

# LAB8 报告

学号：2021K8009929010

姓名：贾城昊

学号：2021K8009929016

姓名：李金明

学号：2021K8009929007

姓名：牛浩宇

箱子号：13

## 一、实验任务

本实验主要实现了对 TLB 模块的结构设计和例外支持，并将其加入到流水线 CPU 中，从而为操作系统的存储管理提供硬件支持。

Exp17 要求实现对单个 TLB 模块的结构设计，使其能够完成最基本的读、写和查找功能，并通过实验提供的测试程序。

Exp18 要求在 CPU 中增添与 TLB 相关的控制寄存器，包括 TLBIDX、TLBEHI、TLBELO0、TLBELO1、ASID、TLBREENTRY；添加对 TLB 相关指令的支持，包括 TLBSRCH、TLBRD、TLBWR、TLBFILL、INVTLB；将 TLB 模块集成到 CPU 内，实现从 CPU 内对 TLB 进行基本的读、写和查找操作。

Exp19 要求在 CPU 中添加对 TLB 例外的支持，包括 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；在 CPU 中添加直接映射配置窗口控制寄存器 DMW0 和 DMW1，并利用 TLB 查找功能实现 CPU 访存过程中的虚实地址转换。

## 二、实验设计

### (一) 总体设计思路

#### 1、Exp17

Exp17 对单个 TLB 模块的设计可根据所需实现的功能分为读、写和查找三部分，下面将从读写功能和查找功能两方面阐述总体设计思路。

#### (1) 读写功能设计

TLB 模块的读写方式与寄存器堆的读写方式十分类似。读 TLB 的方式是根据传入的 index 将对应表项各个域的数据通过组合逻辑读出；写 TLB 的方式是根据传入的 index 和写使能信号，在下一拍将要写的各个域的数据写入对应表项相应位置。但在读写过程中需要注意的是，由于龙芯架构 32 位精简版只支持 4KB 和 4MB 两种页大小，因此尽管 TLB 表项中的 PS 域是 6bit，但其只有 0 和 1 两种取值：0 表示页大小为 4KB（两个物理页大小均为

4KB），1 表示页大小为 4MB（两个物理页大小均为 2MB）。而 TLB 模块输入和输出中的 PS 端口标记了页的真实大小，取值分别是 6'd21 和 6'd12，因此在对页表项进行读写操作时需要将两者进行转换。以读 TLB 为例：

```
// read
assign r_e    = tlb_e    [r_index];
assign r_vppn = tlb_vppn [r_index];
assign r_ps   = tlb_ps4MB[r_index] ? 6'd21 : 6'd12;
assign r_asid = tlb_asid [r_index];
assign r_g    = tlb_g    [r_index];
```

图 1 PS 字段的含义及其赋值逻辑

## (2) 查找功能设计

TLB 模块提供了两个通道用于查找，一个通道对应取指阶段的查找，另一个通道对应访存阶段的查找。查找的流程是并行化的，即同时对 16 个页表项的奇偶相邻页表的物理转换信息进行比较和查找，并用两个 16 位宽的 match 信号表示查找结果。在代码实现上，可以使用 verilog 语言中的 for 循环更方便地实现对页表项的同时匹配和查找过程，匹配和查找的逻辑是：对于每一个页表项，分别要比较和检查以下几点：一个是查看比较部分的存在位 E，如果为 1 则表示页表项存在，若为 0 表示该页表项为空，查找失败；然后还需要查看比较部分的全局标志位 G 和地址空间表示位 ASID，判断该页表项中的数据是否属于本进程，若属于不属于本进程则查找失败；除此之外，还需要查看比较部分的页大小 PS 位和虚双页号 VPPN 位，如果 PS 位为 1，表示页大小为 4MB（两个物理页大小均为 2MB），此时只需要比较 VPPN 的高 10 位是否与 s\_vppn 的高 10 位相等，若相等则查找成功；如果 PS 位为 0，表示页大小为 4KB（两个物理页大小均为 4KB），此时需要比较 VPPN 和 s\_vppn 的全部位数。如果以上全部满足则查找成功。

```
genvar i;
generate
  for(i=0; i<TLBNUM; i=i+1) begin
    assign match0[i] = (s0_vppn[18:9] == tlb_vppn[i][18:9])
      && (tlb_ps4MB[i] || s0_vppn[8:0] == tlb_vppn[i][8:0])
      && ((s0_asid == tlb_asid[i]) || tlb_g[i])
      && tlb_e[i];
    assign match1[i] = (s1_vppn[18:9] == tlb_vppn[i][18:9])
      && (tlb_ps4MB[i] || s1_vppn[8:0] == tlb_vppn[i][8:0])
      && ((s1_asid == tlb_asid[i]) || tlb_g[i])
      && tlb_e[i];
  end
endgenerate
```

图 2 TLB 的查找判断逻辑

通过上述查找过程可以获得两个 match 信号，match0 不为全 0 说明 0 号通道查找到了匹配的页表项（0 号通道用于取指阶段的查找），match1 不为全 0 说明 1 号通道查找到了匹配的页表项（1 号通道用于访存阶段的查找）。

---

match0 或 match1 中哪一位为 1 则说明命中了对应位的页表项。然后再根据输入的 s\_va\_bit12 或者 s\_vppn 决定使用匹配到的 TLB 表项的偶页还是奇页的查找结果。查找的代码实现如下：

```
assign s0_found = (|match0);
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 s0_whichpage = (tlb_ps4MB[s0_index])? s0_vppn[8]: s0_va_bit12;
assign s0_ps      = (tlb_ps4MB[s0_index])? 6'd21 : 6'd12;
assign s0_ppn     = (s0_whichpage) ? tlb_ppn1[s0_index] : tlb_ppn0[s0_index];
assign s0_plv     = (s0_whichpage) ? tlb_plv1[s0_index] : tlb_plv0[s0_index];
assign s0_mat     = (s0_whichpage) ? tlb_mat1[s0_index] : tlb_mat0[s0_index];
assign s0_d       = (s0_whichpage) ? tlb_d1 [s0_index] : tlb_d0 [s0_index];
assign s0_v       = (s0_whichpage) ? tlb_v1 [s0_index] : tlb_v0 [s0_index];
```

图 3 TLB 的查找结果的输出

## 2、Exp18

本实验将 exp17 实现的 TLB 实例化，并集成到 cpu 中。为了实现对 TLB 的操作，本实验增加了 TLBSRCH、TLBRD、TLBWR、TLBFILL、INVTLB 这五条指令。此外，还需要在 csr 模块内增加 TLB 有关的寄存器 TLBIDX、TLBEHI、TLBELO0、TLBELO1、ASID、TLBREENTRY，并将 TLB 的一部分端口与 csr 相连。如果沿用之前的设计，将 csr 模块放在 wb 级中，会使得 wb 级增加很多不必要的接口。为此，我们将 csr 移出 wb 级，成为独立的模块。

### (1) TLB 维护指令

在本实验新增的五条指令中，TLBSRCH、TLBRD、TLBWR、TLBFILL 都会在 TLB 和 CSR 之间进行数据的传递。TLBSRCH 在 EXE 级根据 CSR 中的寄存器查询 TLB，并将查询结果从 EXE 级一路传到 WB 级，在 WB 级将它写入 csr。这种安排使得它可以在 EXE 级复用访存的 TLB 查找通路，在 WB 级复用之前将数据写入 csr 的通路。TLBRD、TLBWR、TLBFILL 均在 WB 级进行读写。csr 寄存器中的相关数据通过端口直接传入 WB 级，cpu 在 WB 级根据这些数据生成读写需要的数据，并和相关控制信号一起通过端口传给 TLB。

不同于以上四条指令，INVTLB 并不负责在 TLB 和 CSR 之间传递数据，它的数据来源也并非 csr 寄存器，而是通用寄存器。它会根据存放在寄存器中的 ASID 和 VA，按照 op 对 TLB 进行无效操作。它也需要对 TLB 进行查找，因此我们让它和 TLBSRCH 使用相同的位于 EXE 级的数据通路进行查找。

### (2) 数据相关

在增加 TLB 后，指令在取指和访存时都有可能查找 TLB 用于虚实地址转换，而 TLBWR、TLBFILL、INVTLB 都会更新 TLB 的内容，二者之间形成围绕 TLB 的写后读相关。此外，所有指令在取指和访存都可能读取 CSR.ASID 来查找 TLB，而 TLBRD 会更新该寄存器，二者之间形成围绕 CSR 的写后读相关。为此，考虑到上述情况出现的频率极低，使用前递或者阻塞的操作，对性能提升很小但使流水线逻辑变的更加复杂，所以处于均衡

的考虑，对于 TLBWR、TLBFILL、TLBRD、INVTLB 以及更改 ASID 或者 CRMD 寄存器的 csr 写指令，本组成员的设计是刷新流水级并重新取指；而刷新流水线的操作可以直接复用在之前实验的 wb\_flush 信号，在 WB 流水级发出，刷掉前面所有流水级，并把 IF 的下一个取值 PC 改为此时 WB 流水级的 PC+4。

```

assign wb_tlbwr_valid      = wb_inst_tlbwr & wb_valid;
assign wb_tlbfill_valid    = wb_inst_tlbfill & wb_valid;
assign wb_tlbrd_valid      = wb_inst_tlbrd & wb_valid;
assign wb_invtlb_valid     = wb_inst_invtlb & wb_valid;
assign wb_csr_tlbwr        = ((wb_csr_num == `CSR_ASID || wb_csr_num == `CSR_CRMD)
                             && wb_csr_we) && wb_valid;

assign wb_tlb_refetch      = wb_tlbwr_valid | wb_tlbfill_valid | wb_tlbrd_valid | wb_invtlb_valid | wb_csr_tlbwr;

assign wb_flush = wb_ertn_flush_valid | wb_excep_valid | wb_tlb_refetch;

```

图 4 WB 流水级对于 TLB 数据相关的处理

除此之外，还需要考虑一种写后读相关：当 TLBSRCH 位于 EXE 级时，它会用 ASID 或 TLBEHI 中的内容查询 TLB。此时如果在 MEM 流水级或者 WB 流水级恰好有条修改 ASID 或 TLBEHI 的指令，或是 TLBRD 指令，就会引发围绕 csr 的数据相关。在这种情况下需要将 TLBSRCH 阻塞在 EXE 级，等前面的指令写入完成后再允许其流入下一级。

```

stage control signal
assign ex_ready_go      = (data_sram_req & data_sram_addr_ok) |
                           (ex_res_from_div & ex_div_complete) |
                           (ex_inst_tlbsrch & ~(mem_csr_tlbrd | wb_csr_tlbrd)) |
                           ~(data_sram_req | ex_res_from_div | ex_inst_tlbsrch);

```

图 5 EXE 流水级的阻塞逻辑

### 3、Exp19

对 exp19 设计思路主要从虚实地址转换的过程和 TLB 例外处理两方面来阐述。

#### (1) 虚实地址转换过程

本实验需要支持的虚实地址翻译模式有两种：直接地址翻译模式和映射地址翻译模式。控制寄存器 CRMD 中的 DA 域与 PG 域决定了采用何种地址翻译模式：当 DA 域为 1，PG 域为 0 时，MMU 处于直接地址翻译模式，在这种映射模式下，物理地址直接等于虚拟地址；当 DA 域为 1，PG 域为 0 时，MMU 处于映射地址翻译模式，此时则先查看配置窗口寄存器 DMW0 和 DMW1，如果能命中且当前特权等级在该配置窗口中被允许，则采用直接映射模式，物理地址等于虚地址低 29 位拼接上被命中的配置窗口寄存器的高 3 位；如果两个配置窗口都不命中或特权等级不合规，则采用页表映射模式，需要根据虚地址的 VPPN 和当前的地址空间标识符 ASID 查找 TLB，并根据查找 TLB 返回的 PPN 和虚地址的 offset 得出物理地址。通过多路选择器并根据当前的地址翻译模式可以得到最终的物理地址，然后将它作为 inst\_sram\_addr 或 data\_sram\_addr 传递给转接桥，进而传递到 AXI 总线进行地址访问。

```
//physical addr  
assign pa      = direct_trans ? va  
      : dmw0_hit    ? dmw_pa0  
      : dmw1_hit    ? dmw_pa1  
      : tlb_pa;
```

图 6 虚拟地址到物理地址的转换

由于 IF 和 EX 流水级均需要进行虚拟地址到物理地址的转换，且两者的逻辑相似，本组成员单独编译一个 MMU 模块，完成虚拟地址到物理地址的映射，同时返回方便判断 TLB 例外的相关信号。然后在 mycpu\_top.v 中分别例化两个 MMU 模块，连接 IF 流水级，EX 流水级和 TLB 模块。

## (2) TLB 例外处理

本实验首先要修改 INE 例外的逻辑，同时还要增添对 PIF、PIL、PIS、PPI、PME 和 TLBR 例外的处理。在之前的实验，INE 取指地址错误仅仅处理译码得到的指令不是以及支持的指令，但在本实验还要考虑 INVTLB 且其 op 不合法的情况。如果是 INVTLB 且其 op 大于 6，也要报出 INE 例外。

TLB 重填例外是指在页表映射模式下没有命中 TLB，并且处理 TLB 重填例外的中断入口区别于其他例外。取指操作页无效例外（PIF）需要在 IF 级进行判断，如果此时处于页表映射模式并且查找到的 TLB 表项无效，即 s\_v 为 0，则报错。load 操作页无效例外（PIL）和 store 操作页无效例外（PIS）需要在 EXE 级进行判断，判断例外的过程和 PIF 类似，只是还需要结合指令类型。页特权等级不合规例外需要比较命中 TLB 表项的 PLV 域和从 CRMD 寄存器中取出的当前 PLV，如果前者小则报出该错。页修改例外也要在 EXE 级进行判断，如果此时执行的是 store 指令且处于页表映射模式，如果命中的 TLB 表项脏位为 0，则报出该例外。

需要注意的是，在 EXE 级如果触发 TLB 相关例外，说明虚实地址翻译环节出现错误，没有得到正确的物理地址，因此此时不应该向总线发起请求，即不能拉高 data\_sram\_req。此时不向总线发起请求，所以也不用等到总线发来的 addr\_ok 和 data\_ok 信号。

此外，在处理例外逻辑时，还要注意这些例外的优先级。根据龙芯架构的指令集手册的说明，IF 阶段触发的例外优先级最高，ID 阶段次之，EX 级再次之。而在 IF 级触发的例外中，ADEF 例外的优先级高于 TLB 相关例外；在 EX 流水级触发的例外中，ALE 例外的优先级高于 TLB 的相关例外；在 TLB 相关例外中，TLBR 的优先级最高，然后为 PIF 或者 PIL 或者 PIS，再其次为 PPI，最后为 PME。在 WB 级中，需要按照上面逻辑，比较各个例外的优先级来设置 ecode 和 esubcode：

```

//----CSR relevant signals and data-----
assign wb_csr_ecode = wb_has_int      ? `ECODE_INT :
                        wb_excp_adef    ? `ECODE_ADE :
                        wb_inst_tlbr_excep? `ECODE_TLBR:
                        wb_inst_pif_excep ? `ECODE_PIF :
                        wb_inst_ppi_excep ? `ECODE_PPI :
                        wb_excp_ine      ? `ECODE_INE :
                        wb_excp_syscall   ? `ECODE_SYS :
                        wb_excp_break     ? `ECODE_BRK :
                        wb_excp_ale       ? `ECODE_ALE :
                        wb_data_tlbr_excep? `ECODE_TLBR:
                        wb_data_pil_excep ? `ECODE_PIL :
                        wb_data_pis_excep ? `ECODE_PIS :
                        wb_data_ppi_excep ? `ECODE_PPI :
                        wb_data_pme_excep ? `ECODE_PME :
                        6'b0;
assign wb_csr_esubcode = 9'b0;

```

图 7 例外处理的优先级

## (二) 重要模块 1 设计：TLB 模块

### 1、工作原理

TLB 采用全相联查找表的组织形式。在 TLB 模块中保存了 TLB 页表的内容，共 16 个页表项。并通过 2 个查找端口，1 个读端口，1 个写端口和一个 INVTLB 指令端口与外部交换数据。实现了对 TLB 页表的 4 种操作：

**查找：**输入虚双页号 vppn、虚地址 12 位 va\_bit12 与地址空间标识 asid，得到对应物理页的内容。

**写：**输入页表项数 index，并将输入的页表项内容写到对应的页表项里。

**读：**输入页表项数 index，并得到对应的页表项中的内容。

**无效指令 INVTLB：**使满足 opcode 对应条件的页表项无效化。

### 2、接口定义

表 1 TLB 模块接口信号

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
<b>Search port 0</b>			
s0_vppn	IN	19	查找虚双页号
s0_va_bit12	IN	1	访存虚地址 12 位
s0_asid	IN	10	地址空间标识
s0_found	OUT	1	查找到信号
s0_index	OUT	4	页表项索引
s0_ppn	OUT	20	物理页号
s0_ps	OUT	6	页大小
s0_plv	OUT	2	权限等级
s0_mat	OUT	2	访存种类
s0_d	OUT	1	脏页信号
s0_v	OUT	1	页有效信号

名称	方向	位宽	功能描述
<b>Search port 1</b>			
s1_vpnn	IN	19	查找虚双页号
s1_va_bit12	IN	1	访存虚地址 12 位
s1_asid	IN	10	地址空间标识
s1_found	OUT	1	查找到信号
s1_index	OUT	4	页表项索引
s1_ppn	OUT	20	物理页号
s1_ps	OUT	6	页大小
s1_plv	OUT	2	权限等级
s1_mat	OUT	2	访存种类
s1_d	OUT	1	脏页信号
s1_v	OUT	1	页有效信号
<b>INVTLB opcode</b>			
invtlb_valid	IN	1	无效指令有效信号
invtlb_op	IN	5	无效指令操作码
<b>Write port</b>			
we	IN	1	写使能信号
w_index	IN	4	写页表项索引
w_e	IN	0	写页表项有效信号
w_vpnn	IN	19	写虚拟页号
w_ps	IN	6	写页大小
w_asid	IN	10	写进程号
w_g	IN	1	写全局信号
w_ppn0	IN	20	写物理页号 0
w_plv0	IN	2	写权限等级 0
w_mat0	IN	2	写访存种类 0
w_d0	IN	1	写脏页信号 0
w_v0	IN	1	写页有效信号 0
w_ppn1	IN	20	写物理页号 1
w_plv1	IN	2	写权限等级 1
w_mat1	IN	2	写访存种类 1
w_d1	IN	1	写脏页信号 1
w_v1	IN	1	写页有效信号 1
<b>Read port</b>			
r_index	IN	4	读页表项索引
r_e	OUT	0	读页表项有效信号
r_vpnn	OUT	19	读虚拟页号
r_ps	OUT	6	读页大小
r_asid	OUT	10	读进程号
r_g	OUT	1	读全局信号
r_ppn0	OUT	20	读物理页号 0
r_plv0	OUT	2	读权限等级 0
r_mat0	OUT	2	读访存种类 0
r_d0	OUT	1	读脏页信号 0
r_v0	OUT	1	读页有效信号 0
r_ppn1	OUT	20	读物理页号 1
r_plv1	OUT	2	读权限等级 1
r_mat1	OUT	2	读访存种类 1
r_d1	OUT	1	读脏页信号 1
r_v1	OUT	1	读页有效信号 1

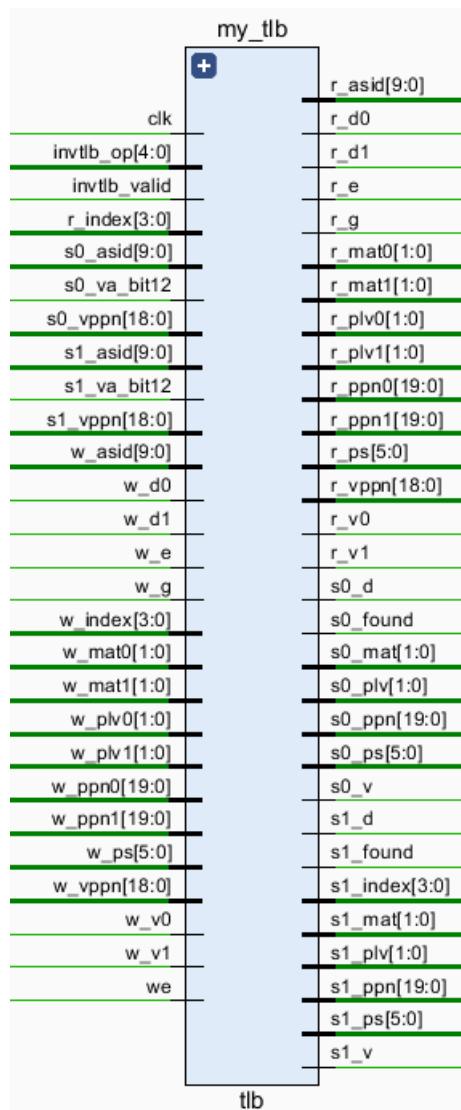


图 8 TLB 模块接口

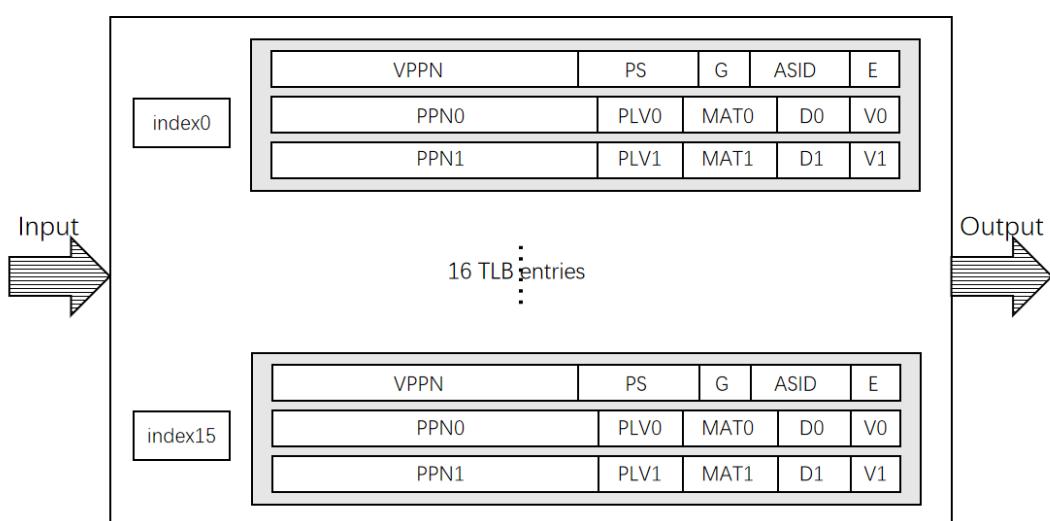


图 9 TLB 模块示意图

### 3、功能描述

#### (1) TLB 结构

TLB 页表中每一个页表项为一个双页，包含比较部分和物理转换部分，结构如下图：

VPPN	PS	G	ASID	E
PPN0	RPLV0 PLV0 MAT0 NX0 NRO DO V0			
PPN1	RPLV1 PLV1 MAT1 NX1 NR1 D1 V1			

图 10 TLB 表项格式

TLB 表项比较部分包括：

VPPN：虚双页号，19位，虚页号最低位不保存在 TLB 中。

PS：页大小（2的幂次），6位，本次实验中有 21（2MB 页）和 12（4KB 页）两种情况。

G：全局标志位，1位，为 1 时该虚拟地址在所有进程间共享。

ASID：地址空间标识，10位，用于区分不同进程中同样的虚地址。

E：存在位，1位，表示该表项存在。

TLB 表项物理转换部分保存一对奇偶相邻页表的物理转换信息，包括：

PPN：物理页号，20位。

PLV：特权等级，2位。

MAT：存储访问类型，2位。

D：脏位，1位，表示该页表项地址范围内有脏数据。

V：有效位，1位，表示该页有效。

RPLV、NX、NR 在本次实验中未被定义。

本次实验中实现的 TLB 页表中包含 16 个页表项，能保存 16 对 32 个页。页大小有 2MB 与 4KB 两种，通过 ps4MB 信号指示。使用 16 组寄存器来保存页表内容：

```

reg [`TLBNUM-1:0] tlb_e;
reg [`TLBNUM-1:0] tlb_ps4MB; //pagesize 1:4MB 0:4KB
reg [18:0] tlb_vppn [`TLBNUM-1:0];
reg [ 9:0] tlb_asid [`TLBNUM-1:0];
reg tlb_g [`TLBNUM-1:0];

reg [19:0] tlb_ppn0 [`TLBNUM-1:0];
reg [ 1:0] tlb_plv0 [`TLBNUM-1:0];
reg [ 1:0] tlb_mat0 [`TLBNUM-1:0];
reg tlb_d0 [`TLBNUM-1:0];
reg tlb_v0 [`TLBNUM-1:0];

reg [19:0] tlb_ppn1 [`TLBNUM-1:0];
reg [ 1:0] tlb_plv1 [`TLBNUM-1:0];
reg [ 1:0] tlb_mat1 [`TLBNUM-1:0];
reg tlb_d1 [`TLBNUM-1:0];
reg tlb_v1 [`TLBNUM-1:0];

```

图 11 TLB 页表存储寄存器组

## (2) TLB 操作

**搜索：**

实现两个相同搜索端口的并行搜索，一个用于取指，一个用于内存读写指令。

输入为虚双页号 vppn，访存虚地址 12 位 va\_bit\_12 与进程号 asid。

首先搜索有没有匹配的页表项，得到页表中每一项是否匹配的 match 信号（01 表示两个搜索端口）：

```

genvar i;
generate
    for(i=0; i<TLBNUM; i=i+1) begin
        assign match0[i] = (s0_vppn[18:9] == tlb_vppn[i][18:9])
            && (tlb_ps4MB[i] || s0_vppn[8:0] == tlb_vppn[i][8:0])
            && ((s0_asid == tlb_asid[i]) || tlb_g[i]);
        assign match1[i] = (s1_vppn[18:9] == tlb_vppn[i][18:9])
            && (tlb_ps4MB[i] || s1_vppn[8:0] == tlb_vppn[i][8:0])
            && ((s1_asid == tlb_asid[i]) || tlb_g[i]);
    end
endgenerate

```

图 12 TLB 页表匹配逻辑

当页大小为 2MB 时，匹配 vppn 的[18:9]位，页大小为 4KB 时，匹配 vppn 的所有位。

如果 G 位为 1 则不用考虑 ASID，否则对 ASID 进行匹配。

然后得到 found, index, whichpage, ps 信号（端口 0 为例）：

```

assign s0_found = (|match0);
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 s0_whichpage = (tlb_ps4MB[s0_index])? s0_vppn[8]: s0_va_bit12;
assign s0_ps      = (tlb_ps4MB[s0_index])? 6'd21 : 6'd12;

```

图 13 found, index, whichpage 与 ps 信号生成逻辑

found: 是否匹配到。

index: 匹配到的页表项的项数。

whichpage: 双页中的哪一页, 页大小为 2MB 时通过 vppn 的第 8 位来判断, 4KB 时通过输入的 va\_bit12 来判断。

ps: 页大小, 2MB 为 21, 4KB 为 12。

最后通过 index 和 whichpage 从页表中得到物理转换信息并输出:

```

assign s0_ppn      = (s0_whichpage) ? tlb_ppn1[s0_index] : tlb_ppn0[s0_index];
assign s0_plv      = (s0_whichpage) ? tlb_plv1[s0_index] : tlb_plv0[s0_index];
assign s0_mat      = (s0_whichpage) ? tlb_mat1[s0_index] : tlb_mat0[s0_index];
assign s0_d        = (s0_whichpage) ? tlb_d1 [s0_index] : tlb_d0 [s0_index];
assign s0_v        = (s0_whichpage) ? tlb_v1 [s0_index] : tlb_v0 [s0_index];

```

图 14 物理转换信息输出

读:

根据输入的项数 r\_index, 将对应页表项内的内容输出即可:

```

// read
assign r_e      = tlb_e [r_index];
assign r_vppn   = tlb_vppn [r_index];
assign r_ps     = tlb_ps4MB[r_index] ? 6'd21 : 6'd12;
assign r_asid   = tlb_asid [r_index];
assign r_g      = tlb_g [r_index];

assign r_ppn0   = tlb_ppn0 [r_index];
assign r_plv0   = tlb_plv0 [r_index];
assign r_mat0   = tlb_mat0 [r_index];
assign r_d0     = tlb_d0 [r_index];
assign r_v0     = tlb_v0 [r_index];

assign r_ppn1   = tlb_ppn1 [r_index];
assign r_plv1   = tlb_plv1 [r_index];
assign r_mat1   = tlb_mat1 [r_index];
assign r_d1     = tlb_d1 [r_index];
assign r_v1     = tlb_v1 [r_index];

```

图 15 TLB 页表读逻辑

写:

根据输入的写使能信号 we 与项数 w\_index，将输入内容填入对应的页表项即可：

```
// write
always @(posedge clk) begin
    if (we) begin
        tlb_e [w_index] <= w_e;
        tlb_ps4MB[w_index] <= (w_ps == 6'd21);
        tlb_vppn [w_index] <= w_vppn;
        tlb_asid [w_index] <= w_asid;
        tlb_g [w_index] <= w_g;

        tlb_ppn0 [w_index] <= w_ppn0;
        tlb_plv0 [w_index] <= w_plv0;
        tlb_mat0 [w_index] <= w_mat0;
        tlb_d0 [w_index] <= w_d0;
        tlb_v0 [w_index] <= w_v0;

        tlb_ppn1 [w_index] <= w_ppn1;
        tlb_plv1 [w_index] <= w_plv1;
        tlb_mat1 [w_index] <= w_mat1;
        tlb_d1 [w_index] <= w_d1;
        tlb_v1 [w_index] <= w_v1;
    end

```

图 16 TLB 页表写逻辑

### 无效指令：

invlb 指令的 opcode 对应的操作如下表：

表 2 INVTLB 指令 opcode 对应操作表

opcode	操作
0x0	清除所有页表项
0x1	清除所有页表项
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	清除所有 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
0x7~0x20	不进行清除操作

由此对每个页表项，定义 4 个条件 cond1~cond4：

```

generate
    for(i=0; i<TLBNUM; i=i+1) begin
        assign cond1[i] = ~tlb_g[i];
        assign cond2[i] = tlb_g[i];
        assign cond3[i] = s1_asid == tlb_asid[i];
        assign cond4[i] = (s1_vppn[18:9] == tlb_vppn[i][18:9])
            && (tlb_ps4MB[i] || s1_vppn[8:0] == tlb_vppn[i][8:0]);
    end
endgenerate

```

图 17 cond 信号生成

根据 cond1~cond4 得到 mask 信号如下：

```

assign invtlb_mask[0] = 16'hffff;
assign invtlb_mask[1] = 16'hffff;
assign invtlb_mask[2] = cond2;
assign invtlb_mask[3] = cond1;
assign invtlb_mask[4] = cond1 & cond3;
assign invtlb_mask[5] = cond1 & cond3 & cond4;
assign invtlb_mask[6] = (cond1|cond3) & cond4;
generate
    for (i = 7; i < 32; i=i+1) begin
        assign invtlb_mask[i] = 16'b0;
    end
endgenerate

```

图 18 mask 信号生成

在 invtlb 指令有效时，根据输入的 opcode 选择 mask，将 mask 对应位为 1 的页表项无效化（e 值清零），为 0 的页表项不变：

```

else if(invtlb_valid) begin
    tlb_e <= ~invtlb_mask[invtlb_op] & tlb_e;
end

```

图 19 无效化操作

### (三) 重要模块 2 设计：CSR 模块

#### 1、工作原理

新增用于虚实地址转换及 TLB 相关例外处理的寄存器

#### 2、接口定义

表 3 CSR 模块接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟输入
resetn	IN	1	复位信号
csr_num	IN	14	控制状态寄存器读写地址

名称	方向	位宽	功能描述
csr_we	IN	1	控制状态寄存器写使能
csr_wmask	IN	32	控制状态寄存器写掩码
csr_wvalue	IN	32	控制状态寄存器写数据
ertn_flush	IN	1	ertn 指令刷新信号
wb_ex	IN	1	例外发生信号
wb_ecode	IN	6	例外发生 ecode 代码
wb_esubcode	IN	9	例外发生 esubcode 代码
wb_pc	IN	32	WB 级 PC 值, 用于记录例外发生的 PC
csr_rvalue	OUT	32	控制状态寄存器读数据
ex_entry	OUT	32	例外跳转入口地址
tlb_ex_entry	OUT	32	TLB 例外跳转入口地址
has_int	OUT	1	发生中断信号
ipi_int_in	IN	1	核间中断输入
coreid_in	IN	32	核编号, 也是 TID 初值
hw_int_in	IN	8	硬件中断输入
wb_vaddr	IN	32	出错地址
csr_tlbhi_vppn	OUT	19	CSR.TLBHI.VPPN, TLB 表项的虚页号
csr_tlbidx_index	OUT	4	CSR.TLBIDX.INDEX, TLB 表项索引
tlbsrch_we	IN	1	TLBSRCH 指令的写使能
tlbsrch_hit	IN	1	TLBSRCH 查询命中
tlbrd_we	IN	1	TLBRD 指令的写使能
tlbsrch_hit_index	IN	4	TLBSRCH 查询命中的索引
csr_crmd_rvalue	OUT	32	CSR.CRMD 的值
csr_asid_rvalue	OUT	32	CSR.ASID 的值
csr_dmw0_rvalue	OUT	32	CSR.DMW0 的值
csr_dmw1_rvalue	OUT	32	CSR.DMW1 的值
r_tlb_e	IN	1	从 TLB 读出的 e 域信息
r_tlb_vppn	IN	19	从 TLB 读出的 vppn 域信息
r_tlb_ps	IN	6	从 TLB 读出的 ps 域信息
r_tlb_asid	IN	10	从 TLB 读出的 asid 域信息
r_tlb_g	IN	1	从 TLB 读出的 g 域信息
r_tlb_ppn0	IN	20	从 TLB 读出的偶页 ppn 域信息
r_tlb_plv0	IN	2	从 TLB 读出的偶页 plv 域信息
r_tlb_mat0	IN	2	从 TLB 读出的偶页 mat 域信息
r_tlb_d0	IN	1	从 TLB 读出的偶页 d 域信息
r_tlb_v0	IN	1	从 TLB 读出的偶页 v 域信息
r_tlb_ppn1	IN	20	从 TLB 读出的奇页 ppn 域信息
r_tlb_plv1	IN	2	从 TLB 读出的奇页 plv 域信息
r_tlb_mat1	IN	2	从 TLB 读出的奇页 mat 域信息
r_tlb_d1	IN	1	从 TLB 读出的奇页 d 域信息
r_tlb_v1	IN	1	从 TLB 读出的奇页 v 域信息
w_tlb_e	OUT	1	TLB 写行为的 e 域信息
w_tlb_vppn	OUT	19	TLB 写行为的 vppn 域信息
w_tlb_ps	OUT	6	TLB 写行为的 ps 域信息
w_tlb_asid	OUT	10	TLB 写行为的 asid 域信息
w_tlb_g	OUT	1	TLB 写行为的 g 域信息
w_tlb_ppn0	OUT	20	TLB 写入偶页的 ppn 域信息
w_tlb_plv0	OUT	2	TLB 写入偶页的 plv 域信息
w_tlb_mat0	OUT	2	TLB 写入偶页的 mat 域信息
w_tlb_d0	OUT	1	TLB 写入偶页的 d 域信息
w_tlb_v0	OUT	1	TLB 写入偶页的 v 域信息
w_tlb_ppn1	OUT	20	TLB 写入奇页的 ppn 域信息
w_tlb_plv1	OUT	2	TLB 写入奇页的 plv 域信息
w_tlb_mat1	OUT	2	TLB 写入奇页的 mat 域信息
w_tlb_d1	OUT	1	TLB 写入奇页的 d 域信息
w_tlb_v1	OUT	1	TLB 写入奇页的 v 域信息

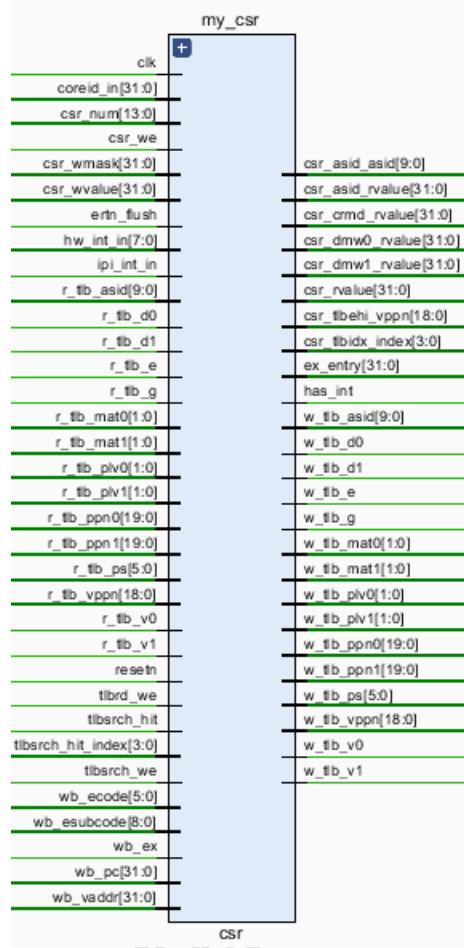


图 20 CSR 模块接口

### 3、功能描述

在虚实地址转换时，地址翻译模式由 CRMD 中的 DA 和 PG 域控制，为此需要让这两个域能被 csr 指令更新。

当 DA=0，PG=1 时，处理器核的 MMU 处于直接地址翻译模式。当 DA=0，PG=1 时，MMU 处于映射地址翻译模式，这种模式又可分为直接映射地址翻译模式和页表映射地址翻译模式。

对于直接映射地址翻译模式，相关配置信息存储在 DMW0、DMW1 寄存器中，每个窗口除了地址范围信息外，还可以配置该窗口在哪些特权等级下可用，以及虚地址落在该窗口上的访存操作的存储访问类型。存储访问类型由 CRMD 的 DATF 和 DATM 域控制。这些域都只需支持 csr 指令的读写。

对于没有落在 DMW0、DMW1 设置的直接映射配置窗口中的地址，需要采用页表映射地址翻译模式，为此也需要在 CSR 模块中增加 TLB 相关的控制状态寄存器。TLBIDX 包含 TLB 指令操作 TLB 时的索引值、PS 以及是否有效等信息。TLBEHI 包含 TLB 表项高位的虚页号，TLBELO0、TLBELO1 则包含表项低位的物页号、有效、脏等信息。ASID 寄存器包含地址空间标识符等信息。TLBSRCH、TLBRD、TLBWR、TLBFILL 指令会对这些寄存器进行读写。于是，这些寄存器更新的逻辑需要考虑两种情况：被 TLB 维护指令更新，或是被之前实验实现的 csr 写指令更新。TLBEHI 在此基础上还需多考虑一种情况，即在发生例外时将触发例外的虚地址记录到其 VPPN 域中。

根据这些 TLB 相关的控制状态寄存器，便可以产生传递给 TLB 模块的各种信号，其中需要注意的是，传递给 TLB 的 w\_e 信号在 TLB 充填例外时恒为 1 而不受 TLBIDX 的 NE 位的影响。

在发生 TLB 重填例外时，处理器核将通过更新 DA 和 PG 来进入直接地址翻译模式。TLB 重填例外的入口地址储存在 TLBENTRY 中，该寄存器也只需支持 csr 指令的读写。在处理完例外后，处理器执行 ETRN 指令，此时需要让 DA=0, PG=1，回到映射地址翻译模式。因此，DA 和 PG 除了要支持 csr 指令的更新，还需考虑在发生 TLB 重填例外和执行 ERTN 时的更新。

## (四) 重要模块 3 设计：MMU 模块

### 1、工作原理

MMU 模块完成虚实地址的转换，包括判断映射模式以及查找 TLB 等操作，同时 MMU 模块还会输出一系列用于判断 TLB 相关异常的信号。

### 2、接口定义

表 4 MMU 模块接口定义

名称	方向	位宽	功能描述
<b>Search port</b>			
s_vppn	OUT	19	查找虚双页号
s_va_bit12	OUT	1	访存虚地址 12 位
s_asid	OUT	10	地址空间标识
s_found	IN	1	查找到信号
s_ppn	IN	20	物理页号
s_ps	IN	6	页大小
s_plv	IN	2	权限等级
s_mat	IN	2	访存种类
s_d	IN	1	脏页信号
s_v	IN	1	页有效信号
<b>virtual addr and physical addr</b>			
va	IN	32	虚拟地址
pa	OUT	32	物理地址
asid_input	IN	10	输入地址空间标识
<b>Data from csr</b>			
csr_crmd_rvalue	IN	32	读 CRMD 寄存器的值
csr_dmwo_rvalue	IN	32	读 DMW0 寄存器的值
csr_dmwi_rvalue	IN	32	读 DMW1 寄存器的值
<b>exception</b>			
page_invalid	OUT	1	页无效
ppi_except	OUT	1	页特权等级不合规例外
page_fault	OUT	1	页不匹配
page_clean	OUT	1	页非脏

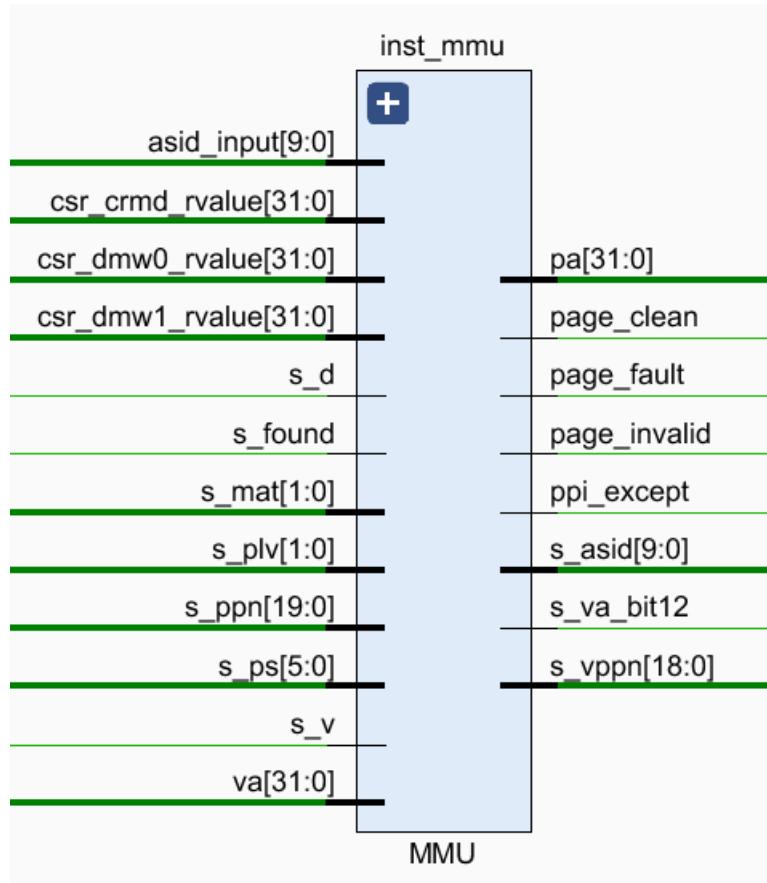


图 21 MMU 模块接口（取指 MMU 为例）

### 3、功能描述

MMU 模块完成虚实地址的转换，地址翻译模式由 CRMD 中的 DA 和 PG 域控制。当 DA=0, PG=1 时，处理器核的 MMU 处于直接地址翻译模式。当 DA=0, PG=1 时，MMU 处于映射地址翻译模式，这种模式又可分为直接映射地址翻译模式和页表映射地址翻译模式。

对于直接映射地址翻译模式，相关配置信息存储在 DMW0、DMW1 寄存器中。每个窗口除了地址范围信息外，还可以配置该窗口在哪些特权等级下可用，以及虚地址落在该窗口上的访存操作的存储访问类型。

当虚地址命中某个有效的直接映射配置窗口时，其物理地址直接等于虚地址的[28:0]位拼接上该映射窗口所配置的物理地址高位。命中的判断方式是：虚地址的最高 3 位 ([31:29]位) 与配置窗口寄存器中的[31:29]相等，且当前特权等级在该配置窗口中被允许。映射地址翻译模式下，除了落在直接映射配置窗口中的地址之外，其余所有合法地址都必须通过页表映射完成虚实地址转换。

```

assign csr_crmd_da = csr_crmd_rvalue[ CSR_CRMD_DA];
assign csr_crmd_pg = csr_crmd_rvalue[ CSR_CRMD_PG];
assign csr_crmd_plv = csr_crmd_rvalue[ CSR_CRMD_PLV];

assign direct_trans = csr_crmd_da && ~csr_crmd_pg;
assign map_trans = ~csr_crmd_da && csr_crmd_pg;

assign dmw0_hit = map_trans && csr_dmw0_rvalue[csr_crmd_plv] &&
                  (csr_dmw0_rvalue[`CSR_DMW_VSEG] == va[`CSR_DMW_VSEG]);
assign dmw1_hit = map_trans && csr_dmw1_rvalue[csr_crmd_plv] &&
                  (csr_dmw1_rvalue[`CSR_DMW_VSEG] == va[`CSR_DMW_VSEG]);

assign dmw_pa0 = {csr_dmw0_rvalue[`CSR_DMW_PSEG], va[28:0]}; //csr_dmw_rvalue[27:25] = csr_dmw_pseg
assign dmw_pa1 = {csr_dmw1_rvalue[`CSR_DMW_PSEG], va[28:0]};

//tlb mapping
assign tlb_map = ~dmw0_hit & ~dmw1_hit & map_trans;

```

图 22 MMU 模块对于地址翻译模式的判断

对于页表映射地址翻译模式，TLB 作为处理器中存放操作系统页表信息的一个临时缓存，用于加速映射地址翻译模式下的取指和 load/store 操作的虚实地址转换过程。所以只需要把虚拟地址的相关信息传给 TLB 模块，如果能匹配到页表项，则根据 TLB 模块返回的 PS 字段的值，把物理页表项 s\_ppn 拼接上对应的 offset，便可以得到页表映射的物理地址。

```

assign tlb_map = ~dmw0_hit & ~dmw1_hit & map_trans;

assign {s_vppn, s_va_bit12} = va[31:12];
assign s_asid = asid_input;

assign tlb_pa = {32{s_ps == 6'd12}} & {s_ppn[19:0], va[11:0]} |
                {32{s_ps == 6'd21}} & {s_ppn[19:9], va[20:0]};

```

图 23 MMU 模块对于页表映射地址翻译模式下的物理地址的产生

如果没有匹配到页表项，则把输出信号 page\_fault 拉高，方便 TLB 重填例外的判断。除此之外，MMU 模块还会根据 TLB 模块返回的页表项的 V, D, PLV 字段的值，对输出信号进行赋值，方便进行 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的判断。

```

assign pa = direct_trans ? va
                      : dmw0_hit ? dmw_pa0
                      : dmw1_hit ? dmw_pa1
                      : tlb_pa;

//for exception
assign page_invalid = tlb_map & ~s_v;
assign ppi_except = tlb_map & (csr_crmd_plv > s_plv);
assign page_fault = tlb_map & ~s_found;
assign page_clean = tlb_map & ~s_d;

```

图 24 MMU 模块物理地址的产生以及用于例外判断的信号

## (五) 重要模块 4 设计：IF 流水级

### 1、工作原理

IF 模块内包含一个伪流水级 pre-IF，pre-IF 级负责发送请求并等待地址握手，IF 级负责接收指令。如果需要取消指令，则将 IF 级或 pre-IF 级的对于 cancel 信号拉高，并把 if\_inst\_reg\_valid 信号置为 0，从而完成对后面返回指令的取消

### 2、接口定义

表 5 IF 流水级接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
resetn	IN	1	复位信号
<b>Inst sram interface</b>			
inst_sram_req	OUT	1	指令 sram 请求信号
inst_sram_wr	OUT	1	指令 sram 读写控制信号
inst_sram_size	OUT	2	指令 sram 该次请求传输的字节数
inst_sram_wstrb	OUT	4	指令 sram 该次请求的字节写使能信号
inst_sram_addr	OUT	32	指令 sram 请求地址
inst_sram_wdata	OUT	32	指令 sram 该次写请求的写数据
inst_sram_addr_ok	IN	1	指令 sram 该次请求的地址传输 OK
inst_sram_data_ok	IN	1	指令 sram 该次请求的数据传输 OK
inst_sram_rdata	IN	32	指令 sram 该次请求返回的读数据
<b>AXI interface</b>			
axi_arid	IN	4	AXI 转接桥的读请求 ID 号
<b>ID interface</b>			
id_allowin	IN	1	ID 流水级是否允许 IF 流水级传入数据
br_taken	IN	1	ID 流水级传来的跳转信号
br_stall	IN	1	ID 流水级传来的跳转阻塞信号
br_target	IN	32	ID 流水级传来的跳转地址
if_to_id_valid	OUT	1	标记 IF 流水级向 ID 流水级传递的数据是否有效
if_to_id_data	OUT	64	IF 模块向 ID 模块传递的数据
if_to_id_excep	OUT	1	IF 模块向 ID 模块传递的异常信息
<b>WB interface</b>			
wb_to_if_csr_data	IN	66	WB 模块传给 IF 模块的 csr 数据
if_flush	IN	1	传给 IF 模块的清空流水线信号
<b>MMU interface</b>			
csr_asid_rvalue	IN	32	从 csr 的 ASID 寄存器中读出地址空间标识
inst_va	OUT	32	指令虚拟地址
inst_pa	IN	32	指令物理地址
if_asid	OUT	10	地址空间标识
inst_page_invalid	IN	1	页无效
inst_ppi_except	IN	1	页特权等级不合规例外
inst_page_fault	IN	1	页不匹配
inst_page_clean	IN	1	页非脏

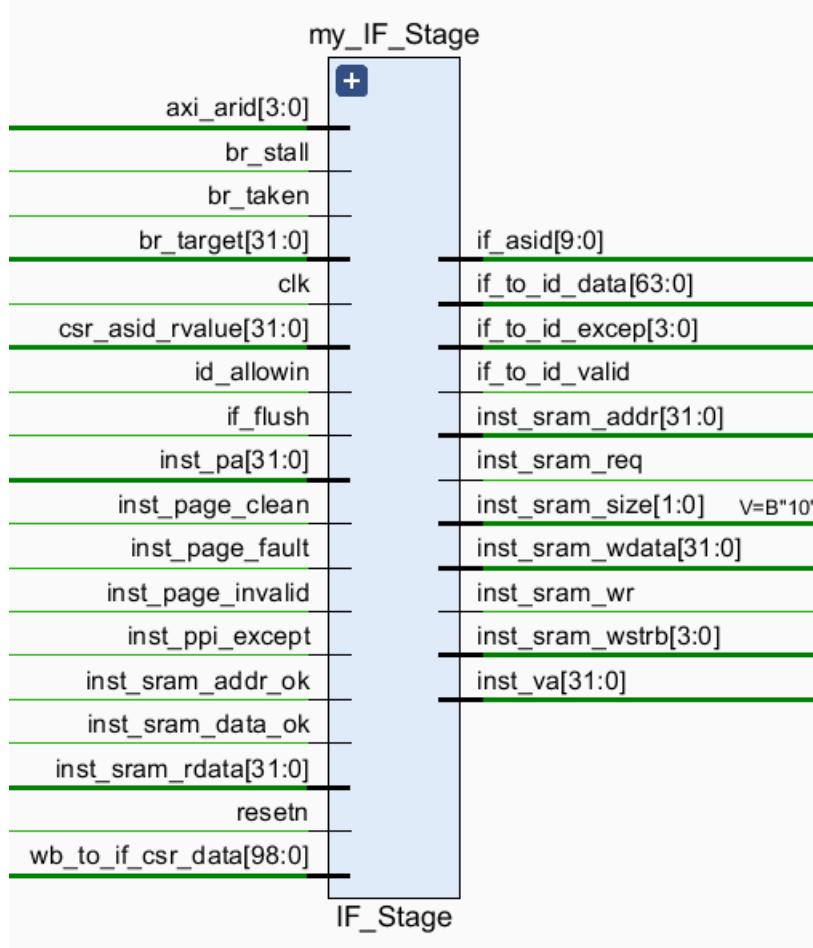


图 25 IF 模块接口

### 3、功能描述

在之前实验的基础上，本次实验在 IF 模块的伪流水级 pre-IF 中增加了对虚地址 nextpc 的地址转换功能。为了支持该功能，还需要在其中增添对 TLB 的查找操作并处理查找过程中出现的例外，然后将这些例外传递给 ID 级进行下一步操作。而虚实地址的转换以及例外的判断则主要通过 MMU 模块处理。

```
//-----to mmu for va->pa and tlb except-----
assign inst_va = nextpc;
assign if_asid = csr_asid_rvalue[`CSR_ASID_ASID];

assign if_pif_excep = inst_page_invalid;
assign if_ppi_excep = inst_ppi_except;
assign if_tlbr_excep = inst_page_fault;

//to id exception
assign if_to_id_excep = {if_adef_excep, if_pif_excep, if_ppi_excep, if_tlbr_excep};
```

图 26 IF 模块的虚实地址转换和 TLB 例外判断

除此之外，由于新添了流水线的回滚操作，所以将 WB 流水级的 PC 传到了 IF 流水级，用于对 nextpc 的选择，

最终逻辑如下：

```
assign seq_pc          = if_pc + 3'h4;
assign nextpc          = wb_ertn_flush_valid ? csr_rvalue_reg
                      : wb_ertn_flush_valid ? csr_rvalue
                      : wb_csr_ex_valid ? ex_entry_reg
                      : wb_csr_ex_valid ? ex_entry
                      : wb_tlb_refetch_valid ? wb_pc_reg + 32'h4
                      : wb_tlb_refetch_valid ? wb_pc + 32'h4
                      : br_taken_reg ? br_target_reg
                      : br_taken ? br_target
                      : seq_pc;
```

图 27 IF 模块 nextpc 更新逻辑的修改

## (六) 重要模块 5 设计：ID 流水级

### 1、工作原理

新增 TLB 指令的译码，TLB 相关例外以及对 INE 例外的修改。其余不变。

### 2、接口定义

表 6 ID 流水级接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
resetn	IN	1	复位信号
id_allowin	OUT	1	ID 流水级允许 IF 流水级传入数据
br_taken	OUT	1	传给 IF 流水级的跳转信号
br_stall	OUT	1	传给 IF 流水级的跳转阻塞信号
br_target	OUT	32	传给 IF 流水级的跳转地址
if_to_id_valid	IN	1	标记 IF 流水级传入 ID 流水级的数据是否有效
if_to_id_data	IN	64	IF 流水级传给 ID 流水级的数据
if_to_id_excep	IN	4	IF 流水级传给 ID 流水级的异常信息
ex_allowin	IN	1	EX 流水级允许 ID 流水级传入数据
id_to_ex_data	OUT	163	ID 流水级传入 EX 流水级的数据
id_to_ex_excep	OUT	89	ID 流水级传入 EX 流水级的异常信息
id_to_ex_tlb	OUT	10	ID 流水级传入 EX 流水级的 TLB 信息
id_to_ex_valid	OUT	1	标记 ID 流水级传入 EX 流水级的数据是否有效
wb_rf_zip	IN	38	WB 流水级向 ID 流水级传递的 regfile 的写回信息和前递数据
mem_rf_zip	IN	39	MEM 流水级前递到 ID 流水级的数据
ex_rf_zip	IN	40	EX 流水级前递到 ID 流水级的数据
id_flush	IN	1	ID 模块收到的清空流水线信号
has_int	IN	1	若 WB 模块判断有中断，将 ID 阶段的指令进行标记

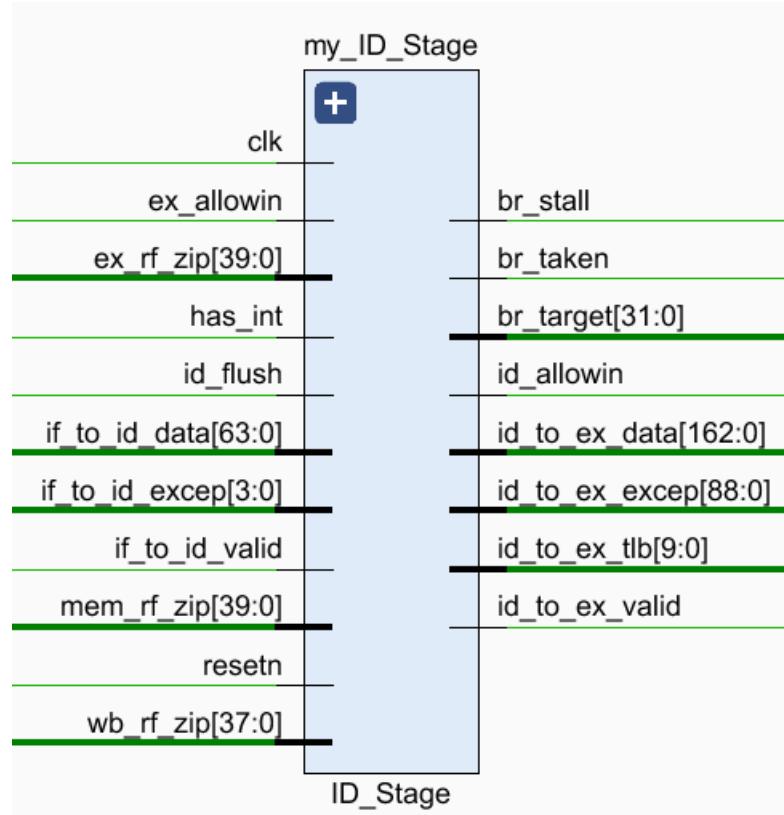


图 28 ID 流水级接口

### 3、功能描述

本实验新增的 5 条 TLB 指令在 ID 级译码，并将控制信号传入 EXE 级：

另外在之前的实验，INE 取指地址错误仅仅处理译码得到的指令不是以及支持的指令，但在本实验还要考虑 INVTLB 且其 op 不合法的情况。如果是 INVTLB 且其 op 大于 6，也要报出 INE 例外

```

assign id_excp_ine      = (~type_calc | type_calc_i | type_branch_uncond | type_branch_cond |
                           type_load | type_store | type_excep | type_tlb | type_others) |
                           inst_invtlb & intvtlb_op_fault) & id_valid;

assign intvtlb_op_fault = rd[4] | rd[3] | (&rd[2:0]);

```

图 29 INE 例外判断逻辑的修改

## (七) 重要模块 6 设计：EX 流水级

### 1、工作原理

新增虚实地址转换的过程以及 TLB 例外的判断。根据指令的需要，进行虚实地址转换后向数据 RAM 发起请求。此外，TLBSRCH 在 EX 级查询 TLB，INVTLB 也在 EX 级进行处理。其余功能不变。

### 2、接口定义

表 7 EX 流水级接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
resetn	IN	1	复位信号
<b>EX stage interface</b>			
ex_allowin	OUT	1	EX 模块允许 ID 模块传入数据
id_to_ex_data	IN	163	ID 模块传入 EX 模块的数据
id_to_ex_excep	IN	89	ID 模块传入 EX 模块的异常信息
id_to_ex_tlb	IN	20	ID 模块传入 EX 模块的 TLB 信息
id_to_ex_valid	IN	1	标记 ID 模块传入 EX 模块的数据是否有效
<b>MEM stage interface</b>			
mem_allowin	IN	1	MEM 模块允许 EX 模块传入数据
ex_to_mem_data	OUT	79	EX 模块传入 MEM 模块的数据
ex_to_mem_excep	OUT	95	EX 模块传入 MEM 模块的异常信息
ex_to_mem_tlb	OUT	20	EX 模块传入 MEM 模块的 TLB 信息
ex_to_mem_valid	OUT	1	标记 EX 模块传入 MEM 模块的数据是否有效
ex_rf_zip	OUT	39	EX 模块前递到 ID 模块的数据
mul_result	OUT	64	乘法器得到的结果
mem_to_ex_excep	IN	1	MEM 模块向 EX 前传的异常信息
<b>Data sram interface</b>			
data_sram_req	OUT	1	数据 ram 请求信号
data_sram_wr	OUT	1	数据 ram 读写控制信号
data_sram_size	OUT	2	数据 ram 请求字节数
data_sram_wstrb	OUT	4	数据 ram 写使能信号
data_sram_addr	OUT	32	数据 ram 请求地址
data_sram_wdata	OUT	32	数据 ram 写数据
data_sram_addr_ok	IN	1	数据 ram 地址握手信号
<b>Flush signal</b>			
ex_flush	IN	1	清空流水线信号
<b>TLB interface</b>			
s1_va_highbits	OUT	20	TLB 搜索虚拟页号
s1_asid	OUT	10	TLB 搜索地址空间标识
invtlb_valid	OUT	1	invtlb 指令有效
invtlb_op	OUT	5	invtlb 指令码
csr_asid_asid	IN	10	从 csr 读地址空间标识
csr_tlbehi_vppn	IN	19	从 csr 读虚拟页号
s1_found	IN	1	TLB 搜索到页表项
s1_index	IN	4	TLB 搜索到页表项索引
mem_csr_tlbrd	IN	1	MEM 流水级为 tlbrd 指令
wb_csr_tlbrd	IN	1	WB 流水级为 tlbrd 指令
ex_to_wb_rand	OUT	4	传到 WB 阶段的伪随机数
<b>MMU interface</b>			
data_va	OUT	32	load/store 指令虚拟地址
data_pa	IN	32	load/store 指令物理地址
ex_asid	OUT	10	地址空间标识
data_page_invalid	IN	1	页无效
data_ppi_except	IN	1	页特权等级不合规例外
data_page_fault	IN	1	页不匹配
data_page_clean	IN	1	页非脏

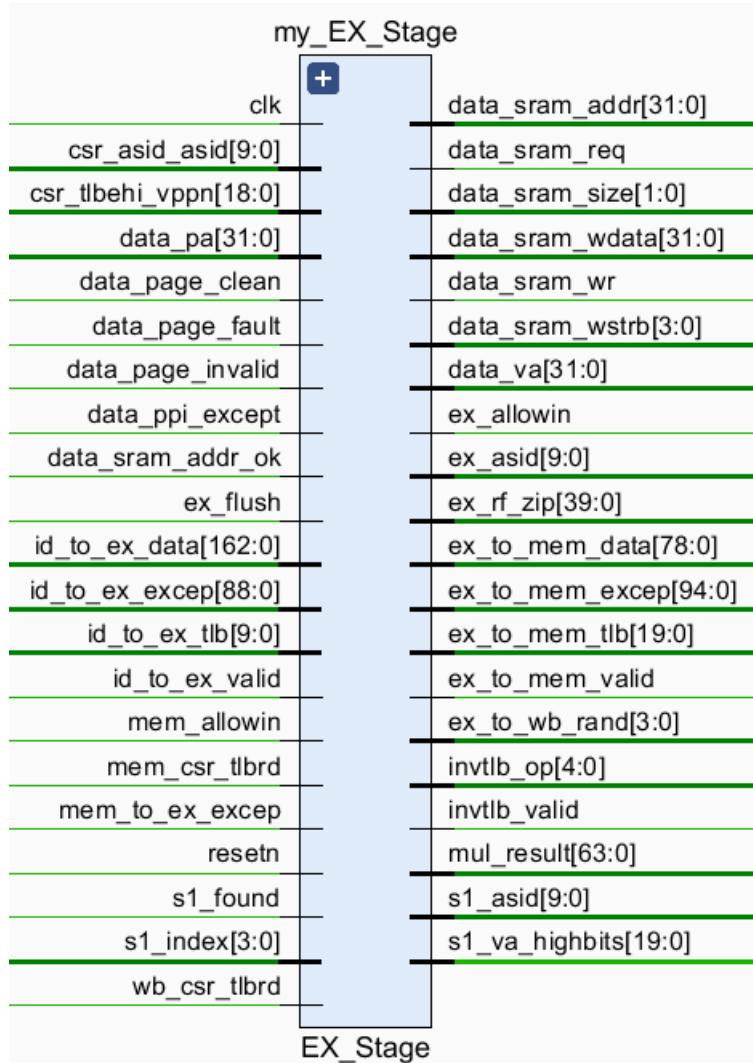


图 30 EX 流水级接口

### 3、功能描述

在之前实验的基础上，本实验在 EXE 级中增加了对虚地址 es\_alu\_result 进行地址转换的功能。和 IF 级类似，虚实地址转换通过 MMU 模块进行。此外，在本流水级还需要处理 TLB 相关例外，也主要结合指令类型和 MMU 返回信号进行判断。此外，如果在 load/store 指令中出现了访存虚地址错误或者 TLB 相关例外，则取消访存，不向总线发送请求。

另外，在 EX 流水级进行了 TLBSRCH 的查找 TLB 操作以及完成对 INVTLB 操作的处理。对于 TLBSRCH 指令，会用 ASID 或 TLBEHI 中的内容查询 TLB，并把返回的 index 和 found 信号传递给 WB 流水级进行进一步处理。对于 INVTLB 指令，则把其 op 传递给 TLB 模块，并把 invtlb\_valid 拉高，让 TLB 模块进行处理。这两条指令均会复用 EX 流水级对 TLB 的虚实地址转换的通路：

```

//----tlb-----
assign invtlb_op = ex_invtl_op;
assign invtlb_valid = ex_inst_invtlb & ex_valid;

//----mmu-----
assign data_va  = {32{ex_inst_tlbsrch}} & {csr_tlbehi_vppn, 13'b0} |
                  {32{ex_inst_invtlb}} & {ex_rkd_value} |
                  {32{~(ex_inst_invtlb | ex_inst_tlbsrch)}} & ex_alu_result;

assign ex_asid  = {10{~ex_inst_invtlb}} & csr_asid_asid |
                  {10{ex_inst_invtlb}} & ex_alu_src1[9:0];

```

图 31 EX 流水级对 TLB 查找的通路的复用

```

assign ex_data_ppi_excep = data_ppi_except && ex_mem_wait;
assign ex_data_tlbr_excep = data_page_fault && ex_mem_wait;
assign ex_data_pil_excep = data_page_invalid && ex_mem_wait && ~data_sram_wr;
assign ex_data_pis_excep = data_page_invalid && ex_mem_wait && data_sram_wr;
assign ex_data_pme_excep = data_page_clean && ex_mem_wait && data_sram_wr;

```

图 32 EX 流水级对 TLB 相关例外的判断

本实验对 ex\_ready\_go 也进行了相关修改，当 TLBSRCH 位于 EXE 级时，此时如果在 MEM 流水级或者 WB 流水级恰好有条修改 ASID 或 TLBEHI 的指令，或是 TLBRD 指令，就会引发围绕 csr 的数据相关。在这种情况下需要将 TLBSRCH 阻塞在 EXE 级，等前面的指令写入完成后再允许其流入 MEM 级。

## (八) 重要模块 7 设计：MEM 流水级

### 1、工作原理

新增传递给 EX 流水级的 TLB 阻塞信号。其余功能不变。

### 2、接口定义

表 8 MEM 流水级接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
resetn	IN	1	复位信号
mem_allowin	OUT	1	MEM 模块允许 EX 模块传入数据
ex_to_mem_data	IN	79	EX 模块传入 MEM 模块的数据
ex_to_mem_excep	IN	95	EX 模块传入 MEM 模块的异常信息
ex_to_mem_tlb	IN	20	EX 模块传入 MEM 模块的 TLB 信息
ex_to_mem_valid	IN	1	标记 EX 模块传入 MEM 模块的数据是否有效
wb_allowin	IN	1	WB 模块允许 MEM 模块传入数据
mem_to_wb_data	OUT	70	MEM 模块传入 WB 模块的数据
mem_to_wb_excep	OUT	127	MEM 模块传入 WB 模块的异常信息
mem_to_wb_tlb	OUT	20	MEM 模块传入 WB 模块的 TLB 信息
mem_to_wb_valid	OUT	1	标记 MEM 模块传入 WB 模块的数据是否有效
data_sram_data_ok	IN	1	数据 ram 数据握手信号

名称	方向	位宽	功能描述
data_sram_rdata	IN	32	数据 ram 读数据
mul_result	IN	64	乘法器运算结果
mem_rf_zip	OUT	38	MEM 模块前递到 ID 模块的数据
mem_reflush	IN	1	清空流水级信号
mem_to_ex_excep	OUT	1	MEM 级向 EX 级传递异常或中断信号

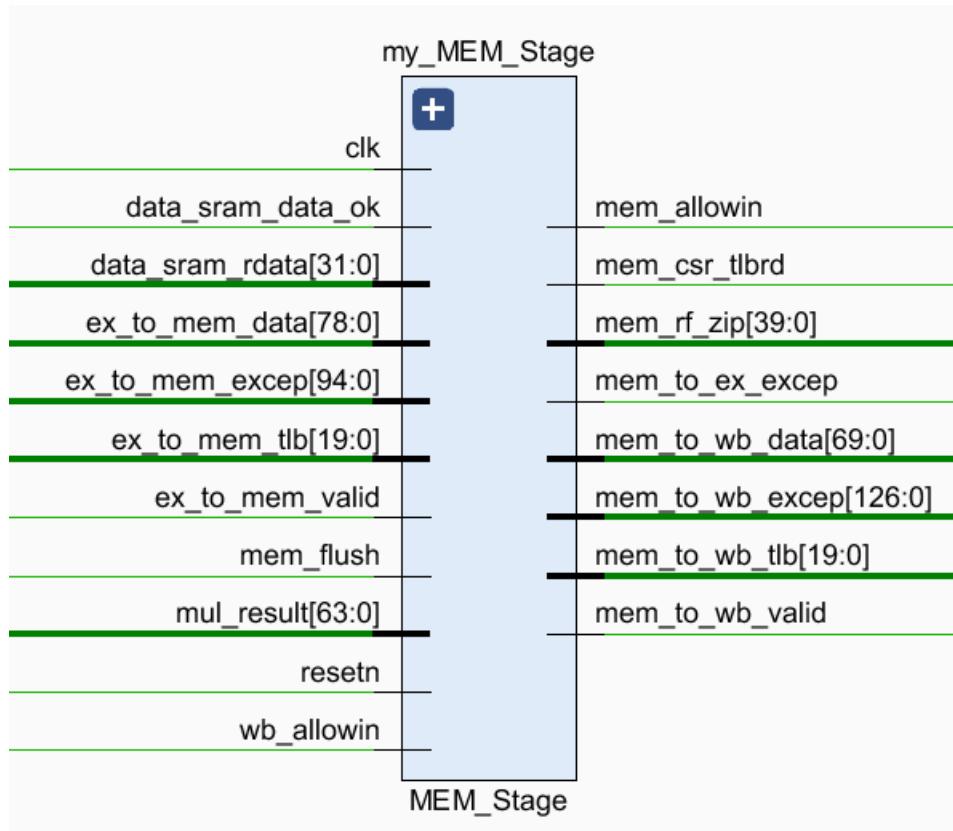


图 33 MEM 流水级接口

### 3、功能描述

新增传递给 EX 流水级的 TLB 阻塞信号，当 TLBSRCH 位于 EXE 级时，此时如果在 MEM 流水级恰好有条修改 ASID 或 TLBEHI 的指令，或是 TLBRD 指令，就会引发围绕 csr 的数据相关。在这种情况下需要将 TLBSRCH 阻塞在 EXE 级。

## (九) 重要模块 8 设计：WB 流水级

### 1、工作原理

本次实验中，TLB 指令在 WB 级完成对 csr 和 TLB 进行读写。此外，本次实验还新添了回滚操作，若需要回滚，把 WB 的 PC 传递给 IF，并清空流水线。除此之外，还新增了传递给 EX 级的 TLB 阻塞信号。其余功能不变。

### 2、接口定义

表 9 WB 流水级接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
resetn	IN	1	复位信号
<b>MEM interface</b>			
wb_allowin	OUT	1	WB 模块允许 MEM 模块传入数据
mem_to_wb_data	IN	70	MEM 模块传入 WB 模块的数据
mem_to_wb_excep	IN	127	MEM 模块传入 WB 模块的异常信息
mem_to_wb_tlb	IN	20	MEM 模块传入 WB 模块的 TLB 信息
mem_to_wb_valid	IN	1	标记 MEM 模块传入 WB 模块的数据是否有效
<b>Trace debug interface</b>			
debug_wb_pc	OUT	32	写回指令 PC 值 (用于 debug)
debug_wb_rf_we	OUT	4	写回指令写使能 (用于 debug)
debug_wb_rf_wnum	OUT	5	写回指令写地址 (用于 debug)
debug_wb_rf_wdata	OUT	32	写回指令写数据 (用于 debug)
<b>Forward data &amp; signal</b>			
wb_rf_zip	OUT	38	WB 模块向 ID 模块传递的 regfile 写回信息和前递数据
wb_to_if_csr_data	OUT	66	WB 模块传给 IF 模块的 csr 数据
wb_flush	OUT	1	WB 模块输出的清空流水线信号
has_int	OUT	1	将 ID 阶段的指令标记为中断
<b>CSR interface</b>			
wb_csr_num	OUT	14	控制状态寄存器读写地址
wb_csr_we	OUT	1	控制状态寄存器写使能
wb_csr_wmask	OUT	32	控制状态寄存器写掩码
wb_csr_wvalue	OUT	32	控制状态寄存器写数据
wb_ertn_flush_valid	OUT	1	ertn 指令刷新信号
wb_excep_valid	OUT	1	例外发生信号
wb_csr_ecode	OUT	6	例外发生 ecode 代码
wb_csr_esubcode	OUT	9	例外发生 esubcode 代码
wb_pc	OUT	31	WB 级 PC 值, 用于记录例外发生的 PC
csr_rvalue	IN	31	控制状态寄存器读数据
ex_entry	IN	31	例外跳转入口地址
ipi_int_in	OUT	1	核间中断输出
hw_int_in	OUT	8	硬件中断输出
coreid_in	OUT	32	核编号, 也是 TID 初值
wb_vaddr	OUT	32	出错地址
has_int	IN	1	发生中断信号
wb_csr_tlbrd	OUT	1	WB 流水级是 tlbrd 指令
csr_tlbidx_index	IN	4	CSR.TLBIDX.INDEX, TLB 表项索引
tlbrd_we	OUT	1	TLBRD 指令的写使能
tlbsrch_we	OUT	1	TLBSRCH 指令的写使能
tlbsrch_hit	OUT	1	TLBSRCH 查询命中
tlbsrch_hit_index	OUT	4	TLBSRCH 查询命中的索引
ex_to_wb_rand	IN	4	传到 WB 阶段的伪随机数
<b>TLB interface</b>			
r_index	OUT	4	TLB 读页表项索引
w_index	OUT	4	TLB 写页表项索引
tlb_we	OUT	1	TLB 写使能信号

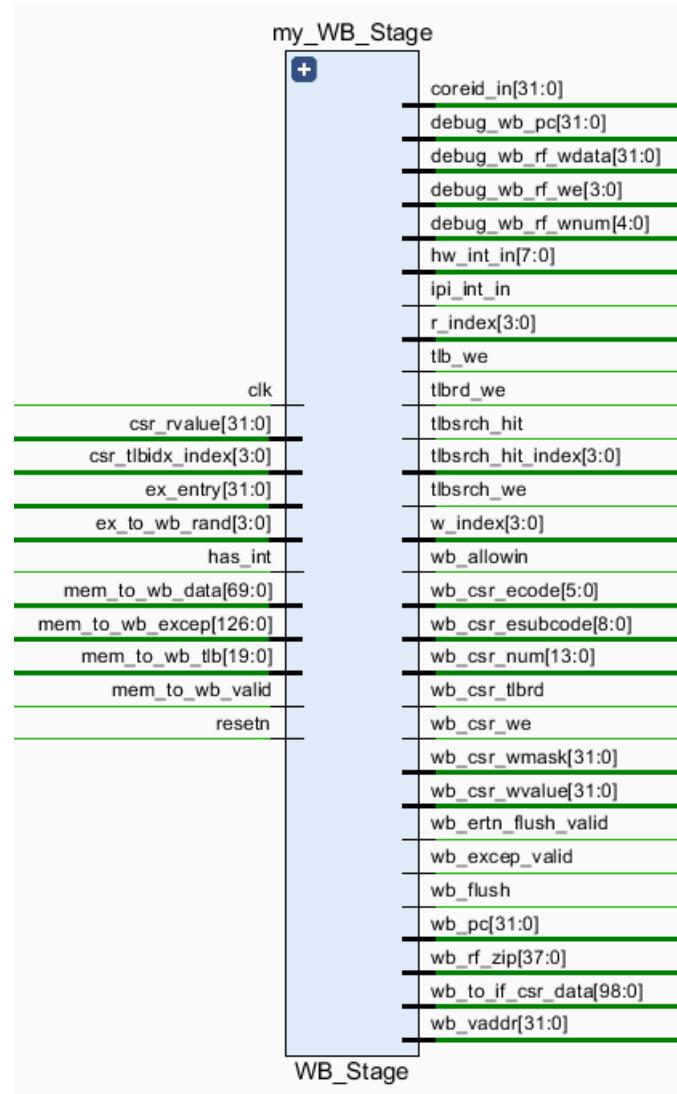


图 34 WB 流水级接口

### 3、功能描述

根据 TLB 指令的类型产生读写使能和索引，并传入 TLB。如果是 TLBSRCH 指令，则将在 EXE 级获得的查询结果写入 CSR。对于 TLBFILL 指令，其会随机选择一个 TLB 项进行写入，此时复用 EX 流水级的计数器的第四位作为写入的 TLB 项的索引，实现随机的效果。

```

// tlbrd
assign tlbrd_we = wb_inst_tlbrd;
assign r_index = csr_tlbidx_index;

// tlbwr and tlbfill
assign w_index = wb_inst_tlbwr ? csr_tlbidx_index : ex_to_wb_rand;
assign tlb_we = wb_inst_tlbwr | wb_inst_tlbfill;

// tlbsrch to csr
assign tlbsrch_we = wb_inst_tlbsrch;
assign tlbsrch_hit = wb_s1_found;
assign tlbsrch_hit_index = wb_s1_index;

```

图 35 WB 级 TLB 指令对 TLB 和 CSR 的读写

根据从上一级流入的重取标志和例外相关的控制信号来进行相应处理。如果重取标志为 1，则需要刷新流水级，为此将重取标志加入 ws\_reflush。如果发生 TLB 重填例外，例外入口地址的来源是从 csr 中读出的 ws\_tlb\_entry，其余情况下依然是 ws\_entry。

对于 TLBWR、TLBFILL、TLBRD、INVTLB 以及更改 ASID 或者 CRMD 寄存器的 csr 写指令，为了解决数据相关，本组成员的设计是刷新流水级并重新取指，直接复用在之前实验的 wb\_flush 信号即可。然后把 wb\_pc 传递给 IF 流水级，让的下一个取值 PC 改为此时 WB 流水级的 PC+4。

```

assign wb_tlbwr_valid      = wb_inst_tlbwr & wb_valid;
assign wb_tlbfill_valid    = wb_inst_tlbfill & wb_valid;
assign wb_tlbrd_valid     = wb_inst_tlbrd & wb_valid;
assign wb_invtlb_valid    = wb_inst_invtlb & wb_valid;
assign wb_csr_tlbwr        = ((wb_csr_num == `CSR_ASID || wb_csr_num == `CSR_CRMD)
                                && wb_csr_we) && wb_valid;

assign wb_tlb_refetch      = wb_tlbwr_valid | wb_tlbfill_valid | wb_tlbrd_valid | wb_invtlb_valid | wb_csr_tlbwr;

assign wb_flush = wb_ertn_flush_valid | wb_excep_valid | wb_tlb_refetch;

```

图 36 WB 级的回滚操作

除此之外，当 TLBSRCH 位于 EXE 级时，如果在 WB 流水级恰好有条修改 ASID 或 TLBEHI 的指令，或是 TLBRD 指令，就会引发围绕 csr 的数据相关。在这种情况下需要将 TLBSRCH 阻塞在 EXE 级。

WB 模块还会根据前面模块传递过来的异常类型信息为 csr\_ecode 和 csr\_esubcode 赋值。在该部分实验中，所有异常类型的 csr\_ecode 均为 0。根据指令集手册，对于该部分实验，当同时存在多个异常时，中断的优先级高于异常，中断的优先级最高。对于异常，取指阶段检测出的优先级最高，译码阶段检测出的优先级次之，执行阶段检测出的优先级再次之。而在 IF 级触发的例外中，ADEF 例外的优先级高于 TLB 相关例外；在 EX 流水级触发的例外中，ALE 例外的优先级高于 TLB 的相关例外；在 TLB 相关例外中，TLBR 的优先级最高，然后为 PIF 或者 PIL 或

者 PIS，再其次为 PPI，最后为 PME。因此使用如下代码逻辑进行赋值：

```
assign wb_csr_ecode = wb_has_int      ? `ECODE_INT :  
    wb_excp_adef     ? `ECODE_ADE :  
    wb_inst_tlbr_excep? `ECODE_TLBR:  
    wb_inst_pif_excep ? `ECODE_PIF :  
    wb_inst_ppi_excep ? `ECODE_PPI :  
    wb_excp_ine      ? `ECODE_INE :  
    wb_excp_syscall   ? `ECODE_SYS :  
    wb_excp_break     ? `ECODE_BRK :  
    wb_excp_ale       ? `ECODE_ALE :  
    wb_data_tlbr_excep? `ECODE_TLBR:  
    wb_data_pil_excep ? `ECODE_PIL :  
    wb_data_pis_excep ? `ECODE_PIS :  
    wb_data_ppi_excep ? `ECODE_PPI :  
    wb_data_pme_excep ? `ECODE_PME :  
    6'b0;  
assign wb_csr_esubcode = 9'b0;
```

图 37 csr\_ecode 和 csr\_esubcode 赋值

### 三、实验过程

#### (一) 实验流水账

李金明

2023.11.30 20: 00-20: 30 完成 exp17 的 debug

2023.12.5 21: 00-23: 00 完成 exp18 的设计

2023.12.5 23: 00-次日 1: 00 进行 exp18 的 debug

2023.12.6 10: 00-11: 00 完成 exp18 的 debug

2023.12.6 23: 30-次日 3: 00 完成 exp19 设计

2023.12.7 10: 00-12: 30, 14: 00-16: 30 完成 exp19 的 debug

2023.12.15 17: 30-22: 10 撰写实验报告

贾城昊

2023.11.30 14: 30-15: 00, 20: 00-20: 30 完成 exp17 的 debug

2023.12.4 13: 00-15: 00 进行 exp18 的设计

2023.12.5 12: 00-15: 00, 21: 00-23: 00 完成 exp18 的设计

2023.12.5 23: 00-次日 1: 00 进行 exp18 的 debug

2023.12.6 11: 00-14: 00 进行 exp19 的设计

2023.12.6 22: 30-次日 3: 00 完成 exp19 的设计

2023.12.15 13:30-17:00, 22: 30-23: 00 撰写实验报告

牛浩宇

2023.11.28-29 完成 exp17 的设计

2023.12.18 晚补充实验报告

## (二) 错误记录

### 1、错误 1：exp17 中 PS 字段赋值出错

#### (1) 错误现象

Console 报错如下：

```
=====
----FAIL!!!
read_test_id is 0
s0_test_id is 0
s1_test_id is 1
$finish called at time : 3845 ns : File "C:/Users/haohao/Desktop/exp17/mycpu_env/module_verify/tl"
```

图 38 错误 1 对应的 Console 报错

#### (2) 分析定位过程

查看波形图，查看 TLB 相关信号的值，发现 r\_ps 的值出现明显的错误，然后查看 r\_ps 的赋值逻辑，很快便定位到了问题所在。

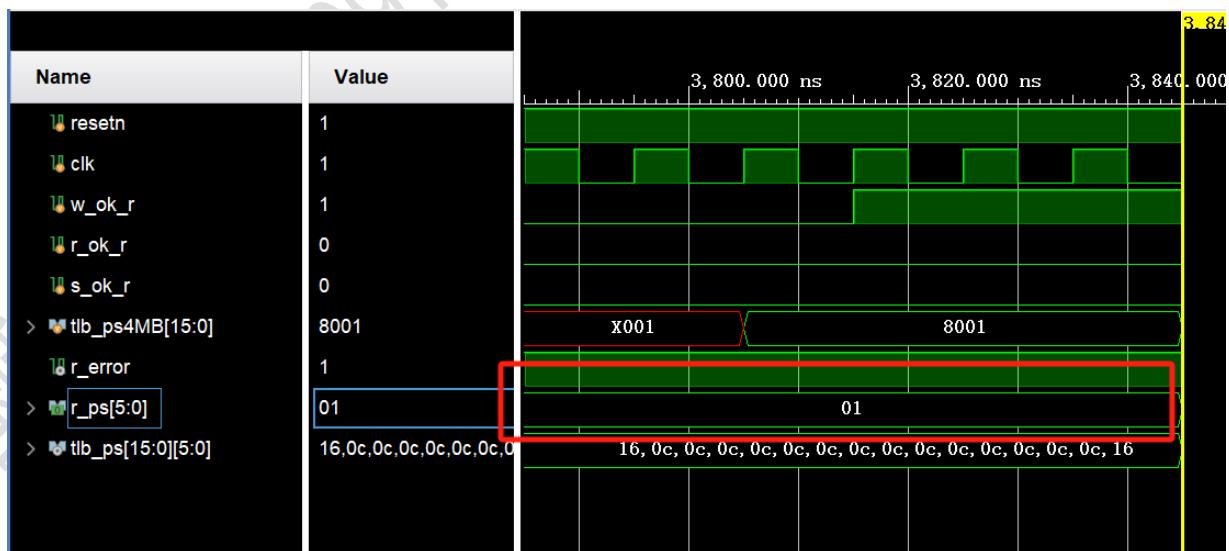


图 39 错误 1 对应的波形图

#### (3) 错误原因

TLB 模块中对 r\_ps 信号的赋值逻辑出错，其实本质原因还是因为本组写 exp17 的成员在写 r\_ps 的赋值逻辑时出现笔误，所以才导致了如下代码的产生：

```
// read  
assign r_e      = tlb_e      [r_index];  
assign r_vppn  = tlb_vppn  [r_index];  
assign r_ps    = tlb_ps4MB[r_index];  
assign r_asid  = tlb_asid  [r_index];  
assign r_g     = tlb_g     [r_index];
```

图 40 错误 1 中 r\_ps 赋值逻辑出错

#### (4) 修正效果

这是很明显的笔误，所以把 TLB 模块中对 r\_ps 信号的赋值逻辑进行修改即可，如下所示：

```
assign r_e      = tlb_e      [r_index];  
assign r_vppn  = tlb_vppn  [r_index];  
assign r_ps    = tlb_ps4MB[r_index] ? 6'd21 : 6'd12;  
assign r_asid  = tlb_asid  [r_index];  
assign r_g     = tlb_g     [r_index];
```

图 41 错误 1 对应的修改代码

该方法有效，exp17 通过。

## 2、错误 2：exp18 流水线回滚时 PC 更新出错

#### (1) 错误现象

Console 报错如下

```
[6122367 ns] Error!!!  
reference: PC = 0x1c071dc0, wb_rf_wnum = 0x1e, wb_rf_wdata = 0x00010000  
mycpu     : PC = 0x1c071dc4, wb_rf_wnum = 0x1e, wb_rf_wdata = 0x00010000
```

图 42 错误 2 对应的 Console 报错

#### (2) 分析定位过程

从 console 报错来看，PC 出错但寄存器写是对的，按照以往的经验，很可能是 PC 的更新或者指令的取消的问

题，但不管怎么说，都得首先查看波形图，如下所示：

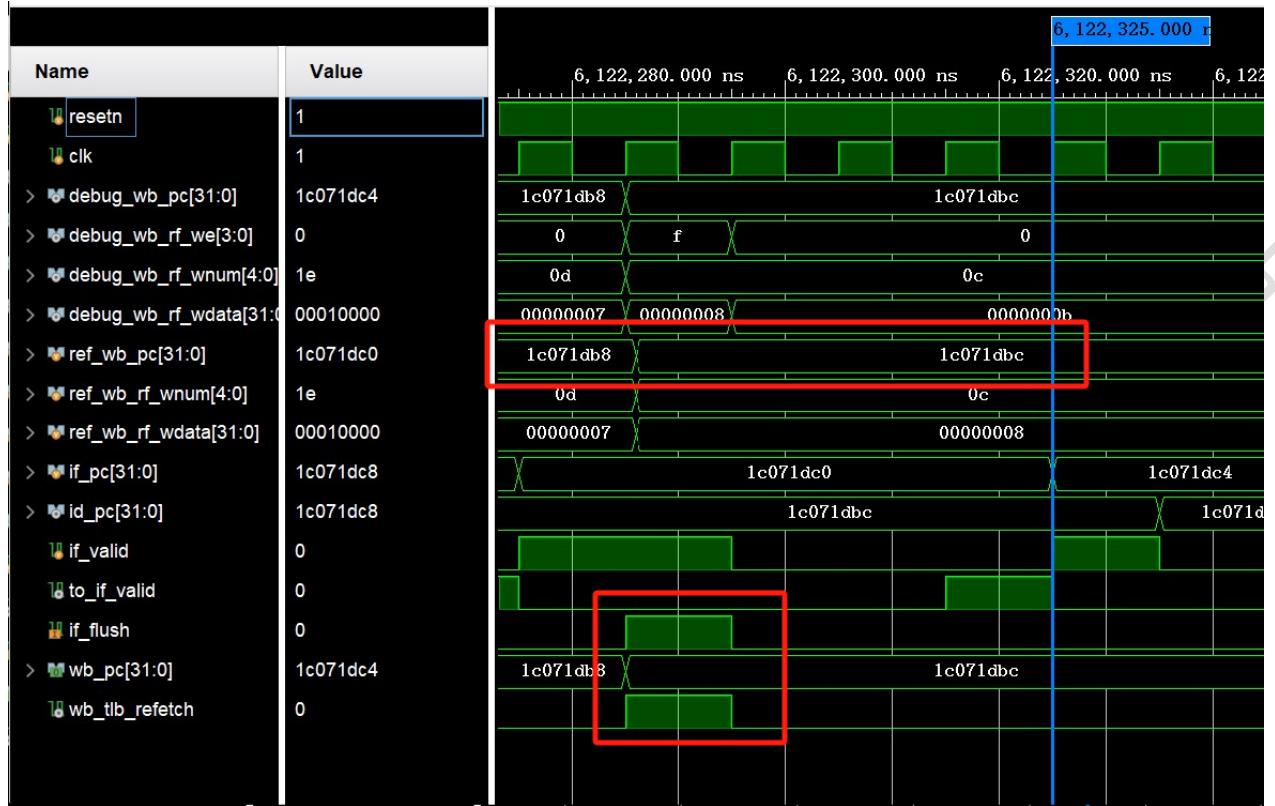


图 43 错误 2 对应的波形图

通过一波向前溯源的操作，发现出现问题的地方在于，当 tlb\_refetch 信号拉高时，本来应该清空流水线，然后重新进行取值，这个时候从理论上来说，应该 PC 会变为 WB 对应指令的下一条指令的 PC，但通过波形图可以看出，此时 PC 仅仅进行了加 4 的操作，而没有回滚到对应的指令的 PC 值，从而导致出错。

其实分析到这后，已经可以确定是设计的问题了（其实写之前本组成员设计时，已经考虑了这个问题，但最终由于工作量比较多，导致 debug 前这个地方忘记编写相关代码了）于是，本组成员查看了 IF 流水级的 PC 更新的代码，果然发现 WB 没有把其 PC 值传过来，而且也没有对应的选择逻辑，所以 PC 进行了默认的加 4 操作，从而导致出错。

### (3) 错误原因

对 TLB 指令进行回滚操作时，需要将 WB 的 PC 传递给 IF 流水级，并把相关控制信号传递给 IF，方便 IF 流水级 nextpc 信号进行选择，但本组成员由于疏忽，debug 前忘记编写与之相关的代码，所以导致出错，出错前 IF 的 nextpc 更新逻辑如下所示：。

```

assign seq_pc      = if_pc + 3'h4;
assign nextpc     = wb_ertn_flush_valid_reg ? csr_rvalue_reg
                  : wb_ertn_flush_valid ? csr_rvalue
                  : wb_csr_ex_valid_reg ? ex_entry_reg
                  : wb_csr_ex_valid ? ex_entry
                  : br_taken_reg ? br_target_reg
                  : br_taken ? br_target
                  : seq_pc;

```

图 44 错误 2 修改前代码(nextpc 赋值逻辑)

#### (4) 修正效果

首先，新增将 WB 的 PC 传递给 IF 流水级的数据通路，并把相关控制信号传递给 IF，方便 IF 流水级 nextpc 信号进行选择。然后，在 IF 流水级新增寄存器存储 WB 的 PC 值和其控制信号，方便 nextpc 进行选择。最后，更改 nextpc 更新逻辑，如下所示：

```

assign seq_pc      = if_pc + 3'h4;
assign nextpc     = wb_ertn_flush_valid_reg ? csr_rvalue_reg
                  : wb_ertn_flush_valid ? csr_rvalue
                  : wb_csr_ex_valid_reg ? ex_entry_reg
                  : wb_csr_ex_valid ? ex_entry
                  : wb_tlb_refetch_valid_reg ? wb_pc_reg + 32'h4
                  : wb_tlb_refetch_valid ? wb_pc + 32'h4
                  : br_taken_reg ? br_target_reg
                  : br_taken ? br_target
                  : seq_pc;

```

图 45 错误 2 修改后代码(nextpc 赋值逻辑)

该方法有效，来到下一个 bug。

### 3、错误 3：exp18 中 WB 级对 MEM 传递来的 TLB 信号数据通路解码出错

#### (1) 错误现象

Console 报错如下

```

[7081677 ns] Error!!!
reference: PC = 0x1c078b34, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x3c00003c
mycpu    : PC = 0x1c078b34, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x3c00003b

```

图 46 错误 3 对应的 Console 报错

## (2) 分析定位过程

这个错误的定位过程花费较久，因为最终出错位置相比于报错位置较远。起初本组成员根据汇编代码，对汇编的预期行为进行判断，最后花费了较长时间仍然没有找到错误所在。于是，本组成员转换思路，考虑到出错在第 60 个测试点，其对应汇编文件如下：

```
1c078a48 <n60_tlbfill_test>:  
n60_tlbfill_test():  
1c078a48: 157f5fec    lu12i.w $r12,-263425(0xbfaaff)  
1c078a4c: 03bcc18c    ori $r12,$r12,0xf30  
1c078a50: 157f5f0d    lu12i.w $r13,-263432(0xbfaf8)  
1c078a54: 03bff1ad    ori $r13,$r13,0ffc  
1c078a58: 298001a0    st.w   $r0,$r13,0  
1c078a5c: 298001a0    st.w   $r0,$r13,0  
1c078a60: 29800180    st.w   $r0,$r12,0  
1c078a64: 298001a0    st.w   $r0,$r13,0  
1c078a68: 298001a0    st.w   $r0,$r13,0  
1c078a6c: 288001a0    ld.w   $r0,$r13,0  
1c078a70: 2880018c    ld.w   $r12,$r12,0  
1c078a74: 028006f7    addi.w $r23,$r23,1(0x1)  
1c078a78: 00150019    move   $r25,$r0  
1c078a7c: 1418000c    lu12i.w $r12,49152(0xc000)  
1c078a80: 1402af0d    lu12i.w $r13,5496(0x1578)  
1c078a84: 141f578e    lu12i.w $r14,64188(0xfabc)  
1c078a88: 038105ce    ori $r14,$r14,0x41  
1c078a8c: 141ff78f    lu12i.w $r15,65468(0xffbc)  
1c078a90: 038105ef    ori $r15,$r15,0x41  
1c078a94: 0400402c    csrwr  $r12,0x10  
1c078a98: 0400442d    csrwr  $r13,0x11  
1c078a9c: 0400482e    csrwr  $r14,0x12  
1c078aa0: 04004c2f    csrwr  $r15,0x13  
1c078aa4: 06483400    tlbfill  
1c078aa8: 06483400    tlbfill  
1c078aac: 06483400    tlbfill  
1c078ab0: 06483400    tlbfill
```

图 47 错误 3 的出错测试点汇编代码

通过汇编代码可知，出错的测试点是为了测试 tlbfill 指令，而在此测试点中出现的新指令只有 tlbfill，其它指令通过之前的测试可知其处理是没有问题的。于是问题的焦点便成为了 tlbfill 的前后一段指令，查看这段指令对应的波形图，观察相关信号，很快便发现了出现问题的信号，如下所示：



图 48 错误 3 对应的波形图

通过波形图，可以发现，此时传递给 TLB 模块的 we, tlb\_we 信号以及传递的 inst\_tlbfill 信号均为 X，这显然是不对的。于是查看 MEM 到 WB 的数据通路以及解码逻辑，果然发现了问题所在：

```
assign {wb_s1_found, wb_s1_index, wb_invtl_op
      wb_inst_tlbfill, wb_inst_tlbwr, wb_inst_tlbfill, wb_inst_tlbrd, wb_inst_invtlb} = mem_to_wb_tlb_reg;
```

图 49 错误 3 对应的错误代码

### (3) 错误原因

WB 流水级对 MEM 流水级传递过来的 TLB 相关信号的解码出错：由于笔误，导致本应是 wb\_inst\_tlbsrch 的位置写成了 wb\_inst\_tlbfill，所以导致 WB 解码出的 wb\_inst\_tlbfill 信号值为 X，导致 tlbfill 的操作出现错误，从而影响到了后面的指令的行为。

### (4) 修正效果

将笔误修改过来，即把第一个 wb\_inst\_tlbfill 改为 wb\_inst\_tlbsrch 即可，如下所示：

```
assign {wb_s1_found, wb_s1_index, wb_invtl_op
      wb_inst_tlbsrch, wb_inst_tlbwr, wb_inst_tlbfill, wb_inst_tlbfill, wb_inst_tlbrd, wb_inst_invtlb} = mem_to_wb_tlb_reg;
```

图 50 错误 3 修改后的代码

该方法有效，来到下一处 bug。

## 4、错误 4：exp18 中未考虑 invtlb 指令 op 值错误的情况

### (1) 错误现象

运行仿真后，console 报错如下：

```
[7305887 ns] Error!!!
reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
mycpu    : PC = 0x1c079110, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00030000
```

图 51 错误 4 对应的 Console 报错

### (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c07910c: 06498007  invtlb 0x7, $r0, $r0
1c079110: 1400060d  lwl2i.w $r13, 48(0x30)
```

图 52 错误 4 对应的汇编指令

显而易见，地址 0x1c07910c 处的 invtlb 指令的 op 值为 0x7，这与要求不符：invtlb 指令的 op 值只能为 0x0-0x6，应该进入异常，从错误的 PC 值也可以看出，我们设计的 CPU 进行的是 invtlb 指令的下一条指令，而正常应该进入异常处理程序。

### (3) 错误原因

当 invtlb 指令的 op 值不合法时，按照指令集手册所述，应当进入保留异常，但我们当时不太确定触发什么异常，因此没有写这个地方。后来经过思考并询问助教老师确认后，决定当 invtlb 指令的 op 值不合法时，触发 INE 指令不存在异常。

### (4) 修正效果

在 ID 模块中更改 INE 异常的触发条件即可，如下所示：

```
assign invtlb_op_fault = rd[4] | rd[3] | (&rd[2:0]);
assign id_excp_ine   = (~ (type_calc | type_calc_i | type_branch_uncond | type_branch_cond |
                           type_load | type_store | type_excp | type_tlb | type_others) |
                           inst_invtlb & invtlb_op_fault) & id_valid;
```

图 53 错误 10 修改后代码

该方法有效，exp18 通过。

## 5、错误 5：exp19 中 EX 模块传到 mmu 的 va 出错

### (1) 错误现象

运行仿真后，console 报错如下：

```
[7320857 ns] Error!!!
reference: PC = 0x1c07b878, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
mycpu    : PC = 0x1c07b878, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x80000000
```

图 54 错误 5 对应的 Console 报错

## (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c07b874: 06482800 tlbsrch
1c07b878: 0400400e csrrd $r14, 0x10
```

图 55 错误 5 对应的汇编指令

可以发现，该部分的功能是首先执行 tlbsrch 指令，tlbsrch 指令会将查找结果写入 CSR.TLBIDX，然后通过 csrrd 指令将该 csr 寄存器中的值读到寄存器中，从而与 golden trace 比较，确定 tlbsrch 执行的结果是否正确。

Golden trace 的结果是 0x00000000，而我们设计输出的是 0x80000000，说明 tlbsrch 本应有命中项，但实际执行的过程中未命中。由于 exp17 和 exp18 已经对 TLB 模块有了充分验证，我们认为是 TLB 模块内部工作出错，而将视角转向参数传递的过程。查看相应的波形图：

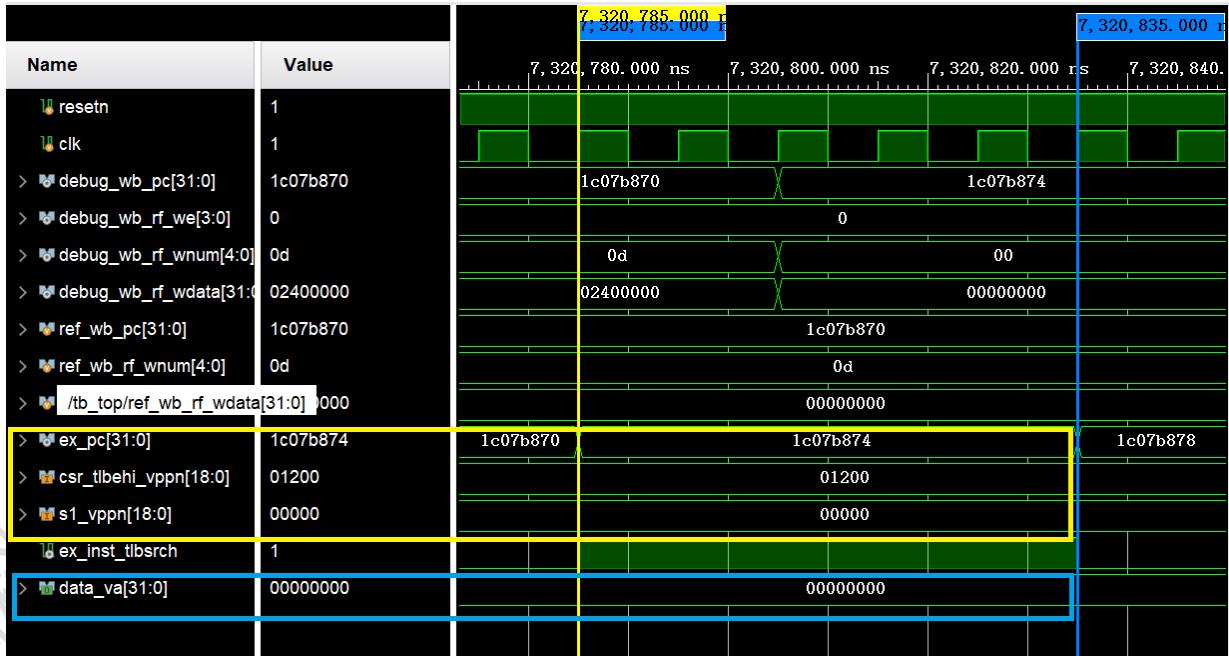


图 56 错误 5 对应的波形图

首先观察黄框中的信号，可以看到，当 EX 模块处理的指令对应地址为 0x1c07b874 时，代表当前指令是 tlbsrch 的信号正常拉高了。但是从 CSR.TLBEHI 中读出的数据是 0x01200，而传递到 TLB 的 vppn 值是 0x00000，二者不匹配。这一信号的传递过程是：EX 模块先得到 CSR.TLBEHI 中的数据，从而确定传递到 MMU 的虚拟地址，再由 MMU 解析，将 vppn 部分传给 TLB 模块。我们认为错误大概率出现在 EX 模块中，调取 EX 模块传递到

MMU 的 va 信号（蓝框部分），可以发现其错误地为 0。查看相关代码：

```
assign data_va = {20{ex_inst_tlbsrch}} & {csr_tlbehi_vppn, 13'b0} |
    {20{ex_inst_invtlb}} & {ex_rkd_value} |
    {20{~(ex_inst_invtlb | ex_inst_tlbsrch)}} & ex_alu_result;
```

图 57 错误 5 中 data\_va 的赋值

### (3) 错误原因

在我们的设计中，当 EX 模块处理虚拟地址时，一开始是将地址的 31 到 12 位传递给 MMU 供其解析，后来决定将整个虚拟地址都传递过去，更改逻辑时忘记因为 data\_va 是 32 位的，因此要将这一部分代码中的 20 改为 32，从而导致出错。

### (4) 修正效果

把 EX 流水级传递给 MMU 的 data\_va 信号赋值逻辑中的 20 改为 32 即可，如下所示：

```
assign data_va = {32{ex_inst_tlbsrch}} & {csr_tlbehi_vppn, 13'b0} |
    {32{ex_inst_invtlb}} & {ex_rkd_value} |
    {32{~(ex_inst_invtlb | ex_inst_tlbsrch)}} & ex_alu_result;
```

图 58 错误 5 修改代码

该方法有效，来到下一个 bug。

## 6、错误 6：exp19 中 csr 模块修改后读 crmd 值出错

### (1) 错误现象

运行仿真后，console 报错如下：

```
[7342527 ns] Error!!!
reference: PC = 0x1c07be14, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x000000088
mycpu      : PC = 0x1c07be14, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x0000000a8
```

图 59 错误 6 对应的 Console 报错

### (2) 分析定位过程

首先查看报错位置的汇编代码：

111401	1c07be08:	040001ac	csrxchg \$r12,\$r13,0x0
111402	1c07be0c:	03804c0c	ori \$r12,\$r0,0x13
111403	1c07be10:	03807c0d	ori \$r13,\$r0,0x1f
111404	1c07be14:	040001ac	csrxchg \$r12,\$r13,0x0

图 60 错误 6 对应的汇编指令

可以发现，该部分的功能是执行 csrxchg 指令，查看的寄存器是 CSR.CRMD 中的值。0x1c07be08 处的 csrxchg 指令将寄存器的内容写入 CSR.CRMD，并通过把与 golden trace 比对判断 CRMD 中的旧值是否正确，0x1c07be14 处的 csrxchg 指令也是如此。考虑到 csrxchg 指令的实现应该不会有问题，不会是后一条 csrxchg 指令读数据错误，同时考虑到该部分汇编代码的工作逻辑，问题大概率出现在 0x1c07be08 处的 csrxchg 指令向 CRMD 写回的过程中。

查看相关波形图：

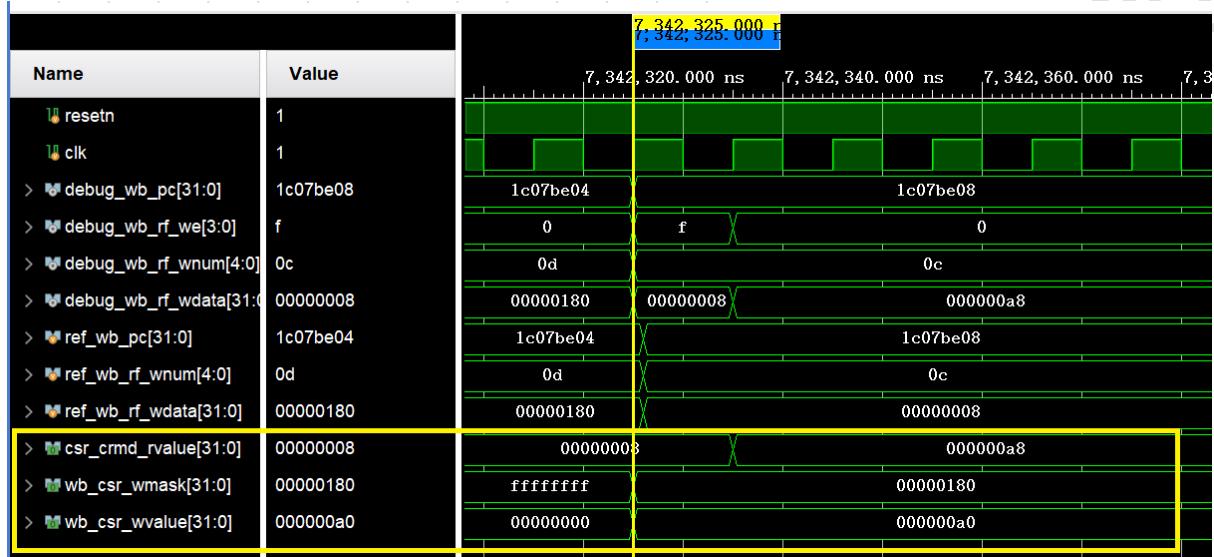


图 61 错误 6 对应的波形图

可以看到，当 debug\_wb\_pc 为 0x1c07be08 时，第一条 csrxchg 指令会向 CRMD 寄存器写回数据。传递给 CSR 模块的写数据 wb\_csr\_wvalue 信号是 0x000000a0，掩码信号 wb\_csr\_wvalue 是 0x000000180，而查看 CSR 模块中的 csr\_crmd\_rvalue 信号，过了一个周期 CRMD 中的数据更新后，其值变为了 0x000000a8，与预期的 0x00000088 不符。故在 CSR 模块中查看 CRMD 寄存器写数据和读数据赋值的相关代码：

```

else if (csr_we && csr_num == `CSR_CRMD) begin
    csr_crmd_da <= csr_wmask[`CSR_CRMD_DA] & csr_wvalue[`CSR_CRMD_DA] |
        ~csr_wmask[`CSR_CRMD_DA] & csr_crmd_da;
    csr_crmd_pg <= csr_wmask[`CSR_CRMD_PG] & csr_wvalue[`CSR_CRMD_PG] |
        ~csr_wmask[`CSR_CRMD_PG] & csr_crmd_pg;
    csr_crmd_datf <= csr_wmask[`CSR_CRMD_DATF] & csr_wvalue[`CSR_CRMD_DATF] |
        ~csr_wmask[`CSR_CRMD_DATF] & csr_crmd_datf;
    csr_crmd_datm <= csr_wmask[`CSR_CRMD_DATM] & csr_wvalue[`CSR_CRMD_DATM] |
        ~csr_wmask[`CSR_CRMD_DATM] & csr_crmd_datm;
end

```

图 62 错误 6 中 CSR.CRMD 写数据的更新逻辑

```
assign csr_crmd_rvalue = {23'b0, csr_crmd_datm, csr_crmd_datm, csr_crmd_pg, csr_crmd_da, csr_crmd_ie, csr_crmd_plv};
```

图 63 错误 6 中 CSR.CRMD 读数据的赋值

### (3) 错误原因

可以发现，写回的过程没有什么问题，问题出现在读数据的赋值。在 CRMD 中，第 5 到 6 位为 DATF，7 到 8 位为 DATM，而我们的代码中错误地将第 5 到 6 位也赋值为了 DATM 部分，导致出错。这一部分代码在以前实验中就存在问题，但是之前不涉及地址翻译模式，不论是 DATF 还是 DATM 都是 0，故不会发生错误。

### (4) 修正效果

把 csr\_crmd\_rvalue 信号赋值逻辑修改正确即可，如下所示：

```
assign csr_crmd_rvalue = {23'b0, csr_crmd_datm, csr_crmd_datf, csr_crmd_pg, csr_crmd_da, csr_crmd_ie, csr_crmd_plv};
```

图 64 错误 6 修改后代码

该方法有效，来到下一个 bug。

## 7、错误 7：exp19 中未配置 tlb 异常入口

### (1) 错误现象

运行仿真后，console 报错如下：

```
[7343087 ns] Error!!!
reference: PC = 0x1c00f000, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000000
mycpu    : PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
```

图 65 错误 7 对应的 Console 报错

### (2) 分析定位过程

由于 0x1c008000 是例外入口地址，我们可以直接通过波形图查看触发这一例外的指令的地址：

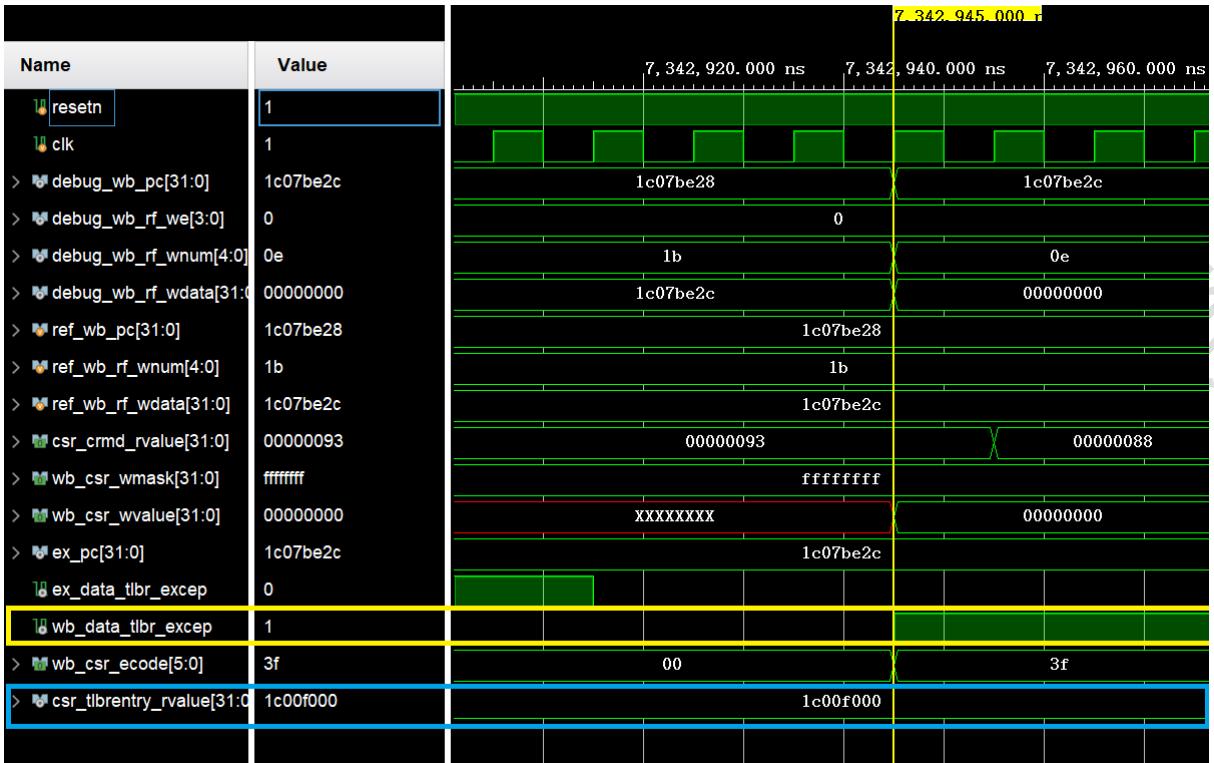


图 66 错误 7 对应的波形图

可以看到，这一例外的类型是 TLB 重填例外。根据指令集手册，当触发 TLB 重填例外时，例外入口应当是 CSR.TLBENTRY 中的值，即 0x1c00f000，但是 CPU 并没有按照预想的跳转到这一地址。查看 CPU 中处理例外时例外入口地址的赋值的相关代码：

```
assign ex_entry = csr_eentry_rvalue;
```

图 67 错误 7 中例外入口地址的赋值

### (3) 错误原因

可以发现，在例外入口地址 `ex_entry` 的赋值中，我们并没有考虑 TLB 重填例外的情况。根据指令集手册，EENTRY 用于配置除 TLB 重填例外之外的例外和中断的入口地址，当触发 TLB 重填时，例外入口地址由 TLBENTRY 寄存器配置。

### (4) 修正效果

把例外入口地址的赋值逻辑修改正确即可，如下所示：

```
assign ex_entry = {32{~tlb_tlbr_excep}} & csr_eentry_rvalue |
    {32{tlb_tlbr_excep}} & csr_tlbrentry_rvalue;
```

图 68 错误 7 修改后代码

该方法有效，来到下一个 bug。

## 8、错误 8：exp19 中 tlbehi 寄存器在触发 tlb 异常时未更新

### (1) 错误现象

运行仿真后，console 报错如下：

```
[7344037 ns] Error!!!
reference: PC = 0x1c00f04c, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x401fe000
mycpu    : PC = 0x1c00f04c, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x1c08a000
```

图 69 错误 8 对应的 Console 报错

### (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c00f04c: 0400440c csrrd $r12, 0x11
```

图 70 错误 8 对应的汇编指令

该条汇编指令读取的是 CSR.TLBEHI，通过指令地址可以看出，当前指令处于 TLB 重填例外的处理阶段，在这部分中没有对 CSR.TLBEHI 没有修改。查看 CSR.TLBEHI 的读写条件：

位	名字	读写	描述
31:13	VPPN	RW	执行 TLBRD 指令时，所读取 TLB 表项的 VPPN 域的值记录到这里。 执行 TLBSRCH 指令时查询 TLB 所用 VPPN 值，以及执行 TLBWR 和 TLBFILL 指令时写入 TLB 表项的 VPPN 域的值来自于此。 当触发 TLB 重填例外、load 操作页无效例外、store 操作页无效例外、取指操作页无效例外、页写允许例外和页特权等级不合规例外时，触发例外的虚地址的[31:13]位被记录到这里。

图 71 CSR.TLBEHI 的读写条件

可以发现，当发生 TLB 相关例外时，要将触发例外的虚地址的 31 到 13 位写到 TLBEHI 寄存器的 VPPN 部分。

根据波形图查找到触发例外的指令对应的地址。查看相关波形图：

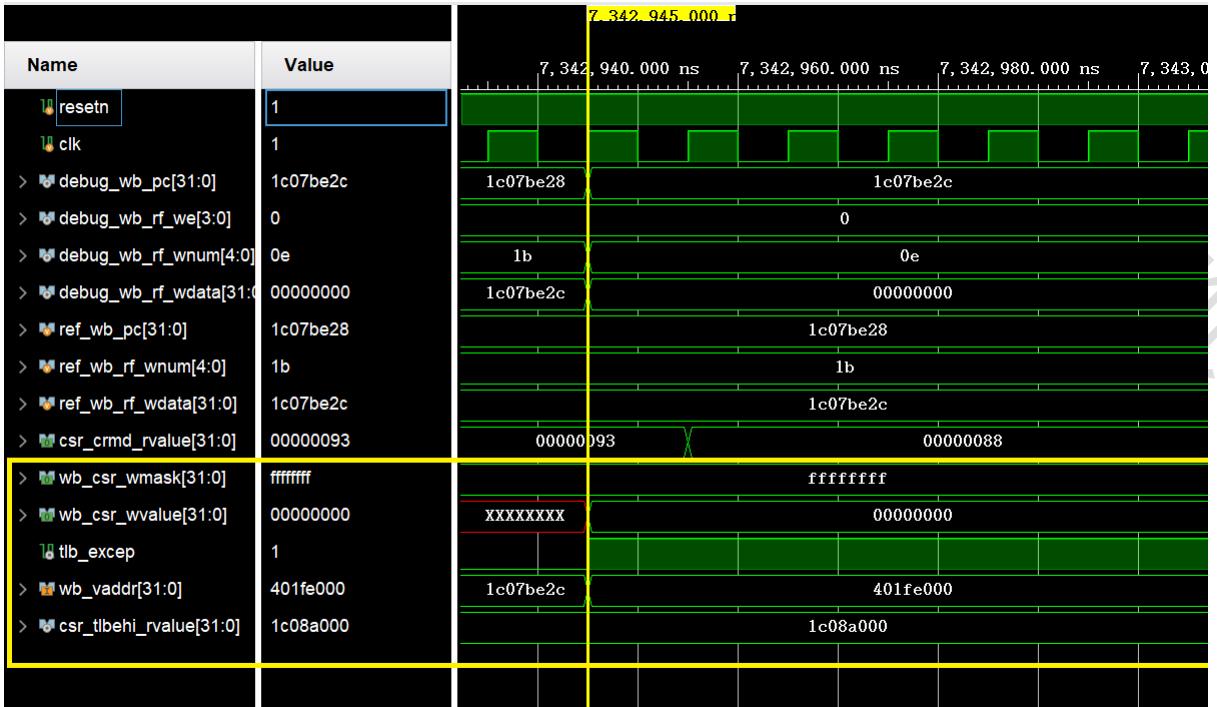


图 72 错误 8 对应的波形图

可以看到，当触发 TLB 例外时，应当将 wb\_vaddr 的 31 到 13 位写入 TLBEHI 寄存器的 VPPN 部分，但是此时 TLBEHI 寄存器的值并没有发生变化。故在 CSR 模块中查看 TLBEHI 寄存器写数据的相关代码：

```
// TLBEHI
always @ (posedge clk) begin
    if (~resetn) begin
        csr_tlbehi_vppn <= 19'b0;
    end
    else if (tlbrd_we) begin
        if(r_tlb_e)
            csr_tlbehi_vppn <= r_tlb_vppn;
        else
            csr_tlbehi_vppn <= 19'b0;
    end
    else if (csr_we && csr_num == `CSR_TLBEHI) begin
        csr_tlbehi_vppn <= csr_wmask[`CSR_TLBEHI_VPPN] & csr_wvalue[`CSR_TLBEHI_VPPN] |
                           ~csr_wmask[`CSR_TLBEHI_VPPN] & csr_tlbehi_vppn;
    end
end
```

图 73 错误 8 中 CSR.TLBEHI 的数据更新逻辑

### (3) 错误原因

可以发现，在 TLBEHI 的更新条件中，并没有考虑有 TLB 例外的情况，这就导致了 TLBEHI 寄存器储存值与预期不符，应当在触发 TLB 相关异常时更新 TLBEHI 的值。

#### (4) 修正效果

把 CSR.TLBEHI 的数据更新逻辑修改正确即可，如下所示：

```
411      always @ (posedge clk) begin
412          if (~resetn) begin
413              csr_tlbehi_vppn <= 19'b0;
414          end
415          else if (tlbrd_we) begin
416              if(r_tlb_e)
417                  csr_tlbehi_vppn <= r_tlb_vppn;
418              else
419                  csr_tlbehi_vppn <= 19'b0;
420          end
421          else if (tlb_excep) begin
422              csr_tlbehi_vppn <= wb_vaddr[31:13];
423          end
424          else if (csr_we && csr_num == `CSR_TLBEHI) begin
425              csr_tlbehi_vppn <= csr_wmask[`CSR_TLBEHI_VPPN] & csr_wvalue[`CSR_TLBEHI_VPPN] |
426                                         ~csr_wmask[`CSR_TLBEHI_VPPN] & csr_tlbehi_vppn;
427          end
428      end
```

图 74 错误 8 修改后代码

该方法有效，来到下一个 bug。

### 9、错误 9：exp19 中 badv 寄存器赋值中混用逻辑运算符导致运算误判优先级

#### (1) 错误现象

运行仿真后，console 报错如下：

```
[7384587 ns] Error!!!
reference: PC = 0x1c0084f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x401fe000
mycpu    : PC = 0x1c0084f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x47000000
```

图 75 错误 9 对应的 Console 报错

#### (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c0084f8: 04001c0c csrrd $r12, 0x7
```

图 76 错误 9 对应的汇编指令

可以发现，出错的原因是读取 BADV 寄存器中的值与预期不符。由于 0x1c008000 是例外入口地址，我们可以直接通过波形图查看触发这一例外的指令的地址：

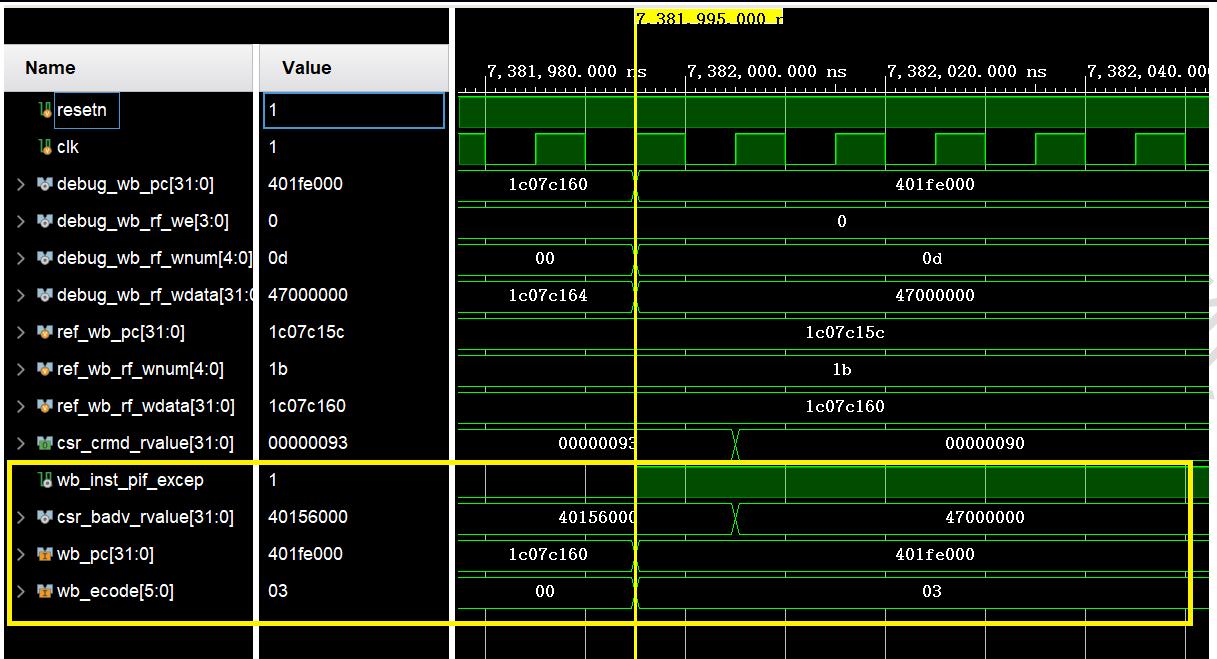


图 77 错误 9 对应的波形图

可以发现，这一例外的类型是取值操作页无效例外。根据指令集手册，BADV 用于触发地址错误相关例外时记录出错的虚地址。对于 PIF 例外，记录的应该是发送的 PC 值。根据波形图，此时除法例外的地址应当为 0x401fe000，其应当被吸入 BADV 寄存器，可是实际上写入 BADV 寄存器的值为 0x47000000。因此可以推断出 BADV 的数据更新逻辑出错，查看相关代码：

```
always @(posedge clk) begin
    if (wb_ex && wb_ex_addr_err)
        csr_badv_vaddr <= (wb_ecode == `ECODE_ADE && wb_esubcode == `ESUBCODE_ADEF |
                            wb_ecode == `ECODE_PIF) ? wb_pc : wb_vaddr;
end
```

图 78 错误 9 中 BADV 寄存器的数据更新逻辑

### (3) 错误原因

对于 BADV 寄存器，当触发 TLB 相关例外时，按照设计，当触发 ADEF 或 PIF 例外时，错误地址是当前的 PC 值，其他情况中出错地址会被存储到 wb\_vaddr 中。需要指出的是，TLBR 例外、页特权等级不合规例外的出错地址也有可能是 PC 值，但也有可能是 EX 模块中发出的地址请求，故处理这两个例外时，PC 值被存储到了 wb\_vaddr 中。回到错误代码，此处代码看起来没有错，实际上犯了一个比较低级的错误：混用了两种逻辑运算符。在 verilog 语法中，| 的运算优先级要高于 &&。当前例外是 ADEF 时，这种判断逻辑能正常运行；但是当前例外是 PIF 时，由于不可能做到 ecode 既为`ECODE\_PID 也为`ECODE\_ADE，从而导致条件为错，从而写入 BADV 的数据时 wb\_vaddr 而非 wb\_pc。

#### (4) 修正效果

把 BADV 的数据更新时写入数据的判断条件修改正确即可，如下所示：

```
always @(posedge clk) begin
    if (wb_ex && wb_ex_addr_err)
        csr_badv_vaddr <= ((wb_ecode == `ECODE_ADE & wb_esubcode == `ESUBCODE_ADEF) |
                            wb_ecode == `ECODE_PIF) ? wb_pc : wb_vaddr;
end
```

图 79 错误 9 修改后代码

该方法有效，来到下一个 bug。

### 10、错误 10：exp19 中 tlbehi 在 pif 异常中赋值出错

#### (1) 错误现象

运行仿真后，console 报错如下：

```
[7384687 ns] Error!!!
reference: PC = 0x1c008500, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x401fe000
mycpu     : PC = 0x1c008500, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x47000000
```

图 80 错误 10 对应的 Console 报错

#### (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c008500: 0400440c csrrd $r12,0x11
```

图 81 错误 10 对应的汇编指令

从这里，我们可以发现错误 10 不止有一点眼熟。一方面，0x11 对应的寄存器是 TLBEHI，这与我们错误 8 中出错的 CSR 寄存器相同；另一方面，读出的数据也很熟悉：和错误 9 中一样。因此我们可以直接查看 TLBEHI 写入数据的相关代码，如图 74 所示。可以通过错误 9 举一反三：触发 PIF 例外时，写入 TLBEHI 的数据应当是 wb\_pc 的 32 到 13 位，而非 wb\_vaddr 的对应位置。

#### (3) 错误原因

当触发 PIF 例外时，触发例外的虚地址被存储在了 wb\_pc 中而非 wb\_vaddr 中，错误的数据逻辑导致出错。

#### (4) 修正效果

把 TLBEHI 的数据更新时写入数据的代码修改正确即可，如下所示：

```

412 |     always @ (posedge clk) begin
413 |         if (~resetn) begin
414 |             csr_tlbhei_vppn <= 19'b0;
415 |         end
416 |         else if (tlbrd_we) begin
417 |             if(r_tlb_e)
418 |                 csr_tlbhei_vppn <= r_tlb_vppn;
419 |             else
420 |                 csr_tlbhei_vppn <= 19'b0;
421 |         end
422 |         else if (tlb_excep) begin
423 |             csr_tlbhei_vppn <= (wb_ecode == `ECODE_PIF) ? wb_pc[31:13] : wb_vaddr[31:13];
424 |         end
425 |         else if (csr_we && csr_num == `CSR_TLBHEI) begin
426 |             csr_tlbhei_vppn <= csr_wmask[`CSR_TLBHEI_VPPN] & csr_wvalue[`CSR_TLBHEI_VPPN] |
427 |                             ~csr_wmask[`CSR_TLBHEI_VPPN] & csr_tlbhei_vppn;
428 |         end
429 |     end

```

图 82 错误 10 修改后代码

该方法有效，来到下一个 bug。

## 11、错误 11：exp19 中 data\_addr 赋值出错

### (1) 错误现象

运行仿真后，console 报错如下：

```

[7394547 ns] Error!!!
reference: PC = 0x1c07c300, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x4c000020
mycpu    : PC = 0x1c07c300, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x0000000f

```

图 83 错误 11 对应的 Console 报错

### (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c07c300: 2880018f 1d.w $r15,$r12,0
```

图 84 错误 10 对应的汇编指令

我们可以发现发生错误的指令是 1d 指令，查看 CPU 对于这条指令的处理情况，相关波形图如下：

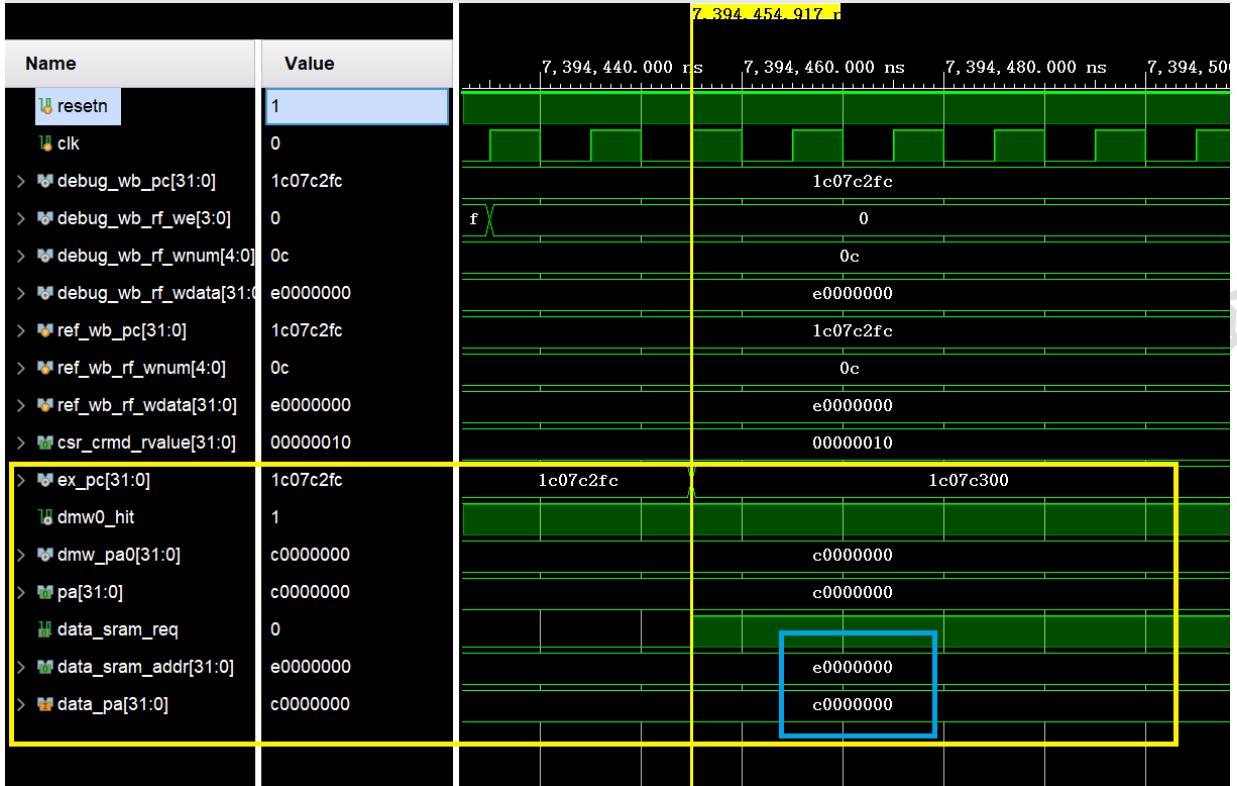


图 85 错误 11 对应的波形图

可以看到，当这条指令在 EX 阶段时，dmw0\_hit 是拉高的，说明这是直接映射。根据映射结果，返回的 data\_pa 是 0xc0000000，而 EX 模块向 AXI 转接桥发送的 data\_sram\_addr 是 0xe0000000，与预期不符。查看 data\_sram\_addr 相关复制逻辑：

```

assign data_sram_req = ex_mem_wait & ex_valid & mem_allowin;
assign data_sram_wr = ex_inst_st_b | ex_inst_st_h | ex_inst_st_w;
assign data_sram_size = ex_sram_size;
assign data_sram_wstrb = ex_mem_strb;
assign data_sram_addr = ex_alu_result ;

```

图 86 错误 11 中 data\_sram\_addr 的赋值逻辑

通过代码我们不难看出，我们仍将 data\_sram\_addr 赋值为 ex\_alu\_result，这是虚地址而非 MMU 返回的物理地址。

### (3) 错误原因

在 EX 模块中，传递给 AXI 转接桥的地址应当为物理地址，而非虚地址。通过进一步查看汇编代码，可以发现该部分代码逻辑是：首先开启直接映射地址翻译模式，将数据写入 0xe0000000 对应的物理地址，而后更改映射方式，将新值写入 0xe0000000 对应的物理地址，再更改映射地址为原来的物理地址，最后加载该地址数据比较是否正确。但是在我们的设计中，所有的数据都写入了物理地址 0xe0000000，导致最后的加载出错。

#### (4) 修正效果

将 data\_sram\_addr 修改正确即可，如下所示：

```
assign ex_sram_size = {ex_inst_ld_w | ex_inst_st_w, (ex_inst_ld_h | ex_inst_ld_hu | ex_inst_st_h)};  
assign data_sram_req = ex_mem_wait & ex_valid & mem_allowin;  
assign data_sram_wr = ex_inst_st_b | ex_inst_st_h | ex_inst_st_w;  
assign data_sram_size = ex_sram_size;  
assign data_sram_wstrb = ex_mem_strb;  
assign data_sram_addr = data_pa;  
assign data_sram_wdata = (ex_inst_st_b)? {4{ex_rkd_value[ 7:0]}} :  
                         (ex_inst_st_h)? {2{ex_rkd_value[15:0]}} :  
                         ex_rkd_value;
```

图 87 错误 11 修改后代码

该方法不好说有没有效，改是肯定要这样改的，但是这个错误出现在第 72 个测试点，而下一个错误出现在第 71 个测试点。

## 12、错误 12：exp19 中 EX 模块中出现 tlb 异常错误向 data sram 发送请求

#### (1) 错误现象

运行仿真后，console 报错如下：

```
[7365957 ns] Error!!!  
reference: PC = 0x1c07bfe4, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x000000ff  
mycpu : PC = 0x1c07bfe4, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000fff
```

图 88 错误 10 对应的 Console 报错

#### (2) 分析定位过程

首先查看报错位置的汇编代码：

```
1c07bfe4: 2880018d 1d.w $r13, $r12, 0
```

图 89 错误 12 对应的汇编指令

查看波形图处理该条指令的波形图：



图 90 错误 12 对应的波形图 (1)

我们可以看到，该条 1d 指令访问的地址为 0x000d0010，搜索该测试点中涉及到这个物理地址的 st 操作，共有两处，给出相应的波形图如下：

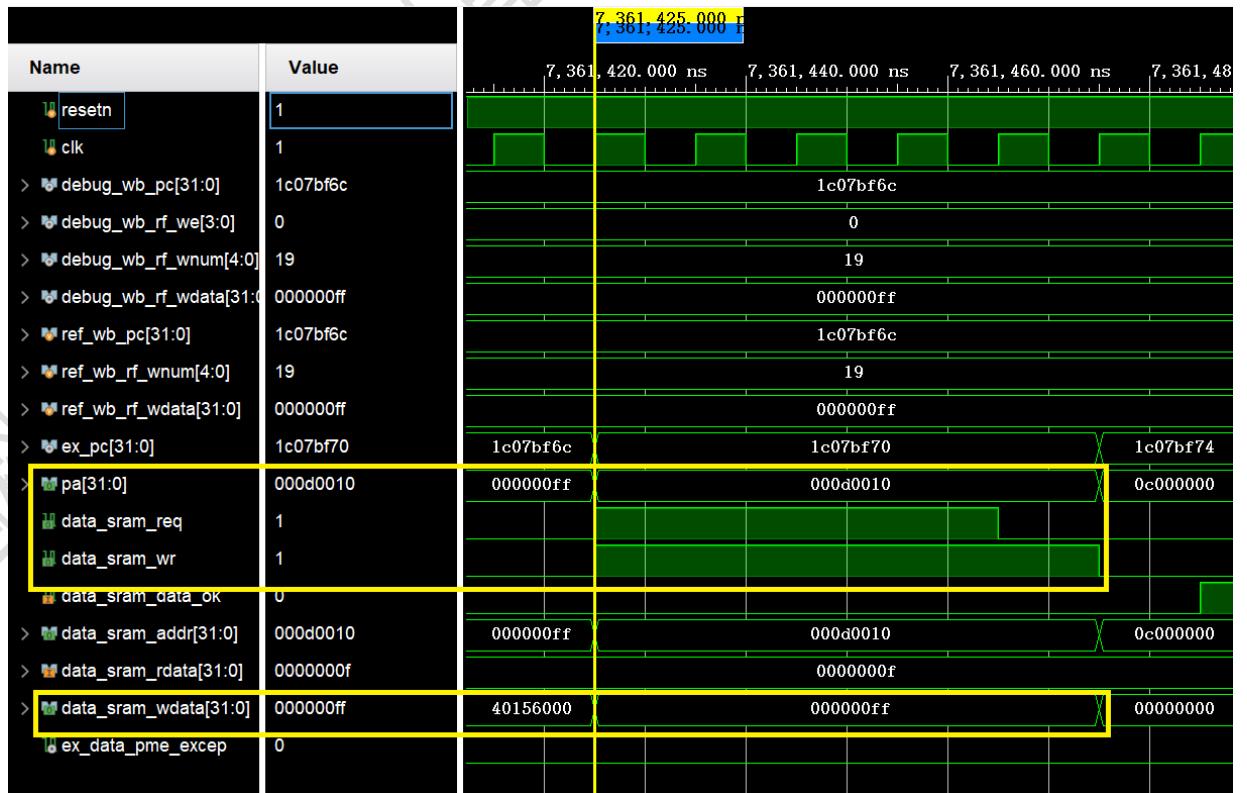


图 91 错误 12 中向对应地址第一次写入数据的波形图

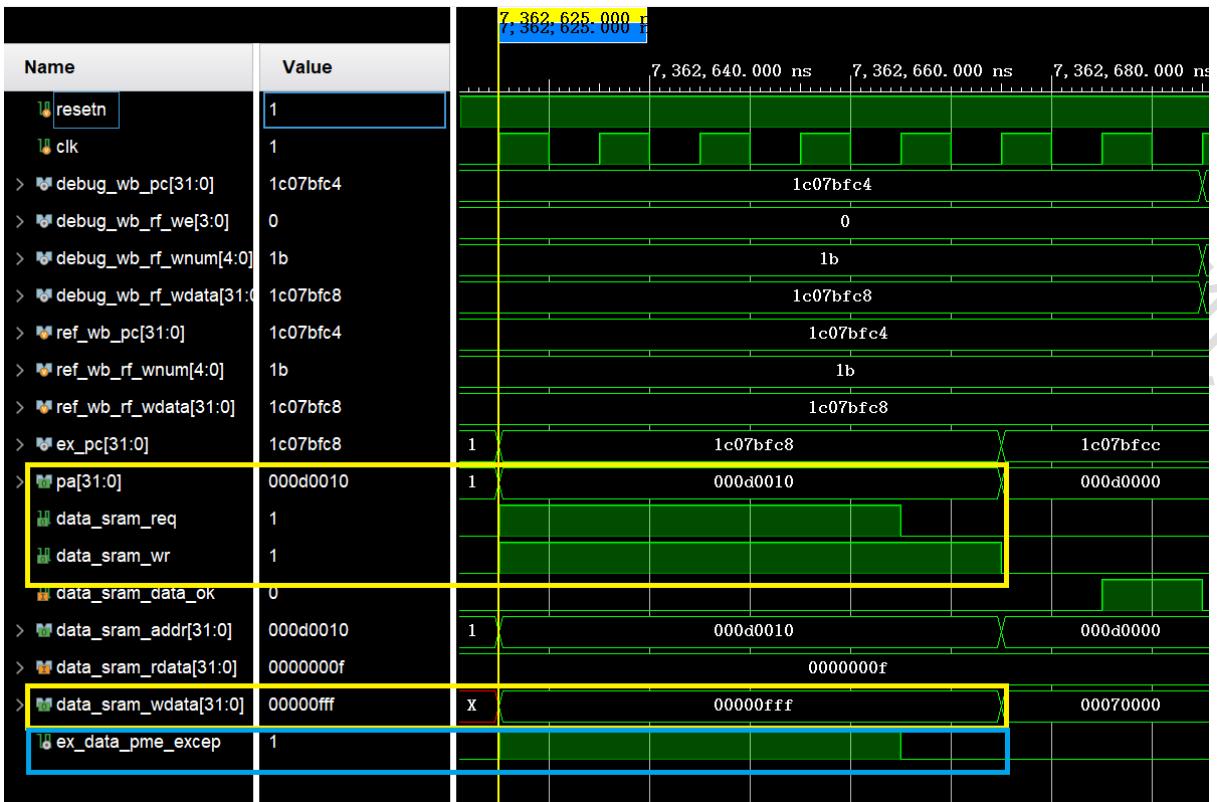


图 92 错误 12 中向对应地址第二次写入数据的波形图

在测试文件的设计中，有两处 store 操作。第一次向 0x000d0010 写入 0x000000ff，第二次写入 0x00000fff。看起来最后访问该物理地址读出来 0x00000fff 是理所当然的，当时我们在这个地方思考了一些时间搞不清为什么，后来想起来查看这之间有没有触发什么例外，而后发现在第二次 store 操作中，触发了 PME 异常。那么错误原因就比较显然了。

### (3) 错误原因

当触发 PME 异常时，向 AXI 转接桥传送的 data\_sram\_req 信号拉高了，data\_sram\_req 信号的赋值逻辑如下：

```

assign ex_mem_wait = (ex_inst_ld_b || ex_inst_ld_bu || ex_inst_ld_h || ex_inst_ld_hu || ex_inst_ld_w || (~ex_mem_strb))
    & ~mem_to_ex_excep & ~ex_flush
    & ~(ex_excp_ale | ex_inst_pif_excep | ex_inst_ppi_excep | ex_inst_tlbr_excep);
assign ex_sram_size = {ex_inst_ld_w | ex_inst_st_w, (ex_inst_ld_h | ex_inst_ld_hu | ex_inst_st_h)};
assign data_sram_req = ex_mem_wait & ex_valid & mem_allowin;
assign data_sram_wr = ex_inst_st_b | ex_inst_st_h | ex_inst_st_w;
assign data_sram_size = ex_sram_size;
assign data_sram_wstrb = ex_mem_strb;
assign data_sram_addr = data_pa;
assign data_sram_wdata = (ex_inst_st_b)? {4{ex_rkd_value[7:0]}} :
    (ex_inst_st_h)? {2{ex_rkd_value[15:0]}} :
    ex_rkd_value;

```

图 93 错误 12 中 data\_sram\_req 信号赋值逻辑

当触发异常时，我们不应当向 AXI 转接桥发起请求。若发起了请求且请求是读请求可能还不会有事，但若是写请求则可能错误地更改对应物理地址的值，最后导致错误。在我们的代码设计中，避免产生异常时拉高

data\_sram\_req 的方式是与上了 ex\_mem\_wait 信号，当有异常时，这个信号会被拉低，但是仔细看赋值逻辑，这个信号只会在除 EX 阶段产生的 TLB 异常外拉低，没有考虑 EX 阶段产生的 TLB 异常，从而导致 data\_sram\_req 错误地拉高。

#### (4) 修正效果

由于 EX 阶段产生 TLB 异常的判断逻辑与 ex\_mem\_wait 信号有关，如下所示：

```
assign ex_data_ppi_excep = data_ppi_except && ex_mem_wait;
assign ex_data_tlbr_excep = data_page_fault && ex_mem_wait;
assign ex_data_pil_excep = data_page_invalid && ex_mem_wait && ~data_sram_wr;
assign ex_data_pis_excep = data_page_invalid && ex_mem_wait && data_sram_wr;
assign ex_data_pme_excep = data_page_clean && ex_mem_wait && data_sram_wr;
```

图 94 错误 12 中 EX 阶段 TLB 相关异常的判断逻辑

因此不能像其他 EX 阶段判断出的异常一样修改 ex\_mem\_wait 的赋值逻辑，否则会形成组合逻辑环，直接修改 data\_sram\_req 的复制逻辑，修改如下：

```
assign ex_mem_wait = (ex_inst_ld_b || ex_inst_ld_bu || ex_inst_ld_h || ex_inst_ld_hu || ex_inst_ld_w || (~ex_mem_strb))
& ~mem_to_ex_excep & ~ex_flush
& ~(ex_excp_ale | ex_inst_pif_excep | ex_inst_ppi_excep | ex_inst_tlbr_excep);
assign ex_sram_size = {ex_inst_ld_w | ex_inst_st_w, (ex_inst_ld_h | ex_inst_ld_hu | ex_inst_st_h)};
assign data_sram_req = ex_mem_wait & ex_valid & mem_allowin &
~(ex_data_ppi_excep | ex_data_tlbr_excep | ex_data_pil_excep | ex_data_pis_excep | ex_data_pme_excep);
assign data_sram_wr = ex_inst_st_b | ex_inst_st_h | ex_inst_st_w;
assign data_sram_size = ex_sram_size;
assign data_sram_wstrb = ex_mem_strb;
assign data_sram_addr = data_pa;
assign data_sram_wdata = (ex_inst_st_b)? {4{ex_rkd_value[7:0]}} :
(ex_inst_st_h)? {2{ex_rkd_value[15:0]}} :
ex_rkd_value;
```

图 95 错误 12 修改后代码

该方法有效，exp19 通过。

## 四、实验总结

截止到上个实验，我们的 CPU 只实现了最简单的直接地址翻译。本实验在 CPU 中增加了映射地址翻译模式，并为其中的页表映射地址翻译模式增加了 TLB 模块，同时也添加了相关的控制状态寄存器，实现了 5 条 TLB 维护指令，并能够处理 TLB 相关的异常。至此，我们的 CPU 中已经实现了 MMU 的功能。

添加映射地址翻译模式，新增 TLB 和 MMU 模块使得流水级也更加复杂，在 IF 流水级和 EX 流水级，均需要实现虚拟地址到物理地址的转换，并判断相关的异常。同时，随机种子也为 debug 增加了一些难度。在 debug 的过程中，我们需要遍历所有可能的情况，认真分析 CPU 的设计，这提高了我们对流水线 CPU, TLB, MMU 以及

CSR 寄存器等各方面的认识，也积累了不少 debug 的经验，并且通过本次实验，本小组成员还进一步熟悉了 git 的使用，对如何使用 git 进行合作有了更深的理解。

國科大B0911009Y計算機體系結構研究課23-24秋季