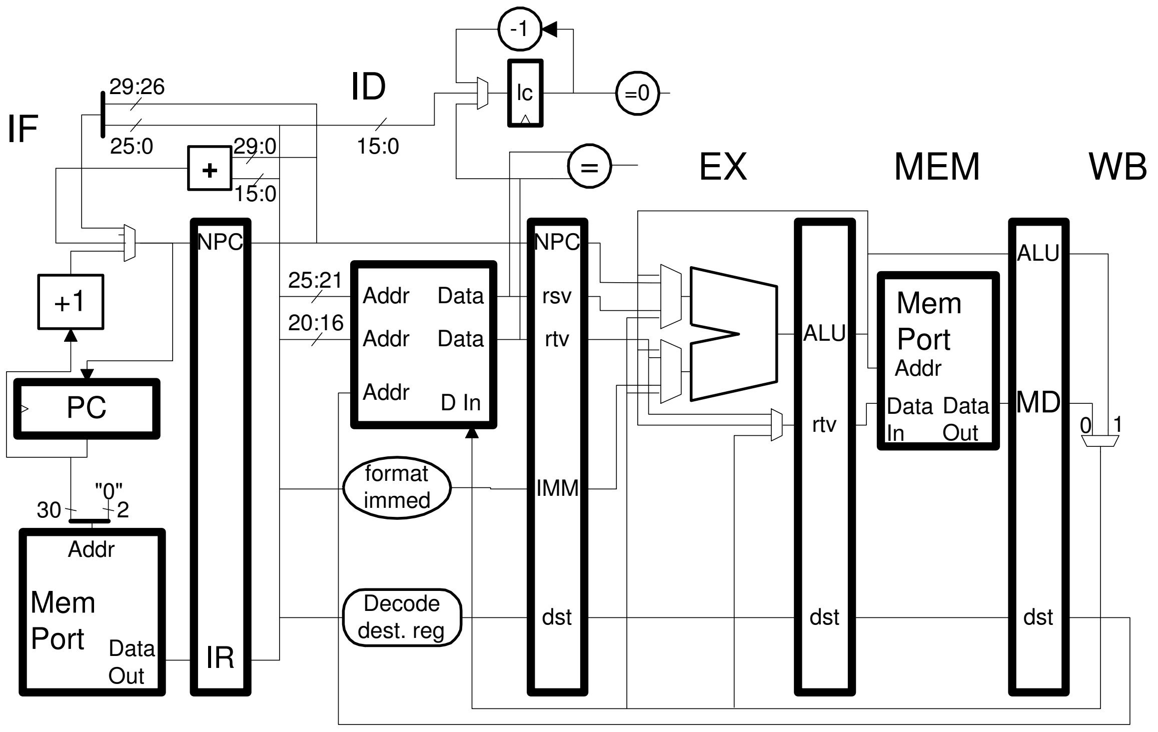
Problem 1: The execution of a new MIPS instruction `blcz TARG`, branch unless loop count register is zero, will result in a delayed control transfer to `TARG` unless the contents of a new register, `lc`, is zero; the target is computed in the same way as ordinary branch instructions. Execution of `blcz` will also decrement `lc` unless it is already zero. The `lc` register is loaded by two new instructions `mtlc` and `mtlci`. The code below uses some of the new instructions and the diagram shows a possible implementation.

```
 mtlc 100         # Load lc register for a 101-iteration loop
LOOP:
 sw r0, 0(r1)
 blcz LOOP        # If lc is not zero branch to LOOP, lc = lc - 1.
 addiu r1, r1, 4
```



(*a*) Re-write the code above using ordinary MIPS instructions and write it so that the loop uses as few instructions as possible. *Hint: A three-instruction loop body is possible.*

The solution is on the next page.

```
 # Solution
 #
 # Re-written code and pipeline execution diagram.
 # Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 addiu r2,r1,400 IF ID EX ME WB
LOOP:
 sw r0, 0(r1)       IF ID EX ME WB
 bne r1, r2, LOOP      IF ID -> EX ME WB
 addiu r1, r1, 4          IF -> ID EX ME WB

# Code below is a repeat of the code above.
 sw r0, 0(r1)                   IF ID EX ME WB
 bne r1, r2, LOOP                  IF ID -> EX ME WB
 addiu r1, r1, 4                      IF -> ID EX ME WB
 sw r0, 0(r1)                               IF ID EX ME WB
 bne r1, r2, LOOP                              IF ID -> EX ME WB
 addiu r1, r1, 4                                  IF -> ID EX ME WB
 # Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

 # Original code and pipeline execution diagram.
 # Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 mtlc 100        IF ID EX ME WB
LOOP:
 sw r0, 0(r1)       IF ID EX ME WB
 blcz LOOP             IF ID EX ME WB
 addiu r1, r1, 4          IF ID EX ME WB

# Code below is a repeat of the code above.
 sw r0, 0(r1)                IF ID EX ME WB
 blcz LOOP                      IF ID EX ME WB
 addiu r1, r1, 4                   IF ID EX ME WB
 # Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*b*) Using pipeline execution diagrams determine the speed of the sample program and your program from the previous part. Only use bypass paths that have been provided.

The original code, which uses blcz, executes without a stall and so it executes at a rate of 0.333 stores per cycle (1 CPI). The re-written code suffers stalls and so only executes at a rate of 0.24 stores per cycle (1.333 CPI).

Note that the question asked for the *speed* of the programs, not the CPI. Since the two programs do the same thing what's important is which one is faster (lower execution time). Since they both do the same number of stores the speed could be measured by stores per cycle. CPI is not a good measure because it only indicates how efficiently the code is running. If two *identical* programs are running on different processors the lower CPI (higher efficiency) would be faster. But since the programs are different CPI is not useful in predicting speed.

(*c*) Unless the control logic is appropriately modified the implementation above may not realize precise exceptions for all integer instructions. In fact, the problem could occur in the example program. Explain what the problem is and show a pipeline execution diagram in which the control logic insures that execution proceeds so that exceptions will be precise. *Hint 1: The exception does not occur in any of the new instructions. Hint 2: One of the two remaining instructions in the example can not raise an exception so it must be the other one.*

```
 # Part of Solution
 # Cycle          0  1  2  3  4  5  6  7  8  9
mtlc 100          IF ID EX ME WB
LOOP:
sw r0, 0(r1)         IF ID EX M*x
blcz LOOP               IF ID --> EX ME WB
addiu r1, r1, 4            IF --> ID EX ME WB
```

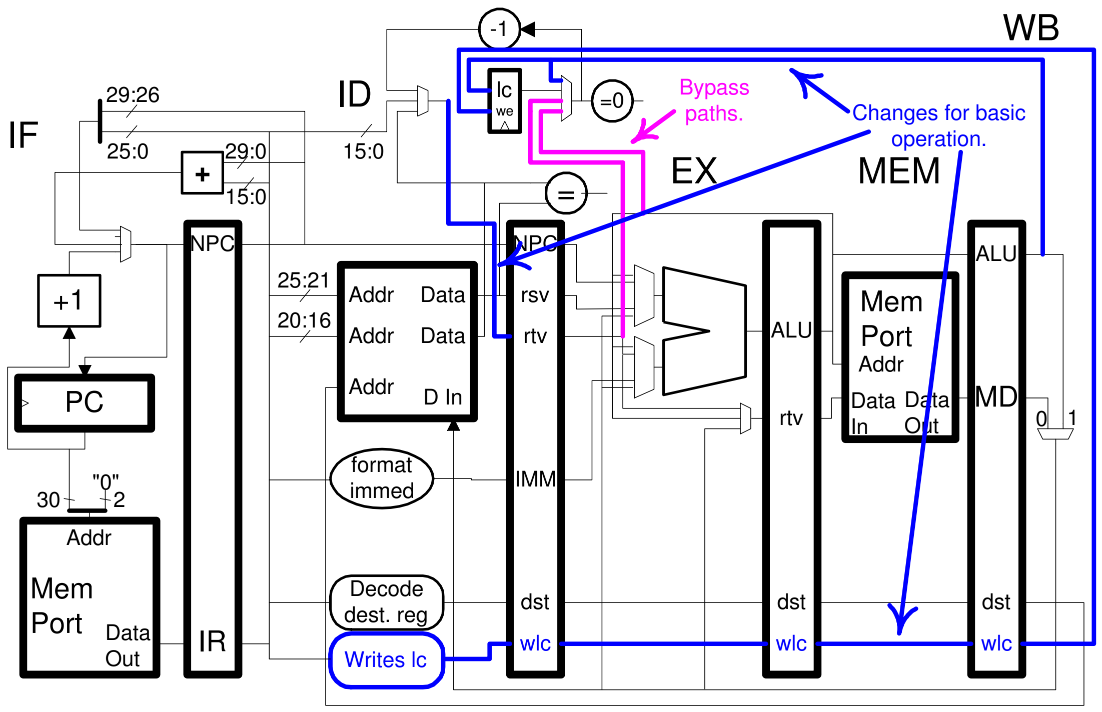
The integer instructions cannot raise precise exceptions with the blcz changes because the lc register is modified in ID, when preceding instructions can still raise exceptions. If they do the handler will see the wrong value in lc.

In the example above sw raises an exception in cycle 4. For the exception to be precise the handler must see the execution of only instructions up to sw, which here is only mtlc, and so the handler should see an lc value of 100. If blcz did not stall in cycle 3 lc would have been decremented and so the handler would have seen an lc value of 99, meaning the exception would not have been precise.

(*d*) Modify the implementation so that precise exceptions are again possible for all integer instructions (while retaining the loop count instructions) without sacrificing performance.

The modifications are shown below. Before the modifications lc would be both read and written in ID. To allow for stall-free precise exceptions modify the implementation so that lc is written in WB, as are the other registers. Those changes are shown in blue. When an instruction that modifies lc, blcz, mtlc, and mtlci, is in ID the new value of lc, rather than writing lc, is put in the ID/EX.rtv pipeline latch. When such an instruction is in EX the ALU is set to pass the lower input through unchanged so that when the instruction reaches WB the new lc value will be in the MEM/WB.ALU latch, that latch is connected to lc's data input. A write enable (we) input is also shown, it is based on the output of $\boxed{\text{Writes lc}}$, which is 1 for instructions that modify lc.

Bypass paths for the lc register are shown in purple.



Grading Notes:

Some solutions tried to repair lc by decrementing its value when an exception is detected. This would fix the problem with `sw` but won't handle cases in which `mtlc` or `mtlci` write `lc`. Also, control logic would have to check if a `blcz` is in the EX and possibly ID stages (depending on timing); many solutions did not point this out.

Another common incorrect solution was keeping the `lc` logic in ID but not modifying `lc` if a preceding instruction raises an exception. That won't work because when an instruction is in ID a doomed preceding instructions might not yet have raised an exception, `sw` is an example. One solution was specific in specifying EX-stage hardware to see if the instruction their would raise a memory-related exception when it reached MEM. That's not possible without doing major damage to the critical path.