浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：

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：

指导教师：卜凯

2022 年　　10 月　　12 日

浙江大学实验报告

课程名称：计算机体系结构　　实验类型：综合

实验项目名称：Lab1：Pipelined CPU supporting RISC-V RVI32 Instructions

学生姓名：　　专业：计算机科学与技术　　学号：

同组学生姓名：　　指导老师：卜凯

实验地点：曹西 301　　实验日期：2022 年 10 月 12 日

1 Tasks and requirements

1.1 Tasks

We will have 5 main tasks in Lab-1, those are:

1. Understand RISC-V RV32I instructions.
2. Master the design methods of pipelined CPU executing RV32I instructions.
3. Master the method of **Pipeline Forwarding Detection** and **bypass unit** design.
4. Master the methods of 1-cycle stall of **Predict-not-taken branch** design.
5. Master methods of program verification of Pipelined CPU executing RV32I instructions.

1.2 Requirements

This experiment would be based on SWORD development board, with xc7k325tffg676-2L FPGA.

We are given a Verilog project which most parts have been finished, but still need to modify some files:

1. **RV32core.v**, as CPU core module.
2. **CtrlUnit.v**, the control unit of the core.
3. **HazardDetectionUnit.v**, in which we need to realize forwarding and branch

prediction functions.

4. **Code2Inst.v**, because still some instructions in the preset program are not included in. But the modification is **not necessary**.

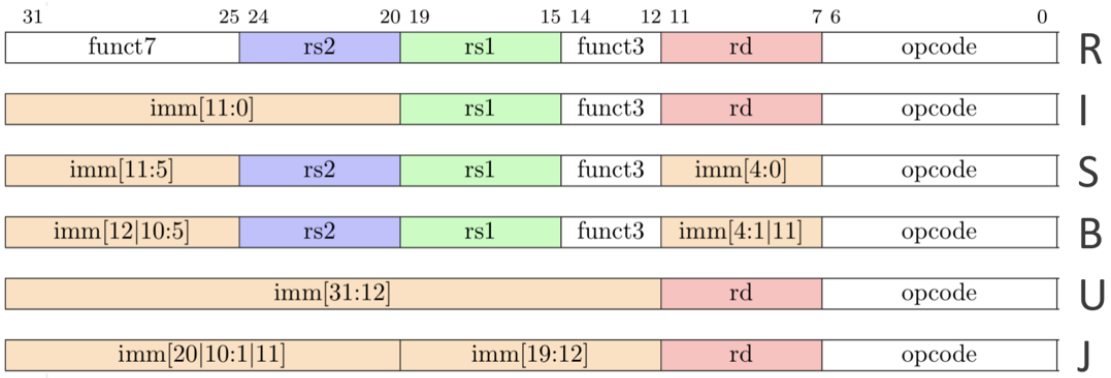
When this work is finished, we shall verify both the simulation results and the on-board results.

2 Contents and principles

All of the principles are based on RISC-V 5-stage pipeline CPU.

2.1 RISC-V instructions

RISC-V ISA has many types of instructions, but all of them have a fixed length of **32-bit**, which makes the hardware design easier. Here we are about to deal with the following types of instructions:



R-type instructions, mainly store register arithmetic results into register.

I-type instructions, store register and immediate arithmetic results into register, or load memory data into register.

S-type instructions, are to save register data to memory.

B-type instructions, deal with branch in program.

U-type instructions, deal with very large immediate.

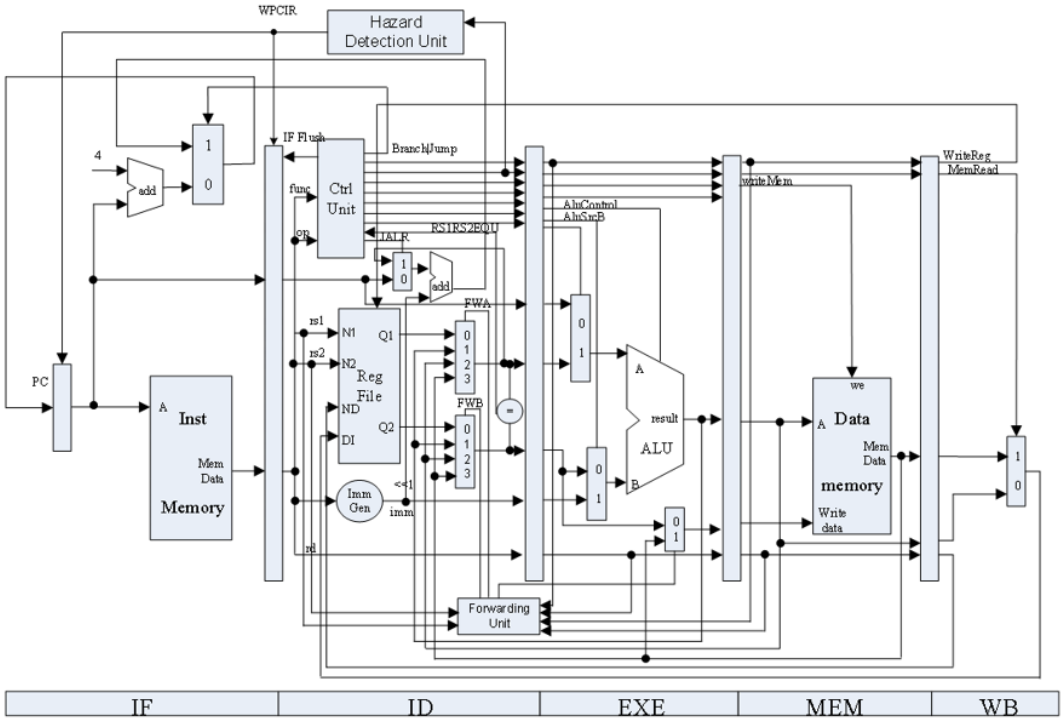
J-type instructions, mean jump to another address.

Each instruction would have different **opcode**, **funct3** or **funct7** bits, and we can read these bits to distinguish different instructions. This process is mainly dealt in **control unit** of CPU. After determining the control signals, the **data path** structure

would complete the work that the instruction informed.

2.2 Data path and control unit

Following is the data path we need to implement.



The data path is typical **5-stage**, that is, divided by some registers into 5 stages: **IF**(Instruction Fetch), **ID**(Instruction Decode), **EXE**(Execute), **MEM**(Memory), **WB**(Write Back).

IF stage. We fetch the instruction from instruction memory.

ID stage. We try to decode the instruction and give out control signals.

EXE stage. We do ALU operations.

MEM stage. We write data into memory, or get data from memory.

WB stage. We write data back into registers.

Pipelining could improve CPU efficiency clearly, however, it may also cause some problems. There are 3 types of **hazards**. **Structure hazard** does not occur in our data path structure. **Data hazard** can be solved by bubbling, or forwarding. And we will try to avoid **control hazard** by prediction and flush.

Control unit is used to identify each instruction and give out control signals. We can see lots of control signals are transported to every stage and intervene in that

stage’s operation. However, most of the signals are not important to this experiment. Now we mainly concern the signals that are used to implement **forwarding** and **prediction**.

The signals are introduced in more concerned parts following.

2.3 Forwarding

Forwarding technique is used to solve data hazards. Using **hazard optype** signal to distinguish instruction types, we could achieve 3 types of **forwarding**:

1. An instruction using a specific register at ID stage after an ALU instruction which writes into the register at EXE stage.
2. An instruction using a specific register at ID stage after an ALU instruction which writes into the register at MEM stage.
3. An instruction using a specific register at ID stage after a LOAD instruction which writes into the register at MEM stage.

The forwarding unit will produce a forward control signal which control the **FWA and FWB MUXs** of ID stage, and then the required register data which is maintained in EXE or MEM stage could be forwarded to avoid data hazard.

However, there is one situation that forwarding cannot be used. We must **bubble**(stall):

4. An ALU or LOAD instruction using a specific register at ID stage after a LOAD instruction which writes into the register at EXE stage.

That’s because the required data could be get only at MEM stage.

If this happened, we would let the ALU or LOAD instruction stop at ID stage, and let a bubble instruction go into EXE stage. Then the situation is converted to condition 3, which is mentioned above.

2.4 Predict not-taken

Predictions is used to solve control hazards. We here will use **predict not-taken** method, which means we will predict all branch instructions as not branching -- the next instruction after a branch instruction will be fetched without any consideration.

Added a compare unit at ID stage, we could distinguish the branch instruction will

really branch or not. If not, then we naturally execute the next instruction. However, if branch, we have to invalidate the current instruction at ID stage.

To do this, we use **flush**. We could make all control signals that would affect the registers or memory invalid, then the instruction which was wrong would not make any effect on our CPU running conditions. It follows the correct instructions.

3 Steps and data records

3.1 Completed Verilog source files

3.1.1 RV32core.v

The filled blanks are as following.

These codes are mainly about MUXs used in data path.

```
//IF
MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));

//ID
MUX4T1_32 mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),.s(forward_ctrl_A),.o(rs1_data_ID));
MUX4T1_32 mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),.s(forward_ctrl_B),.o(rs2_data_ID));

//EXE
MUX2T1_32 mux_A_EXE(.I0(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));
MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));
MUX2T1_32 mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout_EXE));
```

3.1.2 CtrlUnit.v

The filled blanks are as following.

```
//instruction types decode(B, L, S, U, J types)
wire BEQ = Bop & funct3_0;
wire BNE = Bop & funct3_1;
wire BLT = Bop & funct3_4;
wire BGE = Bop & funct3_5;
wire BLTU = Bop & funct3_6;
wire BGEU = Bop & funct3_7;

wire LB =  Lop & funct3_0;
wire LH =  Lop & funct3_1;
wire LW =  Lop & funct3_2;
wire LBU = Lop & funct3_4;
wire LHU = Lop & funct3_5;

wire SB = Sop & funct3_0;
wire SH = Sop & funct3_1;
wire SW = Sop & funct3_2;

wire LUI   = opcode == 7'b0110111;
wire AUIPC = opcode == 7'b0010111;

wire JAL  = opcode == 7'b1101111;
assign JALR = opcode == 7'b1100111;
```

```
//control signals
assign Branch = (JAL | JALR | B_valid) & cmp_res;
assign cmp_ctrl = {3{B_valid}} & funct3;
assign ALUSrc_A = JAL | JALR | AUIPC;
assign ALUSrc_B = I_valid | L_valid | S_valid | AUIPC | LUI;
assign rs1use = R_valid | I_valid | B_valid | L_valid | S_valid | JALR;
assign rs2use = R_valid | B_valid | S_valid;
assign hazard_optype = {2{R_valid | I_valid | JAL | JALR | AUIPC | LUI}} & 2'b01 |
                       {2{L_valid}} & 2'b10 |
                       {2{S_valid}} & 2'b11 ;
```

3.1.3 HazardDetectionUnit.v

This module is as following.

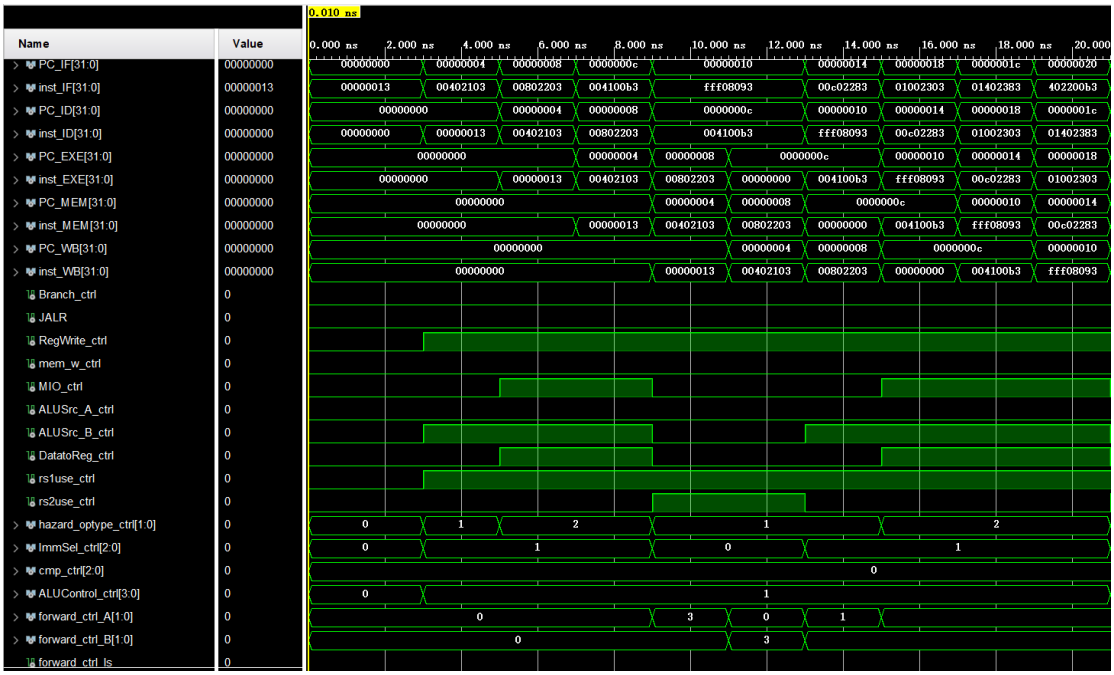
```
module HazardDetectionUnit(
    input clk,
    input Branch_ID, rs1use_ID, rs2use_ID,
    input[1:0] hazard_optype_ID,
    input[4:0] rd_EXE, rd_MEM, rs1_ID, rs2_ID, rs2_EXE,
    output PC_EN_IF, reg_FD_EN, reg_FD_stall, reg_FD_flush,
        reg_DE_EN, reg_DE_flush, reg_EM_EN, reg_EM_flush, reg_MW_EN,
    output forward_ctrl_ls,
    output[1:0] forward_ctrl_A, forward_ctrl_B
);
          //according to the diagram, design the Hazard Detection Unit
    reg[1:0] hazard_optype_EXE, hazard_optype_MEM;
    always@(posedge clk) begin
        hazard_optype_MEM <= hazard_optype_EXE & {2{~reg_EM_flush}};
        hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
    end

    parameter hazard_optype_ALU = 2'd1;
    parameter hazard_optype_LOAD = 2'd2;
    parameter hazard_optype_STORE = 2'd3;

    wire rs1_forward_1     = rs1use_ID && rd_EXE != 5'b00000 && rd_EXE == rs1_ID && hazard_optype_EXE == hazard_optype_ALU;
    wire rs1_forward_stall = rs1use_ID && rd_EXE != 5'b00000 && rd_EXE == rs1_ID && hazard_optype_EXE == hazard_optype_LOAD && hazard_optype_ID != hazard_optype_STORE;
    wire rs1_forward_2     = rs1use_ID && rd_MEM != 5'b00000 && rd_MEM == rs1_ID && hazard_optype_MEM == hazard_optype_ALU;
    wire rs1_forward_3     = rs1use_ID && rd_MEM != 5'b00000 && rd_MEM == rs1_ID && hazard_optype_MEM == hazard_optype_LOAD;
```

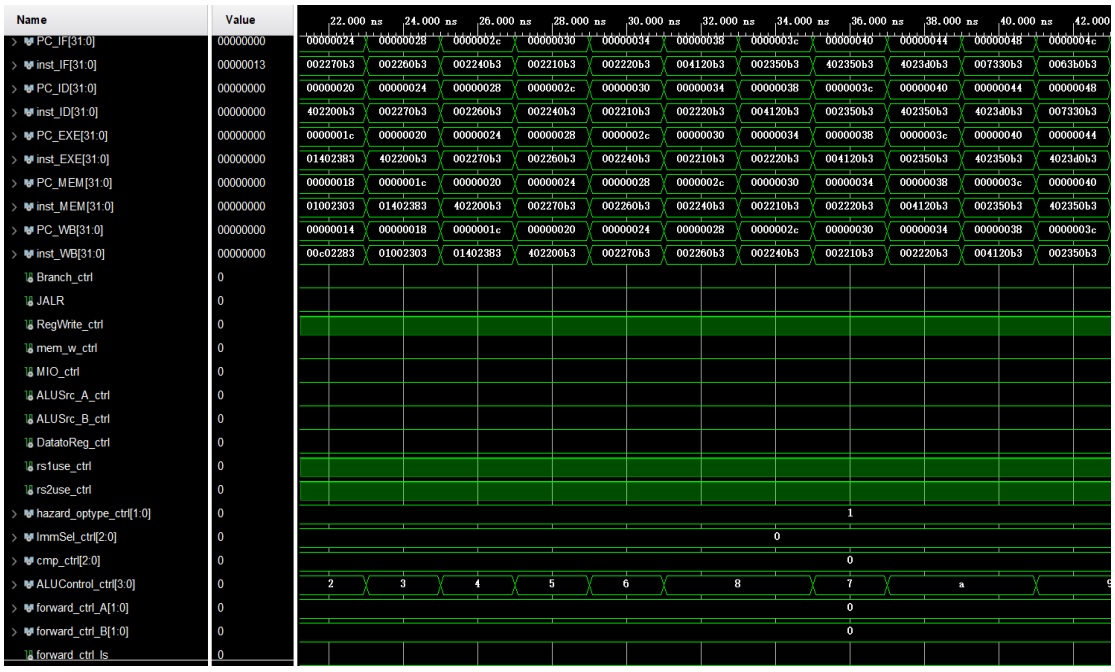
```

    assign forward_ctrl_A = {2{rs1_forward_1}} & 2'b01 |
                            {2{rs1_forward_2}} & 2'b10 |
                            {2{rs1_forward_3}} & 2'b11 ;

    wire rs2_forward_1     = rs2use_ID && rd_EXE != 5'b00000  && rd_EXE == rs2_ID && hazard_optype_EXE == hazard_optype_ALU;
    wire rs2_forward_stall = rs2use_ID && rd_EXE != 5'b00000  && rd_EXE == rs2_ID && hazard_optype_EXE == hazard_optype_LOAD && hazard_optype_ID != hazard_optype_STORE;
    wire rs2_forward_2     = rs2use_ID && rd_MEM != 5'b00000  && rd_MEM == rs2_ID && hazard_optype_MEM == hazard_optype_ALU;
    wire rs2_forward_3     = rs2use_ID && rd_MEM != 5'b00000  && rd_MEM == rs2_ID && hazard_optype_MEM == hazard_optype_LOAD;

    assign forward_ctrl_B = {2{rs2_forward_1}} & 2'b01 |
                            {2{rs2_forward_2}} & 2'b10 |
                            {2{rs2_forward_3}} & 2'b11 ;

    assign forward_ctrl_ls = rd_MEM == rs2_EXE && rd_MEM != 5'b00000 && hazard_optype_EXE == hazard_optype_STORE && hazard_optype_MEM == hazard_optype_LOAD;

    assign PC_EN_IF = ~(rs1_forward_stall | rs2_forward_stall);
    assign reg_FD_stall = rs1_forward_stall | rs2_forward_stall;
    assign reg_FD_flush = Branch_ID;
    assign reg_DE_flush = rs1_forward_stall | rs2_forward_stall;
    assign reg_EM_flush = 1'b0;
    assign reg_FD_EN = 1'b1;
    assign reg_DE_EN = 1'b1;
    assign reg_EM_EN = 1'b1;
    assign reg_MW_EN = 1'b1;

endmodule
```

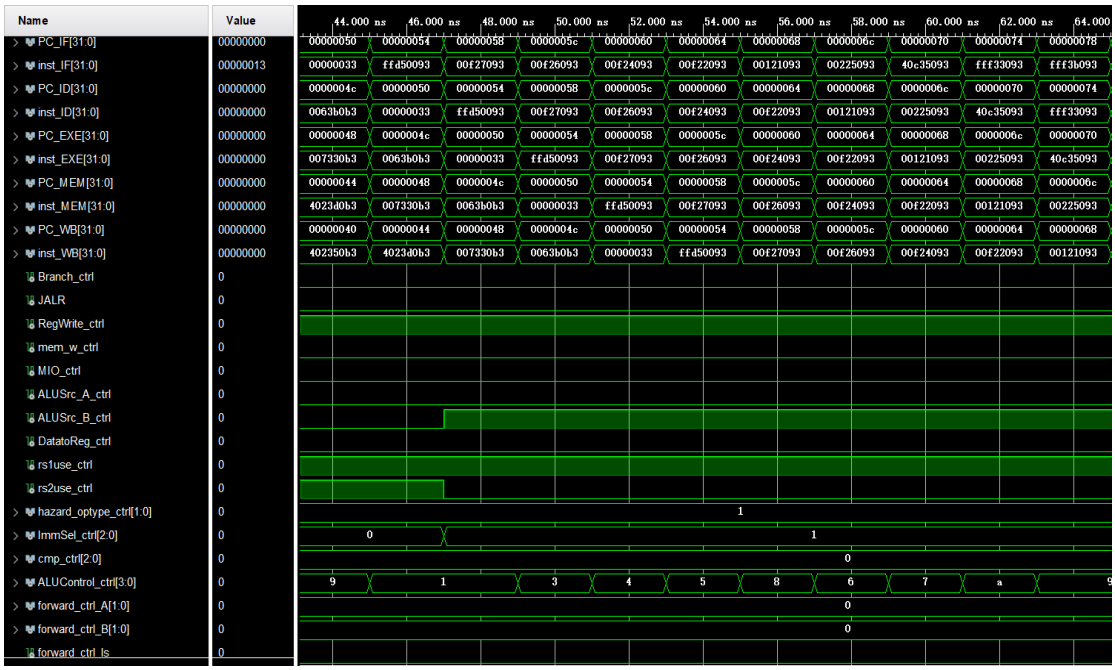
You shall see the 2.3 and 2.4 chapters to get some explanation.

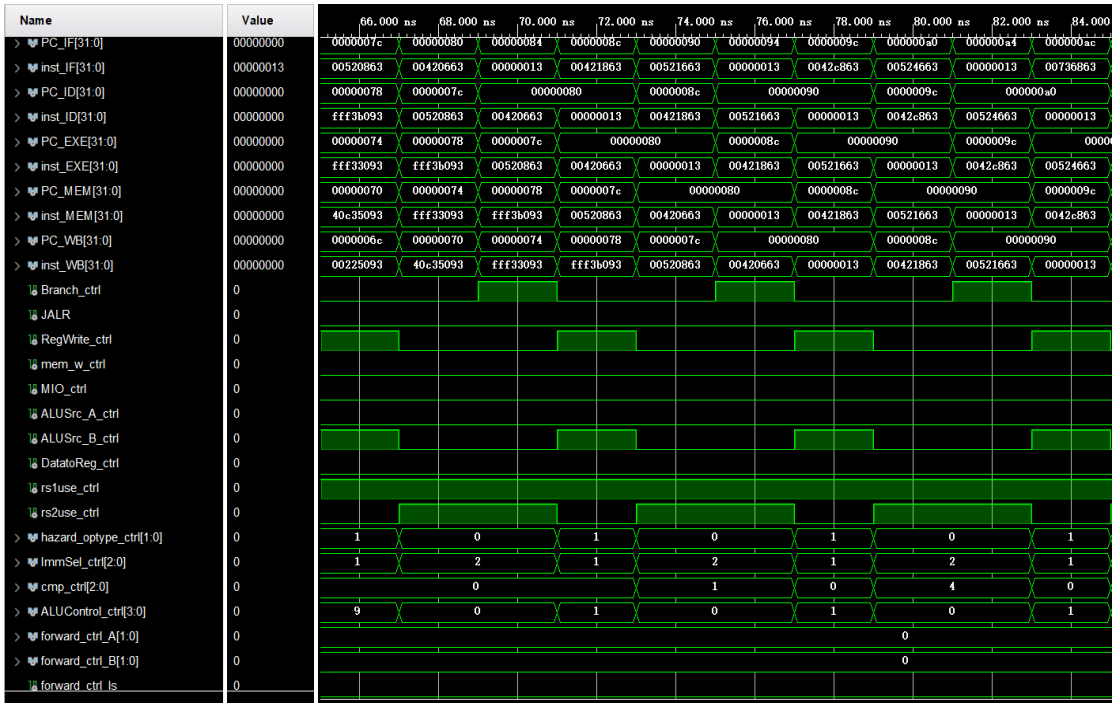
3.2 Implementation results

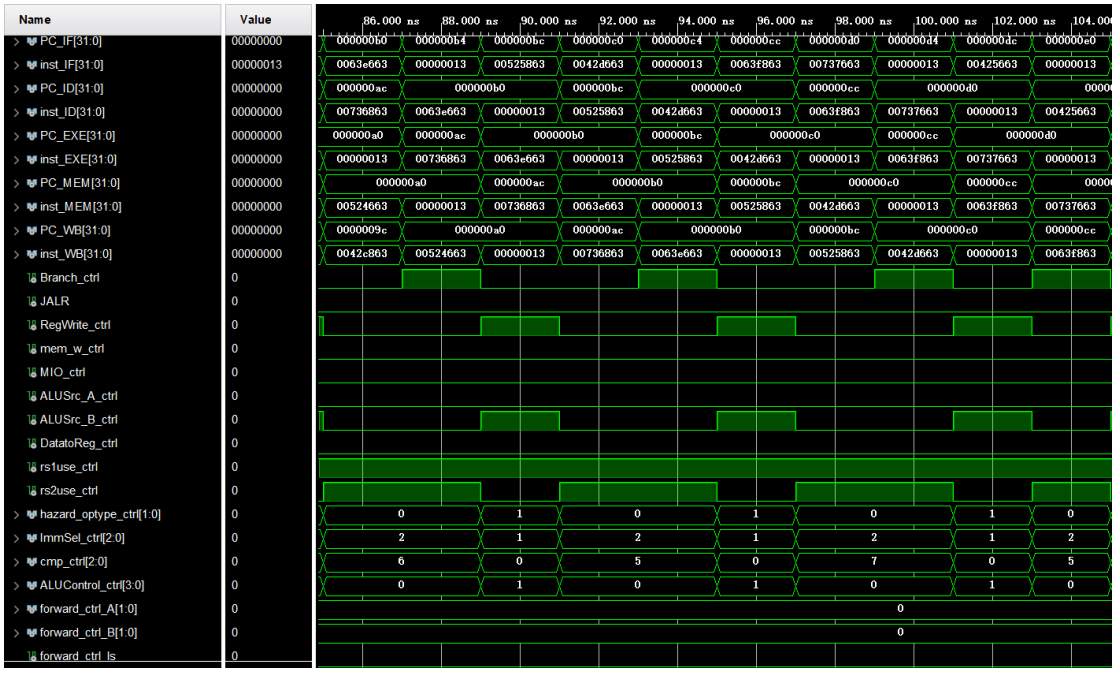
Here we record all our behavioral simulation results. Most will not be used for analysis.

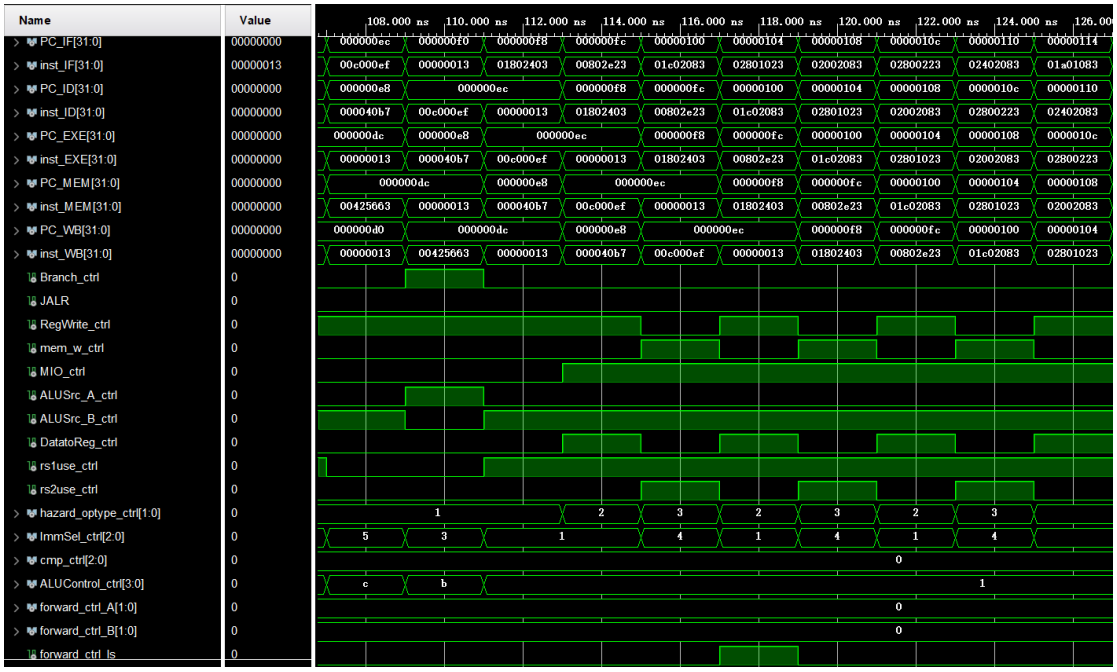


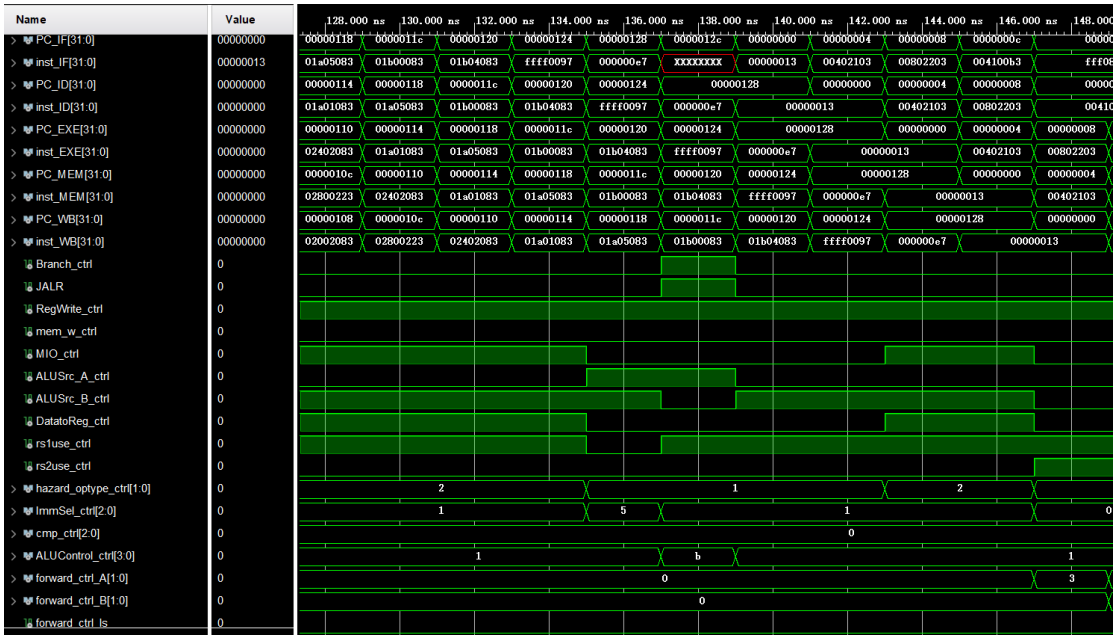



Name
Value
PC_IF[31:0] 00000000
inst_IF[31:0] 00000013
PC_ID[31:0] 00000000
inst_ID[31:0] 00000000
PC_EXE[31:0] 00000000
inst_EXE[31:0] 00000000
PC_MEM[31:0] 00000000
inst_MEM[31:0] 00000000
PC_WB[31:0] 00000000
inst_WB[31:0] 00000000
Branch_ctrl 0
JALR 0
RegWrite_ctrl 0
mem_w_ctrl 0
MIO_ctrl 0
ALUSrc_A_ctrl 0
ALUSrc_B_ctrl 0
DatatoReg_ctrl 0
rs1use_ctrl 0
rs2use_ctrl 0
hazard_optype_ctrl[1:0] 0
ImmSel_ctrl[2:0] 0
cmp_ctrl[2:0] 0
ALUControl_ctrl[3:0] 0
forward_ctrl_A[1:0] 0
forward_ctrl_B[1:0] 0
forward_ctrl_ls 0
44.000 ns
46.000 ns
48.000 ns
50.000 ns
52.000 ns
54.000 ns
56.000 ns
58.000 ns
60.000 ns
62.000 ns
64.000
00000050 00000054 00000058 0000005c 00000060 00000064 00000068 0000006c 00000070 00000074 00000078
00000033 ffd50093 00f27093 00f26093 00f24093 00f22093 00121093 00225093 40c35093 fff33093 fff3b093
0000004c 00000050 00000054 00000058 0000005c 00000060 00000064 00000068 0000006c 00000070 00000074
0063b0b3 00000033 ffd50093 00f27093 00f26093 00f24093 00f22093 00121093 00225093 40c35093 fff33093
00000048 0000004c 00000050 00000054 00000058 0000005c 00000060 00000064 00000068 0000006c 00000070
007330b3 0063b0b3 00000033 ffd50093 00f27093 00f26093 00f24093 00f22093 00121093 00225093 40c35093
00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060 00000064 00000068 0000006c
4023d0b3 007330b3 0063b0b3 00000033 ffd50093 00f27093 00f26093 00f24093 00f22093 00121093 00225093
00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060 00000064 00000068
402350b3 4023d0b3 007330b3 0063b0b3 00000033 ffd50093 00f27093 00f26093 00f24093 00f22093 00121093
1
0
1
0
9 1 3 4 5 8 6 7 a 9
0
0


Name
Value
PC_IF[31:0] 00000000
inst_IF[31:0] 00000013
PC_ID[31:0] 00000000
inst_ID[31:0] 00000000
PC_EXE[31:0] 00000000
inst_EXE[31:0] 00000000
PC_MEM[31:0] 00000000
inst_MEM[31:0] 00000000
PC_WB[31:0] 00000000
inst_WB[31:0] 00000000
Branch_ctrl 0
JALR 0
RegWrite_ctrl 0
mem_w_ctrl 0
MIO_ctrl 0
ALUSrc_A_ctrl 0
ALUSrc_B_ctrl 0
DatatoReg_ctrl 0
rs1use_ctrl 0
rs2use_ctrl 0
hazard_optype_ctrl[1:0] 0
ImmSel_ctrl[2:0] 0
cmp_ctrl[2:0] 0
ALUControl_ctrl[3:0] 0
forward_ctrl_A[1:0] 0
forward_ctrl_B[1:0] 0
forward_ctrl_ls 0
66.000 ns
68.000 ns
70.000 ns
72.000 ns
74.000 ns
76.000 ns
78.000 ns
80.000 ns
82.000 ns
84.000
0000007c 00000080 00000084 0000008c 00000090 00000094 0000009c 000000a0 000000a4 000000ac
00520863 00420663 00000013 00421863 00521663 00000013 0042c863 00524663 00000013 00736863
00000078 0000007c 00000080 0000008c 00000090 0000009c 000000a0
fff3b093 00520863 00420663 00000013 00421863 00521663 00000013 0042c863 00524663 00000013
00000074 00000078 0000007c 00000080 0000008c 00000090 0000009c
fff33093 fff3b093 00520863 00420663 00000013 00421863 00521663 00000013 0042c863 00524663
00000070 00000074 00000078 0000007c 00000080 0000008c 00000090 0000009c
40c35093 fff33093 fff3b093 00520863 00420663 00000013 00421863 00521663 00000013 0042c863
0000006c 00000070 00000074 00000078 0000007c 00000080 0000008c 00000090
00225093 40c35093 fff33093 fff3b093 00520863 00420663 00000013 00421863 00521663 00000013
1 0 1 0 1 0 1
1 2 1 2 1 2 1
0 1 0 4 0
9 0 1 0 1 0 1
0
0


Name
Value
86.000 ns
88.000 ns
90.000 ns
92.000 ns
94.000 ns
96.000 ns
98.000 ns
100.000 ns
102.000 ns
PC_IF[31:0]
inst_IF[31:0]
PC_ID[31:0]
inst_ID[31:0]
PC_EXE[31:0]
inst_EXE[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
Branch_ctrl
JALR
RegWrite_ctrl
mem_w_ctrl
MIO_ctrl
ALUSrc_A_ctrl
ALUSrc_B_ctrl
DatatoReg_ctrl
rs1use_ctrl
rs2use_ctrl
hazard_optype_ctrl[1:0]
ImmSel_ctrl[2:0]
cmp_ctrl[2:0]
ALUControl_ctrl[3:0]
forward_ctrl_A[1:0]
forward_ctrl_B[1:0]
forward_ctrl_ls

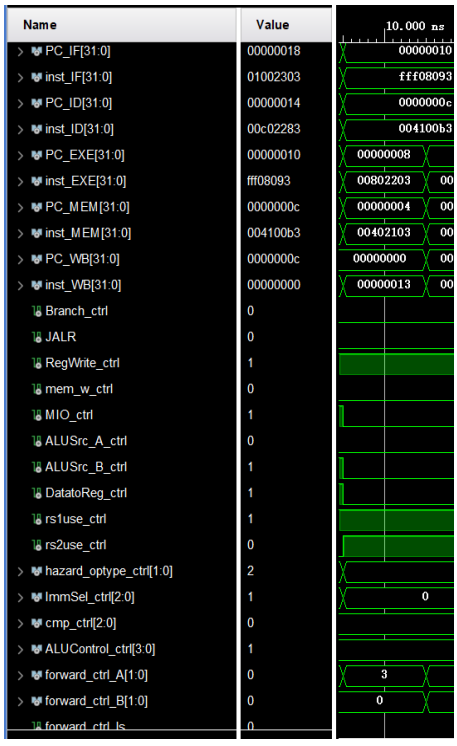

Name
Value
108.000 ns
110.000 ns
112.000 ns
114.000 ns
116.000 ns
118.000 ns
120.000 ns
122.000 ns
124.000 ns
PC_IF[31:0]
inst_IF[31:0]
PC_ID[31:0]
inst_ID[31:0]
PC_EXE[31:0]
inst_EXE[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
Branch_ctrl
JALR
RegWrite_ctrl
mem_w_ctrl
MIO_ctrl
ALUSrc_A_ctrl
ALUSrc_B_ctrl
DatatoReg_ctrl
rs1use_ctrl
rs2use_ctrl
hazard_optype_ctrl[1:0]
ImmSel_ctrl[2:0]
cmp_ctrl[2:0]
ALUControl_ctrl[3:0]
forward_ctrl_A[1:0]
forward_ctrl_B[1:0]
forward_ctrl_ls



4 Analysis of the results

Here we only consider some typical results.

4.1 Successful forwarding



We will see the simulation in 9-11ns. During this period, the instructions in pipeline are:

IF 0xfff08093, **addi x1, x1, -1**

ID 0x004100b3, **add x1, x2, x4**

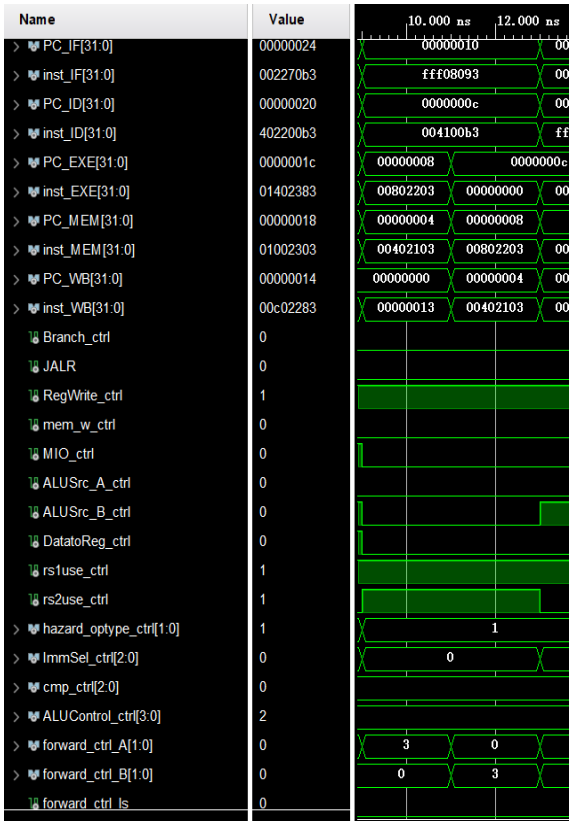
EXE 0x00802203, **lw x4, 8(x0)**

MEM 0x00402103, **lw x2, 4(x0)**

WB 0x00000013, **addi x0, x0, 0**

Here we can see **forward_control_A** is 3. That is, the forwarding condition 3: LOAD x2 at MEM and using x2 at ID. That's a successful detection and forwarding.

4.2 Stall to solve data hazard



We see the simulation in 9-13ns. During 9-11ns, the instructions are as 4.1. During 11-13ns, they are:

IF 0xfff08093, **addi x1, x1, -1**

ID 0x004100b3, **add x1, x2, x4**

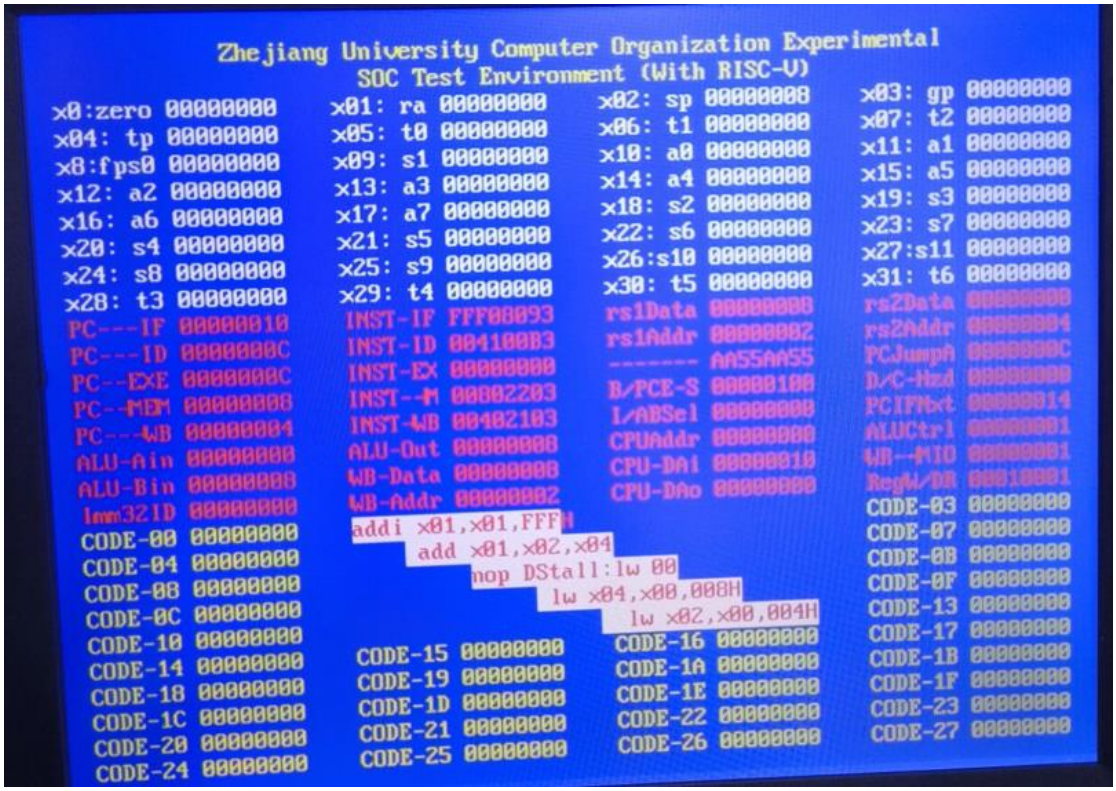
EXE 0x00000000, **bubble**

MEM 0x00802203, **lw x4, 8(x0)**

WB 0x00402103, **lw x2, 4(x0)**

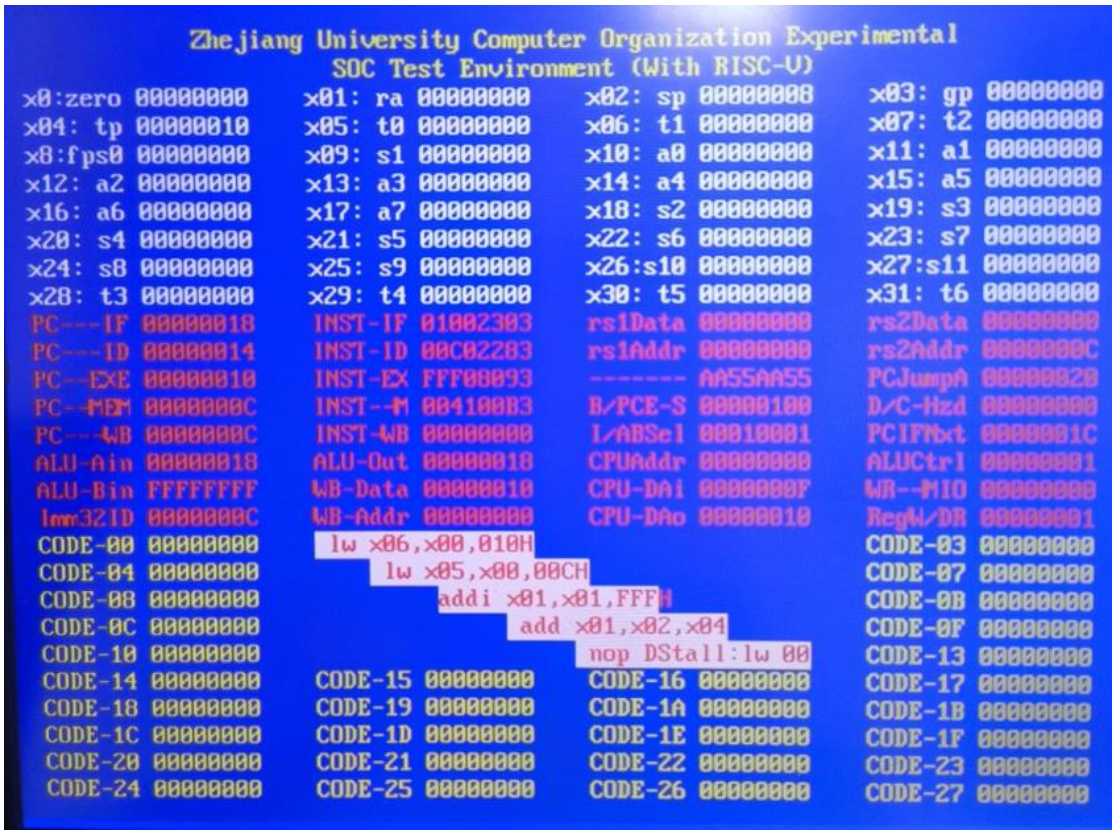
That's because, in 9-11ns, there is a data hazard concerning x4: LOAD at EXE and used at ID. We cannot forward, so we **stall** ID, and send a bubble instruction into EXE.

This result also is saved by us on FPGA:



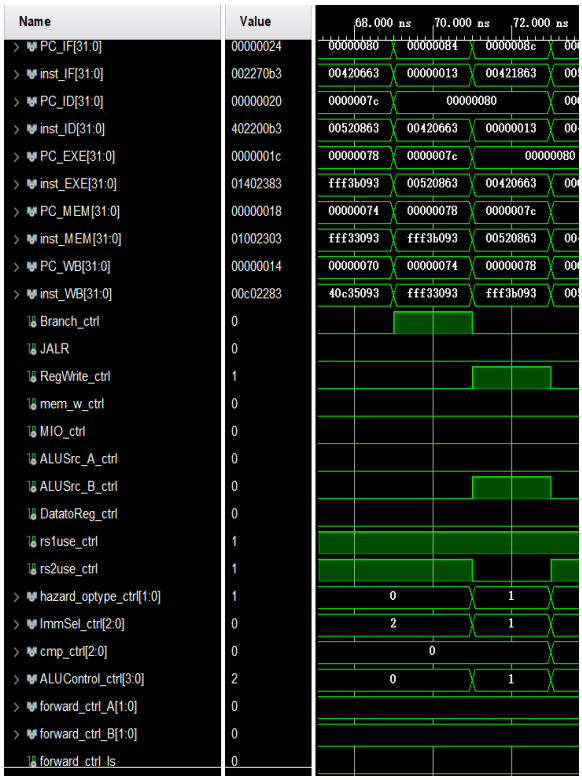
That's the explained condition.

The next time snap, we can see we only stall the ID instruction for one step.



It runs normally after that stall.

4.3successful branch not-taken prediction



We see the simulation in 67-73ns. During 67-69ns, the instructions in pipeline are:

IF 0x00420663, **beq x4, x4, 12**

ID 0x00520863, **bne x4, x4, 16**

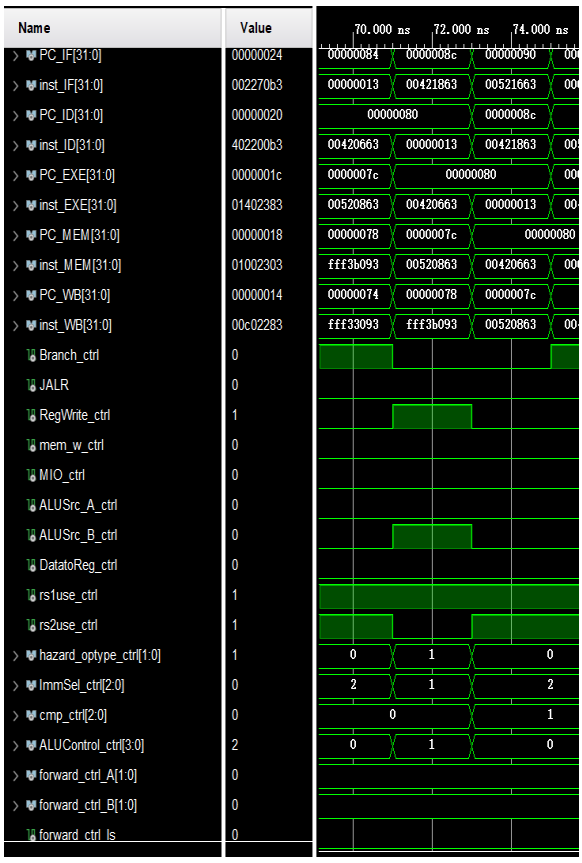
EXE 0xfff3b093, **sltiu x1, x7, -1**

MEM 0xfff33093, **sltiu x1, x6, -1**

WB 0x40c35093, **srai x1, x6, 12**

Now we get a branch instruction at ID, and we get the compare result: not taken. Then we normally execute the IF instruction, which is predicted to be fetched, in 71-73ns.

4.4Unsuccessful branch not-taken prediction



We see the simulation in 69-75ns. During 69-71ns, the instructions in pipeline are:

IF 0x0000013, **addi x0, x0, 0**

ID 0x00420663, **beq x4, x4, 12**

EX 0x00520863, **bne x4, x4, 16**

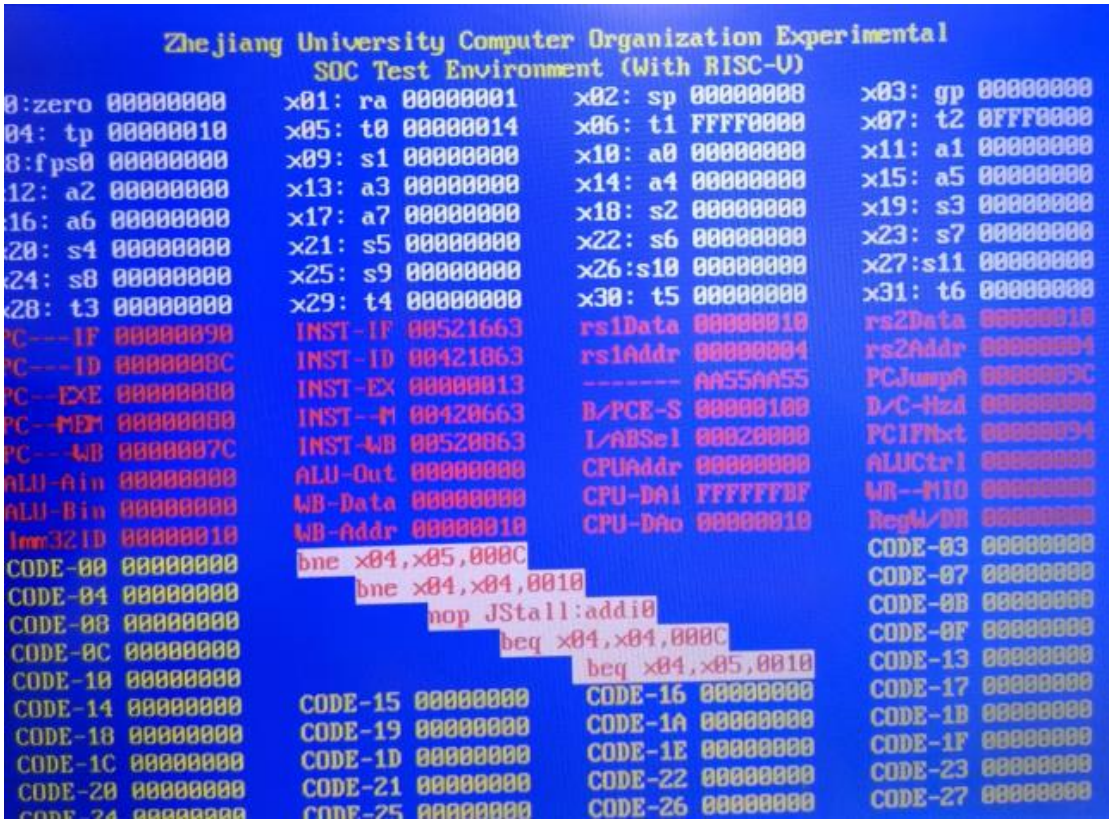
MEM 0xfff3b093, **sltiu x1, x7, -1**

WB 0xfff33093, **sltiu x1, x6, -1**

We get the compare result in ID: taken. Then we should not execute the instruction currently at IF. When this instruction is in EXE in 73-75ns, we **flush** it, that is, set all control signals to invalid.

This result also is saved by us on FPGA:

We see the branch in ID is taken, but an invalid instruction has entered IF.



When it is in other stages, all control signals concerning it have been set invalid.

5 Discussion and Conclusion

5.1 Problems

5.1.1 Problems concerning Verilog language

Problem 1. Of ‘&’ operator

In control unit, I first wrote a line as follows:

assign cmp_ctrl = B_valid & funct3;

That’s a fault. I forgot that *B_valid* and *funct3* do not have the same bits, consequently using the meaning of ‘&&’ operator as ‘&’. In the simulation I found *cmp_ctrl* had only one bit available, and then corrected this problem:

assign cmp_ctrl = {3{B_valid}} & funct3;

5.1.2 Problems concerning the module

Problem 1. Forwarding of x0 register

At first, my forwarding unit didn’t specify x0 register from other registers. However, as x0 is fixed to 0, forwarding x0 may cause some problem:

addi x0, x0, 1

add x1, x0, x0

If using this program, then x1 would become 2 instead of a correct 0 as x0 = 1 is forwarded. After comparing my program with my teammate, I corrected it.

5.1.3 Problems concerning the experiment guides

Problem 1. Of *ALUSrc_A* signal

In the data path shown in guides, *ALUSrc_A* = 1 means that using register data as input. However, in corresponded simulation results, *ALUSrc_A* = 1 means that using PC as input. Here I chose the second way, to achieve more readability when comparing with typical simulation results.

5.2 Achievements and conclusion

In this experiment, I reviewed RISC-V 5-stage pipeline CPU structure, and implemented forwarding and prediction functions. I get more familiar with Verilog language and Xilinx Vivado software as well. The experiment is successful.