CDA 3101: Spring 2020
Project 2 Guide

This is just a suggested way to implement Project 2.
Students are not required to do it this way, but it would help.

1. Set up an array of integers for the instructions. You can have at most 100 instructions, so an array of size 100 should do.
2. Set up an array of size 32 for the register file.
3. Set up an array of size 32 for the data memory.
4. Set up all the required data structures.
   - Set up a structure for the instructions - We can translate the instruction and save it or translate it every time. This is up to you. If you were saving the instruction, this is what's required:
     - The instruction
     - The type of the instruction
     - rs, rt and rd registers
     - immediate field
     - branch target
5. Set up structures for the 4 pipeline registers.
6. The IF/ID pipeline register. This requires the following:
   - The instruction
   - PC+4
7. The ID/EX pipeline register. This requires the following:
   - The instruction
   - PC+4
   - branch target
   - rs, rt, rd registers
   - readData1 and readData2
   - immediate field
8. The EX/MEM pipeline register. This requires the following:
   - The instruction
   - ALU result
   - write data (data in rt register, for store word)
   - write register (the target register - rt or rd, depending on instruction)

9. The MEM/WB pipeline register. This requires the following:
    - The instruction
    - write data from memory (data read from memory for load word)
    - write data from ALU (data forwarded from previous stage, the ALU result)
    - write register (the target register - rt or rd, depending on instruction)
10. You can combine all the pipeline registers into a upper level structure called "state". You will need two of these - one to indicate the state at the beginning of the cycle and one to indicate the state at the end of the cycle (or you can maintain them separately. Up to you).
11. Create a structure for the Branch Predictor
    - The PC
    - branch target
    - state

    Create an array of these. All the instructions could be branches, so we need at most 100 of these.
12. Write some helper functions extract the information from the instruction. You might need one each for the opcode, rs register, rt register, rd register, immediate field, shamt and the funct field.
13. Write a helper function to get the opcode and funct field and get the instruction.
14. Write a helper function to get the register number and return the register name.
15. Write a helper function to print out a state.
16. Read in the program into your setup.
17. Populate the data memory. Calculate the address for dataMem[0]. This is the byte address of the first data word in the program. This is needed because all the load and store addresses will use this. It is an offset for the data memory index.
18. Populate the instruction structure using the helper functions. (if you are using an instruction struct).
19. Populate the branch predictor. Whenever a branch instruction is encountered, record the PC and the branch target. Set the state initially to Weakly Not Taken.
20. Set the PC to 0.
21. Implement the pipeline logic. Basically, you are moving the processor along by one cycle. Look at the current "state" and update the new state. Then print the new state. At the end of the cycle, the new state becomes the current state.
22. Try a program that just halts. That is, the only line in the program is the halt instruction. You would fill up the rest with NOOPs. Make sure this runs as expected and the program halts.
23. Add support for the other instructions (except branches). Test the simulator for cases where there are no hazards. Do this one instruction at a time. Please remember to update the register file at the end of the WB stage and the data memory at the end of the MEM stage.
24. Try and inject stalls into the pipeline. Do this without checking for hazards.

25. Add the logic for mitigating data hazards through stalling. You can check for hazards by looking at the source registers at the beginning of the EX stage. If these registers are the same as the target registers for the previous two instructions (ones at the MEM and WB stage), then we have a hazard (or two).

26. Once you have added support for stalling, add support for forwarding. We have already established the conditions for hazards. All we need to do is copy over data.

27. Test this thoroughly.

28. Add support for branching. First implement branching through stalling. When we recognize a branch at the ID stage, stall until we decide if the branch is taken. Then update the PC with the required target address.

29. Add support for Predict Not Taken. Always fetch the next instruction. If it is determined that the branch is taken, scrub out the wrongly fetched instructions and change them into NOOPs. Fix the PC with the target address.

30. Add support for branch prediction. When we see a branch at the ID stage, look up the PC in the branch predictor. If the state is "taken", update the PC to the branch target, otherwise, let it be the next instruction. When the decision on the branch is made, check if we predicted correctly. If we did, reinforce the state. If not, move the state in the right direction. Then scrub out the wrongly fetched instructions and fix the PC.

31. Make sure you stop and test at each stage. This is not a project where you want to test once at the end.

32. Make sure you make backups at every stage. Using a version control system like git is recommended.