

# 1 CPU TLB MMU 支持

在学习并尝试本章节前，你需要具有以下环境和能力：

- (1) 较为熟练使用 Vivado 工具。
- (2) 一定的汇编编程能力。

通过本章节的学习，你将获得：

- (1) TLB 的结构，及其设计方法。
- (2) TLB 相关寄存器、指令和例外的知识

在本章节的学习过程中，你可能需要查阅：

- (1) MIPS 官方手册。

## 实验提醒：

- (1) 在实现 TLB 相关 CP0 寄存器时，注意各域的读写属性，对于固定为 0 的，表示忽略写入，读出永远为 0。
- (2) TLB 结构里只有一个 G 位，但 EntryLo0 和 EntryLo1 各有 1 个 G 位。在执行 TLBWI 时，需要 EntryLo0 和 EntryLo1 的 G 位都是 1，才将 TLB 对应项的 G 位置 1；在执行 TLBR 时，则是将 TLB 对应项的 G 位读出同时赋值到 EntryLo0 和 EntryLo1 的 G 位。
- (3) 如果 TLB 结构里的 G 位为 1，表示在进行查找时，不需要对比进程号 ASID。
- (4) 在执行 TLBWI 时，需要将 EntryHi.ASID 写入到 TLB 项结构里的 ASID；在执行 TLBR 时，则是将 TLB 项结构里的 ASID 读入到 EntryHi.ASID。
- (5) 在 fetch/load/store 执行需要查找 TLB 进行虚实转换时，虚拟地址与 TLB 项里的 VPN2 比较，与 TLB 项里 ASID 比较的是 EntryHi.ASID。也就是 EntryHi.ASID 记录着当前进程的 ID 号。
- (6) 在执行 TLBP 指令时，如果未查找到，需要将 Index 的最高位置为 1。
- (7) 注意 Kseg0 和 Kseg1 段是 unmapped 的空间，对其进行访问不需要查找 TLB，目前直接访问即可。也就是对虚拟地址 0xbfc00000 访问就是对物理地址 0x1fc00000 访问。
- (8) TLB 的虚实转换可以将虚拟页转换到任意物理页，比如可以将虚拟地址 0x00000000 转换到物理地址 0x1fc00000，这是由软件维护的页表决定的，不需要 CPU 关心。

## 1.1 实验目的

1. 深入理解基于 TLB 的存储管理机制
2. 掌握 TLB 软硬件交互的界面和流程

## 1.2 实验设备

1. 装有 Xilinx Vivado、MIPS 交叉编译环境的计算机一台。
2. 龙芯体系结构教学实验箱（Artix-7）一套。

## 1.3 实验任务

将 myCPU 里的存储管理机制更改为 TLB 映射方式。

第一阶段需要完成：

- (1) CPU 增加 TLBR、TLBWI、TLBP 指令。

- (2) CPU 增加 Index、EntryHi、EntryLo0、EntryLo1、PageMask CP0 寄存器。
- (3) CPU 增加 32 项 TLB 结构，支持的页大小为 4KB。
- (4) 运行专用功能测试 tlb\_func，要求通过前 7 项测试。

第二阶段需要完成：

- (1) CPU 增加 TLB 相关例外：Refill、Invalid、Modified。
- (2) 运行专用功能测试 tlb\_func，要求全部通过，共 10 项测试。

第一、二阶段的测试程序都包含在 tlb\_func 里，在 tlb\_func/start.S 的第 5 行定义了 `"#define TEST_TLB_EXCEPTION 0"`，表示是否测试 TLB 例外。

- (1) `#define TEST_TLB_EXCEPTION 0`：不测试 TLB 例外，tlb\_func 共 7 个功能点测试，供第一阶段使用。
- (2) `#define TEST_TLB_EXCEPTION 1`：测试 TLB 例外，tlb\_func 共 10 个功能点测试，供第二阶段使用。

当前发布的 tlb\_func 默认是 `"#define TEST_TLB_EXCEPTION 0"`，在进行增加 TLB 例外支持后，需设置为 `"#define TEST_TLB_EXCEPTION 1"`，之后重新编译。

## 1.4 实验环境

myCPU 增加 TLB 支持后，需运行 TLB 专用功能测试 tlb\_func 通过。运行该测试的实验环境同 soc\_axi\_func。

如果觉得 AXI RAM 的随机延时影响 debug，可以考虑关闭该延迟，关闭方法是将 soc\_axi\_func/rtl/ram\_wrap/axi\_wrap\_ram.v 中的宏 `"RUN_PERF_TEST"` 和 `"RUN_PERF_NO_DELAY"` 开启。

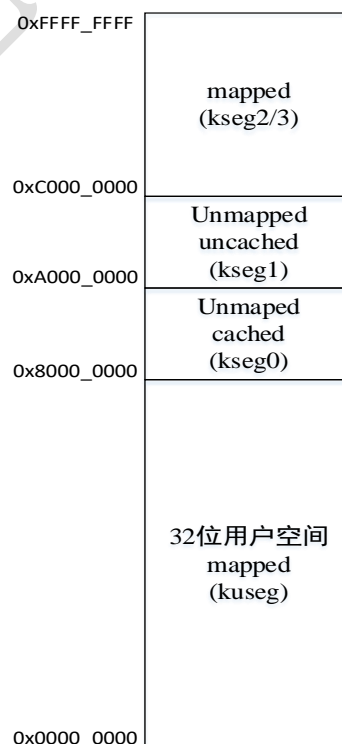
建议运行完 tlb\_func 测试后，再运行一次大赛功能测试，以确保增加 TLB 后未引入其他错误。

## 1.5 实验说明

此处再强调一下 TLB 结构、需要实现的 TLB 相关的 CP0 寄存器、指令和例外。

### 1.5.1 TLB 结构

MIPS 虚拟地址空间的内存映射的 32 位视图如下：



*mapped* 和 *unmapped* 指示是否需要通过 MMU 进行转换。**kseg0** 和 **kseg1** 是 *unmapped*，他们是永远指向物理地址 **0x0000\_0000~0x1fff\_ffff**。而 **kuseg** 和 **kseg2/3** 都是需要经过 MMU 单元进行地址翻译的(从虚拟地址转换为物理地址)。因为处理器刚启动时，MMU 是未设置好的，故我们无法对 **kuseg** 和 **kseg2/3** 进行地址翻译，这也是为什么 MIPS 处理器从 **0xbfc0\_0000(kseg1 段)**启动。换言之，在我们为设置好 MMU 之前，不能使用地址 **kuseg** 和 **kseg2/3**。

MMU 单元有简单形式的，比如固定映射，**gs132** 即实现的该形式；也有复杂形式，比如基于 TLB 的 MMU，这也是大部分 CPU 都会支持的，**gs232** 也实现了该形式。故需要对 TLB 进行初始化（MIPS 规范是要求软件完成初始化，现在通用处理器大多是硬件完成初始化），才能使得程序正常访问 **kuseg** 和 **kseg2/3**，从而可以启动一个操作系统。

TLB 是基于页翻译的，即将一个虚拟页翻译为一个物理页，页内偏移不变，主要是地址高位的转换。对于 4KB 页，页内偏移占 12 位，故需要转换的为高 20 位。MIPS32 中 TLB 结构如下：

EntryHi		PageMask		EntryLo0		EntryLo1	
VPN2	ASID	PageMask	G	PFN0	C、D、V	PFN1	C、D、V
19bit	8bit	12bit	1bit	20bit	5bit	20bit	5bit

上表中第 1 列的 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1** 为 cp0 寄存器。

第 2 列为一项 TLB 的结构，左侧 **VPN2**(虚拟地址高位)、**ASID**(地址空间标识)和 **PageMask**(页大小屏蔽位)结合在一起用于与虚拟地址进行比较，如果命中，则说明该项 TLB 后指示的 **PFN0**(转换后的物理地址高位)和 **PFN1**(转换后的物理地址高位)即为需要对应的奇偶页的物理地址高位，**C、D、V** 为标志位。显然 MIPS32 中为一个 TLB 项一次映射奇偶两个页。具体信息请查阅 MIPS 手册卷 III。

第 3 列为 CPU232 中各域的大小，一页大小为 4KB，页内偏移占 12bit。一项 TLB 同时映射奇偶两个页，故 **VPN2** 为 19bit，而 **PFN0** 和 **PFN1** 为 20bit，与页内偏移结合后正好得到 32bit 的物理地址。也因为页大小为 4KB，故 **PageMask** 为全 0。

本次实验需实现 32 项 TLB，即有 32 项上表中的结构。

关于 TLB 的详细描述，请参阅《计算机体系结构基础》的第五章和《see mips run》的第 6 章。

### 1.5.2 TLB 相关 cp0 寄存器

从 TLB 结构中知道，TLB 相关 cp0 寄存器有 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1**，此外本次实验涉及到的还有 **Index**。在本次实验中，上述寄存器的实现描述如下：

**EntryHi** 结构如下：

31	13 12	8 7	0
VPN2(R/W)		0	ASID(R/W)

**PageMask** 结构如下：

31	25 24	13 12	0
0	Mask(R/W)	0	

**EntryLo0** 结构如下：

31	26 25	6 5	0
0	PFN(R/W)	Flags(R/W)	

此处的 **PFN** 共有 24 位，是因为不同 CPU 中支持的物理地址不一定为 32 位，比如可能为 48 位。但在 CPU232 中就只用到 PFN 的低 20 位。其中 **Flags** 包含 **C**(3bits)、**D**(1bit)、**V**(1bit)、**G**(1bit)等信息：

- **C**: **EntryLo**[5:3]，指示 cache 属性，3：缓存的；2：非缓存的，其他值保留。
- **D**: **EntryLo**[2]，指示页的 Dirty 属性，1：目前不可写；0：目前可写。
- **V**: **EntryLo**[1]，指示有效属性，1：该项虚实转换有效；0：无效。
- **G**: **EntryLo**[0]，指示全局属性，1：该项转换是全局的，因而不需比较 ASID；0：需比较 ASID。

**EntryLo1** 结构同 **EntryLo0**。

Index 结构如下:

31	n	n-1	0
P(R)	0		Index(R/W)

Index 是用来索引 TLB 项的, 故 TLB 有多少项, 表中 n 有相应的值。在实现为 32 项 TLB 时, n 为 5。最高位记为 P, 这是一个只读的域, 在执行 TLBP 指令时, 如果未查找到, 则由硬件将该域置 1。

关于这些 cp0 寄存器的详细描述, 请查阅 MIPS 手册卷 3 第 9 章。

### 1.5.3 TLB 相关指令

本次实验只需实现 TLBWI、TLBR、TLBP 指令。

TLBWI 指令是以 cp0 寄存器 Index 为索引, 将 EntryHi、PageMask、EntryLo0、EntryLo1 写入到寻址到的 TLB 项中。

TLBR 指令则与 TLBWI 相反, 是以 Index 为索引, 读出寻址到的 TLB 项的值写入到 EntryHi、PageMask、EntryLo0、EntryLo1。显然测试 TLB 是否初始化完成也可以使用该指令。

TLBP 指令则是使用 EntryHi 里的 VPN2 和 ASID 去查找整个 TLB, 看是否有某一 TLB 项可以翻译该虚拟地址。如果查找到则将该项 TLB 索引号写入到 cp0 寄存器 Index 中; 如果没找到则将 Index 高位置 1, 其他位任意值。显然该指令测试通过, 表示 TLB 查找通路是没问题的。

### 1.5.4 TLB 相关例外

本次实验需实现的 TLB 相关例外包含: Refill、Invalid、Modified。

#### (1) TLB Refill 例外

当发生下列条件时触发 TLB Refill 例外:

- ◆ 依据虚拟页号在 TLB 中未查找到该项。

例外入口:

例外入口: 0xbfc00200

控制寄存器 Cause 的 ExcCode 域:

0x02 (TLBL): 取指或读数据

0x03 (TLBS): 写数据

响应例外时的额外硬件状态更新:

寄存器	状态更新描述
BadVAddr	记录触发例外的访问内存的虚地址。
EntryHi	VPN2 域更新为 VA <sub>31..12</sub>

#### (2) TLB Invalid 例外

当发生下列条件时触发 TLB Refill 例外:

- ◆ 依据虚拟页号在 TLB 中查找到该项, 但对物理页的 V 位为 0。

例外入口:

例外入口: 0xbfc00380

控制寄存器 Cause 的 ExcCode 域:

0x02 (TLBL): 取指或读数据

0x03 (TLBS): 写数据

响应例外时的额外硬件状态更新:

寄存器	状态更新描述
-----	--------

寄存器	状态更新描述
BadVAddr	记录触发例外的访问内存的虚地址。
EntryHi	VPN2 域更新为 VA <sub>31..12</sub>

### (3) TLB Modified 例外

当发生下列条件时触发 TLB Refill 例外：

- ◆ 依据虚拟页号在 TLB 中查找到该项，且对应物理页的 V 位为 1，D 位为 0，且该访问是 store。

例外入口：

例外入口：0xbfc00380

控制寄存器 Cause 的 ExcCode 域：

0x01 (Mod)：写数据

响应例外时的额外硬件状态更新：

寄存器	状态更新描述
BadVAddr	记录触发例外的访问内存的虚地址。
EntryHi	VPN2 域更新为 VA <sub>31..12</sub>

## 1.5.5 TLB 查找流程

CPU 在什么时候需要取查找 TLB 呢？在 fetch 或 load/store 时，如果访问地址是虚拟空间里 mapped 段时，那么 CPU 需要去查找 TLB 进行地址翻译。查找流程如下：

```

found ← 0
for i in 0...TLBEntries-1
    if ( (TLB[i].VPN2 and not (TLB[i].Mask)) = (va 31..13 and not (TLB[i].Mask))) and (TLB[i].G or (TLB[i].ASID = EntryHi.ASID)) then
        if va12 = 0 then
            pfn ← TLB[i].PFN0
            v ← TLB[i].V0
            c ← TLB[i].C0
            d ← TLB[i].D0
        else
            pfn ← TLB[i].PFN1
            v ← TLB[i].V1
            c ← TLB[i].C1
            d ← TLB[i].D1
        endif
        if v = 0 then
            SignalException(TLBInvalid, reftype)
        endif
        if (d = 0) and (reftype = store) then
            SignalException(TLBModified)
        endif
        # pfn 19..0 corresponds to pa 31..12
        pa ← pfn19..0 || va11..0
        found ← 1
        break
    endif
endfor
if found = 0 then
    SignalException(TLBMiss, reftype)
endif

```