ARM920T 的 MMU 与 Cache

目录

虚拟地址和物理地址的概念

CPU 通过地址来访问内存中的单元，地址有虚拟地址和物理地址之分，如果 CPU 没有 MMU（Memory Management Unit，内存管理单元），或者有 MMU 但没有启用，CPU 核在取指令或访问内存时发出的地址将直接传到 CPU 芯片的外部地址引脚上，直接被内存芯片（以下称为物理内存，以便与虚拟内存区分）接收，这称为物理地址（Physical Address，以下简称 PA），如下图所示。

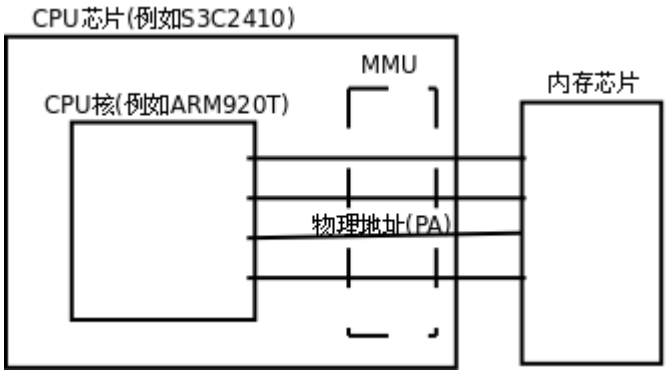


图 1. 物理地址示意图

如果 CPU 启用了 MMU，CPU 核发出的地址将被 MMU 截获，从 CPU 到 MMU 的地址称为虚拟地址（Virtual Address，以下简称 VA），而 MMU 将这个地址翻译成另一个地址发到 CPU 芯片的外部地址引脚上，也就是将虚拟地址映射成物理地址，如下图所示[1]。

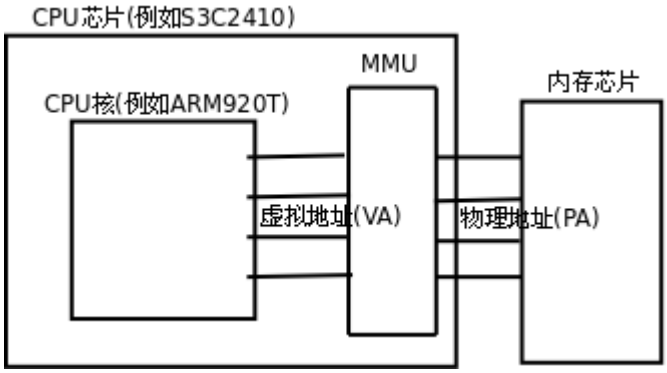


图 2. 虚拟地址示意图

MMU 将虚拟地址映射到物理地址是以页（Page）为单位的，对于 32 位 CPU 通常一页为 4K。例如，虚拟地址 0xb700 1000~0xb700 1fff 是一个页，可能被 MMU 映射到物理地址 0x2000~0x2fff，物理内存中的一个物理页面也称为一个页框（Page Frame）。

虚拟内存管理

现代操作系统充分利用 MMU 提供的 VA 到 PA 的映射机制来做内存管理，以下称为虚拟内存管理（Virtual Memory Management）。首先看下面的例子：

```
$ ps
PID TTY TIME CMD
9612 pts/2 00:00:00 bash
32070 pts/2 00:00:00 ps
$ pmap 9612
9612: bash
08048000 668K r-x-- /bin/bash
080ef000 24K rw--- /bin/bash
080f5000 2056K rw--- [ anon ]
b7c6d000 36K r-x-- /lib/tls/i686/cmov/libnss_files-2.7.so
b7c76000 8K rw--- /lib/tls/i686/cmov/libnss_files-2.7.so
b7c78000 32K r-x-- /lib/tls/i686/cmov/libnss_nis-2.7.so
b7c80000 8K rw--- /lib/tls/i686/cmov/libnss_nis-2.7.so
b7c82000 80K r-x-- /lib/tls/i686/cmov/libnsl-2.7.so
b7c96000 8K rw--- /lib/tls/i686/cmov/libnsl-2.7.so
b7c98000 8K rw--- [ anon ]
```

例 1. 进程的地址空间

这是 bash 进程的虚拟地址空间，32 位 CPU 的虚拟地址空间是 4GB，也就是 0x0000 0000-0xffff ffff，该进程占用的地址范围近似为 0x0000 0000-0xbfff ffff，地址范围 0xc000 0000-0xffff ffff 由内核占用，用户进程不允许访问。在这个 bash 进程的地址空间中，从 0x0804 8000 开始的 668K 的权限为 r-x--，

表示代码段，从 0x080e f000 开始的 24K 的权限是 rw---，表示数据段，从 0x080f 5000 开始的 2056K 的权限也是 rw---，但是没有对应任何磁盘文件，而是用[anon]（anonymous，匿名）来表示，这是堆所占的空间，从 0xb7c6 d000 开始是共享库和资源文件的映射空间，每个共享库也分为代码段和数据段，用不同的权限表示，可以看到，从堆空间到下面的共享库映射空间之间有很大的地址空洞，最末从 0xbfad 4000 开始的 84K 是栈空间。

为什么需要虚拟内存管理呢？可以从以下几个方面来理解。

第一，让每个进程有独立的地址空间是引入虚拟内存管理的最主要目的。所谓独立的地址空间是指，不同进程中的同一个 VA 被 MMU 映射到不同的 PA，并且在某一个进程中访问任何地址都不可能访问到另外一个进程的数据，这样使得任何一个进程由于程序 BUG 或恶意代码所导致的非法内存访问都不会意外改写其它进程的数据，不会影响其它进程的运行，从而保证了整个系统的稳定性。另一方面，每个进程都认为自己独占 4GB 的地址空间，编写程序会比较方便，不必为每个进程分配一个地址范围，而是每个进程都可以使用一个完整的地址空间中的任何地址。

我们继续用上面的例子来理解，再打开一个 shell 窗口，用 pmap 命令看一下这个新的 bash 进程的地址空间，可以发现和刚才的地址空间布局差不多：

```
$ ps
PID TTY TIME CMD
32371 pts/1 00:00:00 bash
32387 pts/1 00:00:00 ps
$ pmap 32371
32371: bash
08048000 668K r-x-- /bin/bash
080ef000 24K rw--- /bin/bash
080f5000 2000K rw--- [ anon ]
b7c71000 36K r-x-- /lib/tls/i686/cmov/libnss_files-2.7.so
b7c7a000 8K rw--- /lib/tls/i686/cmov/libnss_files-2.7.so
......
```

该进程也占用了 0x0000 0000-0xbfff ffff 的地址空间，代码段也是从 0x0804 8000 开始的 668K，数据段也是从 0x080e f000 开始的 24K，共享库的内存布局也差不多。这个进程和刚才的例子是同一个系统中同时运行着的两个进程，它们都认为自己占有 0x0000 0000-0xbfff ffff 的地址空间，并且它们的数据段的地址范围是重合的，但是两个进程各自干各自的事情，显然数据段中的数据是不同的，正是因为不同进程中的同一个 VA 被映射到了不同的 PA，所以两个进程的数据段其实是在不同的物理地址上，如下图所示。

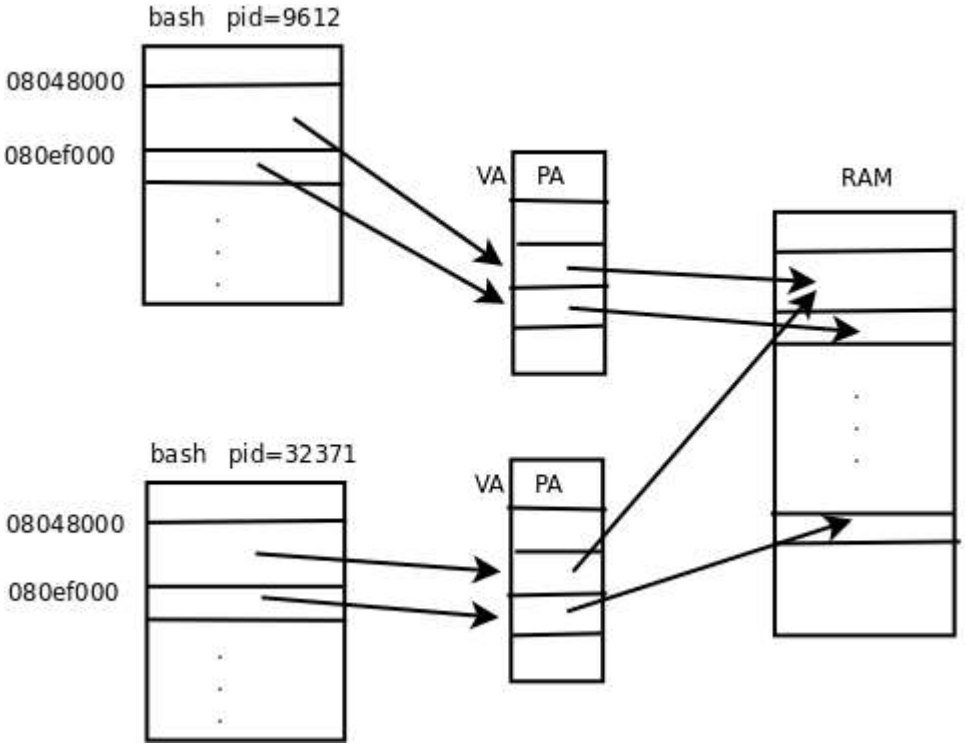


图 4. 进程地址空间是独立的

从图中还可以看到，两个进程都是 bash 进程，代码段是一样的，并且代码段是只读的，不会被改写，因此操作系统会安排两个进程的代码段共享相同的物理内存。由于每个进程都有自己的一套 VA 到 PA 的映射表，整个地址空间中的任何 VA 都在每个进程自己的映射表中查找相应的物理地址，因此不可能访问到其它进程的地址，也就没有可能意外改写其它进程的数据。

第二，引入 VA 到 PA 的映射也会给分配和释放内存带来方便，物理上不连续的空间可以映射为逻辑上连续的虚拟地址空间。比如要 malloc 一块很大的内存空间，而物理内存虽然有足够的空闲内存，却没有足够大的连续空闲内存，这时就可以分配多个不连续的物理页面，而映射为连续的虚拟地址范围。如下图所示。

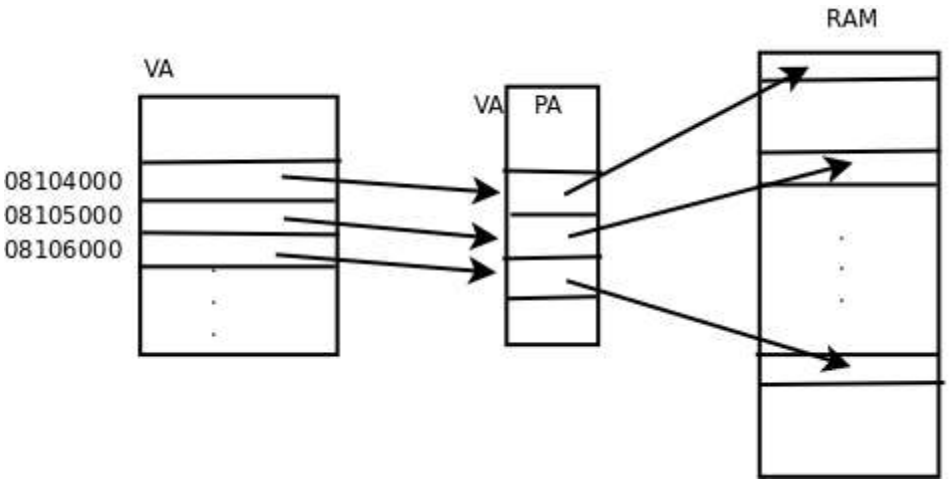


图 5. 不连续的 PA 可以映射为连续的 VA

第三，一个系统如果同时运行着很多进程，为各进程分配的内存之和可能会大于实际可用的物理内存，虚拟内存管理使得这种情况下各进程仍然能够正常运行。因为各进程分配的只不过是虚拟内存的页，这个页的内容可以映射到物理内存的页框，也可以临时保存到磁盘上而不占用物理内存的页框，磁盘上这一部分称为交换设备（Swap Device），可能是一个磁盘分区，也可能是一个磁盘文件。当物理内存不够时将物理内存中不常用的页框临时保存到磁盘上，而当用到这些页框时再从磁盘加载回内存，这称为换页（Paging）因此：

系统中可分配的内存总量 = 物理内存的大小 + 交换设备的大小

如下图所示。第一张图是换出（Page out），将物理页面的内容保存到磁盘，并解除地址映射，释放物理页面。第二张图是换入（Page in），从空闲的物理页面中分配一个，将磁盘暂存的页面加载回内存，并建立地址映射。

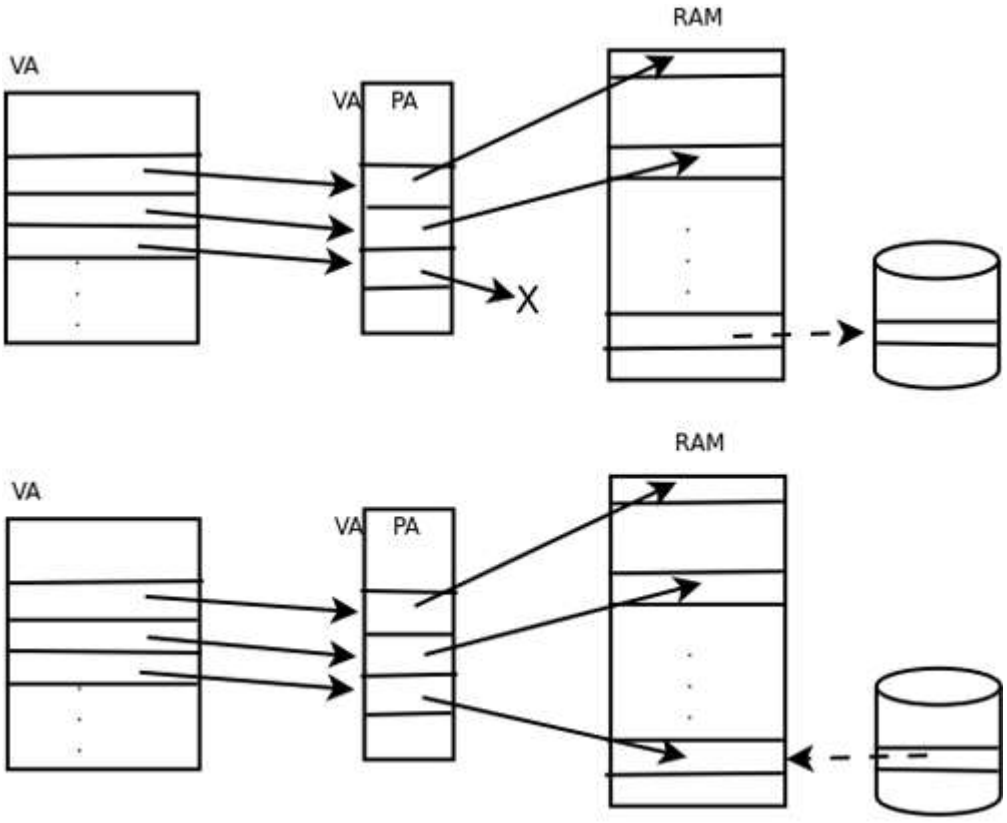
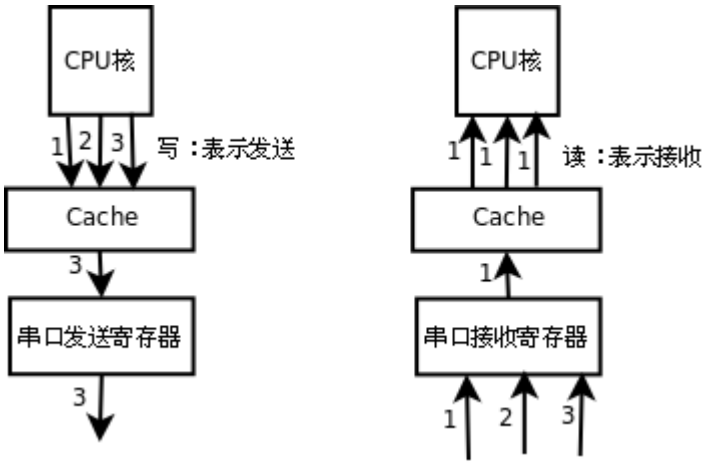


图 6. 换页

第四，虚拟内存管理可以控制物理页面的访问权限。物理内存本身是不限制访问的，任何地址都可以读写，而操作系统要求实现各种不同的访问权限，在先前的例子中我们已经看到，代码段要求是 rx 的，数据段要求是 rw 的，用户进程不能访问属于内核的地址空间，这些都是操作系统和 MMU 配合实现的。

MMU 中还实现了一种访问限制是关于 Cache 的。Cache（高速缓存）是 CPU 内的一小块高速 RAM，用来缓存最近访问过的内存数据，CPU 访问 Cache 的速度是访问内存速度的数十倍，所以有效地利用 Cache 可以大大提高计算机的整体性能。CPU 核要访问数据时首先发出 VA，Cache 利用 VA 查找相应的数据有没有被缓存[2]，如果有就通知 CPU 核，如果是读操作就直接将 Cache 中的数据传给 CPU 核中的寄存器，如果是写操作就直接改写 Cache 中的数据，而不需要访问物理内存。但是，有些 VA 所对应的 PA 并不是物理内存中的地址而是设备寄存器的地址，对这些寄存器进行读写并不是为了保存数据，而是对设备做特殊操作，这种 VA 通常是不允许缓存的，因为如果缓存了，对 VA 的读写将只在 Cache 中起作用，而不会传到设备寄存器对设备进行操作。以串口的收发寄存器为例，如果收发寄存器地址被缓存了会出现什么问题呢？如下图所示。



如果发送寄存器的地址被缓存起来，CPU 核往发送寄存器的地址做写操作都写到 Cache 中去了，发送寄存器并没有及时得到数据，也就不能及时发送，此外，CPU 核先后发出的 1、2、3 三个数据都会写到 Cache 中的同一个地址，最后 Cache 中只保存了第 3 个数据，如果这时 Cache 的数据写回到发送寄存器去，只能把第 3 个数据发送出去，前两个数据就丢失了。与此类似，如果接收寄存器的地址被缓存起来，CPU 核在读第 1 个数据时，Cache 会从接收寄存器读进来缓存，然而接收寄存器后面收到 2、3 两个数据 Cache 并不知道，因为 Cache 把接收寄存器当作内存，并且相信内存中的数据是不会自己变的，所以以后每次 CPU 核读接收寄存器时，Cache 都提供给 CPU 核第 1 个数据。

ARM920T 的 CP15 协处理器

ARM920T 的 MMU 和 Cache 都集成在 CP15 协处理器中，MMU 和 Cache 的联系非常密切，本节首先从总体上介绍 MMU、Cache 和 CPU 核是如何协同工作的，后面两节分别讲解 MMU 和 Cache 的细节。三星公司的 S3C2410 是一种很常见的采用 ARM920T 的芯片，涉及到具体的芯片时我们以 S3C2410 为例。

以下是 CP15 协处理器的寄存器列表（摘自[S3C2410 用户手册]），和 CPU 核的 r0 到 r15 寄存器一样，协处理器寄存器也是用 0 到 15 来编号，在指令中用 4 个 bit 来表示寄存器编号，有些协处理器寄存器有影子寄存器，这种情况下对同一个编号的寄存器使用不同的选项读或者写实际上访问的是不同的寄存器，后文用到某个寄存器时会详细说明它的功能。

寄存器编号	功能
0	ID CODE, Cache Type (RO)
1	Control register
2	Translation table base (TTB) register
3	Domain access control register
4	--Reserved--
5	Fault status register
6	Fault address register
7	Cache operation
8	TLB operation
9	Cache lock down register
10	TLB lock down registe
11-12 & 14	--Reserved--
13	ProcID
15	--Reserved for test purpose--

表 1. CP15 协处理器的寄存器列表

对 CP15 协处理器的操作使用 mcr 和 mrc 两条协处理器指令，这两条指令的记法是从后往前看：mcr 是把 r（CPU 核寄存器）中的数据传送到 c（协处理器寄存器）中，mrc 则是把 c（协处理器寄存器）中的数据传送到 r（CPU 核寄存器）中。对 CP15 协处理器的所有操作都是通过 CPU 核寄存器和 CP15 寄存器之间交换数据来完成的。下图是协处理器的指令格式（摘自[S3C2410 用户手册]）。

MCR/MRC{cond} P15,opcode_1,Rd,CRn,CRm,opcode_2

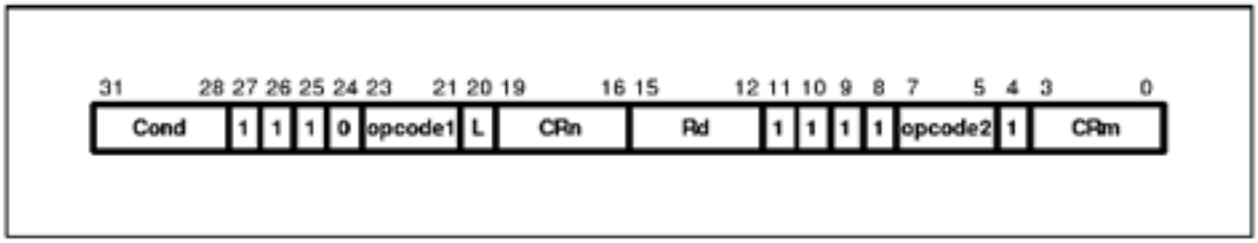


图 8. 协处理器指令格式

和其它 ARM 指令一样，Cond 是条件码，bit 20 是 L 位，表示该指令是读还是写，如果 L=1 就表示 Load，从外面读到 CPU 核中，也就是 mrc 指令，如果 L=0 就表示 Store，也就是 mcr 指令。[11:8]这四个位是协处理器编号，CP15 的编号是 15，因此是 4 个 1。CRn 是 CP15 寄存器编号，Rd 是 CPU 核寄存器编号，各占 4 个位。对于 CP15 协处理器，规定 opcode1 应该为 0，opcode2 和 CRm 是指令的选项，具体含义取决于不同的寄存器。

虽然这里介绍了协处理器的寄存器编号和相关指令，但读者只需了解对协处理器是这样进行操作的就可以了，我们的重点是讲解 MMU 和 Cache 的基本概念，具体各种操作的指令该怎么写可以参考[S3C2410 用户手册]。

MMU 是如何把 VA 映射成 PA 的呢？从图 4 “进程地址空间是独立的”来看，好像是有一张 VA 转 PA 的表，给一个 VA 查表就可以查到 PA，实际上并不是这么简单，通常要有一个多级的查表过程，对于 ARM 体系结构是两级查表，对于一些 64 位体系结构则需要更多级。看下面的图示。

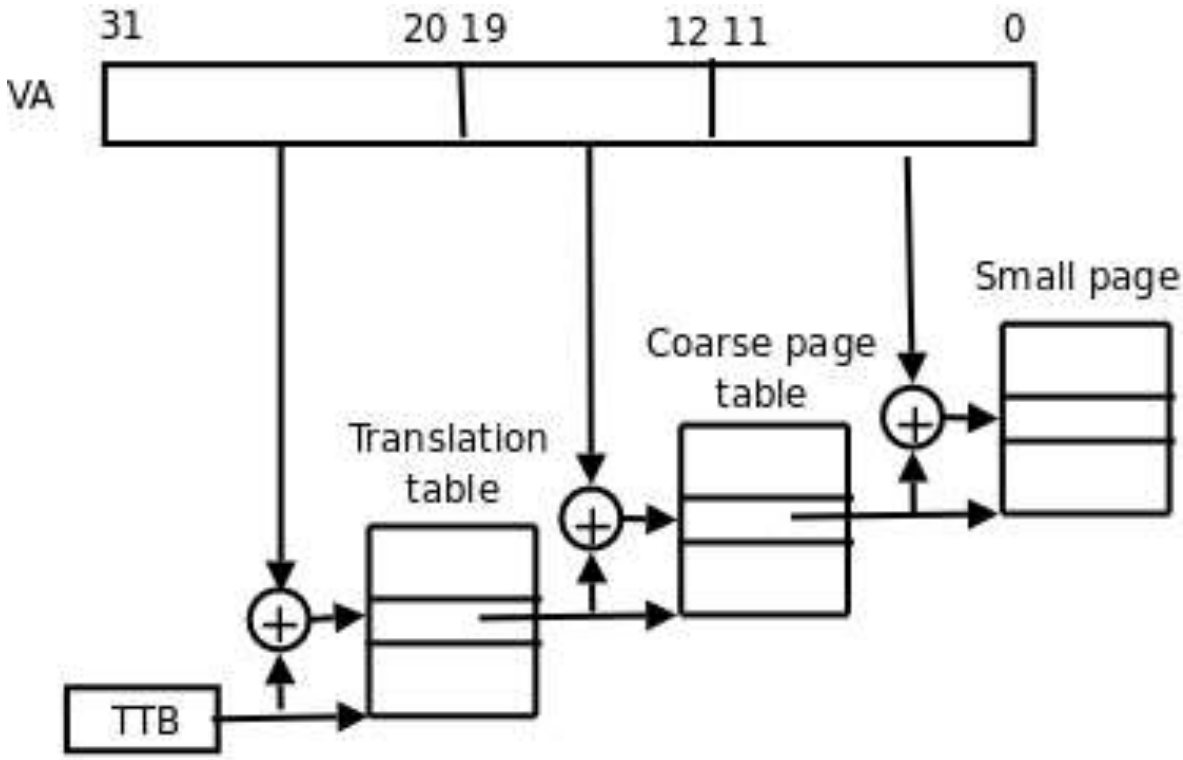


图 9. Translation Table Walk

首先将 32 位的 VA[3]分成三段，前两段[31:20]和[19:12]作为两次查表的索引，第三段[11:0]作为页内的偏移。查表的步骤如下：

1 CP15 协处理器的 TTB 寄存器（看看表 1 “CP15 协处理器的寄存器列表”中这是第几个寄存器？C2）中保存着第一级页表（Translation Table）的基地址，这个基地址指的是 PA，也就是说页表是直接按这个地址存在物理内存中的。

2 以 TTB 中的内容为基地址，以 VA[31:20]为索引在表中查出一项（想一下这个表中一共有多少项？4096 项），这个表项中保存着第二级页表（Coarse Page Table）的基地址，同样是物理地址，也就是说第二级页表也是直接按这个地址存在物理内存中的。

3 以 VA[19:12]为索引在第二级页表中查出一项（想一下这个表中一共有多少项？256 项），这个表项中就保存着物理页面的基地址，先前我们说虚拟内存管理是以页为单位的，一个虚拟内存的页映射到一个物理内存的页框，从这里就可以得到印证，因为查表是以页为单位来查的。

4 有了物理页面的基地址之后，加上 VA[11:0]这个偏移量就可以取出相应地址上的数据（想一下一个页是多少字节？4K）。

这个过程称为 Translation Table Walk，Walk 这个词用得非常形象。从 TTB 走到一级页表，又走到二级页表，又走到物理页面，一次寻址其实是三次访问物理内存。注意这个“走”的过程完全是硬件做的，每次 CPU 寻址时 MMU 就自动完成以上四步，不需要编写指令指示 MMU 去做，前提是操作系统要维护页表项的正确性，每次分配内存时填写相应的页表项，每次释放内存时清除相应的页表项，在必要的时候分配或释放整个页表。

有了以上基本概念，我们来看 CPU 访问内存时的硬件操作顺序（摘自[ARM 参考手册]）。

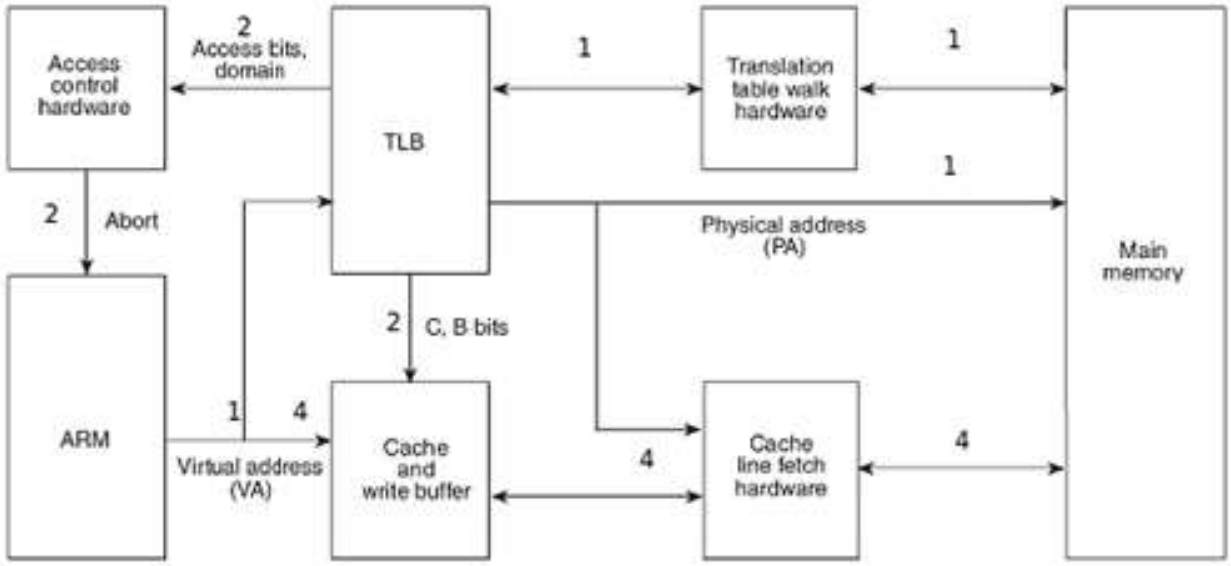


图 10. CPU 访问内存时的硬件操作顺序

我们以 CPU 读内存为例解释一下图中的步骤，各步骤在图中有对应的标号。

1 CPU 核（图中的“ARM”框）发出 VA 请求读数据，TLB（Translation Lookaside Buffer）接收到该地址。TLB 是 MMU 中的一块高速缓存（也是一种

Cache），它缓存最近查找过的 VA 对应的页表项，如果 TLB 里缓存了当前 VA 的页表项就不必做 Translation Table Walk 了，否则去物理内存中读出页表项保存在 TLB 中，TLB 缓存可以减少访问物理内存的次数。

2 页表项中不仅保存着物理页面的基地址，还保存着权限位和是否允许 Cache 的标志。MMU 首先检查权限位，如果没有访问权限，就引发一个异常给 CPU 核。然后检查是否允许 Cache，如果允许 Cache 就启用 Cache 和 CPU 核互操作，图中的“C, B bits”可以理解为直写和回写线，后面再详细解释这两个位的作用。

3 如果不允许 Cache，则直接发出 PA 从物理内存中读取数据到 CPU 核。

4 如果允许 Cache，则以 VA 为索引到 Cache 中查找是否缓存了要读取的数据，如果 Cache 中已经缓存了该数据（称为 Cache Hit）则直接返回给 CPU 核，如果 Cache 中没有缓存该数据（称为 Cache Miss），则发出 PA 从物理内存中读取数据并缓存到 Cache 中，同时返回给 CPU 核。然而 Cache 并不是只取 CPU 核所要的数据，而是把相邻的数据都取上来缓存，这称为一个 Cache Line。ARM920T 的 Cache Line 是 32 字节，例如 CPU 核要读取地址 0x134-0x137 的 4 字节数据，Cache 会把地址 0x120-0x13f（对齐到 32 字节地址边界）的 32 字节都取上来缓存。

MMU

我们已经简单了解了一下查页表的过程，实际上 ARM920T 支持多种尺寸规格的页表，图 9 “Translation Table Walk”所示的只是其中一种情况。下图示意了所有可能的情况（本节的图表均摘自[S3C2410 用户手册]）。

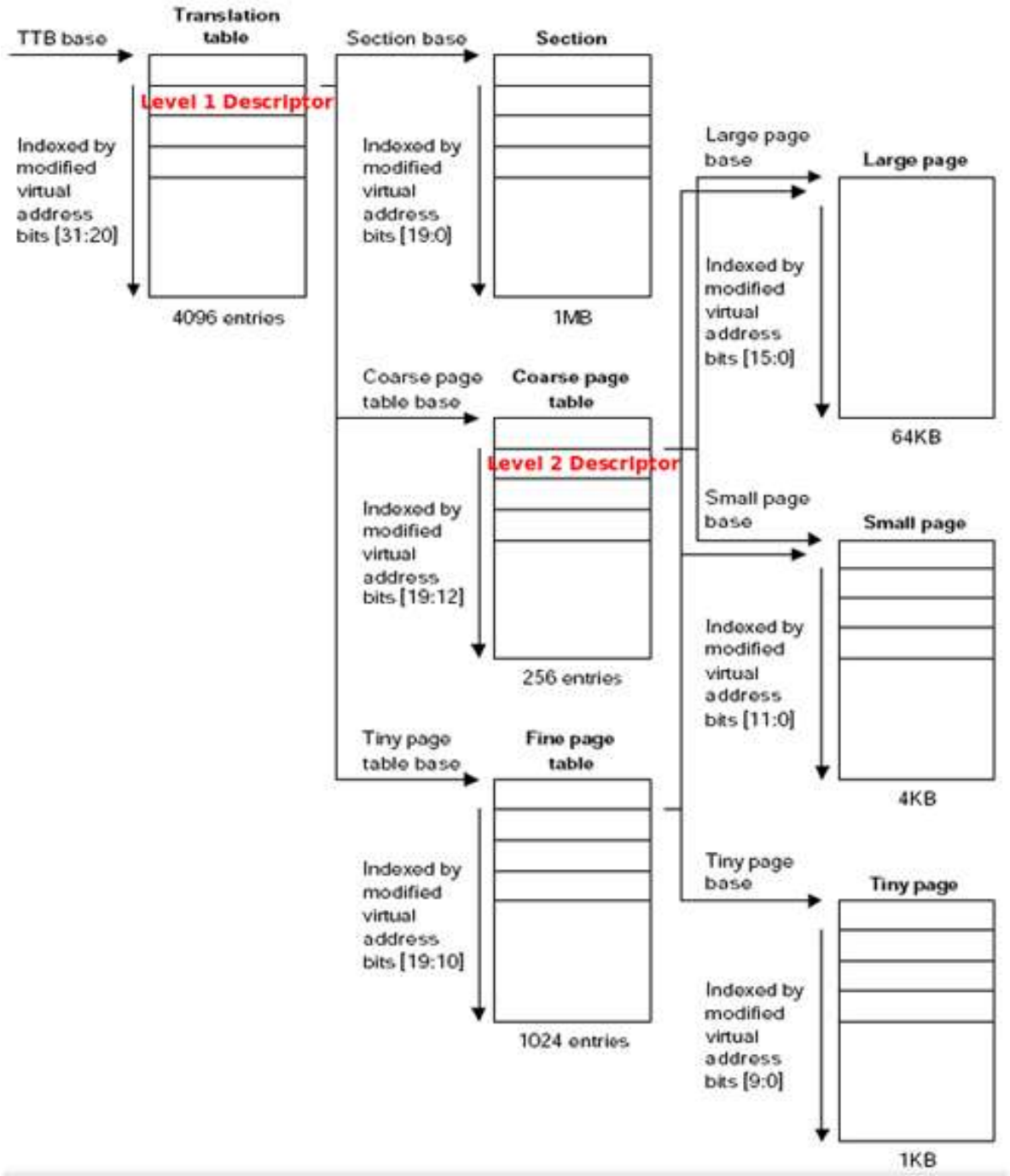


图 11. 查页表的过程

回顾一下查表的过程，首先从 CP15 的 TTB 寄存器找到一级页表的基地址，再把 VA[31:20]（共 4096 项）作为索引从表中找出一项，这个表项称为一级页描述符（Level 1 Descriptor），一个这样的表项占 4 个字节，可以是以下四种格式之一：

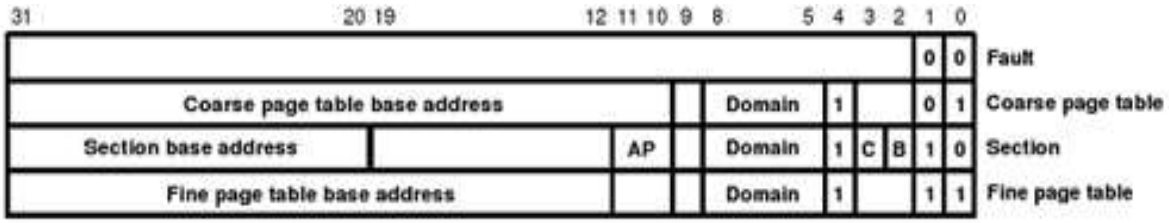


图 12. 一级页描述符

如果描述符的最低两位是 00，属于 Fault 格式，表示该范围的 VA 没有映射到 PA。如果描述符的最低两位是 10，属于 Section 格式，这种格式没有二级页表而是直接映射到物理页面，一个 Section 是 1M 的大页面，描述符中[31:20]位就是这个页面的基地址，基地址的[19:0]低位全为 0，对齐到 1M 地址边界，描述符中的 Domain 和 AP 位控制访问权限，C、B 两位控制缓存，后面再详细解释每个位的含义。如果描述符的最低两位是 01 或 11，则分别对应两种不同规格的二级页表（VA[19:12 则为 256 项]）。根据地址对齐的规律想一下，这两种页表分别是多大？从一级描述符中取出二级页表的基地址，再把 VA 的一部分作为索引去查二级描述符（Level 2 Descriptor）（如果是 Coarse Page Table 则 VA[19:12]是索引，如果是 Fine Page Table 则 VA[19:10]是索引），二级描述符可以是以下四种格式之一：

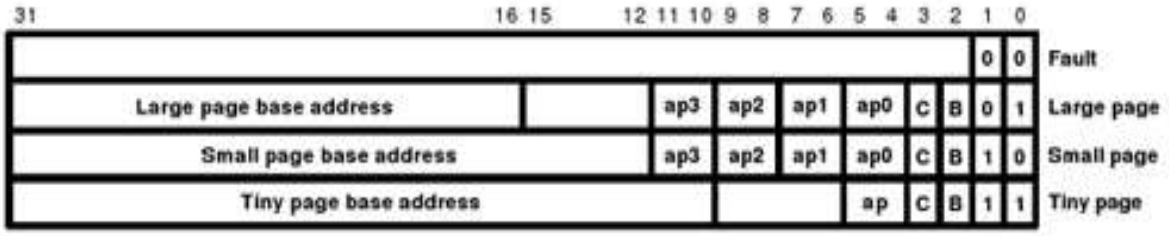


图 13. 二级页描述符

描述符最低两位是 00 属于 Fault 格式，其它三种情况分别对应三种不同规格的物理页面。Large Page 和 Small Page 有四组 AP 权限位，每组两个 bit，这样可以为每 1/4 个物理页面分别设置不同的权限，也就是说，Large Page 可以为每 16K 设置不同的权限，Small Page 可以为每 1K 设置不同的权限。

ARM920T 提供了多种页表和页面规格，但操作系统只采用其中一种，Linux 采用的就是图 9 “Translation Table Walk”所示的规格，一级描述符是 Coarse Page Table 格式，二级描述符是 Small Page 格式，每个物理页面 4K。我们以此为例，结合前面的的解释和页描述符的格式，再看一下 Translation Table Walk 的详细过程：

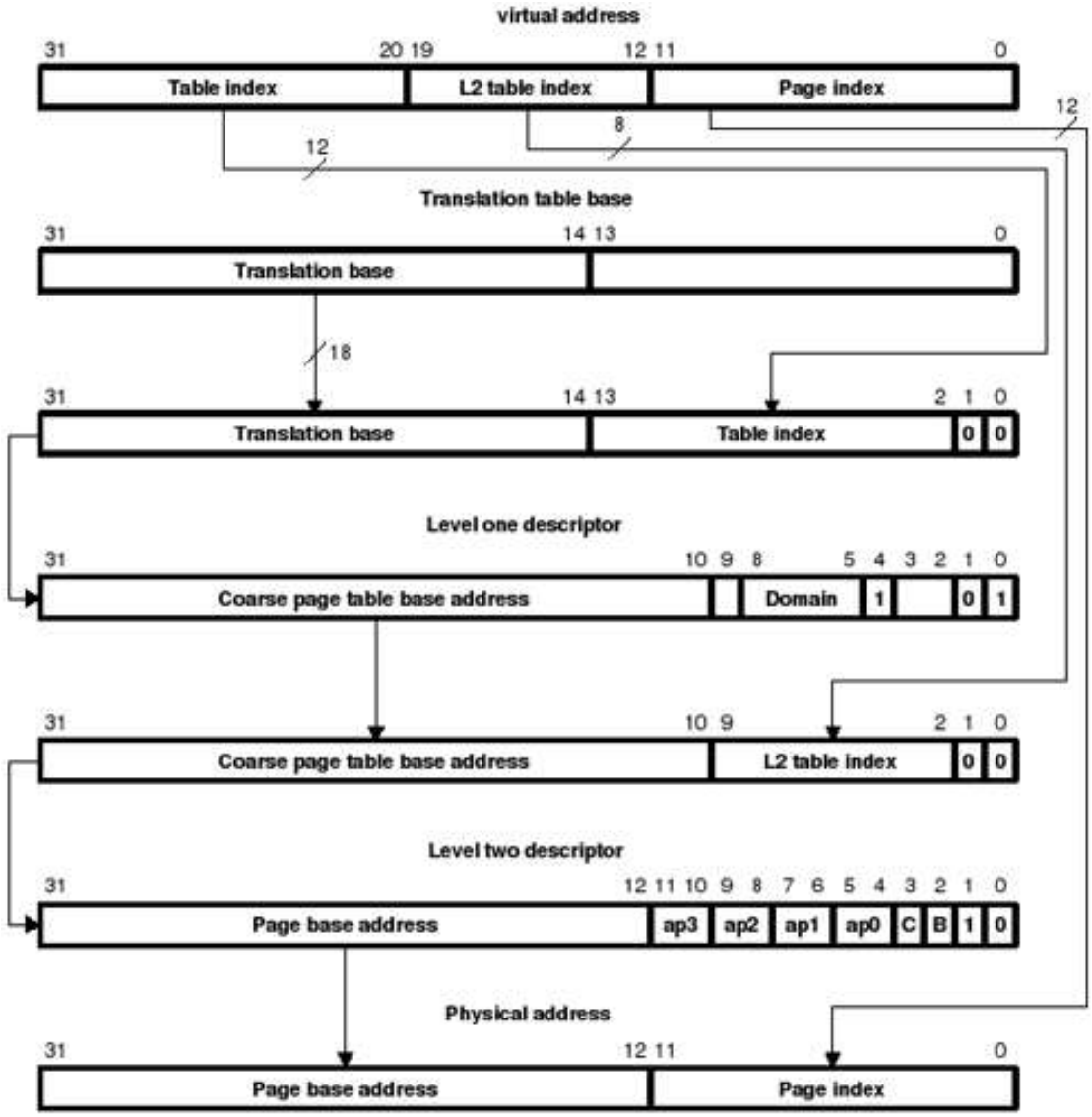


图 14. Translation Table Walk 的详细过程

从上到下依次解释如下：

1 VA 被划分为三段用于地址映射过程，各段的长度取决于页描述符的格式。

2 TTB 寄存器中只有[31:14]位有效，低 14 位全为 0，因此一级页表的基地址对齐到 16K 地址边界，而一级页表的大小也是 16K。

3 一级页表的基地址加上 VA[31:20]左移两位组装成一个物理地址。想一想为什么 VA[31:20]要左移两位占据[13:2]的位置，而空出[1:0]两位呢？类型？

4 用这个组装的物理地址从物理内存中读取一级页描述符，这是一个 Coarse Page Table 格式的描述符。

5 通过 Domain 权限检查后，Coarse Page Table 的基地址再加上 VA[19:12] 左移两位组装成一个物理地址。

6 用这个组装的物理地址从物理内存中读取二级页描述符，这是一个 Small Page 格式的描述符。

7 通过 AP 权限检查后，Small Page 的基地址再加上 VA[11:0]就是最终的物理地址。想一想为什么这次不左移两位了呢？

下面解释一下 Domain 和 AP 位。CP15 的 Domain 访问控制寄存器（见表 1 “CP15 协处理器的寄存器列表”寄存器 3）表示了 16 个域（Domain），每两位表示一个 Domain 的访问权限，以下是该寄存器的格式：

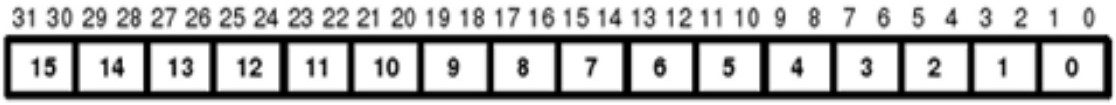


图 15. Domain Access Control Register

每个 Domain 的两个位可以取值为 00、01、10 或 11，如果取值为 00 或 10 则表示该 Domain 不可访问，如果取值为 01 则表示访问该 Domain 需要进一步检查 AP 位，如果取值为 11 则表示可以直接访问该 Domain 而无需检查 AP 位。回想一下，一级页描述符中的 Domain 字段由 4 个位组成，可以有 16 个不同的取值，就表示该描述符所描述的二级页表或 Section 属于这 16 个 Domain 中的哪一个。快速上下文切换、Domain 和多种规格的页表是 ARM 特有的机制，是针对嵌入式系统软件的特点而设计的，其它处理器不一定有类似的机制，例如也许没有 Domain 和快速上下文切换的概念，也许只有一种规格的页表。为了能够在多种不同的平台上移植，Linux 内核代码不会利用 ARM 特有的这些机制。除了这些特例之外，我们在这里介绍的其它机制都具有普遍性，读者应重点把握具有普遍意义的基本原理和基本概念。

CP15 的控制寄存器（见表 1 “CP15 协处理器的寄存器列表”寄存器 1）中的 S 和 R 位与页描述符的 AP 位合在一起决定访问权限，如下所示：

AP	S	R	Supervisor Permissions	User Permissions	Notes
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read only	No access	Supervisor read only permitted
00	0	1	Read only	Read only	Any write generates a permission fault
00	1	1	Reserved		
01	x	x	Read/write	No access	Access allowed only in supervisor mode
10	x	x	Read/write	Read only	Writes in user mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes.
xx	1	1	Reserved		

图 16. AP 权限检查

可见，同样的 AP、S、R 位对用户模式和特权模式来说具有不同的意义，特权模式的权限都不低于用户模式的权限。最后将各种由内存访问产生的异常总结如下：

Alignment Fault——以 Word 为单位的数据访问指令地址未对齐到 4 字节边界，或者以 Half Word 为单位的数据访问指令地址未对齐到 2 字节边界。

Translation Fault——页描述符的[1:0]为 00，属于 Fault 格式，无效表项。

Domain Fault——一级页描述符或 Section 所属 Domain 的权限位为 00 或 10。

Permission Fault——根据 AP 位和 CP15 寄存器 1 的 S、R 位检查访问权限，若所属 Domain 的权限位为 11 则跳过这一步检查。

External Abort——总线异常，例如此物理地址上没有挂 RAM 芯片，或者其它硬件故障。

Cache

ARM920T 有 16K 的数据 Cache 和 16K 的指令 Cache，这两个 Cache 是基本相同的，数据 Cache 多了一些写回内存的机制，后面我们以数据 Cache 为例来介绍 Cache 的基本原理。我们已经知道，Cache 中的存储单位是 Cache Line，ARM920T 的一个 Cache Line 是 32 字节，因此 16K 的 Cache 由 512 条 Cache Line 组成。要了解 Cache 的基本原理，我们从如何设计 Cache 这个问题入手。

设计 Cache 的一种最朴素的想法是，把 VA 分成以 32 字节为单位，从任何一个对齐到 32 字节地址边界的 VA 开始连续的 32 个字节（比如 0x00-0x1f，0x20-0x3f，0x40-0x5f 等等）都可以缓存到 512 条 Cache Line 中的任何一条。那么一条 Cache Line 中的 32 个字节怎么知道是来自哪个 VA 的呢？这就需要把 VA 也保存在 Cache 中，由于这 32 字节的起始地址是对齐到 32 字节

地址边界的，末 5 位全为 0，因此只需要保存 VA[31:5]即可，这称为 VA Tag[4]，Tag 是 VA 的一部分，是 Cache Line 中数据的标识，表明这 32 字节数据来自哪个 VA。这样设计的 Cache 称为全相联 Cache（Fully Associative Cache），图示如下：

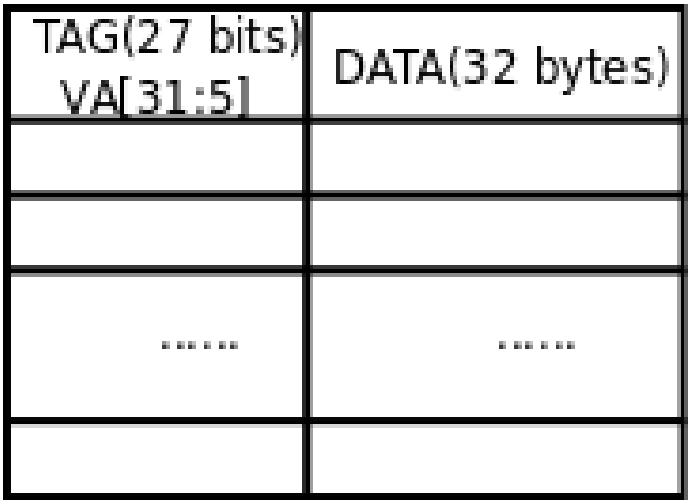


图 17. 全相联 Cache

给定一个 VA，如何在 Cache 中查找对应的数据呢？首先到 Cache 中比较查找哪一行的 Tag 等于 VA[31:5]，找到对应的 Cache Line 后，再根据 VA[4:0]决定要访问的是该 Cache Line 缓存的 32 个字节中的哪一个字节。由于有 512 条 Cache Line，如果这个 VA 没有缓存在 Cache 中则需要比较 512 次才知道，这是最坏的情况，也是最常见的情况，下面我们要改进 Cache 的设计来解决这个问题。

全相联 Cache 的特点是任何 VA 都可以缓存到任何一条 Cache Line，给定一个 VA 做查找时，由于它有可能缓存在 512 条 Cache Line 中的任何一条，就只好全部都找一遍了。如果限定某一个 VA 只允许缓存在某一条 Cache Line 中，那么查找的过程就快多了：检查一下应该缓存这个 VA 的那条 Cache Line，看 Tag 一致不一致，如果一致就是 Cache Hit，如果不一致就是 Cache Miss，可以直接访问物理内存而不必再找其它 Cache Line 了。这种设计称为直接映射 Cache（Direct Mapped Cache），如下图所示：

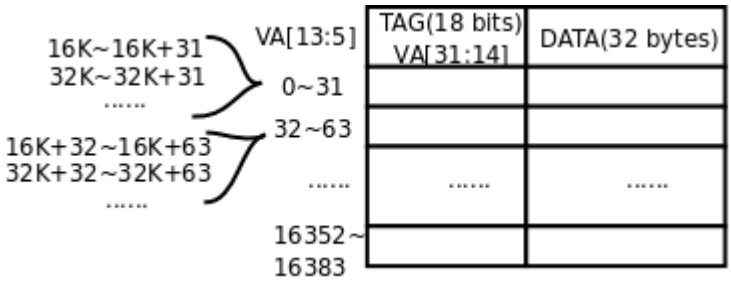


图 18. 直接映射 Cache

地址 0~31 应该缓存在第 1 条 Cache Line 中，地址 32~63 应该缓存在第 2 条 Cache Line 中，依此类推，地址 16352~16383 应该缓存在第 512 条 Cache Line

中，下一个地址应该是 16384（16K）了，我们又回到开头，地址 16K~16K+31 应该缓存在第 1 条 Cache Line 中，地址 16K+32~16K+63 应该缓存在第 2 条 Cache Line 中，依此类推，再次回到开头的地址应该是 32K，32K~32K+31 应该缓存在第 1 条 Cache Line 中，32K+32~32K+63 应该缓存在第 2 条 Cache Line 中，依此类推。读者应该可以总结出规律了：给定一个 VA，将它除以 16K 得的余数决定了它应该缓存在哪一条 Cache Line 中，那么除以 16K 的商数部分就应该是 VA Tag，用以区别 Cache Line 中缓存的到底是 0 还是 16K 还是 32K 地址上的数据。那么除以 16K 的商数和余数怎么表示呢？VA[31:14]就是除以 16K 的商数，VA[13:0]就是余数，所以上图的 Tag 处标着 VA[31:14]。余数 VA[13:0]是 16K Cache 里的一个字节偏移量，而 Cache 是按 32 字节一个 Cache Line 组织的，所以余数中的高位 VA[13:5]决定了是第几条 Cache Line，余数中的低位 VA[4:0]决定了 Cache Line 内的字节偏移量。验算一下，VA[13:5]一共是 9 位，作为 Cache Line 的编号可以表示的 Cache Line 数目正是 512 条。

直接映射 Cache 虽然查找速度很快，但也有缺点。比如，地址 0~31、16K~16K+31、32K~32K+31 都应该缓存到第 1 条 Cache Line 中，假如我们程序第一次访问地址 30，地址 0~31 的数据就从内存加载到第 1 条 Cache Line，以便下次访问能更快一些，但是我们程序第二次访问的却是地址 32770，地址 32K~32K+31 的数据就要从内存加载到第 1 条 Cache Line，把 Cache Line 里原来存的地址 0~31 的数据替换掉，以便下次访问能更快一些，但是我们程序第三次访问的却是地址 16392……这样下去，Cache 起不到任何加速作用，形同虚设，这种问题称为 Cache 抖动（Cache Thrash）。全相联 Cache 就不会有这种问题，因为任何 VA 都可以缓存到任何一条 Cache Line，可以把先后几次访问的 VA 缓存到不同的 Cache Line，就不会相互冲突。

全相联 Cache 和直接映射 Cache 各有优缺点，全相联 Cache 查找很慢，但没有抖动问题，直接映射 Cache 则正相反。为了得到更好的性能，实际 CPU 的 Cache 设计是取两者的折衷，把所有 Cache Line 分成若干个组，每一组有 n 条 Cache Line，称为 n 路组相联 Cache（n-way Set Associative Cache）。ARM920T 采用 64 路组相联 Cache，如下图所示：

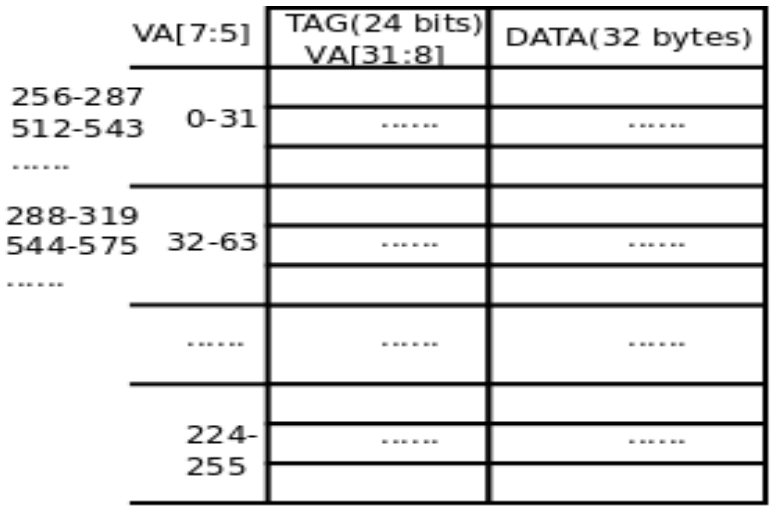


图 19. 64 路组相联 Cache

有了前面两种 Cache 概念的基础，这种 Cache 应该很好理解，512 条 Cache Line 分成 8 组，每组 64 条，地址 0-31、256-587、512-543 等等可以缓存到第 1 组 64 条 Cache Line 中的任何一条，地址 32-63、288-319、544-575 等等可以缓存到第 2 组 64 条 Cache Line 中的任何一条，依此类推。为什么说组相联 Cache 是全相联和直接映射 Cache 的一个折衷呢？如果把组分得很大，把全部 Cache Line 都分到一个组里面去，就变成了全相联 Cache；如果把组分得很小，每组只有一个 Cache Line，就变成了直接映射 Cache。作为练习，请读者自己计算一下为什么 VA Tag 是 VA[31:8]，为什么组的编号用 VA[7:5]表示。

那么，为什么组相联 Cache 的性能比直接映射 Cache 要好呢？一方面，组相联 Cache 把一条 Cache Line 上的冲突分散到了 64 条 Cache Line 上，起到了 64 倍的积极作用。而另一方面，应该缓存到同一个组的 VA 更多了：对于直接映射 Cache，在同一个组（也就是同一条 Cache Line）互相冲突的 VA 有 4G/512 个；对于组相联 Cache，在同一个组（64 条 Cache Line）互相冲突的 VA 有 4G/8 个。从这个数量关系来看，组相联 Cache 又起到了 64 倍的消极作用。难道这两种作用不会完全抵销吗？我不打算从数学上严格证明，这不是本节的重点，读者可以通过一个生活常识的例子来理解：层数一样多的两栋楼，其中一栋楼是一部电梯，每层三户，而另一栋楼是两部电梯，每层六户，每户的平均人数一样多，你认为在哪个楼里等电梯的时间较短呢？

接下来解释一下有关 Cache 写回内存的问题。Cache 写回内存有两种模式：

Write Back：Cache Line 中的数据被 CPU 核修改时并不立刻写回内存，Cache Line 和内存中的数据会暂时不一致，在 Cache Line 中有一个 Dirty 位标记这一情况。当一条 Cache Line 要被其它 VA 的数据替换时，如果不是 Dirty 的就直接替换掉，如果是 Dirty 的就先写回内存再替换。

Write Through：每当 CPU 核修改 Cache Line 中的数据时就立刻写回内存，Cache Line 和内存中的数据总是一致的。如果有多个 CPU 或设备同时访问内存，

例如采用双口 RAM，那么 Cache 中的数据和内存保持一致就非常重要了，这时相关的内存页面通常配置为 Write Through 模式。

通过读写 CP15 的相关寄存器，可以对 Cache 做以下操作：

Clean：将 Cache Line 中的数据写回内存，清除 Dirty 位。在程序中的某些同步点上用于确保 Cache Line 和内存中的数据一致。

Invalidate：在 Cache Line 中有一个 Invalid 位表示无效，将这个位置 1，下次要访问时即使 VA Tag 匹配也重新从内存读取数据。例如进程切换时需要声明前一个进程缓存在 Cache 中的数据无效。

Lock：将某个地址的数据锁定在 Cache 中，确保不被替换掉。在实时系统中，这样做可以保证某个地址的数据能在一个确定的时间内访问到。

从 Cache 中查找要访问的数据时用的是 VA，但是 Cache 写回内存要用 PA，如果写回内存时还需要查一遍页表就太没有效率了，所以实际上每条 Cache Line 中还保存了 PA[31:5]（PA Tag），完整的 Cache 构造如下图所示：

VA[7:5]	TAG(24 bits) VA[31:8]	DATA(32 bytes)	PA TAG(27 bits) PA[31:5]	Dirty	Invalid
0-31					
	……	……	……		
32-63					
	……	……	……		
……	……	……	……	……	……
224-255					
	……	……	……		

图 20. PA Tag

最后解决我们前面遗留的一个问题：页描述符中的 C、B 位具体是什么意思？

C	B	含义
0	0	不允许Cache，也不允许Write Buffer
0	1	不允许Cache，但允许Write Buffer

表 2. 页描述符中 C、B 位的含义

C 位为 1 表示允许 Cache，这种情况下用 B 位来表示 Write Through 还是 Write Back。有些页面不允许 Cache，置 C 位为 0，这种情况下可以用 B 位来选择是否允许使用 Write Buffer。Write Buffer 也是一种简单的 Cache，CPU 核执行写指令时可以把数据交给 Write Buffer，然后由 Write Buffer 负责写回内存，这时 CPU 可以执行后续指令而不必等待写回内存这个较慢的操作结束。

操作 MMU 和 Cache 的内核启动代码

bootloader 加载 linux 内核到内存并解压之后，Linux 内核首先在汇编代码中读取 CPU 的基本信息，对 CPU 做一些基本设置，创建最简单的临时页表，然后开启 MMU 和 Cache，启用虚拟内存管理（此后 CPU 核发出的地址都是虚拟地址），然后跳到 C 代码中完成其它初始化工作，比如创建完整的页表、初始化各种内核子系统、初始化硬件设备等。本节以 Linux 2.4 内核的启动代码为例，了解一下操作 MMU 和 Cache 的具体指令是怎么写的，通过实例来加深对前面内容的理解。本节的内容改编自[ARM Linux 演义]。

假设目标板的 RAM 物理地址是从 0x0800 0000 开始的（也就是说，RAM 芯片连接到 CPU 芯片上从 0x0800 0000 开始的 bank）。经过内核的若干初始化代码之后，寄存器的内容如下：

寄存器	值	含义
r4	0x0800 4000	临时页表的起始地址（物理地址）
r5	0x0800 0000	RAM起始物理地址
r8	0x0000 0c1e	页描述符标志位，后面详细说明

表 3. 寄存器的初始值

接下来的步骤是：

1 创建简单的临时页表和临时映射

2 配置与 MMU 和 Cache 相关的 CP15 寄存器

3 启用 MMU 和 Cache

临时页表存放在物理内存地址 0x0800 4000 开始的 16K（回想一下，第一级页表是 16K，有 4096 个页描述符）。后面将会把页描述符填写成 Section 格式，也就是直接映射到 1M 的大页面，这些都是内核初始化阶段临时用的，为了是写尽可能少的汇编代码，尽快启用 MMU 并跳到 C 代码中做剩下的初始化工作，在完整的两级页表建立之后临时页表就没有用了。首先将 16K 的临时页表清零：

```
mov r0, r4
mov r3, #0
add r2, r0, #0x4000 @ 16k of page table
1: str r3, [r0], #4 @ Clear page table
str r3, [r0], #4
str r3, [r0], #4
str r3, [r0], #4
teq r0, r2
bne 1b
```

下面我们将使用 Section 格式的页描述符来填充表项，由于是内核初始化阶段，还没有用户进程，我们只映射 4M 的地址空间，覆盖内核本身的代码和数据就可以了。思考一下，为什么首先要把这 16K 临时页表清零，即使没用到的表项也要清零？由于 Linux 内核在编译时确定的代码加载地址是 0xc000 8000（虚拟地址），而 bootloader 将内核代码加载到物理地址 0x0800 8000，我们需要把物理地址从 0x0800 0000 开始的 4M 映射到虚拟地址从 0xc000 0000 开始的 4M。

但是这里有一个问题：设置好页表之后，最终有一条指令是启用 MMU 的，假设该指令的 PA 是 0x0800 810c，根据我们要做的映射关系，它的 VA 应该是 0xc000 810c，没有启用 MMU 之前 CPU 核发出的都是物理地址，从 0x0800 810c 地址取这条指令来执行，然而该指令执行之后，CPU 核发出的地址都要被 MMU 拦截，CPU 核就必须用虚拟地址来取指令了，因此下一条指令应该从 0xc000 8110 处取得，然而这时 pc 寄存器（也就是 r15 寄存器）的值并没有变，CPU 核取下一条指令仍然要从 0x0800 8110 处取得，此时 0x0800 8110 已经成了非法地址了。如下图所示。

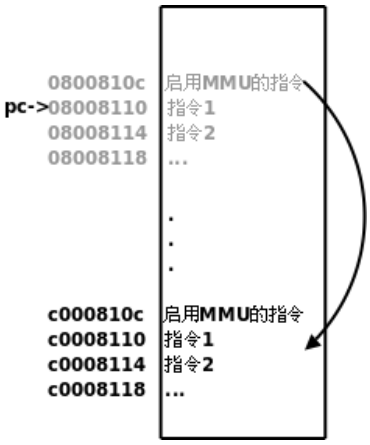


图 21. 启用 MMU 的那条指令导致的问题

为了解决这个问题，要求启用 MMU 的那条指令及其附近的指令虚拟地址跟物理地址相同，这样在启用 MMU 前后，附近指令的地址不会发生变化，从而实现平稳过渡。因此需要将物理地址从 0x0800 0000 开始的 1M 再映射到虚拟地址从 0x0800 0000 开始的 1M，也就是做一个等价映射（identity map）[5]。现在把需要建立

的映射项总结如下：

PA基地址	VA基地址	长度
0x0800 0000	0x0800 0000	1M
0x0800 0000	0x0000 0000	1M
0x0810 0000	0x0010 0000	1M
0x0820 0000	0x0020 0000	1M
0x0830 0000	0x0030 0000	1M

表 4. 需要建立的映射项

以下代码建立上面所说的等价映射。

回头看一下表 3 “寄存器的初始值”，r8 的值是页描述符标志位，r5 的值是 RAM 起始物理地址 0x0800 0000，由于要做的是等价映射，这里的 r5 既是 PA 同时也是 VA，第一条指令将 r5 当作 PA，r3=r8+r5=0x0800 0c1e 得到完整的页描述符，比对一下看看各 bit 的含义。

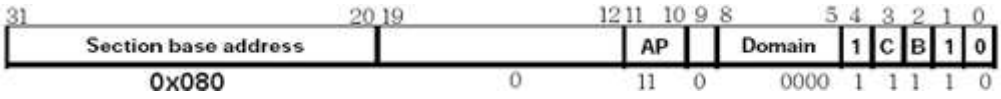


图 22. 等价映射的页描述符

```
add r3, r8, r5 @ mmuflags + start of RAM
add r0, r4, r5, lsr #18
str r3, [r0] @ identity mapping
```

该描述符所描述的 Section 属于第 0 个 Domain，AP 位是 11，可读可写，C、B 位都是 1，允许 Cache，并且 Cache 是 Write Back 方式的。第二条指令，将虚拟地址 r5 右移 18 位（对照图 14 “Translation Table Walk 的详细过程”看一下为什么是右移 18 位（高 12 位是段基址）），加到页表基地址上，得到该描述符在页表中的地址，结果保存在 r0 中。第三条指令，将第一条指令计算出的页描述符的值 r3 保存在第二条指令计算出的 r0 地址处，这样就填写好了页表项。

下面映射物理地址从 0x8000 0000 开始的 4M 到虚拟地址 0xc000 0000，其中 TEXTADDR 是 Linux 内核在编译时确定的代码加载地址 0xc000 8000，PAGE_OFFSET 定义为 0xc000 0000。请读者自己分析以下代码。

```
add r0, r4, #(TEXTADDR & 0xfff00000) >> 18          @ start of kernel
                                注：r0 = r4+ 0x3000 = 0800 4000 + 3000 = 0800 7000
str r3, [r0], #4    @ PAGE_OFFSET + 0MB 注：0800 7000 地址的内容为 0800 0c1e
add r3, r3, #1 << 20          注：r3=0810 0c1e
```

```
str r3, [r0],#4    @ PAGE_OFFSET + 1MB 注：0800 7004 地址的内容为 0810 0c1e
add r3, r3, #1 << 20          注：r3=0820 0c1e
str r3, [r0], #4    @ PAGE_OFFSET + 2MB 注：0800 7008 地址的内容为 0820 0c1e
add r3, r3, #1 << 20          注：r3=0830 0c1e
str r3,[r0],#4    @ PAGE_OFFSET + 3MB 注：0800 700c 地址的内容为 0830 0c1e
```

设置好了页表，接下来设置与 MMU 和 Cache 相关的 CP15 寄存器：

```
mov r0, #0
mcr p15, 0, r0, c7, c7 @ invalidate I,D caches on v4
mcr p15, 0, r0, c7, c10, 4@ drain write buffer on v4
mcr p15, 0, r0, c8, c7 @ invalidate I,D TLBs on v4
mcr p15, 0, r4, c2, c0 @ load page table pointer
mov r0, #0x1f @ Domains 0, 1 = client
mcr p15, 0, r0, c3, c0 @ load domain access register
mrc p15, 0, r0, c1, c0 @ get control register v4
/*
* Clear out 'unwanted' bits (then put them in if we need them)
*/
@ VI ZFRS BLDP WCAM
bic r0, r0, #0x0e00
bic r0, r0, #0x0002
bic r0, r0, #0x000c
bic r0, r0, #0x1000 @ ...0 000. .... 000.
/*
* Turn on what we want
*/
orr r0, r0, #0x0031
orr r0, r0, #0x2100 @ ..1. ...1 ..11 ...1
#ifdef CONFIG_CPU_ARM920_D_CACHE_ON
orr r0, r0, #0x0004 @ .... .... .... .1..
#endif
#ifdef CONFIG_CPU_ARM920_I_CACHE_ON
orr r0, r0, #0x1000 @ ...1 .... .... ....
#endif
```

这一段有很多协处理器指令，请读者对照[S3C2410 用户手册]和代码中的注释查看各指令的含义。大体上来说做了以下事情：首先禁用指令和数据 Cache，等待 Write Buffer 写回内存，然后用 r4 寄存器的值设置 CP15 的 TTB 寄存器，然后设置 Domain 权限位，我们先前填写的页描述符都属于第 0 个 Domain，Domain 寄存器中第 0 个 Domain 的权限位设置为 11，表示访问不必检查 AP 位。接下来读出 CP15 的控制寄存器的值来修改，准备启用 MMU，根据内核配置决定是否启用数据和指令 Cache，修改之后一并写回控制寄存器，使设置生效：

```
mcr p15, 0, r0, c1, c0
```

相关索引

C

M

P

S

T

V

VA，Virtual Address，虚拟地址，虚拟地址和物理地址的概念
(参见 PA，Physical Address，物理地址)
Virtual Memory Management，虚拟内存管理，虚拟内存管理

W

Write Back，Cache(参见 Write Through)
Write Through，Cache(参见 Write Back)

1 对于 32 位的 CPU，从 CPU 核这边看地址线是 32 条（图中只是示意性地画了 4 条地址线），可寻址空间是 4GB，但是通常嵌入式处理器的 CPU 外部地址引脚不会有这么多条地址线，因为引脚是芯片上十分有限而宝贵的资源，而且也不太可能用到 4GB 这么大的物理内存。另一方面，在启用 MMU 的情况下 VA 地址空间和 PA 地址空间是完全独立的，PA 地址空间既可以小于也可以大于 VA 地址空间，例如有些 32 位的服务器可以配置大于 4GB 的物理内存。

2 这里说 Cache 是用 VA 来索引数据的，只是针对一些嵌入式处理器，实际上大多数 PC 和服务器的 CPU 都有两级 Cache，靠近 CPU 核的一级缓存是以 VA 来索引数据的，而靠近物理内存的二级缓存则是以 PA 来索引数据的。

3 如果读者看了[S3C2410 用户手册]可能会注意到有 MVA（Modified Virtual Address）这个概念，这属于 ARM 的快速上下文切换（Fast Context Switch）机制，适用于一些小型的 RTOS，每个进程的地址空间不超过 32M，Linux 并没有利用快速上下文切换机制，因此在我们的讨论当中，VA 和 MVA 不加区分，认为是相同的。

4 我们说 Cache 的大小是 16K，是指 Cache 能缓存 16K 的内存数据，其实 Cache 还需要保存相应的 Tag 和其它标志位，其存储容量应该是大于 16K 的。另外，Cache 的构造和内存不同，不必以字节为单位来存储，例如 VA[31:5]这个 Tag 有 27 个 bit，既不是 3 个字节也不是 4 个字节。

5 事实上，以上解释并不完全正确，这里还有一个更复杂的细节，启用 MMU 的指令在执行时，后面两条指令已经预取到 CPU 流水线里了，如果利用那两条指令跳转到 0xc000 8110 不就行了？但是流水线是靠不住的，跳转和异常都会清空流水线，[ARM 参考手册]的 Chapter A2 详细解释了这种情况，按该手册的建议应该采用等价映射的方法解决这个问题。