中国科学院大学
《计算机体系结构(研讨课)》实验报告

姓名 裴晨皓 竹彦博 纪弘璐 学号 2023K8009916003 2023K8009916001 2023K8009916002
专业 计算机科学与技术 实验项目编号 Project 6 实验名称 TLB MMU 设计专题实验

1 实验简介

这次实验中，本组同学按照讲义提示，先实现了独立的 TLB 模块，又实现了用于虚实地址转换的 MMU 模块，并将新增模块与已有的流水线和 CSR 模块进行协同，又依次完成了 TLB 指令的添加和 TLB 异常的处理，同时在设计过程中兼顾了 CSR 寄存器的添加及其读写逻辑的设计。

2 设计方案介绍

下图是完成本次实验后的处理器结构框图：

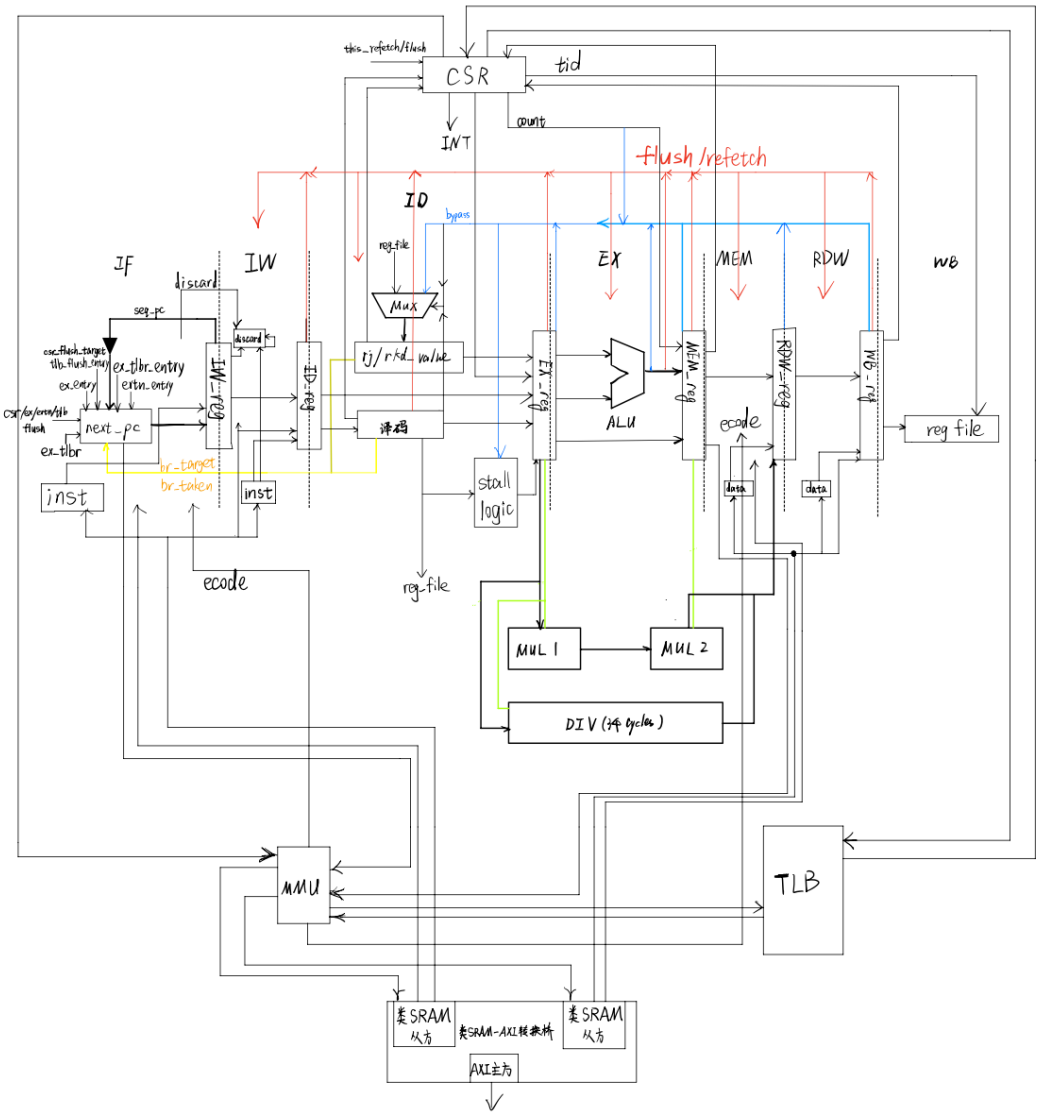

图 1: 处理器结构框图

2.1 总体思路

我们在本次实验新增加了 MMU 和 TLB 模块，上面的结构框图还不够直观，于是我们绘制了下面的逻辑示意图，直观展示新增的两个模块与流水线和 CSR 模块实现协同工作的逻辑：

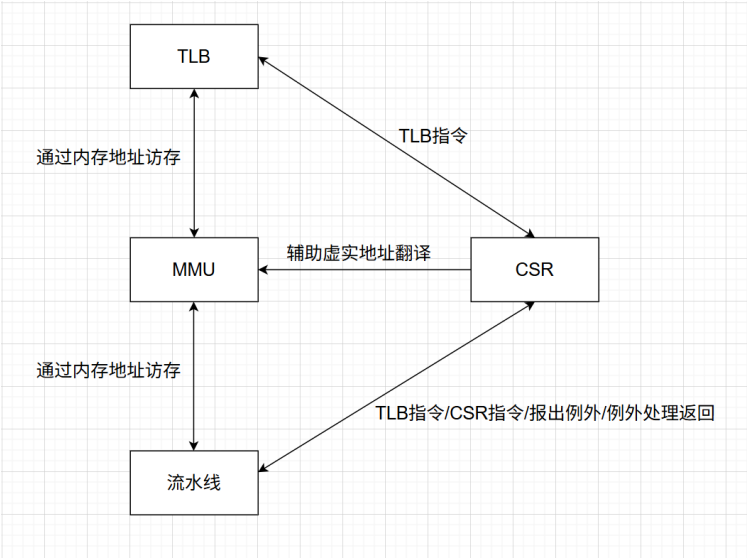

图 2: 新增模块与原有模块协调工作逻辑图

2.2 TLB 模块设计

2.2.1 组织结构与接口

TLB 的主体是一个二维组织结构的查找表，每一项都有多个字段，所以类似寄存器堆的实现，内部要用 reg 对信息进行保存；由于每个字段的含义不同，所以仿照 CSR 模块的写法，不把每一项当成整体实现，而是把不同字段用 reg 分别实现。

对外的接口具体为：

1. 两套查找接口：取指和 Load/Store 访存操作都涉及查找 TLB，为了保证流水线的性能良好，它们需要能够同时对 TLB 进行查找，故它们各自使用一套查找接口；由于 INVTLB 和 TLBSRCH 指令也都需要查找，但它们和 Load/Store 访存指令一样，都在 MEM 阶段进行查找，故复用 Load/Store 访存使用的查找接口 (然后只需要附加它们额外需要的 invtlb_op 输入和 s_index 输出即可)。
2. 读写接口各一套：用于支持专门的根据 index 对 TLB 进行读 (TLBRD) 写 (TLBWR 和 TLBFILL) 的指令。

2.2.2 读写功能

读写功能与寄存器堆的读写类似，都是同步写异步读。需要注意的是要把内部保存的 1 位 ps 字段和读写时输入输出的 6 位 ps 进行转换 (查找时也需要转换)：

```
1    assign r_ps   = tlb_ps4MB[r_index] ? 6'd21 : 6'd12;
2    always @(posedge clk) begin
3        if(we) begin
4                tlb_ps4MB[w_index] <= (w_ps == 6'd21);
5        end
6    end
```

Listing 1: ps 字段的处理

ps 为 21 对应 4MB 页，ps 为 12 对应 4KB 页，模块内每一项的 tlb_ps4MB 信号为 1 时对应 4MB 页，否则对应 4KB 页。

2.2.3 查找功能

关于查找功能，按照讲义的提示，1 号接口 (Load/Store 访存指令在 MEM 级使用) 可以与 INVTLB 指令 op 为 6 的情况复用逻辑。于是，我们先把 INVTLB 用来查找、判断的 condition 进行实现，然后再用它们生成正常查找和 INVTLB 时的查找结果：

```
for(i = 0; i < TLBNUM; i = i + 1)begin
    assign    cond1   [i] =  (tlb_g[i] == 1'b0);
    assign    cond2   [i] =  (tlb_g[i] == 1'b1);
    assign    s1_cond3[i] =  (s1_asid == tlb_asid[i]);
    assign    s0_cond3[i] =  (s0_asid == tlb_asid[i]);
    assign    s1_cond4[i] =  (s1_vppn[18:9] == tlb_vppn[i][18:9])
                          && (tlb_ps4MB[i] || (s1_vppn[8:0] == tlb_vppn[i][8:0]));
    assign    s0_cond4[i] =  (s0_vppn[18:9] == tlb_vppn[i][18:9])
                          && (tlb_ps4MB[i] || (s0_vppn[8:0] == tlb_vppn[i][8:0]));

    assign match1[i] = tlb_e[i] && s1_cond4[i] && (cond2[i] || s1_cond3[i]);
    assign match0[i] = tlb_e[i] && s0_cond4[i] && (cond2[i] || s0_cond3[i]);

    assign inv_match[i] =  (invtlb_op == 5'h0 || invtlb_op == 5'h1) && (cond1[i] || cond2[i])
                        || (invtlb_op == 5'h2) && cond2[i]
                        || (invtlb_op == 5'h3) && cond1[i]
                        || (invtlb_op == 5'h4) && cond1[i] && s1_cond3[i]
                        || (invtlb_op == 5'h5) && cond1[i] && s1_cond3[i] && s1_cond4[i]
                        || (invtlb_op == 5'h6) && match1[i];
end
```

Listing 2: 正常查找与 INVTLB 查找逻辑

- cond1 和 cond2 只与 TLB 内部的信息有关，故各自只有 1 个信号；而 cond3 和 cond4 还额外涉及两个查找接口的外部输入，故它们各自应当有 2 个信号，分别对应两个接口 (s0、s1)。
- 正常查找时 (除了 INVTLB)，命中第 i 项对应于 match0/1[i] 拉高。虽然与 INVTLB 在 op 为 6 时逻辑类似，但并不完全相同——这里要额外考虑 tlb_e，因为这一项如果无效，就不能算命中，也就不能把 match 拉高。
- 对于 INVTLB 用到的查找，命中第 i 项对应于 inv_match[i] 拉高。由于该指令会把所有符合条件的项的 e 位置 0，所以 inv_match 的逻辑中可以不加入 tlb_e，因为本来无效的一项再清 0 一次也无妨。op 为 0 至 5 时根据指令集手册用 cond 逻辑进行设计，op 为 6 时复用 MEM 级 Load/Store 访存用到的 1 号查找接口的 match1 信号。

若是一般查找，那么根据上面提到的 match 信息，就可以判断本次查找是否命中——若某一项的 match 拉高，则说明命中。故我们可以将 match 向量的所有位“或”在一起进行判断：

```
assign s0_found = |match0;
```

Listing 3: 命中判断信号——found(以取指时用到的 0 号查找接口为例)

具体命中哪一项 (s0/1_index)，根据 match 向量的所有位实现一个多路选择器来判断即可，不再赘述。

在查找时还要注意 TLB 里“一项对应两页”的特性——确定了命中项后，还要对其中的两页进行“二选一”：

```
assign s0_select_unit1 = tlb_ps4MB[s0_index] && s0_vppn[8] || !tlb_ps4MB[s0_index] &&
    s0_va_bit12;
```

Listing 4: 对一项中的两页进行选择 (以 0 号查找接口为例)

设同一项中的两页里，虚页号最低位 (第 0 位) 为 0 的一页为第 0 页，虚页号最低位为 1 的一页为第 1 页，则 s0_select_unit1 信号拉高时表示选中第 1 页。此外，在我们的设计中，vppn 信号的第 0 位对应的是 4KB 页的虚双页号的第 0 位，而非 4MB 页的虚双页号的第 0 位，这一点要注意。

关于 s0_select_unit1 信号，下面是两个易错的细节，理解不到位或是数位数时粗心就可能出错：

- 4KB 页：

  这时，页内偏移为虚地址的第 0 到 11 位，虚页号从虚地址第 12 位开始算；虚页号最低位不放入 TLB，虚双页号从第 13 位开始算。所以这里要根据 s0_va_bit12 来选择一项中的两页，而非 vppn 中的某一位。

- 4MB 页：

  根据指令集手册，每一项里的两页实际上都是 2MB 大小。由于 $2MB = 2^{21}B$，所以虚页号的第 0 位应该看虚地址的第 21 位，也就是虚双页号的第 8 位。

特别地，若当前是 INVTLB(invtlb_valid 拉高) 指令用到的查找，直接把命中项的 e 位置 0 即可，不涉及项内页选择：

```
always @(posedge clk) begin
        if(we)
                tlb_e[w_index] <= w_e;
        else if(invtlb_valid) begin
                for(j = 0; j < TLBNUM; j = j + 1) begin
                        if(inv_match[j])
                                tlb_e[j] <= 1'b0;
                end
        end
end
```

Listing 5: INVTLB 对命中项的 e 位置 0 逻辑

2.3 MMU 模块设计

如图 2 所示，我们的 MMU 模块主要负责接收流水线发来的虚地址，进行虚实地址转换操作，并将转换后的物理地址送往 AXI 转接桥。如果转换过程涉及 TLB 查找，则还要对 TLB 相关异常进行判定并发回流水线。

2.3.1 地址翻译逻辑

以取指为例：

```
assign inst_sram_paddr =
(crmd_da_value && !crmd_pg_value) ?
    inst_sram_vaddr :
(inst_sram_vaddr[31: 29] == dmw0_vseg_value && dmw0_plv_cond) ?
    {dmw0_pseg_value, inst_sram_vaddr[28: 0]} :
(inst_sram_vaddr[31: 29] == dmw1_vseg_value && dmw1_plv_cond) ?
    {dmw1_pseg_value, inst_sram_vaddr[28: 0]} :
{tlb_s0_ppn, inst_sram_vaddr[11: 0]};
```

Listing 6: 虚实地址翻译-取指

- 若 DA 为 1 且 PG 为 0，直接把虚地址作为物理地址输出。
- 否则再看两个直接映射窗口，若权限等级符合 (dmw0/1_plv_cond 为 1)，并且虚地址的高 3 位与对应窗口的 dmw0/1_vseg_value 相等，则直接用 dmw0/1_pseg_value 与虚地址的低 29 位拼接得到物理地址。
- 否则，使用 TLB 查找得到的 ppn 与虚地址的低 12 位 (页内偏移) 拼接得到物理地址。

关于使用直接映射窗口时的权限等级判定 (dmw0/1_plv_cond)，只需根据 CRMD 寄存器的 PLV 值与 DMW0/1 寄存器记录的 PLV 情况进行比较即可，二者相同说明权限级别符合，dmw0/1_plv_cond 拉高。此处不再赘述。

2.3.2 TLB 异常判定

TLB 异常发生在借助 TLB 进行地址转换时，所以发生此类异常的必要条件是地址转换“的确使用了 TLB”，同样以取指为例：

```
wire use_tlb_i = !(crmd_da_value && !crmd_pg_value) &&
       !(inst_sram_vaddr[31: 29] == dmw0_vseg_value && dmw0_plv_cond) &&
       !(inst_sram_vaddr[31: 29] == dmw1_vseg_value && dmw1_plv_cond);
```

Listing 7: 对地址翻译是否使用了 TLB 的判定

即：既不是“DA 为 1 且 PG 为 0”的直接地址翻译，也不是落在两个直接映射窗口内的情况，那就一定用到了 TLB。进一步，再结合查找 TLB 得到的信息，就可以产生异常标志信号：

```
wire tlbr_i = use_tlb_i && !tlb_s0_found;
wire pif_i = use_tlb_i && tlb_s0_found && !tlb_s0_v;
wire ppi_i = use_tlb_i && tlb_s0_found && tlb_s0_v && (crmd_plv_value > tlb_s0_plv);
```

Listing 8: 异常标志-以取指为例

这里要注意的是，取指时仅会出现 TLB 重填异常、取指页无效异常和页特权等级不合规异常 (共 3 类)；而访存级还会出现 load/store 页无效异常和页修改异常，但不会出现取指页无效异常 (共 5 类)。

最后，还要根据上面的异常标志，把异常信息通过 ecode 和 esubcode 的形式发回流水线，此处仍以取指的 ecode 为例：

```
assign ecode_i = tlbr_i ? 6'h3F :
                 pif_i  ? 6'h3 :
                 ppi_i  ? 6'h7 :
                          6'h0;
```

Listing 9: ecode 生成逻辑-取指

2.4 TLB 指令的实现与 TLB/CSR 冲突处理

在我们的设计中，TLB 指令对 TLB 的读、写和查找都在 MEM 级进行。下面介绍实现过程中的关键：

2.4.1 重取机制

根据讲义的提示，由于实现的几条 TLB 指令 (除了 TLBSRCH)，都会修改 TLB 的内容或是虚实地址转换所依赖的 CSR 的内容，而这些内容最早在取指级做地址翻译就会被使用，所以这些 TLB 指令后面几条已经取出的/发过取指请求的指令，很可能用的是旧的 TLB/CSR 信息进行的地址翻译，所以需要进行“重取”。

此外，CSR 指令也有可能改写地址翻译依赖的 CRMD、ASID、DMW0 或 DMW1 寄存器，那么后续指令也要重取。

借鉴异常冲刷的处理逻辑，仿照 this_flush 信号 (用于标记当前流水级的指令将来会被冲刷，以此限制指令做出修改内存等改变体系状态的操作)，我们引入 this_tlb_refetch/flush 和 this_csr_refetch/flush 信号，限制因重取而被冲刷的指令的行为。

以 MEM 级的 this_tlb_refetch 和 this_tlb_flush 信号为例：

```
assign this_tlb_refetch = in_valid
                        && (tlbsrch || tlbrd || tlbwr || tlbfill || invtlb
                           || RDW_this_tlb_refetch);
assign this_tlb_flush = in_valid && RDW_this_tlb_refetch;
```

Listing 10: this_tlb_refetch 和 this_tlb_flush 信号-以 MEM 为例

由于无阻塞情况下，TLB 指令对 TLB 的写入在从 MEM 进入 RDW 的上升沿才能完成，所以我们认为其在 RDW 级“提交”重取；同理 CSR 指令导致重取时，在 EX 级“提交”重取——尽早地“提交”有利于提高性能。

- this_tlb_refetch：表明本级的指令与重取相关。

  本级有效，并且本级是 TLB 指令或是后续直到 (做重取提交的)RDW 的所有阶段中，存在与重取有关的情况，那么本级指令也一定与重取有关。

- this_csr_flush：表明本级的指令将来会因重取而被冲刷。

  本级有效，并且后续直到 RDW 的所有阶段中，存在有关于重取的情况，说明本级的指令将来一定会被重取。

refetch 和 flush 的区别在于，前者包含了“本级是引起重取的指令”的情况，而后者不包含。为了防止引起重取的指令作用也被限制，所以在其发挥作用的 MEM 级区分出这两个信号 (用 flush 限制指令的功能而非用 refetch)，而前面的流水级中，引起重取的指令本来也不会产生作用，其作用是否受限不会带来影响，故只使用一个 refetch 信号即可。

重取被提交后，前面各流水级复用异常时的冲刷逻辑进行冲刷即可，冲刷后的 IF 级取指 PC 应当设为被重取的第一条指令 (即引起重取的指令的下一条指令) 的 PC。

2.4.2 “冲突”处理

由于 TLB 指令对 TLB 的读写都在 MEM，因此不会出现围绕 TLB 的读写相关冲突，只需要做好前面提到的重取即可。

然而，原有的设计中，CSR 指令对 CSR 的读写都在 ID 进行；现在 (除了 ID)IF 和 MEM 都会读 CSR，在 MEM 更是有可能出现写 CSR 的 TLBSRCH、TLBRD 指令。由于读与读之间不互斥，而读与写、写与写是互斥的，所以我们重点考虑涉及写的冲突处理。
(特别地，WB 级的异常提交与返回也涉及 CSR 的写，但这种情况下前面的指令会被冲刷，将会被冲刷的指令不会产生作用，相当于不读也不写 CSR，所以没有冲突。)

1. MEM 写，ID 写：

   此时 MEM 为 TLB 指令，前面的流水级的 refetch 信号都会拉高，阻止前面即将被重取冲刷的指令发挥作用，故实际上 ID 不会出现“写”CSR 的行为，所以不必担心：

   ```
   assign csr_we = in_valid && (inst_csrwr || inst_csrxchg) && ready_go && out_ready
                  && !this_flush && !this_tlb_refetch && !this_csr_flush;
   ```

   Listing 11: ID 级 refetch 信号对 csr 写使能的限制

2. MEM 写，ID 读：

   与上一条类似，ID 这时被 refetch 限制，ID 这条指令将来会被冲刷，所以这时 CSR 的读端口读出什么值也就不必在意了。

3. MEM 读，ID 写：

(a) MEM 为 TLB 指令，ID 为 CSR 指令：

同样，ID 的指令被 refetch 限制，不必在意二者冲突。

(b) MEM 为正常访存指令，ID 为 CSR 指令：

这时没有 refetch 对 ID 进行限制 (不需要重取)，若 ID 的写 CSR 操作发生在 MEM 读 CSR 之前，则可能使 MEM 读出错误的值。于是，要对 ID 进行阻塞 (当前设计中 CSR 写操作在 CSR 写指令从 ID 进入 EX 的上升沿发生，所以只要写指令阻塞在 ID，就不会对 CSR 进行写入)。

```
wire csr_affect_mem = in_valid && (inst_csrwr || inst_csrxchg)
                    && (  inst[23: 10] == `CSR_CRMD || inst[23: 10] == `CSR_ASID
                       || inst[23: 10] == `CSR_DMW0 || inst[23: 10] == `CSR_DMW1);
```

Listing 12: csr_affect_mem 信号的生成

首先，要判定 ID 的 CSR 指令是否会改写地址翻译所依赖的 CSR(CRMD、ASID、DMW0、DMW1)，得到上面的 csr_affect_mem 信号。若它不改写这些 CSR，则不会影响 MEM 的地址翻译，为了保障性能，可以不阻塞。

然后，再根据 EX 和 MEM 的指令类型 (是否为 load/store)，生成 ID 级的阻塞信号：

```
assign csr_mem_stall = in_valid && (EX_mem_inst || MEM_mem_inst) && csr_affect_mem;
```

Listing 13: CSR 读写相关的阻塞信号-ID 级

2.4.3 指令功能实现

我们把流水线中关于 TLB 指令的信号都送入 CSR，由 CSR 模块把这些指令的信号送入 TLB。如果它们涉及 CSR 的读写，能够较为方便地在 CSR 模块内进行处理。

对于 TLBSRCH 和 INVTLB 指令，需要关注它们对 TLB 查找接口的共用 (它们与地址翻译过程共用)：

```
assign tlb_s1_asid = MEM_tlbsrch_to_csr ? asid_asid_value :
                     MEM_invtlb_to_csr ? MEM_rj_value[9: 0] :
                     asid_asid_value;
assign tlb_s1_vppn = MEM_tlbsrch_to_csr ? tlbehi_vppn_value :
                     MEM_invtlb_to_csr ? MEM_rkd_value[31: 13] :
                     data_sram_vaddr[31: 13];
```

Listing 14: TLB 查找接口的共用

若是 TLBSRCH，那么就是用 asid 和 tlbehi 的值；若是 INVTLB，则要用到相应的 rj 和 rk 寄存器的值；否则就是默认的地址翻译使用的 asid 寄存器以及访存使用的虚地址的值。此外，根据手册，TLBSRCH 还要根据查找结果更新 TLBIDX 寄存器：

```
else if (tlbsrch) begin
        csr_tlbidx_ne <= !tlb_s1_found;
        if (tlb_s1_found) begin
            csr_tlbidx_index <= tlb_s1_index;
        end
end
```

Listing 15: TLBSRCH 对 TLBIDX 的更新

对于 TLBFILL 和 TLBWR，只需要拉高 TLB 的写使能并把相应 CSR 的值从写端口送入 TLB 即可。二者的区别在于 FILL 是随机选一个位置写入，而 WR 是根据 index 写入，所以写入位置可以都用 TLBIDX 的 INDEX 字段进行指定。

```
assign tlb_we = tlbwr || tlbfill;
assign tlb_w_index = csr_tlbidx_index;
```

Listing 16: TLB 写使能与写 index

根据手册，此处还要注意：

```
assign tlb_w_e = !csr_tlbidx_ne || (csr_estat_ecode == `ECODE_TLBR);
assign tlb_w_g = csr_tlbelo0_g && csr_tlbelo1_g;
```

Listing 17: 写入 TLB 的项的 e 位与 g 位

1. 只有 TLBIDX 的 NE 位为 0 或 ESTAT 的 ecode 对应 TLBREFILL 时 (处于重填例外处理过程中)，才把写入项的 e 位置 1(有效项)，否则写 0；

2. 只有 TLBELO0 和 TLBELO1 的 g 位都为 1 时，才把写入项的 g 位置 1，否则写 0。

对于 TLBRD 指令，如果查找到的是有效项，则把其页表项信息填入相应 csr，并把 TLBIDX 的 NE 置 0；否则把用于保存页表项信息的 csr 清零，同时 TLBIDX 的 NE 置 1。

2.5 对 TLB 异常的处理

2.5.1 异常的判定与提交

在流水线内，涉及发送虚地址进行访存的流水级为 IF 和 MEM，因此需要在这两个流水级根据 MMU 发回的 ecode 进行 TLB 异常的判定。这里以 MEM 为例：

```
always @(posedge clk) begin
    if (rst) begin
        has_exception_out <= 1'b0;
    end
    else if (in_valid && ready_go && out_ready) begin
        has_exception_out <= has_exception || ((|mmu_ecode_d) & (res_from_mem || mem_we));
    end
end
```

Listing 18: 异常检测与传递-MEM

(|mmu_ecode_d) & (res_from_mem || mem_we) 表明当前 MEM 是一条访存指令，并且 MMU 发回的 ecode 不为 0，说明发生了 TLB 异常；has_exception 则表明当前指令在前面流水级是否发生了异常——将本流水级与前面流水级的情况合并，就得到当前 MEM 级指令流入 RDW 后的异常标志 has_exception_out。

异常码 ecode/esubcode 和异常地址 maddr 等异常信息的记录、传递与异常的提交仍沿用之前实验的思路，不再赘述。特别地，IF 级存在的 ADEF 和 TLB 异常中，ADEF 优先级更高，这是因为：进行 TLB 的查找与 TLB 异常判定的前提是取指地址合法。

```
if(ADEF) begin
    ecode_out <= {6{ADEF}} & 6'h8;
end
else begin
    ecode_out <= mmu_ecode_i;
end
```

Listing 19: ADEF 与 TLB 异常的优先级区分

发生异常后，与前面的实验处理方式相同，需要限制异常指令及其前面流水级指令发挥作用，此处不再浪费多余篇幅描述。

这里还要注意的是，TLB 重填异常比较特殊，有单独的异常处理程序入口 (ex_tlbr_entry_preserved)，在 IF 级选择取指 nextpc 时要纳入考虑：

```
assign nextpc = ex_flush_preserved ?
                (ex_tlbr_preserved ? ex_tlbr_entry_preserved : ex_entry_preserved) :
                ertn_flush_preserved ? ertn_entry_preserved :
                tlb_flush_preserved ? tlb_flush_entry_preserved :
                csr_flush_preserved ? csr_flush_target_preserved :
                br_taken_preserved ? br_target_preserved : seq_pc;
```

Listing 20: IF 级取指 PC 选择

(此外，上面这一代码片段中的 tlb_flush_entry_preserved 和 csr_flush_target_preserved 分别对应前面提到的因 TLB 指令/CSR 指令而重取指令的起始 PC。)

2.5.2 对 CSR 的修改

CSR 在应对 TLB 异常时也需要起辅助作用。比如，需要在 TLBEHI 的 VPPN 字段记录发生异常的虚双页号：

```
if(wb_ex && (  wb_ecode==`ECODE_TLBR || wb_ecode==`ECODE_PIL || wb_ecode==`ECODE_PIS
            || wb_ecode==`ECODE_PIF || wb_ecode==`ECODE_PME || wb_ecode==`ECODE_PPI)) begin
    csr_tlbehi_vppn <= wb_vaddr[31:13];
end
```

Listing 21: TLB 异常虚双页号记录

```
assign ex_tlbr_entry = csr_tlbrentry_rvalue;
```

Listing 22: TLB 重填异常处理入口

发生异常时还需要把出错地址填入 BADV 寄存器；如果是 TLB 重填异常，要把 TLB 重填异常的处理入口地址 (TLBRENTRY 的内容) 发送给 IF 级；重填异常提交和返回时还要修改地址翻译模式 (DA、PG)。这些细节上的处理较为简单，不再赘述。

3 Debug 记录 (见下一页)

3.1 忘记在恰当时机修改地址翻译模式

本章实验的特点是难度不大，但细节较多，需要仔细研读手册，容易疏忽——尤其是地址翻译模式的设置。

在 rst 复位时，不能盲目把各种寄存器初始化为 0。我们最初的设计中，忘记在 rst 复位时设置 DA 为 1，PG 为 0，而是把二者都初始化为 0，导致没能进入直接地址翻译模式，影响了取指：

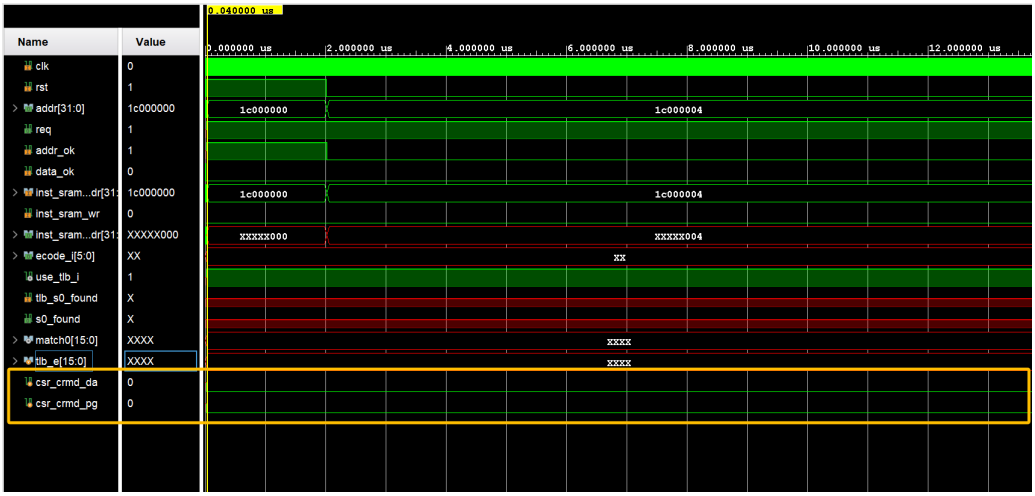

图 3: 复位时 DA 未置为 1-波形图

正确的设置方式为：

```
csr_crmd_da <= 1'b1;
csr_crmd_pg <= 1'b0;
```

Listing 23: 复位时地址翻译模式设置

此外，我们最初忘记了在 TLB 重填异常提交时，要把地址翻译模式改为“直接地址翻译”(DA 为 1，PG 为 0)；在重填异常处理完后，返回时要改回“映射地址翻译模式”(DA 为 0，PG 为 1)。下面展示的片段为修改后的正确逻辑。

```
else if (wb_ex) begin
    csr_crmd_plv <= 2'b0;
    csr_crmd_ie  <= 1'b0;
    if(wb_ecode==`ECODE_TLBR) begin
        csr_crmd_da  <= 1'b1;
        csr_crmd_pg  <= 1'b0;
    end
end
else if (ertn_flush) begin
    csr_crmd_plv <= csr_prmd_pplv;
    csr_crmd_ie  <= csr_prmd_pie;
    if(csr_estat_ecode==`ECODE_TLBR) begin
        csr_crmd_da  <= 1'b0;
        csr_crmd_pg  <= 1'b1;
    end
end
```

Listing 24: TLB 重填异常与地址翻译模式修改

3.2 发生 TLB 异常时忘记限制发送访存请求

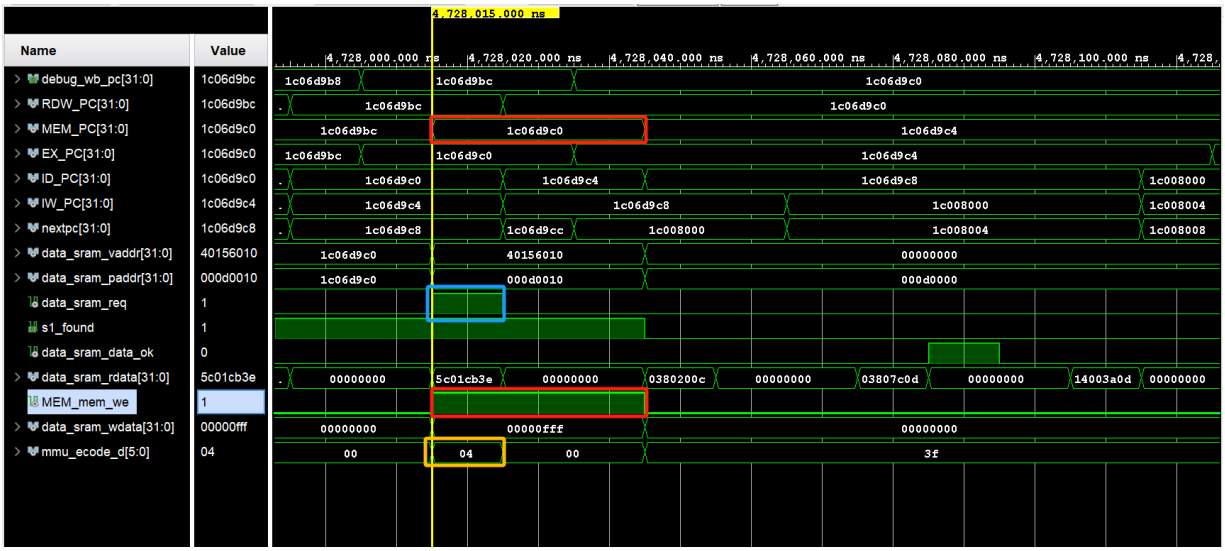

图 4: 发生 TLB 异常时忘记限制发送访存请求-波形图

上图中，红框部分显示 MEM 此时是一条 store 指令，其触发了 PME 异常 (如黄色方框所示，ecode 为 04)，但是访存 req 信号却仍然拉高 (如蓝色方框所示)，导致这条发生异常的指令仍然发送了访存请求，显然是不合理的。

经过排查发现，我们忘记在 TLB 异常发生时限制访存请求 req，正确的逻辑为：

```
assign req = in_valid && !handshake_done && !this_flush && (res_from_mem || mem_we) && !
    this_tlb_flush && !(|mmu_ecode_d);
```

Listing 25: 在 TLB 异常时限制访存请求 req

即：最后额外“与”上“!(|mmu_ecode_d)”，表示只有没发生 TLB 异常的访存指令 (ecode 为 0)，才能拉高 req 请求访存。

3.3 忘记使用“流水级有效”信号 (in_valid) 限制 TLB 指令功能

理论上，只有流水级有效时，其内部的指令信息才有意义，才能发挥作用，但我们最初忘记用 in_valid 信号来限制 TLB 指令，出现下图波形：

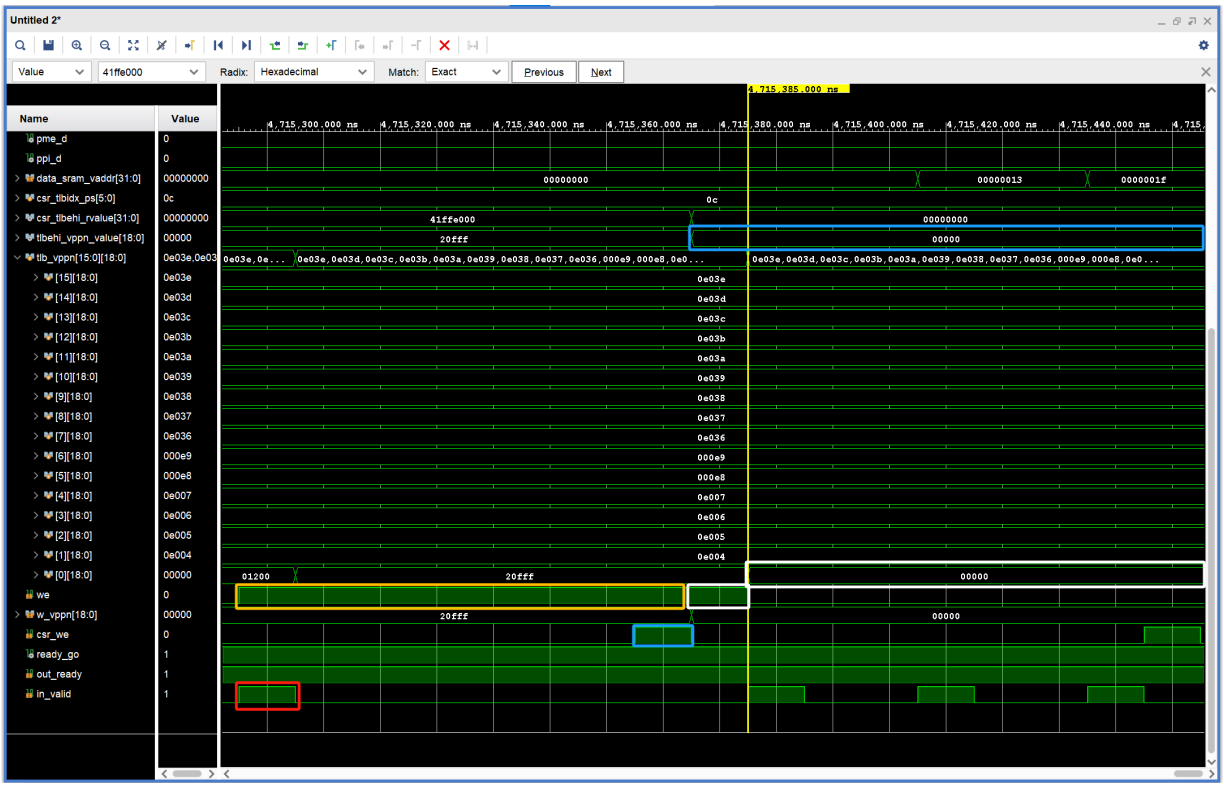

图 5: 忘记使用 in_valid 限制 TLB 指令功能-波形图

红框位置是一条位于 MEM 级的 TLBWR 指令，它把 TLB 的写使能 we 拉高 (见黄框位置)，把 TLB 相关的 CSR 存放的页表项信息写入 TLB(比如图中的 tlbehi_vppn)，下一拍这条 TLBWR 从 MEM 级流出，in_valid 拉低表明这时也没有其他指令流入 MEM，但是 TLB 写使能 we 却仍然拉高 (黄框位置)。如蓝框所示，后来出现了一条 ID 级的 CSR 写指令，改写了 tlbehi 的 vppn 字段；如白框所示，由于 TLB 的 we 仍然拉高，导致这一被修改后的 tlbehi_vppn 被误写入了 TLB。

解决方法十分简单，只要把 TLB 指令“有效”(即会产生作用) 的逻辑里“与”上表明流水级有效的 in_valid 信号即可：

```
1 assign tlbwr_to_csr    = in_valid && tlbwr && !this_flush && !this_tlb_flush;
```

Listing 26: TLB 指令“有效”(即会产生作用) 逻辑-以 TLBWR 为例

这一 bug 具有较强的迷惑性，因为把写好的 TLB 表项又“写坏了”，导致后续本应提交 TLBload 页无效异常，却报出了重填异常，表现为异常处理入口点不同，让人首先怀疑是异常入口选择逻辑写错。

3.4 store 指令误报出 load 页无效异常

如下图所示，黄色方框处的指令报出了红色方框处的 pil 异常 (load 页无效异常)，而非 store 页无效异常 (pis)：

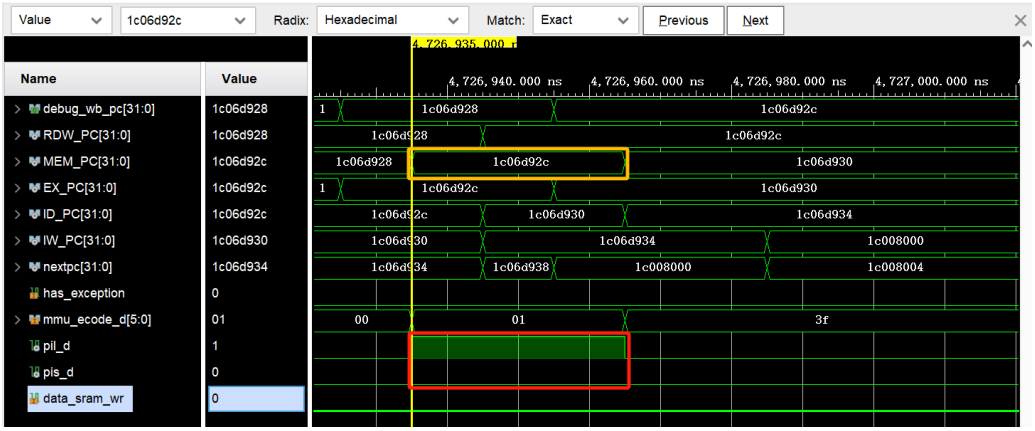

图 6: store 指令误报出 load 页无效异常-波形图

乍看波形毫无异样，但是查找反汇编文件发现这实际上是 st 指令，无论如何不可能出现 load 页无效异常：

```
1c06d928:       0280237b        addi.w  $r27,$r27,8(0x8)
1c06d92c:       298003ec        st.w    $r12,$r31,0
1c06d930:       5c025f3e        bne     $r25,$r30,604(0x25c) # 1c06db8c <inst_error>
1c06d934:       14003a0c        lu12i.w $r12,464(0x1d0)
```

图 7: store 指令误报出 load 页无效异常-反汇编

回看 MEM 级生成的用于内存读写选择的 wr 信号的逻辑——用于生成 wr 信号的 wstrb 信号内部包含了与冲刷相关的信号 this_flush 和 this_tlb_flush，即这条指令如果将来会被冲刷，那么一定会强行使得 wstrb 为 0，进而导致 wr 为 0：

```
assign wr = (|wstrb);
assign wstrb  = {4{mem_we && valid && in_valid && !this_flush && !this_tlb_flush}} & (
                    ({4{mem_op[5]}} & (4'b0001 << alu_result[1: 0])) |  // SB
                    ({4{mem_op[6]}} & (4'b0011 << alu_result[1: 0])) |  // SH
                    ({4{mem_op[7]}} & 4'b1111)  // SW;
                );
```

Listing 27: MEM 级读写选择-wr 信号逻辑

而 MMU 模块内用到 wr 来区分 load/store 页无效异常：

```
wire pil_d = use_tlb_d && tlb_s1_found && !tlb_s1_v && !data_sram_wr;
wire pis_d = use_tlb_d && tlb_s1_found && !tlb_s1_v && data_sram_wr;
```

Listing 28: load/store 页无效异常判定

这两种异常判定的区别仅在于访存操作的类型不同 (是读还是写,wr 为 0 还是 1)。由于前面在指令“将来会被冲刷”的情况下强行把本应为 1 的 wr 置为 0，所以会导致 store 页无效被误判为 load 页无效。

最终的解决方法是：在 wstrb 信号中删除表明指令“将来会被冲刷”的 this_flush 和 this_tlb_flush 信号，从而使得 wr 信号不会被强行错误更改，也就解决了上面提到的异常误判问题。

4 合作说明

本实验由本组成员共同合作完成，组内同学同等贡献。