# 实验 7 报告

学号：2017K8009922026，2017K8009922032
姓名：康齐瀚,赵鑫浩
箱子号：63

## 一、实验任务（10%）

在五级流水线 CPU 中继续添加 BGEZ，BGTZ，BLEZ 等条件转移指令以及 LB，LBU，SB 等访存指令

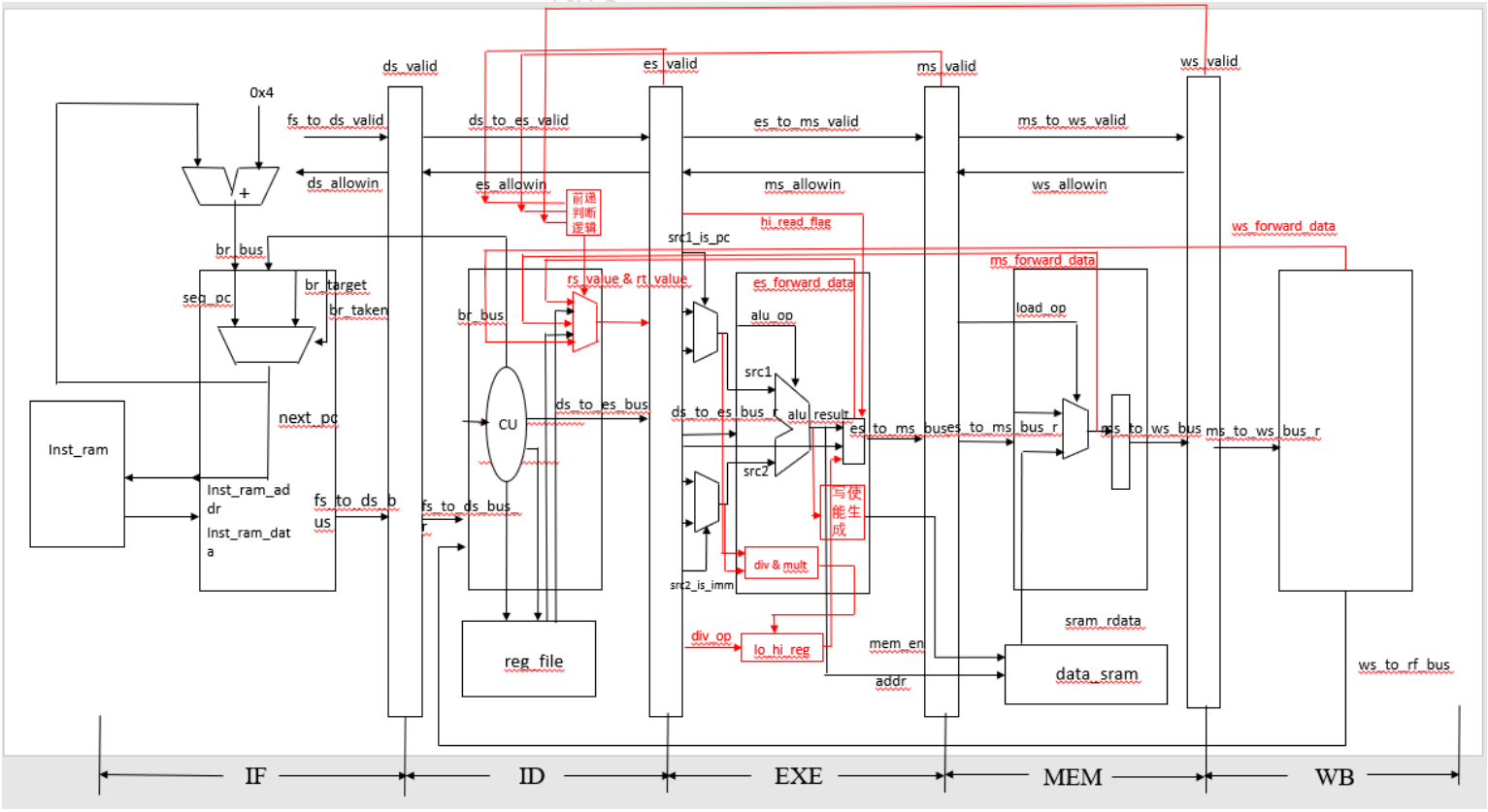
## 二、实验设计（40%）

### （一）总体设计思路

BGEZ，BGTZ 等条件转移指令参考之前 BNE 等指令的设计。在译码级进行译码和条件转移判断并计算出后续 PC，通过 br_bus 信号送回到 IF 级。 BLTZAL 和 BGEZAL 需要在 EXE 级用 ALU 计算 PC+8 的值并写回 31 号寄存器。

访存指令在 EXE 级进行访存时需要将计算地址的最低两位变为零，同时向 MEM 级传递最低两位的信息。如果当前指令是 Store 型指令，则需要根据访存地址的最后两位设置 data_sram_wen 信号。如果是 LWL 和 LWR 指令，则在 MEM 阶段进行拼接和拓展。

实验设计图：



## （二）重要模块 1 设计：跳转指令

### 1、工作原理

译码之后各种跳转指令根据描述来确定是否跳转，跳转的地址以及写使能信号。

### 2、功能描述

使用 rs_less_zero，rs_more_zero 信号分别表示 rs 寄存器中的值是否小于 0 或大于 0。bgezal 指令和 bgez 指令需要判断 rs 寄存器的值是否大于等于 0，这不需要额外增加其他的判断信号了，使用 rs_less_zero 的非即可

```
assign br_taken = (   inst_beq  &&  rs_eq_rt
                   || inst_bne  && !rs_eq_rt
                   || inst_bgez  && !rs_less_zero
                   || inst_bgtz  && rs_more_zero
                   || inst_blez  && !rs_more_zero
                   || inst_bltz  && rs_less_zero
                   || inst_bltzal  && rs_less_zero
                   || inst_bgezal  && !rs_less_zero
                   || inst_jal
                   || inst_jalr
                   || inst_j
                   || inst_jr
                  ) && ds_valid;
```

br_target 信号给出了跳转的目标地址，本次实验需要对新增的指令进行新的设置：

```
assign br_target = (inst_beq || inst_bne || inst_bgez || inst_bgtz || inst_blez || inst_bltz || inst_bltzal || inst_bgezal) ? (fs_pc + {{14{imm[15]}}, imm[15:0], 2'b0}) :
                   (inst_jr| inst_jalr)                                                                                    ? rs_value :
                  /*inst_jal \ inst_j  */                {fs_pc[31:28], jidx[25:0], 2'b0};
```

## （三）重要模块 2 设计：LOAD 模块

### 1、工作原理

通过拼接操作确定各种 load 指令的写入数，之后由多路选择器判断后写入寄存器。

### 2、功能描述

addr_low 信号表示访存地址的最低两位。 在设置写回寄存器的数据时，先对 lb，lh 等指令分别产生 lb_result 和 lh_result 数据，这里区分是否无符号拓展是通过设置 sign 信号得到的。lb_result 和 lh_result 的高位先根据符号拓展的情况设置为全 0 或全 1，再对其低位进行填充，填充的方式是根据 addr_low 的情况做对应的对 data_sram_rdata 信号的截取。

lwr_result 和 lwl_result 的获得需要用 addr_low 信号判断 data_sram_rdata 的截取位置，然后和 rt 寄存器的数据进行拼接。因为这里采用的是第二种设计方法，通过拼接即可。

```
assign lwl_result=(addr_low==2'b00)? {data_sram_rdata[7:0],ms_rt_value[23:0]} :
          (addr_low==2'b01)? {data_sram_rdata[15:0],ms_rt_value[15:0]} :
          (addr_low==2'b10)? {data_sram_rdata[23:0],ms_rt_value[7:0]} :
          (addr_low==2'b11)? data_sram_rdata : 0;

assign lwr_result=(addr_low==2'b00)? data_sram_rdata :
          (addr_low==2'b01)? {ms_rt_value[31:24],data_sram_rdata[31:8]} :
          (addr_low==2'b10)? {ms_rt_value[31:16],data_sram_rdata[31:16]} :
          (addr_low==2'b11)? {ms_rt_value[31:8],data_sram_rdata[31:24]} : 0;

assign addr_low = ms_alu_result[1:0];
assign sign = ms_lb | ms_lh ;
assign lb_result[7:0]=(addr_low==2'b00)? data_sram_rdata[7:0] :
                      (addr_low==2'b01)? data_sram_rdata[15:8] :
                      (addr_low==2'b10)? data_sram_rdata[23:16] :
                      (addr_low==2'b11)? data_sram_rdata[31:24] : 0;

assign lb_result[31:8] = (sign==1)? ((lb_result[7]==1)? 24'hffffff : 24'h000000 )  : 24'h000000;

assign lh_result[15:0]=(addr_low==2'b00)? data_sram_rdata[15:0] :
                       (addr_low==2'b01)? data_sram_rdata[23:8] :
                       (addr_low==2'b10)? data_sram_rdata[31:16] : 0;
assign lh_result[31:16]=(sign==1)? ((lh_result[15]==1)? 16'hffff : 16'h0000 )  : 16'h0000;
```

## （四）重要模块 3 设计：STORE 模块

### 1、工作原理

通过拼接操作确定各种 store 指令的写入数，之后由多路选择器判断后写入寄存器，并判断 4 位写使能信号的值。

### 2、功能描述

store 型的指令统一在 EXE 阶段进行写入，写入数据的生成与 lw 型指令类似，都是通过 addr 的最低两位做相应的写入操作，此处不再赘述。

值得注意的是，store 型指令有写使能信号需要赋值，具体规则是根据 4 种不同的 store 型指令中不同的 addr 低两位地址进行赋值。这里统一处理如果所有条件都不满足的话就将写使能设为 0(可能当前指令非 store 型)

```
assign address_low = es_alu_result[1:0];

assign sb_result = {4{es_rt_value[7:0]}};
assign sh_result = {2{es_rt_value[15:0]}};

assign swl_result=  (address_low==2'b00)? {24'b0,es_rt_value[31:24]} :
        (address_low==2'b01)? {16'b0,es_rt_value[31:16]} :
        (address_low==2'b10)? {8'b0,es_rt_value[31:8]} :
        (address_low==2'b11)? es_rt_value : 0;

assign swr_result=  (address_low==2'b00)? es_rt_value :
        (address_low==2'b01)? {es_rt_value[23:0],8'b0} :
        (address_low==2'b10)? {es_rt_value[15:0],16'b0} :
        (address_low==2'b11)? {es_rt_value[7:0],24'b0} : 0;
```

```
assign sb_byte_wen      = (vaddr == 2'b00) ? 4'h1 :
                          (vaddr == 2'b01) ? 4'h2 :
                          (vaddr == 2'b10) ? 4'h4 :
                          (vaddr == 2'b11) ? 4'h8 : 4'h0;

assign sh_byte_wen      = (vaddr == 2'b00) ? 4'h3 :
                          (vaddr == 2'b10) ? 4'hc : 4'h0;

assign swl_byte_wen     = (vaddr == 2'b00) ? 4'h1 :
                          (vaddr == 2'b01) ? 4'h3 :
                          (vaddr == 2'b10) ? 4'h7 :
                          (vaddr == 2'b11) ? 4'hf : 4'h0;

assign swr_byte_wen     = (vaddr == 2'b00) ? 4'hf :
                          (vaddr == 2'b01) ? 4'he :
                          (vaddr == 2'b10) ? 4'hc :
                          (vaddr == 2'b11) ? 4'h8 : 4'h0;

assign byte_wen         = ({4{op_sw}} & sw_byte_wen)
                        | ({4{op_sb}} & sb_byte_wen)
                        | ({4{op_sh}} & sh_byte_wen)
                        | ({4{op_swl}} & swl_byte_wen)
                        | ({4{op_swr}} & swr_byte_wen);
```



# 三、实验过程（50%）

## （一）实验流水账

2019/10/17 13：00 ~ 17：00　写完代码并仿真成功

2019/10/20 16：00 ~ 18：00　开始撰写实验报告

## （二）错误记录

1、错误 1：PC 对比错误

（1）错误现象

与 trace 文件对比时 PC，写回寄存器编号，写回数据完全错误。

（2）分析定位过程

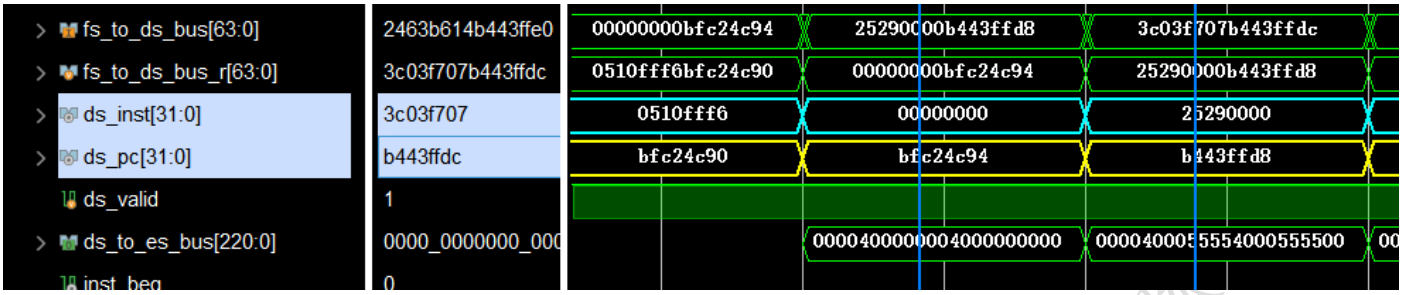
```
[1266147 ns] Error!!!
    reference: PC = 0xbfc24c6c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfc24c98
    mycpu    : PC = 0xb443ffd8, wb_rf_wnum = 0x09, wb_rf_wdata = 0x0000aaaa
```

PC 对比出错，说明这条指令完全执行了但是指令地址不对。考虑是某一条跳转指令未执行或条件判断错误。

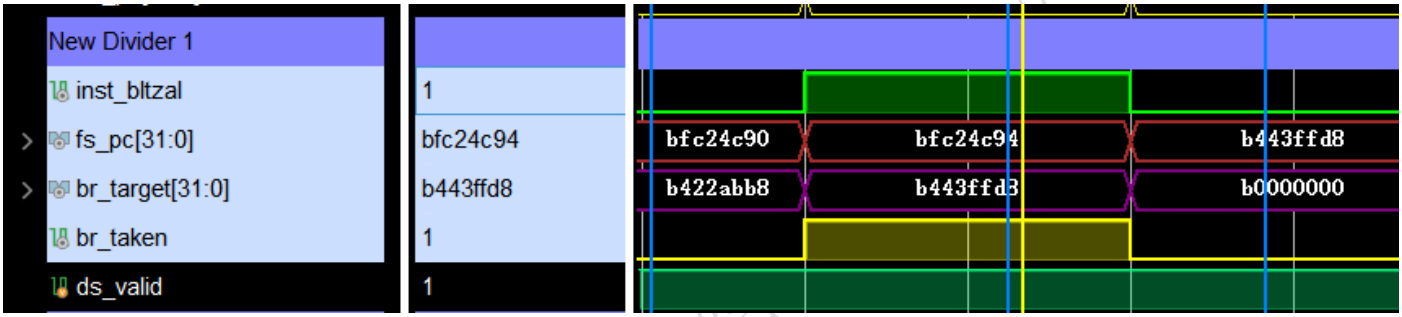
检查 test.s 中 0xbfc24c6c 发现该位置是一条 move 指令。然而检查出错的 0xb443ffd8 位置却发现 test.S 中根本

没有该地址。

从出错位置的波形向前寻找：




出错指令前的一条指令是 0xbfc24c94。 该位置是一条 nop 指令，很可能是延迟槽。再往前面看，发现 0xbfc24c90 位置是一条 bltzal 指令，可以推断是这条指令跳转发生了错误。




可以看出在该位置确定是 bltzal 指令并且也确定了要跳转。但是跳转目标地址错误了。查看代码发现在跳转地址赋值位置有问题：

```
assign br_target = (inst_beq || inst_bne || inst_bgez || inst_bgtz || inst_blez || inst_bltz) ?
                                                    (fs_pc + {{14{imm[15]}}, imm[15:0], 2'b0}) :
                   (inst_jr   || inst_jalr)              ? rs_value :
                  /*inst_jal*/              {fs_pc[31:28], jidx[25:0], 2'b0};
```

在拼接 PC 时漏掉了 inst_bltzal 和 inst_bgezal 的情况。

（3）错误原因

跳转目标地址计算出错

（4）修正效果

将 inst_bltzal 和 inst_bgezal 归为第一类跳转地址拼接情况即可。从仿真效果来看，这个办法是可行的。

## 2、错误 2：指令写回寄存器错误

（1）错误现象

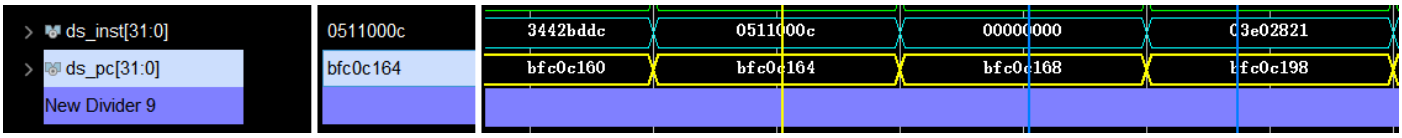
与 trace 文件对比时发现某条指令的写回寄存器编号与写回数据错误。

（2）分析定位过程

与上一个错误类似，不过这次对比出错的 PC 与 trace 文件给出的 PC 很近。

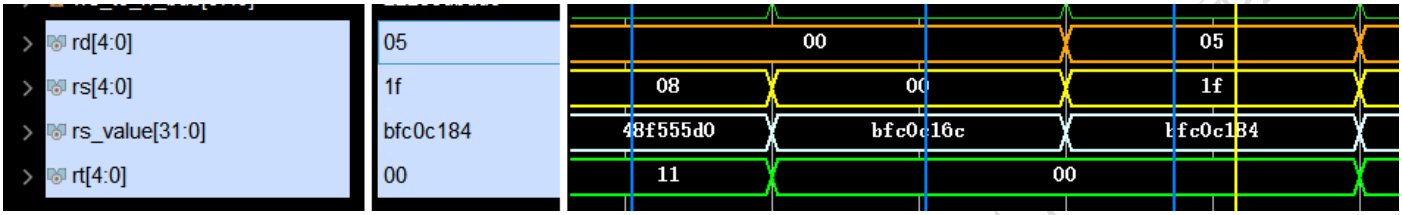
```
[1304507 ns] Error!!!
    reference: PC = 0xbfc0c164, wb_rf_wnum = 0x1f, wb_rf_wdata = 0xbfc0c16c
    mycpu    : PC = 0xbfc0c198, wb_rf_wnum = 0x05, wb_rf_wdata = 0x7f8182f0
```

从 test.S 查到 0xbfc0c164 位置的指令是 bgezal，跳转目标是 0xbfc0c198。

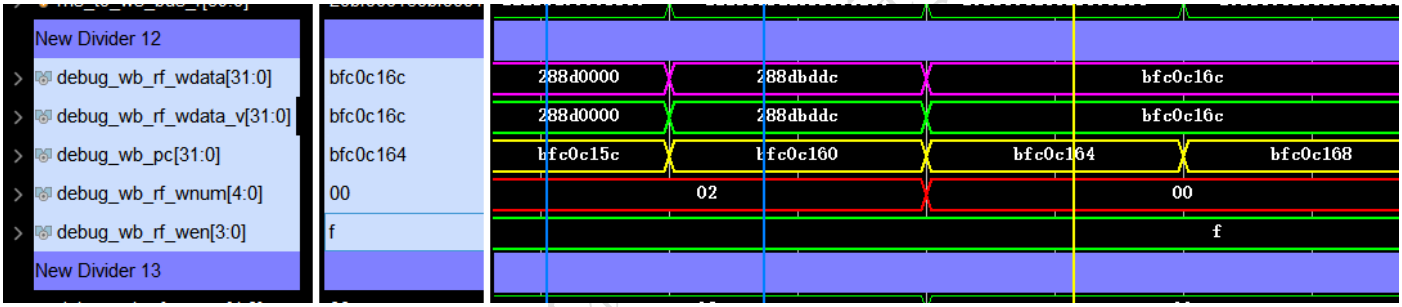



从波形图上来看，这次跳转确实是完成了。

0xbfc0c198 位置的指令是 move a1, ra，是将 ra 寄存器中的数据移动到 a1 寄存器中。




从波形图看出此时从 ra 寄存器中读出的值为 0xbfc0c184. 但理论上写入的应该是 0xbfc0c164+0x8=0xbfc0c16c。因此考虑是 bgezal 指令写回有问题。




在 0xbfc0c164 位置的 bgezal 指令的写回阶段，发现写回寄存器号错误了。考虑是译码阶段没有正确设置。

查看代码发现确实是这样：

```
assign res_from_mem = inst_lw;
assign dst_is_r31   = inst_jal | inst_bltzal;
```

（3）错误原因

bgezal 的写回寄存器号出错，dst_is_r31 信号没有在当前指令是 bgezal 时进行设置

（4）修正效果

在 dst_is_r31 中增加 bgezal 情况时的判断条件。添加后，仿真能够正常通过这条指令的测试。

## 3、错误 3：PC 对比错误

（1）错误现象

与 trace 文件对比时 PC，写回寄存器编号，写回数据完全错误。

（2）分析定位过程

PC 对比出错，从 test.s 中看出是一条简单的 j 指令，跳转地址也完全正确，但是注意到写入寄存器的数发生错误，最后发现未修改该条指令的写使能信号导致错误。

（3）修正效果

修改写使能信号，使得 j 指令时信号的值为 0，通过了仿真。

## 四、实验总结

通过 6，7 两次实验，掌握了向五级流水线中添加指令的方法的一般化方法。通过两次实验观察波形图和定位错误的过程，找 bug 并修复 bug 的能力也得到了很大的提高。对流水线的运行过程有了更深的体会。