# 实验 3 ～ 5 报告

**学号** 2019K8009929019
**姓名** 桂庭辉
**箱子号** 44

## 1 实验任务

本次专题分为三个实验项目，主要工作是在给出的实验框架下修订 bug（lab3），通过阻塞或前递技术处理数据与控制相关 (lab4、5)，通过这些内容对简单流水线 CPU 的设计、调试、优化方式加深理解。检验方式为实验框架给出的基于 trace 的比对验证，在行为仿真与上板验证两个级别验证结果。

## 2 实验设计

以下内容仅包括 myCPU 目录下的设计，即仅考察单发射静态 5 级流水线 CPU 的设计，考察设计版本为 lab5 完成后的最终设计。当然实验过程中简要理解实验框架的其他内容也有助于解决部分错误，如 lab4、5 中可能的 `st.w` 指令内存写数据错误。

### 2.1 总体设计思路

设计目标为基于 SRAM 的支持 LoongArch 精简版中部分指令的单发射静态 5 级流水线 CPU，根据需求将 CPU 划分为 5 个流水级 IF（取指）、ID（译码）、EXE（执行）、MEM（访存）、WB（访存），各流水级间控制与数据传递方式采用实验讲义 3.1.4.12 节介绍过的 valid-allowin 机制。根据流水级功能，在取指级 IF 与指令 RAM 通信，在执行级 EXE 与访存级 MEM 与数据 RAM 通信，在流水级添加 debug 相关接口。根据实验讲义、理论课内容等参考资料可给出下页结构设计简图[①]。各流水级设计在下文分模块介绍。

### 2.2 重要模块设计：IF_stage

#### 2.2.1 功能描述

5 级流水线的取指级，负责 PC 内容的维护以及与指令 SRAM 交互获取指令码。

#### 2.2.2 工作原理

实现上构造一个伪流水级 pre-IF，用于生成 `nextpc`，以 `nextpc` 为指令 RAM 读地址，PC 寄存器在 IF 级更新。同时该模块接收来自下一流水线 ID 级生成的跳转信号 `br_taken`，`br_target`，`nextpc` 的生成兼顾跳转信号的选择，以保证取回正确应执行的指令码。

[①]该简图忽略 debug 相关接口的连线，省略部分逻辑，省略内容在正文将会介绍，各流水级缓存内容简要介绍。

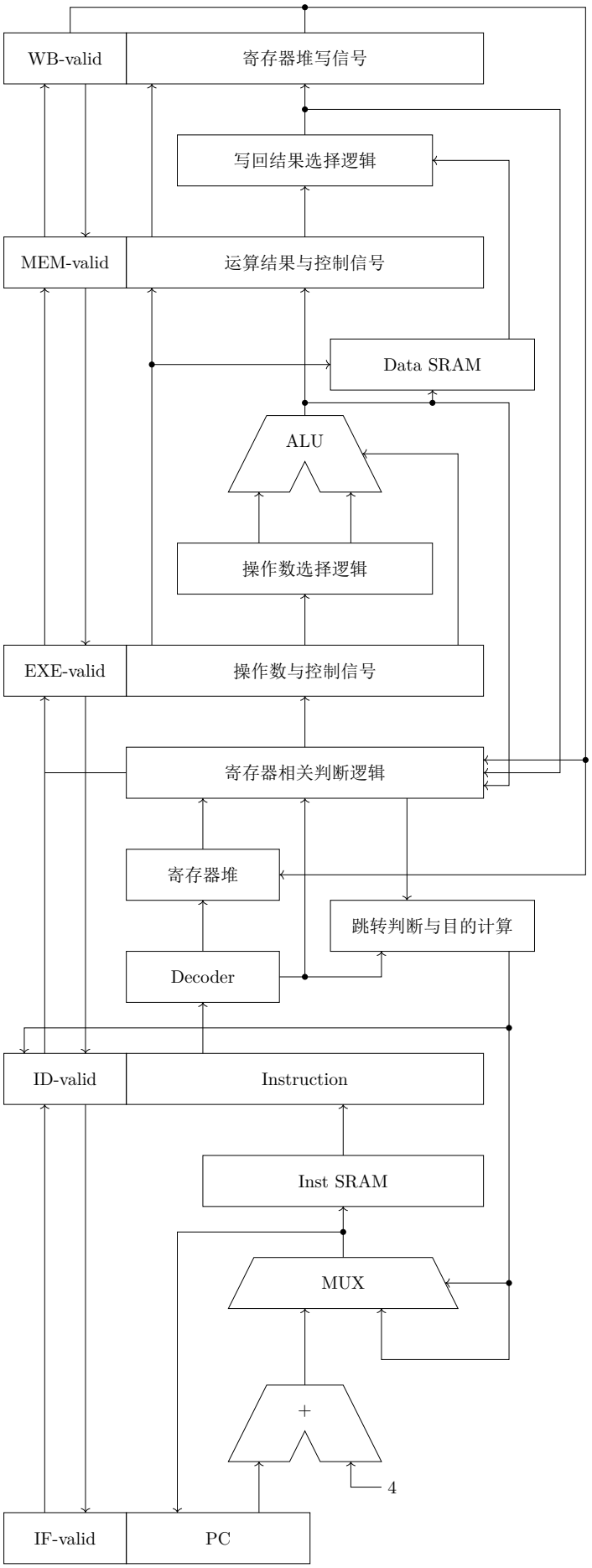


图 1: 简单单发射静态 5 级流水线 CPU 结构简图

### 2.2.3 接口定义

后文其他时序模块的时钟信号 `clk` 与复位信号 `reset` 与此处含义相同，不再赘述。

表 1: IF_stage 接口定义

| 名称 | 方向 | 位宽 | 描述 |
|---|---|---|---|
| `clk` | IN | 1 | 时钟信号 |
| `reset` | IN | 1 | 复位信号 |
| `ds_allowin` | IN | 1 | ID 级运行指令数据流入 |
| `br_bus` | IN | 33 | 来自 ID 级的跳转使能与跳转目的 |
| `fs_to_ds_valid` | OUT | 1 | IF 级向 ID 级传递数据有效 |
| `fs_to_ds_bus` | OUT | 64 | IF 级向 ID 级传递的数据，包括 PC 与指令码 |
| `inst_sram_en` | OUT | 1 | 指令 RAM 使能信号 |
| `inst_sram_wen` | OUT | 4 | 指令 RAM 写使能信号，置为全 0 |
| `inst_sram_addr` | OUT | 32 | 指令 RAM 读写地址，始终为 `nextpc` |
| `inst_sram_wdata` | OUT | 32 | 指令 RAM 写数据，置为全 0 |
| `inst_sram_rdata` | IN | 32 | 指令 RAM 读数据，即取到的指令码 |

## 2.3 重要模块设计：ID_stage

### 2.3.1 功能描述

5 级流水线的译码级，负责对 IF 级取到的指令码进行译码，根据译码结果访问寄存器堆[②]读取寄存器数据，同时生成各种控制信号传递给下游流水级。此外为了解决控制相关与数据相关，需要判断跳转条件、生成跳转目标、判断与下游流水级指令是否存在数据相关、处理前递数据，是本次实验主要修改的模块。

### 2.3.2 工作原理

根据目前选择的 20 条指令与 LoongArch 32 位指令编码方式进行译码，根据指令功能生成 `alu_op`、访存使能、寄存器写使能等控制信号与立即数等数据，生成寄存器堆读地址，并据此访问寄存器堆获取可能的源操作数。

控制相关的处理上，在 `br_taken` 拉高，即发生跳转时，将 `ds_valid` 寄存器置 0，取消已进入 IF 级的指令，阻塞 IF 级一拍重新获取正确的指令。

数据相关的处理是 lab4 与 lab5 主要内容，为实现阻塞与前递技术，我引入了从 EXE 级与 MEM 级到 ID 级的信号（信号的生成留到后文），WB 级到 ID 级已有写回通路，只需如结构图中引出一条参与数据相关判断即可。类比写回通路的数据，EXE 级与 MEM 级的前递数据同样包括寄存器写使能、写地址、写数据，此外 EXE 级还包括阻塞使能信号，处理当前设计下仅有的数据阻塞情况。

结合讲义内容，通过寄存器写使能是否有效、寄存器写地址是否为零、指令对源操作数实际依赖情况筛选排除掉实际不发生数据相关的情况，针对两个源操作数与三个可能的数据来源生成 6 个冲突信号，根

[②]寄存器堆的读写特性在 lab2 实验报告中已有过相关介绍，本次实验报告不再赘述。

据冲突信号在前递数据、寄存器堆读数据间选择，生成最终的源操作数 rj_value、rkd_value 传递给后续流水级。

### 2.3.3 接口定义

流水级间交互信号（即下表前六行）含义大体相同，后续三个流水级不再描述这些信号。

表 2: ID_stage 部分接口定义

| 名称 | 方向 | 位宽 | 描述 |
|---|---|---|---|
| es_allowin | In | 1 | 上游流水级 EXE 级允许数据流入 |
| ds_allowin | Out | 1 | 当前流水级 ID 级允许数据流入 |
| fs_to_ds_valid | In | 1 | 上游流水级 IF 级流入数据有效 |
| fs_to_ds_bus | In | 64 | 上游流水级流入数据 |
| ds_to_es_valid | Out | 1 | 向下游流水线 EXE 级流出数据有效 |
| ds_to_es_bus | Out | 150 | 向下游流水级流出数据 |
| br_to_bus | Out | 33 | 向 IF 级前递的跳转信号 |
| ws_to_rf_bus | In | 38 | 来自 WB 级的寄存器堆写信号 |
| es_fwd_blk_bus | In | 39 | 来自 EXE 级的前递数据 |
| ms_fwd_blk_bus | In | 38 | 来自 MEM 级的前递数据 |

## 2.4 重要模块设计：EXE_stage

### 2.4.1 功能描述

5 级流水线的译码级，负责对 ID 生成的各种数据进行运算，是运算指令的实际操作步骤、访存指令的地址生成步骤，主要实现通过 ALU 完成。同时由于同步 RAM 的读写特性，于其交互的使能信号在该流水级发生，访存指令的相关说明留待下一模块 MEM-stage 一并说明，本节主要关注运算指令与前递信号的生成。

### 2.4.2 工作原理

对于运算指令，实际运算过程由 ALU 完成，为此需要提供 ALU 操作码 alu_op 以及两个 ALU 操作数，这些数据已大半由 ID 级完成，EXE 级仅需对操作数做进一步选取。

ALU 的实现在此处简单说明，不再单列一节介绍。ALU 是纯组合逻辑的运算部件，功能在于根据 alu_op 指示的运算方式处理两个操作数，将运算结果输出到 alu_result 接口上。其实现主要分为三部分：(1) 对 alu_op 译码；(2) 对各种运算计算结果；(3) 根据第一部分的译码结果选择第二部分的计算结果，选择结果即为 alu_result。

前递信号的生成上与写回通路数据基本一致，都需用流水级 valid 信号屏蔽流水级无效时的前递数据。区别在于当前情形下唯一的阻塞需求，即 ld.w 位于 EXE 级且与 ID 级指令发生冲突时，需要阻塞 ID 一拍等待数据 RAM 返回读数据。故而需要再提供一位阻塞使能信号，该信号与 es_res_from_mem 等效，当然与寄存器写使能一样需要通过流水级 valid 信号屏蔽掉未定义状态。

### 2.4.3 接口定义

表 3: EXE_stage 部分接口定义

| 名称 | 方向 | 位宽 | 描述 |
|---|---|---|---|
| data_sram_en | Out | 1 | 数据 RAM 使能信号 |
| data_sram_wen | Out | 4 | 数据 RAM 写使能信号，store 指令置为全 1，其他全 0 |
| data_sram_addr | Out | 32 | 数据 RAM 访存地址 |
| data_sram_wdata | Out | 32 | 数据 RAM 写入数据 |
| es_fwd_blk_bus | Out | 39 | 向 ID 级的前递数据 |

## 2.5 重要模块设计：MEM_stage

### 2.5.1 功能描述与工作原理

实际设计中 MEM 级的逻辑如结构图所示，仅是[3]根据上游传递来的控制信号在 EXE 级计算结果与数据 RAM 读取结果间选择其一作为写回结果传递给下游写回级。此处简要介绍访存指令的处理方式，作为 EXE、MEM 两级与数据 RAM 交互设计的注解。

考虑作为同步 RAM 的数据 RAM 读写特性，在第一拍向其发出使能与地址信号后，等到第二拍才能从读端口获取到读数据。基于这样的特点，选择在 EXE 级向数据 RAM 发送读使能与地址，当前设计中 EXE 级与 MEM 级间不存在阻塞，故而下一拍该指令流入 MEM 级，正好获取到数据 RAM 发送回来的读数据。由于数据 RAM 仅有一个地址接口，store 指令使能、地址、数据的发送也随之迁移到 EXE 级而非 MEM 级完成。这样设计充分利用了同步 RAM 的读写特性，避免了在 MEM 级引入新的阻塞，保证了流水线效率。

### 2.5.2 接口定义

表 4: MEM_stage 部分接口定义

| 名称 | 方向 | 位宽 | 描述 |
|---|---|---|---|
| data_sram_rdata | In | 32 | 数据 RAM 发回的读数据 |
| ms_fwd_blk_bus | Out | 38 | 向 ID 级的前递数据 |

至此 5 级流水线只剩下了写回级，无论从结构图还是实际代码来看，写回级逻辑除流水线控制外，只涉及到将数据写回寄存器堆与将信息呈递给 debug 接口。其中前者在前面几个流水级中已有叙述，后者并非 CPU 设计的核心内容，仅是开发框架下的 debug 工具。故而此处不再对写回级单列一节进行叙述。

[3]当然还有前递给 ID 级的相关数据，但实现与 EXE 级相比仅是去掉了阻塞使能，EXE 级已指出阻塞使能为该级前递特有，故 MEM 级前递与 EXE 级前递其他部分一致，此处不做赘述。

# 3 实验过程

## 3.1 实验流水账

2021.09.12 13:30 ~ 2021.09.12 16:30 ： 完成 lab3 与简单的错误记录

2021.09.15 20:30 ~ 2021.09.05 22:10 ： 完成 lab4

2021.09.22 15:00 ~ 2021.09.22 16:00 ： 完成 lab5

2021.09.25 18:00 ~ 2021.09.26 00:00 ： 完成实验报告撰写与排版

## 3.2 错误记录

此处记录的错误主要来自 lab3 中实验框架设置的错误，仅有最后一个错误来自 lab4 实际设计过程。

### 3.2.1 错误 1：流水线无法正常启动

#### 3.2.1.1 错误现象

如图所示，在未对 lab3 实验框架做任何修改的情况下启动仿真，结果为 `debug` 信号始终全为 X，无法进行比对，仿真始终不停止。

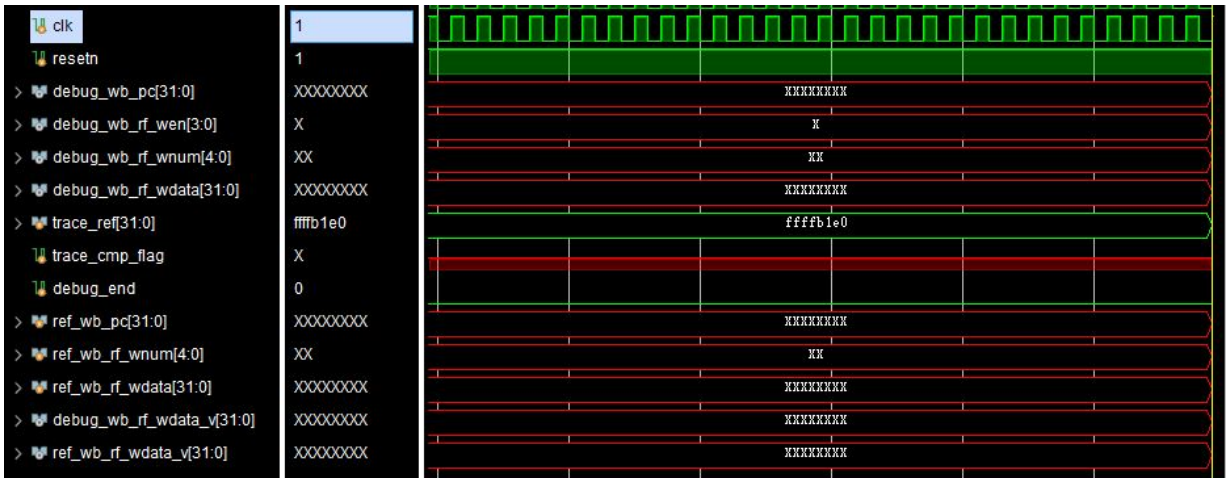


图 2: 错误 1：流水线无法正常启动

#### 3.2.1.2 分析定位过程

实际错误类型为信号为 X，故而从写回级的 debug 信号逻辑上溯，注意到译码级流水线控制逻辑的不足。

#### 3.2.1.3 错误原因

ID 级的流水级 `valid` 寄存器未赋值，导致流水线启动失败。

#### 3.2.1.4 修正效果

此处暂时学习其他流水级 `valid` 寄存器的逻辑增添如下代码：

```
always @ (posedge clk) begin
    if (reset) begin
        ds_valid <= 1'b0;
    end else if (ds_allowin) begin
```

```
            ds_valid <= fs_to_ds_valid;
        end
    end
```

当然，这样设计是不够的，这将引发另一个错误，但此处暂且按下不提。增添上述代码后流水级成功启动。

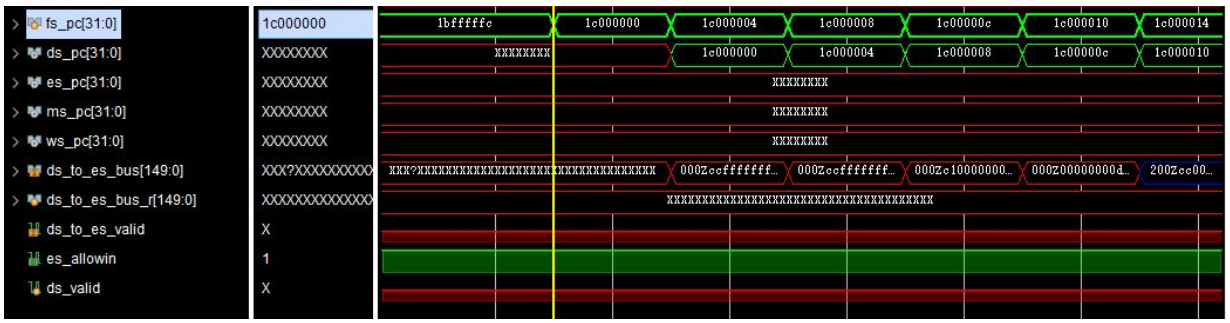


图 3: 错误 1 修订后流水级成功启动

#### 3.2.1.5 归纳总结

这样的错误在编写大型项目时或是易犯的，但事先对逻辑设计具备充分的认识与理解后应能尽量避免。一个或可能的方式是独立编写流水线控制与处理器数据通路设计两个板块，降低单次编写的工作量，避免注意力难以维持导致顾此失彼。

### 3.2.2 错误 2：br_bus 位宽

#### 3.2.2.1 错误现象

如图所示，myCPU 仅能顺序执行，无法跳转至 0x1c00070c 处。

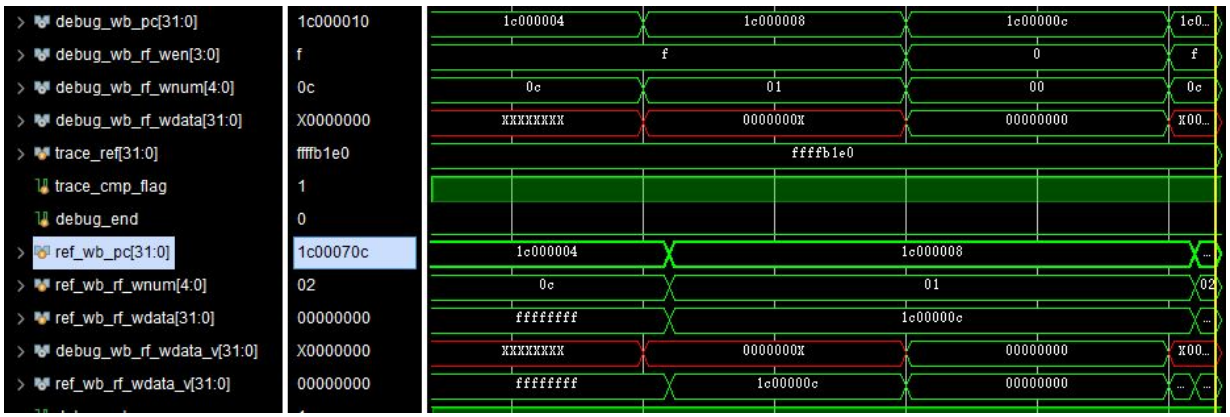


图 4: 无法发生跳转

#### 3.2.2.2 分析定位过程

考察 IF 级与 ID 级跳转相关信号的波形，发现 `br_taken` 其值无效。

#### 3.2.2.3 错误原因

`mycpu.h` 头文件中对 `BR_BUS_WD` 定义为 32 缺失一位。

#### 3.2.2.4 修正效果

修改该宏定义为 33，myCPU 能够跳转至 0x1c00070c 处。

#### 3.2.2.5 归纳总结

该问题本质是位宽未对齐。verilog 支持在 `assign` 或赋值过程时右值短于左值时对右值自动扩展，具体扩展方式取决于是否声明为 `signed`（默认为无符号数据），但左值短于右值的方式右值的高位将被舍弃。使用 verilator 作为 verilog linter 时上述不对齐情况均会给出 warning，根据信息修订错误即可。

### 3.2.3 错误 3：`load_op` 逻辑缺失

#### 3.2.3.1 错误现象

如图所示，写回级的寄存器写数据为 X。

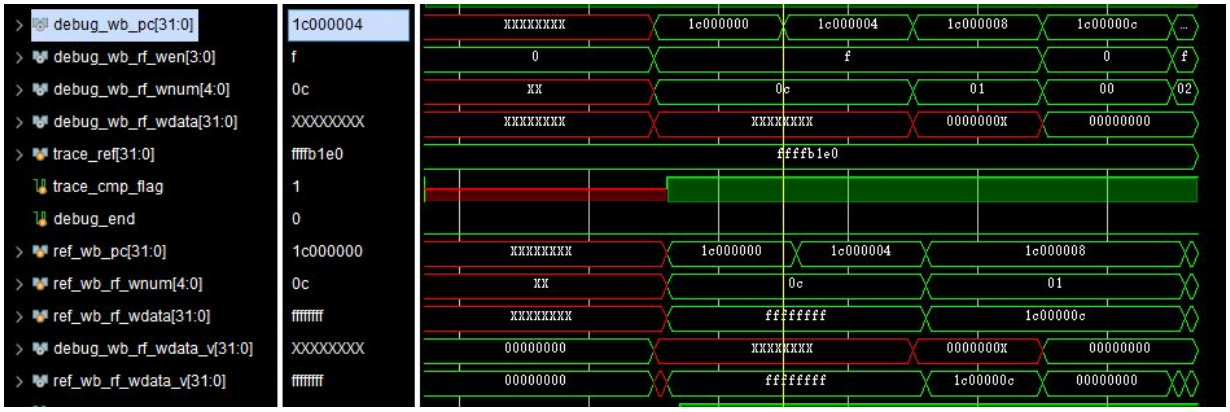


图 5: 写回数据为 X

#### 3.2.3.2 分析定位过程

根据 `debug_wb_rf_wdata` 的逻辑沿流水线上溯，观察相关信号波形注意到如下现象：

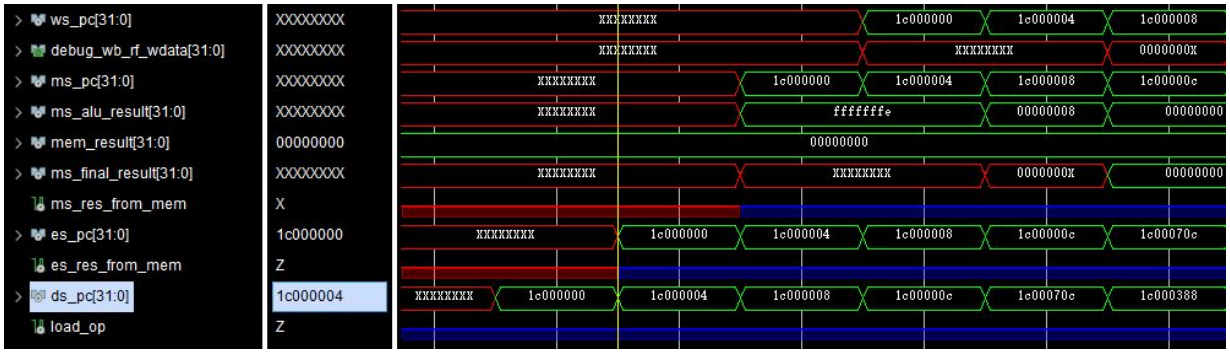


图 6: 自 `load_op` 以来控制信号为 Z

#### 3.2.3.3 错误原因

ID 级中 `load_op` 信号为被 `assign`，始终为 `Z`，最终导致 MEM 级最终选择写回数据时选择器出错。

#### 3.2.3.4 修正效果

根据后续逻辑考察 `load_op` 信号含义，可认识到该信号指示当前指令是否为 `load` 指令，即是否进行读内存操作。在 `ID_stage.v` 中添加如下代码后即可解决该问题：

```
assign load_op = inst_ld_w;
```

#### 3.2.3.5 归纳总结

该错误与错误 1 有类似之处。除错误 1 中给出的方式外，另一种或可行的方式是编写代码时每声明一个或一组信号或寄存器时，随即为其编写 `assign` 语句或 `always` 块完成逻辑设计。即便当下无法完成该处设计，在开发过程中也应当留下 `TODO: xxxxx` 之类的注释提示后续修订代码时该处逻辑未完成。

### 3.2.4 错误 4：ALU 端口

#### 3.2.4.1 错误现象

如图所示，0x1c000000 处指令写回数据错误。

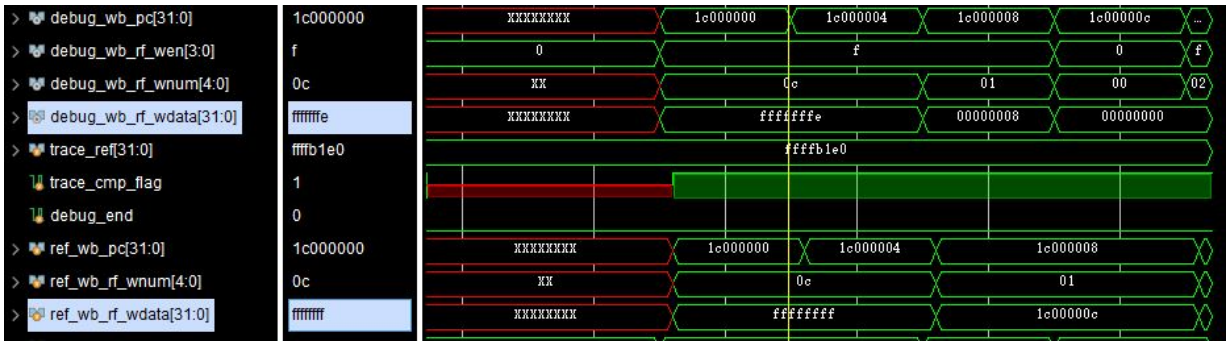


图 7: `addi.w` 指令写回数据错误

#### 3.2.4.2 分析定位过程

查阅反汇编文件可知该指令为 `addi.w`，考察其功能实现相关逻辑可注意到如下现象，即 EXE 级 ALU 操作数生成正确，但 ALU 中操作数错误。

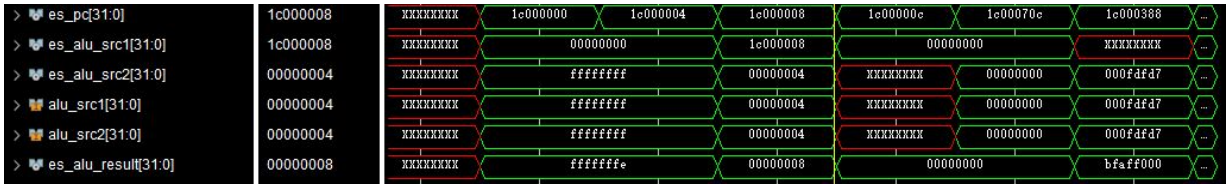


图 8: ALU 模块内外数据不一

#### 3.2.4.3 错误原因

EXE 级实例化 ALU 模块时，端口连线在两个操作数端口均连上信号 `es_alu_src2`。

#### 3.2.4.4 修正效果

修订模块例化如下即可解决该问题。

```
alu u_alu(
    .alu_op     (es_alu_op    ),
    .alu_src1   (es_alu_src1  ),
    .alu_src2   (es_alu_src2  ),
    .alu_result (es_alu_result)
);
```

#### 3.2.4.5 归纳总结

该类错误通常来自笔误，但在模块内外信号名已有如此良好对应关系的情况下出现这种错误难免有刻意之嫌。

### 3.2.5 错误 5：控制相关

#### 3.2.5.1 错误现象

如图所示，PC 在处理 0x1c00070c 处指令后并未顺序执行，而是跳转到 0x1c000388 处执行。

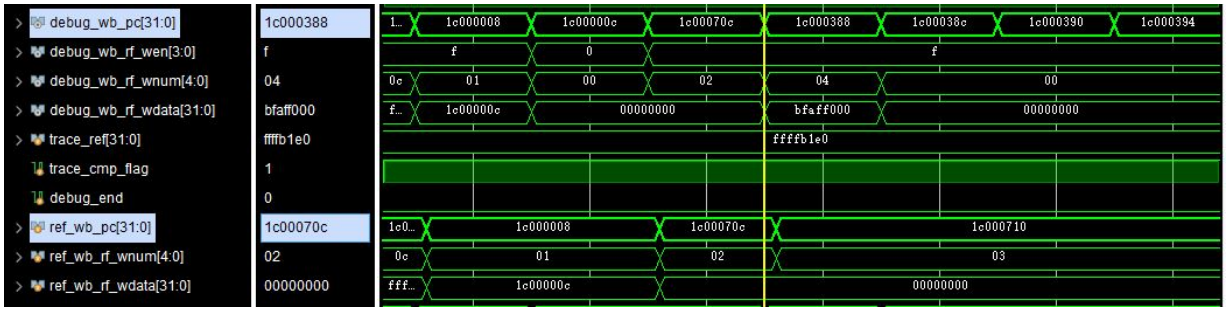


图 9: PC 未正确执行

#### 3.2.5.2 分析定位过程

查阅反汇编文件可知 0x1c00070c 与 0x1c000388 分别为前后两条绝对跳转指令 `bl` 与 `b` 的跳转目标，可以认识到 0x1c00070c 被取指时 `b` 已进入译码级，显然这是不该发生的。

#### 3.2.5.3 错误原因

此处的错误来自于错误 1 的修订方案。错误 1 的临时修正方式仅能使流水线流起来，但是对于流水线 CPU 控制相关的要求却不能满足。时序上产生 `br_taken` 信号时，跳转指令的下一条指令已经被取指，下一拍取到的才是正确的跳转目标，但下一拍错误的指令已经进入 ID，进而引发后续错误。

#### 3.2.5.4 修正效果

阅读实验讲义 4.5.2 小节，将 `ds_valid` 逻辑修改如下：

```
always @(posedge clk) begin
    if (reset) begin
        ds_valid <= 1'b0;
    end else if (br_taken) begin
        ds_valid <= 1'b0;
    end else if (ds_allowin) begin
        ds_valid <= fs_to_ds_valid;
    end
end
```

在 `br_taken` 拉高时阻塞一拍，避免跳转指令的下一条指令（此处即为错误指令）进入 ID 级。至此，由 `ds_valid` 引起的流水线逻辑乃至控制相关的一系列问题得以解决。

#### 3.2.5.5 归纳总结

该错误来自设计时对时序关系考虑的不足。解决方法唯有在编写代码前的设计阶段对各部件、各种情景下的时序关系有充分的认识，基于此给出正确的设计思路然后再编写代码。

### 3.2.6 错误 6：移位指令功能实现

#### 3.2.6.1 错误现象

类似错误 4，左移指令 `slli.w` 写回数据错误。

#### 3.2.6.2 分析定位过程

根据指令功能考察相关逻辑注意到如下现象，即 ALU 运算左移指令出错。

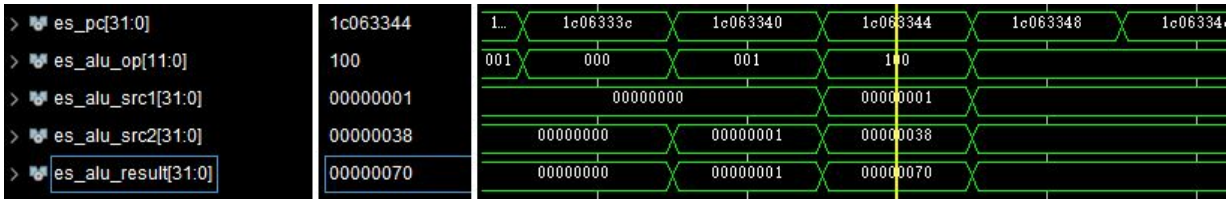


图 10: 左移实现出错

#### 3.2.6.3 错误原因

ALU 实现中左移运算逻辑中两操作数与指令含义相反，同时注意到其下两行右移部分的实现有类似问题与位宽问题，此处一并修正[4]。

#### 3.2.6.4 修正效果

修改该部分代码如下即可解决移位指令相关问题。

```
// SLL result
assign sll_result = alu_src1 << alu_src2[4:0];   //rj << i5

// SRL, SRA result
assign sr64_result = {{32{op_sra & alu_src1[31]}}, alu_src1[31:0]} >> alu_src2[4:0];
    //rj >> i5

assign sr_result   = sr64_result[31:0];
```

#### 3.2.6.5 归纳总结

该问题来自于对指令功能的不熟悉，虽然仍有刻意之嫌，但该错误提醒着我在后续实验中为 CPU 添加对更多指令的支持时一定要把握好指令的具体功能要求。

### 3.2.7 错误 7：波形停止

#### 3.2.7.1 错误现象

修订了上述错误后重启仿真，波形停止于 717985 ns，同时 Vivado 仿真部分进程卡死（体现在 `xsimk.exe` 进程即便在退出 Vivado 后仍在允许且占用大量资源）。

#### 3.2.7.2 分析定位过程

根据 lab2 的经验，该类问题主要来自于组合逻辑环。除了讲义与 lab2 中观察手动结束仿真的方式外，我实际 debug 过程中借助了此前提过的 verilator 作为 verilog linter 时提供的丰富 warning 信息定位错误。

#### 3.2.7.3 错误原因

错误来自于 ALU 中 `or_result` 的生成逻辑中或上了 `alu_result`，而 `alu_result` 的生成为在包含 `or_result` 的一众运算结果中的选择逻辑。

[4]实际实验过程中通过后续行为仿真波形注意到右移的错误，但这两处与已介绍的错误有高度类似，故而此处一并修改，下文不再赘述

#### 3.2.7.4 修正效果

将 or_result 的逻辑修改如下，至此已能通过 lab3 仿真与上板测试。

```
assign or_result  = alu_src1 | alu_src2;
```

#### 3.2.7.5 归纳总结

组合逻辑环的产生同 lab2 中总结的一样，来自于编写代码时对目标设计的认识不足，信号间逻辑关系把握不够。

### 3.2.8 错误 8：decoder 实现问题

该错误不影响通过当前指令要求下的测试，我是在实验过程中浏览实验框架与一些实现时注意到这一错误。

实验框架给出的 tools.v 中 $6 \to 64$ 位译码器的实现中循环条件为 $i < 63$，缺失了输入为全 1，输出为最高位拉高，低位全零的情形。

修改后的 generate for 部分如下：

```
genvar i;
generate for (i=0; i<64; i=i+1) begin : gen_for_dec_6_64
    assign out[i] = (in == i);
end endgenerate
```

### 3.2.9 错误 9：store 指令写数据错误

#### 3.2.9.1 错误现象

该错误来自 lab4，行为仿真过程中控制台报错但并未停止仿真。

#### 3.2.9.2 分析定位过程

认识到该错误来源的过程较为复杂，需要从激励测试文件 mycpu_tb.v 出发，简要阅读以 soc_lite_top.v 为顶层的数个源文件。结合错误发生时间点各流水级中停留的指令，最终认识到错误来自 store 指令 st.w 写数据错误。

#### 3.2.9.3 错误原因

根据讲义内容，最大程度降低阻塞带来的性能损失需要判断指令码中寄存器堆读地址字段冲突的指令是否将引发冲突的字段当作源操作数。故而我引入了 src_reg1、src_reg2 信号来判断当前 ID 级的指令是否将 rj、rk/rd 寄存器数据作为源操作数，进而筛除伪相关。然而初次编写时 src_reg2 中遗漏了 st.w 指令。进而导致在 st.w 与上一条指令在 rd 寄存器上发生数据冲突时未进行阻塞，导致写数据错误。

#### 3.2.9.4 修正效果

在 src_reg2 信号的生成逻辑中增加与 inst_st_w 取或的部分，解决上述问题通过 lab4 仿真与上板测试。

#### 3.2.9.5 归纳总结

该问题与错误 6 类似，都是来源于对指令功能的把握不熟悉，同样警示着我在后续实验中务必在设计前充分考虑指令功能。

# 4 实验总结

## 4.1 阻塞与前递的性能

lab4 我最终的仿真时长为 1322735 ns，lab5 引入前递技术后则为 604915 ns，前递技术带来的加速比为

$$\left(\frac{604915}{1322735}\right)^{-1} = 2.187$$

结合理论课的知识与练习，可以认识到完全用阻塞解决数据相关的流水线效率极低，乃至接近多周期处理器，引入前递技术可以大大减少所需的时钟周期数。但正如实验讲义所述，原本提高了主频的流水级切分工作在当前的前递设计下失去了部分效果，自 ALU 入口出发至最终的两个源操作数构成的长组合逻辑拖累了主频的提升。处理器设计中提升性能的技术还有很多，课程考虑到教学目标浅尝辄止，但我们仍可从实验过程中一窥理论课讲授的那些技术的意义所在。

## 4.2 一些与实验不太有关系的内容

虽然在错误 2 与错误 7 中我均提到我借助 verilator 提供的仿真信息轻松地定位 bug，但 verilator 也并非十全十美的 linter 工具，具体有以下三个例子用于说明其不足。

(1) 我配置 verilator 作为 VSCode 中的 verilog linter 始自计算机组成原理实验课程，在该课程的选做实验中我曾选做过 Cache 设计实验，当时课程对 Cache 访问地址的三个字段示例为 Tag、Set 与 Offset，沿用这一命名进行设计时 verilator 对 set 作为端口名时给出 warning 信息：Symbol matches C++ common word: 'set'。

(2) verilator 对文件名与模块名不匹配的情况支持并不友好。在本课程给出的实验框架中 `tools.v` 下设计了四个 `decoder` 模块，然而在 `ID_stage.v` 中例化译码器处 verilator 未能找到相应的 `decoder` 模块，给出 error 信息。

(3) 即使是在错误 7 中发现组合逻辑环的功能也并非那么准确。在为体系结构理论课作业编写基于 Booth 两位乘与华莱士树的 32 位补码乘法器时，我注意到对于如下情形 verilator 仍会给出可能存在组合逻辑环的 warning 信息。

```
wire [`WD - 1:0] sigs [`NUM - 1:0];


module_name m1(.in(other_sig), .out(sigs[0]));
module_name m2(.in(sigs[0]  ), .out(sigs[1]));
```

当然，上述情景用于最终验证的 FPGA 开发工具都是 Xilinx Vivado，我并未尝试利用 verilator 进行仿真检验上述不足是否会影响仿真验证的正常进行。

基于这些经验，我认识到 linter 始终是代码开发时的辅助工具，难有工具能在代码编写阶段提供尽善尽美的查错功能，Vivado 自带的 xvlog 作为 linter 时给出的信息丰富程度上远不如 verilator。事在人为，减少 verilog 代码中低级错误、设计错误的最终方式始终是开发者对开发目标具有充分且冷静的认识，在编码过程中慎之又慎。当然，不排除大佬的解决方式是自己造个符合自己口味且足够强大的轮子作为 linter 使用（笑）。