

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机体系结构 |
| 姓 名： | 周炜 |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术系 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3210103790 |
| 指导教师： | 卜凯 |

2023年 10 月 1 日

Lab 1: Pipelined CPU supporting RISC-V

RVI32 Instructions

周炜 3210103790

1. **Purpose**

The main task of this experiment is to complete the code and complete a five-level pipeline CPU that supports forwarding and predict-not-taken.

Understand RISC-V RV32I instructions

Master the design methods of pipelined CPU executing RV32I instructions

Master the method of Pipeline Forwarding Detection and bypass unit design

Master the methods of 1-cycle stall of Predict-not-taken branch design

master methods of program verification of Pipelined CPU executing RV32I instructions

1. **Task**

Design of Pipelined CPU executing RV32I instructions.

Design Hazard Detection Unit

Design datapath

Design CPU Controller

Verify the Pipelined CPU with program and observe the execution of program

Overall：**4 files** has blanks needed to be filled

1. **Experimental process and data recording**

**3.1 Experimental principle**

**Hazard**

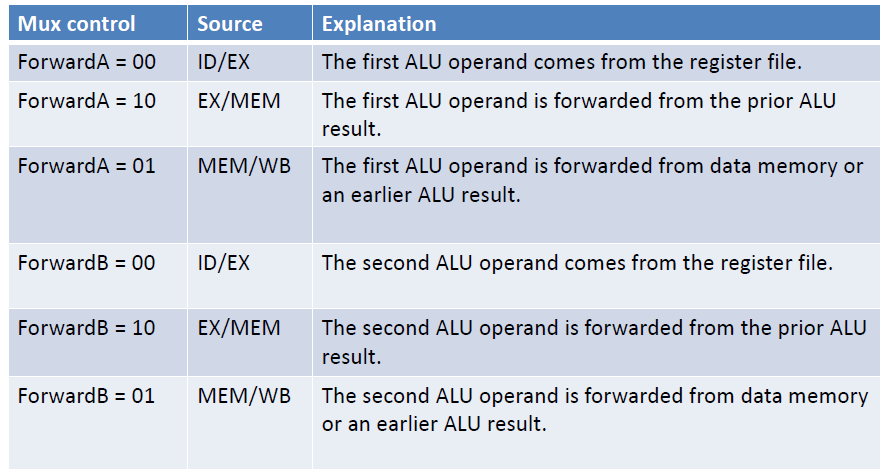
The hazards can be categorized into the following three types:

- Structural hazard: The hardware does not support the execution of multiple instructions in the same clock cycle.

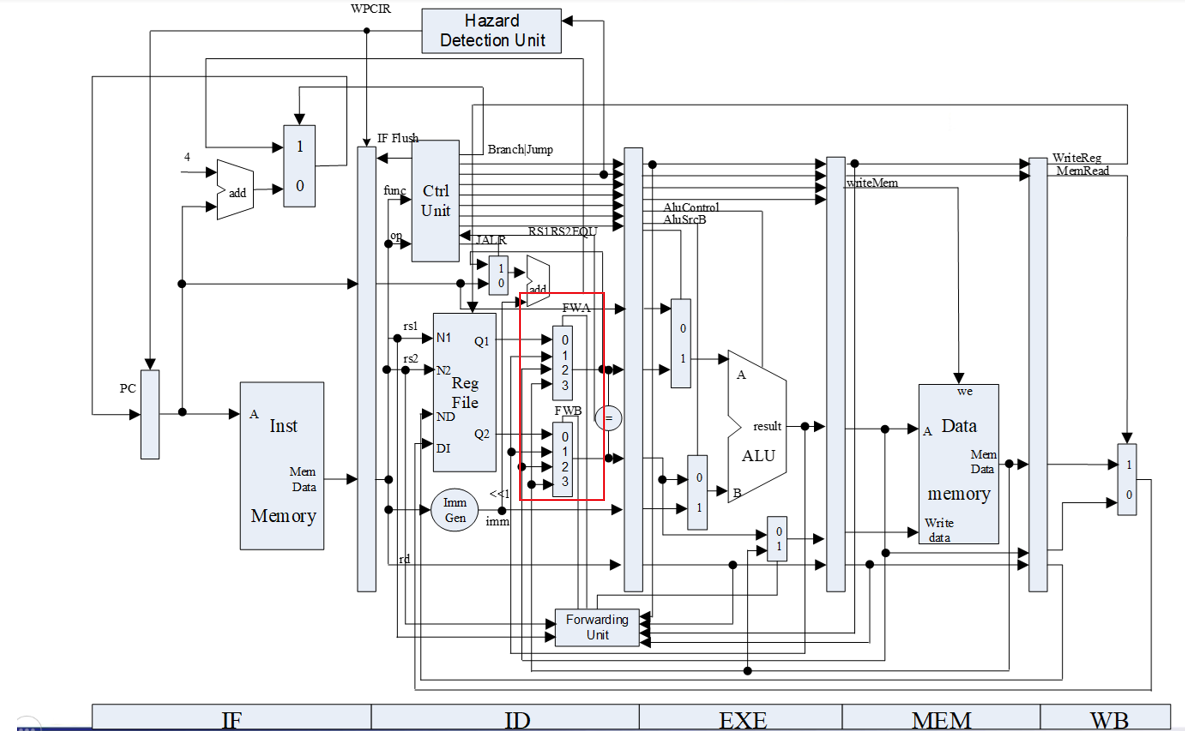
- Data hazard (Data hazard): the execution of the current instruction needs to wait for the data results of the previous instruction

- Control risk (Control hazard): the instruction is not sequential execution and lead to the next execution of the instruction is not the real expectation

**Forwarding Condition**



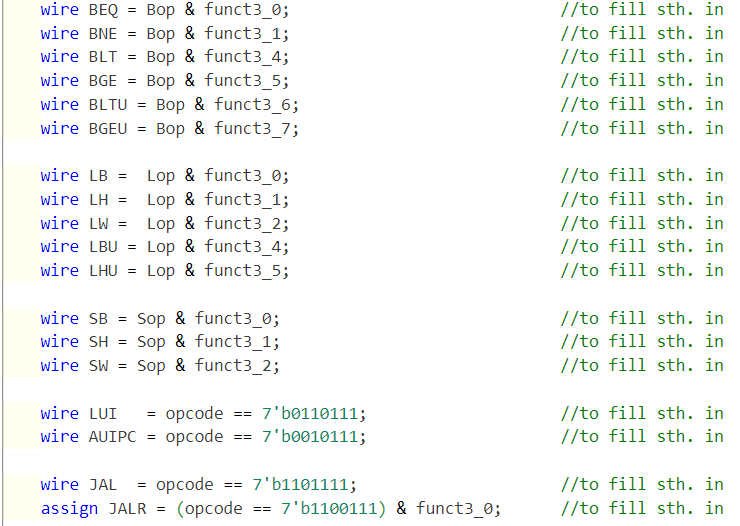
**Circuits Overview with forwarding**



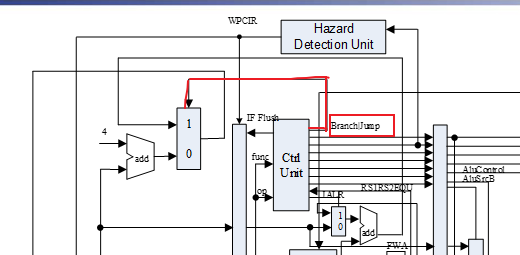
**3.2 Main Work**

1. Controlling Signal in CtrlUnit.v

To complete the classification process of RISC-V instructions, each wire functions as a boolean variable for instruction type determination. This process primarily leverages Opcode, fun3, and fun7 fields (with exceptions in some cases where Opcode alone suffices), and employs logical operations for this classification. The "Valid" wire, specifically, serves to distinguish among five instruction types: R-type, I-type, B-type, L-type, and S-type.



Type level signal to judge whether to jump

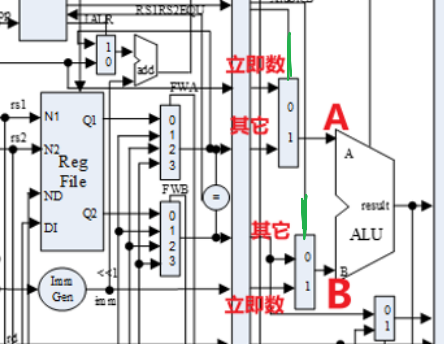
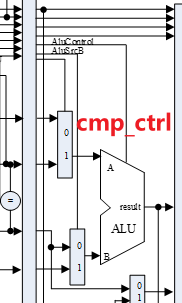


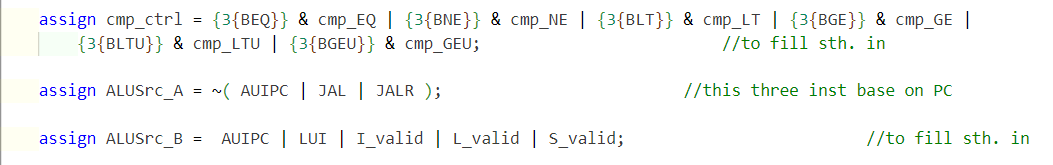


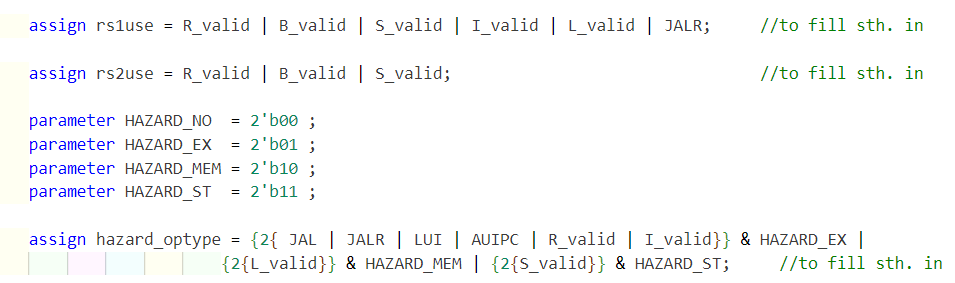
The inputs at both ends of the ALU are selected according to different signals.

Cmp\_ctrl is used to select which ALU operation

{3} represents a replication operation, which copies the elements within the brackets three times. In this particular context, it will copy the expressions {BEQ}, {BNE}, {BLT}, {BGE}, {BLTU}, and {BGEU} three times each



Whether interfaces 1 and 2 of Reg are used, and the signal at which stage Hazard occurs

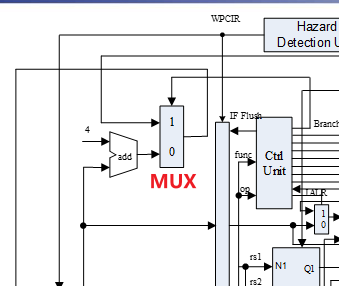
1. Controlling Signal in cmp\_32.v

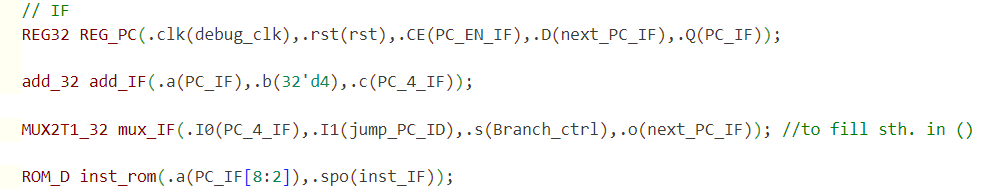
Outputs the result of comparison of B-type instructions in the ALU



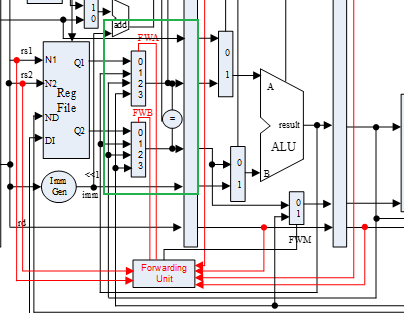
1. Filling in the interface in RV32core.v

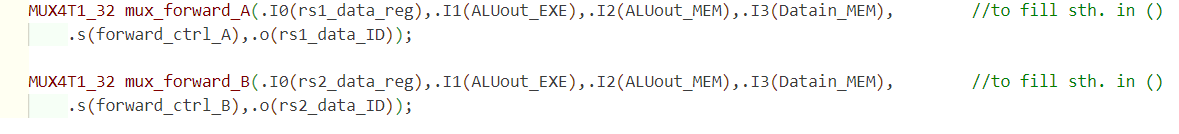
MUX2T1\_32 mux\_IF determines the next instruction



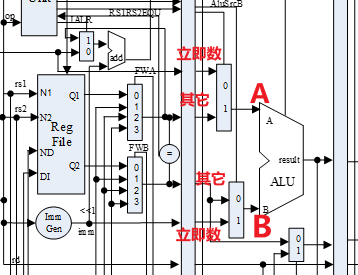


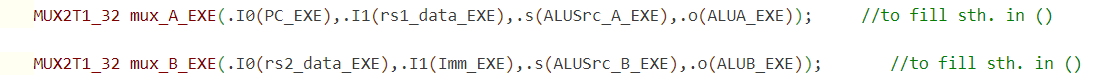
MUX4T1\_32 mux\_forward\_A and MUX4T1\_32 mux\_forward\_B determine whether forward is needed, and where the forward signal comes from



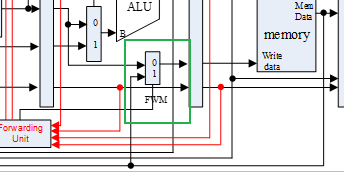


MUX4T1\_32 mux\_forward\_A and MUX4T1\_32 mux\_forward\_B select the inputs of the A and B interfaces of the ALU according to the control signal.





MUX2T1\_32 mux\_forward\_EXE is used to control whether the results of the memory reading phase need to be forwarded to the execution phase.





1. HazardDetectionUnit.v handles Hazard

The forwarding unit and branch prediction rollback are both placed in the hazard detection unit.

input[1:0] hazard\_optype\_ID, is the stage 1: ALU, 2: LOAD, 3: STORE

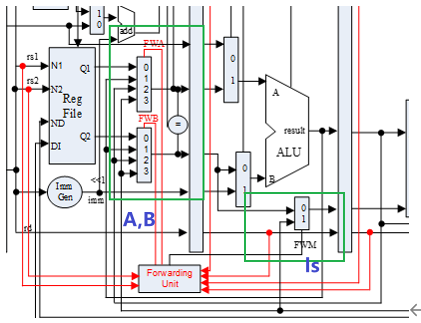
In this lab, there are three primary hazards that require stalling for rs1:

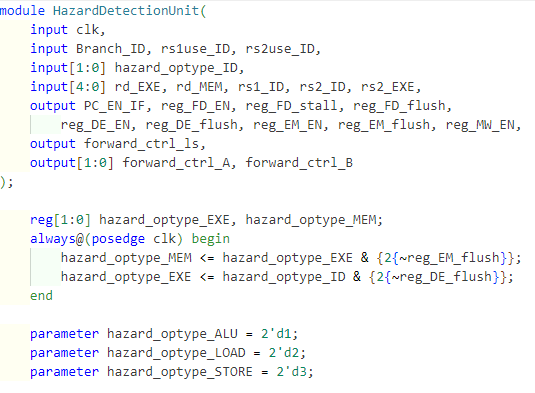
(1) When rs1 matches the destination register (rd) of an instruction currently executing in the EXE (Execute) stage.

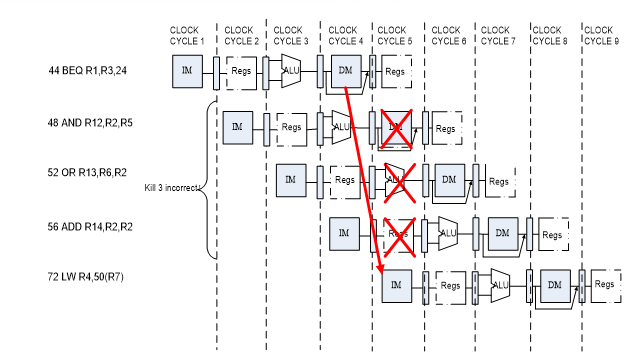
(2) When rs1 matches the destination register (rd) of an instruction currently executing in the MEM (Memory) stage.

(3) When rs1 matches the destination register (rd) of a LOAD instruction currently in the MEM stage.

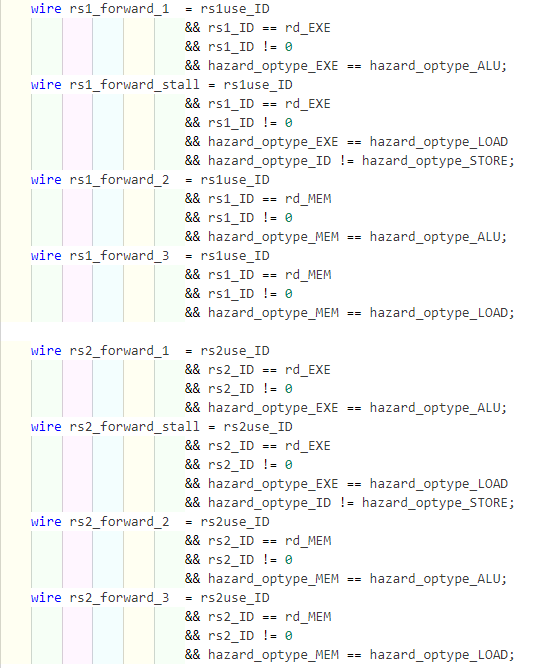
Similar stall conditions apply to rs2.

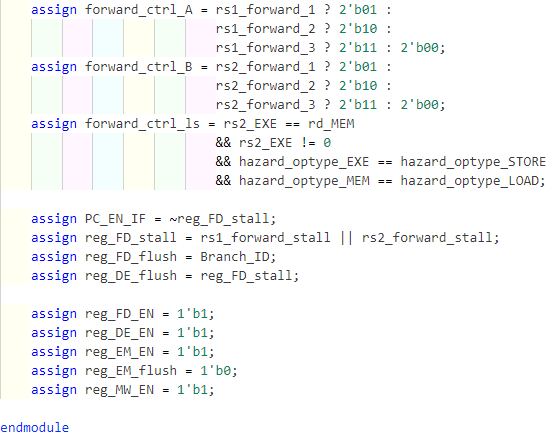






The predict not taken is used, i.e. we assume that it does not jump each time and directly executes the next instruction, which is PC+4. If the result of the final branch instruction execution is that it needs to jump, we flush out the erroneous instruction that entered the pipeline before, as shown in the figure



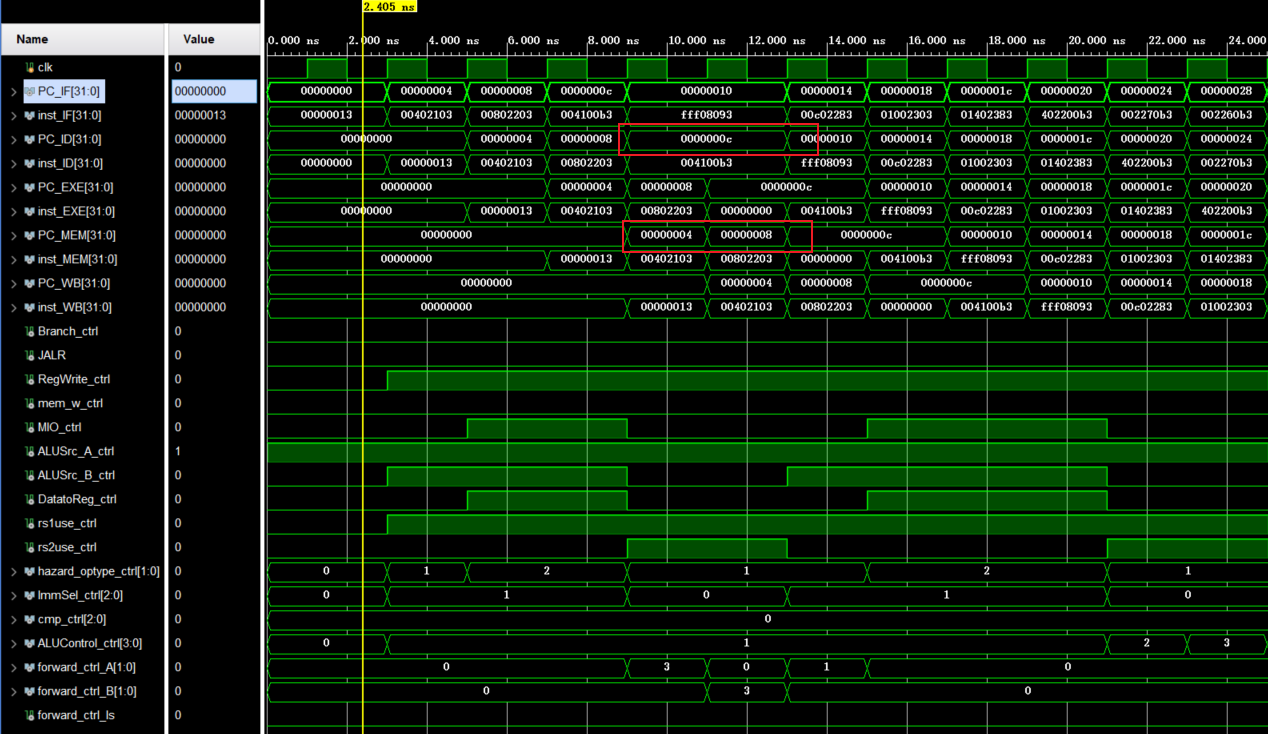


1. **Result and Analysis**

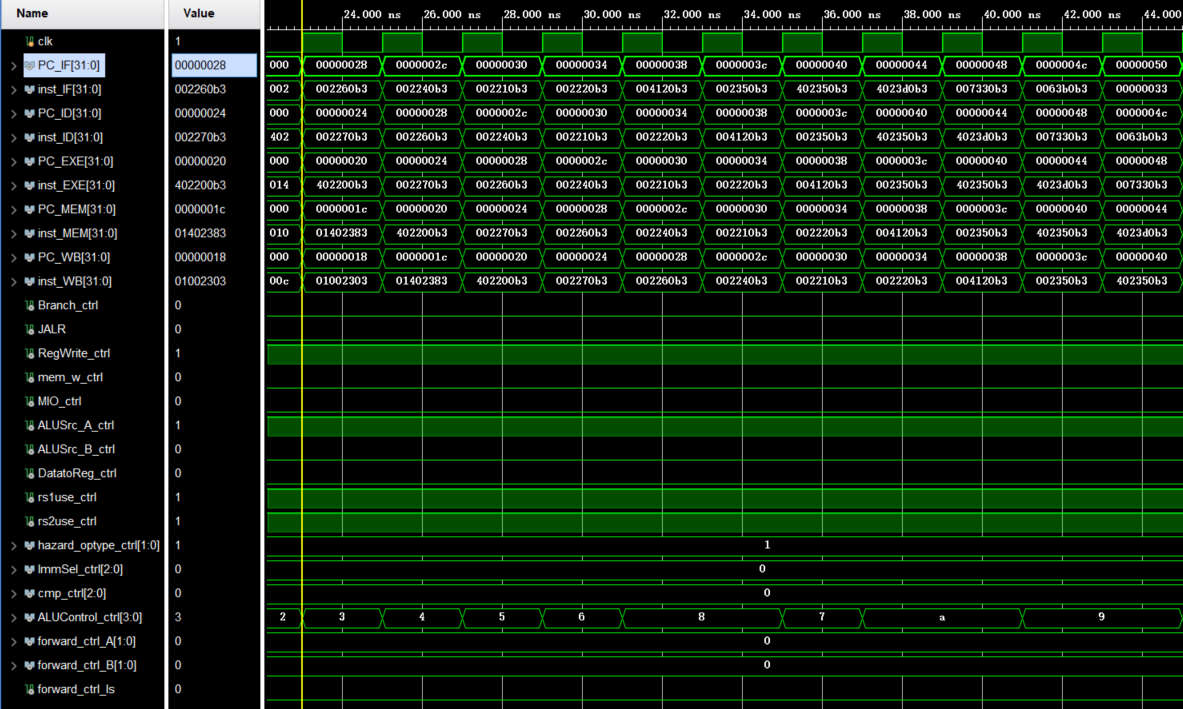
**4.1 Simulation results**

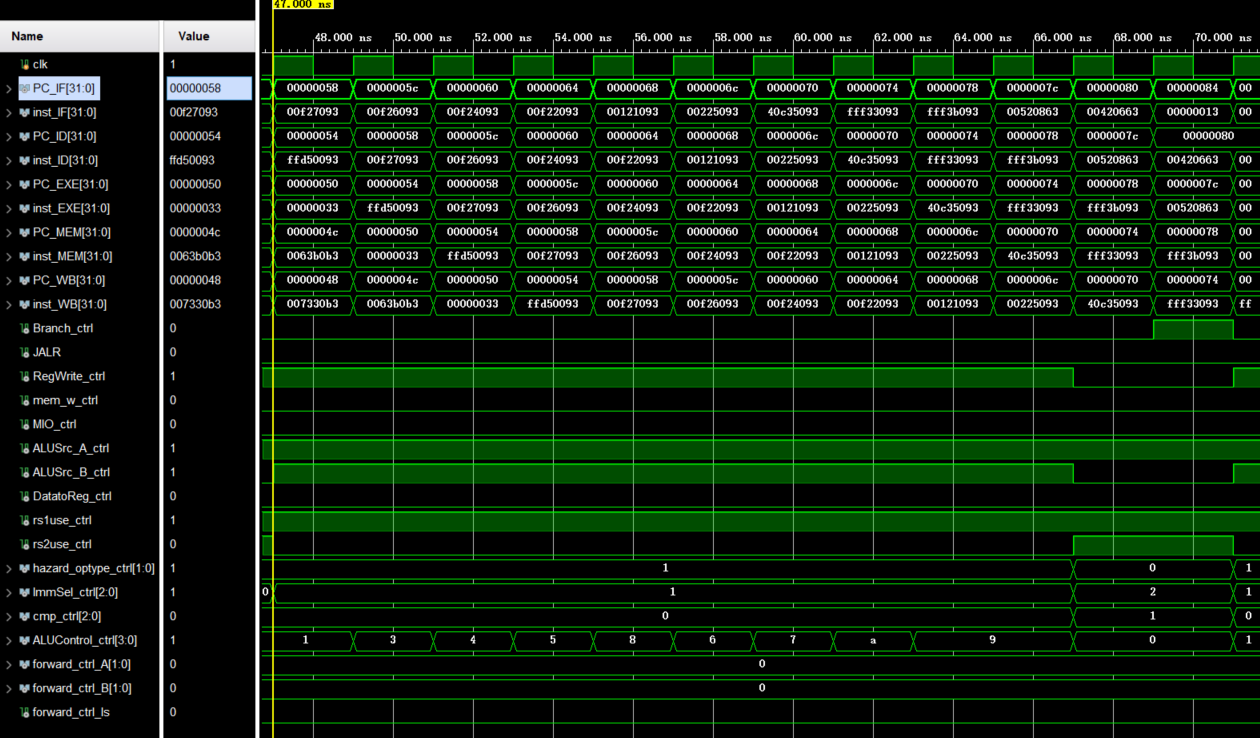


Using the three highlighted branch instructions as an example, it is evident that a data hazard has arisen here. This is because the 'add' instruction relies on data from registers x2 and x4, which depend on the results of the preceding two 'load' instructions fetching data from memory. Examining the simulation diagram, we observe that initially, during the IF (Instruction Fetch) stage, these three instructions enter the pipeline sequentially and each stays for one cycle. Subsequently, they move to the ID (Instruction Decode) stage. Here, we notice that the 'add' instruction incurs an additional cycle in the ID stage, as it must wait for the memory read/write stages of the preceding two 'load' instructions to complete before obtaining the required values. At this point, a pipeline stall occurs.

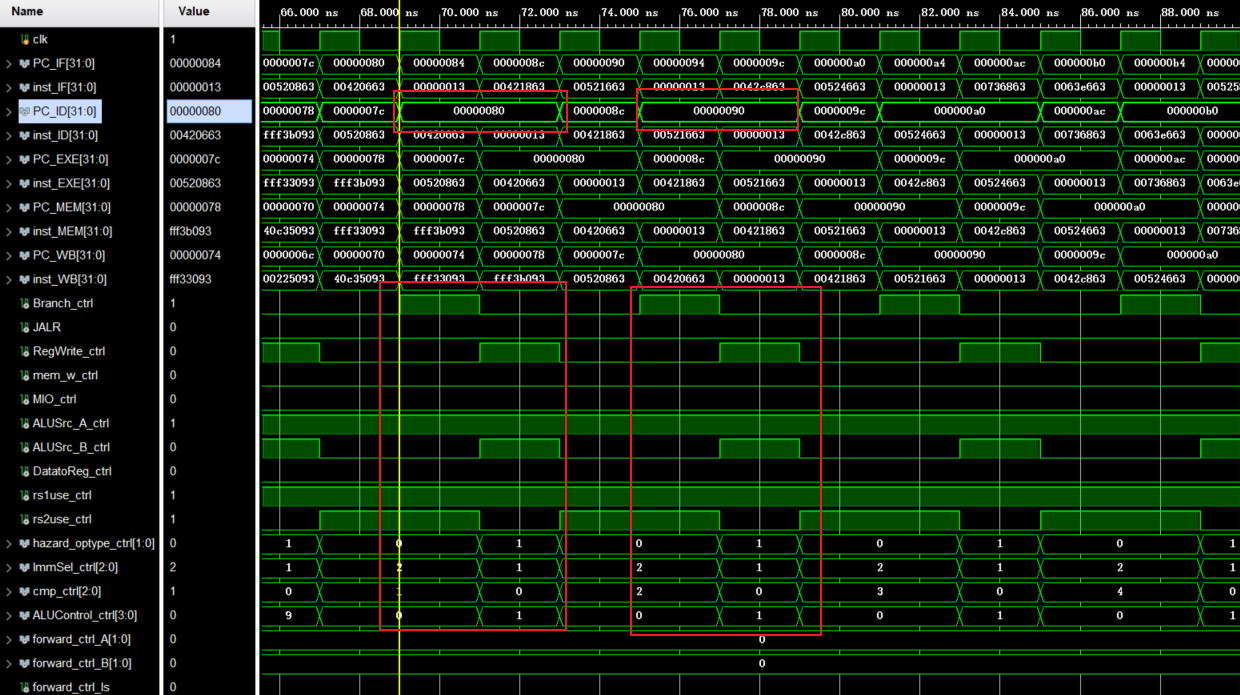


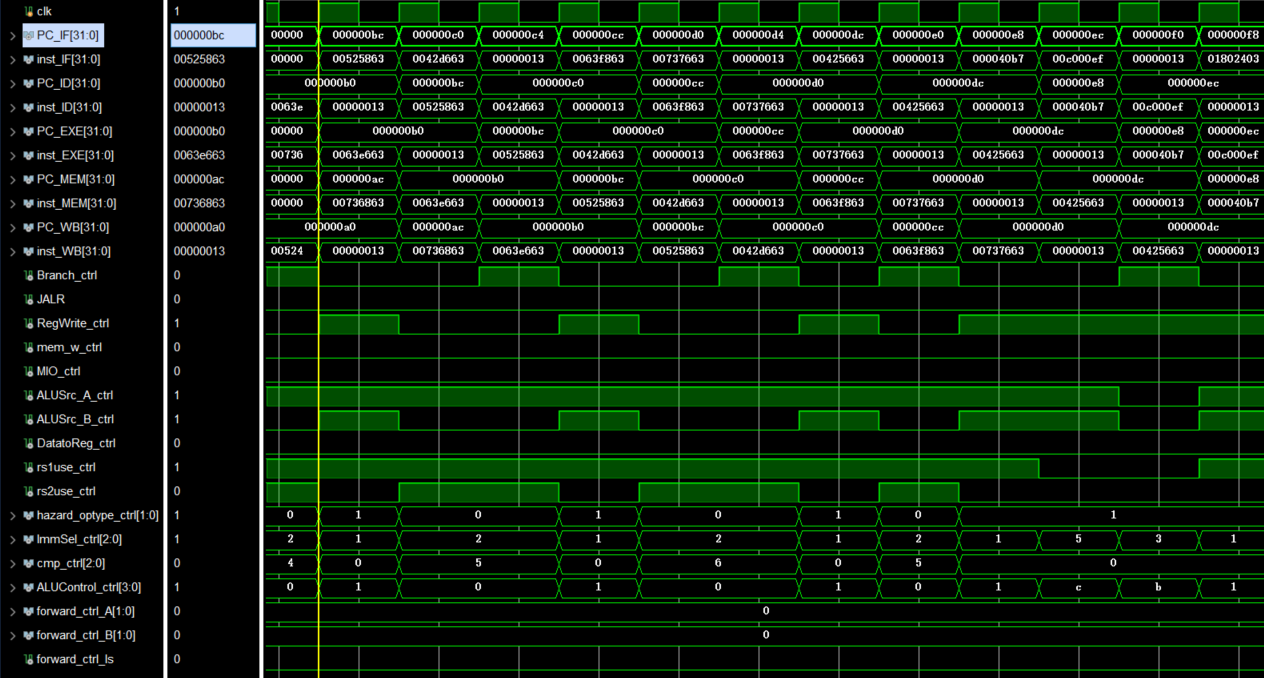
It can be found that the subsequent simulation results can also completely correspond to the results in the ppt.

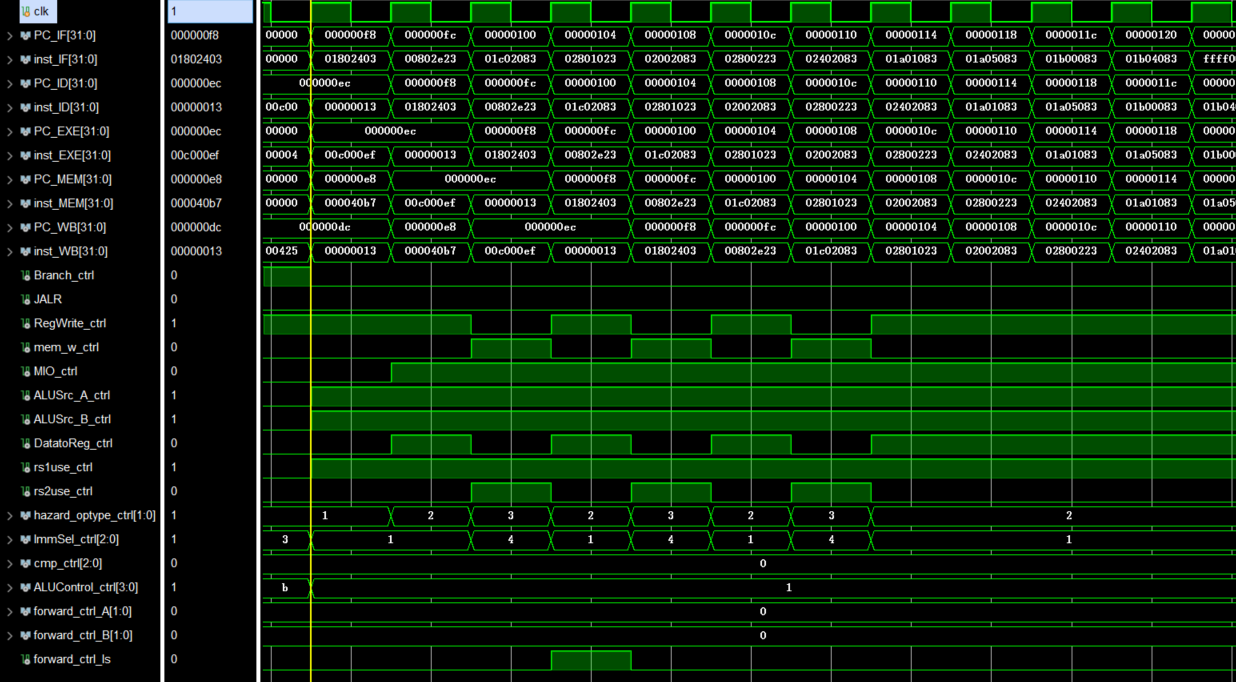


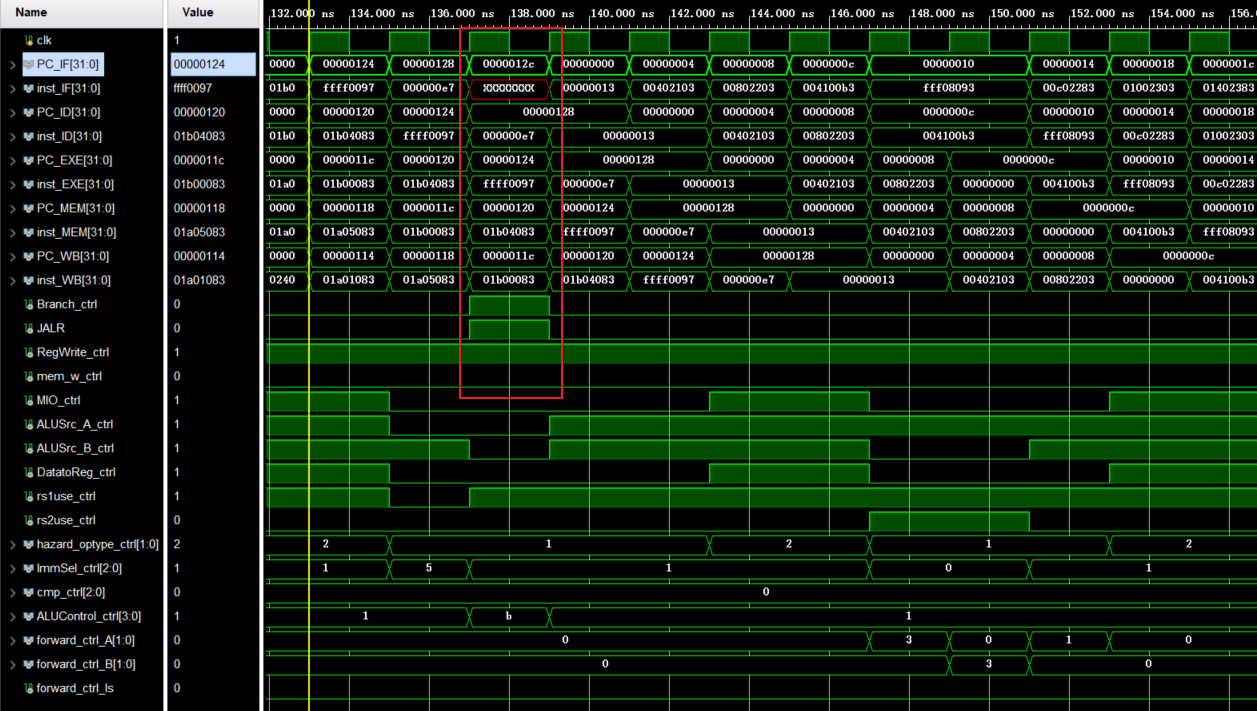


At address 0x80, there is a 'beq' instruction. We have a branch prediction mechanism in place, which employs a predict-not-taken strategy. Nevertheless, the branch instruction is taken in reality. As a result, we discard the instructions that should have been executed sequentially and flush the associated data.









**4.2 Physical verification**

Physical verification is the same as the simulation results, only the initial 7 steps are shown, and you can observe that the changes in each instruction are the same as in the simulation.

You can see that the ID is **0xC** in both 4 and 5, stalled a beat, same as the simulation result.

|  |  |
| --- | --- |
|  |  |
| 1 | 2 |
|  |  |
| 3 | 4 |
|  |  |
| 5 | 6 |
|  |  |
| 7 |  |

1. **Discussion and thoughts**

In this experiment, I learned to design a five-stage pipelined CPU for RISC-V RV32I instructions, focusing on hazard detection, forwarding, and branch prediction. This included understanding different instruction types and handling hazards efficiently, using forwarding to overcome data hazards, and implementing a predict-not-taken strategy for branch instructions. The simulation and physical verification phases were crucial for validating our design. Overall, this experiment provided valuable insights into CPU design complexities and the importance of efficient hazard handling and branch prediction for CPU performance.