中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 朱夏楠 学号 2022K8009929031 专业 计算机科学与技术
实验项目编号 5.1 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` 推送到实验课 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 Verilog HDL 关键代码段{包含注释}及其对应的逻辑电路结构图{自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

• **RISCV32 流水线设计**。

我的流水线处理器为五级流水, 实现分为三个阶段: 第一个阶段可以实现流水线的正常流动, 并且进一步实现了阻塞和气泡插入; 第二个阶段实现了对数据冒险的处理; 第三个阶段则实现了一个 Gshare 分支预测器。

关于如何让流水线正常流动, 我有两种思路, 一是默认流水线的状态为正常流动, 想要对流水线进行阻塞和插入气泡的操作, 需要额外的控制信号对每一级流水线寄存器进行控制, 这样做相对来讲更符合我的直觉, 但缺点是插入气泡时所需要的控制信号有些复杂, 不容易修改; 因此我在依照这个思路写完流水线之后, 又重新根据助教老师在课上提到的基于握手机制的控制方式写了一遍流水线。

在握手的机制下, 有三个信号用于控制流水线寄存器的状态, 分别是 ready、valid 以及 instr_valid。ready 信号代表当前的流水级将在本周期完成对应的指令, 在下一周期可以接收新的指令, 因此在当前流水级阻塞的时候, 需要置零; valid 信号代表前一级允许其指令流入当前的流水级, 因此在上一级内容是气泡时, valid 信号要置零; instr_valid 代表当前流水级内的指令是否有效, 若是一个气泡流入了当前流水级, 会将当前级的 instr_valid 置零, 并且会随着其他信号传入下一级。

```
assign f_valid = ~id_bubble & ~rst & ~WrongPreReg;
    //WrongPreReg代表分支预测错误, 需要冲刷流水线
assign d_valid = ~exbubble & ~rst;
assign e_valid = ~mbubble & ~rst;
assign m_valid = ~wbubble & ~rst;

assign f_ready = ~idblock & ~rst & ~WrongPreReg;
assign d_ready = ~exblock & ~rst;
assign e_ready = ~mblock & ~rst;
assign m_ready = ~wblock & ~rst;
```

```
assign f_instr_valid = ~id_bubble & ~WrongPreReg;
assign d_instr_valid = D_instr_valid & ~exbubble;
assign e_instr_valid = E_instr_valid & ~mbubble;
assign m_instr_valid = M_instr_valid & ~wbubble;
```

第二阶段是实现对于数据冒险的处理，在按顺序流动的流水线中，主要需要关心的是写后读相关，即前一条指令还未向寄存器内写入数据，后一条指令就已经从该寄存器中读取到数据了，从程序执行顺序的角度来看，这显然是错误的。为此，我在 ID 阶段添加了数据冒险判断，并实现了数据前递，即检测到数据冒险之后将未来得及写回的数据直接送给处于 ID 阶段的指令。需要注意的是，对于五级流水线，理论上来讲存在三种可能的冒险，分别为 EX-ID、MEM-ID、WB-ID，但在我的五级流水线实现中，流水的阻塞会导致该流水级之前的全部流水级当场停下来，而根据取指令的握手机制，最快也需要两个时钟周期才能取到下一条指令，因此如图所示，实际上不存在 EX-ID 的冲突。

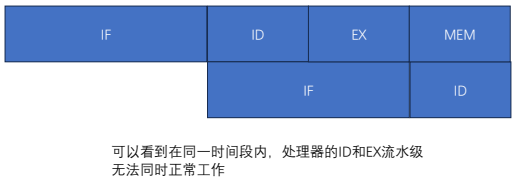

图 1: 一个简单的演示

最终的代码实现如下。需要注意存在一些伪数据冒险，比如要写回的的寄存器是零号寄存器，或者是检测到 ID 阶段的源地址和 MEM 或者是 WB 阶段的 rd 相同，但实际上 ID 阶段的指令并不需要源寄存器，只是恰好在 rs 段的值与与后面的流水级中指令的 rd 相同。这些都需要在实现代码时进行额外考虑。

```
//真实内存访问情况下的数据冒险似乎不可能在ID和EX阶段时发生
//instr_type = {isStype,isBtype,isJtype,isUtype,isItype_L,isItype_J,isItype_C,isRtype}


assign Wdata_risk_s1 = ~|(W_rd ^ D_rs1) & (|W_rd) & |W_instr_type[5:0];
assign Wdata_risk_s2 = ~|(W_rd ^ D_rs2) & (|W_rd)
                  & |W_instr_type[5:0] & (D_instr_type[7] | D_instr_type[6] |
                      D_instr_type[0]);

assign Mdata_risk_s1 = ~|(M_rd ^ D_rs1) & (|M_rd)
                  & |M_instr_type[5:0] & ~M_instr_type[3] & (D_instr_type[7] |
                      D_instr_type[6] | D_instr_type[0]);
assign Mdata_risk_s2 = ~|(M_rd ^ D_rs2) & (|M_rd)
                  & |M_instr_type[5:0] & ~M_instr_type[3] & (D_instr_type[7] |
                      D_instr_type[6] | D_instr_type[0]);
```

```
assign data_risk_s1 = Wdata_risk_s1 | Mdata_risk_s1;
assign data_risk_s2 = Wdata_risk_s2 | Mdata_risk_s2;

assign d_A = (rdata1 & {32{~data_risk_s1}})
           | (m_MEMResult & {32{Mdata_risk_s1 & m_MemtoReg}})
           | (M_EXResult & {32{Mdata_risk_s1 & ~m_MemtoReg}})
           | (w_Result & {32{Wdata_risk_s1 & ~Mdata_risk_s1}});
assign d_B = (rdata2 & {32{~data_risk_s2}})
           | (m_MEMResult & {32{Mdata_risk_s2 & m_MemtoReg}})
           | (M_EXResult & {32{Mdata_risk_s2 & ~m_MemtoReg}})
           | (w_Result & {32{Wdata_risk_s2 & ~Mdata_risk_s2}});
```

第三阶段为分支预测器的实现。分支预测器会在一个条件转移指令之后，预测下一条指令的地址，以保证流水线的正常运行，但如果预测错误，就需要进行冲刷流水线的操作。具体到我的实现，对于五级流水线，分支预测器所给出的结果如果是错误的，最多需要冲刷掉两条指令，冲刷操作即为将对应的流水级信号 instr_valid 拉低，进而使该指令的效果等同于 NOP 指令。我实现的分支预测算法是 Gshare，将一个全局预测表和一个局部分支预测表相结合，综合得出一个分支预测的结果，简要的流程如下。

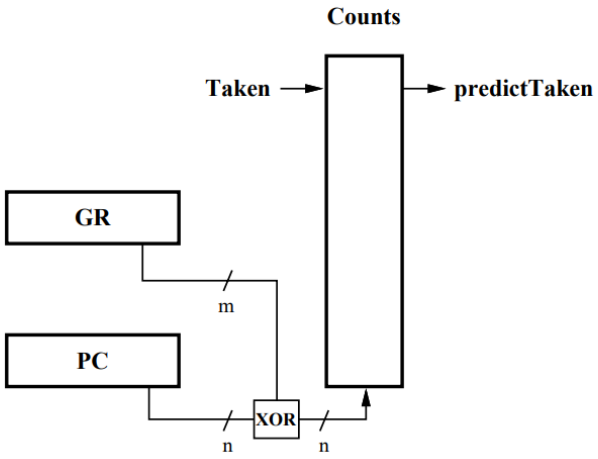

图 2: Gshare 示意

其中 GR 是一个全局分支预测表，用于记录最近所有分支预测的结果，1 代表跳转，0 代表不跳转，Counts 是局部分支预测表，本质是许多 2bit 饱和计数器构成的寄存器堆，进行分支预测时，GR 和 PC 的后几位进行异或运算，得到一个 Hash 值，若对应计数器的值为 2'b11 或 2'b10，则认为可以跳转，否则认为不能跳转。随后进行饱和计数器的更新，若当前分支跳转，则 Hash 值对应的 2bit 饱和计数器就加一，否则就减一。

最后得出的处理器大致结构如下图所示

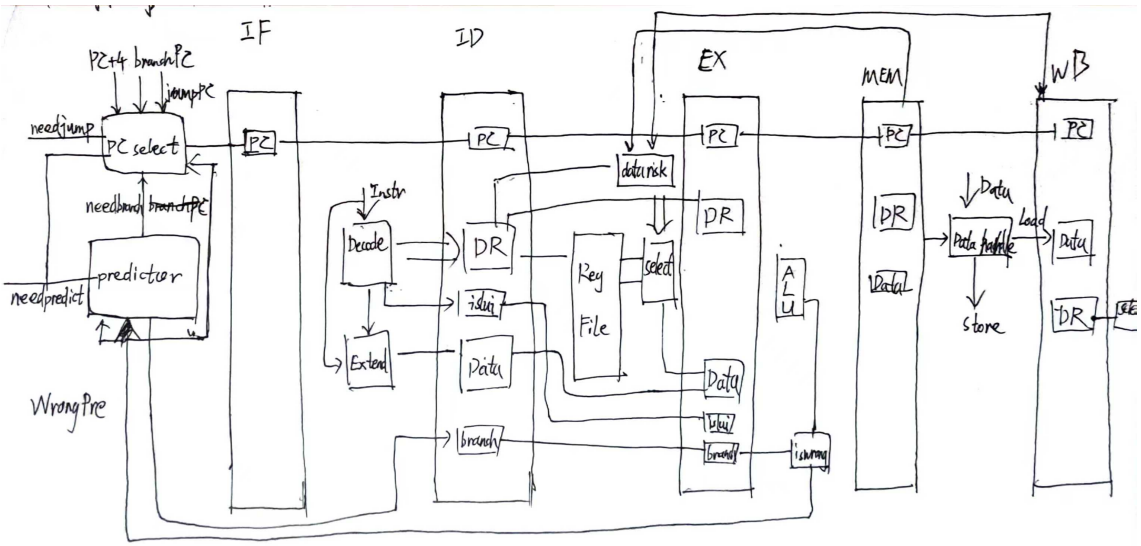

图 3: 流水线处理器结构

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)

• **读指令实现存在问题**。

我在进行该实验之前,认为,读指令时,需要 Inst_Req_Valid 信号和 Inst_Req_Ready 信号同时拉高,才会将地址送给内存。但在进行行为仿真时我发现会出现在分支预测错误之后,进行流水线冲刷,冲刷前错误的分支还未收到 Inst_Req_Ready 的信号,但冲刷后下一条得到指令仍是冲刷前的指令,如图所示,在 16360 到 16370ps 处若保持 Inst_Req_Valid 拉高,未等 Inst_Req_Ready 拉高便会直接将预测错误的 PC 值保存起来,输送给内存。这种情况之所以会出现,是因为我对于握手机制的理解和实验框架的实现不同,实验框架只要指令数据传输队列为空,就会立即保存当前的 PC 值,而 Inst_Req_Ready 信号则是由内存产生,两个行为之间是不同步的。

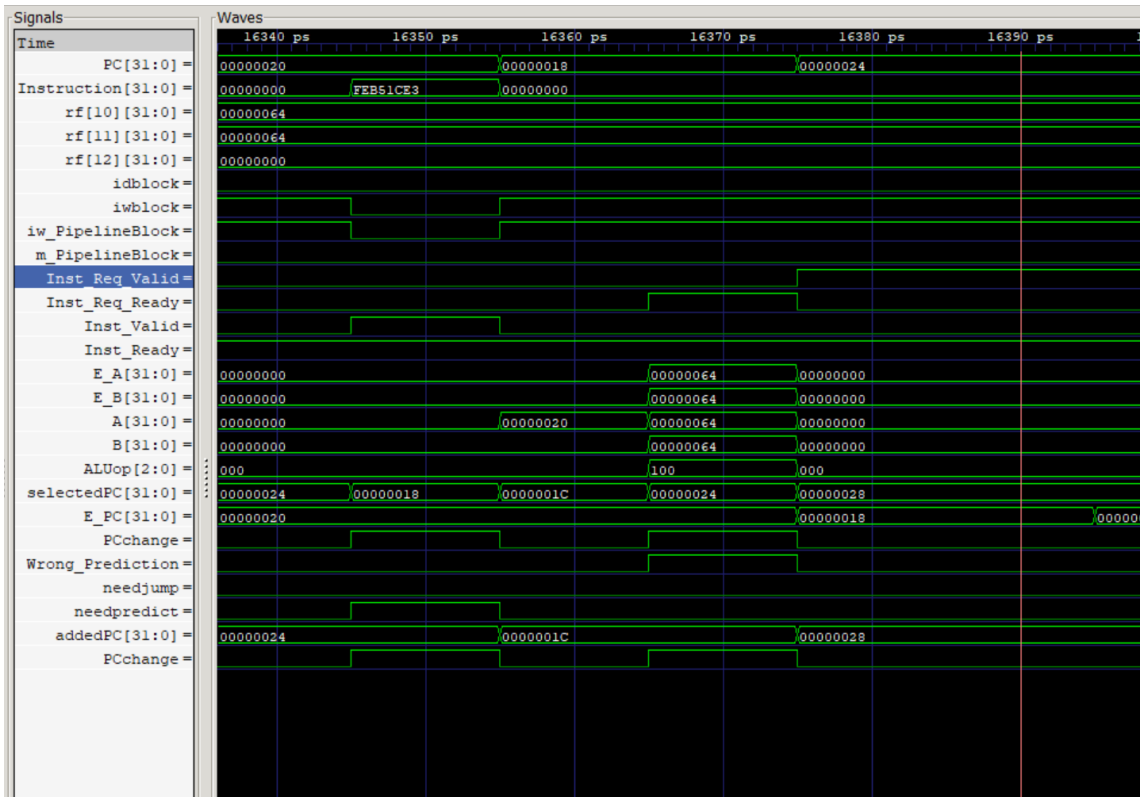

图 4: 错误示例

有两种解决方案，一是不实现分支预测，二是完成指令 Cache。该问题是由于分支预测错误出现的，因此只要不进行分支预测，设计硬件逻辑，使得在取到分支指令后立马给出预测结果即可；而方案二相当于在 CPU 和数据传输队列之间插入了一个中间层，完成指令 Cache 后，CPU 读取指令直接与 Cache 进行交互，因此可以按照我自己的设想设计 Cache，保证 CPU 的取指能按照我的想法进行。而 Cache 由于是多周期的，与数据传输队列的交互不会存在以上的问题。

三、 思考与总结

- **流水线性能评估**。

对比九个 microbench 程序运行所消耗的总时钟周期数，可以发现实现了流水线之后，确实有了一定的提升，经过粗略计算，发现大部分程序性能提升了百分之十左右，小部分提升则不超过百分之三。

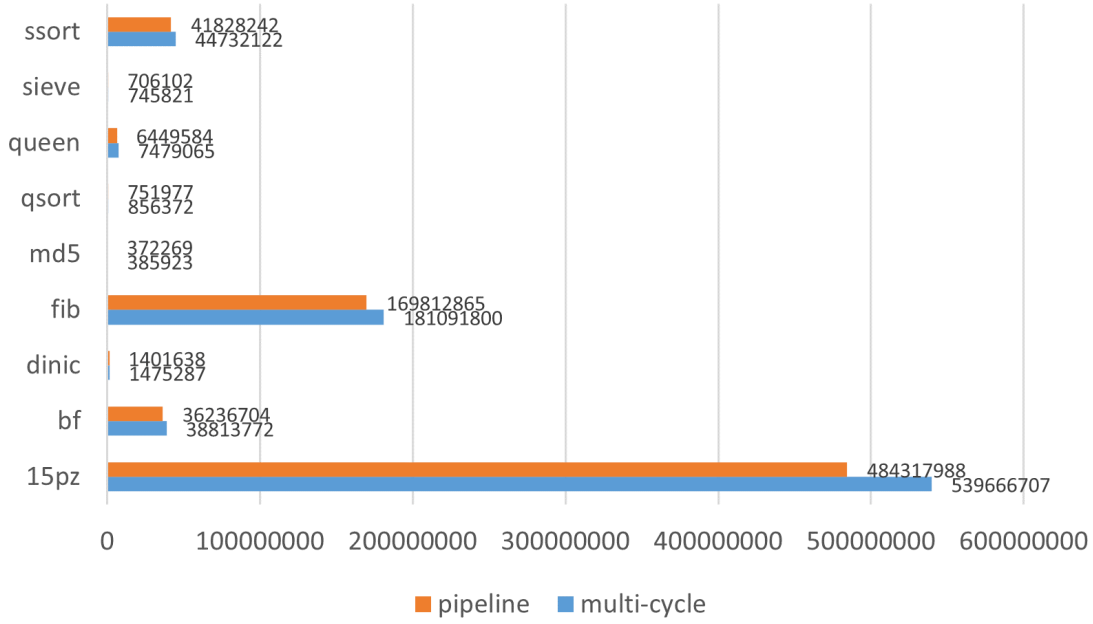

图 5: 流水线与多周期处理器性能对比

- **分支预测器性能评估。**

  根据内置的分支预测性能计数器,可以得到策略为 Gshare 的流水线处理器预测正确率。

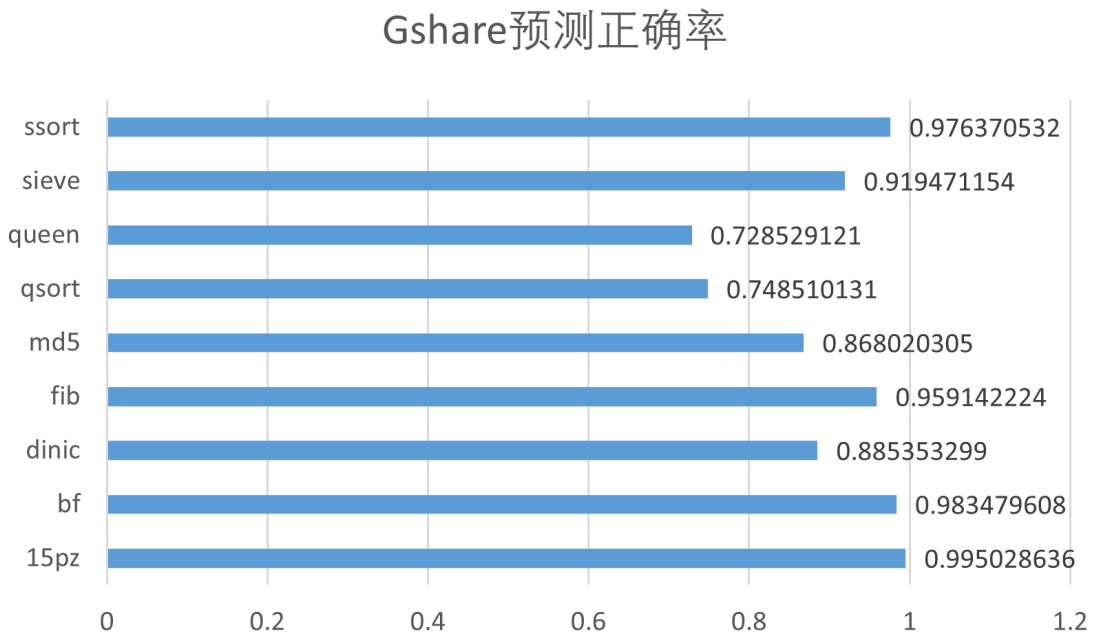

图 6: Gshare 预测正确率

可以看到,在分支预测次数较多的时候,Gshare 的正确率都达到了百分之九十五以上,而程序分支预测次数较少的时候,正确率则波动较大,这是由于在开始时,饱和计数器被置为 2'b00,需要进行大量的更新才能使每一个饱和预测器都被调整到合适的状态。因而总体来看,分支预测器的效果还是相当好的。

四、 实验所耗时间

在课后,你花费了大约____40____ 小时完成此次实验。