实验 5 - Out-of-order Pipeline Design with Scoreboard

实验步骤

本次实验要完成的部分主要在 `CtrlUnit.v` 中，即将控制信号设为正确的值。

- `normal_stall` 用于检测当前是否存在结构冲突或者 WAW 冲突。对于结构冲突，我们可以看当前这条指令是否会用到功能单元，以及对应的功能单元是否正在使用。对于 WAW 冲突，我们看当前指令要写回的寄存器 `dst` 对应的寄存器是否会被功能单元写回。

```
assign normal_stall = (use_FU != `FU_BLANK && FUS[use_FU][`BUSY]) | (|RRS[dst]);
```

- ensure WAR，写回寄存器值时要判断是否存在功能部件，这个部件对应的指令把这个写回寄存器作为源寄存器，这个寄存器已经 ready 了但还没有读。如果有这种情况，`WAR` 信号要设为 0，表明现在不能写回，要等待读取之后才能写。

  这里以 ALU 操作为例，我们需要检查除了 ALU 以外的功能单元是否与 ALU 的写回寄存器产生冲突。其他操作类似。

```
wire ALU_WAR = (
    (FUS[`FU_MEM][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_MEM][`RDY1])  &    //fill sth. here
    (FUS[`FU_MEM][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_MEM][`RDY2])  &    //fill sth. here
    (FUS[`FU_MUL][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_MUL][`RDY1])  &    //fill sth. here
    (FUS[`FU_MUL][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_MUL][`RDY2])  &    //fill sth. here
    (FUS[`FU_DIV][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_DIV][`RDY1])  &    //fill sth. here
    (FUS[`FU_DIV][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_DIV][`RDY2])  &    //fill sth. here
    (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_JUMP][`RDY1]) &    //fill sth. here
    (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS[`FU_JUMP][`RDY2])      //fill sth. here
);
```

- maintain the table
  - IS 阶段

    如果 `RO_en` 为 1，就发射当前这条指令。根据之前已经解码出来的控制信号更新对应的 `FUS` 和 `RRS`.

```
if (|dst) RRS[dst] <= use_FU;
FUS[use_FU][`BUSY] <= 1'b1;
FUS[use_FU][`OP_H:`OP_L] <= op;
FUS[use_FU][`DST_H:`DST_L] <= dst;
FUS[use_FU][`SRC1_H:`SRC1_L] <= src1;
FUS[use_FU][`SRC2_H:`SRC2_L] <= src2;
FUS[use_FU][`FU1_H:`FU1_L] <= fu1;
FUS[use_FU][`FU2_H:`FU2_L] <= fu2;
FUS[use_FU][`RDY1] <= rdy1;
FUS[use_FU][`RDY2] <= rdy2;
FUS[use_FU][`FU_DONE] <= 1'b0;
```

- RO 阶段

  如果某个功能部件对应的源寄存器都已经 ready，说明我们即将读取其值，将 ready 信号更新为 0，并将对应的 Qj Qk 清零即可。
  这里以 ALU 为例，其他功能部件类似。

```
else if (FUS[`FU_ALU][`RDY1] & FUS[`FU_ALU][`RDY2]) begin

    // ALU
    FUS[`FU_ALU][`RDY1] <= 1'b0;
    FUS[`FU_ALU][`RDY2] <= 1'b0;
    FUS[`FU_ALU][`FU1_H:`FU1_L] <= 3'b0;
    FUS[`FU_ALU][`FU2_H:`FU2_L] <= 3'b0;
    // ...
end
```

- EX 阶段

  在某个功能部件执行完后（根据 `done` 信号判断），将对应的功能部件 `FUS` 中的 `done` 也设为 1.
  这里以除法为例，其他功能部件类似。

```
if(DIV_done) begin
    FUS[`FU_DIV][`FU_DONE] <= 1'b1;
end
```

- WB 阶段

  在某个功能部件执行完毕后（即 `FUS` 里对应 `done` 为 1），而且写回不会发生 WAR 冒险时（即对应的 WAR 为 1），我们就会进行写回操作，同时要清空对应的 `FUS`，`RRS`。与此同时，我们会看其他的功能部件是否在等待当前这个功能部件的结果，如果是，我们就把对应的等待部件清除，并将操作数的 ready 设为 1.
  这里以 ALU 为例，其他功能部件类似。

```
else if (FUS[`FU_ALU][`FU_DONE] & ALU_WAR) begin
    FUS[`FU_ALU] <= 32'b0;
    RRS[FUS[`FU_ALU][`DST_H:`DST_L]] <= 3'b0;

    // ensure RAW
```

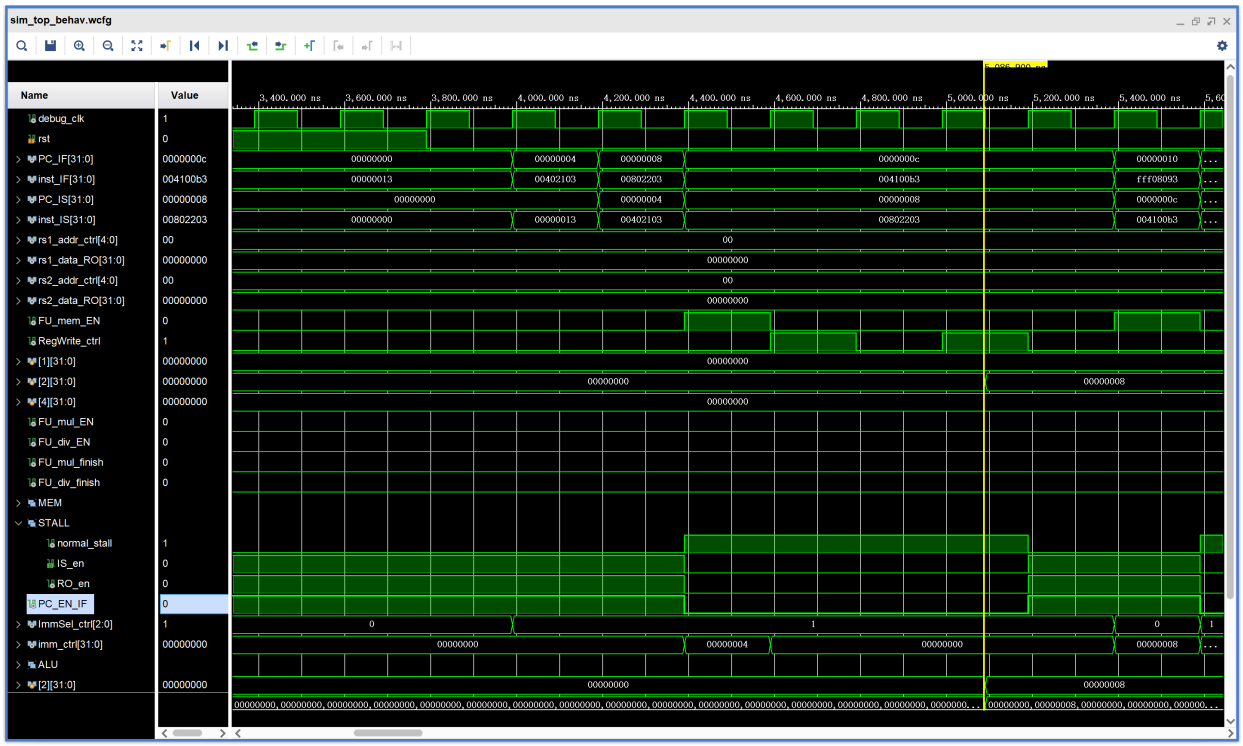
```
    if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_ALU) begin
        FUS[`FU_MEM][`FU1_H:`FU1_L] <= `FU_BLANK;
        FUS[`FU_MEM][`RDY1] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_ALU) begin
        FUS[`FU_MUL][`FU1_H:`FU1_L] <= `FU_BLANK;
        FUS[`FU_MUL][`RDY1] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_ALU) begin
        FUS[`FU_DIV][`FU1_H:`FU1_L] <= `FU_BLANK;
        FUS[`FU_DIV][`RDY1] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_ALU) begin
        FUS[`FU_JUMP][`FU1_H:`FU1_L] <= `FU_BLANK;
        FUS[`FU_JUMP][`RDY1] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_ALU) begin
        FUS[`FU_MEM][`FU2_H:`FU2_L] <= `FU_BLANK;
        FUS[`FU_MEM][`RDY2] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_ALU) begin
        FUS[`FU_MUL][`FU2_H:`FU2_L] <= `FU_BLANK;
        FUS[`FU_MUL][`RDY2] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_ALU) begin
        FUS[`FU_DIV][`FU2_H:`FU2_L] <= `FU_BLANK;
        FUS[`FU_DIV][`RDY2] <= 1'b1;
    end         //fill sth. here
    if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_ALU) begin
        FUS[`FU_JUMP][`FU2_H:`FU2_L] <= `FU_BLANK;
        FUS[`FU_JUMP][`RDY2] <= 1'b1;
    end         //fill sth. here
end
```

实验评估

仿真

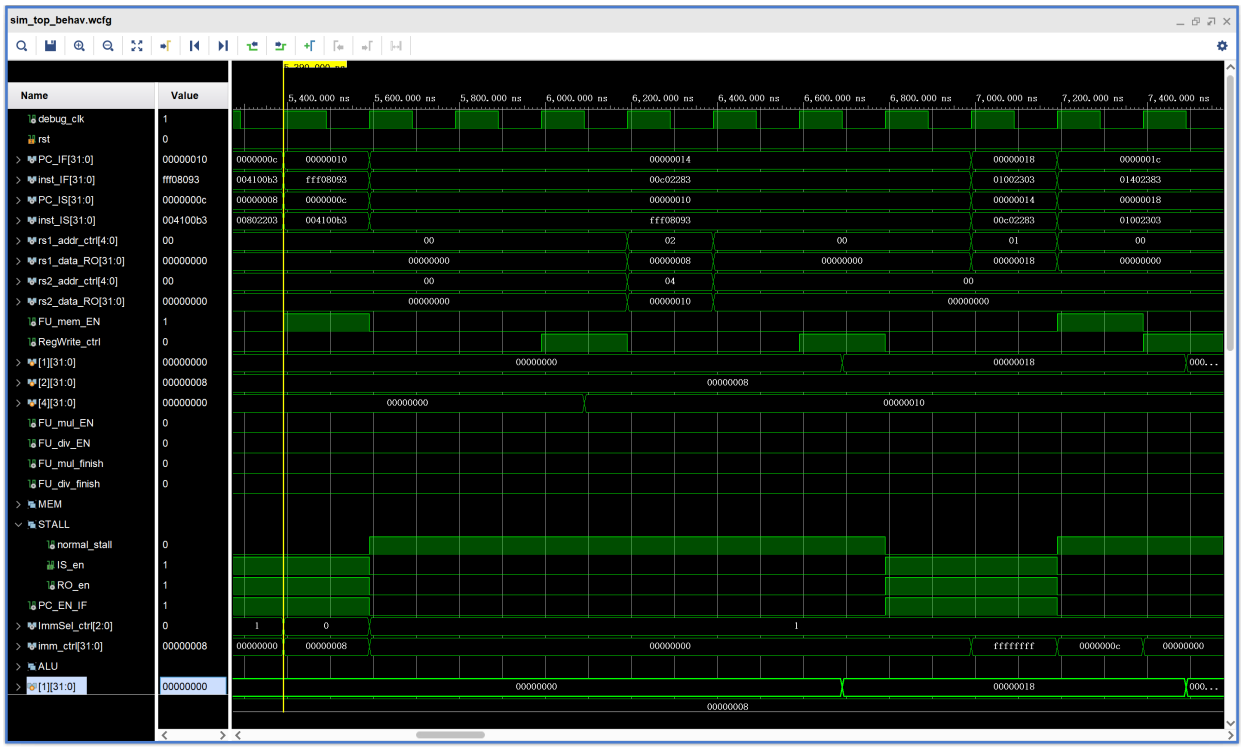
结构冲突：
4400ns 时，此时 `lw x2, 4(x0)` 处于 EX 阶段，`lw x4, 8(x0)` 准备发射，但是因为 MEM 功能单元正在使用，所以发生了结构冲突，需要等前一条指令写回，此时可以看到 `normal_stall` 为 1。四个周期后，即 5200ns 时，此时前一条指令已经完成了写回（`x2` 的值在 5100ns 时已经变为了 8），`IS_en`

为 1，则我们成功发射这条指令。



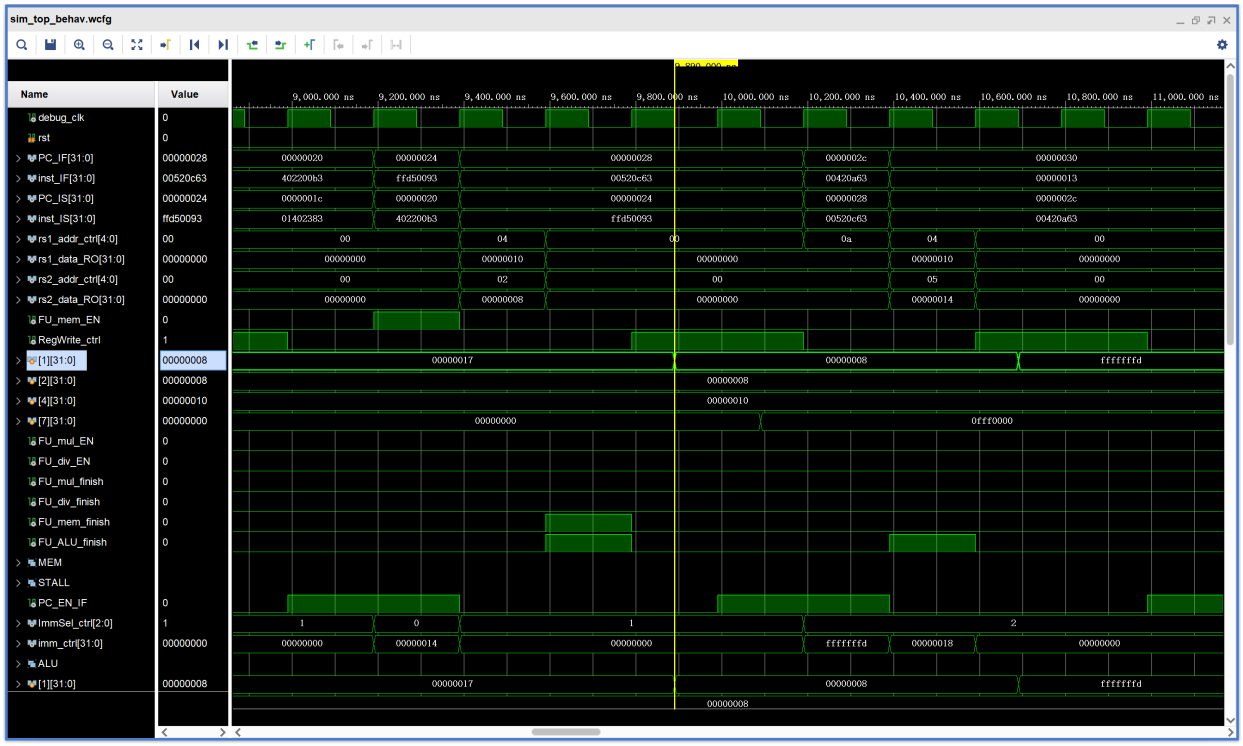
5400ns 时，0xc 这条指令 `add x1, x2, x4` 准备发射，但是此时 `x4` 的值是上一条 `load` 指令的结果，因此他不能立即读操作数。6100ns 时 `load` 指令的结果写回到 `x4` 寄存器，那么下个周期上升沿 0xc 就会进行 RO，再经过一个周期执行，再过半个周期在下降沿（6700ns）时写回，可以看到 `x1` 的值被写为了 0x18，说明 0xc 这条指令执行完毕。

那么下个周期 6800ns 时，WAW 冒险消失（0xc 和 0x10 的指令都要写寄存器 `x1` 因此要等 0xc 执行完毕），0x10 这条指令 `addi x1, x1, -1` 就可以发射了。



同时写：

9600ns 时可以看到 ALU 和 MEM 同时完成，但是我们不允许一个周期内同时往两个寄存器写入。因此这里我们先让 ALU 的结果写入，可以看到 9900ns 时 `x1` 的值变化；经过一个周期后 10100ns 时 `x7` 的值变化，说明 MEM 的结果写入。



同时这里也可以看出来乱序的发生：0x20 对应的 `sub x1, x4, x2`，0x1c 对应的是 `lw x7, 20(x0)`。理论上 0x1c 的指令先执行，应该先结束。但是在后面提交的时候，却是 `x1` 的值先被写入，说明 sub 指令先提交，这与发射顺序不一样，说明我们的确是在做乱序执行。

上板结果

上板结果已在验收时验证，这里不再重复。

思考题

1. Why doesn’t scoreboard use forwarding?

   这里有很多功能模块 FU，在计分板算法的基础上实现前递会带来非常复杂的结构。同时前递技术在这里的实际用途并不大，在实现了 double bump 的基础上，EX 结束后半拍（也就是时钟下降沿）寄存器的值就会被写回，而下一个周期的时钟上升沿，寄存器的值就会被读取。所以使用 forwarding 至多会为当前指令节约一个周期的时间，但是后面的指令不会因为这条指令 stall 就停止发射，此时数据冒险带来的 stall 的 penalty 远没有流水线中大。因此 forwarding 在 scoreboard 中只会带来略微的性能上升，这与其复杂的结构设计以及成本开销不匹配。权衡之下 scoreboard 没有使用 forwarding。

2. If we use a branch predictor, can we just let the CPU execute the predicted instructions? What if the prediction is wrong?

   我认为可以。但是分支预测如果预测失败会很难处理，因为指令在计分板中可能会乱序提交，如果预测错误我们很难恢复处理器状态，最好使用 Reorder Buffer 等技术来实现分支预测。

3. Point out where the out-of-order occurs based on simulation waveform.

见仿真分析部分，最后一张图可以看到乱序的发生：0x20 对应的 `sub x1, x4, x2`，0x1c 对应的是 `lw x7, 20(x0)`。理论上 0x1c 的指令先执行，应该先结束。但是在后面提交的时候，却是 `x1` 的值先被写入，说明 sub 指令先提交，这与发射顺序不一样，说明我们的确是在做乱序执行。

4. Analyze the pros and cons of scoreboard.

   优点：记分牌算法实现了乱序执行指令，解决了乱序执行的时候的冒险问题。
   缺点：记分牌在 WAW 会产生阻塞，后续指令无法发射。指令提交不是顺序的，可能对程序调试提出挑战。