lb`. Refer to the MIPS review notes and MIPS documentation for details.

```
        .data
values: .word 0x11121314
        .word 0xaabbccdd
        .word 0x99887766
        .word 0x41424344

        .text
        la  $s0, values  # Load $s0 with the address of the first value above.
        lw  $s1, 0($s0)    SOLUTION:  0x11121314
        lw  $s2, 4($s0)    SOLUTION:  0xaabbccdd
        sh  $s2, 0($s0)  # Note: this is a store *half*.
        lbu $s3, 0($s0)    SOLUTION:  0xcc
        lbu $s4, 3($s0)    SOLUTION:  0x14
        lb  $s5, 4($s0)    SOLUTION:  0xffffffaa
        lb  $s6, 7($s0)    SOLUTION:  0xffffffdd
```

Problem 2: *Note: The following problem was assigned last year 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.

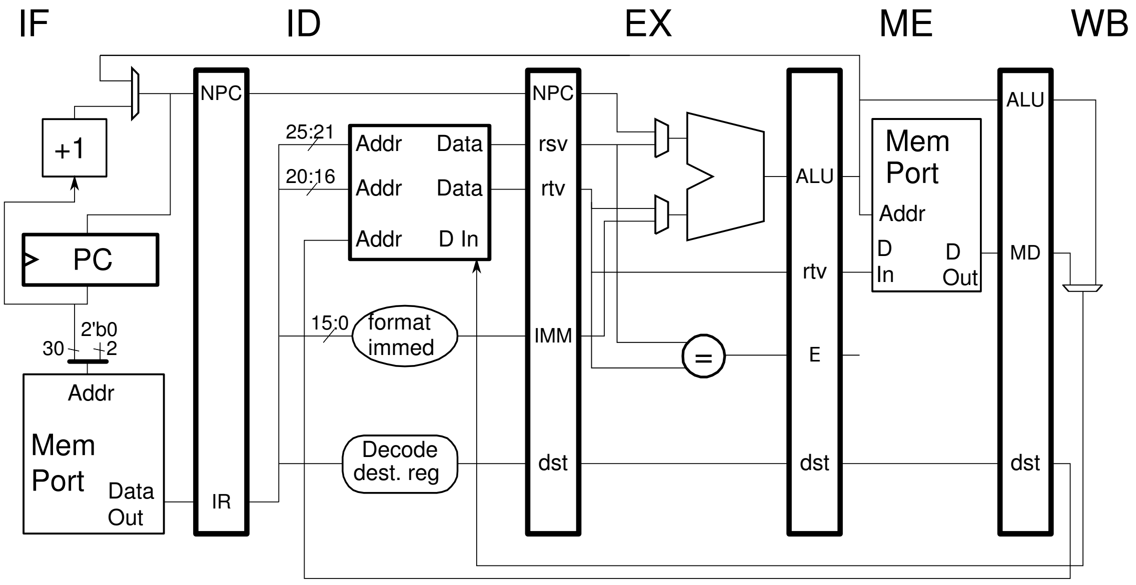
```
LOOP: # Cycles     0  1  2  3  4  5  6  7
 add r1, r2, r3    IF ID EX ME WB
 xor r4, r1, r5       IF ----> ID EX ME WB
LOOP: # Cycles     0  1  2  3  4  5  6  7
```

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 which registers are sources and which are destinations, especially for the `sw` instruction.
- Be sure to stall when necessary.



Solution appears below. Since there are no bypass paths the `lw` must stall in ID until `add` reaches WB. (If the register file were not internally bypassed the `add` would have to stall in ID one more cycle than it does below.) The `sub` stalls for `r4` produced by `lw`, and `sh` stalls for `r5` produced by `sub`.

Common Mistakes:

One common mistake is forgetting that store instructions, such as `sw` and `sh`, do not write registers. That's why `sub` does not need to wait for `r1`.

Another common mistake is assuming that a standard set of bypass paths is available. The implementation in this problem does not have bypass paths which is why the code below suffers from so many stalls.

```
# 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
lw r4, 0(r1)        IF ID ----> EX ME WB
sw r1, 0(r1)           IF ----> ID EX ME WB
sub r5, r4, r1                  IF ID -> EX ME WB
sh r5, 4(r1)                       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
```

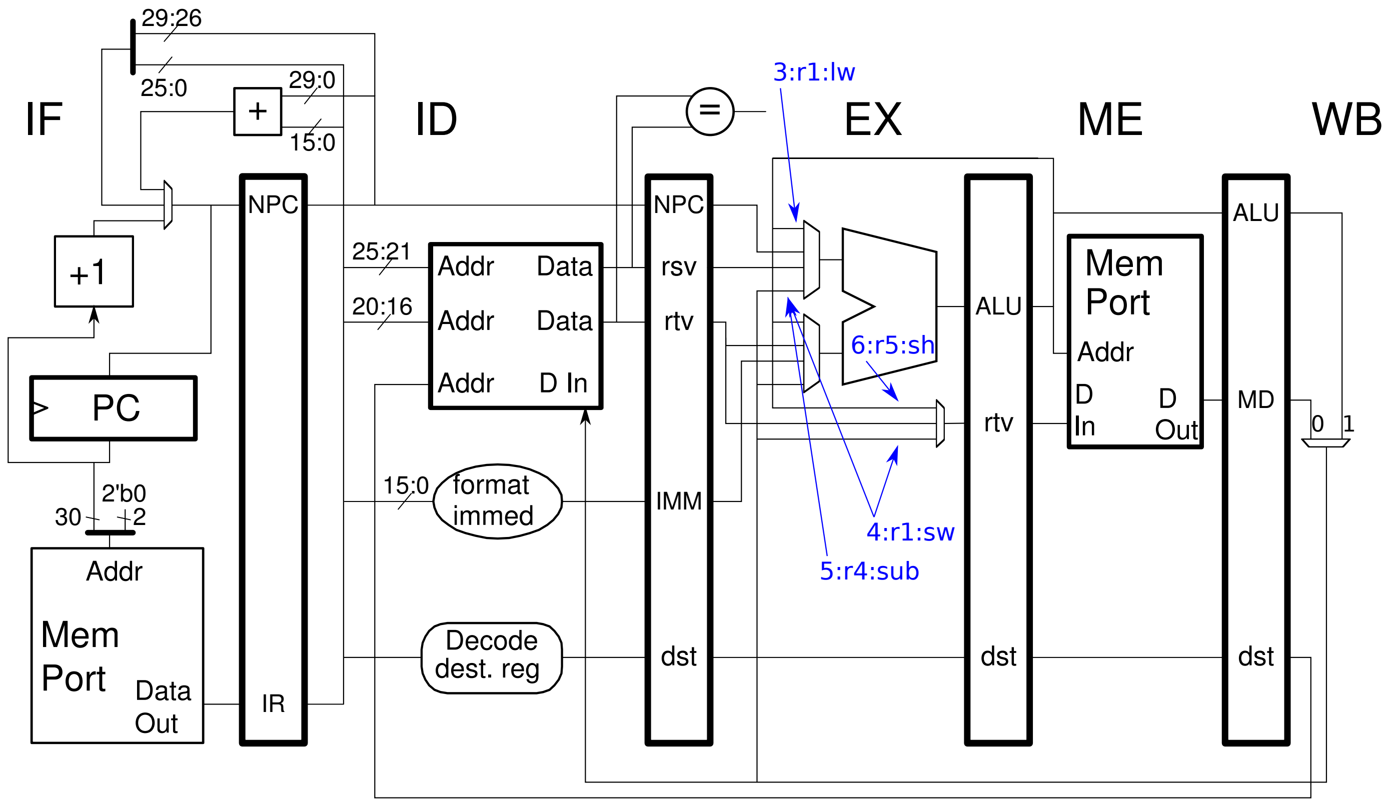
Problem 4: The code below is the same as the code used in the previous problem, but the MIPS implementation is different.

(*a*) Show the execution of the MIPS code below on the illustrated implementation.

Solution appears below.

(*b*) On the diagram label multiplexor data inputs connecting to bypass paths that are used in the execution of this code. The label should include the cycle number, the register, and the instruction consuming the value. For example, the label $\boxed{\text{3:r1:lw}}$ might be placed next to one of the data inputs on the ALU's upper mux.

Solution appears below in blue. Notice that in this case there are no stalls.



```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8
 add r1, r2, r3    IF ID EX ME WB
 lw r4, 0(r1)         IF ID EX ME WB
 sw r1, 0(r1)            IF ID EX ME WB
 sub r5, r4, r1             IF ID EX ME WB
 sh r5, 4(r1)                  IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8
```