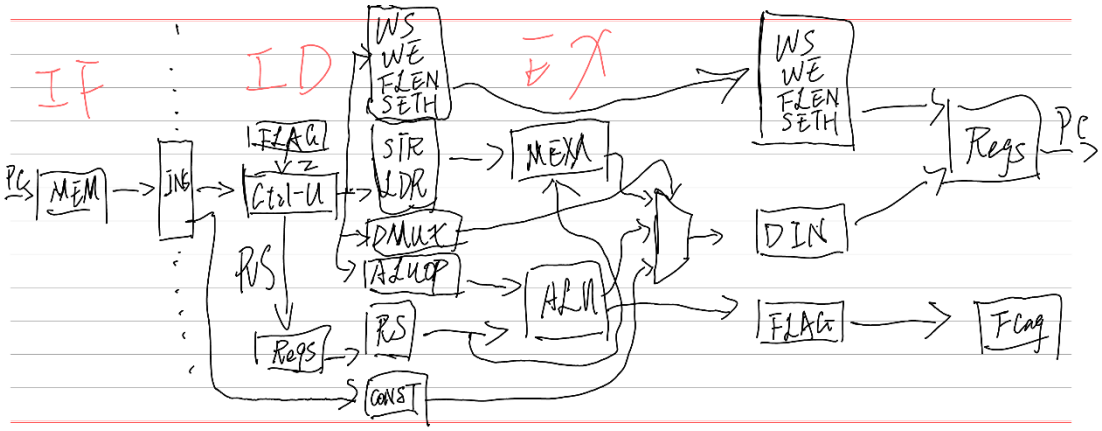
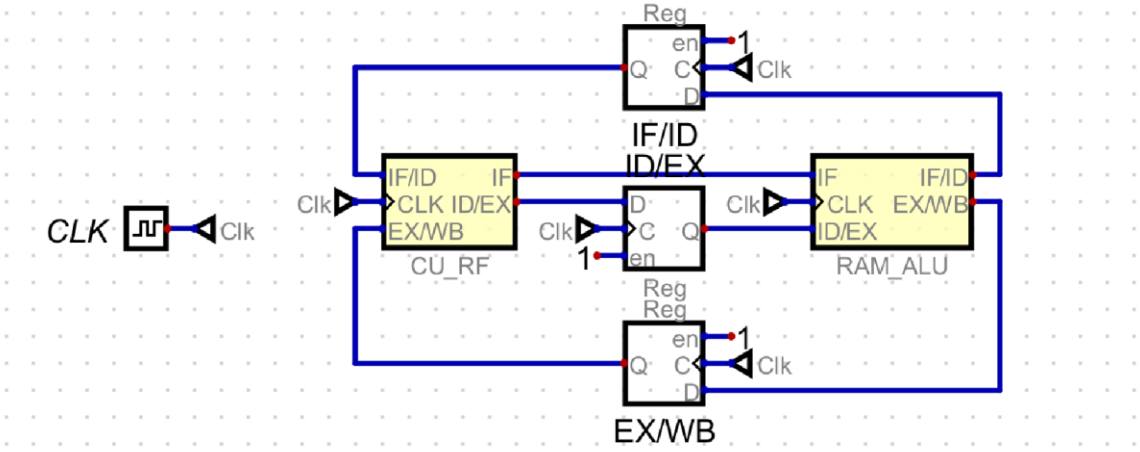
Our submission attempted and completed all three basic tasks, but we attempted none of the extension tasks.

The CPU has 4 stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX) and Write Back (WB). The main difference between this design and the one discussed in the lectures is the merging of EX and MEM stage. This is because in QuAC ISA, every single instruction only requires one of ALU computation and RAM accessing. Because of the merging, there will not be any load-use hazard.

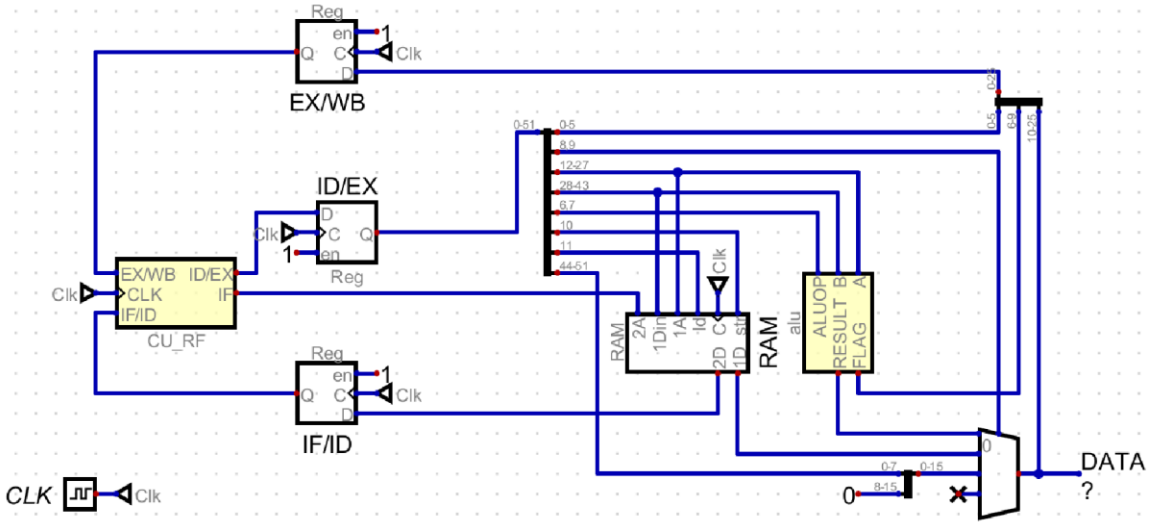
The pipelined version was modified from the single-cycle version. Before doing anything realistic, we first worked out what the data flow of the pipelined CPU should be like.



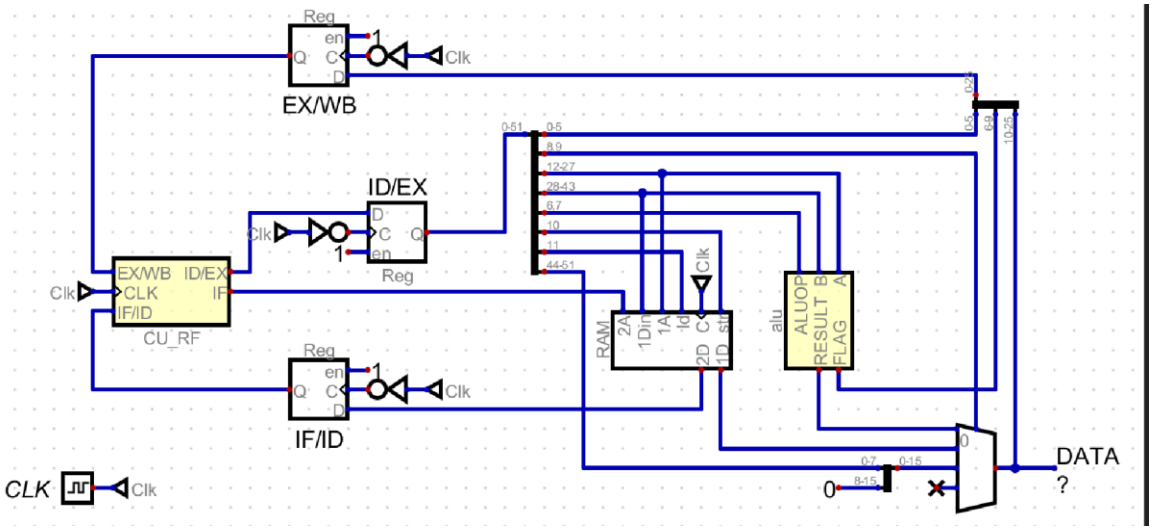
Based on this, we first rearranged the layout of the CPU and added the stage buffers, which completed task 1.



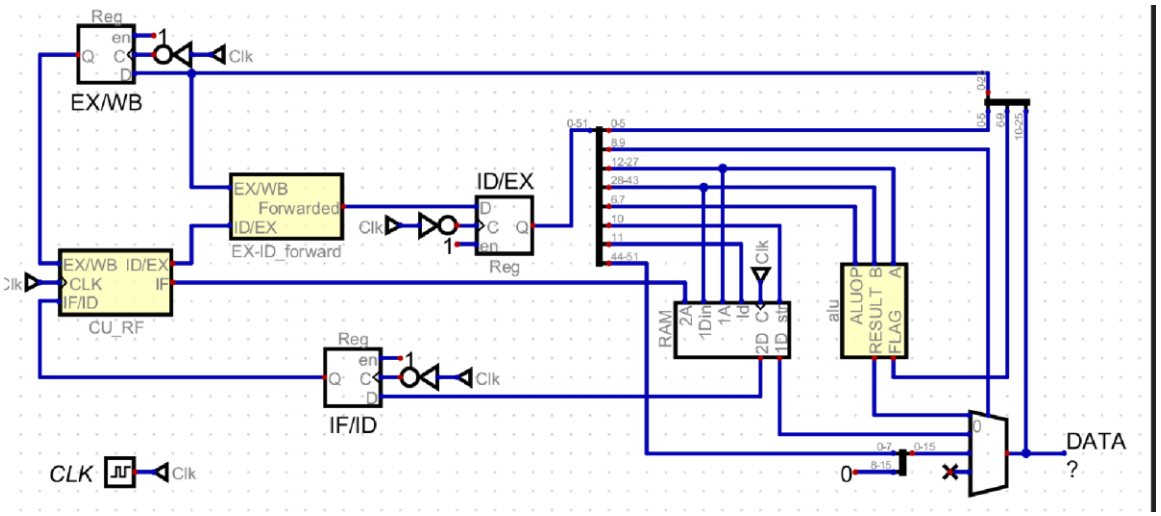
We later put RAM and ALU back to the upper-most layer because otherwise we could not modify the contents in RAM when running simulation.



There are 2 RAW hazards to be resolved, and the first is between WB and ID. As mentioned in the lecture, this can be resolved by writing back at the first half of the cycle and reading at the second half. We achieved this by letting the stage buffers read in data at the falling edge, which is half cycle after the write-back of data to the register file. Notice such change may also improve the speed of CPU slightly, if the clock stays in high stage and low stage for similar amount of time, because some of the works are now done in the second half of the cycle.

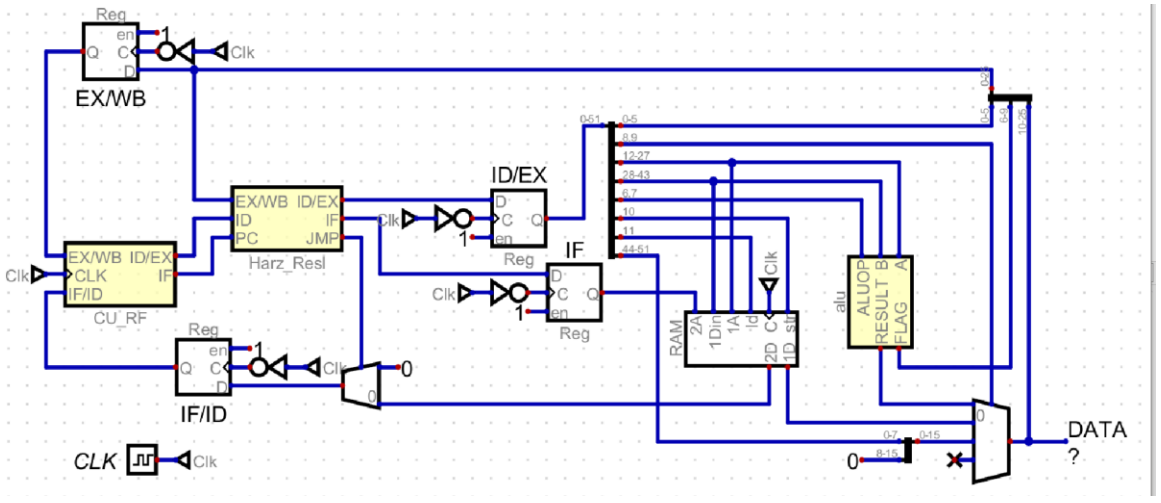


It is necessary to add forwarding to resolve RAW hazard between WB and EX.



In the design of the single-cycle version, the control unit will decode conditional-execution instructions that should not be executed to Nop. Based on this assumption, anything coming out from the EX stage has sufficient and valid information to provide updates to the next instruction to be executed. After the addition of forwarding, task 2 is completed.

Finally, we will need to detect control hazards and add bubbles accordingly. Detecting control hazard is basically the same thing as detecting instructions writing to PC. As mentioned above, the result from the EX stage is enough to decide whether PC will be modified, so we can integrate such detection into the forwarding unit, which makes it the Hazard Resolution Unit. And this is the final implementation.



Now, apart from data forwarding, the unit will also detect control hazard and zero out the next 2 instructions, which should not be executed in most cases.

The clock rate is basically upper-bounded by the speed of reading/writing of the RAM. And there’s hardly any improvement to be made unless making it out-of-order.