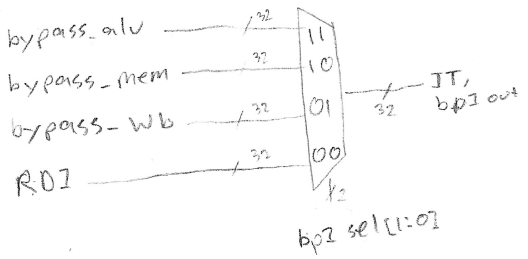
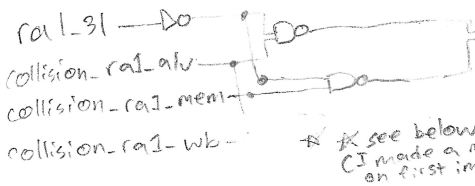
Fully bypassed

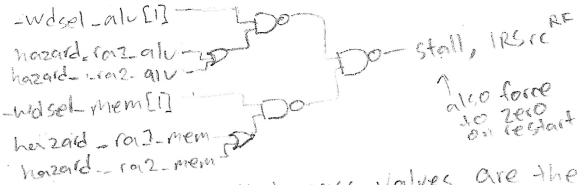




* see below (I made a mistake on first implementation)

bypass mux signal logic, giving priority to collisions with more recent instructions (ALU > MEM > WB).
A similar circuit calculates the bypass for ra2. and bp2sel[1:0]

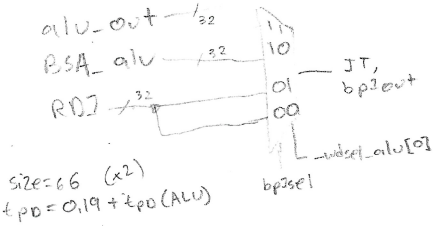
Note *

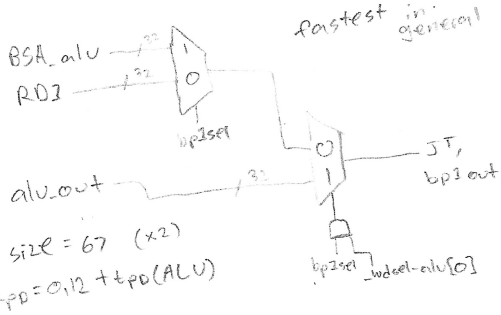


If the collision is with the result of an LD or LDR instruction, we have to wait until it is in the WB stage, so stall if it is in ALU or MEM stage.

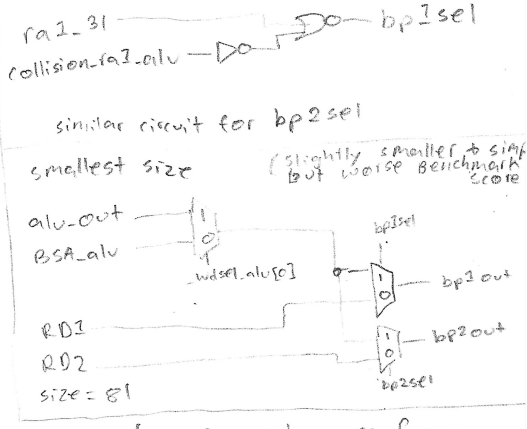
The ALU and MEM bypass values are the _wdsel[0] muxed values between BSA and alu_out
WB bypass is exactly wdata, which is _wdsel[1:0] selected which considers memory output mrd

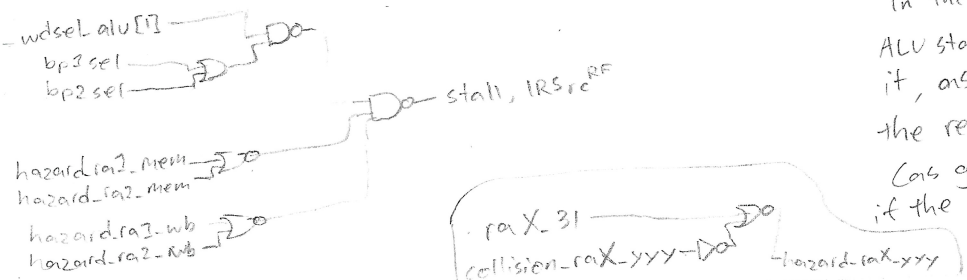
Partially Bypassed





this is the faster implementation, assuming alu_out is the bottleneck input.





In the partially bypassed case, we bypass from ALU stage if there is a non r31 collision with it, as long as it is not a collision with the result of a LD or LDR instruction (as given by _wdsel[1]), we stall either way if the collision is with MEM or WB stage.

Note: a collision is a matching reg addres a hazard is if that match is not r31

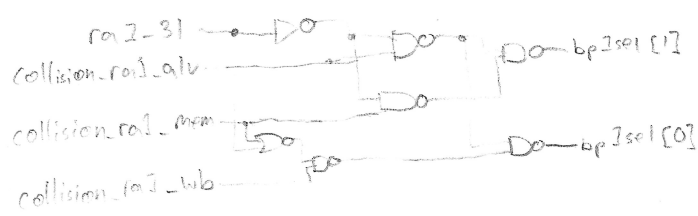
• stall was giving me a buch of problems when the program just started up; of course setting stall to constan 0 gave incorrect program results, but when I added logic to force it to 0 when reset, its value became invalid and ruined everything. I tried a shorter clock period, so the reset signal was stretched over more cycles, and this seemed to allow stall to stay at 0 for long enough until the valid signals had filled the system.

---

* can use
raX_31
collision_raX_yyy
hazard_raX_yyy
→ time optimization might be possible as well

or can reuse signals from bp1sel[1:0] and bp2sel[1:0] computation for size optimization

**



with this fix, full bypassing makes much more sense than partial.