In this project, I ran through machine code, blocks, to pipelining.

In Lab 4, I was responsible for the control unit, instruction memory, and the sign-extension. These modules play an important role as they connect the register and the ALU. Therefore, I became clearer about how this whole CPU works. When working on the control unit, I learned what each control signal is for and how they work. Then I decoded the binary machine code by listing different cases for each output. During debugging, I learned from TA that if-else if statements will easily report errors, as I couldn' t ensure all situations. It would be best to use case statements with default cases so that the compiler can detect all cases.

In the sign-extension module, I initially decided to make ImmSrc itself a 12-bit source of immediate, which will already be decided in the control unit. However, after the lecture where mentioned that this is a one-bit value, I found it more direct and appropriate to decide where to choose from instruction in the control unit first and then do the extraction in the extension module. Apart from this, when I was searching on the given machine code on a whim, I found that the offset in instruction is half of the actual address difference in the Branch instruction. Then I realized that the immediate output is 13-bit, where the last bit is set to 0 by default, which should be revealed when decoding.

As the modules were finally tested by the other two teammates, when we came to the coursework, they would like to continue to work on implementing new logic to these modules. Another concern was that if we all work on implementation, it would be time-consuming to clarify why and how we make this, which turns out to be hard to debug and test at last. So, I chose to work on the machine code with Ethan as we all agreed to enhance our comprehensive ability to try different parts. Our first version was quite naive with only essential instructions. When I put them on the assembler, many problems turned out, for example, the assembler could only identify syntax when branching but not the offset. I also used an online RISC-V interpreter (https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/) to debug and trace the change of each register. After loads of attempts, the final version of the machine code was worked out as shown in the readme. When I was working on this part, I found it quite a logical thing that needs a clear mind on deciding the procedure. It also enhanced my understanding of different RISC-V instructions and their format.

When my teammates were implementing new designs, Ethan and I began to think about pipelining. After testing the machine code on the single cycle, I edited the output registers based on their preferences. Then we started implementing pipeline registers when we figured out newly built multiplexers and some changes to the signals. Next, we implemented top-level modules based on the five divided sectors, including all the blocks. This exercised my ability to systematically write top-level modules and became familiar with the grammar as well. This task also helped me to gain a deep understanding of pipelining. We met difficulties like how to deal with hazards, as we considered whether it would be an integrated pipeline without a hazard unit which we thought was too complicated to implement. We learned from the professor that adding NOP instructions could work as well.

In conclusion, this coursework built a quite clearer structure of CPU for me, combined with the lecture. It was also a nice experience to complete as a group. I benefited from teamwork as everyone had a specific role to work on simultaneously, and we helped each other as well. We had frequent calls to debug together, timely update our process, distribute the next tasks, and set timelines. Next time, I should improve my debugging skill, like understanding the error message, and be more careful with grammar to avoid some basic mistakes which might need much time to debug.