计算机知识普及系列丛书

80386汇编语言编程技术

李 红　陈 星
刘 权　辛 俭　编写

学苑出版社

计算机知识普及系列丛书

80386 汇编语言编程技术

李 红　陈 星
刘 权　辛 俭　编写

燕卫华　审校

学苑出版社

(京)新登字 151 号

内容提要

本书以 Microsoft 的程序开发工具及 Phar Lap 386|DOS 一扩展编程系统为基础，详细地讲述了 8086 系列(包括 80286、80386)汇编语言的结构和指令，并用实例程序来逐步讲解编程过程中的汇编，链接编辑和联机检测。对想学习汇编语言的人来说，本书是很好的入门途径；对软件开发者来说，本书可帮助您开发出更为有用而又高效的应用程序。

本书可供高等院校有关专业及软件开发人员阅读参考。

欲购本书的用户，请直接与北京海淀 8721 信箱书刊部联系，电话：2562329，邮码：100080。

计算机知识普及系列丛书

80386 汇编语言编程技术

编　　写：李红　陈星　刘权　辛俭
审　　校：燕卫华
责任编辑：甄国宪
出版发行：学苑出版社　　**邮政编码**：100036
社　　址：北京市海淀区万寿路西街 11 号
印　　刷：施园印刷厂
开　　本：787×1092　1/16
印　　张：17　　**字　数**：403 千字
印　　数：1～1000 册
版　　次：1993 年 12 月北京第 1 版第 1 次
ISBN7－5077－0821－7/TP・19
本册定价：13.00 元

学苑版图书印、装错误可随时退换

前　言

由于越来越多的人在家庭或工作中使用个人计算机，所以软件开发者必须努力为他们开发出更有用而又高效的应用程序。较新的IBM个人计算机及其核心都是围绕Intel的80286和80386微处理器建立的。对它们可以用高级语言进行编程，但用比机器语言高一级的汇编语言编程，可使编程员更方便地控制计算机基础资源，并且能得到占用内存空间少而运算快的程序，达到充分发挥微处理器硬件的优点，这是其长处。

而汇编语言的短处是其比高级语言难学，甚至很有经验的汇编语言编程员用汇编语言编程都要比用高级语言花更多的时间。当需要一个高效程序时这些付出是很值得的，但当一个应用程序或系统对速度快慢、内存大小要求不高的时候就完全没有必要用汇编了。

预期目的

本书是为帮助编程员学习80286或80386汇编语言编程而设计的。有其它高级语言编程经验对学习汇编是有帮助的，但不需要以前就了解汇编语言。

覆盖范围

本书介绍了在一个现存的操作系统，如OS/2或DOS下开发运行应用程序所需的软硬设施，系统编程不在介绍范围。

本书前两章介绍了微处理器的构成及编程过程。第三到七章详细说明了应用程序的微处理器指令，第八到十一章则讲解了最重要并且是常用的应用程序中的汇编命令，最后，第十二到第十四章将告诉你如何汇编、链接编辑并检测完整的程序。

本书中讲解编程时用了两个主要的编程系统。在OS/2下运行的系统向您介绍了如何用Microsoft的程序开发工具，包括MASM(宏汇编器)，为链接编辑用的LINK，用于交叉引用列表的CREF，及用于检测的CodeView。你还将学习如何给这些程序赋值，使它们能在DOS操作系统下8086系列的任何微处理器上运行。在DOS系统下运行80386机你能学会使用Phar Lap 386|DOS—扩展编程系统，它会使用户在应用DOS操作系统时了解许多386的高级功能。Phar Lap包括有386|ASM汇编器，386|LINK链接编辑器及MINIBUG形式检测器。

OS/2和DOS都提供了大量的操作系统服务，诸如输入、输出及文件管理。本书不想在此详述所有的功能，能在监视屏上显示信息及从键盘输入指令就已足够了。在此过程中，就能学会如何从每个操作系统中调用服务功能。然后，所有要做的只是在适当操作系统参考手册中查寻用户所要的服务并给出程序中应有的指令。

习惯约定

下面的样本格式语句说明了本书中指令的组成部分：

record-name record *field-name* : *width* 〔=*value*〕 〔,...〕

语句中的斜体部分代表编程者提供的信息，在record语句中，编程者必须提供一个名字及至少一个区段名和其宽度。正规的语句类型都应包含这些。在记录语句中应有单词“record”，方括弧（〔〕）中的条目是可以选择的。在记录语句中，每一区段都有一个值，但不是必须的。省略符（...）代表了重复的条目。定义一个记录会用到许多段描述（段名：长度〔=值〕）。除了方括号和省略符以外，其它标点都应在其适当的位置。在上例中，冒号必须连接每个段名和其宽度。如果包含一个值，则需在其前面加上一个等号。如果还有额外的区段描述，则要在其前面用逗号隔开。一个完整的记录描述为：

date record month：4, day：5, year, year：7=90

在汇编语言源程序中大小写是无关紧要的，多数情况用小写字母，有时为了清晰起见也用大小写混合的方式。在源程序中寄存器用小写字母而在文本中用大写字母来表示，如：AH，AX，ESP都是寄存器名。由于同样的原因，操作码、运算符及符号在文本中都以黑体出现。

此为试读，需要完整PDF请访问：www.ertongbook.com

目　录

第一章　80×××系列的结构特性

Intel的微处理器系列从8086开始，随着这个“家族”的不断发展，个人计算机的水平不断提高，现在，一台微型计算机已经与几年前的大型机具有相同的威力。最早的个人计算机是围绕着诸如Intel8080的8位芯片而设计的，它的寄存器、总线（连接计算机内不同部分的电路）都是基于处理8位数据而设计的，较大的数值要被分割成几个8位字节进行处理。8086开始能够进行16位处理，使微处理器有了长足的发展，Intel公司为此设计了一种复合结构，例如，对于8088，在内部进行16位的处理，但为了与其它芯片及I／O外设接口的方便性，它使用了8位的总线结构，由于8088可以与一般PC机的各种外设兼容，IBM公司选择它作为PC计算机的CPU。

16位的微处理器与8位的结构相比较，有很多优点。16位的寄存器使一次处理的数值最大可达65535（8位的寄存器只为256），16位的总线可允许在一次“move”操作中取得16位的数据，作为8088，由于它的总线仅为8位；它使用双倍时间来完成这一动作。

16位的结构体系还影响了微处理器使用内存的数量。内存中的每个字节（8位）都有一地址与之对应，从零开始一直到内存的最大值，因为内存的地址必须能够装载在寄存器中，所以寄存器的位长就能决定内容的总量。对于16位的寄存器，它所对应的最大内存量为65535（64K），但是，如果使用两个寄存器的组合结构来表示20位的地址，8086／8088的内存处理能力就可以扩展到1M字节，而对80286，同样也是16字节微处理器通过两个寄存器组合表达30字节的地址，可以使处理能力提高到1千兆字节。而对于80386，由于它是32位微处理器，它通过寄存器组合表达的46位的地址，使内存处理可达6.4×10^{13}位。

当然，随着大规模集成技术的发展，16位体系结构中，可在一个芯片上集成更多的半导体元件，大大增加了每秒钟处理指令的数量，也就是说8086／8088不仅提供更强的计算能力和内存寻址能力，而且在运算速度方面也有很大的提高。

8086／8088虽然已有了很大的进步，但在某些“数值计算”方面还有缺陷，它的16位寄存器在面对科学计算、高级图形处理等情况时有些力不从心，并且，它的内部系统没有提供方便的加、减、乘、除指令。8087数字协处理器就是与8086／8088在一起，能提供高级数字处理的芯片，它不仅有更多的寄存器，而且，它的高级运算指令可以处理浮点数运算。在系统中使用了协处理器以后，8086／8088会把数字计算转移给它处理，8087将结果送回主处理器。如果没有8087，有些程序根本无法运行，当然，在以后产生的协处理器功能的系统中，可以运行，但是，速度慢得无法忍受。

8086／8088芯片之所以如此成功，主要原因是它使用在个人计算机上，并且围绕着它开发了大量的软件，由于8086／8088的巨大成功，以后发展一些更强的芯片都要求与8086／8088完全兼容。也就是说，在8086／8088和8087上开发运行的软件，在后续新芯片上也要可以运行。由此产生了80×××家族概念，图1—1中体现了此系列微处理器的发展情况。

80186／80188只用于一些特殊的场合，而从没有被接受为通用处理器，因为人们还在关心8086／8088功能的开发。但是，80286与80386的情况就完全不同了。80286成为

IBMAT计算机及其兼容机的基础，并且，80386也越来越多地运用在个人计算机上。

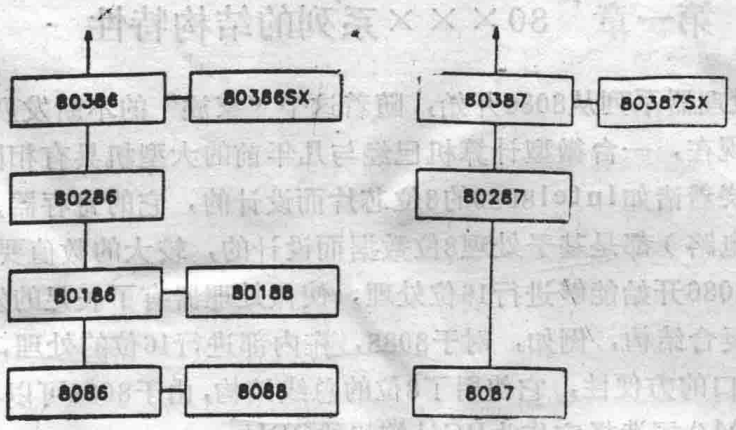

图1—1　80×××家族树枝图

1.1 80286的特点及功能

80286提供了完整的16位体系结构、包括寄存器，总线及地址。芯片的处理速度为8086／8088的三倍，不仅因为它使用了16位的结构，而且其内部的半导体元件数量也是8086／8088的三倍，不仅如此，它的内部还有内存控制单元，并可以将处理指令排序。

8086／8088内部大约有40,000个晶体管，而80286内部的晶体管数量在130,000以上。多出的晶体管可以提供新的功能，但大多数还是用于提高“传统功能”的运行速度。

内存管理单元是用于内存控制的，可提供多任务处理及虚拟空间等功能，由于此单元在CPU内部，而不是通常由外围芯片组成，所以它的访问速度是最快的。

为了更好地利用芯片的高速性，Intel公司设计了一种“管道”结构以提高CPU处理指令的速度。通常，我们在执行一条指令时，首先取得这一指令以及有关的数据，执行指令相应的操作，将操作结果存贮，然后再调用下一条指令。可以看到，总线有一半时间是空闲着的。而CPU的空闲时间更多，而在80286中，当总线执行完当前指令的操作时，在另一个总线管理单元的控制下，自动从内存中读取下一条操作指令，并将其放置在一个on—chip序列中，当CPU执行完某一指令后，它将从on—chip序列中读取下一条指条，而不是等待读取周期完成，这种结构的一个小小的缺点是一个跳转指令或对程序其它部分的调用会“破坏”序列，它必须由CPU直接从内存中读出。但是它的优越性远远地胜于这个小小的缺点，它使每秒中执行的指令数量大大的提高。

1.1.1 内存能力以及虚拟内存

在最早的微处理器时代，标准的内存量为16K字节。当这个数值增长到64K时，人们感到就象发生了奇迹。但是，随着技术的发展，内存量不断地扩展，它所能处理的数据结构以及程序大小也随着不断增加。在很长一段时间，操作系统及应用程序的大小达到了当时个人计算机1M内存的极限。80286有着处理16M物理地址的能力，并且程序并不局限于这个数量，30位的地址长度允许对1千兆的内存量进行访问，内存管理单元提供了产生单位任务使用1千兆虚拟内存的机制，它是通过将存贮内容在物理内存和存储器单元如硬盘之间进行传输而实现的。因此，无论程序装入时占用了多大的空间，它在运行时都好象可以有1千兆的内存空间为之服务，当然，当程序装入时占用的物理空间很小，系

统将不会使用这一能力，因为在物理地址与外设间传输数据是极费时间的。

1.1.2 保护模式

在'286芯片上有两种保护模式：真实模式以及保护性的虚拟地址模式，后者通常被称为保护模式。保护模式的使用可以使芯片的能力得到充分的发挥：例如使程序具有虚拟地址能力以及多任务执行能力（即同时执行多个程序）。

多程序执行能力：在多任务执行环境下，使用者可以在同一时间进行在文字处理器中打印文件、格式化磁盘、通过有线通讯网传输数据、在CAD系统上进行工作等多项工作。多任务执行功能在很长时间里都是大模式系统的标志，因为在286以前的微处理器，在运行速度及内存处理能力方面都无法满足这一功能的需要。

286微处理器之所以可以完成多任务执行功能的秘密在于，使当前执行的各个程序在保护模式下，随机地或连续地访问它们的代码和数据，段的概念是保护柜架的核心。

段：所有在内存中的信息都是以段的方式存贮的，段的最大长度为64K(即可以16位地址进行寻址的量)。一个小程序可能使用一个段就可以满足其代码和数据方面的要求。但是，对于一个较大的程序，如操作系统，就必须使用多个相互分离的代码段和数据段。

对于每个段，都相应的有一名称，它可以表示出段内所装载数据的性质，段中数据量的大小，调用段的上限以及它的属性。这些属性可以告诉我们段是用于执行代码的、只读的、只写的或可读可写的。当某一指令中用到了一个数据或相关指令时，通过段的参数我们就可以知道它是否位于段内，段的属性可以防止对段进行某些非法使用。例如，对于仅用于代码段的执行属性，可以限制对此代码段中代码进行读或写，这样可以保证代码段中的数据不被改变和拷贝————当无法读出时，当然也就无法拷贝。很多代码段都有只读标志，它意味着你只能执行或读出它，但是使用任何方法都无法写入或改变它。对于大多数的数据段都有读写标志，即对它可以进行读操作或改变它的内容。

优先级：对各个段设置的另一个参数就是优先级：从0到3。其中0级的优先级最高，3级的最低，为了避免混乱，本书在谈到优先级时，都是指它的高低，而不是它的代码（0～3）。因此，如果我们说A段比B段有更高的优先级，是指A的优先级号小于B，应用程序一般使用的优先级比较低，而操作系统的例行程序使用的优先级就比较高，一般说来，程序越基本、功能越底层，其优先级越高。

优先级的高低将决定任务执行时指令集中哪条指令被执行先执行与系统核心部分

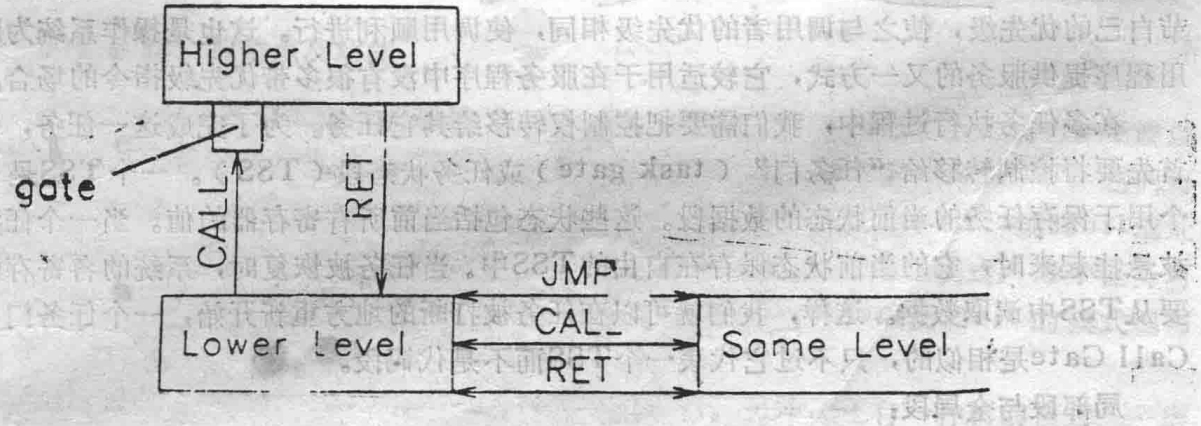

图1—2 不同优先级的段间的转换

此为试读，需要完整PDF请访问：www.ertongbook.com

(如系统的寄存器)相关的指令，一般都以优先级0级执行。因此，这些指令也只能由操作系统来执行，而不能由应用程序完成。在附录B中，列出了指令的优先级，本书将不对这些指令进行讨论，因为它们只能在编写操作系统时使用，本书将不讨论如何编写操作系统。

优先级还控制数据段的访问。当前代码段的优先级就是当前任务的优先级，这样任务可以访问的数据段必须是在此任务中有相同优先级的代码段。代码段将不能访问本任务以外的数据段，无论它的优先级如何，同样，如果数据段包括在本任务内，但优先级不同，代码段也无法访问。

有时，一个代码段会暂时或永久地把执行控制转到另一个代码段上，图1—2中，给出了'286允许的转移方式，'286的保护机制允许一个代码段将控制权转移给同一任务中具有相同优先级的代码段。系统将检查目标段的各个参数以及段的偏移量和大小，如果各种条件都满足，转移工作可以正常进行。

一个代码段也可以把控制权转移给同一任务中优先级更高的代码段。这样，我们的应用程序可以对系统服务（如I／O例行程序）进行访问。只有Call指令才可以完成这一任务，Call指令可以将控制权转移并保留返回调用程序的通道，CAll指令无法直接调用一个优先级更高的任务，而是要依赖于Call Gate。Call Gate用于对具有较高优先级的段进行访问，如果一个具有高优先级的例行程序没有CAll GATE，它将无法被一个低优先级的程序调用。这样，操作系统就可以防止从应用程序发生的非法调用。

Call Gate也有优先级，用于控制谁将访问Call Gate。调用的段必须与Call Gate有相同的优先级。因此，一个优先级为0的例行程序只允许由其它操作系统调用而不允许应用程序调用。如果它的Call Gate为2，它无法被优先级为3的段调用。

一个被调用的高优先级例行程序可以通过ret将控制权返回给较低优先级的程序，ret通过访问保存的返回通道、而不是Call Gate，来找到返回的段址和地址。当ret指令要改变代码段时，它将访问和检查目标段的各个参数，具有较高优先级的数据段的地址将从段址寄存器中移走，这样，返回的段就不会把数据写入这些数据段。

Call指令将无法调用一个低优先级的代码段，它只能调用相同的或具有更高优先级的代码段。而ret指令可以返回到优先级相同或较低的代码段上——而高优先级的代码段不行。因此，通过ret指令对系统进行欺骗，以达到进入高优先级段的尝试是不可能实现的。

但是，有一种称为一致段的特别代码段，可以被任何优先级的段调用。它会自动调节自己的优先级，使之与调用者的优先级相同，使调用顺利进行。这也是操作系统为应用程序提供服务的又一方式，它较适用于在服务程序中没有很多带优先级指令的场合。

在多任务执行过程中，我们需要把控制权转移给其它任务。为了完成这一任务，你首先要将控制转移给“任务门”（task gate）或任务状态段（TSS）。一个TSS是一个用于保存任务的当前状态的数据段。这些状态包括当前所有寄存器的值。当一个任务被悬挂起来时，它的当前状态保存在自由的TSS中。当任务被恢复时，系统的各寄存器要从TSS中读取数据。这样，我们就可以在任务被打断的地方重新开始，一个任务门与Call Gate是相似的，只不过它代表一个TSS而不是代码段。

局部段与全局段：

段又可以被分为局部段和全局段，一个局部段只属于某个任务，不通过任务转换是

无法被其它任务调用的，大多数的应用程序只生成局部段。全局段通常可以被所有任务使用，操作系统的服务通常在全局段中，两个表格记录了段描述符和门：局部描述表（LDT）与全局描述表（GDT），当一个程序将控制制转移给其它段时，处理器会从LDT或GDT中找到正确的段。

1.1.3 真实模式

为了保证与8086形式软件的完全兼容性，'286还提供了另外一种操作模式，我们把它称为真实模式的原因是它只使用真实空间，而不使用虚拟空间。真实模式是在较高速度上对8086／8088功能的模拟，在真实模式中没有虚拟空间以及多任务执行功能。

保护机制在真实模式下是不起作用的，内存空间的地址限制在640K，这对高性能的286是轻而易举的，当IBM AT或兼容机在DOS下运行时，实际上就是处在真实模式中，DOS是针对8088而开发的，所以不使用保护模式。有很多操作系统都能够发挥'286保护模式的作用，如OS／2，但是还没有一个可以将DOS代替。

1.1.4 例外情况以及中断

在继续进行系统正常的处理工作前，对一些例外的系统内部事物必须进行处理，如保护规则冲突或除零错误等，一个"中断信号"也可以中断当前的处理工作，硬件中断一般由外围设备发出，如键盘等，当你按下键盘的某一个键时，它就会向系统发出一个中断信号，并向系统传输所按下键的内容，有些硬件中断，如上面所提到的键盘输入中断，是可屏蔽的。也就是说，如果屏蔽为on，系统就不接收它所发出的中断信号。软件中断一般由程序在运行过程中，根据需要发出。例如，当结果发生溢出错误时，into指令会发出相应的软件中断。

以上谈到的例外情况以及中断处理所使用的方法基本上是相同的，有时我们将它们统称为"中断"，当系统接收到某一中断时，就会使用相应的中断代码进行处理，处理器利用中断代码访问中断字码表(IDT)，其中有处理中断的例行程序的描述符。IDT中主要包括三种字码：中断口陷井以及中断把柄和在完成其它任务时所要使用的任务口。通过它们控制权就可以传给正确的中断处理程序，例行程序的功能取决于操作系统，'286处理器能够接收中断并具备查找IDT的机制，但是IDT的内容以及程序的参数还需要操作系统提供。在一些情况下，中断把柄在处理问题或为硬件服务后，将控制权归还给被中断的任务程序，而在另外一些情况下，中断把柄会产生某种错误信息并终止任务。

1.2 80386 的特点及功能

正如80286以其强大的功能超越了8086一样，80386以其卓越的32位的结构特征超过了80286，为了保证兼容性，它保留了所有的16位寄存器。在不需要兼容的条件下，它们可以扩展为32位，以发挥其强有力的功能，在保护模式下，程序可以选择16bit或32bit地址模式，在16_bit模式下，'386就象一个执行速度很快的'286，在32_bit模式，每个任务可以分配64×10^{12}字节的虚拟空间，这本书将告诉你如何针对16bit以及32bit的模式编写程序。

真实模式依然保留，所以8086程序依然可以运行。另外，还有一种新的模式——虚拟8086模式（V86）可以使用，它允许8086程序在多任务执行环境下运行。也就是说，80386可以执行多个PC形式的软件，（8086的程序由于无法在保护状态下执行，所以在

80286中无法对之进行多任务操作）。

1.3 汇编程序源

汇编语言十分接近于机器内部语言，它可以使用户很好地控制计算机的各种资源——内存、寄存器等等。这是高级语言（如COBOL、BASIC等）所无法比拟的，使用汇编语言编写程序一般比较困难，并且很费时间，但是，在处理某些程序问题时，使用汇编语言是值得的。因为这样不仅可以很好地对系统进行控制，而且程序还会有较高的运行速度，并占用较小的空间。

使用汇编语言开发程序，与用高级语言相似，首先要编写源程序，然后，通过Microsoft公司的宏汇编编译器（MASM）对其进行编译，然后对产生的目标模块进行连接（Link），生成可执行文件（一个EXE文件）。

当在汇编级上工作时，程序员可以使用的资源包括处理器的寄存器，内存空间段以及处理器内部的指令集。

1.3.1 寄存器

寄存器是在处理器内部的用于存贮、计算数据及地址的小区域，由于它位于处理器内部，对其访问不需要利用总线，所以速度极快。例如，使内存中一个值加1需要7个时钟周期（'286）或6个时钟周期（'386），但是对于寄存器加1只需要2个时钟周期。遗憾的是，寄存器的存贮空间是有限的，大多数的数据还是要保存在内存中的。通常，在寄存器中只保存一些当前执行状态需立即使用的数据。

通用寄存器：

大多数的寄存器都是为特定的目的而设计的，一般不能通用。但是，有一组寄存器是为通用程序提供的。如图1—3所示。这些寄存器被称为通用寄存器。图中上部的四个

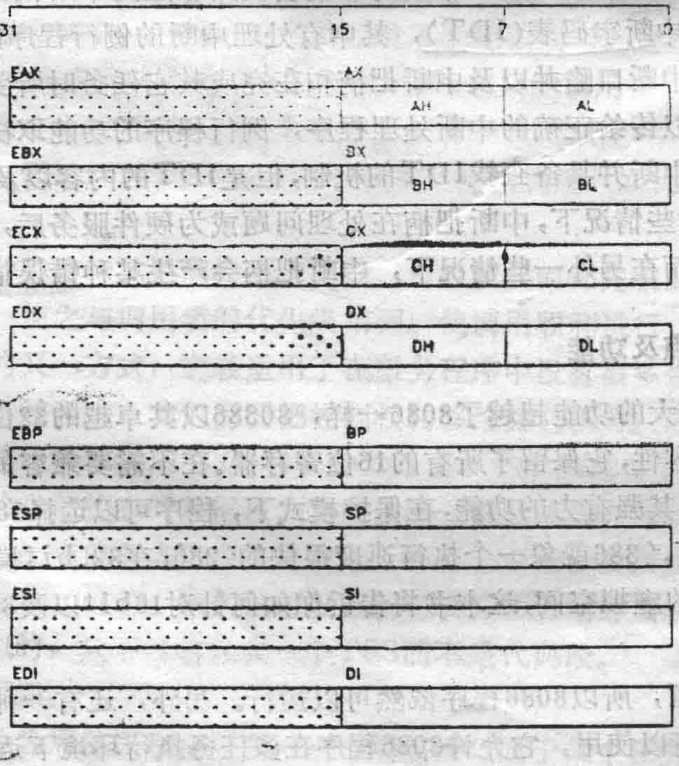

图1—3 一般目的的寄存器

寄存器主要用于数据计算，可以被byte（AL，AH，BL，BH等）、word（AX、BX等）或在'386下的doubleword（EAX，EBX等）访问。在编译时，使用这四个寄存器的机会最大，下面给出的四个寄存器也可以用作数据计算，但是通常把它们用作指针，特别对SP或ESP更是如此。所谓指针，就是在内存中的地址，通过它可以对数据进行访问。

通过图中的标号，用户可以对寄存器进行访问。例如，如果你想使AL寄存器加1，使用下面指令就可以了，

```
inc al
```

我们把16位寄存器的低8位设为一个寄存器，并且命名为xL，相应的高8位称为xH，在'386中，在xH后又进行了16位的扩展，由于'386寄存器的低16访问标号与'286的寄存器相同，这样就保证了'286的程序在'386上也可以顺利执行。换句话说，如果某一程序中使用了32位寄存器（如EAX、EBX等），在'286上，它将无法运行，也就是说，如果你想编写一个使用16位地址的程序，并希望在两种处理器上都可以运行，则你一定不能使用32位的寄存器。

在本书中，当我们使用“EAX”、“EBX”等时，是指此寄存器为32位的，而使用AX、BX等时，是指此寄存器为16位的。因此，当我们说“（E）SP加一”时，我们实际上是说32位的ESP地址加一或16位的SP地址加一。

虽然这八个寄存器可以当作通用寄存器，但是它们各自还有自己的初始定义——特定指令会对它们自动选择使用，由初始定义我们可以给这些寄存器命名：

- (E)AX　用作诸如加、除等代数运算的累加器。
- (E)BX　在'286中可以被用作数据结构的基址寄存器（在'386中，这些寄存器中的任何一个都可以用作基址寄存器，在本章后面我们会解释）。
- (E)CX　用作诸如rep（repeat）与loop等循环指令的计数寄存器。
- （E）DX　用作数据寄存器。在代数运算指令要求使用倍长寄存器时，它被当作累加器的高位。
- （E）BP　用作指向存贮在堆栈中的数据的指针，本章后面会解释。
- （E）SP　作为通用寄存器只能用作堆栈指针，不能用于其它方面。
- （E）SI与（E）DI：在对字符串进行movs或cmps的操作时，这两个寄存器分别用作源索引以及目标索引寄存器。

段寄存器

图1-4中给出了应用程序可以使用的几个系统寄存器，在上部'286有四个，而'386有六个段寄存器，这些16位的寄存器分别指向内存中各段的起点，系统可以通过它们找到任务所需的代码和数据。

CS标定出当前代码段的位置。DS给出主要数据区的位置，在同时使用几个代码段和数据段时，它们是很有用的。ES给出附加数据段的位置。在'386中，用户还可以有另外两个数据段，分别由FS与GS给出位置。SS给出了一个特别的数据段——堆栈段的位置，这个数据段以后会讲到。

系统会决定在一个程序初始化时，在什么地方设置初始段，以及在CS中代码段的初始值。如果程序将控制转移给其它代码段，CS的值会相应变化。在程序开始时，堆

栈的初始值存放在SS中，如果必要的话，程序还要设置自己的数据段（DS～GS）。

在真实模式下，段寄存器给出了内存中的真实物理地址，在保护模式下，段寄存器中存放的是指向某字码表内段字码的指针，字码中包括段的真实基地址。真实基地址可以由基地址加偏移量算出，结果是16bit长还是32bit长将由处理器的地址模式决定。

指令指针：

图1-4中的（E）IP寄存器中存放的是将要执行的指令地址的偏移量。当某一新程序插入时，（E）IP将被设置指向此程序的第一条指令，一般地址偏移量为0，当此地址偏移量上的指令被取走后，（E）IP会前进相应的字节数。例如，在偏移量为0的地方的指令长度为3byte，当此指令被取走后，（E）IP会加3。即由0变为3，如果此时的指令为2-byte长时，(E)IP在下一步会变为5。如果在程序中进行了跳转，（E）IP的

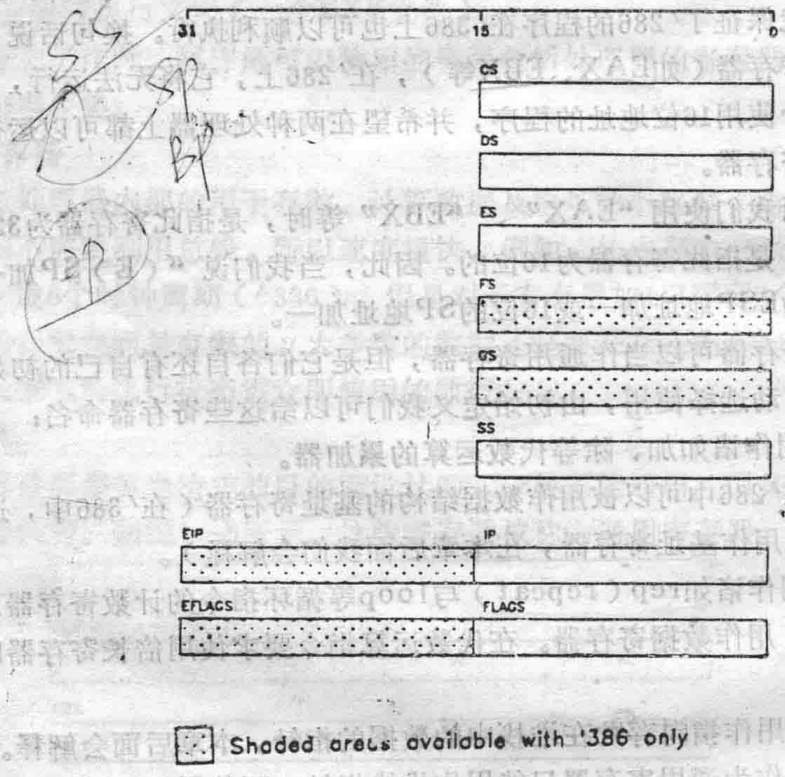

图1—4　一些有用的系统寄存器

值会相应地加、减变化。

系统通过CS与（E）IP的结合，来找到指令在内容中的实际地址。因为CS是直接（真实状态）或间接（保护状态）指向当前代码段基址的指针，CS与（E）IP的结合运用通常写作CS：（E）IP，以确定真实地址。

标志寄存器：

（E）FLAGS寄存器中的各位对系统的状态都有影响。图1—5中给出了标志位的细节内容。位18到位31没有使用。我们将处于低位的标志称为FLAGS，这一点对'286或'386都是相同的。在寄存器扩展部分的标志只对'386有意义。

大多数的标志都是单比特标志，当状态为真时，一个单比特标志被置为1。在状态为非真时清为0。例如：当某计算结果为零时，零标志（IF）被设为1，相反，则清为0。但有输入／输出优先级标志（IOPL）是一个两比特的标志，它可以表示0到3的四个优先

级。

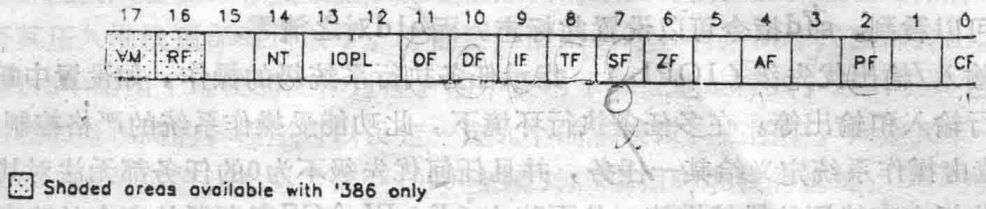

图1—5 （E）FAGS寄存器

有六个标志要受到代数操作及比较操作结果的影响，这些标志通常被称为状态标志，经常在程序中被用作判断标准：

·进位标志（CF）用于指示运算结果是高位比特时是否产生了进位而溢出，使高位数据丢失，或从段外进行了借位操作。当高位产生进位或借位时，此标志被设置为1，相反被清为0。它可以指示在进行运算时是否产生了溢出，符号的值可参考溢出标志。

·奇偶标志（PF）：当结果中比特的数量为奇数时，PF被设置为1，比特的数量为偶数时清为0，所以PF = 1时，说明当前为奇状态，PF = 0时为偶状态。

·附加标志（AF） 当bit 3与bit 4间发生进位和借位时，AF被置为1，此标志在BCD运算中十分的重要。

·零标志（ZF） 如果计算结果为0，ZF会被设置为1，否则为0。

·符号标志（SF） SF的值与结果的最高位值相同，CF为1时表示为负数，CF为零时表示为正数，对于无符号数无意义。

·溢出标志（OF） 指示在次高位与最高位之间是否有进位或借位操作。如果有进位或借位操作，则设置为1，否则清为零。对于有符号数，OF可以指出计算结果是否溢出，对于无符号数，此标志无意义，应该使用进位标志（CF）。

有些指令会影响所有的状态标志，而有些指令则只影响其中的一部分。在附录A中我们可以知道各指令都影响哪些状态标志，如果标志不受当前指令影响，它会保持原值状态不变，但是，也有一些例外，有些指令使标志的状态变得不确定。也就是说，当这些指令执行后，标志的值可能变换，也可能不变，并且变化的结果没有什么意义。这种影响可能比想象的还坏，因为当0标志处于这种状态时，此标志就毫无用处了。附录A中，此类指令用括号标出。

在（E）FLAGS中还剩下系统和控制标志，这些标志的值将影响系统的操作情况，它们与状态标志的区别是，状态标志是随当前状况自动变化的，而系统和控制标志可以由编程者使用软件进行设置或清除，并影响当前进程的情况。

·陷井标志（TF） 控制单步执行，当此标志被设置时，处理器每次只执行一条指令，然后就中断。这种“单步”模式对于调试程序是十分重要的，通常被运用在程序调试器中。

·中断标志（IF） 控制程序是否可以被一个外部中断（可屏蔽的）请求所中断。大多数情况下，它被置为1，程序可以被外部中断所中断，当它清为0时，外部中断无效，此标志只能由处理中断的例行程序控制。

·方向标志（DF） 控制字符串移动的方向，当被清为0时，字符串指令向前移动。即：（E）SI与（E）DI在每步后加1。当它被置为1时，字符串指令向后移动，（E）SI

此为试读，需要完整PDF请访问：www.ertongbook.com