# Lab4 实验报告

## PART1 SCOREBOARD

Lab4 的第一个实验分为两个部分来完成。首先将一个两周期流水线按照实验文档中的要求修改为三周期流水线，形成 F，EX，WB 三级流水线结构；然后再使用 Scoreboard 策略来解决流水线中发生的数据冒险。具体实现过程如下。

第一步将 Decode 和 Register Read 逻辑从第二级转移到第一级。如此一来流水线的第一、二级流水之间传递的数据应该添加 DecodeInst、两个从寄存器堆中读取的数据和一个从协处理器寄存器中读取的数据。修改后数据结构的定义如图 1。

```
typedef struct {
  DecodedInst dInst;
  Data rVal1;
  Data rVal2;
  Data copVal;
  Addr pc;
  Addr ppc;
  Bool epoch;
} Fetch2Execute deriving (Bits, Eq);
```

**图 1 F-EX 级流水线寄存器**

第二步将 WB 逻辑从第二流水级移动到第三流水级。因此 EX 级应该将需要写入的寄存器和数据信息传递到第三级流水线。数据结构如图 2。

```
typedef struct {
  Maybe#(FullIndx) dst;
  Data             data;
} Execute2Writeback deriving(Bits, Eq);
```

**图 2 EX-WB 级流水线寄存器**

第三步就需要解决流水线冒险的问题。流水线冒险分为控制冒险，结构冒险和数据冒险，其中结构冒险在本实验中没有涉及。

首先考虑控制冒险，即由于指令跳转导致的冒险。在 Lab3 中我们已经堆控制冒险进行了处理，并且使用了一个简陋的 BTB 进行分支预测。在 Lab4 中仍

然可以沿用 Lab3 的控制冒险处理机制，唯一需要考虑的问题是加上专门的 WB 流水及之后需不需要添加额外的机制。而实际上，这个问题是不存在的。因为 EX 级向 WB 级传递的描述写入寄存器编号的数据结构是 Maybe，这是一个 Bluespec 内置的数据结构，其中含有一个 valid 标志。当由于分支预测失败对第二级流水线进行清空（实际上就是对 eEpoch 和 fEpoch 寄存器进行取反）时，valid 默认置无效，因此 WB 级便不对流水线进行有效操作。综上，不需要针对控制冒险进行设计。

```
if(execRedirect.notEmpty) begin
  execRedirect.deq;
  pcPred.update(execRedirect.first);
end
// change pc and the fetch's copy of the epoch only on a mispredict
if(execRedirect.notEmpty && execRedirect.first.mispredict) begin
  fEpoch <= !fEpoch;
  pc <= execRedirect.first.nextPc;
end
// fetch the new instruction on a non mispredict
else if(!sb.search1(dInst.src1) && !sb.search2(dInst.src2)) begin
  let ppc = pcPred.predPc(pc);
  pc <= ppc;
  f2Ex.enq(Fetch2Execute{dInst: dInst, rVal1: rVal1, rVal2: rVal2,
                         copVal: copVal, pc: pc, ppc: ppc, epoch: fEpoch});
  let dst = dInst.dst;
  if (isValid(dst) && validValue(dst).regType == Normal)
    sb.insert(dst);
end
else begin
  $display("Block the pipeline at pc: %h", pc);
end
```

**图 3 数据冒险控制逻辑**

其次考虑数据冒险，第一部分要求（暗示）我们使用计分板算法解决数据冒险，当发生数据冒险时将流水线暂停。流水线在 F 级译码之后便能够了解本条指令分别需要读取和写入哪个寄存器，而在 WB 级数据才能够真正地写入寄存器堆，据此我们可以设计我们的计分板算法。

当某条指令确定要将结果写入某个寄存器时（分支预测失败而 load 的指令除外），将此写入寄存器放入计分板中，待到 WB 级将数据写入寄存器之后再

从计分板中删去此项。期间只要 F 级检测到有指令需要读取此寄存器，便停顿流水线。相应的控制逻辑如图 3。

至此，我们便完成了计分板算法的实现。编译并模拟运行之后得到结果存入 l4p1.out（其中删除了一些无关信息，如 pkill 的 log）。从中可以得到信息，总指令数目为 867331，运行周期数为 1417182，显然存在更优的设计方案。

## PART2 BYPASS

Lab4 的第二个实验是添加旁路转发机制。

首先我们对流水线中数据的生产者和消费者进行梳理，值得注意的是，我们这里讨论的数据是指从寄存器堆中读取和写入数据。在此三级流水线中，数据的实际生产者是 EX 级，但是我们实际上最早应该在 EX 级把数据传递给 WB 级之后才能获取或者转发此数据，因此我们可以说实际上数据的生产者是 WB 级。在 F 级中需要读取寄存器堆，而获取数据的 DDL 实际上是指令运行至 EX 流水级时，因此我们可以将 F 级和 EX 级均看作数据的消费者。确定了生产者和消费者之后开始对转发旁路进行设计。

在 F 级中，如果读取的寄存器和要写入的寄存器是同一个寄存器，即处于 WB 级的指令写入的寄存器是处于 F 级的指令需要读取的寄存器，那么我们可以在寄存器内部进行数据的转发，也可以将数据从 WB 级转发至 F 级。此处考虑到我懒得看也懒得改 RFile 的代码，选择将数据从 WB 级转发至 F 级的方案。

在 EX 级中，如果本条指令已经读取的寄存器同时也在 WB 级被写入，说明此刻处于 EX 级的指令读取到的数据是一个脏数据，因此需要将 WB 级正在写入的数据转发到 EX 级参与运算。

综上所述，我们需要将 EX 级正在写入的寄存器和正在写入的数据分别转发至 F 和 EX 级流水级。转发的数据结构 WritebackRedirect 如图 4。

```
typedef struct {
  Maybe#(FullIndx) dst;
  Data             data;
} Execute2Writeback deriving(Bits, Eq);

typedef Execute2Writeback WritebackRedirect;
```

**图 4 WB 流水级的转发数据**

F 和 EX 流水级获取转发数据之后进行一定的判断，选择是否使用 WB 级转发的数据。

```
if(wbRedirect.notEmpty) wbRedirect.deq;
let rDst = wbRedirect.notEmpty? wbRedirect.first.dst : Invalid;
let rData = wbRedirect.notEmpty? wbRedirect.first.data : 32'h0;
```

**图 5 F 级转发处理逻辑**

至此我们完成了旁路转发的设计，而且我们惊奇地发现，完全没有必要将流水线进行暂停。于是我们将计分板相关设计删除。编译并模拟运行的输出保存在 l4p2.out 中，其中总指令条数为 867331，运行周期数为 882103。相对于使用计分板算法来暂停流水线，相同的指令条数运行周期数减少了 37.8%。

## DISCUSSION

**Have you really maximized performance?**

在现有的三周期框架的规定下，已经实现了流水线没有暂停地运行，基本上实现了将流水线性能最大化。可能造成流水线空转的只有控制冒险，这需要使用更加复杂精妙的方法对分支进行预测。

**Do all of your rules fire every cycle?**

是的，除了分支预测失败造成的流水线清空，每个 rule 每个周期都会有效执行。