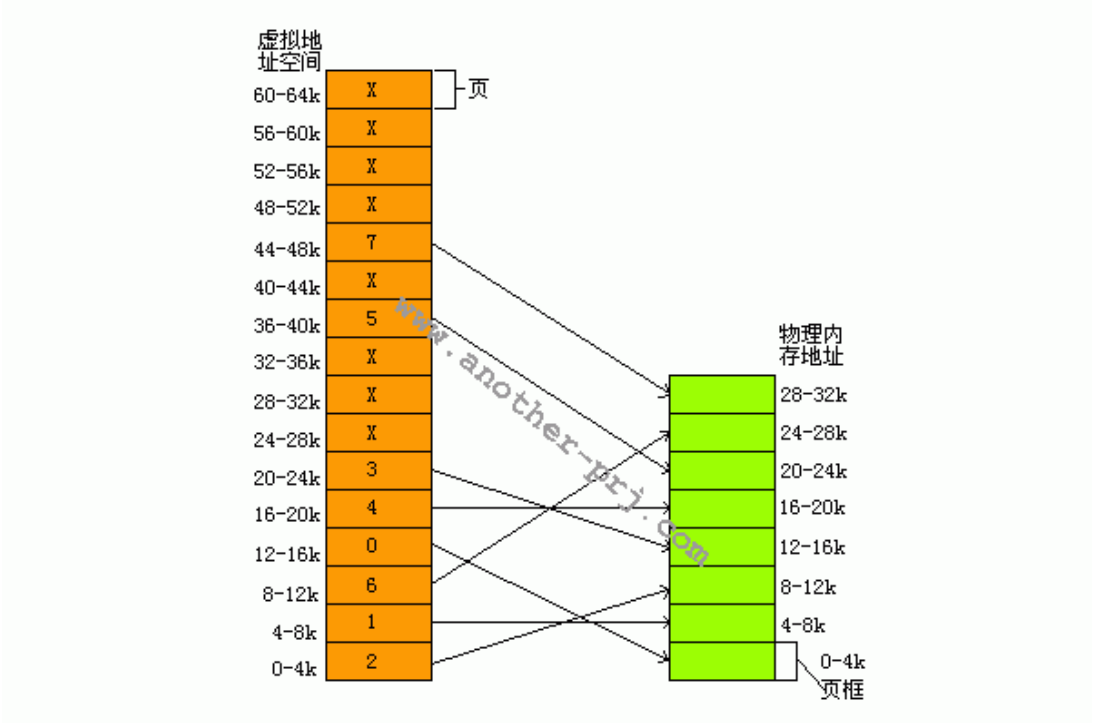
作者：蔡于清
www.another-prj.com

MMU,全称 Memory Manage Unit, 中文名——存储器管理单元。

许多年以前，当人们还在使用 DOS 或是更古老的操作系统的时候，计算机的内存还非常小，一般都是以 K 为单位进行计算，相应的，当时的程序规模也不大，所以内存容量虽然小，但还是可以容纳当时的程序。但随着图形界面的兴起还用用户需求的不断增大，应用程序的规模也随之膨胀起来，终于一个难题出现在程序员的面前，那就是应用程序太大以至于内存容纳不下该程序，通常解决的办法是把程序分割成许多称为**覆盖块（overlay）**的片段。覆盖块 0 首先运行，结束时他将调用另一个覆盖块。虽然覆盖块的交换是由 OS 完成的，但是必须先由程序员把程序先进行分割，这是一个费时费力的工作，而且相当枯燥。人们必须找到更好的办法从根本上解决这个问题。不久人们找到了一个办法，这就是**虚拟存储器(virtual memory).**虚拟存储器的基本思想是程序，数据，堆栈的总的大小可以超过物理存储器的大小，操作系统把当前使用的部分保留在内存中，而把其他未被使用的部分保存在磁盘上。比如对一个 16MB 的程序和一个内存只有 4MB 的机器，OS 通过选择，可以决定各个时刻将哪 4M 的内容保留在内存中，并在需要时在内存和磁盘间交换程序片段，这样就可以把这个 16M 的程序运行在一个只具有 4M 内存机器上了。而这个 16M 的程序在运行前不必由程序员进行分割。

任何时候，计算机上都存在一个程序能够产生的地址集合，我们称之为**地址范围**。这个范围的大小由 CPU 的位数决定，例如一个 32 位的 CPU，它的地址范围是 0~0xFFFFFFFF (4G),而对于一个 64 位的 CPU，它的地址范围为 0~0xFFFFFFFFFFFFFFFF (64T).这个范围就是我们的程序能够产生的地址范围，我们把这个地址范围称为**虚拟地址空间**，该空间中的某一个地址我们称之为**虚拟地址**。与虚拟地址空间和虚拟地址相对应的则是**物理地址空间**和**物理地址**，大多数时候我们的系统所具备的物理地址空间只是虚拟地址空间的一个子集，这里举一个最简单的例子直观地说明这两者，对于一台内存为 256MB 的 32bit x86 主机来说，它的虚拟地址空间范围是 0~0xFFFFFFFF（4G）,而物理地址空间范围是 0x000000000~0x0FFFFFFF（256MB）。在没有使用虚拟存储器的机器上，虚拟地址被直接送到内存总线上，使具有相同地址的物理存储器被读写。而在使用了虚拟存储器的情况下，虚拟地址不是被直接送到内存地址总线上，而是送到内存管理单元——MMU（主角终于出现了：]）。他由一个或一组芯片组成，一般存在与协处理器中，其功能是把虚拟地址映射为物理地址。

大多数使用虚拟存储器的系统都使用一种称为**分页（paging）**。虚拟地址空间划分成称为**页（page）**的单位,而相应的物理地址空间也被进行划分，单位是页**框(frame).**页和页框的大小必须相同。接下来配合图片我以一个例子说明页与页框之间在 MMU 的调度下是如何进行映射的：



在这个例子中我们有一台可以生成 16 位地址的机器，它的虚拟地址范围从 0x0000~0xFFFF(64 K),而这台机器只有 32K 的物理地址，因此他可以运行 64K 的程序，但该程序不能一次性调入内存运行。这台机器必须有一个达到可以存放 64K 程序的外部存储器（例如磁盘或是 FLASH），以保证程序片段在需要时可以被调用。在这个例子中，页的大小为 4K,页框大小与页相同（这点是必须保证的，内存和外围存储器之间的传输总是以页为单位的），对应 64K 的虚拟地址和 32 K 的物理存储器，他们分别包含了 16 个页和 8 个页框。

我们先根据上图解释一下分页后要用到的几个术语，在上面我们已经接触了**页**和**页框**，上图中绿色部分是物理空间，其中每一格表示一个物理页框。橘黄色部分是虚拟空间，每一格表示一个页，它由两部分组成，分别是 **Frame Index(页框索引)**和**位 p（present 存在位）**，Frame Index 的意义很明显，它指出本页是往哪个物理页框进行映射的，位 p 的意义则是指出本页的映射是否有效，如上图，当某个页并没有被映射时（或称“映射无效”，Frame Index 部分为 X），该位为 0，映射有效则该位为 1。

我们执行下面这些指令（本例子的指令不针对任何特定机型，都是伪指令）

例 1：

MOVE REG,0 //将 0 号地址的值传递进寄存器 REG.

虚拟地址 0 将被送往 MMU,MMU 看到该虚地址落在页 0 范围内（页 0 范围是 0 到 4095），从上图我们看到页 0 所对应（映射）的页框为 2（页框 2 的地址范围是 8192 到 12287），因此 MMU 将该虚拟地址转化为物理地址 8192，并把地址 8192 送到地址总线上。内存对 MMU 的映射一无所知，它只看到一个对地址 8192 的读请求并执行它。MMU 从而把 0 到 4096 的虚拟地址映射到 8192 到 12287 的物理地址。

例 2：

MOVE REG,8192

被转换为

MOVE REG,24576

因为虚拟地址 8192 在页 2 中，而页 2 被映射到页框 6（物理地址从 24576 到 28671）

例 3：

MOVE REG,20500

被转换为

MOVE REG,12308

虚拟地址 20500 在虚页 5（虚拟地址范围是 20480 到 24575）距开头 20 个字节处，虚页 5 映射到页框 3（页框 3 的地址范围是 12288 到 16383），于是被映射到物理地址 12288+20=12308。

通过适当的设置 MMU，可以把 16 个虚页隐射到 8 个页框中的任何一个，但是这个方法并没有有效的解决虚拟地址空间比物理地址空间大的问题。从上图中我们可以看到，我们只有 8 个页框（物理地址），但我们有 16 个页（虚拟地址），所以我们只能把 16 个页中的 8 个进行有效的映射。我们看看例 4 会发生什么情况

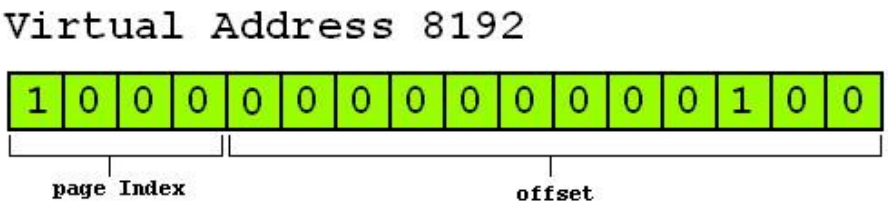
MOV REG,32780

虚拟地址 32780 落在页 8 的范围内，从上图总我们看到页 8 没有被有效的进行映射（该页被打上 X），这是又会发生什么？MMU 注意到这个页没有被映射，于是通知 CPU 发生一个**缺页故障（page fault）.**这种情况下操作系统必须处理这个页故障，它必须从 8 个物理页框中找到 1 个当前很少被使用的页框并把该页框的内容写入外围存储器（这个动作被称为 **page copy**），随后把需要引用的页（例 4 中是页 8）映射到刚才释放的页框中（这个动作称为修改映射关系），然后从新执行产生故障的指令（MOV REG,32780）。假设操作系统决定释放页框 1，那么它将把虚页 8 装入物理地址的 4-8K,并做两处修改：首先把标记虚页 1 未被映射（原来虚页 1 是被影射到页框 1 的），以使以后任何对虚拟地址 4K 到 8K 的访问都引起页故障而使操作系统做出适当的动作（这个动作正是我们现在在讨论的），其次他把虚页 8 对应的页框号由 X 变为 1，因此重新执行 MOV REG,32780 时，MMU 将把 32780 映射为 4108。

我们大致了解了 MMU 在我们的机器中扮演了什么角色以及它基本的工作内容是什么，下面我们将举例子说明它究竟是如何工作的（注意，本例中的 MMU 并无针对某种特定的机型，它是所有 MMU 工作的一个抽象）。

我们已经知道，大多数使用虚拟存储器的系统都使用一种称为分页（paging）的技术，就象我们刚才所举的例子，虚拟地址空间被分成大小相同的一组页，每个页有一个用来标示它的**页号**（这个页号一般是它在该组中的索引，这点和 C/C++中的数组相似）。在上面的例子中 0~4K 的页号为 0，4~8K 的页号为 1，8~12K 的页号为 2，以此类推。而虚拟地址（注意：是一个确定的地址，不是一个空间）被 MMU 分为 2 个部分，第一部分是**页号索引（page Index），**第二部分

则是相对该页首地址的**偏移量（offset）．**。我们还是以刚才那个 16 位机器结合下图进行一个实例说明，该实例中，虚拟地址 8196 被送进 MMU,MMU 把它映射成物理地址。16 位的 CPU 总共能产生的地址范围是 0~64K,按每页 4K 的大小计算，该空间必须被分成 16 个页。而我们的虚拟地址第一部分所能够表达的范围也必须等于 16（这样才能索引到该页组中的每一个页）,也就是说这个部分至少需要 4 个 bit。一个页的大小是 4K(4096),也就是说偏移部分必须使用 12 个 bit 来表示(2^12= 4096，这样才能访问到一个页中的所有地址),8196 的二进制码如下图所示：



该地址的页号索引为 0010（二进制码），既索引的页为页 2，第二部分为 000000000100（二进制），偏移量为 4。页 2 中的页框号为 6（页 2 映射在页框 6，见上图），我们看到页框 6 的物理地址是 24~28K。于是 MMU 计算出虚拟地址 8196 应该被映射成物理地址 24580（页框首地址+偏移量=24576+4=24580）。同样的，若我们对虚拟地址 1026 进行读取，1026 的二进制码为 0000010000000010，page index=0000=0,offset=010000000010=1026。页号为 0，该页映射的页框号为 2，页框 2 的物理地址范围是 8192~12287，故 MMU 将虚拟地址 1026 映射为物理地址 9218（页框首地址+偏移量=8192+1026=9218）

以上就是 MMU 的工作过程。

下面我们针对 s3c2410 的 MMU(注 1)进行讲解。

S3c2410 总共有 4 种内存映射方式，分别是：

1．Fault (无映射)

2．Coarse Page (粗表)

3．Section (段)

4．Fine Page (细表)

我们以 **Section(段)**进行说明。

ARM920T 是一个 32bit 的 CPU,它的虚拟地址空间为 2^32=4G。而在 **Section 模式**，这 4G 的虚拟空间被分成一个一个称为**段（Section）**的单位(与我们上面讲的页在本质上其实是一致的),每个段的长度是 1M (而我们之前所使用的页的长度是 4K)。4G 的虚拟内存总共可以被分成 4096 个段（1M*4096=4G）,因此我们必须用 4096 个描述符来对这组段进行描述，每个描述符占用 4 个 Byte,故这组描述符的大小为 16KB (4K*4096),这 4096 个描述符构为一个表格，我们称其为 **Tralaton Table.**

Section base address		AP		Domain	1	C	B	1	0

上图是描述符的结构

Section base address:段基地址（相当于页框号首地址）

AP: 访问控制位 Access Permission

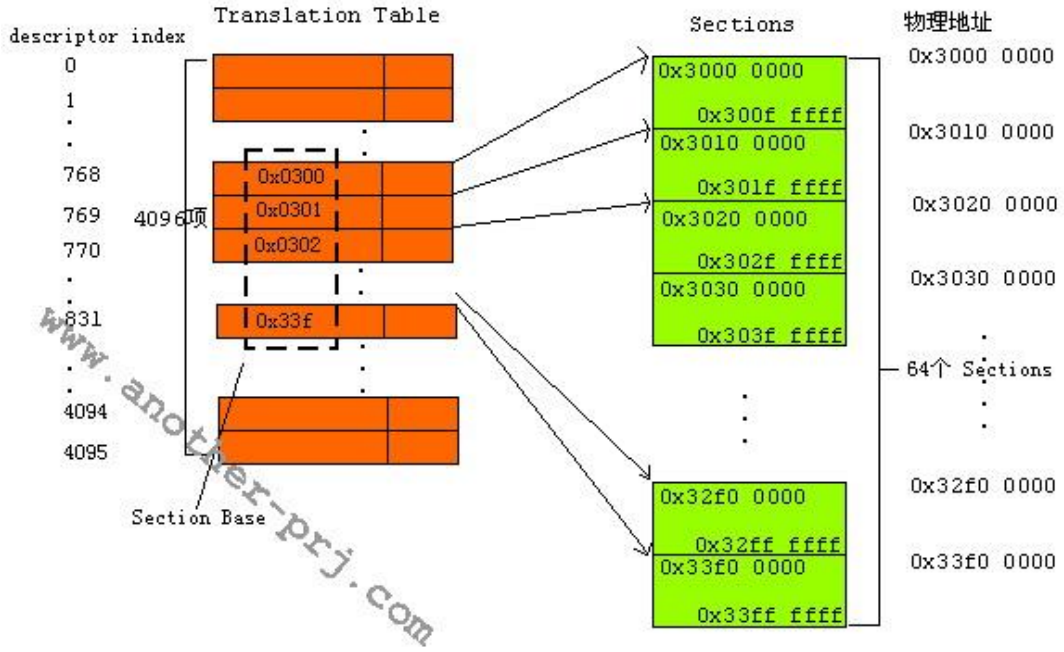
Domain: 访问控制寄存器的索引。Domain 与 AP 配合使用，对访问权限进行检查

C:当 C 被置 1 时为 write-through (WT)模式

B: 当 B 被置 1 时为 write-back (WB)模式

（C,B 两个位在同一时刻只能有一个被置 1）

下面是 s3c2410 内存映射后的一个示意图：



我的 s3c2410 上配置的 SDRSAM 大小为 64M,该 SDRAM 的物理地址范围是 0x3000 0000~0x33FF FFFF(属于 Bank 6)，由于 1 个 Section 的大小是 1M,所以该物理空间可以被分成 64 个物理段(页框).

在 Section 模式下，送进 MMU 的虚拟地址**(注 1)**被分为两部分（这点和我们上面举的例子是一样的），这两部分为 **Descriptor Index**(相当于上面例子的 Page Index)和 **Offset**,descript index 长度为 12bit(2^12=4096,从这个关系式你能看出什么？：))，Offset 长度为 20bit（2^20=1M，你又能看出什么？：)）.观察一下一个描述符（Descriptor）中的 Section Base Address 部分，它长度为 12 bit，里面的值是该虚拟段（页）映射成的物理段（页框）的物理地址前 12bit，

由于每一个物理段的长度都是 1M，所以物理段首地址的后 20bit 总是为 0x00000(每个 Section 都是以 1M 对齐)，确定一个物理地址的方法是 物理页框基地址+虚拟地址中的偏移部分=Section Base Address<<20+Offset ,呵呵，可能你有点糊涂了，还是举一个实际例子说明吧。假设现在执行指令

MOV REG, 0x30000012

虚拟地址的二进制码为 00110000 00000000 00000000 00010010

前 12 位是 Descriptor Index= 00110000 0000=768,故在 Translation Table 里面找到第 768 号描述符，该描述的 Section Base Address=0x0300,也就是说描述符所描述的虚拟段（页）所映射的物理段（页框）的首地址为 0x3000 0000（物理段（页框）的基地址=Section Base Address 左移 20bit=0x0300<<20=0x3000 0000），而 Offset=000000 00000000 00010010=0x12,故虚拟地址 0x30000012 映射成的物理地址=0x3000 0000+0x12=0x3000 0012（物理页框基地址+虚拟地址中的偏移）。你可能会问怎么这个虚拟地址和映射后的物理地址一样？这是由我们定义的映射规则所决定的。在这个例子中我们定义的映射规则是把虚拟地址映射成和他相等的物理地址。我们这样书写映射关系的代码：

```
void mem_mapping_linear(void)
{
    unsigned long descriptor_index, section_base, sdram_base, sdram_size;
    sdram_base=0x30000000;
    sdram_size=0x 4000000;
    for (section _base= sdram_base,descriptor_index = section _base>>20;
        section _base < sdram_base+ sdram_size;
        descriptor_index+=1;section _base +=0x100000)
    {
        *(mmu_tlb_base + (descriptor_index)) = (section _base>>20) | MMU_OTHER_SE
CDESC;
    }
}
```

上面的这段段代码把虚拟空间 0x3000 0000~0x33FF FFFF 映射到物理空间 0x3000 0000~0x33FF FFFF，由于虚拟空间与物理空间空间相吻合，所以虚拟地址与他们各自对应的物理地址在值上是一致的。当初始完 Translation Table 之后，记得要把 Translation Table 的首地址(第 0 号描述符的地址)加载进协处理器 CP15 的 Control Register2(2 号控制寄存器)中,该控制寄存器的名称叫做 Translation table base (TTB) register。

以上讨论的是 descriptor 中的 Section Base Address 以及虚拟地址和物理地址的映射关系,然而 MMU 还有一个重要的功能，那就是**访问控制机制(Access Permission)。**

简单说访问控制机制就是 CPU 通过某种方法判断当前程序对内存的访问是否合法（是否有权限对该内存进行访问），如果当前的程序并没有权限对即将访问的内存区域进行操作，则 CPU 将

引发一个异常，s3c2410 称该异常为 **Permission fault**，x86 架构则把这种异常称之为**通用保护异常（General Protection）**，什么情况会引起 Permission fault 呢？比如处于 User 级别的程序要对一个 System 级别的内存区域进行写操作，这种操作是越权的，应该引起一个 Permission fault，搞过 x86 架构的朋友应该听过**保护模式（Protection Mode）**,保护模式就是基于这种思想进行工作的，于是我们也可以这么说：s3c2410 的访问控制机制其实就是一种保护机制。那 s3c2410 的访问控制机制到底是由什么元素去参与完成的呢？它们间是怎么协调工作的呢？这些元素总共有：

1．协处理器 CP15 中 Control Register3：DOMAIN ACCESS CONTROL REGISTER

2．段描述符中的 AP 位和 Domain 位

3．协处理器 CP15 中 Control Register1(控制寄存器 1)中的 S bit 和 R bit

4．协处理器 CP15 中 Control Register5(控制寄存器 5)

5．协处理器 CP15 中 Control Register6(控制寄存器 6)

DOMAIN ACCESS CONTROL REGISTER 是**访问控制寄存器**，该寄存器有效位为 32，被分成 16 个区域，每个区域由两个位组成，他们说明了当前内存的访问权限检查的级别，如下图所示：

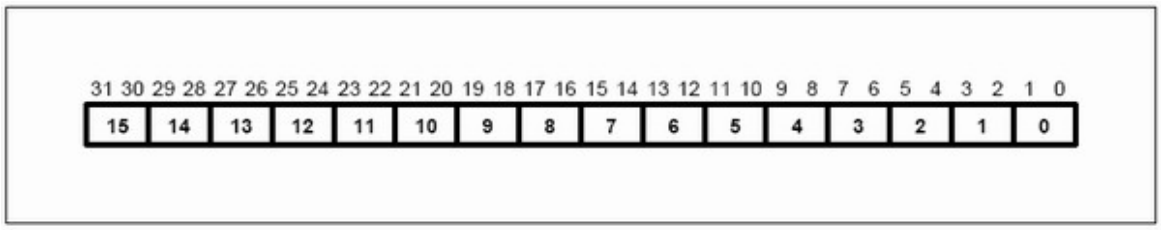


Figure 3-10. Domain Access Control Register Format

每区域可以填写的值有 4 个，分别为 00,01,10,11(二进制)，他们的意义如下所示：

Table 3-5. Interpreting Access Control Bits in Domain Access Control Register

Value	Meaning	Notes
00	No Access	Any access will generate a domain fault.
01	Client	Accesses are checked against the access permission bits in the section or page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are *not* checked against the access permission bits so a permission fault cannot be generated.

00：当前级别下，该内存区域不允许被访问，任何的访问都会引起一个 domain fault

01：当前级别下，该内存区域的访问必须配合该内存区域的段描述符中 AP 位进行权检查

10：保留状态（我们最好不要填写该值，以免引起不能确定的问题）

11：当前级别下，对该内存区域的访问都不进行权限检查。

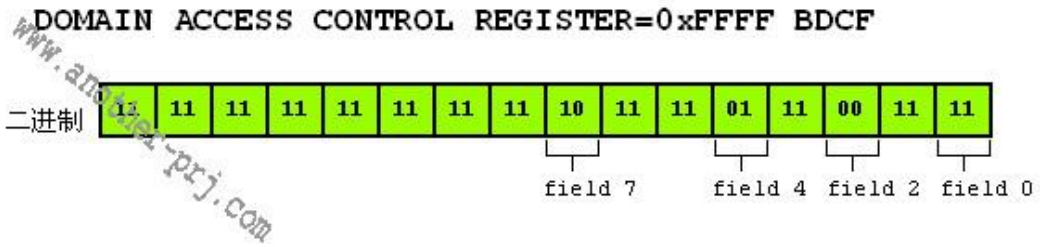
我们再来看看 discriptor 中的 Domain 区域，该区域总共有 4 个 bit,里面的值是对 DOMAIN ACCESS CONTROL REGISTER 中 16 个区域的索引.而 **AP 位配合 S bit 和 A bit** 对当前描述符描述的内存区域被访问权限的说明,他们的配合关系如下图所示:

Table 3-6. Interpreting Access Permission (AP) Bits

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		

AP 位也是有四个值,我结合实例对其进行说明.

在下面的例子中，我们的 DOMAIN ACCESS CONTROL REGISTER 都被初始化成 0xFFFF BDCF,如下图所示：



例 1：

Discriptor 中的 domain=4,AP=10(这种情况下 S bit ,A bit 被忽略)

假设现在我要对该描述符描述的内存区域进行访问：

由于 domain=4,而 DOMAIN ACCESS CONTROL REGISTER 中 field 4 的值是 01，系统会对该访问进行访问权限的检查。

假设当前 CPU 处于 Supervisor 模式下，则程序可以对该描述符描述的内存区域进行读写操作。

假设当前 CPU 处于 User 模式下，则程序可以对该描述符描述的内存进行读访问，若对其进行写操作则引起一个 permission fault.

例 2：

Discriptor 中的 domain=0,AP=10(这种情况下 S bit ,A bit 被忽略)

domain=0,而 DOMAIN ACCESS CONTROL REGISTER 中 field 0 的值是 11，系统对任何内存区域的访问都不进行访问权限的检查。

由于统对任何内存区域的访问都不进行访问权限的检查，所以无论 CPU 处于合种模式下（Supervisor 模式或是 User 模式），程序对该描述符描述的内存都可以顺利地进行读写操作

例 3：Discriptor 中的 domain=4,AP=11(这种情况下 S bit ,A bit 被忽略)

由于 domain=4,而 DOMAIN ACCESS CONTROL REGISTER 中 field 4 的值是 01，系统会对该访问进行访问权限的检查。

由于 AP=11，所以无论 CPU 处于合种模式下（Supervisor 模式或是 User 模式），程序对该描述符描述的内存都可以顺利地进行读写操作

例 4：

Discriptor 中的 domain=4,AP=00, S bit=0,A bit=0

由于 domain=4,而 DOMAIN ACCESS CONTROL REGISTER 中 field 4 的值是 01，系统会对该访问进行访问权限的检查。

由于 AP=00，S bit=0,A bit=0,所以无论 CPU 处于合种模式下（Supervisor 模式或是 User 模式），程序对该描述符描述的内存都只能进行读操作，否则引起 permission fault.

通过以上 4 个例子我们得出两个结论：

1．对某个内存区域的访问是否需要进行权限检查是由该内存区域的描述符中的 Domain 域决定的。

2．某个内存区域的访问权限是由该内存区域的描述符中的 AP 位和协处理器 CP15 中 Control Register1(控制寄存器 1)中的 S bit 和 R bit 所决定的。

关于访问控制机制我们就讲到这里.

注 1:对于 s3c2410 来说,MMU 是以 Modify Visual Address(MVA)进行寻址的,这个地址是 Virtual Address 的一个变换,我将在以后谈论到进程切换的时候中向大家介绍 MVA

下一篇文档我将介绍 s3c2410 的 Caches,Write Buffer 并给出开启 MMU 的源代码,请把本文档和下一篇文档配合阅读。