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 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

本次实验要求我们实现加入了 cache 的 CPU 流水线

1. 理解支持多周期操作流水线的设计原理
2. 掌握支持多周期操作流水线的设计方法
3. 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支持多周期操作的 CPU 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/ID/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
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

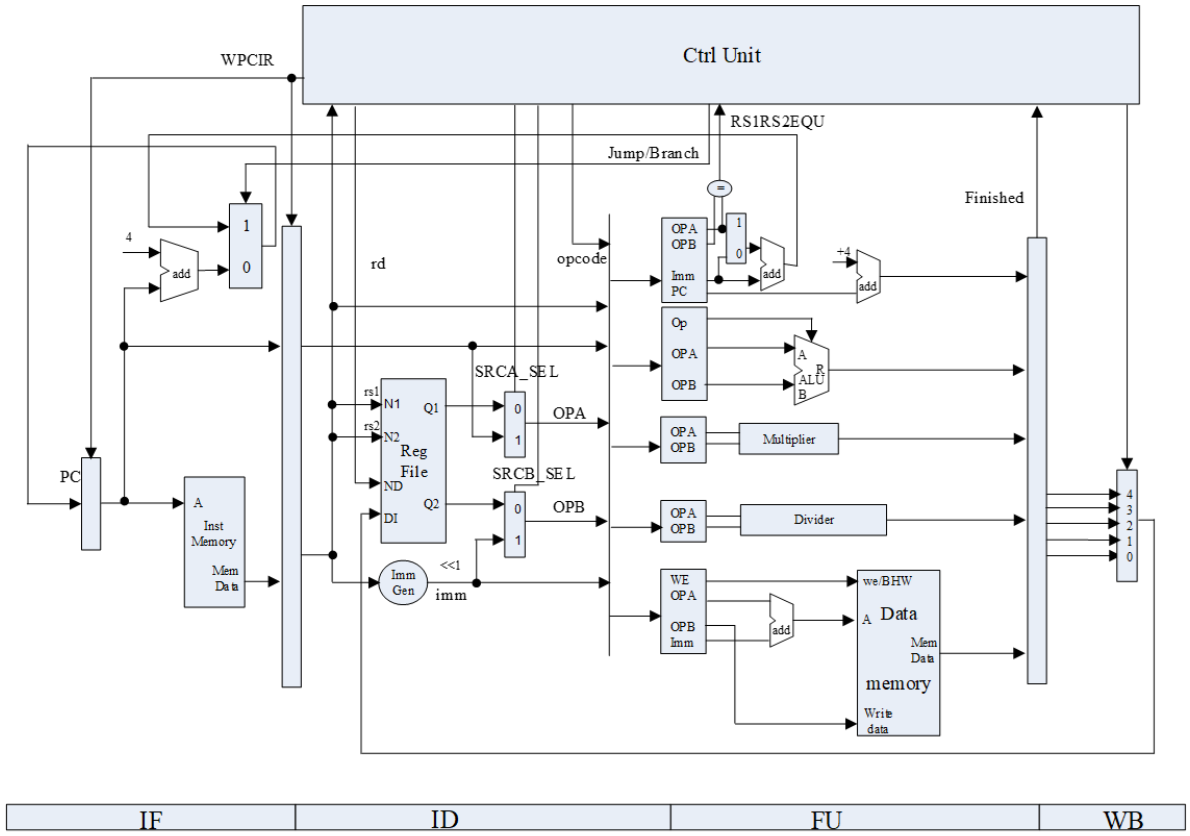

图 3.1: Datapath

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(
```

```
        debug_step),.debug_clk(debug_clk));
19
20      wire reg_IF_EN, reg_ID_EN, reg_ID_flush, FU_ALU_EN, FU_mem_EN,
            FU_mul_EN, FU_div_EN, FU_jump_EN;
21      wire RegWrite_ctrl, ALUSrcA_ctrl, ALUSrcB_ctrl, mem_w_ctrl,
            branch_ctrl;
22      wire[2:0] ImmSel_ctrl, DatatoReg_ctrl;
23      wire[3:0] ALUControl_ctrl, Jump_ctrl;
24      wire[4:0] rd_ctrl;
25
26
27      wire [31:0] PC_IF, next_PC_IF, PC_4_IF, inst_IF;
28
29      wire valid_ID;
30      wire[31:0]inst_ID, PC_ID, Imm_out_ID, rs1_data_ID, rs2_data_ID,
            ALUA_ID, ALUB_ID;
31
32      wire FU_ALU_finish, FU_mem_finish, FU_mul_finish, FU_div_finish,
            FU_jump_finish, cmp_res_FU;
33      wire[31:0]ALUout_FU, mem_data_FU, mulres_FU, divres_FU, PC_jump_FU
            , PC_wb_FU;
34
35      wire[31:0]ALUout_WB, mem_data_WB, mulres_WB, divres_WB, PC_wb_WB,
            wt_data_WB;
36
37
38      // IF
39      REG32 REG_PC(.clk(debug_clk),.rst(rst),.CE(reg_IF_EN),.D(
            next_PC_IF),.Q(PC_IF));
40
41      add_32 add_IF(.a(PC_IF),.b(32'd4),.c(PC_4_IF));
42
43      MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(PC_jump_FU),.s(branch_ctrl),.o(
            next_PC_IF));
44
45      ROM_D inst_rom(.a(PC_IF[8:2]),.spo(inst_IF));
46
47
48      //Issue
49      REG_ID reg_ID(.clk(debug_clk),.rst(rst),.EN(reg_ID_EN),
50              .flush(reg_ID_flush),.PCOUT(PC_IF),.IR(inst_IF),
51
```

```
52                  .IR_ID(inst_ID),.PCurrent_ID(PC_ID),.valid(valid_ID));
53
54          CtrlUnit ctrl(.clk(debug_clk),.rst(rst),.inst(inst_ID),.valid_ID(
                valid_ID),
55                  .ALU_done(FU_ALU_finish),.MEM_done(FU_mem_finish),.
                        MUL_done(FU_mul_finish),
56                  .DIV_done(FU_div_finish),.JUMP_done(FU_jump_finish),.
                        cmp_res_FU(cmp_res_FU),
57
58                  .reg_IF_en(reg_IF_EN),.branch_ctrl(branch_ctrl),.reg_ID_en
                        (reg_ID_EN),
59                  .reg_ID_flush(reg_ID_flush),.ImmSel(ImmSel_ctrl),.ALU_en(
                        FU_ALU_EN),
60                  .MEM_en(FU_mem_EN),.MUL_en(FU_mul_EN),.DIV_en(FU_div_EN),.
                        JUMP_en(FU_jump_EN),
61                  .JUMP_op(Jump_ctrl),.ALU_op(ALUControl_ctrl),.MEM_we(
                        mem_w_ctrl),
62                  .ALUSrcA(ALUSrcA_ctrl),.ALUSrcB(ALUSrcB_ctrl),
63                  .write_sel(DatatoReg_ctrl),.reg_write(RegWrite_ctrl),.
                        rd_ctrl(rd_ctrl));
64
65          ImmGen imm_gen(.ImmSel(ImmSel_ctrl), .inst_field(inst_ID), .
                Imm_out(Imm_out_ID));              //to fill sth.in
66
67          Regs register(.clk(debug_clk),.rst(rst),
68                  .R_addr_A(inst_ID[19:15]),.rdata_A(rs1_data_ID),
69                  .R_addr_B(inst_ID[24:20]),.rdata_B(rs2_data_ID),
70                  .L_S(RegWrite_ctrl),.Wt_addr(rd_ctrl),.Wt_data(wt_data_WB)
                        ,
71                  .Debug_addr(debug_addr[4:0]),.Debug_regs(debug_regs));
72
73          MUX2T1_32 mux_imm_ALU_ID_A(.I0(rs1_data_ID), .I1(PC_ID), .s(
                ALUSrcA_ctrl), .o(ALUA_ID));              //to fill sth.in
74
75          MUX2T1_32 mux_imm_ALU_ID_B(.I0(rs2_data_ID), .I1(PC_ID), .s(
                ALUSrcB_ctrl), .o(ALUB_ID));              //to fill sth.in
76
77
78          // FU
79          FU_ALU alu(.clk(debug_clk),.EN(FU_ALU_EN),.finish(FU_ALU_finish),
80                  .ALUControl(ALUControl_ctrl),.ALUA(ALUA_ID),.ALUB(ALUB_ID)
                        ,.res(ALUout_FU),
```

```
81                  .zero() ,.overflow());
82
83          FU_mem mem(.clk(debug_clk) ,.EN(FU_mem_EN) ,.finish(FU_mem_finish),
84                  .mem_w(mem_w_ctrl) ,.bhw(inst_ID[14:12]) ,.rs1_data(
                        rs1_data_ID) ,.rs2_data(rs2_data_ID),
85                  .imm(Imm_out_ID) ,.mem_data(mem_data_FU));
86
87          FU_mul mu(.clk(debug_clk) ,.EN(FU_mul_EN) ,.finish(FU_mul_finish),
88                  .A(rs1_data_ID) ,.B(rs2_data_ID) ,.res(mulres_FU));
89
90          FU_div du(.clk(debug_clk) ,.EN(FU_div_EN) ,.finish(FU_div_finish),
91                  .A(rs1_data_ID) ,.B(rs2_data_ID) ,.res(divres_FU));
92
93          FU_jump ju(.clk(debug_clk) ,.EN(FU_jump_EN) ,.finish(FU_jump_finish)
                ,
94                  .JALR(Jump_ctrl[3]) ,.cmp_ctrl(Jump_ctrl[2:0]) ,.rs1_data(
                        rs1_data_ID) ,.rs2_data(rs2_data_ID),
95                  .imm(Imm_out_ID) ,.PC(PC_ID) ,.PC_jump(PC_jump_FU) ,.PC_wb(
                        PC_wb_FU) ,.cmp_res(cmp_res_FU));
96
97
98          // WB
99          REG32 reg_WB_ALU(.clk(debug_clk) ,.rst(rst) ,.CE(FU_ALU_finish) ,.D(
                ALUout_FU) ,.Q(ALUout_WB));
100
101         REG32 reg_WB_mem(.clk(debug_clk) ,.rst(rst) ,.CE(FU_mem_finish) ,.D(
                mem_data_FU) ,.Q(mem_data_WB));
102
103         REG32 reg_WB_mul(.clk(debug_clk) ,.rst(rst) ,.CE(FU_mul_finish) ,.D(
                mulres_FU) ,.Q(mulres_WB));
104
105         REG32 reg_WB_div(.clk(debug_clk) ,.rst(rst) ,.CE(FU_div_finish) ,.D(
                divres_FU) ,.Q(divres_WB));
106
107         REG32 reg_WB_jump(.clk(debug_clk) ,.rst(rst) ,.CE(FU_jump_finish) ,.D
                (PC_wb_FU) ,.Q(PC_wb_WB));
108
109         MUX8T1_32 mux_DtR(.s(DatatoReg_ctrl), .I0(32'd0), .I1(ALUout_WB),
                .I2(mem_data_WB), .I3(mulres_WB),
110         .I4(divres_WB), .I5(PC_wb_WB), .I6(32'd0), .I7(32'd0), .o(
                wt_data_WB));  //to fill sth.in
111
```

```
112
113         always @* begin
114                 case (debug_addr[4:0])
115                         0:  Test_signal = PC_IF;
116                         1:  Test_signal = inst_IF;
117                         2:  Test_signal = PC_ID;
118                         3:  Test_signal = inst_ID;
119
120                         4:  Test_signal = inst_ID[19:15];
121                         5:  Test_signal = rs1_data_ID;
122                         6:  Test_signal = inst_ID[24:20];
123                         7:  Test_signal = rs2_data_ID;
124
125                         8:  Test_signal = ImmSel_ctrl;
126                         9:  Test_signal = Imm_out_ID;
127                         10: Test_signal = ALUout_FU;
128                         11: Test_signal = reg_IF_EN;
129
130                         12: Test_signal = {15'b0, FU_ALU_EN, 15'b0,
                                FU_ALU_finish};
131                         13: Test_signal = ALUControl_ctrl;
132                         14: Test_signal = ALUA_ID;
133                         15: Test_signal = ALUB_ID;
134
135                         16: Test_signal = {15'b0, FU_mem_EN, 15'b0,
                                FU_mem_finish};
136                         17: Test_signal = mem_w_ctrl;
137                         18: Test_signal = inst_ID[14:12];
138                         19: Test_signal = mem_data_FU;
139
140                         20: Test_signal = {15'b0, FU_mul_EN, 15'b0,
                                FU_mul_finish};
141                         21: Test_signal = mulres_FU;
142                         22: Test_signal = {15'b0, FU_div_EN, 15'b0,
                                FU_div_finish};
143                         23: Test_signal = divres_FU;
144
145                         24: Test_signal = {15'b0, FU_jump_EN, 15'b0,
                                FU_jump_finish};
146                         25: Test_signal = Jump_ctrl;
147                         26: Test_signal = PC_jump_FU;
148                         27: Test_signal = PC_wb_FU;
```

```
149
150                             28: Test_signal = RegWrite_ctrl;
151                             29: Test_signal = rd_ctrl;
152                             30: Test_signal = DatatoReg_ctrl;
153                             31: Test_signal = wt_data_WB;
154
155                             default: Test_signal = 32'hAA55_AA55;
156                     endcase
157             end
158
159 endmodule
```

3.2 Data Hazard

但流水线支持多周期操作后，势必会更改 Data Hazard 的情形和解决方式，为了解决这一问题，此实验中简单的使用了 FU 阶段的各操作的 finish 信号进行 Data Hazard 的 resolve，只有当一条指令的 FU 阶段结束之后才允许进入 WB 阶段，当然仅仅只控制当前执行指令停顿是远远不够的，此次实验中还将 FU 的结束信号传入了 Ctrl Unit 中，作为 Data Hazard 的检测信号，当 FU 阶段未结束时，Ctrl Unit 中的 FU_in_use 信号会升起，并且会使得后续指令同样等待 FU 阶段结束。

具体例子如下所示：

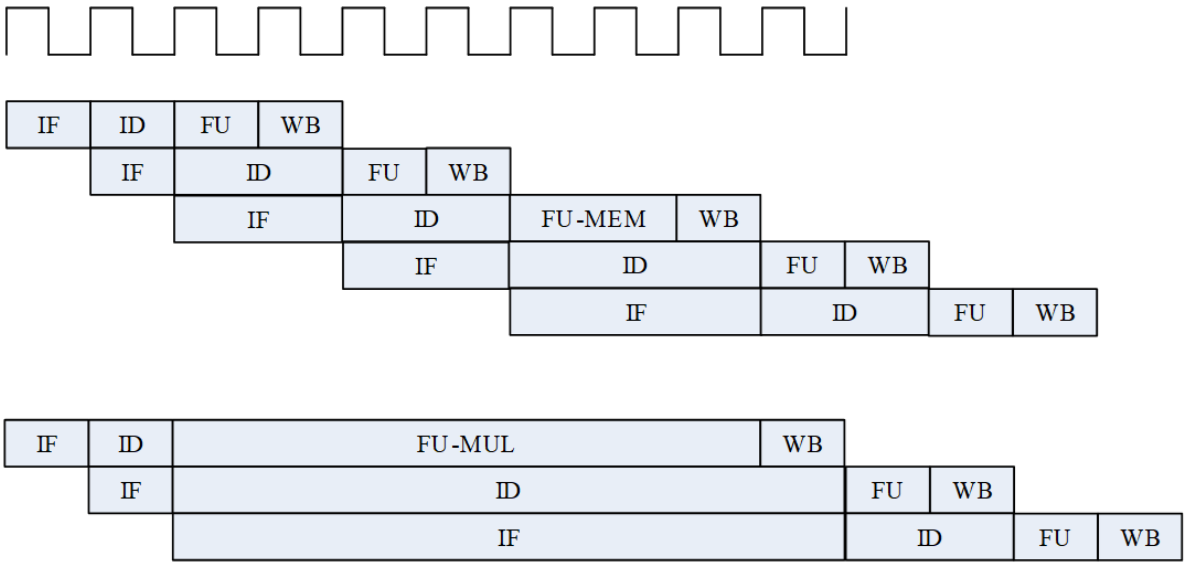

图 3.2: DataHazard

3.3 Control Hazard

Ctrl Unit 中不仅进行 Data Hazard 检测与解决，还实现了 Control Unit 的 resolve 操作，此部分的关键点就在于：在本次实验中有关跳转操作的地址计算与跳转条件判断均是在 FU 阶段计算完成的，结合 FU 阶段的计算结果和 FU 结束的信号，可以进行 Control Hazard 的解决。

本实验还实现了 Predict Not Taken 策略，实现过程同样也是在 Ctrl Unit 中完成，默认认为跳转不会执行，在跳转指令进入 FU 阶段前正常取指与执行但 FU 阶段结束，跳转需要被执行时，通过 Ctrl Unit 中的 reg_ID_flush 信号升起，来 flush 掉 ID 段寄存器，并且将跳转到的指令取入 IF 段寄存器。
总得来说，过程如下图所示：

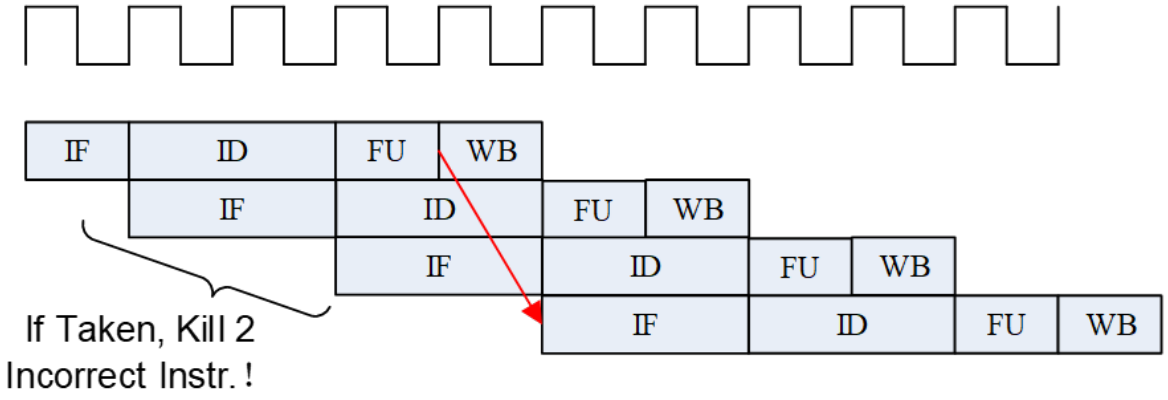

图 3.3: Control Hazard

根据上述设计思想，Ctrl Unit 设计如下：

```
1  `timescale 1ns / 1ps
2
3  module CtrlUnit(
4          input clk,
5          input rst,
6
7          input[31:0] inst,
8          input valid_ID,
9
10         input ALU_done,
11         input MEM_done,
12         input MUL_done,
13         input DIV_done,
14         input JUMP_done,
```

```
15          input cmp_res_FU,
16
17          // IF
18          output reg_IF_en, branch_ctrl,
19
20          // ID
21          output reg_ID_en, reg_ID_flush,
22          output[2:0] ImmSel,
23          output ALU_en, MEM_en, MUL_en, DIV_en, JUMP_en,
24
25          // FU
26          output[3:0] JUMP_op,
27          output[3:0] ALU_op,
28          output ALUSrcA,
29          output ALUSrcB,
30          output MEM_we,
31
32          // WB
33          output reg[2:0] write_sel,
34          output reg[4:0] rd_ctrl,
35          output reg reg_write
36  );
37                  // instruction field
38          wire[6:0] funct7 = inst[31:25];
39          wire[2:0] funct3 = inst[14:12];
40          wire[6:0] opcode = inst[6:0];
41          wire[4:0] rd = inst[11:7];
42          wire[4:0] rs1 = inst[19:15];
43          wire[4:0] rs2 = inst[24:20];
44
45          // type specification
46          wire Rop = opcode == 7'b0110011;
47          wire Iop = opcode == 7'b0010011;
48          wire Bop = opcode == 7'b1100011;
49          wire Lop = opcode == 7'b0000011;
50          wire Sop = opcode == 7'b0100011;
51
52          wire funct7_0  = funct7 == 7'h0;
53          wire funct7_1  = funct7 == 7'h1;
54          wire funct7_32 = funct7 == 7'h20;
55
56          wire funct3_0 = funct3 == 3'h0;
```

```
57          wire funct3_1 = funct3 == 3'h1;
58          wire funct3_2 = funct3 == 3'h2;
59          wire funct3_3 = funct3 == 3'h3;
60          wire funct3_4 = funct3 == 3'h4;
61          wire funct3_5 = funct3 == 3'h5;
62          wire funct3_6 = funct3 == 3'h6;
63          wire funct3_7 = funct3 == 3'h7;
64
65          wire ADD  = Rop & funct3_0 & funct7_0;
66          wire SUB  = Rop & funct3_0 & funct7_32;
67          wire SLL  = Rop & funct3_1 & funct7_0;
68          wire SLT  = Rop & funct3_2 & funct7_0;
69          wire SLTU = Rop & funct3_3 & funct7_0;
70          wire XOR  = Rop & funct3_4 & funct7_0;
71          wire SRL  = Rop & funct3_5 & funct7_0;
72          wire SRA  = Rop & funct3_5 & funct7_32;
73          wire OR   = Rop & funct3_6 & funct7_0;
74          wire AND  = Rop & funct3_7 & funct7_0;
75
76          wire MUL    = Rop & funct3_0 & funct7_1;
77          wire MULH   = Rop & funct3_1 & funct7_1;
78          wire MULHSU = Rop & funct3_2 & funct7_1;
79          wire MULHU  = Rop & funct3_3 & funct7_1;
80          wire DIV    = Rop & funct3_4 & funct7_1;
81          wire DIVU   = Rop & funct3_5 & funct7_1;
82          wire REM    = Rop & funct3_6 & funct7_1;
83          wire REMU    = Rop & funct3_7 & funct7_1;
84
85          wire ADDI  = Iop & funct3_0;
86          wire SLTI  = Iop & funct3_2;
87          wire SLTIU = Iop & funct3_3;
88          wire XORI  = Iop & funct3_4;
89          wire ORI   = Iop & funct3_6;
90          wire ANDI  = Iop & funct3_7;
91          wire SLLI  = Iop & funct3_1 & funct7_0;
92          wire SRLI  = Iop & funct3_5 & funct7_0;
93          wire SRAI  = Iop & funct3_5 & funct7_32;
94
95          wire BEQ = Bop & funct3_0;
96          wire BNE = Bop & funct3_1;
97          wire BLT = Bop & funct3_4;
98          wire BGE = Bop & funct3_5;
```

```
99          wire BLTU = Bop & funct3_6;
100         wire BGEU = Bop & funct3_7;
101
102         wire LB =  Lop & funct3_0;
103         wire LH =  Lop & funct3_1;
104         wire LW =  Lop & funct3_2;
105         wire LBU = Lop & funct3_4;
106         wire LHU = Lop & funct3_5;
107
108         wire SB = Sop & funct3_0;
109         wire SH = Sop & funct3_1;
110         wire SW = Sop & funct3_2;
111
112         wire LUI   = opcode == 7'b0110111;
113         wire AUIPC = opcode == 7'b0010111;
114
115         wire JAL  =  opcode == 7'b1101111;
116         wire JALR = (opcode == 7'b1100111) && funct3_0;
117
118         wire R_valid = AND | OR | ADD | XOR | SLL | SRL | SRA | SUB | SLT
                | SLTU
119                 | MUL | MULH | MULHSU | MULHU | DIV | DIVU | REM | REMU;
120         wire I_valid = ANDI | ORI | ADDI | XORI | SLLI | SRLI | SRAI |
               SLTI | SLTIU;
121         wire B_valid = BEQ | BNE | BLT | BGE | BLTU | BGEU;
122         wire L_valid = LW | LH | LB | LHU | LBU;
123         wire S_valid = SW | SH | SB;
124
125         wire use_ALU = AND | OR | ADD | XOR | SLL | SRL | SRA | SUB | SLT
                | SLTU
126                 | I_valid | LUI | AUIPC;
127         wire use_MEM = L_valid | S_valid;
128         wire use_MUL = MUL | MULH | MULHSU | MULHU;
129         wire use_DIV = DIV | DIVU | REM | REMU;
130         wire use_JUMP = B_valid | JAL | JALR;
131
132         wire[2:0] use_FU =  {3{use_ALU}}  & 3'd1 |
133                                                 {3{use_MEM}}  & 3'd2 |
134                                                 {3{use_MUL}}  & 3'd3 |
135                                                 {3{use_DIV}}  & 3'd4 |
136                                                 {3{use_JUMP}} & 3'd5 ;
137
```

```
138         reg FU_in_use, to_writeback, B_in_FU, J_in_FU;
139         initial begin
140                 FU_in_use = 0;
141                 to_writeback = 0;
142                 B_in_FU = 0;
143                 J_in_FU = 0;
144                 write_sel = 0;
145                 rd_ctrl = 0;
146                 reg_write = 0;
147         end
148
149         always @(posedge clk or posedge rst) begin
150                 if(rst) begin
151                         FU_in_use <= 0;
152                         reg_write <= 0;
153                         B_in_FU <= 0;
154                         J_in_FU <= 0;
155                 end
156                 else if(~FU_in_use) begin
157                         reg_write <= 0;
158                         if(valid_ID) begin
159                                 FU_in_use <= 1;
160                                 rd_ctrl <= inst[11:7];
161                                 write_sel <= use_FU;
162                                 to_writeback <= R_valid | I_valid | JAL |
                                        JALR | L_valid | LUI | AUIPC;
163                                 B_in_FU <= B_valid;
164                                 J_in_FU <= JAL | JALR;
165                         end
166                         else begin
167                                 B_in_FU <= 0;
168                                 J_in_FU <= 0;
169                         end
170                 end
171                 else if(ALU_done | MEM_done | MUL_done | DIV_done |
                        JUMP_done) begin
172                         FU_in_use <= 0;
173                         reg_write <= to_writeback;
174                         B_in_FU <= 0;
175                         J_in_FU <= 0;
176                 end
177                 else begin
```

```
178                         reg_write <= 0;
179                         B_in_FU <= 0;
180                         J_in_FU <= 0;
181                 end
182         end
183
184
185         assign reg_IF_en = ~FU_in_use | branch_ctrl;
186
187         assign reg_ID_en = reg_IF_en;
188
189         assign branch_ctrl = (B_in_FU & cmp_res_FU) |  J_in_FU;
190
191         assign reg_ID_flush = branch_ctrl;
192
193         localparam Imm_type_I = 3'b001;
194         localparam Imm_type_B = 3'b010;
195         localparam Imm_type_J = 3'b011;
196         localparam Imm_type_S = 3'b100;
197         localparam Imm_type_U = 3'b101;
198         assign ImmSel = {3{JALR | L_valid | I_valid}} & Imm_type_I |
199                                         {3{B_valid}}                    &
                                                Imm_type_B |
200                                         {3{JAL}}                        &
                                                Imm_type_J |
201                                         {3{S_valid}}                    &
                                                Imm_type_S |
202                                         {3{LUI | AUIPC}}                &
                                                Imm_type_U ;
203
204         assign ALU_en = reg_IF_en & use_ALU & valid_ID;
205         assign MEM_en = reg_IF_en & use_MEM & valid_ID;
206         assign MUL_en = reg_IF_en & use_MUL & valid_ID;
207         assign DIV_en = reg_IF_en & use_DIV & valid_ID;
208         assign JUMP_en = reg_IF_en & use_JUMP & valid_ID;
209
210         localparam JUMP_BEQ  = 4'b0_001;
211         localparam JUMP_BNE  = 4'b0_010;
212         localparam JUMP_BLT  = 4'b0_011;
213         localparam JUMP_BGE  = 4'b0_100;
214         localparam JUMP_BLTU = 4'b0_101;
215         localparam JUMP_BGEU = 4'b0_110;
```

```
216         localparam JUMP_JAL  = 4'b0_000;
217         localparam JUMP_JALR = 4'b1_000;
218         assign JUMP_op ={4{BEQ}}  & JUMP_BEQ  |
219                                     {4{BNE}}  & JUMP_BNE  |
220                                     {4{BLT}}  & JUMP_BLT  |
221                                     {4{BGE}}  & JUMP_BGE  |
222                                     {4{BLTU}} & JUMP_BLTU |
223                                     {4{BGEU}} & JUMP_BGEU |
224                                     {4{JAL}}  & JUMP_JAL  |
225                                     {4{JALR}} & JUMP_JALR ;
226
227         localparam ALU_ADD  = 4'b0001;
228         localparam ALU_SUB  = 4'b0010;
229         localparam ALU_AND  = 4'b0011;
230         localparam ALU_OR   = 4'b0100;
231         localparam ALU_XOR  = 4'b0101;
232         localparam ALU_SLL  = 4'b0110;
233         localparam ALU_SRL  = 4'b0111;
234         localparam ALU_SLT  = 4'b1000;
235         localparam ALU_SLTU = 4'b1001;
236         localparam ALU_SRA  = 4'b1010;
237         localparam ALU_Ap4  = 4'b1011;
238         localparam ALU_Bout = 4'b1100;
239         assign ALU_op = {4{ADD | ADDI | AUIPC}} & ALU_ADD  |
240                                     {4{SUB}}                & ALU_SUB
                                                |
241                                     {4{AND | ANDI}}         & ALU_AND
                                                |
242                                     {4{OR | ORI}}           & ALU_OR
                                                |
243                                     {4{XOR | XORI}}         & ALU_XOR
                                                |
244                                     {4{SLL | SLLI}}         & ALU_SLL
                                                |
245                                     {4{SRL | SRLI}}         & ALU_SRL
                                                |
246                                     {4{SLT | SLTI}}         & ALU_SLT
                                                |
247                                     {4{SLTU | SLTIU}}       & ALU_SLTU
                                                |
248                                     {4{SRA | SRAI}}         & ALU_SRA
                                                |
```

```
249                                                {4{LUI}}                & ALU_Bout
                                                        ;
250
251         assign ALUSrcA = AUIPC;
252
253         assign ALUSrcB = I_valid | LUI | AUIPC;
254
255         assign MEM_we = S_valid;
256 endmodule
```

3.4 FU 阶段

FU_ALU FU_ALU 执行的是 ALU 操作，由两个时钟周期完成，基本操作与 ALU 操作相同，但通过 state 和一系列 reg 寄存器来控制结束周期。具体设计如下：

```
1  `timescale 1ns / 1ps
2
3  module FU_ALU(
4          input clk, EN,
5          input[3:0] ALUControl,
6          input[31:0] ALUA, ALUB,
7          output[31:0] res,
8          output zero, overflow,
9          output finish
10 );
11
12         reg state;
13         assign finish = state == 1'b1;
14         initial begin
15                 state = 0;
16         end
17
18         reg[3:0] Control;
19         reg[31:0] A, B;
20
21         always@(posedge clk) begin
22                 if(EN & ~state) begin // state == 0
23                         A <= ALUA;
24                         B <= ALUB;
25                         Control <= ALUControl;
26                         //...to fill sth.in
27                         state <= 1;
```

```
28                  end
29                  else state <= 0;
30          end
31
32          localparam ALU_ADD  = 4'b0001;
33          localparam ALU_SUB  = 4'b0010;
34          localparam ALU_AND  = 4'b0011;
35          localparam ALU_OR   = 4'b0100;
36          localparam ALU_XOR  = 4'b0101;
37          localparam ALU_SLL  = 4'b0110;
38          localparam ALU_SRL  = 4'b0111;
39          localparam ALU_SLT  = 4'b1000;
40          localparam ALU_SLTU = 4'b1001;
41          localparam ALU_SRA  = 4'b1010;
42          localparam ALU_Ap4  = 4'b1011;
43          localparam ALU_Bout = 4'b1100;
44
45          wire[4:0] shamt = B[4:0];
46          wire[32:0] res_subu = {1'b0,A} - {1'b0,B};
47
48          wire[31:0] res_ADD  = A + B;
49          wire[31:0] res_SUB  = A - B;
50          wire[31:0] res_AND  = A & B;
51          wire[31:0] res_OR   = A | B;
52          wire[31:0] res_XOR  = A ^ B;
53          wire[31:0] res_SLL  = A << shamt;
54          wire[31:0] res_SRL  = A >> shamt;
55
56          wire add_of = A[31] & B[31] & ~res_ADD[31] | // neg + neg = pos
57                                       ~A[31] & ~B[31] & res_ADD[31]; //
                                            pos + pos = neg
58          wire sub_of = ~A[31] & B[31] & res_SUB[31] | // pos - neg = neg
59                                       A[31] & ~B[31] & ~res_SUB[31]; //
                                            neg - pos = pos
60
61          wire[31:0] res_SLT  = {31'b0, res_SUB[31] ^ sub_of};
62          wire[31:0] res_SLTU = {31'b0, res_subu[32]};
63          wire[31:0] res_SRA  = {{32{A[31]}}, A} >> shamt;
64          wire[31:0] res_Ap4  = A + 4;
65          wire[31:0] res_Bout = B;
66
67          wire ADD  = Control == ALU_ADD ;
```

```
68          wire SUB  = Control == ALU_SUB ;
69          wire AND  = Control == ALU_AND ;
70          wire OR   = Control == ALU_OR  ;
71          wire XOR  = Control == ALU_XOR ;
72          wire SLL  = Control == ALU_SLL ;
73          wire SRL  = Control == ALU_SRL ;
74          wire SLT  = Control == ALU_SLT ;
75          wire SLTU = Control == ALU_SLTU;
76          wire SRA  = Control == ALU_SRA ;
77          wire Ap4  = Control == ALU_Ap4 ;
78          wire Bout = Control == ALU_Bout;
79
80
81          assign zero = ~|res;
82
83          assign overflow = (Control == ALU_ADD && add_of) |
84                                          (Control == ALU_SUB && sub_of);
85
86          assign res = {32{ADD }} & res_ADD  |
87                                          {32{SUB }} & res_SUB  |
88                                          {32{AND }} & res_AND  |
89                                          {32{OR  }} & res_OR   |
90                                          {32{XOR }} & res_XOR  |
91                                          {32{SLL }} & res_SLL  |
92                                          {32{SRL }} & res_SRL  |
93                                          {32{SLT }} & res_SLT  |
94                                          {32{SLTU}} & res_SLTU |
95                                          {32{SRA }} & res_SRA  |
96                                          {32{Ap4 }} & res_Ap4  |
97                                          {32{Bout}} & res_Bout ;
98
99  endmodule
```

FU_mem FU_mem 模块负责内存访问，需要三个时钟周期完成，同样是通过 state 与 reg 寄存器实现内存读写时钟周期的控制。

```
1  `timescale 1ns / 1ps
2
3  module FU_mem(
4          input clk, EN, mem_w,
5          input[2:0] bhw,
6          input[31:0] rs1_data, rs2_data, imm,
```

```
7         output[31:0] mem_data,
8         output finish
9 );
10
11        reg[1:0] state;
12        assign finish = state[0] == 1'b1;
13        initial begin
14                state = 0;
15        end
16
17        reg mem_w_reg;
18        reg[2:0] bhw_reg;
19        reg[31:0] rs1_data_reg, rs2_data_reg, imm_reg;
20
21        //to fill sth.in
22        always @(posedge clk) begin
23                if(EN & ~state) begin
24                        mem_w_reg <= mem_w;
25                        bhw_reg <= bhw;
26                        rs1_data_reg <= rs1_data;
27                        rs2_data_reg <= rs2_data;
28                        imm_reg <= imm;
29                        state[1] <= 1'b1;
30                end
31                else if(~finish) begin
32                        state = state >> 1;
33                end
34                else begin
35                        state <= 0;
36                end
37        end
38
39        RAM_B ram(.clka(clk),.addra(rs1_data_reg + imm_reg),.dina(
              rs2_data_reg),.wea(mem_w_reg),
40                .douta(mem_data),.mem_u_b_h_w(bhw_reg));
41
42 endmodule
```

FU_div FU_div 复杂除法模块，其中除法器由 IP 核实现，值得注意的是除法器的完成信号是通过 state 状态与除法器的 res_valid 信号共同控制。

```
1         `timescale 1ns / 1ps
```

```
2
3  module FU_div(
4      input clk, EN,
5      input[31:0] A, B,
6      output[31:0] res,
7      output finish
8  );
9
10     wire res_valid;
11     wire[63:0] divres;
12
13     reg state;
14     assign finish = res_valid & state;
15     initial begin
16         state = 0;
17     end
18
19     reg A_valid, B_valid;
20     reg[31:0] A_reg, B_reg;
21
22
23     always @(posedge clk) begin
24         if(EN & ~state) begin
25             A_valid = ~(A == 0);
26             B_valid = ~(B == 0);
27             A_reg = A;
28             B_reg = B;
29             state <= 1;
30         end
31         else if(res_valid)
32             state <= 0;
33         else state <= state;
34     end
35     //...             //to fill sth.in
36
37
38     divider div(.aclk(clk),
39         .s_axis_dividend_tvalid(A_valid),
40         .s_axis_dividend_tdata(A_reg),
41         .s_axis_divisor_tvalid(B_valid),
42         .s_axis_divisor_tdata(B_reg),
43         .m_axis_dout_tvalid(res_valid),
```

```
        .m_axis_dout_tdata(divres)
    );

    assign res = divres[63:32];

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。

```
`timescale 1ns / 1ps

module FU_jump(
        input clk, EN, JALR,
        input[2:0] cmp_ctrl,
        input[31:0] rs1_data, rs2_data, imm, PC,
        output[31:0] PC_jump, PC_wb,
        output cmp_res, finish
);

        reg state;
        assign finish = state == 1'b1;
        initial begin
                state = 0;
        end

        reg JALR_reg;
        reg[2:0] cmp_ctrl_reg;
        reg[31:0] rs1_data_reg, rs2_data_reg, imm_reg, PC_reg;

        always @(posedge clk) begin
                if(EN & ~state) begin
                        JALR_reg <= JALR;
                        cmp_ctrl_reg <= cmp_ctrl;
                        rs1_data_reg <= rs1_data;
                        rs2_data_reg <= rs2_data;
                        imm_reg <= imm;
                        PC_reg <= PC;
                        state <= 1;
                end
                else state <= 0;
        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 :
                tmp_res;
42          assign PC_wb = PC_reg + 4;
43          assign PC_jump = JALR_reg ? rs1_data_reg + imm_reg : PC_reg +
                imm_reg;
44  endmodule
```

FU_mul FU_mul 阶段需要 7 个周期完成，通过 IP 核与 state 右移实现。

```
1           `timescale 1ns / 1ps
2
3   module FU_mul(
4       input clk, EN,
5       input[31:0] A, B,
6       output[31:0] res,
7       output finish
8   );
9
10      reg[6:0] state;
11      assign finish = state[0] == 1'b1;
12      initial begin
13          state = 0;
14      end
15
16      reg[31:0] A_reg, B_reg;
17      wire[63:0] mulres;
18
19      //to fill sth.in
20      always @(posedge clk) begin
21          if (EN & ~state) begin
22              A_reg <= A;
23              B_reg <= B;
24              state[6] <= 1'b1;
25          end
```

```
26          else if(~finish) begin
27              state <= state >> 1;
28          end
29          else state <= 0;
30      end
31      multiplier mul(.CLK(clk),.A(A_reg),.B(B_reg),.P(mulres));
32
33      assign res = mulres[31:0];
34
35  endmodule
```

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 或者点击上上的绿色二进制标志。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 bistream 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：

```
module core_sim;
        reg clk, rst;

        RV32core core(
                .debug_en(1'b0),
                .debug_step(1'b0),
                .debug_addr(7'b0),
                .debug_data(),
                .clk(clk),
                .rst(rst),
                .interrupter(1'b0)
        );

        initial begin
        clk = 0;
        rst = 1;
        #2 rst = 0;
        end
        always #1 clk = ~clk;

endmodule
```

测试使用的代码如下

```
| addi  x0, x0, 0  |
| lw  x2, 4(x0)    |
| lw  x4, 8(x0)    |
| add  x1, x2, x4  |
| addi  x3, x1, -1 |
| lw  x5, 12(x0)   |
| lw  x6, 16(x0)   |
| 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, 0xffff0 |
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 load 操作中，FU 阶段需要两个周期

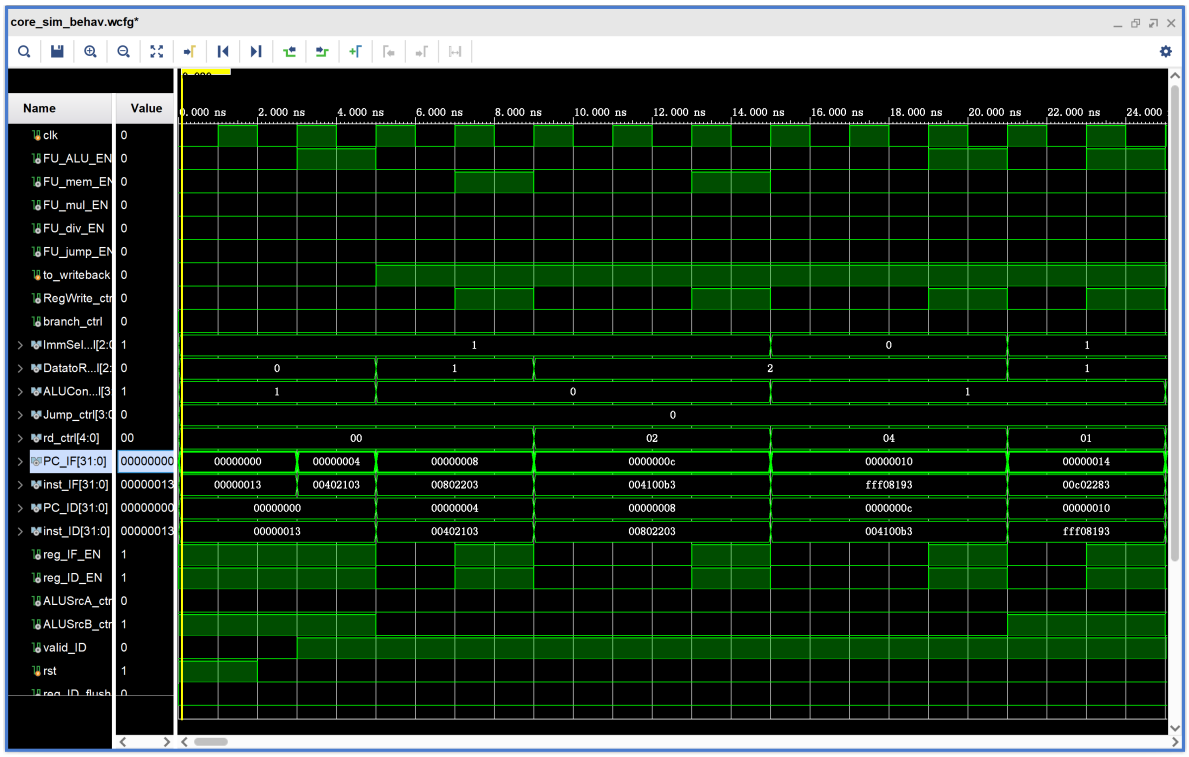

图 5.4: load

load 操作之后的 ALU 操作，由于需要用到之前的数据，发生了数据冒险。因此需要等待之前的指令完成之后才能执行。

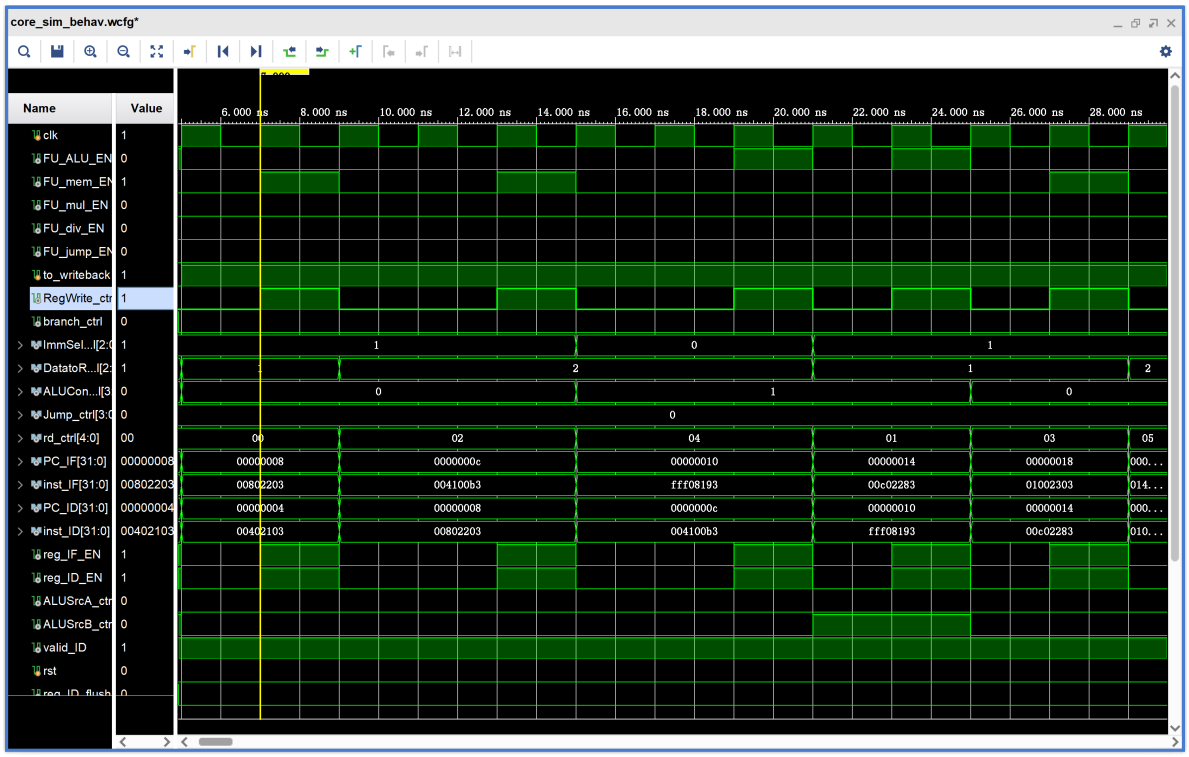

图 5.5: ALU

在遇到 branch 指令时，采用 predict not taken 的方式。首先让程序继续执行，若发现需要进行跳转，则杀死 branch 指令之后的指令，并跳转至下一条指令的位置。

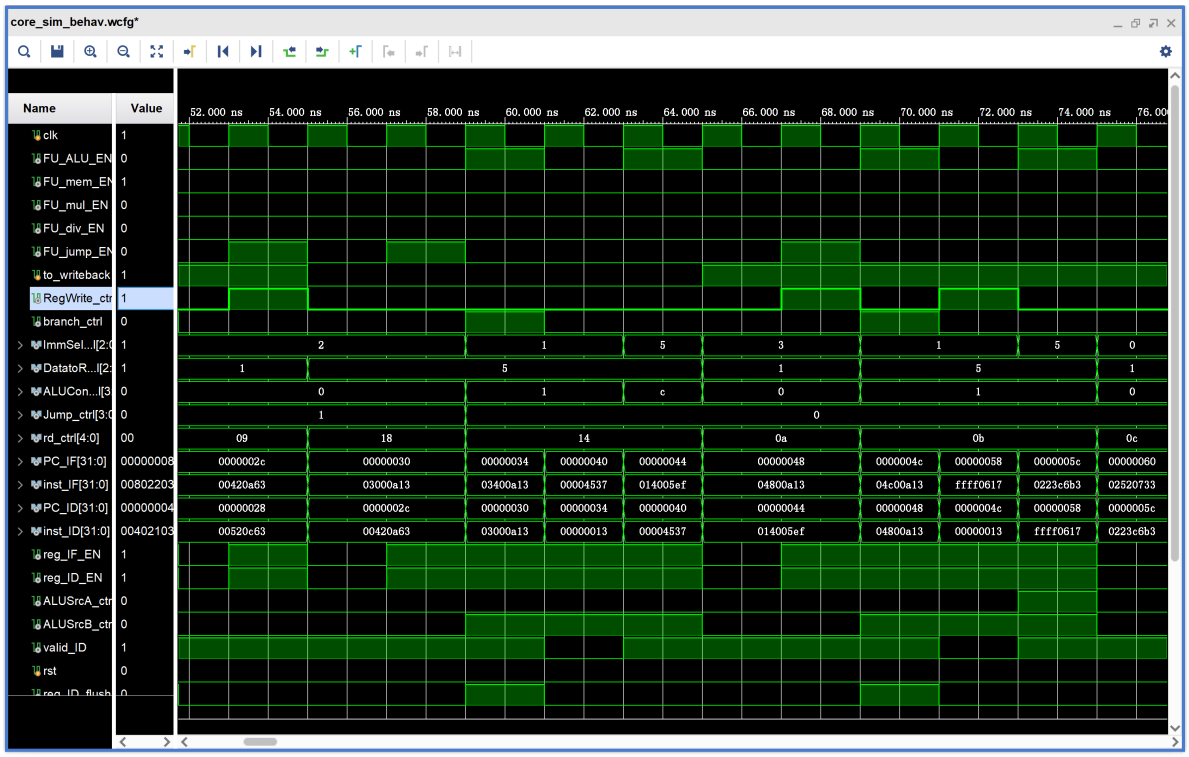

图 5.6: branch

在遇到 jal 指令时，采取同样的操作

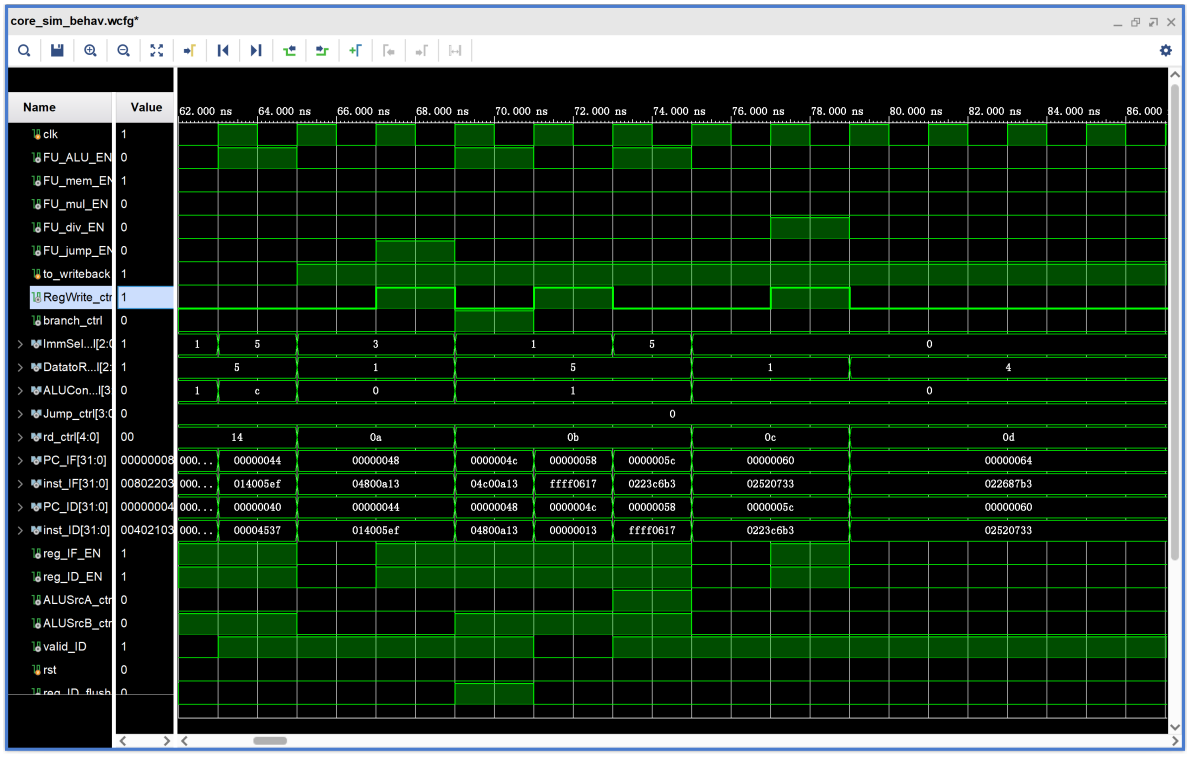

图 5.7: jal

之后进入除法指令，一次除法需要很长的时间，具体实现由 IP 核提供

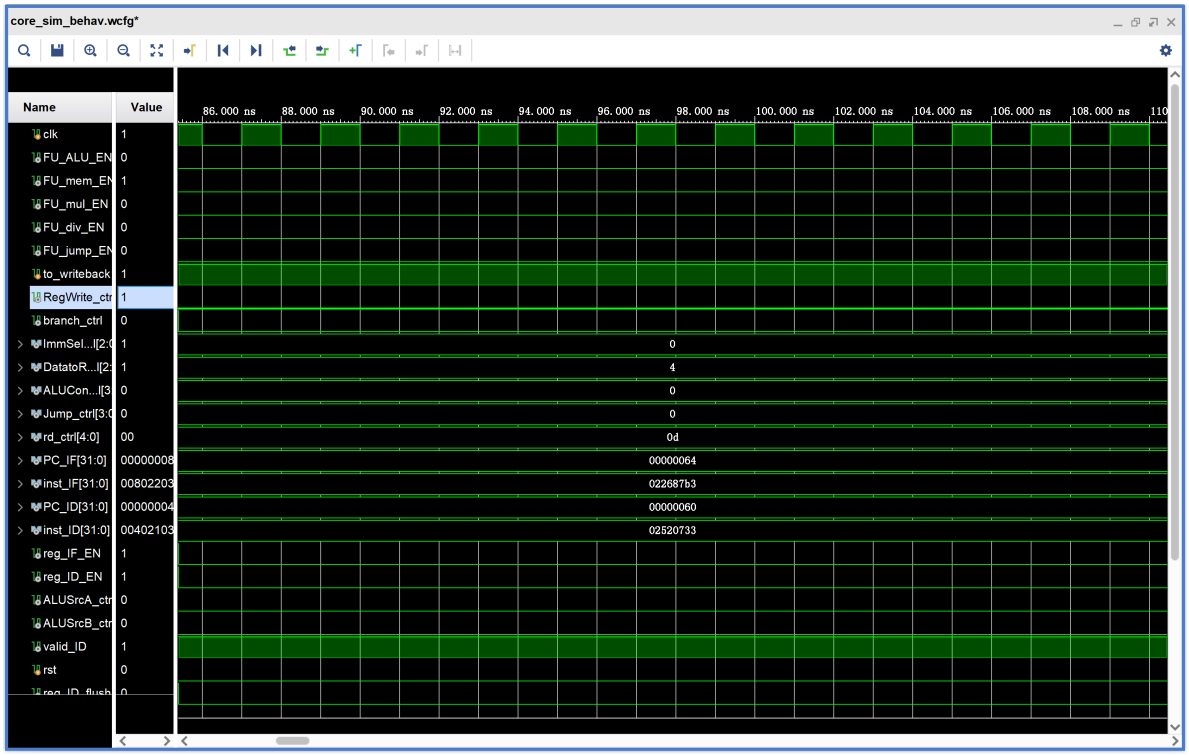

图 5.8: div

后面是两条乘法指令，每条乘法指令的 FU 阶段需要 7 个周期

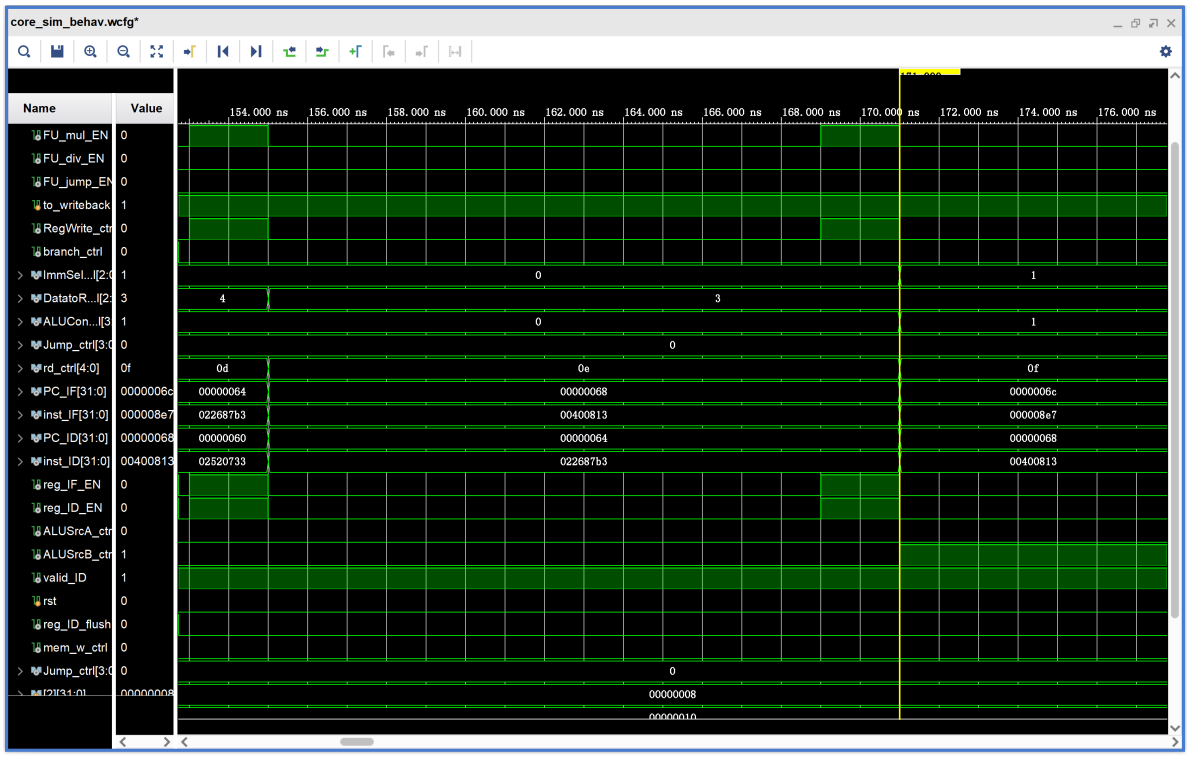

图 5.9: mul1

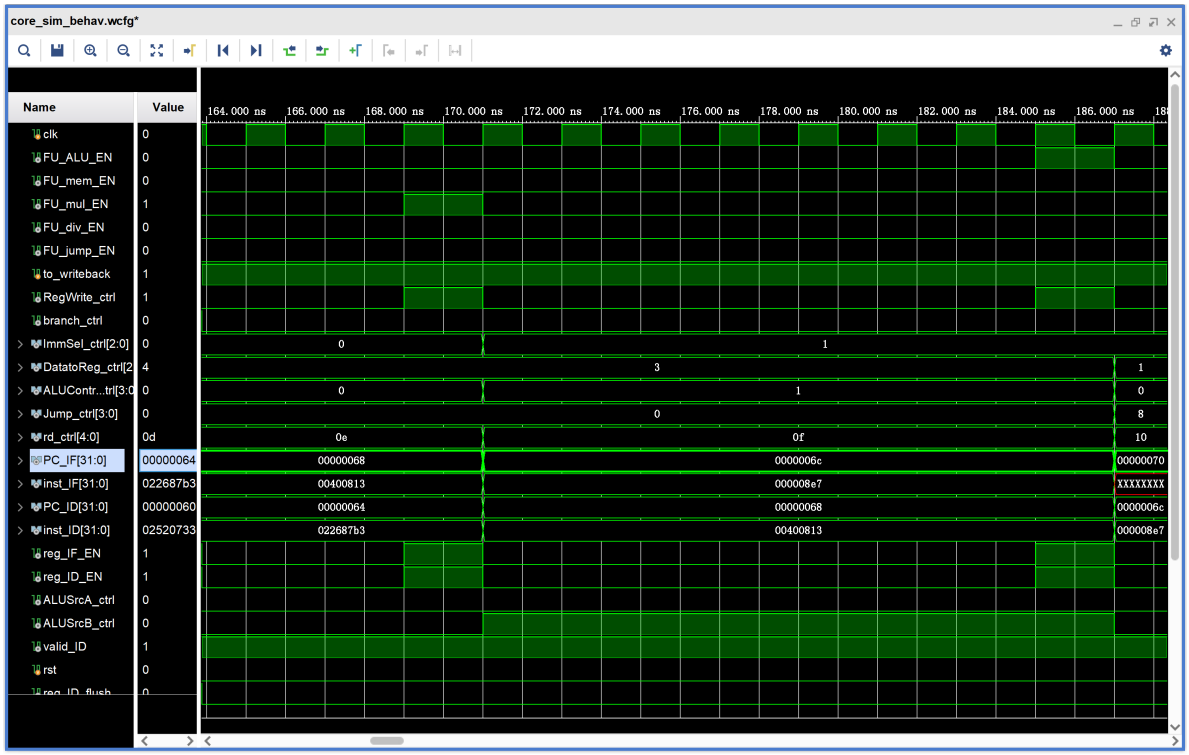

图 5.10: mul2

最后，jalr 指令使 PC 回到程序开头，进行下一次运行。由于在跳转回去之前，CPU

会读取 jalr 之后的指令，但 jalr 之后没有新的指令，因此读取会出现错误，会出现大片红色。

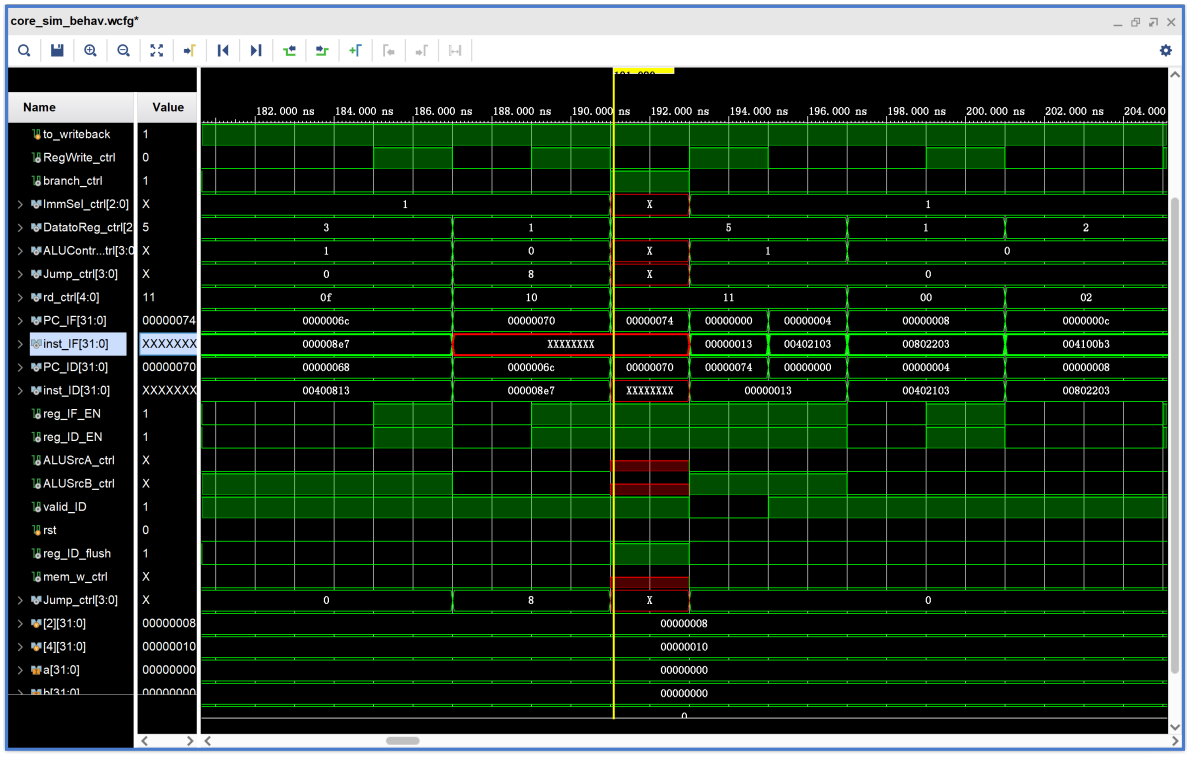

图 5.11: jalr

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。

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，我们在流水线中加入了浮点数模块。由于每个指令所需要的周期数不同，我们添加了新的模块来控制程序的停止与运行，以实现每种指令运行不同的时间。另外，在这次实验中，我们不再采用的原来的五个模块，而是将 EXE 与 MEM 模块合并成了 FU 模块，这样可以方便浮点数指令进行操作，并减少每个周期的时间。这让我认识到，CPU 流水线的结构是可以根据我们的需要进行调整的。