Lab 1: Pipelined CPU Supporting RISC-V RV32I Instructions

3220106039 Hanxuan Li(李瀚轩)

1. Steps of the Experiment

1.1 Implementing all instructions from RV32I

Based on the provided code framework, we filled in the necessary parts, which are mostly straightforward. The code is as follows:

- `CtrlUnit.v` :

```
wire BEQ = Bop & funct3_0;  //to fill sth. in
wire BNE = Bop & funct3_1;  //to fill sth. in
wire BLT = Bop & funct3_4;  //to fill sth. in
wire BGE = Bop & funct3_5;  //to fill sth. in
wire BLTU = Bop & funct3_6; //to fill sth. in
wire BGEU = Bop & funct3_7; //to fill sth. in

wire LB =  Lop & funct3_0;  //to fill sth. in
wire LH =  Lop & funct3_1;  //to fill sth. in
wire LW =  Lop & funct3_2;  //to fill sth. in
wire LBU = Lop & funct3_4;  //to fill sth. in
wire LHU = Lop & funct3_5;  //to fill sth. in

wire SB = Sop & funct3_0;   //to fill sth. in
wire SH = Sop & funct3_1;   //to fill sth. in
wire SW = Sop & funct3_2;   //to fill sth. in

wire LUI   = opcode == 7'b0110111;  //to fill sth. in
wire AUIPC = opcode == 7'b0010111;  //to fill sth. in

wire JAL  = opcode == 7'b1101111;  //to fill sth. in
assign JALR = opcode == 7'b1100111 && funct3_0; //to fill sth. in
assign Branch = JAL | JALR | (B_valid & cmp_res); //to fill sth. in
assign cmp_ctrl = BEQ ? 3'b001 :
                  BNE ? 3'b010 :
                  BLT ? 3'b011 :
                  BLTU ? 3'b100 :
                  BGE ? 3'b101 :
                  BGEU ? 3'b110 : 3'b000; //to fill sth. in

assign ALUSrc_A = JAL | JALR | AUIPC; //to fill sth. in
assign ALUSrc_B = I_valid | L_valid | S_valid | LUI | AUIPC; //to fill sth. in
assign rs1use = R_valid | I_valid | B_valid | L_valid | S_valid | JALR; //to fill sth. in
assign rs2use = R_valid | B_valid | S_valid; //to fill sth. in
assign hazard_optype = (R_valid | I_valid | JAL | JALR | LUI | AUIPC) ? 2'b01 :
```

```
                         L_valid ? 2'b10 :
                         S_valid ? 2'b11 : 2'b00; //to fill sth. in
```

Here, we primarily added the decoding signals for Branch, Load, Store instructions, as well as the LUI, AUIPC, JAL, and JALR instructions. The `cmp_ctrl` signal is sent to the comparator to indicate the kind of comparison required for branch instructions. If `ALUSrc_A` is 1, the ALU uses the PC as input on the A port instead of the operand passed from the ID stage. Similarly, if `ALUSrc_B` is 1, the ALU uses an immediate value as input on the B port instead of the value passed from the ID stage. The signals `hazard_optype` , `rs1use` , and `rs2use` are sent to the Forwarding Unit for hazard detection.

- `cmp_32.v` :

```

assign c = (EQ & res_EQ) | (NE & res_NE) | (LT & res_LT) | (LTU & res_LTU) |
(GE & res_GE) | (GEU & res_GEU); //to fill sth. in
```

This code determines the type of comparison based on the `cmp_ctrl` signal. The logic is straightforward.

1.2 Implementing pipeline forwarding

Based on the logic of the datapath, we designed the following code:

```
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_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
       hazard_optype_MEM <= hazard_optype_EXE;
    end

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

    wire load_stall = ((rs1use_ID && rs1_ID == rd_EXE) || (rs2use_ID && rs2_ID
== rd_EXE && hazard_optype_ID != hazard_optype_STORE)) && rd_EXE &&
hazard_optype_EXE == hazard_optype_LOAD;

    assign forward_ctrl_A = (rs1use_ID && rs1_ID == rd_EXE && rd_EXE &&
hazard_optype_EXE == hazard_optype_ALU) ? 2'd1 :
```

```
                              (rs1use_ID && rs1_ID == rd_MEM && rd_MEM &&
hazard_optype_MEM == hazard_optype_ALU) ? 2'd2 :
                              (rs1use_ID && rs1_ID == rd_MEM && rd_MEM &&
hazard_optype_MEM == hazard_optype_LOAD) ? 2'd3 : 2'd0;
    assign forward_ctrl_B = (rs2use_ID && rs2_ID == rd_EXE && rd_EXE &&
hazard_optype_EXE == hazard_optype_ALU) ? 2'd1 :
                              (rs2use_ID && rs2_ID == rd_MEM && rd_MEM &&
hazard_optype_MEM == hazard_optype_ALU) ? 2'd2 :
                              (rs2use_ID && rs2_ID == rd_MEM && rd_MEM &&
hazard_optype_MEM == hazard_optype_LOAD) ? 2'd3 : 2'd0;
    assign reg_FD_EN = 1'b1;
    assign reg_DE_EN = 1'b1;
    assign reg_EM_EN = 1'b1;
    assign reg_MW_EN = 1'b1;
    assign reg_EM_flush = 1'b0;
    assign PC_EN_IF = ~load_stall;
    assign reg_FD_stall = load_stall;
    assign reg_FD_flush = Branch_ID;
    assign reg_DE_flush = load_stall;
    assign forward_ctrl_ls = (rs2_EXE == rd_MEM) && (hazard_optype_STORE ==
hazard_optype_EXE) && (hazard_optype_MEM == hazard_optype_LOAD);
endmodule
```

We first added two pipeline registers to store the `hazard_optype` signals for the current clock cycle in the EXE and MEM stages. Next, we determined the `load_stall` signal, which halts the ID stage if the MEM stage instruction has not completed yet. This happens when the current instruction needs the result of the previous instruction, is not a STORE, and the previous instruction is a LOAD.

We then determine the forwarding signals `forwardA` and `forwardB` . If the current instruction requires the result from the previous instruction, the forwarding signal is 1. If it requires the result from two instructions ago, we set `forward` based on the type of hazard. If the ALU result needs forwarding, the signal is 2; if the memory result needs forwarding, the signal is 3. Otherwise, no hazard occurs, and the forwarding signal is 0. The `forward_ctrl_ls` signal is used for cases like `lw r1, xxx` followed by `sw r1, xxx` .

Finally, based on the `load_stall` and `Branch_ID` signals, we decide whether to stall the IF stage or flush the ID stage.

1.3 Pipeline Integration

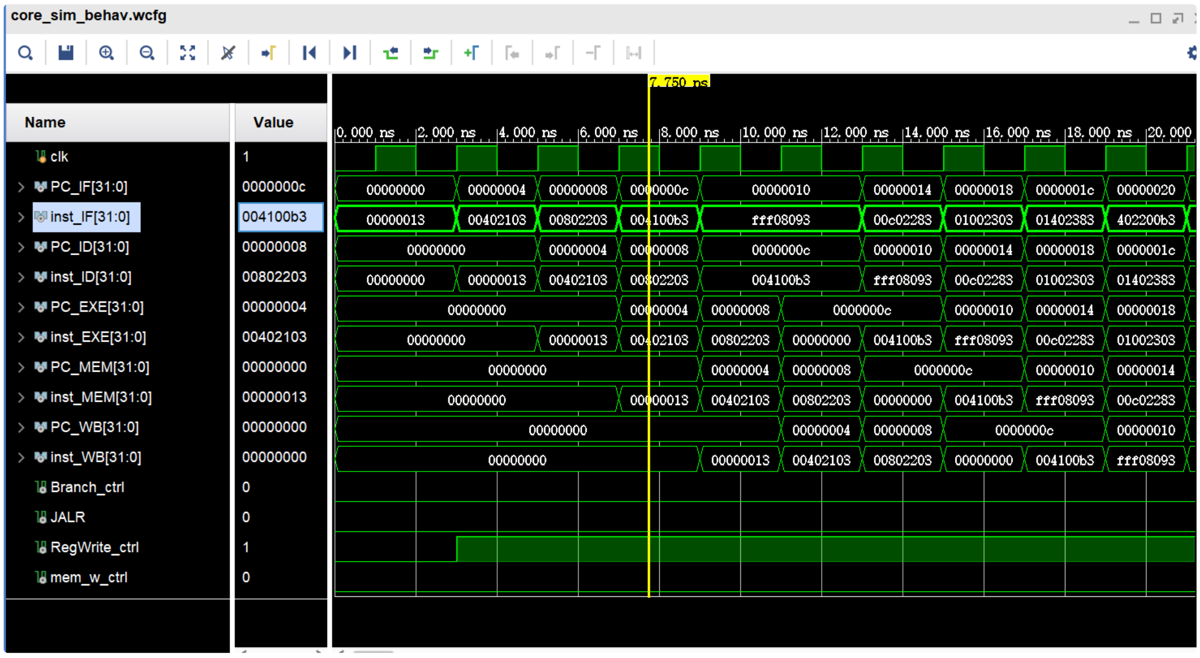
In `RV32core.v` , we connected the forwarding and control signals to the complete datapath. The logic is simple, and the code is as follows:

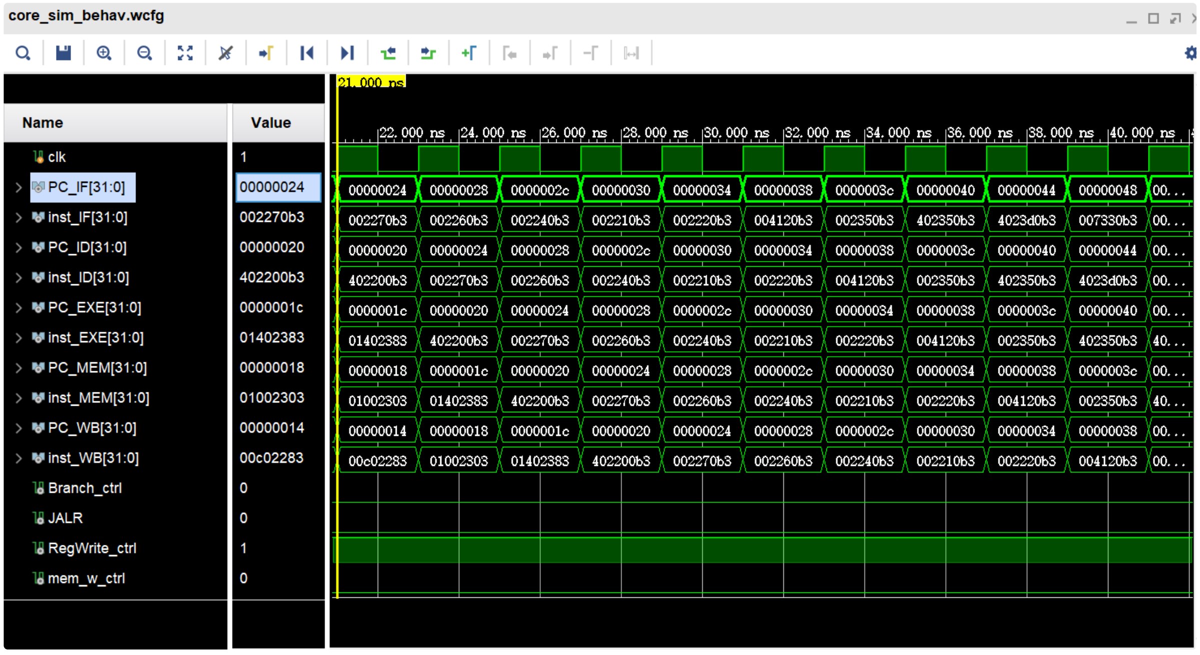
```
//IF
MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));
//to fill sth. in ()

//ID
MUX4T1_32 mux_forward_A(.I0(rs1_data_reg), .I1(ALUout_EXE), .I2(ALUout_MEM),
.I3(Datain_MEM), //to fill sth. in ()
        .s(forward_ctrl_A),.o(rs1_data_ID));
```

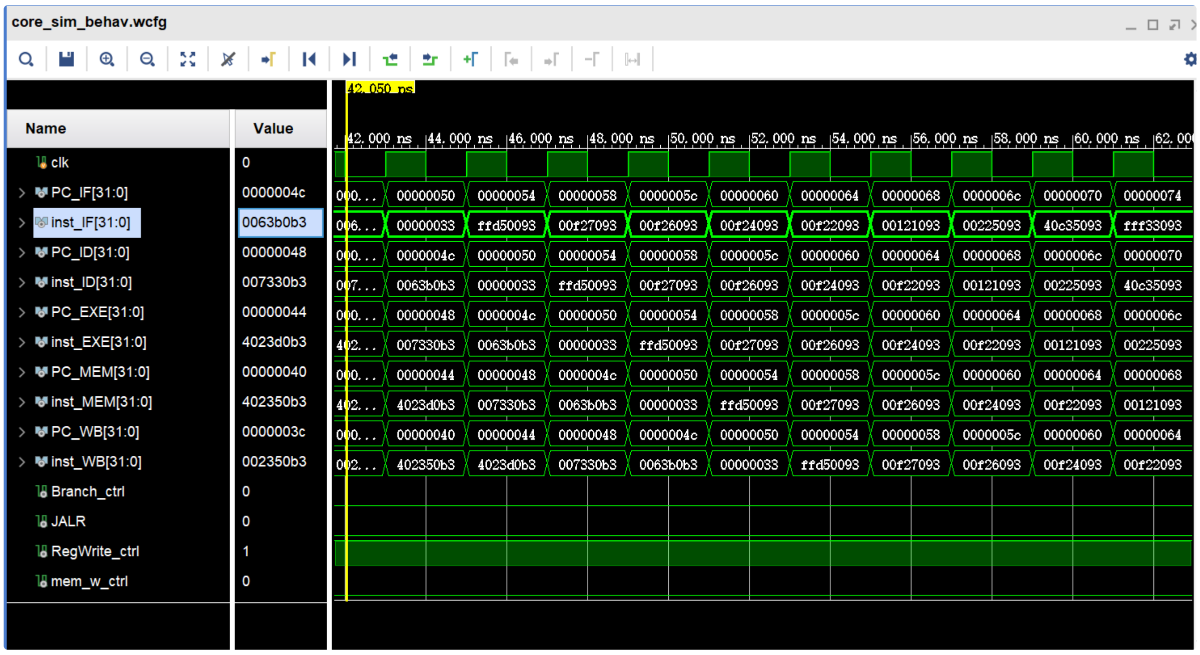
```
MUX4T1_32 mux_forward_B(.I0(rs2_data_reg), .I1(ALUout_EXE), .I2(ALUout_MEM),
.I3(Datain_MEM), //to fill sth. in ()
        .s(forward_ctrl_B),.o(rs2_data_ID));

//EX
MUX2T1_32 mux_A_EXE(.I0(rs1_data_EXE), .I1(PC_EXE), .s(ALUSrc_A_EXE),
.o(ALUA_EXE)); //to fill sth. in ()

MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE), .I1(Imm_EXE), .s(ALUSrc_B_EXE),
.o(ALUB_EXE));
```

2. Analysis of Experimental Results

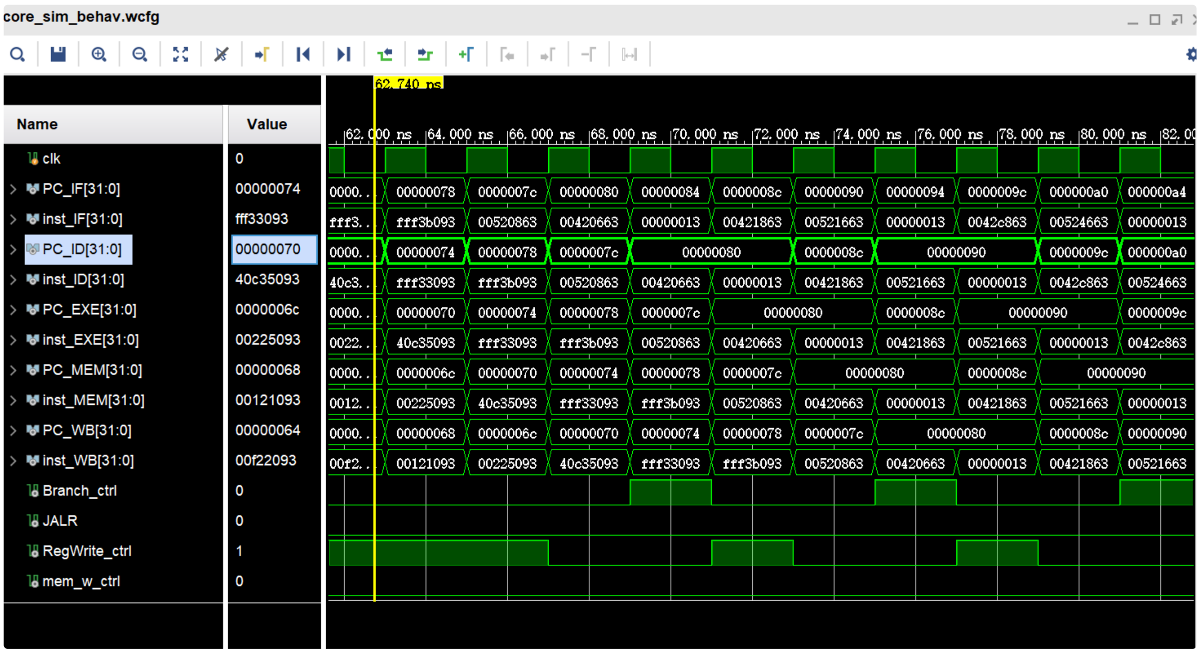
Using the simulation framework provided, we obtained the following waveform:







Name	Value
clk	0
PC_IF[31:0]	0000004c
inst_IF[31:0]	0063b0b3
PC_ID[31:0]	00000048
inst_ID[31:0]	007330b3
PC_EXE[31:0]	00000044
inst_EXE[31:0]	4023d0b3
PC_MEM[31:0]	00000040
inst_MEM[31:0]	402350b3
PC_WB[31:0]	0000003c
inst_WB[31:0]	002350b3
Branch_ctrl	0
JALR	0
RegWrite_ctrl	1
mem_w_ctrl	0



Name	Value
clk	0
PC_IF[31:0]	00000074
inst_IF[31:0]	fff33093
PC_ID[31:0]	00000070
inst_ID[31:0]	40c35093
PC_EXE[31:0]	0000006c
inst_EXE[31:0]	00225093
PC_MEM[31:0]	00000068
inst_MEM[31:0]	00121093
PC_WB[31:0]	00000064
inst_WB[31:0]	00f22093
Branch_ctrl	0
JALR	0
RegWrite_ctrl	1
mem_w_ctrl	0

As shown in the waveform, the simulation results match our expectations.(Due to space constraints, only a portion of the simulation waveform is displayed.)

Next, we performed on-board verification, and the correctness of the results was confirmed during the verification phase.