Lab 5 报告

学号 2020K8009929013
姓名史文轩、安金鹏、陈卓
箱子号 69

一、实验任务（10%）

在已有的简单流水线 CPU 上添加更多的普通用户态指令。具体包括:

- 算术逻辑运算类指令 slti, sltui, andi, ori, xori, sll, srl, sra, pcaddu12i;
- 乘除运算类指令 mul.w, mulh.w, mulh.wu, div.w, mod.w, div.wu, mod.wu;
- 转移指令 blt, bge, bltu, bgeu;
- 访存指令 ld.b, ld.h, ld.bu, ld.hu, st.b, st.h.

二、实验设计（40%）

（一）设计思路

1.算术逻辑运算类指令：

添加这几条算数逻辑指令比较简单，主要是在译码阶段作修改。

首先是添加指令在译码阶段的标识信号如下图：

```
assign inst_slti   = op_31_26_d[6'h00] & op_25_22_d[4'h8];
assign inst_sltui  = op_31_26_d[6'h00] & op_25_22_d[4'h9];
assign inst_addi_w = op_31_26_d[6'h00] & op_25_22_d[4'ha];
assign inst_andi   = op_31_26_d[6'h00] & op_25_22_d[4'hd];
assign inst_ori    = op_31_26_d[6'h00] & op_25_22_d[4'he];
assign inst_xori   = op_31_26_d[6'h00] & op_25_22_d[4'hf];
```

然后根据指令标识，修改 alu_op

```
235
236 assign alu_op[ 0] = inst_add_w | inst_addi_w | inst_ld_w | inst_st_w
237                  | inst_jirl | inst_bl | inst_pcaddu12i;
238 assign alu_op[ 1] = inst_sub_w;
239 assign alu_op[ 2] = inst_slt | inst_slti;
240 assign alu_op[ 3] = inst_sltu | inst_sltui;
241 assign alu_op[ 4] = inst_and | inst_andi;
242 assign alu_op[ 5] = inst_nor ;
243 assign alu_op[ 6] = inst_or | inst_ori;
244 assign alu_op[ 7] = inst_xor | inst_xori;
245 assign alu_op[ 8] = inst_slli_w |inst_sll_w;
246 assign alu_op[ 9] = inst_srli_w | inst_srl_w;
247 assign alu_op[10] = inst_srai_w | inst_sra_w;
248 assign alu_op[11] = inst_lu12i_w;
```

由于新添加的运算逻辑指令涉及到新的零扩展的立即数，因此，下面要修改立即数相关的信号：

首先添加了新的立即数类型 ui12,是对 12 位立即数作无符号扩展:

```
assign need_ui12  =  inst_andi | inst_ori | inst_xori;
assign imm = src2_is_4 ? 32'h4                      :
             need_si20 ? {i20[19:0], 12'b0}         :
             need_ui12 ? {20'b0,i12 }               :
/*need_ui5 || need_si12*/{{20{i12[11]}}, i12[11:0]} ;
```

接下来修改两个操作数相关的控制信号:

```
assign src1_is_pc    = inst_jirl | inst_bl | inst_pcaddu12i;

assign src2_is_imm   = inst_slli_w | inst_slti |
                       inst_srli_w | inst_sltui |
                       inst_srai_w | inst_andi  |
                       inst_addi_w | inst_ori   |
                       inst_ld_w   | inst_xori  |
                       inst_st_w   | inst_pcaddu12i |
                       inst_lu12i_w|
                       inst_jirl   |
                       inst_bl     ;
```

至此，译码阶段的工作就结束了，新添加的相应的立即数，以及控制信号，可以按照原来的数据通路传到执行阶段，其他阶段的逻辑无需更改。

2. 乘除运算类指令:

相比于上面的，乘除取模指令的实现比较复杂。首先要更改的是译码阶段的生成信号，这里不再赘述。由于这 7 条指令所作的计算都不是 ALU 进行计算的，所以就把这些指令的判断标识传到了执行阶段，来做相应的控制信号:

```
assign {
      es_inst_mul_w,
      es_inst_mulh_w,
      es_inst_mulh_wu,
      es_inst_div_w,
      es_inst_mod_w,
     es_inst_div_wu,
     es_inst_mod_wu,
     es_alu_op       ,
     es_load_op      ,
     es_src1_is_pc   ,
     es_src2_is_imm  ,
     es_gr_we        ,
     es_mem_we       ,
     es_dest         ,
     es_imm          ,
     es_rj_value     ,
     es_rkd_value    ,
     es_pc
} = ds_to_es_data_r;
```

首先是乘法的设计，选用最简单的调用 Xilinx IP 实现乘法运算部件的方法，如下图，实现了无符号和有符号乘

法：

```
assign mul_src1 = es_rj_value;
assign mul_src2 = es_rkd_value;
assign unsigned_mul_res = mul_src1 * mul_src2;
assign signed_mul_res   = $signed(mul_src1) * $signed(mul_src2);
```

接下来是除法和取模，我们按照讲义上的方法生成了两个 IP 核：

```
u_unsigned_div u_unsigned_divider (
    .aclk                   (clk),
    .s_axis_dividend_tdata  (divider_dividend),
    .s_axis_dividend_tready (unsigned_dividend_tready),
    .s_axis_dividend_tvalid (unsigned_dividend_tvalid),
    .s_axis_divisor_tdata   (divider_divisor),
    .s_axis_divisor_tready  (unsigned_divisor_tready),
    .s_axis_divisor_tvalid  (unsigned_divisor_tvalid),
    .m_axis_dout_tdata      (unsigned_divider_res),
    .m_axis_dout_tvalid     (unsigned_dout_tvalid)
);

u_signed_div u_signed_divider (
    .aclk                   (clk),
    .s_axis_dividend_tdata  (divider_dividend),
    .s_axis_dividend_tready (signed_dividend_tready),
    .s_axis_dividend_tvalid (signed_dividend_tvalid),
    .s_axis_divisor_tdata   (divider_divisor),
    .s_axis_divisor_tready  (signed_divisor_tready),
    .s_axis_divisor_tvalid  (signed_divisor_tvalid),
    .m_axis_dout_tdata      (signed_divider_res),
    .m_axis_dout_tvalid     (signed_dout_tvalid)
);
```

接下来是两个除法器的控制信号设计，由于有符号和无符号的控制信号设计逻辑一样，所以这里就以无符号设计为例：

如下图所示，是除数和被除数的 valid 信号的设计逻辑，指令有效之外，还设计了两个 sent 信号，这两个 sent 信号由下面的 always 块控制，意思是，一旦除法器的输入信号握手成功，那么除数和被除数的 valid 信号就要变成无效，不然除法器还会继续进行除法运算。

```
reg  unsigned_dividend_sent;
reg  unsigned_divisor_sent;

assign unsigned_dividend_tvalid = es_valid && (es_inst_div_wu | es_inst_mod_wu) && !unsigned_dividend_sent;
assign unsigned_divisor_tvalid = es_valid &&( es_inst_div_wu | es_inst_mod_wu) && !unsigned_divisor_sent;

always @ (posedge clk) begin
    if (reset) begin
        unsigned_dividend_sent <= 1'b0;
    end else if (unsigned_dividend_tready && unsigned_dividend_tvalid) begin
        unsigned_dividend_sent <= 1'b1;
    end else if (es_ready_go && ms_allowin) begin
        unsigned_dividend_sent <= 1'b0;
    end

    if (reset) begin
        unsigned_divisor_sent <= 1'b0;
    end else if (unsigned_divisor_tready && unsigned_divisor_tvalid) begin
        unsigned_divisor_sent <= 1'b1;
    end else if (es_ready_go && ms_allowin) begin
        unsigned_divisor_sent <= 1'b0;
    end
end
```

由于除法器计算所需周期不确定，因此在计算结果出来，也就是信号 dout_tvalid 有效之前，流水线要阻塞，这里的设计如下图，直接用 dout_tvalid 信号来生成 es_readygo 的信号：

```
assign es_ready_go    =
    (es_inst_div_w|es_inst_mod_w)     ? signed_dout_tvalid  :
    (es_inst_div_wu|es_inst_mod_wu)    ? unsigned_dout_tvalid :
    1'b1;
```

再添加完除法和乘法指令的运算之后，那么计算结果就来自不同的器件，需要进行数据选择：

下图是三种运算的生成的数据和选择控制信号的设计：

```
assign reg_mul_rdata = es_inst_mul_w ? signed_mul_res[31:0]  :
                       es_inst_mulh_w? signed_mul_res[63:32]  :
                       /*es_inst_mulh_wu*/  unsigned_mul_res[63:32];
assign reg_div_rdata =  es_inst_div_w ?  signed_divider_res[63:32] :
                                        unsigned_divider_res[63:32];
assign reg_mod_rdata = es_inst_mod_w ?   signed_divider_res[31:0]:
                                        unsigned_divider_res[31:0];
assign es_res_from_mul = es_inst_mul_w | es_inst_mulh_w | es_inst_mulh_wu ;
assign es_res_from_mod = es_inst_mod_w | es_inst_mod_wu;
assign es_res_from_div =es_inst_div_w | es_inst_div_wu;
```

然后再跟原来的 ALU 的输出加一个数据选择器，作为执行阶段的最终的数据：

```
assign es_exe_result =
    es_res_from_mul  ? reg_mul_rdata  :
    es_res_from_div  ? reg_div_rdata  :
    es_res_from_mod  ? reg_mod_rdata  :
    es_alu_result;
```

由于作除法运算的时候，指令阻塞再执行阶段，所以数据前递的部分不用作修改，当阻塞结束的时候，前面前递的错误的数据会被相应的覆盖掉。

3. 转移类指令：

转移类指令的添加相对来说比较简单，首先是在译码阶段加上相关的指令信号：

```
assign inst_jirl   = op_31_26_d[6'h13];
assign inst_b      = op_31_26_d[6'h14];
assign inst_bl     = op_31_26_d[6'h15];
assign inst_beq    = op_31_26_d[6'h16];
assign inst_bne    = op_31_26_d[6'h17];
assign inst_blt    = op_31_26_d[6'h18];
assign inst_bge    = op_31_26_d[6'h19];
assign inst_bltu   = op_31_26_d[6'h1a];
assign inst_bgeu   = op_31_26_d[6'h1b];
```

由于此次添加的转移指令（上图的下面四条）与之前的已经存在的 beq 和 bne 指令数据通路几乎是完全一样的，因而我们可以大规模复用原有的数据通路，并在 src_reg_is_rd 信号等与跳转指令相关的控制信号处添加上新的几条跳转指令。

与之前的转移指令不同的地方在于跳转条件发生了变化，由于要解决控制相关，我们需要在译码阶段就完成对跳转条件的判断，因而我们还要针对新添加的这些跳转指令在译码阶段加上新的跳转条件判断模块，如下：

```
wire [32:0] sub_result;
wire        rj_lt_rd;
wire        rj_ltu_rd;

assign sub_result = {1'b0, rj_value} + {1'b0, ~rkd_value} + 1'b1;


// SLT result
assign rj_lt_rd = (rj_value[31] & ~rkd_value[31])
                | ((rj_value[31] ~^ rkd_value[31]) & sub_result[31]);

// SLTU result
assign rj_ltu_rd = ~sub_result[32];
```

这里要判断有符号数和无符号数的大小，我采用了与提供的 ALU 模块类似的方法，即用补码加法模拟减法，再用减法结果和操作数的最高位的逻辑组合判断比较的结果，这种方法相对来说需要硬件资源不多，运算也较快。

3. 访存类指令：

相对来说，访存类的指令比较复杂，首先是 store 类指令，在译码阶段，我们要做的是修改部分需要传送到之后级流水线的控制信号，比如决定 exe 阶段是否是 load 类指令从而影响前递的 load_op 信号和是否写回寄存器的 gr_we 信号等，由于这些控制信号比较多且很多逻辑上比较简单，在此不一一赘述，只截取部分控制信号的修改部分供举例说明：

```
-assign src2_is_imm   = inst_slli_w | inst_slti |
-                       inst_srli_w | inst_sltui |
-                       inst_srai_w | inst_andi  |
-                       inst_addi_w | inst_ori   |
-                       inst_ld_w   | inst_xori |
-                       inst_st_w   | inst_pcaddu12i |
+assign src2_is_imm   = inst_slli_w | inst_slti | inst_st_b |
+                       inst_srli_w | inst_sltui | inst_ld_hu |
+                       inst_srai_w | inst_andi  | inst_ld_h |
+                       inst_addi_w | inst_ori   | inst_ld_bu |
+                       inst_ld_w   | inst_xori | inst_ld_b |
+                       inst_st_w   | inst_pcaddu12i | inst_st_h |
                        inst_lu12i_w|
                        inst_jirl   |
                        inst_bl     ;
-assign load_op       = inst_ld_w;
-assign res_from_mem  = inst_ld_w;
+assign load_op       = inst_ld_w | inst_ld_b | inst_ld_bu | inst_ld_hu | inst_ld_h;
+assign res_from_mem  = load_op;
 assign dst_is_r1     = inst_bl;
-assign gr_we         = ~inst_st_w & ~inst_beq & ~inst_bne & ~inst_b;
-assign mem_we        = inst_st_w;
+assign gr_we         = ~inst_st_w & ~inst_beq & ~inst_bne & ~inst_b & ~inst_blt &
+                       ~inst_st_b & ~inst_st_h & ~inst_bltu & ~inst_bge & ~inst_bgeu;
+assign mem_we        = inst_st_w | inst_st_b | inst_st_h;
```

为了后续流水线的设计，我们还要把访存类指令的指令类型信号传到下一级。

在执行阶段，由于要在这个阶段完成对数据存储器接口信号的赋值，因而指定了要往数据存储器中写入什么数据以及写入哪几个字节的 store 指令就需要在这个阶段进行处理。我设计的 store 指令处理模块如下：

```
// store
wire [1:0] st_addr;
wire [3:0] st_op;
wire [3:0] st_b_op;
wire [3:0] st_h_op;
wire [3:0] st_w_op;
wire [31:0] st_data;

assign st_addr = es_alu_result[1:0];
assign st_b_op = {4{(!st_addr[0] & !st_addr[1])}} & 4'b0001 |
                 {4{( st_addr[0] & !st_addr[1])}} & 4'b0010 |
                 {4{(!st_addr[0] &  st_addr[1])}} & 4'b0100 |
                 {4{( st_addr[0] &  st_addr[1])}} & 4'b1000 ;
assign st_h_op = (!st_addr[0] & !st_addr[1]) ? 4'b0011 : 4'b1100;
assign st_w_op = 4'b1111;

assign st_op   = {4{es_inst_st_b}} & st_b_op |
                 {4{es_inst_st_h}} & st_h_op |
                 {4{es_inst_st_w}} & st_w_op ;

assign st_data = es_inst_st_b ? {4{es_rkd_value[7:0]}} :
                 es_inst_st_h ? {2{es_rkd_value[15:0]}}:
                                es_rkd_value[31:0] ;


assign data_sram_we    = (es_valid & es_mem_we) ? st_op : 4'h0;
assign data_sram_addr  = es_alu_result;
assign data_sram_wdata = st_data;
```

利用地址的后两位对不同 store 指令下要写入的字节（we 信号）进行多路选择，然后针对不同的 store 指令再进行一次多路选择，并用这个选择出来的写入信号当作 we 信号。而写入信号则可以把需要的字节直接复写到整个

wdata 中，这样选择的时候只通过 we 信号就可以选到需要的字节部分。

然后是 load 类指令。我们同样需要从译码阶段开始就把 load 类指令的类型信号一级一级往后传，由于 load 的内容是在访存阶段从数据存储器中取出，因此 load 类指令需要在访存阶段处理。这里的处理方式跟 store 指令的处理方式比较相像，都是采用了对不同的 load 指令分别进行四字节的多路选择，最后再用一个针对指令本身的多路选择得到最终的 load 内容。

在这里的设计中值得一提的是，我们对 ld_b 和 ld_bu，以及 ld_h 和 ld_hu 两组指令进行了选择的归一，这样在进行多路选择的时候可以节省部分资源。具体的设计如下：

```
// load
wire [1:0] ld_addr;
wire [7:0] mem_byte;
wire [15:0] mem_half;
wire [31:0] mem_word;
wire [31:0] ld_b_res;
wire [31:0] ld_h_res;
wire [31:0] ld_w_res;

assign ld_addr = ms_exe_result[1:0];
assign mem_byte = {8{!ld_addr[0] & !ld_addr[1]}} & data_sram_rdata[7:0]   |
                  {8{ ld_addr[0] & !ld_addr[1]}} & data_sram_rdata[15:8]  |
                  {8{!ld_addr[0] &  ld_addr[1]}} & data_sram_rdata[23:16] |
                  {8{ ld_addr[0] &  ld_addr[1]}} & data_sram_rdata[31:24] ;
assign mem_half = ld_addr[1] ? data_sram_rdata[31:16] : data_sram_rdata[15:0];
assign mem_word = data_sram_rdata;
assign ld_b_res[31:8] = {24{ms_inst_ld_b & mem_byte[7]}};
assign ld_b_res[7:0]  = mem_byte;
assign ld_h_res[31:16]= {16{ms_inst_ld_h & mem_half[15]}};
assign ld_h_res[15:0] = mem_half;
assign ld_w_res       = mem_word;

assign mem_result = {32{ms_inst_ld_b || ms_inst_ld_bu}} & ld_b_res |
                    {32{ms_inst_ld_h || ms_inst_ld_hu}} & ld_h_res |
                    {32{ms_inst_ld_w}}                  & ld_w_res ;
```

三、实验过程（50%）

（一）实验流水账

由于我们组三人是按照每人写一个实验并辅助以写上一个实验的同学给其他两个同学讲解的方法，因而本次实验由我和安金鹏两人完成，由安金鹏同学完成算数逻辑和乘除运算的添加，由我（史文轩）完成访存和转移指令的添加。

安金鹏同学从 9.29 开始读相关部分的讲义理解本次实验的目的和大体操作方法；

9.30 完成了大部分的设计内容；

10.1 完成 debug 通过测试与我进行交接；

我（史文轩）从 10.4 号开始阅读相关部分讲义并在安金鹏同学的帮助下理解他的代码；

10.5 号完成了大部分设计内容；

10.6 号完成 debug 通过测试；

10.8 号 10.9 号两人合力写完实验报告。

（二）错误记录

1、错误 1：除法器 IP 核设置出错

（1）错误现象

写回数据出错

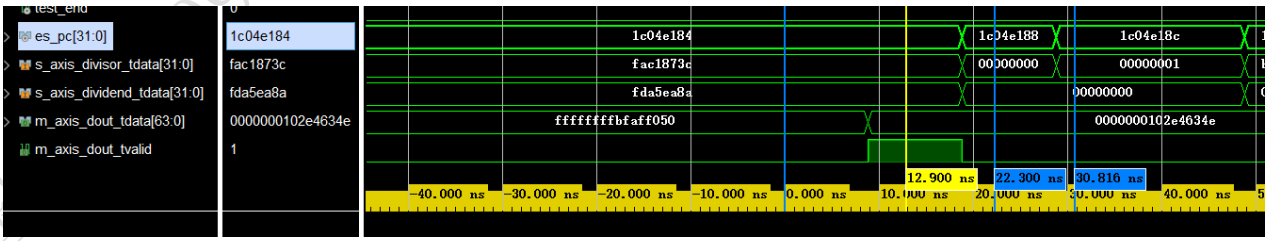
```
------------------------------------------------------------
[ 822927 ns] Error!!!
    reference: PC = 0x1c04e184, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c04e184, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x00000001
```

（2）分析定位过程

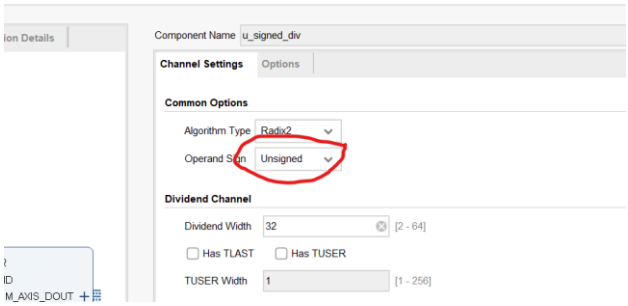
因为在设计的时候只修改了译码和执行阶段两个模块，所以首先根据出错的周期，直接前推到执行阶段。接下来根据 PC 去 test.s 中找到对应的指令发现是有符号除法指令：

```
1c04e178:  03aa298c      ori     $r12,$r12,0xa8a
1c04e17c:  15f5830d      lu12i.w $r13,-21480(0xfac18)
1c04e180:  039cf1ad ori  $r13,$r13,0x73c
1c04e184:  0020358f      div.w   $r15,$r12,$r13
1c04e188:  00150010      move    $r16,$r0
1c04e18c:  5c0d1e0f bne  $r16,$r15,3356(0xd1c) # 1c04eea8 <in
1c04e190:  14a7d68c      lu12i.w $r12,343732(0x53eb4)
```

查看执行阶段的波形图，发现 es_pc,以及两个操作数并未出错。



自己手动计算了一下，被除数比除数的绝对值要小，商理应为 0，但是计算结果却不是，因此一定是除法器的定义出错。查看信号接口并未出错，就想到可能是设置的时候，哪里没有设置对，发现，有符号除法器 IP 设置出错：



（3）错误原因

除法器 IP 核设置出错。

（4）修正效果

更正一下设计，重新跑一下仿真，发现该问题解决

（5）归纳总结（可选）

在调用 IP 核实现乘除法的时候，其中设置的时候一定要认真认真再认真，不然，当出现问题的时候，我们的第一反应是代码哪里写错了，但是去看波形，看代码却迟迟找不到错误，耽误了很长时间才发现是 IP 核设置的问题。

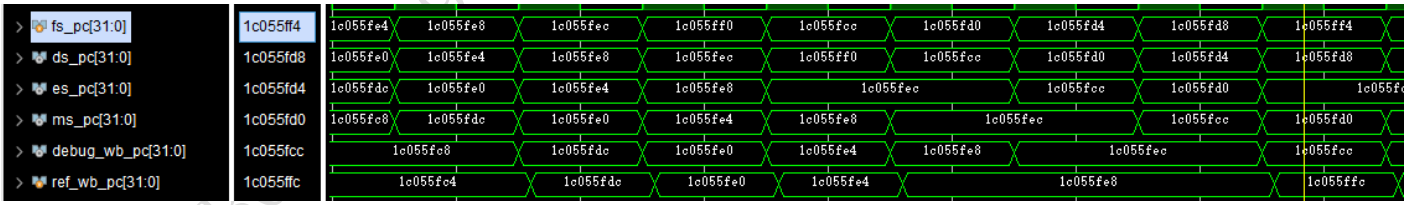
2、错误 2：译码阶段的跳转指令判断出错

（1）错误现象

PC 跳转的结果出错

```
[1147867 ns] Error!!!
    reference: PC = 0x1c055ffc, wb_rf_wnum = 0x1c, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c055fcc, wb_rf_wnum = 0x0f, wb_rf_wdata = 0xede1b000
```

（2）分析定位过程



首先要从波形看出错的指令，PC=0x1c055fcc，给出的 ref 信号为 0xlc055ffc，我们在取值阶段找到 0x1c55fcc，其前一条指令 PC=0x1c055ff0。同时我们在 test.s 中找到 PC=0x1c055fe8 的指令如下：

```
1c055fe8:   03990dad    ori $r13,$r13,0x643
1c055fec:   63ffe18d    blt $r12,$r13,-32(0x3ffe0) # 1c055fcc <n37_blt_test+0x14>
1c055ff0:   50000c00    b   12(0xc) # 1c055ffc <n37_blt_test+0x44>
1c055ff4:   15230930    lu12i.w $r16,-452535(0x91849)
```

可以看到这里有一条 blt 指令和一条 b 指令，我们通过分析容易得出，我们的波形在 blt 处发生了跳转，因而 PC=0x1c055ff0 的这条 b 指令被取消了，从而进入了跳转并取到了 PC=0x1c055fcc 的指令。而若 blt 不发生跳转，就会跳转到 PC=0x1c55fcc 处，这与 ref 信号是一致的。因此，我们合理推测有可能是 blt 指令的跳转条件判断出错，

从而导致出现了不该跳转的跳转。

有这种想法之后，我们就去 ID 阶段的代码中寻找有关 blt 指令的条件判断，发现如下代码：

```
assign rj_lt_rd = ($signed(rj_value) > $signed(rkd_value));
assign rj_ltu_rd = (rj_value < rkd_value);
```

这里在判断大小处出现了明显的失误，本该是小于号由于手误写成了大于号。

（3）错误原因

跳转条件判断出错，进而导致了不该跳转的跳转，使得 PC 错误。

（4）修正效果

本来是打算直接把大于号改成小于号，但是仔细思考后发现这样比较浪费资源，同时又考虑到在 ALU 中有关于大于小于的有符号数和无符号数判断，因而想到是不是可以进行模仿。最终的代码如下：

```
wire [32:0] sub_result;
wire        rj_lt_rd;
wire        rj_ltu_rd;

assign sub_result = {1'b0, rj_value} + {1'b0, ~rkd_value} + 1'b1;


// SLT result
assign rj_lt_rd = (rj_value[31] & ~rkd_value[31])
                | ((rj_value[31] ~^ rkd_value[31]) & sub_result[31]);

// SLTU result
assign rj_ltu_rd = ~sub_result[32];
```

这样只用一个全加器和一些逻辑门就实现了判断。

（5）归纳总结（可选）

这里的错误本身是一个很低级的手误，但是由此引发了我的一些思考，并最终得到了更好的解决方案。

四、实验总结（可选）

此次实验的两部分整体来说难度都不是很大，只是在原来的基础上新加入了部分指令，很多都是复用了原有的数据通路和控制信号，不同的部分也由于去年的计组实验有一定的相似之处而变得不是很难。但是这次是我们组首次进行合作，主要由我和安金鹏两人完成。在进行代码的交接时，难免会对对方的代码有些不熟悉，这些困难都靠着充分的互相交流解决了。总的来说，本次实验受益匪浅，也让我们认识到了代码的格式规范美观和较强的可读性、扩展性对合作项目的重要性！