中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929034 姓名：侯汝垚 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

1. 七级流水线设计

a. IF

取指阶段的主要工作就是发送指令请求，并根据条件将下一条指令地址写入 PC，除了复位外共有 5 种可能：①取数指令时若发生数据冒险 PC 不变；②ID 阶段译码获得跳转或分支指令且 take，那么 PC 变为 ID 阶段计算出的 PC_branch；③遇到 MRET 指令时将 MEPC 寄存器的值赋给 PC；④携带中断指令在写回阶段时将中断服务程序地址赋给 PC；⑤不属于上述任意一种情况时 pc+4。

```
reg  [31:0] PC_reg;
wire [31:0] PC_plus = PC + 4;

always @(posedge clk) begin
        if (rst)
                PC_reg <= 32'b0;
        else if (Go && load_hazard_control)
                PC_reg <= PC_reg;
        else if (Go && (branchValid || jump))
                PC_reg <= PC_branch;
        else if (Go && RDW_WB_MRET)
                PC_reg <= MEPC;
        else if (Go && RDW_WB_intr)
                PC_reg <= 32'b100;
        else if (Go)
                PC_reg <= PC_plus;
end
```

b. IW

IW 阶段主要工作是接收指令，握手成功后需要用一个额外寄存器存指令，因为握手成功时可能其他流水级的任务没有完成，仍然需要等待。

c. ID

译码阶段生成大部分控制信号并存入寄存器，同时也读取寄存器堆将数据存入寄存器。这些控制信号依次流到后续流水级，某些不再被用到的信号就不必再传递。这些信号基本与单周期处理器需要的信号相同，其中 func3 因为在后续流水级使用较多，因此直接传递 func3。

```
reg [ 4:0] ID_EX_rd;
reg [4:0] ID_EX_rs1, ID_EX_rs2; // two index of read registers to deal with forwarding
reg [31:0] ID_EX_rdata1, ID_EX_rdata2, ID_EX_PC, ID_EX_Imm; // 32 bit data
reg ID_EX_ALU_SrcA, ID_EX_MemRead, ID_EX_MemWrite, ID_EX_RegWrite, ID_EX_PCSrc, ID_EX_ShifterOp;
(* max_fanout = 50 *) reg ID_EX_ALU_SrcB;
reg ID_EX_intr, ID_EX_MRET, ID_EX_Cache;
reg [2:0] ID_EX_ALUOp, ID_EX_func3;
reg [1:0] ID_EX_ALUOutSrc;
```

为了减少分支预测失败（没有分支预测器，因此总是预测分支不发生）时的指令损失，选择在 ID 阶段用一个额外的加法器判断出分支是否成立，同时计算跳转地址，这样预测失败时损失两条指令，若在 EX 阶段判断则损失三条指令，代价是增加了处理数据冒险和控制冒险的电路，对冒险的处理在后面详细介绍。ID 阶段需要多次加法计算（PC_branch 和分支成立条件），组合逻辑路径长，用 bit_gen 进行时序分析时若不考虑乘法指令，该 cpu 的关键路径就是 ID 阶段处理分支指令的这段电路，这样判断也会降低最大主频。

```
assign {cout, ID_result} = RF_rdata1_valid + ~RF_rdata2_valid + 1'b1;

wire ID_slt = RF_rdata1_valid[31] && ~RF_rdata2_valid[31] || ((RF_rdata1_valid[31] ^ ~RF_rdata2_valid[31]) && ID_result[31]);
wire ID_sltu = cout;
wire ID_Zero = ~(|ID_result);
wire branchValid = branch && (~func3[2] && (ID_Zero ^ func3[0]) || func3[2] && ~func3[1] && (ID_slt ^ func3[0]) || func3[1] && (ID_sltu ^ func3[0]));
wire jump = J_type || JALR;
wire [31:0] PC_JALR = RF_rdata1_valid + I_imm;
wire [31:0] PC_branch = (JALR) ? {PC_JALR[31:1], 1'b0} : IW_ID_PC + PC_imm;
```

d. EX

执行阶段计算出访存指令的地址或者算术指令、移位指令的结果。和单周期 CPU 基本一致。

e. MEM

访存阶段发送内存请求和地址，等待内存 ready。根据上述提到的 func3 可以获得 Write_strb。

f. RDW

RDW 阶段等待内存返回有效数据。根据 func3 和 ALUOut 后两位生成掩码选取 32 位数中需要的部分。

g. WB

WB 阶段根据 RDW_WB_RegWrite 信号决定是否将结果写入 RegFile，并完成指令提交。RF_wen 信号只拉高一个周期。

2. 数据冒险处理

在译码阶段处理数据冒险，读取的数据可能是 EX、MEM、RDW、WB 四个阶段的计算结果，判断条件就是后面几个流水级的 rd 是否等于 ID 阶段的 rs1 或者 rs2，是否需要写回（RegWrite 拉高），并且还要不能为全 0。MEM 阶段和 RDW 阶段的结果可以直接通过流水级之间的寄存器获取，而 EX 阶段则直接将计算结果连到 ID 中的 rdata1 或 rdata2。若当前读取的数据是当前写入的数据，即 ID 阶段 rs1 或 rs2 等于 WB 阶段的 rd，只需加一个判断逻辑，若寄存器堆读地址和写地址一致，则直接将写入的数据作为读取数据输出。

```
// four possible cases can lead to data hazard

// to judge whether raddr1 == waddr or raddr2 == waddr;
wire data1_hazard_rw, data2_hazard_rw;
assign data1_hazard_rw = (rs1 == RF_waddr) && RDW_WB_RegWrite && |rs1;
assign data2_hazard_rw = (rs2 == RF_waddr) && RDW_WB_RegWrite && |rs2;

wire [3:0] data1_hazard, data2_hazard;

assign data1_hazard[2:0] = {rs1 == MEM_RDW_rd && MEM_RDW_RegWrite && |rs1,
                            rs1 == EX_MEM_rd  && EX_MEM_RegWrite  && |rs1,
                            rs1 == ID_EX_rd   && ID_EX_RegWrite   && |rs1};
assign data1_hazard[3] = (data1_hazard[2:0] == 3'b0);

assign data2_hazard[2:0] = {rs2 == MEM_RDW_rd && MEM_RDW_RegWrite && |rs2,
                            rs2 == EX_MEM_rd  && EX_MEM_RegWrite  && |rs2,
                            rs2 == ID_EX_rd   && ID_EX_RegWrite   && |rs2};
assign data2_hazard[3] = (data2_hazard[2:0] == 3'b0);
```

如果相关的数据中前面的指令是取数指令，那么必须等到 RDW 阶段结束后才能获得结果，因此必须将流水线 IF、IW、ID 阶段停止，直到读取数据完成。load_hazard 信号拉高时，PC 不变。若发送冒险的取数指令在 MEM 阶段，则需要停止三个指令周期，一个指令周期后改指令到达 RDW 阶段，此时仍需停止两个指令周期，以此类推，只有后续流水线中不存在取数相关的数据冒险，load_hazard_control 才为 0。

```
// the data hazard could not be solved by forwarding
// read a reg which will be written by a load instruction
wire [2:0] load_hazard;
assign load_hazard[2] = MEM_RDW_MemRead && (MEM_RDW_rd == rs1 && |rs1 || MEM_RDW_rd == rs2 && |rs2);
assign load_hazard[1] = EX_MEM_MemRead  && (EX_MEM_rd  == rs1 && |rs1 || EX_MEM_rd  == rs2 && |rs2);
assign load_hazard[0] = ID_EX_MemRead   && (ID_EX_rd   == rs1 && |rs1 || ID_EX_rd   == rs2 && |rs2);
wire load_hazard_control = |load_hazard || (MemWrite_delay && S_type);
```

最后 rdata1 和 rdata2 共有五个来源。

```
wire [31:0] RF_rdata1_valid, RF_rdata2_valid;

assign RF_rdata1_valid = {32{data1_hazard[0]}} & Result
                       | {32{data1_hazard[1:0] == 2'b10}} & EX_MEM_ALUOut
                       | {32{data1_hazard[2:0] == 3'b100}} & MEM_RDW_ALUOut
                       | {32{data1_hazard[3] &&  data1_hazard_rw}} & RDW_WB_MDR
                       | {32{data1_hazard[3] && ~data1_hazard_rw}} & RF_rdata1;

assign RF_rdata2_valid = {32{data2_hazard[0]}} & Result
                       | {32{data2_hazard[1:0] == 2'b10}} & EX_MEM_ALUOut
                       | {32{data2_hazard[2:0] == 3'b100}} & MEM_RDW_ALUOut
                       | {32{data2_hazard[3] &&  data2_hazard_rw}} & RDW_WB_MDR
                       | {32{data2_hazard[3] && ~data2_hazard_rw}} & RF_rdata2;
```

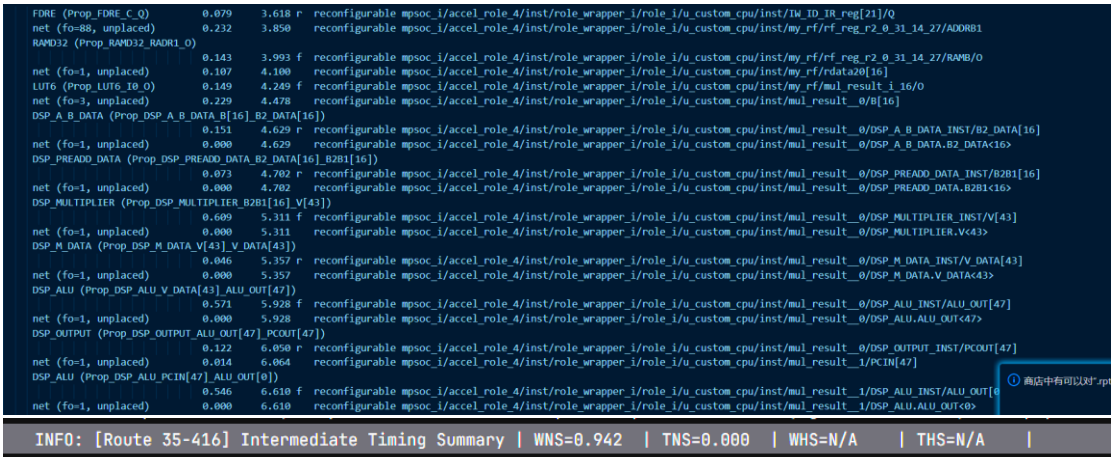
3. 控制冒险处理

```
reg [31:0] IW_ID_IR;
always @(posedge clk) begin
        if (rst)
                IW_ID_IR <= 32'b0;
        else if ((load_hazard_control || IW_intr_block) && Go)
                IW_ID_IR <= IW_ID_IR;
        else if ((branchValid || jump || pre_failed || MRET) && Go)
                IW_ID_IR <= 32'b0;
        else if (Go)
                IW_ID_IR <= IR;
end
```

没有精力和时间再完成分支预测器，因此总是预测分支不发生，一旦分支发生，即 jump 或 branchValid 拉高，那么 IF 和 IW 中的数据要作废，此时只需连续往 IR 中填入两次全 0 的 NOP 指令即可。

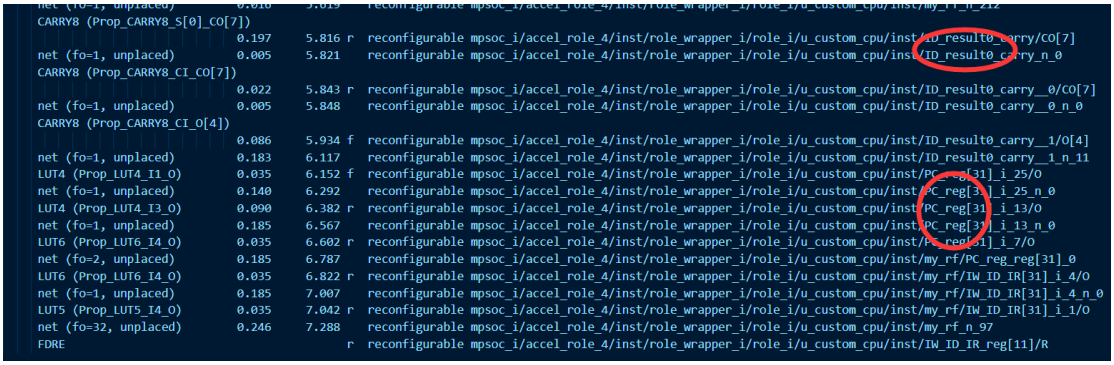
4. 时序分析

频率为 100MHz 时，WNS 大于零，关键路径是乘法的计算。



不考虑乘法指令，将频率调为 250MHz 时 WNS 小于 0，为 200MHz 时大于 0，最大频率大约在 200MHz，此时关键路径为 ID 阶段判断分支的组合逻辑。在扇出过多的信号前加前缀限制，然后将一些长的表达式转化为二分形式减少组合逻辑路径延时。

```
(* max_fanout = 50 *) reg ID_EX_ALU_SrcB;
```



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）

1. 流水线如何“流动”？

流水线中 IF、IW、MEM、RDW 需要多个周期才能完成，且每个阶段完成的时间不一定一致，而流水线必须统一步骤，因此定义一个信号 Go，当上述四个流水级均完成后拉高 Go，使流水线统一运作。

2. 访存请求和等待数据错杂导致 cpu 卡死。

本实验采用七级流水线最开始是为了提高指令并行性，特别是 IF 和 IW，MEM 和 RDW 间的并行性，然而写完后的调试过程中发现，发送访存请求和接受数据这两个阶段的并行性不是很强，并且很容易紊乱，当连续发送多次访存请求时，会出现发送请求、获取数据在同一个周期完成，然后访存信号就紊乱。在运行初期基本都是有序进行先获得指令后发送读取下一条指令的请求，但是运行到某一时刻就会紊乱，根据微信群中同学交流，问题可能在于发送指令请求过于频繁。完成 cache 后，我的 cache 也不能同时处理访存和访存请求，为了正确握手，我将发送访存请求和获取数据两个阶段分开，只有当上次数据已经取到才发送访存请求。因此我的七级流水线实际上和经典的五级流水线性能应该相当甚至不如五级流水线，吞吐率并没有得到改善。但是如果内存模块可以并行处理请求和发送数据，那么七级流水线应该会优于五级流水线。

3. 行为仿真中最后一条指令写回前上一条存数指令已经完毕。

采用七级流水线后，存数指令在 MEM 阶段就存入数据，因此存数指令实际上在前一条指令写回前就已经完成。根据仿真的 custm_cpu_test.v 文件的设计，trace_end 信号只有在最后一条指令提交后才拉高，但是此时前一条存数指令已经完成，仿真模块检测不到往 0xC 写 0 的这一个动作，因此行为仿真将会卡死。解决这一问题的一个简单办法是将所有存数指令延迟一个周期，这样就可以保证存数指令不会早于下一条指令完成，但这样也损失了一定的性能，好在一般程序中存数指令占比不高。另一个方法是改写仿真顶层模块，提前一个流水阶段拉高 trace_end。根据查到的一些资料，RISC-V 采用的是弱内存顺序模型，理论上我们可以不必保证存数一定要严格在上一条指令后才完成。

三、 在课后，你花费了大约 30 小时完成此次实验。

四、 对于此次实验的心得、感受和建议

流水线实验初看似乎很容易，思路就是在单周期处理器的基础上将处理器分级并插入几排寄存器即可，但是实际写起来难度还是很大，首先最大的问题依然是访存信号的问题，微信群里大部分同学遇到的问题也是关于握手信号，希望可以拥有更加真实的仿真访存模型。其次的难点就在于对数据冒险和控制冒险的处理，理清某个阶段的数据来源、数据去向非常关键。