

Lab 3

佟铭洋

23300240009

需要注意的是，在 3 月 31 日本人对跳转逻辑做了一些修改，此文档在最后保留了修改前的相关说明

思路与过程

对于任务中的这么多指令，其实可以分成几种：

branch:

`beq bne blt bge bltu bgeu`

compare and set:

`slti sltiu slt sltu`

shift:

`slli srli srai` - I type, RV32I

`slliw srliw sraiw` - I type, RV64I

`sll srl sra` - R type, RV32I

`sllw srlw sraw` - R type, RV64I

jump:

`jalr jal`

other:

`auipc`

AUIPC

首先，`auipc` 指令的实现已经在上次实验中完成了，只要把 `lui` 的 `srcA` 设置为 `PC` 即可。

SHIFT

先考虑 **shift** 类的指令，毕竟这种指令还是在玩 ALU。

实际上，注意到带有 `w` 的指令其实就是只关注数据的低 32 位，按照一个 32 位处理器的行为输出一个 32 位的结果，然后再将其扩展为 64 位。因为移位指令会出现一些非常麻烦的问题（不太好通过 64 位的结果截断取得数据，主要是要用输入的符号位进行扩展这一需求）。因此决定制作两套 ALU。

我们发现 ALU 正好可以用 3 位表示 `ALUop`，分别表示：

- 加法

- 减法
- 与
- 或
- 异或
- 左移
- 逻辑右移
- 算术右移

COMPARE AND SET

需要注意的是，后续的 branch 也需要用到比较，因此我们制作的比较部分应该也能满足后续的需求。类似于 x86 的，我们实际上可以在 `execute` 部分产生一个 `flags` 用来表示各种比较结果。

注意到，我们需要比较的为：等于，有符号意义下的小于，无符号意义下的小于。我们可以将比较的结果（一个 3 位二进制数）放在 `execute` 的 `flags` 中，传递给后续步骤（给 branch 使用）。对于 compare and set，我们只需要在 `execute` 阶段进行判断，以确认传递给接下来传递给 `memory` 的数据是 0 还是 1。

另外，需要在 `decode` 阶段额外为后续提供两个值：使用哪个 `flag`，是否需要取反。

```
u3 flags;//0: ia<ib 1: (unsigned)(ia<ib) 2: ia=ib

u64 compB;

assign compB = moduleIn.cmpSrcB ? moduleIn.imm : moduleIn.rs2;

always_comb begin
    if(moduleIn.rs1[63]^compB[63]) begin
        flags[0] = moduleIn.rs1[63];
    end else begin
        flags[0] = moduleIn.rs1<compB;
    end
    flags[1] = moduleIn.rs1<compB;
    flags[2] = moduleIn.rs1==compB;
end
```

对 `execute` 的输出进行一个统一的 mux:

```
always_comb begin
    if(moduleIn.rvm) begin
        if (moduleIn.rv64) begin
            datUse = {{32{mulOut32[31]}},mulOut32};
        end else begin
            datUse = mulOut;
        end
    end else begin
        if (moduleIn.rv64) begin
            datUse = {{32{aluOut32[31]}},aluOut32};
        end else begin
            if (moduleIn.cns) begin
```

```

        if(moduleIn.flagInv^(flags[moduleIn.useflag])) begin
            datUse = 1;
        end else begin
            datUse = 0;
        end

        end else begin
            datUse = aluOut;
        end
    end
end
end
end

```

BRANCH

本部分进行过更新，更新前的内容放在最后

对于 branch 指令，我们需要在 ALU 中计算出内存地址（加法），注意这条指令虽然用了 ALU，但并不需要写回。

由于 jump 指令需要写回，我们将跳转统一放在 **memory** 中进行。此时下一条指令在 **execute** 中。

我们要干两件事情：向控制线写跳转相关内容（组合逻辑），这用于将信号传递回 **fetch**，以及将正在 **decode** 和 **execute** 的指令记为无效；将现在这条指令继续传递给 **writeback**。

```

assign JumpEn = (moduleIn.isJump|(moduleIn.isBranch&moduleIn.flagResult)) &
moduleIn.valid;
assign JumpAddr = {moduleIn.aluOut[63:1],1'b0};

```

在 **decode** 和 **execute** 使用以下方法使指令作废

```

moduleOut.valid <= moduleIn.valid & ~JumpEn;

```

对于 Fetch 阶段的修改：增加

```

if(JumpEn) begin
    moduleOut.valid <= 0;
    curPC <= JumpAddr;
    nextPC <= JumpAddr + 4;
    ibus_req.addr <= JumpAddr;
    ibus_req.valid <= 1;
    instr_ok <= 0;
end

```

JUMP

注意到需要在 ALU 中计算一个地址用来跳转。但是写回的值不是 ALU 的结果，需要做特殊处理。

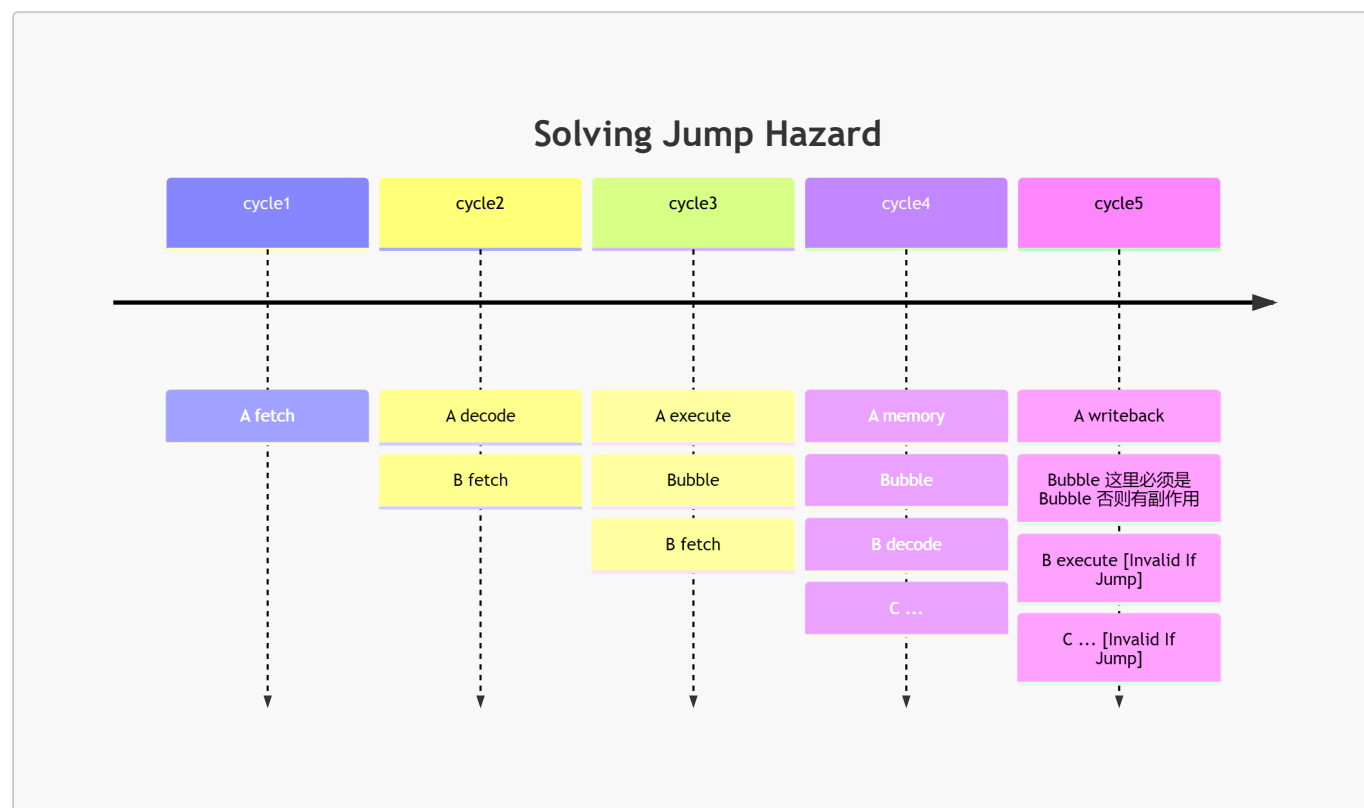
遇到的问题

大致思路是对的，没有遇到什么特殊的问题，只是出现了一点点由于编码时的疏忽产生的小问题（主要是在 **fetch** 已经在取指令了而总线还没有返回，这时候想要跳转需要等待总线空闲再进行，这里解决冲突的时候出现了一小问题），对照波形图进行修改即可。

注意到我们修改了跳转的逻辑，上述问题仅在修改前出现。

性能优化

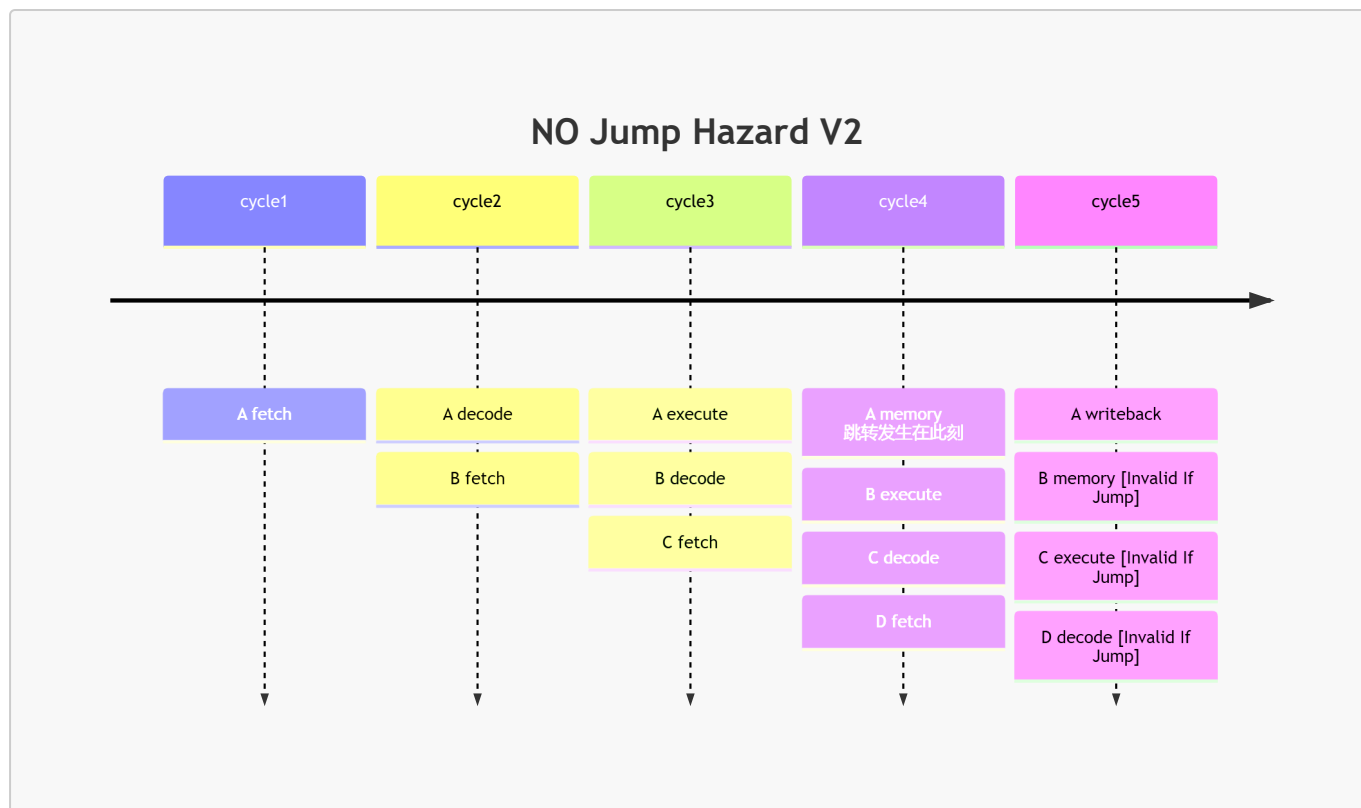
第一版的逻辑是这样：



并且 **writeback** 是在**上升沿之后**才发送信号给 **fetch** 的，此时 **fetch** 已经在取指令了，因此必须等现在这个指令取完（其实毫无意义，因为这个指令已经无效了）才能取跳转后的指令。

并且如果你去看修改之前的描述，你就会发现为了实现这个逻辑，逻辑是非常复杂的。

修改如下：



因此，这样优化带来两个好处：

- 如果跳转，减少一次无效的访存时间（取指）
- 如果不跳转，减少一个 Bubble（也是减少一次无效的访存时间）

总结

其实这里的 branch 使用了**静态分支预测**（其实就是不预测），默认他不跳转，后面继续往里塞指令，如果跳转就把前面的指令清掉。

此代码也可以跑 **extra** 测试，但其实乘除法的实现直接使用了乘号，在 verilog 中使用是没有问题的，但在 FPGA 中乘号的逻辑门太多了，必须要实现多周期的乘法才能降低周期。但由于乘除法的实现并不是课程要求的一部分，因此暂且这样，等到上板如果影响性能就去掉乘法。

本人目前有一些乘法实现的思路，也就是使用协处理器，并且协处理器正常不阻塞正常处理器的执行（除非对结果有依赖）。

过期的内容

BRANCH

对于 branch 指令，我们需要在 ALU 中计算出内存地址（加法），注意这条指令虽然用了 ALU，但并不需要写回。

由于 jump 指令需要写回，我们将跳转统一放在 **writeback** 中进行。因此跳转指令就都存在一个问题：进入到了 **writeback** 阶段，但其后一条指令已经进入了 **memory** 阶段并产生副作用了。因此我们必须类似于 load，在 **decode** 阶段进行判断，如果是一个跳转指令，则让 **fetch** 等待一个周期（相当于插入一个 nop）。

在 **writeback** 阶段获取 **flag** 即可。

注意到如果跳转，需要即刻将 `decode`，`execute` 阶段的指令记为无效，需要即刻更改 `pc`，以及正在 `fetch` 的指令。因此增加几个控制线。

```
if (ibus_resp.addr_ok & ibus_resp.data_ok) begin
    instr_ok <= 1;
    instr_n <= ibus_resp.data;
    ibus_req.valid <= 0;
end

if (JumpEn & ~jspc_ok) begin
    curPC <= JumpAddr;
    nextPC <= JumpAddr + 4;
    jspc_ok <= 1;
end
```

实际上需要用一套非常复杂的控制线来解决 PC 已经在取指的问题：

两套独立的逻辑

修改 fetch 的 pc

如果需要跳转，那么：如果当前 fetch 的 pc 已经是是跳转后的地址，则 `jspc_ok` 置为 1。因此如果需要跳转但 `jspc_ok` 为 0，则修改 pc 且将 `jspc_ok` 置为 1。

处理总线取指

只有在 总线取指完成 并且（要么不需要跳转，要么 `jump_ok` 为 1）时，才认为取指完成。

`jump_ok` 表示已经将跳转地址发送给总线了。

因此，如果总线在忙，此时要等待总线处理完毕再将跳转地址发送给总线。然后再将 `jump_ok` 置为 1。

```
if(JumpEn & ~jump_ok & instr_ok) begin
    instr_ok <= 0;
    jump_ok <= 1;
    ibus_req.addr <= JumpAddr;
    ibus_req.valid <= 1;
end
```

需要把 `execute` 和 `decode` 阶段的输出做以下处理：

```
moduleOut.valid <= moduleIn.valid & ~JumpEn;
```

也就是说，如果需要跳转，则将 `decode` 和 `execute` 此时的指令无效。