实验 13 ~ 15 报告

学号　2019K8009906001
2019K8009929051
姓名　孙彬　代瀚堃
箱子号　49

一、实验任务（10%）

1. 明确 TLB 模块的功能，理解 TLB 模块的读、写和查找操作，按照规定的接口设计 TLB 模块，并集成到该实验提供的模块级验证环境中，完成仿真和上板测试；

2. 掌握 MMU 的相关知识，理解 LoongArch 架构中 MMU 相关的状态寄存器和指令，在流水线中提供支持；

3. 为 CPU 增加 TLB 相关异常：TLB 重填例外、load/store/取指操作页无效例外、页修改例外、页特权等级不合规例外；并实现映射地址翻译模式。

二、实验设计（40%）

（一）总体设计思路

Lab13: TLB 本身是一个二维的查找表，是页表的高速缓存，用于实现虚实地址的快速转换。TLB 模块的功能设计主要分为三个部分：读、写和查找 TLB。其中，我们为读和写分别设置一组通道，为查找 TLB 设置了两组通道（支持取指和取数同时查找）。此外，还实现了 INVTLB 指令所需的操作，但**本次实验并不会用到该指令**。具体设计见以下的 TLB 模块设计介绍。

Lab14: 为 CSR.CRMD 寄存器添加 DA 和 PG 位，添加 CSR.TLBIDX，CSR.TLBEHI，CSR.TLBELO0，CSR.TLBELO1，CSR.ASID 和 CSR.TLBRENTRY 寄存器，并实现 TLBRD，TLBWR，TLBFILL，TLBSRCH 和 INVTLB 五条指令，并按照指令手册的定义去读写上述 CSR 寄存器，其中 TLBRD 和 TLBWR、TLBFILL 指令在 WB 级发起读写请求，而 TLBSRCH 和 INVTLB 指令在 EXE 级发起查找请求，TLBSRCH 需要使用 TLBIDX 寄存器为索引，可能存在**写后读相关**，此处使用阻塞的方式解决。另外，除 TLBSRCH 外的四条指令都有可能修改 TLB 项或者与地址映射有关的 CSR，影响后续指令的取指和访存，还需要引入重取指机制（此处未实现完整的地址映射，所以暂未考虑 CSRWR 和 CSRXCHG 对上述 CSR 的修改）。

Lab15：为 CSR 模块增加 DMW0 和 DMW1 寄存器，并修改 BADV 的 vaddr 以及 TLBTLBEHI 的 vppn 部分，在条件内补充新增的例外。在 IF 级和 EXE 级添加 mmu 模块用于虚实地址的转换。在 IF 级根据地址转换的结果添加 TLB 重填例外（TLBR）、取指操作页无效例外（PIF）、页特权等级不合规例外（PPI），并修改取指地址错例外（ADEM），增加用户态内存地址空间范围 0~2^{31}-1 的限制；在 EXE 级根据地址转换的结果添加 TLB 重填例外

（TLBR）、load+store 操作页无效例外（PIL、PIS）、页特权等级不合规例外（PPI）、页修改例外（PME），并修改访存指令地址错例外（ADEM），同样增加用户态内存地址空间范围 0~2^{31}-1 的限制。

（二）重要模块 1 设计：TLB 模块

1. 工作原理

我们所设计的 TLB 模块主要实现了读、写和查找 TLB 三种功能。读和写 TLB 的逻辑与寄存器堆非常类似，只需要给出目标的 index，再读出或写入 TLB 中对应项即可。读写逻辑采取同步写、异步读的方式。查找 TLB，首先根据 vppn、asid 和 g 域判断查找是否命中，如果命中则计算出命中项的索引 index；然后根据 s0_va_bit12 或是 vppn[9]信号（与 ps 有关）生成 oddpage 信号来选择奇数页或者偶数页，并将其 ppn、ps、plv、mat、d 以及 v 域读出，并返回 found 信号以表示是否查找成功。

2、接口定义

TLB 模块的接口信号如表 1：

名称	方向	位宽	功能描述
s0_vppn	input	19	TLB 查找通道 1：
s0_va_bit12	input	1	s0_va_bit12 指示需要查找的是奇数页还是
s0_asid	input	10	偶数页；s0_asid 表示该页所属的进程；
s0_found	output	1	s0_found 表示该次查找是否查找成功，成功
s0_index	output	4	则为 1，否则为 0；s0_index 表示查找命中
s0_ppn	output	20	的项；
s0_ps	output	6	s0_ps 表示页大小的幂指数；s0_plv 表示特
s0_plv	output	2	权等级；
s0_mat	output	2	s0_mat 表示存储访问类型；s0_d 为“脏”位；
s0_d	output	1	s0_v 是有效位。
s0_v	output	1	
s1_vppn	input	19	TLB 查找通道 2。
s1_va_bit12	input	1	具体信号的含义同 TLB 查找通道 1
s1_asid	input	10	
s1_found	output	1	

s1_index	output	4	
s1_ppn	output	20	
s1_ps	output	6	
s1_plv	output	2	
s1_mat	output	2	
s1_d	output	1	
s1_v	output	1	
invtlb_valid	input	1	INVTLB 指令及其操作码。
invtlb_op	input	5	
we	input	1	TLB 写通道。此处为奇数页和偶数页均提供了写数据。
w_index	input	4	
w_e	input	1	
w_vppn	input	19	
w_asid	input	10	
w_g	input	1	
w_ppn0	input	20	
w_plv0	input	2	
w_mat0	input	2	
w_d0	input	1	
w_v0	input	1	
w_ppn1	input	20	
w_plv1	input	2	
w_mat1	input	2	
w_d1	input	1	
w_v1	input	1	
r_index	input	4	TLB 读通道。将奇数页和偶数页的数据一同读出。
r_e	output	1	
r_vppn	output	19	

r_ps	output	6	
r_asid	output	10	
r_g	output	1	
r_ppn0	output	20	
r_plv0	output	2	
r_mat0	output	2	
r_d0	output	1	
r_v0	output	1	
r_ppn1	output	20	
r_plv1	output	2	
r_mat1	output	2	
r_d1	output	1	
r_v1	output	1	

表 1　TLB 模块的接口定义

3. 功能描述

① index 信号逻辑

如果 TLB 查找成功，那么需要寻找出命中的是哪一项并输出。具体可以根据 match 这个信号中哪一位为 1 来确定 index 信号的值，代码逻辑使用同优先级的多路选择器，将 match 信号每一项的值拓展为 4 位并“与”上其下标，然后把每一项的都“并”起来，最后得到结果，具体代码如下：

```
    assign s0_index = ({4{match0[ 0]}} & 4'd00)
                    | ({4{match0[ 1]}} & 4'd01)
                    | ({4{match0[ 2]}} & 4'd02)
                    | ({4{match0[ 3]}} & 4'd03)
                    | ({4{match0[ 4]}} & 4'd04)
                    | ({4{match0[ 5]}} & 4'd05)
                    | ({4{match0[ 6]}} & 4'd06)
                    | ({4{match0[ 7]}} & 4'd07)
                    | ({4{match0[ 8]}} & 4'd08)
                    | ({4{match0[ 9]}} & 4'd09)
                    | ({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);
```

② 输出数据的选择逻辑

对于 s0_ppn、s0_plv、s0_mat、s0_d、s0_v，他们需要根据 s0_oddpage 的值来确定是输出奇数页还是偶数页数据，如果 s0_odd_page 的值为 1，则输出奇数页的数据，否则输出偶数页的数据，具体代码如下：

```
    assign s0_oddpage = tlb_ps4MB[s0_index] ? s0_vppn[9] : s0_va_bit12;

    assign s0_ps    = tlb_ps4MB[s0_index] ? 6'd22 : 6'd12;
    assign s0_ppn   = s0_oddpage ? tlb_ppn1[s0_index] : tlb_ppn0[s0_index];
    assign s0_plv   = s0_oddpage ? tlb_plv1[s0_index] : tlb_plv0[s0_index];
    assign s0_mat   = s0_oddpage ? tlb_mat1[s0_index] : tlb_mat0[s0_index];
    assign s0_d     = s0_oddpage ? tlb_d1  [s0_index] : tlb_d0  [s0_index];
    assign s0_v     = s0_oddpage ? tlb_v1  [s0_index] : tlb_v0  [s0_index];
```

② INVTLB 指令的实现

INVTLB 指令的查找也将采用并行查找机制，即同时对 TLB 中各项进行匹配判断。设计要点转为每一项的匹配该如何处理。通过分析 INVTLB 指令各操作的定义，发现可以将各操作的匹配分解成若干“子匹配”的逻辑组合，具体来说，可得到 4 个“子匹配”的判断条件：（1）cond1——G 域是否等于 0；（2）cond2——G 域是否等于 1；（3）cond3——s1_asid 是否等于 ASID 域；（4）cond4——s1_vppn 是否匹配 VPPN 和 PS 域。那么 invtlb op=0、1 的匹配条件就可以表达为 cond1||cond2，op=4 的匹配条件可以表达为 cond1&&cond3，op=5 的匹配条件可以表达为 cond1&&cond3&&cond4，op=6 的匹配条件可以表达为 (cond2||cond3)&&cond4。同时我们也很容易得知 op=6 的匹配条件就是取指和访存进行虚实地址转换时的查找匹配条件。对于 INVTLB 指令来说，将对应 TLB 表项无效的操作就是将 inv_match[i] 等于 1 对应的 tlb_e[i] 置为 0，具体代码如下：

```
    generate
        genvar k;
        for(k=0;k<TLBNUM;k=k+1)
        begin: asid_match_gen
            assign asid_match[k] = s1_asid==tlb_asid[k];
        end
    endgenerate

    generate
        genvar l;
        for(l=0;l<TLBNUM;l=l+1)
        begin: vppn_match_gen
            assign vppn_match[l] = (s1_vppn[18:10]==tlb_vppn[l][18:10]) && (tlb_ps4MB[l] ||
s1_vppn[9:0]==tlb_vppn[l][9:0]);
```

```
        end
    endgenerate

    generate
        genvar m;
        for(m=0;m<TLBNUM;m=m+1)
        begin: inv_match_gen
            assign inv_match[m] = (invtlb_op==5'd0)
                                || (invtlb_op==5'd1)
                                || (invtlb_op==5'd2 &&  tlb_g[m])
                                || (invtlb_op==5'd3 && ~tlb_g[m])
                                || (invtlb_op==5'd4 && ~tlb_g[m] && asid_match[m])
                                || (invtlb_op==5'd5 && ~tlb_g[m] && asid_match[m]  &&
vppn_match[m])
                                || (invtlb_op==5'd6 && (tlb_g[m] || asid_match[m]) &&
vppn_match[m]);

        end
    endgenerate

    always @(posedge clk)
    begin
        if (invtlb_valid)
        begin
            tlb_e <= ~inv_match & tlb_e;
        end
        else if (tlbwr_valid | tlbfill_valid)
        begin
            tlb_e[wr_addr] <= w_e;
        end
    end
```

（三）重要模块 2 设计：MMU 相关 CSR 与指令的实现

1. 工作原理

CSR 的变化如下：

CRMD：增加 DA 位和 PG 位，分别为直接和映射翻译模式的使能，在 Lab14 中，软件可以通过 CSR 指令来配置，当执行 ERTN 指令时根据 ESTAT 的 Ecode 来修改这两位；

TLBIDX：存放 TLB 指令操作 TLB 时相关的索引值等信息，包括 Index，PS 和 NE 等域；

TLBEHI：包含 TLB 指令操作时与 TLB 表项高位部分虚页号 VPPN 相关的信息；

TLBELO0/TLBELO1：包含 TLB 指令操作时 TLB 表项低位部分物理页号相关的信息，分别存放偶、奇页号，包含 G、PPN、V、PLV、MAT、D 等域；

ASID：包含用于访存操作和 TLB 指令的地址空间标识符（ASID）信息，包含 ASID 和 ASIDBITS 等域；

TLBRENTRY：供软件配置 TLB 重填例外的入口物理地址。

五条 TLB 指令的工作原理如下：

TLBSRCH：用 ASID 和 TLBEHI 中的信息去查询 TLB，将匹配上的项在 TLB 中的索引值写入到 TLBIDX 的 Index 域，并置 NE 域为 0，若无匹配项，则置 1；

TLBRD：TLBIDX 的 Index 指定了一个 TLB 中的项，如果该项在 TLB 中有效，则将其内容读出至 TLBEHI、TLBELO0、TLBELO1 及 TLBIDX.PS，并置 TLBIDX.NE=0；否则置 TLBIDX.NE=1，其余内容不更新；

TLBWR：TLBIDX 的 Index 指定了一个 TLB 中的位置，将 TLBEHI、TLBELO0、TLBELO1、TLBIDX.PS 以及 TLBIDX.NE 中的内容填到该 TLB 项中；

TLBFILL：所填内容与 TLBWR 相同，只是填入的 TLB 项的位置由硬件决定；

INVTLB：在 TLB 中按照 op 指定的规则查找匹配项（规则在报告第 5 页已经详述，此处不再展开），将其 E 位置为 0，设置为无效项。

2、接口定义

首先我们对原先 TLB 模块的接口稍作了些调整以支持 Lab14 的变化：

名称	方向	位宽	功能描述
tlbsrch_valid	input	1	用于发起查找命令，高电平有效
tlbsrch_done	output	1	查找的数据是否返回，用于更新 CSR 寄存器
tlbwr_valid	input	1	即原来的 we 信号，用于发起写命令
tlbfill_valid	input	1	用于发起 TLBFILL 填 TLB 项的命令
tlbrd_valid	input	1	用于发起读命令
tlbrd_done	output	1	读的数据是否返回，用于更新 CSR 寄存器

表 2　Lab14 中 TLB 模块的接口调整

当然，由于 TLB 的读和查找在发起请求的当拍就能完成，所以上面的 done 就直接等于相应的 valid，此处设置 done 信号的目的是和返回的数据一起传到 CSR 模块中用于更新 CSR 寄存器。

另外，由于这 5 条指令可能需要一次性读出或写入多个 CSR 寄存器，所以我们为 CSR 模块增设如下 3 个通道：

名称	方向	位宽	功能描述

tlb_to_csr_bus	input	105	传输 TLBSRCH 和 TLBRD 从 TLB 中返回的数据，及数据是否有效的信号
csr_to_exe_bus	output	19	TLBSRCH 和 INVTLB 指令在 EXE 级使用的 TLBEHI.VPPN
csr_to_tlb_bus	output	108	执行 TLBWR 和 TLBFILL 时写入到 TLB 中的信息

表 3　Lab14 中 CSR 模块的接口调整

同时，在设计 MMU 模块时，需要 CSR 寄存器的相关信心，所以我们为 CSR 模块增设如下 4 个信号：

名称	方向	位宽	功能描述
csr_crmd_rvalue	output	32	执行地址转换时写入到 MMU 中的信息
csr_asid_rvalue	output	32	
csr_dmw0_rvalue	output	32	
csr_dmw1_rvalue	output	32	

表 4　Lab15 中 CSR 模块的接口调整

3. 功能描述

5 条 TLB 指令的实现及 CSR 寄存器的增加和调整在工作原理中已经叙述，此处不再重复。表 3 的 3 条通道中，tlb_to_csr_bus 上的信号由 WB 级从 TLB 模块中接收并整理后传入，在 CSR 模块中分离出各个信号。其中，WB 级中的赋值如下：

```
    assign tlb_to_csr_bus = {tlbsrch_done,  //104:104
                             s1_index    ,  //103:100
                             ~s1_found   ,  //99:99
                             tlbrd_done  ,  //98:98
                             r_vppn      ,  //97:79
                             r_v0        ,  //78:78
                             r_d0        ,  //77:77
                             r_plv0      ,  //76:75
                             r_mat0      ,  //74:73
                             r_g         ,  //72:72
                             r_ppn0      ,  //71:48
                             r_v1        ,  //47:47
                             r_d1        ,  //46:46
                             r_plv1      ,  //45:44
                             r_mat1      ,  //43:42
```

```
                        r_g         ,  //41:41
                        r_ppn1      ,  //40:17
                        r_asid      ,  //16:7
                        r_ps        ,  //6:1
                        ~r_e           //0:0
                        };
```

csr_to_tlb_bus 上信号的整合由 CSR 模块完成，而在 WB 级中拆分，重新整理后对接到 TLB 模块上：

```
    assign csr_to_tlb_bus = {csr_estat_ecode,   //107:102
                             csr_tlbidx_idx ,   //101:98
                             csr_tlbidx_ps  ,   //97:92
                             csr_tlbidx_ne  ,   //91:91
                             csr_tlbehi_vppn,   //90:72
                             csr_tlbelo0_v  ,   //71:71
                             csr_tlbelo0_d  ,   //70:70
                             csr_tlbelo0_plv,   //69:68
                             csr_tlbelo0_mat,   //67:66
                             csr_tlbelo0_g  ,   //65:65
                             csr_tlbelo0_ppn,   //64:41
                             csr_tlbelo1_v  ,   //40:40
                             csr_tlbelo1_d  ,   //39:39
                             csr_tlbelo1_plv,   //38:37
                             csr_tlbelo1_mat,   //36:35
                             csr_tlbelo1_g  ,   //34:34
                             csr_tlbelo1_ppn,   //33:10
                             csr_asid_asid      //9:0
                            };
```

TLB 的 w_e 和 w_g 端口需要稍加处理，其他端口可以直接对接：

```
    assign w_e           = (csr_estat_ecode ==`ECODE_TLBR) ? 1'b1 : ~csr_tlbidx_ne;
    assign w_g           = csr_tlbelo0_g & csr_tlbelo1_g;
```

CSR 模块将 EXE 级需要使用的信息传入 csr_to_exe_bus，经 WB 级的传递，到 EXE 级中拆分出来，用于 INVTLB 和 TLBSRCH 发起命令：

```
    assign csr_to_exe_bus = csr_tlbehi_vppn;
```

在这一部分中需要详细描述的是我们如何处理新加入的指令和控制寄存器引发的冲突。首先，对于 TLBSRCH 指令的写后读冲突，我们采用阻塞的方式来解决。类似于此前使用的中断标志，当前一条指令处在 ID 级时，如果识别出修改了上述寄存器，就置起 tlbsrch_revised 标志，并跟随该指令在流水线中传递，如果后面的 TLBSRCH 在 EXE 级读取到 MEM 级和 WB 级存在上述标志，就置 es_ready_go 为 0，将 TLBSRCH 阻塞在 EXE 级，并置 tlbsrch_valid 为 0，不发起查找命令，直至上述标志解除。

TLBWR、TLBFILL 和 INVTLB 会修改 TLB 项，可能导致后续指令的地址映射发生改变，而 TLBRD、CSRW

R 和 CSRXCHG 可能会修改与地址映射有关的寄存器，也会影响后续指令的取指和访存，我们用重取指机制解决。当这些指令在 ID 级译出时，如果与地址映射有关的寄存器被修改，则置起 memmap_revised 标记（由于 Lab14 还未实现完整的地址映射，所以此处我们暂未考虑 CSRWR 和 CSRXCHG），表示下一条指令需要重取，类似于中断标记和 ERTN 指令刷新流水线，该标记以及下一条指令的 PC 虚地址跟随指令在流水线中传递，到达 WB 级时传递至 pre-IF 级用于重取指，当 pre-IF 级的 do_refetch 信号有效时，按 pc_refetch 取指。另外，如果当前指令在 EXE 级检测到前面处于 MEM 和 WB 级的指令存在上述 memmap_revised 标记时，不应发起访存和 TLB 查找请求，以免出错。由于 TLBWR、TLBFILL 和 TLBRD 的读写请求都在 WB 级发起，所以不存在写后读相关。

同时，当 EXE 级执行 INVTLB 指令时，若 MEM 级有 TLBRD\WR\FILL 指令在执行的话，需要将 INVTLB 指令阻塞一拍，直到 TLBRD\WR\FILL 进入 WB 级。

最后，INVTLB 指令有可能会触发例外，指令手册中定义了 op=0~6 的情况，如果指令中带有的 op 不在该范围内，最早在 ID 级识别出来，此时置起 illegal_opcode 信号，生成 inst_no_existing=1’b1，并置起例外标志，复用之前实验的例外处理流程即可。

（四）重要模块 3 设计：MMU 转换

1. 工作原理

我们所设计的 MMU 模块支持直接地址翻译模式、直接映射地址翻译模式和页表映射地址翻译模式三种地址翻译模式，并根据前文顺序进行比较，确定虚拟地址所对应的物理地址。同时，如果是页表映射地址翻译模式三种地址翻译模式，还需要根据讲义所述确定 TLB 相关的例外信息。

2、接口定义

TLB 模块的接口信号如表 5：

名称	方向	位宽	功能描述
vaddr	input	32	虚拟地址
is_inst	input	1	取指操作
is_load	input	1	load 操作
is_store	input	1	store 操作
csr_crmd	input	32	CRMD 寄存器
csr_asid	input	32	ASID 寄存器
csr_dmw0	input	32	DMW0 寄存器

csr_dmw1	input	32	DMW1 寄存器
paddr	output	32	物理地址
tlb_ex_bus	output	6	tlb 相关例外信息
s_vppn	output	19	传入 TLB 模块，用于查找相应页表项
s_va_bit12	output	1	
s_asid	output	10	
s_found	input	1	tlb 是否命中及相应信息
s_index	input	4	
s_ppn	input	20	
s_ps	input	6	
s_plv	input	2	
s_mat	input	2	
s_d	input	1	
s_v	input	1	
dmw_hit	output	1	直接映射模式是否命中

表 5　MMU 模块的接口定义

需要关注的是，在 IF 级中，tlb 相关信息所走的是 s0 通道，而 EXE 级中走的是 s1 通道。

3. 功能描述

① 虚实地址翻译

首先根据 CRMD 中的 DA 和 PG 位来确定是直接地址翻译模式还是映射地址翻译模式，具体代码如下：

```
    assign direct     = csr_crmd[`CSR_CRMD_DA] & ~csr_crmd[`CSR_CRMD_PG];
```

接着再确定 DMW0 与 DMW1 寄存器是否命中，主要过程就是先看当前特权级是否支持，再看地址的高三位是否命中，从而确定直接映射地址翻译模式下的物理地址，具体代码如下：

```
    assign dmw0_hit   = csr_dmw0[csr_crmd[`CSR_CRMD_PLV]] && (csr_dmw0[`CSR_DMW0_VSEG] ==
vaddr[31:29]);
    assign dmw1_hit   = csr_dmw1[csr_crmd[`CSR_CRMD_PLV]] && (csr_dmw1[`CSR_DMW1_VSEG] ==
vaddr[31:29]);
    assign dmw0_paddr = {csr_dmw0[`CSR_DMW0_PSEG],vaddr[28:0]};
    assign dmw1_paddr = {csr_dmw1[`CSR_DMW1_PSEG],vaddr[28:0]};
    assign dmw_hit    = dmw0_hit | dmw1_hit;
```

最后根据 PS 位来拼接出页表映射翻译模式下对应的物理地址，代码如下：

```
    assign tlb_paddr = (s_ps == 6'd12) ? {s_ppn[19: 0],vaddr[11:0]}
                                       : {s_ppn[19:10],vaddr[21:0]};
```

那么，最后的物理地址即为：

```
    assign paddr = direct   ? vaddr
                 : dmw0_hit ? dmw0_paddr
                 : dmw1_hit ? dmw1_paddr
                            : tlb_paddr;
```

② TLB 相关例外

关于 tlb 的可能例外，可以根据讲义的伪代码得出相应结果，具体代码如下：

```
    assign ecode_pil  = ~dmw_hit & is_load  & ~s_v;
    assign ecode_pis  = ~dmw_hit & is_store & ~s_v;
    assign ecode_pif  = ~dmw_hit & is_inst  & ~s_v;
    assign ecode_pme  = ~dmw_hit & is_store & ~s_d;
    assign ecode_ppi  = ~dmw_hit & (csr_crmd[`CSR_CRMD_PLV] > s_plv) & (is_inst | is_load
| is_store);
    assign ecode_tlbr = ~dmw_hit & ~s_found & (is_inst | is_load | is_store);

    assign tlb_ex_bus = direct ? 6'b0
                               : {ecode_pil,
                                  ecode_pis,
                                  ecode_pif,
                                  ecode_pme,
                                  ecode_ppi,
                                  ecode_tlbr};
```

三、实验过程（50%）

（一）实验流水账

Lab13:

2021/11/17	14:00 ~ 18:00	阅读讲义
2021/11/18	20:00 ~ 24:00	设计 tlb 模块
2021/11/19	9:00 ~ 12:00	调试
2021/11/30	20:00 ~ 22:00	书写实验报告

Lab14:

2021/11/27	8:00 ~ 12:00	阅读讲义及指令手册

2021/11/27　　14:00 ~ 18:00　调整 TLB 模块及模块间接线

2021/11/28　　8:00 ~ 20:00　调整 CSR 模块并实现 5 条新增指令

2021/11/29　　19:00 ~ 23:00　调试

2021/11/30　　13:00 ~ 15:00　调试

2021/12/1　　18:30 ~ 24:00　书写实验报告

Lab15:

2021/12/3　　19:00 ~ 23:00　阅读讲义及指令手册

2021/12/4　　13:30 ~ 24:00　增加 MMU 模块及模块连接线

2021/12/5　　14:00 ~ 22:00　调整流水线并实现 6 种 TLB 例外

　　22:00 ~ 24:00　调试

2021/12/6　　16:00 ~ 22:00　调试

2021/12/7　　8:30 ~ 12:00　书写实验报告

（二）错误记录

1、错误 1：写操作与 INVTLB 冲突

（1）错误现象

在仿真时显示写操作出错。

（2）分析定位过程

查看先前的写操作，发现其与 INVTLB 指令相冲突：

在 tlb_top.v 文件中可以看到：

```
.invtlb_op   (5'd0          ),
```

而根据指令手册中的操作码可知：

op	操作
0x0	清除所有页表项。
0x1	清除所有页表项。此时操作效果与 op=0 完全一致。
0x2	清除所有 G=1 的页表项。
0x3	清除所有 G=0 的页表项。
0x4	清除所有 G=0，且 ASID 等于寄存器指定 ASID 的页表项。
0x5	清除 G=0，且 ASID 等于寄存器指定 ASID，且 VA 等于寄存器指定 VA 的页表项。
0x6	清除所有 G=1 或 ASID 等于寄存器指定 ASID，且 VA 等于寄存器指定 VA 的页表项。

图 1　invtlb 指令操作码

该操作即为将所有页表项的 e 位置 0。而写操作是将所有页表项的 e 位置 0，所以查看波形就会发现，每段时间只有一个页表项的 e 位被置 0：

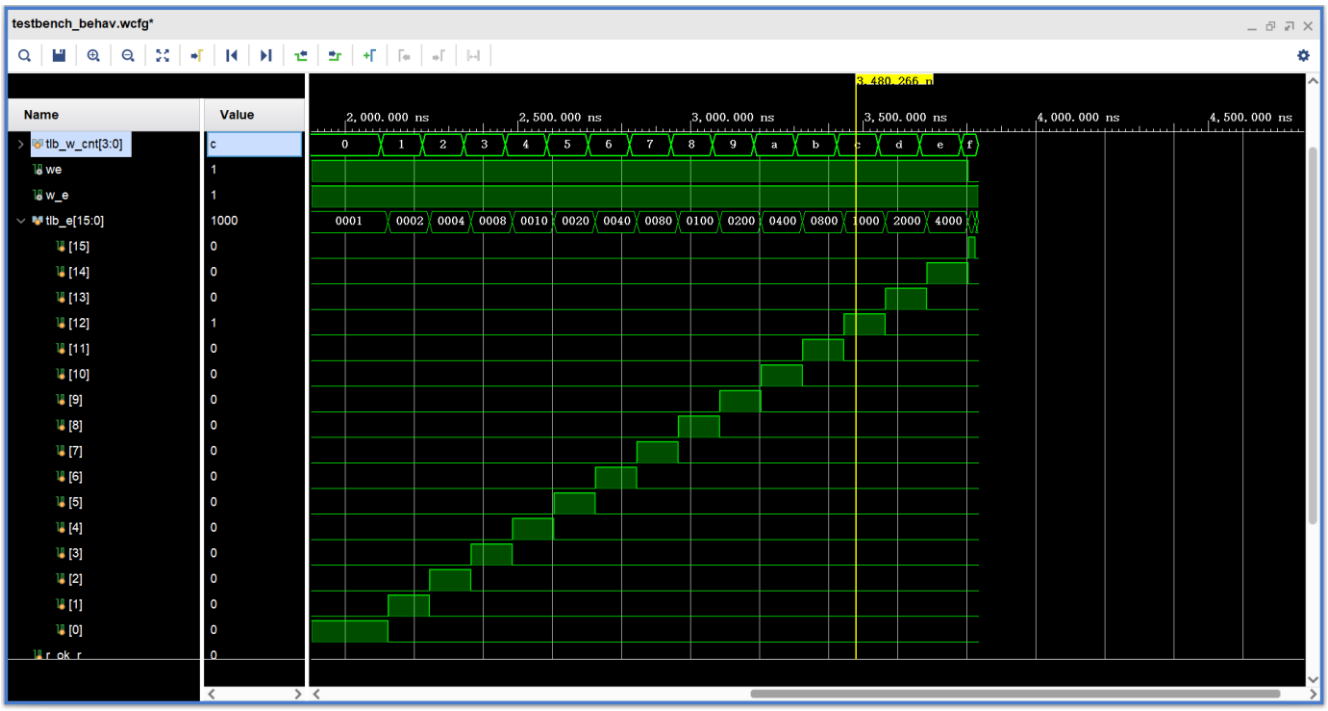

图 2　写操作波形

但是，在读操作的测试程序中，观察到所用的测试代码的 tlb_e 为全 1，因此构成矛盾。

（3）错误原因

由于实验开始时间较早，所以仍以为本实验需要处理 INVTLB 指令，导致其与 tlb 写操作构成冲突。

（4）修正效果

加入一个 invtlb_valid 信号，表示开始执行 invtlb_op 操作。并在模块里使该信号接 0。

（5）总结

太早写实验也不是件好事啊（笑），走了不少弯路。

2、错误 2：va_bit12 未分情况讨论

（1）错误现象

查找操作时出错。

（2）分析定位过程

查看波形后发现问题出在第一次查找操作，本应查看的是偶数页，但查找结果却是奇数页：

图 3　查找操作波形

（3）错误原因

继续查看测试文件：

```
//search
// s_vppn  |  s_va_bit12  | s_asid  |  s_found  |  s_index  |  s_ppn  | s_ps | s_plv s_mat s_d s_v |
//  0x1000 |        0x1  |   0x0  |       1  |       0  |  0x1000 | 0x16 |     0,1,1,1         |
```

一开始觉得并没有出错，s_va_bit12 是 1，那就应该是奇数页，后来才发现此时的 ps 位是 22，也就是说页大小为 4M，此时的奇偶页判断位并不是 s_va_bit12，而是 vppn[9]。

（4）修正效果

分类讨论，根据 tlb_ps4MB[s_index]的值来选择 s_oddpage：

```
assign s1_oddpage = tlb_ps4MB[s1_index] ? s1_vppn[9] : s1_va_bit12;
```

3、错误 3：未更新指令不存在例外的产生逻辑

（1）错误现象

在第 59 个测试点中，PC=0 处死循环：

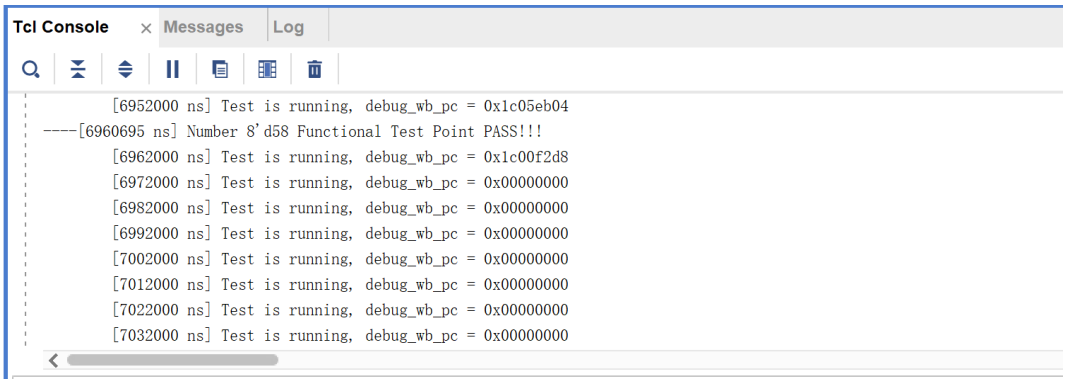

图 4　错误 3 的错误现象

（2）分析定位过程

我们找到 PC 进入 0 的死循环之前执行完成的最后一条指令，查看波形图，发现在 0x1c027b90 处：

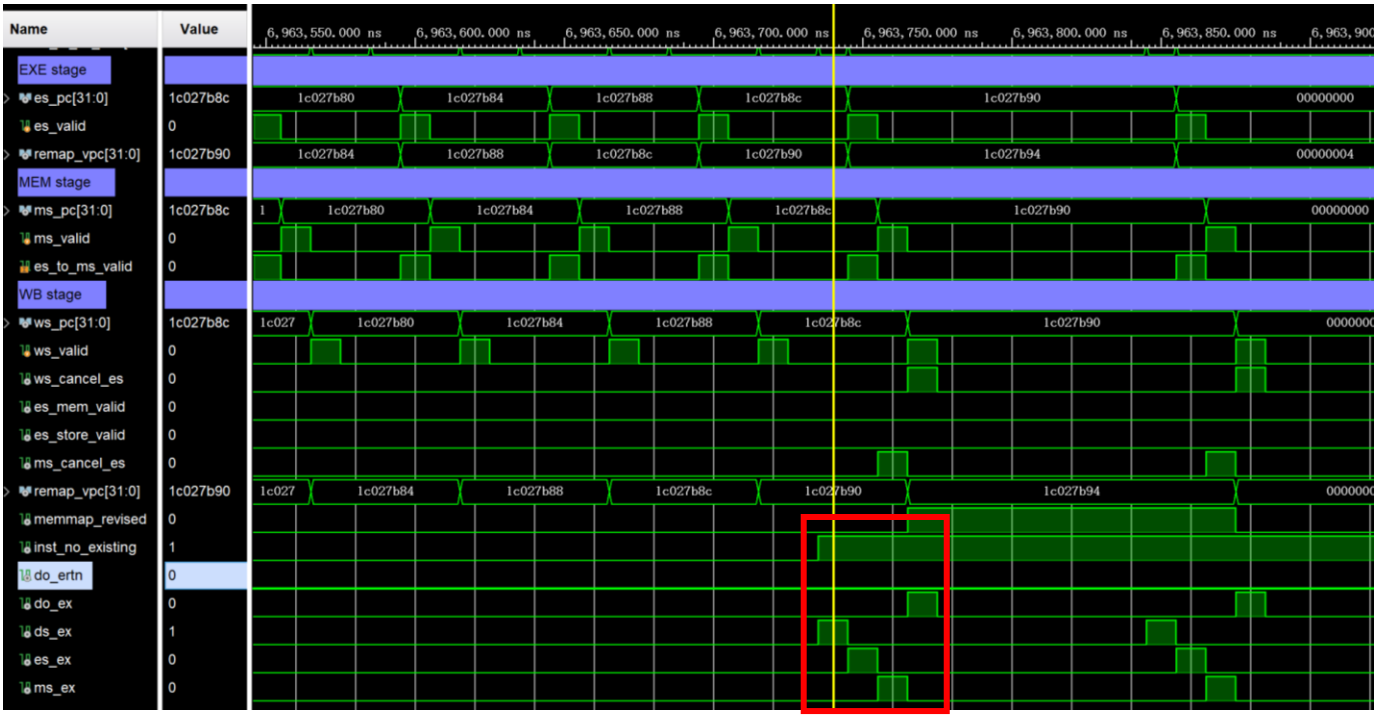

图 5　错误 3 的波形图

查看汇编文件，这是一条 tlbwr 指令：

```
1c027b90:   06483000    tlbwr
```

跳转到奇怪的地方取指，说明取指的地址/条件有问题，按照我们的设计，pre-IF 级用于取指的 nextPC 有 6 个来源，依次为复位后的初始值、ertn 的返回地址、ex_entry 的例外入口、重取指的 PC、分支跳转的 PC 和顺序 PC。初始值不会出错，我们先来看表示 ERTN 返回的信号 do_ertn 是否有效，从图 5 的红框中可以看出，它为低电平，所以应该不是由 do_ertn 引起的，于是我们查看第二个条件 do_ex 是否有效，发现该信号为高电平，于是我们联想到，此时例外入口可能还未配置，当 do_ex 有效后，nextPC 就被置成了 0，再顺着向上查看前几个流水级的 ms_ex、es_ex 和 ds_ex，发现都为有效，在 ID 级 ds_ex 的来源不外乎 4 个，即 fs_ex、break、syscall 和指令不存在引发的例

外（中断标志为单独的 has_int 信号），想到这里，我们已经反应过来了，我们新加了 5 条指令，但这 5 条指令并没有在 inst_no_existing 的产生逻辑中排除掉，于是 tlbwr 被识别成一条无效指令，将 inst_no_existing 信号添加到波形图中，发现它确实被置成了有效，验证了我们的想法。

（3）错误原因

如上所述，添加 5 条新指令后，没有在 inst_no_existing 中排除，被误认为是无效指令，触发例外

（4）修正效果

将 inst_tlbwr 等 5 个信号取非后再取与接到原来 inst_no_existing 的后面，另一方面，我们还意识到，指令手册中指出了 INVTLB 的操作码无效时也应该触发例外，于是增加相应的判断逻辑，重新仿真，运行到下一个错误点。

4、错误 4：未更新指令不存在例外的产生逻辑

（1）错误现象

在第 59 个测试点的 0x1c027af8 处出错，debug_wb_rf_wdata 正确值应为 0x00012000，但实际写入 0x0001c000

（2）分析定位过程

我们查看汇编文件中前面的几条指令：

```
1c027ae8:   1418000c    lu12i.w $r12,49152(0xc000)
1c027aec:   0380058c    ori $r12,$r12,0x1
1c027af0:   0400402c    csrwr   $r12,0x10
1c027af4:   06482c00    tlbrd
1c027af8:   0400440d    csrrd   $r13,0x11
```

0x10 号 CSR 是 TLBIDX，0x11 号 CSR 是 TLBEHI，从这几条指令中可以看出，我们先向 TLBIDX 中配置 Index=0x1，PS=0xc，然后用 tlbrd 把 TLB 中的第 1 项读取出来，这表明我们从 TLB 中读取到了错误的内容。根据 TLBEHI 的设计，我们读到的错误数据是 VPPN=0x0000f，而正确值应该是 VPPN=0x00009。一般来说，一条指令出错很有可能是由它附近的几条指令引起的，但我们分析了前面的很多条指令，发现它们执行的结果都没有问题，查看 TLB 中 tlb_vppn 的波形，第 1 项的内容也一直都是 0x0000f，说明这一错误应该不是来自前面的这些指令，可能在一段时间之前的某个地方就往 TLB 中写入了错误的内容，但直到 0x1c027af8 处才被发现。既然正确值是 0x00009，我们就来查看金标准的波形，查找这一正确值最先被写到 TLB 中是在什么位置。我们添加了金标准的 tlb_0_VPPN ~ tlb_3_VPPN 的波形，发现 tlb_0_VPPN 最早在 0x1c027b90 后被赋值成 0x00009，查看附近的几条汇编指令：

```
1c027b80:   0400402e    csrwr   $r14,0x10
1c027b84:   0400442f    csrwr   $r15,0x11
1c027b88:   04004830    csrwr   $r16,0x12
1c027b8c:   04004c31    csrwr   $r17,0x13
1c027b90:   06483000    tlbwr
```

我们在此处向 CSR 中配置了页表项，然后用 TLBWR 写入到 TLB 中。查看我们自己的波形，该值在 0x1c027b90 处确实写入到了 tlb_vppn 的 0 号位置，但在紧接着的一段波形中，我们发现了异常：

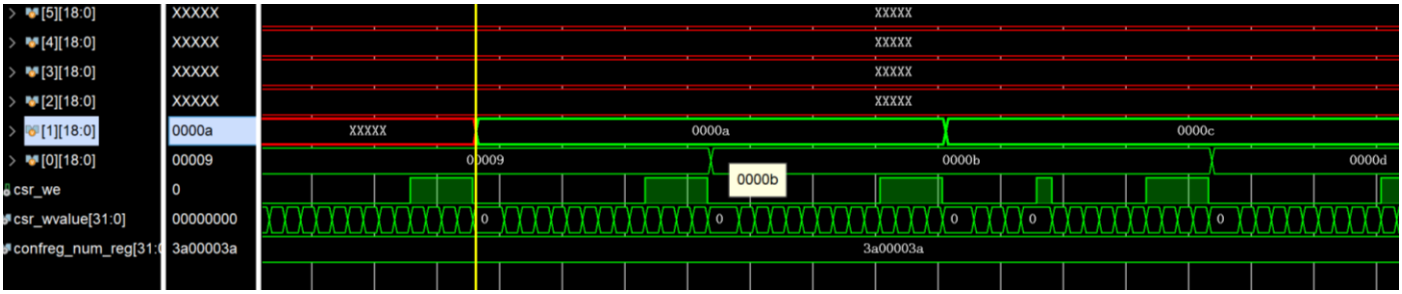

图 6　错误 4 的波形图

即原来的值不久后就被 0x0000b 覆盖了，然后又被 0x0000d 覆盖，然后又经过几次覆盖，最后读到的值不再是原先的 0x00009。根据文件中的标号，这是一段 TLB 初始化的函数，说明我们向 TLB 中写初始值时写错了位置，于是查看金标准的波形，检查这些初始值正确的写入位置：

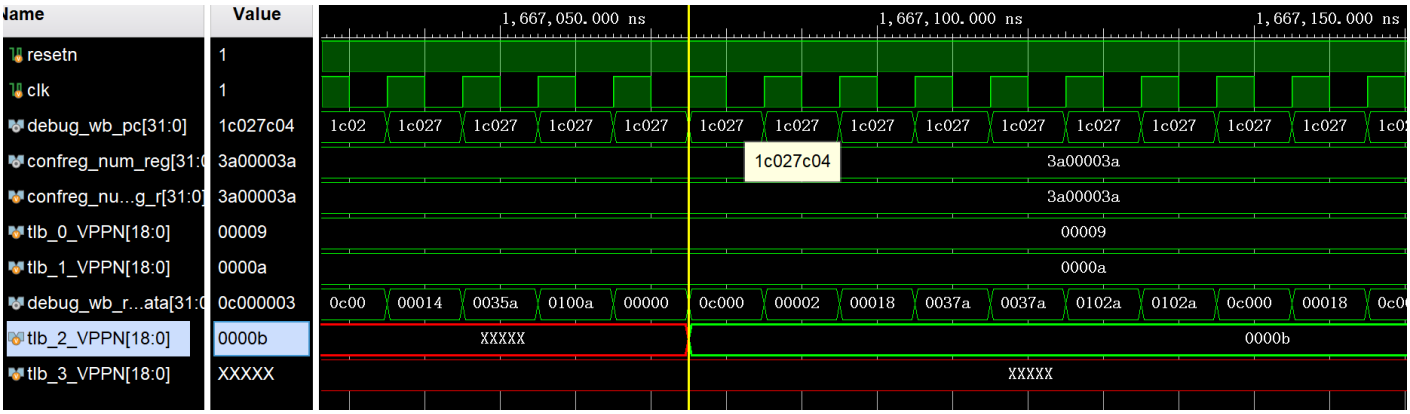

图 7　错误 4 的金标准波形

不久后的 0x1c027c04 处也是一条 TLBWR 指令，0x0000b 这个值应该写到 TLB 中的第 2 项，但我们写在了第 0 项，于是出错。我们来查看代码中 tlb_vppn 的更新逻辑：

```
        if (tlbwr_valid | tlbfill_valid)
        begin
            tlb_vppn [wr_addr] <= w_vppn;
        end
```

将 tlbwr_valid 添加至波形，发现它已经是有效，说明 wr_addr 有问题，查看代码，发现我们将这一信号声明为 wire 后就直接赋了值，没有声明位宽。

（3）错误原因

没有声明 wr_addr 信号的位宽，其位宽为 4，导致它只有最低位的值，只能在 TLB 的第 0 和第 1 项的位置写。

（4）修正效果

声明了该信号的位宽，重新仿真，前进到下一个错误点。

5、错误 5：对 TLBRD 读出内容的屏蔽保护理解有误

（1）错误现象

在 PC=0x1c01a628 处写入 GPR 的数据出错：

图 8　错误 5 的错误信息

（2）分析定位过程

我们查看汇编文件中附近的几条指令：

```
1c01a60c:   06498000    invtlb  0x0,$r0,$r0
1c01a610:   1418000c    lu12i.w $r12,49152(0xc000)
1c01a614:   02804190    addi.w  $r16,$r12,16(0x10)
1c01a618:   00100000    add.w   $r0,$r0,$r0
1c01a61c:   0280018d    addi.w  $r13,$r12,0
1c01a620:   0400402d    csrwr   $r13,0x10
1c01a624:   06482c00    tlbrd
1c01a628:   0400440d    csrrd   $r13,0x11
```

根据文件中的标号，这一错误发生在第 62 号测试，主要是测试 INVTLB 指令。前面 0x1c01a60c 处的 INVTLB 指令将所有 TLB 项置为无效，然后配置了 TLBIDX，将其中的某一项读出来，这中间没有别的操作会改 TLB，所以读出来的这项应该是无效的，说明我们在读到无效 TLB 项时的屏蔽保护没有做好，或者是 INVTLB 指令的实现有问题，导致没有将 TLB 项无效掉。我们先来检查后者，查看执行这条 TLBRD 指令时 TLB 中 E 位的波形：

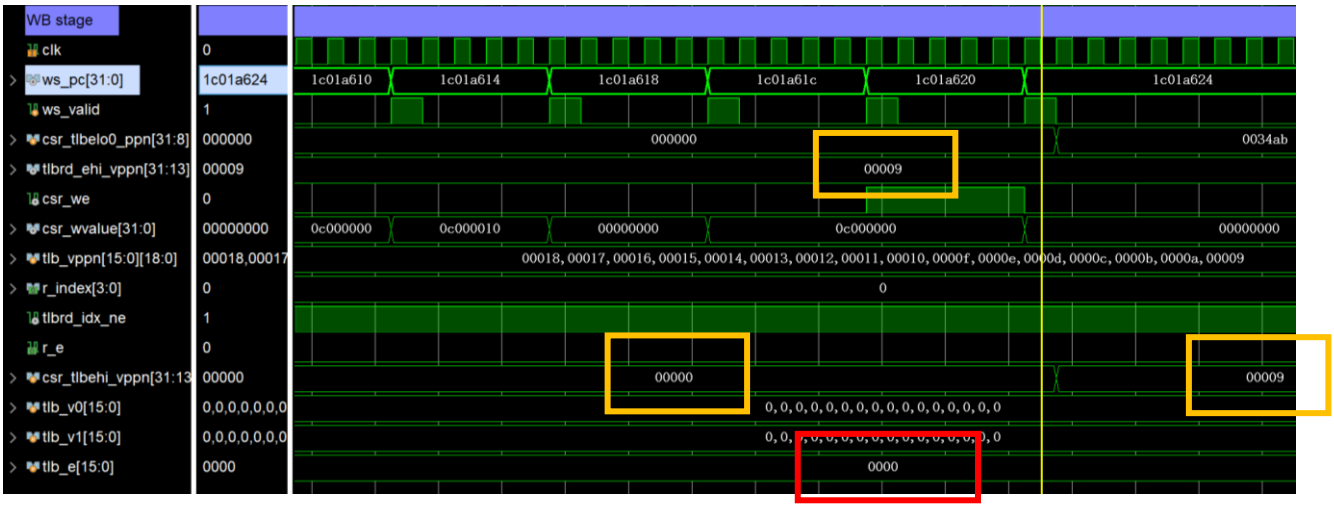

图 9　错误 5 的波形图

我们在图 9 中用红框圈出，此时 TLB 项确实无效，所以应该不是 INVTLB 的问题，可能是屏蔽保护的实现有误。为了进一步确认这一点，我们来看金标准的波形，如图 10：

图 10　错误 5 的金标准波形

我们用红框圈出，在执行完 0x1c01a60c 处的 INVTLB 指令后，tlb 中所有项（这里只选了 0 和 1）的 E 位被置为 0，而执行到 0x1c01a624 处的 TLBRD 指令时，由于读到的是一个无效项，所以应该对 TLBEHI.VPPN 进行屏蔽保护，如图 10 中黄框所示，由于 TLBEHI.VPPN 中原来的内容就是 0x00000，所以读出的 0x00009 不应该覆盖掉它。我们再回过来看图 9，如黄框所示，我们从第 0 项中读出了 0x00009，是无效项，但它把原来的 0x00000 覆盖掉了。这时我们才意识到我们此前的理解有误，我们一直认为，不更新 TLBEHI 中内容的意思是，把读到的内容原封不动地搬到 TLBEHI 中，但这实际上相当于没有屏蔽保护，无效的内容还是被搬到了 TLBEHI，正确的理解应该是，不更新就是保持原来寄存器中的内容不变。这一错误听上去有些让人难以理解，但我们小组的成员在调试时确实没能转过这个弯来。

（3）错误原因

没有正确理解 TLBRD 读到无效项的屏蔽保护，导致无效项覆盖了 CSR 中原有的内容

（4）修正效果

将 TLBEHI 和 TLBELO 等寄存器的更新条件改为 tlbrd_done & ~tlbrd_idx_ne 后，仿真通过

6、错误 6：tlb_e 的赋值存在多驱动

（1）错误现象

仿真通过，而实现失败，错误报告显示 tlb_e 寄存器存在多驱动：

图 11　错误 6 的错误报告

（2）分析定位过程

查看 tlb_e 的赋值逻辑，发现我们在两个 always 块内更新了 tlb_e，第一次是 INVTLB 的清空，第二次是在另一个 always 块中，在 TLBWR 指令下和 TLB 项的其他域一起更新，在我们的设计中，这两个更新条件不会同时有效，仿真也通过了，但在实现阶段被检测了出来。

（3）错误原因

用 always 块更新寄存器时按照指令的功能来划分，而没有把一个寄存器放在一个 always 块内，更新条件相同时才合并。

（4）修正效果

只将 tlb_e 的更新放在一个 always 块内：

```
always @(posedge clk)
begin
    if (invtlb_valid)
    begin
        tlb_e <= ~inv_match & tlb_e;
    end
    else if (tlbwr_valid | tlbfill_valid)
    begin
        tlb_e[wr_addr] <= w_e;
    end
end
```

重新仿真和实现，并上板通过。

7、错误 7：tlbelo 模块中的 mat 位写操作出错

（1）错误现象

```
------------------------------------------------------------
[7257977 ns] Error!!!
    reference: PC = 0x1c06e344, wb_rf_wnum = 0x12, wb_rf_wdata = 0x01c0084d
    mycpu    : PC = 0x1c06e344, wb_rf_wnum = 0x12, wb_rf_wdata = 0x01c0087d
------------------------------------------------------------
```

图 12　错误 7 的错误报告

（2）分析定位过程

根据错误报告给出的 PC 值查看汇编指令：

```
1c06e344:  04004832   csrwr   $r18,0x12
```

可以看出，是在向 TLBELO0 寄存器写值的时候读出的值发生了错误，于是我们查看波形中 csr_tlbelo0_rvalue 的值最后改变的位置，发现此时写入的值错误，但是再次查看波形我们可以发现，csr_wvalue，也就是写入的值并没有出错，那么也就说明是在 csr 内部模块中的 TLBELO0 写逻辑出现了错误。

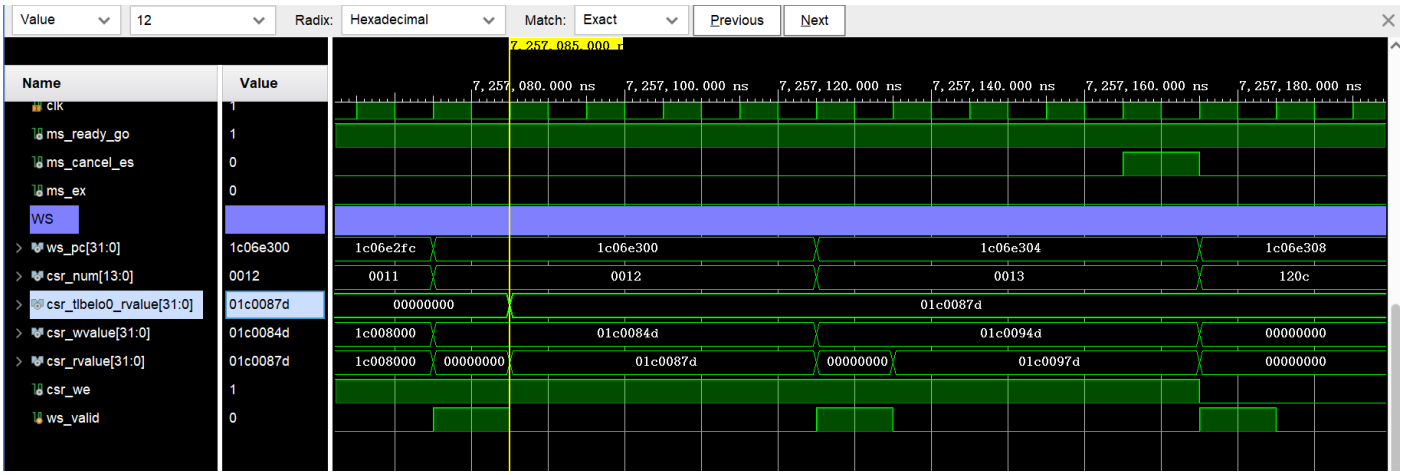

图 13　错误 7 的波形

（3）错误原因

根据 0x1c0084d 和 0x1c0087d 的差别，我们可以发现出错的位是 mat 部分，查看该部分的写逻辑：

```
    always @(posedge clock)
        begin
            if (csr_we && csr_num==`CSR_TLBELO0)
            begin
                csr_tlbelo0_mat <= csr_wmask[`CSR_TLBELO0_PLV] &
csr_wvalue[`CSR_TLBELO0_PLV]
                                | ~csr_wmask[`CSR_TLBELO0_PLV] & csr_tlbelo0_plv;
            end

            if (csr_we && csr_num==`CSR_TLBELO1)
            begin
                csr_tlbelo1_mat <= csr_wmask[`CSR_TLBELO1_PLV] &
csr_wvalue[`CSR_TLBELO1_PLV]
```

```
                                | ~csr_wmask[`CSR_TLBELO1_PLV] & csr_tlbelo1_plv;
        end
```

可以看出标红的部分还没有被修改过来

（4）修正效果

将（3）中标红的 PLV 改为 MAT

8、错误 8：例外 ecode 设置错误

（1）错误现象

```
------------------------------------------------------------
[7295287 ns] Error!!!
    reference: PC = 0x1c0083f0, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00030003
    mycpu    : PC = 0x1c0083f0, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000003
------------------------------------------------------------
```

图 14　错误 8 的错误报告

（2）分析定位过程

根据错误报告给出的 PC 值查看汇编指令：

```
1c0083f0:   0400140c    csrrd   $r12,0x5
```

查看 csr_num 为 0x5 的寄存器，发现是 ESTAT 寄存器。查看波形，发现从 ESTAT 寄存器中读出的值出错，本应是 0x00030003，却变成了 0x00000003，推测之前往 ESTAT 寄存器中写入的值错了，于是在波形中查看 csr_badv_rvalue 信号改变的最后位置，波形如下：

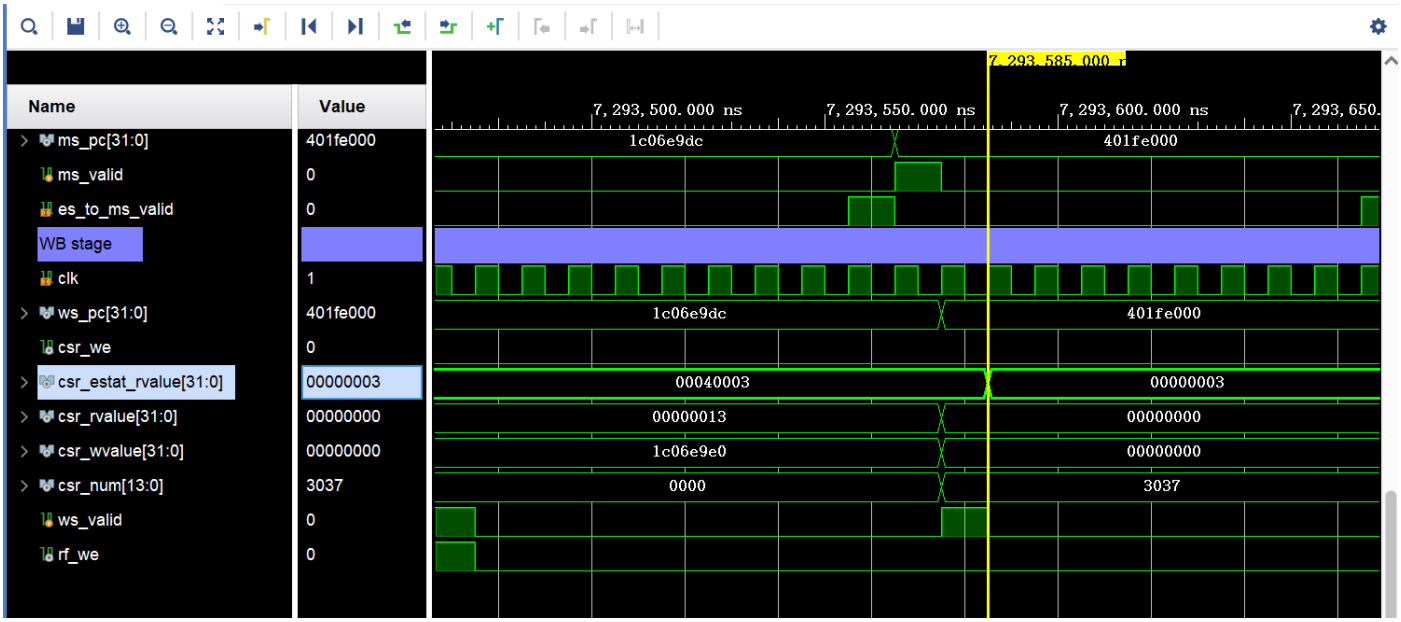

图 15　错误 8 的波形

可以看出此时的 PC 值为 0x401fe000，有些奇怪，也就在这条指令下 ESTAT 寄存器被改写了。于是我们继续向

前查看：

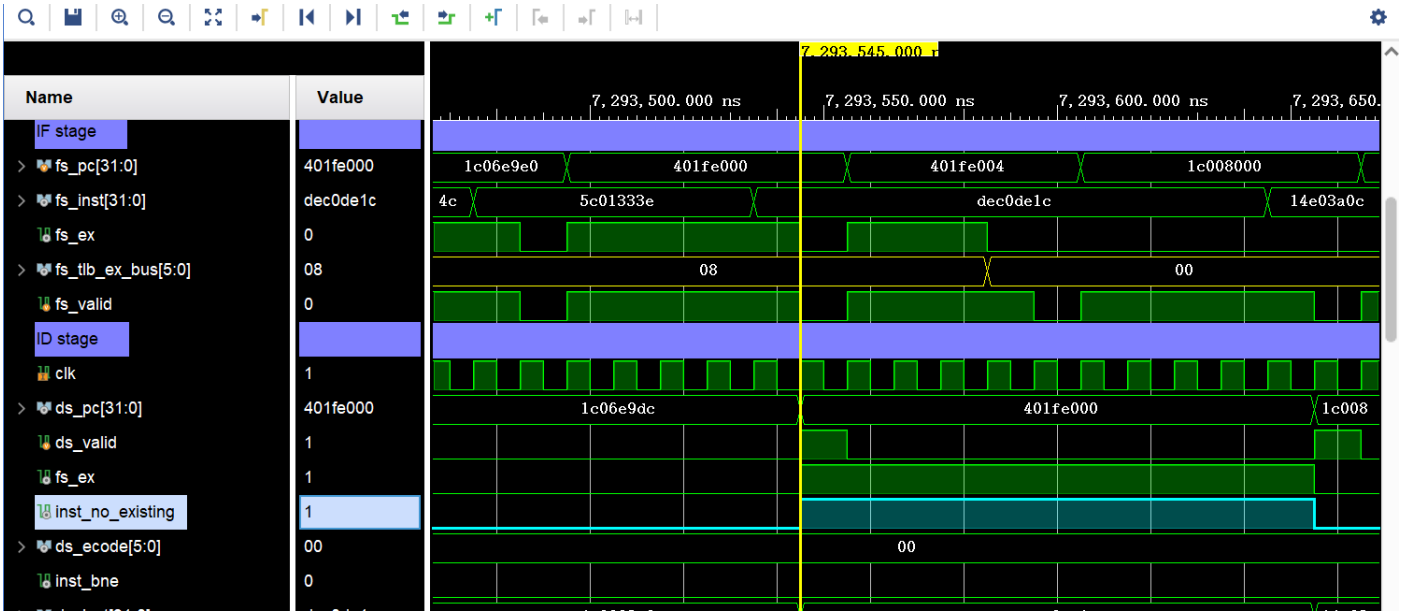

图 16　错误 8 的波形

到 ID 级时，我们发现蓝色的 inst_no_existing 信号被拉高，此时我以为是因为指令不存在例外导致 ESTAT 寄存器被改写，但是去翻看金标准与出错的区别位，也就是 ecode 时，我发现此时出错的的 ecode（0x0）对应的是中断例外，而不是指令不存在例外（0xD），同时金标准又是取值操作页无效例外（0x3），于是继续向上查看 IF 级的例外。

可以看出，黄色的信号线，也就是储存 tlb 相关例外信息的信号值为 0x08，说明取值操作页无效例外有效，又因为例外的优先级是 IF 级＞ID 级>EXE 级，所以此时的例外应该就是取值操作页无效例外，那么问题又来了，为什么 ecode 不是 0x3 呢？查看 ID 级对 ds_ecode 的赋值语句，我们发现了答案。

（3）错误原因

在取指级时只是确定了取指地址错例外，并没有细分 tlb 例外，导致在 ID 级由于优先级原因并没有修改成功 ecode：

```
    assign fs_ecode    = faddr_error ? `ECODE_ADE
                       : `ECODE_UNKNOWN;

    assign ds_ecode      = has_int ? `ECODE_INT
                         : fs_ex ? fs_ecode
                         : fs_tlb_ex_bus[0] ? `ECODE_TLBR
                         : fs_tlb_ex_bus[3] ? `ECODE_PIF
                         : fs_tlb_ex_bus[1] ? `ECODE_PPI
                         : inst_no_existing ? `ECODE_INE
                         : inst_break       ? `ECODE_BRK
```

```
                            : inst_syscall     ? `ECODE_SYS
                            : `ECODE_UNKNOWN;
```

（4）修正效果

在 IF 中就确定有关取指的例外类型，将其改为：

```
    assign fs_ecode       = faddr_error ? `ECODE_ADE
                          : fs_tlb_ex_bus[0] ? `ECODE_TLBR
                          : fs_tlb_ex_bus[3] ? `ECODE_PIF
                          : fs_tlb_ex_bus[1] ? `ECODE_PPI
                          : `ECODE_UNKNOWN;

    assign ds_ecode       = has_int ? `ECODE_INT
                          : fs_ex ? fs_ecode
                          : inst_no_existing ? `ECODE_INE
                          : inst_break       ? `ECODE_BRK
                          : inst_syscall     ? `ECODE_SYS
                          : `ECODE_UNKNOWN;
```

9、错误 9：badv 未考虑取值级例外

（1）错误现象

```
------------------------------------------------------------
[7295987 ns] Error!!!
    reference: PC = 0x1c008428, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x401fe000
    mycpu    : PC = 0x1c008428, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
------------------------------------------------------------
```

图 17　错误 9 的错误报告

（2）分析定位过程

根据错误报告给出的 PC 值查看汇编指令：

```
1c008428:  04001c0c   csrrd  $r12,0x7
```

查看 csr_num 为 0x7 的寄存器，发现是 BADV 寄存器。查看波形，发现从 BADV 寄存器中读出的值出错，本应是 0x401fe000，却变成了 0x00000000，推测之前往 BADV 寄存器中写入的值错了，于是在波形中查 csr_badv_rvalue 信号改变的最后位置，波形如下：

7.4.7 出错虚地址（BADV）

该寄存器用于触发地址错误相关例外时，记录出错的虚地址。此类例外包括:

- TLB 重填例外
- 取指地址错例外（ADEF），此时记录的是该指令的 PC
- load/store 操作地址错例外（ADEM）
- 地址对齐错例外（ALE）
- load 操作页无效例外（PIL）
- store 操作页无效例外（PIS）
- 取指操作页无效例外（PIF）
- 页修改例外（PME）
- 页特权等级不合规例外（PPI）

表 7-9 出错虚地址寄存器定义

位	名字	读写	描述
GRLEN-1:0	VAddr	RW	当触发 TLB 重填例外和地址错误相关例外时，硬件将出错的虚地址记录于此。

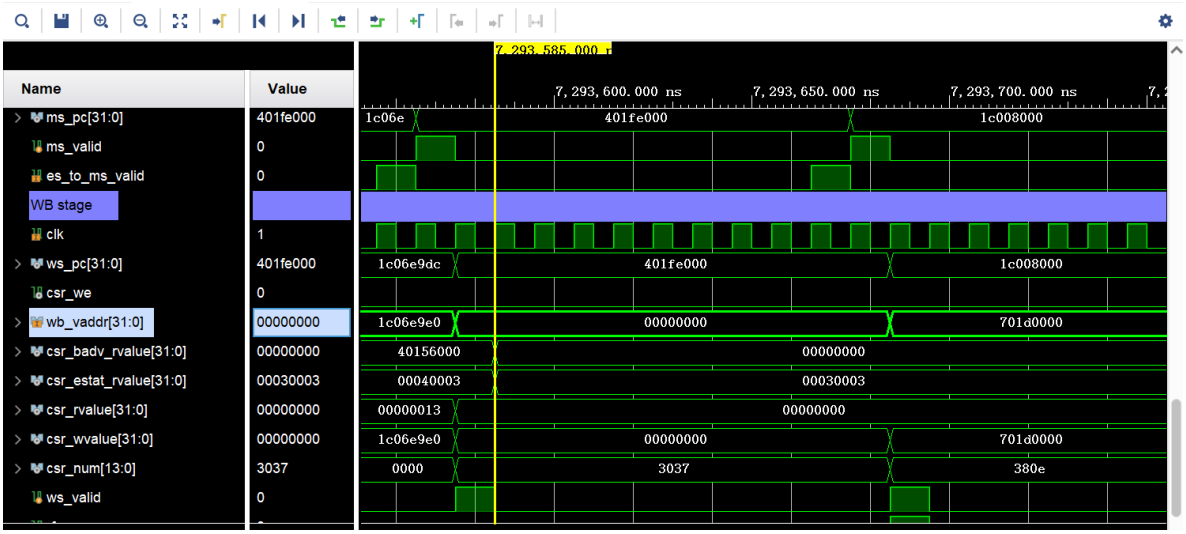

图 18　错误 9 的波形

这时我们惊讶地发现，向 BADV 寄存器写入 0 值的指令就是错误 8 中的不存在指令，这也就可以解释为什么金标准是 0x401fe000，也就是此时的 PC 值。于是我们沿着 BADV 的数据通路向前查看，首先是写入的记录出错地址的信号 wb_vaddr，接着是 EXE 级的 es_vaddr，在这里我们发现了端倪，根据指令手册中对 BADV 寄存器作用的描述：

可以看出在这里我们需要区分指令例外以及访存例外的出错地址，但是对于 es_vaddr 的赋值却并没有考虑。

（3）错误原因

对 es_vaddr 信号的赋值并未考虑取值级例外：

```
    assign es_vaddr = es_alu_result;
```

（4）修正效果

将 IF 级的取指例外信号 fs_ex 传到 EXE 级，对于记录错误地址的信号 es_vaddr 分情况判断。

```
assign es_vaddr         = fs_ex ? es_pc : es_alu_result;
```

四、实验总结

1. 此次实验过程中，由于上手写的比较早，所以遇到了不少讲义上未注意到的或是未提及的错误，卡了一会，好在在 piazza 上提问后，老师很快就给出了回答。

2. 在完成 Lab14 时，由于指令手册有错误，加上我们有一些地方也没有理解到位，导致在调试时改来改去总是改不对。尤其是在调试 INVTLB 指令时，我们一开始并没有明白 TLB 项无效和 TLB 中的页表项无效有什么区别，后来看了体系结构的教材之后才理解，前者和 TLB 某一项的 E 位有关，后者和这一项 TLB 中两个页表项的 V 位相关。

3. 在写 lab15 时饱受信号位宽之苦，大部分时间都是在调位宽设置或是计算错误所导致的 bug，希望以后能重视起来，在写的时候就有意识地注意一下。