Lab 8 报告

学号：2020K8009929034、2020K8009929019、2020K8009929037
姓名：胡康、李子恒、吕星宇
箱子号：05

一、实验任务（10%）

存储管理是现代操作系统的重要功能之一. 本实验通过三阶段来完成一个简单的 MMU 管理:

1. 完成 TLB 模块的设计；
2. 将 TLB 集成到现有 CPU 中， 实现 TLB 相关指令和控制寄存器；
3. 添加异常处理和虚实地址转换模块，使 CPU 能够处理 MMU 相关异常。

二、实验设计（40%）

（一）总体设计思路

1. TLB 的设计本质和寄存器区别不大，可以理解成另类的寄存器堆，不过需要比对 index 来选出合适的输出。

2. 设计虚实地址转换模块加入到 IF 和 EXE 级，在 cause_bus 上添加相关异常支持。

（二）重要模块 1 设计：TLB

1、工作原理

TLB 一共需要五类端口，取指和访存的两类查询端口和用于 TLB 指令和异常处理的两类读写端口，最后是专门给 invtlb 指令的端口。

取指和访存端口一致（在下面的接口定义中只介绍 s 端口，即取指为 s0 端口，访存为 s1 端口，两者端口功能上一致），通过输入的虚页号，虚地址的 12 位和 asid 信息来查找，并输出 found，index 等信号。

其工作原理为：通过传入的虚页号和 asid 对所有项进行查找比对：

```
generate // ! 用来对比TLB表项，生成两个match regs
for(tlb_index = 0; tlb_index < TLBNUM; tlb_index=tlb_index+1)
begin: __TLB_compare
        assign match0[tlb_index] = // ! 虚拟地址对上 + 页数对上 + asid对上 + exist
        (s0_vppn[18:10] == tlb_vppn[tlb_index][18:10]) && tlb_e[tlb_index]
        && (tlb_ps4MB[tlb_index] || s0_vppn[9:0]==tlb_vppn[tlb_index][9:0])
        && (s0_asid == tlb_asid[tlb_index] || tlb_g[tlb_index]);

        assign match1[tlb_index] =
        (s1_vppn[18:10] == tlb_vppn[tlb_index][18:10]) && tlb_e[tlb_index]
        && (tlb_ps4MB[tlb_index] || s1_vppn[9:0]==tlb_vppn[tlb_index][ 9: 0])
        && (s1_asid == tlb_asid[tlb_index] || tlb_g[tlb_index]);
end
endgenerate
```

图 1：tlb 比对代码

并得到对应项的 index，否则 found=0：

```
// 是否found
assign s0_found = |match0;
assign s1_found = |match1;
// ! 采用书上介绍的 "经典" 选择逻辑
assign s0_index = ({4{match0[0]}} & 4'd0)   | ({4{match0[1]}} & 4'd1)
                | ({4{match0[2]}} & 4'd2)   | ({4{match0[3]}} & 4'd3)
                | ({4{match0[4]}} & 4'd4)   | ({4{match0[5]}} & 4'd5)
                | ({4{match0[6]}} & 4'd6)   | ({4{match0[7]}} & 4'd7)
                | ({4{match0[8]}} & 4'd8)   | ({4{match0[9]}} & 4'd9)
                | ({4{match0[10]}} & 4'd10) | ({4{match0[11]}} & 4'd11)
                | ({4{match0[12]}} & 4'd12) | ({4{match0[13]}} & 4'd13)
                | ({4{match0[14]}} & 4'd14) | ({4{match0[15]}} & 4'd15);
assign s1_index = ({4{match1[0]}} & 4'd0)   | ({4{match1[1]}} & 4'd1)
                | ({4{match1[2]}} & 4'd2)   | ({4{match1[3]}} & 4'd3)
                | ({4{match1[4]}} & 4'd4)   | ({4{match1[5]}} & 4'd5)
                | ({4{match1[6]}} & 4'd6)   | ({4{match1[7]}} & 4'd7)
                | ({4{match1[8]}} & 4'd8)   | ({4{match1[9]}} & 4'd9)
                | ({4{match1[10]}} & 4'd10) | ({4{match1[11]}} & 4'd11)
                | ({4{match1[12]}} & 4'd12) | ({4{match1[13]}} & 4'd13)
                | ({4{match1[14]}} & 4'd14) | ({4{match1[15]}} & 4'd15);
```

图 2：tlb 寻找 index 代码

随后根据对应项的 ps 位来判定是大页小页和奇偶项，最后输出：

```
// !! 奇偶项选择
    /*
     ! 这一步逻辑需要仔细观察
     ? 仔细思考4KB和4MB下，对于[31:0]的虚地址，该在哪里得到0/1
     * 对于4KB，ps4MB = 0，则va[11:0]为页偏移，va[12]为标识奇偶页号，va[31:13]为虚拟页号，也就是传入的vppn
     * 对于4MB，ps4MB = 1，则va[21:0]为页偏移，va[22]为标识奇偶页号，va[31:23]为虚拟页号，也就是传入的vppn[18:10]
     ! 所以4KB，奇偶号为va[12]--> va_bit12; 4MB，奇偶号为va[22]-->vppn[9]
     */
    assign pick_num0 = tlb_ps4MB[s0_index] ? s0_vppn[9] : s0_va_bit12;
    assign pick_num1 = tlb_ps4MB[s1_index] ? s1_vppn[9] : s1_va_bit12;
    // 得到实际的输出
    assign s0_ppn   = pick_num0 ? tlb_ppn1[s0_index] : tlb_ppn0[s0_index];
    assign s0_ps    = tlb_ps4MB[s0_index] ? 6'd22 : 6'd12;
    assign s0_plv   = pick_num0 ? tlb_plv1[s0_index] : tlb_plv0[s0_index];
    assign s0_mat   = pick_num0 ? tlb_mat1[s0_index] : tlb_mat0[s0_index];
    assign s0_d     = pick_num0 ? tlb_d1[s0_index] : tlb_d0[s0_index];
    assign s0_v     = pick_num0 ? tlb_v1[s0_index] : tlb_v0[s0_index];

    assign s1_ppn   = pick_num1 ? tlb_ppn1[s1_index] : tlb_ppn0[s1_index];
    assign s1_ps    = tlb_ps4MB[s1_index] ? 6'd22 : 6'd12;
    assign s1_plv   = pick_num1 ? tlb_plv1[s1_index] : tlb_plv0[s1_index];
    assign s1_mat   = pick_num1 ? tlb_mat1[s1_index] : tlb_mat0[s1_index];
    assign s1_d     = pick_num1 ? tlb_d1[s1_index] : tlb_d0[s1_index];
    assign s1_v     = pick_num1 ? tlb_v1[s1_index] : tlb_v0[s1_index];
```

图 3：tlb 奇偶项选择代码

读写端口和寄存器十分类似，工作原理不再介绍（只给出写端口定义，读只需把 we 信号删去，除开 index 输入外的 io 反转，w 前缀改成 r 即可）。

Invtlb 指令则是通过 op 信号来选出所有需要被删除的项:

```
generate // * 首先得到匹配的表项
    for(tlb_index = 0; tlb_index < TLBNUM; tlb_index=tlb_index+1)
    begin: __flush_TLB_prepare
        // ! 三项分别代表, G==1; asid对应; va一致(也就是虚拟页表对应, 要么)
        assign attr[tlb_index][0] = tlb_g[tlb_index];
        assign attr[tlb_index][1] = s1_asid == tlb_asid[tlb_index];
        assign attr[tlb_index][2] = (s1_vppn[18:10]==tlb_vppn[tlb_index][18:10]) // 高位必须一一对应
           && (tlb_ps4MB[tlb_index] || s1_vppn[9:0]==tlb_vppn[tlb_index][ 9: 0]); // 低位则当4KB时, 才需要一一对应

        assign inv_match[tlb_index] = ((invtlb_op==0||invtlb_op==1) & 1'b1)  // all
                                    |((invtlb_op==2) & (attr[tlb_index][0]))  // G = 1
                                    |((invtlb_op==3) & (!attr[tlb_index][0]))  // G = 0
                                    |((invtlb_op==4) & (!attr[tlb_index][0]) & (attr[tlb_index][1])) // G=0, ASID一致
                                    |((invtlb_op==5) & (!attr[tlb_index][0]) & attr[tlb_index][1] & attr[tlb_index][2])
                                    |((invtlb_op==6) & (attr[tlb_index][0] | attr[tlb_index][1]) & attr[tlb_index][2]);
    end
endgenerate
```

图 4：inv 指令 match 代码

然后清零：

```
always @(posedge clk ) begin
    if(inv_match[0] & invtlb_valid)
            tlb_e[0] <= 1'b0;
    if(inv_match[1] & invtlb_valid)
            tlb_e[1] <= 1'b0;
    if(inv_match[2] & invtlb_valid)
            tlb_e[2] <= 1'b0;
    if(inv_match[3] & invtlb_valid)
```

图 5：inv 指令清除 tlb 的存在位代码

2、接口定义

表 1：TLB 接口信号表

名称	位宽	方向	功能描述
clk	1	input	时钟信号
s_vppn	19	input	虚双页号，来自访存虚地址[31:13]
s_va_bit12	1	input	来自访存虚地址的第 12 位，在 4KB 页下用来选择奇偶页
s_asid	10	output	标识进程
s_found	1	output	是否找到对应页
s_index	log(表项)	output	对应页的索引
s_ppn	20	output	对应页的物理页号
s_ps	6	input	对应页的大小页标志
s_plv	2	input	对应页的权限
s_mat	2	input	对应页的访存类型
s_d	1	output	对应页的脏位
s_v	1	output	对应页的有效位
...	...	...	分隔线----------
we	1	input	写使能信号
w_index	log(表项)	input	写目的 tlb 项的索引
w_e	1	input	写入 tlb 的存在位
w_ps	6	input	写入 tlb 的页框大小位
w_vppn	19	input	写入 tlb 的虚拟页框号
w_asid	10	input	写入 tlb 的进程标识位
w_g	1	input	写入 tlb 的全局有效位
w_ppn0/1	20	input	写入 tlb 的物理页号（奇偶）
w_plv0/1	2	input	写入 tlb 的权限（奇偶）

名称	位宽	方向	功能描述
w_mat0/1	2	input	写入 tlb 的访存类型（奇偶）
w_d0/1	1	input	写入 tlb 的脏位（奇偶）
w_v0/1	1	input	写入 tlb 的有效位（奇偶）
...	...	...	...
invtlb_valid	1	input	invtlb 指令有效
invtlb_op	5	input	invtlb 指令类型

3、功能描述

支持 IF 和 EXE 阶段的访存需求，根据 crmd 相应位提供虚实地址转换功能。

（三）重要模块 2 设计：vaddr_transfer 虚实地址转换

1、工作原理

通过传入的 crmd 确定翻译模式，通过传入的虚拟地址，操作类型，还有 asid 分别进行三类地址查找：

```
// * 直接映射地址翻译模式
    assign dmw_hit  = dmw_hit0 | dmw_hit1;
    assign dmw_hit0 = csr_dmw0[csr_crmd[1:0]] && (csr_dmw0[31:29]==va[31:29]);
    assign dmw_hit1 = csr_dmw1[csr_crmd[1:0]] && (csr_dmw1[31:29]==va[31:29]);
    assign dmw_pa0  = {csr_dmw0[27:25],va[28:0]};
    assign dmw_pa1  = {csr_dmw1[27:25],va[28:0]};
// * 页表的虚实转换
    // ! output wire
    assign s_vppn =  va[31:13];
    assign s_va_bit12 = va[12];
    assign s_asid =  csr_asid[9:0];
    // ! input wire--翻译
    assign tlb_pa = (s_ps==6'd12)? {s_ppn[19:0],va[11:0]} : {s_ppn[19:10],va[21:0]
    // ! 异常:{PME,PPE,PIS,PIL,PIF,TLBR}
    assign tlb_ex_bus = {6{!direct_mode}} & // 如果是直接模式，就不会由tlb报异常
            {6{!dmw_hit}} &  // 如果dmw命中，也不会考虑tlb异常
            {6{|inst_op}} & {// 如果inst_op有效
            inst_op[1]    & ~s_d, // 页修改异常:: PME
            csr_crmd[1:0] > s_plv,// 页权限异常:: PPI
            inst_op[1]    & ~s_v, // store页无效例外:: PIS
            inst_op[2]    & ~s_v, // 取指页无效:: PIF
            inst_op[0]    & ~s_v, // load页无效:: PIL
            ~s_found};// 页重填例外
```

图 6：vaddr_transfer 的三类地址查找

最后根据模式和命中来输出实地址，并输出 tlb 异常信号线。

```
assign pa = direct_mode ? va:
            (dmw_hit0 ? dmw_pa0 :
            (dmw_hit1 ? dmw_pa1 : tlb_pa));
```

图 7：选择输出物理地址

2、各通道接口信号定义

表 2：Vaddr_transfer 接口信号表

名称	位宽	方向	功能描述
va	32	input	传入的虚拟地址
inst_op	3	input	传入的操作类型{load, store, inst}
pa	32	output	输出的物理地址
tlb_ex_bus	6	output	期间发生的异常{PME,PPE,PIS,PIL,PIF,TLBR}
s_vppn	19	output	和 TLB 相连（下列 s_均是），输出虚拟页号
s_va_bit12	1	output	输出虚拟地址的第 12 位
s_asid	10	output	输出当前的 asid 号

名称	位宽	方向	功能描述
s_found	1	input	从 TLB 输入，是否查找到
s_index	4	input	查找到的 index
s_ppn	20	input	查找到的页表项物理页号
s_ps	6	input	查找到的页表项页框大小
s_plv	2	input	查找到的页表项权限
s_mat	2	input	查找到的页表项访存类型
s_d	1	input	查找到的页表项的脏位
s_v	1	input	查找到的页表项的有效位
csr_asid	32	input	从 csr 传入的 asid
csr_crmd	32	input	从 csr 传入的 crmd
dmw_hit	1	output	是否有 dmw 命中，判断 ade 错
csr_dmw0	32	input	dmw0 信息
csr_dmw1	32	input	dmw1 信息

3、功能描述

与 IF 和 EXE 的访存地址交互, 生成物理地址。

三、实验过程（50%）

（一）实验流水账

2022.11.18　19：00——22：00　exp17 写完

2022.11.21　20：00——21：00　exp17 上板，debug

2022.11.23　15：30——19：30　完成 exp18 代码

2022.11.24　8：00——15：30　调试 exp18 代码，上板

2022.11.29　9：30——20：00　完成 exp19 代码

2022.11.30　11：00——20：58　exp19 的 debug

2022.12.01　13：00——20：20　完成三个实验报告

（二）错误记录

1、错误 1：exp17 上板错

（1）错误现象

数码管最后停在右侧的数码管从 0x00 累加到 0x0f 的地方。

（2）分析定位过程

查询 piazza 得知，检查是否在不同的 always 块里对同一个寄存器赋值

（3）错误原因

在多个 always 块里对同一个寄存器赋值了:

```
        if(we)
                tlb_e[w_index] <= w_e;
    end


// ! 写端口
    always @(posedge clk) begin
        if (we) begin :__TLB_Write

        //      tlb_e[w_index] <= w_e;
```

图 8：错误 1 代码截图

（4）修正效果

将下面注释，并且移动到上方 invtlb 逻辑即可。

（5）归纳总结

这次出错是因为两者功能恰好分开，结果就变成按功能分成两个 always 块了，以后得设计清晰。

2、错误 2：出现了始料未及的 tlb 异常

（1）错误现象

Pc 出错：

```
[8262407 ns] Error!!!
    reference: PC = 0x1c07c7cc, wb_rf_wnum = 0x11, wb_rf_wdata = 0x00000003
    mycpu    : PC = 0x1c00f000, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000000
```

图 9：错误 2 出错情况截图

（2）分析定位过程

查看波形发现是 pc 迟迟没有拉高 we 信号提交：

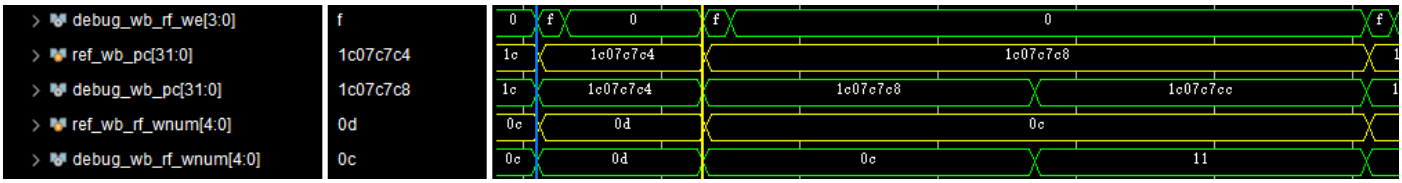

图 10：错误 2 出错波形截图-1

查看指令为：

```
113018    1c07c7c0:    03804c0c    ori $r12,$r0,0x13
113019    1c07c7c4:    03807c0d    ori $r13,$r0,0x1f
113020    1c07c7c8:    040001ac    csrxchg $r12,$r13,0x0
```

图 11：错误 2 出错指令截图

理应产生写寄存器信号，但是没有，于是查找 es 级发现了问题：

```
// ! 增加错误判断：出现异常则置写使能无效
assign es_to_ms_gr_we = es_gr_we & ~(|es_ex_cause_bus_r[15:0]);
```

图 12：错误 2 出错代码截图-1

那么是什么异常呢？答案是 tlb 异常：

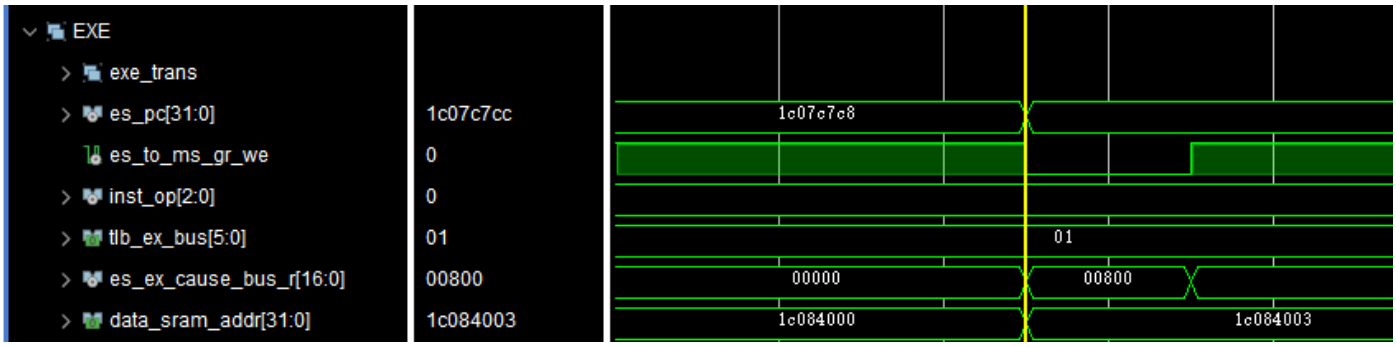

图 13：错误 2 出错波形截图-2

在 inst_op 为 0 的情况下，仍然将异常有效，所以会出 bug。

（3）错误原因

```
// ! 异常:{PME,PPE,PIS,PIL,PIF,TLBR}
assign tlb_ex_bus = {6{!direct_mode}} & // 如果是直接模式, 就不会由tlb报异常
       {6{!dmw_hit}} &  // 如果dmw命中, 也不会考虑tlb异常
       {6{|inst_op}} & {// 如果inst_op有效
       inst_op[1]    & ~s_d, // 页修改异常:: PME
       csr_crmd[1:0] > s_plv,// 页权限异常:: PPI
       inst_op[1]    & ~s_v, // store页无效例外:: PIS
       inst_op[2]    & ~s_v, // 取指页无效:: PIF
       inst_op[0]    & ~s_v, // load页无效:: PIL
       ~s_found};// 页重填例外
```

图 14：错误 2 出错代码截图-2

图中本来是没有对{6{|inst_op}}后来发现，哪怕是没有对总线访问，某些地址照样会产生异常，影响流水级。

（4）修正效果

需要 inst_op 有效后，tlb 异常线才能有效，这样就规避了非法异常的影响。

（5）归纳总结

需要考虑什么时候会发生异常，什么时候不可能发生某类异常，这样才能让处理器尽可能少的出现非法情况。

3、错误 3：没有出对应的异常

（1）错误现象

如图，未发生异常跳转：

```
--------------------------------------------------------------
[8315777 ns] Error!!!
    reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
    mycpu    : PC = 0x1c07cb94, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x02bffc0c
--------------------------------------------------------------
```

图 15：错误 3 出错现象截图

（2）分析定位过程

首先查看汇编指令序列，如图 9 所示，虽然是 b90 地址出问题，但为 b94 指令出问题，因为我们期待这条 ld 指令触发异常，但是我在纸上跑了一遍寄存器，发现不应该出现异常。

```
113260    1c07cb88:    15001f9f    lu12i.w $r31,-524036(0x800fc)
113261    1c07cb8c:    1c00001b    pcaddu12i    $r27,0
113262    1c07cb90:    0280237b    addi.w  $r27,$r27,8(0x8)
113263    1c07cb94:    288003ec    ld.w    $r12,$r31,0
113264    1c07cb98:    5c00b33e    bne $r25,$r30,176(0xb0) # 1c07cc48 <inst_error>
113265    1c07cb9c:    14003a0c    lu12i.w $r12,464(0x1d0)
```

图 15：错误 3 汇编指令序列

百思不得其解下，我查询了 n71_tlb_ex.S 文件，直接锁定了测试用例：

```
### TLB address error in memory access: adem

    //let DA=1 and PG=0 and prl=0
    li.w      t0, 0x08
    li.w      t1, 0x1f
    csrxchg t0, t1, csr_crmd

    LI_EXIMM(t0,s2,IMM_TLBADEM)

    ###fill in a present and valid entry
    FILL_TLB_ITEM(0x0c000000,0x800fc000, 0x000f004d, 0x000a004d)
    csrwr    zero, csr_tlbehi

    //let DA=0 and PG=1 and prl=3
    li.w      t0, 0x13
    li.w      t1, 0x1f
    csrxchg t0, t1, csr_crmd

    li.w       t5, 0x3
    lu12i.w  s7, 0x81
    li.w       s8, 0x800fc000
    la.local s4, 7f
7:
    ld.w  t0, s8, 0x0
    bne   s2, s7, inst_error
    LI_EXIMM(t0,s2,IMM_KERNEL)
    syscall 0  // return to kernel mode
```

图 16：错误 3 测试指令序列

原来是 ADEM 异常，可惜，我搜索了手册，没有找到 ADEM 异常说明。

（3）错误原因

上网搜索后发现，如果不是最高特权等级，访问 0x80000000 的地址是不被允许的。(地址空间限制)

（4）修正效果

加入相关异常的数据通路和判断（在 IF 阶段也是同理），直接 pass！！

```
assign es_ex_cause_bus_r[6'hd/*ADEM    */] =
        (es_res_from_mem||es_mem_we) & es_alu_result[31] & (csr_crmd_rvalue[1:0]!=0) & ~dmw_hit;
```

图 17：错误 3 错误代码修正

4、错误 4：地址错异常中 BADV 寄存器出错

（1）错误现象

在异常处理指令中 BADV 出错:

出错 PC: 0x1c0084f8, ref:0x401fe000, mycpu: 0xffffffff

（2）分析定位过程

查询指令得知为 csrrd $r12, 0x7

这是一个读 BADV 寄存器的指令, 那么第一个反映就是 vaddr 不对. 于是开始查询相关逻辑线路.

（3）错误原因

在原来的逻辑中, ms_alu_result 是直接连到 ms_vaddr 上了, 这也是之前没有出现过指令错的情况造成的后果:

```
// !exp19:: 进行一个选择
assign ms_vaddr = ms_ex_cause_bus[12]? ms_pc : ms_alu_result;
```

图 18：错误 4 代码截图

（4）修正效果

在 ex_cause_bus 中新加入一位来表征是取指错还是访存错, 这样就完成了对错误地址的选择.

（5）归纳总结

这次出错是因为在之前的设计中, 设计人员没有考虑到后面的实现和实验完整性, 只是想通过测试造成的遗漏代码. 另一方面, 也是后续设计人员没有充分考虑前面设计而直接动手, 没有更多地和之前模块的设计人员沟通, 造成的结果. 所以在多人作业中, 合作和沟通是重中之重.

5、错误 5：wb 阶段处理异常有优先级安排

（1）错误现象

当异常传递总线上出现多个异常时, CSR.STAT 寄存器内容出错.

（2）分析定位过程

STAT 寄存器中存储的是 ecode 和 subcode, 很直接地就可以定位到 wb 阶段的 ecode 和 subecode 逻辑.

（3）错误原因

修改异常返回逻辑的优先级即可.

（4）修正效果

将第十二位的 IF/EXE 地址错判断位放到最后即可, 因为这是一个判断位, 而不是决定发生地址异常的异常位, 所以放到相应的异常后面即可.

（5）归纳总结

由于异常的设计有一些问题, 导致总线部分也没有对异常有很好的适应, 最后需要判断异常发生的优先级, 整体看下来, 实现并不是很漂亮, 也许后续有时间可以修改得更好.

四、实验总结

这次任务最难的是而 exp19，它是对整个总线和异常处理系统的挑战。由于初始异常的设计有些问题，在这个实验上花费的时间就多得超乎预期。此外，还有很多没预料的 bug 出现。此外，还需要看手册和 PPT，这样才能迅速解决问题。