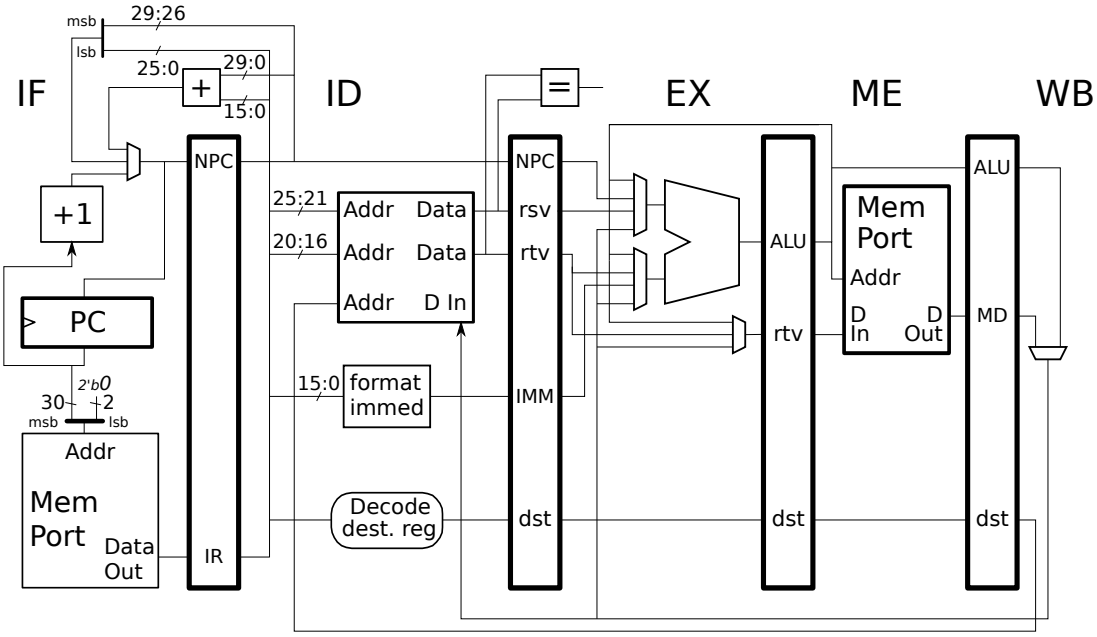
LSU EE 4720 **Homework 3** Solution **Due: 9 March 2022**

Note: The following problems (or very similar problems) were assigned in 2020 and 2021, and their solutions are available. DO NOT look at the solutions unless you are lost and can't get help elsewhere. Even in that case just glimpse.

Problem 1: Appearing below are **incorrect** executions on the illustrated implementation. Notice that this implementation is different than the one from the previous problem. For each execution explain why it is wrong and show the correct execution.



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

```
# Cycle          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
```

There is a bypass path available so that there is no need to stall.

```
# Cycle          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
```

(*b*) The execution of the branch below has **two** errors. One error is due to improper handling of the `andi` instruction. (That is, if the `andi` were replaced with a `nop` there would be no problem in the execution below.) The other is due to the way the `beq` executes. As in all code fragments in this problem, the program is correct, the only problem is with the illustrated execution timing.

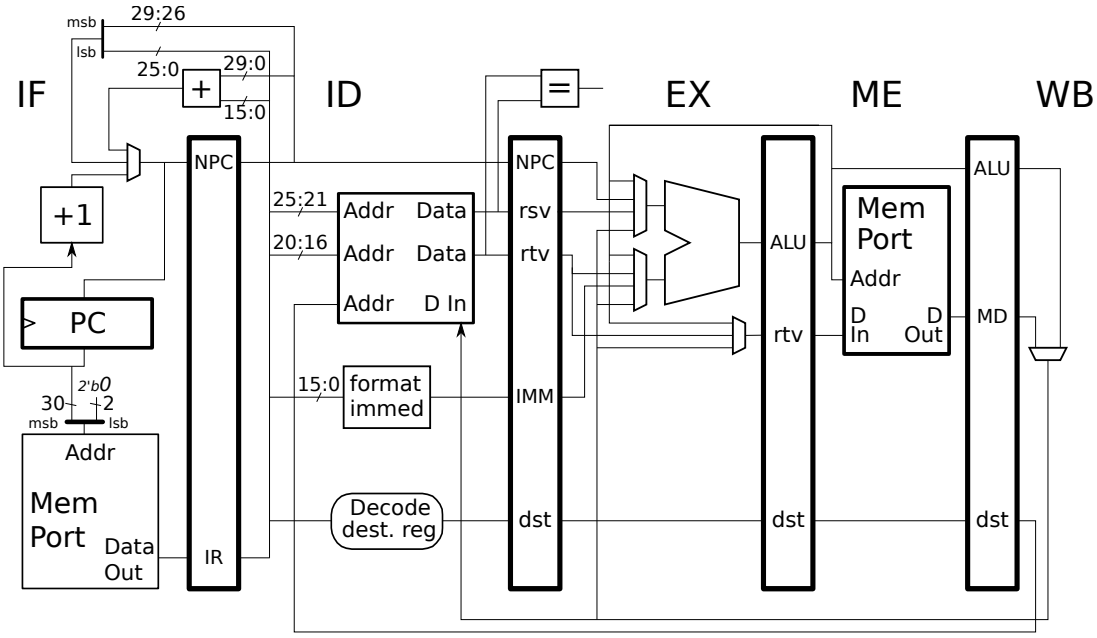
```
 # Cycle:            0  1  2  3  4  5  6  7  8
 andi r2, r2, 0xff   IF ID EX ME WB
 beq r1, r2, TARG       IF ID EX ME WB
 add r3, r4, r5            IF ID EX ME WB
 xor                          IFx
TARG:
 sw r6, 7(r8)                    IF ID EX ME WB
 # Cycle:            0  1  2  3  4  5  6  7  8
```

Briefly, the two problems are the lack of a stall for the `andi`/`beq` dependence carried by `r2` and because the branch target is fetched one cycle later than it should be. The correct execution appears below.

Detailed explanation: In the illustrated implementation the $\boxed{=}$ in `ID` is used to compute the branch condition for `beq` (and `bne`). When the branch reaches `ID`, in cycle 2, the value of `r2` retrieved from the register file is outdated, it needs to use the value computed by `andi`. Since there are no bypass paths to the $\boxed{=}$ logic the branch will need to stall until `andi` reaches writeback. The stalls occur in cycles 2 and 3.

The illustrated implementation resolves the branch in `ID`, and so the branch target should be in `IF` when the branch is in `EX`. In the execution above the target isn't fetched until the branch is in `ME`, in cycle 4. That is fixed below by fetching the target a cycle earlier. The `xor` is no longer fetched and squashed.

```
 # Cycle:            0  1  2  3  4  5  6  7  8  9  SOLUTION
 andi r2, r2, 0xff   IF ID EX ME WB
 beq r1, r2, TARG       IF ID ----> EX ME WB
 add r3, r4, r5            IF ----> ID EX ME WB
 xor
TARG:
 sw r6, 7(r8)                    IF ID EX ME WB
 # Cycle:            0  1  2  3  4  5  6  7  8  9
```



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

```
# Cycle          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
```

The `add` depends on the `lw` through `r2`, and for the illustrated implementation the `add` has to stall in `ID` until the `lw` reaches `ME` so that the `add` can bypass from `WB`.

```
# Cycle          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
```

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

```
# Cycle          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
```

There is no need for a stall because `r1` is not a source register of `lw`. Note that `r1` is a destination of `lw`.

```
# Cycle          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
```

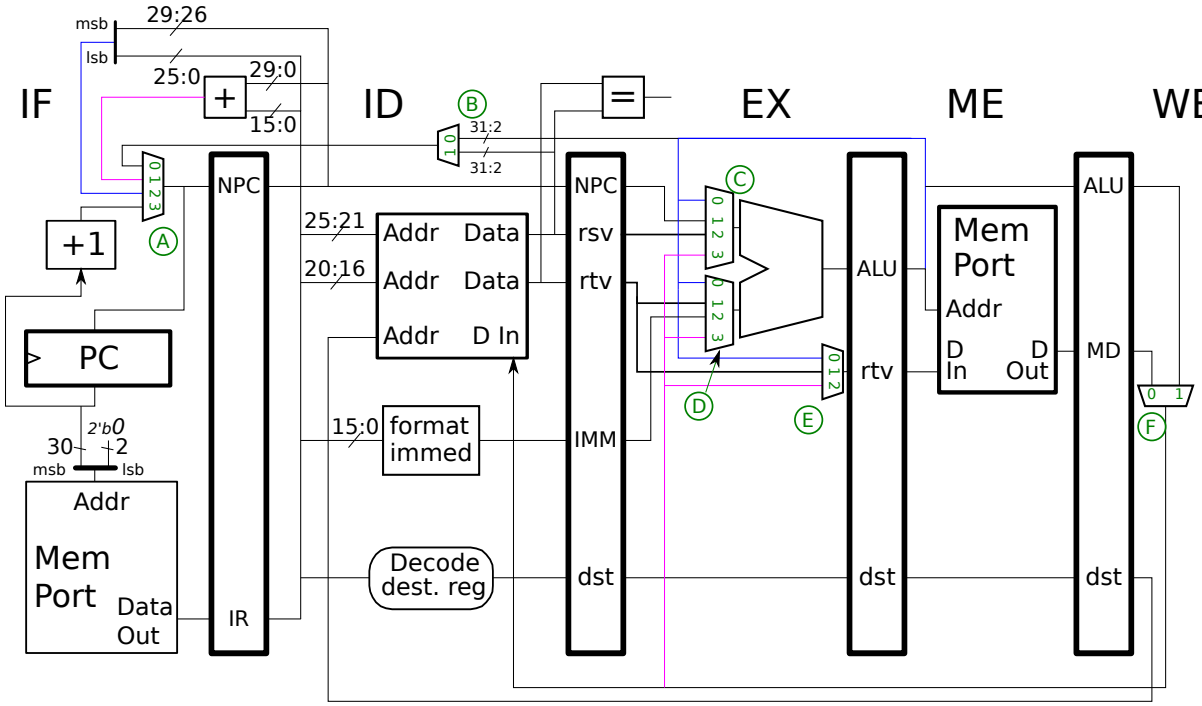
(*e*) Explain error and show correct execution.

```
# Cycle          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
```

No stall is needed here because the **sw** can use the **ME**-to-**EX** bypass path.

```
# Cycle          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
```

Problem 2: Appearing below is the labeled MIPS implementation from 2018 Midterm Exam Problem 2(b), and as in that problem each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.



(*a*) Use F0. Don't be fancy about it, just one instruction is all it takes.

Solution appears below. F0 is used by values being loaded from memory into the pipeline. Load instructions, include `lw`, use F0.

```
# SOLUTION
# Cycle       0  1  2  3  4
lw r1, 0(r2)  IF ID EX ME WB
```

(*b*) Use F0, C2, and D3 at the same time. The code **should not** suffer a stall. More than one instruction is needed for the solution. *Note: This is new in 2022.*

The solution appears below. The F0 mux input is used by load instructions. For that a `lw` instruction is included in the solution. The D3 mux input is used to bypass something from WB to the second ALU operand. To use F0 and D3 at the same time the load instruction must be in `WB` at the same time as the other instruction (an `add` in the example below) is in `EX`. The `add` instruction uses the D3 bypass to get the value of `r1` written by the `lw`. To use C2 the instruction must use an unbypassed value for the first source. The first source of the `add` is `r9` which has not been written by the two prior instructions, and so it can use the value from the register file.

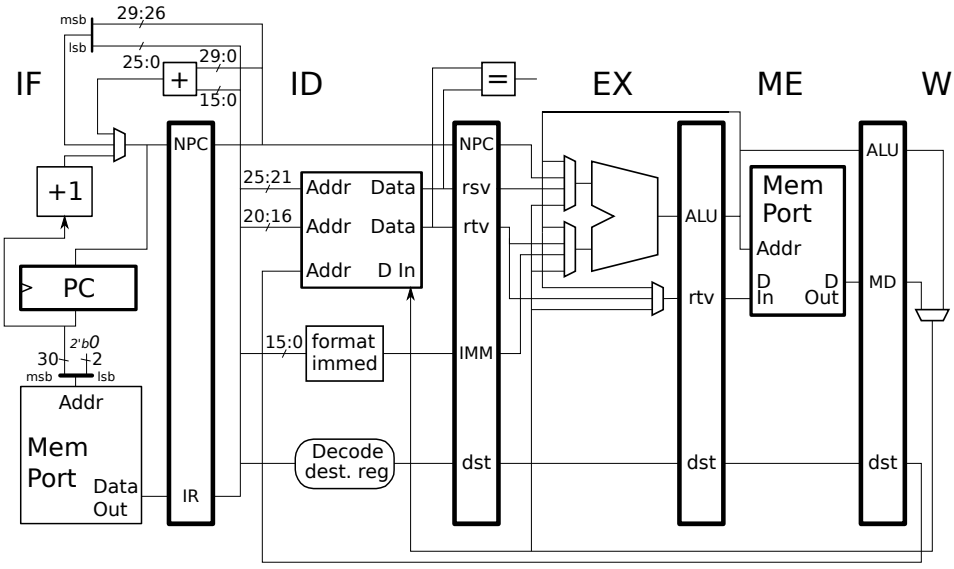
```
# SOLUTION
# Cycle         0  1  2  3  4  5  6
lw r1, 0(r2)    IF ID EX ME WB
xor r5, r6, r7     IF ID EX ME WB
add r3, r9, r1        IF ID EX ME WB
# Cycle         0  1  2  3  4  5  6
```

(*c*) Explain why its impossible to use E0 and D0 at the same time.

If E0 is in use then there must be a store instruction in EX. If D0 is in use then a value is being bypassed to the second ALU source operand of the instruction in EX. But store instructions use an immediate for the second ALU input, so a store in EX can only use D2, it can't use D0 (nor D1 nor D3).

Problem 3: *This problem appeared as Problem 2c on the 2020 final exam.* Appearing below is our bypassed, pipelined implementation followed by a code fragment.

It might be helpful to look at Spring 2019 Midterm Exam Problem 4a. That problems asks for the execution of a loop and for a performance measure based upon how fast that loop executes.



(*a*) Show the execution of the code below on the illustrated implementation up to the point where the first instruction, `addi r2,r2,16`, reaches `WB` in the second iteration.

The execution appears below. The execution is shown until the beginning of the third iteration. (A full-credit solution would only need to show execution until cycle 10, when the `addi r2,r2,16` reaches `WB` in the second iteration.) The only stall is a 1-cycle load/use stall suffered by the `sw`. The first iteration starts in cycle 0 (when the first instruction, `addi`, is in `IF`), the second iteration starts at cycle 6, and the third at cycle 12.

Note that the pattern of stalls in the second iteration is the same as the pattern in the first. We can expect this pattern to continue because the contents of the pipeline is the same at the beginning of the second and third iterations. (The second iteration begins in cycle 6. In that cycle the `addi r2` is in `IF`, the `addi r3` is in `ID`, etc. The contents of the pipeline is the same in cycle 12.)

SOLUTION

```
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 addi r2, r2, 16    IF ID EX ME WB                          First Iteration
 lw r1, 8(r2)          IF ID EX ME WB
 sw r1, 12(r3)            IF ID -> EX ME WB
 bne r3, r4, LOOP            IF -> ID EX ME WB
 addi r3, r3, 32                   IF ID EX ME WB
 sub r10, r3, r2
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 addi r2, r2, 16                      IF ID EX ME WB        Second Iteration
 lw r1, 8(r2)                            IF ID EX ME WB
 sw r1, 12(r3)                              IF ID -> EX ME WB
 bne r3, r4, LOOP                              IF -> ID EX ME WB
 addi r3, r3, 32                                     IF ID EX ME WB
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 addi r2, r2, 16    Third Iteration                        IF ID EX ME WB
```

(*b*) Based on your execution determine how many cycles it will take to complete n iterations of the loop.

The time for n iterations of the loop is n times the duration of one iteration of the loop. The key to solving this correctly is using the correct duration for an iteration. The duration of an iteration if the time between the start of two consecutive iterations. In this class the start time of an iteration is the time at which the first instruction is in `IF`. Using that definition the duration of the first iteration is $6 - 0 = 6\,\text{cyc}$ and the duration of the second is $12 - 6 = 6\,\text{cyc}$. So the number of cycles to complete n iterations is $\boxed{6n\,\text{cyc}}$.

An important point to understand is that the definition of duration above insures that iterations don't overlap. That is, by defining an iteration duration as starting in the `IF` of the first instruction of the iteration, there is no possibility that two iterations overlap and there is no time gap between them. That's what enables us to multiply a duration by the number of iterations to get a total time.

Some might be tempted to add another four cycles to account for the `addi r3` instruction completing execution. No credit would be lost for that in a solution, but that is not useful for our purposes because we might want to add together the duration of different pieces of code, so for us the important thing is when the next instruction can be fetched.