浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：刘思锐

学　　院：计算机科学与技术学院

系：

专　　业：计算机科学与技术

学　　号：3200102708

指导教师：陈文智

2022 年　10 月　10 日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：Pipelined CPU supporting RV32I instructions

学生姓名：刘思锐 专业：计算机科学与技术 学号：3200102708

同组学生姓名：陈镛屹 指导老师：陈文智

实验地点：曹西 301 实验日期：2022 年 10 月 10 日

一、 实验目的和要求

(1) Understand RISC-V RV32I instructions

(2) Master the design methods of pipelined CPU executing RV32I instructions

(3) Master the method of Pipeline Forwarding Detection and bypass unit design

(4) Master the methods of 1-cycle stall of Predict-not-taken branch design

(5) master methods of program verification of Pipelined CPU executing RV32I instructions

二、 实验内容和原理

(1) 实验内容

i. Design of Pipelined CPU executing RV32I instructions.

1. Design datapath
2. Design Bypass Unit
3. Design CPU Controller

ii. Verify the Pipelined CPU with program and observe the execution of program

(2) 实验原理

i. 标准 5 级流水线中的 forward 情况有以下 4 种

1. 前一指令 EXE 的结果立即 forward 到下一条指令的 EXE 操作数。
2. 前一指令 EXE 的结果在 MEM 级才 forward 到下两条指令的 EXE

操作数。

3. 前一指令从 MEM 读出的结果 forward 到下一条指令的 MEM 写数据。
4. 前一指令从 MEM 读出的结果 forward 到下两条指令的 EXE 操作数。

ii. 标准 5 级流水线中必须 stall 的情况是：

1. 前一指令是 load，后一指令在 EXE 级需要使用前一指令读出的数据。
2. 无条件跳转指令（包括 JAL、JALR）

iii. 遇到分支指令时可行的做法有：

1. Stall 一个时钟等待分支条件判断
2. Predict taken 默认跳转条件成立，如果发现条件不成立则把已经读入的指令 flush
3. Predict not-taken 默认跳转条件不成立，如果发现条件成立则把已经读入的指令 flush

三、 实验过程和数据记录

(1) 补全 cmp_32.v 文件

i. cmp_32.v 是一个多功能的比较单元，关键是要先输入比较的项目，才能返回相应结果的 bool 值。

```
wire EQ  = ctrl == cmp_EQ ;
wire NE  = ctrl == cmp_NE ;
wire LT  = ctrl == cmp_LT ;
wire LTU = ctrl == cmp_LTU;
wire GE  = ctrl == cmp_GE ;
wire GEU = ctrl == cmp_GEU;

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

(2) 补全 CtrlUnit.v 文件

i. 补全解码部分，根据指令中的 opcode 与 func3 确定指令的类型。（图中仅展示部分代码）

```
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
```

ii. 连接 CtrlUnit 和 cmp_32

```
assign cmp_ctrl = BEQ? cmp_EQ:
                  BNE? cmp_NE:
                  BLT? cmp_LT:
                  BLTU? cmp_LTU:
                  BGE? cmp_GE:
                  cmp_GEU;                                  //to fill sth. in
```

iii. 根据 DataPath 图补全控制信号

```
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
```

iv. 分析当前指令类型并生成相应信号，用于竞争的判断。

```
localparam hazard_optype_ALU = 2'b01;
localparam hazard_optype_LOAD = 2'b10;
localparam hazard_optype_STORE = 2'b11;
//to fill sth. in
assign hazard_optype = {2{R_valid | I_valid | JAL | JALR | LUI | AUIPC}}
                                                    & hazard_optype_ALU  |
                       {2{L_valid}} & hazard_optype_LOAD |
                       {2{S_valid}} & hazard_optype_STORE;
```

(3) 补全 HazardDetectionUnit.v 文件

i. 根据指令类型和寄存器使用情况判断当前是否需要 forward 和 stall。

```
//rs1的forward与stall的情况分类
wire rs1_stall     = rs1use_ID && rd_EXE && rs1_ID == rd_EXE && hazard_optype_EXE == hazard_optype_LOAD
                   && hazard_optype_ID != hazard_optype_STORE;    //上一条指令是load，下一条指令的EXE级要使用load的结果，无法forward，只能stall
wire rs1_forward_1 = rs1use_ID && rd_EXE && rs1_ID == rd_EXE && hazard_optype_EXE == hazard_optype_ALU;     //ALU运算，EXE级的结果forward
wire rs1_forward_2 = rs1use_ID && rd_MEM && rs1_ID == rd_MEM && hazard_optype_MEM == hazard_optype_ALU;     //ALU运算，MEM级的结果forward
wire rs1_forward_3 = rs1use_ID && rd_MEM && rs1_ID == rd_MEM && hazard_optype_MEM == hazard_optype_LOAD;    //LOAD数据，MEM级的结果forward
//rs2的forward与stall的情况分类
wire rs2_stall     = rs2use_ID && rd_EXE && rs2_ID == rd_EXE && hazard_optype_EXE == hazard_optype_LOAD
                   && hazard_optype_ID != hazard_optype_STORE;    //上一条指令是load，下一条指令的EXE级要使用load的结果，无法forward，只能stall
wire rs2_forward_1 = rs2use_ID && rd_EXE && rs2_ID == rd_EXE && hazard_optype_EXE == hazard_optype_ALU;     //ALU运算，EXE级的结果forward
wire rs2_forward_2 = rs2use_ID && rd_MEM && rs2_ID == rd_MEM && hazard_optype_MEM == hazard_optype_ALU;     //ALU运算，MEM级的结果forward
wire rs2_forward_3 = rs2use_ID && rd_MEM && rs2_ID == rd_MEM && hazard_optype_MEM == hazard_optype_LOAD;    //LOAD数据，MEM级的结果forward

assign forward_ctrl_ls = rs2_EXE == rd_MEM && hazard_optype_EXE == hazard_optype_STORE
                       && hazard_optype_MEM == hazard_optype_LOAD;    // load出来的数据立刻store回去，forward到mem
```

ii. 根据 forward 情况生成 mux 的控制信号

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

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

iii. 根据分支跳转情况和 stall 情况判断各级流水线之间的寄存器是否需要 flush、PC 应该如何取值。**这里对跳转指令使用 predict not-taken 策略。**

```
assign PC_EN_IF = ~load_stall;      //stall时PC停一个时钟，不取新的指令

assign reg_FD_stall = load_stall;   //stall则保持存的指令和PC的值不变
assign reg_DE_flush = load_stall;   //DE寄存器在stall时没有值，指令变成addi x0, x0, 0

assign reg_FD_flush = Branch_ID;    //前一时钟取到了错误的指令，flush
```

(4) 补全 RV32core.v 文件，主要是各路 mux 的控制信号与实际接线相对应。

i. 跳转相关的 mux

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

ii. Forward 相关的 mux

```
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));

MUX2T1_32 mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout_EXE));       //to fill sth. in ()
```

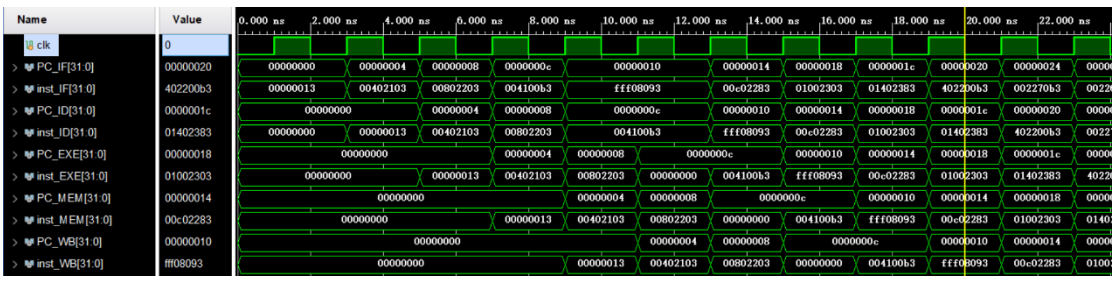
iii. EXE 相关的 mux

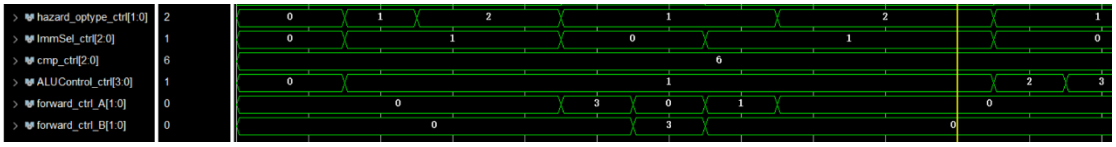
```
MUX2T1_32 mux_A_EXE(.I0(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));     //to fill sth. in ()

MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));       //to fill sth. in ()
```

(5) 进行仿真实验

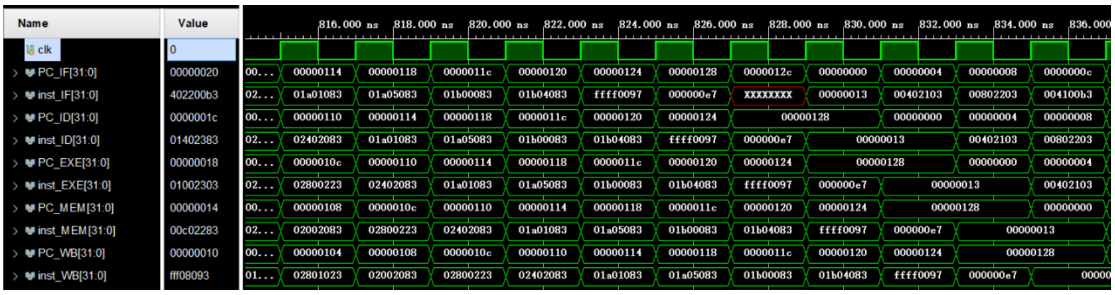
四、 实验结果分析

(1) 第一种 stall 情况，load 后立刻取做操作数，同时也有第一、第三种 forward 情况。



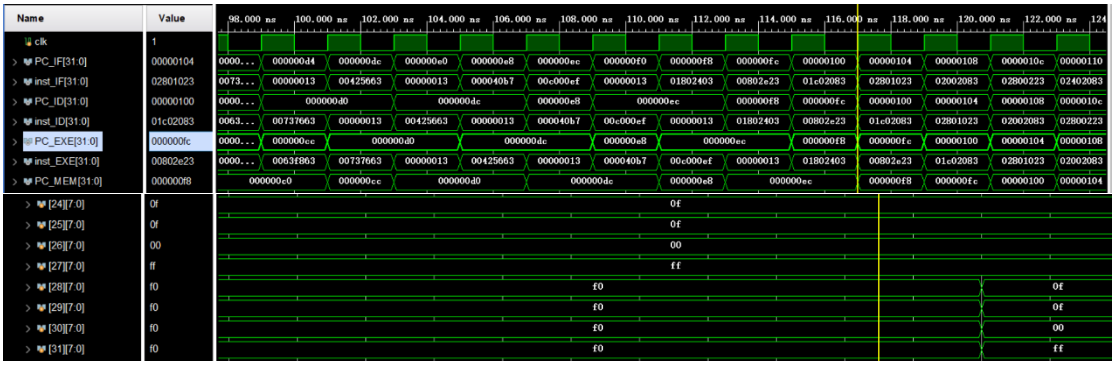


(2) 第二种 stall 情况，JAL 指令



(3) 第二种 forward 情况（ALU 运算的结果从 MEM 级 forward 给下两条指令的 EXE 级）在 rom 里似乎没有出现。

(4) 第四种 forward 情况，Load 的结果立即 store。



五、 讨论与心得

第一次实验的内容并没有超出上个学期学过的计算机组成，但是复杂的连线和 HazardUnit 中多种多样的条件判断对细心和耐心的要求很高，加之 Vivado 重新生成一次 bit 文件要花将近十分钟，为了完成这次实验还是在实验室蹲了相当长时间。