Jesse Chen, Ojan Thonrycroft

Assignment 3

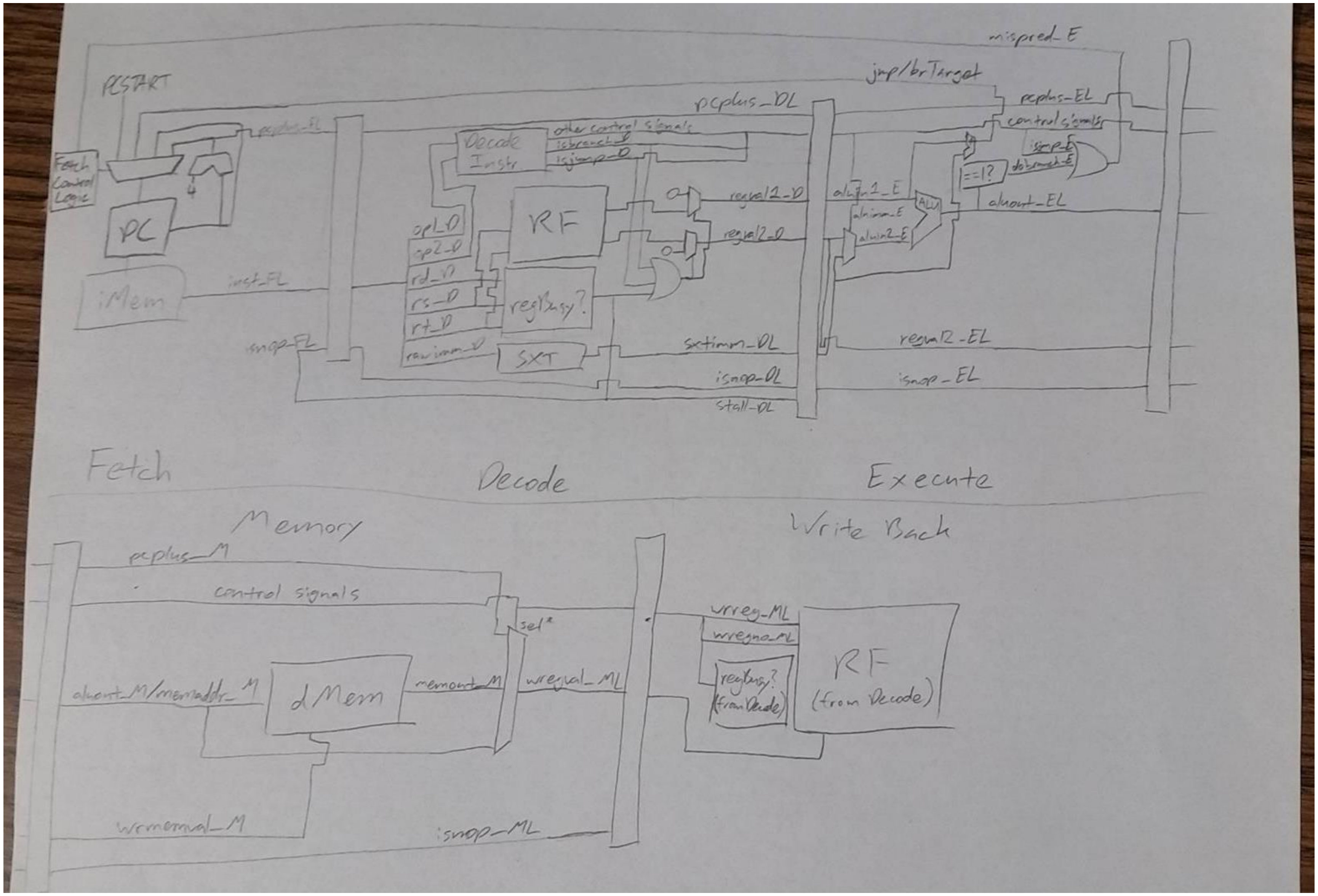
For the most part, the only major design options to consider were how to handle data dependencies and branch and jump instructions. Aside from those components, our design followed the design that was discussed in class and provided in the given frame. We could have changed the number of stages in the pipeline as well, but we decided it would be best to stick with the five stages in the classic RISC pipeline for our processor: instruction fetch, instruction decode, execute, memory access, and writeback.

For our design, we handled data dependencies by stalling the fetch stage and sending NOPs from the decode stage buffer until the data dependency was resolved. Stalling the fetch stage means that the everything inside the fetch stage buffer, and the decode stage as a result, will remain locked in place, so new instructions cannot move forward. Sending NOPs from the decode stage buffer ensures that the current instruction in the decode stage does not get executed. In order to detect data dependencies, we chose to implement the scoreboard method. The scoreboard keeps track of registers that have a pending write issued to them by previous instructions and marks them as busy. If a new instruction is trying to read from a busy register, a data dependency is triggered. Once the old instruction has finished writing to a register in the writeback stage, the scoreboard's value for that register is reset, and the new instruction can then finish decoding.

For branch and jump instructions, we also stalled the fetch stage but sent NOPs from the fetch stage as well, instead of from the decode stage. This ensures that no new instructions are fetched until the branch or jump instruction is resolved after the execute stage. Simultaneously, in the execute stage, the target PC is also calculated and fed back to the fetch stage, so the jump, and branch if it's taken, goes to the right address after the instruction completes the execute stage. As a result, two NOPs are sent through the pipeline for each branch and jump instruction, and we never have to flush the pipeline.

The most difficult problem we had was handling data dependencies. As simple as the scoreboard technique sounds, its implementation was somewhat tricky to get right. At first we were having trouble figuring out how to use the scoreboard register in both the decode and writeback stages. This obviously cannot work because that same register would be written to in two different sequential blocks. To solve this, we ended up declaring and reading from the scoreboard register in the decode stage to check for busy registers. Meanwhile, the wrreg and wregno signals from the decode stage were forwarded directly to the writeback stage in order to flag registers as busy. Then, when the writing instruction itself finally makes it to the writeback stage, the scoreboard can reset the status of the register being written to.

My contribution to this project consisted mainly of filling out the initial logic in all five stages, excluding anything directly involved with stalling and sending NOPs. Most of the initial logic worked, and I started working on the data dependencies, but my partner ended up figuring out most of that, along with implementing the branch and jump logic, which I thought were the most difficult parts. In terms of debugging, I believe we contributed roughly equally. In the end, I think my contribution was about 45%.



mispred_E
jmp/brTarget
PCSTART
pcplus_DL
pcplus_FL
Fetch Control Logic
4
PC
iMem
inst_FL
isnop_FL
Decode Instr
other control signals
isbranch_D
isjump_D
op1_D
op2_D
rd_D
rs_D
rt_D
rawimm_D
RF
regBusy?
SXT
0
regval1_D
regval2_D
sxtimm_DL
isnop_DL
stall_DL
pcplus_EL
control signals
isjmp_E
dobranch_E
==1?
aluin1_E
aluimm_E
aluin2_E
ALU
aluout_EL
regval2_EL
isnop_EL
Fetch
Decode
Execute
Memory
Write Back
pcplus_M
control signals
sel*
aluout_M/memaddr_M
dMem
memout_M
wregval_ML
wrmemval_M
isnop_ML
wrreg_ML
wregno_ML
regBusy? (from Decode)
RF (from Decode)