中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929010 姓名：贾城昊 专业：计算机科学与技术

实验序号：5.3 实验名称：处理器性能增强设计

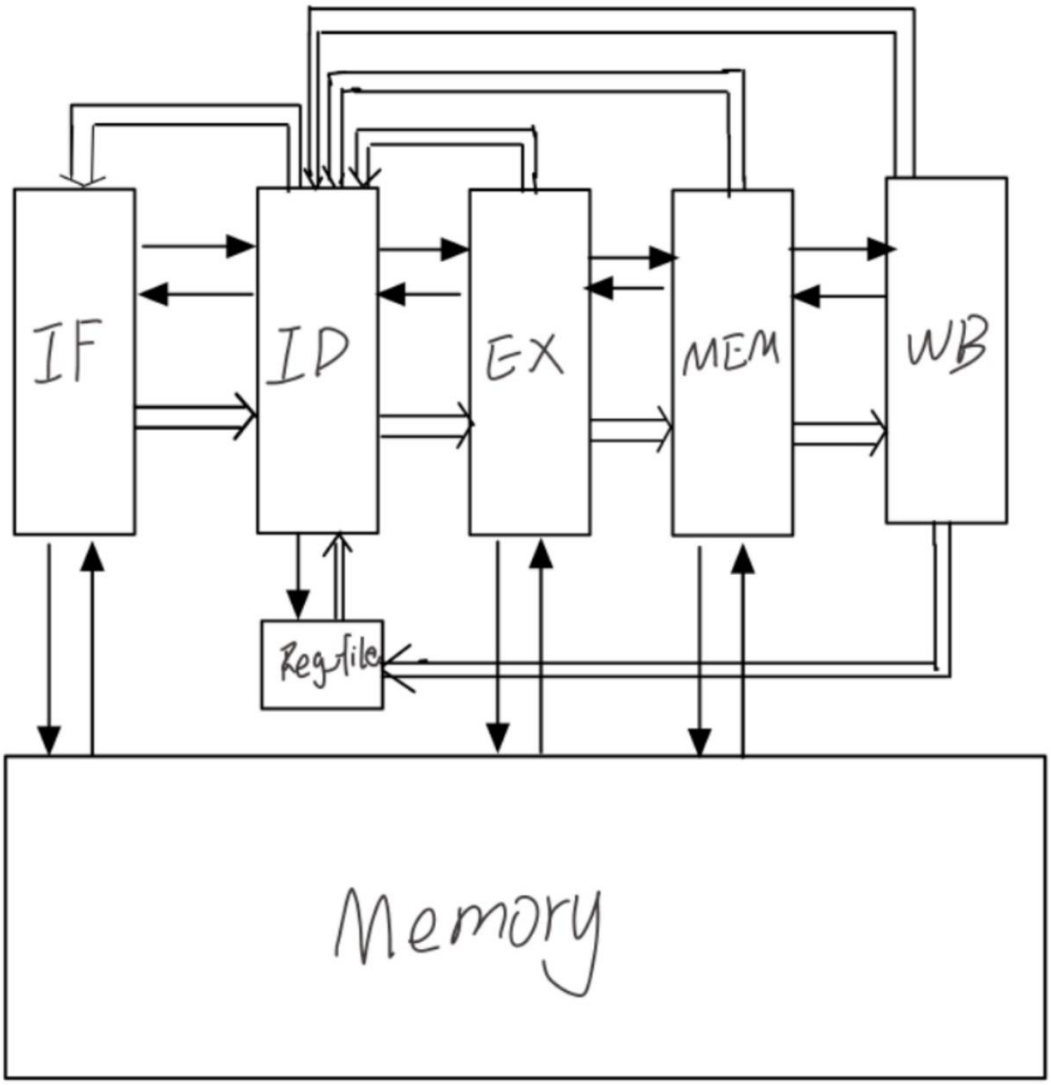
注 1：撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 /home/serve-ide/cod-lab/reports 目录下（注意：reports 全部小写）。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图{自行画图，推荐用 PPT 画逻辑结构框图，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

本人复杂处理器设计实验使用 RSICV 指令集,使用流水线架构为五级流水线。复杂处理器设计实验中一共使用了 9 个硬件设计文件: custom_cpu.v, IF_State.v, ID_State.v, EX_State.v, MEM_State.v, WB_stage.v, reg_file.v, alu.v, shifter.v。其中 reg_file.v, alu.v, shifter.v 直接复用 pri4 中的对应文件;其余文件为本次实验中编写。以下为复杂处理器设计实验中设计的 RSICV 五级流水线处理器整体逻辑电路结构图：



1） 流水线的控制信号

I. 各流水线模块之间:

后一级流水向前一级发送 Y_Allow_in 信号（X 和 Y 代表模块名）：高电平表示后一级流水可以正常接收数据;低电平表示后一级流水处于工作状态或阻塞状态。前一级流水向后一级发送 X_to_Y_valid 信号 :高电平表示前一级流水正常工作,输出数据正确;低电平表示前一级流水处于工作状态或阻塞状态。前一级流水向后一

级发送 X_to_Y_Bus ：当 X_to_Y_valid 为高电平时 bus 中传递数据有效。

II. 各流水线模块以内

各流水线模块内部由 X_Allow_in，X_Ready 信号以及 X_Valid 寄存器控制 ：X_Allow_in 表示该级流水能否正常读入数据,高电平时修改 X_Valid 寄存器。X_Allow_in 信号由后一级流水的 Y_Allow_in 信号以及本级流水的 X_Ready 控制。X_Ready 表示该级流水工作是否完成完成。X_Ready 信号由该级流水所需动作具体决定。通过 X_Ready 信号来实现阻塞控制。X_Valid 表示该级流水线工作是否有效。X_Valid 寄存器由 X_Allow_in 信号以及上一级流水传入的 Z_to_X_Valid 信号控制与决定。当 X_Valid 为高电平时，更新流水线中寄存器。X_Valid 寄存器与 X_Ready 信号共同控制 X_to_Y_Valid 信号。

下面是各模块握手信号具体代码：

IF:

```
//IF_Valid
always @(posedge clk) begin
        if(rst) begin
                IF_Valid <= 1'b0;
        end
        else if(IF_Allow_in) begin
                IF_Valid <= to_IF_Valid & Inst_Req_Ready;
        end
end
```

```
//HandShake Signals (IF <-> ID)
assign IF_Ready    = (current_state == DONE); //访存请求成功且访存成功
assign IF_Allow_in     = !IF_Valid || IF_Ready && ID_Allow_in;  //IF和ID握手成功
assign IF_to_ID_Valid = IF_Valid & IF_Ready; //非跳转分支指令且IF工作正常
```

ID:

```
//ID_Valid
always @(posedge  clk) begin
        if(rst) begin
                ID_Valid <= 1'b0;
        end
        else if(ID_Allow_in) begin
                ID_Valid <= IF_to_ID_Valid;
        end
end
```

```
//HandShake Signals
assign ID_Ready   = ~((B_Type & (~rs_from_RegFile | ~rt_from_RegFile)) |       //Branch需要得到正确的数据
                      (~rdw_EX_addr_valid & (rs_from_EX | rt_from_EX)) |     //与EX冲突
                      (~rdw_MEM_addr_valid & (rs_from_MEM | rt_from_MEM)) |  //与MEM冲突
                      (~rdw_WB_addr_valid & (rs_from_WB | rt_from_WB))       //与WB冲突
                      );

assign ID_Allow_in     = !ID_Valid || ID_Ready && EX_Allow_in; //ID与EX握手成功
assign ID_to_EX_Valid = ID_Valid & ID_Ready; //ID阶段工作正常
```

EX:

```
//EX_Valid
always @(posedge clk) begin
        if(rst) begin
                EX_Valid <= 1'b0;
        end
        else if(EX_Allow_in) begin
                EX_Valid <= ID_to_EX_Valid;
        end
end
```

```
//HandShake Signals between MEM and EX
assign EX_Ready     = MEM_req_succ | ~LOAD & ~STORE;  //访存请求成功或者不是访存指令
assign EX_Allow_in      = !EX_Valid || EX_Ready && MEM_Allow_in; //EX和MEM握手成功
assign EX_to_MEM_Valid = EX_Valid & EX_Ready; //EX阶段工作正常
```

MEM:

```
//MEM_Valid
always @(posedge clk) begin
        if(rst) begin
                MEM_Valid <= 1'b0;
        end
        else if(MEM_Allow_in) begin
                MEM_Valid <= EX_to_MEM_Valid;
        end
end
```

```
//HandShake Signals
assign MEM_Ready        =  ~LOAD | Read_data_Valid; //非LOAD指令或访存成功
assign MEM_Allow_in     =  !MEM_Valid || MEM_Ready && WB_Allow_in; //MEM和WB握手成功
assign MEM_to_WB_Valid  =  MEM_Valid & MEM_Ready; //MEM工作正常
```

WB:

```
//WB_Valid
always @(posedge clk) begin
        if(rst) begin
                WB_Valid <= 1'b0;
        end
        else if(WB_Allow_in)begin
                WB_Valid <= MEM_to_WB_Valid;
        end
end
```

```
//HandShake Signals
assign WB_Ready = 1'b1;
assign WB_Allow_in  = !WB_Valid || WB_Ready;
```

2） 流水线的数据传递

通过X_to_Y_Bus和X_to_Y_Bus_reg完成，当握手成功和，更新寄存器中的值，下面以WB流水级为例：

```
//MEM_to_WB_Bus_reg
always @(posedge clk) begin
        if(MEM_to_WB_Valid & WB_Allow_in) begin
                MEM_to_WB_Bus_reg <= MEM_to_WB_Bus;
        end
end
```

3） RAW（旁路）设计：

由于在 ID 阶段访问寄存器时,对应寄存器中内容可能已经被 EX, MEM,WB 三个阶段中改变,故需要三条从 EX, MEM, WB 到 ID 的旁路来保证 ID 阶段读取寄存器中数据的正确性。而判断有无冲突只用对比传过去的写寄存器地址和当前的读寄存器地址是否相同即可。特别的,对于 LOAD 指令,由于其在 MEM 阶段才能得到正确数据,并且内存访问存在延时,故需要添加 rdw_xx_addr_valid 以及 rdw_xx_data_valid 信号来确定旁路传回的寄存器地址以及数据的有效性。

在 ID 阶段,寄存器中正确内容由旁路和当时寄存器内容确定。其间优先级关系：EX (by_path) >MEM (by_path) >WB (by_path) >ID (reg_file)特别的，对于 load 指令，由于其在 MEM 阶段才能得到正确数据，并且内存访问存在延时，故需要在其未得到正确数据时将 ID 以及 IF 阻塞，直到 LOAD 取到正确数据。

以下为相关的代码：

```
//DataPath (EX --> ID)
assign rdw_EX_Bus  = {  ~LOAD,                  //38:38
                        WB_wen & EX_Valid,      //37:37
                        RF_waddr,               //36:32
                        Result                  //31:0
                        };
```

```
//DataPath (MEM --> ID)
assign rdw_MEM_Bus =  { MEM_Ready,              //38:38
                        WB_wen & MEM_Valid,     //37:37
                        RF_waddr,               //36:32
                        final_result            //31:0
                        };
```

```
//DataPath (WB --> ID)
assign rdw_WB_Bus =   { 1'b1,           //38:38
                        RF_wen,         //37:37
                        RF_waddr,       //36:2
                        final_result    //31:0
                        };
```

ID 流水级：

寄存器数据选择：

```
//Reg_File_Data Selected Signals
assign rs_from_EX  = (|rs) & rdw_EX_data_valid & (rdw_EX_addr == rs);
assign rs_from_MEM = (|rs) & ~rs_from_EX & rdw_MEM_data_valid & (rdw_MEM_addr == rs);
assign rs_from_WB  = (|rs) & ~rs_from_EX & ~rs_from_MEM & rdw_WB_data_valid & (rdw_WB_addr == rs);
assign rs_from_RegFile  = ~(rs_from_EX |rs_from_MEM | rs_from_WB);

assign rt_from_EX  = (|rt) & rdw_EX_data_valid & (rdw_EX_addr == rt);
assign rt_from_MEM = (|rt) & ~rt_from_EX & rdw_MEM_data_valid & (rdw_MEM_addr == rt);
assign rt_from_WB  = (|rt) & ~rt_from_EX & ~rt_from_MEM & rdw_WB_data_valid & (rdw_WB_addr == rt);
assign rt_from_RegFile  = ~(rt_from_EX |rt_from_MEM | rt_from_WB);
```

```
//Reg_File_Data Selected
assign RF_Final_data1  = ({32{rs_from_EX}} & rdw_EX_data) |
                         ({32{rs_from_MEM}} & rdw_MEM_data) |
                         ({32{rs_from_WB}} & rdw_WB_data) |
                         ({32{rs_from_RegFile}} & RF_rdata1);

assign RF_Final_data2  = ({32{rt_from_EX}} & rdw_EX_data) |
                         ({32{rt_from_MEM}} & rdw_MEM_data) |
                         ({32{rt_from_WB}} & rdw_WB_data) |
                         ({32{rt_from_RegFile}} & RF_rdata2);
```

阻塞处理：

```
//HandShake Signals
assign ID_Ready    = ~((B_Type & (~rs_from_RegFile | ~rt_from_RegFile)) |       //Branch需要得到正确的数据
                       (~rdw_EX_addr_valid & (rs_from_EX | rt_from_EX)) |      //与EX冲突
                       (~rdw_MEM_addr_valid & (rs_from_MEM | rt_from_MEM)) |   //与MEM冲突
                       (~rdw_WB_addr_valid & (rs_from_WB | rt_from_WB))        //与WB冲突
                       );
```

4） Branch 和 Jump 指令处理：

branch 与 jump 相关处理在 ID 流水级完成，并将结果直接传回 IF 流水级。

branch:为在 ID 得到 branch 跳转结果,例化 CLA 模块，用于计算 branch 比较结果（实际上是一个并行加法器，它会返回结果符号位，overflow 信号， carryout 信号和结果为零信号）。

jump: jalr 需要通过计算来求得跳转的 PC 值，也可复用 CLA 实现。

相关代码如下：

```
//CLA例化 (for branch and jump)
assign CLA_A = RF_Final_data1;
assign CLA_B = ({32{B_Type}} & ~RF_Final_data2) |
               ({32{JALR}} & imm);

CLA CLA(
        .A(CLA_A),
        .B(CLA_B),
        .CIN(B_Type),
        .SF(CLA_SF),    //符号位
        .ZF(CLA_ZF),    //零标志位
        .OF(CLA_OF),    //Carryout标志位
        .CF(CLA_CF),    //Overflow标志位
        .S(CLA_result)
);
```

```
//Branch_or_not
assign Branch_or_not = B_Type & (((~|funct3) & CLA_ZF) |                    //BEQ
                                 ((funct3 == 3'b001) & ~CLA_ZF) |           //BNE
                                 ((funct3 == 3'b100) & (CLA_OF ^ CLA_SF)) | //BLT
                                 ((funct3 == 3'b101) & ~(CLA_OF ^ CLA_SF)) | //BGE
                                 ((funct3 == 3'b110) & CLA_CF) |            //BLTU
                                 ((&funct3) & ~CLA_CF)                      //BGEU
                                 );


//Branch_Address & Jump_Address
assign Branch_Address = PC + imm;
assign Jump           = J_Type | JALR;
assign Jump_Address   = {32{J_Type}} & (PC + imm) |
                        {32{JALR}} & (CLA_result & {~31'b0, 1'b0});

//Branch_or_Jump_Bus
assign Branch_or_Jump_wen        = (Branch_or_not | Jump) & ID_Valid;
assign Branch_or_Jump_PC         = ({32{Branch_or_not}} & Branch_Address) |
                                   ({32{Jump}} & Jump_Address) |
                                   ({32{~Jump & ~Branch_or_not}} & (PC + 4));

assign Branch_or_Jump_Bus        = {Branch_or_Jump_wen, Branch_or_Jump_PC};
```

CLA 设计：

```
module CLA(
        input [31:0] A,
        input [31:0] B,
        input CIN,
        output SF,          //符号位
        output ZF,          //零标志位
        output CF,          //Carryout标志位
        output OF,          //Overflow标志位
        output [31:0] S
);
        wire [32:0] cout;
        wire Cin;
        wire COUT;

        assign cout[0] = CIN;

        //并行加法器
        genvar i;
        generate
                for(i = 0; i < 32; i = i + 1)
                begin : CLA
                        wire p, g;
                        assign p = ~A[i] & B[i] | A[i] & ~B[i];
                        assign g = A[i] & B[i];
                        assign S[i] = ~p & cout[i] | p & ~cout[i];
                        assign cout[i + 1] = g | p & cout[i];
                end
        endgenerate

        assign COUT = cout[32];
        assign Cin = cout[31];

        //SF:符号位 ZF:零标志 CF:进位标准 OF:溢出标准
        assign SF = S[31];
        assign ZF = ~|S;
        assign CF = ~COUT;
        assign OF =  Cin ^ COUT;
endmodule
```

5)　PC 更新

PC 在 IF 流水级进行更新，如果 ID 传过来的信号表示是 Branch 或 Jump 指令，需要跳转或者分支，则更新为其对应的地址，否则 PC 更新为 PC+4。

具体的实现是通过设计一个状态机，如下：

```
always @ (posedge clk) begin
        if (rst)
                current_state <= INIT;
        else
                current_state <= next_state;
end


/* FSM 2 */
always @ (*) begin
        case (current_state)
                INIT:
                        next_state = IF;
                IF:
                        if (Inst_Req_Ready & Inst_Req_Valid) //握手成功
                                next_state = IW;
                        else
                                next_state = IF;
                IW:
                        if (Inst_Valid) begin
                                if (ID_Valid && Branch_or_Jump || Branch_or_Jump_reg)
                                        /* Branch will happen */
                                        next_state = IF;
                                else
                                        next_state = DONE;
                        end
                        else begin
                                next_state = IW;
                        end
                DONE:   begin
                                if(ID_Allow_in) begin
                                        next_state = IF;
                                end
                                else begin
                                        next_state = DONE;
                                end
                        end
                default:        next_state = INIT;
        endcase
end
```

当发现需要分支或跳转时，会从 IW 重新回到 IF 阶段，重新进行握手。

PC 具体的更新代码如下：

```
//PC更新
always @ (posedge clk) begin
        if (rst)
                PC <= 32'd0;
        else if (current_state == IW
                && (ID_Valid && Branch_or_Jump || Branch_or_Jump_reg))
                        PC <= PC_abnormal;
        else if (current_state == DONE) begin
                if (ID_Valid && Branch_or_Jump || Branch_or_Jump_reg)
                        PC <= PC_abnormal;  /* Branch */
                else if (next_state == IF) begin
                        PC <= PC + 32'd4;
                end
        end
end
```

如果时跳转分支指令，PC 在接受到信号后更新为对应的跳转分支的地址，如果不是，则在完成对 ID 流水级的数据传递后，更新为 PC+4

6） 访存信号：

IF 流水级完成跟内存的取指令握手，在 EX 阶段完成写内存握手和读内存请求握手，在 MEM 阶段完成写内存握手，具体握手信号逻辑如下：

```
//访存信号
assign Inst_Req_Valid = (current_state == IF) & ~MemRead; //行为仿真需要错开
assign Inst_Ready = (current_state == IW || current_state == INIT);
```

```
//MEM_req_succ(访存请求握手成功)
always @(posedge clk) begin
        if(~MEM_req_succ | EX_Allow_in) begin //握手未成功时需要一直判断
                MEM_req_succ <= MEM_Req_Ready & (MEMRead | MEMWrite);
        end
end
```

```
//Memory Request HandShake Signals
assign MEMRead    = LOAD & EX_Valid & ~MEM_req_succ;  //LOAD指令，且访存请求未成功，EX阶段有效
assign MEMWrite   = STORE & EX_Valid & ~MEM_req_succ; //STORE指令，且访存请求未成功，EX阶段有效
```

```
assign Read_data_Ready  =  LOAD & MEM_Valid & WB_Allow_in | rst; //WB允许进入，拉高Ready信号，获取数据
```

（注：上面 Inst_Req_Valid 信号原本应该与 MemRead 无关，但似乎是行为仿真框架的原因，当依次发生 Inst_Req_Valid， MemRead，Read_data_Ready 握手成功时， Inst_Valid 永远不会拉高，希望该问题早点得到解决）

7） 性能计数器

由于流水线不再具有统一的状态机，因此之前的计数器逻辑会发生一定的变化，且由于没有状态机，在本次实验中，本人只设计了六个性能计数器，分别计算周期数、指令数、读内存指令数、写内存指令数、跳转指令数、分支指令数，对应代码如下:

```
//周期计数器
reg  [31:0] cycle_cnt;
always @ (posedge clk) begin
        if (rst) begin
                cycle_cnt <= 32'd0;
        end else begin
                cycle_cnt <= cycle_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_0 = cycle_cnt;

//指令计数器
reg  [31:0] inst_cnt;
always @ (posedge clk) begin
        if (rst) begin
                inst_cnt <= 32'd0;
        end else if (Inst_Valid & Inst_Ready) begin
                inst_cnt <= inst_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_1 = inst_cnt;

//读内存计数器
reg  [31:0] mr_cnt;
always @ (posedge clk) begin
        if (rst) begin
                mr_cnt <= 32'd0;
        end else if (MemRead & Mem_Req_Ready) begin
                mr_cnt <= mr_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_2 = mr_cnt;
```

```
//写内存计数器
reg  [31:0] mw_cnt;
always @ (posedge clk) begin
        if (rst) begin
                mw_cnt <= 32'd0;
        end else if (MemWrite & Mem_Req_Ready) begin
                mw_cnt <= mw_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_3 = mw_cnt;

//分支指令计数器
reg  [31:0] branch_inst_cnt;
always @ (posedge clk) begin
        if (rst) begin
                branch_inst_cnt <= 32'd0;
        end else if (ID_to_EX_Valid && ID_to_EX_Bus[71]) begin
                branch_inst_cnt <= branch_inst_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_9 = branch_inst_cnt;

//跳转指令计数器
reg  [31:0] jump_inst_cnt;
always @ (posedge clk) begin
        if (rst) begin
                jump_inst_cnt <= 32'd0;
        end else if (ID_to_EX_Valid && (ID_to_EX_Bus[67] | ID_to_EX_Bus[66])) begin
                jump_inst_cnt <= jump_inst_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_10 = jump_inst_cnt;
```

```
//辅助周期计数器(帮助处理周期计数器溢出的问题)
reg  [31:0] sub_cycle_cnt;
always @ (posedge clk) begin
        if (rst) begin
                sub_cycle_cnt <= 32'd0;
        end else if (&cycle_cnt) begin //存周期数的高位
                sub_cycle_cnt <= sub_cycle_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_11 = sub_cycle_cnt;
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，逻辑仿真和 FPGA 调试过程中的难点等）

本次实验的硬件部分代码 debug 较为麻烦，因为流水线的握手较为

复杂，且还需要考虑与内存的握手，所以 debug 耗费了本人特别多的时间。对于本次实验中遇到的 bug，主要有以下几个：

遇到的重要 bug 以及解决过程：

1. PC 更新出错：

本人的最开始没有用状态机对 IF 阶段进行阻塞，导致 PC 的更新出错，根本原因还是在于不用状态机，逻辑较为混乱，当加入状态机后，PC 更新逻辑清晰了许多，本人最开始代码如下：

```
//Branch or Jump
always @(posedge clk) begin
        if(rst) begin
                Branch_or_Jump_reg <= 1'b0;
        end
        else if(ID_Valid | IF_Allow_in) begin
                Branch_or_Jump_reg <= Branch_or_Jump;
        end
end

always @(posedge clk) begin
        Branch_or_Jump_temp <= Branch_or_Jump_reg;
end


//PC_new
always @(posedge clk) begin
        if(rst) begin
                PC_new_reg <= PC_normal;
        end
        else if (ID_Valid | (~Branch_or_Jump_reg & Branch_or_Jump_temp) | (~|IF_PC)) begin
                PC_new_reg <= PC_new;
        end
end


  //PC
  always @(posedge clk) begin
         if(rst) begin
                 IF_PC <= 32'hfffffffc;
         end
         else if(Inst_Req_Valid & Inst_Req_Ready | (Branch_or_Jump_reg & ~Branch_or_Jump_temp) ) begin
                 IF_PC <= PC_new_reg;
         end
  end


 //PC更新
 assign PC_normal                        = IF_PC + 4;
 assign PC_new                           = Branch_or_Jump ? PC_abnormal : PC_normal;

 //读存信号
```

但实际上，这种做法正确需要在 ID 传来的跳转分支信号后的两个周期内 Req 握手成功，条件太为苛刻（虽然在 fpga 仿真中这个条件是满足的）。

解决过程：在没有解决思路一段时间后，有同学跟我说，为什么不尝试用状态机呢，在使用状态机后，一切迎刃而解，不得不感慨，状态机真好用，最后代码在之前已经展示了，这里就不赘述了。

2. RF_wen 连续两个周期拉高，导致行为仿真出错（框架原因）：

由于行为仿真的框架的原因，当连续两个周期 RF_wen 拉高时，pc_golden 不会进行更新，这样就导致了行为仿真的出错。本人最开始代码如下：

```
assign retired   = WB_Valid;
assign fifo_data = {RF_wen,             //69:69
                    RF_waddr,           //68:64
                    final_result,       //63:32
                    PC                  //31:0
                    };
```

解决过程：找了很久没有找到自己出错的原因，最后通过别人得知，是 RF_wen 连续拉高的原因，也是因为行为仿真的框架不完善，找到原因后并进行修改，新添加了两个寄存器用来保存值，使两个 RF_wen 拉高周期之间加上一个间隔，最后代码如下：

```
//Inst_retired(防止连续两个周期内RF_wen均拉高，无其它作用，希望框架尽快改善)
always @(posedge clk) begin
        if (Temp_inst_retire[69]) begin
                inst_retire <= Temp_inst_retire;
        end
        else if(~(inst_retire[69] & RF_wen)) begin
                inst_retire <= retire_data;
        end
        else begin
                inst_retire <= 70'b0;
        end
end

always @(posedge clk) begin
        if(inst_retire[69] & RF_wen) begin
                Temp_inst_retire <= retire_data;
        end
        else begin
                Temp_inst_retire <= 70'b0;
        end
end
```

```
//retired_instruction
assign retired    = WB_Valid;
assign retire_data = {RF_wen,              //69:69
                      RF_waddr,            //68:64
                      final_result,        //63:32
                      PC                   //31:0
                      };
```

3. 握手信号（Inst_Req_Valid）出错（框架问题）：

本人最开始的 Inst_Req_Valid 握手信号如下所示：

```
//访存信号
assign Inst_Req_Valid = (current_state == IF); //行为仿真需要错开
assign Inst_Ready = (current_state == IW || current_state == INIT);
```

后来在大群里得知：似乎是行为仿真框架的原因，当依次发生 Inst_Req_Valid， MemRead， Read_data_Ready 握手成功时，

Inst_Valid 永远不会拉高，希望该问题早点得到解决，最后代码如下：

```
//访存信号
assign Inst_Req_Valid = (current_state == IF) & ~MemRead; //行为仿真需要错开
assign Inst_Ready = (current_state == IW || current_state == INIT);
```

4. 其它错误

除上述错误外，本人也犯过一些小错误，大多是因为流水线设计不够完善（如旁路返回数据的优先级），以及直接搬以前实验的代码而未加以修改或者就是因为粗心导致（如字段在 Instruction 的位置搞错），这里就不一一赘述了。

三、 对讲义中思考题（如有）的理解和回答

性能对比如下：

I. 五级流水处理器性能计数器：

```
[queen] Queen placement: * Passed.
Result:
        Cycles: at least 6103882
        Instruction Count: at least 87084
        Memory Read Instruction Count: at least 14419
        Memory Write Instruction Count: at least 12356
        Branch Instruction Count: at least 6078
        Jump Instruction Count: at least 4118
```

```
[qsort] Quick sort: * Passed.
Result:
       Cycles: at least 760499
       Instruction Count: at least 10900
       Memory Read Instruction Count: at least 1523
       Memory Write Instruction Count: at least 944
       Branch Instruction Count: at least 1678
       Jump Instruction Count: at least 170
benchmark finished
```

```
reset: MMIO accessed
[md5] MD5 digest: * Passed.
Result:
       Cycles: at least 369020
       Instruction Count: at least 5279
       Memory Read Instruction Count: at least 544
       Memory Write Instruction Count: at least 101
       Branch Instruction Count: at least 394
       Jump Instruction Count: at least 68
benchmark finished
```

II. 对应多周期处理器的性能计数器：

```
[queen] Queen placement: * Passed.
Result:
       Cycles: at least 6950886
       Instruction Count: at least 81489
       Memory Read Instruction Count: at least 14418
       Memory Write Instruction Count: at least 12355
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 5466794
       Memory Read Request Delay Cycles: at least 14423
       Read Data Delay Cycles: at least 1003900
       Memory Write Request Delay Cycles: at least 12355
       Branch Instruction Count: at least 6078
       Jump Instruction Count: at least 4118
benchmark finished
```

```
[qsort] Quick sort: * Passed.
Result:
       Cycles: at least 785666
       Instruction Count: at least 9479
       Memory Read Instruction Count: at least 1522
       Memory Write Instruction Count: at least 943
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 630293
       Memory Read Request Delay Cycles: at least 1527
       Read Data Delay Cycles: at least 103778
       Memory Write Request Delay Cycles: at least 943
       Branch Instruction Count: at least 1678
       Jump Instruction Count: at least 170
benchmark finished
```

```
[md5] MD5 digest: * Passed.
Result:
       Cycles: at least 386325
       Instruction Count: at least 4914
       Memory Read Instruction Count: at least 543
       Memory Write Instruction Count: at least 100
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 325447
       Memory Read Request Delay Cycles: at least 548
       Read Data Delay Cycles: at least 36272
       Memory Write Request Delay Cycles: at least 100
       Branch Instruction Count: at least 394
       Jump Instruction Count: at least 68
benchmark finished
```

可以看出，流水线的效果并不是很显著，本人认为是因为访问真实内存时,延时较高。当频繁进行真实内存访问时,五级流水线处于频繁处于部分阻塞的状态，导致并行的指令数减少，以至于和多周期处理器性能差异不大。同时，由于运行状态稳定性的波动，可能导致部分时候五级流水线处理器性能劣于多周期处理器的情况。

因此,五级流水线处理器的平均CPI相较多周期处理器有所减少,但减

少幅度不大（部分程序甚至会增加），需要 cache 的辅助来实现性能的进一步提升。

四、 在课后，你花费了大约___55__小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

这次实验较为复杂，主要在于各种握手信号的设计（实现流水级阻塞的关键），再就是 PC 更新较为复杂，既要考虑与内存的握手，又要考虑与下一流水级的握手，且需要根据跳转分支的信号进行阻塞。在这个过程中，本人开始没有使用状态机，发现逻辑较为混乱，后面在使用状态机后，逻辑清晰了许多，也很快解决了问题。而且在本次实验的过程中，本人发现了很多行为仿真框架的小错误，因为这些小错误，导致本人 debug 时间较久。

首先就是，当连续两个周期 RF_wen 拉高时，pc_golden 不会进行更新，这样就导致了行为仿真的出错；其次就是，当依次发生 Inst_Req_Valid， MemRead， Read_data_Ready 握手成功时，

Inst_Valid 永远不会拉高，导致取值失败。

上述的错误都是经过了对测试代码的研究以及与很多具有相同问题的同学讨论才找出来的问题，所以还是希望能尽快完善一下吧，不然出现这些问题，却不知道自己错在哪真的特别折磨。

最后特别感谢老师，助教们和实验群的同学对本人提出的问题的帮助与解答。