实验 13 ～ 15 报告

学号	2019K8009929019	2019K8009929026
姓名	桂庭辉	高梓源
箱子号	44	

1 实验任务

为了抽象物理内存，使得用户进程拥有完整的可用地址空间，现代操作系统提供了基于虚拟内存的存储管理机制。如实现上述机制，传递给 CPU 的指令携带的地址信息将为虚地址而非物理地址（开启虚存时），但 CPU 与物理内存间的交互始终只能基于物理地址，故而需要引入存储管理单元 MMU 进行虚地址到物理地址的翻译工作以及存储管理的部分维护机制。

现代操作系统通常采用页式内存管理，即虚拟内存与物理内存间的映射以页为单位，由页表存储进程地址空间中虚实页的所有对应关系，特定页的虚实映射关系由页表项（Page Table Entry, PTE）记录。页表记录于内存中，但由于其需要被频繁使用，如每次取指均需由虚拟地址的 PC 翻译至物理地址请求物理内存，所以为了减小访问页表的开销，在 CPU 中使用 TLB（Translation Lookaside Buffer）记录部分页表项以加快虚实地址翻译。

本专题实验基于上述思想，从 TLB 模块设计入手，基于 LoongArch 32 位精简版指令集首先完成 TLB 的集成与维护（TLB 相关指令），其后基于上述工作完成虚实地址翻译功能。

2 实验设计

2.1 重要模块设计：TLB

2.1.1 功能描述

TLB 的功能在于在硬件上记录部分页表项以加快页表查询速度，具体需要记录指定项数的页表项 PTE，根据 LoongArch 32 位精简版 TLB 需支持的指令，设计同步写、异步读的读写端口以及用于取指、访存两处地址转换的查找功能。

TLB 模块的接口与讲义一致，此处不再赘述。

2.1.2 细节实现

TLB 的读写逻辑与通用寄存器堆 regfile 的逻辑类似。读操作根据 `r_index` 异步读取 TLB 记录页表项各域即可，写操作在 `we` 拉高时根据 `w_index` 将页表项各域同步更新记录。

查找过程需要将输入的虚页号与 TLB 记录的虚页号相匹配，生成各表项匹配成功与否的 `match0/1` 信号逻辑与讲义一致，具体而言对每个 TLB 表项需检查存在位 `e`、可见性 `g` 或 `asid` 匹配、虚页号相等与否，其中虚页号对于页大小为 4 MB 时比对高 9 位即可，而为 4 KB 时需匹配所有 19 位。

```
generate for (i = 0; i < TLBNUM; i = i + 1) begin
    assign match0[i] = tlb_e[i] && (tlb_g[i] || tlb_asid[i] == s0_asid) &&
```

```
                        // cmp high bits whatever ps is
                        s0_vppn[18:10] == tlb_vppn[i][18:10]            &&
                        // if ps == 4 KB, cmp low bits
                       (s0_vppn[ 9: 0] == tlb_vppn[i][ 9: 0] || tlb_ps[i]);
    assign match1[i] = tlb_e[i] && (tlb_g[i] || tlb_asid[i] == s1_asid) &&
                        s1_vppn[18:10] == tlb_vppn[i][18:10]            &&
                       (s1_vppn[ 9: 0] == tlb_vppn[i][ 9: 0] || tlb_ps[i]);
end
endgenerate
```

是否查找成功信号 found 生成逻辑简单，只需将 match 按位或（等效于 $\neq 0$）即可。由 match 信号获取匹配的 TLB 表项索引 index 可以使用多路选择器实现，但在个人设计中未能编写出兼顾代码可读性与参数化（即选择器路数由参数 TLBNUM 决定）的多路选择器，故而以另一种形式实现该过程，即下文将要讨论的模块 mylog2。

获取到查找索引 s_index 后，由于一个 TLB 表项记录两个物理页，接下来需要判断奇偶页的选择。标记奇偶页的虚地址位为 va[PS]，对于 4 KB 页，其为虚地址的第 12 位，对于 4 MB 页，其为虚地址的第 22 页，即查找端口输入的虚页号第 9 位。根据页大小获取奇偶标志位后即可选出查找结果的物理页。

```
assign s0_ppg_sel = tlb_ps[s0_index] ? s0_vppn[9] : s0_va_bit12;
assign s0_ps      = tlb_ps[s0_index] ? 6'd22 : 6'd12;

assign s0_ppn   = s0_ppg_sel ? tlb_ppn1[s0_index] : tlb_ppn0[s0_index];
assign s0_plv   = s0_ppg_sel ? tlb_plv1[s0_index] : tlb_plv0[s0_index];
assign s0_mat   = s0_ppg_sel ? tlb_mat1[s0_index] : tlb_mat0[s0_index];
assign s0_d     = s0_ppg_sel ? tlb_d1  [s0_index] : tlb_d0  [s0_index];
assign s0_v     = s0_ppg_sel ? tlb_v1  [s0_index] : tlb_v0  [s0_index];
```

接下来考虑 INVTLB 指令功能的实现，如讲义所言，将其查找逻辑拆分为四个子匹配逻辑：

```
generate for (i = 0; i < TLBNUM; i = i + 1) begin
   assign inv_match[0][i] = ~tlb_g[i];
   assign inv_match[1][i] =  tlb_g[i];
   assign inv_match[2][i] = s1_asid == tlb_asid[i];
   assign inv_match[3][i] = s1_vppn[18:10] == tlb_vppn[i][18:10]            &&
                            (s1_vppn[ 9: 0] == tlb_vppn[i][ 9: 0] || tlb_ps[i]);
end
endgenerate
```

由子匹配结果容易获得一套各 invtlb_op 对应的掩码，INVTLB 指令的无效实现只需要将 op 对应的掩码中为 1 处的 tlb_e 域抹零即可。[①]

```
assign inv_op_mask[0] = 16'hffff;
assign inv_op_mask[1] = 16'hffff;
assign inv_op_mask[2] = inv_match[1];
assign inv_op_mask[3] = inv_match[0];
```

[①]INVTLB 功能在 lab13 中不做验证，故而在 lab13 的代码中此处 [0] 置为全 0。

```
assign inv_op_mask[4] = inv_match[0] & inv_match[2];
assign inv_op_mask[5] = inv_match[0] & inv_match[2] & inv_match[3];
assign inv_op_mask[6] = (inv_match[0] | inv_match[2]) & inv_match[3];
// set rest to 16'h0000

always @ (posedge clk) begin
    if (we) begin
        // write logic
    end else if (invtlb_valid) begin
        tlb_e <= ~inv_op_mask[invtlb_op] & tlb_e;
    end
end
```

2.2 重要模块设计：mylog2

2.2.1 功能描述与设计实现

设计本意是检测输入中的 1，返回其位置。思路可简单概括为折半查找。

折半查找的每一步检索结果与输出结果自高到低每一位一一对应。以 32 位输入为例，输出应有 5 位，折半查找第一次判断 1 落在高 16 位还是低 16 位，若高则将输出最高位置 1，依此类推，每次折半判断的高 (1) 低 (0) 与作为结果的索引一一对应。判断 1 落在哪一半的实现通过判断高半部分是否有 1（是否不等于 0），由此在输入有多个 1 时进行了高位优先，最终获取的结果为最高 1 的索引，在数值上等同于运算 $\log_2$。

在具体实现上，对此前提到的高半部分的选取通过语法 `+:` 实现，由此需要获取到每次判断的基址。以 32 位为例，首次基址应为 16，判断输入的 `[16+:16]` 部分是否有 1。类似的对于任意[2]位宽 `WIDTH` 的输入，其索引位宽 `IDX_WIDTH` 为 `WIDTH` 对 2 取对数，首基址只需将 `IDX_WIDTH` -1 位置 1，其他位置 0 即可（即对 `WIDTH` 数值取半）。

```
localparam IDX_WIDTH = $clog2(WIDTH);

assign base[IDX_WIDTH-1] = {1'b1, {IDX_WIDTH-1{1'b0}}};
```

对于折半查找每层需要做的工作，(1) 根据基址判断高半部分是否有 1，生成结果 `res` 的相应位；(2) 计算下一层应使用的基址。前者的实现较为简单。后者归纳下来即高半部分有 1 则当前层所用基址加上 (1) 中检测宽度的一半，否则减去。此处的加减法具有一定的特殊性，记当前层对应位索引为 `i`，次层对应位为 `i-1`。加数或减数定为在 `i-1` 处为 1，其他位为 0。而被加/减数的 `i` 位以后均为 0，所以计算结果中 `i-1` 处必定为 1，根据这个结论可以发现不会出现连续借位的情况，即 `i` 位是上一层的 `i-1` 位，必定为 1 可以被借位，若发生加法则该位保持 1，若发生减法则被借位置 0。由此可以注意到 `base[i]` 与 `res[i]` 的值是一致的。

```
generate
    for (i = IDX_WIDTH-1; i > 0; i = i - 1) begin
        assign res[i] = src[base[i]+:({{IDX_WIDTH-1{1'b0}}, 1'b1} << i)] != 0;
```

[2]实际应用上最好指定位宽 `WIDTH` 为 2 的次幂，非 2 次幂时 `+:` 的越界问题我并未检验与解决。

```
            for (j = 0; j < IDX_WIDTH; j = j + 1) begin
                if (i == j) begin
                    assign base[i - 1][j] = res[i];
                end else if (i - 1 == j) begin
                    assign base[i - 1][j] = 1'b1;
                end else begin
                    assign base[i - 1][j] = base[i][j];
                end
            end
        end
    endgenerate
```

结果的末位与根据最后一次计算出的基址 `base[0]` 索引输入的结果一致。

```
assign res[0] = src[base[0]];
```

2.2.2 接口定义

表 1: `mylog2` 模块接口定义

名称	方向	位宽	描述
`src`	IN	WIDTH	$\texttt{res} = \log_2 \texttt{src}$
`res`	OUT	$\log_2$ WIDTH	

2.3 TLB 集成与 TLB 指令的添加

2.3.1 TLB 的集成

TLB 设计中预留两套查找端口，分别供取指（pre-IF 级）和访存（EXE 级）使用，读写端口分别供 TLB 指令 tlbrd、tlbwr/tlbfill 使用，如讲义中讨论的指令实现细节，为尽量减少围绕 TLB 和相关 CSR 寄存器的数据相关，读写过程均在写回 (WB) 级完成读写动作。由此可以注意到 TLB 模块与 CSR 模块类似，需要与多个流水级进行交互，故而同样将其例化于流水级之外而非某个特定流水级之内，从而便于其与各流水级交互。

在仅进行集成而不考虑虚实地址翻译功能的实现时，用于取指的 s0 端口暂无需考虑，用于访存的 s1 端口仅需考虑复用其的 tlbsrch、invtlb 指令功能支持即可，读写功能中读写使能来自流水级，读地址来自 CSR，写地址除 tlbwr 使用 TLBIDX 内容外，还有 tlbfill 使用随机数的情形需要考虑。

2.3.2 TLB 相关 CSR 寄存器

需要实现的五条 TLB 相关指令中，除了 invtlb 指令应用了部分来自通用寄存器的数据外，其他四条指令均是指示 TLB 与相关 CSR 寄存器间的交互。CSR 寄存器中 TLBEHI、TLBELO0/1 记录一个 TLB 表项的主要信息，TLB 表项中其他内容，如 asid 域、ps 域、e 位由 CSR.ASID 寄存器、CSR.TLBIDX 寄存器记录。读写过程即将上述 CSR 寄存器内容与 TLB 表项进行读写（“读/写”动作均相对 TLB 而言）。

tlbsrch 则应用其中 asid、vppn 信息进行查找，仅记录查找成功与否、查找结果的 e 位与查找结果的 TLB 索引。

具体实现上引入 tlbsrch、tlbrd、tlbwr、tlbfill 指令的相应使能（invtlb 指令不维护 CSR 寄存器，仅需将其使能传递给 TLB 模块即可），根据指令手册中对相关寄存器各域的描述，在相应的 `always` 块中加上对应分支即可。此处逻辑与指令手册一致，不再给出具体代码描述。

2.3.3 TLB 指令的添加

如前所述，除 invtlb 外的四条 tlb 相关指令均是指示 TLB 与相关 CSR 寄存器交互，并不需要应用通用寄存器与其他存储单元，在数据通路的构造上较为简单，只需将其译码结果作为前文提到的对 CSR 使能传递到写回级发送给 CSR 即可。其中 tlbsrch 在执行级发出查找请求后需将收到的查找结果传递到写回级一并发给 CSR。

invtlb 指令使用的数据来自指令码与通用寄存器，由于其复用执行级的 s1 端口，该处已有 `rj`、`rk` 寄存器数据，直接连线使用即可。其操作类型单独自译码级传递至执行级。

本次实验引入 TLB 与一众 CSR 寄存器作为新的存储结构，自然需要考量其数据相关问题。这些数据相关问题可以分为两类：(1) TLB 与 CSR.CRMD、CSR.DMW0/1 等影响取指的存储结构；(2) 其他 CSR 寄存器。

前者影响取指过程，由于流水线设计不可避免地在获取到指令类型时已经取到一条新指令，故而在指令修改（即写）上述结构时，引入重取机制将该写指令后续流水线内指令全部刷掉重新从其后一条指令处开始取指。具体代码实现上，在译码级即可判断出是否需要进行重取标记：

```
assign ds_refetch_flg = inst_tlbfill || inst_tlbwr || inst_tlbrd || inst_invtlb ||
                        ds_csr_we && (ds_csr_wnum == `CSR_CRMD || ds_csr_wnum == `CSR_ASID ||
                                      ds_csr_wnum == `CSR_DMW0 || ds_csr_wnum == `CSR_DMW1);
```

当携带着重取标志的指令到达写回级后复用发异常的数据通路清空流水线，只是该“异常”不会修改任何 CSR 寄存器，而且其入口为当前 PC + 4。

对于其他 CSR 寄存器引起的数据相关采用此前处理 CSR 相关的阻塞思路一致。由于我们的设计在 ID 级读 CSR 寄存器，故而新引入的读写关系分为三类：(1) tlbrd 指令修改 CSR 寄存器，后续 CSR 指令读 TLB 相关寄存器；(2) tlbsrch 修改 CSR.TLBIDX，后续 CSR 指令读该寄存器；(3) tlbsrch 指令在执行级使用 CSR.ASID、CSR.TLBEHI 寄存器内容查找 TLB，后续流水级中有修改该寄存器的指令。其中 (1) 已经被此前的重取机制解决，(2) 采用与此前完全一致的阻塞机制，(3) 同样采用阻塞机制，但复用原 ID 级的阻塞逻辑与其他情况略有不同。对于其他的阻塞来源，读者通常在 ID 级获取到读数据，故而对于其后每一级中写者均需进行判断，但 tlbsrch 并不需要在 ID 级获取到读数据，其在 EXE 级才会使用 CSR.ASID、CSR.TLBEHI 数据，故而不用判断写回级的写者，当 tlbsrch 自 ID 级流向 EXE 级时写回级对这些 CSR 寄存器的写动作已经完成。

```
assign csr_blk = ds_csr_re & (es_csr_blk | ms_csr_blk | ws_csr_blk);
assign es_csr_blk = es_csr_we    &&  csr_rnum == es_csr_wnum  ||
                    es_eret      &&  csr_rnum == `CSR_CRMD    ||
                    es_tlbsrch   &&  csr_rnum == `CSR_TLBIDX  ||
                    inst_tlbsrch && (es_csr_wnum == `CSR_ASID || es_csr_wnum == `CSR_TLBEHI);
assign ms_csr_blk = ms_csr_we    &&  csr_rnum == ms_csr_wnum  ||
```

```
                    ms_eret      &&  csr_rnum == `CSR_CRMD    ||
                    ms_tlbsrch   &&  csr_rnum == `CSR_TLBIDX  ||
                    inst_tlbsrch && (ms_csr_wnum == `CSR_ASID || ms_csr_wnum == `CSR_TLBEHI);
assign ws_csr_blk = ws_csr_we    &&  csr_rnum == ws_csr_wnum  ||
                    ws_eret      &&  csr_rnum == `CSR_CRMD    ||
                    ws_tlbsrch   &&  csr_rnum == `CSR_TLBIDX;
```

2.4 虚实地址翻译

LoongArch 32 位精简版支持的地址翻译模式与其相应逻辑通过指令手册可以获得较为清晰的梳理，此处不再赘述，以下仅以 pre-IF 级为例给出主要代码实现：

```
    assign inst_sram_addr = pfs_da       ? nextpc         :
                            pfs_dmw0_hit ? pfs_dmw0_paddr :
                            pfs_dmw1_hit ? pfs_dmw1_paddr :
                                           pfs_tlb_paddr;
    assign pfs_da = csr_crmd_da & ~csr_crmd_pg;
    assign pfs_dmw0_hit = nextpc[31:29] == csr_dmw0_vseg &&
                          (csr_crmd_plv == 2'd0 && csr_dmw0_plv0 ||
                           csr_crmd_plv == 2'd3 && csr_dmw0_plv3);
    assign pfs_dmw1_hit = nextpc[31:29] == csr_dmw1_vseg &&
                          (csr_crmd_plv == 2'd0 && csr_dmw1_plv0 ||
                           csr_crmd_plv == 2'd3 && csr_dmw1_plv3);
    assign pfs_dmw0_paddr = {csr_dmw0_pseg, nextpc[28:0]};
    assign pfs_dmw1_paddr = {csr_dmw1_pseg, nextpc[28:0]};
    assign pfs_tlb_paddr = s0_ps == 6'd22 ? {s0_ppn[19:10], nextpc[21:0]} :
                                            {s0_ppn, nextpc[11:0]};
```

此处简化了直接地址翻译模式与映射地址翻译模式的判断逻辑，还简化了页大小对 TLB 结果的拼接影响。

2.5 TLB 相关例外

TLB 相关的异常包括 TLB 重填（TLBR）、Load/Store/取指操作页无效（PIL/PIS/PIF）、页修改（PME）、页特权等级不合规（PPE[3]），此外由于完善了访存地址（包括取指地址）的特权等级区分，故而 ADEF 与 ADEM 异常的逻辑也可以进行完善。

异常优先级的考虑需要区分 TLBR、PPE 指令发生在取指还是访存[4]，同时可以将 TLB 相关的五条例外间的互斥在优先级判断中处理，减小电路开销。

```
    // pre_if_stage.v
    assign pfs_exc_flgs[`EXC_FLG_ADEF] = (|nextpc[1:0]) | (nextpc[31] & csr_crmd_plv == 2'd3);
    assign pfs_exc_flgs[`EXC_FLG_TLBR_F] = pfs_tlb_trans & ~s0_found;
    assign pfs_exc_flgs[`EXC_FLG_PIF]  = pfs_tlb_trans & ~s0_v;
    assign pfs_exc_flgs[`EXC_FLG_PPE_F] = pfs_tlb_trans & (csr_crmd_plv > s0_plv);
```

[3]对于该例外的简称，指令手册中部分（CSR.ESTAT 的 Ecode 域）为 PPI，部分（5.4.4 节基于 TLB 的虚实地址转换过程小节）为 PPE，代码实现与本报告中的命名均采用 PPE。

[4]当然测试集中好像没测出来这些 ()

```
// exe_stage.v
assign es_exc_flgs[`EXC_FLG_ADEM] = es_inst_ls & es_alu_result[31] & (csr_crmd_plv == 2'd3);
assign es_exc_flgs[`EXC_FLG_TLBR_M] = es_inst_ls & es_tlb_trans & ~s1_found;
assign es_exc_flgs[`EXC_FLG_PIL]  = (|es_load_op) & es_tlb_trans & ~s1_v;
assign es_exc_flgs[`EXC_FLG_PIS]  = es_mem_we & es_tlb_trans & ~s1_v;
assign es_exc_flgs[`EXC_FLG_PME]  = es_mem_we & es_tlb_trans & ~s1_d;
assign es_exc_flgs[`EXC_FLG_PPE_M] = es_inst_ls & es_tlb_trans & (csr_crmd_plv > s1_plv);

// wb_stage.v
assign wb_exc      = (|ws_exc_flgs) & ws_valid;
assign wb_ecode    = ws_exc_flgs[`EXC_FLG_INT ] ? `ECODE_INT :
                     ws_exc_flgs[`EXC_FLG_ADEF] ? `ECODE_ADE :
                     ws_exc_flgs[`EXC_FLG_TLBR_F] ? `ECODE_TLBR :
                     ws_exc_flgs[`EXC_FLG_PIF ] ? `ECODE_PIF :
                     ws_exc_flgs[`EXC_FLG_PPE_F] ? `ECODE_PPE :
                     ws_exc_flgs[`EXC_FLG_INE ] ? `ECODE_INE :
                     ws_exc_flgs[`EXC_FLG_SYS ] ? `ECODE_SYS :
                     ws_exc_flgs[`EXC_FLG_BRK ] ? `ECODE_BRK :
                     ws_exc_flgs[`EXC_FLG_ALE ] ? `ECODE_ALE :
                     ws_exc_flgs[`EXC_FLG_ADEM] ? `ECODE_ADE :
                     ws_exc_flgs[`EXC_FLG_TLBR_M] ? `ECODE_TLBR :
                     ws_exc_flgs[`EXC_FLG_PIL ] ? `ECODE_PIL :
                     ws_exc_flgs[`EXC_FLG_PIS ] ? `ECODE_PIS :
                     ws_exc_flgs[`EXC_FLG_PPE_M] ? `ECODE_PPE :
                     ws_exc_flgs[`EXC_FLG_PME ] ? `ECODE_PME : 6'h00;
assign wb_esubcode = {9{ws_exc_flgs[`EXC_FLG_ADEF]}} & `ESUBCODE_ADEF |
                     {9{ws_exc_flgs[`EXC_FLG_ADEM]}} & `ESUBCODE_ADEM;
```

此外在 LoongArch 32 位精简版中 TLB 重填例外入口与其他例外入口不同，由 CSR.TLBRENTRY 指定，需要对其进行选择。

3 实验过程

3.1 实验流水账

2021.10.28 10:00 ~ 2021.10.28 10:30 ： 根据讲义内容与 gitee 仓库 tlb_verify[5]完成 TLB 初步设计，由于验证环境问题未完全验证设计。

2021.11.18 16:00 ~ 2021.11.18 16:45 ： 编写设计 `mylog2` 模块，根据正式实验环境通过 TLB 功能验证。

2021.11.29 00:00 ~ 2021.11.29 02:00 ： 编写 lab14 中新增 CSR 寄存器部分。

2021.11.29 08:00 ~ 2021.11.29 11:30 ： 编写 lab14 中新增指令与集成 TLB 部分。

2021.11.29 15:20 ~ 2021.11.29 16:20 ： lab14 debug 完成。

[5]https://gitee.com/ucas-ca-edu-lab/tlb_verify

2021.12.02 09:10 ~ 2021.12.02 15:10 ： 完成 lab15。

2021.12.05 18:00 ~ 2021.12.05 21:30 ： 完成本实验报告撰写。

3.2 错误记录

本次实验报告不再讨论笔误与简单逻辑错误。

3.2.1 错误 1：重取入口与异常入口的优先级

3.2.1.1 错误现象

`invtlb_op` 测试点中未按理进入异常入口。

3.2.1.2 分析定位过程

`invtlb_op` 测试点中测试 `op` 不合法时的 invtlb 指令运行情况，invtlb 在 `op` 域不合法时应发指令不存在例外（INE），该异常被正确检测，但刷新流水线后的入口为重取入口而非异常入口。即如下代码中的优先级错误：

```
assign wb_flush_target = wb_refetch ? wb_pc + 32'd4 :
                         wb_exc     ? exc_entry     :
                                      exc_retaddr;
```

3.2.1.3 错误原因

编写代码时未考虑 invtlb 指令同时重取与发 INE 异常的情形，优先级直接乱写。重新考虑刷新流水线的三种情形间优先级，重取为指令正常执行行为，而异常为“错误”行为，改错（处理异常）应优先于正常执行的重取，异常返回的 `ertn` 本身不引起重取，即异常返回与重取不会同时发生。

3.2.1.4 修正效果

调整优先级关系如下：

```
assign wb_flush_target = wb_exc     ? exc_entry     :
                         wb_refetch ? wb_pc + 32'd4 :
                                      exc_retaddr;
```

完善了刷新流水线相关逻辑。

3.2.2 错误 2：CSR.BADV 与 CSR.TLBEHI 维护错误

3.2.2.1 错误现象

lab15 新引入的异常测试点中读 CSR.BADV 或 CSR.TLBEHI 寄存器错误。

3.2.2.2 分析定位过程与错误原因

重新查阅指令手册中有关这两个寄存器的维护逻辑，注意到 lab9、lab14 中对其实现的逻辑并非完善，对 lab15 新增的异常未做支持。

3.2.2.3 错误修订

在写回级根据异常类型判断 `badv` 的来源为 PC 或访存地址，在 CSR 模块中修改 CSR.BADV、CSR.TLBEHI 维护逻辑如下：

```
always @ (posedge clk) begin
    if(rst)begin
        csr_badv_vaddr <= 32'b0;
    end
    if (wb_exc) begin
        if (badv_is_pc) begin
            csr_badv_vaddr <= wb_pc;
        end else if (badv_is_mem) begin
            csr_badv_vaddr <= wb_badvaddr;
        end
    end
end

always @ (posedge clk) begin
    if (rst) begin
        csr_tlbehi_vppn <= 19'b0;
    end else if (tlbrd_we && r_tlb_e) begin
        csr_tlbehi_vppn <= r_tlb_vppn;
    end else if (wb_exc) begin
        if (badv_is_pc && wb_ecode != `ECODE_ADE) begin
            csr_tlbehi_vppn <= wb_pc[31:13];
        end else if (badv_is_mem && wb_ecode != `ECODE_ALE && wb_ecode != `ECODE_ADE) begin
            csr_tlbehi_vppn <= wb_badvaddr[31:13];
        end
    end else if (csr_we && csr_wnum == `CSR_TLBEHI) begin
        csr_tlbehi_vppn <= csr_wmask[`CSR_TLBEHI_VPPN] & csr_wval[`CSR_TLBEHI_VPPN] |
                           ~csr_wmask[`CSR_TLBEHI_VPPN] & csr_tlbehi_vppn;
    end
end
```

3.2.2.4 归纳总结

该错误来自于迭代开发过程中未记录不完善的实现，以跨实验的视角来合理注释 TODO 还是有必要的。

4 实验总结

本专题实验整体难度不高，按步就班根据指令手册与实验讲义进行实现即可，具体实现过程中还是要多关注实验细节与迭代开发过程中隐藏的问题，不然就容易出现上述错误。