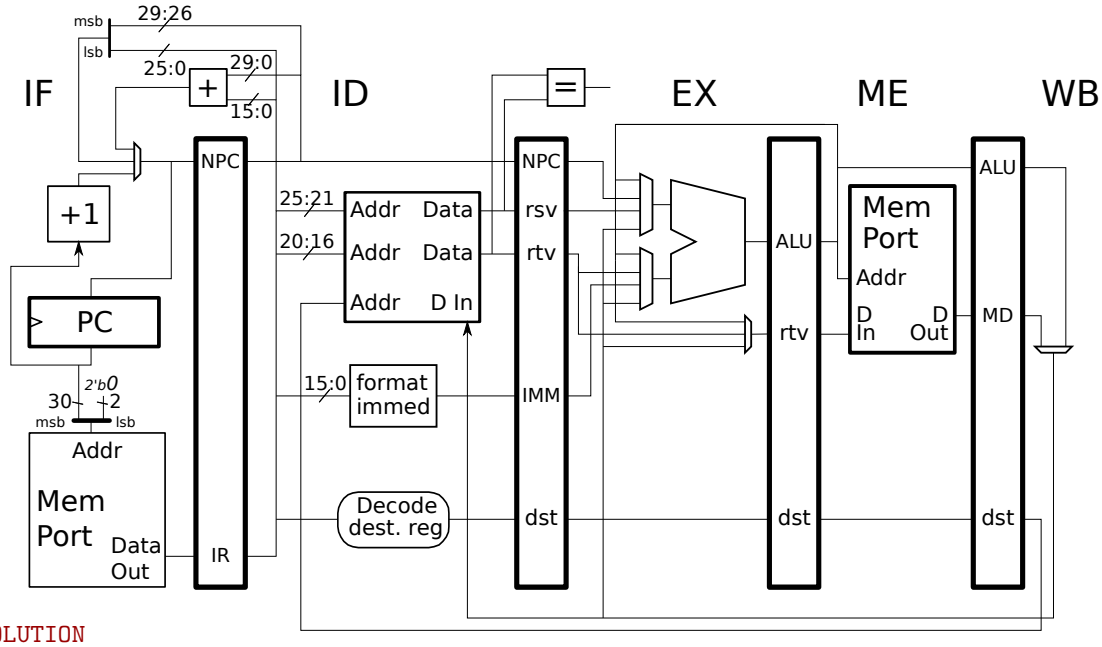
LSU EE 4720 **Homework 2** Solution **Due: 26 February 2016**

Problem 1: The code fragment below is to execute on the illustrated MIPS implementation. Unfamiliar instructions can be looked up on the MIPS ISA manual linked to the course references page. Show the execution of the code fragment below on the illustrated MIPS implementation. All branches are taken.

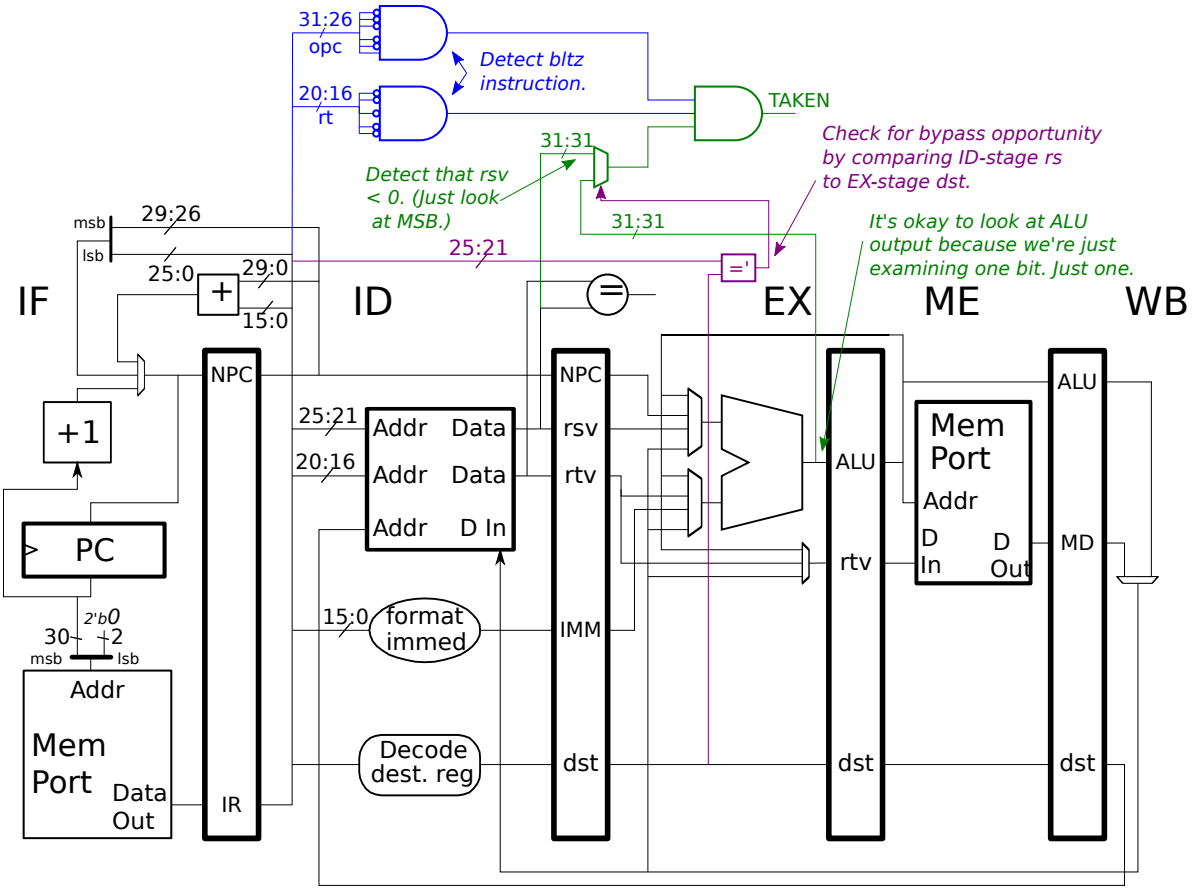
- Pay close attention to dependencies, including those for the branch.
- Note that unnecessary stalls are just as incorrect as not stalling when a stall is necessary.




```
# SOLUTION
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
 add r4, r2, r3      IF ID EX ME WB
 lw r6, 8(r4)           IF ID EX ME WB
 sub r1, r6, r5            IF ID -> EX ME WB
 bltz  r1 TARG                IF -> ID ----> EX ME WB
 and r8, r7, r10                    IF ----> ID EX ME WB
 or  r11, r12, r13
 xor r14, r11, r8
TARG:
 sw r1, 0(r2)                                  IF ID EX ME WB
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
```

Solution appears above. The `lw` does not stall because the `r4` register value can be bypassed. That's not possible for the `sub`, since there is no bypass for the `r6` value it needs because the `r6` value isn't available until the end of cycle 4, too late to bypass. The `bltz` stalls two cycles, waiting for the value of `r1` that it needs to arrive in `ID`, where it needs it. The `and` is in the delay slot and so executes normally. Since the branch resolves in `ID` (as we can tell by the connections from `ID` to the PC mux) the target, `sw`, will be in `IF` when the branch is in `EX`.

Problem 2: The implementation below (which is the same as the implementation for the previous problem) lacks hardware needed for the `bltz` instruction. In this problem design such hardware as described in the parts below. *Note: An Inkscape SVG version of the implementation can be found at* `https://www.ece.lsu.edu/ee4720/2016/mpipei3.svg`.



(*a*) Add the hardware needed to detect when a `bltz` is taken. The hardware should have an output labeled `TAKEN`, which should be set to logic 1 if there is a taken `bltz` in `ID`. Include control logic, including the logic for detecting `bltz`.

The solution appears above. The logic in blue is used for detecting a `bltz` instruction. That instruction has an opcode of 1, but also requires that the `rt` field be zero. In the solution above AND gates are used to detect these, rather than the usual =bltz and =0 boxes. Either would be correct. The branch is taken if the `rsv` is negative, that can be detected by looking at the MSB. That's shown in green (the solution to the part below is also shown). The rightmost AND gate detects the taken condition for this branch.

(*b*) The solution to the previous problem (not the previous part to this problem) should have included a stall due to the branch instruction. Add a bypass path to the hardware designed above so that the branch from the previous problem can execute without stalling.

Logic also shown above in green. In particular the green mux and the connection to the output of the ALU. We can get away with using the ALU output in this case because we only need examine one bit. If we needed to look at all 32 bits there would not be enough time left in the clock cycle.

An important thing to remember is that this is one of the few cases where we can get away with using the ALU output. See Problem 3a.

(*c*) Design control logic for the bypass path.

Logic shown in purple.

Problem 3: The code below is similar to the code from the first problem, the only difference is in the branch instruction. In this problem explain some bad news and good news about that branch.

```
 add r4, r2, r3
 lw r6, 8(r4)
 sub r1, r6, r5
 beq r0, r1 TARG
 and r8, r7, r10
 or  r11, r12, r13
 xor r14, r11, r8

TARG:
 sw r1, 0(r2)
```

(*a*) The bad news is that adding bypass paths for a `beq` would not be a good idea, even though adding bypass paths for the `bltz` was a good idea. Explain why.

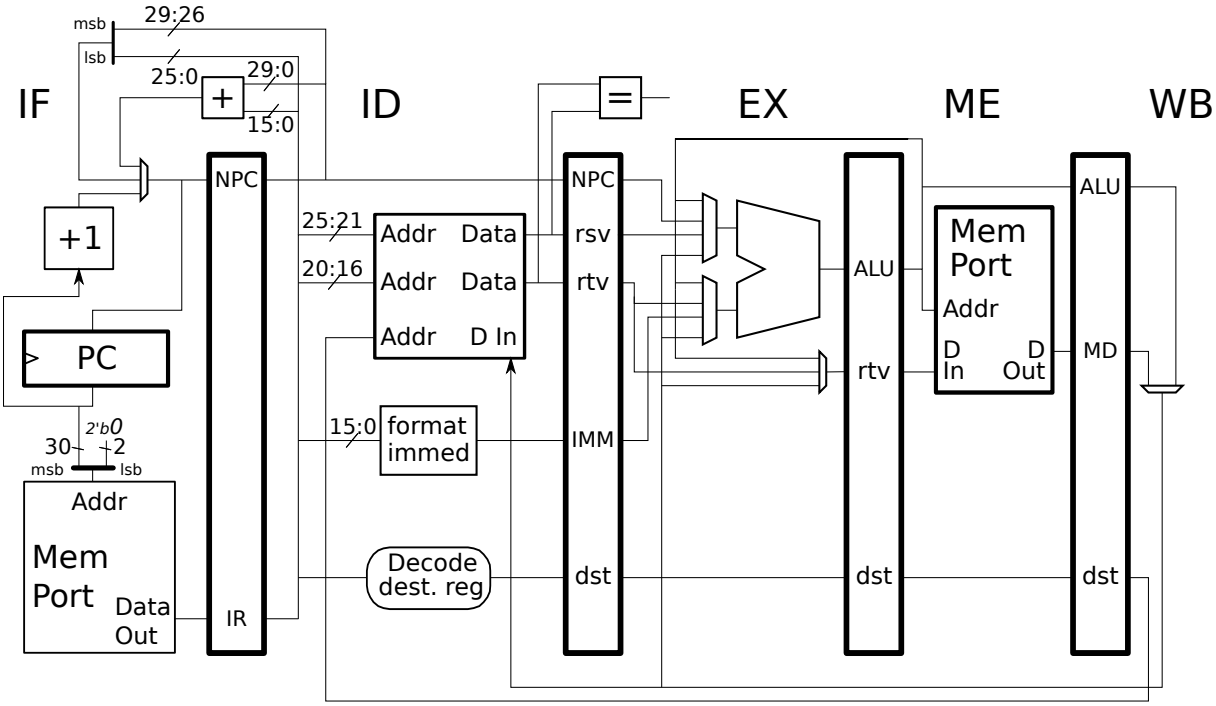
Testing the condition for a `bltz` instruction is simple: just check if the MSB of the rs value is 1. That's a good thing because the bypass path added in the previous problem was from the output of the ALU, and so was available at the end of the cycle. In contrast, the `beq` must compare two registers, which requires about 7 layers of logic. (An XNOR to test equality of each bit pair and a 32-input AND gate to check that all XNOR output are 1. The 32-input AND gate might be realized with five layers of two-input and gates.)

As pointed out in class, the ALU output will not be available until close to the end of the clock period, and so there is no time for testing equality.

Grading Note: Many answered that the difficulty was in the bypass paths since two values had to be bypassed. Though that's true it ignores two things. First, for `bltz` *all we need is the MSB, so the difference between the two cases is more like a factor of 64 than a factor of 2 in bypass path cost. It also ignores the more significant problem of testing equality, which stretches the critical path. Nevertheless, full credit was given for the "two values" answer.*

(*b*) The good news is that the program above can easily avoid the stalls by just changing the branch instruction. Explain how. (Of course, it should go without saying that the changed program must do the same thing as the original one.)

Change the branch to a `beq r6, r5 TARG`, thus avoiding the need for a bypass. The `sub` can not be removed because the `sw` uses the value of `r1`.


IF
ID
EX
ME
WB
msb
lsb
29:26
25:0
29:0
15:0
+
=
+1
PC
NPC
IR
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out