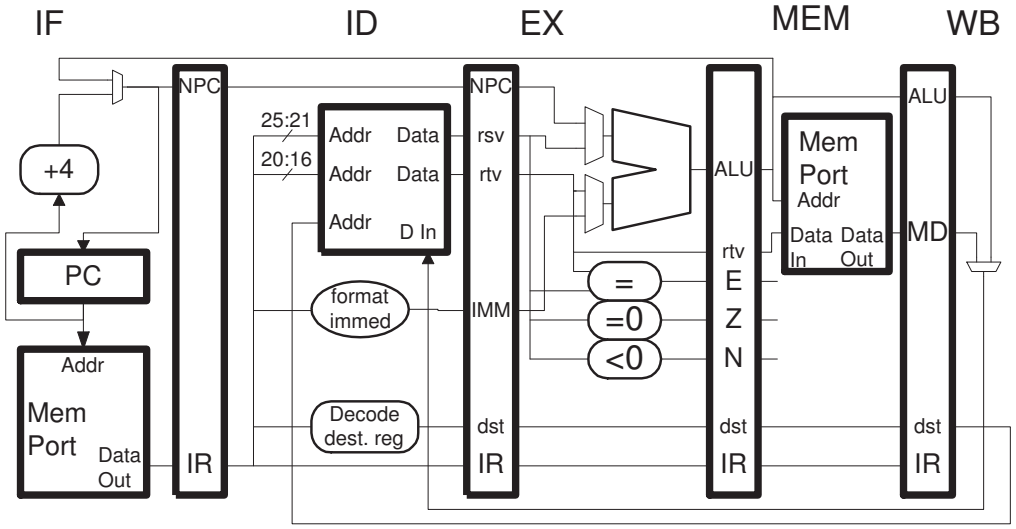
LSU EE 4720 **Homework 1** Solution **Due: 10 February 2014**

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.




```
 lw r1, -6(r3)

 lw r5, -2(r3)
LOOP:
 add r5, r5, r1

 lw r1, 2(r3)

 bneq r3, r4, LOOP

 addi r3, r3, 4

 jr r31

 sw r5, 0(r6)
```

Solution on next page.

(*a*) Show the execution of the code above on the illustrated implementation up to and including the first instruction of the third iteration (that is, the third time that the `add` instructions is fetched).

- Carefully check the code for dependencies.
- Be sure to stall when necessary.
- Pay careful attention to the timing of the fetch of the branch target.

Solution appears below.

```
 lw r1, -6(r3)       IF ID EX ME WB
 lw r5, -2(r3)          IF ID EX ME WB
LOOP:  # Cycles      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1            IF ID ----> EX ME WB
 lw r1, 2(r3)                 IF ----> ID EX ME WB
 bneq r3, r4, LOOP                     IF ID EX ME WB
 addi r3, r3, 4                           IF ID EX ME WB
 jr r31                                      IF IDx
 sw r5, 0(r6)                                   IFx
LOOP:  # Cycles      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1                                    IF ID EX ME WB
 lw r1, 2(r3)                                         IF ID EX ME WB
 bneq r3, r4, LOOP                                       IF ID EX ME WB
 addi r3, r3, 4                                             IF ID EX ME WB
 jr r31                                                        IF IDx
 sw r5, 0(r6)                                                     IFx
LOOP:  # Cycles      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1                                                      IF ID ..
```

(*b*) Compute the CPI for a large number of iterations.

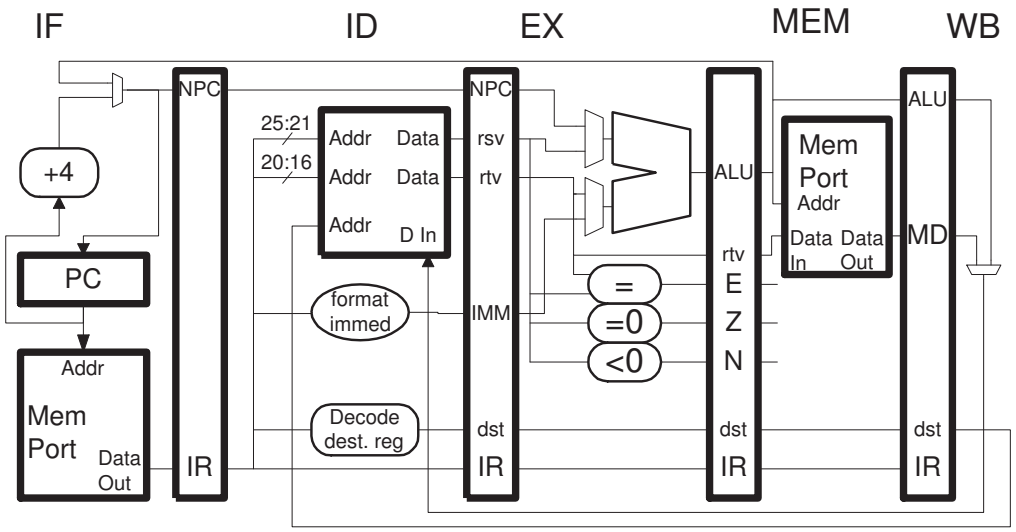
Recall that we define an iteration to start when the first instruction is in `IF`. In the execution above the first iteration starts in cycle 2 and the second iteration starts in cycle 10, and the third starts in cycle 16.

Notice that the first and second iterations are different: in the first there is a stall, in the second there isn't a stall. The first iteration takes $10 - 2 = 8$ cycles and the second takes $16 - 10 = 6$ cycles.

To compute the CPI for a large number of iterations we need a repeating pattern. The stalls in the first iteration are caused by instructions before the loop, they won't affect subsequent iterations. Even so, how can we be sure that the third and subsequent iterations will be like the second? By looking at the state of the pipeline when the first instruction in the loop is fetched. For the first iteration the state is `add` in `IF`, `lw r5` in `ID` and `lw r1` in `EX`. In the second iteration we have `add` in `IF`, `addi` in `ME` and `bneq` in `WB`. The third iteration starts exactly the same way as the second. Therefore the third will look like the second, and by induction all future iterations. So we can use 6 cycles as the iteration time.

The CPI is then $\frac{6\,\text{cyc}}{4\,\text{insn}} = 1.5\,\text{CPI}$.

Problem 2: 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
```