

1 实验四 CPU Cache 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 Verilog 编程能力。

通过本章节的学习，你将获得：

- (1) Cache 的结构，及其设计方法。
- (2) Cache 相关寄存器、指令的知识

在本章节的学习过程中，你可能需要查阅：

- (1) MIPS 官方手册。

1.1 实验目的

1. 深入理解 Cache 的结构。
2. 掌握 Cache 工作机理。

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

在 myCPU 里增加一级 ICache 和 DCache。

第一阶段需要完成：

- (1) CPU 增加 ICache。比如 ICache 可以实现为 2 路组相连，每路大小 4KB，LRU 或伪随机替换算法，硬件初始化。

第二阶段需要完成：

- (1) CPU 增加 DCache。比如 DCache 可以实现为 2 路组相连，每路大小 4KB，LRU 或伪随机替换算法，硬件初始化，写回+写分配。

第三阶段需要完成：

- (1) CPU 增加 2 个 CP0 寄存器：Config 和 Config1。此时 Kseg0 依据 Config.k0 确定 Cache 属性。
- (2) CPU 增加专用 ICache 指令：I_Index_Inv、I_Index_Store_Tag、I_Hit_Inv。
- (3) CPU 增加专用 DCache 指令：D_Index_Wb_Inv、D_Index_Store_Tag、D_Hit_Inv、D_Hit_Wb_Inv。

1.4 实验说明

1.4.1 Cache 性能

Cache 有很多实现方法，有性能高的，也有性能低的。在本实验过程中应当尽量实现高性能版本的 Cache。本小节先简单说明 Cache 性能相关的几点细节，可先简单看下，等后续设计 Cache 的过程中再来详细考虑这些信息。

Cache 性能相关，有以下几点要求必须达到：

- Cache 命中应该时流水的，也就是连续命中时应当可以不间断的返回数据。
- 对一次访问，Cache 命中时的延迟，CPU 内部应该完全消化掉。也就是当取指和访存全部命中时，不考虑数据相关，CPU 内部流水线应该完全流水起来，对于单发射 CPU，此时 IPC 为 1。

有以下几点细节可以考虑实现：

- Cache Miss 时，使用关键字优先返回。这个需要 AXI 的 Burst 传输类型为 Wrap 访问。
- 考虑 Store buffer 的实现，Store 无论 Miss 还是 Hit，都可以不阻塞后续访存指令。

1.4.2 AXI 的 Burst 传输

在实现 Cache 时，需要修改 CPU 的 AXI 接口，需要支持 Burst 传输。关于 AXI 的 Burst 传输请参考 AMBA 官方文档。

1.4.3 Cache 容量计算

Cache 容量计算公式是：

$$\text{Cache 缓存容量} = \text{路数} * \text{路大小}$$

在本次实验中，ICache 和 DCache 都是 2 路、每路 4KB，所以其可缓存的数据容量是 $2 * 4\text{KB} = 8\text{KB}$ 。

上述计算得到的容量是其可缓存数据的大小，并不是实际实现该 Cache 所使用的 RAM 的总大小。实际实现所需 RAM 的大小还需要考虑 Cache 的 TAG、Dirty 等域。

1.4.4 Index、Tag 位数计算

本次实验实现的 Cache 是采用虚索引、实 Tag。图 1-1 说明了一个进行 Cache 访问的 32 位地址的 Tag、Index 和 Offset 域的划分：

- Offset: Cache 行内偏移。宽度为 $\log_2(\text{Cache 行大小})$ 。
- Index: Cache 组索引。宽度为 $(\log_2(\text{路大小}) - \log_2(\text{Cache 行大小}))$ 。
- Tag: Cache 行的 Tag 域。宽度为 $(\text{物理地址宽度} - \log_2(\text{路大小}))$ 。

图 1-1 包含以下重要信息

1. 物理地址宽度与虚拟地址宽度不一定相等；
2. 如果页大小小于路大小，图中虚线会落在 Index 域中，此时存在 Cache 别名的问题（Cache 中存在同一物理地址的多个缓存）。

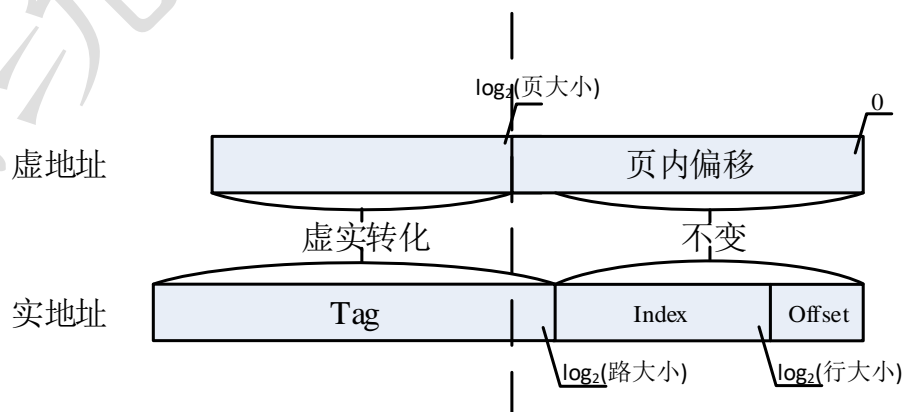


图 1-1 虚实转换中的 Tag、Index 和 Offset 域划分

如果 Cache 实现为 2 路组相连，每路 4KB，Cache 行为 32 字节，物理地址设定为 32 位。则 Cache 使用地址 [31:0] 中的 [11:5] 进行索引。剩余的高 20 位 ([31:12]) 包含在 TAG 域中。地址形式如下：

31	12 11	5 4	0
Tag		Index(索引)	offset 块内偏移

1.4.5 Cache 逻辑组织结构

在确定 Cache 的 Index、Tag 位数后，我们来考虑对应一个 Index 下的 Cache 组织（也就是一个 Cache 组的结构）：

- 每路的 Tag 域，2 路共有 2 个 Tag 域。index 宽度是 $(\log_2(4*2^{10})-5)=7$ ，TAG 域的宽度是 $32-7-5=20$ 。
- 每路的 V 域，2 路共有 2 个 V 域。V 域指示该 Cache 行是否有效，只需要 1bit。
- 每路的数据域，2 路共有 2 个 Data 域。DATA 域分为 8 个 bank，每个 bank 为 32bits。
- 对于 Dcache，每路的 Dirty 域，2 路共有 2 个 Dirty 域。Dirty 域指示该 Cache 行是否被写过，只需要 1bit。
- 采用 LRU 替换算法，还需要 LRU 域，2 路共同相同的 LRU 域。2 路组相连的 LRU 域只需要 1bit。

示意图如下：

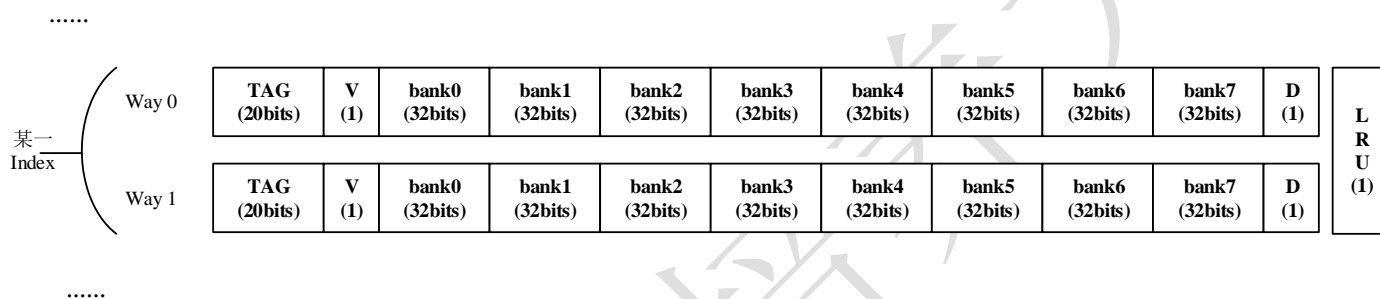


图 1-2 2 路*4KB 的 CACHE 逻辑组织结构

上图展示的 Cache 各域通常都是存放在 Ram 中的，Ram 的组织将在本章的 1.4.7 节介绍。但是有时候，对于部分很少的域，也是可以自己编写 RTL 代码使用寄存器组建，比如 LRU 域是可以使用 RTL 自行建模组建。像 Tag、Data 这些宽度比较多的，都是需要使用 Ram IP 进行组建的。

1.4.6 Cache 访问阶段划分

本小节的内容和 Cache 设计方案有所结合，按照我们预想的你们的设计方案（或者说是我们希望你们做出的设计方案）将 Cache 的访问阶段分为六个阶段，这一划分方案并不是通用的，仅供参考。这六个阶段是：

- Look up: 判断是否在 Cache 中，并读出 Data 供命中时返回读数据。
- Hit Update: 命中 Cache 时，更新信息。
- Miss handle: 从主存取回 Miss 的 Cache 行。
- Select: 选出被替换的 Cache 行。
- Replace: 将某一 Cache 行读出来，后续会替换出去。
- Refill: 将新取回的 Cache 行填入 Replace 后剩余的空行中。

上述 5 个阶段并不是流水阶段的划分，有些阶段其实是隐含了 2 个流水级。比如“Look up”其实隐含了“发送地址请求”和“Tag 比较”这 2 级；再比如“Replace”隐含了“发送地址请求”和“获取数据”。

显然“Look up”发现命中时会去往“Hit Update”；如果 Miss 会去往“Miss 处理单元”，“Miss 处理单元”会对外取回新的 Cache 行（“Miss handle”），取回之后才会进行后续的“Select”、“Replace”和“Refill”。

一般“Select”、“Replace”和“Refill”是连续的 3 个阶段，它们由“Miss 处理单元”进行维护，是在 Miss 时从外部取回新的 Cache 行之后的步骤，三个步骤一起是真正完成一个新的 Cache 行替换某一个旧的 Cache 行。通常在“Refill”之后，被替换出来的 Cache 行如果是脏的，会对外进行写回。

考虑每个阶段对本章 1.4.5 节描述的 Cache 各域的访问，如下表：

表 1-1 Cache 访问各阶段对各域的访问¹

域	Look up	Hit Update	Miss handle	Select	Replace	Refill
Tag	读 2 路的 Tag	-	-	-	读某一路的 Tag	更新某一路的 Tag
V	读 2 路的 V	-	-	-	读某一路的 V	更新某一路的 V
Data (8 个 Bank)	读 2 路的某一 bank	更新某一路的 某一 bank	-	-	读某一路的所有 bank	更新某一路的所有 bank
Dirty	-	更新某一路的 Dirty	-	在“Select”或者“Replace”时读 Dirty		更新某一路的 Dirty
LRU	-	更新 LRU	-	读出 LRU	-	更新 LRU

上表中，在部分设计中，“Select”和“Replace”的“发送地址请求”是可以在同一拍做的，也就是在“Select”选出被替换的 Cache 行的当拍发送“Replace”的读请求。“Replace”的“获取数据”是可以和“Refill”的“发送写请求”在同一拍的。在这些同一拍同时做的事情中，注意不要出现 Cache Ram 的 Output 送回到 Input 的路径，这样时序会差。

理清清楚表 1-1 的逻辑，可以有助于帮助我们设计 Cache 的 Ram 划分。比如，显然 Tag 和 V 访问情况完全一致，可以使用同一块 Ram 进行组建。

1.4.7 Cache 的 Ram 组织

在理解本章 1.4.5 和 1.4.6 节的内容后，可以考虑 Cache 的 Ram 组织结构，这是 Cache 设计之初的关键。在这之前，我们需要定义一个 Ram 的原语单元。

(1) 原语 Ram

原语 Ram 是最基本的 Ram，实验中的 Cache 都使用该原语 Ram 进行搭建。本次实验中，推荐大家使用单端口的 Ram 实现 Cache，基本的单端口 RAM 原语定义为：

表 1-2 搭建 CACHE 的 RAM 原语定义

名称	宽度	方向	描述
clk	1	input	时钟
en	1	input	使能信号，片选作用。高有效。
wen	待定	input	写使能，可能是 bit、byte 或 data 写使能。高有效。
addr	待定	input	地址，取决于 RAM 的深度
wdata	待定	input	写数据，取决于 RAM 的宽度
rdata	待定	output	读数据，取决于 RAM 的宽度

其中使能信号 en 即管读请求又管写请求。写使能信号 wen 只管写请求，其宽度依据需求而定：对于 bit mask 则是 wen 的每 1bit 对应 1bit 的数据；对于 byte mask 则是 wen 的每 1bit 对应 1byte 的数据；对于 data mask 则是 wen 只有 1bit，对应整个数据的写使能，也是常说的全局写使能。

通常，我们认定 Ram 的单周期返回的，也就是在第 n 拍，发送读地址请求，在第 n+1 拍获得读数据。通常，我们认定 Ram 没有读保持的功能，也就是说，在第 n+1 拍获得读数据后，而且 n+1 拍没有新的读写请求，不认为读数据会在 n+2 拍的时候保持在 rdata 的端口。

上表的原语 Ram 可以使用 Vivado 的 IP 定制 Block Memory Generator 生成，但在生成该 Ram 时，注意不要勾选“Primitives Output Register”和“Core Output Register”，否则该 Ram 不再是单周期返回了，如下图。

请大家在 Cache 实验中自行定制所需要的各个规格的 Ram。

¹表中“-”表示不需要访问。

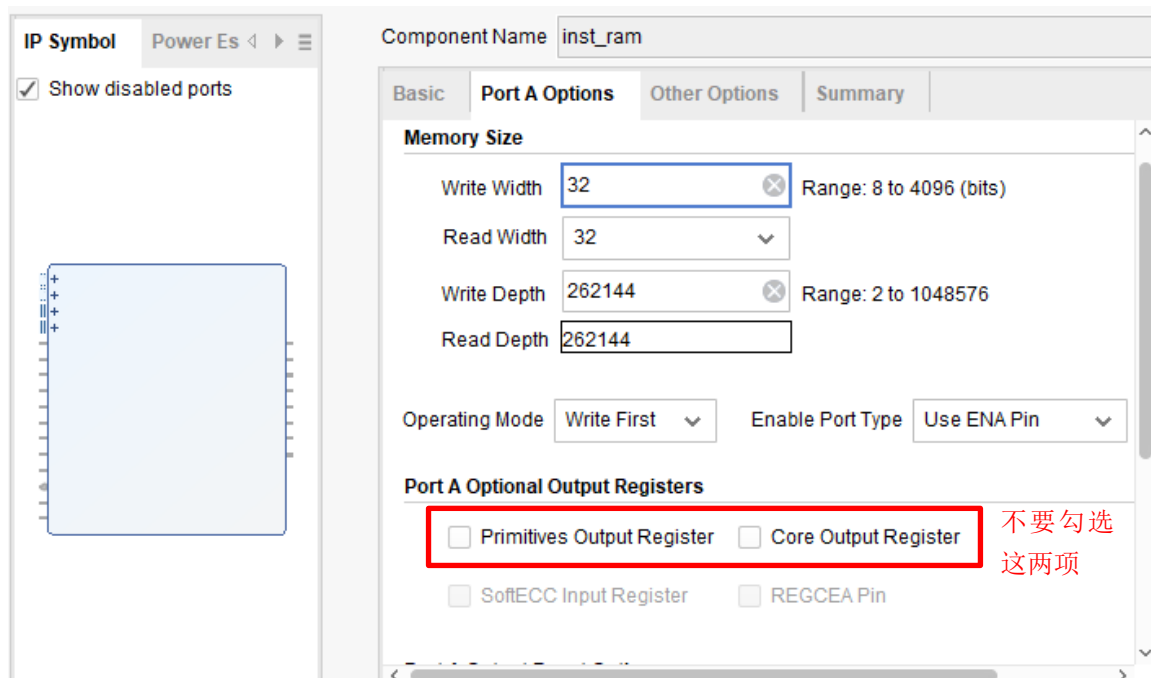


图 1-3 定制 Ram 时的注意事项

(2) Ram 组织结构

依据本章 1.4.5 和 1.4.6 节的讨论，我们可以给出一个最简单的 Ram 组织方案：

- TAGV Ram: 选用 Ram 128*21（深度*宽度），共需要 2 块。
- DATA Ram: 选用 Ram 128*32（深度*宽度），共需要 16 块。
- Dirty Ram（仅 DCache 需要）：选用 Ram 128*1（深度*宽度），共需要 2 块。
- LRU Ram（仅 LRU 替换算法需要）：选用 Ram 128*1（深度*宽度），共需要 1 块。

实际的 Ram 组织结构见下图。显然 TAGV 和 DATA Ram 适合使用定制的 Ram IP 组建，而 Dirty 和 LRU Ram 宽度很小，是可以自行编写 RTL 使用寄存器组建的。

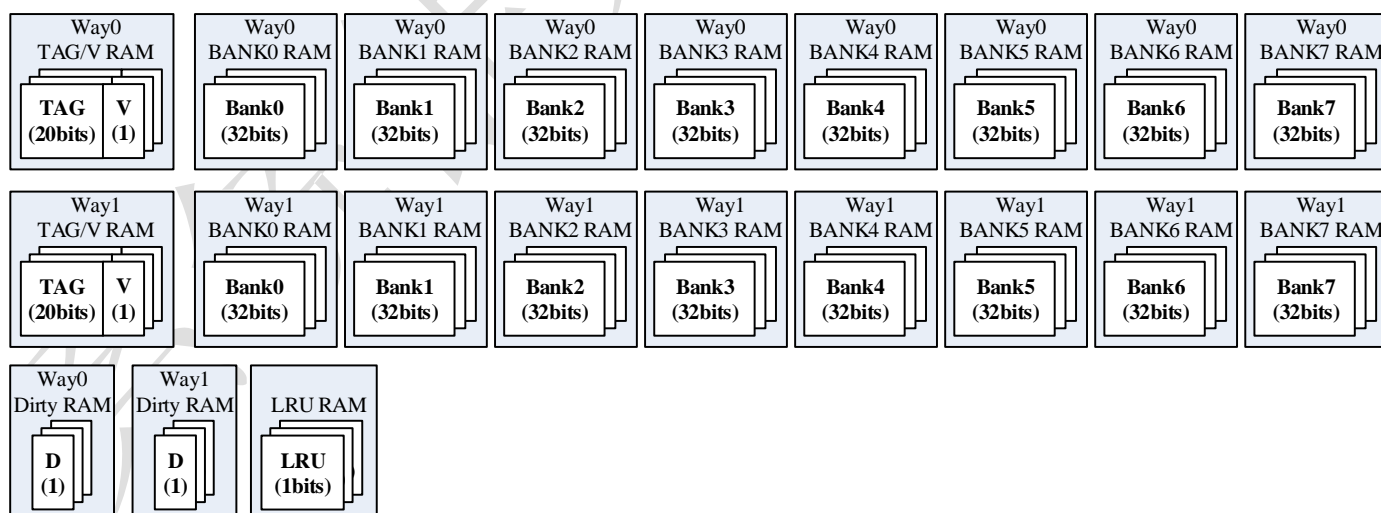


图 1-4 一种 2 路*4KB 的 DCache 的 RAM 组织结构

上图的方案并不是唯一的：

- 比如，2 个 Dirty Ram 和 LRU Ram 是完全可以合并成一个规格为 128*3 的 Ram。
- 再比如，way0 和 way1 的相同 bank 都可以合并成一个规格为 128*64 的 Ram。
- 再比如，way0 的每一对奇偶 bank（如 bank0 和 bank1）可以合并成一个规格为 128*64 的 Ram。

在编写 Cache 代码前，一定要规划好 Cache 的 Ram 组织方案。

1.4.8 Cache 属性配置

MIPS 虚拟地址空间的内存映射的 32 位视图如下：

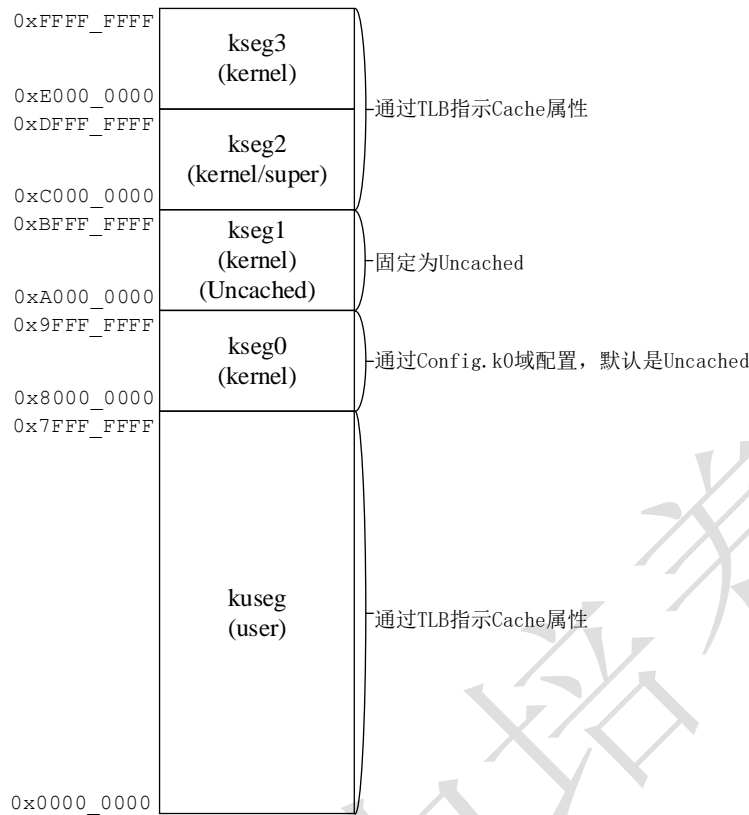


图 1-5 通过 Config 相关域配置段属性

kseg0 和 kseg1 是 unmapped，他们是永远指向物理地址 0x0000_0000~0x1fff_ffff，不需要经过 MMU 单元（如 TLB）转换。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)。

对于 Cache 属性，在这 5 个段（kuseg、kseg0、kseg1、kseg2 和 kseg3）中，kseg1 段是固定是 Uncached 属性的，无法更改；Kseg0 可以通过 CP0 寄存器 Config 的 K0 域更改 Cache 属性的；kuseg 和 kseg2/3 的地址则在 TLB 转换时由对应表项里的 C 域指定 Cache 属性。

Config.k0 和 TLB 表项里的 C 域都是 3bit 的域：0x2 表示 Uncached；0x3 表示 Cached；其他值为保留，当作 Uncached。

在实现 Cache 时一定要注意：

- Uncached 的访问不可以经过 Cache 缓存；
- Uncached 的 Load 不能使用之前的 Store 前递过来的数据，所有 Uncached Load 必须从目标设备获取数据；
- Uncached 的访问需要确保顺序性，比如 Uncached Load 不要越过其之前的 Uncached Store 先执行。

1.4.9 Config 寄存器 (CP0 Register 16, Select 0)

Config 寄存器中定义了一些处理器的配置信息。Config 寄存器中除 K0 域可通过软件读写外，其它域均由硬件在复位期间予以初始化并保持只读状态不变。

图 1-6 说明了 Config 寄存器的格式；表 1-3 Config 寄存器域描述表 1-3 对 Config 寄存器各域进行了描述。

图 1-6 Config 寄存器格式

31	30	16	15	14	13	12	10	9	6	3	2	0
M	0	BE	AT	AR	MT	0	K0					

表 1-3 Config 寄存器域描述

域名称	位	功能描述	读/写	复位值
M	31	值为 1，表示存在 Config1 寄存器。	R	0x1
0	30..16	只读恒为 0。	0	0
BE	15	值为 0，表示采用小尾端寻址方式。	R	0x0
AT	14..13	值为 0，示实现 MIPS32 结构。	R	0x0
AR	12..10	值为 0，兼容 MIPS32 release1 规范，具体各细节实现的情况软件可通过读取其它配置寄存器或其它寄存器的配置域获得。	R	0x0
MT	9..7	值为 1，MMU 采用标准的 TLB。	R	0x1
0	6..3	只读恒为 0。	0	0
K0	2..0	指定 Kseg0 段在 Status.ERL=0 时的访存类型。2: Uncached; 3: Cached; 其它: Uncached	R/W	0x2

1.4.10 Config1 寄存器(CP0 Register 16, Select 1)

Config1 寄存器用于提供处理器的一些配置信息。Config1 寄存器中的所有域均为只读。

图 1-7 说明了 Config1 寄存器的格式；表 1-4 对 Config1 寄存器各域进行了描述。

图 1-7 Config1 寄存器格式

31	30	25	24	22	21	19	18	16	15	13	12	10	9	7	6	5	4	3	2	1	0
M	MMU Size-1				IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP				

表 1-4 Config1 寄存器域描述

域名称	位	功能描述	读/写	复位值
M	31	值为 0，表示不存在 Config2 寄存器。	R	0x1
MMUSize-1	30..25	指示了 CPU 里实现的 TLB 大小，值为 0~63，对应 TLB 项数为 1~64。	R	预置的
IS	24..22	ICache 每路的组数目，等于 路大小/Cache 行大小： 0: 64 1: 128 2: 256 3: 512 4: 1024 5: 2048 6: 4096 7: 32	R	预置的
IL	21..19	ICache 的 Cache 行大小： 0: 没有 ICache 1: 4 字节 2: 8 字节 3: 16 字节 4: 32 字节 5: 64 字节 6: 128 字节 7: 保留	R	0x4
IA	18..16	ICache 的相连度： 0: 直接相连 1: 2 路 2: 3 路 3: 4 路 4: 5 路 5: 6 路 6: 7 路 7: 8 路	R	预置的
DS	15..13	DCache 每路的组数目，等于 路大小/Cache 行大小。编码类似 IS 域。	R	预置的
DL	12..10	DCache 的 Cache 行大小。编码类似 IL 域。	R	0x4
DA	18..16	DCache 的相连度。编码类似 IA 域。	R	预置的
C2	6	值为 0，表示没有协处理器 2 (COP2)。	R	0x0
MD	5	值为 0，表示未实现 MDMX ASE 指令集。	R	0x0
PC	4	值为 0，表示未实现性能计数器。	R	0x0

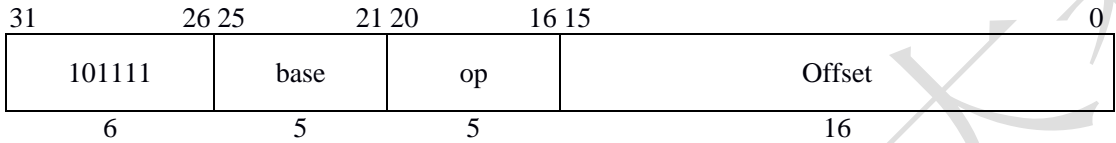
域名称	位	功能描述	读/写	复位值
WR	3	值为 0，表示未实现 Watch 寄存器。	R	0x0
CA	2	值为 0，表示未实现 MIPS16e 指令集。	R	0x0
EP	1	值为 0，表示未实现了 EJTAG。	R	0x0
FP	0	值为 0，表示未实现浮点协处理器。	R	0x0

1.4.11 Cache 指令

本次实验要求实现的 Cache 相关指令只有 Cache 指令。Cache 指令用于管理 Cache，包含很多操作，以下对其作重点介绍。

(1) Cache 指令格式

MIPS 中 Cache 指令的格式如下：



汇编格式：CACHE op, offset(base)

指令说明：依据 op 做不同的 Cache 操作

$GPR^2[base] + \text{sign_extend}(\text{offset})$ 用于计算得到一个地址，该地址依据 op 的不同有以下两类意义：

- 对 Index 类 op，使用图 1-8 展示的 Index 域索引 Cache 组，使用 Way 确定访问该组里的哪路 Cache 行，Tag 域并不使用。
- 对 Hit 类 op，需要将计算后的地址转换为实地址，之后同正常 Cache 访问一样，使用图 1-8 展示的 Index 和 Tag 域查找 Cache，判断是否命中并进行相应操作。

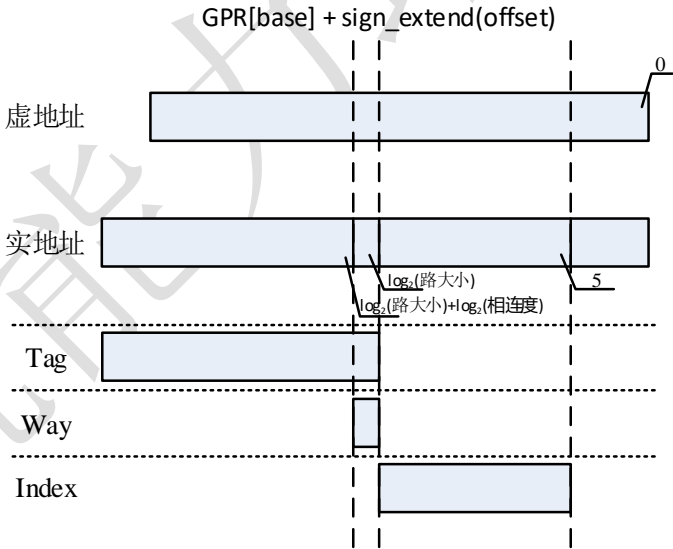


图 1-8 Cache 指令里地址的使用示意

(2) ICache 操作

本次实验要求实现的 Cache 指令需要支持的 ICache 操作有 3 类操作，见表 1-5 所示。

表 1-5 支持的 ICache 操作一览表

op 编码	操作名称	地址使用方式	功能描述
0b00000	Index Invalid	Index	使用地址索引到 Cache 行后，将该行无效掉。
0b01000	Index Store Tag	Index	使用地址索引到 Cache 行后，将 CP0 寄存器 TagLo 指定的 Tag、V 和 D 域更新进该 Cache 行。
0b10000	Hit Invalid	Hit	使用地址查找 ICache，如果命中，则将该行无效掉。

² GPR：通用寄存器堆的缩写。

(3) DCache 操作

本次实验要求实现的 Cache 指令需要支持的 DCache 操作有 4 类操作，见表 1-6 所示。

表 1-6 支持的 DCache 操作一览表

op 编码	操作名称	地址使用方式	功能描述
0b00001	Index Writeback Invalid	Index	使用地址索引到 Cache 行后，将该行无效掉。如果该行是有效且脏的，则需要先写回到系统内存里。
0b01001	Index Store Tag	Index	使用地址索引到 Cache 行后，将 CP0 寄存器 TagLo 指定的 Tag、V 和 D 域更新进该 Cache 行。
0b10001	Hit Invalid	Hit	使用地址查找 DCache，如果命中，则将该行无效掉。即使该行是有效且脏的，也不需要写回到系统内存里。
0b10101	Hit Writeback Invalid	Hit	使用地址查找 DCache，如果命中，则将该行无效掉。如果该行是有效且脏的，则需要先写回到系统内存里。