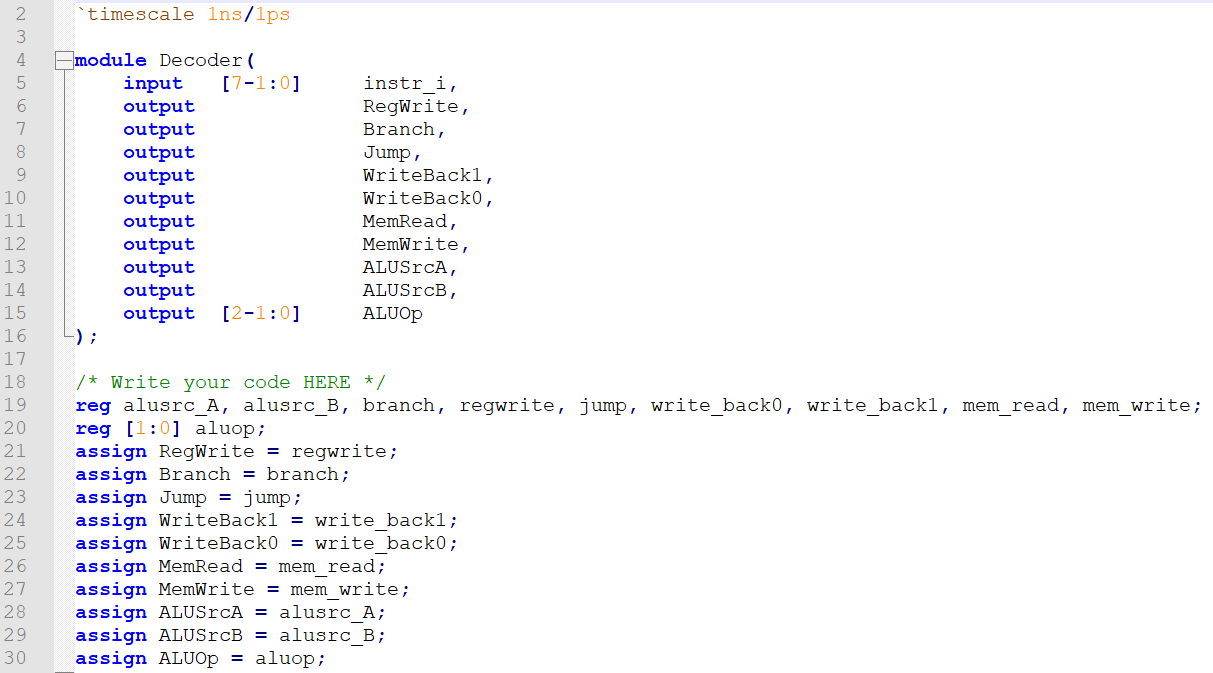
**LAB 4 : Single-Cycle CPU**

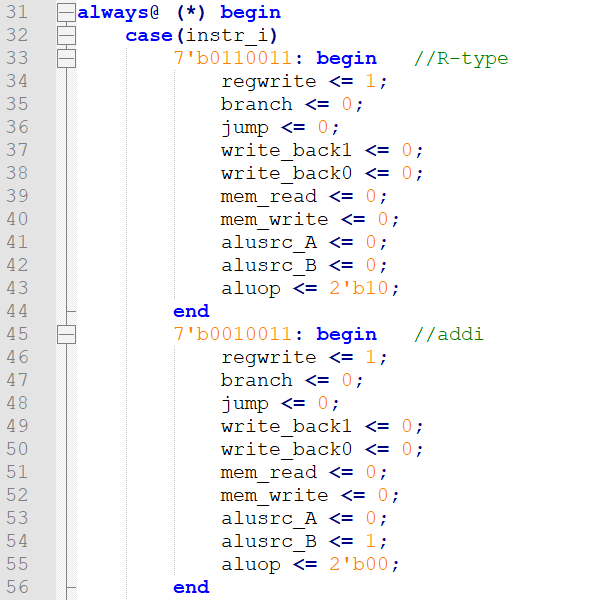
**Group 17 109550135 范恩宇 109550156 曾偉杰**

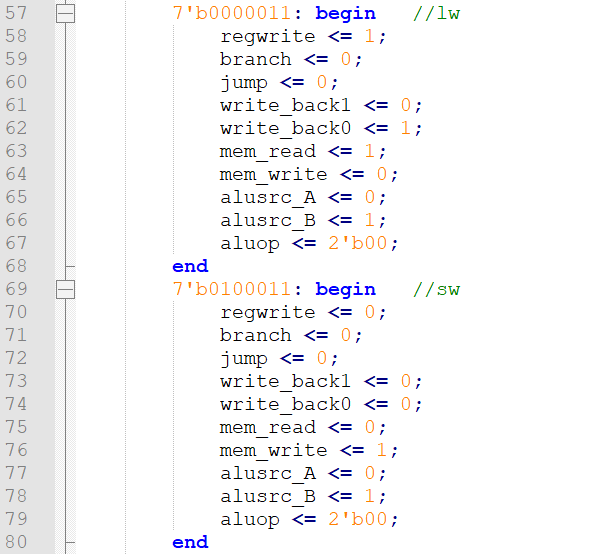
**Part I. Detailed description of the implementation :**

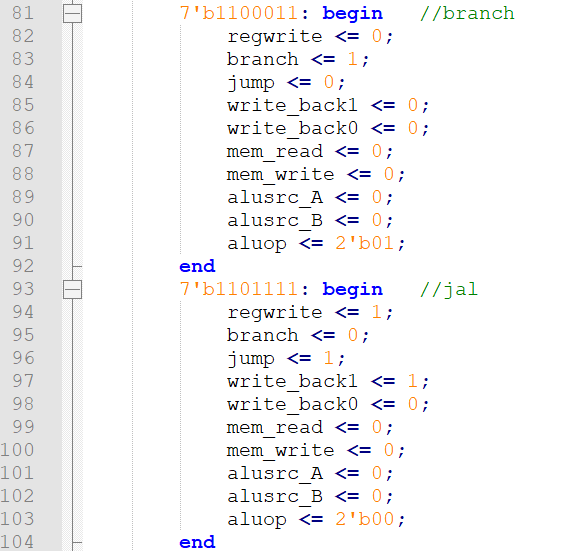
1. Decoder.v:

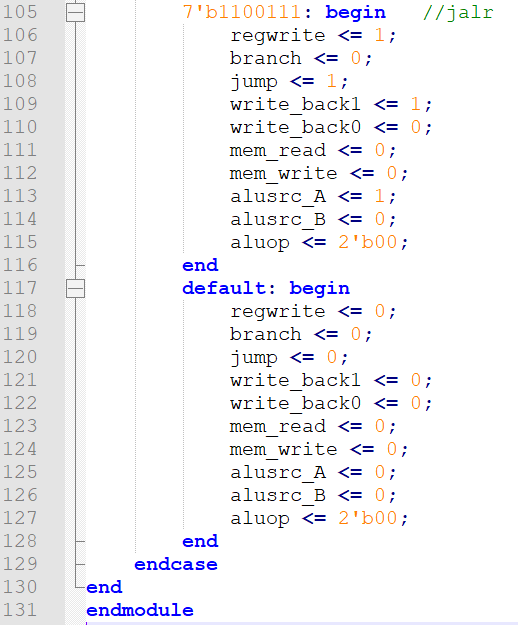
First assign each outputs with a temporary signal , then deal with the cases through different “instr\_i” . Then for different operations we asked to do in this lab , we set the required signals as true , and make it output corresponding ALUOp .







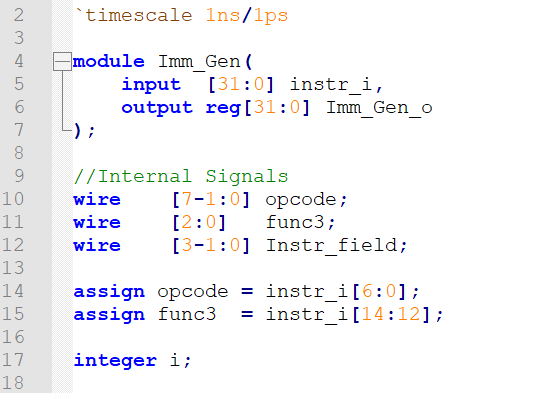


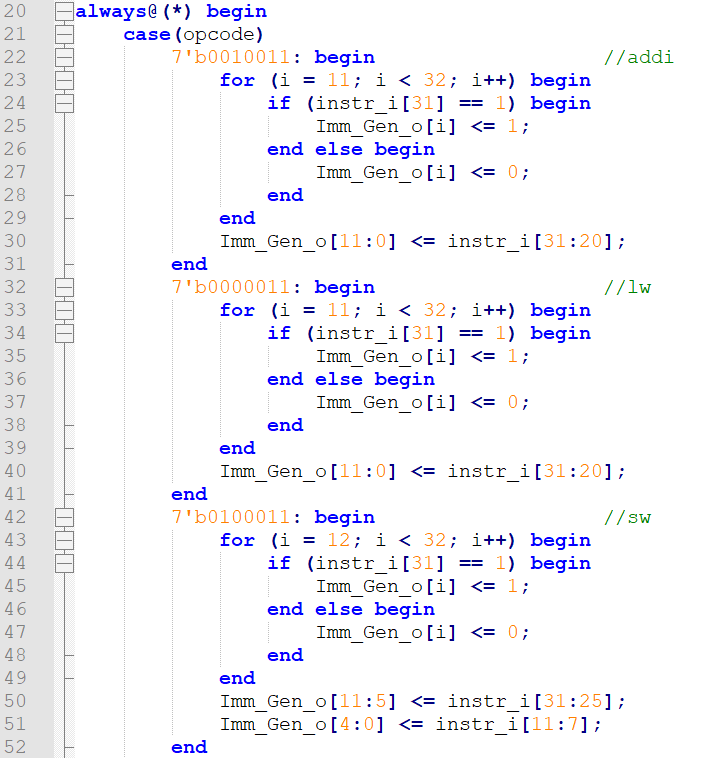


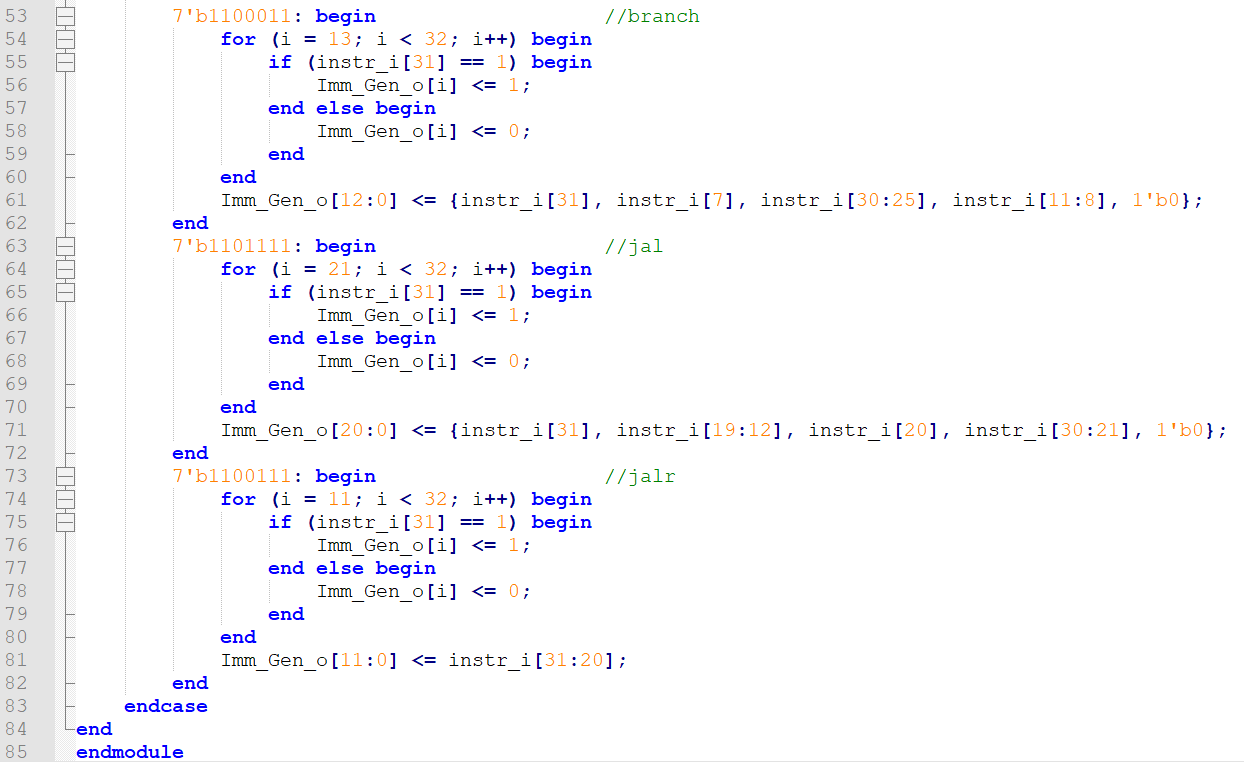
1. Imm\_Gen.v :

We choose different case of options through the “opcode” . Then in each case section , we set “Imm\_Gen\_o[i]” as 1 or 0 in the range that shown on the slide . Since this is a file for sign-extension , the former actually means that we make it do sign-extension by Identifying the sign bit of “Imm\_Gen\_o” . And if sign bit is 1, we fill 1 into the remaining bits , otherwise we fill 0 .

As for the range we didn’t do the former operation , we set it with different parts of “instr\_i[x:y]” , based on the type of the command and its corresponding structure that is shown on the slide .





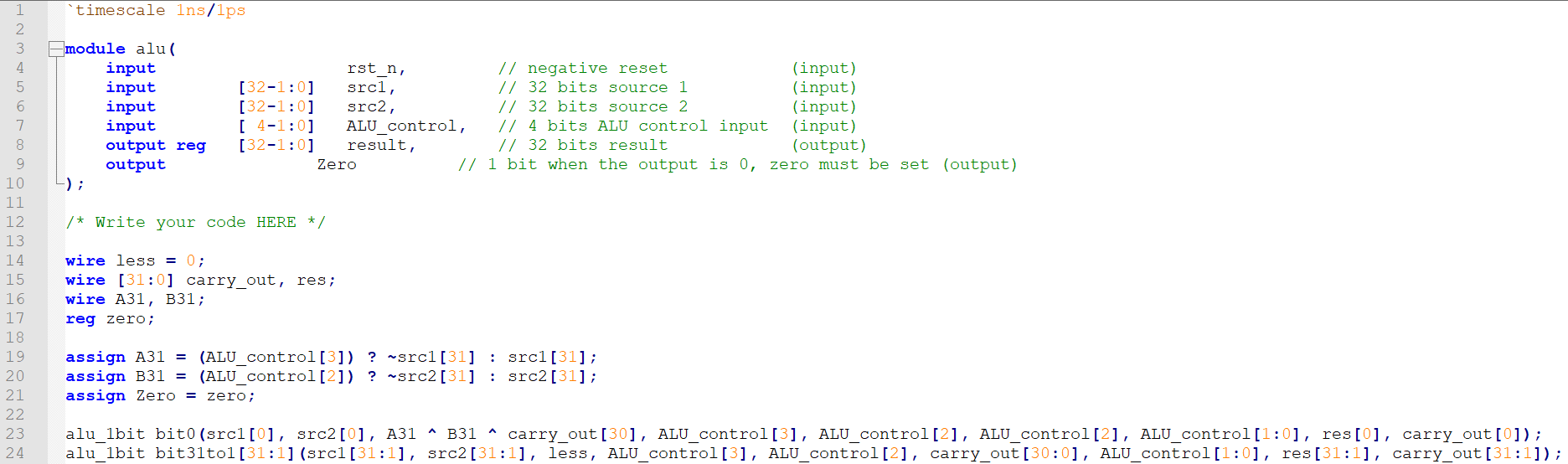


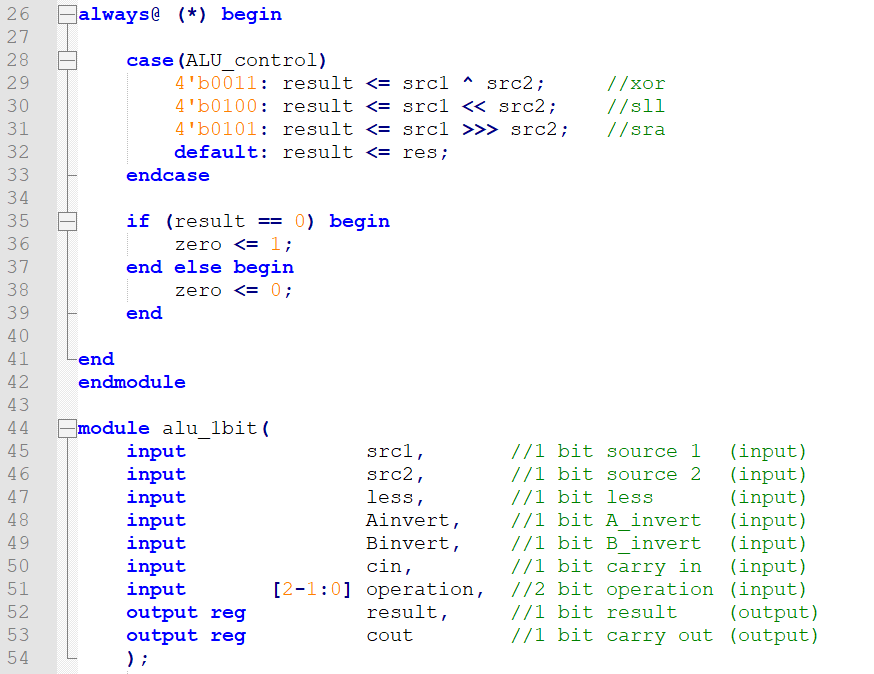
1. alu.v :

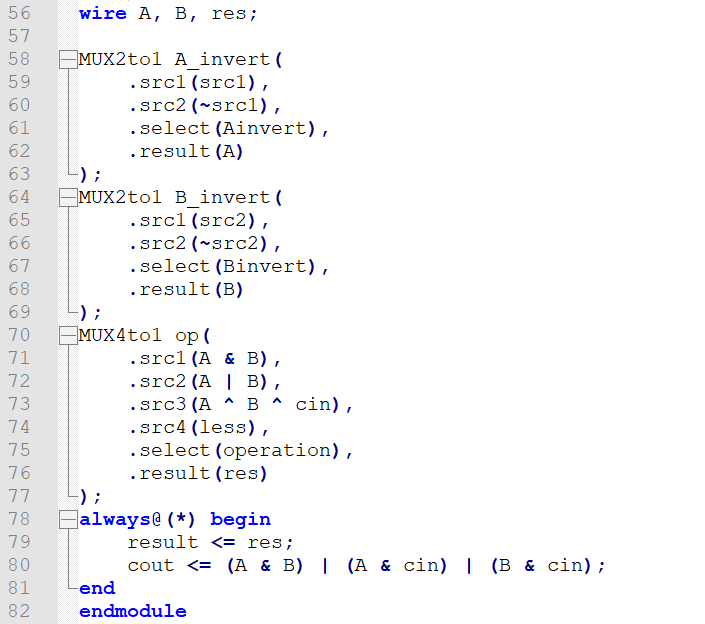
We construct the complete ALU with 32 1-bit ALUs . For each 1-bit ALU , we add one full adder , two 2x1 multiplexers and one 4x1 multiplexer just like we did in lab2 . Then the result and carry out will be outputs of the 4x1 multiplexer and full adder respectively .

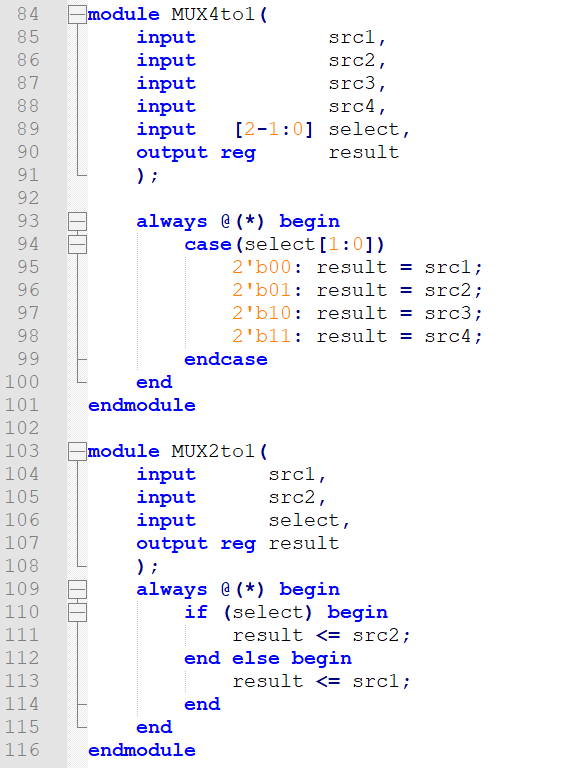
For the connected 1-bit alus , first let “A31” and “B31” be “ALU\_control[3]” and “ALU\_control[2]” determined by “src1[31]” and “src2[31]” respectively . As for the first 1-bit alu , set “less” = A31 ^ B31 ^ carry\_out[30] , “cin” = ALU\_control[2] . Then connect the 1-bit alus with the carryout of their former 1-bit alu as cin , and make the result of each 1-bit alu as the corresponding part of full result .

Finally for the running process , result will be the ones produced by the alus , but setting xor , sll , sra with self-defined operation . In the end , “zero” will be 1 once “result” equals to 1 .





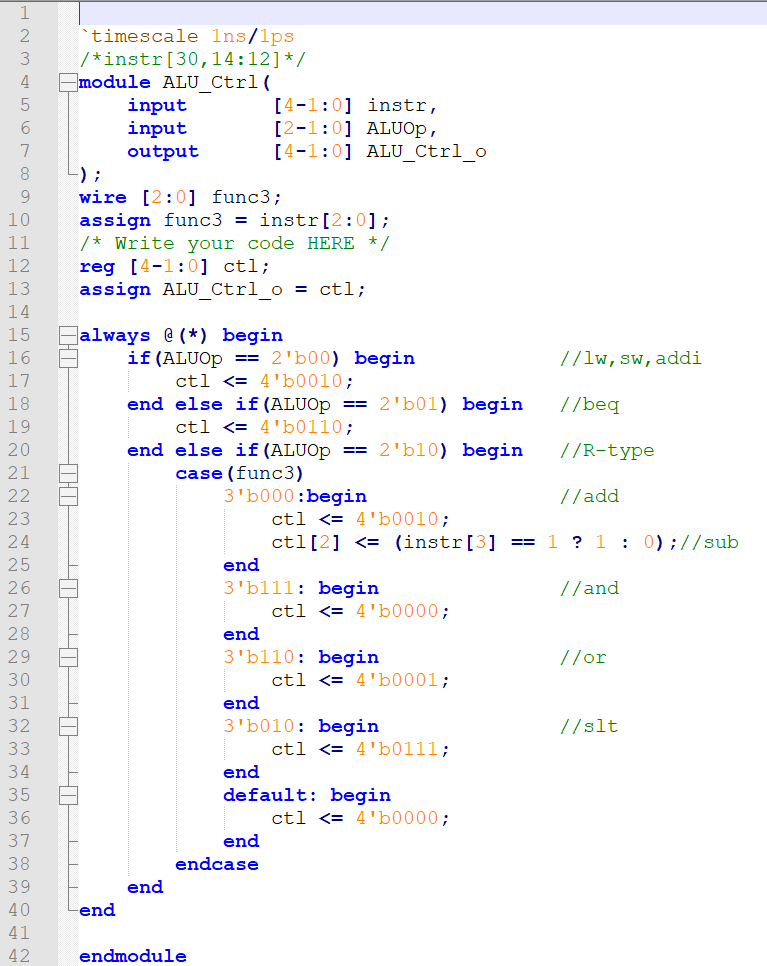




1. ALU\_Ctrl.v :

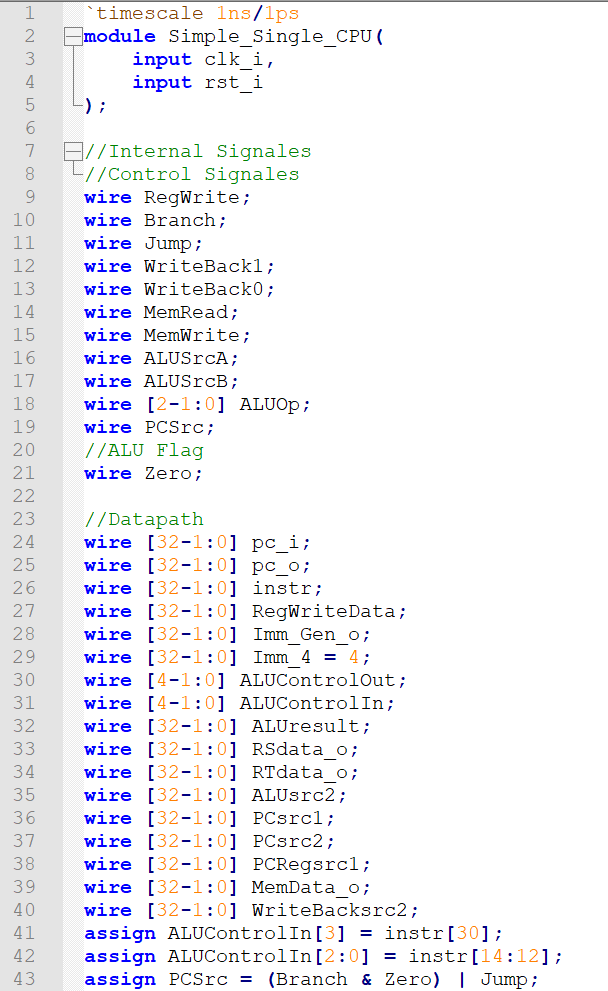
In case of making the output “ALU\_Ctrl\_o” be covered by undesired data , we first declare a register “ctl” and assign it to the original output “ALU\_Ctrl\_o”.

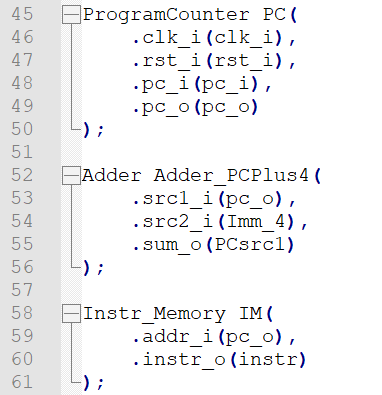
Then the main part is using ALUOP signal and function 3 to choose different operations , according to the lecture slides . 2’b00 is for S-type and addi , 2’b01 is for B-type , and 2’b10 is for R-type , then use function 3 to decide different operations in R-type . In part of R-type , although the slide doesn’t ask us to implement operations such like subtraction,and,or … directly , they’re still required for some commands , thus we also implement then . For subtraction of R-type , since (I30+fun3)[3] equals to alu\_control[2] and (I30 + fun3 = fun7) , we use “ALU\_Ctrl\_o[2] = (instr[3]==1 ? 1:0) to distinguish it from addition .



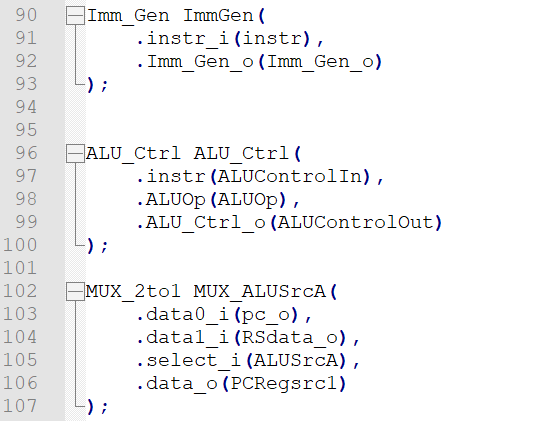
1. Simple\_Single\_CPU.v :

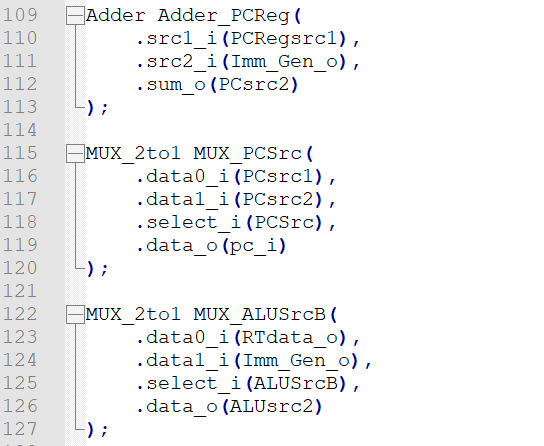
Basically consists of all the other modules whose corresponding unit exist in the structure of CPU . Connect each modules with each other through the required datapaths and assign them with specific signals to make the whole cpu run correctly .

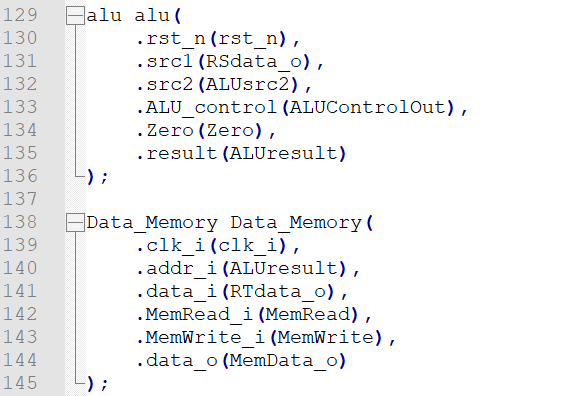


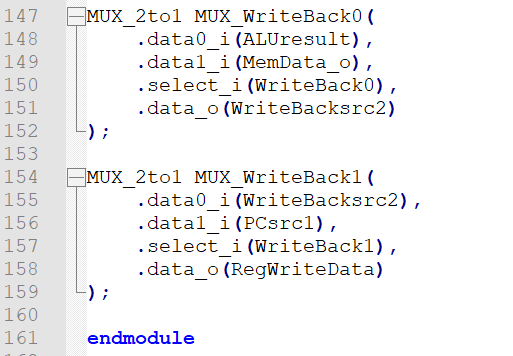




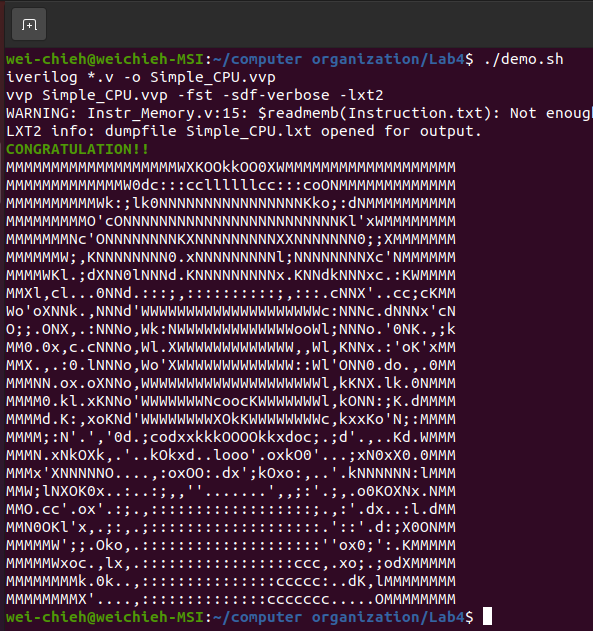








**Part II. Implementation results :**

****

**Part III. Problems encountered and solutions :**

Although Lab 4 is similar to Lab 3 , there are much more details that we need to be cautious of . The most significant problem we met is that we were once unable to read new pc signals , making we have no idea about how to debug . We thought that it may result from the nop of the commands . After trying to include the conditions of it in our code , we eventually got the desired result . In addition , the structure of this cpu is much more complicated than the last one , we do have to be more careful and patient to check the original diagram and connect the datapaths as the diagram asks we to do .