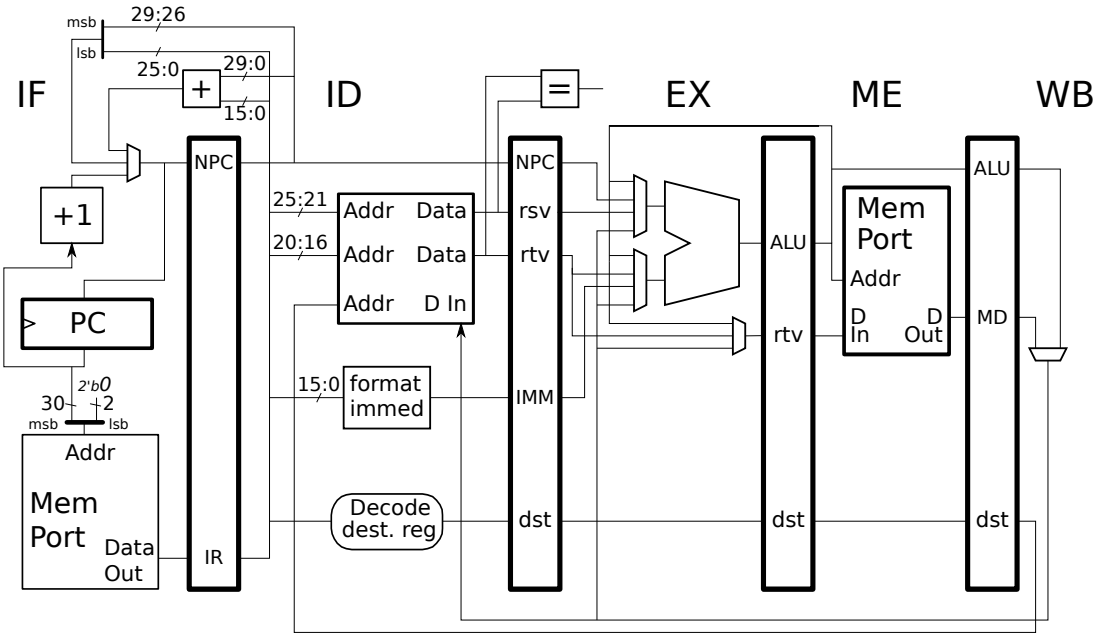
Problem 1: Appearing below are **incorrect** executions on the illustrated implementation. For each execution explain why it is wrong and show the correct execution. *Note: This problem was assigned in 2020, 2021, and 2022, and their solutions are available. DO NOT look at the solutions unless you are lost and can't get help elsewhere. Even in that case just glimpse.*



(*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.

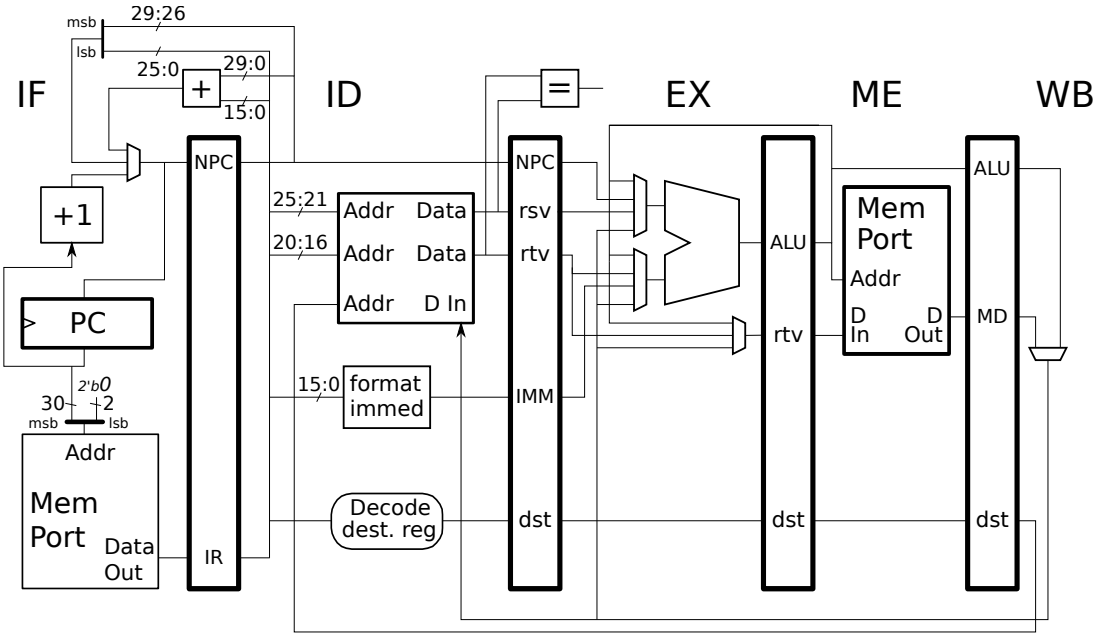
```
 # Cycle:            0  1  2  3  4  5  6  7  8
 andi r2, r2, 0xff   IF ID EX ME WB
 beq r1, r2, TARG       IF ID EX ME WB
 add r3, r4, r5            IF ID EX ME WB
 xor                          IFx
TARG:
 sw r6, 7(r8)                    IF ID EX ME WB
 # Cycle:            0  1  2  3  4  5  6  7  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 When the branch reaches `ID`, in cycle 2, the value of `r2` retrieved from the register file is outdated, it needs to use the value computed by `andi`.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.

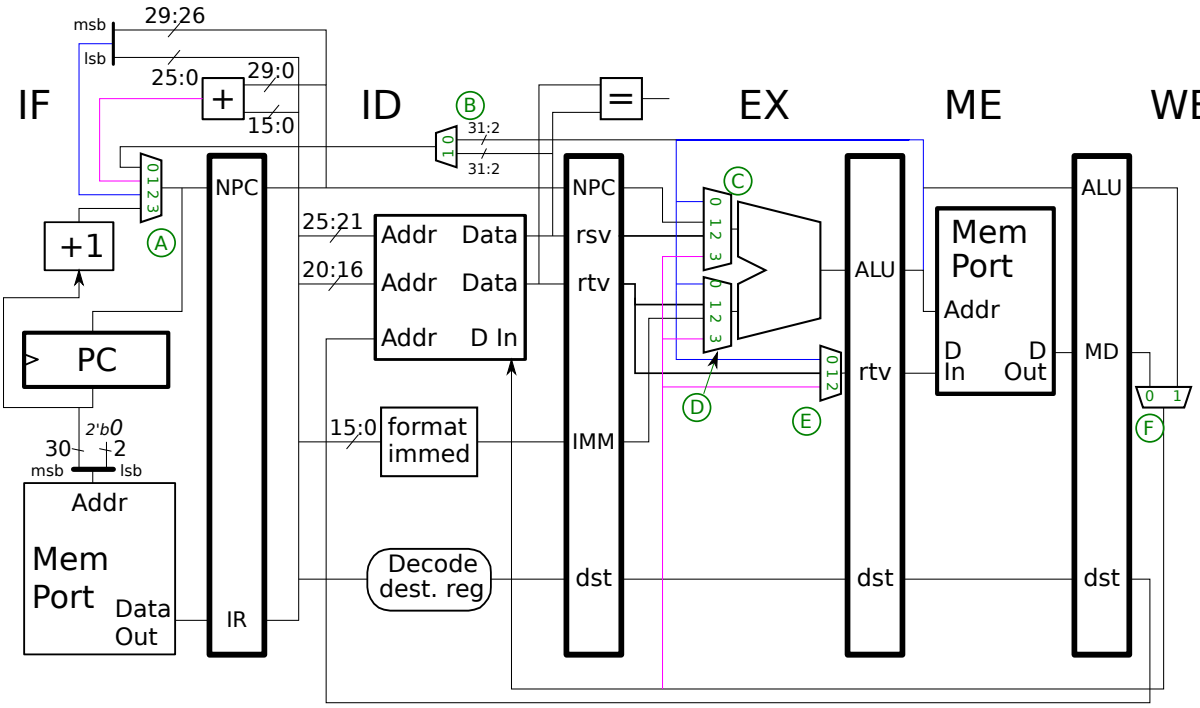
```
# Cycle          0  1  2  3  4  5  6  7
add r1, r2, r3   IF ID EX ME WB
sw r1, 0(r4)        IF ID -> EX ME WB
```

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: Illustrated below is a MIPS implementation in which each multiplexor has a label, such as a circled A at the multiplexor providing a value for the PC. (The implementation debuted on the 2018 midterm exam.) The multiplexor inputs are also numbered. Below the illustration an execution of the program on the implementation is shown for two iterations of a loop. Below the execution is a table with one row for each labeled multiplexor. Complete the table so that it shows the values on the multiplexors' select signals at each cycle based on the execution. Leave an entry blank if its value does not make a difference.

Wire thicknesses and colors have been varied to make it easier to trace them through the diagram. *Before attempting this problem, solve 2018 Midterm Exam Problem 2b, which also appeared as 2022 Homework 3 Problem 2. Also see the 2014 Midterm Exam Problem 1 for a similar problem.*



Continued on the next page.

☑ Complete the table (the rows starting with `A:`, `B:`, etc.) based on the execution below.

☑ Omit select signal values if they do not matter. For example, omit values for `E` for cycles in which there is not a store instruction in `EX`.

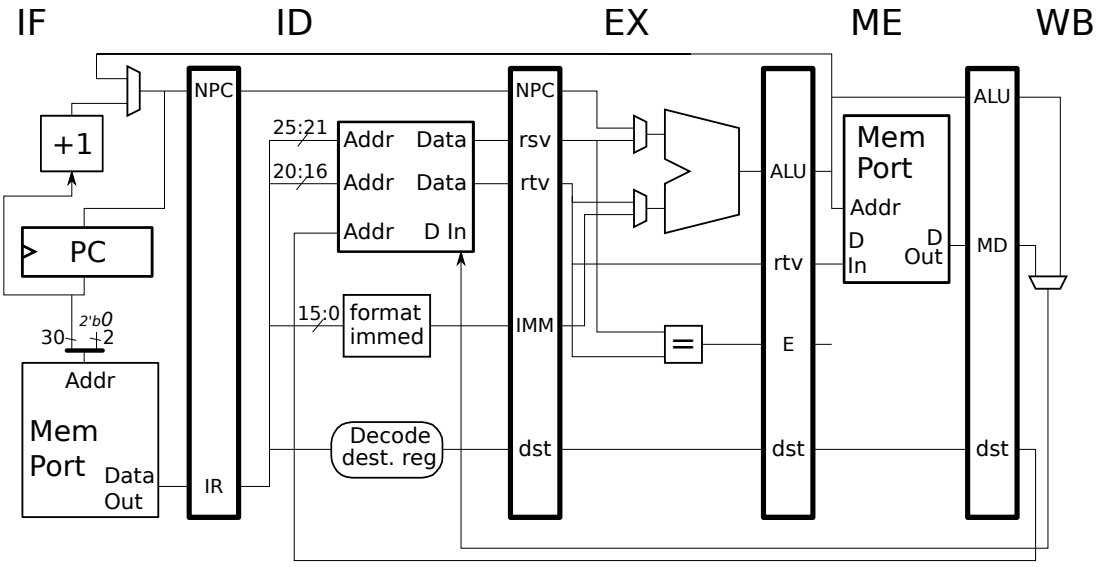
☑ Assume that the branch is taken the second time it appears. (No assumption needed for its first appearance.)

```
 addi r1, r1, -4   IF    ID    EX    ME    WB
LOOP: # Cycle      0     1     2     3     4     5     6     7     8     9
 sw r2, 4(r1)            IF    ID    EX    ME    WB
 lw r1, 8(r2)                  IF    ID    EX    ME    WB
 bne r2, r3, LOOP                    IF    ID    EX    ME    WB
 add r2, r2, r6                            IF    ID    EX    ME    WB
LOOP: # Cycle      0     1     2     3     4     5     6     7     8     9
 sw r2, 4(r1)                                    IF    ID    EX    ME    WB
 lw r1, 8(r2)                                          IF    ID    EX    ME    WB
 bne r2, r3, LOOP                                            IF    ID    EX    ME    WB
 add r2, r2, r6                                                    IF    ID    EX    ME    WB
      # Cycle      0     1     2     3     4     5     6     7     8     9     10    11    12

      # SOLUTION
      # Cycle      0     1     2     3     4     5     6     7     8     9     10    11    12
 A:                3     3     3     3     1     3     3     3     1
 B:
LOOP: # Cycle      0     1     2     3     4     5     6     7     8     9     10    11    12
 C:                            2     0     2           2     2     3           2
 D:                            2     2     2           1     2     2           1
LOOP: # Cycle      0     1     2     3     4     5     6     7     8     9     10    11    12
 E:                                  1                       0
 F:                                        1           0           1           0           1
LOOP: # Cycle      0     1     2     3     4     5     6     7     8     9     10    11    12
```

Problem 3: Show the execution of the code fragments on the following implementations for enough iterations to determine the instruction throughput (IPC). **As always, base the behavior of branches and the availability of bypasses on the implementations. Also, don't forget that MIPS branches have a delay slot.** Sorry for yelling, but I hate it when students miss things.

This problem appeared as most of Problem 1 on the 2022 Final Exam. A solution is not yet available.



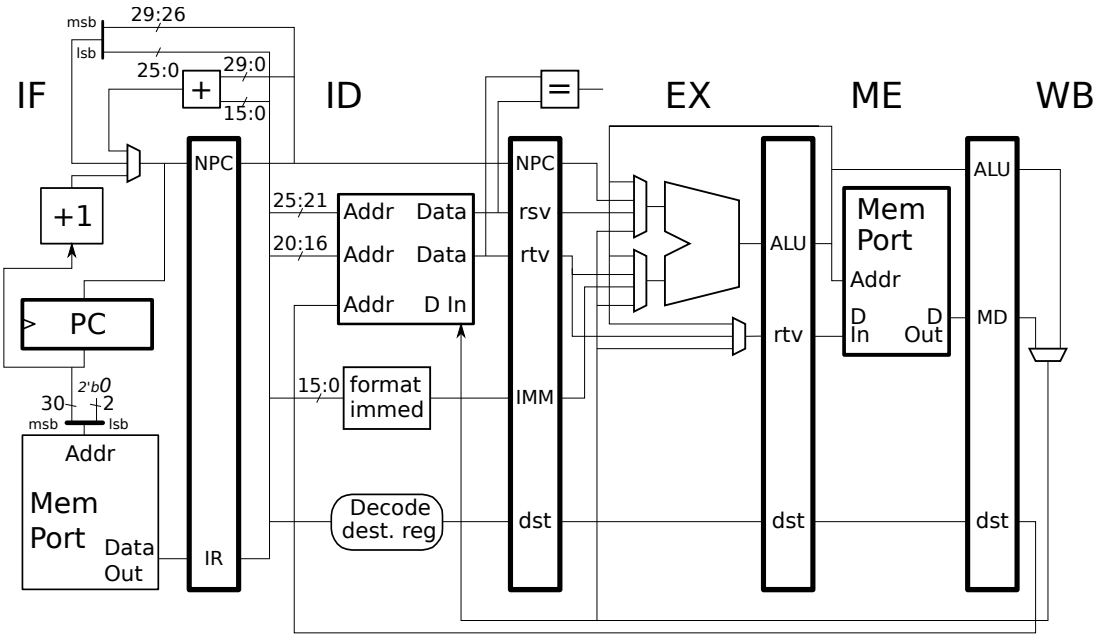
☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

The solution appears below. The branch is resolved in ME, and so the target is fetched (in IF) in the next cycle, when the branch is in WB. Two wrong-path instructions are fetched, `xor` and `sub`. They are squashed when the branch is resolved. (Of course, they would not be squashed if the branch were not taken.)

The instruction throughput is $\frac{2\,\text{insn}}{(8-4)\,\text{cyc}} = \frac{2}{4}\,\text{insn/cycle}$ based on the second iteration starting at cycle 4 and the third iteration starting at cycle 8.

```
 # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12
 bne r1, r2, LOOP  IF ID EX ME WB                # First Iteration
 addi r1, r1, 4       IF ID EX ME WB
 xor r5, r6, r7          IF IDx
 sub r8, r9, r10            IFx
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12
 bne r1, r2, LOOP              IF ID EX ME WB    # Second Iteration
 addi r1, r1, 4                   IF ID EX ME WB
 xor r5, r6, r7                      IF IDx
 sub r8, r9, r10                        IFx
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12
 bne r1, r2, LOOP                          IF ID EX ME WB
 ...

 # These instructions will be completely executed after the last iteration.
 xor r5, r6, r7
 sub r8, r9, r10
```



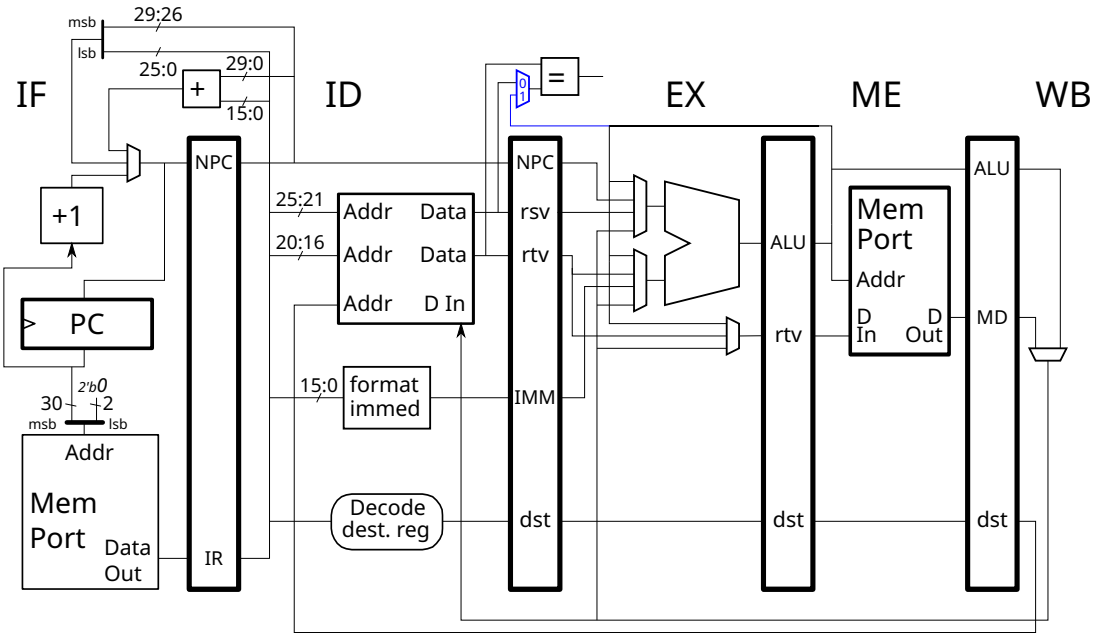
☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

The solution appears below. The good news in this pipeline the branch is resolved in `ID`, meaning that zero wrong-path instructions are fetched. The bad news is that there is a dependence carried by `r1` that stalls `bne` in `ID` for two cycles. For this reason, the instruction throughput is the same: $\frac{2\,\text{insn}}{(6-2)\,\text{cyc}} = \frac{2}{4}\,\text{insn/cycle}$ based on the second iteration starting at cycle 2 and the third iteration starting at cycle 6.

```
LOOP:  # Code in Static Instruction Order
 bne r1, r2, LOOP
 addi r1, r1, 4
 xor r5, r6, r7
 sub r8, r9, r10


 # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 bne r1, r2, LOOP  IF ID EX ME WB                  # First Iteration
 addi r1, r1, 4       IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 bne r1, r2, LOOP        IF ID ----> EX ME WB       # Second Iteration
 addi r1, r1, 4             IF ----> ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 bne r1, r2, LOOP                    IF ID ----> EX ME WB
 addi r1, r1, 4                         IF ----> ID EX ME WB

 # These instructions will be executed after the last iteration.
 xor r5, r6, r7
 sub r8, r9, r10
```



☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

In this implementation there is a bypass that helps with the branch condition dependence, reducing the stall from two cycles to one cycle. The instruction throughput is higher, $\frac{2\,\text{insn}}{(5-2)\,\text{cyc}} = \frac{2}{3}\,\text{insn/cycle}$ based on the second iteration starting at cycle 2 and the third iteration starting at cycle 5.

```
LOOP:  # Code in Static Instruction Order
 bne r1, r2, LOOP
 addi r1, r1, 4
 xor r5, r6, r7
 sub r8, r9, r10

 # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11
 bne r1, r2, LOOP  IF ID EX ME WB                    # First Iteration
 addi r1, r1, 4       IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11
 bne r1, r2, LOOP        IF ID -> EX ME WB          # Second Iteration
 addi r1, r1, 4             IF -> ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11
 bne r1, r2, LOOP                 IF ID -> EX ME WB
 addi r1, r1, 4                      IF -> ID EX ME WB
```