Lab 5 报告

马迪峰 2021K8009929033
饶嘉怡 2021K8009929005
范子墨 2021K8009929006
箱子号 5

一、实验任务（10%）

本次实验分为两部分任务，实践任务 10 为在已有单发射五级流水线的基础上添加算术逻辑类指令，包括 slti、sltui、andi、ori、xori、sll、srl、sra、pcaddu12i，和乘除运算指令，包括 mul.w、mulh.w、mulh.wu、div.w、mod.w、mod.wu、div.wu。实践任务 11 在任务 10 的基础上添加转移指令包括 blt、bge、bltu、bgcu，及访存指令包括 ld.b、ld.h、ld.bu、ld.hu、st.b、st.h。

该实验的正确性同样由基于 trace 比对的调试辅助手段进行验证，通过将 myCPU 中的 WB 阶段的 PC 值与 golden_trace 的 PC 值进行比对，如果不一致则立即报错并且停止仿真。在仿真正确后进行上板验证。

二、实验设计（40%）

（一）总体设计思路

对于算术逻辑运算指令，要实现的指令都尽量复用了原有指令的数据通路和控制信号。这部分指令需要在译码阶段进行对应数据通路的更改，在后续流水中只需复用原 alu_result 的数据通路鄂控制信号。

在乘除运算指令方面，乘法器采用了 booth 两位乘+华莱士树并且进行了二级流水划分，对于除法器使用的是恢复余数法，固定 33 拍算出结果。

对于转移指令，与原有的转移指令类似，只需添加跳转条件判断。

在访存指令方面，首先需要在译码阶段增加 load_op 和 store_op，并在发起访存请求前和后分别根据访存指令进行数据的修改使数据符合指令需求。

实验中为了使时序结果更好，尽量使用与或逻辑，而非三目运算符。

（二）重要模块 1 设计：算术逻辑运算类指令

1、工作原理

增添了 9 条算术逻辑运算类指令，所有指令均可以复用原有的数据通路和控制相关，只需对操作数进行选择，因此只需在译码阶段进行更改。

2、功能描述

具体来说，slti 可以复用 slt 的数据通路，sltui 可以复用 sltu 的数据通路，andi 可以复用 and 的数据通路，ori 可

以复用 or 的数据通路，xori 可以复用 xor 的数据通路，sll 可以复用 slli 的数据通路，srl 可以复用 srli 的数据通路，sra 可以复用 srai 的数据通路，pcaddul12i 可以复用 add 的数据通路。

而 slti、sltui、andi、ori 和 xori 的两个操作数分别为 rj 寄存器中值和立即数，其中 andi、ori 和 xori 新增了一种类型的操作数为 ui12，即零扩展立即数。

```
assign need_ui12  =  inst_andi | inst_ori | inst_xori;
```

因此对于立即数的选择需要增添对应 ui12 的判断。以及增添操作数 2 是否来自立即数的判断和操作数 1 是否来自 PC 的判断。

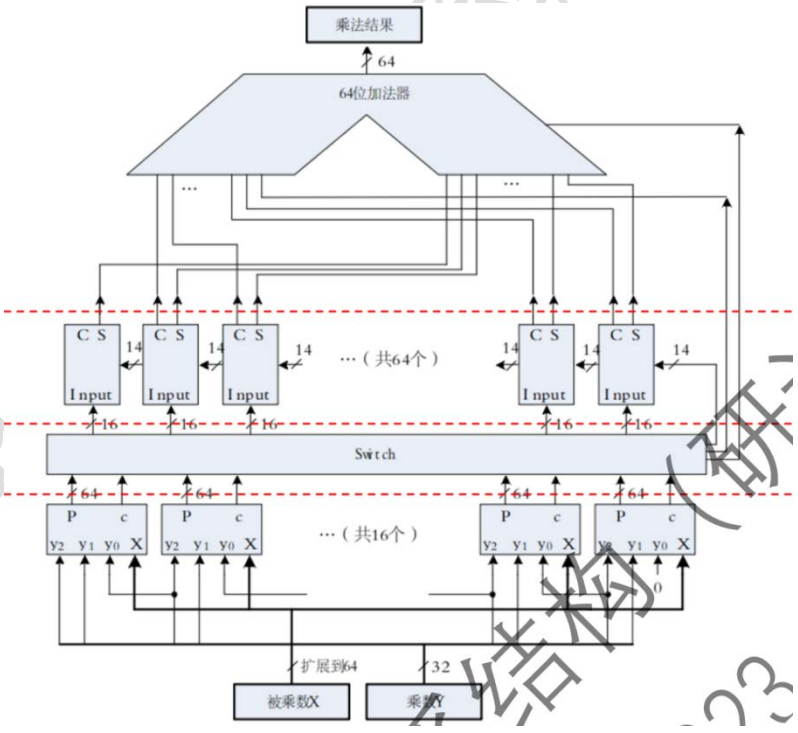
```
assign imm =                    src2_is_4 ? 32'h4                      :
                                need_si20 ? {i20[19:0], 12'b0}         :
  /*need_ui5 || need_si12*/     (need_ui5 || need_si12) ? {{20{i12[11]}}, i12[11:0]} :
                                {20'b0, i12[11:0]};
```

（三）重要模块 2 设计：乘法器

1、工作原理

（1） 乘法器

乘法器采用讲义推荐的两位 Booth 编码+华莱士树的设计原理，如下图所示。



图表 1：乘法器

本次实验需要实现的乘法指令有三条：

MUL.W：进行有符号乘法，将低 32 位作为结果。

MULH.W：进行有符号乘法，将高 32 位作为结果。

MULH.WU：进行无符号乘法，将高 32 位作为结果。

由此可知，该乘法器需要能够实现有符号和无符号乘法，因此可以将操作数符号拓展为 34 位，计算得到 68 位结果。这里之所以拓展成偶数位，是为了配合 booth 两位乘法。

该乘法器大致分为四部分：生成部分积、转置、华莱士树、64 位加法器。其中转置和 64 位加法器在代码设计阶段都好实现，主要难点在生成部分积和华莱士树。该实验中我们将生成部分积与华莱士树分别封装成了模块。

在生成部分积模块中，用输入的三位乘数判断要对被乘数进行的操作，逐位算出该部分积。

在华莱士树模块中，对 17 个 1bit 数，使用六层华莱士树将加数减少为 2 个 1bit 数相加，得到 17 个 1bit 数相加的结果。

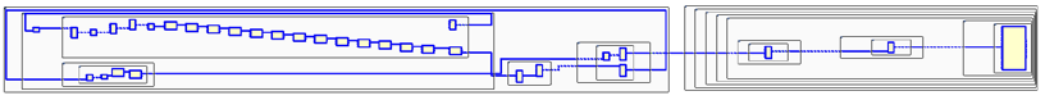
只需在 booth 乘法顶层模块中例化 17 个生成部分积模块，算出 17 个部分积；将 17 个 68bits 数转置成 68 个 17bits 数；再例化 68 个华莱士树模块，将每个 17bits 数送入一个华莱士树，算出 17 个 1bit 数的和；最后，将计算结果送入 64 位加法器即可。

（2） 切分流水

该模块最难的部分在于如何切分流水使得时序更好，切成几拍和在哪里切是最主要的问题。我组最后采用的方案是：

1. 切成两拍，在 MEM 级拿到结果，不阻塞，只在冲突时阻塞一拍；
2. 在 64 位加法器之后切分流水，即第一拍算出乘法结果，将结果存入寄存器送到第二拍。

之所以选择在 64 位加法器之后切分流水，是因为我们的 slack 违约最严重的关键路径，是“乘法器的结果前递回 ID 级的跳转指令，计算跳转地址之后送入指令 RAM”，所以如果把切割流水级改在乘法器的结尾，时序会好很多。



图表 2：路径

尝试最佳的切分流水级的过程会写在实验过程中，在此不赘述。

（3） 乘法模块外部处理

添加了两级流水乘法器之后的 CPU 大致如下图所示。之所以选择将乘法器直接例化在 EXE 模块，而不是例化在 alu 模块内部，就是因为乘法器是两级流水的，而 alu 内部是组合逻辑，如果将乘法器模块例化在 alu 模块里，需要给 alu 也切分出两级流水，而这是没有必要的。

在 EXE 模块里例化乘法器之后，需要在乘法器模块外部添加的操作有：将 alu 的源操作数位拓展至 34 位之后送入乘法器作为源操作数，将乘法器模块计算得到的 68 位结果送入 MEM 级，在 MEM 级中根据从 ID 级传来的 alu_op_reg 选择高 32 位或低 32 位作为乘法结果。将 final_result 改为 alu、乘法器、数据 RAM 三者结果的三选一逻辑。

由于乘法的结果在 MEM 级才能得到，因此乘法的前递操作与 ld 指令相同。（但我的乘法器在 EXE 其实就得到结果了，感觉是不是能不阻塞，直接在 EXE 拿到 z_temp 前递）如果在 EXE 时与 ID 级的指令冲突，就将 ID 级的指令阻塞一拍，之后再将 MEM 级的结果前递回 ID 级。

2、接口定义

表格 1：乘法器接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
x	IN	34	符号拓展的 34 位被乘数
y	IN	34	符号拓展的 34 位乘数
z	OUT	68	68 位乘法结果

3、功能描述

输入信号来自 EXE 级，在 EXE 和 MEM 两拍计算，输出信号于 MEM 级被获取。如果乘法指令处于 EXE 阶段时与当前处于 ID 阶段的指令冲突，将 ID 阶段的指令阻塞一拍，之后前递。

（四）重要模块 3 设计： 除法器

1、工作原理

除法器模块设计采用的是恢复余数法的思路，需要完整的 33 拍才能计算出结果

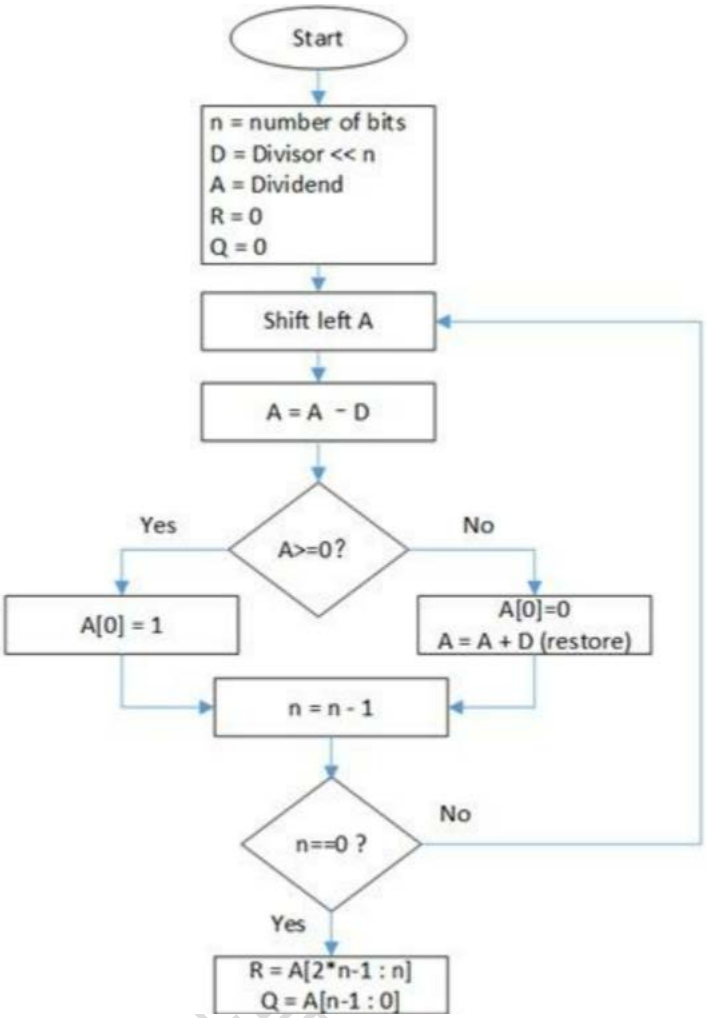
2、接口定义

表格 2：除法器接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
resetn	IN	1	复位信号
div_en	IN	1	除法器使能信号
sign	IN	1	除法器操作类型，为 1 代表有符号除法否则为无符号
divisor	OUT	32	被除数
dividend	OUT	32	除数
result	OUT	64	运算结果，高 32 位为商，底 32 位为余数
complete	OUT	1	除法器完成信号

3、功能描述

恢复余数法的流程图如下：



图表 3

恢复余数法，每次都需要判断结果寄存器是否小于 0，再做是否恢复 A 寄存器的操作；最小需要 33 个时钟才能计算商和余数；利用 cunter 信号进行计数，当除法器开始工作且操作数有效时，计数开始，计数达到 33 时代表除法运算结束，complete 信号拉高向外输出到 ALU 模块表明除法器已算得结果，不必再阻塞可以输出运算结果。

除法器设计中遇到的问题主要在于结果的保持和在 alu 中的例化，因为我们的设计中并未将乘法器例化至 ALU 中，所以 ALU 发生阻塞时只会是除法器运算导致，需要在 alu 模块中中引入 alu_flag 信号标明其运算是否完成。

```
assign alu_flag = ~resetn | div_finish & div_en /*| mul_finish & mul_en*/ | ~div_en /*& ~mul_en*/;
```

图表 4

除法器的其余数据通路与其他在 alu 完成的运算基本一致，不需要做其他改变。

（五）重要模块 4 设计：转移类指令

1、工作原理

由于本次跳转指令所用的数据通路于之前的一致，因此只需要增加针对于这次新增指令的跳转条件判断，而不需要更改寄存器数据是否有效等部分。以及本组采用直接在译码阶段增加加法器进行判断，而不是复用 alu。

2、功能描述

```
assign {cout, cout_test} = {1'b0, rj_value} + {1'b0, ~rkd_value} + 1'b1;
assign rj_lt_rd = rj_value[31] ^ ~rkd_value[31] ^ cout;
assign rj_ltu_rd = ~cou.t;
```

首先对两个需要比较的操作数进行减法，具体来说是 rj_value 加上 rkd_value 的补码，而后将减法结果赋值为 cout_test，而进位赋给 cout。对于有符号数的两个数，利用补码计算的进位机制，当 rj_value<rkd_value 时不会发生进位。对于无符号数，如果有借位 cout 为 0，即小于，无借位 cout 为 1，即大于等于。

而后对是否跳转的信号进行补充。

```
assign br_taken = (   inst_beq  &&  rj_eq_rd
                   || inst_bne  && !rj_eq_rd
                   || inst_blt  && rj_lt_rd
                   || inst_bge  && !rj_lt_rd
                   || inst_bltu && rj_ltu_rd
                   || inst_bgeu && !rj_ltu_rd
                   || inst_jirl
                   || inst_bl
                   || inst_b
                  ) && ds_valid && ds_ready_go;
```

（六）重要模块 5 设计：访存类指令

1、工作原理

对于 load 型指令，定义 load_op，由于需要存到寄存器中的指令需要在 MEM 阶段才能读出，因此 load_op 需要从 ID 阶段译出后一直传递到 MEM 阶段后进行赋值。而对于 store 指令，定义 store_op，虽然在 ID 阶段就已经可以进行 store 数据的选择，但考虑到组合逻辑路径可能过长导致时序不好，因此放在 EXE 阶段进行赋值。

2、功能描述

load 型一共五种指令，因此 load_op 位宽为 5 位并分别赋值。

```
assign load_op[0] = inst_ld_b;
assign load_op[1] = inst_ld_h;
assign load_op[2] = inst_ld_w;
assign load_op[3] = inst_ld_bu;
assign load_op[4] = inst_ld_hu;
```

而后传递到 MEM 阶段后，将五种 load 指令分为三种，lb_data，lh_data 和 mem_result，对于在结果中填入不同长度的 mem_result，其中前两种通过对 zero_ext 的赋值来进行零扩展或者符号扩展的判断，可以节省资源。

```
assign zero_ext = ~(mem_load_op[3] | mem_load_op[4]);
assign lb_data = {32{ld_sel[0]}} & {{24{mem_result[7] & zero_ext}}, mem_result[7:0]}
```

```
            | {32{ld_sel[1]}} & {{24{mem_result[15] & zero_ext}}, mem_result[15:8]}
            | {32{ld_sel[2]}} & {{24{mem_result[23] & zero_ext}}, mem_result[23:16]}
            | {32{ld_sel[3]}} & {{24{mem_result[31] & zero_ext}}, mem_result[31:24]};

assign lh_data = {32{ld_sel[0]}} & {{16{mem_result[15] & zero_ext}}, mem_result[15:0]}
               | {32{ld_sel[2]}} & {{16{mem_result[31] & zero_ext}}, mem_result[31:16]};
```

最后对写入 ld_data 的数据进行选择。

```
assign ld_data = {32{mem_load_op[0] | mem_load_op[3]}} & lb_data
               | {32{mem_load_op[1] | mem_load_op[4]}} & lh_data
               | {32{mem_load_op[2]}} & mem_result;
```

而 store 指令共三种指令，因此 store_op 为三位并分别赋值。

```
assign store_op[0] = inst_st_b;
assign store_op[1] = inst_st_h;
assign store_op[2] = inst_st_w;
```

在 EXE 阶段，首先对 alu 计算出的地址后两位对要写入的字节进行多路选择，而后针对不同的 store 指令进行赋值。再将要写入的数据进行扩充到 32 位写入数据种，方便选择时可以直接选到所需要的部分。

```
assign st_strb = {4{store_op_reg[0]}} &  st_sel
               | {4{store_op_reg[1]}} & (st_sel[0] ? 4'b0011 : 4'b1100)
               | {4{store_op_reg[2]}} &  4'b1111;

assign st_data = {32{store_op_reg[0]}} & {4{rkd_value_reg[7:0]}}
               | {32{store_op_reg[1]}} & {2{rkd_value_reg[15:0]}}
               | {32{store_op_reg[2]}} & rkd_value_reg;
```

三、实验过程（50%）

（一）实验流水账

exp10 实验流水账

- 2023/10/13 19:00 开始 exp10 实验，确定分工
  - 马迪峰负责除法模块和数据通路的设计
  - 饶嘉怡负责乘法模块和数据通路的设计
  - 范子墨负责其余逻辑运算的设计
- 2023/10/14 10:00 完成除法模块的设计和测试
- 2023/10/14 15:00 完成乘法模块的设计和测试
- 2023/10/16 10:00-22:00 完成调试和 debug >.<

- 2023/10/16 22:00 完成 exp10 实验仿真，但未通过实现。
- 2023/10/17 8:00 开始 exp11。

exp11 实验流水账

- 2023/10/17 10:00 完成 exp11 实验，开始 debug
- 2023/10/17 19:00 exp11 调试完成，开始综合与实现
- 2023/10/23 17:00 上板测试通过，开始撰写实验报告
- 2023/10/23 22:00 实验报告撰写完毕，开始 exp12 分工

三、错误记录

1、错误 1：乘法器时序违约过多

（1）错误现象

乘法器模块级仿真通过，将其嵌入整个 CPU 的代码后，仿真通过。但综合实现之后的时序报告中，可以看到有高达 630 条违约的关键路径，WNS 违约值高达 1.724ns。

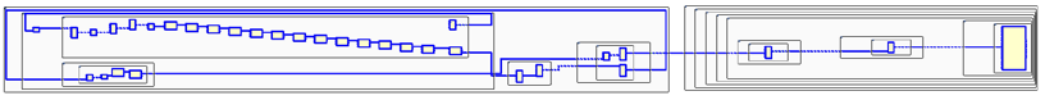
Setup		Hold	
Worst Negative Slack (WNS):	-1.724 ns	Worst Hold Slack (WHS):	0.049 ns
Total Negative Slack (TNS):	-231.498 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	630	Number of Failing Endpoints:	0
Total Number of Endpoints:	9007	Total Number of Endpoints:	9007

Timing constraints are not met.

图表 5

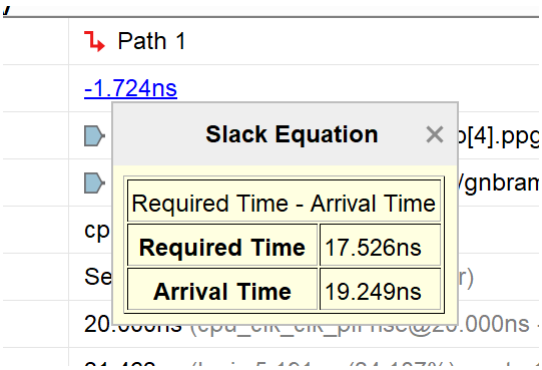
（2）分析定位过程

打开 Design Timing Summary，看到 Setup 时间违约最高的关键路径，查看其 Schematic 原理图，如下。



图表 6

可以看到，这是“用华莱士树算出乘法结果，前递给 ID 级的跳转指令，去算跳转地址送入指令 RAM”的路径。由于其组合逻辑路径过长，难以在建立时间开始之前计算完成（也就是一个时钟周期内难以完成）。如下图所示，Arrival Time 大于 Required Time，也就是说计算结果到达的时间晚于要求的时间。



图表 7

鉴于这条路径中华莱士树所占的路径过长，很自然地能够想到，如果在乘法器的结尾切一个流水级，这条关键路径就能短很多，也许就能解决这个时序问题。

现在就需要考虑，是把乘法器改成三拍，还是仍旧保留两级，把切流水的地方从部分积和华莱士树之间改到乘法器结尾。鉴于三拍需要增加阻塞，修改的地方过多，因此我先尝试了后者。再次综合实现，得到时序报告，发现违约路径已经减到只剩 3 条，WNS 减小到-0.092ns，已经不存在和乘法器有关的违约路径，说明乘法器已经修改到对时序来说的最佳状态。

Setup		Hold	
Worst Negative Slack (WNS):	-0.092 ns	Worst Hold Slack (WHS):	0.061 ns
Total Negative Slack (TNS):	-0.209 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	3	Number of Failing Endpoints:	0
Total Number of Endpoints:	8472	Total Number of Endpoints:	8472

Timing constraints are not met.

图表 8

（3）错误原因

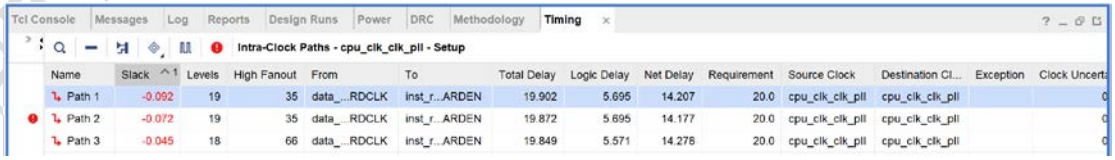
“用华莱士树算出乘法结果，前递给 ID 级的跳转指令，去算跳转地址送入指令 RAM”这条关键路径过长。

（4）修正效果

时序报告中，违约路径只剩 3 条，WNS 减小到-0.092ns，已经不存在和乘法器有关的违约路径。

2、错误 2：仍然存在三条与前递有关的违约路径

（1）错误现象



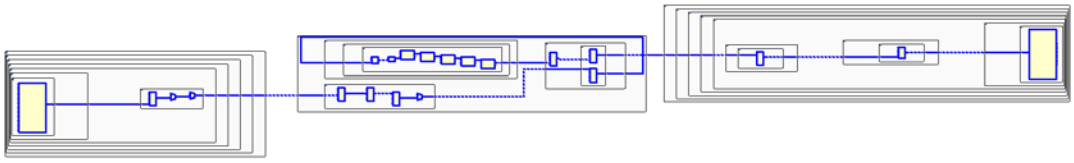
图表 9

修改完上一个错误之后，前面提到仍然剩三条关键路径是违约的。

（2）分析定位过程

仍然从 Design Timing Summary 中找出违约最严重的关键路径的 Schematic，只要在关键路径上合适的地方切

流水，就能够解决这个问题。



图表 10

但是，从原理图上看到数据经过的组合逻辑路径是“dataRAM→MEM→ID(RF)→EXE(alu div)→ID(RF)→instRAM”。这条数据通路很像是：ld 指令从数据 RAM 取回数据之后送给 MEM，MEM 将结果前递给 ID 的寄存器堆，某个数据从 ID 传递到 EXE 并进入 alu 中的乘法器进行计算，再次前递给 ID 的寄存器堆，ID 将跳转地址送给指令 RAM。但这听起来就很不合理，因为

①前递回 ID 不需要进寄存器堆；

②不会有两个前递在同一个时钟周期内发生。

这说明，这条逻辑路径并不是表面看起来的这样，只能打开完整的 Schematic 图去探寻真正的路径。

经过调研，发现 FPGA 的内部的结构单位是可配置逻辑块 CLB，它由 LUT6、寄存器、MUX、加法器进位链组成。也就是说，在综合实现的时候，不会简单地布局布线成零散的逻辑门，而是将代码中变量拆开成零散的位，去利用这些可配置逻辑块。也就是说，综合实现可能会将某些变量的某些位拉进某个模块里，但实际上代码里并未如此设计。因此，可能会造成，Schematic 图虽然显示数据通路进了某个模块，但代码里的数据其实并没有进到我们写的模块里的“假象”。

CLB（可配置逻辑块）

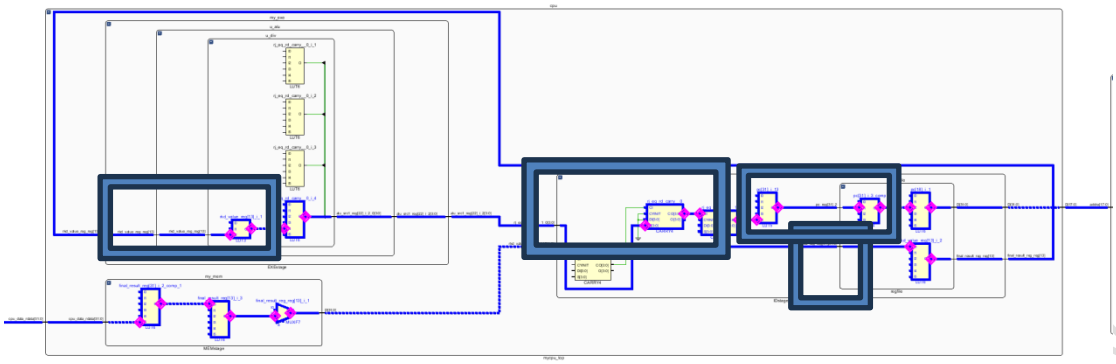
- CLB组成
  - 1CLB=2SLICE
  - 1SLICE = 4*LUT6+8*storage elements+3MUX+CARRY4进位链

1个CLB里面有2个SLICE，SLICE可分为SLICEM和SLICEL，M代表memory，L代表logic，顾名思义，带M的SLICE可以用于组成disturbuted ram和shift register，带L的SLICE用于逻辑资源。

1个SLICE中有4个LUT6，8个寄存器，3个2选1MUX和一个4bit的加法器进位链，这些后续会进行详细介绍。

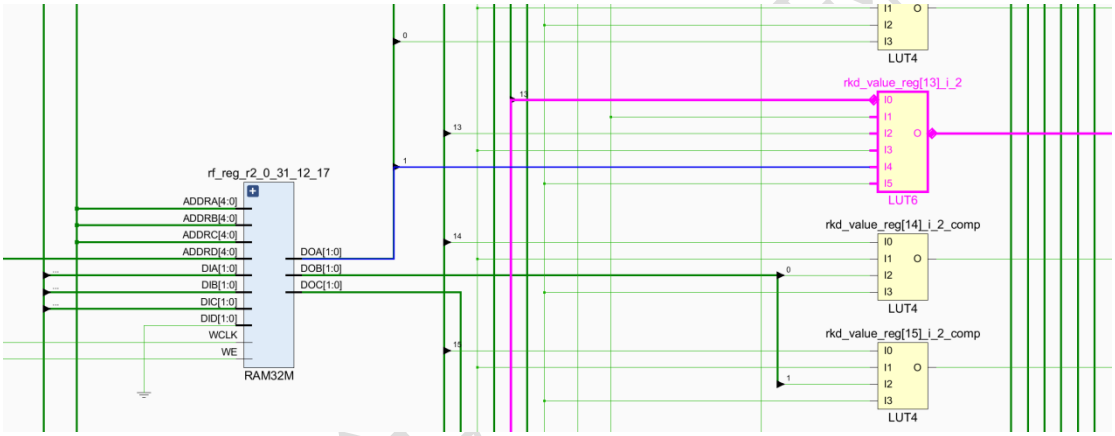
图表 11

有了这个认知，打开 Schematic 全图，再次追踪数据通路的走向，就能解决之前的很多疑问。



图表 12

① 路径第一次进 ID 的时候，其实并没有进寄存器堆。路径看似进了寄存器堆，是因为 LUT6 的一个输入是从 RF 出来的，这可能是综合实现的时候为了在 FPGA 上布局布线作出的安排。如下图，粉色的是关键路径，蓝色的是从寄存器堆来的输入。



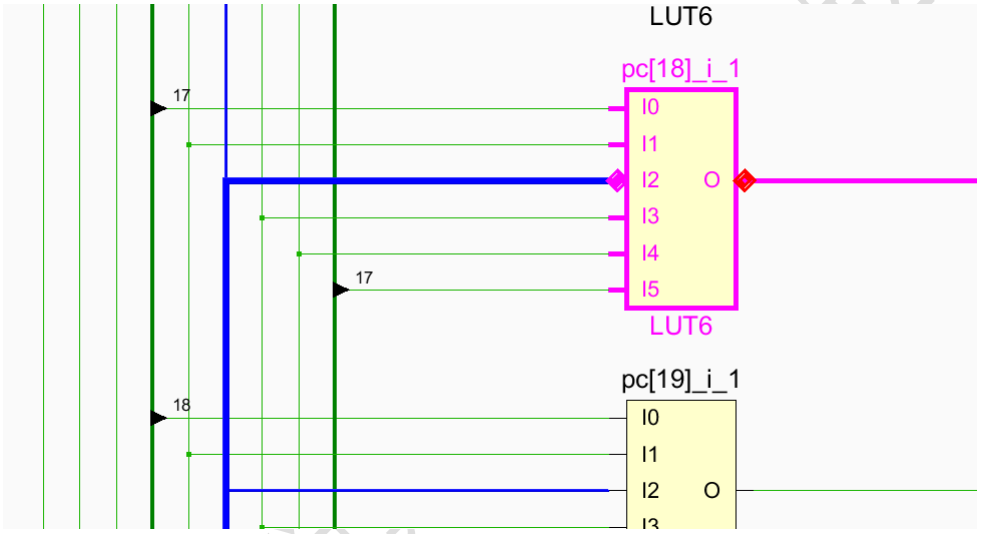
图表 13

② 路径其实并没有进入 EXE 模块，更没有进入 alu 和除法器。进了除法器之后的第一个模块，除了 rkd_value 之外，还有一个叫 quotient_reg 的输入，我猜这个叫 quotient_reg 的输入就是为什么信号莫名其妙拉进了除法器的原因。进除法器之后的第二个模块，除了 rkd_value 之外，还有一个输入信号叫 exe_gr_we，这是为了判断冲突从 EXE 拉回 ID 的信号。所以很自然地就能联想到，虽然我写代码的时候把 exe_gr_we 改成了 exe_rf_we 拉回了 ID，但在 FPGA 上布局布线的时候处于某些考虑，直接带着 rkd_value（这应该也不是真的 rkd_value，而是 rkd_value 生成之前的某个中间信号）进了 EXE，甚至还进到了除法器模块里，“舍近求远”地来找 exe_gr_we，最后才算出来真正的 rkd_value，再回到 ID。在 schematic 的模块上右键点“go to source”，回到的也是下图这部分，根本不是 EXE，说明该猜想很正确。也就是说，布局布线时把 rkd_value 的生成逻辑放到了除法器里。

```
assign rkd_value =
    (exe_rf_we && exe_dest == rf_raddr2)? alu_result :
    (mem_rf_we && mem_dest == rf_raddr2)? final_result :
    (rf_we  && dest_reg   == rf_raddr2)?  rf_wdata   :
    rf_rdata2;
```

图表 14

③ 路径第二次进 ID 的时候，也并没有进寄存器堆。寄存器堆里的合格 LUT6 模块，输入信号有 br_zip, pc, ds_valid,seq_pc，很像选择下一拍的 pc 的逻辑，go to source 之后也确实来到这里了。但这个逻辑其实没有道理会在寄存器堆里。我点了它的每一个输入信号，都是直接或间接从 RF 的外面进来的，它的输出信号也会直接出 RF。所以相当于布局布线时出于某些考虑，把选下一拍 PC 的逻辑从 IF 模块挪进了 ID 的寄存器堆模块里。



图表 15

```
assign nextpc = br_taken ? br_target : seq_pc;
```

因此，这条关键路径实际上是 dataRAM→MEM→ID→IF→instRAM。它做的事情就是，从数据 RAM 取回 ld 的结果到 MEM，算出 final_result，前递回 ID，得到 rkd_value，又算出 br_target 传给 IF，在 IF 里算出 nextpc，送给指令 RAM。仔细想想这些事情确实是在一拍里完成的。

也就是说，这个过长的关键路径是从 MEM 前递造成的，如果要解决，要做的事情就是换成阻塞，但这势必又会影响流水线效率。目前上板是可以通过的，因此我们决定暂时不修改。

（3）错误原因

从 MEM 级前递回 ID 的逻辑的关键路径过长。

（4）修正效果

决定不予修改。

（5）归纳总结

在解决时序违约的问题时，我们由于要阅读时序报告，调研了时序相关的内容。

复述一下时序的基本概念：

Tsu：建立时间，时钟有效沿到来之前，数据必须保持稳定的最小时间

Th：保持时间，时钟有效沿到来之后，数据必须保持稳定的最小时间

Tco：数据输出延迟，寄存器内部输出延时

Tskew：是指一个时钟源到达两个不同的寄存器的时钟端的时间偏移，Tskew = Tclk2 － Tclk1
Tskew为正时，有利于SetUp满足条件

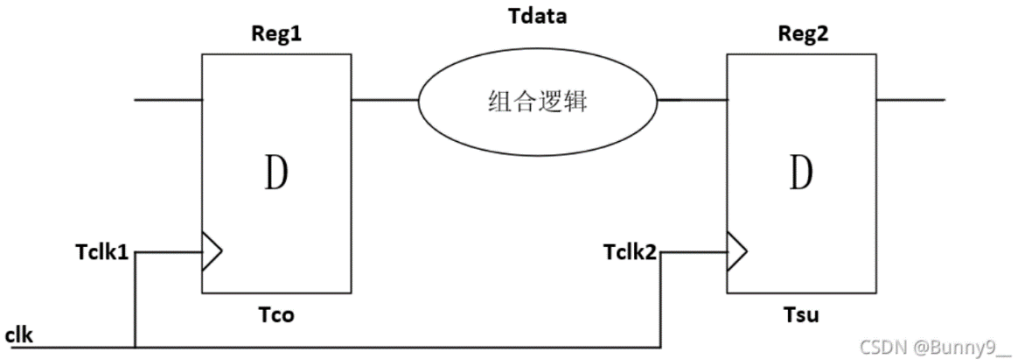
Tdata：走线延迟和组合逻辑延迟的总和，Tdata = Tlogic + Trouting

时序需要满足：要求 ≥ 到达

Tclk ≥ Tco + Tdata + Tsu － Tskew

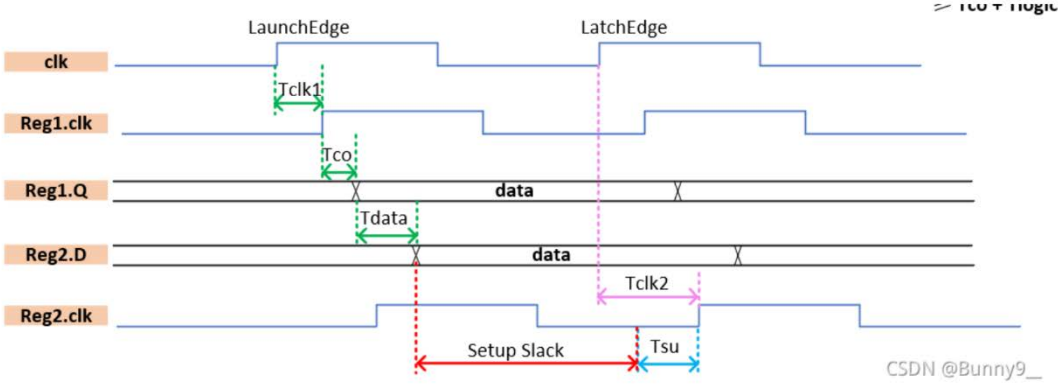
≥ Tco + Tlogic + Trouting + Tsu - Tskew

图表 16



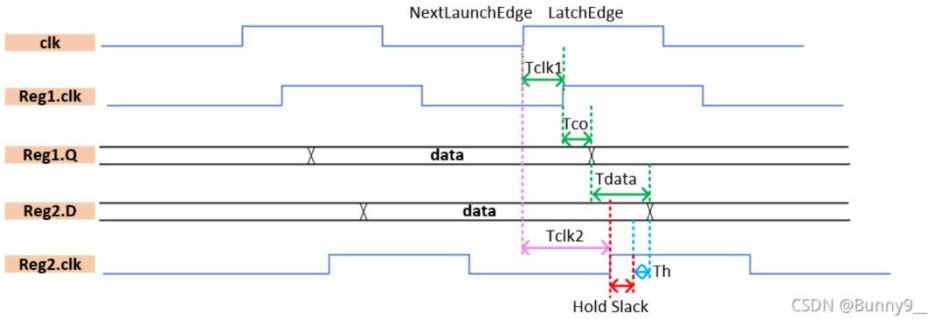
图表 17

建立余量：



图表 18

保持余量：



图表 19

(Hold) Data Require Time = LatchEdge + Tclk2 + Th

Data Arrival Time = NextLaunchEdge + Tclk1 + Tco + Tdata

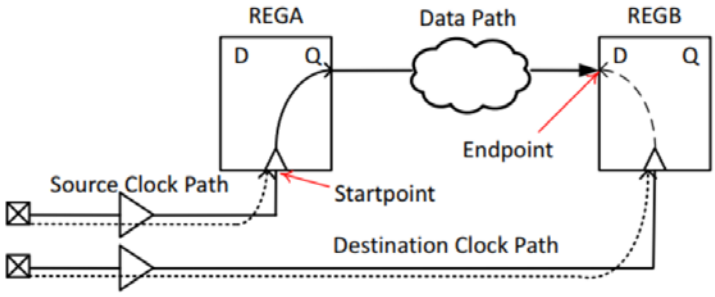
Hold Slack = Data Arrival Time － (Hold)Data Require Time
= (NextLaunchEdge + Tclk1 +Tco + Tdata) - (LatchEdge +Tclk2 + Th)
= (NextLaunchEdge - LatchEdge) + (Tclk1 － Tclk2) + (Tco + Tdata + Th)
= Tco + Tdata + Th - Tskew

图表 20

通过看 level 和 fanout，可以看到路径时序违例的原因，level 值过大，则表示逻辑层数太多，需要考虑将这条路径对应 HDL 代码分成几拍完成；如果 fanout 值过大，则表示该寄存器的扇出过大。理解建立余量的原理后，如果时序过大，那就需要尽可能减少两个除法器之间的组合逻辑级数，减低 Tdate.

每个时序路径分为三段：

- 源时钟路径（source clock path）
- 数据路径（data path）
- 目的时钟路径（destination clock path）



图表 21

1、 源时钟路径

源时钟路径：源时钟 从【源点（通常是输入端口）】到 【启动时序单元的时钟引脚】的路径。

对于一个从输入端口（port）开始的时序路径，没有源时钟路径。

2、 数据路径

数据路径：数据从路径起点（path startpoint）到路径终点（path endpoint）的传递路径。

路径起点：时序单元 的【时钟引脚 或 数据输入端口（port）】。

路径终点：时序单元 的【数据输入 或 输出 port】。

3、 目的时钟路径

目的时钟路径：目的时钟 从 【源点（通常是输入 port）】 到 【捕获时序单元的时钟引脚】的路径。

对于一个在输出 port 结束的时序路径，没有目的时钟路径。

3、错误 3：时延改进

ID 阶段中前递信号的处理部分中，可以看到运用了许多三元运算符，考虑到其时延可能会受到较大影响，考虑到可以改成与或逻辑继续选择，但是这样的操作将会倒置结果与预期完全不符！

原因是如果改成选择信号，就失去了 EXE MEM WB 阶段之间的优先级，导致两个阶段同时冲突且需要前递时发生意想不到的错误结果，因此为保证三个阶段之间的优先级，这里的三元运算符并不允许修改。（或者引入其他与或逻辑来代表优先级）

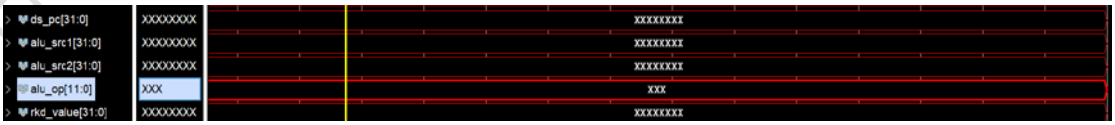
```
assign rj_value =
    (exe_rf_we && exe_dest == rf_raddr1)? alu_result :
    (mem_rf_we && mem_dest == rf_raddr1)? final_result :
    (rf_we  && dest_reg   == rf_raddr1)? rf_wdata   :
    rf_rdata1;
assign rkd_value =
    (exe_rf_we && exe_dest == rf_raddr2)? alu_result :
    (mem_rf_we && mem_dest == rf_raddr2)? final_result :
    (rf_we  && dest_reg   == rf_raddr2)?  rf_wdata   :
    rf_rdata2;
```

图表 22

4、错误 4： alu_op 错误位宽

```
module alu(
  input clk,
  input resetn,
  input  wire [11:0] alu_op,
  input  wire [31:0] alu_src1,
  input  wire [31:0] alu_src2,
  output wire [31:0] alu_result,
  output wire [31:0] alu_flag
);
```

因为需要扩充额外的运算，所以 alu_op 位数应该有所增加，但是在进行 exp10 实验时，遗漏了这点，因此出现在 alu 中进行结果选择时，选出了含有不定态值的结果，写回值出现不定态。写使能信号未拉高，导致未被采样，但并未报错。



图表 23

```
[1242000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[1252000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[1262000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[1272000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[1282000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
```

图表 24

将所有 alu_op 定义部分和将其加入传输 bus 信号位宽增加到需要的位宽即可。我们的设计中乘法器并不例化在 ALU 中，因此实际上的 alu_op 仅有 16 位。

5、错误 5：除法器功能错误

（1）错误现象

除法器算出的结果出现错误，提前输出结果

（2） 分析定位过程

最后一次循环内 counter 值为 32， [31 - counter] 会导致数组下标越界，因此最后一次循环时将当前余数赋值为 recover_r。

33 的二进制码是 6’b100001，其最后一位和最高位均为 1，最开始时判断的结束信号是 31 拍，导致提前输出结果

（3）修正效果

将两部分代码改成如下所示即可，即可

```
remainder_reg <=  (counter[5]&(~|counter[4:0])) ? recover_r : {recover_r, divisor_pad[31 - counter]};
```

```
assign complete = counter[5]&counter[0]&(~|counter[4:1]);
```

修改后结果正确，仿真验证通过。

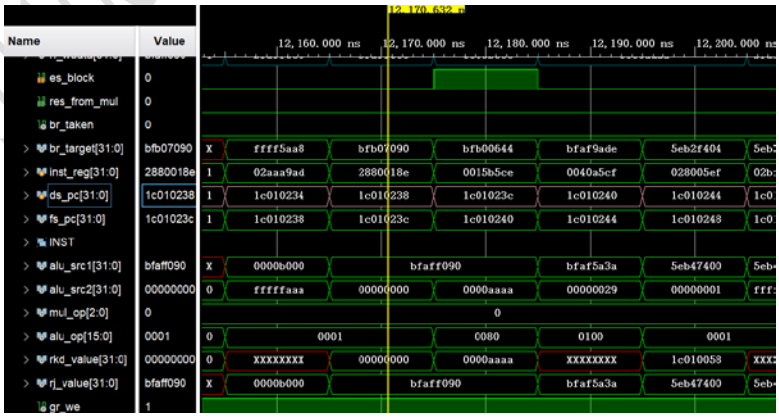
6、错误 6：前递信号判断遗漏

（1）错误现象

因为未考虑乘法器信号可能出现的前递，导致前递回错误的值发生错误

```
    reference: PC = 0x1c01023c, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c01023c, wb_rf_wnum = 0x0e, wb_rf_wdata = 0xbfaf5a3a
-----------------------------------------------------------------
```

图表 25



图表 26

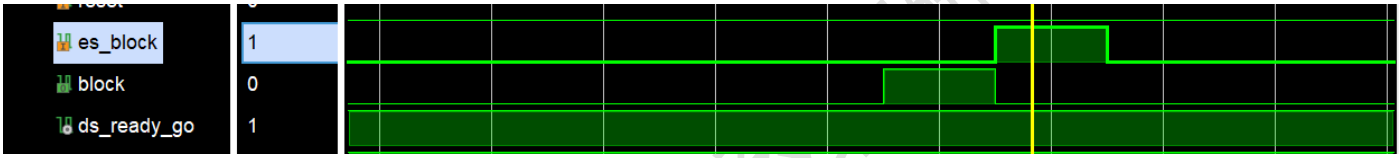
（2）分析定位过程

```
1c010228:    157f5fec    lu12i.w $r12,-263425(0xbfaff)
1c01022c:    0282418c    addi.w  $r12,$r12,144(0x90)
1c010230:    1400016d    lu12i.w $r13,11(0xb)
1c010234:    02aaa9ad    addi.w  $r13,$r13,-1366(0xaaa)
1c010238:    2880018e    ld.w    $r14,$r12,0
1c01023c:    0015b5ce    xor $r14,$r14,$r13
1c010240:    0040a5cf    slli.w  $r15,$r14,0x9
1c010244:    028005ef    addi.w  $r15,$r15,1(0x1)
```

图表 27

观察汇编文件，可以看到出现了较多的写后读访问冲突，按照我们的流水线前递设计，将会直接从所需操作数的对应阶段取回，减少流水线引起的阻塞损失。但是在这里，操作数的运算并没有发生错误，流水线也正常前递，可是结果并不对应。一开始想的是是否可能因为存数指令出错，导致 ld.w 指令取出错误的操作数，因为存数指令并没有写回寄存器的操作，不会进行调试比对。分析后发现，这不可能，因为前面任意一条存数指令都是从寄存器中取数，而寄存器中的值必定是某条指令写入的值，因此如果存数出错一定会在前面某条指令报错。

所以调试陷入了瓶颈，考虑到此处的写后读操作较多，因此再次从前递技术上着手，而后发现，ld.w 发生了冲突，理应在 ID 阶段阻塞，但是此时的 ds_ready_go 并未拉低，于是猜测就是因为这个原因导致尽管前递了数据，但是却不是正确的数据。



图表 28

（3）错误原因

在 ds_ready_go 信号的赋值部分，仔细检查，恍然大悟，是因为考虑阻塞条件时对 mul_op 信号的引入并不严谨，因为阻塞发生时当前的指令应该是需要操作数并且进行运算的指令，而因为 mul_op 和 alu_op 互斥，所以它们必定不会同时满足，所以 ds_ready_go 始终为高，出现错误。

```
assign ds_ready_go    =~(ds_valid && ((exe_rf_we && es_block &&
                        (exe_dest == rf_raddr1 && |rf_raddr1 && (~src1_is_pc & ~inst_lu12i_w &|alu_op &|mul_op) ||
                         exe_dest == rf_raddr2 && |rf_raddr2 && ~src2_is_imm & |alu_op &|mul_op))));
```

（4）修正效果

考虑到 mul_op 的引入后，重现修改赋值逻辑即可。

```
assign ds_ready_go    =~(ds_valid && ((exe_rf_we && es_block &&
                        (exe_dest == rf_raddr1 && |rf_raddr1 && (~src1_is_pc & ~inst_lu12i_w & |alu_op |(|mul_op)) ||
                         exe_dest == rf_raddr2 && |rf_raddr2 && ~src2_is_imm & |alu_op|(|mul_op)))));
```

（5）归纳总结（可选）

这个 bug 是因为前递信号设计遗漏而导致的，所以在控制逻辑部分引入新的信号时理应考虑到兼容和向后延展。

7、错误 7：赋值逻辑错误

（1）错误现象

```
[   2057 ns] Error!!!
    reference: PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
    mycpu    : PC = 0x0c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
```

图表 29

2057sPC 值和写数据均发生错误

（2）分析定位过程

首先是 PC 值的问题，发现主要是因为在执行到访存阶段的数据的传递的位宽出现问题，更改后查找写回数据问题。写回数据应该来自于 alu_result，而后赋值给 final_result，可以看到在赋值阶段出现了错误。而后发现 mem_res_from_mem 拉高了，但此时不应该拉高。查找 mem_res_from_mem 找到错误。

图表 30

（3）错误原因

assign mem_res_from_mem = ~(|mem_load_op);

右侧式子意思为对首先查找 load_op 是否为全 0，而后进行非操作。但是当 load_op 非全零时应该给 mem_res_from_mem 赋值为 1，因此不应该取反。

修改为 assign mem_res_from_mem = (|mem_load_op);

（4）修正效果

修改后该时间点不报错。

8、错误 8：load_op 错位

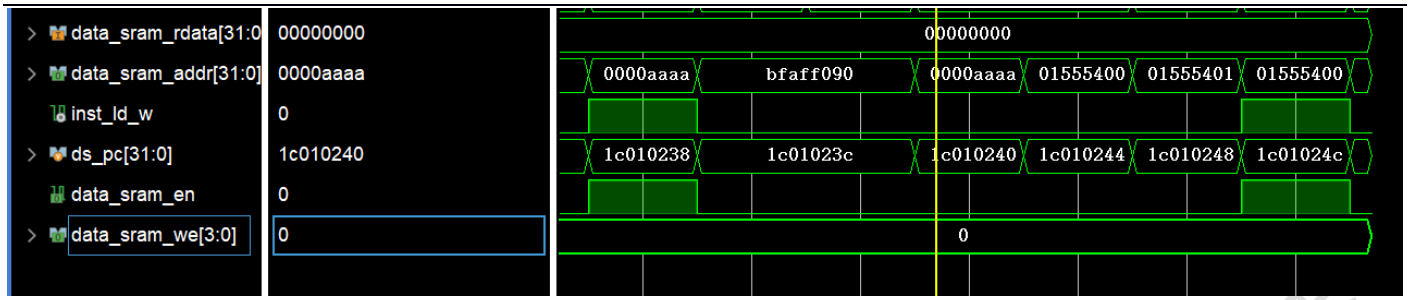
（1）错误现象

```
[  12197 ns] Error!!!
    reference: PC = 0x1c010238, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x0000aaaa
    mycpu    : PC = 0x1c010238, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
```

图表 31

写入数据错误。

（2）分析定位过程



图表 32

发现使能信号为 0，因此是有问题的。而后查找发现使用的是 load_op，即下一拍的 load_op，因此是错位的，应该使用 load_op_reg 为对应的 load 信号。

（3）错误原因

assign data_sram_en = (mem_we_reg || (|load_op)) & es_valid;

（4）修正效果

assign data_sram_en = (mem_we_reg || (|load_op_reg)) & es_valid;

修改后顺利通过测试点

9、错误 9： Ld.b 指令出错，立即数信号给错

（1）错误现象

```
[1235697 ns] Error!!!
    reference: PC = 0x1c066e6c, wb_rf_wnum = 0x10, wb_rf_wdata = 0x0000000c
    mycpu    : PC = 0x1c066e6c, wb_rf_wnum = 0x10, wb_rf_wdata = 0x00000000
```

图表 33

（2）分析定位过程

```
1c066e68:   29a030a5    st.w    $r5,$r5,-2036(0x80c)
1c066e6c:   28203190    ld.b    $r16,$r12,-2036(0x80c)
1c066e70:   28a03085    ld.w    $r5,$r4,-2036(0x80c)
```

图表 34

通过查看汇编文件，可以看到该地址是一条访存指令，取出的数据与预期金标准不符，考虑访存指令的数据通路，可能是掩码操作出现错误或者访存地址出错。

（3）错误原因

先考虑简单的情况，检查立即数选取发现 need_sil2 拉高未考虑新加入的访存指令，导致地址出错。

```
assign need_ui5   =  inst_slli_w | inst_srli_w | inst_srai_w;
assign need_ui12  =  inst_andi | inst_ori | inst_xori;
assign need_si12  =  inst_addi_w | inst_ld_w | inst_st_w | inst_slti | inst_sltui;
assign need_si16  =  inst_jirl | inst_beq | inst_bne;
assign need_si20  =  inst_lu12i_w | inst_pcaddu12i;
assign need_si26  =  inst_b | inst_bl;
assign src2_is_4  =  inst_jirl | inst_bl;
```

（4）修正效果

加入新的访存指令后，仿真通过。

（5）归纳总结

除此之外，还有诸多因为未考虑引入新指令的控制信号出错，例如 rd 寄存器选择信号中，需要考虑新引入的跳转指令和访存指令

```
assign src_reg_is_rd = inst_beq | inst_bne | inst_blt | inst_bge | inst_bltu | inst_bgeu | (|store_op );
```

以及执行阶段和访存阶段，数据 RAM 的读写使能信号未考虑新加入的访存指令导致结果错误。

```
assign data_sram_en    =  ((|exe_load_op)|mem_we) & es_valid;//1'b1;
```

四、实验总结

本次是第一次进行小组合作实验，由于之前还从未尝试过如此大面积地进行小组合作，因此效率并不是特别高，相信下一次实验会更顺利和有效率。

本实验添加了算术逻辑运算、乘除法运算、转移、访存四类指令，基本上都可以复用现有的数据通路，只需对控制信号作一些修改，但也有一些地方需要设计新的通路和器件。这次设计总体上并不难，如果单纯利用 IP 核提供的乘除法器，但是如果这样 vivado 例化和布线时往往会增加一些不必要的组合逻辑，造成一定的时延损失，所以在单独实现乘除法器时遇到了不小的困难。乘法器在于是如何实现华莱士树和划分二级流水以让其效率更高，对于除法器更多的按部就班，没有太多的可优化空间。

最后需要提到的是，希望可以修改一下测试文件，这次实验的仿真速度非常慢，而刚好新增的指令是在靠后的测试点，导致每次在跑仿真 debug 的时候，需要等待非常久的时间才能检测修正是否有效。因此如果可以，测试点可以减少或新增的检测指令往前挪动。