



本科实验报告

课程名称: 计算机体系结构

设计名称: Pipelined CPU

姓 名: 曾帅王异鸣

学 号: 3190105729 3190102782

学 院: 计算机学院

专 业: 计算机科学与技术

班 级: 图灵 1901

指导老师: 姜晓红

2021 年 10 月 9 日

目录

| | |
|---|----|
| 1 实验内容 | 3 |
| 2 实验原理 | 3 |
| 2.1 指令格式 | 3 |
| 2.2 Control Unit | 4 |
| 2.3 Comperator | 7 |
| 2.4 数据通路 | 7 |
| 2.5 HazardUnit | 14 |
| 3 实验步骤与调试 | 17 |
| 3.1 仿真 | 17 |
| 3.2 综合 | 17 |
| 3.3 实现 | 17 |
| 3.4 验证设计 | 19 |
| 3.5 生成二进制文件 | 20 |
| 3.6 烧写上板 | 20 |
| 4 实验结果与分析 | 20 |
| 4.1 ALU 与 ls 指令的 forwarding 和 stall | 20 |
| 4.2 ALU 指令检验 | 20 |
| 4.3 branch 指令检验 | 20 |
| 4.4 ls 指令、lui、auipc 指令 | 20 |
| 4.5 jalr 指令检验 | 20 |
| 5 讨论与心得 | 24 |

1 实验内容

实验的基本要求是实现 RISC-V 架构下，指令集为 RV32I 并支持 Hazard, forwarding 和 Predict-not-taken 的流水线 CPU。

本实验已给出主要框架，需要完成的实验内容如下：

1. 补充数据通路
2. 设计 ByPass Unit
3. 设计 CPU Controller
4. 利用测试程序验证 CPU 并观察指令的执行情况

2 实验原理

2.1 指令格式

RV32I 的指令格式大致分为一下 6 种，每种指令都是 32 位指令，所以指令在内存中必须以 4 字节对齐。

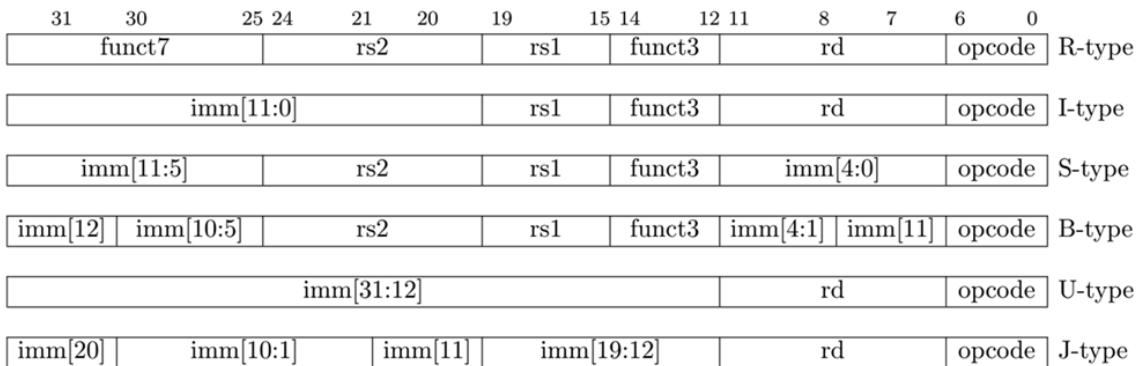


图 2.1: 指令格式

每种指令格式对应者一类或者多类指令，根据每种指令实现需要的条件，对指令格式也有不同的要求，操作码也各不相同

| 操作 | 操作码 | 包含指令 |
|-------------------|---------|--|
| R-type | 0110011 | add, sub, sll, slt, sltu, xor, srl, sra, or, and |
| Arithmetic I-type | 0010011 | addi, slli, srli, srai, andi, ori, xori, slti, sltui |
| load I-type | 0000011 | lw |
| jump I-type | 1100111 | jalr |
| S-type | 0100011 | sw |
| B-type | 1100011 | beq, bne, blt, bge, bltu, bgeu |
| U-type(lui) | 0110111 | lui |
| U-type(auipc) | 0010111 | auipc |
| J-type | 1101111 | jal |

根据指令格式，我们可以从中知道立即数的存储方式：

| 操作类型 | 立即数组织方式 |
|--------|--|
| I-type | $\text{Imm} = \{ 21\{\text{Ins}[31]\}, \text{Ins}[30:25], \text{Ins}[24:21], \text{Ins}[20] \}$ |
| S-type | $\text{Imm} = \{ \{21\{\text{Ins}[31]\}\}, \text{Ins}[30:25], \text{Ins}[11:8], \text{Ins}[7] \}$ |
| U-type | $\text{Imm} = \{ \text{Ins}[31], \text{Ins}[30:20], \text{Ins}[19:12], 12'b0 \}$ |
| J-type | $\text{Imm} = \{ \{12\{\text{Ins}[31]\}\}, \text{Ins}[19:12], \text{Ins}[20], \text{Ins}[30:25], \text{Ins}[24:21], 1'b0 \}$ |
| B-type | $\text{Imm} = \{ \{20\{\text{Ins}[31]\}\}, \text{Ins}[7], \text{Ins}[30:25], \text{Ins}[11:8], 1'b0 \}$ |

需要注意的是，之所以 U-type、J-type、B-type 指令的立即数需要末尾加 0，与 PC 地址恒为 2 的倍数有关，省略恒为 0 的最末尾可以使得立即数表示大一倍，可能我们会注意到 PC 地址实际上恒为 4 的倍数，为什么不直接省略末尾两个 0 呢？是因为还有一种 2Byte 的压缩指令，如果省略两位，那它将无法表示。

2.2 Control Unit

本实验中，ControlUnit 复杂产生指令的控制信号、ALU 的操作信号与比较器的操作信号在此部分仅说明需要填充的内容，首先是用于判断跳转指令类型的信号，根据指令类型 (OpCode) 与 funct3 判断其类型。

| funct3 | 跳转指令类型 | Load 类型 | Store 类型 |
|--------|--------|---------|----------|
| 3'b000 | BEQ | LB | SB |
| 3'b001 | BNE | LH | SH |
| 3'b010 | - | LW | SW |
| 3'b100 | BLT | LBU | - |
| 3'b101 | BGE | LHU | - |
| 3'b110 | BLTU | - | - |
| 3'b111 | BGEU | - | - |

信号 LUI, AUIPC, JAL 与 JALR 只需判断输入 OpCode 是否等于对应 OpCode 即可。而 Branch 跳转控制信号则是 JAL, JALR 与 Branch 是否跳转信号 (Branch & cmp_res) 的或运算，比较器的控制信号直接将输入的 funct3 传入即可。

比较重要的是 ALU 的两个输入口的操作数选择信号，A 口选择器有两个输入：PC 值 (0) 与 rs1_data(1)，根据指令操作方式我们可以判断出只有 AUIPC, JAL 与 JALR 需要在 EXE 阶段对 PC 值进行运算，所以它们的控制信号应该置 0，其他指令均置 1；B 口选择器也有两个输入 rs2_data 与立即数，根据指令操作方式我们判断出 I 类型指令，Load 类型指令，Store 类型指令，LUI 与 AUIPC 均需要在 EXE 阶段利用立即数进行计算。

rs1use 与 rs2use 分别是寄存器的 rs1 与 rs2 是否被使用的信号，根据前述指令类型我们可以比较容易的得到结果。

最后是 hazard_optype 信号，此信号用于传递给 HazardUnit 以便识别当前指令的类型，这个信号的使用可以减少很多信号的传递。

```

1 module CtrlUnit(
2     input [31:0] inst ,
3     input cmp_res ,
4     output Branch , ALUSrc_A , ALUSrc_B , DatatoReg , RegWrite , mem_w ,
5             MIO , rs1use , rs2use ,
6     output [1:0] hazard_optype ,
7     output [2:0] ImmSel , cmp_ctrl ,
8     output [3:0] ALUControl ,
9     output JALR
10 );
11
12     //跳转指令的类型判断信号
13     wire BEQ = Bop & funct3_0;                                //to fill sth. in
14     wire BNE = Bop & funct3_1;                                //to fill sth. in
15     wire BLT = Bop & funct3_4;                                //to fill sth. in
16     wire BGE = Bop & funct3_5;                                //to fill sth. in

```

```

17   wire BLTU = Bop & funct3_6;           //to fill sth. in
18   wire BGEU = Bop & funct3_7;           //to fill sth. in
19   //Load指令的类型判断
20   wire LB = Lop & funct3_0;           //to fill sth. in
21   wire LH = Lop & funct3_1;           //to fill sth. in
22   wire LW = Lop & funct3_2;           //to fill sth. in
23   wire LBU =Lop & funct3_4;          //to fill sth. in
24   wire LHU =Lop & funct3_5;          //to fill sth. in
25   //Store指令的类型判断
26   wire SB = Sop & funct3_0;           //to fill sth. in
27   wire SH = Sop & funct3_1;           //to fill sth. in
28   wire SW = Sop & funct3_2;           //to fill sth. in
29
30   wire LUI    = opcode == 7'b0110111;      //to fill sth. in
31   wire AUIPC = opcode == 7'b0010111;      //to fill sth. in
32
33   wire JAL    = opcode == 7'b1101111;      //to fill sth. in
34   assign JALR = opcode == 7'b1100111;      //to fill sth. in
35   //跳转总控制信号
36   assign Branch = (cmp_res & B_valid) | JAL | JALR; //to fill sth. in
37   //比较器控制信号
38   assign cmp_ctrl = funct3;                //to fill sth. in
39   //ALU的A操作数选择信号
40   assign ALUSrc_A = (AUIPC | JAL | JALR) ? 0 : 1; //to fill
41   sth. in
42   //ALU的B操作数选择信号
43   assign ALUSrc_B = (I_valid | L_valid | S_valid | LUI | AUIPC)? 1 : 0;
44   //to fill sth. in
45
46   assign rs1use = R_valid|S_valid|B_valid|L_valid|I_valid; //to
47   fill sth. in
48
49   assign rs2use = R_valid|S_valid|B_valid; //to
50   fill sth. in
51
52   assign hazard_optype = (R_valid | I_valid | LUI | AUIPC | JAL | JALR)
53   ? 2'b00:
54   S_valid ? 2'b01:
55   L_valid ? 2'b10:2'b11; //to
56   fill sth. in
57
58 endmodule

```

2.3 Comperator

比较器是根据输入控制信号 (ctrl) 进行相应的比较操作并将结果输出，比较器的操作比较简单，就是将对应比较操作的结果输出即可，默认为 0

```
1 module cmp_32(  input [31:0] a,
2                  input [31:0] b,
3                  input [2:0] ctrl ,
4                  output c
5 );
6
7 // to fill sth. in ()
8 assign c = EQ ? res_EQ : 
9           NE ? res_NE : 
10          LT ? res_LT : 
11          LTU ? res_LTU : 
12          GE ? res_GE : 
13          GEU ? res_GEU : 
14           0;
15
16 endmodule
```

2.4 数据通路

完成了个部件的设计，数据通路就是将各个部分连接起来，保证各功能正常实现的关键部分，下面是总体的原理图

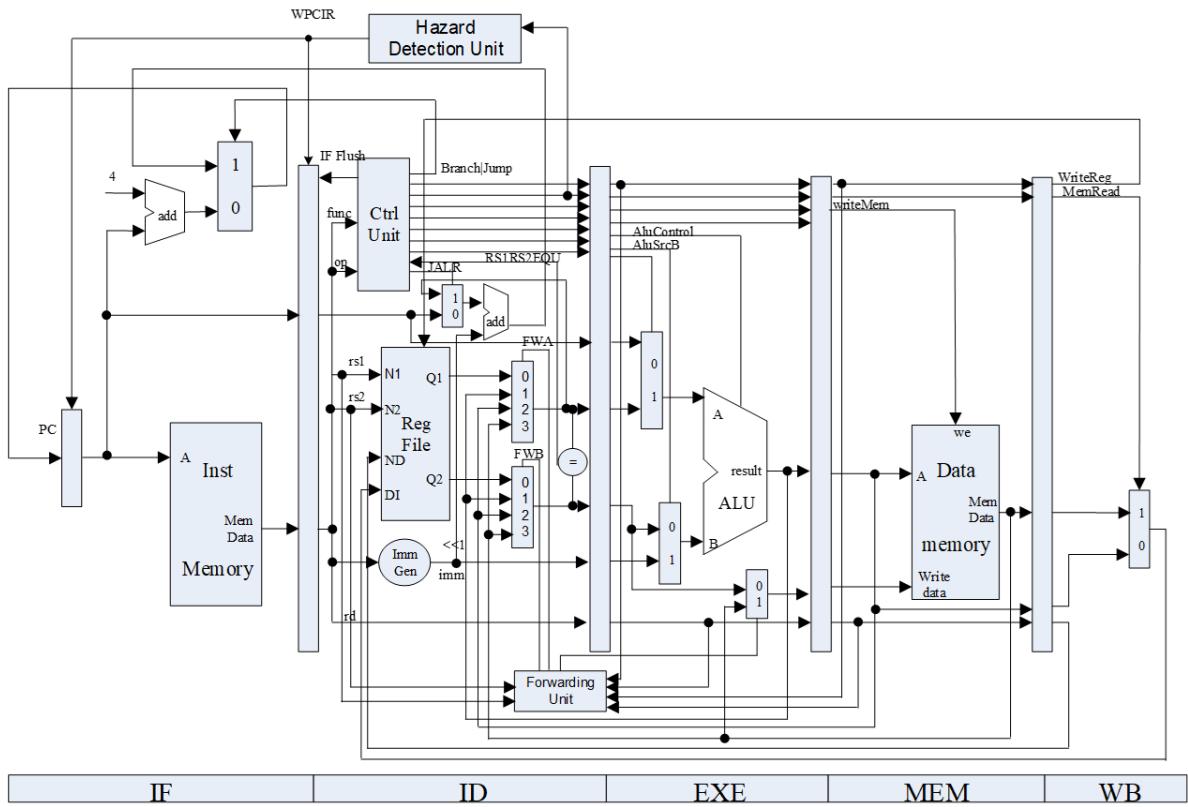


图 2.2: Datapath

数据通路的补充就是根据原理图连线即可，具体的源代码如下

```

1 module RV32core(
2     input debug_en, // debug enable
3     input debug_step, // debug step clock
4     input [6:0] debug_addr, // debug address
5     output[31:0] debug_data, // debug data
6     input clk, // main clock
7     input rst, // synchronous reset
8     input interrupter // interrupt source, for future use
9 );
10
11 wire debug_clk;
12
13 debug_clk clock (.clk(clk), .debug_en(debug_en), .debug_step(debug_step),
14     , .debug_clk(debug_clk));
15
16 wire Branch_ctrl, JALR, RegWrite_ctrl, mem_w_ctrl, MIO_ctrl,
17     ALUSrc_A_ctrl, ALUSrc_B_ctrl, DatatoReg_ctrl, rs1use_ctrl,
     rs2use_ctrl;
18 wire [1:0] hazard_optype_ctrl;

```

```

18   wire [2:0] ImmSel_ctrl , cmp_ctrl;
19   wire [3:0] ALUControl_ctrl;
20
21   wire forward_ctrl_ls ;
22   wire [1:0] forward_ctrl_A , forward_ctrl_B ;
23
24   wire PC_EN_IF;
25   wire [31:0] PC_IF, next_PC_IF, PC_4_IF, inst_IF ;
26
27   wire reg_FD_EN, reg_FD_stall ,reg_FD_flush , cmp_res_ID ;
28   wire [31:0] jump_PC_ID, PC_ID, inst_ID , Debug_regs , rs1_data_reg ,
29     rs2_data_reg ,
30     Imm_out_ID , rs1_data_ID , rs2_data_ID , addA_ID ;
31
32   wire reg_DE_EN, reg_DE_flush , RegWrite_EXE, mem_w_EXE, MIO_EXE,
33     ALUSrc_A_EXE, ALUSrc_B_EXE, ALUzero_EXE, ALUoverflow_EXE ,
34     DatatoReg_EXE;
35   wire [2:0] u_b_h_w_EXE;
36   wire [3:0] ALUControl_EXE;
37   wire [4:0] rs1_EXE , rs2_EXE , rd_EXE;
38   wire [31:0] ALUout_EXE, PC_EXE, inst_EXE , rs1_data_EXE , rs2_data_EXE ,
39     Imm_EXE,
40     ALUA_EXE, ALUB_EXE, Dataout_EXE ;
41
42   wire reg_EM_EN, reg_EM_flush , RegWrite_MEM, DatatoReg_MEM, mem_w_MEM,
43     MIO_MEM;
44   wire [2:0] u_b_h_w_MEM;
45   wire [4:0] rd_MEM;
46   wire [31:0] ALUout_MEM, PC_MEM, inst_MEM , Dataout_MEM , Datain_MEM;
47
48
49   wire reg_MW_EN, RegWrite_WB , DatatoReg_WB ;
50   wire [4:0] rd_WB;
51   wire [31:0] wt_data_WB, PC_WB, inst_WB , ALUout_WB, Datain_WB ;
52
53 // IF
54 REG32 REG_PC(.clk(debug_clk) ,.rst(rst) ,.CE(PC_EN_IF) ,.D(next_PC_IF) ,.Q
  (PC_IF));

```

```

55 //to fill sth. in ()
56 MUX2T1_32 mux_IF(.I0(PC_4_IF), .I1(jump_PC_ID), .s(Branch_ctrl), .o(
      next_PC_IF));
57
58 ROM_D inst_rom(.a(PC_IF[8:2]), .spo(inst_IF));
59
60
61 // ID
62 REG_IF_ID reg_IF_ID(.clk(debug_clk), .rst(rst), .EN(reg_FD_EN), .
      Data_stall(reg_FD_stall),
63     .flush(reg_FD_flush), .PCOUT(PC_IF), .IR(inst_IF),
64
65     .IR_ID(inst_ID), .PCurrent_ID(PC_ID));
66
67 CtrlUnit ctrl(.inst(inst_ID), .cmp_res(cmp_res_ID), .Branch(Branch_ctrl),
      .ALUSrc_A(ALUSrc_A_ctrl),
68     .ALUSrc_B(ALUSrc_B_ctrl), .DatatoReg(DatatoReg_ctrl), .RegWrite(
          RegWrite_ctrl),
69     .mem_w(mem_w_ctrl), .MIO(MIO_ctrl), .rs1use(rs1use_ctrl), .rs2use(
          rs2use_ctrl),
70     .hazard_optype(hazard_optype_ctrl), .ImmSel(ImmSel_ctrl), .cmp_ctrl(
          cmp_ctrl),
71     .ALUControl(ALUControl_ctrl), .JALR(JALR));
72
73 Regs register(.clk(debug_clk), .rst(rst), .L_S(RegWrite_WB), .R_addr_A(
      inst_ID[19:15]),
74     .R_addr_B(inst_ID[24:20]), .rdata_A(rs1_data_reg), .rdata_B(
          rs2_data_reg),
75     .Wt_addr(rd_WB), .Wt_data(wt_data_WB),
76     .Debug_addr(debug_addr[4:0]), .Debug_regs(Debug_regs));
77
78 ImmGen imm_gen(.ImmSel(ImmSel_ctrl), .inst_field(inst_ID), .Imm_out(
      Imm_out_ID));
79
80 //to fill sth. in ()
81 MUX4T1_32 mux_forward_A(.I0(rs1_data_reg), .I1(ALUout_EXE), .I2(
      ALUout_MEM), .I3(Datain_MEM),
82     .s(forward_ctrl_A), .o(rs1_data_ID));
83
84 //to fill sth. in ()
85 MUX4T1_32 mux_forward_B(.I0(rs2_data_reg), .I1(ALUout_EXE), .I2(
      ALUout_MEM), .I3(Datain_MEM),

```

```

86     . s ( forward_ctrl_B ) ,. o ( rs2_data_ID ) );
87
88 MUX2T1_32 mux_branch_ID (. I0 (PC_ID) ,. I1 (rs1_data_ID) ,. s (JALR) ,. o (
89     addA_ID ) );
90
91 add_32 add_branch_ID (. a (addA_ID) ,. b (Imm_out_ID) ,. c (jump_PC_ID) );
92
93 cmp_32 cmp_ID (. a (rs1_data_ID) ,. b (rs2_data_ID) ,. ctrl (cmp_ctrl) ,. c (
94     cmp_res_ID ) );
95
96 HazardDetectionUnit hazard_unit (. clk (debug_clk) ,. Branch_ID (Branch_ctrl
97     ) ,. rs1use_ID (rs1use_ctrl) ,
98     . rs2use_ID (rs2use_ctrl) ,. hazard_optype_ID (hazard_optype_ctrl) ,.
99     rd_EXE (rd_EXE) ,
100    . rd_MEM (rd_MEM) ,. rs1_ID (inst_ID [19:15]) ,. rs2_ID (inst_ID [24:20]) ,.
101      rs2_EXE (rs2_EXE) ,
102    . PC_EN_IF (PC_EN_IF) ,. reg_FD_EN (reg_FD_EN) ,. reg_FD_stall (
103      reg_FD_stall) ,
104    . reg_FD_flush (reg_FD_flush) ,. reg_DE_EN (reg_DE_EN) ,. reg_DE_flush (
105      reg_DE_flush) ,
106    . reg_EM_EN (reg_EM_EN) ,. reg_EM_flush (reg_EM_flush) ,. reg_MW_EN (
107      reg_MW_EN) ,
108    . forward_ctrl_ls (forward_ctrl_ls) ,. forward_ctrl_A (forward_ctrl_A) ,
109    . forward_ctrl_B (forward_ctrl_B));
110
111 // EX
112 REG_ID_EX reg_ID_EX (. clk (debug_clk) ,. rst (rst) ,. EN (reg_DE_EN) ,. flush (
113     reg_DE_flush) ,. IR_ID (inst_ID) ,
114     . PCurrent_ID (PC_ID) ,. rs1_addr (inst_ID [19:15]) ,. rs2_addr (inst_ID
115         [24:20]) ,. rs1_data (rs1_data_ID) ,
116     . rs2_data (rs2_data_ID) ,. Imm32 (Imm_out_ID) ,. rd_addr (inst_ID [11:7])
117         ,. ALUSrc_A (ALUSrc_A_ctrl) ,
118     . ALUSrc_B (ALUSrc_B_ctrl) ,. ALUC (ALUControl_ctrl) ,. DatatoReg (
119         DatatoReg_ctrl) ,
120     . RegWrite (RegWrite_ctrl) ,. WR (mem_w_ctrl) ,. u_b_h_w (inst_ID [14:12])
121         ,. MIO (MIO_ctrl) ,
122     . PCurrent_EX (PC_EXE) ,. IR_EX (inst_EXE) ,. rs1_EX (rs1_EXE) ,. rs2_EX (
123         rs2_EXE) ,
124     . A_EX (rs1_data_EXE) ,. B_EX (rs2_data_EXE) ,. Imm32_EX (Imm_EXE) ,. rd_EX (
125         rd_EXE) ,
126     . ALUSrc_A_EX (ALUSrc_A_EXE) ,. ALUSrc_B_EX (ALUSrc_B_EXE) ,. ALUC_EX (

```

```

        ALUControl_EXE) ,
113    .DatatoReg_EX(DatatoReg_EXE) ,.RegWrite_EX(RegWrite_EXE) ,.WR_EX(
        mem_w_EXE) ,
114    .u_b_h_w_EX(u_b_h_w_EXE) ,.MIO_EX(MIO_EXE)) ;
115
116 //to fill sth. in ()
117 MUX2T1_32 mux_A_EX(.I0(PC_EXE) ,.I1(rs1_data_EXE) ,.s(ALUSrc_A_EXE) ,.o(
        ALUA_EXE)) ;
118 //to fill sth. in ()
119 MUX2T1_32 mux_B_EX(.I0(rs2_data_EXE) ,.I1(Imm_EXE) ,.s(ALUSrc_B_EXE) ,.o(
        ALUB_EXE)) ;
120
121 ALU alu (.A(ALUA_EXE) ,.B(ALUB_EXE) ,.Control(ALUControl_EXE) ,
122     .res(ALUout_EXE) ,.zero(ALUzero_EXE) ,.overflow(ALUoverflow_EXE)) ;
123
124 //to fill sth. in ()
125 MUX2T1_32 mux_forward_EX(.I0(rs2_data_EXE) ,.I1(Datain_MEM) ,.s(
        forward_ctrl_ls) ,.o(Dataout_EXE)) ;
126
127
128 // MEM
129 REG_EX_MEM reg_EX_MEM(.clk(debug_clk) ,.rst(rst) ,.EN(reg_EM_EN) ,.flush(
        reg_EM_flush) ,
130     .IR_EX(inst_EXE) ,.PCurrent_EX(PC_EXE) ,.ALUO_EX(ALUout_EXE) ,.B_EX(
        Dataout_EXE) ,
131     .rd_EX(rd_EX) ,.DatatoReg_EX(DatatoReg_EXE) ,.RegWrite_EX(
        RegWrite_EX) ,
132     .WR_EX(mem_w_EXE) ,.u_b_h_w_EX(u_b_h_w_EXE) ,.MIO_EX(MIO_EXE) ,
133
134     .PCurrent_MEM(PC_MEM) ,.IR_MEM(inst_MEM) ,.ALUO_MEM(ALUout_MEM) ,.
        Datao_MEM(Dataout_MEM) ,
135     .rd_MEM(rd_MEM) ,.DatatoReg_MEM(DatatoReg_MEM) ,.RegWrite_MEM(
        RegWrite_MEM) ,
136     .WR_MEM(mem_w_MEM) ,.u_b_h_w_MEM(u_b_h_w_MEM) ,.MIO_MEM(MIO_MEM)) ;
137
138 RAM_B data_ram (.addra(ALUout_MEM) ,.clka(debug_clk) ,.dina(Dataout_MEM) ,
139     .wea(mem_w_MEM) ,.douta(Datain_MEM) ,.mem_u_b_h_w(u_b_h_w_MEM)) ;
140
141
142 // WB
143 REG_MEM_WB reg_MEM_WB(.clk(debug_clk) ,.rst(rst) ,.EN(reg_MW_EN) ,.IR_MEM(
        inst_MEM) ,

```

```

144 .PCurrent_MEM(PC_MEM) ,.ALUO_MEM(ALUout_MEM) ,.Datai(Datain_MEM) ,.
145     rd_MEM(rd_MEM) ,
146 .DatatoReg_MEM(DatatoReg_MEM) ,.RegWrite_MEM(RegWrite_MEM) ,
147 .PCurrent_WB(PC_WB) ,.IR_WB(inst_WB) ,.ALUO_WB(ALUout_WB) ,.MDR_WB(
148     Datain_WB) ,
149 .rd_WB(rd_WB) ,.DatatoReg_WB(DatatoReg_WB) ,.RegWrite_WB(RegWrite_WB
150     )) ;
151
152
153 wire [31:0] Test_signal;
154 assign debug_data = debug_addr[5] ? Test_signal : Debug_regs;
155
156 CPUTEST U1_3(.PC_IF(PC_IF) ,
157                 .PC_ID(PC_ID) ,
158                 .PC_EXE(PC_EXE) ,
159                 .PC_MEM(PC_MEM) ,
160                 .PC_WB(PC_WB) ,
161                 .PC_next_IF(next_PC_IF) ,
162                 .PCJump(jump_PC_ID) ,
163                 .inst_IF(inst_IF) ,
164                 .inst_ID(inst_ID) ,
165                 .inst_EXE(inst_EXE) ,
166                 .inst_MEM(inst_MEM) ,
167                 .inst_WB(inst_WB) ,
168                 .PCEN(PC_EN_IF) ,
169                 .Branch( Branch_ctrl) ,
170                 .PCSource( Branch_ctrl) ,
171                 .RS1DATA(rs1_data_reg) ,
172                 .RS2DATA(rs2_data_reg) ,
173                 .Imm32(Imm_out_ID) ,
174                 .ImmSel(ImmSel_ctrl) ,
175                 .ALUC(ALUControl_ctrl) ,
176                 .ALUSrc_A(ALUSrc_A_ctrl) ,
177                 .ALUSrc_B(ALUSrc_B_ctrl) ,
178                 .A(ALUA_EXE) ,
179                 .B(ALUB_EXE) ,
180                 .ALU_out(ALUout_MEM) ,
181                 .Datai(Datain_MEM) ,

```

```

182     . Datao(Dataout_MEM) ,
183     . Addr(Addr) ,
184     . WR(MWR) ,
185     . MIO(MIO_MEM) ,
186     . WDATA(wt_data_WB) ,
187     . DataoReg(DatatoReg_WB) ,
188     . RegWrite(RegWrite_WB) ,
189     . data_hazard(reg_FD_stall) ,
190     . control_hazard(Branch_ctrl) ,
191
192     . Debug_addr(debug_addr[4:0]) ,
193     . Test_signal(Test_signal)
194 );
195
196 endmodule

```

2.5 HazardUnit

在此次的实验中，Hazard 由结构竞争、数据冒险和控制竞争三部分组成，出于效率与实现复杂度的考虑本模块被设计在 ID 阶段，利用段寄存的 Flush 与 Stall 功能实现插入 bubble 或者清空段寄存器数据并且本次实验还设计了 Forward Units 以减少停顿次数，提高流水线效率，同时本模块中实现了 Predict-Not-Taken 的跳转策略，在 ID 阶段识别并执行跳转语句。具体的设计分几部分阐述如下：

ForwardUnits 前送只涉及到 EX, MEM, WB 三个阶段的数据前送，在此实验中分为 ALU 的两个数据输入口进行讨论，其中 ALU 的 1 号操作数口接受来自：

1. ID 阶段读取的寄存器值的 rs1_val
2. EXE 阶段的上一条指令 EX 阶段的 ALU 计算结果
3. MEM 阶段的上一条指令 EX 阶段的 ALU 计算结果
4. WB 阶段将要写回 Memory 的数据结果

当没有前递时，直接选择 ID 阶段读取的寄存器值，但只涉及到两条运算指令的 RAW 并且 EX 阶段运算得到的结果在 ID 阶段就需要使用时选择 2 号数据，当两条产生冲突的普通运算指令中间隔了一条无关指令，也即 MEM 阶段还未写入的数据在 ID 阶段又需要使用时选择 3 号数据，4 号数据是提供给涉及到 Load 与一条普通运算指令之间产生的数据冲突使用的，也即当 Load 指令所对应的内存数据还未写入寄存器但

ID 阶段又使用到此寄存器时选择数据 4。

所以前送 ALU 操作数 1 口的条件应该是:

1. regw_exe & rs1_ID==rd_EXE & rs1use_ID
2. regw_mem & rs1_ID==rd_MEM & rs1use_ID & mem_load
3. regw_mem & rs1_ID==rd_MEM & rs1use_ID & mem_load
4. others

ALU 操作数的 2 口分析与 1 口类似, ALU 的 2 号操作数口接受来自:

1. ID 阶段读取的寄存器值的 rs2_val
2. EXE 阶段的上一条指令 EX 阶段的 ALU 计算结果
3. MEM 阶段的上一条指令 EX 阶段的 ALU 计算结果
4. WB 阶段将要写回 Memory 的数据结果

前送 ALU 操作数 2 口的条件应该是:

1. regw_exe & rs2_ID==rd_EXE & rs2use_ID
2. regw_mem & rs2_ID==rd_MEM & rs2use_ID & mem_load
3. regw_mem & rs2_ID==rd_MEM & rs2use_ID & mem_load
4. others

这样使得涉及到算数指令的 Read After Write 冲突无需任何停顿即可得到正确结果,使得涉及到 Load 指令的 RAW 情形仅需一次停顿即可避免数据冲突。

除了 ALU 的前递之外,此次实验还存在一种特殊的前递, Load 指令之后紧接 Store 指令并发生冲突时,需要通过 ls 信号控制将 Load 阶段将要写入内存的寄存器值前递给 Store 指令。

Predict-Not-Taken 对于跳转指令的判断,我组均是在 ID 阶段判断并跳转,若 ID 阶段检测到不满足跳转条件 (Branch 指令不跳转且不为 JAL, JALR 指令),则流水线继续执行,若 ID 阶段检测到跳转 (Branch 满足跳转条件或 Jal, Jalr 必定跳转),则将下一条 PC 地址的值修改为跳转的目标地址,并且将 IF/ID 段寄存器的值 Flush 掉(清空),这样可以使得无效指令全部被清空,并且跳转指令的剩下部分继续向后流动直至执行完毕 (例如 jal, jalr 指令还需将 PC + 4 的值写入寄存器) 所以此实验实现了只 Flush 一个段寄存器无需 Stall 的 Predict-Not-Taken 策略

DataHazard 尽管拥有了前递单元，实验中仅需将涉及到 Load 指令的 RAW 情形还是无法仅通过前递解决，也就是说 Load 与后续一条季运算指令的数据冲突需要使得后续指令停顿一拍，在 EX 阶段判断并插入 bubble，并且插入 bubble 的方式就是将 EX/MEM 段寄存器 Flush 掉，并使得 IF/ID 与 ID/EX 段寄存器 Stall(停顿)一个周期

综上原理，我组设计的 HazardUnit 代码如下：

```

1 `timescale 1ps/1ps
2
3 module HazardDetectionUnit(
4     input clk ,
5     input Branch_ID , rs1use_ID , rs2use_ID ,
6     input [1:0] hazard_optype_ID ,
7     input [4:0] rd_EXE, rd_MEM, rs1_ID , rs2_ID , rs2_EXE,
8     output PC_EN_IF, reg_FD_EN, reg_FD_stall , reg_FD_flush ,
9         reg_DE_EN, reg_DE_flush , reg_EM_EN, reg_EM_flush , reg_MW_EN,
10    output forward_ctrl_ls ,
11    output [1:0] forward_ctrl_A , forward_ctrl_B
12 );
13     wire regw_exe , regw_mem;
14     reg [1:0] hazard_optype_EXE;
15     reg [1:0] hazard_optype_MEM;
16     always@(posedge clk) begin
17         hazard_optype_MEM=hazard_optype_EXE;
18         hazard_optype_EXE=hazard_optype_ID ;
19     end
20     wire id_store=hazard_optype_ID==2'b01 ? 1:0;
21     wire exe_load=hazard_optype_EXE==2'b10 ? 1:0;
22     wire mem_load=hazard_optype_MEM==2'b10 ? 1:0;
23     wire exe_store=hazard_optype_EXE==7'b01 ? 1:0;
24     wire mem_store=hazard_optype_MEM==7'b01 ? 1:0;
25     assign regw_exe=(hazard_optype_EXE==2'b00) || (hazard_optype_EXE==2'
26         b10);
27     assign regw_mem=(hazard_optype_MEM==2'b00) || (hazard_optype_MEM==2'
28         b10);
29     wire stall=exe_load && !id_store && ((rs1_ID==rd_EXE && rs1use_ID) ||
30         (rs2_ID==rd_EXE && rs2use_ID));
31
32     assign forward_ctrl_A=(regw_exe && rs1_ID==rd_EXE && rs1use_ID)? 2'b01
33         :
34             (regw_mem && rs1_ID==rd_MEM && rs1use_ID)? (
35                 mem_load ? 2'b11:2'b10):2'b00;

```

```

31  assign forward_ctrl_B=(regw_exe && rs2_ID==rd_EXE && rs2use_ID)? 2'b01
32  :
32  :           (regw_mem && rs2_ID==rd_MEM && rs2use_ID)? (
33  :               mem_load ? 2'b11:2'b10):2'b00;
33  assign forward_ctrl_ls=(rs2_EXE==rd_MEM && mem_load && exe_store)?1:0;
34
35  assign PC_EN_IF!=stall;
36  assign reg_FD_EN=1;
37  assign reg_FD_stall=stall;
38  assign reg_FD_flush=Branch_ID;
39  assign reg_DE_EN=1;
40  assign reg_DE_flush=stall;
41  assign reg_EM_EN=1;
42  assign reg_EM_flush=stall;
43  assign reg_MW_EN=1;
44
45 endmodule

```

3 实验步骤与调试

3.1 仿真

根据已经写好的代码，进行仿真模拟

当 load 指令后紧接着 alu 指令时，既要 stall 又要 forwarding。由图 3.3 可以验证 forwarding 功能已经实现。

branch 指令验证如图 3.4 由图中可以看到 branch 指令已经实现

代码最后，使用 jalr 指令跳转至代码开头 由图 3.5 中可以看到，auipc 指令与 jalr 指令的实现没有错误。

3.2 综合

选择左侧面板的 Run Synthesis 或者点击上方的绿色小三角，选择 Synthesis

3.3 实现

选择左侧面板的 Run Implementation 或者点击上方的绿色小三角，选择 Implementation。值得注意的是执行 implementation 之前应该确保引脚约束存在且正确，同时之前已经综合过最新的代码。

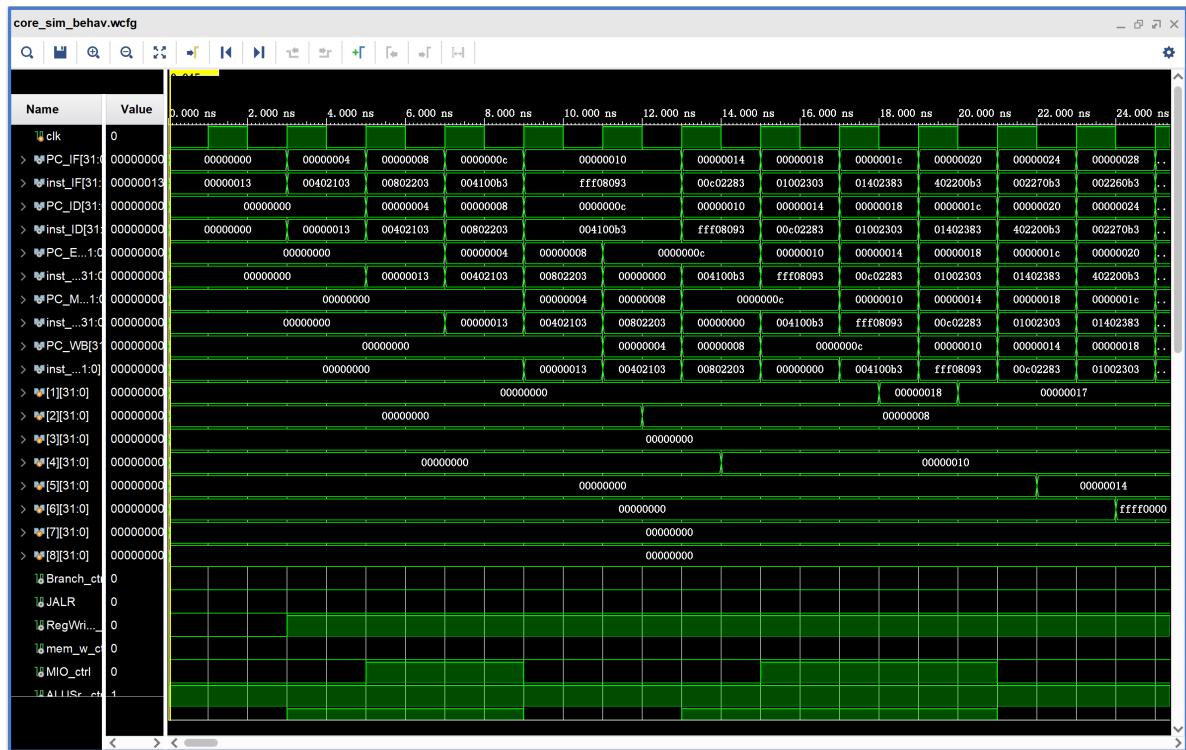


图 3.3: 仿真结果图

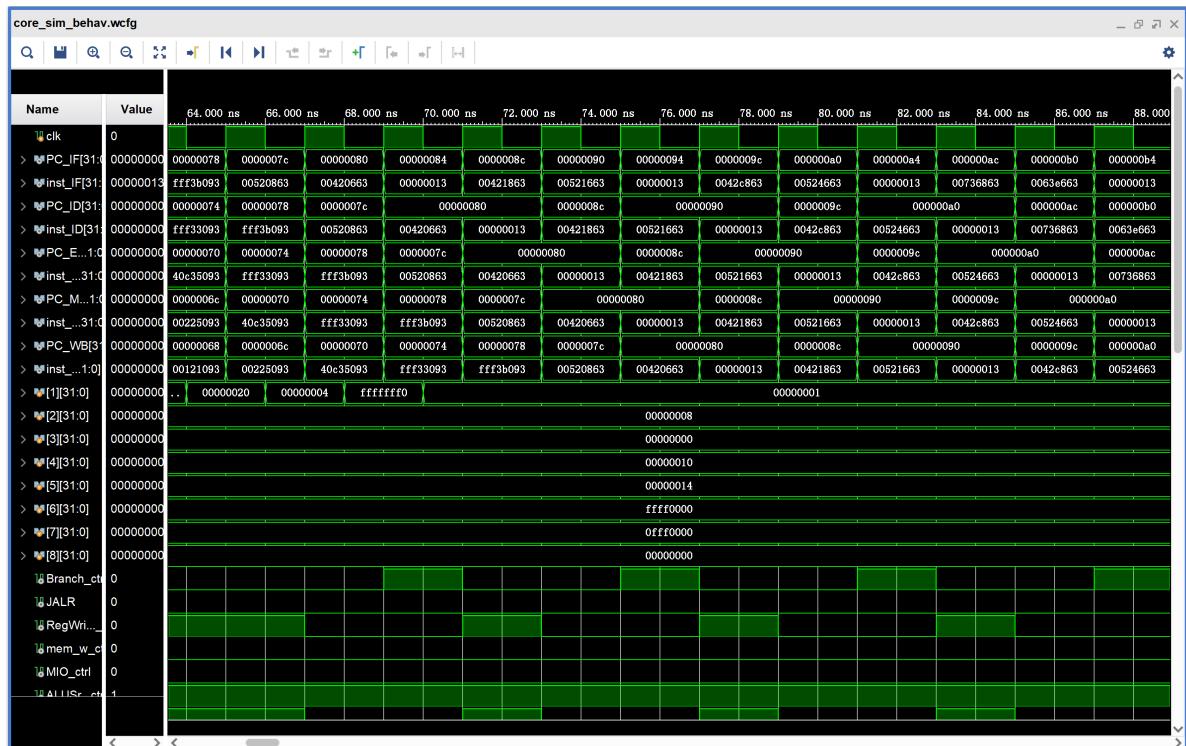


图 3.4: 仿真结果图

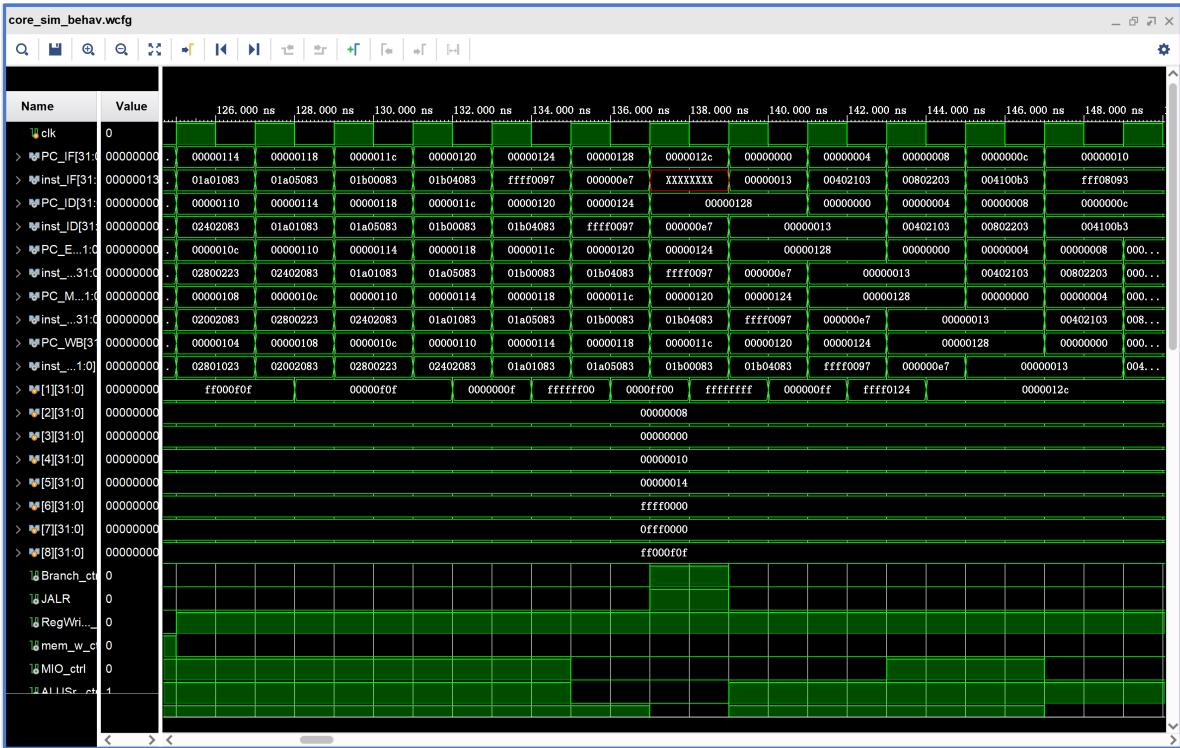


图 3.5: 仿真结果图

3.4 验证设计

选择左侧面板的 Open Elaborated Design, 输出的结果如下, 根据原理图来判断, 基本没有问题

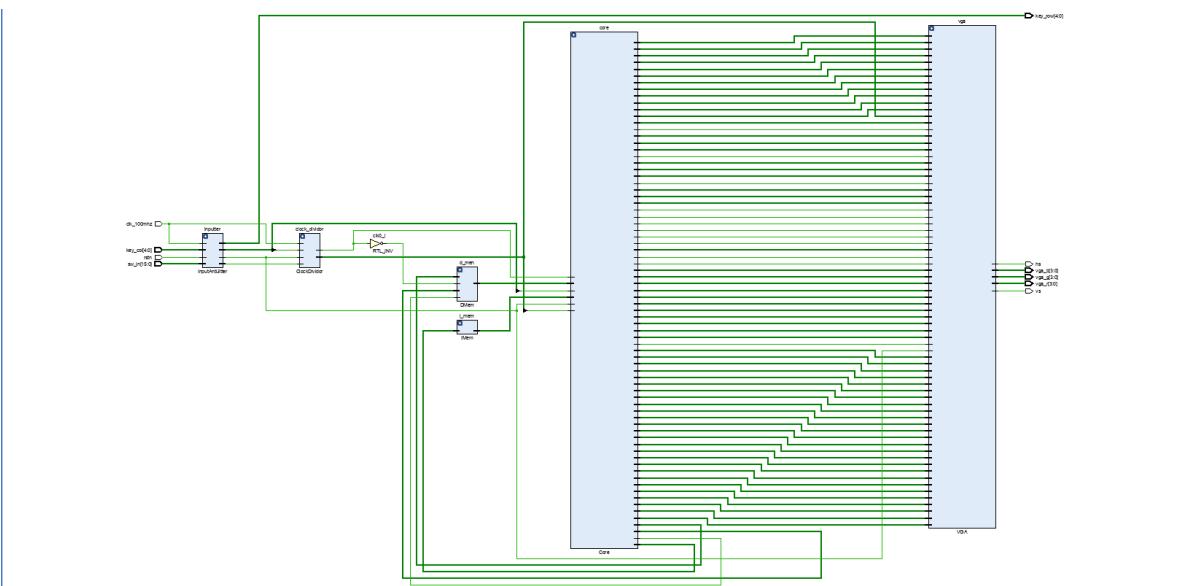


图 3.6: 验证结果图

3.5 生成二进制文件

选择左侧面板的 Generate Bitstream 或者点击上方的绿色二进制标志。同时生成 Bitstream 前要确保：之前已经综合、实现过最新的代码。如没有，直接运行会默认从综合、实现开始。此过程还要注意生成的 bit 文件默认存放在.runs 下相应的 implementation 文件夹中。

3.6 烧写上板

点击左侧的 Open Hardware Manager → 点击 Open Target → Auto Connect → 点击 Program Device → 选择 bistream 路径，烧写。验证结果见实验结果部分。

4 实验结果与分析

经历综合实现，我们可以在开发板上看到测试的结果。我们分以下几个部分验证实验结果的正确性。

4.1 ALU 与 ls 指令的 forwarding 和 stall

forwarding 后紧接着需要用到读入的寄存器的 alu 指令时，需要在 stall 之后再进行 forwarding。在图 4.7 中，load 指令后紧接了一条 ALU 指令，因此我们在 load 指令后插入了一个 stall，之后再进行 forwarding 操作。

4.2 ALU 指令检验

之后是一长串正常的 ALU 指令，通过检测寄存器数值，确定运行结果正确，见图 4.8

4.3 branch 指令检验

之后进入 branch 阶段，我们的 branch 在 ID 阶段实现，并且使用 predict not taken，当需要跳转时，会删除已经读取的指令并插入一个 stall。见图 4.9

4.4 ls 指令、lui、auipc 指令

之后是一系列 load, store, lui, auipc 等指令，经检验，运行结果正确。见图 4.10、4.11

4.5 jalr 指令检验

最后，jalr 指令跳回程序开头。见图 4.12

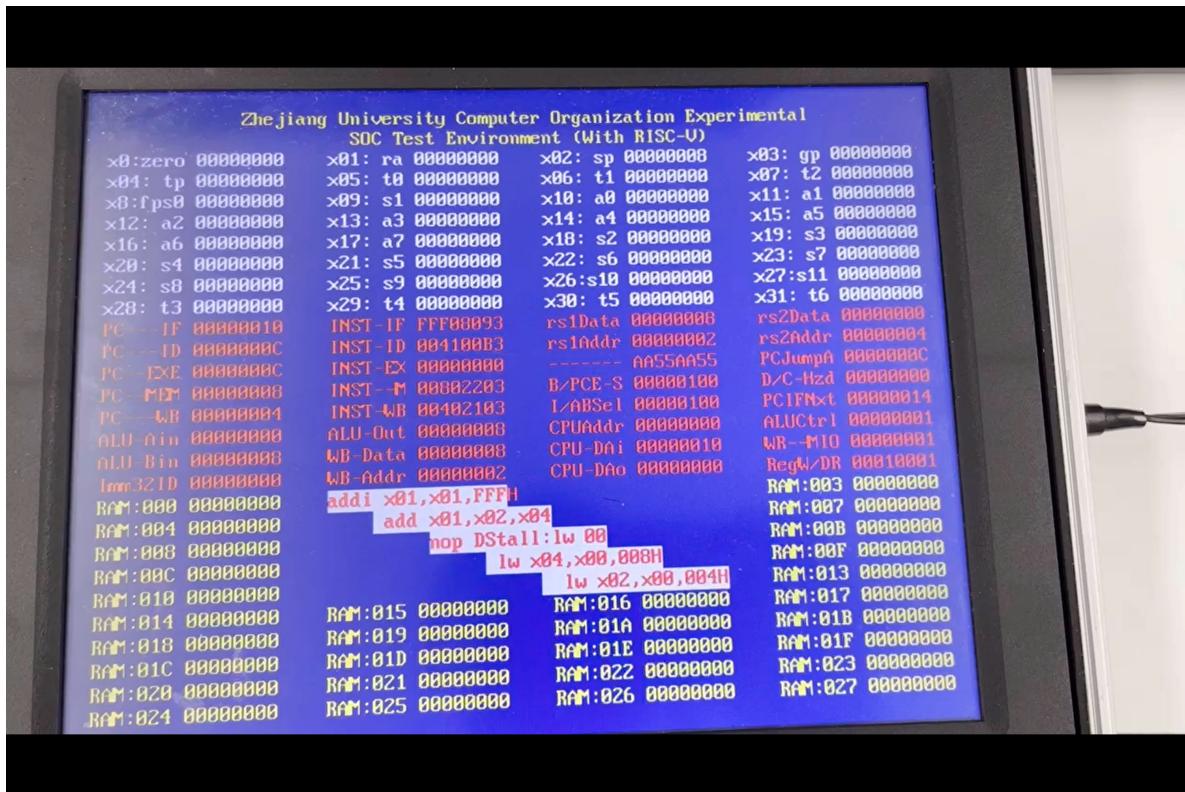


图 4.7: 验证结果图

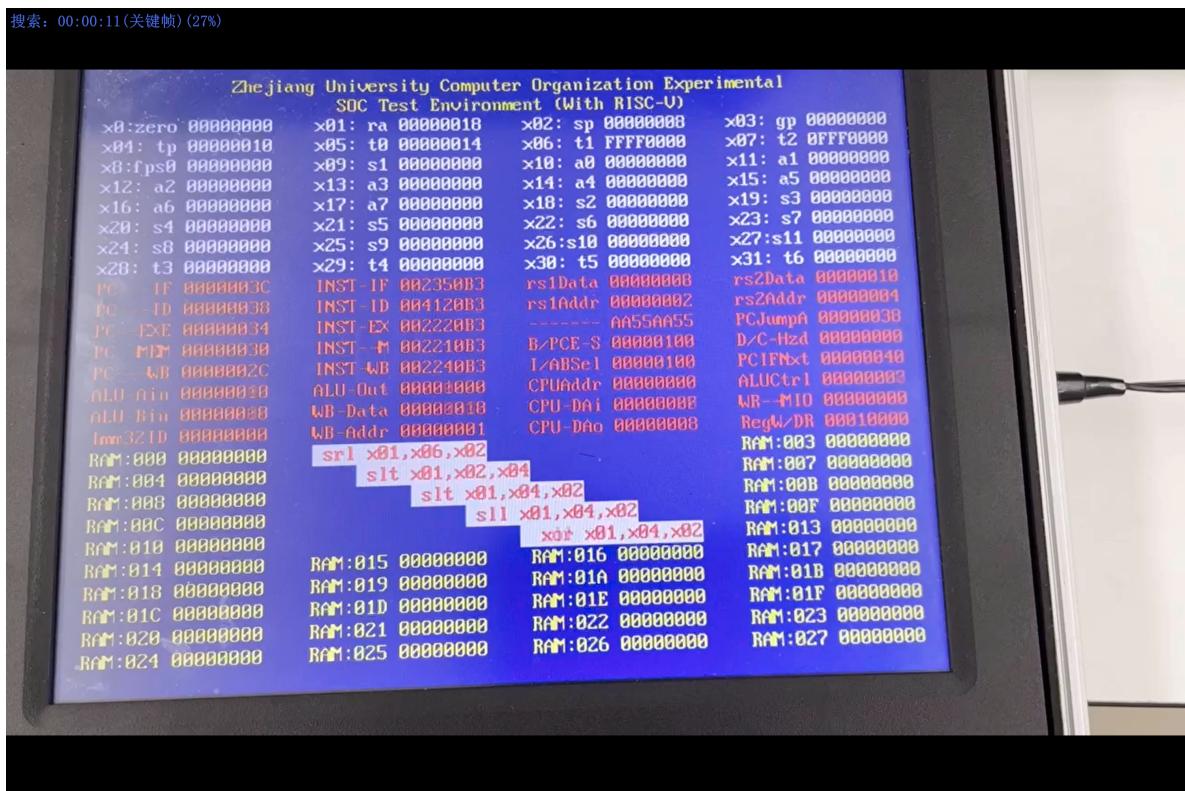


图 4.8: 验证结果图

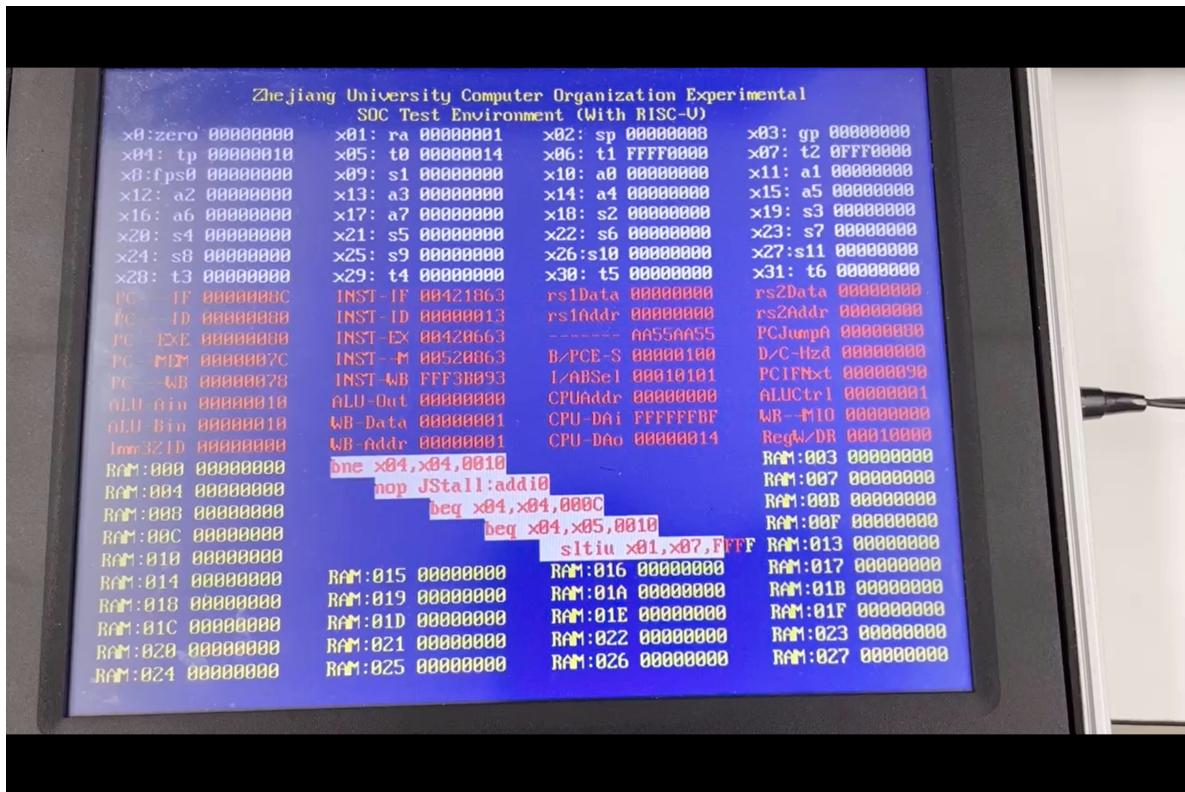


图 4.9: 验证结果图

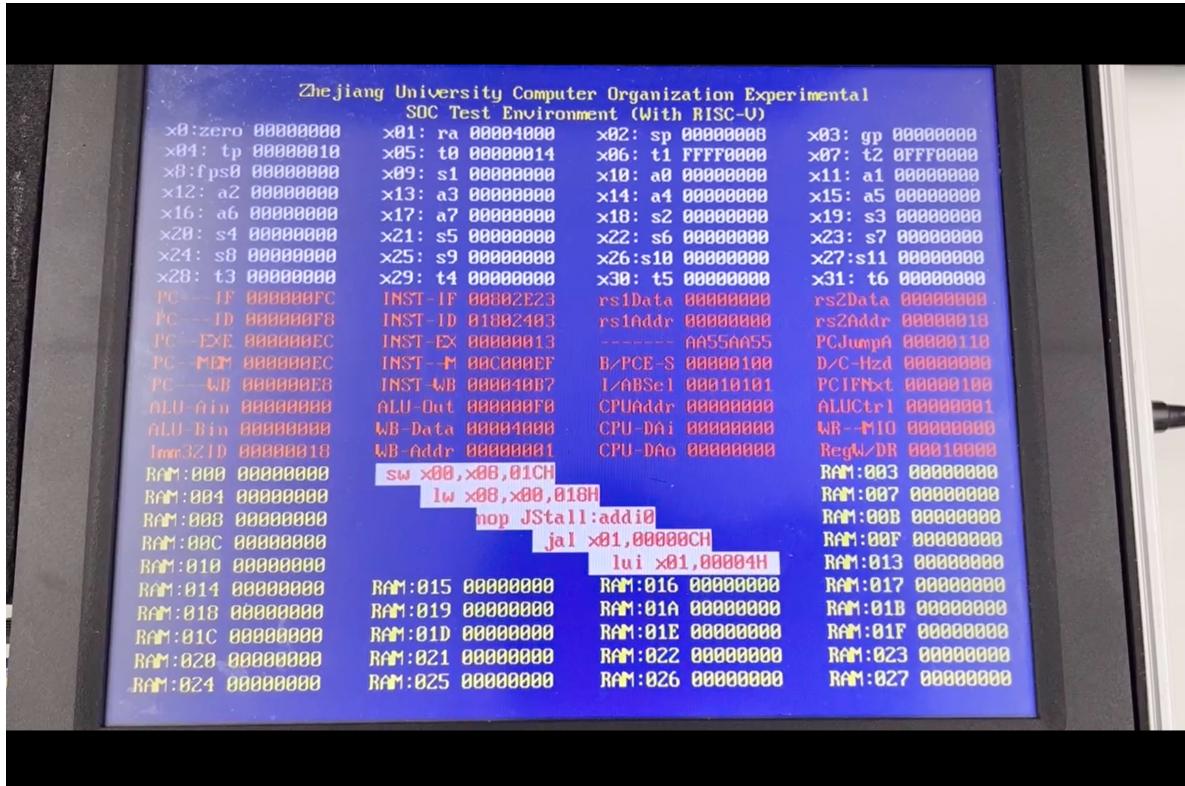


图 4.10: 验证结果图

搜索: 00:00:33(关键帧) (7%)

```

Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-U)

x0:zero 00000000  x01: ra 00000F0F  x02: sp 00000000  x03: gp 00000000
x04: tp 00000010  x05: t0 00000014  x06: t1 FFFF0000  x07: t2 0FFF0000
x08:fps0 FF000F0F  x09: s1 00000000  x10: a0 00000000  x11: a1 00000000
x12: a2 00000000  x13: a3 00000000  x14: a4 00000000  x15: a5 00000000
x16: a6 00000000  x17: a7 00000000  x18: s2 00000000  x19: s3 00000000
x20: s4 00000000  x21: s5 00000000  x22: s6 00000000  x23: s7 00000000
x24: s8 00000000  x25: s9 00000000  x26:s10 00000000  x27:s11 00000000
x28: t3 00000000  x29: t4 00000000  x30: t5 00000000  x31: t6 00000000
PC---IF 00000011C INST--IF 01B00003 rs1Data 00000000
PC---ID 000000118 INST--ID 01A005003 rs1Addr 00000000
PC---EXE 000000114 INST--EX 01A001003 rs1Data AA55AA55
PC---MEM 000000118 INST--M 02A02003 B/PCE-S 00000100
PC---WB 00000010C INST--WB 02B00223 L/ABSel 00010101
ALU-Ain 00000000 ALU-Out 00000024 CPUAddr 00000000
ALU-Bin 00000010 WB-Data 00000024 CPU-Dai 0000000F
Imm32ID 000000010 WB-Addr 00000004 CPU-Dao 00000010
RegW/DR 00000000
RAM:000 00000000 RAM:003 00000000
RAM:004 00000000 RAM:007 00000000
RAM:008 00000000 RAM:00B 00000000
RAM:00C 00000000 RAM:00F 00000000
RAM:010 00000000 RAM:013 00000000
RAM:014 00000000 RAM:017 00000000
RAM:018 00000000 RAM:01B 00000000
RAM:01C 00000000 RAM:01F 00000000
RAM:020 00000000 RAM:023 00000000
RAM:024 00000000 RAM:027 00000000
lw x01,<00,01BH
lw x01,<00,01AH
lw x01,<00,01AH
lw x01,<00,024H
sw x00,<00,024H

```

图 4.11: 验证结果图

```

Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-U)

x0:zero 00000000  x01: ra 0000012C  x02: sp 00000000  x03: gp 00000000
x04: tp 00000010  x05: t0 00000014  x06: t1 FFFF0000  x07: t2 0FFF0000
x08:fps0 FF000F0F  x09: s1 00000000  x10: a0 00000000  x11: a1 00000000
x12: a2 00000000  x13: a3 00000000  x14: a4 00000000  x15: a5 00000000
x16: a6 00000000  x17: a7 00000000  x18: s2 00000000  x19: s3 00000000
x20: s4 00000000  x21: s5 00000000  x22: s6 00000000  x23: s7 00000000
x24: s8 00000000  x25: s9 00000000  x26:s10 00000000  x27:s11 00000000
x28: t3 00000000  x29: t4 00000000  x30: t5 00000000  x31: t6 00000000
PC---IF 000000008 INST--IF 000002203 rs1Data 00000000
PC---ID 000000004 INST--ID 00402103 rs1Addr 00000000
PC---EXE 000000000 INST--EX 000000013 ----- AA55AA55
PC---MEM 000000000 INST--M 000000013 B/PCE-S 00000100
PC---WB 000000000 INST--WB 0000000E7 L/ABSel 00010101
PC1FNxt 0000000C
ALU-Ain 000000000 ALU-Out 000000000 CPUAddr 000000000
ALU-Bin 000000000 WB-Data 00000012C CPU-Dai FFFFFFFF
WR-M10 00000000
Imm32ID 000000004 WB-addr 000000001 CPU-Dao 000000000
RegW/DR 00010000
RAM:000 00000000 RAM:003 00000000
RAM:004 00000000 RAM:007 00000000
RAM:008 00000000 RAM:00B 00000000
RAM:00C 00000000 RAM:00F 00000000
RAM:010 00000000 RAM:013 00000000
RAM:014 00000000 RAM:017 00000000
RAM:018 00000000 RAM:01B 00000000
RAM:01C 00000000 RAM:01F 00000000
RAM:020 00000000 RAM:023 00000000
RAM:024 00000000 RAM:027 00000000
lw x04,<00,000H
lw x02,<00,004H
nop JStall:addi0
nop JStall:addi0
jalr x01,<00,000H

```

图 4.12: 验证结果图

5 讨论与心得

本实验是在上学期的最后一个实验的基础上进行了功能的添加与改进。由于在本次实验中，我们已经有了框架代码，因此我们需要阅读框架代码并了解其主要意思，并在此框架的基础上完成我们最终的代码，这要求我们有一定的代码阅读与理解能力。另外，我们还需要尽快捡起上学期已经学过的知识，打好之后学习和实验的基础。