

浙江大学

本科实验报告

课程名称: 计算机体系结构

设计名称: Scoreboard

姓 名: 曾帅王异鸣

学 号: 3190105729 3190102780

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 16 日

目录

1 实验目的	3
2 实验内容	3
3 实验原理	3
3.1 Datapath	3
3.2 结构竞争与 WAW 解决	10
3.3 WAR 解决	11
3.4 RO 阶段	14
3.5 EX 阶段	14
3.6 WB 阶段	15
4 实验步骤与调试	18
4.1 仿真	18
4.2 综合	18
4.3 实现	18
4.4 验证设计	19
4.5 生成二进制文件	19
4.6 烧写上板	19
5 实验结果与分析	19
5.1 Cache 仿真结果分析	19
5.2 上板结果分析	27
6 讨论与心得	30

1 实验目的

本次实验要求我们实现使用 Scoreboard 的动态调度流水线

1. 理解支持多周期操作流水线的设计原理
2. 理解 Scoreboard 动态调度流水线的设计原理
3. 掌握支持多周期操作流水线的设计方法
4. 掌握 Scoreboard 动态调度流水线的设计方法
5. 掌握验证流水线正确性的方法，并根据设计思路进行验证

2 实验内容

本次实验要求我们实现使用 Scoreboard 的动态调度流水线

本实验已给出主要框架，需要完成的实验内容如下：

1. 重新流水线的 IF/IS/RO/FU/WB 阶段和 FU 阶段使得其支持多周期操作流水线
2. 重新设计 CPU 控制模块，设计 Scoreboard 模块并整个进流水线中
3. 验证 CPU 的正确性并且观察 CPU 的执行情况

3 实验原理

3.1 Datapath

为了支持多周期操作本实验对数据通路进行了修改，想要成功完成此次实验，对 Datapath 的熟练掌握自然是必要的。下面就是本次实验所参照的 Datapath

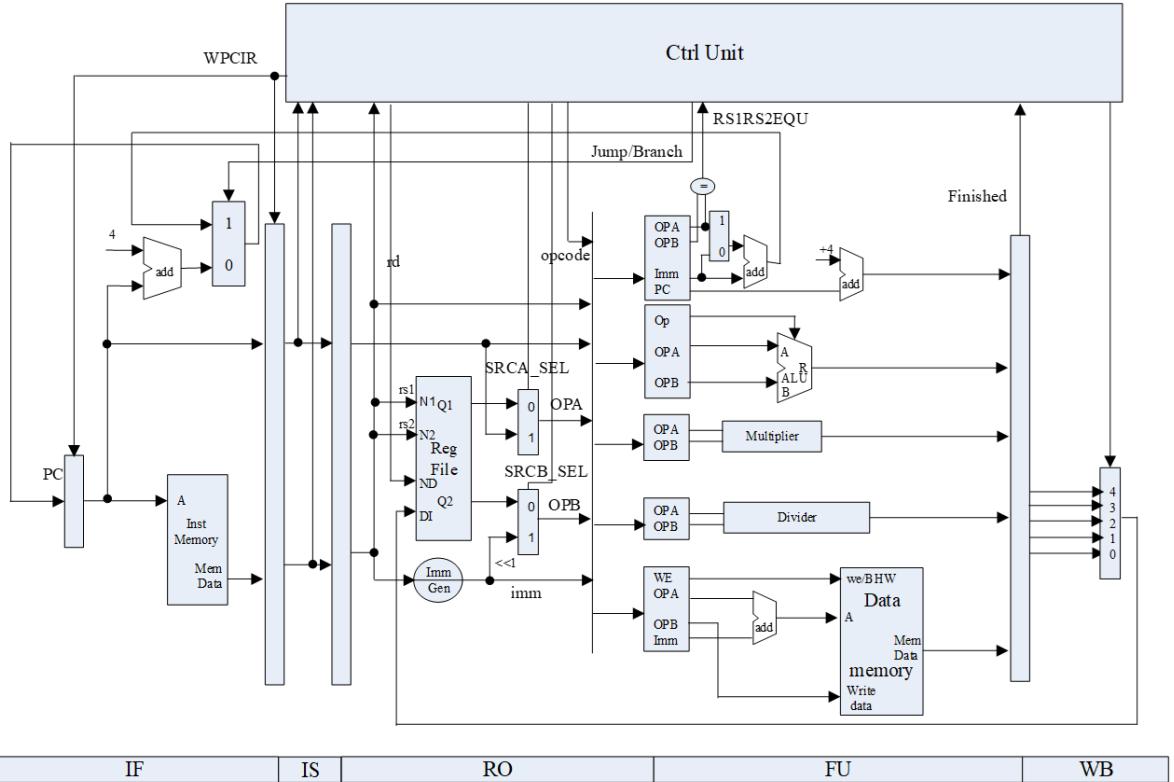


图 3.1: Datapath

根据此图，我们就能够将数据通路部分完成，完成的 RV32Core.v 代码如下所示：

```

1 `timescale 1ns / 1ps
2
3 module RV32core(
4     input debug_en, // debug enable
5     input debug_step, // debug step clock
6     input [6:0] debug_addr, // debug address
7     output[31:0] debug_data, // debug data
8     input clk, // main clock
9     input rst, // synchronous reset
10    input interrupter // interrupt source, for future use
11 );
12
13   wire debug_clk;
14   wire[31:0] debug_regs;
15   reg[31:0] Test_signal;
16   assign debug_data = debug_addr[5] ? Test_signal : debug_regs;
17
18   debug_clk clock (.clk(clk), .debug_en(debug_en), .debug_step(
19     debug_step), .debug_clk(debug_clk));

```

```

19
20     wire PC_EN_IF, IS_EN, FU_ALU_EN, FU_mem_EN, FU_mul_EN, FU_div_EN,
21         FU_jump_EN;
22     wire RegWrite_ctrl, ALUSrcA_ctrl, ALUSrcB_ctrl, mem_w_ctrl;
23     wire [2:0] ImmSel_ctrl, bhw_ctrl, DatatoReg_ctrl;
24     wire [3:0] ALUControl_ctrl;
25     wire [4:0] Jump_ctrl;
26     wire [4:0] rs1_addr_ctrl, rs2_addr_ctrl, rd_ctrl;
27     wire [31:0] PC_ctrl, imm_ctrl;
28
29
30     wire [31:0] PC_IF, next_PC_IF, PC_4_IF, inst_IF ;
31
32     wire [31:0] inst_IS , PC_IS, Imm_out_IS;
33
34     wire [31:0] rs1_data_RO, rs2_data_RO , ALUA_RO, ALUB_RO;
35
36     wire FU_ALU_finish, FU_mem_finish, FU_mul_finish , FU_div_finish ,
37         FU_jump_finish , is_jump_FU;
38     wire [31:0] ALUout_FU, mem_data_FU, mulres_FU , divres_FU , PC_jump_FU
39         , PC_wb_FU;
40
41     // IF
42     assign PC_EN_IF = IS_EN | FU_jump_finish & is_jump_FU;
43
44     REG32 REG_PC(. clk(debug_clk) ,. rst(rst) ,. CE(PC_EN_IF) ,. D(next_PC_IF)
45             ) ,. Q(PC_IF));
46
47     add_32 add_IF (. a(PC_IF) ,. b(32'd4) ,. c(PC_4_IF));
48
49     MUX2T1_32 mux_IF (. I0(PC_4_IF) ,. I1(PC_jump_FU) ,. s(FU_jump_finish &
50         is_jump_FU) ,. o(next_PC_IF));
51
52     ROM_D inst_rom (. a(PC_IF[8:2]) ,. spo(inst_IF));
53
54     // Issue
      REG_IF_IS reg_IF_IS (. clk(debug_clk) ,. rst(rst) ,. EN(IS_EN) ,

```

```

55      . flush ( 1 'b0 ) ,.PCOUT(PC_IF) ,. IR( inst_IF ) ,
56
57      . IR_IS( inst_IS ) ,. PCurrent_IS(PC_IS)) ;
58
59 ImmGen imm_gen( . ImmSel(ImmSel_ctrl) ,. inst_field(inst_IS) ,. Imm_out(
60           Imm_out_IS));
61
62 CtrlUnit ctrl (. clk(debug_clk) ,. rst(rst) ,. PC(PC_IS) ,. inst(inst_IS)
63           ,.imm(Imm_out_IS) ,
64           .ALU_done(FU_ALU_finish) ,.MEM_done(FU_mem_finish) ,.
65           MUL_done(FU_mul_finish) ,.DIV_done(FU_div_finish) ,.
66           JUMP_done(FU_jump_finish) ,.is_jump(is_jump_FU) ,
67           .IS_en(IS_EN) ,. ImmSel(ImmSel_ctrl) ,. ALU_en(FU_ALU_EN) ,
68           .MEM_en(FU_mem_EN) ,. MUL_en(FU_mul_EN) ,. DIV_en(FU_div_EN) ,.
69           JUMP_en(FU_jump_EN) ,
70           .PC_ctrl(PC_ctrl) ,. imm_ctrl(imm_ctrl) ,. rs1_ctrl(
71               rs1_addr_ctrl) ,. rs2_ctrl(rs2_addr_ctrl) ,
72           .JUMP_op(Jump_ctrl) ,. ALU_op(ALUControl_ctrl) ,. ALU_use_PC(
73               ALUSrcA_ctrl) ,. ALU_use_imm(ALUSrcB_ctrl) ,
74           .MEM_we(mem_w_ctrl) ,. MEM_bhw(bhw_ctrl) ,. MUL_op() ,. DIV_op()
75           ,
76           . write_sel(DatatoReg_ctrl) ,. reg_write(RegWrite_ctrl) ,.
77           rd_ctrl(rd_ctrl)
78   );
79
80
81
82
83
84 // FU

```

```

85 FU_ALU alu( .clk(debug_clk) ,.EN(FU_ALU_EN) ,.finish(FU_ALU_finish) ,
86             .ALUControl(ALUControl_ctrl) ,.ALUA(ALUA_RO) ,.ALUB(ALUB_RO)
87             ,.res(ALUout_FU) ,
88             .zero() ,.overflow() );
89
90 FU_mem mem( .clk(debug_clk) ,.EN(FU_mem_EN) ,.finish(FU_mem_finish) ,
91             .mem_w(mem_w_ctrl) ,.bhw(bhw_ctrl) ,.rs1_data(rs1_data_RO) ,.
92             rs2_data(rs2_data_RO) ,
93             .imm(imm_ctrl) ,.mem_data(mem_data_FU) );
94
95 FU_mul mu( .clk(debug_clk) ,.EN(FU_mul_EN) ,.finish(FU_mul_finish) ,
96             .A(rs1_data_RO) ,.B(rs2_data_RO) ,.res(mulres_FU));
97
98 FU_div du( .clk(debug_clk) ,.EN(FU_div_EN) ,.finish(FU_div_finish) ,
99             .A(rs1_data_RO) ,.B(rs2_data_RO) ,.res(divres_FU));
100
101 FU_jump ju( .clk(debug_clk) ,.EN(FU_jump_EN) ,.finish(FU_jump_finish)
102             ,
103             .JALR(Jump_ctrl[4]) ,.cmp_ctrl(Jump_ctrl[3:0]) ,.rs1_data(
104               rs1_data_RO) ,.rs2_data(rs2_data_RO) ,
105               .imm(imm_ctrl) ,.PC(PC_ctrl) ,.PC_jump(PC_jump_FU) ,.PC_wb(
106                 PC_wb_FU) ,.is_jump(is_jump_FU));
107
108 // WB
109 REG32 reg_WB_ALU(.clk(debug_clk) ,.rst(rst) ,.CE(FU_ALU_finish) ,.D(
110   ALUout_FU) ,.Q(ALUout_WB));
111
112 REG32 reg_WB_mem(.clk(debug_clk) ,.rst(rst) ,.CE(FU_mem_finish) ,.D(
113   mem_data_FU) ,.Q(mem_data_WB));
114
115 REG32 reg_WB_mul(.clk(debug_clk) ,.rst(rst) ,.CE(FU_mul_finish) ,.D(
116   mulres_FU) ,.Q(mulres_WB));
117
118 REG32 reg_WB_div(.clk(debug_clk) ,.rst(rst) ,.CE(FU_div_finish) ,.D(
119   divres_FU) ,.Q(divres_WB));
120
121 REG32 reg_WB_jump(.clk(debug_clk) ,.rst(rst) ,.CE(FU_jump_finish) ,.D(
122   (PC_wb_FU) ,.Q(PC_wb_WB));
123
124 MUX8T1_32 mux_DtR(.s(DatatoReg_ctrl) ,.I0(ALUout_WB) ,.I1(
125   mem_data_WB) ,.I2(mulres_WB) ,.I3(divres_WB) ,

```

```

116     .I4(PC_wb_WB) ,.I5(32'd0) ,.I6(32'd0) ,.I7(32'd0) ,.o(
117         wt_data_WB)) ;
118
119     always @* begin
120         case (debug_addr[4:0])
121             0: Test_signal = PC_IF;
122             1: Test_signal = inst_IF ;
123             2: Test_signal = PC_IS;
124             3: Test_signal = inst_IS ;
125
126             4: Test_signal = rs1_addr_ctrl;
127             5: Test_signal = rs1_data_RO ;
128             6: Test_signal = rs2_addr_ctrl;
129             7: Test_signal = rs2_data_RO ;
130
131             8: Test_signal = ImmSel_ctrl;
132             9: Test_signal = imm_ctrl;
133             10: Test_signal = ALUout_FU;
134             11: Test_signal = PC_EN_IF;
135
136             12: Test_signal = {15'b0, FU_ALU_EN, 15'b0,
137                             FU_ALU_finish};
138             13: Test_signal = ALUControl_ctrl;
139             14: Test_signal = ALUA_RO;
140             15: Test_signal = ALUB_RO;
141
142             16: Test_signal = {15'b0, FU_mem_EN, 15'b0,
143                             FU_mem_finish};
144             17: Test_signal = mem_w_ctrl;
145             18: Test_signal = bhw_ctrl;
146             19: Test_signal = mem_data_FU;
147
148             20: Test_signal = {15'b0, FU_mul_EN, 15'b0,
149                             FU_mul_finish};
150             21: Test_signal = mulres_FU;
151             22: Test_signal = {15'b0, FU_div_EN, 15'b0,
152                             FU_div_finish};
153             23: Test_signal = divres_FU;
154
155             24: Test_signal = {15'b0, FU_jump_EN, 15'b0,
156                             FU_jump_finish};

```

```

152          25: Test_signal = Jump_ctrl;
153          26: Test_signal = PC_jump_FU;
154          27: Test_signal = PC_wb_FU;
155
156          28: Test_signal = RegWrite_ctrl;
157          29: Test_signal = rd_ctrl;
158          30: Test_signal = DatatoReg_ctrl;
159          31: Test_signal = wt_data_WB;
160
161      default: Test_signal = 32'hAA55_AA55;
162  endcase
163 end
164
165 endmodule

```

FU_jump FU_jump 模块负责跳转，需要两个时钟周期完成，同样是通过 state 与 reg 寄存器实现跳转时钟周期的控制。

```

1 `timescale 1ns / 1ps
2
3 module FU_jump(
4     input clk, EN, JALR,
5     input [2:0] cmp_ctrl,
6     input [31:0] rs1_data, rs2_data, imm, PC,
7     output [31:0] PC_jump, PC_wb,
8     output cmp_res, finish
9 );
10
11     reg state;
12     assign finish = state == 1'b1;
13     initial begin
14         state = 0;
15     end
16
17     reg JALR_reg;
18     reg [2:0] cmp_ctrl_reg;
19     reg [31:0] rs1_data_reg, rs2_data_reg, imm_reg, PC_reg;
20
21     always @(posedge clk) begin
22         if(EN & ~state) begin
23             JALR_reg <= JALR;
24             cmp_ctrl_reg <= cmp_ctrl;

```

```

25         rs1_data_reg <= rs1_data;
26         rs2_data_reg <= rs2_data;
27         imm_reg <= imm;
28         PC_reg <= PC;
29         state <= 1;
30     end
31     else state <= 0;
32 end
33 // ...           // to fill sth.in
34 wire tmp_res;
35 cmp_32 cmp(.a(rs1_data_reg),
36             .b(rs2_data_reg),
37             .ctrl(cmp_ctrl_reg),
38             .c(tmp_res)
39 );
40
41 assign cmp_res = (JALR_reg | cmp_ctrl_reg == 3'b000) ? 1'b1 :
42     tmp_res;
43 assign PC_wb = PC_reg + 4;
44 assign PC_jump = JALR_reg ? rs1_data_reg + imm_reg : PC_reg +
    imm_reg;
44 endmodule

```

3.2 结构竞争与 WAW 解决

在 Scoreboard 的 IS 阶段，需要检测结构冲突与 WAW 的情形，因此在 Scoreboard 中使用一个信号 normal_stall 来表示结构冲突与 WAW 的情形，并在 IS 阶段检测到 normal_stall 信号为 1 时，将该信号置为 0，当 normal_stall 信号为 1 时 IS 的指令无法进入 RO，会 stall 在 IS 阶段。

判断结构竞争比较简单，只需要查看 FUS 中的对应结构 BUSY 位是否为 1，如果为 1，则说明结构竞争，否则不结构竞争。判断 WAW 需要将 FU 中的写入地址与当前指令的 rd 进行比较。

```

1 // normal stall: structural hazard or WAW
2 assign normal_stall = (FUS[`FU_ALU][`BUSY] & use_ALU) |
3                                         (FUS[`FU_MEM][`BUSY] &
4                                         use_MEM) |
5                                         (FUS[`FU_MUL][`BUSY] &
                                         use_MUL) |
6                                         (FUS[`FU_DIV][`BUSY] &
                                         use_DIV) |

```

```

6          (FUS[`FU_JUMP][`BUSY] &
7              use_JUMP) |
8          (FUS[`FU_ALU][`DST_H:
9              `DST_L] == rd && FUS[
10                 `FU_ALU][`BUSY]) |
11          (FUS[`FU_MEM][`DST_H:
12             `DST_L] == rd && FUS[
13                 `FU_MEM][`BUSY]) |
14          (FUS[`FU_MUL][`DST_H:
15             `DST_L] == rd && FUS[
16                 `FU_MUL][`BUSY]) |
17          (FUS[`FU_DIV][`DST_H:
18             `DST_L] == rd && FUS[
19                 `FU_DIV][`BUSY]) |
20          (FUS[`FU_JUMP][`DST_H:
21             `DST_L] == rd && FUS[
22                 `FU_JUMP][`BUSY]);
23
24 assign IS_en = IS_flush | ~normal_stall & ~ctrl_stall;
25 assign RO_en = ~IS_flush & ~normal_stall & ~ctrl_stall;

```

3.3 WAR 解决

Scoreboard 会在 WB 阶段检查 WAR 的情形，ALU/MEM/DIV/MUL/JUMP 的 WAR 信号设置需要检查其他 FU 部件的读取地址，如果当前指令的写入地址与其他 FU 部件的读取地址相同，并且其他部件仍然没有 ready，则说明 WAR，否则不 WAR。

```

1 // ensure WAR:
2 // If an FU hasn't read a register value (RO), don't write to it.
3 wire ALU_WAR = (FUS[`FU_ALU][`DST_H:`DST_L]==0) ||
4     (FUS[`FU_MUL][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L] || !
5         FUS[`FU_MUL][`RDY1]) & // fill sth. here
6     (FUS[`FU_MUL][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L] || !
7         FUS[`FU_MUL][`RDY2]) & // fill sth. here
8     (FUS[`FU_DIV][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L] || !
9         FUS[`FU_DIV][`RDY1]) & // fill sth. here
10    (FUS[`FU_DIV][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L] || !
11        FUS[`FU_DIV][`RDY2]) & // fill sth. here
12    (FUS[`FU_JUMP][`SRC1_H:`SRC1_L]!= FUS[`FU_ALU][`DST_H:`DST_L] || !
13        FUS[`FU_JUMP][`RDY1]) & // fill sth. here
14    (FUS[`FU_JUMP][`SRC2_H:`SRC2_L]!= FUS[`FU_ALU][`DST_H:`DST_L] || !
15        FUS[`FU_JUMP][`RDY2]) & // fill sth. here

```

```

10      (FUS[`FU_MEM][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L] || !
11          FUS[`FU_MEM][`RDY1]) & // fill sth. here
12      (FUS[`FU_MEM][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L] || !
13          FUS[`FU_MEM][`RDY2])
14 );
15
16 wire MEM_WAR = (
17     (FUS[`FU_ALU][`SRC1_H:`SRC1_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
18         FUS[`FU_ALU][`RDY1]) & // fill sth. here
19     (FUS[`FU_ALU][`SRC2_H:`SRC2_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
20         FUS[`FU_ALU][`RDY2]) & // fill sth. here
21     (FUS[`FU_MUL][`SRC1_H:`SRC1_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
22         FUS[`FU_MUL][`RDY1]) & // fill sth. here
23     (FUS[`FU_MUL][`SRC2_H:`SRC2_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
24         FUS[`FU_MUL][`RDY2]) & // fill sth. here
25     (FUS[`FU_DIV][`SRC1_H:`SRC1_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
26         FUS[`FU_DIV][`RDY1]) & // fill sth. here
27     (FUS[`FU_DIV][`SRC2_H:`SRC2_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
28         FUS[`FU_DIV][`RDY2]) & // fill sth. here
29     (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
30         FUS[`FU_JUMP][`RDY1]) & // fill sth. here
31     (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_MEM][`DST_H:`DST_L] || !
32         FUS[`FU_JUMP][`RDY2]) & // fill sth. here
33 );
34
35 wire MUL_WAR = (
36     (FUS[`FU_ALU][`SRC1_H:`SRC1_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
37         FUS[`FU_ALU][`RDY1]) & // fill sth. here
38     (FUS[`FU_ALU][`SRC2_H:`SRC2_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
39         FUS[`FU_ALU][`RDY2]) & // fill sth. here
40     (FUS[`FU_MEM][`SRC1_H:`SRC1_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
41         FUS[`FU_MEM][`RDY1]) & // fill sth. here
42     (FUS[`FU_MEM][`SRC2_H:`SRC2_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
43         FUS[`FU_MEM][`RDY2]) & // fill sth. here
44     (FUS[`FU_DIV][`SRC1_H:`SRC1_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
45         FUS[`FU_DIV][`RDY1]) & // fill sth. here
46     (FUS[`FU_DIV][`SRC2_H:`SRC2_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
47         FUS[`FU_DIV][`RDY2]) & // fill sth. here
48     (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
49         FUS[`FU_JUMP][`RDY1]) & // fill sth. here
50     (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_MUL][`DST_H:`DST_L] || !
51         FUS[`FU_JUMP][`RDY2]) & // fill sth. here
52 );
53
54

```

```

34 );
35
36 wire DIV_WAR = (
37     (FUS[`FU_ALU][`SRC1_H:`SRC1_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
38         FUS[`FU_ALU][`RDY1]) &           // fill sth. here
39     (FUS[`FU_ALU][`SRC2_H:`SRC2_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
40         FUS[`FU_ALU][`RDY2]) &           // fill sth. here
41     (FUS[`FU_MEM][`SRC1_H:`SRC1_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
42         FUS[`FU_MEM][`RDY1]) &           // fill sth. here
43     (FUS[`FU_MEM][`SRC2_H:`SRC2_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
44         FUS[`FU_MEM][`RDY2]) &           // fill sth. here
45     (FUS[`FU_MUL][`SRC1_H:`SRC1_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
46         FUS[`FU_MUL][`RDY1]) &           // fill sth. here
47     (FUS[`FU_MUL][`SRC2_H:`SRC2_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
48         FUS[`FU_MUL][`RDY2]) &           // fill sth. here
49     (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
50         FUS[`FU_JUMP][`RDY1]) &           // fill sth. here
51     (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_DIV][`DST_H:`DST_L] || !
52         FUS[`FU_JUMP][`RDY2]) &           // fill sth. here
53     (FUS[`FU_MUL][`SRC1_H:`SRC1_L] != FUS[`FU_JUMP][`DST_H:`DST_L] || !
54         FUS[`FU_MUL][`RDY1]) &           // fill sth. here
55     (FUS[`FU_MUL][`SRC2_H:`SRC2_L] != FUS[`FU_JUMP][`DST_H:`DST_L] || !
56         FUS[`FU_MUL][`RDY2]) &           // fill sth. here
);

```

3.4 RO 阶段

只有当 IS 阶段中指令所需的操作数均可用时才能够进入 RO 阶段，否则等待下一个周期。这一个阶段可以动态地解决 RAW 冲突，具体的实现代码如下：

```
1 // RO
2 if (FUS[`FU_JUMP][`RDY1] & FUS[`FU_JUMP][`RDY2]) begin
3     // JUMP
4     FUS[`FU_JUMP][`RDY1] <= 1'b0;
5     FUS[`FU_JUMP][`RDY2] <= 1'b0;
6 end
7 else if (FUS[`FU_ALU][`RDY1] & FUS[`FU_ALU][`RDY2]) begin // fill sth. here.
8     // ALU
9     FUS[`FU_ALU][`RDY1] <= 0;                                // fill sth. here
10
11
12 else if (FUS[`FU_MEM][`RDY1] & FUS[`FU_MEM][`RDY2]) begin // fill sth. here.
13     // MEM
14     FUS[`FU_MEM][`RDY1] <= 0;                                // fill sth. here
15
16
17 else if (FUS[`FU_MUL][`RDY1] & FUS[`FU_MUL][`RDY2]) begin // fill sth. here.
18     // MUL
19     FUS[`FU_MUL][`RDY1] <= 0;                                // fill sth. here
20
21
22 else if (FUS[`FU_DIV][`RDY1] & FUS[`FU_DIV][`RDY2]) begin // fill sth. here.
23     // DIV
24     FUS[`FU_DIV][`RDY1] <= 0;                                // fill sth. here
25
26 end
```

3.5 EX 阶段

在此实验中，EX 阶段等待各 FU 的结果，等到 FU_Done 信号，具体实现代码如下：

```

1 // EX
2 if (!FUS[`FU_ALU][`FU_DONE]) FUS[`FU_ALU][`FU_DONE] <= ALU_done; // fill
   sth. here
3 if (!FUS[`FU_MEM][`FU_DONE]) FUS[`FU_MEM][`FU_DONE] <= MEM_done;
4 if (!FUS[`FU_MUL][`FU_DONE]) FUS[`FU_MUL][`FU_DONE] <= MUL_done;
5 if (!FUS[`FU_DIV][`FU_DONE]) FUS[`FU_DIV][`FU_DONE] <= DIV_done;
6 if (!FUS[`FU_JUMP][`FU_DONE]) FUS[`FU_JUMP][`FU_DONE] <= JUMP_done;

```

3.6 WB 阶段

当没有 WAR Hazard 时才能够进入 WB 阶段，WB 阶段将使用的操作数的 ready bit 置为 1，并且将此指令从 FSU 与 RRS 中移除，具体实现代码如下：

```

1 // WB
2 if (FUS[`FU_JUMP][`FU_DONE] & JUMP_WAR) begin
3   FUS[`FU_JUMP] <= 32'b0;
4   RRS[FUS[`FU_JUMP][`DST_H:`DST_L]] <= 3'b0;
5
6   // ensure RAW
7   if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_ALU][`RDY1]
8     <= 1; // fill sth. here
9   if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_MEM][`RDY1]
10    <= 1; // fill sth. here
11   if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_MUL][`RDY1]
12     <= 1; // fill sth. here
13   if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_DIV][`RDY1]
14     <= 1; // fill sth. here
15
16   if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_ALU][`RDY2]
17     <= 1; // fill sth. here
18   if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_MEM][`RDY2]
19     <= 1; // fill sth. here
20   if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_MUL][`RDY2]
21     <= 1;
22
23   FUS[`FU_JUMP][`BUSY] <= 0; // fill sth. here
24 end
25 // ALU
26 // fill sth. here
27 else if (FUS[`FU_ALU][`FU_DONE] & ALU_WAR) begin

```

```

22   FUS[`FU_ALU] <= 32'b0;
23   RRS[FUS[`FU_ALU][`DST_H:`DST_L]] <= 3'b0;
24
25   // ensure RAW
26   if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_JUMP][`RDY1]
27     <= 1;           // fill sth. here
28   if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_MEM][`RDY1] <=
29     1;           // fill sth. here
30   if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_MUL][`RDY1] <=
31     1;           // fill sth. here
32   if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_DIV][`RDY1] <=
33     1;           // fill sth. here
34
35   if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_JUMP][`RDY2]
36     <= 1;           // fill sth. here
37   if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_MEM][`RDY2] <=
38     1;           // fill sth. here
39   if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_MUL][`RDY2] <=
40     1;           // fill sth. here
41   if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_DIV][`RDY2] <=
42     1;
43
44   FUS[`FU_ALU][`BUSY] <= 0;           // fill sth. here
45 end
46 // MEM
47 // fill sth. here
48 else if (FUS[`FU_MEM][`FU_DONE] & MEM_WAR) begin
49   FUS[`FU_MEM] <= 32'b0;
50   RRS[FUS[`FU_MEM][`DST_H:`DST_L]] <= 3'b0;
51
52   // ensure RAW
53   if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_JUMP][`RDY1]
54     <= 1;           // fill sth. here
55   if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_ALU][`RDY1] <=
56     1;           // fill sth. here
57   if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_MUL][`RDY1] <=
58     1;           // fill sth. here
59   if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_DIV][`RDY1] <=
60     1;           // fill sth. here
61
62   if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_JUMP][`RDY2]
63     <= 1;           // fill sth. here

```

```

51      if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_ALU][`RDY2] <=
52          1;           // fill sth. here
53      if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_MUL][`RDY2] <=
54          1;           // fill sth. here
55      if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_DIV][`RDY2] <=
56          1;
57
58      FUS[`FU_MEM][`BUSY] <= 0;           // fill sth. here
59  end
60 // MUL
61 // fill sth. here
62 else if (FUS[`FU_MUL][`FU_DONE] & MUL_WAR) begin
63     FUS[`FU_MUL] <= 32'b0;
64     RRS[FUS[`FU_MUL][`DST_H:`DST_L]] <= 3'b0;
65
66     // ensure RAW
67     if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_JUMP][`RDY1]
68         <= 1;           // fill sth. here
69     if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_MEM][`RDY1] <=
70         1;           // fill sth. here
71     if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_ALU][`RDY1] <=
72         1;           // fill sth. here
73     if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_DIV][`RDY1] <=
74         1;           // fill sth. here
75
76     if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_JUMP][`RDY2]
77         <= 1;           // fill sth. here
78     if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_MEM][`RDY2] <=
79         1;           // fill sth. here
80     if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_ALU][`RDY2] <=
81         1;           // fill sth. here
82
83     FUS[`FU_MUL][`BUSY] <= 0;           // fill sth. here
84  end
85 // DIV
86 // fill sth. here
87 else if (FUS[`FU_DIV][`FU_DONE] & DIV_WAR) begin
88     FUS[`FU_DIV] <= 32'b0;
89     RRS[FUS[`FU_DIV][`DST_H:`DST_L]] <= 3'b0;

```

```

82    // ensure RAW
83    if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_JUMP][`RDY1]
84        <= 1;           // fill sth. here
85    if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_MEM][`RDY1] <=
86        1;           // fill sth. here
87    if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_MUL][`RDY1] <=
88        1;           // fill sth. here
89    if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_ALU][`RDY1] <=
90        1;           // fill sth. here
91
92    if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_JUMP][`RDY2]
93        <= 1;           // fill sth. here
94    if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_MEM][`RDY2] <=
95        1;           // fill sth. here
96    if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_MUL][`RDY2] <=
97        1;           // fill sth. here
98    if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_ALU][`RDY2] <=
99        1;
100
101    FUS[`FU_DIV][`BUSY] <= 0;           // fill sth. here
102
103 end

```

4 实验步骤与调试

4.1 仿真

根据已经写好的代码，进行仿真模拟

4.2 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

4.3 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。

4.4 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

4.5 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

4.6 烧写上板

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

5 实验结果与分析

5.1 Cache 仿真结果分析

仿真代码设计 为了验证实验结果的正确性，我们采用以下代码对工程进行仿真模拟：

```
1 module sim_top;
2
3     // Inputs
4     reg clk;
5     reg rstn;
6
7     top uut (
8         .CLK_200M_P(clk),
9         .CLK_200M_N(~clk),
10        .SW(16'b0),
11        .RSTN(rstn),
12        .BTN_X(),
13        .BTN_Y(4'b0),
14        .SEGLED_CLK(),
15        .SEGLED_DO(),
16        .SEGLED_PEN(),
17        .LED_PEN(),
18        .LED_DO(),
```

```

19          .LED_CLK() ,
20          .VGA_B() ,
21          .VGA_G() ,
22          .VGA_R() ,
23          .HS() ,
24          .VS()
25      );
26
27      initial begin
28      // Initialize Inputs
29      clk = 0;
30      rstn = 0;
31
32      // Wait 100 ns for global reset to finish
33      #95 rstn = 1;
34
35      // Add stimulus here
36      end
37
38      initial forever #10 clk = ~clk;
39
40 endmodule

```

测试使用的代码如下

```

1 | addi  x0 , x0 , 0 |
2 | lw    x2 , 4(x0)   |
3 | lw    x4 , 8(x0)   |
4 | add   x1 , x2 , x4 |
5 | addi  x3 , x1 , -1 |
6 | lw    x5 , 12(x0)  |
7 | lw    x6 , 16(x0)  |
8 | lw    x7 , 20(x0)  |
9 | sub   x8 ,x4 ,x2   |
10 | addi  x9 ,x10,-3   |
11 | beq   x4 ,x5 ,label0 |
12 | beq   x4 ,x4 ,label0 |
13 | addi  x20 ,x0 ,48   |
14 | addi  x20 ,x0 ,52   |
15 | addi  x20 ,x0 ,56   |
16 | addi  x0 , x0 , 0   |
17 | lw    x2 , 4(x0)   |
18 | lw    x4 , 8(x0)   |

```

```
19 | add  x1 , x2 , x4   |
20 | addi x3 , x1 , -1   |
21 | lw   x5 , 12(x0)   |
22 | lw   x6 , 16(x0)   |
23 | lw   x7 , 20(x0)   |
24 | sub  x8 ,x4 ,x2   |
25 | addi x9 ,x10,-3   |
26 | beq  x4 ,x5 ,label0 |
27 | beq  x4 ,x4 ,label0 |
28 | addi x20 ,x0 ,48   |
29 | addi x20 ,x0 ,52   |
30 | addi x20 ,x0 ,56   |
31 | addi x20 ,x0 ,60   |
32 label0:
33 | lui   x10 ,4        |
34 | jal   x11 ,20       |
35 | addi x20 ,x0 ,72   |
36 | addi x20 ,x0 ,76   |
37 | addi x20 ,x0 ,80   |
38 | addi x20 ,x0 ,84   |
39 | auipc x12 , 0xfffff0 |
40 | div   x13 , x7 , x2   |
41 | mul   x14 , x4 , x5   |
42 | mul   x15 , x13 , x2   |
43 | addi x16 , x0 , 4     |
44 | jalr x17 ,0(x0)   |
```

仿真结果分析 根据上述仿真代码，实验的仿真结果如下，在此部分我组将对仿真结果进行逐一分析。在程序开始时，有一长串的 rst 时间供程序初始化

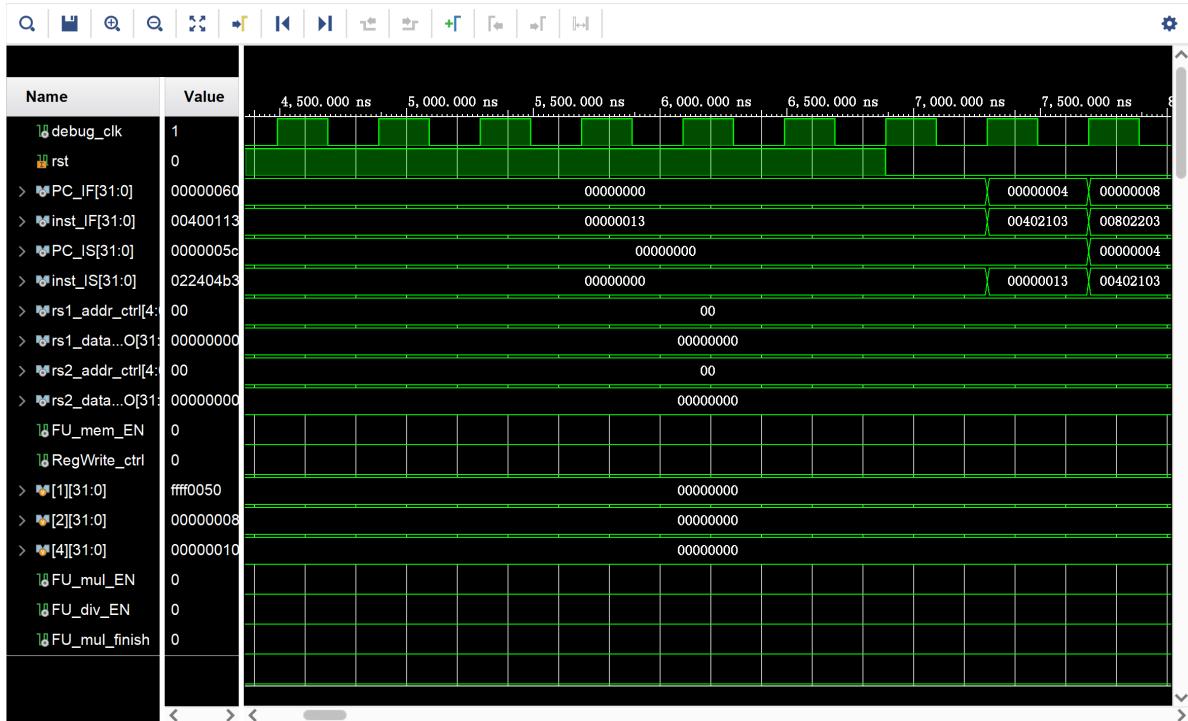


图 5.2: 初始化

之后程序开始运行。程序开头有两条连续的 load 指令。由于我们只有一个 load 单元，在第二条 load 指令 issue 时，程序检测到结构竞争，一直 stall 直到上一条 load 指令执行完毕。

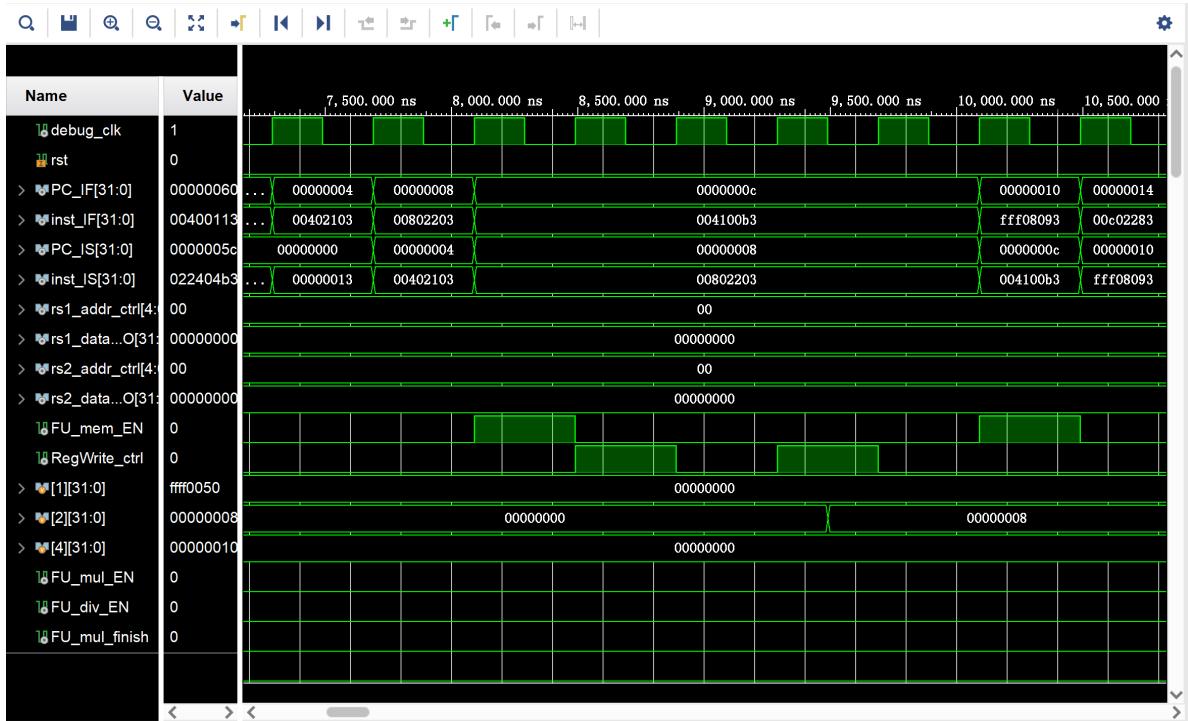


图 5.3: load

之后是两条连续的 ALU 指令，由于两条指令写入的是同一个寄存器，发生了 WAW 冲突，需要 stall 直到第一条 ALU 指令执行完成。

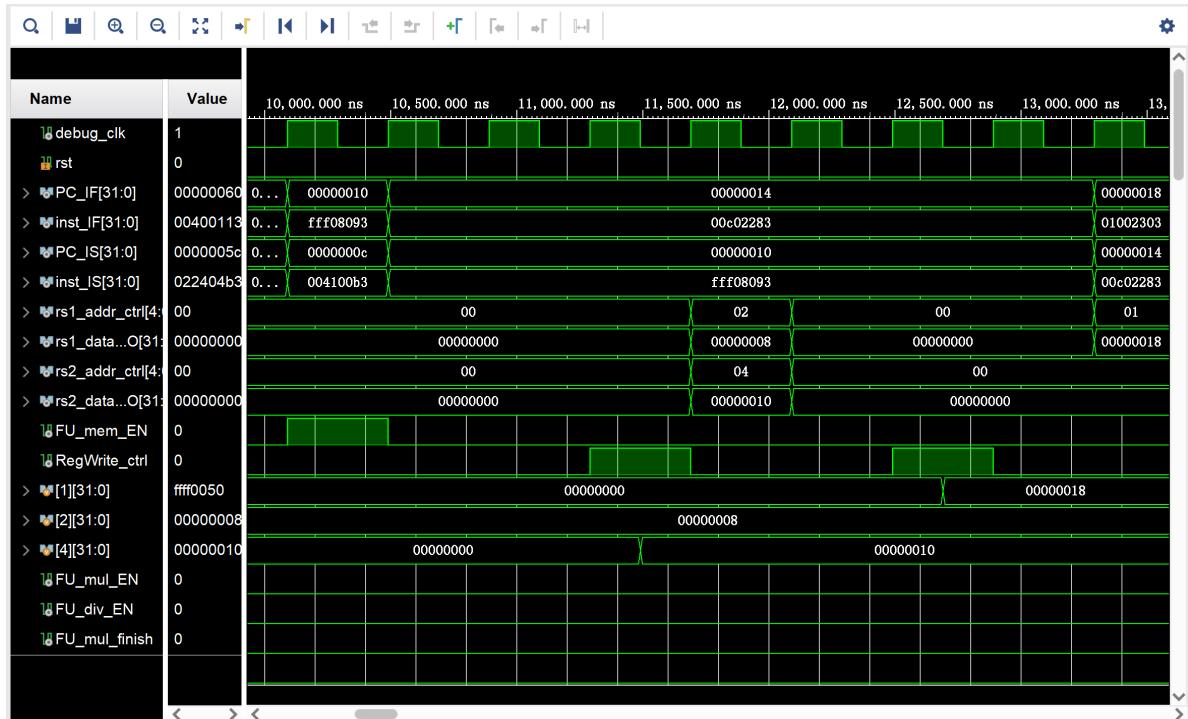


图 5.4: ALU

之后，仍然是一连串的 load 与 ALU 指令，直到程序遇到 branch 指令。当程序执行 jump 指令时，程序 stall 并且停止 issue，直至 jump 指令计算出结果，若不跳转，则继续执行，若跳转则跳转到新的位置继续运行程序。

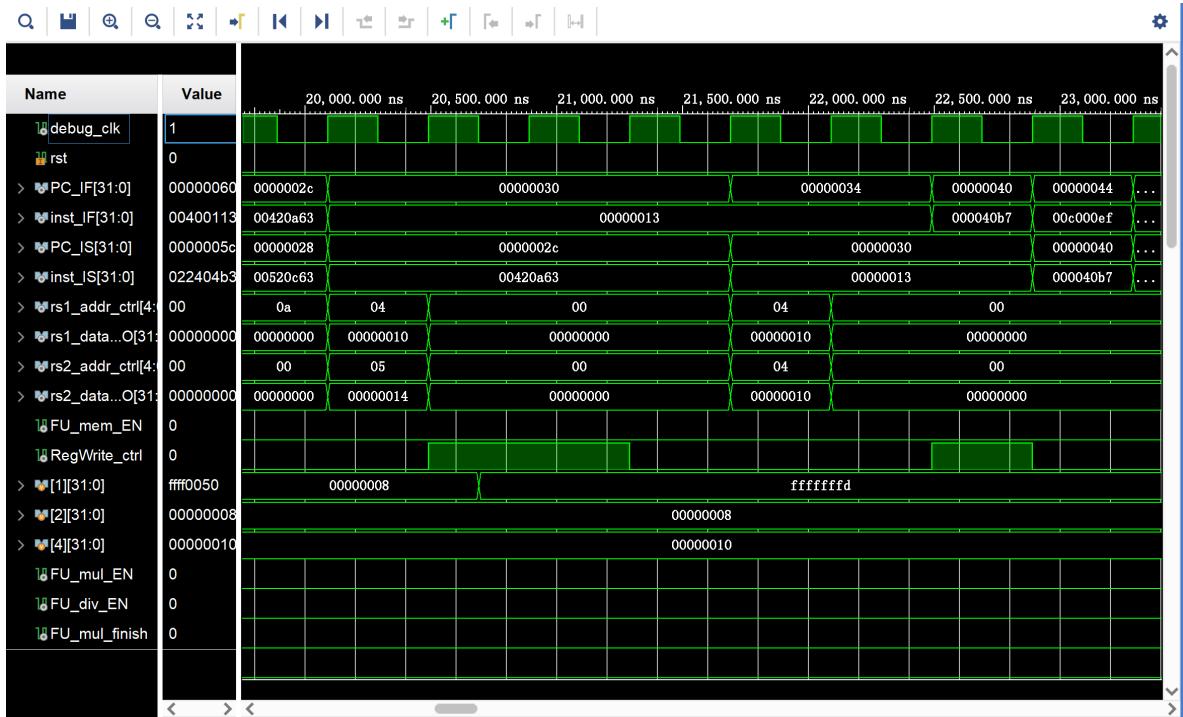


图 5.5: branch

之后程序继续运行，运行时使用的 FU 模块依然与上一次实验相同。

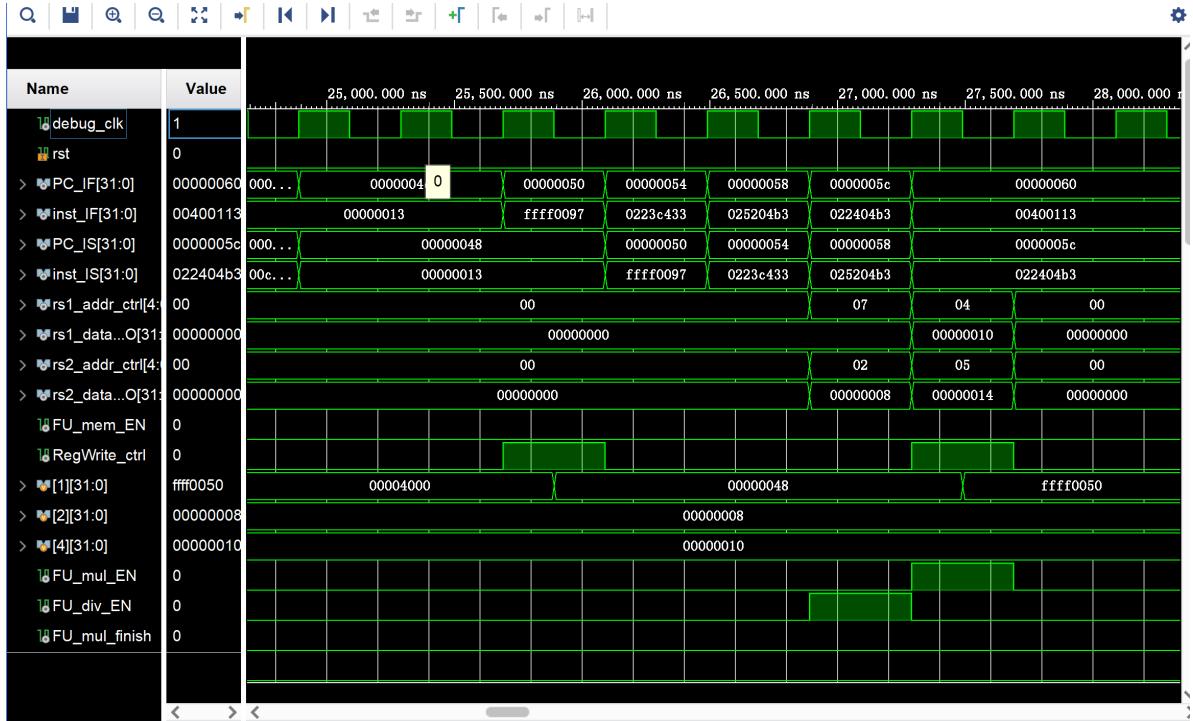


图 5.6: inst

当程序运行至第二条乘法指令时，同时发生了 RAW 和 WAW，因为这条指令写入的寄存器与上一条指令相同，且运算使用到的寄存器在上上条指令写入。

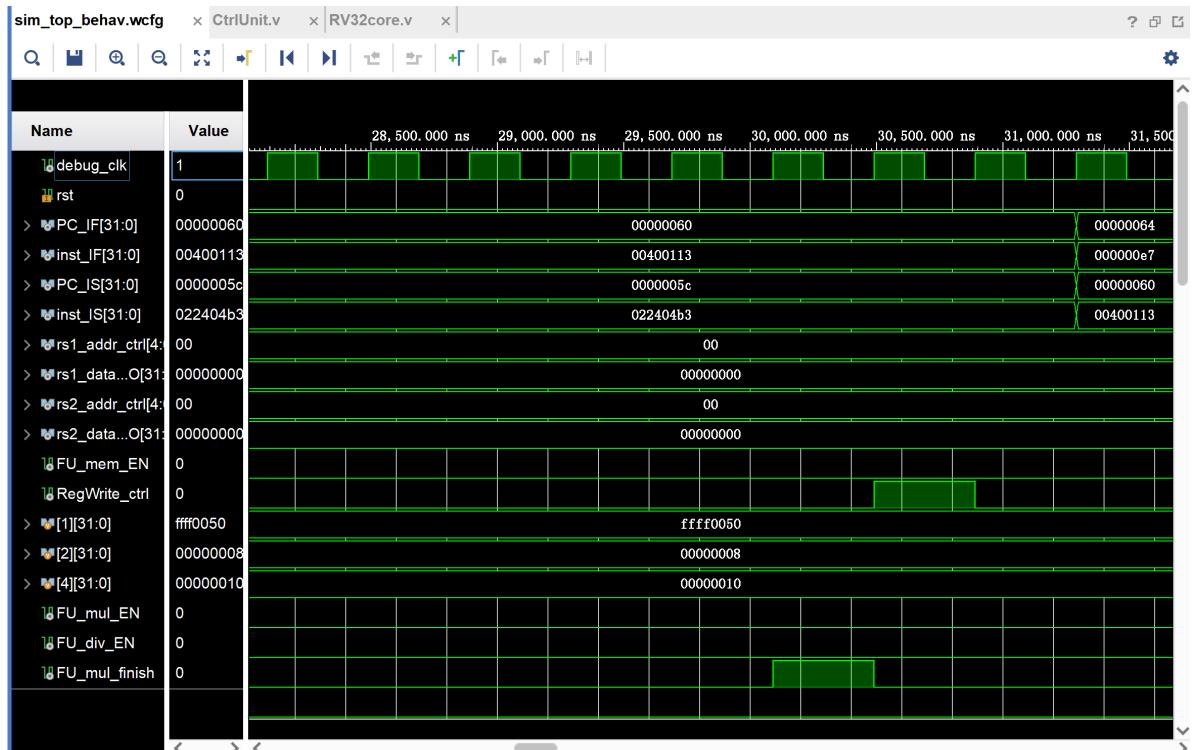


图 5.7: RAW&WAW

下一条 add 指令，写入的寄存器会在 mul 指令中用到，因此检测到 WAR 冲突，因此需要在 mul 指令取出寄存器中的值后才能写入。

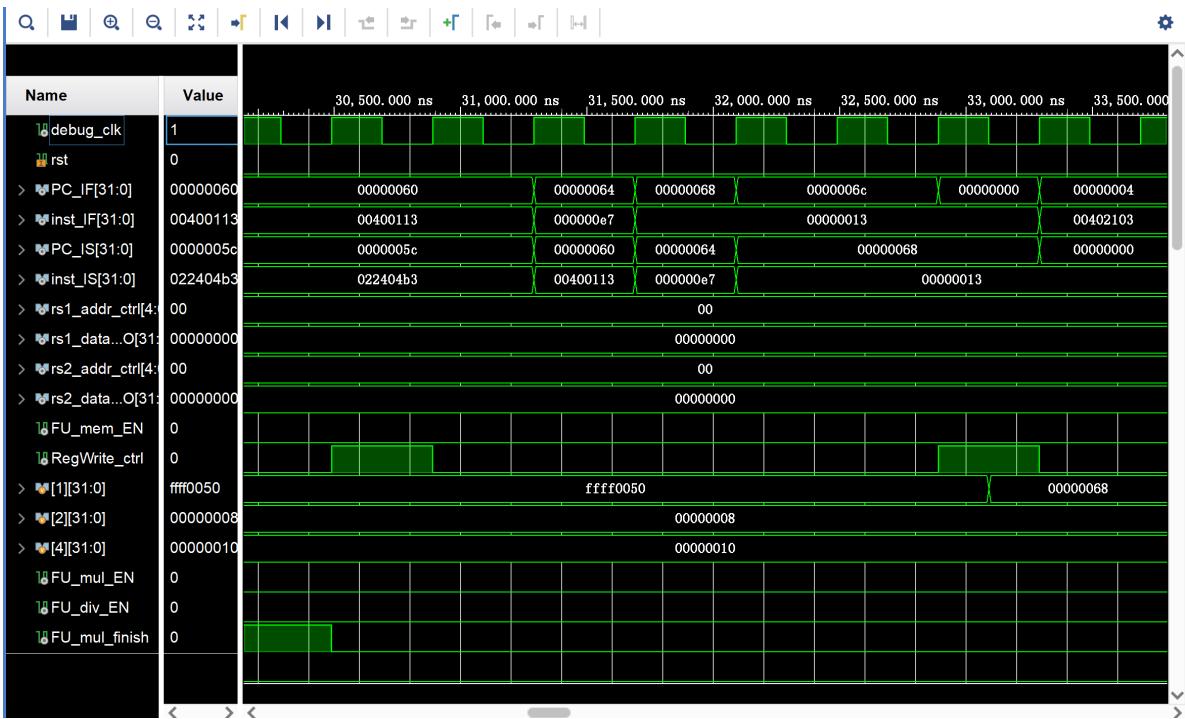


图 5.8: WAR

最后，程序读取到 jalr 指令，跳回程序开头。由于 addi 指令在等待 mul 指令读取寄存器，而 mul 指令在等待 div 指令写入寄存器，ALU 单元一直在使用中，新的 ALU 指令无法发射。一直要等到 div 指令运行完，且 mul 读取完寄存器后才能使用。

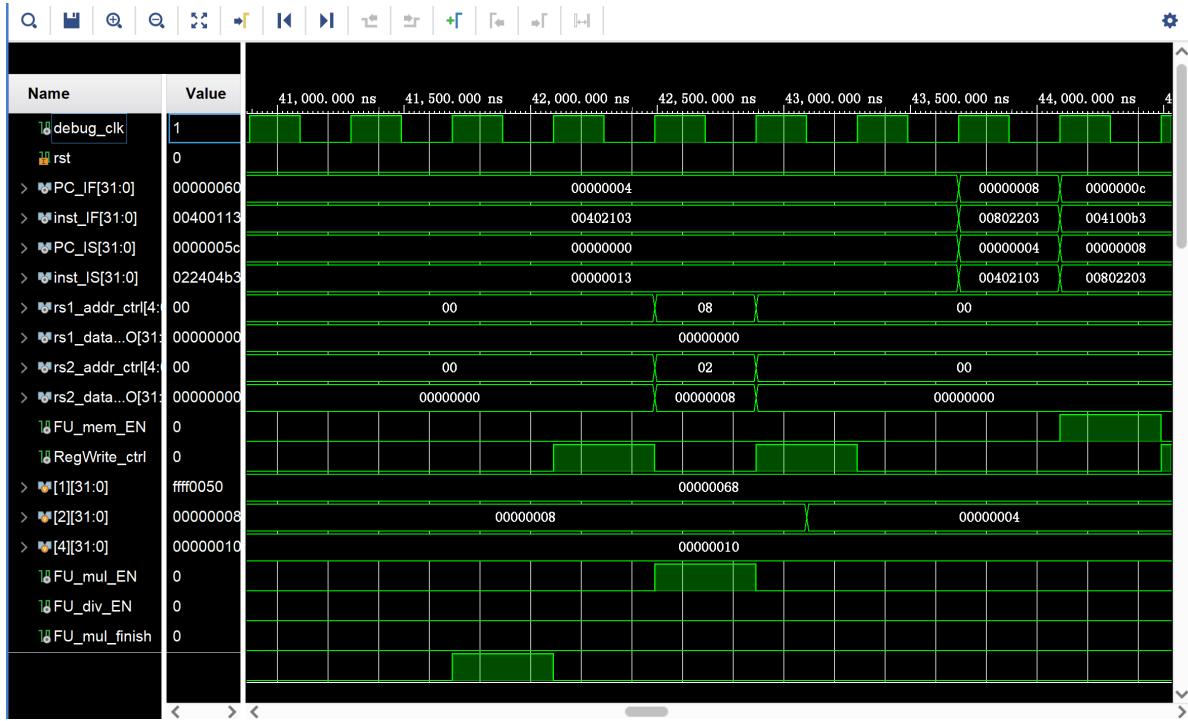


图 5.9: jalr

5.2 上板结果分析

仿真结果正确已经基本验证了实验的正确性，下面我组再简单的通过几张结果截图分析正确性。

在 load 结构冲突时，需要 stall



图 5.10: load

写入同一寄存器时，由于 WAW 造成 stall



图 5.11: WAW

用到仍在运行的指令要写的寄存器，由于 RAW 造成 stall

搜索: 00:00:29(关键帧) (56%)



图 5.12: RAW

跳转回程序开始时，需要等待除法指令运行完才能继续运行



图 5.13: jalr

后续的流程类似，均符合上部分仿真结果，具体流程会在验收时展示。

6 讨论与心得

在本次实验中，我们实现了简单的 scoreboard。我们实现了简单的记分卡，在各个记分卡中记录各个执行单元的状态，并以此为依据完成指令的并发操作。不过，在本次实验中，对于每种指令，仅有一个执行单元，这使得我们的 scoreboard 性能相比之前并没有提升特别大。