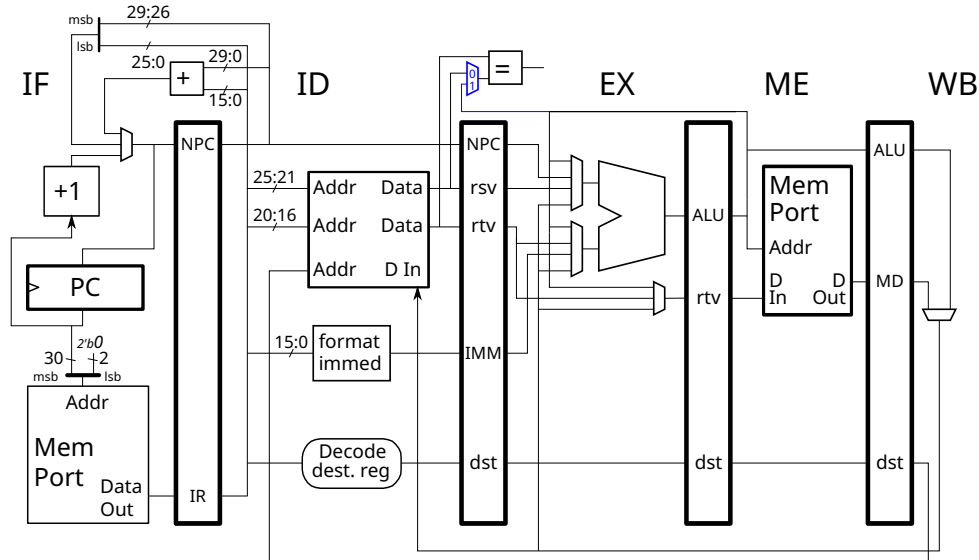


Problem 1: Appearing below is our familiar five stage MIPS implementation with a new branch bypass path shown in blue. For this problem assume that `orc.b` is executed by the ALU.



(a) The code below is based on a solution to Homework 1. Show a pipeline execution diagram of this code on the illustrated hardware. Pay close attention to the behavior of the branch including behavior due to dependencies with prior instructions. Show enough of the execution to compute the instruction throughput in units of IPC.

- Show execution on the illustrated hardware.
- Compute the instruction throughput (IPC).
- Pay attention to dependencies and available bypass paths.

Solution appears below. The only unbypassable dependency was from `orc.b` to `beq`. The `beq` needs the value of `t1` when `beq` is in ID but it cannot be bypassed until `orc.b` reaches ME (using the blue mux) and so the `beq` stalls one cycle.

The instruction throughput is $\frac{4 \text{ insn}}{(6-1) \text{ cyc}} = .8 \text{ insn/cycle}$. Note that the number of cycles in an iteration is computed by using the fetch of the first instruction in the loop body, the `addi`. So the number of cycles is $6 - 1 = 5$.


```

# SOLUTION
lw $t0, 0($a0)      IF ID EX ME WB
LOOPB: # Cycle      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
addi $a0, $a0, 4    IF ID EX ME WB # 1st ITERATION
orc.b $t1, $t0      IF ID EX ME WB
beq $t1, $t3, LOOPB IF ID -> EX ME WB
lw $t0, 0($a0)      IF -> ID EX ME WB
LOOPB: # Cycle      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
addi $a0, $a0, 4    IF ID EX ME WB # 2nd ITERATION
orc.b $t1, $t0      IF ID EX ME WB
beq $t1, $t3, LOOPB IF ID -> EX ME WB
lw $t0, 0($a0)      IF -> ID EX ME WB
    
```

(b) The code below should have executed more slowly on the illustrated implementation. Explain why. *Hint: The only difference in the code is the branch instruction.*

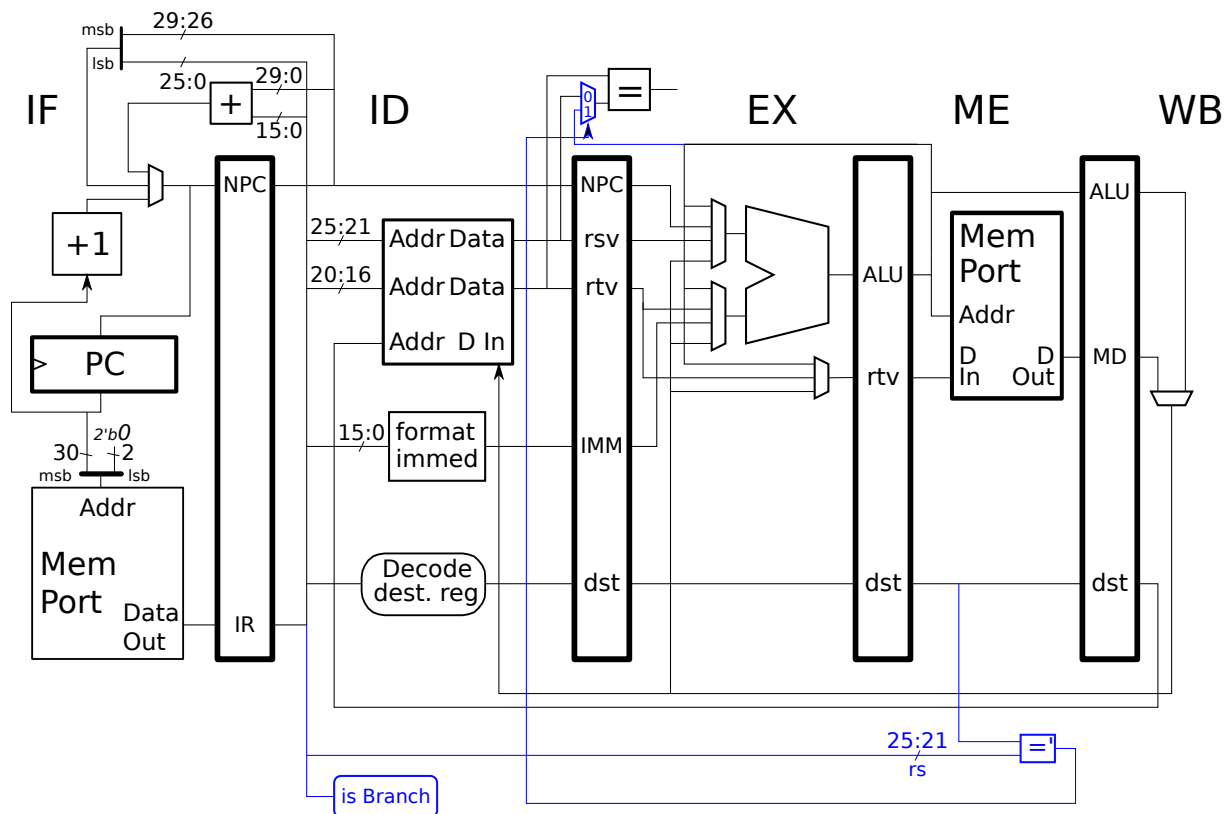
```
lw $t0, 0($a0)
LOOPB:
addi $a0, $a0, 4
orc.b $t1, $t0
beq $t3, $t1, LOOPB
lw $t0, 0($a0)
```

Explain why the code above executes more slowly.

The comparison unit used to evaluate the branch condition (the  box in the ID stage) can bypass an **rs** register value (though only from **ME**) but not an **rt** value. In both code fragments the **beq** needs the value of **t1** written by the **orc.b**, but in the code fragment immediately above **t1** is the **beq rt** register and so there is no bypass path.

Problem 2: Appearing below is the implementation used in the previous problem. Add control logic for the branch condition multiplexor (shown in blue). Feel free to insert an `is Branch` logic block to detect the presence of a branch based on the instruction opcode. For an Inkscape SVG version of the implementation follow <https://www.ece.lsu.edu/ee4720/2022/hw04-br-byp.svg>.

The solution appears below in blue. Note that it was not necessary to check whether the instruction in `ID` is a branch because only a branch instruction would use that mux. It is assumed that logic already exists to generate a stall signal when there is a dependence with the instruction in `EX`.



Problem 3: Appearing below is our MIPS implementation (the one we use, we're not taking credit for inventing it) with an `orc.b` unit in the EX stage. Unlike the first problem in this assignment, here the `orc.b` instruction is executed by its own unit, not by the ALU. One reason is because `orc.b` is fairly easy to compute, and so its output can be available much sooner than the ALU's output. In fact, it will be available early enough to be bypassed to ID for use in determining the branch condition.

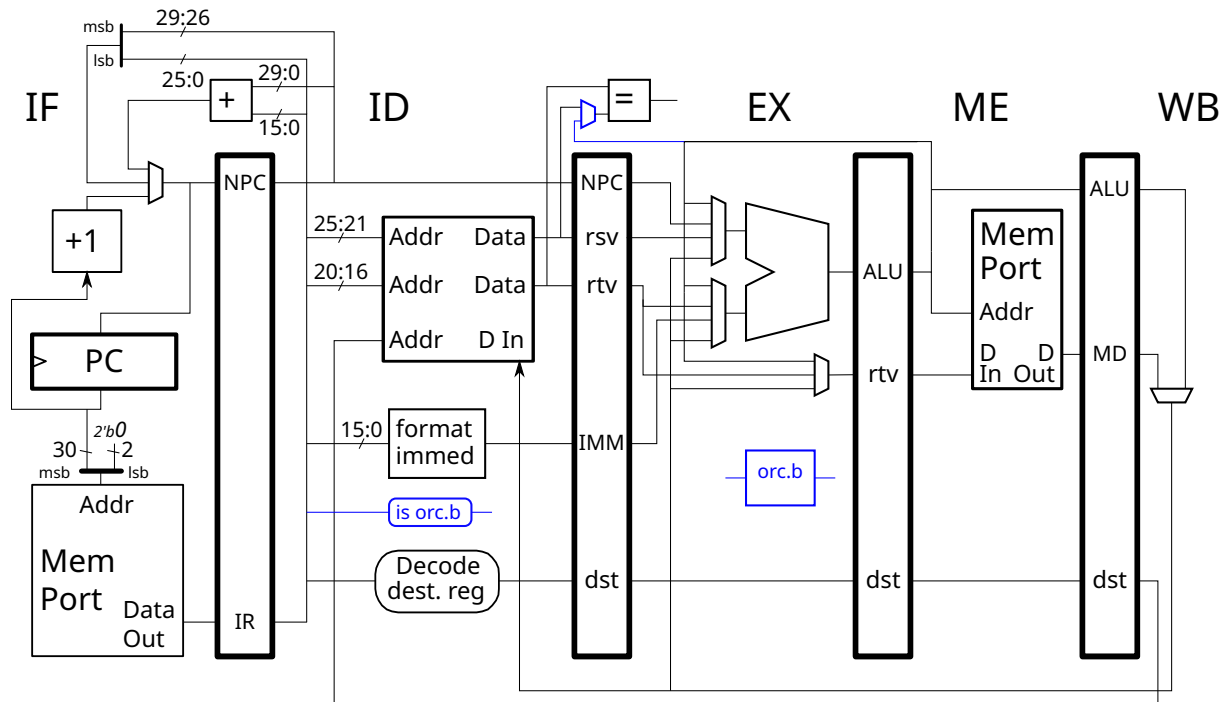
Connect the `orc.b` functional unit so that it can be used by `orc.b` instructions. Paying attention to cost, connect it so that the following bypasses are possible: (1) A bypass so that an immediately following dependent branch does not stall. This would eliminate a stall in a solution to Problem 1, and avoid a stall in Case 1 in the code fragment below. (2) Bypasses to the next two arithmetic/logical instructions. See Case 2 below.

When weighing design alternatives assume that one pipeline latch bit cost twice as much as one multiplexor bit. Don't overlook opportunities to reuse existing hardware. The Inkscape SVG source for the diagram below is at <https://www.ece.lsu.edu/ee4720/2022/hw04-orc.svg>.

```
# Case 1
orc.b R1, r9
beq R1, r10, TARG
```

```
# Case 2
orc.b R1, r9
add r2, R1, r3 # Bypass from ME
xor r4, R1, r5 # Bypass from WB
or r6, R1, r7 # No bypass needed.
```

- ✓ Connect `orc.b` unit so code above executes without a stall.
- ✓ Show control logic for any multiplexors added. (Control logic does not need to be shown for the branch condition mux.)
- ✓ As always, avoid costly, inefficient, and unclear solutions.



The solution appears on the next page.

The solution appears below in [turquoise](#). The `orc.b` instruction has one source, the `rs` register value. That value is taken at the output of the upper ALU mux, this way we can take care of the existing bypass paths and control logic.

The output of `orc.b` connects to two places. For the important `orc.b / beq` use case the output is connected directly to the branch = mux for a stall-free bypass to the branch instruction. For other cases we need to put it on a path to the register file. The chosen solution first puts it in the `rtv` mux, because that path is not otherwise used by the `orc.b` instruction. Then, in the `ME` stage a mux is used to put the value on the "main" path back to the register file. The `is orc.b` signal is used as a select input to the mux. Of course, the signal travels with the instruction by using pipeline latches.

There are two efficiency issues worth noting. First, it is possible to place a mux between the ALU and the pipeline latch, and have the `orc.b` output connect to an input to that mux. That would have received full credit. But, it would be correct to argue that the output of the ALU was on the critical path and so that should be avoided. That is why in the solution below the path from the ALU output to the latch is not touched. For the same reason the path from `ME.ALU` to the memory port address input is not touched.

