中国科学院大学
《计算机体系结构基础(研讨课)》实验报告

姓名 艾华春,李霄宇,王敬华

学号 2022K8009916011,2022K8009929029,2022K8009925009

实验项目编号 4 实验名称 异常和中断的支持

一、 逻辑电路结构与仿真波形的截图及说明

- **添加系统调用异常支持、指令和中断处理。**

1. 添加控制寄存器 CRMD, PRMD, ESTAT, ERA, EENTRY, SAVE0-3

   新建一个 csrfile 的模块,里面包括所有控制寄存器的实现,并在`mycpu_top`中进行例化,与各个流水级处于并列地位。

   模块接口包括两类,用于指令访存的接口和与处理器硬件电路逻辑直接交互的接口。

接口名称	接口类型	含义	位宽	输入或输出
csr_re	指令访问	读使能	1	input
csr_num	指令访问	寄存器号	14	input
csr_rvalue	指令访问	寄存器返回值	32	output
csr_we	指令访问	写使能	1	input
csr_wmask	指令访问	写掩码	32	input
csr_wvalue	指令访问	写数据	32	input
wb_ex	硬件电路交互	异常处理触发信号	1	input
wb_ecode	硬件电路交互	异常类型	6	input
wb_esubcode	硬件电路交互	异常类型	9	input
wb_pc	硬件电路交互	触发异常的 pc 值	32	input
wb_vaddr	硬件电路交互	触发异常的虚地址	32	input
ertn_flush	硬件电路交互	ertn 指令	1	input
....				

   在写回级流水级模块 wb_reg 中,创建上述所有的接口。在 cpu_top 模块中,将两个模块 wb_reg 和 csrfile 中对应的接口相连接。

   最后,在 csrfile 模块中创建了所有输入输出接口后,按照讲义上的内容,以每个 CSR 的各个域作为基本单位,依次实现各个控制寄存器初始化,被指令访问和修改,被硬件电路逻辑访问和修改。

2. 增加控制寄存器 ECFG, BADV, TID, TCFG, TVAL, TICLR

   补全 ECFG 和 BADV 寄存器。

```
/*--------------------------ECFG-----------------------------------------------------*/
always @(posedge clk)begin
    if(~resetn)
        csr_ecfg_lie <= 13'b0;
    else if (csr_we && csr_num == `CSR_ECFG)     // csr_ecfg_lie[10] == 0
        csr_ecfg_lie <=  csr_wmask[`CSR_ECFG_LIE] & 13'h1bff & csr_wvalue[`CSR_ECFG_LIE]
                       | ~csr_wmask[`CSR_ECFG_LIE] & 13'h1bff & csr_ecfg_lie;
```

```
end
/*--------------------------ERA-----------------------------------------------------*/
always @(posedge clk)begin
    if(wb_ex)
        csr_era_pc <=    wb_pc;
    else if(csr_we && csr_num == `CSR_ERA)
        csr_era_pc <=    csr_wmask[`CSR_ERA_PC] & csr_wvalue[`CSR_ERA_PC]
                       | ~csr_wmask[`CSR_ERA_PC] & csr_era_pc;
end
```

假设定时器位数为 32 位，从全 f 开始递减，补全 TID, TCFG, TVAL, TICLR 寄存器。

```
/*--------------------------TID-----------------------------------------------------*/
    //add TID
always @(posedge clk)begin
    if(~resetn)
        csr_tid_tid <= 32'b0;
    else if(csr_we && csr_num == `CSR_TID)
        csr_tid_tid <= csr_wmask[`CSR_TID_TID] & csr_wvalue[`CSR_TID_TID]
                     | ~csr_wmask[`CSR_TID_TID] & csr_tid_tid;
end

/*--------------------------TCFG-----------------------------------------------------*/
    //add TCFG
always @(posedge clk)begin
    if(~resetn)
        csr_tcfg_en <= 1'b0;
    else if(csr_we && csr_num==`CSR_TCFG)
        csr_tcfg_en <= csr_wmask[`CSR_TCFG_EN] & csr_wvalue[`CSR_TCFG_EN]
                     | ~csr_wmask[`CSR_TCFG_EN] & csr_tcfg_en;

    if(csr_we && csr_num==`CSR_TCFG)begin
        csr_tcfg_periodic <= csr_wmask[`CSR_TCFG_PERIOD] & csr_wvalue[`CSR_TCFG_PERIOD]
                           | ~csr_wmask[`CSR_TCFG_PERIOD] & csr_tcfg_periodic;
        csr_tcfg_initval <= csr_wmask[`CSR_TCFG_INITV] & csr_wvalue[`CSR_TCFG_INITV]
                           | ~csr_wmask[`CSR_TCFG_INITV] & csr_tcfg_initval;
    end
end

/*--------------------------TVAL-----------------------------------------------------*/
    //add TVAL
assign tcfg_next_value = csr_wmask[31:0] & csr_wvalue[31:0]
                       | ~csr_wmask[31:0] & {csr_tcfg_initval,csr_tcfg_periodic,csr_tcfg_en};
                           //value of TCFG in the next clk

always @(posedge clk)begin
    if(~resetn)
        timer_cnt <= 32'hffffffff;
    else if(csr_we && csr_num==`CSR_TCFG && tcfg_next_value[`CSR_TCFG_EN])
        timer_cnt <= {tcfg_next_value[`CSR_TCFG_INITV],2'b0};
```

```
    else if(csr_tcfg_en && timer_cnt!=32'hffffffff) begin
        if(timer_cnt[31:0]==32'b0 && csr_tcfg_periodic)
            timer_cnt <= {csr_tcfg_initval,2'b0};
        else
            timer_cnt <= timer_cnt -1'b1;
    end
end

assign csr_tval = timer_cnt[31:0];

/*--------------------------TICLR-----------------------------------------------------------*/
    //add TICLR
assign csr_ticlr_clr = 1'b0;
```

3. 增加 csrrd,csrwr,csrxchg 指令

在 ID 流水级模块中，通过译码操作，解析出csr_we，csr_re，csr_num，csr_wmask信息，并创建从 ID 流水级到 WB 流水级的逐级传递的数据通路，用来传递这些信号。

```
assign id_csr_re = inst_csrrd || inst_csrwr || inst_csxchg || inst_ertn; // 控制寄存器读使能
assign id_csr_num = inst_ertn ? 14'h6        // CSR_ERA
                  :id_excep_en ? 14'hc         // CSR_EENTRY
                  :id_read_TID ? 14'h40        // CSR_TID
                               :csr_num;
assign id_csr_we = inst_csrwr || inst_csxchg; // 控制寄存器写使能
assign id_csr_wmask = inst_csxchg ? rj_value: ~32'b0; // 控制寄存器写掩码
```

在原来的通用寄存器读出数据 rj_valuw, rkd_value 的逐流水级传递数据通路的基础上，延长其至写回级，作为控制寄存器访问的数据和掩码。

例如，原来 rkd_value 的逐级传递到 wb 流水级为止，作为写回 mem 的数据。将其继续逐级传递到 wb 流水级，并且作为 csr 控制寄存器的写数据。

```
assign csr_wvalue = rkd_value; // rd 寄存器的旧值作为控制寄存器的写数据。
```

4. 添加 ertn 和 syscall 指令

在 id 级添加相应的译码逻辑，生成将当前指令是 ertn 指令的信号，以及异常使能信号，异常类型相关信号。并依附于流水级逐级传递至写回级模块, 通过第一步定义的接口，与 csrfile 模块中的对应接口相连接。

```
assign id_ertn_flush = inst_ertn; // 当前指令是ertn

assign id_excep_INT  =  has_int;      // 记录中断信号
assign id_excep_SYSCALL = inst_syscall; // 记录该条指令是否存在SYSCALL异常
assign id_excep_BRK  =  inst_break;   // 记录该条指令是否存在BRK异常
assign id_excep_INE  =  no_inst;      // 记录该条指令是否存在INE异常
assign id_excep_en =    id_excep_INT | id_excep_SYSCALL | id_excep_BRK | id_excep_INE |
    if_excep_en; //只要有一个异常就置1
```

并向每个流水级增加从 wb 发出的清空流水级的信号 flush。

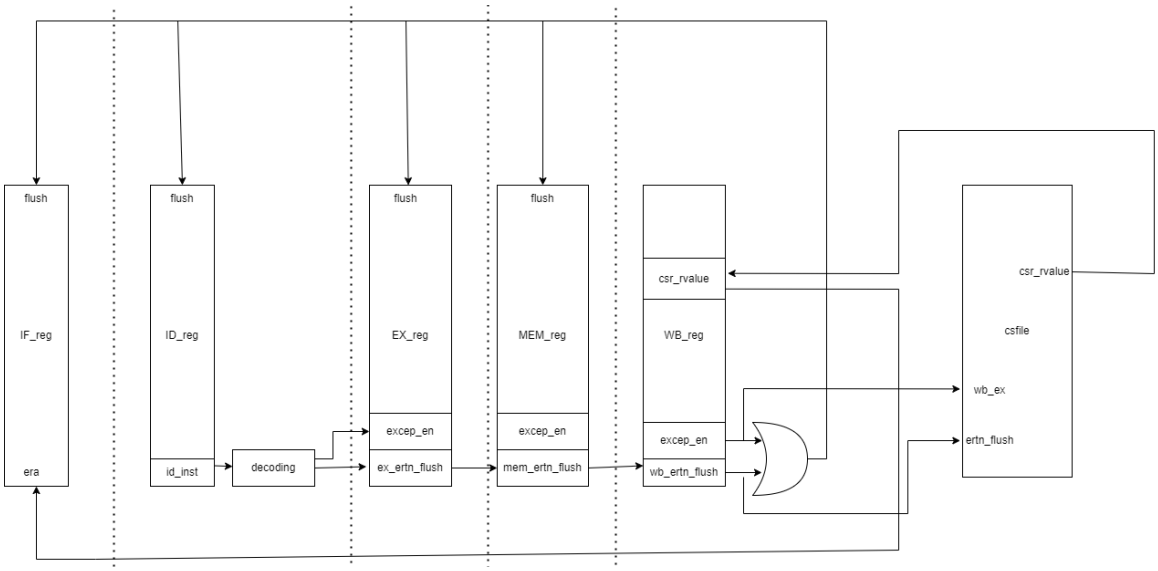


图 1: syscall 和 ertn 增加的部分数据通路

从 wb 流水级向 id 流水级传递异常跳转地址 era，如图所示 每个流水级，接受到清空流水线的信号时，将当前流水级的 valid 置 0。

在 IF 流水级，如果接受到清空流水线的信号，将下一个 pc 值设置为从 wb 传来的返回的异常处理地址。

```
assign pre_pc       =  flush ? wb_csr_rvalue // 将下一个pc值设置为从wb传来的返回的异常处理地址
                     : br_taken ? br_target
                     : seq_pc;
```

5. 添加取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE）异常的支持

从每个例外对应的判断阶段开始，其和其之后的每个阶段都添加一个 1 位宽的用于记录该条指令是否发生该种例外的控制信号。

ADEF 在 pre-IF 级进行判断，当取值地址不为 4 字节对齐时，产生 ADEF 例外，并在 WB 阶段把错误地址传给 BADV 寄存器。

```
assign pre_if_excep_ADEF =  pre_pc[0] | pre_pc[1]; // 记录该条指令是否存在ADEF异常
assign pre_if_excep_en =   pre_if_excep_ADEF;
```

INE 和 BRK 均在 ID 阶段判断，当发现是相应指令时产生例外。

```
// 中断,系统调用,断点, 指令不存在异常处理
    assign id_excep_INT  =  has_int;      // 记录中断信号
    assign id_excep_SYSCALL = inst_syscall; // 记录该条指令是否存在SYSCALL异常
    assign id_excep_BRK  =  inst_break;   // 记录该条指令是否存在BRK异常
    assign id_excep_INE  =  no_inst;      // 记录该条指令是否存在INE异常
    assign id_excep_en =    id_excep_INT | id_excep_SYSCALL | id_excep_BRK | id_excep_INE |
        if_excep_en; //只要有一个异常就置1
    assign id_excep_esubcode = 9'h0;
```

ALE 在 EX 阶段判断，当取半字时地址最低位为 1，或取字时地址最后两位不全为 0，则产生 ALE 例外，并记录错误地址，等到 WB 阶段将其传给 BADV 寄存器。

```
// 地址非对齐异常处理
    assign ex_excep_ALE = (ex_op_st_ld_h & ex_alu_result[0]) | (ex_op_st_ld_w &
        (ex_alu_result[1] | ex_alu_result[0])); // 记录该条指令是否存在ALE异常
    assign ex_excep_en = ex_excep_ALE | id_excep_en;

    assign ex_vaddr = {32{ex_read_counter && ~ex_read_counter_low}} & counter[63:32] |
                    {32{ex_read_counter && ex_read_counter_low}} & counter[31: 0] |
                    {32{~ex_read_counter}} & ex_alu_result;
```

6. 计数器的添加

添加 counter.v 文件，存放计数器 Stable_Counter，没经过一个时钟周期自增 1。

```
always @(posedge clk)begin
    if(~resetn)
        time_counter <= 64'b0;
    else
        time_counter <= time_counter + 1'b1;
end
```

7. 添加 rdcntvl.w、rdcntvh.w 和 rdcntid 指令

添加控制信号 id_read_counter, id_read_counter_low, id_read_TID，分别用于记录指令是否需要读取计数器的值，是否要读取计数器的低 32 位，指令是否要读取计数器 ID。

```
    assign id_read_counter = inst_rdcntvl_w | inst_rdcntvh_w;
    assign id_read_counter_low = inst_rdcntvl_w;
    assign id_read_TID     = inst_rdcntid;
```

计数器的值在 EX 阶段根据 ex_read_counter_low 完成读入。

```
// 读计数器
    assign ex_counter_result = ex_read_counter_low ? counter[31:0] : counter[63:32];
        //处理rdcntvl.w rdcntvh.w指令
```

TID 的值在 WB 阶段完成读入。

```
    assign final_rf_wdata = wb_csr_re ? csr_rvalue :
                            wb_read_TID ? csr_rvalue : wb_rf_wdata;      //add csr_tid_rvalue for
                                rdcntid.w
```

8. 添加中断处理

实际上，中断处理可以看作一种特殊的异常，其主要的任务也与异常的处理一致首先在控制寄存器 (csr-file.v) 中设置中断处理信号：

```
    assign has_int = ((csr_estat_is[12:0] & csr_ecfg_lie[12:0]) != 13'b0)&& (csr_crmd_ie ==
        1'b1);
```

硬件中断通过设备或中断控制器将高电平有效的中断信号连接到 8 个输入引脚上，ESTAT 控制状态寄存器 IS 的 9..2 这八位直接对中断输入引脚的信号采样。软件中断通过 CSR 写指令对 ESTAT 状态控制寄存器 IS 域的 1..0 这两位写 1 或写 0 进行控制。定时器中断的状态记录在 ESTAT 控制状态寄存器 IS 域的第 11 位，其中这些寄存器一一对应着 cpu 外部的引脚，

```
    assign hw_int_in = 8'b0;//硬中断接口
    assign ipi_int_in = 1'b0;//定时器中断接口
    always @(posedge clk)begin
    if(~resetn)
        csr_estat_is[1:0] <= 2'b0;
    else if(csr_we && csr_num == `CSR_ESTAT)
        csr_estat_is[1:0] <= csr_wmask[`CSR_ESTAT_IS10] & csr_wvalue[`CSR_ESTAT_IS10]
                          | ~csr_wmask[`CSR_ESTAT_IS10] & csr_estat_is[1:0];

    csr_estat_is[9:2] <= hw_int_in[7:0];      // come from hardware sampling
    csr_estat_is[10] <= 1'b0;                 // reserved

    if(timer_cnt[31:0] == 32'b0)               // time counter interrupt
        csr_estat_is[11] <= 1'b1;
    else if(csr_we && csr_num == `CSR_TICLR && csr_wmask[`CSR_TICLR_CLR] &&
         csr_wvalue[`CSR_TICLR_CLR])
        csr_estat_is[11] <= 1'b0;

    csr_estat_is[12] <= ipi_int_in;

end
```

然后中断处理信号在 ID 阶段生成，并且和其他异常一起组成 excep_en 信号流向后面的流水级

```
    // 中断,系统调用,断点, 指令不存在异常处理
    assign id_excep_INT  =   has_int;      // 记录中断信号
    assign id_excep_SYSCALL = inst_syscall; // 记录该条指令是否存在SYSCALL异常
    assign id_excep_BRK  =   inst_break;   // 记录该条指令是否存在BRK异常
    assign id_excep_INE  =   no_inst;      // 记录该条指令是否存在INE异常
    assign id_excep_en =     id_excep_INT | id_excep_SYSCALL | id_excep_BRK | id_excep_INE |
        if_excep_en; //只要有一个异常就置1
    assign id_excep_esubcode = 9'h0;
```

最后中断与其他所有的异常处理在 WB 阶段生成相应的 ecode 和 wb_ex 信号（控制流水线清空）

```
    assign wb_ecode = wb_excep_INT  ? 6'h0 :     //INT 中断
    wb_excep_ADEF  ? 6'h8 :     //ADEF
    wb_excep_SYSCALL ? 6'hb :   //SYSCALL
    wb_excep_BRK   ? 6'hc :     //BRK
    wb_excep_INE   ? 6'hd :     //INE
    6'h9;                       //ALE
    assign wb_esubcode= wb_excep_esubcode;
    assign wb_ex =    wb_excep_en & wb_valid;
```

wb_ex 信号和 ertn 指令信号（ertn_flush）从 WB 流水级传到 cpu 顶部，指导各个流水级进行清空（flush）

操作，也就是将相应流水及 valid 拉低

```
.flush(ertn_flush || wb_ex),//IF为例
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，逻辑仿真和 FPGA 调试过程中的难点等）

- **wb 流水级的 flush 有效信号只持续一个 clk。**

在清空流水线的设计时，将 wb 流水级在发出 flush 信号时的下一个 clk 时，也要将 wb_valid 置为 0，避免重复清空流水线。

使得如果 IF 流水级的 allowin 由于读后写数据相关等原因为 0 时，不能及时把`pre_pc = wb_csr_rvalue`发送给`inst_sram`，而随着下一个 clk 的 wb_valid 拉低，使从 wb 传到 if 的 wb_csr_rvalue 信号也失效。

在这种情况下，不能正确地进行跳转到异常处理地址。

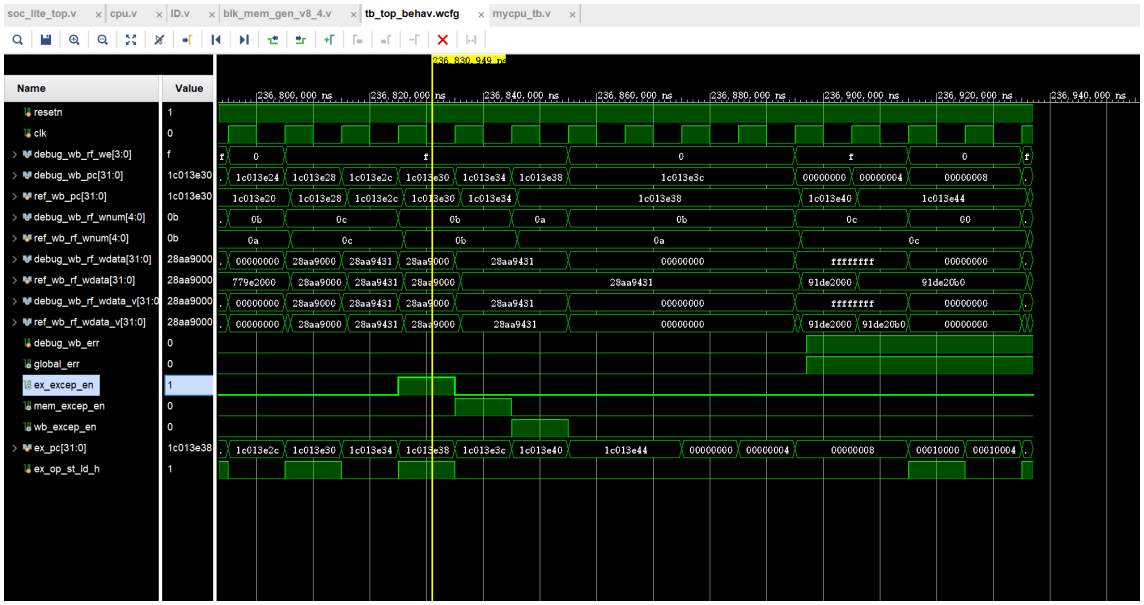
在分析波形图后，确定上述 bug 后，为 if_allowin 添加上规则，使得接收到清空流水线时，立马将 allowin 拉高，在当前 clk 发送出 pc

```
assign if_allowin    =   ~if_valid            // valid是reg类型，接受flush后最快下一个clk才能拉低
                       | if_ready_go & id_allowin // id_allowin可能由于读后写阻塞，拉低
                       | flush;   // 添加上规则，立马将allowin拉高，在当前clk发送出pc
```

- **指令译码定义太宽泛导致异常判断出错。**

问题报错如下：

```
----------------------------------------------------------------
[ 236897 ns] Error!!!
    reference: PC = 0x1c013e40, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x91de2000
    mycpu    : PC = 0x00000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
----------------------------------------------------------------
$finish called at time : 236937 ns : File "C:/Users/86156/Desktop/calab/prj4_exp13/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 169
) run: Time (s): cpu = 00:00:08 ; elapsed = 00:00:15 . Memory (MB): peak = 1991.660 ; gain = 19.238
```



查看信号波形如上，按照错误信号定位，可以追溯到是 slli.w 指令（1c013e38: 0040818a slli.w $r10,r12,0x0$ ），这条指令本不应该出现异常，但是错误的发生了异常信号 ex_excep_en 进一步追溯发现是由 ex_op_st_ld_h 信号导致的。

```
// 地址非对齐异常处理
    assign ex_excep_ALE = (ex_op_st_ld_h & ex_alu_result[0]) | (ex_op_st_ld_w & (ex_alu_result[1] | ex_alu_result[0]));
    assign ex_excep_en = ex_excep_ALE | id_excep_en;
```

查看原来的代码发现，原来的信号定义的过于宽泛，导致出现了很多错误的拉高

```
    assign id_op_st_ld_b       = op_25_22[1:0] == 2'd0;
    assign id_op_st_ld_h       = op_25_22[1:0] == 2'd1;
    assign id_op_st_ld_w       = inst_ld_w | inst_st_w;
    assign id_op_st_ld_u       = op_25_22[3];
```

进行如下修改之后，通过该测试点。

```
    assign id_op_st_ld_b       = inst_st_b | inst_ld_b |inst_ld_bu;
    assign id_op_st_ld_h       = inst_st_h | inst_ld_h |inst_ld_hu;
    assign id_op_st_ld_w       = inst_ld_w | inst_st_w;
    assign id_op_st_ld_u       = inst_ld_bu | inst_ld_hu;
```

- **控制寄存器写入逻辑出错。**

  报错信息如下：

```
----[1375225 ns] Number 8'd48 Functional Test Point PASS!!!
--------------------------------------------------------------
[1379007 ns] Error!!!
    reference: PC = 0x1c0081ac, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000004
    mycpu    : PC = 0x1c0081ac, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
--------------------------------------------------------------
```

查看信号波形如上，按照错误信号定位，可以定位到指令（1c0081ac: 0400040c csrrd $r12,0x1）出错，错误的原因是控制寄存器 prmd 存储的的值出错，向前寻找上一次写改变该值的指令（图中黄线），发现在这个周期虽然没有写使能信号，但是寄存器的值发生了神秘的变化，仔细观察代码逻辑，发现原来的代码忘记加 begin 和 end 导致寄存器赋值逻辑出错，修改后通过。

```
else if(csr_we && csr_num == `CSR_PRMD)  begin    // inst access
    csr_prmd_pplv <=     csr_wmask[`CSR_PRMD_PPLV] & csr_wvalue[`CSR_PRMD_PPLV
                       | ~csr_wmask[`CSR_PRMD_PPLV] & csr_prmd_pplv;
    csr_prmd_pie  <=     csr_wmask[`CSR_PRMD_PIE] & csr_wvalue[`CSR_PRMD_PIE]
                       | ~csr_wmask[`CSR_PRMD_PIE] & csr_prmd_pie;
end
```

- **控制寄存器读出逻辑出错。**

  报错信息如下：

```
[1379517 ns] Error!!!
    reference: PC = 0x1c0722f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00001bff
    mycpu    : PC = 0x1c0722f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
------------------------------------------------------------
```

查看信号波形如上，按照错误信号定位，可以定位到指令（1c0722f8: 0400102c csrwr $r12,0x4）出错，错误的原因是读控制寄存的读出值 csr_rvalue 出错，但是 ecfg 寄存器内部寄存的值是正确的，仔细观察赋值逻辑，发现一处笔误错误的将 csr_ecfg_rvalue 写作 csr_tcfg_rvalue，修改之后通过。

```
assign csr_rvalue =         {32{csr_num == `CSR_CRMD}} & csr_crmd_rvalue
                           |{32{csr_num == `CSR_PRMD}} & csr_prmd_rvalue
                           |{32{csr_num == `CSR_ESTAT}} & csr_estat_rvalue
                           |{32{csr_num == `CSR_ERA}} & csr_era_rvalue
                           |{32{csr_num == `CSR_EENTRY}} & csr_eentry_rvalue
                           |{32{csr_num == `CSR_SAVE0}} & csr_save0_rvalue
                           |{32{csr_num == `CSR_SAVE1}} & csr_save1_rvalue
                           |{32{csr_num == `CSR_SAVE2}} & csr_save2_rvalue
                           |{32{csr_num == `CSR_SAVE3}} & csr_save3_rvalue
                           |{32{csr_num == `CSR_ECFG}}  & csr_tcfg_rvalue
                           |{32{csr_num == `CSR_BADV}}  & csr_badv_rvalue
                           |{32{csr_num == `CSR_TID}}   & csr_tid_rvalue
                           |{32{csr_num == `CSR_TCFG}}  & csr_tcfg_rvalue
                           |{32{csr_num == `CSR_TVAL}}  & csr_tval_rvalue
                           |{32{csr_num == `CSR_TICLR}} & csr_ticlr_rvalue;
```

- **trace 比对提交的条件出错。**

  报错信息如下：

```
------------------------------------------------------------
[1915277 ns] Error!!!
    reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
    mycpu    : PC = 0x1c074d94, wb_rf_wnum = 0x1f, wb_rf_wdata = 0x00000000
------------------------------------------------------------
```

按照错误信号定位，可以定位到指令（1c074d94: ffffffff 0xffffffff）出错，错误的原因是：这是一条错误的指令，cpu 测出这是一种异常，但是 trace 比对写回值结果的时候是以 debug_wb_rf_we 为使能信号的，在异常产生的

时候，应该优先处理异常，拉低这个信号，因此在这个信号内部 & wb_excep_en 信号表示没有出现异常的时候才进行比对，并且对 wb_rf_we 的前身 gr_we 进行了补充

```
assign debug_wb_pc = wb_pc;
assign debug_wb_rf_wdata = final_rf_wdata;
assign debug_wb_rf_we = {4{wb_rf_we & wb_valid & ~wb_excep_en}};
assign debug_wb_rf_wnum = wb_rf_waddr;
```

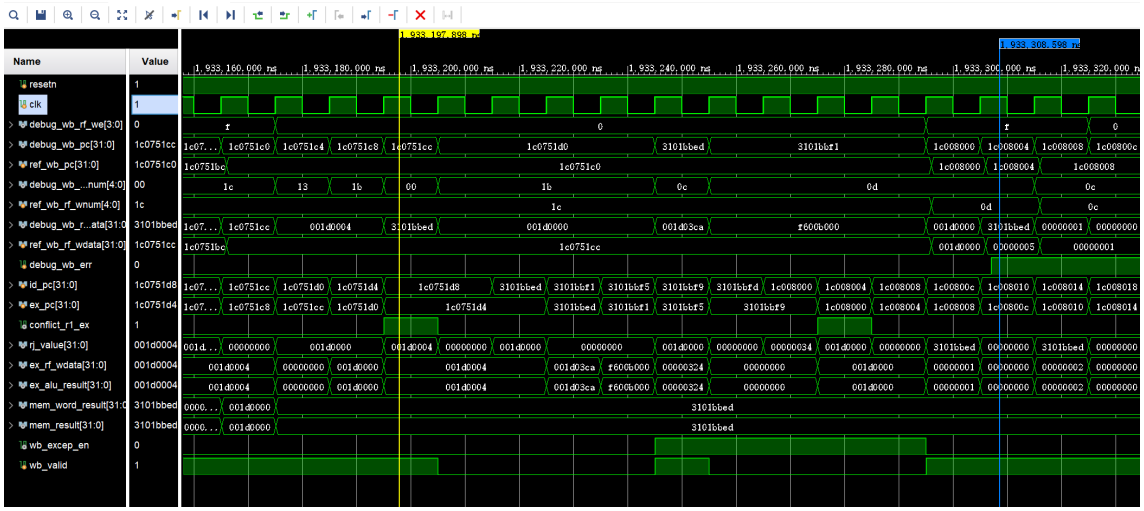
```
//寄存器的写地址和写使能
assign gr_we          =   ~inst_st_w & ~inst_st_b & ~inst_st_h & ~inst_beq & ~inst_bne
                        & ~inst_blt & ~inst_bge & ~inst_bltu & ~inst_bgeu & ~inst_b & id_valid ;
```

```
//寄存器的写地址和写使能
assign gr_we          =   ~inst_st_w & ~inst_st_b & ~inst_st_h
                        & ~inst_beq & ~inst_bne & ~inst_blt & ~inst_bge & ~inst_bltu & ~inst_bgeu & ~inst_b
                        & ~inst_syscall & ~inst_ertn & ~inst_break
                        & id_valid;
```

- 在 **EXE、MEM 判断出 ERTN 后未拉低写使能信号** 。

  报错信息如下：

```
------------------------------------------------------------
[1933307 ns] Error!!!
    reference: PC = 0x1c008004, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000005
    mycpu    : PC = 0x1c008004, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x3101bbed
------------------------------------------------------------
```



查看信号波形如上，按照错误信号定位，查看当前指令类型 (1c008004: 288001ad ld.w $r13$,r13,0) load 读数据出错说明错误的原因是上一次访存指令错误的写入数据，猜测前序写内存时有误，查看对应数据最近一次被写入的时机：找到图中黄线的时刻, 是一条 st.w 指令后面跟着 ertn 指令

```
1c0751c8:  2980127b   st.w $r27,$r19,4(0x4)
1c0751cc:  06483800   ertn
```

这下破案了，原因是 ertn 指令判断后，没有取消内存写使能的信号，导致错误的数据被写入

> 常之后的指令修改了，这就违反了精确异常。那么该如何处理呢？现阶段一种简单有效的方式是：store 指令若想在执行级发出写命令，那么需要检查当前访存级和写回级上是否存在已标记为异常或可能标记为异常的指令，也要检查自己有没有产生异常或标记异常。庆幸的是，就目前支持的指令和异常类型来说，指令在访存级和写回级不会再判断出新的异常了。也就是说，位于执行级

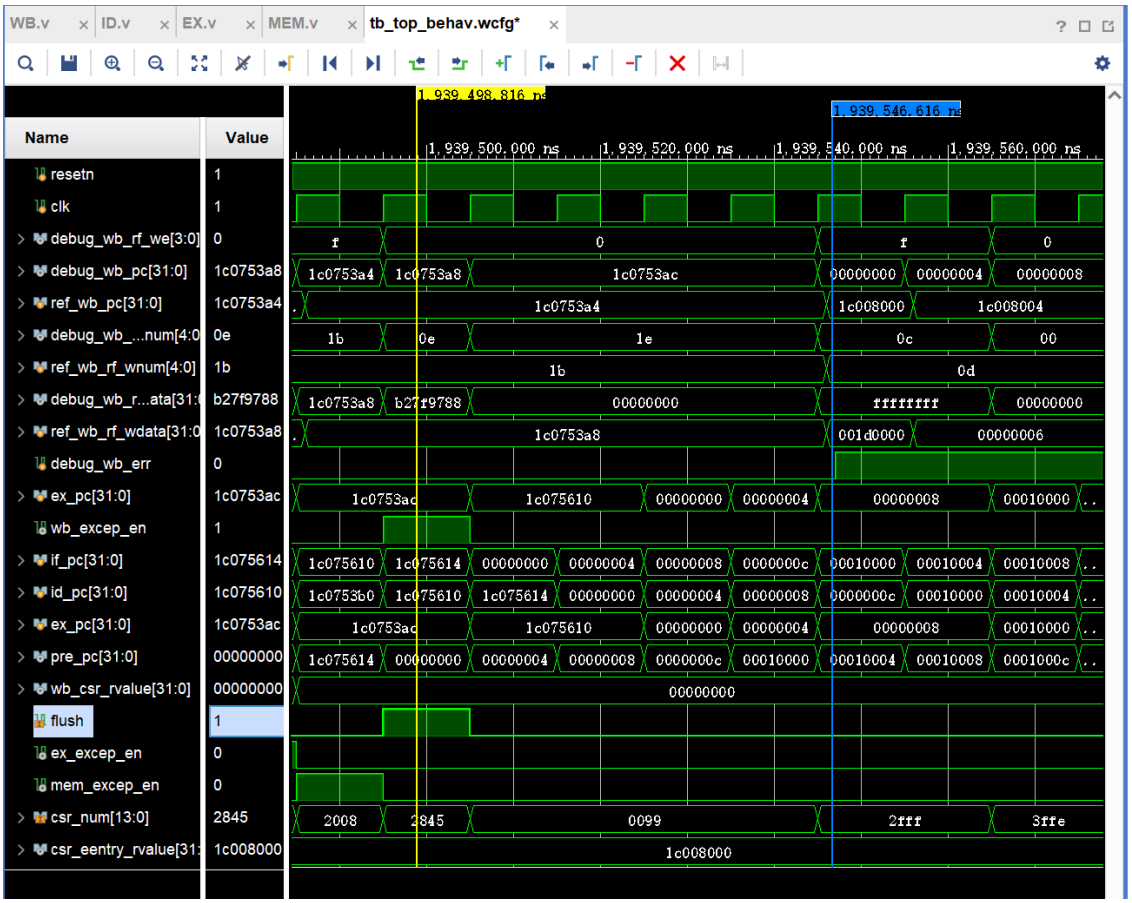
```
assign data_sram_en  = (ex_res_from_mem || ex_mem_we) && ex_valid && ~mem_excep_en &&
    ~wb_excep_en &&~mem_ertn_flush&& ~wb_ertn_flush && ~ex_excep_ALE;
```

按照书上的要求进行修改，在内存片选信号上面添加对 ertn 的检查信号之后，通过该测试点。

- **异常处理写入的 PC 值入口出错。**

  报错信息如下：

```
------------------------------------------------------------
[1939547 ns] Error!!!
    reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
    mycpu    : PC = 0x00000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
------------------------------------------------------------
```



查看信号波形如上，按照 PC 错误跳转信号定位，可以定位到指令（1c0753a8: 28a1148e ld.w *r*14,r4,-1979(0x845)）这条指令出现了 ALE 异常，将 flush 信号拉高，因此清空流水线并且 prePC 进行取指令，错误的原因是异常处理的 pc 地址出错。追根溯源，可以发现是错误的不加区分的直接使用了 csr 的读出值 csr_rvalue 作为了 PC。正确的方式应该是根据是 ertn 还是异常采用区分 era 或者 eenery 的值

```
    assign excep_entry = wb_ex ? csr_eentry_rvalue //PC在flush时候取指的地址
        /*ertn_flush*/:csr_era_pc;
```

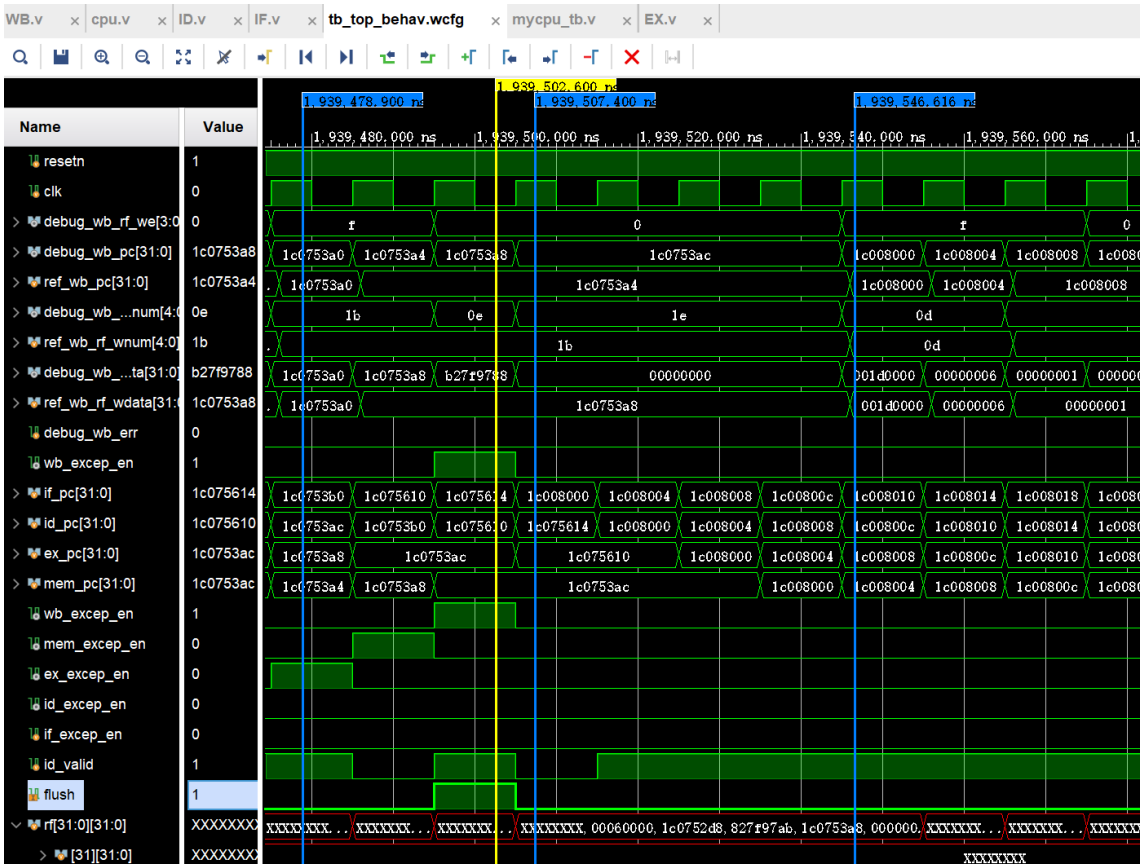
修改之后，通过测试点。

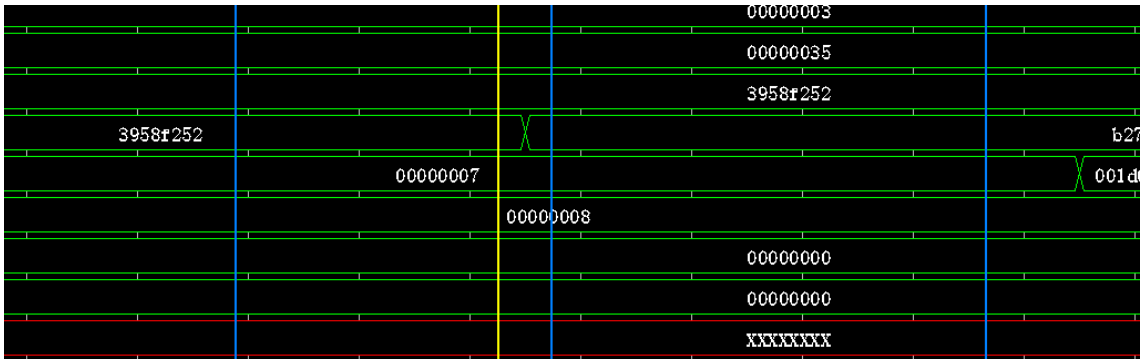
- **控制寄存器写入使能信号出错。**

  报错信息如下：

```
------------------------------------------------------------
[1940127 ns] Error!!!
    reference: PC = 0x1c0753b4, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x001d0000
    mycpu    : PC = 0x1c075610, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x35000000
------------------------------------------------------------
```

根据错误出现的位置，可以发现指令为（1c0753ac: 5c02673e bne $r25,$r30,612(0x264) 1c075610 <inst_error>）指令发生了错误的跳转导致出错，原因是通用寄存器的值被前面的指令错误的修改了，定位到上次这个寄存器被修改的位置：





查看信号波形如上，这里的指令是产生了 flush 信号，按理来说应该将通用寄存器的写使能拉低，但是这里并没有，这就导致了寄存器值的改变。

```
assign wb_to_id_bus = {wb_rf_we & wb_valid & ~wb_ex & ~ertn_flush, wb_rf_waddr,
    final_rf_wdata};
```

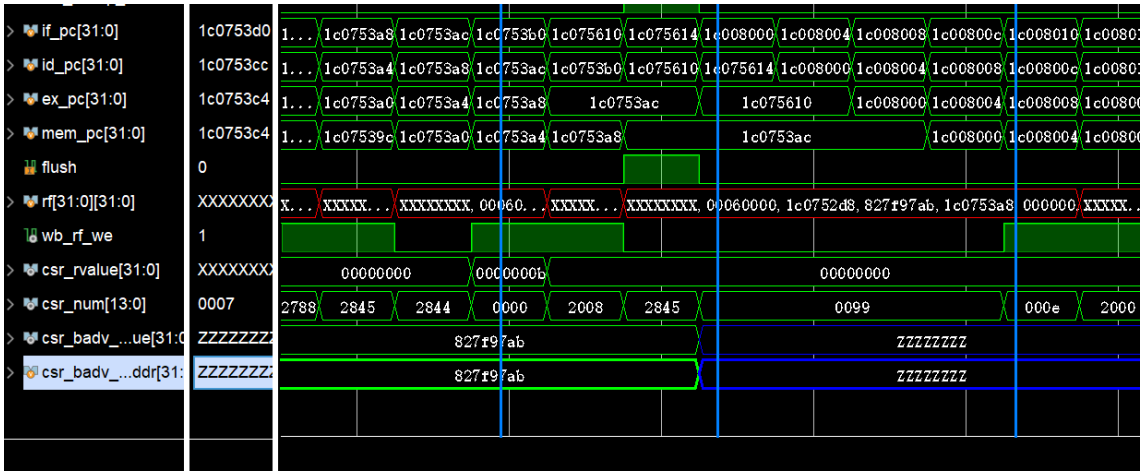
在写使能处加上了 & ertn_flush 即可，通过测试点。

- **控制寄存器信号悬空。**

报错信息如下：

```
------------------------------------------------------------
[1940647 ns] Error!!!
    reference: PC = 0x1c0753c4, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x000cfde1
    mycpu    : PC = 0x1c0753c4, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xxxxxxxxx
------------------------------------------------------------
```

这里发生错误的指令是 (1c0753c4: 04001c0c csrrd $r12,0x7)，错误的原因是 07 控制寄存器 CSR_BADV 的值出现高阻态导致无法读出。




```
always @(posedge clk)begin
 if(wb_ex && wb_ex_addr_err)
    csr_badv_vaddr <=    (wb_ecode == `ECODE_ADE && wb_esubcode == `ESUBCODE_ADEF) ?
                        wb_pc          //  inst fecth error
                        :wb_vaddr;     // mem access error and so on
    end
```

查看信号波形如上，通过代码这里可以看出是因为 wb_vaddr 信号悬空没有赋值（实际上通路也没有打通）导致出现高阻态。

```
assign ex_vaddr = {32{ex_read_counter && ~ex_read_counter_low}} & counter[63:32] |
{32{ex_read_counter && ex_read_counter_low}} & counter[31: 0] |
{32{~ex_read_counter}} & ex_alu_result;
```

实现信号并且打通通路之后，错误解除。

- **中断未复位 ALU**。

  报错信息如下：

```
----------------------------------------------------------------
[1943867 ns] Error!!!
    reference: PC = 0x1c0754f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c0754f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00004094
----------------------------------------------------------------
```

这是一条除法指令 (1c0754f8: 0020418c div.w $r12,$r12,$r16) 查看信号波形发现恰好该 div 指令在 wb 阶段判断出异常：异常发生时 alu 中乘法和除法器的计数器未被复位，导致后续操作出错。应该令 alu 的 resetn 信号与上 wb_excep_en 和 wb_ertn_flush。

```
    //alu的实例化
alu u_alu(
    .clk          (clk     ),
    .resetn       (resetn && ~wb_excep_en && ~wb_ertn_flush ),
    .alu_op       (ex_alu_op  ),
    .alu_src1     (ex_alu_src1 ),
    .alu_src2     (ex_alu_src2 ),
    .alu_result   (ex_alu_result),
    .complete     (alu_complete)
);
```

修改之后，测试 PASS！

三、 小组成员分工合作情况

王敬华负责 exp13 的中断处理和整体的 debug 工作
李霄宇负责 exp13 的异常处理和计时器指令的实现
艾华春负责 exp12：添加系统调用异常支持
实验报告为根据每人负责代码的部分，写相应部分的报告。