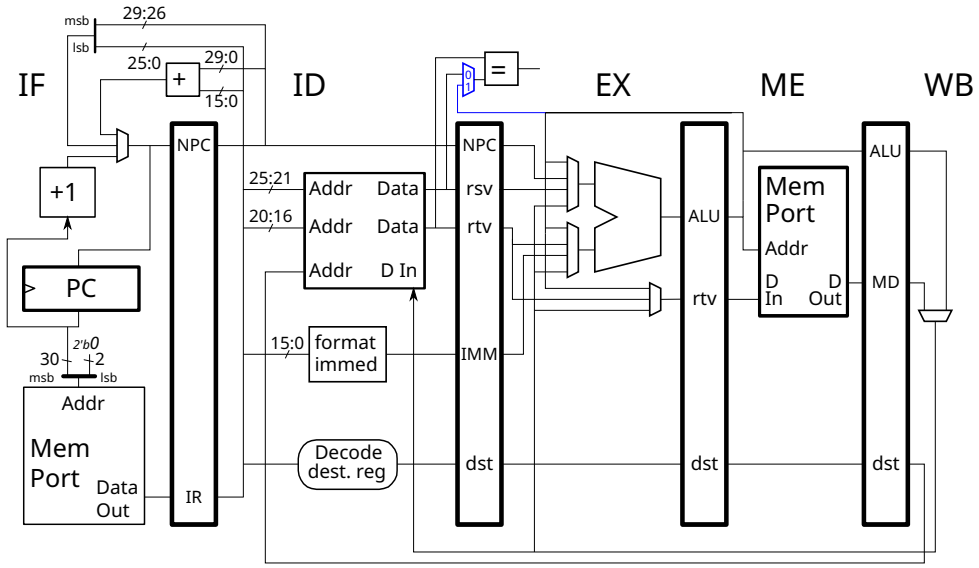
LSU EE 4720 **Homework 4** **Due: 25 March 2022**

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.

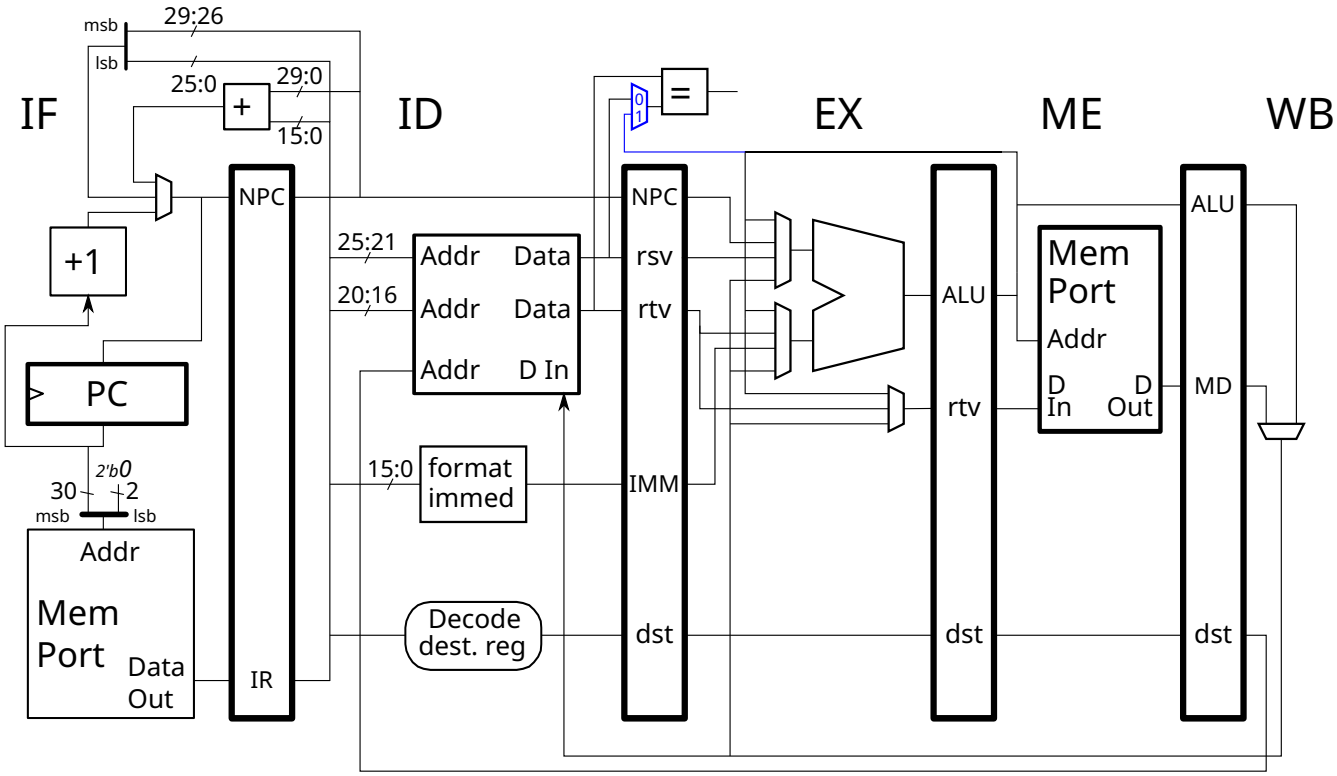
```
        lw $t0, 0($a0)

LOOPB:
        addi $a0, $a0, 4

        orc.b $t1, $t0

        beq $t1, $t3, LOOPB

        lw $t0, 0($a0)
```

(*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.

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`.



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.