Lab 8 报告

马迪峰 2021K8009929033
饶嘉怡 2021K8009929005
范子墨 2021K8009929006
箱子号 5

一、实验任务（10%）

操作系统和 CPU 硬件需要实现存储管理来管理内存。操作系统的主要工作是有效地管理内存，记录哪些内存是正在使用的，在进程需要时分配内存以及在进程完成时回收内存。CPU 通过地址转换机制将虚拟地址转换为物理地址，然后访问物理地址取出操作系统所需要的指令和数据。

虚实地址转换是存储管理中的重要一环，它能够建立虚拟地址空间和物理地址空间之间的对应关系，从而使得在较大的虚地址空间中编写的程序能够在较小的实地址空间运行，以便节省物理存储器；或者使得在较小的虚地址空间中编写的程序能够在较大的实地址空间运行，以便提高系统的处理能力。

更进一步地，进行虚实地址转换的过程中，有两个部件发挥着极大的作用。MMU（Memory Management Unit，内存管理单元）是一种硬件模块，其主要功能是将虚拟地址转换为物理地址，同时提供访问权限的控制和缓存管理等功能。TLB（Translation Lookaside Buffer）是 MMU 的一个高速缓存，用于缓存页表转换的结果，从而缩短页表查询的时间

本实验主要实现了对 TLB 模块的结构设计和例外支持以搭建一个 MMU 模块，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。

具体来说，本实验总共分为三个阶段。

Exp17:单独设计 TLB 模块，并通过功能仿真测试。

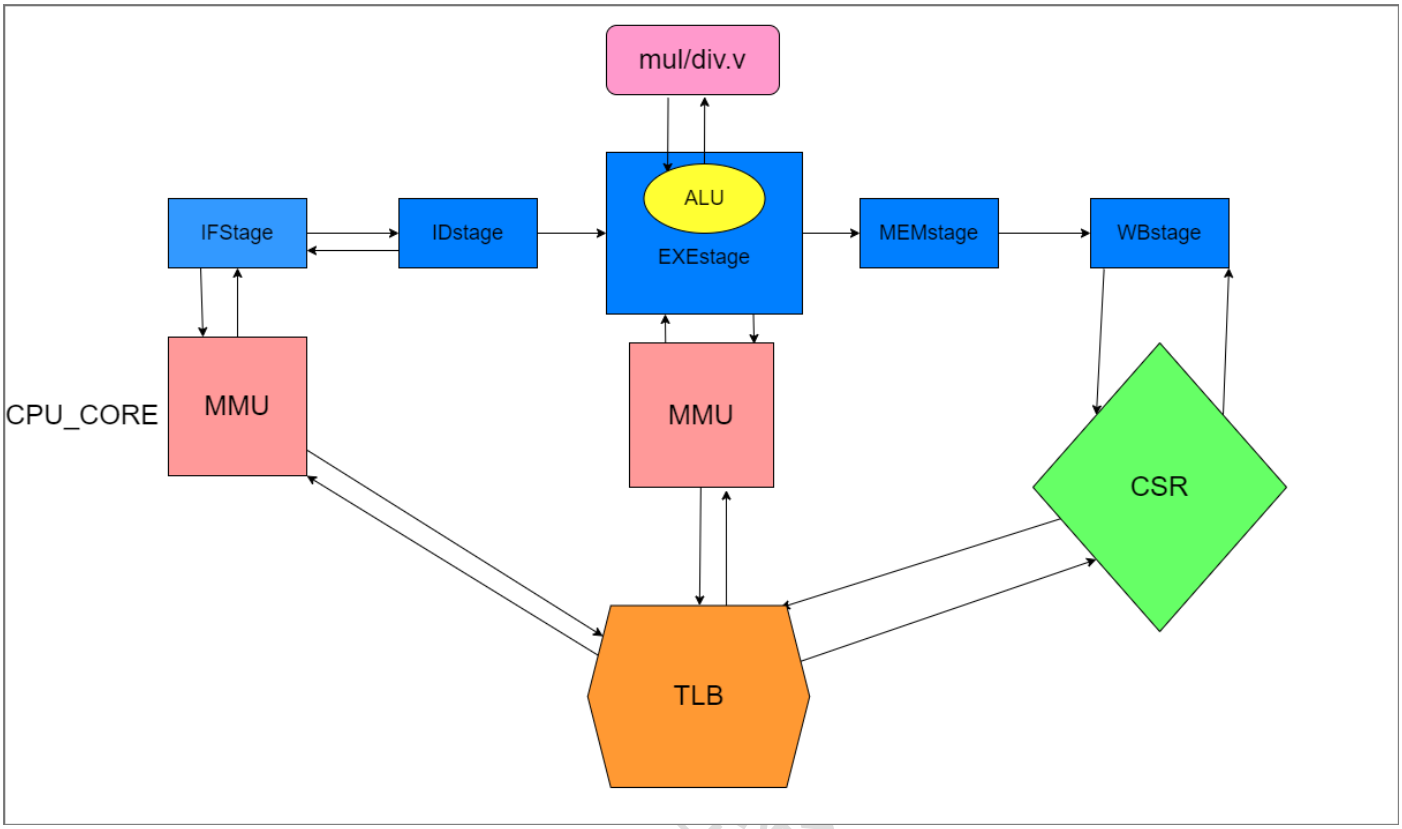
Exp18:在 CPU 中添加五条 TLB 指令，包括 TLBSRCH、TLBRD、TLBWR、TLBFILL、INVTLB；同时在 CPU 中增加 TLBIDX、TLBEHI、TLBELO0、TLBELO1、ASID、TLBRENTRY 等 CSR 寄存器以支持上述指令的实现。

Exp19: 需要在 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等；同时在 CPU 中添加直接映射配置窗口控制寄存器 DMW0 和 DMW1。最终搭建出一个完整的 MMU 模块，以支持内存管理。

二、实验设计（40%）

（一）总体设计思路

总体设计图：省去 SRAM 数据通路和 CPU_TOP 模块等。



图表 1 CPU_CORE 简易设计图

Exp17

见下面 TLB 模块的设计。

Exp18

TLB 当中设计了取指和访存两套查找端口、读端口和写端口。对于新实现的这五条 TLB 指令，首先要考虑其使用哪一套 TLB 端口，其次考虑其作为生产者和消费者涉及到的写后读相关，从而给其安排一个合适的流水级来执行操作，并且选择一个合适的消除相关的方法。

TLBSRCH 指令：应当复用一套查找端口，为了复用时不阻塞其他指令访问 TLB，显然应当复用访存查找端口，并与访存一样安排在 EXE 级。该指令作为消费者要读取 CSR.ASID 和 CSR.TLBEHI 的内容，而 CSR 写指令可能会在 WB 级修改二者的内容。因此，TLBSRCH 指令与处在 MEM 和 WB 级的 CSR 写指令形成写后读相关，可采用阻塞解决。同时，该指令查找的 TLB 可能会被 TLB 写指令修改，这个会通过 TLB 写指令的重取机制解决，后续讨论。该指令作为生产者要修改 CSR.TLBIDX.Index 和 CSR.TLBIDX.NE 的内容，会与 MEM 和 WB 级的 CSR 读指令形成写后读相关，可通过流水级传递推迟到 WB 级，避免冲突。

TLBRD 指令：直接使用读端口。为了尽量避免围绕 CSR 的写后读相关，该指令可安排在 WB 级与 CSR 读写指令一起。但该指令会修改 CSR.ASID，这是取指时就需要读的 CSR 寄存器。由于在 ID 级才能译码出这是 TLBRD 指令，此时 IF 级已经根据旧的信息取回了指令，无法通过阻塞解决，只能将 TLBRD 指令后面发射的所有指令都

取消，在它执行完之后重取。

TLBWR 指令：直接使用写端口。与 TLBRD 相同，为了尽量避免围绕 CSR 的写后读相关，该指令可安排在 WB 级与 CSR 读写指令一起。与 TLBRD 不同的是，TLBWR 会修改 TLB 的内容，而取指时需要读 TLB 的内容，形成围绕 TLB 的写后读相关，也只能通过重取解决。

TLBFILL 指令：与 TLBWR 的不同仅仅在于修改的 TLB 项是指定的还是随机的，因此可直接复用写端口。其操作和避免相关的方式与 TLBWR 完全相同。

INVTLB 指令：由于其操作数全部由指令码携带的信息得到，不需要与 CSR 交互，因此不需要考虑围绕 CSR 的写后读相关，可以在 EXE 复用访存查找端口。又由于其会修改 TLB 的内容，因此存在围绕 TLB 的写后读相关，需要通过重取解决。

总结一下。这五条指令，尽量都安排在 WB 级读写 CSR，以避免围绕 CSR 的写后读相关。唯一做不到的就是 TLBSRCH 需要在 EXE 读 CSR 的内容，但可以通过阻塞解决。这五条指令中，后三条都会修改 TLB 的内容，存在围绕 TLB 的写后读相关，TLBRD 会修改 CSR.ASID 的内容。因此这四条指令都会影响到后续指令的取指，需要通过取消重取机制解决。

在每个流水级里需要做的事情是：在 EXE 级执行 INVTLB 或 TLBSRCH，在 WB 级写回 CSR。在 WB 级执行 TLBRD、TLBWR 和 TLBFILL，并在 WB 级的下一拍真正写入，但实际上写入的过程是在 CSR 模块中完成。

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。

特权状态控制寄存器 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，则采用直接窗口映射模式，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，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。

这一部分的设计主要封装在了 MMU 模块中，将会在后文 MMU 模块设计描述介绍更多具体细节。

另外，为了实现直接窗口映射模式和地址翻译模式，还需要在 CSR 模块中完成对 CSR.DMW0-1 的写行为，以

及对 CSR.CRMD 特权寄存器中的 DA、PG、DATF、DATM 域的写入。这部分的实现需要在 CSR 模块中补全相应的寄存器和对应的读写逻辑。这部分将在对 CSR 模块的设计中进行更细致的描述。

（2）**TLB 例外处理**

对于例外处理，本实验要增添对 ADEM、PIF、PIL、PIS、PPI、PME 和 TLBR 例外的处理。新增的这些例外可以支持在之前的实验实现的流水线传递中实现数据通路，但是需要注意到 ADEF 取指和 ADEM 指令地址错误仅仅指取指地址不对齐，但在本实验还要考虑取指地址不在当前特权等级所允许的取指范围内的例外。另外对于 TLB 产生的例外相关还需要单独进行考虑。

这里强调一下 TLB 重填例外，该例外是指在页表映射模式下没有命中 TLB，但**该例外的处理地址入口与其他例外处理入口是不同的，需要单独考虑，因此需要更改 CSR 模块中的例外入口地址的逻辑**。对于其他例外，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。**同时，由于只有 load 和 store 指令需要进行访存和查找 TLB 操作，所以只有在遇到这两种指令触发的 TLB 例外是有效的**。

需要注意的是，在 EXE 级如果触发了 ADEM 或 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_en。既然无法向总线发起请求，所以无法等到总线发来的 addr_ok 和 data_ok 信号，因此需要修改 EXE 级和 MEM 级的 ready_go 信号，使得流水线能够正常工作。

此外，在处理例外逻辑时，还要注意这些例外的优先级。IF 阶段触发的例外优先级最高，ID 阶段次之，EXE 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 TLB 相关例外中，TLBR 的优先级最高。否则就会报告错误的例外状态。

（二）重要模块设计：TLB 模块

1、工作原理

单个 TLB 模块的设计根据需要分为读、写和查找三部分，且同时支持取指和访问同时进行，因此有两套查找端口。TLB 模块内部是一个二维组织结构的查找表，查找表的每一项分为两部分，一部分存储的信息既参与读写又参与查找比较，另一部分只参与读写。

2、接口定义

表格 1：TLB 接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
取指查找端口			

名称	方向	位宽	功能描述
s0_vppn	IN	19	来自访存虚地址的 31..13 为
s0_va_bit12	IN	1	来自访存虚地址 12 位
s0_asid	IN	10	来自 CSR.ASID 的 ASID 域
s0_found	OUT	1	用于判定是否产生 TLB 重填异常
s0_index	OUT	4	记录命中在查找表的第几项
s0_ppn	OUT	20	用于产生最终的物理地址
s0_ps	OUT	6	用于产生最终的物理地址
s0_plv	OUT	2	和 s_found 一起判定是否产生页特权等级不合规异常
s0_mat	OUT	2	TLB 模块输出信息，存储访问类型
s0_d	OUT	1	和 s_found、s_d 一起判定是否产生页修改异常
s0_v	OUT	1	和 s_found 一起判定是否产生页无效异常
访存查找端口，端口功能与取指查找端口功能相同			
s1_vppn	IN	19	来自访存虚地址的 31..13 为
s1_va_bit12	IN	1	来自访存虚地址 12 位
s1_asid	IN	10	来自 CSR.ASID 的 ASID 域
s1_found	OUT	1	用于判定是否产生 TLB 重填异常
s1_index	OUT	4	记录命中在查找表的第几项
s1_ppn	OUT	20	用于产生最终的物理地址
s1_ps	OUT	6	用于产生最终的物理地址
s1_plv	OUT	2	和 s_found 一起判定是否产生页特权等级不合规异常
s1_mat	OUT	2	TLB 模块输出信息，存储访问类型
s1_d	OUT	1	和 s_found、s_d 一起判定是否产生页修改异常
s1_v	OUT	1	和 s_found 一起判定是否产生页无效异常
invtlb_valid	IN	1	指令有效
invtlb_op	IN	5	标识 invtlb 的具体操作类型
we	IN	1	写使能输入信号
w_index	IN	4	写地址
w_e	IN	1	写入的有效性标志
w_vppn	IN	19	写入的虚双页号
w_ps	IN	6	写入的页大小
w_asid	IN	10	写入的地址空间标识（区分不同进程中同样的虚地址）
w_g	IN	1	写入的全局标志位（1 时不进行 ASID 一致性检查）
w_ppn0	IN	20	写入的物理页号
w_plv0	IN	2	写入的特权等级
w_mat0	IN	2	写入的存储访问类型
w_d0	IN	1	写入的脏位
w_v0	IN	1	写入的有效位
w_ppn1	IN	20	写入的物理页号
w_plv1	IN	2	写入的特权等级
w_mat1	IN	2	写入的存储访问类型
w_d1	IN	1	写入的脏位
w_v1	IN	1	写入的有效位
r_index	IN	4	读地址
r_e	OUT	1	输出读的有效性标志
r_vppn	OUT	19	输出读的虚双页号
r_ps	OUT	6	输出读的页大小
r_asid	OUT	10	输出读的地址空间标识

名称	方向	位宽	功能描述
r_g	OUT	1	输出读的全局标志位
r_ppn0	OUT	20	输出读的物理页号
r_plv0	OUT	2	输出读的特权等级
r_mat0	OUT	2	输出读的存储访问类型
r_d0	OUT	1	输出读的脏位
r_v0	OUT	1	输出读的有效位
r_ppn1	OUT	20	输出读的物理页号
r_plv1	OUT	2	输出读的特权等级
r_mat1	OUT	2	输出读的存储访问类型
r_d1	OUT	1	输出读的脏位
r_v1	OUT	1	输出读的有效位

3、功能描述

（1） 读写功能设计

由于 TLB 模块的读写方式与寄存器堆的读写方式类似，因此一样是同步写异步读，只需将 TLB 的相关信息使用非阻塞赋值写入或使用组合逻辑读出即可。其中需要注意由于在 LoogArch 精简版中只支持 4KB 和 4MB 两种页大小，TLB 模块内部只用 1bit 来存放页大小信息，因此需要进行一个简单的转换。

```
assign r_ps    = tlb_ps4MB[r_index] ? 6'd21 : 6'd12;
```

（2） 查找功能设计

在进行查找时，查找流程并非串行化，而是同时比较所有项，在讲义中已经给出了代码，我们在这里使用 generate 进行代码的缩减。要判断是否匹配，则对每一个页表项，比较 vppn 的高十位，而后根据页大小是 4MB 还是 4KB 判断是否需要比较 vppn 的低十位。具体来说，是在 LoogArch 精简版中，每个页表项存放了相邻的一奇偶相邻页表信息，所以 TLB 页表项中存放虚页号的是系统中虚页号/2 的内容，也就是 2MB 大小只需比较 32-21-1=10 位，4KB 只需比较 32-12-1=19 位。虚页号的最低位不需要存放在 TLB 中，查找 TLB 时在根据被查找虚页号的最低位决定是选择奇数号页还是偶数号页的物理转换信息。

```
assign s1_odd = tlb_ps4MB[s1_index] ? s1_vppn[8] : s1_va_bit12;
```

最后比较全局标志位 g 和地址空间表示位 asid（区分不同进程中同样的虚地址），判断是否属于本进程。若以上全部满足，则说明命中。

```
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
```

最后对于 INVTLB 指令的支持，在讲义中也已经大致写出，将该指令个操作分成四种“子匹配”判断条件，

而后再进行组合。由于 op 为 5 位，因此设置的 invtlb_mask 为 32 位，需要将高 25 为置 0。

（三）重要模块设计：MMU 模块

1、工作原理

该模块的功能即为地址翻译模块，是一个组合逻辑电路块，其作用就是将输入的虚拟地址转化为物理地址，并输出对应的例外信息。需要注意到的是，对于 LOONGARCH 指令集，该 MMU 模块只向 TLB 模块内进行查找，而实际一些其他体系结构的 MMU 可能还需要通过全局页表进行逐级页表查找，最终访存到真正的物理地址。

2、接口定义

表格 2：MMU 接口定义

名称	方向	位宽	功能描述
flag	IN	2	地址类型，10：取指 01：取值
csr_crmd_rvalue	IN	32	当前模式信息
csr_asid_rvalue	IN	32	当前进程 ASID 信息
csr_dmw0_rvalue	IN	32	直接映射窗口 0
csr_dmw1_rvalue	IN	32	直接映射窗口 1
s_found	IN	1	访存通道查找是否命中
s_index	IN	4	访存通道查找得到的 index
s_ppn	IN	20	访存通道查找得到的 ppn 域
s_ps	IN	6	访存通道查找得到的 ps 域
s_plv	IN	2	访存通道查找得到的 plv 域
s_mat	IN	2	访存通道查找得到的 mat 域
s_d	IN	1	访存通道查找得到的 d 域
s_v	IN	1	访存通道查找得到的 v 域
va	IN	32	进行转换的虚地址
exc_ecode	OUT	6	虚地址翻译产生的例外信息
dmw_hit	IN	1	直接映射窗口是否命中
plv	IN	2	特权信息
pa	IN	32	转换后的物理地址
s_asid	IN	10	对应页表的 asid 信息

3、功能描述

根据通过 TLB 引出的 search 通道进行查找，虚拟地址同时送往直接地址翻译、直接映射窗口地址和 TLB 地址翻译三处，同时分别进行各自的虚实地址转换，最后如果是直接地址翻译模式，就选择直接翻译的结果，否则查看直接映射窗口是否命中，若命中则翻译结束，否则就选择 TLB 翻译的结果。

```
//paddr
assign pa =       ({32{direct}} & va) |
                  ({32{~direct & dmw_hit0}} & dmw_paddr0) |
                  ({32{~direct & ~dmw_hit0 & dmw_hit1}} & dmw_paddr1) |
                  ({32{~direct & ~dmw_hit0 & ~dmw_hit1}} & tlb_paddr);
```

如果 TLB 未命中或特权状态和读写行为出错则触发例外，将例外信息传送至 IF 流水级和 EXE 级进行处

理。flag 信息是控制例外信息的状态，因为 IF 级和 EXE 级的地址翻译有些许不同，主要是 IF 级不进行写行为，不会出现 pis，pil 和 pme 错，这可以通过 flag 信号来控制。

```
assign ecode_pif = flag[1] & tlb_trans & ~s_v;
assign ecode_ppi = tlb_trans & ((csr_crmd_plv > s_plv));//
assign ecode_tlbr = tlb_trans & ~s_found;
assign ecode_pil = flag[1]?1'b0:tlb_trans & ~s_v;
assign ecode_pis = flag[1]?1'b0:tlb_trans & ~s_v;
assign ecode_pme = flag[1]?1'b0:tlb_trans & ~s_d;
```

输出的 dmw_hit 是为了生成访存地址错例外，这一部分的实现如下：

虚拟地址第 31 位是为了区分内核态地址或者用户态地址，它们对应特权访问等级不同。

```
assign es_adem = es_need_mem & es_alu_result[31] & (plv == 2'd3) & ~(dmw_hit);
assign fs_adef = fs_pc[1] | fs_pc[0] |(plv == 2'd3 & fs_pc[31] & ~(dmw_hit));
```

将该模块例化在 my_cpu_core 模块内，分别连接 IF 模块和 EXE 模块，承担地址翻译工作，减少 IF 和 EXE 级排线，也更方便对 MMU 进行功能扩展。对于复用 EXE 访存数据通路的 TLBSRCH、TLBSRCH、TLBRD、TLBWR、TLBFILL、INVTLB 等指令可以通过更改 MMU 接口，增加一些额外的控制逻辑即可实现，这样就可以在 IF 级或者 EXE 级来进行控制，将数据延流水线进行传递，最终在 CSR 模块中写入。不过，单独为这几条指令更改 MMU 模块的逻辑实在有点得不偿失，所以 exp19 进行封装的时候并未考虑将这些指令移至 MMU 模块中实现，而是继续沿用在 exp18 中单独在 EXE 级设计的数据通路。

（三）重要模块设计：CSR 模块

1、工作原理

该实验的 CSR 主要更改是新增虚实地址转换及 TLB 相关例外处理的寄存器，并直接控制 TLB 的更新逻辑。

2、接口定义

表格 3：CSR 接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟输入
reset	IN	1	复位信号
ertn_flush	IN	1	ertn 指令刷新信号
wb_ex	IN	1	例外发生信号
ws2csr_bus	IN	210	WB 流水级传递给 CSR 模块的数据
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
ertn_entry	OUT	32	例外返回入口地址
has_int	OUT	1	发生中断信号
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

名称	方向	位宽	功能描述
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
r_index	OUT	4	TLB 表项索引，即 CSR.TLBIDX.INDEX，
w_index	OUT	4	TLB 进行写操作的索引，随机写或者指定表项
we	OUT	1	TLB 写使能信号
csr_asid_asid	OUT	32	TLB 表项 ASID 号，用于判断例外
csr_tlbehi_vppn	OUT	32	TLB 表项虚拟地址高位
csr_tlbidx_index	OUT	32	TLB 表项索引

3、功能描述

对于虚实转换的翻译逻辑前文已经反复描述过，在此不再赘述。

这里主要强调的是，TLB 是直接与 CSR 模块相连的，不需要经过 WB 流水级，如果某个流水级需要 TLB 中的接口，则直接引出即可，例如在 EXE 级的数据通路。因此我们需要将 WB 中关于更新 TLB 表项的指令控制信息交给 CSR 模块，这部分只需要沿用 ws2csr_bus 即可，具体来说就是将 csr_tlb_ctrl 从 bus 中译码出来，然后在对具体的控制信号进行译码，之后直接与 TLB 对应的接口相连即可。

```
wire we;
wire [ 3:0] w_index;
wire tlbrd_we;
wire [ 3:0]r_index;
wire [ 1:0]tlbwe_op;//10:tlbfill;01:tlbwrite
wire tlbsrch_we;
wire tlbsrch_hit;
wire [ 3:0]tlbsrch_hit_index;
assign {tlbrd_we, tlbwe_op, tlbsrch_we, tlbsrch_hit, tlbsrch_hit_index}=csr_tlb_ctrl;
assign we = |tlbwe_op;
assign r_index = csr_tlbidx_index;
assign w_index = {4{tlbwe_op[0]}}& csr_tlbidx_index | {4{tlbwe_op[1]}}& rand_idx;
```

另外，前文提到 TLB 重填指令的入口地址与其他例外入口地址是不同的，因此需要进行一个判断。

在 WB 级译码出来的 tlb_ entry_en 信号进行二路选择,tlb_entry_en 信号即重填例外对应的 ecode 位拉高。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。例外入口地址选择逻辑为：

```
assign ex_entry = tlb_entry_en? tlb_ex_entry:csr_eentry_rvalue;
assign tlb_ex_entry = csr_tlbrentry_rvalue;
```

另外还需要考虑到的是，由于引入了 tlb 模块，csr_crmd 特权寄存器的写入模式也需要重新考虑。对于 tlb 模块的更新，则直接对 tlb 引出的接口赋值即可。

```
    // TLB entry
assign w_tlb_e    = ~csr_tlbidx_ne;
assign w_tlb_ps   =  csr_tlbidx_ps;
assign w_tlb_vppn =  csr_tlbehi_vppn;
assign w_tlb_asid =  csr_asid_asid;
assign w_tlb_g    =  csr_tlbelo0_g & csr_tlbelo1_g;

assign w_tlb_ppn0 = csr_tlbelo0_ppn[19:0];
assign w_tlb_plv0 = csr_tlbelo0_plv;
assign w_tlb_mat0 = csr_tlbelo0_mat;
assign w_tlb_d0   = csr_tlbelo0_d;
assign w_tlb_v0   = csr_tlbelo0_v;

assign w_tlb_ppn1 = csr_tlbelo1_ppn[19:0];
assign w_tlb_plv1 = csr_tlbelo1_plv;
assign w_tlb_mat1 = csr_tlbelo1_mat;
assign w_tlb_d1   = csr_tlbelo1_d;
assign w_tlb_v1   = csr_tlbelo1_v;
```

（四）重要模块设计：其余流水级模块

1、IF 级：

对于 IF 级，对于该实验来说没有什么变化，主要是需要考虑因为引入 TLB 而产生的 adef 例外，

```
即 assign fs_adef = fs_pc[1] | fs_pc[0] |(plv == 2'd3 & fs_pc[31] & ~(dmw_hit));
```

其余的只需要将 MMU 翻译产生的 fs_exc_ecode 延流水级向下传递即可。另外 IF 级实际上我们并未考虑到取指例外产生后取消取指指令的请求，我们默认每一条指令的地址都是合法的，否则流水线将无法正常运转，这与 EXE 流水级的数据 SRAM 有些许不同。

2、ID 级：

对于 ID 级，译码逻辑需要做的事情主要是将几条新增的 TLB 刷新指令进行译码，并且生成其写后读相关的重取信号：

```
assign ds_refetch_flg = inst_tlbfill || inst_tlbwr || inst_tlbrd || inst_invtlb ||
```

```
                        (ds_csr_we) && (ds_csr_num == `CSR_CRMD || ds_csr_num ==
`CSR_ASID);
assign invtlb_op = rd;
```

这里的重取是无视具体冲突的重取，即只要有一条可能产生冲突的指令，在 EXE 流水级就不加以区分地进行阻塞，这样考虑主要是一方面这些指令的频率不高，产生写后读相关阻塞造成的损失较小，不必大费周章去具体地考虑阻塞控制。

3、EXE 级：

EXE 级是改动较大的部分，TLBSRCH、INVTLB 指令主要是在 EXE 级实现，TLBSRCH 产生的阻塞也将在 EXE 级进行处理。前文已述，该指令是在 EXE 级从 TLB 单独引出接口实现的，但实际上它也应该在 MMU 中复用其与 TLB 交互的接口，不过我们并未如此设计（原因前文已述）。

更进一步地，现在 EXE 流水级的控制信号变得更加复杂

```
assign es_ready_go = es_need_mem ? (es_reflush || es_finish || data_sram_en &&
data_sram_addr_ok && !tlbsrch_blk|| (|es_tlb_ecode) || es_adem)
                                   : (es_reflush || alu_flag && es_valid
&& !tlbsrch_blk);
```

现在需要考虑 TLBSRCH 指令产生的阻塞，以及发生例外时将 EXE 流水线刷新。

对 EXE 级还需要考虑对访存 SRAM 接口的控制，也就是错误的取值地址不应该发送读写请求。

```
assign data_sram_en    =  ~es_finish && es_need_mem && ms_allowin &&~(|es_tlb_ecode) &&
~es_adem;
```

除此之外就是和 IF 级一样将从 MMU 产生的例外向后传递。

4、MEM 级：

MEM 流水级没有什么改动，它主要是向 EXE 流水级产生一个前递，以指导其发生写后读相关时进行阻塞。

```
assign ms_csr_tlbrd = ( ( mem_csr_num == `CSR_ASID || mem_csr_num == `CSR_TLBEHI) &&
(mem_csr_we)|| ms_inst_tlbrd) && ms_valid;
```

5、WB 级：

WB 级是新增的 TLBSRCH、TLBRD、TLBWR、TLBFILL、INVTLB 等指令真正与 TLB 进行交互的位置，

```
wire tlbrd_we;
// tlbrd
assign tlbrd_we = ws_inst_tlbrd;

// tlbwr and tlbfill
assign tlbwe_op[0] = ws_inst_tlbwr;
assign tlbwe_op[1] = ws_inst_tlbfill;

// tlbsrch
assign tlbsrch_we         = ws_inst_tlbsrch;
assign tlbsrch_hit        = ws_tlbsrch_hit;
assign tlbsrch_hit_index  = ws_tlbsrch_hit_index;
```

```
assign ws_csr_tlbrd = ( ( ws_csr_num == `CSR_ASID || ws_csr_num == `CSR_TLBEHI) &&
(ws_csr_we)|| ws_inst_tlbrd) && ws_valid;
```

WB 产生的控制信号，通过 ws2csr_bus 传递给 CSR 模块，CSR 模块再利用这些控制信号实现对 TLB 表项的读写和刷新。这部分已经在 CSR 模块中提到。

```
assign ws_csr_tlb_ctrl = {tlbrd_we, tlbwe_op, tlbsrch_we, tlbsrch_hit, tlbsrch_hit_index};
```

另外，WB 流水级和 MEM 流水级均需要考虑 tlbsrch 产生的写后读相关，也需要向 EXE 流水级发送一个前递信号以指挥其进行阻塞。

三、实验过程（50%）

（一）实验流水账

2023/11/30 18：00 开始撰写 exp17 部分代码，仿真及上板通过后完成该部分实验

2023/12/09 23：00 exp18 仿真及上板通过

2023/12/10 18:00 开始 exp19 实验代码

2023/12/10 23:00 完成 exp19 代码部分，开始仿真调试

2023/12/11 21:00 调试结束，仿真上板均通过

2023/12/12 19:00 重新设计 MMU 模块，完成 exp19-v2

2023/12/12/23:00 exp19 仿真上板测试均通过。

（二）错误记录

EXP17：

1、错误 1：页大小错误

（1）错误现象

```
OK!!!write
======================
----FAIL!!!
read_test_id is  0
s0_test_id is  0
s1_test_id is  1
```

在新的测试文件中进行测试，发现原代码无法通过。

（2）分析定位过程

根据老师在群里所发，结合指令集手册，可以看出原来对于 4MB 页大小理解出现问题。通过指令集手册中所说，龙芯架构 32 位精简版只支持 4KB 和 4MB 两种页大小，对应 TLB 表项中的 PS 值分别是 12 和 21。其中 4MB 页大小对应的是透明大页的页表项，其在填入 TLB 过程中等分为 2 个 2MB 大小相同页表属性的表项。因此需要将

4MB 页大小相关的代码进行修改。

（3）修正效果

首先是检索匹配项时，4MB 页大小需要匹配高十位而不是高九位。而后对于判断奇偶页，是对于虚地址的第 22 位进行判断，也就是 vppn 的第九位。

```
assign s0_odd = tlb_ps4MB[s0_index] ? s0_vppn[8] : s0_va_bit12;
```

在 cond 匹配条件时，需要更改 cond4 的匹配项，与 match 的更改类似。

```
assign cond[i][3] = (s1_vppn[18:9] == tlb_vppn[i][18:9]) && (tlb_ps4MB[i] || s1_vppn[8:0] == tlb_vppn[i][8:0]);
```

最后要更改所有 ps 域的赋值，当是 4MB 页大小时，ps 赋值为 21。

EXP18：

2、错误 2：INVTLB 指令出错

（1）错误现象

```
------------------------------------------------------------
[8421077 ns] Error!!!
    reference: PC = 0x1c078d60, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c078d60, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x0002a000
------------------------------------------------------------
```

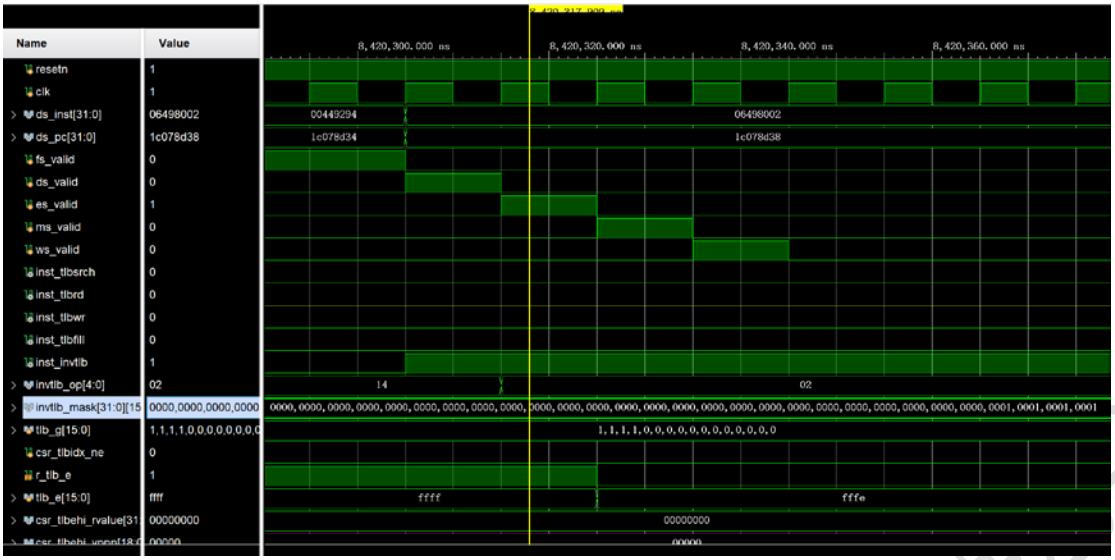
指令 0x1c078d60 的写回值错误。

（2）分析定位过程

出错的指令是 TLBRD 之后的 CSRRD，要读的寄存器是（0x11）TLBEHI。本该读出来的是 0x00000000，实际读出来的是 0x0002a000。也就是说，TLBEHI.VPPN 本该是 0，而错误地存为了 10101（0x15）。

继续往前找，正好执行 TLBRD 时，所读取表项的 VPPN 被写在这里，所以一定是前一个 TLBRD 的 VPPN 错了。TLBRD 在 WB 级的下一拍将 VPPN 写入 CSR.TLBEHI.VPPN，经追溯，确实在上一个 TLBRD 执行到 WB 的下一拍，CSR.TLBEHI.VPPN 的值变为 0x0002a000。再追溯，TLB 的 tlb_vppn[12]确实是 0x15。那么就只有可能是上次写或重填或者 INVTLB 出错，导致 TLB 里的内容错误。

最近一次出现的对 index=12 的修改是一次 TLBWR，确实从 CSR 写入的是 0x15，那只有可能是再上一次的对 11 的 CSR 写指令错了。但这是第一次对 11 号 CSR 寄存器的指令。经检查，上一次重填指令也没什么问题。因此，最有可能是之前的某次 INVTLB 指令出错。



追溯到上图所示的 INVTLB 指令，invtlb_op 的号是 02，清掉所有 g=1 的页表项。看到 tlb_g 是 1111000000000000，那么 invtlb_mask[2]就应该是 f000，清掉的就是 12~15 位，tlb_e 理应变成 0fff。那之后去查 tlb 第 12 项的时候读出来的就会是 0。这指向错误应当出现在 TLB 模块中。

（3）错误原因

```
// invtlb
wire [3:0] cond [TLBNUM - 1:0];
wire [TLBNUM - 1:0] invtlb_mask [31:0];
generate for(i = 0; i < TLBNUM; i = i + 1) begin
        assign cond[i][0] = ~tlb_g[i];
        assign cond[i][1] = tlb_g[i];
        assign cond[i][2] = s1_asid == tlb_asid[i];
        assign cond[i][3] = (s1_vppn[18:9] == tlb_vppn[i][18:9]) && (tlb_ps4MB[i] || s1_vppn[8:0] == tlb_vppn[i][8:0]);
```

TLB 模块中，cond 的位宽和个数写反了。正确写法应当是，32 个 16 位掩码和 4 个 16 位 cond。

（4）修正效果

```
// invtlb
wire [TLBNUM - 1:0] cond [3:0];
wire [TLBNUM - 1:0] invtlb_mask [31:0];
generate for(i = 0; i < TLBNUM; i = i + 1) begin
        assign cond[0][i] = ~tlb_g[i];
        assign cond[1][i] = tlb_g[i];
        assign cond[2][i] = s1_asid == tlb_asid[i];
        assign cond[3][i] = (s1_vppn[18:9] == tlb_vppn[i][18:9]) && (tlb_ps4MB[i] || s1_vppn[8:0] == tlb_vppn[i][8:0]);

end
endgenerate
```

将 TLB 模块中 invtlb 的部分修改为如上样子。

```
// TLBEHI
always @ (posedge clk) begin
    if (reset) begin
        csr_tlbehi_vppn <= 19'b0;
    end else if (tlbrd_we) begin
        csr_tlbehi_vppn <= r_tlb_e ? {19{r_tlb_e}} & r_tlb_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
```

将 CSR 模块中 TLBEHI 部分的 tlbrd_we 有效时，添上根据 r_tlb_e 的选择。

```
----[8600425 ns] Number 8'd70 Functional Test Point PASS!!!
       [8602000 ns] Test is running, debug_wb_pc = 0x1c0102bc
==============================================================
Test end!
----PASS!!!
$finish called at time : 8602885 ns : File "D:/CA_local_depository/lab8_exp18/mycpu_env/soc_verify/soc_axi/testbench/mycpu_tb.v" Line 267
run: Time (s): cpu = 00:14:33 ; elapsed = 00:15:33 . Memory (MB): peak = 2788.277 ; gain = 0.000
```

重新仿真，仿真通过。

（5）归纳总结

之所以会出现这个历史遗留错误，是因为之前的 TLB 模块单独的测试文件中，没有再 INVTLB 指令进行了错误的清除之后，用一条 CSR 读指令将其读出到寄存器中，从而没有检测到错误。这同时也反应了，在出现了越来越多的不写回的特权指令后，由于金标准只比对写回寄存器的值是否正确，debug 就越来越难，比如像这个 bug 一样跨越了两条指令才最终定位到 bug。

EXP19：

3、错误 3：CSR 模块未实现对特权寄存器的写操作

（1） 错误现象

写数据与参考值不符合。

```
------------------------------------------------------------
[8622537 ns] Error!!!
    reference: PC = 0x1c07bdfc, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000088
    mycpu    : PC = 0x1c07bdfc, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000008
------------------------------------------------------------
```

（2） 分析定位过程

查看测试文件，该指令是一条 csrxchg 指令，读写 CSR_CRMD 特权寄存器，可以看到参考值为 0x00000088,这表明当前映射地址模式为 DA,直接地址翻译模式下取指操作的访问类型，已指出软件置 PG 为 1 时，需要将 DATF 域置为 0b01，猜测是 csr 模块遗忘新增该逻辑。

4	PG	RW	映射地址翻译模式的使能，高有效。 当触发 TLB 重填例外时，硬件将该域置为 0。 当执行 ERTN 指令从例外处理程序返回时，如果 CSR.ESTAT.Ecode=0x3F，则硬件将该域置为 1。 PG 位和 DA 位的合法组合情况为 0、1 或 1、0，当软件配置成其它组合情况时结果不确定。
6:5	DATF	RW	直接地址翻译模式时，取指操作的存储访问类型。 当软件将 PG 置为 1 时，推荐同时将 DATF 域置为 0b01，即一致可缓存类型。
8:7	DATM	RW	直接地址翻译模式时，load 和 store 操作的存储访问类型。 当软件将 PG 置为 1 时，推荐同时将 DATM 置为 0b01，即一致可缓存类型。

```
111397    1c07bdf8:    03807c0d    ori $r13,$r0,0x1f
111398    1c07bdfc:    040001ac    csrxchg $r12,$r13,0x0
111399    1c07be00:    03800c11    ori $r17,$r0,0x3
```

（3）错误原因

查看 csr 模块，发现确实未考虑到对 crmd 寄存器进行写操作时的情况。应该进行更改。

```
// DA, PG, DATF, DATM domains of CRMD: consider reset and write
always @(posedge clk) begin
    if(reset) begin
        csr_crmd_da   <= 1'b1;
        csr_crmd_pg   <= 1'b0;
        csr_crmd_datf <= 2'b0;
        csr_crmd_datm <= 2'b0;
    end
    else if (csr_we && csr_estat_ecode == 6'h3f) begin // ?
        csr_crmd_da   <= 1'b0;
        csr_crmd_pg   <= 1'b1;
        csr_crmd_datf <= 2'b01;
        csr_crmd_datm <= 2'b01;
    end
end
```

（4）修正效果

修改如下：

```
// DA, PG, DATF, DATM domains of CRMD: consider reset and write
always @(posedge clk) begin
    if(reset) begin
        csr_crmd_da   <= 1'b1;
        csr_crmd_pg   <= 1'b0;
        csr_crmd_datf <= 2'b0;
        csr_crmd_datm <= 2'b0;
    end else if (wb_ex && wb_ecode == `ECODE_TLBR) begin
        csr_crmd_da <= 1'b1;
        csr_crmd_pg <= 1'b0;
    end else if (ertn_flush && csr_estat_ecode == `ECODE_TLBR) begin
        csr_crmd_da <= 1'b0;
        csr_crmd_pg <= 1'b1;
    end 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
end
```

修改后该测试点仿真通过。

4、错误 4：TLB 异常处理测试异常

（1）、错误现象

PC 与金标准中的参考 PC 值不符合。

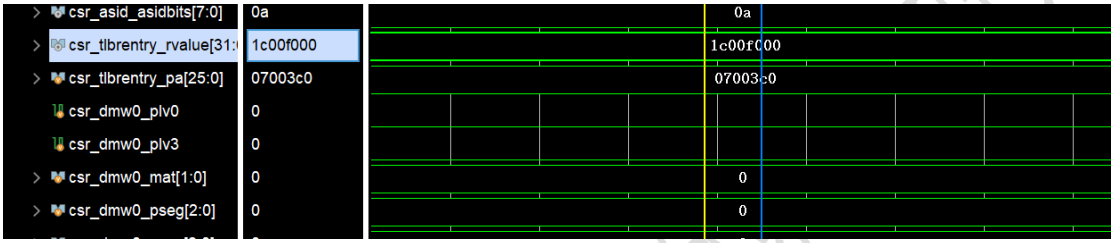
```
--------------------------------------------------------------
[8623057 ns] Error!!!
    reference: PC = 0x1c00f000, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c07be14, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
--------------------------------------------------------------
```

（2）、分析定位过程

PC 出现错误，0x1c07be14 是一条 load 指令，上一条指令 0x1c07be10 是一个立即数加指令，而参考的 PC 值 0x1c00f000 是一条异常处理指令。

```
1c07be10:    0280237b    addi.w  $r27,$r27,8(0x8)
1c07be14:    288003ee    ld.w    $r14,$r31,0
```

并且是 tlb 异常处理入口，观察 csr 波形可以看到此时的 tlb_entry 正是 0x1c00f000,



这表明该指令会触发一个 tlb 异常，此时将会把异常处理入口设置为 tlb_entry，显然此时并未触发 tlb 异常。猜测：查看测试源文件，该测试源文件的原理就是通过修改 csr 寄存器改变 tlb 的表项，执行 ld.w 时会触发一个 tlb refill，从而转到中断处理地址。既然当前没有触发错误，这表明 tlb 错误地命中了。检查 TLB 其行为均符合预期，TLB 进行重填测试时会首先执行一条 invtlb 指令，该指令将 TLB 清空后，该测试文件 TLB 重填了 15 次，也就是说 TLB 中表现已满，接下来合理的行为应该是 tlb 未命中导致触发 tlb 重填。

```
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r
FILL_TLB_ITEM_r

//set dotm=1
li.w      t0, 0xa0
li.w      t1, 0x188
csrxchg t0, t1, csr_crmd

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

## TLB refill error
li.w      t5, 0x3
lu12i.w s7, 0xe
li.w s8, 0x200ff << 13
la.local s4, 1f

ld.w  t2, s8, 0x0
bne   s2, s7, inst_error
LI_EXIMM(t0,s2,INM_KERNEL)
syscall 0   //return to kernel mode
```

之所以没有报出重填异常，可能的原因显然只有两种，一是 TLB 行为异常，但已排除，二是传给 TLB 的虚拟地址是错误的。

（3）、错误原因

重新查看 EXE 阶段，终于发现，错误的原因是因为 exp18 时进行 invtlb 指令进行的匹配时，未更新其赋值逻辑，也就是说传给 s1_va_highbits 的指令与访问地址无关，而由于测试刚开始未出现访存指令，所以并未报错。

```
assign s1_va_highbits = invtlb_valid ? es_rkd_value[31:12] : {csr_tlbehi_vppn, 1'b0};
assign s1_asid        = invtlb_valid ? es_rj_value [ 9: 0] : csr_asid_asid;
```

将其赋值更新为：

```
assign s1_va_highbits = es_need_mem ? es_alu_result[31:12] : invtlb_valid ? es_rkd_value[31:12] : {csr_tlbehi_vppn, 1'b0};
assign s1_asid = (es_need_mem | ~es_need_mem & ~invtlb_valid) ? csr_asid_asid : es_rj_value[9:0];
```

（4）、修正效果

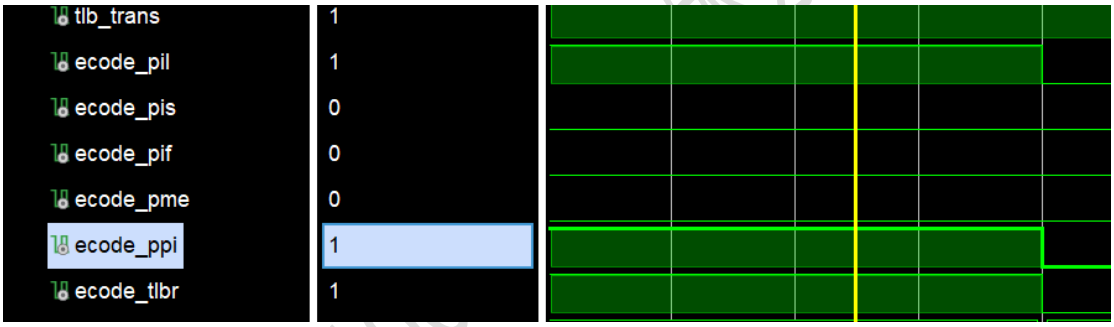
该测试点不再报错，转而出现 pc 不更新的情况。

5、错误 4（续）：TLB 异常处理测试异常

（1）、分析定位过程

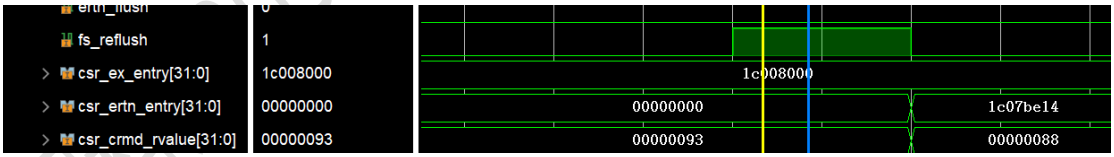
```
----[8600425 ns] Number 8'd70 Functional Test Point PASS!!!
        [8602000 ns] Test is running, debug_wb_pc = 0x1c0102b8
        [8612000 ns] Test is running, debug_wb_pc = 0x1c07bb6c
        [8622000 ns] Test is running, debug_wb_pc = 0x1c07bddc
        [8632000 ns] Test is running, debug_wb_pc = 0x1c07be18
        [8642000 ns] Test is running, debug_wb_pc = 0x1c07be18
        [8652000 ns] Test is running, debug_wb_pc = 0x1c07be18
        [8662000 ns] Test is running, debug_wb_pc = 0x1c07be18
        [8672000 ns] Test is running, debug_wb_pc = 0x1c07be18
        [8682000 ns] Test is running, debug_wb_pc = 0x1c07be18
        [8692000 ns] Test is running, debug_wb_pc = 0x1c07be18
```

继续查看该指令执行的情况



（2）、错误原因

发现出现了三个异常，而我们需要的异常应该是 ecode_tlbr，最后传给 IF 阶段的异常入口地址应该 0x1c00f000，但是可以看到此时异常处理入口是 1c008000，检查 CSR 模块，发现其最后选择入口地址的赋值出现错误。



将其修改为：

```
assign ex_entry = tlb_entry_en? tlb_ex_entry:csr_eentry_rvalue;
assign tlb_ex_entry = csr_tlbrentry_rvalue;
```

之所以 pc 不更新就是因为出现异常，流水线被卡住，而异常处理程序并未找到跳转入口，异常无法被处理导致堵塞。

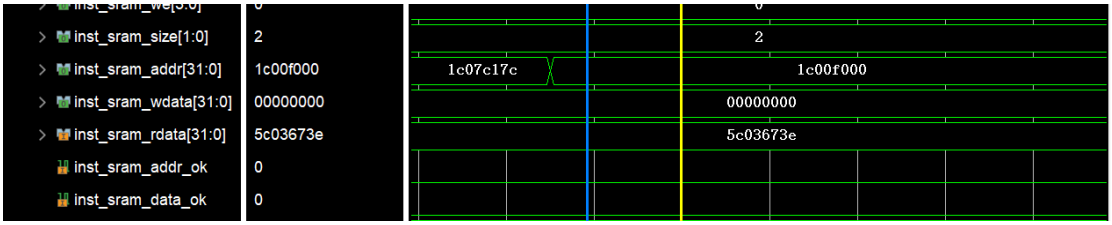
（3）、修正效果

修改后依然无法继续进行仿真。（这个 bug 真的有够长的>_<）

6、错误 4（续 2）：TLB 异常处理测试异常

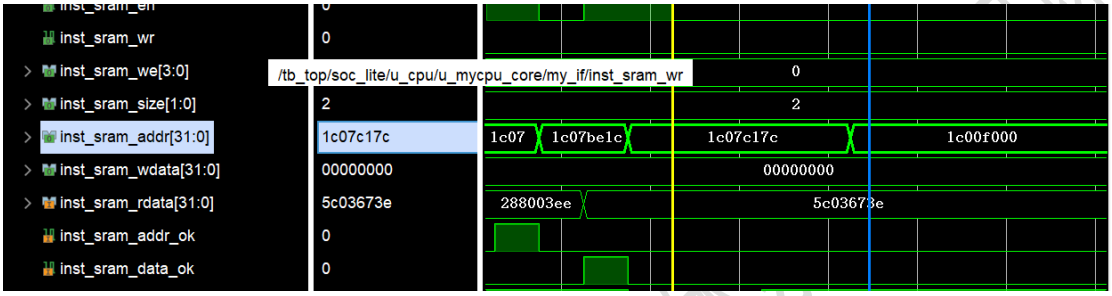
（1）、分析定位过程

继续检查，注意到之所以无法继续进行仿真，是因为 addr_ok 一直不拉高，无法取指。



为什么不会拉高呢？重新检查转接桥模块的设计，状态进行转移的时候，发现不拉高是因为指令请求状态未开始，这是因为 inst_sram_en 未被拉高，而读使能信号未拉高又是因为 pfs_reflush 信号一直拉高，即 fs_reflush 信号一直拉高，但是正常情况下，这个信号理应只拉高一拍。

为什么会这样呢，这其实是 IFstage 遗留下来的问题，现在的状况刚好是一条跳转指令，然后它将跳转到 1c07c17c 这个指令入口地址，但是刚好由于之前出现的指令发生了 tlb 重填，异常处理入口这个时候发送给了 IFstage，但是前一条跳转指令尚未被 inst_sram 接收，但它遗留的 pfs_reflush 却一直拉高（拉高直到指令返回）。



（2）、错误原因

查看以下对 pfs_reflush 的赋值逻辑，发现其拉高的时的条件会有问题，只要有 br_taken 且~br_stall 时就会拉高，但实际上你必须保证 inst_sram_addr_ok 此时也是拉高的才是正确的逻辑，否则就会出现拉高之后被其他异常打断，而无法拉低最终阻塞住流水线。

```
always @(posedge clk) begin
    if(~resetn)
        pfs_reflush <= 1'b0;
    else if(inst_sram_en && (fs_reflush | br_taken &~ br_stall | (br_stall | fs_br_stall)&inst_sram_addr_ok))
        pfs_reflush <= 1'b1;
    else if(inst_sram_data_ok)
        pfs_reflush <= 1'b0;
end
```

修改其逻辑为：

```
always @(posedge clk) begin
    if(~resetn)
        pfs_reflush <= 1'b0;
    else if(inst_sram_en && (fs_reflush | ((br_taken &~ br_stall) | (br_stall | fs_br_stall))&inst_sram_addr_ok))
        pfs_reflush <= 1'b1;
    else if(inst_sram_data_ok)
        pfs_reflush <= 1'b0;
end
```

（3）、修正效果

即保证了地址被接收后才拉高 pfs_reflush 信号，修改后该测试点终于通过了。

（有意思的是，修改完这个 bug 后，跑到同一条指令所用的仿真时间大大减少了）

7、错误 5： st.w 存数指令出错

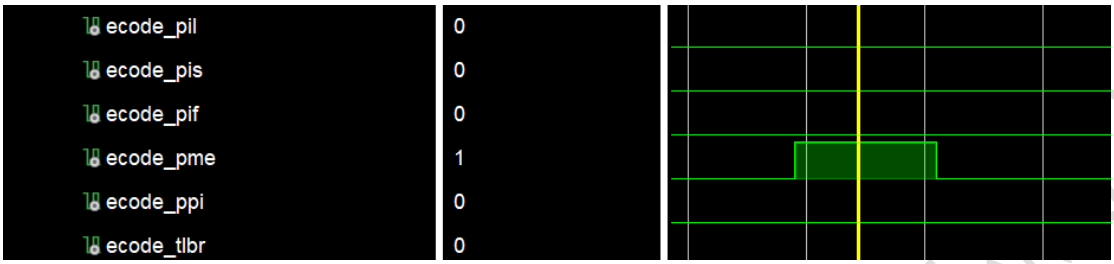
（1）、错误现象

取数指令 ld.w 取回数据与参考值不符合。

```
[8133087 ns] Error!!!
    reference: PC = 0x1c07bfcc, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x000000ff
    mycpu    : PC = 0x1c07bfcc, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000fff
```

（2）、分析定位过程

这是一条 ld.w 指令，读出来的数据是错误的，显然可能是因为存入了错误的值。查看上一条往同样地址写入值的 st.w 指令，按照测试点的位置，该测试将会引发一个 pme 错误，可以看到其已正确引发，但是显然不能往其中存入值，因为此时的页表已经是一个为脏的页表项。



这说明对 data_sram_en 的赋值逻辑错误。

（3）、错误原因

```
assign data_sram_en    = ~es_finish && es_need_mem && ms_allowin;//1'b1;
```

其未更新当存在异常的时候，不会往其中写入值。

修改如下：

```
assign data_sram_en    = ~es_finish && es_need_mem && ms_allowin &&~(|es_exc_ecode) && ~es_adem;//1'b1;
```

同时更新 ds_ready_go 信号为：

（4）、修正效果

修改后该测试点仿真通过。

8、错误 6： 访存地址错 BADV 寄存器出错

（1）、错误现象

BADV 寄存器值与预期不符合。

```
[8154317 ns] Error!!!
    reference: PC = 0x1c0084f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x401fe000
    mycpu    : PC = 0x1c0084f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
--------------------------------------------------------------
```

（2）、分析定位过程

这是一条 csrrd 指令，读取的特权寄存器为 CSR_BADV 寄存器，而这个寄存器是用来存异常地址，这说明异常地址的填写出现错误。

```
1c0084f8:    04001c0c    csrrd   $r12,0x7
1c0084fc:    5c6b619f    bne $r12,$r31,27488(0x6b60) # 1c00f05c <ex_finish>
1c008500:    0400440c    csrrd   $r12,0x11
1c008504:    5c6b599f    bne $r12,$r31,27480(0x6b58) # 1c00f05c <ex_finish>
1c008508:    14000079    lu12i.w $r25,3(0x3)
1c00850c:    506b5000    b   27472(0x6b50) # 1c00f05c <ex_finish>
```

当前指令是在测试 TLB page invalid error in inst fetch，这说明异常触发成功，但是异常地址未成功保存。

（3）、错误原因

这也是一个历史遗留问题，原因是存入 wrong_addr 时，之前仅仅考虑 IF 取指出现错误，所以在 exe_stage 只需要看 adef 有没有被置起即可，但是现在不能仅仅只考虑 adef 的情况，还可能因为取指地址触发 tlb 无效产生的例外也会需要传递 wrong_addr,因此修改

Exe_stage 的赋值逻辑为：

```
assign es_wrong_addr  = es_adef ? ds_wrong_addr : es_alu_result;
```

修改为：

```
assign es_wrong_addr  = ds_ex ? ds_wrong_addr : es_alu_result;
```

（4）修正效果

预感这是最后一个 bug, ٩(ᐛ)و （还真是）

修改后仿真测试通过！

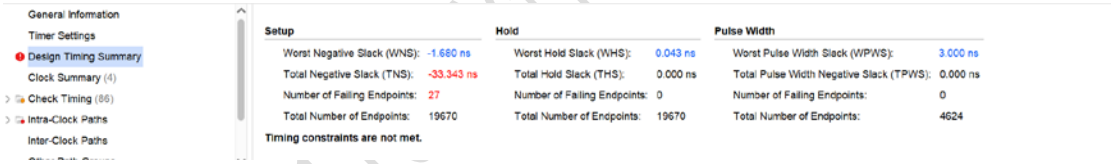
调试总结：

有了以往调试的经验，本次实验开始时写代码的时候就额外小心，尽力避免出现位宽错，未连线等等低级错误，因此这次实验确实没有一开始出现 0xxxxxxxxx 的报错 PC 了，算是伟大的一步。在进行 MMU 封装时，也出现了一些 bug，但是基本上都是没有什么代表性 的小问题，在此就不再指出了。

还需要提到的是，本次进行仿真调试的时候，参照了汇编源文件，能够清除地看到每个测试点是如何进行测试的，一旦理解了测试的原理，那么找到问题的源头就变得相对容易了起来。不过也很明显，这个测试文件无法测出某些潜在的小问题，严格来说也不能算是一个问题。

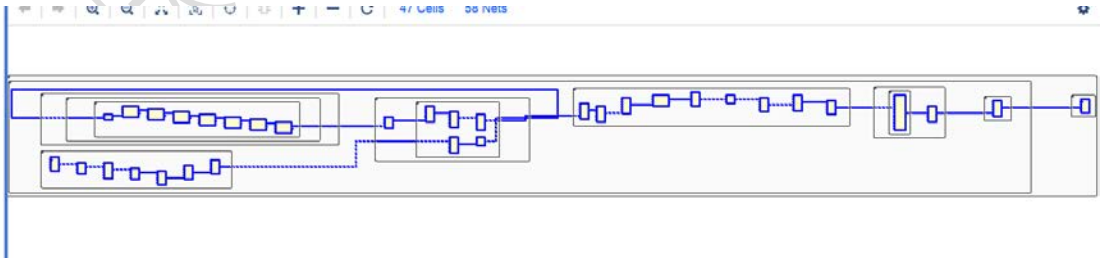
Exp18 中 tlbsrch 指令的阻塞是只考虑了 MEM 级是否有相关的指令，但是未考虑 WB 级，理论上来说如果每个流水级可以一拍内做完，那么 WB 级的指令是会存在写后读相关的。对应的测试文件确实是没有出现跨度两个流水级的写后读相关的指令的，当然就算有可能因为现在实现的 CPU 性能太烂也不会真正产生错误的结果。

时序：



Exp19 实验之后，时序变得更差了了些，最差的一条路径是：

这应该是在 IF 阶段引入对 tlb 的译码组合逻辑电路，导致时延大大地提升了。



后续将 MMU 单独封装成模块后：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.480 ns	Worst Hold Slack (WHS):	0.054 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	-36.135 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	99	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	19669	Total Number of Endpoints:	19669	Total Number of Endpoints:	4628

因为优化了许多组合逻辑电路，所以最长违约路径的时序变得稍微更好了一些。

总体来说并没有特别差，上板测试顺利通过。貌似其他组或多或少会出现调速不过的情况，但我测试数个种子，均未出现此问题，姑且认为 exp19 完美收官了。

四、实验总结（可选）

由于很多特权指令不写回通用寄存器，无法在一个指令之内判断正误。再加上测试文件不够完善，很多错误检查不出来，导致存在很多“隐患”在后面的某个实验中才会暴露出来，或者压根暴露不出来。这样增加了 debug 负担，也让代码留有不完备的隐患。希望之后给的测试文件能够更加完善。

就实验难度而言，这次实验并不算特别复杂，甚至没有上个实验那么棘手，不过如果没有理解页表机制，那么在实现 TLB 模块的设计上就会比较有难度，实现 MMU 模块也会一头雾水。虚实地址映射是操作系统和硬件协调完成的，操作系统负责建立页表映射关系，硬件则通过该关系来进行虚实地址转化，所以单独从硬件设计的角度上来考虑，比较摸不着头脑。因此也愈发觉得，体系结构研讨和操作系统研讨课非常适合搭配食用。