第 3 部分

32 位保护模式

- ✧ 学习处理器 32 位保护模式的工作原理，包括分段、分页、特权级、保护、中断和异常中断等。
- ✧ 学习 32 位保护模式下的汇编语言程序设计技术。
- ✧ 通过多个实例了解操作系统如何在保护模式下加载应用程序，并提供各种管理服务。
- ✧ 学会用 Bochs 虚拟机调试 32 位保护模式下的程序。

第 10 章　32 位 x86 处理器编程架构

所谓处理器架构，或者处理器编程架构，是指一整套的硬件结构，以及与之相适应的工作状态，这其中的灵魂部分就是一种设计理念，决定了处理器的应用环境和工作模式，也决定了软件开发人员如何在这种模式下解决实际问题。架构内的资源对程序员来说是可见的、可访问的，受程序的控制以改变处理器的运行状态；非架构的资源取决于具体的硬件实现。

处理器架构实际上是不断扩展的，新处理器必须延续旧的设计思路，并保持兼容性和一致性；同时还会有所扩充和增强。

Intel 32 位处理器架构简称 IA-32（Intel Architecture，32-bit），是以 1978 年的 8086 处理器为基础发展起来的。在那个时候，他们只是想造一款特别牛的处理器，也没考虑到架构。尽管那些人是专家，但和我们一样不是千里眼，这是很正常的。

正如我们已经知道的，8086 有 20 根地址线，可以寻址 1MB 内存。但是，它内部的寄存器是 16 位的，无法在程序中访问整个 1MB 内存。所以，它也是第一款支持内存分段模型的处理器。还有，8086 处理器只有一种工作模式，即实模式。当然，在那时，还没有实模式这一说。

由于 8086 处理器的成功，推动着 Intel 公司不断地研发更新的处理器，32 位的时代就这样到来了。到目前为止，到底有多少种类型，我也说不清楚。尽管 8086 是 16 位的处理器，但它也是 32 位架构内的一部分。原因在于，32 位的处理器架构是从 8086 那里发展来的，是基于 8086 的，具有延续性和兼容性。

就我们曾经用过的产品而言，32 位的处理器有 32 根地址线，数据线的数量是 32 根或者 64 根。特别是最近最新的处理器，都是 64 根。因此，它可以访问 2^{32}，即 4GB 的内存，而且每次可以读写连续的 4 字节或者 8 字节，这称为双字（Double Word）或者 4 字（Quad Word）访问。当然，如果你要按字节或者字来访问内存，也是允许的。

我总说，处理器虽小，功能却异常复杂。要想把 32 位处理器的所有功能都解释清楚，不是一件简单的事情。它不单单是地址线和数据线的扩展，实际上还有更多的部分，包括高速缓存、流水线、浮点处理部件、多处理器（核）管理、多媒体扩展、乱序执行、分支预测、虚拟化、温度和电源管理等。在这本书里，我的一个基本原则是，如果你不能讲清楚，干脆就不要提它。因此，我只讲那些现在用得上的东西。

10.1　IA-32 架构的基本执行环境

10.1.1　寄存器的扩展

在 16 位处理器内，有 8 个通用寄存器 AX、BX、CX、DX、SI、DI、BP 和 SP，其中，前 4 个还可以拆分成两个独立的 8 位寄存器来用，即 AH、AL、BH、BL、CH、CL、DH 和 DL。如图 10-1 所示，32 位处理器在 16 位处理器的基础上，扩展了这 8 个通用寄存器的长度，使之达到 32 位。

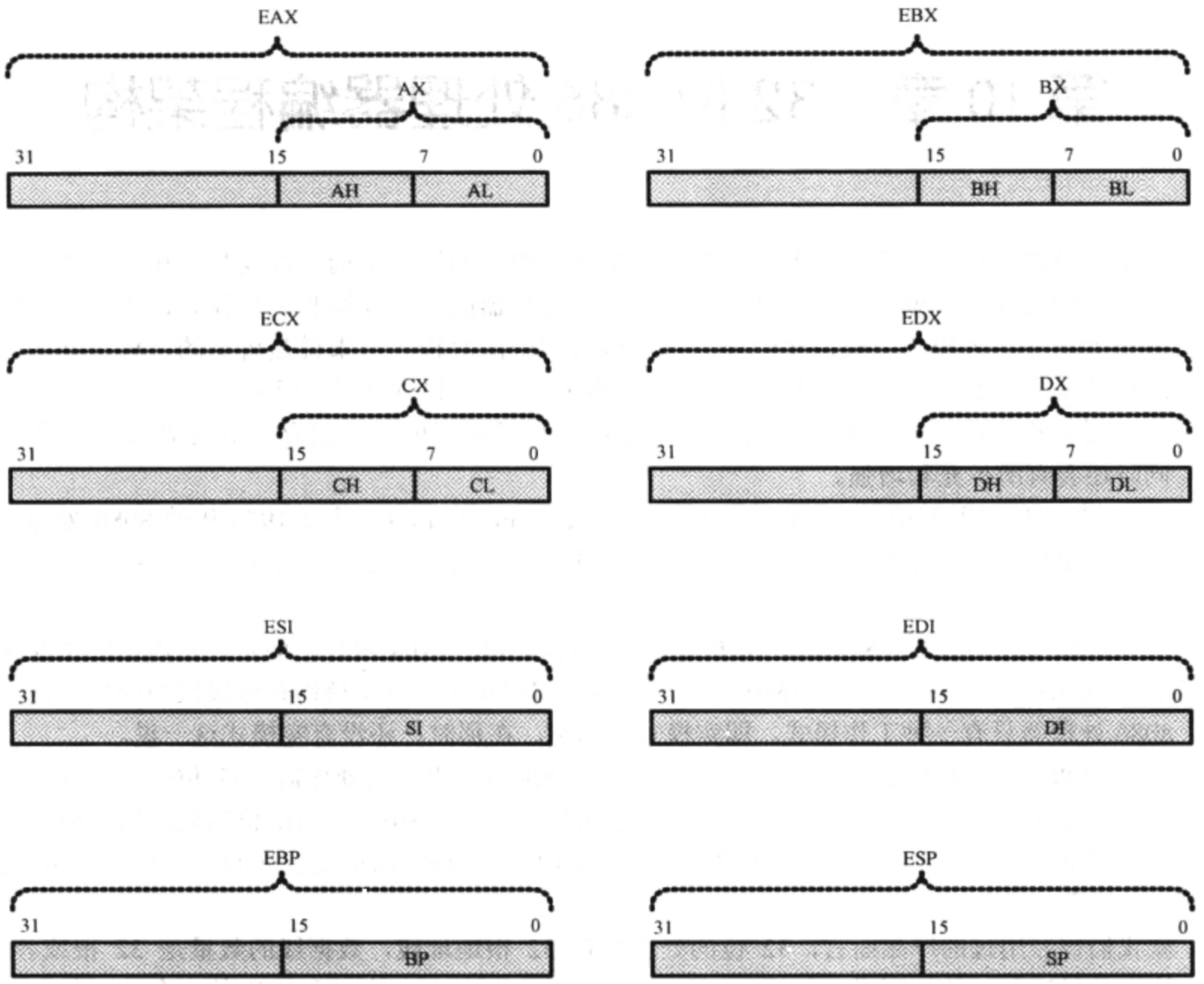

图 10-1　32 位处理器内部的通用寄存器

为了在汇编语言程序中使用经过扩展（Extend）的寄存器，需要给它们命名，它们的名字分别是 EAX、EBX、ECX、EDX、ESI、EDI、ESP 和 EBP。可以在程序中使用这些寄存器，即使是在实模式下：

```
mov eax,0xf0000005
mov ecx,eax
add edx,ecx
```

但是，就像以上指令所示的那样，指令的源操作数和目的操作数必须具有相同的长度，个别特殊用途的指令除外。因此，像这样的搭配是不允许的，在程序编译时，编译器会报告错误：

```
mov eax,cx              ;错误的汇编语言指令
```

如果目的操作数是 32 位寄存器，源操作数是立即数，那么，立即数被视为 32 位的：

```
mov eax,0xf5            ;EAX←0x000000f5
```

32 位通用寄存器的高 16 位是不可独立使用的，但低 16 位保持同 16 位处理器的兼容性。因此，在任何时候它们都可以照往常一样使用：

```
mov ah,0x02
mov al,0x03
add ax,si
```

可以在 32 位处理器上运行 16 位处理器上的软件。但是，它并不是 16 位处理器的简单增强。事实上，32 位处理器有自己的 32 位工作模式，在本书中，32 位模式特指 32 位保护模式。在这种模式下，可以完全、充分地发挥处理器的性能。同时，在这种模式下，处理器可以使用它全部的 32 根地址线，能够访问 4GB 内存。

如图 10-2 所示，在 32 位模式下，为了生成 32 位物理地址，处理器需要使用 32 位的指令指针寄存器。为此，32 位处理器扩展了 IP，使之达到 32 位，即 EIP。当它工作在 16 位模式下时，依然使用 16 位的 IP；工作在 32 位模式下时，使用的是全部的 32 位 EIP。和往常一样，即使是在 32 位模式下，EIP 寄存器也只由处理器内部使用，程序中是无法直接访问的。对 IP 和 EIP 的修改通常是用某些指令隐式进行的，这些指令包括 JMP、CALL、RET 和 IRET 等等。

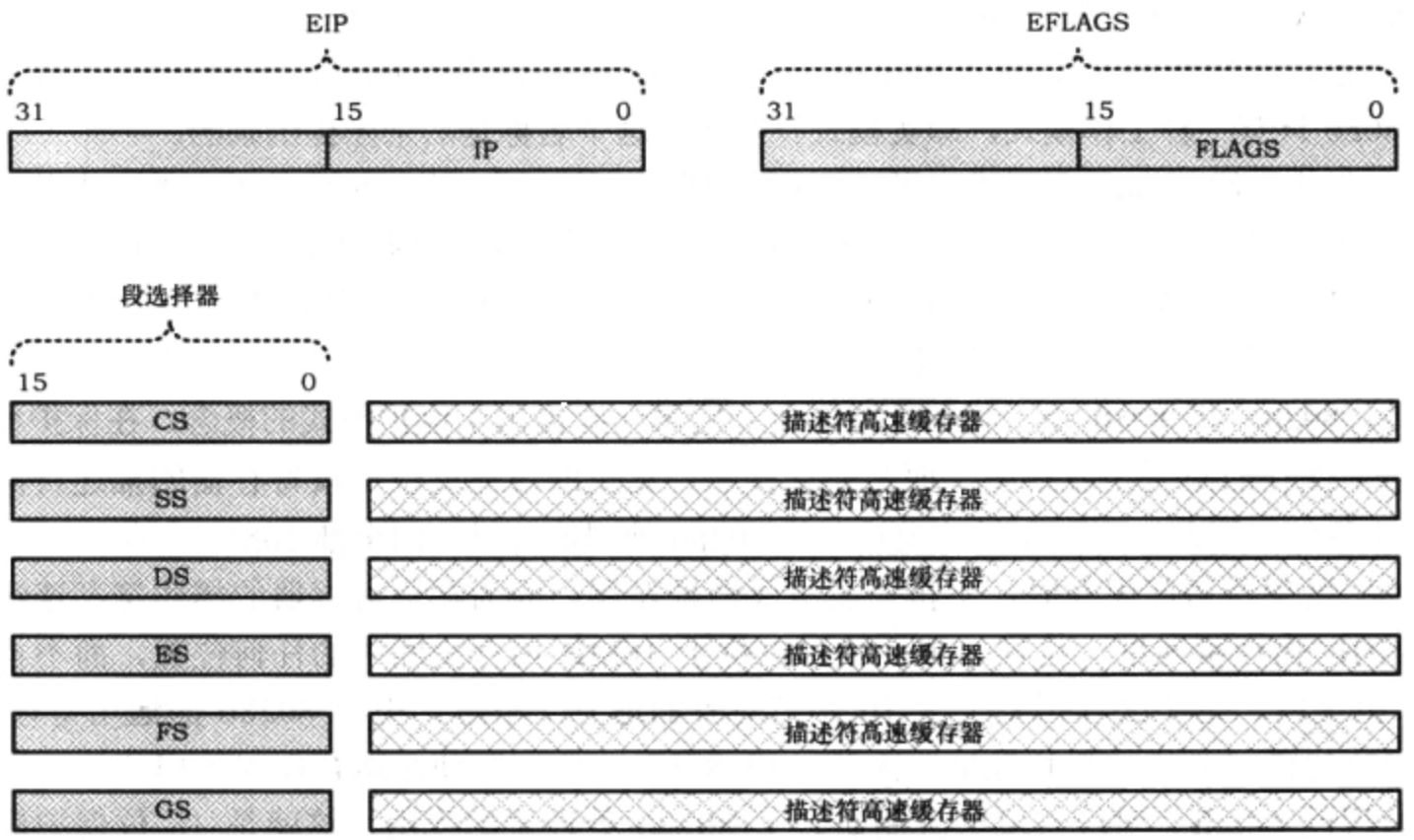

图 10-2 32 位处理器的指令指针、标志和段寄存器

另外，在 16 位处理器中，标志寄存器 FLAGS 是 16 位的，在 32 位处理器中，扩展到了 32 位，低 16 位和原先保持一致。关于 EFLAGS 中的各个标志位，将在后面的章节中逐一介绍。

在 32 位模式下，对内存的访问从理论上来说不再需要分段，因为它有 32 根地址线，可以自由访问任何一个内存位置。但是，IA-32 架构的处理器是基于分段模型的，因此，32 位处理器依然需要以段为单位访问内存，即使它工作在 32 位模式下。

不过，它也提供了一种变通的方案，即，只分一个段，段的基地址是 0x00000000，段的长度（大小）是 4GB。在这种情况下，可以视为不分段，即平坦模型（Flat Mode）。

每个程序都有属于自己的内存空间。在 16 位模式下，一个程序可以自由地访问不属于它的内存位置，甚至可以对那些地方的内容进行修改。这当然是不安全的，也不合法，但却没有任何机制来限制这种行为。在 32 位模式下，处理器要求在加载程序时，先定义该程序所拥有的段，然后允许使用这些段。定义段时，除了基地址（起始地址）外，还附加了段界限、特权级别、类型等属性。当程序访问一个段时，处理器将用固件实施各种检查工作，以防止对内存的违规访问。

如图 10-2 所示，在 32 位模式下，传统的段寄存器，如 CS、SS、DS、ES，保存的不再是 16 位段基地址，而是段的选择子，即，用于选择所要访问的段，因此，严格地说，它的新名字叫做

段选择器。除了段选择器之外，每个段寄存器还包括一个不可见部分，称为描述符高速缓存器，里面有段的基地址和各种访问属性。这部分内容程序不可访问，由处理器自动使用。

有关 32 位模式下的段和段的访问方法，将在后面的章节中予以详述，你在看这段文字的时候，也许有迷迷糊糊的感觉，没关系，这是正常的，到后面你就会感觉豁然开朗了。

最后，32 位处理器增加了两个额外的段寄存器 FS 和 GS。对于某些复杂的程序来说，多出两个段寄存器可能会令它们感到高兴。

10.1.2 基本的工作模式

8086 具有 16 位的段寄存器、指令指针寄存器和通用寄存器（CS、SS、DS、ES、IP、AX、BX、CX、DX、SI、DI、BP、SP），因此，我们称它为 16 位的处理器。尽管它可以访问 1MB 的内存，但是只能分段进行，而且由于只能使用 16 位的段内偏移量，故段的长度最大只能是 64KB。8086 只有一种工作模式，即实模式。当然，这个名称是后来才提出来的。

1982 年的时候，Intel 公司推出了 80286 处理器。这也是一款 16 位的处理器，大部分的寄存器都和 8086 处理器一样。因此，80286 和 8086 一样，因为段寄存器是 16 位的，而且只能使用 16 位的偏移地址，在实模式下只能使用 64KB 的段；尽管它有 24 根地址线，理论上可以访问 2^{24}，即 16MB 的内存,但依然只能分成多个段来进行。

但是，80286 和 8086 不一样的地方在于，它第一次提出了保护模式的概念。在保护模式下，段寄存器中保存的不再是段地址，而是段选择子，真正的段地址位于段寄存器的描述符高速缓存中，是 24 位的。因此，运行在保护模式下的 80286 处理器可以访问全部 16MB 内存。

80286 处理器访问内存时，不再需要将段地址左移，因为在段寄存器的描述符高速缓存器中有 24 位的段物理基地址。这样一来，段可以位于 16MB 内存空间中的任何位置，而不再限于低端 1MB 范围内，也不必非得是位于 16 字节对齐的地方。不过，由于 80286 的通用寄存器是 16 位的，只能提供 16 位的偏移地址，因此，和 8086 一样，即使是运行在保护模式下，段的长度依然不能超过 64KB。对段长度的限制妨碍了 80286 处理器的应用，这就是 16 位保护模式很少为人所知的原因。

实模式等同于 8086 模式，在本书中，实模式和 16 位保护模式统称 16 位模式。在 16 位模式下，数据的大小是 8 位或者 16 位的；控制转移和内存访问时，偏移量也是 16 位的。

1985 年的 80386 处理器是 Intel 公司的第一款 32 位产品，而且获得了极大成功，是后续所有 32 位产品的基础。本书中的绝大多数例子，都可以在 80386 上运行。和 8086/80286 不同，80386 处理器的寄存器是 32 位的，而且拥有 32 根地址线，可以访问 2^{32}，即 4GB 的内存。

80386，以及所有后续的 32 位处理器，都兼容实模式，可以运行实模式下的 8086 程序。而且，在刚加电时，这些处理器都自动处于实模式下，此时，它相当于一个非常快速的 8086 处理器。只有在进行一番设置之后，才能运行在保护模式下。

在保护模式下，所有的 32 位处理器都可以访问多达 4GB 的内存，它们可以工作在分段模型下，每个段的基地址是 32 位的，段内偏移量也是 32 位的，因此，段的长度不受限制。在最典型的情况下，可以将整个 4GB 内存定义成一个段来处理，这就是所谓的平坦模式。在平坦模式下，可以执行 4GB 范围内的控制转移，也可以使用 32 位的偏移量访问任何 4GB 范围内的任何位置。32 位保护模式兼容 80286 的 16 位保护模式。

除了保护模式，32 位处理器还提供虚拟 8086 模式（V86 模式），在这种模式下，IA-32 处理器被模拟成多个 8086 处理器并行工作。V86 模式是保护模式的一种，可以在保护模式下执行多个

8086 程序。传统上，要执行 8086 程序，处理器必须工作在实模式下。在这种情况下，为 32 位保护模式写的程序就不能运行。但是，V86 模式提供了让它们在一起同时运行的条件。

V86 模式曾经很有用，因为在那个时候，8086 程序很多，而 32 位应用程序很少，这个过渡期是必需的。现在，这种工作模式已经基本无用了。

在本书中，32 位模式特指 IA-32 处理器上的 32 位保护模式。不存在所谓的 32 位实模式，实模式的概念实质上就是 8086 模式。

10.1.3 线性地址

为 IA-32 处理器编程，访问内存时，需要在程序中给出段地址和偏移量，因为分段是 IA-32 架构的基本特征之一。传统上，段地址和偏移地址称为逻辑地址，偏移地址叫做有效地址（Effective Address，EA），在指令中给出有效地址的方式叫做寻址方式（Addressing Mode）。比如：

```
inc word [bx+si+0x06]
```

在这里，指令中使用的是基址加变址的方式来寻找最终的操作数。

段的管理是由处理器的段部件负责进行的，段部件将段地址和偏移地址相加，得到访问内存的地址。一般来说，段部件产生的地址就是物理地址。

IA-32 处理器支持多任务。在多任务环境下，任务的创建需要分配内存空间；当任务终止后，还要回收它所占用的内存空间。在分段模型下，内存的分配是不定长的，程序大时，就分配一大块内存；程序小时，就分配一小块。时间长了，内存空间就会碎片化，就有可能出现一种情况：内存空间是有的，但都是小块，无法分配给某个任务。为了解决这个问题，IA-32 处理器支持分页功能，分页功能将物理内存空间划分成逻辑上的页。页的大小是固定的，一般为 4KB，通过使用页，可以简化内存管理。

如图 10-3 所示，当页功能开启时，段部件产生的地址就不再是物理地址了，而是线性地址（Linear Address），线性地址还要经页部件转换后，才是物理地址。

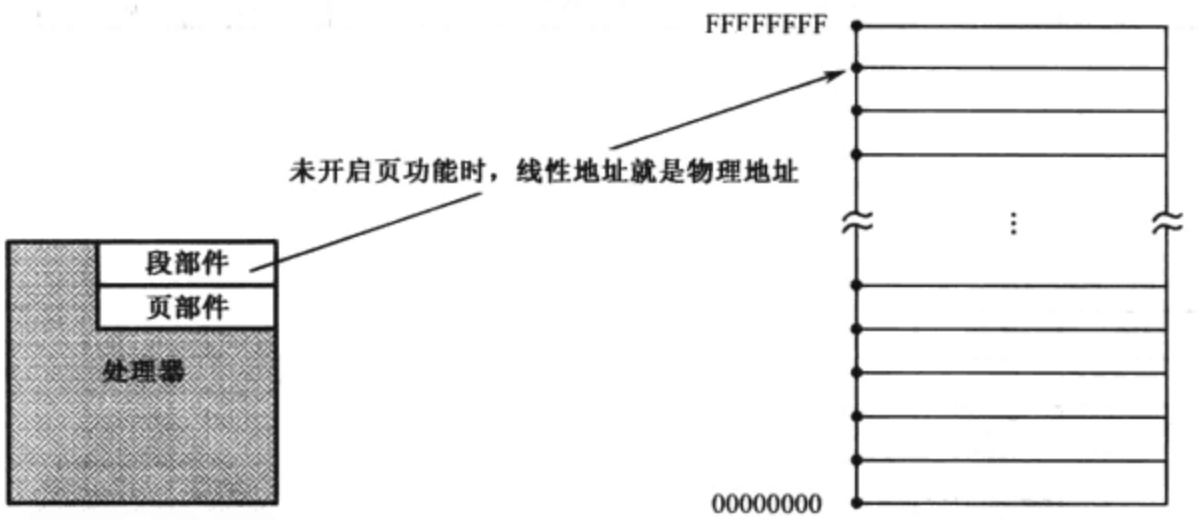

图 10-3 线性地址和线性地址空间

线性地址的概念用来描述任务的地址空间。如图 10-3 所示，IA-32 处理器上的每个任务都拥有 4GB 的虚拟内存空间，这是一段长 4GB 的平坦空间，就像一段平直的线段，因此叫线性地址空间。相应地，由段部件产生的地址，就对应着线性地址空间上的每一个点，这就是线性地址。

IA-32 架构下的任务、分段、分页等内容，是本书的重点，要在后半部分详细论述。

10.2 现代处理器的结构和特点

10.2.1 流水线

处理器的每一次更新换代，都会增加若干新特性，这是很自然的。同时我们也会发现，老软件在新的处理器上跑得更快。这里面的原因很简单，处理器的设计者总是在想尽办法加快指令的执行。

早在 8086 时代，处理器就已经有了指令预取队列。当指令执行时，如果总线是空闲的（没有访问内存的操作），就可以在指令执行的同时预取指令并提前译码，这种做法是有效的，能大大加快程序的执行速度。

处理器可以做很多事情，换言之，能够执行各种不同的指令，完成不同的功能，但这些事情大都不会在一个时钟周期内完成。执行一条指令需要从内存中取指令、译码、访问操作数和结果，并进行移位、加法、减法、乘法以及其他任何需要的操作。

为了提高处理器的执行效率和速度，可以把一条指令的执行过程分解成若干个细小的步骤，并分配给相应的单元来完成。各个单元的执行是独立的、并行的。如此一来，各个步骤的执行在时间上就会重叠起来，这种执行指令的方法就是流水线（Pipe-Line）技术。

比如，一条指令的执行过程分为取指令、译码和执行三个步骤，而且假定每个步骤都要花 1 个时钟周期，那么，如图 10-4 所示，如果采用顺序执行，则执行三条指令就要花 9 个时钟周期，每 3 个时钟周期才能得到一条指令的执行结果；如果采用 3 级流水线，则执行这三条指令只需 5 个时钟周期，每隔一个时钟周期就能得到一条指令的执行结果。

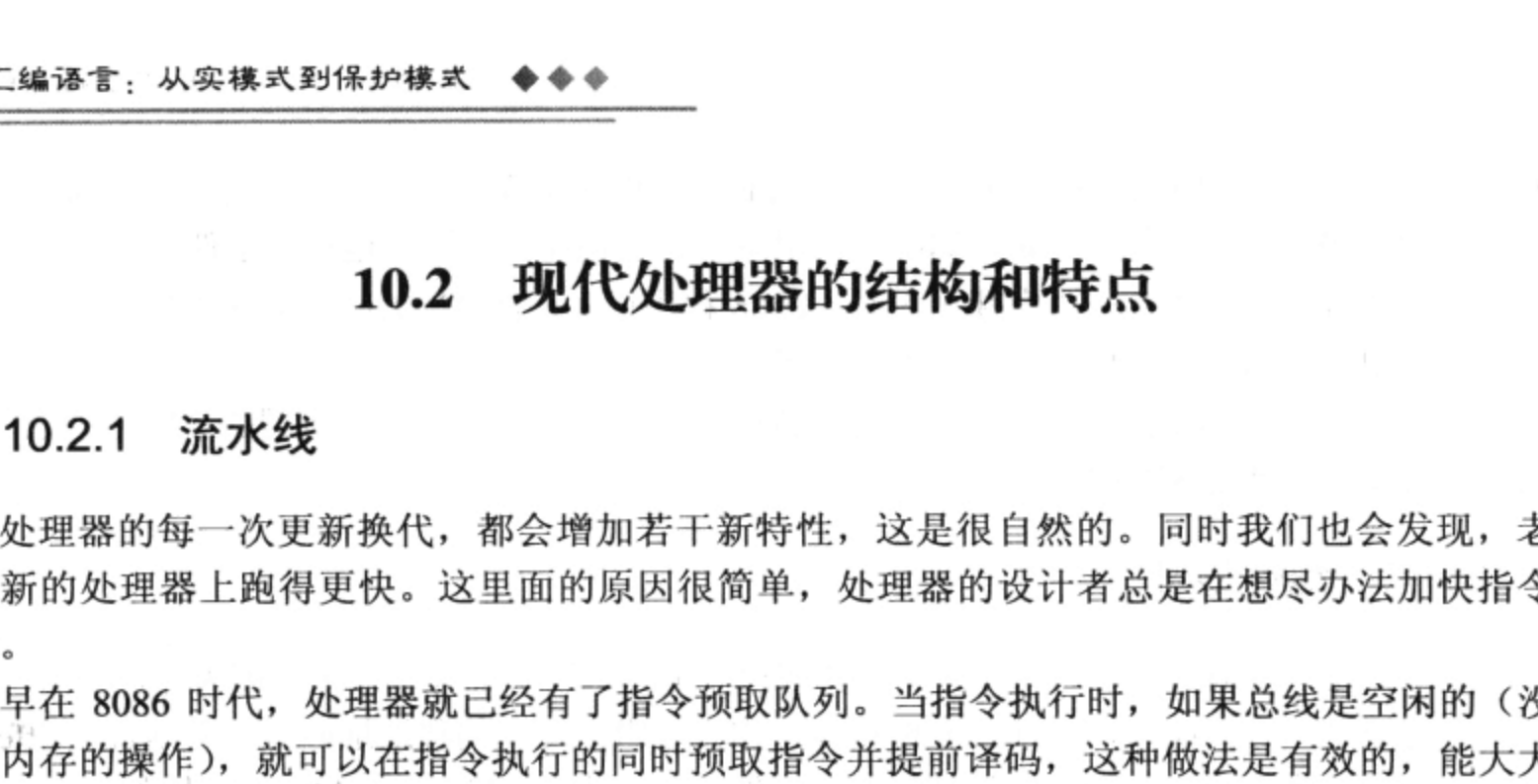

图 10-4 流水线的基本原理

一个简单的流水线其实不过如此，但是，它仍有很大的改进空间。原因很简单，指令的执行过程仍然可以继续细分。一般来说，流水线的效率受执行时间最长的那一级的限制，要缩短各级的执行时间，就必须让每一级的任务减少，与此同时，就需要把一些复杂的任务再进行分解。比如，2000 年之后推出的 Pentium 4 处理器采用了 NetBurst 微结构，它进一步分解指令的执行过

程，采用了 31 级超深流水线。

10.2.2 高速缓存

影响处理器速度的另一个因素是存储器。从处理器内部向外看，它们分别是寄存器、内存和硬盘。当然，现在有的计算机已经用上了固态磁盘。

寄存器的速度是最快的，原因在于它使用了触发器，这是一种利用反馈原理制作的存储电路，在《穿越计算机的迷雾》那本书里，介绍得很清楚。触发器的工作速度是纳秒（ns）级别的，当然也可以用来作为内存的基本单元，即静态存储器（SRAM），缺点是成本太高，价格也不菲。所以，制作内存芯片的材料一般是电容和单个的晶体管，由于电容需要定时刷新，使得它的访问速度变得很慢，通常是几十个纳秒。因此，它也获得了一个恰当的名字：动态存储器（DRAM），我们所用的内存芯片，大部分都是 DRAM。最后，硬盘是机电设备，是机械和电子的混合体，它的速度最慢，通常在毫秒级（ms）。

在这种情况下，因为需要等待内存和硬盘这样的慢速设备，处理器便无法全速运行。为了缓解这一矛盾，高速缓存（Cache）技术应运而生。高速缓存是处理器与内存（DRAM）之间的一个静态存储器，容量较小，但速度可以与处理器匹配。

高速缓存的用处源于程序在运行时所具有的局部性规律。首先，程序常常访问最近刚刚访问过的指令和数据，或者与它们相邻的指令和数据。比如，程序往往是序列化地从内存中取指令执行的，循环操作往往是执行一段固定的指令。当访问数据时，要访问的数据通常都被安排在一起；其次，一旦访问了某个数据，那么，不久之后，它可能会被再次访问。

利用程序运行时的局部性原理，可以把处理器正在访问和即将访问的指令和数据块从内存调入高速缓存中。于是，每当处理器要访问内存时，首先检索高速缓存。如果要访问的内容已经在高速缓存中，那么，很好，可以用极快的速度直接从高速缓存中取得，这称为命中（Hit）；否则，称为不中（miss）。在不中的情况下，处理器在取得需要的内容之前必须重新装载高速缓存，而不只是直接到内存中去取那个内容。高速缓存的装载是以块为单位的，包括那个所需数据的邻近内容。为此，需要额外的时间来等待块从内存载入高速缓存，在该过程中所损失的时间称为不中惩罚（miss penalty）。

高速缓存的复杂性在于，每一款处理器可能都有不同的实现。在一些复杂的处理器内部，会存在多级 Cache，分别应用于各个独立的执行部件。

10.2.3 乱序执行

为了实现流水线技术，需要将指令拆分成更小的可独立执行部分，即拆分成微操作（micro-operations），简写为 μops。

有些指令非常简单，因此只需要一个微操作。如：

```
add eax,ebx
```

再比如：

```
add eax,[mem]
```

可以拆分成两个微操作，一个用于从内存中读取数据并保存到临时寄存器，另一个用于将 EAX 寄存器和临时寄存器中的数值相加。

再举个例子，这条指令：

```
add [mem],eax
```

可以拆分成三个微操作，一个从内存中读数据，一个执行相加的动作，第 3 个用于将相加的结果写回到内存中。

一旦将指令拆分成微操作，处理器就可以在必要的时候乱序执行（Out-Of-Order Execution）程序。考虑以下例子：

```
mov eax,[mem1]
shl eax,5
add eax,[mem2]
mov [mem3],eax
```

这里，指令 add eax,[mem2]可以拆分为两个微操作。如此一来，在执行逻辑左移指令的同时，处理器可以提前从内存中读取 mem2 的内容。典型地，如果数据不在高速缓存中（不中），那么处理器在获取 mem1 的内容之后，会立即开始获取 mem2 的内容，与此同时，shl 指令的执行早就开始了。

将指令拆分成微操作，也可以使得栈的操作更有效率。考虑以下代码片断：

```
push eax
call func
```

这里，push eax 指令可以拆分成两个微操作，即可以表述为以下的等价形式：

```
sub esp,4
mov [esp],eax
```

这就带来了一个好处，即使 EAX 寄存器的内容还没有准备好，微操作 sub esp,4 也可以执行。call 指令执行时需要在当前栈中保存返回地址，在以前，该操作只能等待 push eax 指令执行结束，因为它需要 ESP 的新值。感谢微操作，现在，call 指令在微操作 sub esp,4 执行结束时就可以无延迟地立即开始执行。

10.2.4 寄存器重命名

考虑以下例子：

```
mov eax,[mem1]
shl eax,3
mov [mem2],eax
mov eax,[mem3]
add eax,2
mov [mem4],eax
```

以上代码片断做了两件事，但互不相干：将 mem1 里的内容左移 3 次（乘以 8），并将 mem3 里的内容加 2。如果我们为最后三条指令使用不同的寄存器，那么将更明显地看出这两件事的无关性。并且，事实上，处理器实际上也是这样做的。处理器为最后三条指令使用了另一个不同的临时寄存器，因此，左移（乘法）和加法可以并行地处理。

IA-32 架构的处理器只有 8 个 32 位通用寄存器，但通常都会被我们全部派上用场（甚至还觉得不够）。因此，我们不能奢望在每个计算当中都使用新的寄存器。不过，在处理器内部，却有大量的临时寄存器可用，处理器可以重命名这些寄存器以代表一个逻辑寄存器，比如 EAX。

寄存器重命名以一种完全自动和非常简单的方式工作。每当指令写逻辑寄存器时，处理器就为那个逻辑寄存器分配一个新的临时寄存器。再来看一个例子：

```
mov eax,[mem1]
mov ebx,[mem2]
add ebx,eax
shl eax,3
mov [mem3],eax
mov [mem4],ebx
```

假定现在 mem1 的内容在高速缓存里，可以立即取得，但 mem2 的内容不在高速缓存中。这意味着，左移操作可以在加法之前开始（使用临时寄存器代替 EAX）。为左移的结果使用一个新的临时寄存器，其好处是 EAX 寄存器中仍然是以前的内容，它将一直保持这个值，直到 EBX 寄存器中的内容就绪，然后同它一起做加法运算。如果没有寄存器重命名机制，左移操作将不得不等待从内存中读取 mem2 的内容到 EBX 寄存器以及加法操作完成。

在所有的操作都完成之后，那个代表 EAX 寄存器最终结果的临时寄存器的内容被写入真实的 EAX 寄存器，该处理过程称为引退（Retirement）。

所有通用寄存器，栈指针、标志、浮点寄存器，甚至段寄存器都有可能被重命名。

10.2.5 分支目标预测

流水线并不是百分之百完美的解决方案。实际上，有很多潜在的因素会使得流水线不能达到最佳的效率。一个典型的情况是，如果遇到一条转移指令，则后面那些已经进入流水线的指令就都无效了。换句话说，我们必须清空（Flush）流水线，从要转移到的目标位置处重新取指令放入流水线。

在现代处理器中，流水线操作分为很多步骤，包括取指令、译码、寄存器分配和重命名、微操作排序、执行和引退。指令的流水线处理方式允许处理器同时做很多事情。在一条指令执行时，下一条指令正在获取和译码。

流水线的最大问题是代码中经常存在分支。举个例子来说，一个条件转移允许指令流前往任意两个方向。如果这里只有一个流水线，那么，直到那个分支开始执行，在此之前，处理器将不知道应该用哪个分支填充流水线。流水线越长，处理器在用错误的分支填充流水线时，浪费的时间越多。

随着复杂架构下的流水线变得越来越长，程序分支带来的问题开始变得很大。让处理器的设计者不能接受，毕竟不中处罚的代价越来越高。

为了解决这个问题，在 1996 年的 Pentium Pro 处理器上，引入了分支预测技术（Branch Prediction）。分支预测的核心问题是，转移是发生还是不会发生。换句话说，条件转移指令的条件会不会成立。举个例子来说：

```
jne branch5
```

在这条指令还没有执行的时候，处理器就必须提前预测相等的条件在这条指令执行的时候是否成立。这当然是很困难的，几乎不可能。想想看，如果能够提前知道结果，还执行这些指令干嘛。

但是，从统计学的角度来看，有些事情一旦出现，下一次还会出现的概率较大。一个典型的例子就是循环，比如下面的程序片断：

```
    xor si,si
lops:
```

```
        ......
        cmp si,20
        jnz lops
```

当 jnz 指令第一次执行时，转移一定会发生。那么，处理器就可以预测，下一次它还会转移到标号 lops 处，而不是顺序往下执行。事实上，这个预测通常是很准的。

在处理器内部，有一个小容量的高速缓存器，叫分支目标缓存器（Branch Target Buffer，BTB）。当处理器执行了一条分支语句后，它会在 BTB 中记录当前指令的地址、分支目标的地址，以及本次分支预测的结果。下一次，在那条转移指令实际执行前，处理器会查找 BTB，看有没有最近的转移记录。如果能找到对应的条目，则推测执行和上一次相同的分支，把该分支的指令送入流水线。

当该指令实际执行时，如果预测是失败的，那么，清空流水线，同时刷新 BTB 中的记录。这个代价较大。

10.3 32 位模式的指令系统

10.3.1 32 位处理器的寻址方式

在 16 位处理器上，指令中的操作数可以是 8 位或者 16 位的寄存器、指向 8 位或者 16 位实际操作数的 16 位内存地址，以及 8 位或 16 位的立即数。

如果指令中包含了内存地址操作数，那么，它必然是一个 16 位的段内偏移地址，称为有效地址。通过有效地址，可以间接取得 8 位或者 16 位的实际操作数。指定有效地址可以使用基址寄存器 BX、BP，变址（索引）寄存器 SI 和 DI，同时还可以加上一个 8 位或 16 位的偏移量。比如：

```
mov ax,[bx]
mov ax,[bx+di]
mov al,[bx+si+0x02]
```

以上，第 1 条指令，寄存器 BX 中的内容是指向 16 位实际操作数的 16 位地址；第 2 条指令，寄存器 BX 和 DI 的内容相加，形成指向 16 位实际操作数的 16 位地址；第 3 条指令，寄存器 BX、SI 和 8 位偏移量共同形成指向 8 位实际操作数的 16 位地址。

如图 10-5 所示，这是 16 位处理器的内存寻址方式示意图。从图中可以看出，允许使用基址寄存器 BX 或者 BP，同变址寄存器 SI 或者 DI 结合，再加上 8 位或者 16 位偏移量来寻址内存操作数。

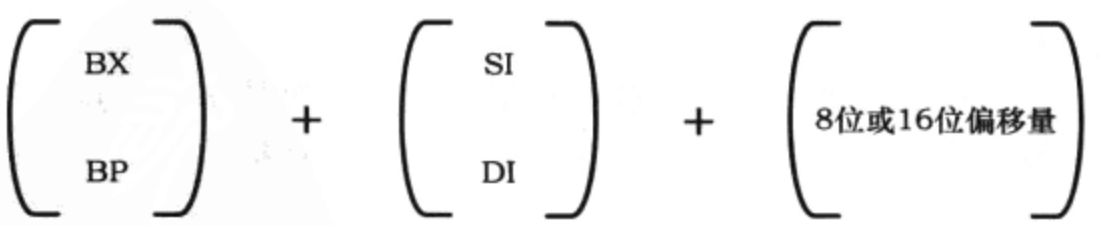

图 10-5 16 位处理器的内存寻址方式

16 位处理器的寻址方式本来就很复杂，当 32 位处理器出现后，寄存器和偏移地址的宽度都扩展了，相应地，要继续扩展原有的寻址方式。但是，原有的 16 位方案已经成型，再进行修补是非常困难的。一个可行的解决方案是，让 16 位指令和 32 位指令共用相同的指令码，但通过不同的

指令前缀，结合处理器当前的运行状态来决定该指令的寻址方式。

比如，当处理器运行在 16 位模式时，如果没有指令前缀 0x66，则认为指令是传统的 16 位寻址方式；若有指令前缀 0x66，则指令是新的 32 位寻址方式。如果处理器当前运行在 32 位模式下且没有指令前缀 0x66，则视为默认的 32 位寻址方式，否则就是传统的 16 位寻址方式。

32 位处理器兼容 16 位处理器的工作模式，可以运行传统的 16 位代码。但是，它有自己独立的 32 位运行模式，而且只有在这种模式下才能发挥最高的运行效率。

在 32 位模式下，默认使用 32 位宽度的寄存器。如：

```
mov eax,ebx
```

如果指令中使用了立即数，那么，该数值默认是 32 位的：

```
mov ecx,0x55        ;ECX←0x00000055
```

还有，如果指令中的操作数是指向内存单元的地址，那么，该地址默认是 32 位的段内偏移地址，或者叫段内偏移量：

```
mov edx,[mem]       ;mem 是一个 32 位的段内偏移地址
```

这就是说，如果指令中包含了内存地址操作数，那么，它必然默认地是一个 32 位的有效地址。通过有效地址，可以间接取得 32 位的实际操作数。如图 10-6 所示，指定有效地址可以使用全部的 32 位通用寄存器作为基址寄存器。同时，还可以再加上一个除 ESP 之外的 32 位通用寄存器作为变址寄存器。变址寄存器还允许乘以 1、2、4 或者 8 作为比例因子。最后，还允许加上一个 8 位或者 32 位的偏移量。

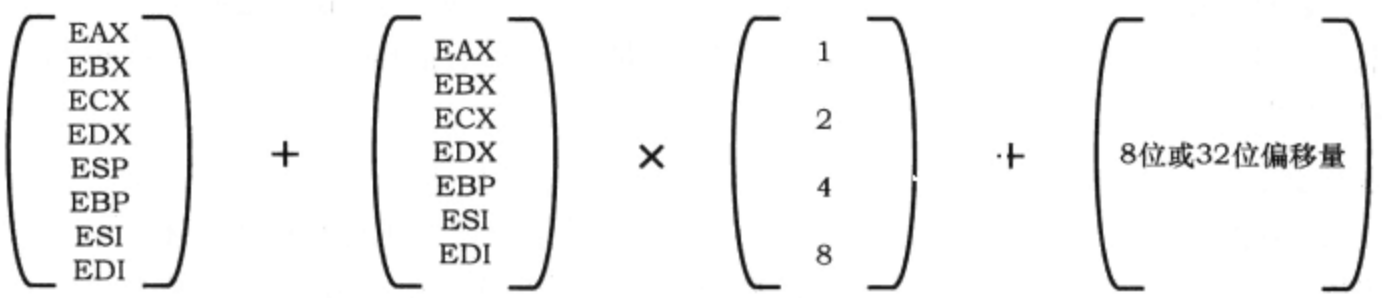

图 10-6 32 位处理器的内存寻址方式

以下是几个例子：

```
add eax,[0x2008]              ;有效地址为 0x00002008
sub eax,[eax+0x08]            ;有效地址是 32 位的
mov ecx,[eax+ebx*8+0x02]      ;有效地址是 32 位的
```

值得说明的是，在 16 位模式下，内存寻址方式的操作数不允许使用栈指针寄存器 SP。因此，象这条指令就是不正确的：

```
mov ax,[sp]
```

但是，在 32 位模式下，允许在内存操作数中使用栈指针寄存器 ESP。因此，下面的指令形式是合法的：

```
mov eax,[esp]
```

10.3.2 操作数大小的指令前缀

Intel 处理器的指令系统比较复杂，这种复杂性来源于两个方面，一是指令的数量较多，二是寻址方式也很多。可以想象，为了组成这些众多的指令，必须有一套同样复杂的指令格式。

如图 10-7 所示，每一条处理器指令都可以拥有前缀，比如重复前缀（REP/REPE/ REPNE）、

段超越前缀（如 ES:）、总线封锁前缀（LOCK）等。前缀是可选的，每个前缀的长度是 1 字节，每条指令可以有 1～4 个前缀，或者不使用前缀。

前缀（如果有的话）的后面是操作码部分，指示执行什么样的操作，比如传送、加法、减法、乘法、除法、移位等。根据指令的不同，操作码的长度是 1～3 字节。同时，操作码还可以用来指示操作的字长，即数据宽度为字节还是字。

操作码之后是操作数类型和寻址方式部分。这部分是可选的，简单的指令不包含这一部分，稍微复杂一点的指令，这一部分只有 1 字节；最复杂的指令，可能有 2 字节。这部分给出了指令的寻址方式，以及寄存器的类型（用的是哪个寄存器）。

指令的最后是立即数和偏移量。如果指令中使用了立即数，那么立即数就在这一部分给出；如果指令使用了带偏移量的寻址方式，如：

```
mov cx,[0x2000]
mov ecx,[eax+ebx*8+0x02]
```

那么，偏移量 0x2000 和 0x02 也在这部分出现。取决于具体的指令，立即数可以是 1、2 或者 4 字节，偏移量部分与此相同。

图 10-7　IA-32 的指令格式

上述的指令编码格式发源于 16 位处理器时代，并在 32 位处理器出现之后做了修改，主要是扩展了数据的宽度，其他都保持不变。毕竟，兼容性是首要考虑的因素。但是，这也带来了一些问题。考虑以下指令：

```
mov dx,[bx+si+0x02]
```

在 16 位指令编码格式中，这种内存单元到寄存器的传送指令使用了操作码 0x8B。如图 10-8（a）所示，在操作码 0x8B 之后是 1 字节的寻址方式和操作数类型部分。位 7 和位 6 的值是 01，表示使用了基地址变址的寻址方式，而且带有 8 位偏移量；位 5～位 3 的值是 010，指示目的操作数为寄存器 DX；位 2～位 0 的值是 000，表示寻址方式为“BX+SI+8 位偏移量”。在该字节之后，是 1 字节的偏移量 0x02。因此，这条指令编译后的机器代码是

```
8B 50 02
```

32 位处理器使用相同的编码格式，但是，寻址方式和寄存器的定义却是另起炉灶的，完全不同于 16 位指令。如图 10-8（b）所示，在 32 位处理器上，位 7 和位 6 的值是 01，表示使用了基址寻址方式，而且带有 8 位偏移量；位 5～位 3 的值是 010，指示目的操作数为寄存器 EDX；位 2～位 0 的值是 000，表示寻址方式为 EAX+8 位偏移量。在该字节之后，是 1 字节的偏移量 0x02。因此，同样的机器指令码，却对应着不同的 32 位指令：

```
mov edx,[eax+0x02]
```

这就是说，相同的机器指令，在 16 位模式下和 32 位模式下的解释和执行效果是不同的。但是，别忘了，32 位处理器可以执行 16 位的程序，包括实模式和 16 位保护模式。为此，在 16 位模式下，处理器把所有指令都看成是 16 位的。举个例子，机器指令码 0x40 在 16 位模式下的含义是

```
inc ax
```

当处理器在 16 位模式下运行时，也可以使用 32 位的寄存器，执行 32 位的运算。为此，必须使用指令前缀 0x66 来临时改变这种默认状态，因为同一个指令码，在 16 位模式下和 32 位模式下具有不同的解释。因此，当处理器在 16 位模式下运行时，机器指令码

```
66 40
```

对应的指令不再是 inc ax，而是

```
inc eax
```

相反地，如果处理器运行在 32 位模式下，那么，处理器认为指令的操作数都是 32 位的，如果你加了前缀，这个前缀就用来指示指令是 16 位的。因此，指令前缀 0x66 具有反转当前默认操作数大小的作用。

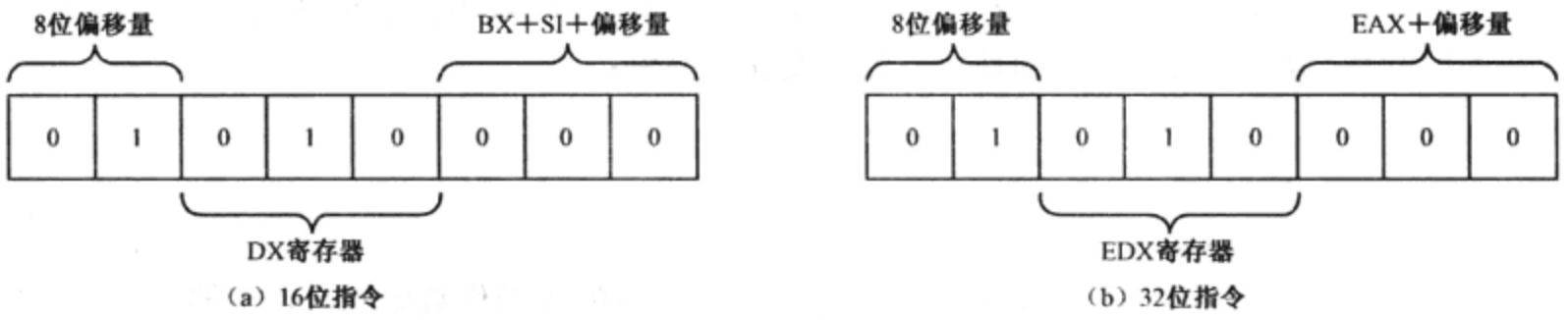

图 10-8 16 位指令和 32 位指令的寻址方式和操作数类型编码对比

在编写程序的时候，就应当考虑到指令的运行环境。为了指明程序的默认运行环境，编译器提供了伪指令 bits，用于指明其后的指令应该被编译成 16 位的，还是 32 位的。比如：

```
bits 16
mov cx,dx      ;89 D1
mov eax,ebx    ;66 89 D8

bits 32
mov cx,dx      ;66 89 D1
mov eax,ebx    ;89 D8
```

注意，bits 16 或者 bits 32 可以放在方括号中，也可以没有方括号。以下两种方式都是允许的：

```
[bits 32]
mov ecx,edx

bits 16
mov ax,bx
```

最后，16 位模式是默认的编译模式。如果没有指定指令的编译模式，则默认是“bits 16”的。

有关寻址方式和指令前缀的话题比较复杂，在后面的章节里，我们将在适当的时候，结合程序和具体的指令进行讲解。

10.3.3 一般指令的扩展

由于 32 位的处理器都拥有 32 位的寄存器和算术逻辑部件，而且同内存芯片之间的数据通路至少是 32 位的，因此，所有以寄存器或者内存单元为操作数的指令都被扩充，以适应 32 位的算术逻辑操作。而且，这些扩展的操作即使是在 16 位模式下（实模式和 16 位保护模式）也是可用的。比如加法指令 ADD，在 32 位处理器上，除了允许 8 位或者 16 位的操作数外，32 位的操作数现在也是可用的：

```
add al,bl
add ax,bx
```

```
add eax,ebx
add dword [ecx],0x0000005f
```

除了双操作数指令，单操作数指令也同样允许 32 位操作数。比如：

```
inc al
inc dword [0x2000]
dec dword [eax*2]
```

我们已经接触过的逻辑移动指令，如 shl、shr 等，目的操作数也扩展至 32 位，但用于指定移动次数的源操作数足够应付 32 位的环境，没有变化。举例：

```
shl eax,1
shl eax,9
shl dword [eax*2+0x08],cl
```

和 16 位时代一样，在 32 位处理器上，逻辑移动指令的源操作数如果是寄存器的话，则依然必须使用 CL。同时，32 位处理器在实际执行时，要先将源操作数（在 CL 寄存器内）同 0x1F 做逻辑与。也就是说，仅保留源操作数的低 5 位，因此，实际移动的次数最大为 31。

在 16 位处理器上，loop 指令的循环次数在寄存器 CX 中。在 32 位处理器上，如果当前的运行模式是 16 位的（bits 16，8086 实模式或者 16 位保护模式），那么，loop 指令执行时，依然使用 CX 寄存器；否则，如果运行在 32 位模式下（bits 32），则使用的是 ECX 寄存器。

在 16 位处理器上，无符号数乘法指令 mul 的格式为

```
mul r/m8        ;AX ← AL×r/m8
mul r/m16       ;DX:AX ← AX×r/m16
```

在 32 位处理器上，除了依然支持上述操作外，还支持以下扩展的格式：

```
mul r/m32       ;EDX:EAX ← EAX×r/m32
```

这样，两个 32 位的数相乘，得到一个 64 位的结果。这里有个例子：

```
mov eax,0x10000
mov ebx,0x20000
mul ebx
```

有符号数乘法指令 imul 与此相同。

相应地，无符号数和有符号数除法也做了 32 位扩展：

```
div r/m32
idiv r/m32
```

在这里，被除数是 64 位的，高 32 位在 EDX 寄存器；低 32 位在 EAX 寄存器。除数是 32 位的，位于 32 位的寄存器，或者存放有 32 位实际操作数的内存地址。指令执行后，32 位的商在 EAX 寄存器，32 位的余数在 EDX 寄存器。

32 位处理器的栈操作指令 push 和 pop 也有所扩展，允许压入双字操作数。特别是，它现在支持立即数压栈操作。立即数压栈操作的指令格式为

```
push imm8       ;操作码为 6A
push imm16      ;操作码为 68
push imm32      ;操作码为 68
```

举个例子可能更清楚一些。比如：

```
push byte 0x55
```

在这里，关键字“byte”仅仅是给编译器用的，告诉它，压入的是字节（毕竟立即数 0x55 可以解释为字 0x0055 或者双字 0x00000055），而不是用来在编译后的机器指令前添加指令前缀。

这条指令的 16 位形式（用 bits 16 编译）和 32 位形式（用 bits 32 编译）是一样的，机器代码都是

```
6A 55
```

但是，当它执行时，就不同了。注意，无论在什么时候，处理器都不会真的压入一字节，要么压入字，要么压入双字。因此，在 16 位模式下，默认的操作数字长是 16，处理器在执行时，将该字节的符号位扩展到高 8 位，然后压入栈，压栈时使用 SP 寄存器，且先将 SP 的内容减去 2。这就是说，实际压入栈中的数值是 0x0055；在 32 位模式下，压入的内容是该字节操作数符号位扩展到高 24 位的结果，即 0x00000055。压栈时使用 ESP 寄存器，且先将 ESP 的内容减去 4。

如果压入的是字操作数，则必须用关键字“word”来修饰。如：

```
push word 0xfffb
```

在 16 位模式下，默认的操作数字长是 16，处理器在执行时，直接压入该字，压栈时使用 SP 寄存器，且先将 SP 的内容减去 2；在 32 位模式下，压入的内容是该操作数符号位扩展到高 16 位的结果，即 0xFFFFFFFB，压栈时使用 ESP 寄存器，且先将 ESP 的内容减去 4。

如果压入的是双字操作数，则必须用关键字“dword”来修饰。如：

```
push dword 0xfb
```

则无论是在 16 位模式下，还是在 32 位模式下，压入的都是 0x000000FB，而且栈指针寄存器（SP 或者 ESP）都先减去 4。

对于实际操作数位于通用寄存器，或者位于内存单元的情况，只能压入字或者双字，指令格式为：

```
push r/m16
push r/m32
```

如果是寄存器，则可以使用 16 位或者 32 位的通用寄存器。比如：

```
push ax
push edx
```

如果被压入的 16 位或者 32 位操作数位于内存单元中，则必须用关键字“word”或者“dword”修饰，以指示操作数的大小：

```
push word [0x2000]
push dword [ecx+esi*2+0x02]
```

无论被压入的数位于寄存器，还是位于内存单元，在 16 位模式下，如果压入的是字操作数，那么先将 SP 的内容减去 2；如果压入的是双字，应当先将 SP 的内容减去 4。在 32 位模式下，如果压入的是字操作数，那么先将 ESP 的内容减去 2；如果压入的是双字，应当先将 ESP 的内容减去 4。

压入段寄存器的操作比较特殊。以下是压入段寄存器的 push 指令格式：

```
push cs          ;机器指令为 0E
push ds          ;机器指令为 1E
push es          ;机器指令为 06
push fs          ;机器指令为 0F A0
push gs          ;机器指令为 0F A8
push ss          ;机器指令为 16
```

在 16 位模式下，先将 SP 的内容减去 2，然后直接压入段寄存器的内容；在 32 位模式下，要

先将段寄存器的内容用零扩展到 32 位，即高 16 位为全零。然后，将 ESP 的内容减去 4，再压入扩展后的 32 位值。

本章习题

1．在编译阶段，如果指定的编译模式是 bits 16，那么，mov bx,16 的机器码为 BB 10 00。相反，mov ebx,16 的机器码为 66 BB 10 00 00 00。试问，如果指定了编译模式 bits 32，这两条指令编译后的机器码又分别是什么？

2．以下程序片断：

```
bits 16
mov bx,16          ;BB 10 00
mul bx             ;F7 E3
```

将生成机器指令序列 BB 10 00 F7 E3。

当处理器在 32 位保护模式下执行这些代码时，会有什么问题？