

LAB 3

学号:2020K8009915008

姓名:林孟颖 箱子号:79

LAB 3

一.实验任务概览

- (一) 实验目的
- (二) 检验方式

二.实验设计

- (一) 总体设计思路
- (二) 重要模块1设计: 指令地址的生成
 - 1.工作原理
 - 2.接口定义
 - 3.功能描述
- (三) 重要模块2设计: ALU相关数据的生成
 - 1.工作原理
 - 2.接口定义
 - 3.思路描述&对应代码
- (四) 重要模块3设计: 访存 (Data RAM) 信号的生成
 - 1.工作原理
 - 2.接口定义
 - 3.思路描述&对应代码
- (五) 重要模块4设计: 写回相关信号的生成
 - 1.工作原理
 - 2.接口定义
 - 3.思路描述&对应代码
- (五) 重要模块5设计: 和debug模块的交互
 - 1.工作原理
 - 2.接口定义
 - 3.思路描述&对应代码

三.实验过程

- (一) 实验流水
- (二) 环境配置记录
 - 1. 关于定制inst_ram IP核
 - 2. 交叉编译工具链配置的补充说明
- (三) 实践任务6: 20条指令单周期CPU
 - 错误1: 信号为'X'
 - 错误2: 信号为'Z' (拼写错误)
 - 错误3: 信号未定义+1 (拼写错误)
 - 错误4: ALU接口连接错误
 - 错误5: 错用时序逻辑
 - 错误6: 代码逻辑有误 (gr_we)

错误7：ALU操作数颠倒（移位指令）

错误8：组合逻辑环

错误9：位宽截取有误

错误10：confreq信号重复定义

四.实验总结

一. 实验任务概览

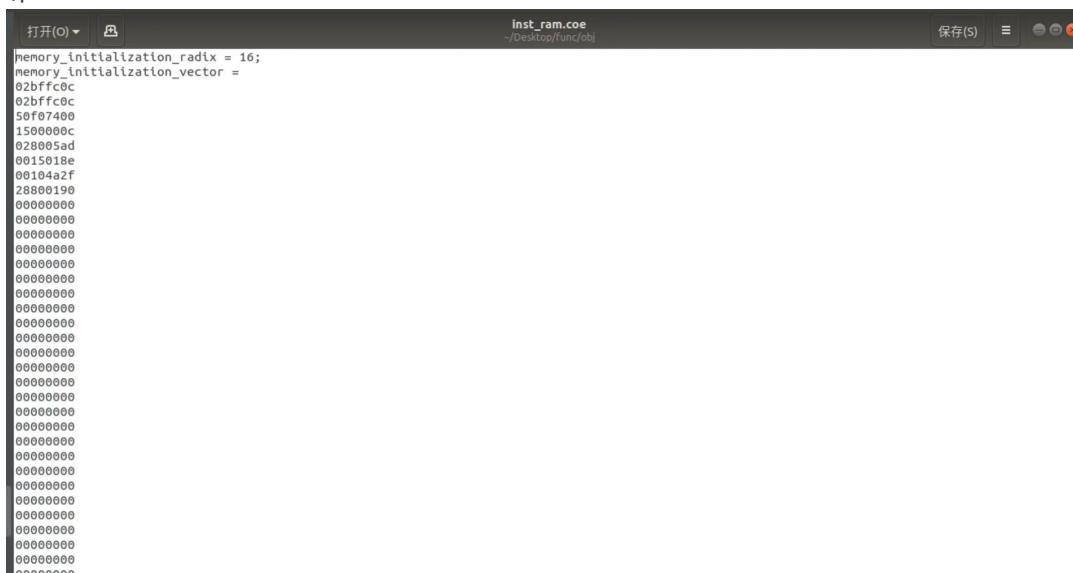
• (一) 实验目的

1. 理解Loongarch指令级的基本指令实现；
 2. 完成20条指令单周期CPU设计；
 3. 进一步熟悉看波形+反汇编代码debug的操作；
 4. 熟悉基于trace的检验模式。

• (二) 检验方式

1. 对测试用例进行交叉编译，处理后得到coe文件和mif文件

- 对比coe文件和反汇编代码可知其相当于一个“指令堆”，给出了指令在内存中的分布：



```

obj > ASM test.s
1
2 ./obj/main.elf:      文件格式 elf32-loongarch
3 ./obj/main.elf
4
5
6 Disassembly of section .text:
7
8 1c000000 <_start>:
9 kernel_entry():
10 1c000000: 02bffc0c addi.w $r12,$r0,-1(0xffff)
11 1c000004: 02bffc0c addi.w $r12,$r0,-1(0xffff)
12 1c000008: 50f07400 b 61556(0xf074) # 1c00f07c <Locate>
13 1c00000c: 1500000c lu12i.w $r12,-524288(0x80000)
14 1c000010: 028005ad addi.w $r13,$r13,1(0x1)
15 1c000014: 0015018e move $r14,$r12
16 1c000018: 00104a2f add.w $r15,$r17,$r18
17 1c00001c: 28800190 ld.w $r16,$r12,0
18 ...
19 1c0000ec: 1500000c lu12i.w $r12,-524288(0x80000)
20 1c0000f0: 028005ad addi.w $r13,$r13,1(0x1)
21 1c0000f4: 0015018e move $r14,$r12
22 1c0000f8: 00104a2f add.w $r15,$r17,$r18
23 1c0000fc: 28800190 ld.w $r16,$r12,0
24
25 1c000100 <test_finish>:

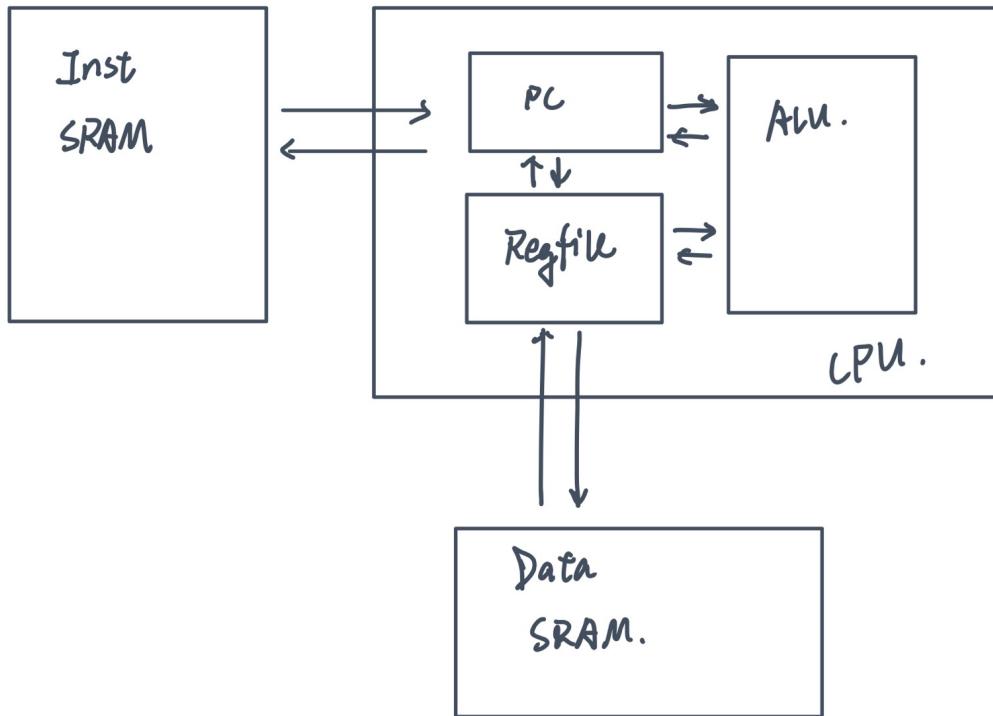
```

2. 启动gettrace工程并跑仿真，将写回寄存器的相关信息（含此时的pc、写回地址、写使能、写回数据）存储在golden_trace.txt里作为标签；
 - 当写回寄存器为0号寄存器时不做记录，因为后续从中只能读出0。
3. 运行CPU的testbench，其会在每次写回时将相关信息与golden_trace.txt中记录的标签作比较，若错误则报“Error”，并可根据debug_wb_err拉高的时间定位出错位置。

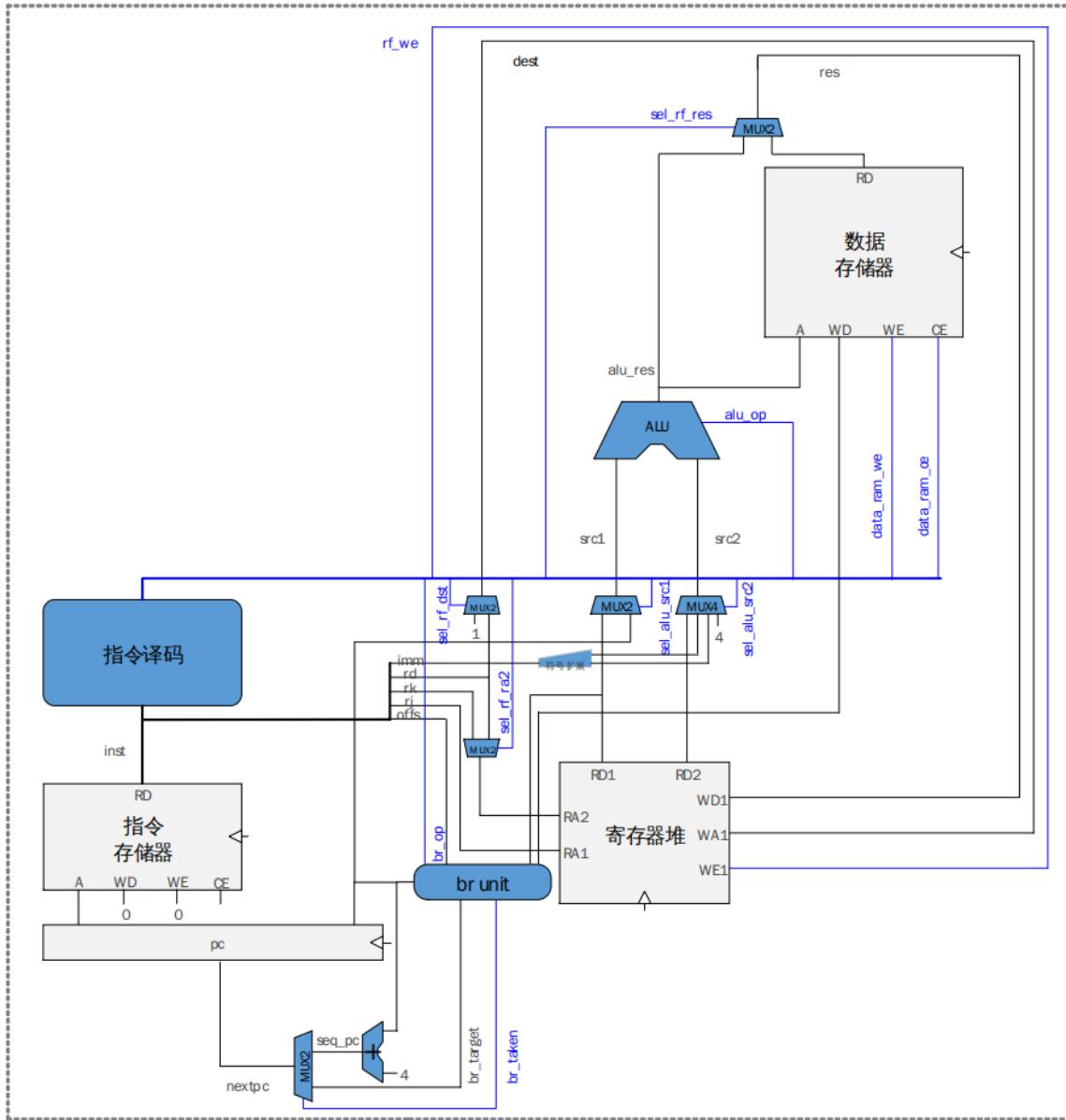
二. 实验设计

- **(一) 总体设计思路**

首先明确，CPU时不时在与内存做交互，其给内存提供地址与读写使能信号，从内存中读取数据或修改内存中数据，大致示意图如下（此图未完全描述本实验的CPU，其未画出CPU与debug模块交互的界面）。



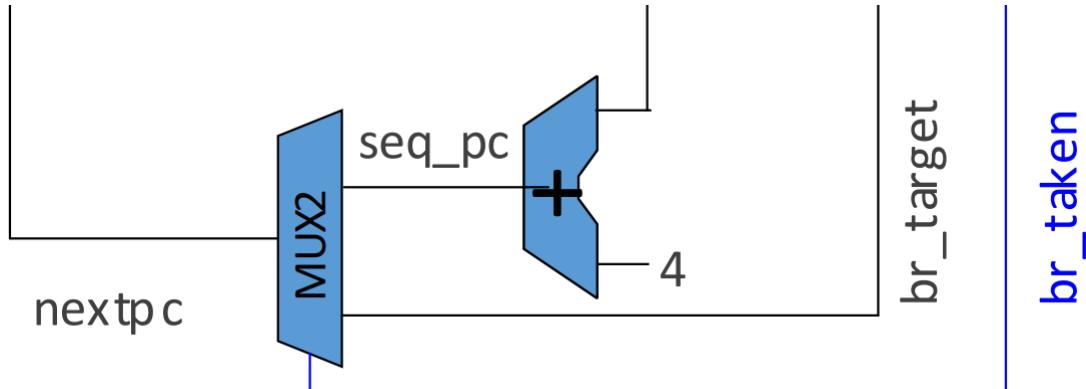
要实现该功能，需要完整的“数据通路”和“控制逻辑”的设计，图中已经给出所需的五大部件：PC、指令RAM、数据RAM、ALU、通用寄存器堆（由于本次实验采取直接映射的方式，故未画出地址映射的部件），只需再完善控制逻辑，具体包括PC的生成、指令译码、ALU计算的源数据和操作码的生成、取数相关信号的确定、写回的地址与数据的生成等。



• (二) 重要模块1设计：指令地址的生成

- 1. 工作原理

通过二选一的选择器选出下一个PC值，其两个源操作数为顺序执行的PC以及分支跳转的目的地址，其控制信号为 `br_taken`，译码时可得。



- 2. 接口定义

Name	I/O	Func
nextpc	OUT	给出下一个pc值
seq_pc	IN	顺序执行的pc值
br_taken	IN	跳转的目标pc值 (含b-type和j-type指令)
br_target	IN	选择pc来源的控制信号

- 3. 功能描述

功能：确定下一个取指地址；

对应代码：

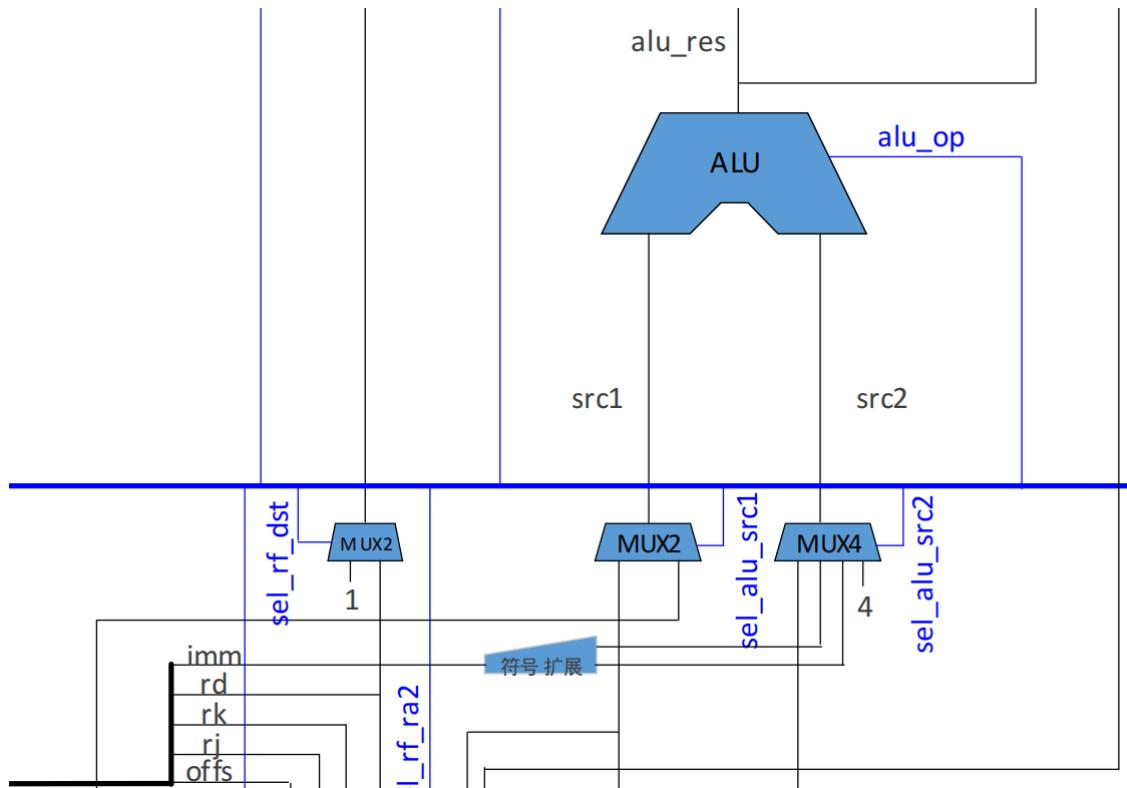
```
assign br_taken = ( inst_beq && rj_eq_rd
                    || inst_bne && !rj_eq_rd
                    || inst_jirl
                    || inst_bl
                    || inst_b
                    );
//      ) && ds_valid;
assign br_target = (inst_beq || inst_bne || inst_bl || inst_b) ? (pc +
br_offs) :
                                         /*inst_jirl*/
(rj_value + jirl_offs);
assign seq_pc      = pc + 3'h4;
assign nextpc     = br_taken ? br_target : seq_pc;

always @(posedge clk) begin
    if (~resetn) begin
        // if (reset) begin
        pc <= 32'h1c000000;
    end
    else begin
        pc <= nextpc;
    end
end
```

• (三) 重要模块2设计：ALU相关数据的生成

- 1. 工作原理

ALU有三个输入端口，一个输出端口，其三个输入都需在译码时通过选择器生成。



- 2. 接口定义

Name	I/O	Func
<code>alu_result</code>	OUT	alu的计算结果
<code>alu_src1</code>	IN	alu的源操作数1
<code>alu_src2</code>	IN	alu的源操作数2
<code>alu_op</code>	IN	alu的操作码，指定执行何种运算

- 3. 思路描述&对应代码

- `alu_src1`: 候选数据包括当前pc值与寄存器堆读出的rdata1；
- `alu_src2`: 候选数据包括符号扩展的12位立即数、常值立即数（4）、左移12位的20位立即数与寄存器堆读出的rdata2；
 - 先确定imm，再在imm和 rdata2间做选择，使得思路更清晰
- `alu_op`: 含12种操作，采用独热码译码；

```

assign imm = src2_is_4 ? 32'h4 : need_si20 ? {i20[19:0], 12'b0} : /*need_si5 || need_si12*/{{20{i12[11]}}}, i12[11:0}} ;
assign alu_src1 = src1_is_pc ? pc[31:0] : rj_value;
assign alu_src2 = src2_is_imm ? imm : rk_value;

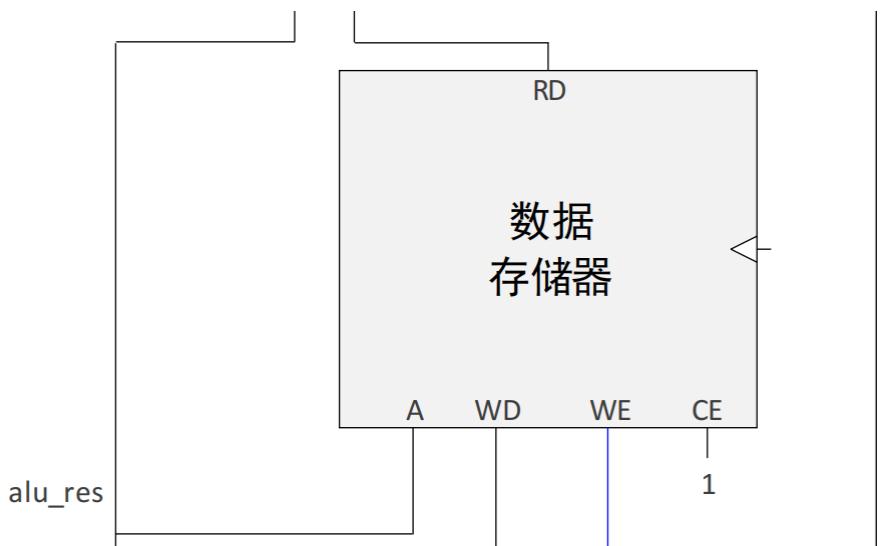
assign alu_op[ 0] = inst_add_w | inst_addi_w | inst_ld_w | inst_st_w
                  | inst_jirl | inst_b1;
assign alu_op[ 1] = inst_sub_w;
assign alu_op[ 2] = inst_slt;
assign alu_op[ 3] = inst_sltu;
assign alu_op[ 4] = inst_and;
assign alu_op[ 5] = inst_nor;
assign alu_op[ 6] = inst_or;
assign alu_op[ 7] = inst_xor;
assign alu_op[ 8] = inst_slli_w;
assign alu_op[ 9] = inst_srli_w;
assign alu_op[10] = inst_srai_w;
assign alu_op[11] = inst_lu12i_w;

```

- **(四) 重要模块3设计：访存 (Data RAM) 信号的生成**

- **1. 工作原理**

数据存储器异步读同步写，通过使能信号控制写。本次实验无读使能信号，且只有Load和Store类指令需要访存（对应ld.w和st.w）。



- 2. 接口定义

Name	I/O	Func
data_sram_rdata	OUT	读出的数据
data_sram_wdata	IN	待写入的数据
data_sram_addr	IN	待写入的地址
data_sram_we	IN	写使能信号

- 3. 思路描述&对应代码

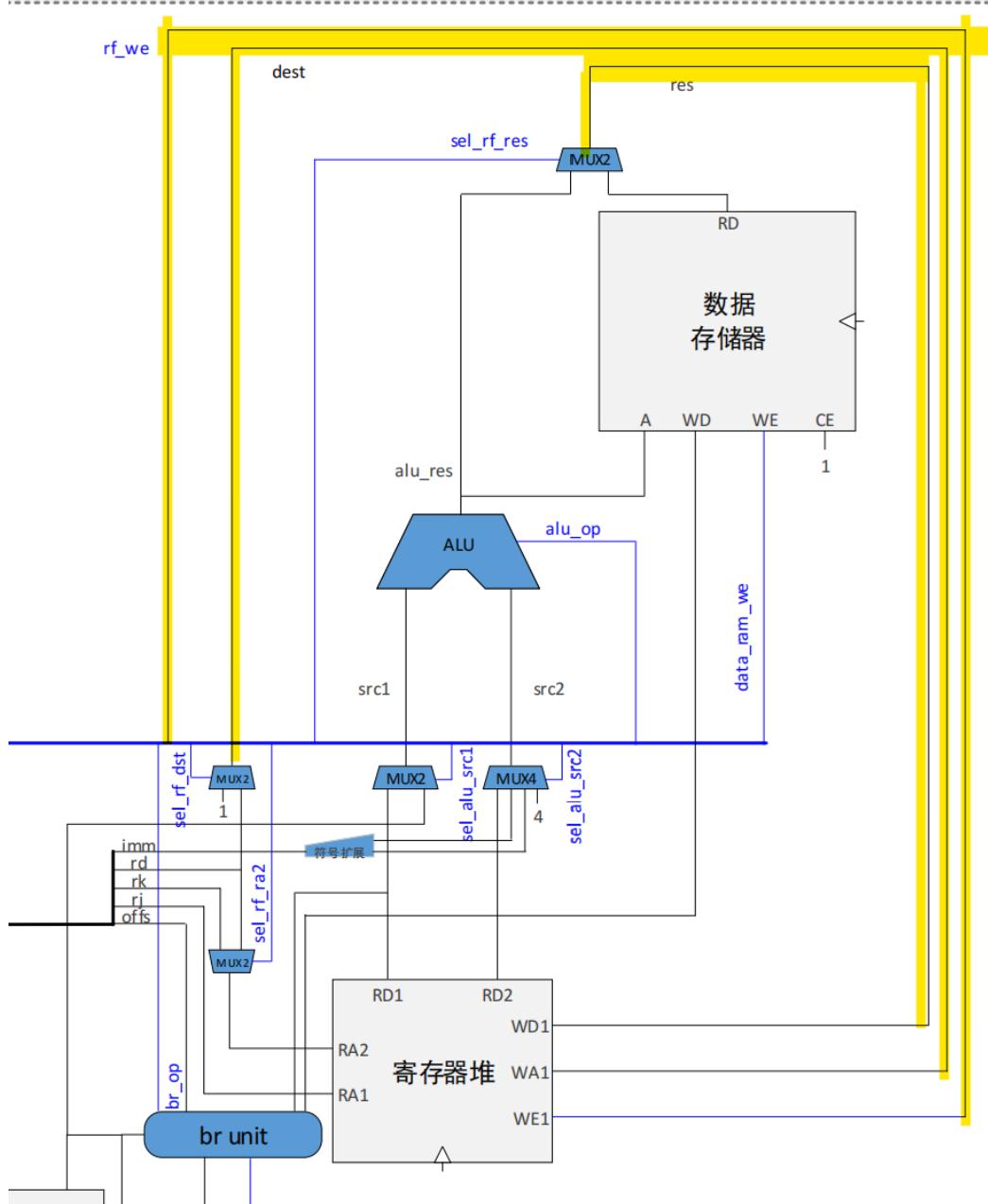
- data_sram_we：本次实验只有st.w需写内存，故可将data_sram_we在译码出st.w时置为1
- data_sram_addr：待写入的地址由alu计算结果给出（此次实验给出的数据刚好是4byte对齐的）
- data_sram_wdata：由寄存器堆读出的rf_rdata2给出

```
assign data_sram_we      = mem_we;
assign data_sram_addr   = alu_result;
assign data_sram_wdata = rkd_value;
```

• (五) 重要模块4设计：写回相关信号的生成

- 1. 工作原理

此部分需确定三个信号，写使能、写地址、写数据。即图中三条黄线：



- 2. 接口定义

Name	I/O	Func
rf_we	IN	寄存器堆写使能
rf_waddr	IN	待写入寄存器的地址
rf_wdata	IN	写入寄存器堆的数据

- 3. 思路描述&对应代码

- rf_we: 在20条指令种，除却store类型指令、除bl外的分支指令，其余都需写回
- rf_waddr: 写回地址的候选项有两个，一号寄存器（bl指令）和rd寄存器
- rf_wdata: 写回数据的候选项有两个，从内存中读出的数据和alu计算的结果（本实验只设计ld.w指令，故mem_result直接等于data_sram_rdata，不涉及mask的

设计)。

```
assign gr_we      = ~inst_st_w & ~inst_beq & ~inst_bne & ~inst_b;
assign dest       = dst_is_r1 ? 5'd1 : rd;
assign mem_result = data_sram_rdata;
assign final_result = res_from_mem ? mem_result : alu_result;

assign rf_we     = gr_we;
assign rf_waddr = dest;
assign rf_wdata = final_result;
```

• (五) 重要模块5设计：和debug模块的交互

- 1.工作原理

当前操作需要写回寄存器且目的寄存器非0号寄存器时需要读取下一条golden_trace记录，比较当前pc、写地址、写数据和标签，有错误则终止仿真。

- 2.接口定义

Name	I/O	Func
debug_wb_pc	OUT	写回时的pc值
debug_wb_rf_we	OUT	写使能信号
debug_wb_rf_wnum	OUT	写回时的寄存器号
debug_wb_rf_wdata	OUT	写回寄存器的数据

- 3. 思路描述&对应代码

```
// debug info generate
assign debug_wb_pc      = pc;
assign debug_wb_rf_we   = {4{rf_we}};
assign debug_wb_rf_wnum = dest;
assign debug_wb_rf_wdata = final_result;
```

本部分只实现了写回时的信息update，但未排除寄存器号为0的情况，这一部分判断在testbench中实现（即 `debug_wb_rf_wnum!=5'd0` 的约束条件）：

```
always @(posedge soc_clk)
begin
#2;
```

```

if(!resetn)
begin
    debug_wb_err <= 1'b0;
end
else if(| debug_wb_rf_we && debug_wb_rf_wnum!=5'd0 && !debug_end &&
`CONFREG_OPEN_TRACE)
begin
    if( ( debug_wb_pc!=ref_wb_pc) ||
(debug_wb_rf_wnum!=ref_wb_rf_wnum)
    ||(debug_wb_rf_wdata_v!=ref_wb_rf_wdata_v) )
begin
    $display("-----");
    $display("[%t] Error!!!",$time);
    $display("      reference: PC = 0x%8h, wb_rf_wnum = 0x%2h,
wb_rf_wdata = 0x%8h",
            ref_wb_pc, ref_wb_rf_wnum, ref_wb_rf_wdata_v);
    $display("      mycpu      : PC = 0x%8h, wb_rf_wnum = 0x%2h,
wb_rf_wdata = 0x%8h",
            debug_wb_pc, debug_wb_rf_wnum,
debug_wb_rf_wdata_v);
    $display("-----");
    debug_wb_err <= 1'b1;
#40;
$finish;
end
end
end

```

三.实验过程

- **(一) 实验流水**

- 2022.08.28 20:00-21:00 阅读讲义
- 2022.08.29 8:00- 9:20 配置环境
- 2022.08.29 20:00-24:00 Debug
- 2022.08.30 8:40-9:40 Debug

• (二) 环境配置记录

- 1. 关于定制inst_ram IP核

讲义中提及可按照自己需求选择GLOBAL或者OOC模式，此前我对两种模式没什么概念，查找资料整理如下：

- GLOBAL：工程综合时会对IP核的源码进行综合，Global模式不独立产生.dcp文件（design checkpoint），综合时耗时较久；
 - OOC：out of context per IP，指定再综合时对IP单独综合，生成.dcp文件，工程在后续使用IP时只需将IP视作单独模块，从.dcp文件解析出对应IP的网表文件。
- OOC方式定制后续综合时更快，但相应的在单独IP重新定制的时候耗时就会很长，好处是后续若需多次综合的话可重复利用.dcp文件，故建议使用OOC模式。

- 2. 交叉编译工具链配置的补充说明

按照讲义将安装的工具链地址写入 `~/.bashrc` 文件后系统仍然无法识别loongarch32r-linux-gnusf系列指令，上网搜索配置其他交叉编译环境配置的教程，得知可以在按照讲义解压压缩包到opt文件夹后再尝试如下方法：

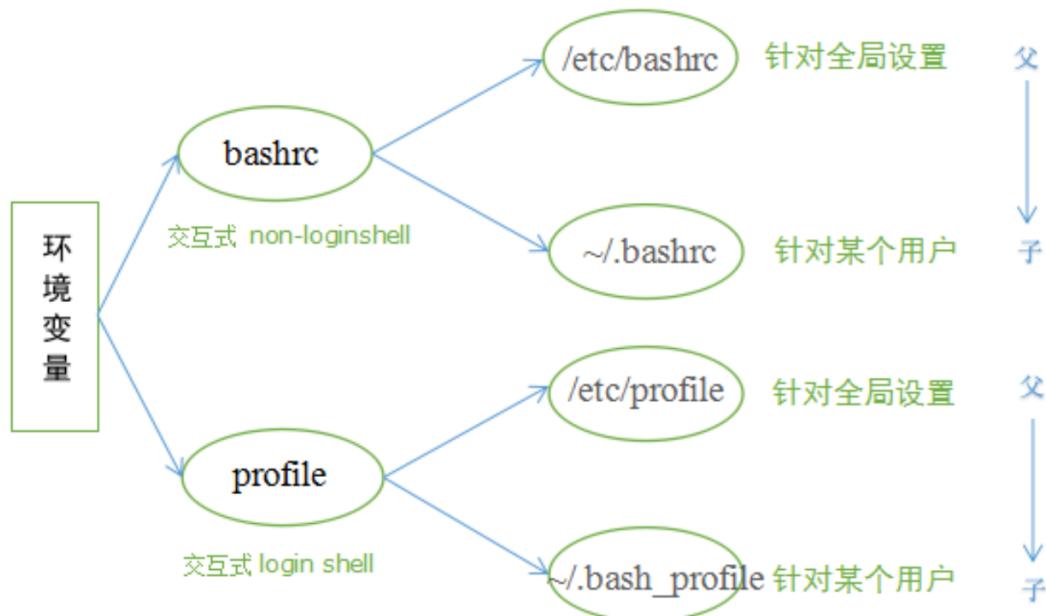
打开profile文件：

```
sudo vim /etc/profile
```

在文件末尾加上：

```
export PATH=$PATH:/opt/loongarch32r-linux-gnusf-2022-05-20/bin
```

补充的查找资料：



- 四种文件的区别详见：
[【笔记】etc/profile和~/.bashrc的区别](#)
[ZoeYen的博客-CSDN博客 /etc/profile和.bashrc的区别](#)

• (三) 实践任务6：20条指令单周期CPU

- 错误1：信号为'X'

(1) 错误现象

Vivado的Tcl console报错如下：

```

[D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:237]
`ic' is not declared [D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:237]
`m_mem' is not declared [D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:250]
`e'mycpu_top' ignored due to previous errors [D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:1]
`e' step finished in '2' seconds
:sults log file:'D:/Desktop/cdp_edc_local/soc_verify/soc_dram/run_vivado/project/loongson.sim/sim_1/beav/xsim/xvlog.log'
le' step failed with error(s). Please check the Tcl console output or 'D:/Desktop/cdp_edc_local/mycpu_env/soc_verify/soc_dram/run_v
ected error while running simulation. Please correct the issue and retry this operation.
`ich_simulation' failed due to earlier errors.

```

(2) 错误原因&定位过程

Console提示将错误内容重定向到了xvlog.log文件，查看该文件可知是存在若干信号未定义的情况：

```

[D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:264]
ERROR: [VRFC 10-2989] 'fs_pc' is not declared
[D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:107]
ERROR: [VRFC 10-2989] 'ds_valid' is not declared
[D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:236]
ERROR: [VRFC 10-2989] 'ds_pc' is not declared
[D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:237]
ERROR: [VRFC 10-2989] 'rfrom_mem' is not declared
[D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:250]
ERROR: [VRFC 10-2865] module 'mycpu_top' ignored due to previous
errors [D:/Desktop/cdp_edc_local/mycpu_env/myCPU/mycpu_top.v:1]

```

(3) 修正方式与效果

根据代码功能，反推各信号的应有实现（其中未定义的信号皆是在设计流水线CPU的时候才会被使用，此处修改无需补充定义）：

- fs_pc和ds_pc：其分别为流水线CPU中处于取指和译码阶段对应的pc值，此处暂不考虑，一律改为pc即可。原代码：

```

assign seq_pc      = fs_pc + 3'h4;
assign br_target = (inst_beq || inst_bne || inst_bl || inst_b) ?
(ds_pc + br_offs) :/*inst_jirl*/ (rj_value + jirl_offs);

```

修改后：

```

assign seq_pc      = pc + 3'h4;
assign br_target = (inst_beq || inst_bne || inst_bl || inst_b)
? (pc + br_offs) :/*inst_jirl*/ (rj_value + jirl_offs);

```

- ds_valid：原代码：

```

assign br_taken = ( inst_beq && rj_eq_rd
|| inst_bne && !rj_eq_rd
|| inst_jirl
|| inst_bl
|| inst_b
) && ds_valid;

```

可知该信号用于判断流水线CPU的译码操作是否有效，此处直接去掉即可

- rfrom_mem和valid：

原代码中：

```

assign data_sram_en = (rfrom_mem || mem_we) && valid;

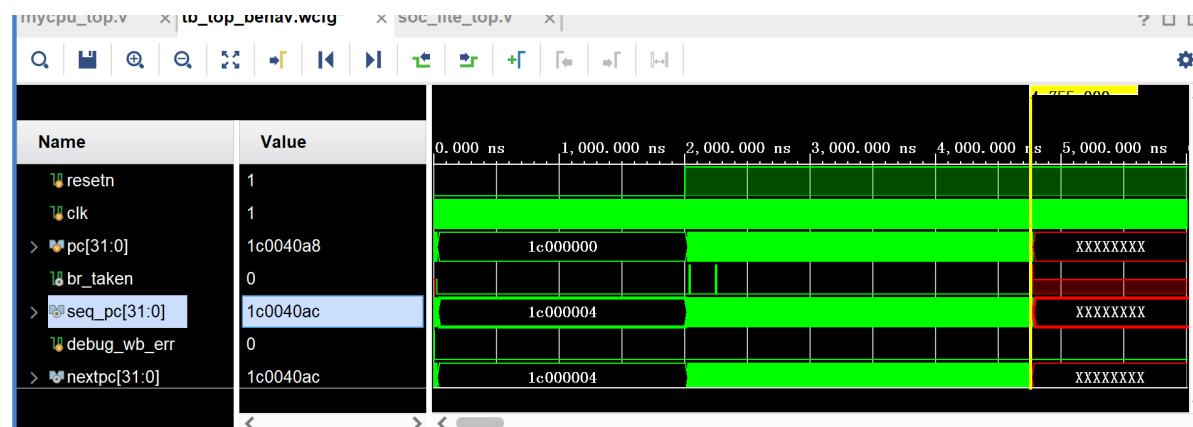
```

而观察代码全文可知暂未用到data_sram_en信号，可先直接注释掉。

- 错误2：信号为'Z'（拼写错误）

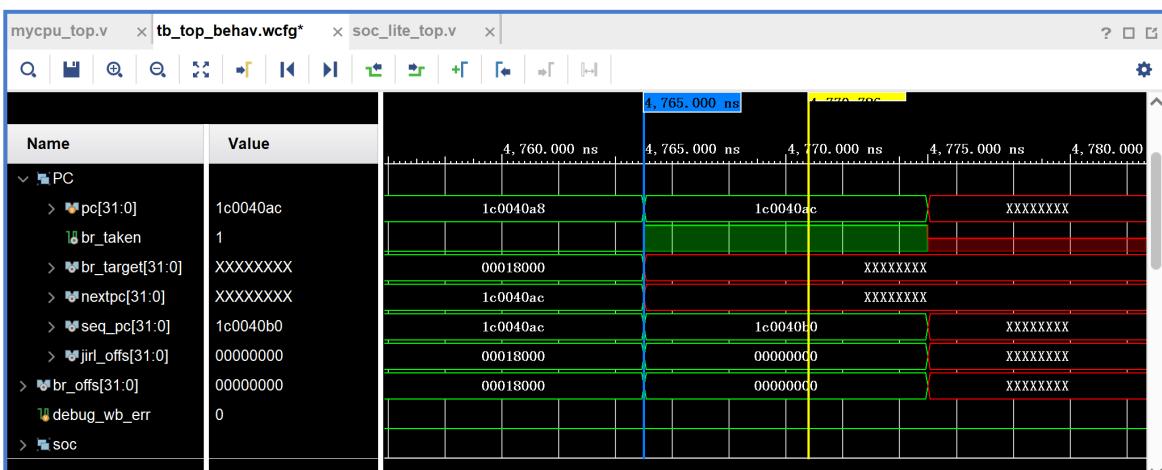
(1) 错误现象

PC一段时间后无效，但在此之前 debug_wb_err 始终为0，即写回寄存器时的处理皆没有问题。



(2) 分析定位过程

- 仔细观察波形，此时marker指示处分支信号有效，应该跳转，但此时跳转目标地址 `br_target` 出现未定义的情况：



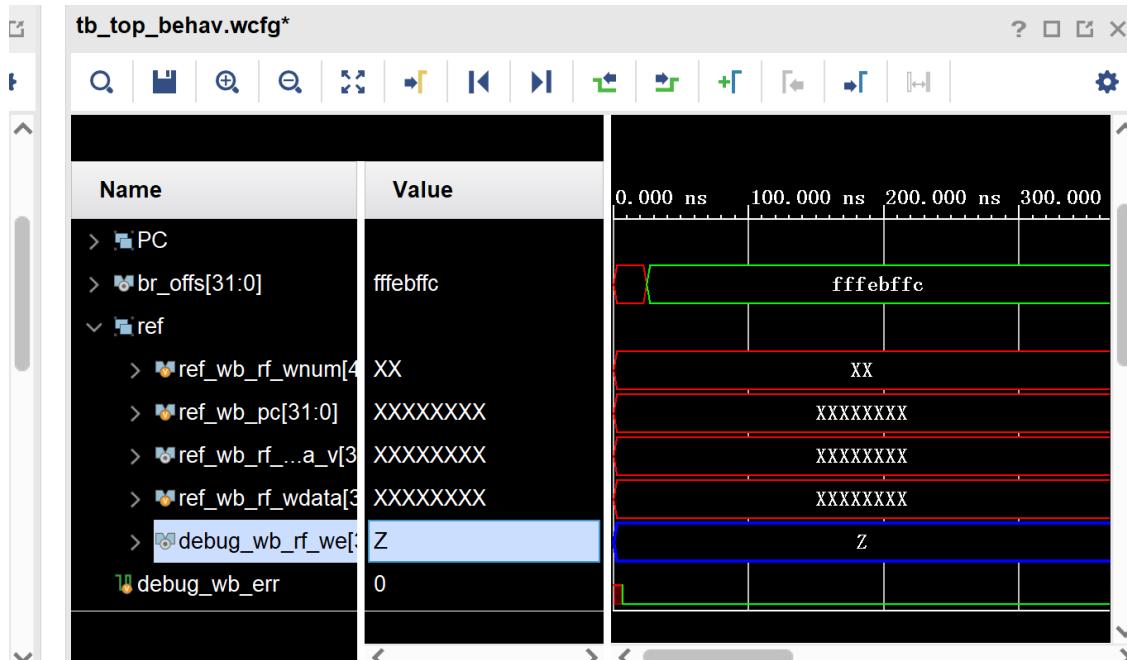
- 根据当前pc值查看反汇编代码：

```

1c0040a0 <inst_error>:
1c0040a0: 0040e2ed slli.w $r13,$r23,0x18
1c0040a4: 001569ac or $r12,$r13,$r26
1c0040a8: 2980030c st.w $r12,$r24,0
1c0040ac: 4c000020 jirl $r0,$r1,0

```

发现已经跑进inst_error里了（不应该呀前面都没报错来着？），再细看对比信号，发现全部未定义：



反观CPU设计代码以及trace比对的代码，再结合波形得知是因为

debug_wb_rf_we 未定义。

```
//compare result in rising edge
always @(posedge soc_clk) //上升沿将 debug 信号与 trace 信号对比
begin
    if(!resetn)
        begin
            debug_wb_err <= 1'b0;
        end
    else if(!debug_wb_rf_we && debug_wb_rf_wnum!=5'd0 && !debug_end && ~CONFREG_OPEN_TRACE //对比时机与采样时机相同
        begin
            if ( (debug_wb_pc!=ref_wb_pc) || (debug_wb_rf_wnum!=ref_wb_rf_wnum) ||(debug_wb_rf_wdata_v!=ref_wb_rf_wdata_v) ) //对比时机与采样时机相同
                begin
                    $display("-----");
                    $display("[%t] Error!!!",$time);
                    $display(" reference: PC = 0x%8h, wb_rf_wnum = 0x%2h, wb_rf_wdata = 0x%8h",
                            ref_wb_pc, ref_wb_rf_wnum, ref_wb_rf_wdata_v);
                    $display(" mycpu : PC = 0x%8h, wb_rf_wnum = 0x%2h, wb_rf_wdata = 0x%8h",
                            debug_wb_pc, debug_wb_rf_wnum, debug_wb_rf_wdata_v);
                    $display("-----");
                    debug_wb_err <= 1'b1; //标记出错
                #40;
            end
        end
    end
end
```

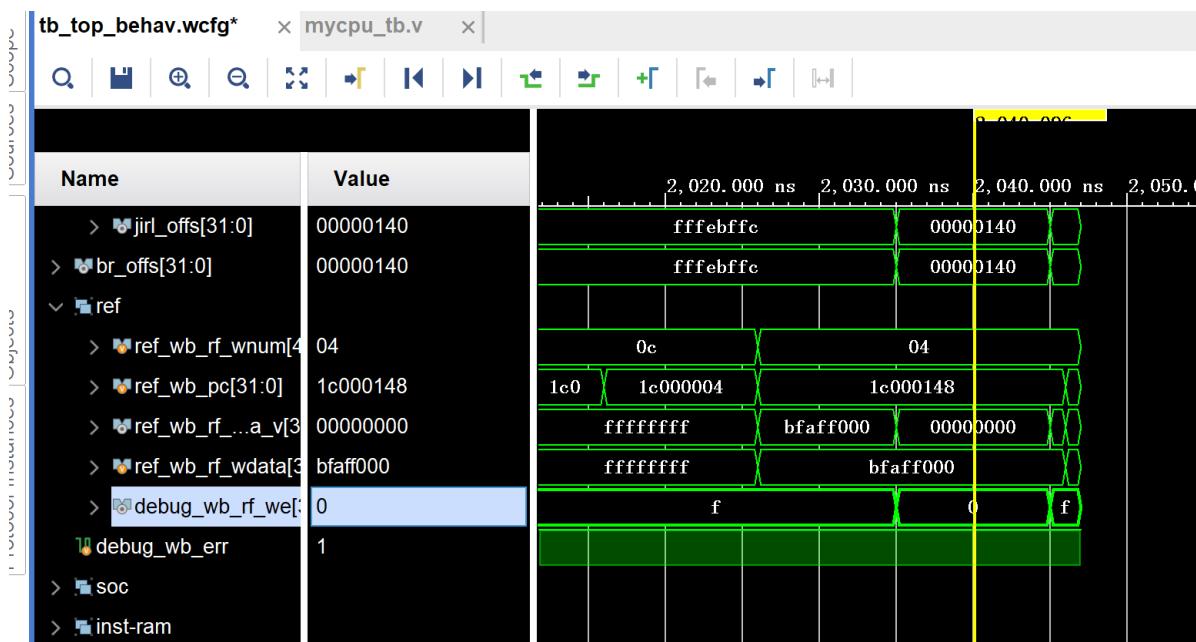
(3) 错误原因

纯纯只是拼写错了，把 debug_wb_rf_we 写成了 debug_wb_rf_wen

```
assign debug_wb_rf_wen = {4{rf_we}};
```

(4) 修正效果

修改上述错误后可正常读取参考信号



- 错误3：信号未定义+1 (拼写错误)

(1) 错误现象

Console报错如下

```
=====
Test begin!
=====
[ 2007 ns] Error!!!
reference: PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
mycpu : PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
=====
WARNING in tb_top.soc_lite.data_ram.inst at time 2045000 ns:
Reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023
```

(2) 分析定位过程

`wb_rf_wdata` 出错，观察其赋值逻辑：

```
'0
'1 assign mem_result = data_sram_rdata;
'2 assign final_result = res_from_mem ? mem_result : alu_result;
'3
'4 assign rf_we = gr_we;
'5 assign rf_waddr = dest;
'6 assign rf_wdata = final_result;
'7
'8 // debug info generate
'9 assign debug_wb_pc = pc;
'0 assign debug_wb_rf_we = {4{rf_we}};
'1 assign debug_wb_rf_wnum = dest;
'2 assign debug_wb_rf_wdata = final_result;
'3
```

定位到 `final_result`。

(3) 错误原因

未定义 `final_result` 的wire类型，排查过去发现一个未使用的 `ms_final_result`。

(4) 修正效果

将 `ms_final_result` 修改为 `final_result` 后继续relaunch，得到新的报错（虽然还在同一个地方，好歹错误变了……）

```
=====
Test begin!
=====
[ 2007 ns] Error!!!
reference: PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
mycpu : PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xfffffff
=====
WARNING in tb_top.soc_lite.data_ram.inst at time 2045000 ns:
```

- 错误4：ALU接口连接错误

(1) 错误现象

如错误3的修正效果处，`wb_rf_data` 值有误。

(2) 分析定位过程

观察反汇编代码，此时指令为

```
addi.w $r12,$r0,-1(0xffff)
```

得知是alu计算结果出错，继续查看alu实现逻辑：

```
alu u_alu(
    .alu_op      (alu_op      ),
    .alu_src1    (alu_src1    ),
    .alu_src2    (alu_src2    ),
    .alu_result  (alu_result)
);

// assign data_sram_en = (rfrom_mem ||
```

(3) 错误原因

如上图，alu的操作数1接口接错，应改接为 `alu_src1`

(4) 修正效果

reference的PC更新到下一步了，可见此bug已经解决。

```
Test begin!
-----
[ 2017 ns] Error!!!
  reference: PC = 0x1c000004, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
  mycpu     : PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
-----
WARNING in tb_top.soc_lite.data_ram.inst at time          2045000 ns:
  . . . . .
```

- 错误5：错用时序逻辑

(1) 错误现象

如错误4中更正接口后的图，console输出显示PC未正常更新。

(2) 分析定位过程

由于PC有误，自然而然查看PC的时序逻辑：

```
always @(posedge clk) begin
    if (reset) begin
        pc <= 32'h1c000000;
    end
    else begin
        pc <= nextpc;
    end
end
```

再寻找reset信号

```
reg      reset;
always @(posedge clk) reset <= ~resetn;
```

(3) 错误原因

重新定义了一个reset，并且使用时序逻辑对之赋值（其会在resetn拉高后下一个时钟上升沿才更新），将导致pc更新滞后一个周期。

(4) 修正方法&效果

上述问题有两种修改方法，一种是直接使用~resetn作为PC重置的条件，一种是将reset改为wire类型变量，并直接使用组合逻辑对之赋值。为少引入一个变量本处采用前一种方法。

- 错误6：代码逻辑有误 (gr_we)

(1) 错误现象

PC报错，初步猜测是转移时PC和reference不一致。

```
running from out of range address. max address in to_top.svc_trace.data痕 first is
-----
[ 2247 ns] Error!!!
reference: PC = 0x1c000198, wb_rf_wnum = 0x01, wb_rf_wdata = 0x1c00019c
mycpu    : PC = 0x1c003cc0, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000001
-----
mycpu    : PC = 0x1c003cc0, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000001
-----
```

(2) 分析定位过程

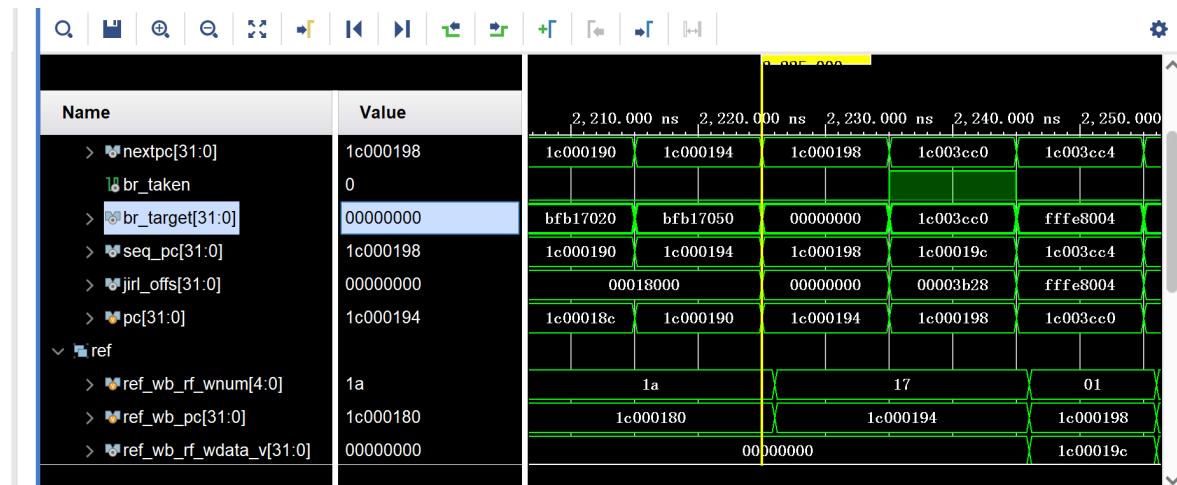
查看此时反汇编代码

```

54  1c000198 <inst_test>;
55  inst_test();
56  1c000198: 543b2800 bl 15144(0x3b28) # 1c003cc0 <n1_Lu12i_w_test>
57  1c00019c: 54010400 bl 260(0x104) # 1c0002a0 <idle_1s>
58  1c0001a0: 541c2000 bl 7200(0x1c20) # 1c001dc0 <n2_add_w_test>

```

得知是bl指令，观察波形如下：



此时确实应该有跳转，但注意到reference的pc不知为何延迟了一个周期才读入，反观其wen信号进行溯源：

```

assign debug_wb_rf_we = {4{rf_we}}; ---->

rf_we = gr_we ---->

assign gr_we = ~inst_st_w & ~inst_beq & ~inst_bne & ~inst_b &
~inst_bl;

```

定位到 gr_we。

(3) 错误原因

bl指令应写寄存器，故gr_we赋值中应去掉 ~inst_bl 一项。

(4) 修正效果

进行上述修改后console输出如下：

```

Reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023
-----
[ 4727 ns] Error!!!
    reference: PC = 0x1c0040a0, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x01000000
    mycpu   : PC = 0x1c0040a0, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000070
-----
[ 4765 ns] Error( 0)!!! Occurred in number 8'd00 Functional Test Point!

```

可见原错误点已经解决

- 错误7：ALU操作数颠倒（移位指令）

(1) 错误现象

如错误6中修正后的输出，`wb_rf_wdata` 有误。

(2) 分析定位过程

查看反汇编代码

```
1c0040a0: 0040e2ed slli.w $r13,$r23,0x18
```

在alu中查看slli.w指令实现：

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

(3) 错误原因

rj中的数据是alu_src1，而移位位数应为alu_src2[4:0]，上述将二者搞反。同时注意到右移操作也将二者搞反，一并修正。

(4) 修正效果

```
[ 4737 ns] Error!!!  
reference: PC = 0x1c0040a4, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x01000001  
mycpu : PC = 0x1c0040a4, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x05000001
```

如图，来到下一个错误点。

- 错误8：组合逻辑环

(1) 错误现象

如错误7中修正后的现象，`wb_rf_wdata` 梅开二度。

(2) 分析定位过程

查看反汇编代码：

```
1c0040a4: 001569ac or $r12,$r13,$r26
```

检查alu的or指令设计：

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

(3) 错误原因

使用 `alu_result` 向 `or_result` 赋值，又将 `or_result` 拉回 `alu_result`，形成组合逻辑环。

(4) 修正效果

改为 `assign or_result = alu_src1 | alu_src2;` 后通关到下一个bug：

```
-----[reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023-----[ 52897 ns] Error!!!  
reference: PC = 0x1c0065a8, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x840418eb  
mycpu : PC = 0x1c0065a8, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x040418eb-----
```

- 错误9：位宽截取有误

(1) 错误现象

如错误8中修正后的现象，`wb_rf_wdata` 再次有误。

(2) 分析定位过程

看反汇编代码：

```
1c0065a8: 0044818a srli.w $r10,$r12,0x0
```

再结合错误8中修正后的console输出，大概率是没处理好算术右移和逻辑右移，查看alu实现：

```
assign sr64_result = {{32{op_sra & alu_src1[31]}}, alu_src1[31:0]} >>  
alu_src2[4:0]; //rj >> i5  
assign sr_result = sr64_result[30:0];
```

(3) 错误原因

`sr_result` 只截取了31位，导致符号位丢失，应改为32位。

(4) 修正效果

Tcl Console × Messages Log

```

Reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023
----[ 121585 ns] Number 8'd20 Functional Test Point PASS!!!
WARNING in tb_top.soc_lite.data_ram.inst at time 121605000 ns:
Reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023
WARNING in tb_top.soc_lite.data_ram.inst at time 121825000 ns:
Reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023
WARNING in tb_top.soc_lite.data_ram.inst at time 121845000 ns:
Reading from out-of-range address. Max address in tb_top.soc_lite.data_ram.inst is 1023
=====
Test end!
----PASS!!!
$finish called at time : 121945 ns : File "D:/Desktop/cdp_ede_local/mycpu_env/soc_verify/soc_dram/testbench/mycpu_tb.v" Line 267
run: Time (s): cpu = 00:00:08 ; elapsed = 00:00:06 . Memory (MB): peak = 970.703 ; gain = 13.586

```

PASS! ! !

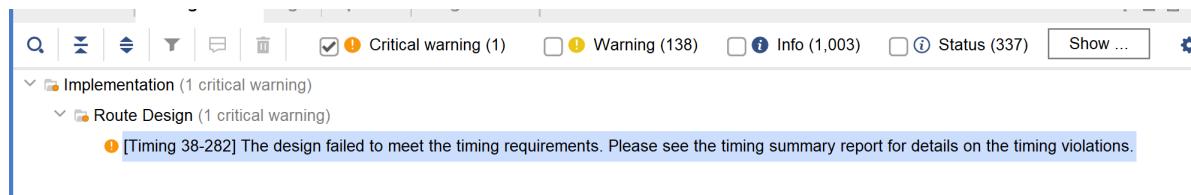
- 错误10：confreg信号重复定义

(1) 错误现象

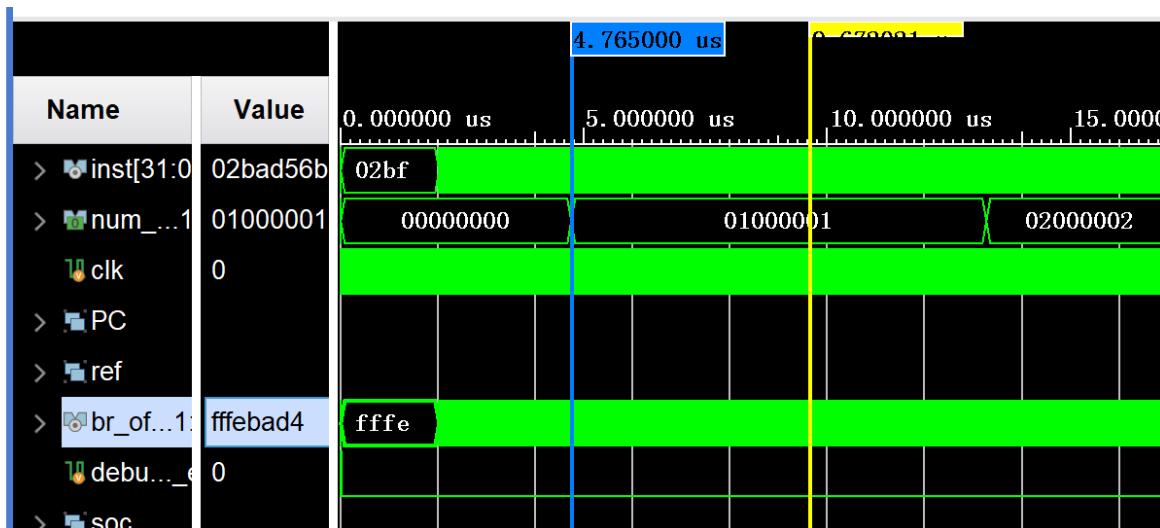
pass后仿真发现违约很严重，如图：

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Fai
✓ synth_1 (active)	constrs_1	synth_design Complete!							
✓ impl_1	constrs_1	write_bitstream Complete!	-1.802	-763.949	0.038	0.000	0.000	0.307	
Out-of-Context Module Runs									
✓ clk_pll_synth_1	clk_pll	synth_design Complete!							
✓ data_ram_synth_1	data_ram	synth_design Complete!							

(2) 分析定位过程



依据critical warning查看时序报告，未看到有效定位信息，上板强行跑一发，没有示数。查看显示信号 num_data，波形正常：

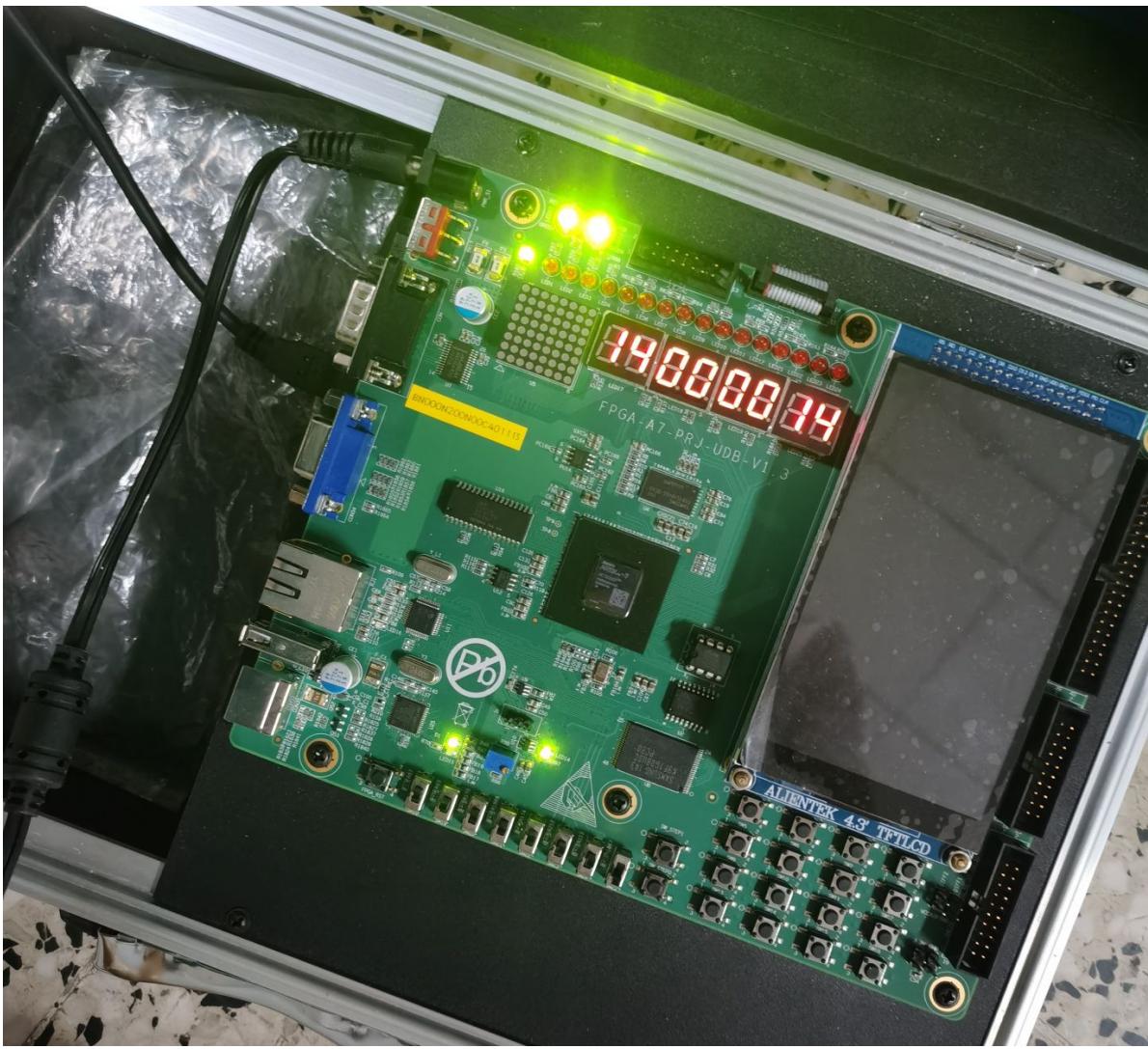


实在摸不着头脑，再查看confreg里的具体实现，发现信号重复定义：

```
    output reg  [6 :0] num_a_g,
    output reg  [31:0] num_data,
    input  wire [7 :0] switch,
    output wire [3 :0] btn_key_col,
    input  wire [3 :0] btn_key_row,
    input  wire [1 :0] btn_step
);
    reg  [31:0] cr0;
    reg  [31:0] cr1;
    reg  [31:0] cr2;
    reg  [31:0] cr3;
    reg  [31:0] cr4;
    reg  [31:0] cr5;
    reg  [31:0] cr6;
    reg  [31:0] cr7;

    reg  [31:0] led_data;
    reg  [31:0] led_rg0_data;
    reg  [31:0] led_rg1_data;
    reg  [31:0] num_data;
    wire [31:0] switch_data.
```

(3) 修改效果



上板可正常工作。

四.实验总结

整体实验感受：

在debug的时候结合console输出和反汇编代码效率基本可以定位大部分错误，还有少数很坑的错误很难注意到，比方说移位时ALU的两个操作数改反，还有那个confreg的重复定义是真的难受（是不是应该稍微提醒一下除了myCPU之外的文件也需要修改？）。

接下来是几点建议：

- 感觉上非常多的拼写错误，实际上大多数都涉及后续流水线CPU的设计。起初那些信号定义让人很困惑，后续阅读了流水线CPU部分的讲义才知道某些信号的含义。

个人认为如果是想设置bug理应把当前实验和后续实验的界限划分开。如某些同学在debug时不是采取注释掉错误代码的方式，而是直接将某些信号（如 `fs_pc` 以及 `ds_pc`）删去，后续写流水线CPU就可能会比较麻烦；

- 组合逻辑环的设置有些牵强（把 `alu_result|alu_src1|alu_src2` 赋值给 `or_result`），感觉这种情况肉眼debug的效率会比看波形的稍高；

- 环境更新可以再及时一点，最好能在学生开始实验前完成。否则在debug出了部分错误后，每次有更新都需手动找出更新的文件并替换。