

浙江大学实验报告

课程名称: 计算机体系结构 实验类型: 综合

实验项目名称: Topic 6. Dynamically Scheduled Pipelines using Scoreboarding

指导教师: 何水兵 完成时间: 2023.12.19

姓名	詹含蓓	学号	3210106333
同组学生姓名	居圣桐	学号	3210105812
分工信息	共同完成		

一、 实验目的和要求

- Understand the principle of pipelines that support multicycle operations.
- Understand the principle of Dynamic Scheduling With a Scoreboard.
- Master the design methods of pipelines that support multicycle operations.
- Master the design methods of Dynamically Scheduled Pipelines using Scoreboarding.
- Master verification methods of Dynamically Scheduled Pipelines using Scoreboarding.

要求:

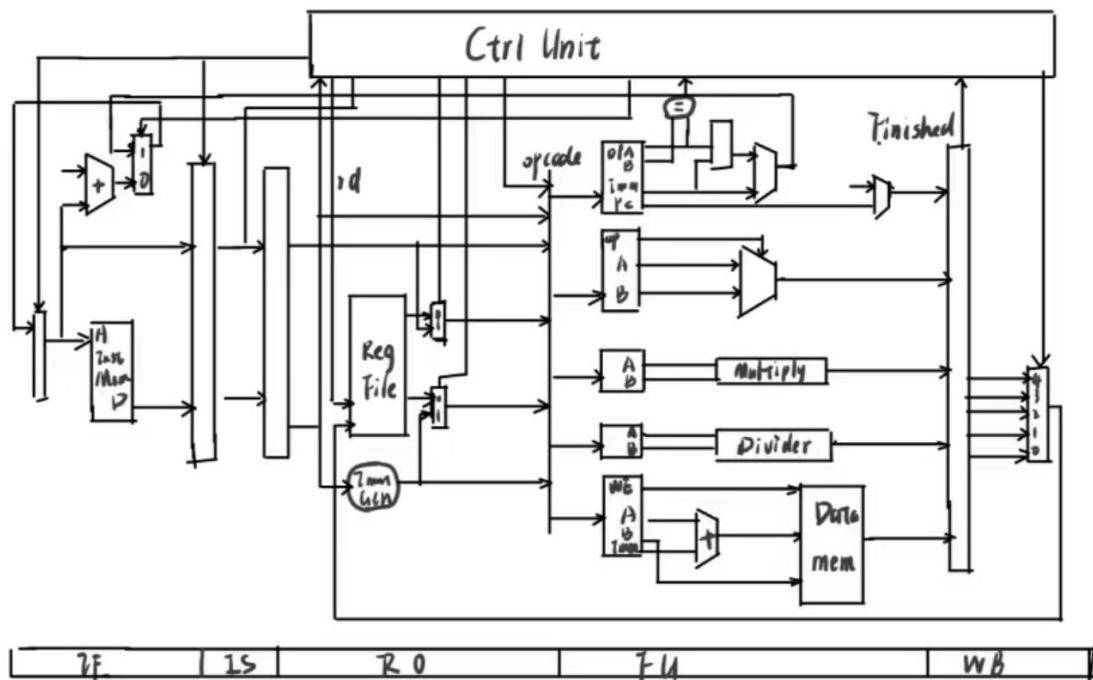
- Redesign the pipelines with IF/IS/RO/FU/WB stages and supporting multicycle operations.
- Design of a scoreboard and integrate it to CPU.
- Verify the Pipelined CPU with program and observe the execution of program.

二、 实验内容和原理

1. 本实验实现了基于 lab5 的 scoreboard 动态流水线的实现。流水线分为五级 IF/IS/RO/FU/WB。

(1) IF: 取指令, 将指令从存放指令的内存中取出。

- (2) IS: 发射, 对指令进行解码, 并观察各个功能部件的占用情况, 和寄存器堆的写情况。如果指令对应的功能部件空闲, 且指令要写的目标寄存器没有别的指令将要写, 则把指令信息存进部件寄存器, 同时改写记分牌, 把指令相关信息进入记分牌。
- (3) RO: 取数, 观察记分牌, 了解当前指令需要哪些寄存器的数值、该数值是否准备好、如果数值没有准备好的话数值正在哪个功能部件进行运算。
- (4) FU: 执行, 计算过程可能维持很多个周期, 在第一个计算周期结束时, 记分牌的 Rj、Rk 被修改。在得到计算结果的那个周期结束时, 结果会被存进结果寄存器。
- (5) WB: 写回, 如果别的指令不需要读当前计算结果即将写入的寄存器, 把结果写回到寄存器堆, 同时会清空记分牌中的信息。



2. 三种冒险

- (1) 结构冒险: 同时读写寄存器。在 scoreboard 中用寄存器状态表格中记录寄存器被哪个功能单元占用, 避免结构冒险。
- (2) 数据冒险
- ① RAW (read after write): 先写后读, 前一条指令的目的寄存器是下一条指令的操作数。如果第二条指令, 在第一条指令写之前, 第二条

指令先读，就会引起逻辑错误。

- ② WAW (write after write): 先写后写，前一条指令的目的寄存器和下一条指令的目的寄存器相同。如果第二条指令，在第一条指令写之前，第二条指令先写，就会引起逻辑错误。
 - 1) 在本实验当中用 `normal_stall` 来解决
 - ③ WAR (write after read): 先读后写，前一条指令的操作数是下一条指令的目的寄存器。如果第二条指令比第一条指令先写，则第一条指令就会读出错误的值。
- (3) 控制冒险：控制是否跳转取决于前面指令的结果。

三、实验过程和数据记录及结果分析

(一) 实验过程

1. **Normal_stall:**

在发生结构竞争或者 WAW 的时候，采用 `normal_stall` 来解决。这些冲突会在 IS 阶段和 RO 阶段发现。

```
assign IS_en = IS_flush | ~normal_stall & ~ctrl_stall;
assign RO_en = ~IS_flush & ~normal_stall & ~ctrl_stall;
```

- (1) 结构竞争：通过判断要使用的 FU 是否为 0 来判断是否到 fu 阶段。如果到达并且需要使用的 fu 中 busy 位是 1，则说明发生结构竞争，stall。
- (2) WAW：判断结果寄存器状态表格中目的寄存器是否被占用。

```
// normal stall: structural hazard or WAW
assign normal_stall = (use_FU == 0 ? 0 : FUS[use_FU][`BUSY]) | (|RRS[dst]);
```

2. **WAR**

这里为 1 表示不存在 WAR 冲突可以运行。在 WB 阶段进行应用。

- (1) JUMP_WAR：判断 JUMP 运算的 FU 状态寄存器当中目的寄存器和所有种类 FU 的源操作数寄存器不相同，或者其他 FU 的源操作数寄存器已经完成读数，则 JUMP_WAR 置于 1 表示不存在 WAR 冲突

```

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

```

(2) ALU_WAR: 判断 ALU 运算的 FU 状态寄存器当中目的寄存器和所有种类 FU 的源操作数寄存器不相同, 或者其他 FU 的源操作数寄存器已经完成读数, 则 ALU_WAR 置于 1 表示不存在 WAR 冲突

```

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
);

```

(3) MEM_WAR: 判断 ALU 运算的 FU 状态寄存器当中目的寄存器和所有种类 FU 的源操作数寄存器不相同, 或者其他 FU 的源操作数寄存器已经完成读数, 则 MEM_WAR 置于 1 表示不存在 WAR 冲突

```

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

```

(4) MUL_WAR: 判断 ALU 运算的 FU 状态寄存器当中目的寄存器和所有种类 FU 的源操作数寄存器不相同, 或者其他 FU 的源操作数寄存器已经完成读数, 则 MUL_WAR 置于 1 表示不存在 WAR 冲突

```

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

```

(5) DIV_WAR: 判断 ALU 运算的 FU 状态寄存器当中目的寄存器和所有种类 FU 的源操作数寄存器不相同, 或者其他 FU 的源操作数寄存器已经完成读数, 则 DIV_WAR 置于 1 表示不存在 WAR 冲突

```

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

```

3. RAW:

- (1) 在 WB 阶段判断是否有在等待该 FU 结果的源寄存器，如果有则把这个源寄存器 ready 设置为 1 表示已经准备好了来确保 RAW。

```

// WB
// JUMP
if (FUS[~FU_JUMP][`FU_DONE] & JUMP_WAR) begin
    FUS[~FU_JUMP] <= 32'b0;
    RRS[FUS[~FU_JUMP][`DST_H:`DST_L]] <= 3'b0;

    // ensure RAW
    if (FUS[~FU_ALU][`FU1_H:`FU1_L] == `FU_JUMP) FUS[~FU_ALU][`RDY1] <= 1'b1; //fill sth. here
    if (FUS[~FU_MEM][`FU1_H:`FU1_L] == `FU_JUMP) FUS[~FU_MEM][`RDY1] <= 1'b1; //fill sth. here
    if (FUS[~FU_MUL][`FU1_H:`FU1_L] == `FU_JUMP) FUS[~FU_MUL][`RDY1] <= 1'b1; //fill sth. here
    if (FUS[~FU_DIV][`FU1_H:`FU1_L] == `FU_JUMP) FUS[~FU_DIV][`RDY1] <= 1'b1; //fill sth. here

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

```

4. IS 阶段 scoreboard 更新:

- (1) 如果目标寄存器没有被占用，则将结果寄存器状态表格更新
(2) 将 FU 状态寄存器的各个位进行更新，标记为本条指令需要用的操作，源寄存器和目的寄存器，数据来源 fu，ready 位等，同时将 DONE 设置为 0 表示没有完成。

```

else begin
    // IS
    if (RO_en) begin
        // not busy, no WAW, write info to FUS and 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;
        IMM[use_FU] <= imm;
        PCR[use_FU] <= PC;
    end

```

5. RO 阶段 scoreboard 更新

- (1) 如果两个源寄存器的 ready 都为 1 则将其置于 0，表示使用完成

```
// RO
if (FUS[~FU_JUMP][`RDY1] & FUS[~FU_JUMP][`RDY2]) begin
    // JUMP
    FUS[~FU_JUMP][`RDY1] <= 1'b0;
    FUS[~FU_JUMP][`RDY2] <= 1'b0;
end
else if (FUS[~FU_ALU][`RDY1] & FUS[~FU_ALU][`RDY2]) begin    //fill sth. here.
    // ALU
    FUS[~FU_ALU][`RDY1] <= 1'b0;
    FUS[~FU_ALU][`RDY2] <= 1'b0;
end
else if (FUS[~FU_MEM][`RDY1] & FUS[~FU_MEM][`RDY2]) begin    //fill sth. here.
    // MEM
    FUS[~FU_MEM][`RDY1] <= 1'b0;
    FUS[~FU_MEM][`RDY2] <= 1'b0;
end
else if (FUS[~FU_MUL][`RDY1] & FUS[~FU_MUL][`RDY2]) begin    //fill sth. here.
    // MUL
    FUS[~FU_MUL][`RDY1] <= 1'b0;
    FUS[~FU_MUL][`RDY2] <= 1'b0;
end
else if (FUS[~FU_DIV][`RDY1] & FUS[~FU_DIV][`RDY2]) begin    //fill sth. here.
    // DIV
    FUS[~FU_DIV][`RDY1] <= 1'b0;
    FUS[~FU_DIV][`RDY2] <= 1'b0;
end
```

6. EX (FU) 阶段 scoreboard 更新

如果判断出来原来没有标记，则将 FU 表格中存的状态赋值给它。

```
// EX
if (~FUS[~FU_ALU][`FU_DONE]) FUS[~FU_ALU][`FU_DONE] <= ALU_done;
if (~FUS[~FU_MEM][`FU_DONE]) FUS[~FU_MEM][`FU_DONE] <= MEM_done;    //fill sth. here
if (~FUS[~FU_MUL][`FU_DONE]) FUS[~FU_MUL][`FU_DONE] <= MUL_done;
if (~FUS[~FU_DIV][`FU_DONE]) FUS[~FU_DIV][`FU_DONE] <= DIV_done;
if (~FUS[~FU_JUMP][`FU_DONE]) FUS[~FU_JUMP][`FU_DONE] <= JUMP_done;
```

7. WB 阶段 scoreboard 更新

- (1) 利用之前实现的 WAR 信号和 done 信号来判断是否可以写回。如果满足条件则清空 FU 状态，并且将结果寄存器表格中对应寄存器清空。
- (2) 遍历所有其他 FU，判断是否有在等待该 FU 结果的源寄存器，如果有则把这个源寄存器 ready 设置为 1 表示已经准备好了。

下面只对 JUMP 做代码展示，其他 FU 都相似。

```

// WB
// JUMP
if (FUS[~FU_JUMP][~FU_DONE] & JUMP_WAR) begin
    FUS[~FU_JUMP] <= 32'b0;
    RRS[FUS[~FU_JUMP][~DST_H:DST_L]] <= 3'b0;

    // ensure RAW
    if (FUS[~FU_ALU][~FU1_H:~FU1_L] == `FU_JUMP) FUS[~FU_ALU][`RDY1] <= 1'b1;           //fill sth. here
    if (FUS[~FU_MEM][~FU1_H:~FU1_L] == `FU_JUMP) FUS[~FU_MEM][`RDY1] <= 1'b1;           //fill sth. here
    if (FUS[~FU_MUL][~FU1_H:~FU1_L] == `FU_JUMP) FUS[~FU_MUL][`RDY1] <= 1'b1;           //fill sth. here
    if (FUS[~FU_DIV][~FU1_H:~FU1_L] == `FU_JUMP) FUS[~FU_DIV][`RDY1] <= 1'b1;           //fill sth. here

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

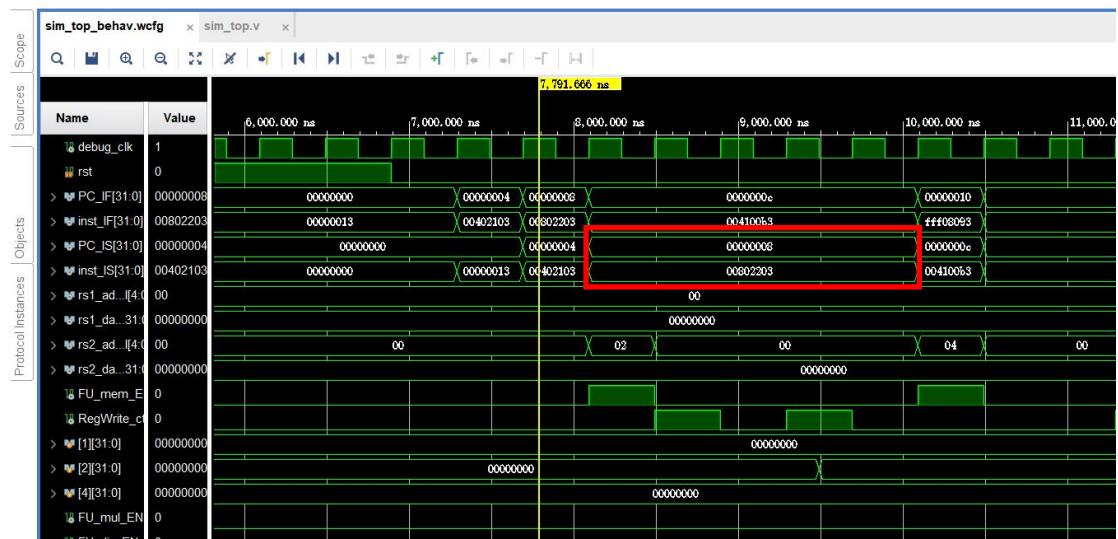
```

(二) 实验验证

一、仿真验证

- 前两条指令发生了结构冲突，两条指令都需要 MEM 的 FU。因此 pc 为 8 的指令在 IS 阶段 stall，需要等待 pc 为 4 的指令进行 RO+FU+WB = 1+2+1=4 个周期。在 WB 阶段寄存器的 ready 信号置于 1，MEM_en 信号也就被置于 1，并送入寄存器当中，在下一个周期让程序执行 RO。

1	00402103	4	lw x2, 4(x0)	
2	00802203	8	lw x4, 8(x0)	Structural Hazard



- PC 为 C 的指令顺序发射后会在 RO 的时候被卡住，因为发生了 RAW，寄存器 ready 信号为 0，RO 阶段无法发出 MEM_en 信号。此时需要等待 PC 为 8 的信号进行 FU+WB=2+1=3 个周期，在加上 RO_FU 中间的寄存器的 1 个周期，PC 为 C

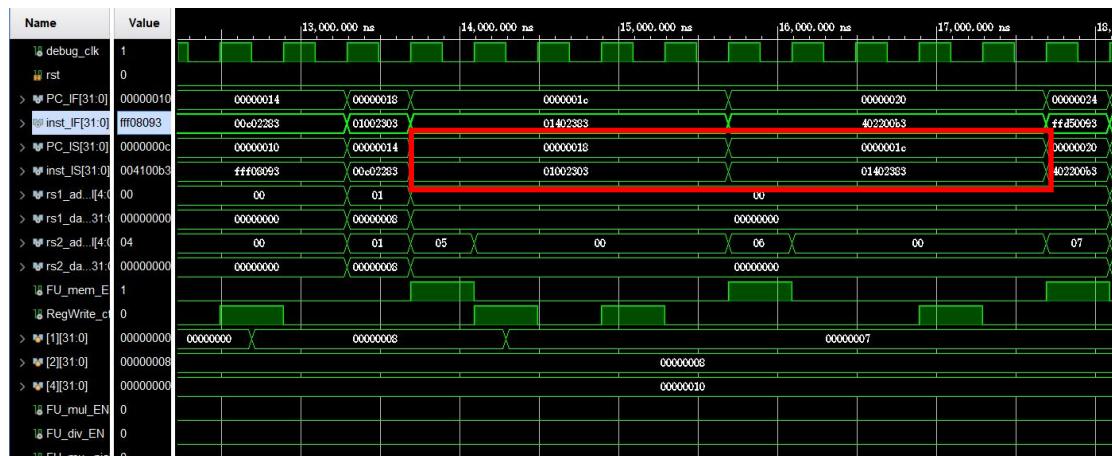
的指令的 RO 阶段一共有 4 个周期。之后指令继续执行，pc 为 10 的指令在 IS 阶段发现 WAW 后 stall，需要等待 pc 为 C 的指令进行 RO+FU+WB=4+1+1=6 个周期，再加上寄存器转存 1 个周期，因此 Pc 为 10 的指令的 IS 阶段一共进行了 7 个周期。

2	00802203	8	lw x4, 8(x0)	Structural Hazard
3	004100b3	C	add x1, x2, x4	
4	fff08093	10	addi x1, x1, -1	WAW



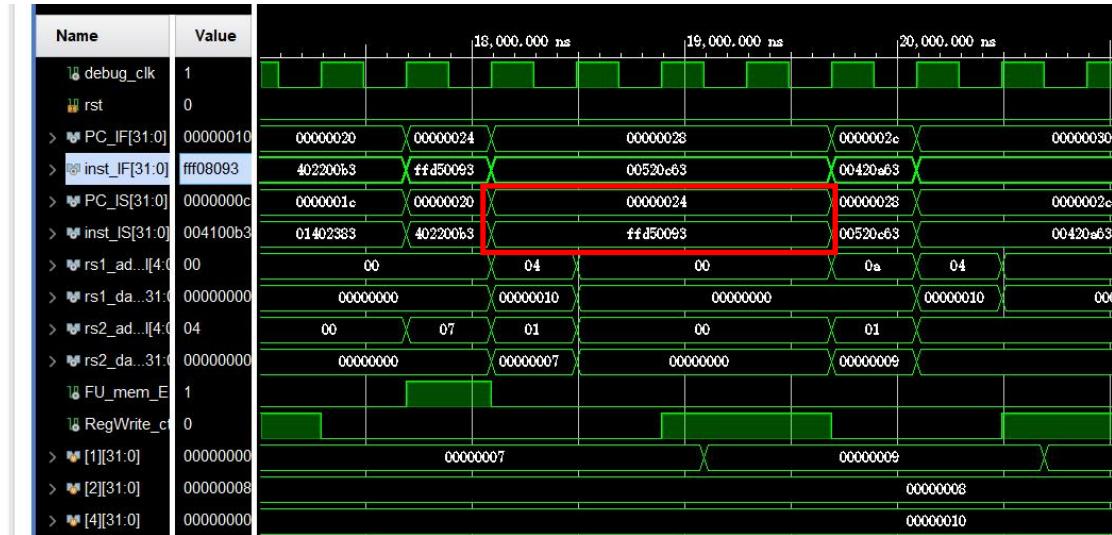
3. PC 为 18 和 1C 的指令同 PC 为 4 和 8 的指令，都会发生结构冲突，并且在 IS 的时候发生 5 个周期的 stall。

5	00c02283	14	lw x5, 12(x0)
6	01002303	18	lw x6, 16(x0)
7	01402383	1C	lw x7, 20(x0)



4. PC 为 24 的指令发生 WAW 冲突，在 IS 阶段进行 stall，需要等待 PC 为 20 的指令进行 RO+FU+WB=3 个周期，加上寄存器转化 1 个周期，一共需要 4 个周期。

8	402200b3	20	sub x1,x4,x2
9	ffd50093	24	addi x1,x10,-3



5. PC 为 28 的跳转指令跳转失败，2C 跳转成功。但是 28 到 2C 发生结构冲突，2C 的 IS 阶段需要等待 4 个周期，之后执行 RO 和 FU2 个周期之后将跳转 PC 填装到 IF 阶段。

10	00520c63	28	beq x4,x5,label0
11	00420a63	2C	beq x4,x4,label0
12	000000010	2C	-
16	000040b7	40	label0: lui x1,4
17	00c000ef	44	jal x1,12
18	000000010	40	addi x0,x0,0



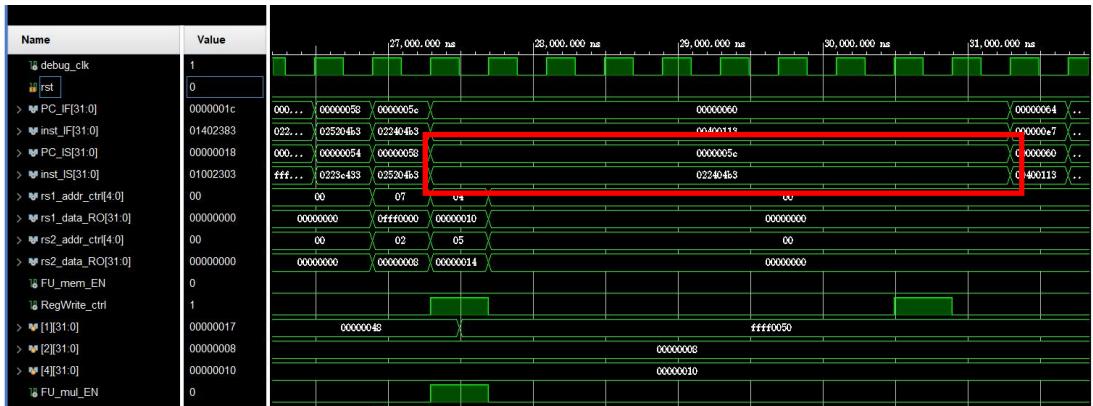
6. PC 为 44 的指令发生结构冲突，在 IS 阶段等待前一条指令 4 个周期，同时 flush 掉了 IF 阶段的指令。在 jal 执行了 RO，FU 两个周期之后将新的 PC 填装到 IF 阶段，也就是 PC 为 50 时的指令。



7. PC 为 5C 的指令和 PC 为 54 的指令发生了 RAW 冲突，并且和 PC 为 58 的指令发生了结构冲突。

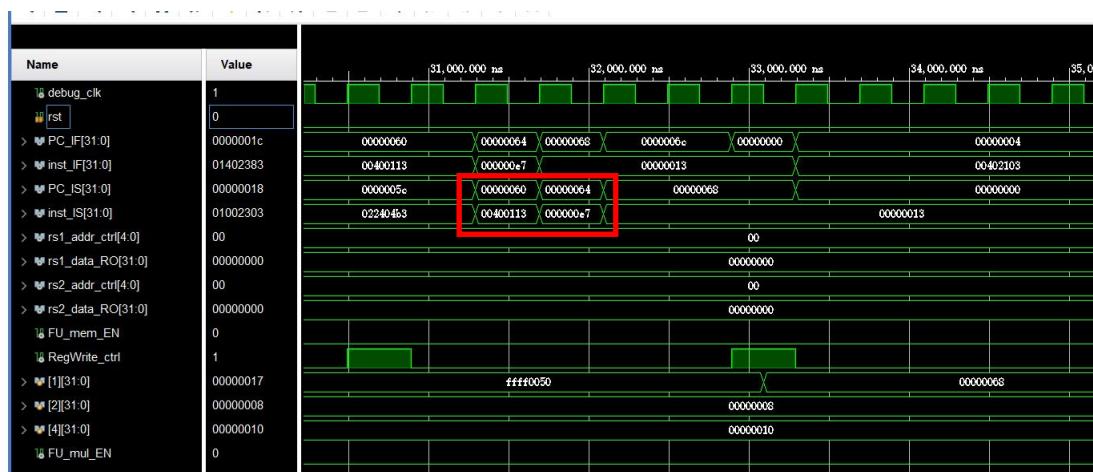
- (1) IS 阶段会被结构冲突出中断，需要等待前一条指令进行 RO+FU+WB=1+7+1 个周期，再加上寄存器转化 1 个周期一共 10 个周期。
- (2) RO 阶段由于 x2 寄存器要等到除法指令 WB 完成之后才可以进行，而除法指令的 FU+WB=39，其中 10 个周期在 5C 指令的 IS 阶段和等待结构冲突阶段完成，再加上 1 个寄存器转化周期，因此还需要 30 个周期。

21	0223c433	54		div x8, x7, x2
22	025204b3	58		mul x9, x4, x5
23	022404b3	5C		mul x9, x8, x2 St. Ha./RAW/WAW



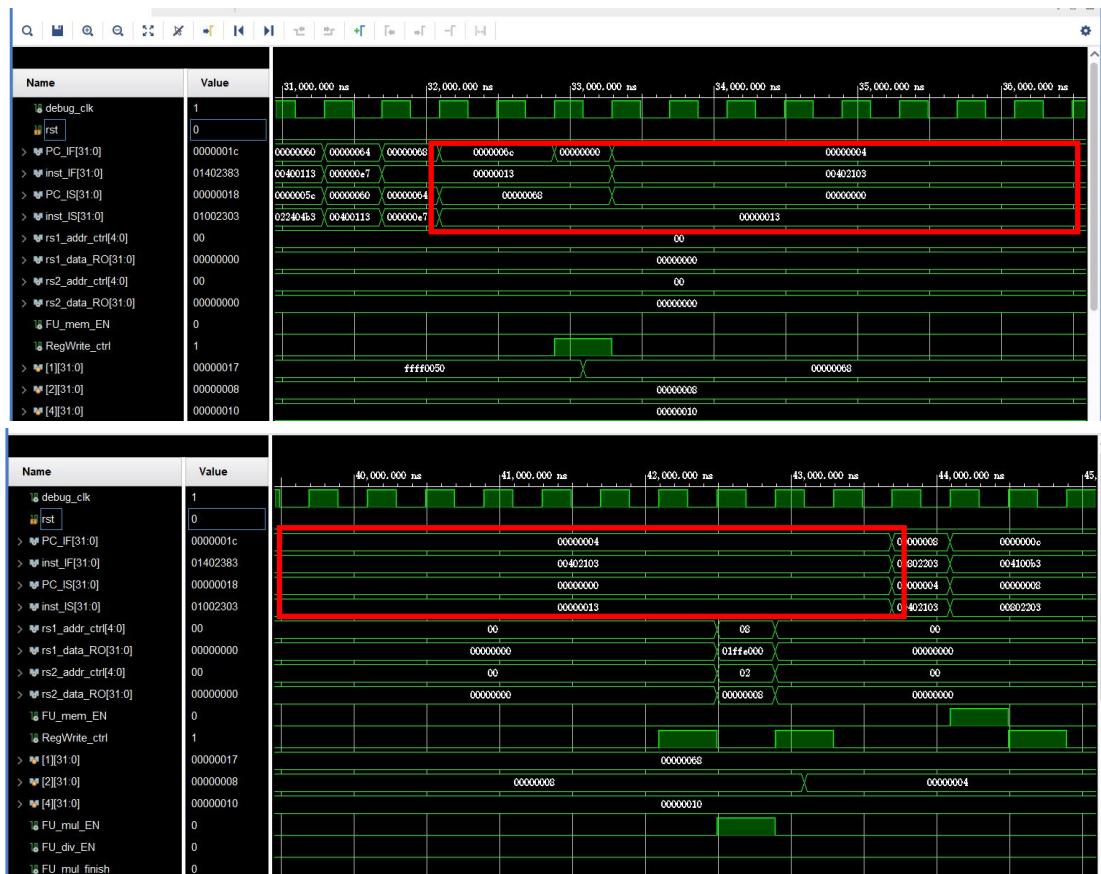
8. PC 为 60 的指令发生 WAR 冲突，在 WB 阶段才会发现，因此仿真上看是正常进行的。这个时候 addi 指令需要等待 mul 指令 WB 结束，而前者的 IS+RO+FU+WB=10+30+7+1=48，

23	022404b3	5C	mul x9, x8, x2	St. Ha./RAW/WAW
24	00400113	60	addi x2, x0, 4	WAR



9. PC 为 64 的指令跳转后进入 PC 为 0 的指令，这时候需要使用 ALU 的 FU，即发生了结构冲突。需要等待前面 PC 为 5C 的指令执行完之后才可以进行，所以等待了很多个周期。

24	00400113	60	addi x2, x0, 4	WAR
25	000000e7	64	jalr x1,0(x0)	



二、实验板验证

在仿真当中以及对代码和相应的周期数进行了很详尽的分析，因此实验板的照片就不做详细分析，主要为证明实验上板成功。



Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)

x0:zero 00000000	x01: ra 00000068	x02: sp 00000000	x03: gp 00000000
x04: tp 00000010	x05: t0 00000014	x06: t1 FFFF0000	x07: t2 0FFF0000
x08:fps8 01FFE000	x09: s1 0FFF0000	x10: a0 00000000	x11: a1 00000000
x12: a2 00000000	x13: a3 00000000	x14: a4 00000000	x15: a5 00000000
x16: a6 00000000	x17: a7 00000000	x18: s2 00000000	x19: s3 00000000
x20: s4 00000000	x21: s5 00000000	x22: s6 00000000	x23: s7 00000000
x24: s8 00000000	x25: s9 00000000	x26:s10 00000000	x27:s11 00000000
x28: t3 00000000	x29: t4 00000000	x30: t5 00000000	x31: t6 00000000

PC---IF 00000014 INST-IF 00C02203 PC---IS 00000010 INST-IS 0FFF0003
rs1addr 00000002 rs1data 00000000 rs2data 00000004 rs2addr 00000010
Imm-Sel 00000001 Imm-FU 00000000 ALUout 00000000 PC---EN 00000000
ALUE/Fi 00010000 ALUCtrl 00000001 ALU-RD 00000000 ALUB-RD 00000010
memE/Fi 00000000 mem-wri 00000000 u-b-h-w 00000000 memdata 00000010
mulE/Fi 00000000 mulres 0FFF0000 d1o/E/Fi 00000000 dires- 01FFF000
jopl/E/Fi 00000000 jmpCtrl 00000000 PC-jump 00000000 PC-wrtb 00000060
RegMrit 00000000 rd-addr 00000000 DaToReg 00000000 wt-data 00000000
CODE-00 00000000 CODE-01 00000000 CODE-02 00000000 CODE-03 00000000
CODE-04 00000000 CODE-05 00000000 CODE-06 00000000 CODE-07 00000000
CODE-08 00000000 CODE-09 00000000 CODE-0A 00000000 CODE-0B 00000000
CODE-0C 00000000 CODE-0D 00000000 CODE-0E 00000000 CODE-0F 00000000
CODE-10 00000000 CODE-11 00000000 CODE-12 00000000 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-1C 00000000 CODE-1D 00000000 CODE-1E 00000000 CODE-1F 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000

POWER VOL- VOL+ MENU CH- CH+ MODE

Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)

x0:zero 00000000	x01: ra 00000018	x02: sp 00000000	x03: gp 00000000
x04: tp 00000010	x05: t0 00000014	x06: t1 FFFF0000	x07: t2 0FFF0000
x08:fps8 01FFE000	x09: s1 0FFF0000	x10: a0 00000000	x11: a1 00000000
x12: a2 00000000	x13: a3 00000000	x14: a4 00000000	x15: a5 00000000
x16: a6 00000000	x17: a7 00000000	x18: s2 00000000	x19: s3 00000000
x20: s4 00000000	x21: s5 00000000	x22: s6 00000000	x23: s7 00000000
x24: s8 00000000	x25: s9 00000000	x26:s10 00000000	x27:s11 00000000
x28: t3 00000000	x29: t4 00000000	x30: t5 00000000	x31: t6 00000000

PC---IF 00000018 INST-IF 01002203 PC---IS 00000014 INST-IS 00C02203
rs1addr 00000001 rs1data 00000010 rs2data 00000000 rs2addr 00000000
Imm-Sel 00000001 Imm-FU FFFFFFFF ALUout 00000010 PC---EN 00000001
ALUE/Fi 00010000 ALUCtrl 00000001 ALU-RD 00000010 ALUB-RD FFFFFFFF
memE/Fi 00000000 mem-wri 00000000 u-b-h-w 00000000 memdata 00000010
mulE/Fi 00000000 mulres 0FFF0000 d1o/E/Fi 00000000 dires- 01FFF000
jopl/E/Fi 00000000 jmpCtrl 00000000 PC-jump 00000000 PC-wrtb 00000060
RegMrit 00000000 rd-addr 00000000 DaToReg 00000000 wt-data 00000010
CODE-00 00000000 CODE-01 00000000 CODE-02 00000000 CODE-03 00000000
CODE-04 00000000 CODE-05 00000000 CODE-06 00000000 CODE-07 00000000
CODE-08 00000000 CODE-09 00000000 CODE-0A 00000000 CODE-0B 00000000
CODE-0C 00000000 CODE-0D 00000000 CODE-0E 00000000 CODE-0F 00000000
CODE-10 00000000 CODE-11 00000000 CODE-12 00000000 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000

POWER VOL- VOL+ MENU CH- CH+ MODE

Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)

x0:zero 00000000	x01: ra 00000017	x02: sp 00000000	x03: gp 00000000
x04: tp 00000010	x05: t0 00000013	x06: t1 FFFF0000	x07: t2 0FFF0000
x08:fps8 01FFE000	x09: s1 0FFF0000	x10: a0 00000000	x11: a1 00000000
x12: a2 00000000	x13: a3 00000000	x14: a4 00000000	x15: a5 00000000
x16: a6 00000000	x17: a7 00000000	x18: s2 00000000	x19: s3 00000000
x20: s4 00000000	x21: s5 00000000	x22: s6 00000000	x23: s7 00000000
x24: s8 00000000	x25: s9 00000000	x26:s10 00000000	x27:s11 00000000
x28: t3 00000000	x29: t4 00000000	x30: t5 00000000	x31: t6 00000000

PC---IF 0000001C INST-IF 01402383 PC---IS 00000010 INST-IS 01002383
rs1addr 00000000 rs1data 00000000 rs2data 00000000 rs2addr 00000000
Imm-Sel 00000001 Imm-FU 00000000 ALUout 00000017 PC---EN 00000000
ALUE/Fi 00010000 ALUCtrl 00000000 ALU-RD 00000000 ALUB-RD 00000000
memE/Fi 00000001 mem-wri 00000000 u-b-h-w 00000000 memdata 00000014
mulE/Fi 00000000 mulres 0FFF0000 d1o/E/Fi 00000000 dires- 01FFF000
jopl/E/Fi 00000000 jmpCtrl 00000000 PC-jump 00000000 PC-wrtb 00000060
RegMrit 00000000 rd-addr 00000000 DaToReg 00000000 wt-data 00000012
CODE-00 00000000 CODE-01 00000000 CODE-02 00000000 CODE-03 00000000
CODE-04 00000000 CODE-05 00000000 CODE-06 00000000 CODE-07 00000000
CODE-08 00000000 CODE-09 00000000 CODE-0A 00000000 CODE-0B 00000000
CODE-0C 00000000 CODE-0D 00000000 CODE-0E 00000000 CODE-0F 00000000
CODE-10 00000000 CODE-11 00000000 CODE-12 00000000 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000

POWER VOL- VOL+ MENU CH- CH+ MODE



POWER VOL- VOL+ MENU CH- CH+ MODE



POWER VOL- VOL+ MENU CH- CH+ MODE



POWER VOL- VOL+ MENU CH- CH+ MODE



四、 讨论与心得

本次实验是在 lab5 的基础上实现了 scoreboard 动态流水线。实验由于 ppt 内容非常的少，所以最开始理解了很久到底要做什么事情。这次代码也比 lab5 更有难度。其中还有一部分内容其实也不是很确定。好在代码当中有很多的部分是相似的，只不过是 FU 之间的变化。但是在最后解读仿真的时候还是有一定的困难的。