第 1 章 基 础 知 识

在众多的计算机程序设计语言中，汇编语言是唯一一种能够充分利用计算机硬件特性的面向机器的低级语言，它因机器结构的不同而不同。因此，要学会一种汇编语言，就必须先了解该机器的硬件结构、数据类型及表示方法等。本章将从汇编语言程序设计的角度出发，介绍有关的预备知识，如什么是汇编语言、Intel 8086 微处理器中的寄存器组、主存储器的编址方式及物理地址的形成方式、数和符号在计算机中的表示方法等，并以几个源程序为实例介绍汇编源程序的基本结构和格式。这些都是学习以后各章的必备知识。

1.1 计算机语言的发展

自 1946 年第一台电子计算机问世以来，计算机已被广泛地应用于社会生产、生活的各个领域，推动着社会的进步与发展。特别是 Internet 出现后，传统的信息收集、传输及交换方式发生了革命性的改变。

计算机科学的发展依赖于计算机硬件和软件技术的发展，硬件是计算机的“躯体”，软件是计算机的“灵魂”。没有软件，计算机只是一台“裸机”，什么也不能干；有了软件，计算机才有“思想”，才能做相应的事。软件是用计算机语言编写的。计算机语言的发展经历了从机器语言、汇编语言到高级语言的历程。

1.1.1 机器语言

计算机能够直接识别的数据是由二进制数 0 和 1 组成的代码。机器指令就是用二进制代码组成的指令，一条机器指令控制计算机完成一个基本操作。机器指令由操作码和操作数组成，操作码指明做什么操作，操作数指明被操作对象的地址。

使用机器语言编程非常麻烦，特别是在程序有错需要修改时更是如此。例如，将内存中地

址为100H的单元中的内容加2，再存入该地址单元，完成这一操作的二进制指令如下：

1000 0011B
0000 0011B
0110 0100B
0000 0000B
0000 0000B
0000 0010B

其中，第1、2行中的两个8位二进制数是操作码，表示要进行“加”操作，还指明了以何种方式取得两个加数；第3、4行中的两个8位二进制数指出了第一个加数所存放的偏移地址是100H，因为结果也存放在该偏移地址中，所以该加数称为目的操作数；第5、6行中的两个8位二进制数指出了第二个加数（称为源操作数）是02H。

可见，使用机器指令非常麻烦、难懂，不易普及。由于每台计算机的指令系统各不相同，在一台计算机上执行的程序要想在另一台计算机上执行，必须另外编写程序，这造成了重复工作。

机器语言是第一代计算机语言。由于机器语言是针对特定型号计算机的语言，故而运算效率是所有语言中最高的。用机器语言编写的程序是计算机唯一能够直接识别并执行的程序。

1.1.2 汇编语言

为了减轻使用机器语言编程的麻烦，人们进行了一种有益的改进：用一些简洁的英文字母、符号串来替代一个特定的指令的二进制串，如用“ADD”代表加法，“MOV”代表数据传递等，用变量来代替操作数的存放地址，并且与机器指令一一对应。这样一来，人们很容易读懂并理解程序在干什么，纠错及维护也都变得方便了，这种程序设计语言就是汇编语言，是第二代计算机语言。上面的例子可以用一条汇编语言表示：

```
ADD   WORD PTR DS:[100H],2
```

其中，ADD为加指令的助记符，代表了机器指令中的操作码；DS:[100H]表示在当前数据段中，偏移地址为100H的单元中的内容是目的操作数；WORD PTR说明了这个目的操作数是16位二进制数；而源操作数是2H，相加的结果送入目的操作数所在的单元中。

然而计算机是不认识这些汇编语言符号的，用汇编语言编写出的源程序不能直接被计算机识别，必须将它翻译成由机器指令组成的程序后，计算机才能识别并执行。由源程序经过翻译转换生成的机器语言程序称为目标程序。目标程序中的二进制代码（即机器指令）称为目标代码。

用汇编语言编写的程序称为汇编源程序。将汇编源程序翻译成机器语言程序的过程称为“汇编”，这种把汇编源程序翻译成目标程序的语言加工程序称为汇编程序。

汇编语言是一种符号语言，比机器语言容易理解和掌握，其编写的源程序也容易阅读和调试。它与机器语言一一对应，所占用的存储空间、执行速度与机器语言相仿，是一种最接近硬件的计算机语言。

汇编语言同样十分依赖于机器硬件，移植性不好，但效率十分高。针对计算机特定硬件编制的汇编语言程序精炼且质量高、占用空间小、运行速度快，能准确发挥计算机硬件的功能和特长，所以至今仍是一种常用而有力的软件开发工具。在通信、工业、计量、测试、监控等领域，由于输入/输出处理复杂、实时性强，有许多操作都是采用汇编语言来实现的。在多

媒体技术、网络通信技术、计算机安全、病毒防治等方面，也由于需要与硬件设备打交道而使用汇编语言。此外，实用中还有许多用汇编语言编写的系统程序和应用程序需要阅读、分析和修改，因此，几乎每一个计算机系统，都把汇编语言作为系统的基本配置，汇编程序成为系统软件的核心成分之一。汇编语言程序设计的方法与技巧又是进行其他高级语言程序设计的基础，是学习后继课程的基础。对于从事计算机研制和应用的广大科技工作者来说，熟练且牢固地掌握汇编语言程序设计技术是必要的，一个未经过汇编语言程序设计严格训练的计算机专业人员是不能很好地胜任其工作的。

1.1.3 高级语言

汇编语言虽然较机器语言直观，但仍然是面向硬件的语言，并且烦琐难懂，于是人们研究出了高级程序设计语言。高级程序设计语言接近于人类自然语言的语法习惯，与计算机硬件无关，易被用户掌握和使用。高级语言的发展也经历了从早期语言到结构化程序设计语言，从面向过程到非过程化程序语言的过程。相应地，软件的开发也由最初的个体手工作坊式的封闭式生产，发展为产业化、流水线式的工业化生产。目前广泛应用的高级语言有多种，如 BASIC、FORTRAN、PASCAL、C、C++等。

高级语言的下一个发展方向是面向应用，也就是说，只需要告诉程序你要干什么，程序就能自动生成算法，自动进行处理，这就是非过程化的程序语言。

1.2 Intel 8086 微处理器简介

80x86 是美国 Intel 公司生产的微处理器系列。该公司自 1969 年开始设计 4 位的 4004 芯片，1973 年开发出 8 位的 8080 芯片，1977 年推出 16 位的 8086 微处理器芯片，由此 Intel 公司开始了 80x86 微处理器系列的生产历史。

微型计算机主要由中央处理器(CPU)、主存储器、外部设备及互连设备组成，总线(数据总线、地址总线和控制总线)在各部件之间提供通信，其系统结构如图 1-1 所示。其中，CPU 是核心部分，主要由微处理器组成；主存储器用来保存程序和数据。

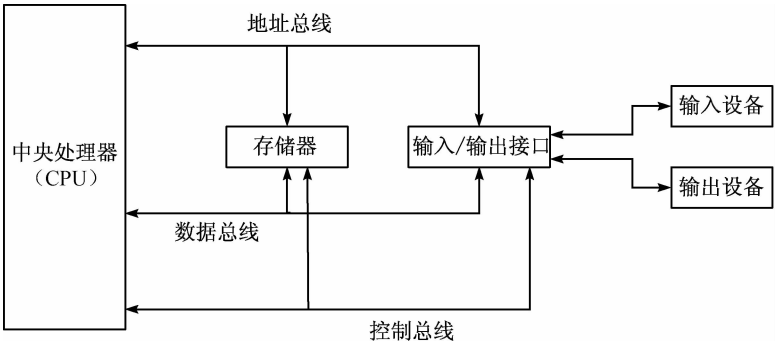


图 1-1 微型计算机的系统结构图

Intel 8086/8088 微处理器按功能可分为两大部分：执行部件（execution unit，EU）和总线接口部件（bus interface unit，BIU），其内部结构如图 1-2 所示。

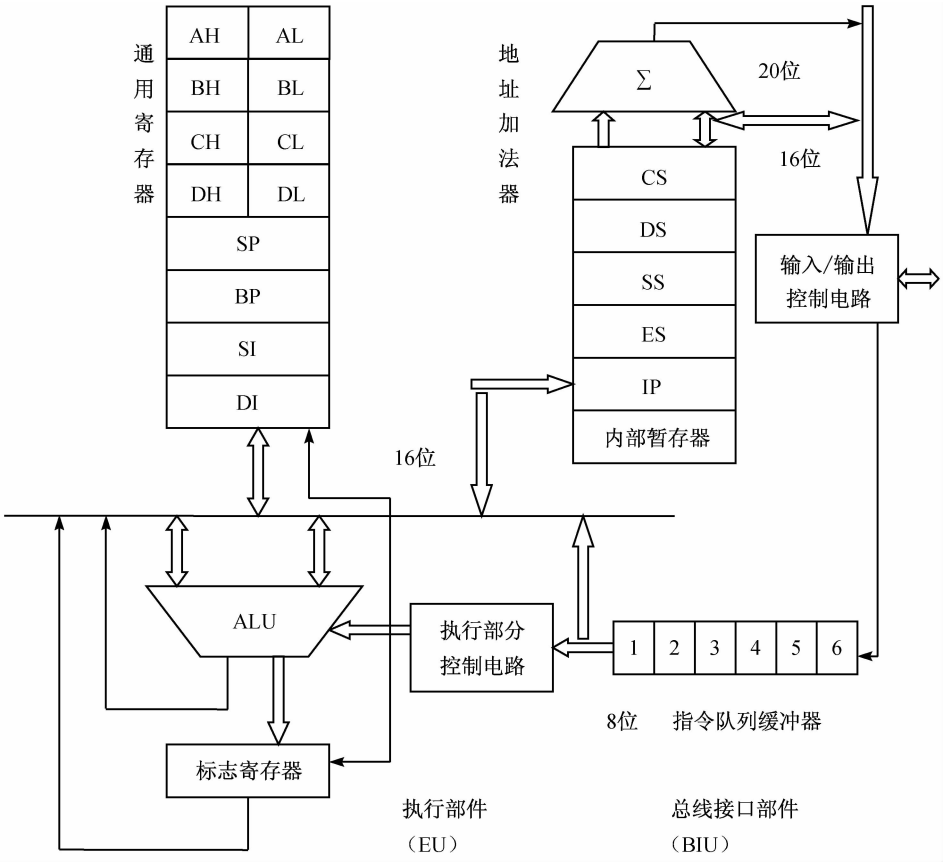


图 1-2　8086/8088 微处理器内部结构示意图

1.2.1　Intel 8086 微处理器结构

计算机主要由运算器、控制器、存储器和输入/输出设备构成。20 世纪 70 年代初期，由于大规模集成电路技术的发展，已经开始把运算器和控制器集成在一个芯片上，构成中央处理器（central processing unit，CPU），80x86 就是这样一组微处理器系列。80x86 微处理器主要由执行部件（EU）和总线接口部件（BIU）构成。

（1）执行部件：由算术逻辑单元（ALU）、通用寄存器组、状态寄存器及操作控制器电路组成。

（2）总线接口部件：由专用寄存器、指令队列缓冲器、地址加法器等功能部件组成，形成对外总线，与存储器、I/O 接口电路进行数据传输。

EU 与 BIU 可独立工作，BIU 在保证 EU 与片外传送操作数前提下，可进行指令预取，可与 EU 重叠操作。

1）执行部件

执行部件负责指令的执行，并进行算术逻辑运算等。EU 从 BIU 中的指令队列中取得

指令，当指令要求将数据放在寄存器或输出到外部设备，或者要从寄存器或外部设备读取数据时，EU 就向 BIU 发出请求，BIU 根据 EU 发来的请求完成这些操作。执行部件中含有 8 个 16 位的寄存器，这些寄存器属于 CPU 的专用寄存器，按其用途可将它们分成两组：数据寄存器组、指针和变址寄存器组。

(1)数据寄存器组(AX、BX、CX、DX)。数据寄存器主要用来保存操作数或运算结果等信息。它们的存在减少了为存取操作数所需访问总线和主存的时间，加快了机器的运行速度。

(2)指针和变址寄存器组(SI、DI、SP、BP)。这 4 个寄存器均为 16 位寄存器，它们一般用来存放操作数的偏移地址，用做指示器或变址寄存器。

2)总线接口部件

总线接口部件包括一组段寄存器、一个指令指示器、指令队列(8086 长 6 字节，8088 长 4 字节)、地址加法器和总线控制器等。BIU 根据执行部件 EU 的请求，完成 CPU 与存储器或 I/O 设备之间的数据传送。在 EU 执行指令的过程中，BIU 根据需要从存储器中预先取一些指令，保存到指令队列中。如果 EU 执行一条转移指令，那么存放在指令队列中的预先取得的指令就不再有用，BIU 会根据 EU 的指示从新的地址重新开始取指令。

EU 所提供的存储器地址是 16 位的，而 8086 访问 1 MB(2^{20})存储空间需要 20 位地址，为了形成这 20 位地址，在 BIU 中设立了 4 个段寄存器(CS、DS、ES、SS)。CPU 在当前每一时刻可以直接访问 4 个存储段：一个代码段、一个数据段、一个附加段、一个堆栈段。其中，每个段最大可达 64 KB。

在 BIU 中，有一个很重要的寄存器——指令指示器(IP)，它总是保存着下一次将从主存中取出指令的偏移地址，其值为该指令到所在段段首址的字节距离。

(1)段寄存器。用于存放段地址。

①代码段(CS)寄存器。存放当前被执行的程序的段地址。

②数据段(DS)寄存器。存放当前被执行的程序所用数据的段地址。

③堆栈段(SS)寄存器。存放当前被执行的程序所用堆栈操作数的段地址。

④附加段(ES)寄存器。存放当前被执行的程序附加数据段的段地址。

(2)指令指针(IP)寄存器。存放将要执行的下一条指令的偏移量，与 CS 联合形成下一条指令的物理地址。

(3)地址加法器(20 位)。按以下算式计算存储单元的物理地址：

$$\text{物理地址} = \text{段地址} * 10\text{H} + \text{偏移地址}$$

(4)指令队列缓冲器(6 B)。在 EU 不使用总线时，BIU 从存储器中读取指令填充指令队列缓冲器。

(5)输入/输出控制电路。它是处理器与外部总线的接口，根据地址码经数据总线进行操作数或指令代码的传输。

1.2.2 Intel 8086 微处理器寄存器组

8086 微处理器内部共有 14 个 16 位可编程寄存器，按大的功能分为 4 组，即数据寄存器组、指针及变址寄存器组、段寄存器组和控制寄存器组，如图 1-3 所示。

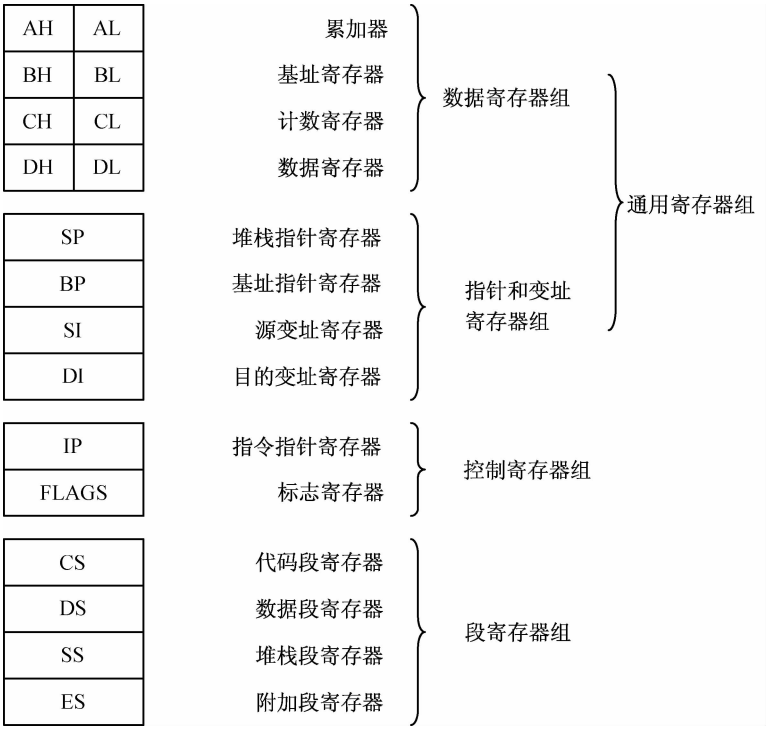


图 1-3　8086 微处理器寄存器组

1)数据寄存器组

数据寄存器组包括 4 个 16 位的寄存器 AX、BX、CX、DX,它们和指针和变址寄存器组构成通用寄存器组。它们既可以作为 16 位寄存器使用,也可以分为两个 8 位寄存器使用,即高 8 位寄存器 AH、BH、CH、DH 和低 8 位寄存器 AL、BL、CL、DL。这些寄存器既可以作为算术逻辑运算的源操作数,向 ALU 提供参与运算的原始数据,也可以作为目标操作数,保存运算的中间结果或最后结果。在有些指令中,这些寄存器具有特定的用途,例如,AX 作为累加器,BX 作为基址寄存器,CX 作为计数寄存器,DX 作为数据寄存器。

2)指针和变址寄存器组

指针和变址寄存器组(pointer and index registers)分为两个指针寄存器——堆栈指针(stack pointer,SP)寄存器、基址指针(base pointer,BP)寄存器和两个变址寄存器——源变址(source index,SI)寄存器、目的变址(destination index,DI)寄存器。这组寄存器通常用来存放存储器单元的 16 位偏移地址(即相对于段起始地址的距离,简称偏移地址)。

(1)指针寄存器。在 8086 CPU 内存中有一个按照“先进后出”原则进行数据操作的区域,称为堆栈。CPU 对堆栈的操作有两种:压入(PUSH)和弹出(POP)。在进行堆栈操作的过程中,SP 用来指示堆栈栈顶的偏移地址,称为堆栈指针;而 BP 用来存放位于堆栈段中的一个数据区的“基址”的偏移量,称为基址指针。

(2)变址寄存器。变址寄存器用来存放当前数据所在存储单元的偏移地址。在串操作指令中,SI 用来存放源操作数地址的偏移量,DI 用来存放目标操作数地址的偏移量。

3)段寄存器组

在8086 CPU中有4个16位的段寄存器(segment registers):代码段(code segment,CS)寄存器、数据段(data segment,DS)寄存器、堆栈段(stack segment,SS)寄存器和附加段(extra segment,ES)寄存器。这些寄存器指明了一个特定的现行段,用来存放各段的段基址。当用户用指令设定了它们的初值后,实际上已经确定了一个64 KB的存储区段。其中,CS用来存放当前使用的代码段的段基址,用户编写的程序必须存放在代码段中,CPU将会依次从代码段中取出指令代码并执行。DS用来存放当前使用的数据段的段基址,程序运行所需的原始数据以及运算的结果应存放在数据段中。ES用来存放当前使用的附加段的段基址,它通常也用来存放数据,但在进行数据串操作时,用来存放目标数据串(此时DS用来存放源数据串)。SS用来存放当前使用的堆栈段的段基址,所有堆栈操作的数据均保存在这个段中。

4)指令指针寄存器

指令指针(instruction pointer, IP)寄存器为16位寄存器,其内容总是指向BIU将要取的下一条指令代码的16位偏移地址。当取出1字节指令代码后,IP自动加1并指向下一条指令代码的偏移地址。它的内容是由BIU来修改的,用户不能通过指令预置或修改IP的内容,但有些指令的执行可以修改它的内容,也可以将其内容压入堆栈或由堆栈中弹出。

5)标志寄存器

8086 CPU中有一个16位的标志寄存器(flag register,FR),但只使用了9位。其中6位为状态标志位,用来反映算术运算或逻辑运算结果的状态;另外3位为控制位,用来控制CPU的操作。8086 CPU标志寄存器各位的定义如图1-4所示。

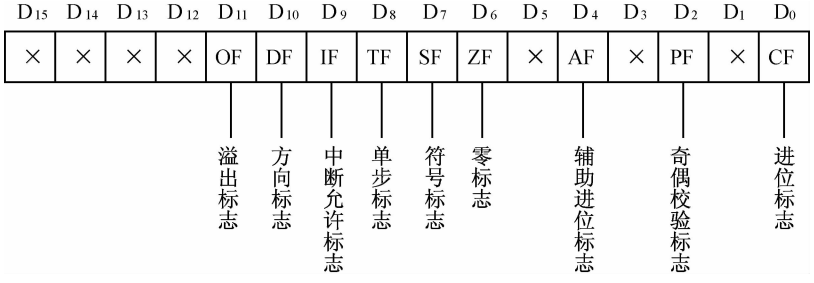


图1-4 8086 CPU标志寄存器中各位的定义

(1)进位标志(carry flag, CF):状态标志位,表示本次加法或减法运算中最高位(D_7或D_{15})产生进位或借位的情况。(CF)=1表示有进位,(CF)=0表示无进位(减法时,表示借位的情况)。

(2)奇偶校验标志(parity flag, PF):表示本次运算结果中包含"1"的个数。(PF)=1表示有偶数个"1",(PF)=0表示有奇数个"1"。

(3)辅助进位标志(auxiliary carry flag, AF):表示加法或减法运算结果中D_3位向D_4位产生进位或借位的情况。(AF)=1表示有进位,(AF)=0表示无进位(减法时,表示借位的情况)。

(4)零标志(zero flag, ZF):表示当前的运算结果是否为零。(ZF)=1 表示运算结果为零,(ZF)=0 表示运算结果不为零。

(5)符号标志(sign flag, SF):表示运算结果的正、负情况。(SF)=1 表示运算结果为负,(SF)=0 表示运算结果为正。

(6)溢出标志(overflow flag, OF):表示运算过程中产生溢出的情况。(OF)=1 表示当前正在进行的补码运算有溢出,(OF)=0 表示无溢出。

(7)方向标志(direction flag, DF):控制标志位,用来设定和控制字符串操作指令的步进方向。(DF)=1 时,串操作过程中的地址会自动递减 1;(DF)=0 时,地址自动递增 1。

(8)中断允许标志(interrupt enable flag, IF):用来控制可屏蔽中断的标志位。(IF)=1 时,开中断,CPU 可以接收可屏蔽中断请求;(IF)=0 时,关中断,CPU 不能接收可屏蔽中断请求。

(9)单步标志(trap flag, TF):用来控制 CPU 进入单步工作方式。(TF)=1 时,8086 CPU 处于单步工作方式,每执行完一条指令就自动产生一次内部中断;(TF)=0 时,CPU 不能以单步方式工作。CPU 的单步工作方式为程序调试提供了一种重要的方法。

1.3 存 储 器

1.3.1 存储单元的地址和内容

存储单元有字节单元和字单元两种。其中,一个字节单元只有一个字节,而一个字单元包含几个字节单元,具体取决于机器字长。字长为 16 的机器,字单元包含连续的两个字节单元;字长为 32 的机器,字单元包含 4 个连续的字节单元;其他字长的机器依此类推。

存储单元的编号称为存储单元的地址,一个存储单元中存放的信息称为该存储单元的内容,同一个地址既可看做字节单元的地址,又可看做字单元的地址,这要根据使用情况确定。可以看出,字单元的地址可以是偶数,也可以是奇数。但是,在机器里,访问存储器(要求取数或存数)都是以字为单位进行的,也就是说,机器是以偶地址访问存储器的。这样,对于奇地址的字单元,取一个字需要访问两次存储器,当然这样做要花费较多的时间。

如上所述,如果用 X 表示某存储单元的地址,则 X 单元的内容可以表示为(X)。假如 X 单元中存放着 Y,而 Y 又是一个地址,则可用(Y)=((X))来表示 Y 单元的内容。存储器有这样的特性,所以它的内容是取之不尽的。也就是说,从某个单元取出其内容后,该单元仍然保存着原来的内容不变,可以重复取出,存入新的信息后,原来保存的内容就会自动丢失了。

1.3.2 堆栈

堆栈是一种线性表(就是像一条线一样存储的序列,见图 1-5),它只允许在一端进行插

入和删除。允许插入和删除的一端称为栈顶(top),另一端称为栈底(buttom)。堆栈的插入称为入栈,删除称为出栈。根据堆栈的特点可知,最先入栈的总是最后出栈的,最后入栈的总是最先出栈的,即“先进后出”。堆栈中的数据也称为元素或栈顶。

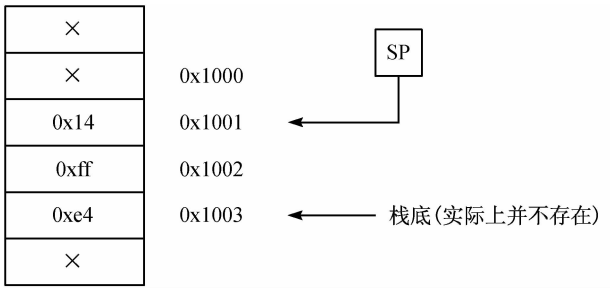


图 1-5　堆栈

堆栈段(SS)寄存器存储的是堆栈段的基地址,堆栈指针(SP)寄存器则存储堆栈段的栈顶。

注意:堆栈是向低地址生长的。

为什么说栈底并不存在? 假设现在 SP 指向栈底 0x1003,如果再进行一次出栈操作(弹出 1 字节),SP 就会指向 0x1004,即栈底并不存在。

1)堆栈操作

向堆栈中存入数据必须采用专门的指令进行,而且只能是字操作。堆栈操作使用了两个指令:PUSH(入栈)和 POP(出栈)。

(1)PUSH　操作数。将 SP 指向前一个字单元(或字节单元),并把操作数放在 SP 所指的字单元(或字节单元)中。

(2)POP　操作数。将 SP 所指的字单元(或字节单元)中的数据放到操作数中,这时的操作数不能是立即数,只能是寄存器或内存地址,然后让 SP 指向下一个字单元(或字节单元)。

当然,也可以使用 mov 指令把 SP 所指向的数据挪到目的操作数中,而不改变 SP 的值。堆栈操作如图 1-6 所示。

现在假设堆栈为图 1-6(b)所示的情况,SP 指向 0x1004,再假设 0x1002 中存储的数据为 0x5555。

问题:执行 mov ax,[SP+2]之后,ax 中的数据是什么? 执行 mov ax,[SP−2]之后,ax 中的数据是什么?

答案:SP+2 指向 0x1006(0x1004+0x2),所以 ax 中的数据是 0x1234;同理,SP−2 指向 0x1002,所以 ax 中的数据为 0x5555。

2)堆栈的作用

堆栈的作用主要有如下 3 个:

(1)调用函数。

(2)暂时存储数据。

(3)保护寄存器数据。

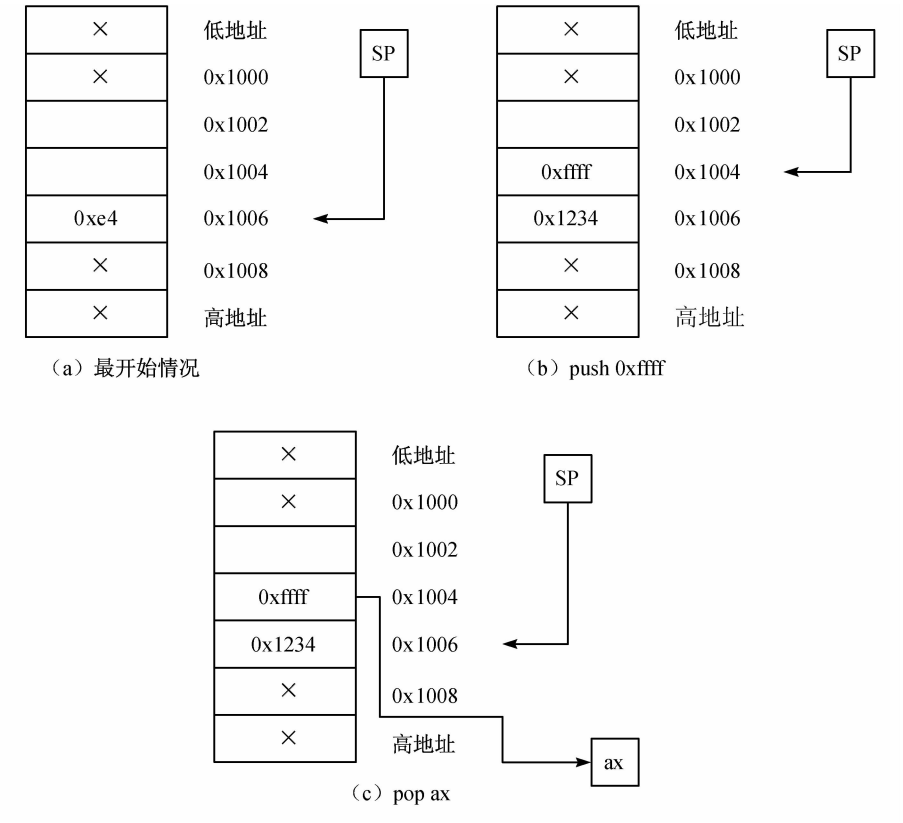


图 1-6　堆栈操作

调用函数必然会有 IP 寄存器的变化，有时还需要存储参数，所以用到堆栈；有时为了保护一些暂时数据，使用堆栈无疑再好不过；为了防止函数改变寄存器的值，可以在进入函数时再把寄存器入栈，离开函数时再把寄存器出栈。

1.3.3　存储器物理地址的形成

8086/8088 CPU 的地址线有 20 根，可以直接寻址 $2^{20}=1$ MB 存储器空间。而 8086/8088 CPU 寄存器的字长为 16 位，只能直接寻址 $2^{16}=64$ KB，无法寻址 1 MB。如何使 16 位的寄存器寻址 20 位存储器物理地址？8086/8088 采用了存储器地址分段的方法。

1 MB 存储器的所有物理地址可表示如下：

```
00000H   00001H   00002H  …        0000FH
00010H   00011H   00012H  …        0001FH
                          ⋮
0FFFF0H  0FFFF1H  0FFFF2H  …      0FFFFFH
```

将整个存储器地址分成许多逻辑段，每个逻辑段的容量最多为 64 KB，允许它们在整个存储器空间浮动，各个逻辑段可以紧密相连，也可以重叠。对于任何一个物理地址来说，可以唯一地被包含在一个逻辑段中，也可以被包含在多个相互重叠的逻辑段中，只要能得到它

所在段的首地址和段内的相对地址，就可以对它进行访问。在8086/8088存储空间中，从0地址开始，把每16个连续字节的存储空间称为小节。为了简化操作，逻辑段必须从任一小节的首地址开始。这样划分的特点是：在十六进制表示的地址中，最低位为0（即20位地址中的低4位为0）。这样，在1 MB的地址空间中，共有64 KB小节。

综上所述，分段的原则如下：

(1)每个段的最大长度为64 KB。

(2)段的首地址能被16整除。

在8086/8088中，每一个存储单元都有一个唯一的20位地址，此地址为该存储单元的物理地址。CPU访问存储器时，必须先确定所要访问的存储单元地址才能取得该单元的内容。20位的物理地址由16位的段地址和16位的段内偏移地址计算得到。段地址是每一逻辑段的起始地址，必须是每个小节中的首地址，其低4位一定是0，于是在保留段地址时，可以只取段地址的高16位。偏移地址则是在段内相对于段起始地址的偏移值。因此任一存储单元物理地址的计算方法如下：

物理地址＝段基址×10H＋段内偏移地址

前面所述的CS、DS、ES、SS段寄存器即是用来存放段基址的。

1.4 数据在机内的表示形式

数据是计算机处理的对象。这里的“数据”含义非常广泛，包括数值、文字、图形、图像、视频等各种数据形式。计算机内部一律采用二进制表示数据。下面来讨论数值和字符数据在机内的表现形式。

1.4.1 数值在计算机内的表示形式

数值数据在计算机内的表示形式有两种：定点表示法和浮点表示法。计算机处理的数值数据多数带有小数，小数点在计算机中通常有两种表示方法：一种是约定所有数值数据的小数点隐含在某一个固定位置上，称为定点表示法，简称定点数；另一种是小数点位置可以浮动，称为浮点表示法，简称浮点数。浮点表示法比定点表示法所表示的数的范围大，精度高。在选择计算机的数值数的表示方式时，需要考虑以下几个因素：

(1)要表示的数的类型（小数、整数、实数和复数）。

(2)可能遇到的数值范围。

(3)数值精确度。

(4)数据存储和处理所需要的硬件代价。

8086宏汇编语言中的数值数据均是指无符号定点数，由于是将小数点固定在第0位的后面，因此，在不特别说明时所提到的数都是整数；对于有符号数则一律采用n位二进制补码表示，n可以是16位，也可以是8位。下面讨论有符号数值数据在字节及字中的表示方法及表示范围。

【例 1-1】 设 M=60=3CH,则 M 的 8 位补码表示为:$[M]_{补}$=3CH。

在主存中的存放形式为:

0 0 1 1 1 1 0 0

而 M 的 16 位补码表示为:$[M]_{补}$=003CH。

在主存中的存放形式为:

0 0 1 1 1 1 0 0

0 0 0 0 0 0 0 0

【例 1-2】 设 M=-60=-3CH,则 M 的 8 位补码表示为:$[M]_{补}$=0C4H。

在主存中的存放形式为:

1 1 0 0 0 1 0 0

而 M 的 16 位补码表示为:$[M]_{补}$=0FFC4H。

在主存中的存放形式为:

1 1 0 0 0 1 0 0

1 1 1 1 1 1 1 1

从以上两例均可看出,M 的 16 位补码实际上是其 8 位补码的符号扩展。由此可得出一个很重要的结论:一个二进制的补码表示的最高位(符号位)向左扩展若干位(符号扩展)之后,所得到的数仍然是该数的补码。

对于 8086 微处理器来说,16 位补码所能表示的最大正数为:$[M]_{补max}$=7FFFH。

在主存中的存放形式为:

1 1 1 1 1 1 1 1

0 1 1 1 1 1 1 1

即 M_{max}=7FFFH=32 767=$2^{15}-1$。

16 位补码所能表示的最小负数为:$[M]_{补min}$=8000H。

在主存中的存放形式为:

0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0

即 M_{min}=8000H=-32 768=-2^{15}。

所以,16 位补码所能表示的最大范围为:8000H≤$[M]_{补}$≤7FFFH,也可写为:

$$-2^{15}\leqslant M\leqslant 2^{15}-1 \text{ 或 } -32\ 768\leqslant M\leqslant 32\ 767$$

但 8086 所允许的数的范围为-65 535~+65 535,具体的处理方法是:如果使用的有符号数在-65 535~-32 769 时,将被作为 1~32 767 之间的正数存放,其映射关系是-65 535=1,-65 534=2,…,-32 769=32 767。

同理,8 位补码所能表示数的范围为:80H≤$[M]_{补}$≤7FH,也可写为:

$$-2^{7}\leqslant M\leqslant 2^{7}-1 \text{ 或 } -128\leqslant M\leqslant 127$$

而 8086 所允许的范围为-256~+256,具体的处理方法与 16 位补码相同。

计算机在进行算术逻辑运算时,总是把参与运算的、用补码表示的操作数作为无符号数处理,这时,数的表示范围则与前面讨论的完全不同。

当 n=16 时,为 0~0FFFFH。

当 n=8 时,为 0~0FFH。

1.4.2 字符数据在计算机内的表示形式

1)ASCII码

计算机中处理的信息并不全是数值,有时需要处理字符或字符串,例如,从键盘输入的信息或打印输出的信息都是以字符方式输入/输出的。因此,计算机必须能表示字符。字符包括如下几种:

(1)字母:A,B,…,Z,a,b,…,z。

(2)数字:0,1,…,9。

(3)专用字符:+、-、*、/、SP(space,空格)…

(4)非打印字符:BEL(bell,响铃)、LF(line feed,换行)、CR(carriage return,回车)…

这些字符在机器中必须用二进制数表示。8086采用最常用的美国标准信息交换代码(American Standard Code for Information Interchange,ASCII)来表示。这种代码用1字节(8位二进制数)来表示一个字符,其中,低7位为字符的ASCII值,最高位一般用做校验位。8086/8088ASCII码表参见附录Ⅰ。

为了区别数值数据,程序中的字符数据全部以单引号或双引号括起来。

2)BCD码

BCD码的特点是利用二进制形式来表示十进制数,即用4位二进制数(0000B~1001B)表示1位十进制数(0~9),而每4位二进制数之间的进位又是十进制的形式。因此BCD码既具有二进制的特点,又具有十进制的特点。

BCD码的使用为十进制数在计算机内的表示提供了一种简单而实用的手段,特别是8088具有直接处理BCD码的指令,更带来了方便。

在8086中,根据在存储器中的存放方式不同,BCD码又分为未压缩的BCD码和压缩的BCD码。未压缩的BCD码每字节只存放1个十进制字位,而压缩的BCD码是在1字节中存放两个十进制字位。

例如,将十进制数8 679用压缩的BCD码表示,则为:

1000011001111001

在主存中的存放形式为:

0 1 1 1 1 0 0 1

1 0 0 0 0 1 1 0

而用未压缩的BCD码表示为:

00001000000001100000011100001001

在主存中的存放形式为:

0 0 0 0 1 0 0 1

0 0 0 0 0 1 1 1

0 0 0 0 0 1 1 0

0 0 0 0 1 0 0 0

1.5 汇编源程序举例

一个汇编源程序一般由几个段组成，其中，必不可少的是代码段和堆栈段，如果程序中需要使用数据存储区，还要定义数据段，必要时还要定义附加数据段。下面以3个完整的汇编源程序实例来说明汇编语言的有关规定和格式。

【例1-3】 二进制加法程序。

两个多字节的二进制数分别放在以ADD1和ADD2为首地址的存储单元中，两个数的字长度放在CONT单元中，最后相加结果放在以SUM为首地址的单元中(所有数的低字节在前，高字节在后)。

解 参考程序如下：

```
DATA SEGMENT                                    ;数据段
     ADD1 DB FEH, 86H, 7CH, 44H, 56H, 1FH
     ADD2 DB 56H, 49H, 4EH, 0FH, 9CH, 22H
     SUM DB 6 DUP(0)
     CONT DB 3
DATA ENDS
STACK SEGMENT PARA STACK 'STACK'                ;堆栈段
     DB 100 DUP(?)
STACK ENDS
CODE SEGMENT                                    ;代码段
     ASSUME: CS: CODE,DS: DATA, ES: DATA, SS: STACK
MADDB: MOV AX, DATA
     MOV DS, AX                                 ;初始化数据段寄存器
     MOV ES, AX                                 ;初始化附加段寄存器
     MOV SI, OFFSET ADD1                        ;被加数地址→(SI)
     MOV DI, OFFSET ADD2                        ;加数地址→(DI)
     MOV BX, OFFSET SUM                         ;和地址→(BX)
     MOV CL, BYTE PTR CONT
     MOV CH, 0                                  ;初始化相加字长度
CLCMADDB1:MOV AX,[SI]
     ADC AX, [DI]                               ;16位相加
     INC SI
     INC SI
     INC DI
     INC DI
     MOV [BX], AX                               ;相加结果送结果单元
     INC BX
```

```
        INC BX
        LOOP MADDB1                          ;执行循环
        HLT
CODE ENDS
END MADDB
```

【例 1-4】 将二进制数转换成 ASCII 码的程序。

编写程序,将一个字长的二进制数转换成一个 ASCII 码表示的字符串。二进制数放在 BINNUM 中,其转换结果放在 ASCBCD 中。

解 参考程序如下:

```
DATA SEGMENT                                 ;数据段
        BINNUM DW 4FFFH
        ASCBCD DB 5 DUP(0)
DATA ENDS
STACK  SEGMENT  PARA STACK 'STACK'           ;堆栈段
        DB 200 DUP(?)
STACK ENDS
CODE SEGMENT                                 ;代码段
        ASSUME:CS: CODE, DS: DATA, ES: DATA, SS: STACK
BINASC: MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
        MOV CX, 5
        XOR DX, DX
        MOV AX, BINNUM
        MOV BX, 10
        MOV DI, OFFSET ASCBCD
BINASC1: DIV BX
        ADD DL, 30H
        MOV [DI], DL
        INC DI
        AND AX, AX
        JZ STOP
        MOV DL, 0
        LOOP BINASC1
        STOP: HLT
CODE ENDS
END BINASC
```

【例 1-5】 ASCII 码转换成二进制数的程序。

将一个 4 位 ASCII 码数字转换成二进制数,ASCII 码数字放在以 ASCSTG 为首地址的

内存单元中(共有4位),转换结果放入以BIN为首地址的内存单元中。

解 参考程序如下:

```
DATA SEGMENT
     ASCSTG DB '5', 'A', '6', '1'
     BIN DB 2 DUP(0)
DATA ENDS
STACK SEGMENT PARA STACK 'STACK'
     DB 100 DUP(?)
STACK ENDS
CODE SEGMENT
     ASSUME:CS: CODE, DS: DATA, SS: STACK
ASCB: MOV AX, DATA
     MOV DS, AX
     MOV CL, 4
     MOV CH, CL
     MOV SI, OFFSET ASCSTG
     CLD
     XOR AX, AX
     XOR DX, DX
ASCB1: LODS ASCSTG
     AND AL, 7FH
     CMP AL, '0'
     JL ERROR
     CMP AL, '9'
     JG ASCB2
     SUB AL, 30H
     JMP SHORT ASCB3
ASCB2: CMP AL, 'A'
     JL ERROR
     CMP AL, 'F'
     JG ERROR
     SUB AL, 37H
ASCB3: OR DL, AL
     ROR DX, CL
     DEC CH
     JNZ ASCB1
     MOV WORD PTR BIN, DX
     HLT
CODE ENDS
END ASCB
```

习 题

(1)简述计算机系统的硬件组成及各部分的作用。

(2)什么是汇编语言源程序、汇编程序、目标程序?

(3)8086 微处理器有哪些寄存器? 如何分组? 各有什么用途?

(4)将下列十六进制数转换为二进制数和十进制数表示:

FFH,0H,5EH,EFH,2EH,10H,1FH,ABH

(5)将下列十进制数转换为 BCD 码表示:

12,24,68,127,128,255,1 234,2 458

(6)将下列十进制数分别用 8 位二进制数的原码、反码和补码表示:

0,-127,127,-57,126,-126,-128,68

(7)计算机中有一个"01100001"编码,如果把它认为是无符号数,它是十进制的什么数? 如果认为它是 BCD 码,则表示什么数? 如果它是某个 ASCII 码,则代表哪个字符?

(8)什么是标志寄存器? 它有什么用途? 状态标志和控制标志有什么区别? 画出标志寄存器(FLAGS),说明各个标志的位置和含义。

(9)什么是 8086 中的逻辑地址和物理地址? 逻辑地址如何转换成物理地址? 请将如下逻辑地址用物理地址表达:

FFFFH:0,40H:17H,2000H:4500H,B821H:4567H

(10)8086 有哪 4 种逻辑段? 各种逻辑段分别是什么用途?

(11)已知 x_1 和 x_2 的值,求$[x_1]_{补}+[x_2]_{补}$,指出结果的符号,判断是否产生了溢出和进位。

①$x_1=+0110010B$,$x_2=+1001011B$。

②$x_1=-0101011B$,$x_2=-1011110B$。

③$x_1=+1100001B$,$x_2=-1011101B$。

第 2 章 寻 址 方 式

计算机中的指令由操作码字段和操作数字段两部分组成。操作码字段指示计算机所要执行的操作，操作数字段则指出在指令执行操作过程中所需要的操作数，即操作的对象。操作数字段可以是操作数本身，也可以是操作数地址或地址的一部分，还可以是指向操作数地址的指针或其他有关操作数的信息。“寻址”就是查找操作数存放在什么地方。操作数所采取的寻址方式会影响机器运行的速度和效率。

2.1 操作数类型

指令系统设计了多种操作数的来源，寻找操作数的过程就是操作数的寻址方式。指令中操作数字段实质上指出参加操作运算的操作数存放在何处。一般来说，存放在指令代码中的操作数称为立即数，存放在 CPU 内部寄存器中的操作数称为寄存器操作数，存放在内部存储器中的操作数称为存储器操作数，绝大部分操作数属于最后一种类型。对于一部分输入/输出指令来说，操作数可以存放在接口电路的寄存器中，每个寄存器都有一个端口号。下面主要介绍 3 种操作数类型：立即数、寄存器操作数及存储器操作数。

2.1.1 立即数

立即数是作为指令代码的一部分出现在指令中的，它通常被作为源操作数使用。为寄存器赋初值、地址赋初值等都是立即数的例子。在汇编指令中，立即数可以用二进制、十进制或十六进制形式表示，也可以写成一个可求出确定值表达式的形式来表示。从键盘输入二进制或十六进制立即数时，后面一定要跟 B 或 H 标识符，输入十进制的立即数时，其标识符 D 可写可不写，没有标识符的立即数，机器都默认为十进制数。

2.1.2 寄存器操作数

寄存器操作数是把操作数存放在CPU中的寄存器内，即用寄存器存放源或目的操作数。在汇编指令中给出了寄存器的名称。在双操作数指令中，寄存器操作数可以作为源操作数，也可以作为目的操作数。有的指令虽然没有明确给出寄存器名，但它隐含着在某个通用寄存器中。如DAA十进制调整指令，其隐含的寄存器一定是AL。使用寄存器操作数可以提高CPU的运行速度。可用的寄存器主要包括通用寄存器组和段寄存器组(CS不能用于DST(目的操作数))。对于16位操作数，可以用AX、BX、CX、DX、SI、DI、SP和BP；对于8位操作数，可以用AL、AH、BL、BH、CL、CH、DL、DH；对于386及其后继机型，可以用EAX、EBX、ECX、EDX、ESI、EDI、ESP和EBP。

2.1.3 存储器操作数

存储器操作数是指操作数存放在主存储器中，因此，在汇编指令中应给出存储器的地址。

需要说明的是，存储器操作数所在的存储器地址应该是物理地址，但在汇编指令中，通常只给出有效地址(EA，它是以各种寻址方式给出的)，而段地址(在段寄存器中)是通过隐含方式(或段超越)使用的。

程序中使用大量的存储器操作数，CPU访问主存必须先传送地址，发出相应的控制信号，等待存储器功能就绪等，所以对存储器操作数进行寻址的速度较慢。另外，从程序运行时的数据结构来看，存储器操作数常常不是单个数，而是成组的以表格或数组的形式存放在主存储器的某一个特定区域中。

2.2 有效地址和段超越

当操作数存放在存储器中时，存储器存储单元的物理地址由两部分组成：偏移地址和段地址。在8086/8088的各种寻址方式中，寻找的存储单元所需的偏移地址称为有效地址(EA)。在实模式下提供有效地址需要使用的有关寄存器分别为：

(1)段寄存器CS、DS、ES和SS。段寄存器中存放逻辑段的基址的高16位。

(2)通用寄存器BX。

(3)基址指针寄存器BP。

(4)变址寄存器SI和DI。

其中，(2)、(3)、(4)中存放逻辑段的偏移地址。

存储器操作数寻址时，存储单元的物理地址的另一部分是段地址。8086/8088指令系统中对段地址有个基本规定，即所谓的Default(默认)状态：在正常情况下，由寻址方式中有效地址规定的基地址寄存器来确定段寄存器，即只要寻址方式中出现BP寄存器作为基地

址，段寄存器就一定采用堆栈段(SS)寄存器；其余的情况都采用数据段(DS)寄存器。

指令中的操作数也可以不在基本规定的段区内，但必须在指令中指定段寄存器，这就是段超越。例如，MOV AL,[2000H]，此存储单元的物理地址为 10H×DS+2000H，数据是存放在数据段中的；而对于 MOV AL,ES:[2000H]，其存储单元的物理地址为 10H×ES+2000H，该指令的源数据在附加段中。段寄存器使用的相关约定如表 2-1 所示。

表 2-1　段基址和偏移量的约定情况

存储器操作数操作类型	约定段寄存器	允许指定的段寄存器	偏 移 量
指令	CS	无	IP
堆栈操作	SS	无	SP
普通变量	DS	ES、SS、CS	EA
字符串操作指令的源串地址	DS	ES、SS、CS	SI
字符串操作指令的目标串地址	ES	无	DI
BP 作为基址寄存器	SS	DS、ES、CS	EA

2.3　与数据有关的寻址方式

操作数是计算机的操作对象，但通常并不是直接给出操作数，而是给出操作数的地址，甚至地址也不直接给出，而是给出计算操作数有效地址的方法，这种方法称为寻址方式。寻址的目的是寻找操作数。学习并掌握操作数的寻址方式是极其重要的。

2.3.1　立即寻址方式

有些操作数紧跟在指令操作码后面，作为指令的操作数字段存放在指令代码中，这种操作数就是立即数。立即数可以是 8 位、16 位或 32 位数。

汇编格式：m

功能：指令下一单元的内容为操作数 m，如图 2-1 所示。

操作：将一个常数送到寄存器中或者存储器中的一个单元里。

图 2-1　立即寻址方式功能示意图

其中，m 即立即操作数。

这种方式就是将操作数直接写在指令中，例如：MOV AH,20。其中，目的操作数地址是 AH，源操作数是 20，其地址是指令的下一单元。源操作数的地址如图 2-2 所示。

图 2-2 源操作数的地址示意图

源操作数 20 紧跟着指令操作码存放在代码段之中。该指令的功能是将操作数 20 送入 AH 之中。

【例 2-1】 假设有指令 MOV AL,8,问该指令执行后,(AL)的值是什么?

程序代码段所在的存储器如图 2-3 所示。

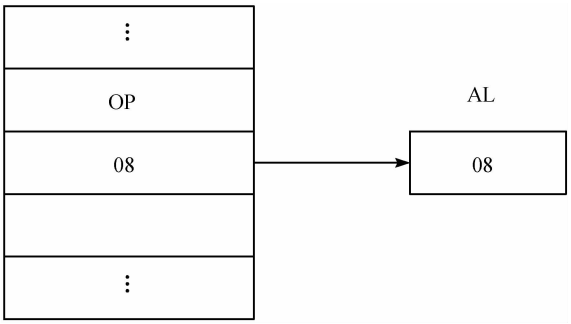


图 2-3 MOV AL,8 指令立即寻址执行示意图

因此指令执行后(AL)的值为 8。

【例 2-2】 假设有指令 MOV AX,1234H,问该指令执行后,(AX)的值是什么?

程序代码段所在的存储器如图 2-4 所示。

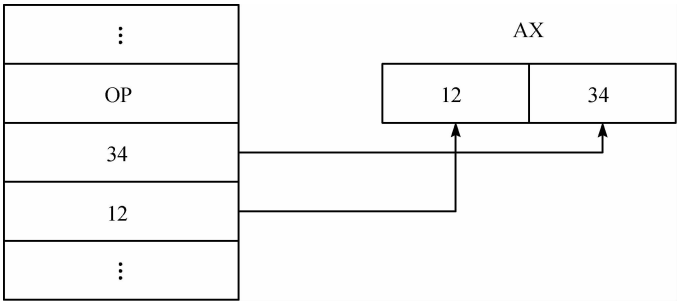


图 2-4 MOV AX,1234H 指令立即寻址执行示意图

因此指令执行后(AX)的值为 1234H。

注意: 目的操作数寻址时不能使用立即寻址方式。指令的源操作数长度应与目的操作数长度一致。

2.3.2 寄存器寻址方式

寄存器寻址方式又称寄存器直接寻址方式,寄存器的内容就是要找的操作数,指令指定

寄存器名。对于16位操作数，寄存器可以使用AX、BX、CX、DX、SP、BP、SI和DI等；对于8位操作数，寄存器可以选用AH、AL、BH、BL、CH、CL、DH和DL。

该寻址方式下由于操作数存储在寄存器中，不需要访问主存储器来取得操作数，所以其存取效率较高。对于那些需要经常存取的操作数，采用该方式较为合适。

汇编格式：R

功能：寄存器R的内容是操作数。

寄存器寻址方式如图2-5所示。

图2-5　寄存器寻址方式示意图

【例2-3】 假设有指令INC AX，执行前(AX)＝26H。其中，INC为加1指令的操作符，其操作数地址为寄存器AX，即操作数在AX之中。问该指令执行后，(AX)的值是什么？

解　执行前：(AX)＝26H，如图2-6所示。

图2-6　INC AX指令寄存器寻址执行前的示意图

执行：(AX)＋1→(AX)

执行后：(AX)＝27H

因此该指令执行后(AX)的值是27H。

【例2-4】 假设有指令DEC BX，执行前(BX)＝68H。其中，DEC为减1指令的操作符，其操作数地址为寄存器BX，即操作数在BX之中。问该指令执行后，(BX)的值是什么？

解　执行前：(BX)＝68H

执行：(BX)－1→(BX)

执行后：(BX)＝67H

因此该指令执行后(BX)的值是67H。

【例2-5】 假设有指令MOV BX，AX，在执行前(AX)＝1987H。问该指令执行后，(BX)的值是什么？

解　这是一条双操作数指令，BX为目的操作数地址，AX为源操作数地址。

执行：(AX)→(BX)

执行后：(BX)＝(AX)＝1987H，(AX)中的内容不变。

因此该指令执行后(BX)的值为1987H。

2.3.3　直接寻址方式

在直接寻址方式中，指令直接指明操作数的有效地址，即操作数在主存中的偏移地址，

而其实际的物理地址应由段基值与有效地址的组合来决定。如果采用段超越前缀，则操作数也可含在数据段外的其他段中。直接寻址方式的用途是存取单个变量。

汇编格式：含有变量的地址表达式或段寄存器名[EA]

功能：指令下一字单元的内容是操作数的偏移地址EA，如图2-7所示。

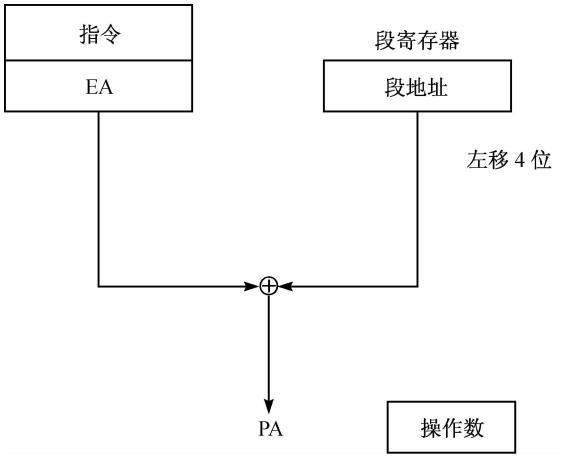


图2-7 直接寻址方式示意图

【例2-6】 假设有指令MOV AX，[0078H]，默认段寄存器为DS，在执行前(DS)＝3000H，内存单元30078H的值为3050H。问该指令执行后，(AX)的值是什么？

解 若用符号VAR表示该地址，则上述指令可以写成MOV AX，VAR或MOV AX，[VAR]。其中源操作数寻址方式仍是直接寻址方式。

其功能与下面两条指令的功能等价：

```
MOV  AL，[0078H]
MOV  AH，[0079H]
```

执行前：(DS)＝3000H，字地址30078H中的内容(30078H)＝3050H。

执行：物理地址PA＝10H×(DS)＋EA＝10H×3000H＋0078H＝30078H。

执行后：(AX)＝(30078H)＝3050H。

执行结果如图2-8所示。

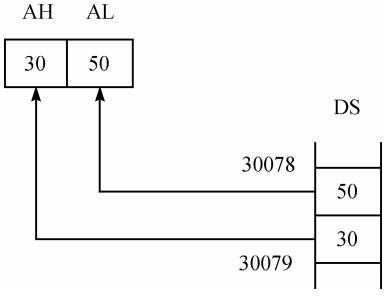


图2-8 执行结果图

因此该指令执行后，(AX)的值为3050H。

注意：直接寻址和立即数寻址在书写方式上是不同的，即直接寻址的地址要放在方括号

中。在直接寻址方式中，当用常量给出操作数的偏移地址时，其前必须写明相应的段寄存器名，否则，会与立即寻址方式混淆。

2.3.4 寄存器间接寻址方式

寄存器间接寻址方式中，操作数的有效地址存放在 BX、BP、SI、DI 4 个寄存器之一中，也即此时寄存器的内容就是操作数的有效地址，操作数在主存储器中。

如果有效地址在 BX、SI、DI 中，则段基值在 DS 寄存器中；如果有效地址在 BP 中，则段基值在 SS 寄存器中。BX、BP 和 SI、DI 均允许使用段超越。对于 386 以上 CPU，这种寻址方式允许使用任何 32 位通用寄存器。

对于寄存器间接寻址方式给出的操作数，其偏移地址 EA 按下面的公式计算：

$$EA=\begin{cases}(SI) & \text{用 SI 作为间址寄存器时}\\(DI) & \text{用 DI 作为间址寄存器时}\\(BX) & \text{用 BX 作为间址寄存器时}\\(BP) & \text{用 BP 作为间址寄存器时}\end{cases}$$

若用寄存器 BX、DI 或 SI 间接寻址，则操作数在当前数据段中，即 DS 寄存器的内容左移 4 位，加上 BX、DI 或 SI 中的偏移地址，形成操作数的物理地址。若用寄存器 BP 间接寻址，则操作数在堆栈中，即 SS 寄存器的内容左移 4 位，加上 BP 中的偏移地址，形成操作数的物理地址。

汇编格式：[R]

功能：R 的内容为操作数的偏移地址。

寄存器间接寻址方式如图 2-9 所示。

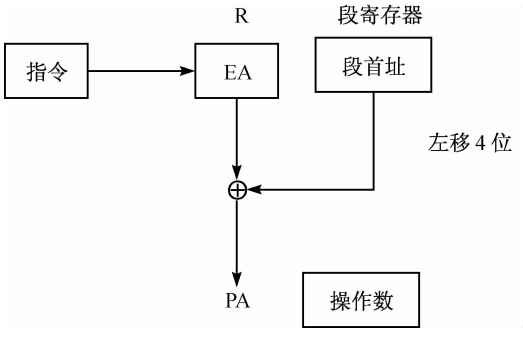


图 2-9 寄存器间接寻址方式示意图

【例 2-7】 假设有指令 MOV AX，[BX]，默认段寄存器为 DS，在执行前(DS)＝3000H，(BX)＝78H，内存单元 30078H 的值为 12H。问该指令执行后，(AX)的值是什么？

解 注意它与寄存器寻址方式指令 MOV AX，BX 在形式上的区别。

执行前：(DS)＝3000H，(BX)＝78H，(30078H)＝12H。

执行：源操作数的物理地址 PA＝(DS)×10H＋(BX)＝3000H×10H＋78H＝30078H。

执行后：(AL)＝(30078H)＝12H。

因此该指令执行后，(AX)的值为 12H。

【例 2-8】 假设有指令 MOV AX,[BP],默认段寄存器为 SS,在执行前(SS)=2000H,(BP)=80H,内存单元 20080H 的值为 5612H。问该指令执行后,(AX)的值是什么?

解 执行前:(SS)=2000H,(BP)=80H,(20080H)=5612H。

执行:源操作数的物理地址 PA=(SS)×10H+(BP)=20080H。

执行后:(AX)= (20080H)=5612H。

因此该指令执行后,(AX)的值为 5612H。

利用这种寻址方式,再配合修改寄存器内容的指令可以方便地处理一维数组,如图 2-10 所示。

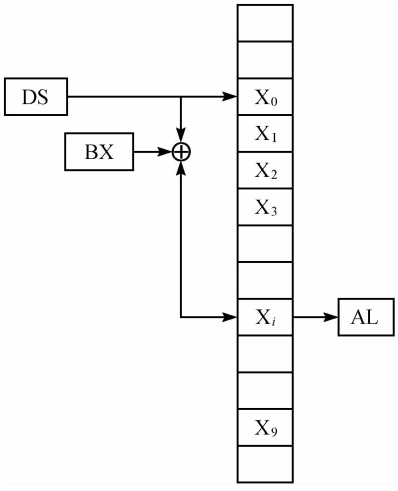


图 2-10 寄存器间接寻址方式处理一维数组的情况

2.3.5 寄存器相对寻址方式

寄存器相对寻址方式的操作数在存储器中,操作数的有效地址是基址寄存器 BX、BP 或变址寄存器 SI、DI 的内容加上指令中指定的 8 位或 16 位位移量(DISP)之和,即

$$EA=\left\{\begin{matrix}(BX)\\(BP)\\(SI)\\(DI)\end{matrix}\right\}+\left\{\begin{matrix}8\text{ 位位移量}\\16\text{ 位位移量}\end{matrix}\right\}$$

在一般情况下,如果使用 BX、SI 或 DI 寄存器,那么段寄存器默认是 DS;如果使用 BP 寄存器,那么段寄存器默认是 SS。

在指令中给定的 8 位或 16 位位移量采用补码形式表示。在计算有效地址时,若位移量为 8 位,则其符号位自动扩展为 16 位带符号数。

这种寻址方式可用于对数组、表格的处理。数组和表格的首地址可以设置为指令中的位移量,利用修改基址或变址寄存器的内容来存取数组或表格中的项值。

【例 2-9】 假设有指令 MOV AL，TABLE[BX]或 MOV AL，[BX＋TABLE]，其中 TABLE 为位移量的符号表示。问该指令执行后，(AL)的值是什么？

解 该指令执行结果是：(AL)←(DS：[BX＋TABLE])，其中，DS 为默认段寄存器，[BX＋TABLE]为内存单元 BX＋TABLE 的值。

使用这种寻址方式可以访问一维数组，其中，TABLE 是数组首地址的偏移量，寄存器中存放数组元素的下标乘以元素长度的积(一个元素占用的字节数)，下标从 0 开始计数。

【例 2-10】 假设有指令 MOV AX，8 [BX](也可以表示为 MOV AX，[BX＋8])，默认段寄存器为 DS，在执行前(DS)＝3000H，(BX)＝70H，内存单元 30078H 的值为 1234H。问该指令执行后，(AX)的值是什么？

解 执行前：(DS)＝3000H，(BX)＝70H，(30078H)＝1234H。

执行：源操作数的物理地址 PA＝10H×(DS)＋(BX)＋DISP＝30078H。

执行后：(AX)＝(30078H)＝1234H。

因此该指令执行后，(AX)的值为 1234H。

该指令执行情况如图 2-11 所示。

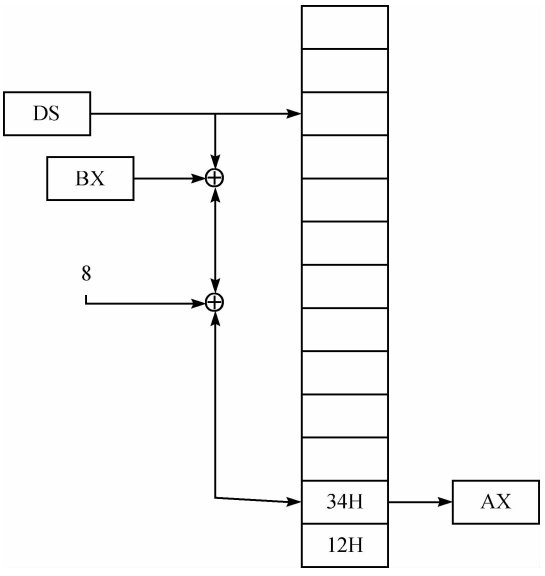


图 2-11 寄存器相对寻址方式的执行情况

注意：位移量可以是一个字节或一个字的带符号数。该方式关于段寄存器的约定和寄存器间接寻址方式的约定相同。

2.3.6 基址变址寻址方式

基址变址寻址方式的操作数在存储器中，操作数的有效地址由一个基址寄存器的内容和一个变址寄存器的内容相加得到，即

$$EA=\begin{Bmatrix}(BX)\\(BP)\end{Bmatrix}+\begin{Bmatrix}(SI)\\(DI)\end{Bmatrix}$$

如果使用基址寄存器BX,那么段寄存器默认为DS;如果使用基址寄存器BP,那么段寄存器默认为SS。80386以上32位基址变址寻址方式组合如图2-12所示。

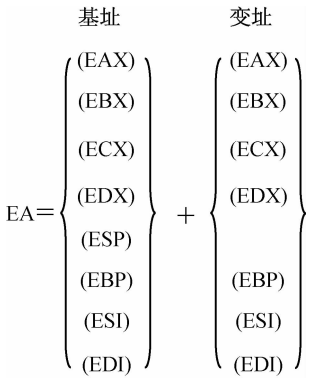


图2-12 80386以上32位基址变址寻址方式组合

该寻址方式允许使用段超越前缀。下面指令的源操作数采用基址加变址寻址方式:

```
MOV  AX, ES:[BX+SI]
```

或写成:

```
MOV  AX, ES:[BX][SI]
```

该寻址方式也适用于数组和表格的处理,首地址存放于基地址寄存器中,而用变址寄存器来访问数组或表格中的各个元素,或反之。由于两个寄存器都可以修改,所以能更灵活地访问数组或表格中的元素。

【例2-11】 假设有指令MOV AL,[BX][SI](也可表示为MOV AL,[BX+SI]),问该指令执行后,(AL)的值是什么?

解 该指令执行结果是:(AL)←(DS:[BX+SI]),其中DS为默认段寄存器,[BX+SI]为内存单元BX+SI的值。

使用该寻址方式可以访问一维数组。用BX存放数组起始地址的偏移量,SI存放数组元素的下标乘以元素的长度,下标从0开始计数,则数组中任一元素的有效地址等于(BX)+(SI)。因为数组的首地址存放在寄存器中,所以可以访问存储器中任意已知地址指针的数组。基址变址寻址方式的执行情况如图2-13所示。

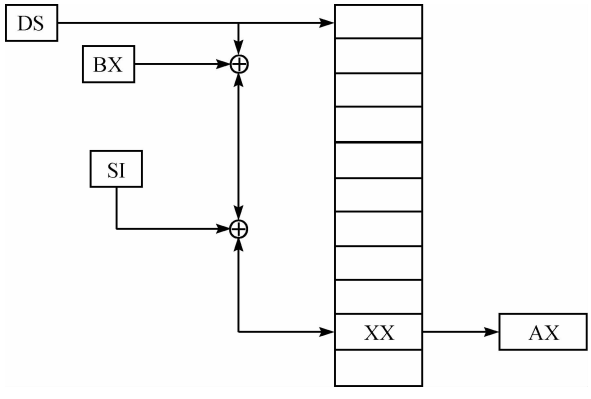


图2-13 基址变址寻址方式的执行情况

2.3.7 相对基址变址寻址方式

相对基址变址寻址方式的操作数存在存储器中，操作数的有效地址由一个基址寄存器(BX 或 BP)和一个变址寄存器(SI 或 DI)的内容及指令中给定的 8 位或 16 位位移量相加得到，即

$$EA=\left\{\begin{matrix}(BX)\\(BP)\end{matrix}\right\}+\left\{\begin{matrix}(SI)\\(DI)\end{matrix}\right\}+\left\{\begin{matrix}8\text{ 位位移量}\\16\text{ 位位移量}\end{matrix}\right\}$$

如果使用基址寄存器 BX，那么段寄存器默认为 DS；如果使用基址寄存器 BP，那么段寄存器默认为 SS。在指令中给定的 8 位或 16 位位移量采用补码形式表示。在计算有效地址时，若位移量是 8 位，则被扩展成 16 位带符号数。

汇编格式：X [BR＋IR]

功能：BR 的内容加上 IR 的内容，再加上 X，所得之和是操作数的偏移地址，如图 2-14 所示。

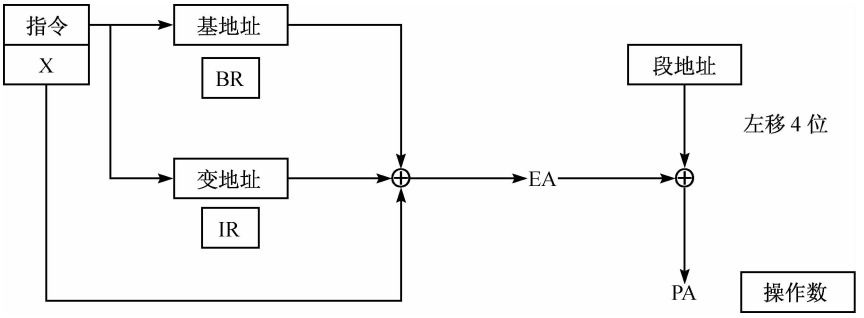


图 2-14　相对基址变址寻址方式功能示意图

其中，X 表示位移量，其值是用 8 位或 16 位二进制补码表示的有符号数；BR 表示基址寄存器，只能选用 BX、BP 之一；IR 表示变址寄存器，只能选用 SI、DI 之一。

基址寄存器选用 BP 或 BX，决定了是从堆栈段中还是从数据段中获取操作数。也就是说，若选用 BP 作为基址寄存器，则将 SS 寄存器的内容左移 4 位与偏移地址相加，形成操作数的物理地址；若选用 BX 作为基址寄存器，则将 DS 寄存器的内容左移 4 位与偏移地址相加，形成操作数的物理地址。

【例 2-12】 假设有指令 MOV AX，6 [BX＋SI]，默认段寄存器为 DS，在执行前(AX)＝48H，(BX)＝40H，(SI)＝30H，(DS)＝1000H，内存单元 10076H 的值为 80H。问该指令执行后，(AX)的值是什么？

解　该例中，目的操作数地址是 AX。源操作数地址用基址加变址方式给出，位移量为 6，基址寄存器选用了 BX，变址寄存器选用了 SI。由于源操作数选用 BX 作为基址寄存器，所以其物理地址由 DS 寄存器的内容左移 4 位与偏移地址相加形成。

执行前：(AX)＝48H，(BX)＝40H，(SI)＝30H，(DS)＝1000H，(10076H)＝80H。

源操作数的寻址过程如图 2-15 所示。

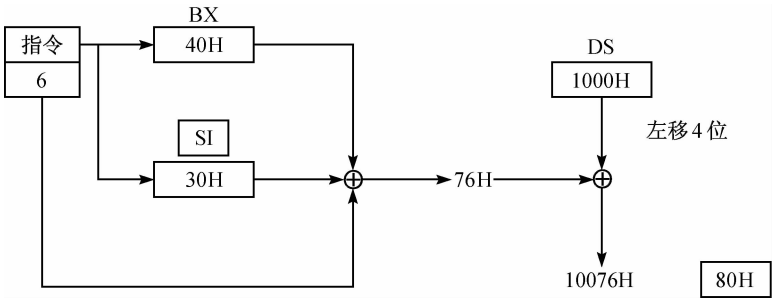


图 2-15 相对基址变址寻址方式的执行情况

执行:(10076H)→(AX)。

执行后:(AX)=80H,(BX)、(SI)、(DS)、(10076H)未变。

因此该指令执行后,(AX)的值为 80H。

【例 2-13】 计算下列 4 条指令中操作数的地址并指出执行结果。

(1)MOV 5 [BX+SI], AX

(2)MOV 6 [BP+SI], BX

(3)MOV 7 [BX+DI], CX

(4)MOV 8 [BP+DI], DX

解 上述 4 条指令的目的操作数都是相对基址变址寻址方式。第(1)、(3)条指令选用了 BX 作为基址寄存器,第(2)、(4)条指令选用了 BP 作为基址寄存器,所以,AX、CX 的内容将被送往数据段中的相应单元,BX、DX 的内容将被送往堆栈段中的相应单元。

假定执行前:

(DS)=1000H,(SS)=2000H,(BX)=280H,(BP)=360H,(SI)=72H,(DI)=50H,(AX)=11H,(CX)=28H,(DX)=20H,则各条指令的操作数地址计算过程及执行结果如下。

(1)目的操作数地址:EA=[BX]+[SI]+5=280H+72H+5=2F7H

PA=(DS)×10H+EA=102F7H

源操作数地址:AX

执行:(AX)→(102F7H)

(2)目的操作数地址:EA=[BP]+[SI]+6=360H+72H+6=3D8H

PA=(SS)×10H +EA=203D8H

源操作数地址:BX

执行:(BX)→(203D8H)

(3)目的操作数地址:EA=[BX]+[DI]+7=280H+50H+7=2D7H

PA=(DS)×10H +EA=102D7H

源操作数地址:CX

执行:(CX)→(102D7H)

(4)目的操作数地址:EA=[BP]+[DI]+8=360H+50H+8=3B8H

PA=(SS)×10H +EA=203B8H

源操作数地址:DX
执行:(DX)→(203B8H)
上述4条指令执行后,存储器中相应单元的内容如下:
(102F7H)=11H　　;由第(1)条指令送入
(203D8H)=280H　;由第(2)条指令送入
(102D7H)=28H　　;由第(3)条指令送入
(203B8H)=20H　　;由第(4)条指令送入

2.4 与转移地址有关的寻址方式

这种寻址方式用来确定无条件转移指令JMP和子程序调用指令CALL的转移地址。转移地址是借助与数据有关的各种寻址方式得到转移目标地址的。

无条件转移或子程序调用的目标转移地址分为段内和段间两种情形。若为段内转移或段内调用子程序,则其特点为只改变逻辑地址中的IP值,因为只是段内转移时,CS的值不会改变;若为段间转移或段间子程序调用,则目标转移地址的CS与IP都要改变。与转移地址有关的寻址方式中的目标地址只存在寻找与计算的问题。

2.4.1 段内直接寻址

段内直接寻址的目标转移地址的有效地址由当前IP寄存器内容和指令中指定的8位或16位位移量相加得到。

这种寻址方式在指令中直接指出转向地址,如:

```
JMP SHORT NEXT
```

其中,NEXT表示转移的符号地址,SHORT为属性操作符,它指示后面跟着的是一个带符号的8位位移量,称为短转移。

再如:

```
JMP NEAR PTR PROC
```

其中,PROC表示转移的符号地址,操作符NEAR PTR指示后面是一个带符号的16位位移量,称为近转移。

这种寻址方式适用于条件转移及无条件转移,也适用于段内子程序调用指令。条件转移指令只能用短转移寻址方式,子程序调用只能用近转移寻址方式。

2.4.2 段内间接寻址

这种寻址方式是把目标转移地址的有效地址直接存放在一个寄存器或一个存储单

元中。

段内间接寻址转移指令的汇编格式如下：

```
JMP  BX                    ;寄存器 BX 中的内容就是目标 IP
JMP  WORD PTR[BX]          ;按寄存器间接寻址方式寻找目标 IP
```

其中，WORD PTR 为操作符，用以说明其后的寻址方式所取得的转移地址是一个字的有效地址。

段内间接寻址方式指令格式及举例如表 2-2 所示。

表 2-2 段内间接寻址方式指令格式及举例

格　式	举　例	注　释
JMP 通用寄存器	JMP BX	16 位转移地址在 BX 中
JMP 内存单元	JMP VAR	16 位转移地址在 VAR 字型内存变量中
	JMP WORD PTR [BX]	16 位转移地址在 BX 所指向的内存变量中

【例 2-14】 假设有指令 JMP BX，在执行前(BX)＝0300H，问该指令执行后的作用？

解 CPU 把寄存器 BX 中的内容送到目标 IP，然后执行 CS:300H 处的指令，实现段内间接转移。

【例 2-15】 假设有指令 JMP WORD PTR[BX]，默认段寄存器为 DS。在执行前(DS)＝2000H，(BX)＝0300H，(IP)＝0100H，(20300H)＝0500H。问该指令执行后的作用？

解 CPU 首先根据寻址方式计算存放转移目标地址的内存单元的地址：PA＝(DS)×10H＋(BX)＝20300H，从(PA)取转移目标地址的 EA 送到 IP 中，EA＝(20300H)＝0500H→(IP)，然后执行 CS:500H 处的指令，实现段内间接转移。

注意：这种寻址方式以及后面要介绍的两种段间寻址方式都不能用于条件转移指令。

2.4.3 段间直接寻址

段间直接寻址是在指令操作码后直接提供了转移地址的偏移地址(目标 IP)和段基值(目标 CS)，所以只要用指令中指定的偏移地址取代 IP 寄存器内容，用指令中指定的段基值取代 CS 寄存器的内容就完成了从一个代码段到另一个代码段的转移操作。

指令的汇编格式为：

```
JMP FAR PTR PROC
```

其中，PROC 代表转移地址的符号地址，FAR PTR 是指示段间转移操作符。段间直接寻址方式如图 2-16 所示。

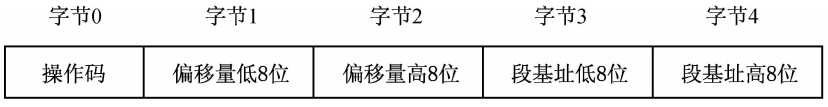


图 2-16 段间直接寻址方式

2.4.4 段间间接寻址

这种寻址方式仍然是将相继两个字的内容装入 IP 和 CS 来达到段间转移的目的，但这两个字的存储器地址是通过指令中的数据寻址方式来取得的。

指令的汇编格式为：

```
JMP  DWORD PTR  [SI]
JMP  DWORD PTR  [BX+TABLE]
```

其中，DWORD PTR 为双字操作符，说明转移地址需取双字为段间转移。第一个字为转移地址的偏移地址，送到 IP 中，第二个字则送到 CS 中。

【例 2-16】 若(DS)＝2000H，(BX)＝0300H，(20300H)＝0500H，(20302H)＝6010H，则指令 JMP DWORD PTR [BX]执行后，(CS)＝6010H，(IP)＝0500H，因此程序转到 6010:0500 处执行。

上面介绍的与转移地址有关的寻址方式完全适用于 386 以上的实模式环境。保护模式下的虚拟 8086 方式转移地址的形成与此类似，只是 32 位的偏移量送给 EIP，高 16 位清 0，这是因为段长不能超过 64 KB。而段间转移的目标地址采用 48 位全指针形式，即 32 位的偏移量和 16 位的段选择子进行运算。

2.5 寻址方式总结

至此，已介绍了 7 种基本的寻址方式，下面简单总结寻址方式的使用情况。

1)立即寻址方式

立即寻址方式主要用在以下几个方面：

(1)为寄存器赋值，如“MOV CX，12”。

(2)为存储单元赋值，如“MOV A，5”。

(3)与寄存器中的内容进行算术或逻辑运算，如“ADD AL，5”。

(4)与存储单元中的内容进行算术或逻辑运算，如“ADD C，10”。

2)直接寻址方式

通常，用直接寻址方式可以方便地访问存储器中某一数据存储单元。被访问的单元一般由变量或含有变量的地址表达式给出地址。

3)寄存器寻址方式

寄存器寻址方式通常用来存放运算对象、中间及最后运算结果、计数值等。实际上这种方式是与 CPU 的特定存储单元(即寄存器)进行数据存取，所以其运行速度快。

4)其他寻址方式

其他寻址方式主要包括寄存器间接寻址、寄存器相对寻址、基址变址寻址和相对基址变址寻址方式4种。这4种方式通常用来访问存储器中的一片连续单元,这些单元的地址不便用变量一一给出,而是为其存储区首址定义一个变量名。

这7种寻址方式的比较如表2-3所示。

表2-3 7种寻址方式的比较

寻址方式	汇编格式	操作数位置	操作数地址	举 例
立即寻址	n	指令	指令操作码的下一单元	MOV BX,20H
寄存器寻址	R	寄存器R的内容	指令所指明的寄存器	MOV BX,AX
直接寻址	含有变量的地址表达式或段寄存器名:[EA]	数据段或其他段	其所在段的段寄存器内容左移4位与指令中给出的偏移地址(指令下一字单元的内容)相加而成	MOV AX,[06H]
寄存器间接寻址	[R]	存储器	数据段或者堆栈段寄存器内容左移4位与操作数的偏移地址(指令指明的寄存器内容)相加	MOV AX,[BX]
寄存器相对寻址	X[R]	存储器	数据段或者堆栈段寄存器内容左移4位与操作数的偏移地址(R的内容与X相加之和)相加	MOV AX,6[BX]
基址变址寻址	[BR+IR]	存储器	数据段或者堆栈段寄存器内容左移4位与操作数的偏移地址(BR的内容加上IR的内容)相加	MOV AL,[BX][SI]
相对基址变址寻址	X[BR+IR]	存储器	数据段或者堆栈段寄存器内容左移4位与操作数的偏移地址(BR的内容加上IR的内容再加上X)相加	ADD AX,7[BP+DI]

习 题

(1)分别指出下列指令中源操作数和目的操作数的寻址方式。

①MOV DI, 8　　②MOV SI, [DI]

③ADD AX, 6[BX]　　④SUB AX, 3[BX+SI]

⑤MOV [DI+2], DX　　⑥MOV SI, [12H]

⑦ADD [DI], DI　　⑧ADD BX,9[BP+SI]

(2)试分别指出下列各指令语句的语法是否有错,如有错,指明是什么错误。

①MOV [BX][BP], AX　　②TEST [BP], BL

③ADD SI, ABH　　④AND DH, DL

⑤CMP CL, 1234H　　⑥SHR [BX][DI], 3

⑦NOT CX, AX　　⑧LEA DS, 35[SI]

⑨INC CX, 1　　⑩PUSH 45[DI]

(3)假定(BX)=637DH,(SI)=2A9BH,位移量 D=3237H,试确定在以下各种寻址方式下的有效地址。

①立即寻址

②直接寻址

③使用 BX 的寄存器寻址

④使用 BX 的间接寻址

⑤使用 BX 的寄存器相对寻址

⑥基址变址寻址

⑦相对基址变址寻址

(4)已知(DS)=091DH,(SS)=1E4AH,(AX)=1234H,(BX)=0024H,(CX)=5678H,(BP)=0024H,(SI)=0012H,(DI)=0032H,(09226H)=00F6H,(09228H)=1E40H,(1E4F6H)=091DH。下列各指令或程序段分别执行后的结果如何?

①MOV CL, 20H[BX][SI]　　②MOV [BP][DI], CX

③LEA BX,20H[BX][SI]　　④MOV AX, 2[BX]

⑤LDS SI, [BX][DI]　　⑥MOV [SI], BX

⑦XCHG CX, 32H[BX]　　⑧XCHG 20H[BX][SI], AX

(5)现有(DS)=2000H,(BX)=0100H,(SI)=0002H,(20100H)=12H,(20101H)=34H,(20102H)=56H,(20103H)=78H,(21200H)=2AH,(21201H)=4CH,(21202H)=B7H,(21203H)=65H,试说明下列各条指令执行完后 AX 寄存器的内容。

①MOV AX,1200H

②MOV AX,BX

③MOV AX,[1200H]

④MOV AX,[BX]

⑤MOV AX,[BX+1100]

⑥MOV AX,[BX+SI]

⑦MOV AX,[BX+SI+1100]

(6)已知(SS)=0915H,(DS)=0930H,(SI)=0A0H,(DI)=1C0H,(BX)=80H,(BP)=470H。现有一指令“MOV AX,OPRD”,如源操作数的物理地址为 095C0H,试用 4 种不同的寻址方式改写此指令(要求上述每个已知条件至少要使用一次)。

(7)试述指令 MOV AX, 2010H 和 MOV AX, DS:[2010H]的区别。

(8)JMP FAR PTR ABCD(ABCD是符号地址)的转移方式是什么?

(9)MOV AX, ES:[BX][SI]的源操作数的物理地址是多少?(用CS、DS、ES、SS、BX、SI表示出即可)

(10)运算型指令寻址和转移型指令寻址的不同点表现在哪里?

(11)如果TABLE为数据段中0032单元的符号名,其中存放的内容为1234H,当执行指令"MOV AX, TABLE"和"LEA AX, TABLE"后,AX中的内容分别为多少?

(12)在1000H单元中有一条二字节指令JMP SHORT LAB,如果其中的偏移量分别为30H、6CH、0B8H,则转向地址LAB的值分别为多少?

第 3 章 8086 基本指令系统

指令系统是计算机所能执行的全部指令的集合，它描述了计算机内全部的控制信息和“逻辑判断”能力，是表征一台计算机性能的重要因素，它的格式与功能不仅直接影响到机器的硬件结构，而且直接影响到系统软件，影响到机器的适用范围。8086/8088 CPU 指令系统包含的指令助记符约上百种，它们与第 2 章介绍的各种寻址方式相结合，可以构成上千条不同的指令。一般来说，8086 全部指令按功能可分成六大类：数据传送指令、算术运算指令、逻辑指令、串操作指令、控制转移指令和处理器控制指令。这六类指令在附录Ⅱ中详细列出，请读者在学习过程中前后进行对照。本章对前四类基本指令进行介绍，后面两类指令在后面章节中介绍。

3.1 数据传送指令

数据传送指令负责将数据、地址或立即数传送到寄存器或存储单元中。数据传送是计算机最基本、最重要的一种操作，在实际程序中，数据传送指令使用的比例是最高的。所以，数据传送是否方便灵活、速度是否快，是指令系统要考虑的重要问题。在 8086/8088 CPU 指令系统中，数据传送指令共有 14 条，其中除标志寄存器传送指令外，该类指令均不影响任何标志位。

3.1.1 机器指令格式

在介绍具体指令前，先来了解 8086/8088 CPU 的机器指令格式。

计算机中的一条指令就是机器语言的一条语句，它是一组有意义的二进制代码，由操作码字段和操作数字段两个部分组成。其中，操作码字段表示该指令应进行什么性质的操作，操作数字段则指出指令执行的参与者，也就是各种操作的对象。例如，对于一条加法指令来说，除了需要在操作码字段指定执行加法操作外，还必须在操作数字段提供参与加法运算的

加数和被加数。指令的一般格式如图 3-1 所示。

操作码	操作数	…	操作数

图 3-1　指令的一般格式

指令的操作码字段在机器里的表示相对比较简单，只需要对每一种操作指定一个确定且唯一的二进制代码即可。不同的指令用不同的编码表示，每一种编码代表一种指令。组成操作码字段的位数一般取决于计算机指令系统的规模。

指令的操作数字段的情况相对比较复杂。

首先，从操作数字段的数量来说，可以有 1 个、2 个或 3 个，通常称为一地址指令、二地址指令或三地址指令。例如，单操作数指令就是一地址指令，它只需要指定一个操作数，如逻辑非指令只需要指出求非的操作数即可。此外，有的双操作数指令也可能以一地址指令的形式出现，这种情况下指令中只给出一个操作数，而另一个操作数是隐含默认的，如乘法指令。大多数运算指令都是双操作数指令，对于这种指令，有的计算机使用三地址指令，除了给出参加运算的两个操作数之外，还在指令中指出运算结果的存放地址。而以 IBM-PC 为代表的近代多数机器则使用二地址指令，此时分别称两个操作数为源操作数和目的操作数。指令执行之后，结果存放到目的操作数的地址之中。这意味着经过运算后，参加运算的一个操作数将会被覆盖丢失。一般来说，这个问题不会有什么影响，如果以后的运算中还会用到这个操作数，则应在运算前先对其保存一个副本。

其次，从操作数存放的位置来看，不同位置对应的地址码长度相差很大，有时可能还要使用以表格或数组形式存放的成组的数据。这时就要结合第 2 章介绍的各种寻址方式来指定地址，使得指令尽可能简短，使操作数提取尽可能方便。

3.1.2　通用数据传送指令

通用数据传送指令负责 CPU 内部的寄存器以及内存单元之间的数据传递，是数据传送指令中使用最多的一组指令。该类指令主要包括：MOV、XCHG 和堆栈操作指令。

1）MOV 传送指令

格式：MOV DST，SRC

执行操作：(DST)←(SRC)

其中，DST 表示目的操作数，SRC 表示源操作数。

该指令为双操作数指令，源操作数与目的操作数的寻址方式必须满足以下规定：

(1)源操作数与目的操作数的类型必须明确且一致。当两个操作数只有一个类型明确时，另一个与其视为同一类型；当两个操作数类型均不明确时，必须用 BYTE PTR(字节属性)或 WORD PTR(字属性)对其中一个存储器操作数进行定义。

(2)源操作数可采用 7 种寻址方式中的任何一种，目的操作数可采用除立即数寻址之外的任何一种寻址方式。

(3)若源操作数采用立即数寻址，则目的操作数不能为段寄存器。

(4)若源操作数不采用立即数寻址，那么源操作数与目的操作数之中必须至少有一个采

用寄存器寻址。也就是说,MOV 指令不能用于两个存储单元之间直接传递数据。

(5)源操作数与目的操作数不能同为段寄存器。

(6)目的操作数不能为 CS 寄存器和 IP 寄存器。

注意:以上的一些规定在后面其他指令中也必须遵循,因此在后面的指令介绍中,对于某些共性的规定将不再重复列举。

该指令不影响任何标志位。

【例 3-1】 判断下列指令是否合法。

MOV	DS,AX	√	
MOV	CS,AX	×	参见规定(6)
MOV	AL,0FFFFH	×	参见规定(1)
MOV	AX,5EH	√	
MOV	DX,AL	×	参见规定(1)
MOV	100,DL	×	参见规定(2)
MOV	DS,1234H	×	参见规定(3)
MOV	MEM1,MEM2	×	参见规定(4)
MOV	DS,SS	×	参见规定(5)

2)XCHG 交换指令

格式:XCHG　OPR1,OPR2

执行操作:(OPR1)←→(OPR2)

其中,OPR1 和 OPR2 分别表示两个操作数。

该指令为双操作数指令,源操作数与目的操作数的寻址方式必须满足以下规定:

(1)两个操作数必须至少有一个在寄存器中,即该指令可实现寄存器之间或寄存器与存储器之间交换信息。

(2)不允许使用段寄存器。

(3)两个操作数应同为字节或同位字。

该指令不影响任何标志位。

【例 3-2】 若指令执行前(BX)=1234H,(BP)=0200H,(SI)=0046H,(SS)=2F00H,(2F246H)=5678H,则下面指令执行后会发生什么变化?

XCHG　BX,[BP+SI]

解　由寻址方式可知:

OPR2 的物理地址=2F00H×10H+0200H+0046H=2F246H。

则指令执行后,寄存器 BX 中的内容与地址为 2F246H 存储单元中的内容互换,如下所示:

(BX)=5678H,(2F246H)=1234H。

提示:当寄存器中的内容和累加器 AX 的内容进行交换时,指令的机器代码为 1 字节,执行速度比一般交换要快。

3.1.3　堆栈操作指令

前面已经介绍过,所谓堆栈是在内存中按照“后进先出”原则组织的一个专门区域,通过

一个始终指向堆栈顶部的指针 SP 来进行访问。由于堆栈只有进栈和出栈两个基本操作，因此堆栈操作指令也只有压入堆栈指令 PUSH 和弹出堆栈指令 POP 两条。

注意：堆栈操作指令一般也划分到通用传送指令中，但考虑到其所操作对象的特殊性，本书单独分出介绍。

1）PUSH 进栈指令

格式：PUSH　SRC

执行操作：(SP)←(SP)－2

((SP)＋1，(SP))←(SRC)

该指令为单操作数指令，其中目的操作数是隐含的，即指针 SP 所指向的堆栈顶部的一个字单元。该指令中源操作数的寻址方式必须满足以下规定：

(1)不能使用立即数寻址。

(2)操作必须以字为单位，不允许以字节为单位。

该指令不影响任何标志位。

【例 3-3-0】 若(AX)＝2107H，(SP)＝0200H，执行下面指令后堆栈如何变化？

PUSH　AX

解　指令执行情况如图 3-2 所示。

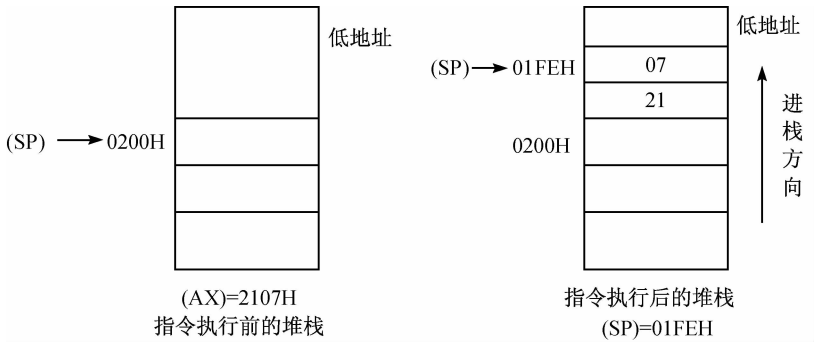


图 3-2　PUSHAX 指令的执行情况

2）POP 出栈指令

格式：POP　DST

执行操作：(DST)←((SP)＋1，(SP))

(SP)←(SP)＋2

该指令为单操作数指令，其中源操作数是隐含的，即指针 SP 所指向的堆栈顶部的一个字单元。该指令的目的操作数的寻址方式必须满足以下规定：

(1)不能使用立即数寻址。

(2)操作必须以字为单位，不允许以字节为单位。

(3)不允许使用 CS 寄存器。

该指令不影响任何标志位。

【例 3-3-1】 若指令执行前(SP)＝0200H，执行如下指令后堆栈如何变化？

POP　AX

解 指令执行情况如图 3-3 所示。

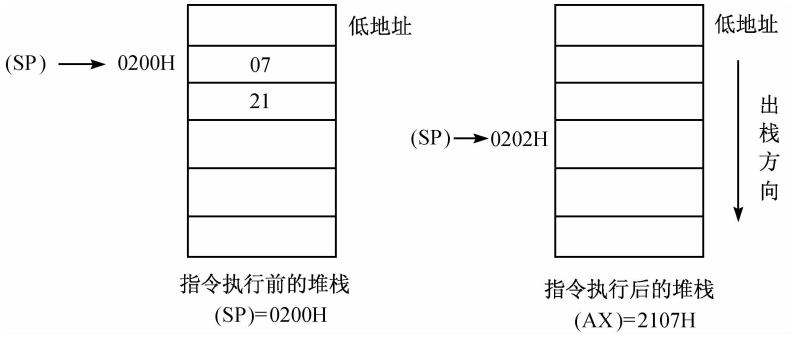


图 3-3 POPAX 指令的执行情况

3.1.4 标志传送指令

标志寄存器是一个存放条件标志、控制标志的寄存器，主要用于反映处理器的状态和运算结果的某些特征及控制指令的执行。标志传送指令主要用于对标志寄存器的内容进行读取和修改，共包括 4 条指令：LAHF、SAHF、PUSHF 和 POPF。

1)LAHF 读标志指令

格式：LAHF

执行操作：(AH)←(FR 的低 8 位)

该指令的两个操作数均是隐含的，源操作数为标志寄存器的低字节，目的操作数为 AH 寄存器。其中，标志寄存器的低字节包含 SF、ZF、AF、PF 和 CF，分别对应第 7、6、4、2 和 0 位，而第 5、3、1 位无定义。

该指令不影响任何标志位。

2)SAHF 取标志指令

格式：SAHF

执行操作：(AH)→(FR 的低 8 位)

该指令的两个操作数均是隐含的，源操作数为 AH 寄存器中的内容，目的操作数为标志寄存器的低字节。该指令将影响 SF、ZF、AF、PF 和 CF 标志位。

3)PUSHF 标志进栈指令

格式：PUSHF

执行操作：(SP)←(SP)－2

((SP)＋1,(SP))←(FR)

该指令的两个操作数均是隐含的，源操作数为标志寄存器的低字节，目的操作数为指针 SP 所指向的堆栈顶部的一个字单元。其功能是将标志寄存器的内容压入堆栈。

该指令不影响任何标志位。

4)POPF 标志出栈指令

格式：POPF

执行操作：(FR)←((SP)＋1,(SP))

(SP)←(SP)＋2

该指令的两个操作数均是隐含的，源操作数为指针 SP 所指向的堆栈顶部的一个字单元，目的操作数为标志寄存器。其功能是将堆栈顶部字单元内容弹出到标志寄存器中。

该指令将影响所有的标志位。

提示：标志传送指令中 SAHF 和 POPF 两条指令将直接影响标志寄存器的内容，利用这一特性，可以快速方便地改变标志寄存器的指定位。

3.1.5　地址传送指令

这类指令用于将地址传送到指定的寄存器中，以提供访问变量的工具。共包括 3 条指令：LEA、LDS 和 LES。其中 LEA 使用得最多。

1)LEA 取有效地址指令

格式：LEA　DST,SRC

执行操作：(DST)←(SRC)

该指令为双操作数指令，功能是将源操作数的有效地址传递给目的操作数。源操作数与目的操作数的寻址方式必须满足以下规定：

(1)源操作数不能使用立即数寻址和寄存器寻址方式。

(2)目的操作数必须使用寄存器寻址方式，且必须使用 16 位寄存器。

(3)目的操作数不能为段寄存器。

该指令不影响任何标志位。

【例 3-4】 若指令执行前(BX)＝0400H,(SI)＝003CH，则执行如下指令后(BX)的值为多少？

```
LEA   BX,[BX+SI+0F62H]
```

解　指令执行后 BX 寄存器的值应为源操作数的有效地址，即

(BX)＝0400H＋003CH＋0F62H＝139EH。

注意：这里(BX)得到的是该存储单元的有效地址而不是其内容。

2)LDS 传送偏移地址及数据段段地址指令

格式：LDS　DST,SRC

执行操作：(DST)←(SRC)

(DS)←((SRC)＋2)

该指令中的源操作数必须为双字存储器操作数，即相继 4 字节的存储单元，一般低字单元存放偏移地址，高字单元存放段地址。指令功能是将源操作数的低字单元传递给目的操作数，将高字单元传递给 DS 寄存器。源操作数与目的操作数的寻址方式必须满足的规定同 LEA 指令。

该指令不影响任何标志位。

【例 3-5】 若指令执行前(DS)＝1000H,(10010H)＝80H,(10011H)＝01H,(10012H)＝00H,(10013H)＝20H，则执行如下指令后(SI)和(DS)的值为多少？

LDS　SI,[0010H]

解　指令执行后 SI 寄存器的值应为源操作数的低字单元,DS 寄存器的内容应为源操作数的高字单元,即

(SI)=0180H,(DS)=2000H。

注意:在 LDS 指令中,目的操作数通常使用 SI 寄存器。

3)LES 传送偏移地址及附加段段地址指令

格式:LES　DST,SRC

执行操作:(DST)←(SRC)

(ES)←((SRC)+2)

该指令中的源操作数必须为双字存储器操作数,即相继 4 字节的存储单元,一般低字单元存放偏移地址,高字单元存放段地址。指令功能是将源操作数的低字单元传递给目的操作数,将高字单元传递给 ES 寄存器。源操作数与目的操作数的寻址方式必须满足的规定同 LEA 指令。

该指令不影响任何标志位。

【例 3-6】　若指令执行前(DS)=B000H,(BX)=080AH,(0B080AH)=05AEH,(0B080CH)=4000H,则执行如下指令后(DI)和(ES)的值为多少?

LES　DI,[BX]

解　指令执行后 DI 寄存器的值应为源操作数的低字单元,ES 寄存器的内容应为源操作数的高字单元,即

(DI)=05AEH,(ES)=4000H。

注意:在 LES 指令中,目的操作数通常使用 DI 寄存器。

3.1.6　输入/输出指令

8086/8088 CPU 由于采用独立编址方式,因此必须使用专门的输入/输出指令。这类指令利用累加器 AL 或 AX 和外设接口中的端口以字节或字为单位进行信息的传输。由于专门用到了累加器,所以又称累加器专用传送指令。共包括两条指令:IN 和 OUT。

1)IN 输入指令

格式:IN　AL,PORT　　长格式字节操作

IN　AX,PORT　　长格式字操作

IN　AL,DX　　短格式字节操作

IN　AX,DX　　短格式字操作

执行操作:(AL)←(PORT)　　长格式字节操作

(AX)←(PORT+1,PORT)　　长格式字操作

(AL)←((DX))　　短格式字节操作

(AX)←((DX)+1,(DX))　　短格式字操作

该指令为双操作数指令,其中目的操作数只能是累加器 AL 或 AX,源操作数为端口号。由于 8086/8088 CPU 外设最多可以有 65 536 个端口,端口号为 0000H～0FFFFH,因此源

操作数有两种格式：当端口号为 00H～0FFH 时(即前 256 个端口)，端口号以立即数的形式直接在指令中指定，这就是长格式中的 PORT；当端口号位于 0100H～0FFFFH 时，应先用 MOV 指令将端口号传送到 DX 寄存器中，再使用短格式访问端口。

该指令不影响任何标志位。

2)OUT 输出指令

```
格式：OUT   PORT,AL        长格式字节操作
      OUT   PORT,AX        长格式字操作
      OUT   DX,AL          短格式字节操作
      OUT   DX,AX          短格式字操作
执行操作：(AL)→(PORT)                    长格式字节操作
          (AX)→(PORT+1,PORT)             长格式字操作
          (AL)→((DX))                    短格式字节操作
          (AX)→((DX)+1,(DX))             短格式字操作
```

该指令为双操作数指令，其中源操作数只能是累加器 AL 或 AX，目的操作数为端口号。长格式和短格式的使用规定与 IN 指令相同。

该指令不影响任何标志位。

【例 3-7】 编写代码段，通过端口 10H 采集一个 16 位数据，并将该数据传递给端口 1000H。

解　代码如下：

```
MOV   DX,1000H
IN    AX,10H
OUT   DX,AX
```

注意：这里的 PORT 或 DX 的内容均为地址，而传送的是该地址所对应端口中的信息，而且短格式中 DX 的内容就是端口号本身，不需要再通过任何段寄存器来进行任何计算。另外这两条指令到底选用字操作还是字节操作，取决于外设端口的宽度。

3.2　算术运算指令

算术运算指令负责执行加、减、乘、除四则基本运算。它包括无符号数、有符号数的二进制算术运算和十进制算术运算调整指令，它们中有双操作数指令，也有单操作数指令。如前所述，双操作数指令的两个操作数除源操作数为立即寻址外，必须有至少一个操作数在寄存器中；单操作数指令不允许使用立即寻址方式。所有算术运算指令的寻址方式均遵循这一规则。

3.2.1　加法指令

加法指令用于实现两个操作数求和或单个操作数的自增。共包括 3 条指令：ADD、ADC 和 INC。

1)ADD 加法指令

格式:ADD　DST,SRC

执行操作:(DST)←(DST)+(SRC)

该指令为双操作数指令,功能为将源操作数与目的操作数相加,结果保存在目的操作数中。

该指令将影响所有条件码标志位。CF 标志位的置位规则为当最高有效位有向高位进位时(CF)=1,否则(CF)=0;OF 标志位的置位规则为若两个操作数的符号相同,而结果的符号与之相反时(OF)=1,否则(OF)=0。其他标志位设置比较简单,这里不再赘述。

【例 3-8】 若(CX)=7029H,(2000H)=94EDH,则执行如下指令后 CX 寄存器的值和各条件标志位的值为多少?

```
ADD  CX,[2000H]
```

解　7029H+94EDH

=0111000000101001B+1001010011101101B

=(1)0000010100010110B

=(1)0516H

则(CX)=0516H,(SF)=0,(ZF)=0,(AF)=1,(PF)=0,(CF)=1,(OF)=0。

注意:通常来说,当两个无符号数相加时,根据 CF 标志位是否为 1 判断结果是否溢出;当两个有符号数相加时,根据 OF 标志位是否为 1 判断结果是否溢出。

2)ADC 带进位加法指令

格式:ADC　DST,SRC

执行操作:(DST)←(DST)+(SRC)+(CF)

该指令为双操作数指令,功能是将源操作数与目的操作数相加,再加上 CF 标志位的值,结果保存在目的操作数中。该指令主要和 ADD 指令配合使用,用于计算超过 CPU 字长的两个操作数的加法和。

和 ADD 指令一样,该指令将影响所有条件码标志位。各标志位置位规则也与 ADD 指令相同。

【例 3-9】 编写代码段,实现两个双精度数的加法。假设目的操作数存放在 DX 和 AX 寄存器中,其中 DX 中为高位字;源操作数存放在 BX 和 CX 寄存器中,其中 BX 中为高位字。指令执行前:

(DX)=0002H　　(AX)=0F365H

(BX)=0005H　　(CX)=0E024H

解

```
ADD  AX,CX
ADC  DX,BX
```

第一条指令执行后:

(AX)=0D389H,(SF)=1,(ZF)=0,(CF)=1,(OF)=0。

第二条指令执行后:

(DX)=0008H,(SF)=0,(ZF)=0,(CF)=0,(OF)=0。

提示:为实现双精度加法,必须用两条指令分别完成低位字和高位字的加法。一般先用

ADD 指令求低位字之和，再用 ADC 指令把前一条 ADD 指令所产生的进位加入到高位字的求和运算中。另外，带符号的双精度数的溢出应根据 ADC 指令执行后 OF 标志位的值来判断，低位 ADD 指令所产生的溢出是无意义的。

3)INC 加 1 指令

格式：INC　OPR

执行操作：(OPR)←(OPR)＋1

该指令为单操作数指令，功能为操作数自增 1，主要用于修改地址指针。

该指令将影响除 CF 标志位之外的所有条件码标志位，各标志位置位规则也与 ADD 指令相同。

注意：作为单操作数指令，操作数 OPR 在使用直接寻址、寄存器间接寻址、寄存器相对寻址、基址变址寻址和相对基址变址寻址 5 种寻址方式时，会出现操作数类型不明确的现象，此时应用 WORD PTR 或 BYTE PTR 指定其类型。后面的 DEC 和 NEG 也是如此。

3.2.2　减法指令

减法指令用于实现两个操作数求差或比较，以及单个操作数的自减或求补。共包括 5 条指令：SUB、SBB、DEC、NEG 和 CMP。

1)SUB 减法指令

格式：SUB　DST，SRC

执行操作：(DST)←(DST)－(SRC)

该指令为双操作数指令，功能是用目的操作数减去源操作数，结果保存在目的操作数中。

该指令将影响所有条件码标志位。CF 标志位的置位规则为当最高有效位有向高位借位时(CF)＝1，否则(CF)＝0；OF 标志位的置位规则为若两个操作数的符号相反，且结果的符号与减数相同时(OF)＝1，否则(OF)＝0。其他标志位设置比较简单，这里不再赘述。

【例 3-10】 若(DH)＝41H，(SS)＝0000H，(BP)＝00E4H，(000E8H)＝5AH，则执行如下指令后 DH 寄存器的值和各条件标志位的值为多少？

SUB　DH，[BP＋4]

解　(DH)＝41H－5AH＝0E7H

则(SF)＝1，(ZF)＝0，(AF)＝1，(CF)＝1，(OF)＝0。

注意：通常来说，当两个无符号数相减时，根据 CF 标志位是否为 1 判断结果是否溢出；当两个有符号数相减时，根据 OF 标志位是否为 1 判断结果是否溢出。

2)SBB 带借位减法指令

格式：SBB　DST，SRC

执行操作：(DST)←(DST)－(SRC)－(CF)

该指令为双操作数指令，功能是用目的操作数减去源操作数，再减去 CF 标志位的值，结果保存在目的操作数中。该指令主要和 SUB 指令配合，用于计算超过 CPU 字长的两个操

作数的减法。

和SUB指令一样，该指令将影响所有条件码标志位。各标志位置位规则也与SUB指令相同。

【例3-11】 编写代码段，实现两个双精度数的减法。假设目的操作数存放在DX和AX寄存器中，其中DX中为高位字；源操作数存放在BX和CX寄存器中，其中BX中为高位字。指令执行前：

(DX)＝7A45H　(AX)＝3BEFH

(BX)＝5BD9H　(CX)＝0C689H

解

```
SUB  AX,CX
SBB  DX,BX
```

第一条指令执行后：

(AX)＝7566H，(SF)＝0，(ZF)＝0，(CF)＝1，(OF)＝0。

第二条指令执行后：

(DX)＝1E6BH，(SF)＝0，(ZF)＝0，(CF)＝0，(OF)＝0。

提示：为实现双精度减法，必须用两条指令分别完成低位字和高位字的减法。一般先用SUB指令求低位字之差，再用SBB指令把前一条SUB指令所产生的借位考虑到高位字的求差运算中。

3)DEC减1指令

格式：DEC　OPR

执行操作：(OPR)←(OPR)－1

该指令为单操作数指令，功能为操作数自减1。该指令主要用于修改地址指针。

该指令将影响除CF标志位之外的所有条件码标志位。各标志位置位规则也与SUB指令相同。

4)NEG求补指令

格式：NEG　OPR

执行操作：(OPR)←－(OPR)

该指令为单操作数指令，功能是将操作数按位取反后，末位加1，也可理解为100H－(OPR)(字节操作)或10000H－(OPR)(字操作)，因此也划入减法指令中。该指令常用于求操作数的相反数。

该指令将影响所有条件码标志位。其中只有操作数为0时，执行该指令后，(CF)＝0，其他情况下，(CF)＝1；只有操作数为－128(字节操作)或－32 768(字操作)时，执行该指令后，(OF)＝1，其他情况下，(OF)＝0。

5)CMP比较指令

格式：CMP　OPR1，OPR2

执行操作：(OPR1)－(OPR2)

该指令为双操作数指令，功能是用目的操作数减去源操作数，但不保存结果，只是根据结果去影响标志位。

该指令将影响所有条件码标志位。

该指令通常用于比较两个操作数的大小。指令执行后，能根据标志位的设置情况来判断两个数的大小关系：

(1)若无符号数比较大小：当(ZF)＝1 时，OPR1＝OPR2；否则，当(CF)＝0 时，OPR1＞OPR2；当(CF)＝1 时，OPR1＜OPR2。

(2)若有符号数比较大小：当(ZF)＝1 时，OPR1＝OPR2；否则，当(OF)＝(SF)时，OPR1＞OPR2；当(OF)≠(SF)时，OPR1＜OPR2。

3.2.3　乘法指令

乘法指令用于实现两个二进制操作数的乘法运算。由于操作数有无符号数和有符号数两种，所以共提供两条指令：MUL 和 IMUL。

1)MUL 无符号数乘法指令

格式：MUL　SRC

执行操作：(AX)←(AL)×(SRC)　　字节操作

　　(DX,AX)←(AX)×(SRC)　　字操作

该指令为单操作数指令，其中目的操作数隐含，两个操作数及结果均为无符号数。当为字节操作时，用(AL)乘以(SRC)，将得到的字乘积送入(AX)中；当为字操作时，用(AX)乘以(SRC)，将得到的双字乘积的高位字送入(DX)中，低位字送入(AX)中。

该指令对除 CF 和 OF 以外的条件码标志位无定义。如果乘积的高一半为 0，即字节操作的(AH)或字操作的(DX)为 0，则(CF)和(OF)均为 0；否则，(CF)和(OF)均为 1。因此可通过标志位判断字节乘法的结果是字还是字节，或字乘法的结果是字还是双字。

注意：对条件码标志位无定义与不影响条件码是不同的。无定义是指指令执行后，条件码位的值不确定；不影响则是指指令执行后，条件码位的值保持原值不变化。

【例 3-12】 若(AL)＝0B4H，(BL)＝11H，执行如下指令，乘积为多少？

MUL　BL

解　(AL)＝0B4H，即无符号数 180D；(BL)＝11H，即无符号数 17D；

执行指令后：(AX)＝(AL)×(BL)＝0BF4H＝3060D；

(CF)＝(OF)＝1。

2)IMUL 有符号数乘法指令

格式：IMUL　SRC

执行操作：(AX)←(AL)×(SRC)　　字节操作

　　(DX,AX)←(AX)×(SRC)　　字操作

该指令为单操作数指令，其中目的操作数隐含，两个操作数及结果均为有符号数。当为字节操作时，用(AL)乘以(SRC)，将得到的字乘积送入(AX)中；当为字操作时，用(AX)乘以(SRC)，将得到的双字乘积的高位字送入(DX)中，低位字送入(AX)中。

该指令对除 CF 和 OF 以外的条件码标志位无定义。如果乘积的高一半为低一半的符号扩展，则(CF)和(OF)均为 0；否则，(CF)和(OF)均为 1。

【例 3-13】 若(AL)=0B4H,(BL)=11H,执行如下指令后乘积为多少?

```
IMUL   BL
```

解 (AL)=0B4H,即有符号数-76D;(BL)=11H,即有符号数 17D;

执行指令后:(AX)=(AL)*(BL)=0FAF4H=-1292D;

(CF)=(OF)=1。

注意:由【例 3-12】和【例 3-13】可以看出,同样的二进制串,作为无符号数做乘法和作为有符号数做乘法的结果是完全不同的。

3.2.4 除法指令

除法指令用于实现两个二进制操作数的除法运算,由于操作数有无符号数和有符号数两种,所以共提供两条指令:DIV 和 IDIV。

1)DIV 无符号数除法指令

格式:DIV SRC

执行操作:字节操作

(AL)←(AX)/(SRC)的商

(AH)←(AX)/(SRC)的余数

字操作

(AX)←(DX,AX)/(SRC)的商

(DX)←(DX,AX)/(SRC)的余数

该指令为单操作数指令,其中目的操作数隐含,两个操作数及结果均为无符号数。当为字节操作时,以 16 位的(AX)为被除数,8 位的(SRC)为除数,除法运算后,将得到的 8 位商送入(AL)中,8 位余数送入(AH)中;当为字操作时,以 32 位的(DX,AX)为被除数,16 位的(SRC)为除数,除法运算后,将得到的 16 位商送入(AX)中,16 位余数送入(DX)中。

该指令执行后,标志寄存器中各标志位不确定,但商可产生溢出。一般来说,当被除数的高一半绝对值大于除数绝对值时,商就会产生溢出。此溢出会产生 0 号中断(除法错误中断),转入除法错误中断处理,又称除法溢出。

【例 3-14】 若(AX)=0062H,(BL)=04H,执行如下指令后商和余数为多少?

```
DIV   BL
```

解 (AX)=0062H,即无符号数 98D;(BL)=04H,即无符号数 04D;

执行指令后:(AL)=(AX)/(BL)的商=18H=24D;

(AH)=(AX)/(BL)的余数=02H=2D。

2)IDIV 有符号数除法指令

格式:DIV SRC

执行操作:字节操作

(AL)←(AX)/(SRC)的商

(AH)←(AX)/(SRC)的余数

字操作

(AX)←(DX,AX)/(SRC)的商

(DX)←(DX,AX)/(SRC)的余数

该指令为单操作数指令,其中目的操作数隐含,两个操作数及结果均为有符号数,且余数的符号和被除数的符号必须相同。当为字节操作时,以 16 位的(AX)为被除数,8 位的(SRC)为除数,除法运算后,将得到的 8 位商送入(AL)中,8 位余数送入(AH)中;当为字操作时,以 32 位的(DX,AX)为被除数,16 位的(SRC)为除数,除法运算后,将得到的 16 位商送入(AX)中,16 位余数送入(DX)中。

该指令执行后,标志寄存器中各标志位不确定,但商可产生溢出。一般来说,当被除数的高一半绝对值大于除数绝对值时,商就会产生溢出。此溢出会产生 0 号中断(除法错误中断),转入除法错误中断处理,又称除法溢出。

【例 3-15】 若(AX)=0FFEBH,(BL)=05H,执行如下指令后商和余数为多少?

```
IDIV   BL
```

解 (AX)=0FFEBH,即有符号数-66D;(BL)=05H,即有符号数 05D;

执行指令后:(AL)=(AX)/(BL)的商=0F3H=-12D;

(AH)=(AX)/(BL)的余数=0FFH=-1D。

3.2.5　符号扩展指令

在算术运算指令中,对参加运算的数据长度往往有一定的要求,因此需要通过符号扩展指令来获得算术运算指令所需要的数据格式。例如,加法、减法和乘法指令都要求两个操作数位数一致,除法指令字节操作时要求被除数为 16 位,字操作时要求被除数为 32 位等。该类指令共包括两条:CBW 和 CWD。

1)CBW 字节扩展为字指令

格式:CBW

执行操作:将 AL 内容的符号位扩展到 AH,形成 AX 中的一个字。

若(AL)最高有效位为 0,则(AH)=00H;

若(AL)最高有效位为 1,则(AH)=0FFH。

该指令操作数隐含为 AL,在对某操作数进行符号扩展前,必须先把该操作数送入 AL 寄存器中。符号扩展不会改变有符号数的大小。

该指令对所有的条件码标志位均不影响。

2)CWD 字扩展为双字指令

格式:CWD

执行操作:将 AX 的内容符号扩展到 DX,形成由 DX,AX 构成的一个双字。

若(AX)最高有效位为 0,则(DX)=0000H;

若(AX)最高有效位为 1,则(DX)=0FFFFH。

该指令操作数隐含为 AX,在对某操作数进行符号扩展前,必须先把该操作数送入 AX 寄存器中。符号扩展不会改变有符号数的大小。

该指令对所有的条件码标志位均不影响。

【例 3-16】 算术运算综合举例，试编写指令序列计算：

(V－(X×Y＋Z－360))/W

其中，X、Y、Z、V、W 均为 16 位符号数，分别已装入 X、Y、Z、V、W 单元中，要求结果商存入 AX 寄存器，余数存入 DX 寄存器。

解 指令序列如下：

```
MOV  AX,X
IMUL  Y          ;求解 X×Y,积在(DX,AX)中
MOV  CX,AX
MOV  BX,DX       ;复制积值至(BX,CX)中
MOV  AX,Z
CWD              ;将 Z 扩展为(DX,AX)中的 32 位双字
ADD  CX,AX
ADC  BX,DX       ;将 Z 与乘积双字求和
SUB  CX,360D
SBB  BX,0        ;和值减去 360 的差在(BX,CX)中
MOV  AX,V
CWD              ;将 V 扩展为(DX,AX)中的 32 位双字
SUB  AX,CX
SBB  DX,BX       ;V 减去(BX,CX)中的差值
IDIV   W         ;除以 W,商存入 AX 寄存器,余数存入 DX 寄存器
```

3.2.6 十进制调整指令

前面提到的所有算术运算指令都是二进制运算指令，但最常用的是十进制数，这意味着在计算前必须先把十进制数转化为二进制数，计算完毕后再将二进制数转化为十进制数输出。为了便于直接对计算机中以 BCD 码表示的十进制数进行计算，8086/8088 CPU 提供了一组十进制调整指令，又称 BCD 码调整指令，共包括 6 条指令：DAA、DAS、AAA、AAS、AAM 和 AAD。

1)BCD 码

在说明这组指令前，进一步介绍 BCD 码(之前的介绍见 1.4.2)。

BCD(binary-coded decimal)码是一种二进制的数字编码形式，是用二进制编码的十进制数，又称二进码十进数。它利用了 4 个二进制位来储存一个十进制的数码，由于常用的 BCD 码从左到右每一位的 1 分别表示 8、4、2、1，所以这种代码又称 8421BCD 码。十进制数码对应的 8421BCD 码如图 3-4 所示。

十进制数码	0	1	2	3	4	5	6	7	8	9
8421BCD 码	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

图 3-4　十进制数码与 8421BCD 码对应表

在 IBM-PC 中，表示十进制数的 BCD 码有压缩 BCD 码和非压缩 BCD 码两种格式。所谓压缩 BCD 码格式(packed BCD format)，是指用 4 位二进制数表示一位 BCD 码，用 1 字节表示两位 BCD 码，如十进制数 56 的压缩 BCD 码是 0101 0110B。非压缩型 BCD 码格式(unpacked BCD format)是指 1 字节只存放 1 位十进制数，其中高 4 位的内容不做规定(也有部分书籍要求为 0，二者均可)，低 4 位二进制数表示该位十进制数，如十进制数 56 的非压缩型 BCD 码是 00000101 00000110B，必须存放在一个字中。可以看出数字的 ASCII 码是一种典型的非压缩型 BCD 码，其高 4 位均为 0011，而低 4 位是用 8421BCD 表示的十进制数。

2)DAA 压缩 BCD 码加法调整指令

格式：DAA

执行功能：将(AL)中的和值调整为压缩 BCD 码并送回(AL)中。指令执行前必须先执行 ADD 或 ADC 指令，将两个压缩 BCD 码相加，且和值保存在(AL)中。

调整方法：若(AL)的低 4 位大于 9 或(AF)＝1，则(AL)←(AL)＋06H，并使(AF)＝1；若(AL)的高 4 位大于 9 或(CF)＝1，则(AL)←(AL)＋60H，并使(CF)＝1；其余情况下(AL)的值不变。

该指令对 OF 标志位无定义，但影响其他条件码标志位。

【例 3-17】 若(BCD1)＝1834D，(BCD2)＝2789D，均为压缩 BCD 码，编写指令序列执行(BCD3)←(BCD1)＋(BCD2)。

解　由于 BCD1 和 BCD2 均为压缩 BCD 码，且都是 4 位十进制数，所以每个数占 2 字节，其存放方式为：

(BCD1)＝34　(BCD1＋1)＝18

(BCD2)＝89　(BCD2＋1)＝27

指令序列如下：

```
MOV   AL,BCD1
ADD   AL,BCD2
DAA
MOV   DCB3,AL
MOV   AL,BCD1＋1
ADC   AL,BCD2＋1
DAA
MOV   DCB3＋1,AL
```

第一组的 4 条指令完成低位字节加法并经调整后送入 BCD3，其中 ADD 指令执行后(AL)＝34＋89＝BDH，(CF)＝0，(AF)＝0，经 DAA 调整后(AL)＝23，(CF)＝1，(AF)＝1；第二组的 4 条指令完成高位字节加法并经调整后送入 BCD3＋1，其中 ADC 指令执行后

(AL)=18+27+(CF)=40H,(CF)=0,(AF)=1,经 DAA 调整后(AL)=46,(CF)=0,(AF)=1;最后结果(BCD3)=4623,结果正确无误。

3)DAS 压缩 BCD 码减法调整指令

格式:DAS

执行功能:将(AL)中的差值调整为压缩 BCD 码并送回(AL)中。指令执行前必须先执行 SUB 或 SBB 指令,将两个压缩 BCD 码相减,且差值保存在(AL)中。

调整方法:若(AL)的低 4 位大于 9 或(AF)=1,则(AL)←(AL)-06H,并使(AF)=1;若(AL)的高 4 位大于 9 或(CF)=1,则(AL)←(AL)-60H,并使(CF)=1;其余情况下(AL)的值不变。

该指令对 OF 标志位无定义,但影响其他条件码标志位。

【例 3-18】 若(BCD1)=1234D,(BCD2)=4612D,均为压缩 BCD 码,编写指令序列执行(BCD3)←(BCD1)-(BCD2)。

解 由于 BCD1 和 BCD2 均为压缩 BCD 码,且都是 4 位十进制数,所以每个数占 2 字节,其存放方式为:

(BCD1)=34　(BCD1+1)=12

(BCD2)=12　(BCD2+1)=46

指令序列如下:

```
MOV  AL,BCD1
SUB  AL,BCD2
DAS
MOV  DCB3,AL
MOV  AL,BCD1+1
SBB  AL,BCD2+1
DAS
MOV  DCB3+1,AL
```

第一组的 4 条指令完成低位字节减法并经调整后送入 BCD3,其中 SBB 指令执行后(AL)=34-12=22H,(CF)=0,(AF)=0,经 DAS 不做任何调整;第二组的 4 条指令完成高位字节减法并经调整后送入 BCD3+1,其中 SBB 指令执行后(AL)=12-46-(CF)=0CCH,(CF)=1,(AF)=1,经 DAS 调整后(AL)=66,(CF)=1,(AF)=1;最后结果(BCD3)=6622,为十进制数-3378 的补码,结果正确无误。

4)AAA 非压缩 BCD 码加法调整指令

格式:AAA

执行功能:将(AL)中的和值调整为非压缩 BCD 码并送回(AL)中。指令执行前必须先执行 ADD 或 ADC 指令,将两个非压缩 BCD 码相加,且和值保存在(AL)中。

调整方法:若(AL)的低 4 位大于 9 或(AF)=1,则(AL)←(AL)+06H,(AH)←(AH)+1,并使(AF)=(CF)=1,且(AL)高 4 位清 0;否则,(AF)=(CF)=0,且(AL)高 4 位清 0。

该指令对除 CF 和 AF 之外的其他条件码标志位无定义。

【例 3-19】 若(AX)＝0535H,(BL)＝39H,分析下列指令序列的执行过程：

```
ADD   AL,BL
AAA
```

解 由题可知,AL 寄存器和 BL 寄存器中分别为 5 和 9 的 ASCII 码,第一条指令执行完后,(AL)＝6EH,(AF)＝0;第二条指令进行调整后使得(AX)＝0604H,(AF)＝(CF)＝1。

5)AAS 非压缩 BCD 码减法调整指令

格式:AAS

执行功能:将(AL)中的差值调整为非压缩 BCD 码并送回(AL)中。指令执行前必须先执行 SUB 或 SBB 指令,将两个非压缩 BCD 码相减,且差值保存在(AL)中。

调整方法:若 AL 的低 4 位大于 9 或(AF)＝1,则(AL)←(AL)－06H,(AH)←(AH)－1,并使(AF)＝(CF)＝1,且 AL 高 4 位清 0;否则,(AF)＝(CF)＝0,且 AL 高 4 位清 0。

该指令对除 CF 和 AF 之外的其他条件码标志位无定义。

【例 3-20】 编写程序段实现下式：

(DX)←UP1＋UP2－UP3

其中,参加运算的数均为两位十进制数。如要求计算 25＋48－19,每个十进制数以非压缩 BCD 格式存入存储器,每个数占有一个字,所以(UP1)＝0205H,(UP2)＝0408H,(UP3)＝0109H。

解 指令序列如下：

```
MOV   AX,0          ;初始化
MOV   AL,UP1
ADD   AL,UP2        ;低位求和
AAA                 ;非压缩 BCD 码加法调整
MOV   DL,AL
MOV   AL,UP1＋1
ADC   AL,UP2＋1     ;高位求和
AAA                 ;非压缩 BCD 码加法调整
XCHG  AL,DL
SUB   AL,UP3        ;低位求差
AAS                 ;非压缩 BCD 码减法调整
XCHG  AL,DL
SBB   AL,UP3＋1     ;高位求差
AAS                 ;非压缩 BCD 码减法调整
MOV   DH,AL
```

6)AAM 非压缩 BCD 码乘法调整指令

格式:AAM

执行功能:将(AX)中的积值调整为非压缩 BCD 码并送回(AX)中。指令执行前必须先执行 MUL 指令,将两个非压缩 BCD 码相乘(此时要求其高 4 位为 0),且积值保存在(AX)中。

调整方法:把 AL 寄存器的内容除以 0AH,商放在 AH 寄存器中,余数保存在 AL 寄存器中。

该指令根据 AL 寄存器的内容设置 SF、ZF 和 PF 标志位,对 CF、AF 和 OF 标志位无定义。

【例 3-21】 若(AL)=07H,(BL)=08H,分析下列指令序列的执行过程:

```
MUL    BL
AAM
```

解 由题可知,AL 寄存器和 BL 寄存器中分别为 7 和 8 的 ASCII 码,第一条指令执行完后,(AL)=38H;第二条指令进行调整后使得(AH)=05H,(AL)=06H。

该指令的另一个用途是将 0000H 到 0063H 的二进制数转换为对应的非压缩 BCD 码。如执行前 AX 寄存器的值为 0060H,AAM 指令执行后它的内容将是 0906H,这是非压缩的等效十进制数 96。如果再加上 3030H,则可得到其 ASCII 码 3936H。

7)AAD 非压缩 BCD 码除法调整指令

格式:AAD

执行功能:除法运算前,先对被除数 AX 的内容进行调整。

调整方法:(AL)←(AH)×0AH+(AL),(AH)←0。

和前面介绍的调整指令不同,AAD 指令是在除法操作前进行调整的。它是针对以下情况而设定的:若被除数是存放在 AX 中的两位非压缩 BCD 数,其中 AH 中为十位,AL 中为个位,且 AH 和 AL 的高 4 位均为 0;除数是一位非压缩 BCD 数,同样要求高 4 位为 0,则这两个数用 DIV 指令相除前,必须先用 AAD 指令将 AX 中的被除数调整为二进制数,放在 AL 寄存器中。

该指令根据 AL 寄存器的内容设置 SF、ZF 和 PF 标志位,对 CF、AF 和 OF 标志位无定义。

【例 3-22】 若(AX)=0605H,分析下列指令序列的执行后的变化:

```
AAD
```

解 执行后(AX)=0041H。

3.3 位操作指令

前面介绍的指令都是以字或字节为单位,将其整体作为一个数据进行运算。然而在某些情况下,需要将某个字或字节中的每个二进制位看成一个独立的部分来进行处理,这种方式称为按位操作。相应地,8086/8088 CPU 提供了一组按位进行操作的位操作指令,包括进行逻辑运算的逻辑运算指令和用于移位处理的移位指令两大类。

3.3.1 逻辑运算指令

逻辑运算指令可以对字或字节执行逻辑运算。由于逻辑运算是按位操作的,因此其操

作数应看做是二进制位串而不是数。逻辑运算指令共包括 5 条：AND、OR、XOR、TEST 和 NOT。它们中有双操作数指令，也有单操作数指令。如前所述，双操作数指令的两个操作数除源操作数为立即寻址外，必须有至少一个操作数在寄存器中；单操作数指令不允许使用立即寻址方式。所有逻辑运算指令的寻址方式均遵循这一规则。

1）AND 逻辑与指令

格式：AND　DST，SRC

执行操作：(DST)←(DST)∧(SRC)

该指令为双操作数指令，功能是将目的操作数与源操作数对位相与，结果保存在目的操作数中。该指令主要用于以下两种情况：一是使一个操作数的若干位保持不变（和 1 与），其余位清 0（和 0 与）；二是将操作数与自身相与，操作数不变，但将 CF 标志位清 0。

该指令执行后，CF 和 OF 标志位为 0，AF 标志位不确定，SF、ZF 和 PF 标志位根据结果设置。

【例 3-23】 若(AL)＝0BDH，试用一条 AND 指令将(AL)的第 0、2、5 位清 0。

解　指令执行前：(AL)＝0BDH＝10111101B。

指令为：AND　AL，11011010B。

指令执行后：(AL)＝10011000B＝98H。

2）OR 逻辑或指令

格式：OR　DST，SRC

执行操作：(DST)←(DST)∨(SRC)

该指令为双操作数指令，功能是将目的操作数与源操作数对位相或，结果保存在目的操作数中。该指令主要用于以下两种情况：一是使一个操作数的若干位保持不变（和 0 或），其余位置 1（和 1 或）；二是将操作数与自身相或，操作数不变，但将 CF 标志位清 0。

该指令执行后，CF 和 OF 标志位为 0，AF 标志位不确定，SF、ZF 和 PF 标志位根据结果设置。

【例 3-24】 若(AL)＝94H，试用一条 OR 指令将(AL)的第 1、3、6 位置 1。

解　指令执行前：(AL)＝94H＝10010100B。

指令为：OR　AL，01001010B。

指令执行后：(AL)＝11011110B＝0DEH。

3）NOT 逻辑非指令

格式：NOT　OPR

执行操作：(OPR)←($\overline{\text{OPR}}$)

该指令为单操作数指令，功能是将操作数各位按位取反，结果仍然保存在该操作数中。

该指令不影响条件标志位。

4）XOR 逻辑异或指令

格式：XOR　DST，SRC

执行操作：(DST)←(DST)⊕(SRC)

该指令为双操作数指令，功能是将目的操作数与源操作数对位相异或，结果保存在目的操作数中。该指令主要用于以下 3 种情况：一是使一个操作数的若干位保持不变（和 0 异

或),其余位取反(和1异或);二是将操作数与自身异或,操作数清0,CF标志位也清0,常用于使操作数初值置0;三是用于测试某一操作数是否与另一确定的操作数相等(若两数相等,异或后(ZF)=1;否则,(ZF)=0),这种操作在检查地址是否匹配时是最常用的。

该指令执行后,CF和OF标志位为0,AF标志位不确定,SF、ZF和PF标志位根据结果设置。

【例3-25】 若(AL)=39H,试用一条XOR指令将(AL)的低4位取反。

解 指令执行前:(AL)=39H=00111001B。

指令为:XOR AL,00001111B。

指令执行后:(AL)=00110110B=36H。

5)TEST逻辑测试指令

格式:TEST DST,SRC

执行操作:(DST)∧(SRC)

该指令为双操作数指令,功能是将目的操作数与源操作数对位相与,结果不保存。该指令功能与AND指令类似,但执行完后不会改变目的操作数的值。该指令常用于在不希望改变原有操作数的前提下,检测操作数的某一位或某几位是否满足条件。具体编程时,常与条件转移指令连用。

该指令执行后,CF和OF标志位为0,AF标志位不确定,SF、ZF和PF标志位根据结果设置。

【例3-26】 若(AX)为一个带符号数,求(AX)的绝对值。

解 求一个数的绝对值,应该首先测试该数的正负,即该数最高有效位的值。若最高有效位为1,则是负数,绝对值为该数求补;若最高有效位为0,则是正数,绝对值为该数本身。指令如下:

```
TEST  AX,8000H
JZ  NEXT            ;测试AX最高位,为0该数为正,转到NEXT
NEG  AX             ;否则,该数为负,求补
NEXT:…
```

3.3.2 移位指令

移位指令包括逻辑移位指令、算术移位指令和循环移位指令。所有这些指令都对目的操作数按指令规定的方向和方式向左或向右移动指定位数。所有移位指令中的第一个操作数,不允许使用立即寻址方式,也不能使用段寄存器。

1)SHL逻辑左移指令

格式:SHL OPR,COUNT

执行操作:将操作数(OPR)向左移动COUNT指定的位数,每左移1位,最低位补0,最高位移出送至CF标志位。每逻辑左移1位,相当于无符号数乘以2一次。

该指令的操作数(OPR)在使用直接寻址、寄存器间接寻址、寄存器相对寻址、基址变址寻址和相对基址变址寻址5种寻址方式时,会出现操作数类型不明确的现象。此时应用WORD PTR或BYTE PTR指定其类型。COUNT是移位的次数,8086/8088规定:若COUNT=1,则在指令中直接写上1代替COUNT;若COUNT>1,则要用MOV指令先将

移位次数送入 CL 寄存器，然后在移位指令中用 CL 代替 COUNT。

该指令执行后，AF 标志位无定义；SF、ZF、PF 和 CF 标志位根据移位后的结果设置；OF 只有在 COUNT＝1 时有效，否则该位也无定义。当 COUNT＝1 时，若移位后符号位发生了变化，则(OF)＝1，否则(OF)＝0。

2)SHR 逻辑右移指令

格式：SHR　OPR，COUNT

执行操作：将操作数(OPR)向右移动 COUNT 指定的位数，每右移 1 位最高位补 0，最低位移出送至 CF 标志位。每逻辑右移 1 位，相当于无符号数除以 2 一次。

该指令对操作数的要求及对标志位的影响同 SHL 指令。

3)SAL 算术左移指令

格式：SAL　OPR，COUNT

执行操作：将操作数(OPR)向左移动 COUNT 指定的位数，每左移 1 位，最低位补 0，最高位移出送至 CF 标志位。每逻辑左移 1 位，相当于有符号数乘以 2 一次。该指令在物理上与 SHL 指令完全一致。

该指令对操作数的要求及对标志位的影响同 SHL 指令。

4)SAR 算术右移指令

格式：SAR　OPR，COUNT

执行操作：将操作数(OPR)向右移动 COUNT 指定的位数，每右移 1 位，最高位保持不变(即用原符号位填充)，最低位移出送至 CF 标志位。每逻辑右移 1 位，相当于有符号数除以 2 一次。

该指令对操作数的要求及对标志位的影响同 SHL 指令。

5)ROL 循环左移指令

格式：ROL　OPR，COUNT

执行操作：将操作数(OPR)向左循环移动 COUNT 指定的位数，每左移 1 位，左移前的最高位移出送至最低位及 CF 标志位。

该指令对操作数的要求同 SHL 指令。

该指令执行后，当 COUNT＝1 时，若循环移位后结果的最高位不等于 CF 标志位的值，则(OF)＝1，否则(OF)＝0。这用来说明循环移位前后的符号位是否改变。CF 与 OF 标志位根据移位结果设置，其他标志位无影响。

6)ROR 循环右移指令

格式：ROR　OPR，COUNT

执行操作：将操作数(OPR)向右循环移动 COUNT 指定的位数，每右移 1 位，右移前的最低位移出送至最高位及 CF 标志位。

该指令对操作数的要求同 SHL 指令，对标志位的影响同 ROL 指令。

7)RCL 带进位循环左移指令

格式：RCL　OPR，COUNT

执行操作：将操作数(OPR)连同 CF 标志位一起向左循环移动 COUNT 指定的位数，每

左移 1 位，左移前 CF 标志位的值移出送至最低位，左移前的最高位移出送至 CF 标志位。

该指令对操作数的要求同 SHL 指令，对标志位的影响同 ROL 指令。

8）RCR 带进位循环右移指令

格式：RCR　OPR，COUNT

执行操作：将操作数（OPR）连同 CF 标志位一起向右循环移动 COUNT 指定的位数，每右移 1 位，右移前 CF 标志位的值移出送至最高位，右移前的最低位移出送至 CF 标志位。

该指令对操作数的要求同 SHL 指令，对标志位的影响同 ROL 指令。

各种移位指令的操作如图 3-5 所示。

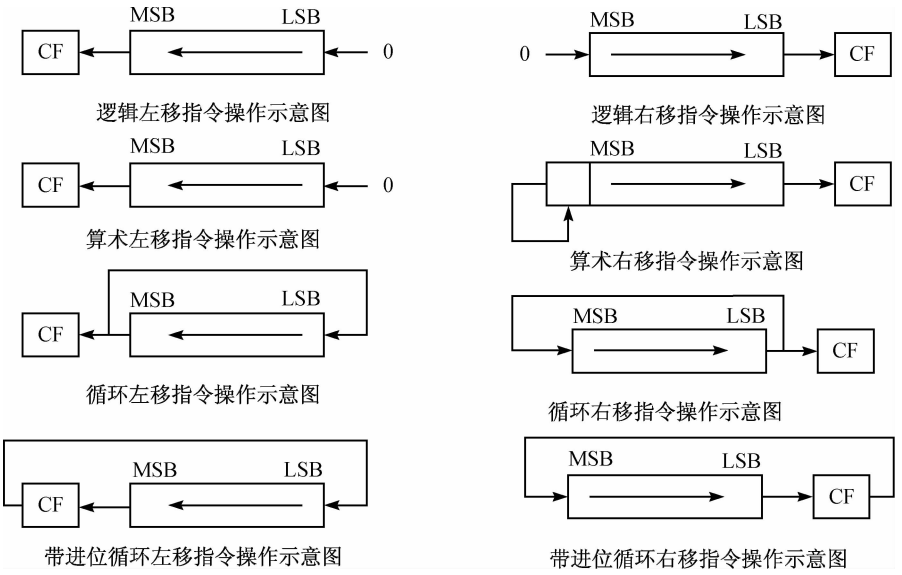


图 3-5　各种移位指令操作图解

3.4　字符串操作

字符串是字符的一个序列，串操作就是对这一序列字或字节进行某种相同的操作。这一序列字或字节存放的内容可能是字符或数据，它们也是一种普通的数据类型。对字符串的操作处理一般包括复制、检索、插入、删除和替换等。为了便于对字符串进行有效的处理，8086/8088 提供了专门用于处理字符串的指令，称为字符串操作指令，简称串操作指令。

3.4.1　串操作指令简介

串操作指令不能使用第 2 章中所讲的寻址方式，它有自己独特的寻址方式。串操作指

令中的源操作数地址由 DS:SI 提供,目的操作数地址由 ES:DI 提供。每条串操作指令执行时会自动调整作为指针使用的寄存器 SI 或 DI 中的值,且每次仅对串中一个字或字节进行操作。如串操作的单元是字节,则调整值为 1;如串操作的单元是字,则调整值为 2。此外,字符串操作的方向(处理字符串中单元的次序)由标志寄存器中的方向标志(DF)控制。当(DF)复位为 0 时,按递增方式调整寄存器 SI 或 DI 中的值;当(DF)置位为 1 时,按递减方式调整寄存器 SI 或 DI 中的值。其具体规定如下:

(1)当(DF)=0 时,变址寄存器 SI(和 DI)的内容增加 1、2 或 4。

(2)当(DF)=1 时,变址寄存器 SI(和 DI)的内容减小 1、2 或 4。

3.4.2　串操作指令

为了处理连续存储单元中的字符串或数串,地址指针需要连续地增量或减量,指针增量或减量决定了串处理的方向。设置方向标志的指令是 CLD 和 STD,则有:

```
CLD    ;DF 置 0(clear direction flag)
STD    ;DF 置 1(set direction flag)
```

当用 CLD 指令使(DF)=0 时,源串的指针 SI 和目的串的指针 DI 自动增量(+1 或+2);当用 STD 指令使(DF)=1 时,指针 SI 和 DI 自动减量(−1 或−2)。地址指针是±1 还是±2,取决于串操作数是字节还是字。处理字节串时,地址指针每次+1 或−1;处理字串时,地址指针每次+2 或−2。

8088 指令系统中共设计有 5 条串操作指令,分别用于完成从串中取出数据、向串中存入数据、串复制、串比较、串扫描等操作,下面依次具体介绍。

1)LODS 指令——从串中取出数据

按照串中存放的是字节型数据还是字型数据,有两条指令分别用于从串中取出一个元素。

格式:LODSB 或 LODSW

功能:

(1)LODSB 进行字节型串操作,从内存中 DS:SI 所确定的逻辑地址处取出一个字节的数据,送到(AL)中。当(DF)=0 时,令(SI)←(SI)+1;当(DF)=1 时,令(SI)←(SI)−1。

(2)LODSW 进行字型串操作,从内存中 DS:SI 所确定的逻辑地址处取出一个字型数据,送到(AX)中。当(DF)=0 时,令(SI)←(SI)+2;当(DF)=1 时,令(SI)←(SI)−2。

这里把 LODSB 指令和 LODSW 指令统称为 LODS 指令,以下各串操作指令也做类似处理。LODS 指令要求把串放在 DS 所指向的段中,SI 则存放将要处理的元素的偏移地址。字节型的串每个元素占 1 字节,所以每执行一次 LODSB 指令,SI 中的值会根据 DF 的情况自动加 1 或减 1;而字型的串中每个元素占 2 字节,SI 需要加 2 或减 2 后才能指向下一个元素。

串操作指令 LODS 实际上是把一条 MOV 指令和一条 ADD(或 SUB、INC、DEC 等)指令综合在一起,可以说,没有串操作指令同样可以编写数组操作的程序,但串操作指令会使这种操作简化。

【例 3-27】 设 DS 段中的变量 arr 中存放了一个带符号的字型数组，元素个数已放在字型变量 arrlen 中(个数大于 0)。编写程序段，利用串操作指令，统计出该数组中正数、0 和负数各多少个，结果分别放在 DS 段中的字型变量 countp、count0 和 countn 中。

解 程序如下：

```
        MOV     CX,[arrlen]
        MOV     [countp],0
        MOV     [count0],0
        MOV     [countn],0
        LEA     SI,arr          ;(DS)已有正确值,只要把(SI)指向串首地址
        CLD                     ;清方向标志
lab1:   LODSW
        CMP     AX,0
        JG      lab2            ;大于 0 跳转
        JL      lab3            ;小于 0 跳转
        INC     [count0]
        JMP     lab4
lab2:   INC     [countp]
        JMP     lab4
lab3:   INC     [countn]
lab4:   LOOP lab1
```

2)STOS 指令——向串中存入数据

格式：STOSB 或 STOSW

功能：

(1) STOSB 进行字节型串操作，把(AL)的值送入由 ES:DI 所确定的内存中。当(DF)=0 时，令(DI)←(DI)+1；当(DF)=1 时，令(DI)←(DI)−1。

(2)STOSW 进行字型串操作，把(AX)的值送入由 ES:DI 所确定的内存中。当(DF)=0 时，令(DI)←(DI)+2；当(DF)=1 时，令(DI)←(DI)−2。

STOS 指令主要用于把一段连续的存储区域以 AL 或 AX 中的值填充，特别的是，存储区的段地址必须放在 ES 寄存器中。STOS 与 LODS 指令配合，还可以从一个串中取出数据，有选择地存到另一个串中。

【例 3-28】 设 DS 段中的变量 arr1 中存放了一个带符号的字型数组，元素个数已放在字型变量 arr1len 中(个数大于 0)。编写程序段，试利用串操作指令，把该数组中非 0 元素复制到 DS 段中的另一个字型变量 arr2 中，要求在 arr2 中连续存放，并统计出非 0 元素的个数填在变量 arr2len 中。

分析：首先把(DS)、(SI)、(ES)和(DI)指向正确的位置，然后利用循环指令，每次从 arr1 中取出一个数，若不是 0，则存往 arr2。由于是字型数据，循环结束后 DI 的值减去 arr2 的偏移地址可得到保存下来的数据占据了多少字节，除以 2 后即得元素个数。

解　程序如下：

```
        PUSH    DS
        POP     ES              ;令(ES)←(DS)
        LEA     SI,[arr1]
        LEA     DI,[arr2]
        MOV     CX,[arr1len]
        CLD                     ;准备好取出数据的串和存入数据的串的首地址
lab1:   LODSW
        TEST    AX,AX
        JZ      lab2            ;AX 为 0 跳转
        STOSW
lab2:   LOOP lab1
        SUB     DI,OFFSET arr2
        SHR     DI,1            ;除以 2
        MOV     [arr2len],DI
```

3)MOVS 指令——串复制

格式：MOVSB 或 MOVSW

功能：

(1)MOVSB 进行字节型串复制，把 DS:SI 所指向的一个字节型数据送往 ES:DI 所指向的内存中。当(DF)＝0 时，令(SI)←(SI)＋1，(DI)←(DI)＋1；当(DF)＝1 时，令(SI)←(SI)－1，(DI)←(DI)－1。

(2)MOVSW 进行字型串复制，把 DS:SI 所指向的一个字型数据送往 ES:DI 所指向的内存中。当(DF)＝0 时，令(SI)←(SI)＋2，(DI)←(DI)＋2；当(DF)＝1 时，令(SI)←(SI)－2，(DI)←(DI)－2。

MOVS 指令可以实现把内存中的一个数据，不经过寄存器的过渡由一处复制到另一处，这一点是 MOV 指令做不到的。MOVS 指令与循环控制指令配合，可以完成数据块的复制。被复制的数据串称为源串，复制到目的地的串称为目标串。如果源串与目标串所占据的内存是完全分离的，数据传递可以按由串首至串尾的次序进行，也可以按相反的方向进行。但是，当两者占据的内存区域有部分重叠时，需要注意用 DF 控制方向：当源串首地址小于目标串首地址时，应由串尾至串首进行传送；源串首地址大于目标串首地址时，则由串首至串尾传送。

【例 3-29】　设字节型变量 str 中存放了 100 个字符，编写程序段完成下列操作：

(1)删除串中前 5 个字符，并把后续字符前移。

(2)把串中各字符向后移一个字节，在串首插入一个空格符。

分析：第(1)题要把串的后 95 个字节向前移动，是源串首址大于目标串首址的情况，需要由串首至串尾进行移动；第(2)题正相反，源串首址小于目标串首址，只能按由串尾至串首的方向移动。

解　第(1)题程序如下：

```
        MOV     AX,SEG str      ;取变量 str 所在的段地址
```

```
        MOV     DS,AX
        MOV     ES,AX
        LEA     SI,[str+5]      ;取源串首偏移地址
        LEA     DI,[str]        ;取目标串首偏移地址
        MOV     CX,95           ;置复制字节数
        CLD
lab:    MOVSB   ;字节型复制
        LOOP lab
```

第(2)题程序如下：

```
        MOV     AX,SEG str
        MOV     DS,AX
        MOV     ES,AX
        LEA     SI,[str+99]     ;取源串尾的偏移地址
        LEA     DI,[str+100]    ;取目标串尾的偏移地址
        MOV     CX,100          ;复制 100 个字节
        STD
lab:    MOVSB
        LOOP lab
        MOV     [str],' '
```

4)CMPS 指令——串比较

格式：CMPSB 或 CMPSW

功能：

(1)CMPSB 进行字节型串比较，把 DS:SI 所指向的一个字节型数据与 ES:DI 所指向的一个字节型数据相减，把相减结果反映到条件标志位上。当(DF)＝0 时，令(SI)←(SI)＋1，(DI)←(DI)＋1；当(DF)＝1 时，令(SI)←(SI)－1，(DI)←(DI)－1。

(2)CMPSW 进行字型串比较，把 DS:SI 所指向的一个字型数据与 ES:DI 所指向的一个字型数据相减，相减结果反映到条件标志位上。当(DF)＝0 时，令(SI)←(SI)＋2，(DI)←(DI)＋2；当(DF)＝1 时，令(SI)←(SI)－2，(DI)←(DI)－2。

程序设计中经常会遇到比较问题，比较两个字符串是否完全相同，或者比较两个字符串按字典顺序的大小，这正是 CMPS 指令发挥作用的地方。

【例 3-30】 编写子程序，按字典排序法比较两个已知长度的字符串的大小。

解 程序如下：

```
;入口参数:DS:SI 和 ES:DI 分别存放第 1 个串和第 2 个串的起始逻辑地址
; CX 和 DX 分别存放两个串的串长
;出口参数:AL 为 1 表示第 1 个串大,AL 为－1 表示第 2 个串大,AL 为 0 表示两者相等
strcmp      PROC    NEAR
            CLD
            MOV     AH,0            ;记载串长 1<串长 2
            CMP     CX,DX
```

```
                JB      lab1
                MOV     AH,1            ;记载串长相等
                JE      lab1
                MOV     CX,DX           ;按第 2 个串的长度进行比较
                MOV     AH,2            ;记载串长 1>串长 2
lab1:           JCXZ    lab2
                CMPSB
                JA      lab3            ;串 1>串 2 跳转
                JB      lab4            ;串 1<串 2 跳转
                LOOP    lab1
lab2:           CMP     AH,1
                JB      lab4            ;串 1<串 2 跳转
                JA      lab3            ;串 1>串 2 跳转
                MOV     AL,0
                JMP     lab5
lab3:           MOV     AL,1
                JMP     lab5
lab4:           MOV     AL,-1
lab5:           RET
                strcmp  ENDP
```

5)SCAS 指令——串扫描

格式:SCASB 或 SCASW

功能:

(1)SCASB 把(AL)与字节型串中数据比较,用(AL)减去 ES:DI 所指向的一个字节型数据,相减结果反映到条件标志位上。当(DF)=0 时,令(DI)←(DI)+1;当(DF)=1 时,令(DI)←(DI)-1。

(2)SCASW 把(AX)与字型串中数据比较,用(AX)减去 ES:DI 所指向的一个字,结果反映到条件标志位上。当(DF)=0 时,令(DI)←(DI)+2;当(DF)=1 时,令(DI)←(DI)-2。

SCAS 指令通常用于查找一个数组中是否存在某个指定的值。该指令不改变数组中的任何数据,也不改变(AX)或(AL)的值,可以用循环控制的方法连续查找。

【例 3-31】 编写子程序,查找一个字型数组中是否存在一个给定的值。

解　程序如下:

```
;入口参数:ES:DI 存放字型数组的首地址,CX 存放串中元素的个数
; AX 存放指定查找的值
;出口参数:(CF)为 1 表示找到,(CF)为 0 表示没找到
search    PROC    NEAR
          JCXZ    lab0
          CLD
lab1:     SCASW
```

```
            JE      lab2
            LOOP    lab1
lab0:       CLC
            JMP     lab3
lab2:       STC
lab3:       RET
search      ENDP
```

3.4.3 串重复前缀

串操作指令是对内存中连续存放的一批数据进行处理的一种高效、快捷的方法，它往往需要循环控制指令的配合。对于那些单纯是数据块复制、查找、比较的操作，汇编语言中还设计了 3 个串操作重复前缀，以进一步提高编程和数据处理的效率。

串操作前缀是附加在串操作前面的指令，它是一种以 CX 为计数器的重复操作指示器，用以简化循环操作控制。使用串操作前缀的一般格式是：

串前缀　串操作指令

1)REP 前缀

功能：当 CX 的值不是 0 时，重复执行后面的串操作指令，每执行一次，把 CX 的值减 1，直到(CX)＝0 为止。图 3-6 描述了 REP 串前缀的功能。

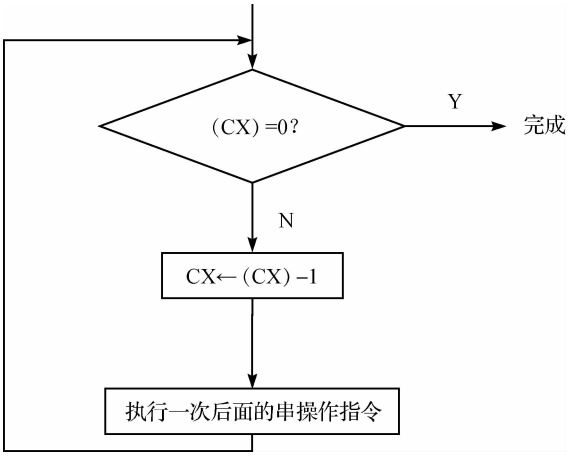


图 3-6　REP 前缀的功能

从图中可以看到，这是一种先判断后重复的循环，如果 CX 的值是 0，则串操作指令一次都不执行，这与 LOOP 指令控制的循环是不同的。

REP 前缀通常加在 MOVS 或 STOS 串操作指令的前面，用以把一个串复制到内存的另一个地方，或者把一段内存区域用一个特定值填充。REP 前缀一般不与另外 3 条串操作指令连用。

2)REPZ 和 REPNZ 前缀

REPZ 和 REPNZ 也是串操作指令前缀，与 REP 一样都是用于控制后面的串操作指令重复执行，但重复执行不仅依赖于 CX 的值，还依赖于标志寄存器中的 ZF 标志位。

带有 REPZ 前缀的串操作指令按下列方式执行：

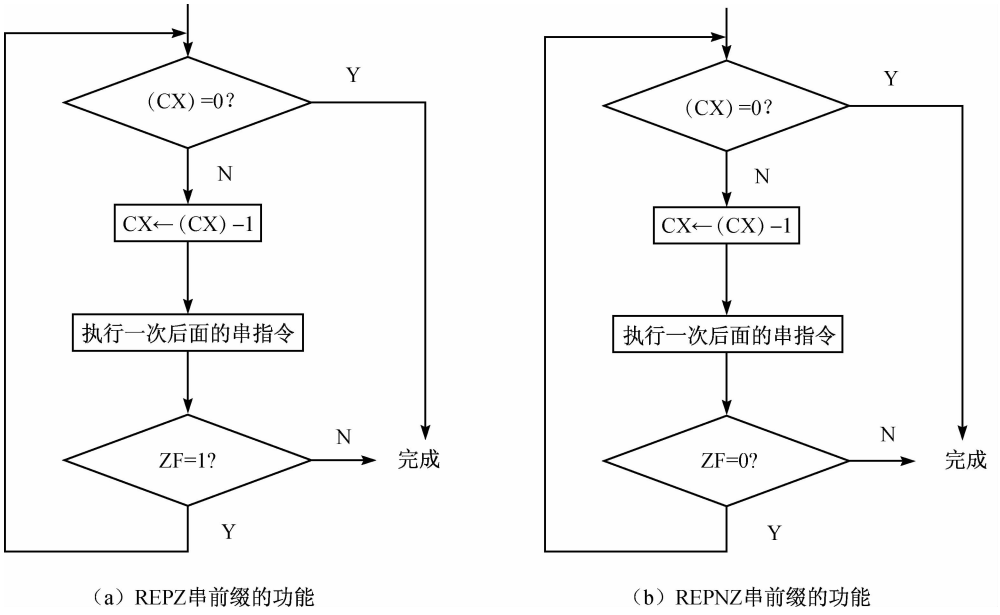
(1)若(CX)＝0，则结束指令的执行，否则转(2)。

(2)CX←(CX)－1。

(3)执行一次串操作指令。

(4)若 ZF＝0，则结束指令的执行，否则转(1)。

REPNZ 的功能与 REPZ 仅在第(4)步不同，REPZ 是在 ZF＝1 时控制串操作重复执行，而 REPNZ 是在 ZF＝0 时控制串操作重复执行。REPZ 和 REPNZ 的功能可以用图 3-7 描述。



（a）REPZ串前缀的功能　　（b）REPNZ串前缀的功能

图 3-7　REPZ 和 REPNZ 前缀的功能

REPZ 和 REPNZ 前缀通常加在 CMPS 或 SCAS 串操作指令的前面，完成连续比较操作。这两个前缀各自又有一种功能完全相同的变形，REPZ 可以写作 REPE，REPNZ 可以写作 REPNE。

3)串前缀的应用

根据串前缀与串操作指令的功能，串操作指令前面配置串前缀的类型是有一定限制的。表 3-1 列出了串前缀与串操作指令之间的配合关系，其中的“√”表示对应的串操作指令与串前缀可以配合使用，“×”表示不能，“△”表示相应的用法没有实用价值。

表 3-1　串前缀与串操作指令之间的配合关系

串前缀 / 串操作指令	REP	REPZ (REPE)	REPNZ (REPNE)
LODS	△	×	×
STOS	√	×	×
MOVS	√	×	×
CMPS	×	√	√
SCAS	×	√	√

从表 3-1 中可以看出，LODS 指令一般不与串前缀配合使用，因为 LODS 是从串中取出数据放到 AL 或 AX 中，每取一个数据就应该做适当的处理，然后再去取下一个数据，否则后取出的数据将取代 AL 或 AX 中的原有数据，使得只有最后一次取出的数据被保留下来。

实际上，8088 的指令系统中只有两个机器码与串前缀对应，并且，对于 MOVS、STOS 和 LODS，不论串操作指令的前面加的是什么前缀，都按 REP 进行处理，汇编程序在翻译时既不报错也不警告；对 CMPS 和 SCAS 指令，如果前面加上了前缀 REP，汇编程序将按 REPZ 进行翻译。

串前缀的用途是简化控制串操作的循环结构，【例 3-32】和【例 3-33】就是这种简化的典型用法。

【例 3-32】 把字型变量 v1 中存放的 50 个整数复制到变量 v2 中，先用 LOOP 指令编写程序段完成复制操作，再用带前缀的串操作指令简化。

解 用循环控制方法编写的程序段如下：

```
        MOV  AX,SEG v1
        MOV  DS,AX            ;准备源串的段地址
        MOV  SI,OFFSET v1     ;准备源串的起始偏移地址
        MOV  AX,SEG v2
        MOV  ES,AX            ;准备目标串的段地址
        LEA  DI,v2            ;准备目标串的起始偏移地址
        MOV  CX,50
lab:    MOV  AX,[SI]
        MOV  ES:[DI],AX
        LOOP lab
```

该程序段的最后 3 行可以用一个带前缀的串操作指令简化，写为：

```
        REP  MOVSW
```

【例 3-33】 编写一个子程序，判断一个数据串中是否存在一个给定的值。要求子程序对字节型和字型的串都能判断，以(CF)作为出口参数，如果在串中找到目标值，则(CF)位置 1，否则令(CF)清 0。

解 程序如下：

```
;入口参数:ES:DI=数据串的首地址
;(AX)=查找目标值,字节型数据串则用(AL)存放目标值
```

```
;(CX)=串中元素个数
;(CF)=0 表示串中元素是字型,(CF)=1 则表示串中元素是字节型
;出口参数:(CF)=1 表示在串中找到了给定值,(CF)=0 表示没找到
search      PROC    NEAR
            JCXZ    s2          ;串长为 0,串中不存在给定数据,跳转
            CLD                 ;清方向标志,准备按增量方向查找
            JC      s3          ;入口参数(CF)为 1,转字节型查找
            REPNZ   SCASW       ;字型查找
            JZ      s1          ;找到跳转
            JMP     s2          ;未找到跳转
s3:         REPNZ   SCASB       ;字节型查找
            JZ      s1          ;找到跳转
s2:         CLC                 ;置未找到标记
            JMP     s4
s1:         STC                 ;置找到标记
s4:         RET
search      ENDP
```

从图 3-7 的流程可以看出,带有 REPZ 和 REPNZ 前缀的串操作指令可以在两种情况下结束串操作:一是已执行到(CX)为 0 时,二是当(ZF)不符合要求时。【例 3-33】中,在带有 REPNZ 前缀的串扫描指令的后面用条件跳转指令进行判断,此时必须能够分辨出是哪一种情况导致串操作结束。可以想到的指令除了 JZ、JNZ 之外,还有 JCXZ,究竟用哪一个为好呢? 如果串操作指令执行完后(CX)不是 0,可以肯定是由于(ZF)不满足重复条件而导致串操作提前结束的,这种情况下 JCXZ 或 JZ、JNZ 指令都可使用;反之若(CX)是 0,表示串操作已经处理到串的最后一个元素,并且最后一次处理的结果已设置在(ZF)上,但这时如果用 JCXZ 指令进行判断,就不能分辨最后一次串操作比较或查找的结果是相等还是不等。

总之,在带有 REPZ 或 REPNZ 前缀的串操作指令的后面,必须用 JZ 或 JNZ 指令判断比较或查找的情况,而不能用 JCXZ 指令。

注意:

(1)串操作指令能对存储区中一块(串)字节或字进行操作,其块的长度可达 64 KB,任一个这样的基本操作指令前还能用一个重复前缀使它们重复地操作。

(2)所有的基本串操作指令都约定:源串用寄存器 SI 进行寻址,在无超越前缀时,段地址取自于 DS 寄存器;目的串则用寄存器 DI 进行寻址,其段地址总是取自于 ES 寄存器中。

(3)串操作指令在每一次操作之后能自动修改地址指针 SI、DI 的值,以便指向串中下一个元素的地址。但按增量还是按减量来修改地址,则取决于方向标志(DF)。若(DF)=0,则 SI、DI 自动增量(字节操作加 1,字操作加 2);反之,SI、DI 自动减量。DF 的状态由指令 STD(置方向标志)和 CLD(清除方向标志)来控制。

(4)MOVS、LODS、STOS 指令对标志位无影响,CMPS、SCAS 指令对标志位的影响同 CMP 指令。

3.5 32位CPU扩展的指令

Intel公司于1985年正式发布了32位微处理器80386，并将其32位指令系统的结构命名为英特尔结构(intel architecture，IA)，同时宣布其作为后续80x86微处理器的标准。随后，从Intel 80486至Pentium 4的一系列产品都继承了该结构，并新增了若干条专用指令。此外，在32位整数指令系统的基础上还加入了浮点指令、整数多媒体MMX(multimedia extensions，多媒体扩展指令集)指令、单精度浮点多媒体SSE(streaming SIMD extensions，单一指令多数据流扩展)指令和双精度浮点多媒体SSE2指令，极大地丰富了Intel 80x86微处理器的指令系统，增强了Intel 80x86微处理器的功能。

3.5.1 32位指令运行环境

32位的80x86微处理器有3种工作模式：实模式、保护模式和虚拟8086模式。

1)实模式

实模式是为了和8086处理器兼容而设置的。在实模式下，80386处理器就相当于一个快速的8086处理器。

80386处理器被复位或加电时以实模式启动，这时处理器中的各寄存器以实模式的初始化值工作。80386处理器在实模式下的存储器寻址方式和8086是一样的，由段寄存器的内容乘以10H当做基地址，加上段内的偏移地址形成最终的物理地址，这时候它的32位地址线只使用了低20位。在实模式下，80386处理器不能对内存进行分页管理，所以指令寻址的地址就是内存中实际的物理地址。在实模式下，所有的段都是可以读、写和执行的。

实模式下80386不支持优先级，所有的指令相当于工作在特权级(优先级0)，所以它可以执行所有特权指令，包括读写控制寄存器CR0等。实际上，80386就是通过在实模式下初始化控制寄存器、GDTR、LDTR、IDTR与TR等管理寄存器以及页表，然后再通过置CR0的保护模式位(PE位)进入保护模式。实模式下不支持硬件上的多任务切换。

实模式下的中断处理方式和8086处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和8086处理器一样，每4字节组成一个中断向量，其中包括两字节的段地址和两字节的偏移地址。

从编程的角度看，实模式的80386处理器有以下好处：首先，可以使用80386的32位寄存器，用32位的寄存器进行编程可以使计算程序更加简捷，加快了执行速度；其次，80386中增加的两个辅助段寄存器FS和GS在实模式下也可以使用，这样，同时可以访问的段达到了6个，从而不必考虑重新装入的问题；最后，很多80386的新增指令也使一些原来不很方便的操作得以简化，当然，使用了这些新指令的程序是无法回到8086处理器上执行的，因为这些指令的编码在8086处理器上未定义。

2)保护模式

保护模式是80386处理器的主要工作模式。

当 80386 工作在保护模式下时，它的所有功能都是可用的。这时 80386 的 32 根地址线都可供寻址，物理寻址空间高达 4 GB。在保护模式下，支持内存分页机制，提供了对虚拟内存的良好支持。虽然与 8086 可寻址的 1 MB 物理地址空间相比，80386 可寻址的物理地址空间很大，但实际的微机系统不可能安装如此大的物理内存，所以，为了运行大型程序和真正实现多任务，虚拟内存是一种必需的技术。

保护模式下 80386 支持多任务，可以依靠硬件在一条指令中实现任务切换。任务环境的保护工作是由处理器自动完成的。在保护模式下，80386 处理器还支持优先级机制，不同的程序可以运行在不同的优先级上。优先级有 0～3 共 4 个级别，操作系统运行在最高的优先级 0 上，应用程序则运行在比较低的级别上；配合良好的检查机制后，既可以在任务间实现数据的安全共享，也可以很好地隔离各个任务。从实模式切换到保护模式是通过修改控制寄存器 CR0 的控制位 PE(位 0)来实现的。在这之前还需要建立保护模式必需的一些数据表，如全局描述符表(GDT)和中断描述符表(IDT)等。

3)虚拟 8086 模式

为了在保护模式下继续提供和 8086 处理器的兼容，80386 又设计了一种虚拟 8086 模式，以便可以在保护模式的多任务条件下，有的任务运行 32 位程序，有的任务运行 MS-DOS 程序。

虚拟 8086 模式是以任务形式在保护模式上执行的，在 80386 上可以同时支持由多个真正的 80386 任务和虚拟 8086 模式构成的任务。在虚拟 8086 模式下，80386 支持任务切换和内存分页。在 Windows 操作系统中，有一部分程序专门用来管理虚拟 8086 模式的任务，称为虚拟 8086 管理程序。

既然虚拟 8086 模式以保护模式为基础，它的工作方式实际上是实模式和保护模式的混合。为了和 8086 程序的寻址方式兼容，虚拟 8086 模式采用和 8086 一样的寻址方式，即用段寄存器乘以 10H 当做基址，再配合偏移地址形成线性地址，寻址空间为 1 MB。但显然多个虚拟 8086 任务不能同时使用同一位置的 1 MB 地址空间，否则会引起冲突。操作系统利用分页机制将不同虚拟 8086 任务的地址空间映射到不同的物理地址上，这样每个虚拟 8086 任务看起来都认为自己在使用 0～1 MB 的地址空间。

8086 代码中有相当一部分指令在保护模式下属于特权指令，如屏蔽中断的 CLI 和中断返回指令 IRET 等。这些指令在 8086 程序中是合法的，如果不让这些指令执行，8086 代码就无法工作。为了解决这个问题，虚拟 8086 管理程序采用模拟的方法来完成这些指令。这些特权指令执行时引起了保护异常。虚拟 8086 管理程序在异常处理程序中检查产生异常的指令，如果是中断指令，则从虚拟 8086 任务的中断向量表中取出中断处理程序的入口地址，并将控制转移过去；如果是危及操作系统的指令，如 CLI 等，则简单地忽略这些指令，在异常处理程序返回时直接返回到下一条指令。通过这些措施，8086 程序既可以正常地运行下去，又可以在执行这些指令时觉察不到已经被虚拟 8086 管理程序做了手脚。MS-DOS 应用程序在 Windows 操作系统中就是这样工作的。

80386 处理器的 3 种工作模式各有特点且相互联系。实模式是 80386 处理器工作的基础，这时 80386 当做一个快速的 8086 处理器工作。在实模式下可以通过指令切换到保护模式，也可以从保护模式退回到实模式。虚拟 8086 模式则以保护模式为基础，在保护模式和虚拟 8086 模式之间可以互相切换，但不能从实模式直接进入虚拟 8086 模式或从虚拟 8086

模式直接退到实模式。DOS 操作系统运行于实模式下，而 Windows 操作系统运行于保护模式下。

3.5.2 32 位 CPU 寄存器组和寻址方式的变化

Intel 32 位处理器在原 16 位处理器的寄存器组和寻址方式的基础上做了相应的扩充，以适应 32 位微处理器的需求，同时又对原有的 16 位微处理器保持兼容。

80386 微处理器共有 7 类 34 个寄存器，包括通用寄存器组、段寄存器、指令指针和标志寄存器、系统地址寄存器、控制寄存器、调试寄存器、测试寄存器。通常应用程序主要使用前三类，如图 3-8 所示。

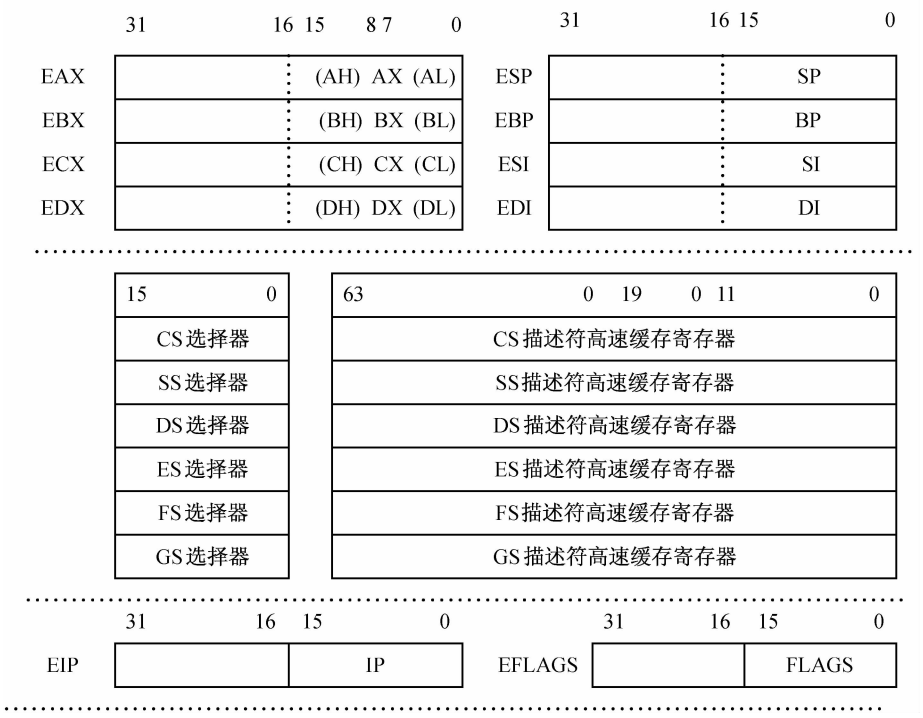


图 3-8 通用寄存器组、段寄存器、指令指针和标志寄存器示意图

通用寄存器组共有 8 个 32 位寄存器：EAX、EBX、ECX、EDX、ESP、EBP、ESI、EDI，它们由 8086 的 16 位寄存器扩展而来。这些通用寄存器的低 16 位还可以作为 16 位寄存器存取，并不受影响。而以前的 AX、BX、CX 和 DX 寄存器还可以单独使用，其高 8 位和低 8 位分别是 AH、AL，BH、BL，CH、CL，DH、DL。在 80386 中，8 个 32 位通用寄存器都可以作为指针寄存器使用，所以 32 位通用寄存器更加通用。

段寄存器组共有 6 个 16 位的段寄存器：CS、DS、SS、ES、FS、GS。与这 6 个段寄存器对应的有 6 个 64 位描述符寄存器，它是 80x86 处理器提供的一种附加的非编程的寄存器，用来装 64 位段描述符，每当一个段选择符被装入段寄存器时，相应的段描述符就由内存装入到对应的非编程的 CPU 寄存器中。其中 CS、DS、SS、ES 与 8086 的段寄存器完全相同，在实模式下，使用方法也与 8086 相同；在保护模式下，这些寄存器中的值是“段选择符”，需要查

全局描述符表(GDT)或者局部描述符表(LDT)来获得段基址,再加上偏移地址才能得到线性地址。FS 和 GS 是新加的附加数据段寄存器,可以由用户将 FS、GS 定义为其他数据段。

指令指针和标志寄存器包括:指令指针寄存器 EIP,由 8086 的 IP 寄存器扩展而来;标志寄存器 EFLAGS,由 8086 的 FLAGS 寄存器扩展而来,包含一组状态标志、一个控制标志、一组系统标志,其具体内容如图 3-9 所示。

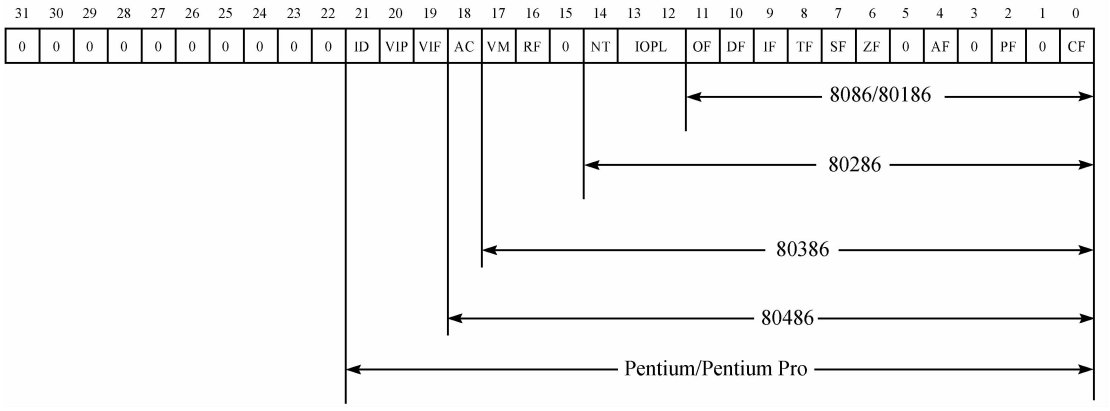


图 3-9　标志寄存器 EFLAGS 的标准位

标志寄存器 EFLAGS 的低 12 位标志位的意义与 8086 的标志寄存器 FLAGS 的低 12 位含义完全相同,其他各标志位的含义如下。

IOPL(I/O privilege level)占两个二进制位,表示 I/O 特权级。如果当前特权级小于或等于 IOPL,就可以执行 I/O 操作,否则将出现一个保护性异常。IOPL 只能由特权级为 0 的程序或任务来修改。

NT(nested task)表示嵌套任务,用于控制中断返回指令 IRET。当(NT)=0 时,用堆栈中保存的值恢复 EFLAGS、CS 和 EIP,从而实现返回;若(NT)=1,则通过任务切换实现中断返回。

VM(virtual-8086 mode)表示虚拟 8086 模式。如果 VM 被置位且 80386 已处于保护模式下,则 CPU 切换到虚拟 8086 模式,此时,对段的任何操作又回到了实模式,如同在 8086 下运行一样。

RF(resume flag)表示恢复标志(又称重启标志),与调试寄存器一起用于断点和单步操作。当(RF)=1 时,下一条指令的任何调试故障将被忽略,不产生异常中断;当(RF)=0 时,调试故障被接受,并产生异常中断(常用于调试失败后,强迫程序恢复执行,在成功执行每条指令后,(RF)自动复位)。

AC(alignment check)表示对齐检查(只有运行在特权级 3 的程序才执行地址对齐检查,特权级 0、1、2 忽略该标志)。当(AC)=1 且 CR0 中的(AM)=1 时,允许存储器进行地址对齐(指当访问一个字时,其地址必须是偶数;当访问双字时,其地址必须是 4 的倍数)检查,若发现地址未对齐,则将产生异常中断。

VIF(virtual interrupt flag)表示虚拟中断标志。当(VIF)=1 时,可以使用虚拟中断;当(VIF)=0 时,不能使用虚拟中断。

VIP(virtual interrupt pending flag)表示虚拟中断挂起标志。当(VIP)=1 时,VIF 有

效;当(VIP)=0时,VIF无效。

ID(identification flag)表示鉴别标志。该标志用来指示Pentium CPU是否支持CPUID的指令。

实际上,如果不编写操作系统,大部分标志可能很难用到。

注意:由于在实模式下,段的最大范围是64 KB,所以EIP的高16位必须全是0,仍相当于16位的IP作用。

8086的16位寻址方式同样适用于32位80x86 CPU,只不过是将立即数、寄存器、存储器有效地址扩充到了32位,而且8个32位通用寄存器都可以当做基址寄存器,除了ESP外的另外7个通用寄存器都可以当做变址寄存器使用。除此之外,Intel 32位CPU还提供3种带比例因子、仅适用于32位CPU的寻址方式:比例变址寻址、基址比例变址寻址和相对基址比例变址寻址。

比例因子可以是1×、2×、4×或8×。默认比例因子是1×,不要求包含在汇编语言指令中。比例因子2×用来寻址字存储器数组,比例因子4×用来寻址双字存储器数组,而比例因子8×用来寻址4字存储器数组。

(1)比例变址寻址中操作数的有效地址是变址寄存器内容乘以指令中指定的比例因子再加上位移量之和,所以有效地址由3种成分组成。变址寄存器为EAX、EBX、ECX、EDX、ESI、EDI时,操作数默认在数据段;变址寄存器为EBP时,操作数默认在堆栈段。该指令可加段超越前缀。该寻址方式举例如下:

```
MOV EAX,COUNT[SI*4]
```

(2)基址比例变址寻址中操作数的有效地址是变址寄存器内容乘以指令中指定的比例因子再加上基址寄存器内容。基址寄存器可为8个32位通用寄存器,变址寄存器为除了ESP外的另外7个通用寄存器。基址寄存器为EAX、EBX、ECX、EDX、ESI、EDI时,操作数默认在数据段;基址寄存器为EBP、ESP时,操作数默认在堆栈段。该指令可加段超越前缀。该寻址方式举例如下:

```
MOV EAX,[EBX][SI*4]
```

(3)相对基址比例变址寻址中操作数的有效地址是变址寄存器内容乘以指令中指定的比例因子加上基址寄存器内容,再加上指定位移量。基址寄存器可为8个32位通用寄存器,变址寄存器为除了ESP外的另外7个通用寄存器。基址寄存器为EAX、EBX、ECX、EDX、ESI、EDI时,操作数默认在数据段;基址寄存器为EBP、ESP时,操作数默认在堆栈段。该指令可加段超越前缀。该寻址方式举例如下:

```
MOV EAX,COUNT[EBX][SI*4]
```

32位存储器寻址方式的组成公式可总结为:

32位有效地址= 基址寄存器+(变址寄存器×比例)+位移量

其中的3个组成部分是:

(1)基址寄存器——任何8个32位通用寄存器之一。

(2)变址寄存器——除ESP之外的任何32位通用寄存器之一。

(3)比例——可以是1/2/4/8(因为操作数的长度可以是1/2/4/8字节)。

3.5.3　32 位扩展增强指令

32 位 80x86 CPU 对 8086 某些指令的操作数进行了扩展，同时在 8086 指令系统的基础上引入了很多新指令，实现了原有功能的增强。主要的新引入的指令如下。

1)数据传送类指令

(1)从 80186 开始 PUSH 指令可以将立即数压入堆栈。

```
PUSH    i8/i16/i32
```

该指令把 16 位或 32 位立即数 i16/i32 压入堆栈。若是 8 位立即数，则经符号扩展成 16 位后再压入堆栈。

注意：下面的描述中，i 表示立即数，r 表示寄存器，m 表示存储单元，后面的数字为其位数。

(2)由于 32 位 80x86 CPU 新增了 FS 和 GS 段寄存器，所以可对它们进行堆栈操作。

```
PUSH   FS
PUSH   GS
POP    FS
POP    GS
```

(3)新增 16 位通用寄存器进栈 PUSHA 和出栈 POPA 指令。

```
PUSHA   ;顺序压入 AX,CX,DX,BX,SP(指令执前),BP,SI,DI
POPA    ;栈顶的 16 字节依次出栈到 DI,SI,BP,SP,BX,DX,CX,AX,其中应进入
        ;SP 的值丢弃,SP 通过加 16 来恢复
```

(4)新增 32 位通用寄存器进栈 PUSHAD 和出栈 POPAD 指令。

```
PUSHAD  ;顺序压入 EAX,ECX,EDX,EBX,ESP(指令执前),EBP,ESI,EDI
POPAD   ;栈顶的 32 字节依次出栈到 EDI,ESI,EBP,ESP,EBX,EDX,ECX,EAX,
        ;其中应进入 ESP 的值丢弃,SP 通过加 32 来恢复
```

(5)新增 PUSHFD 和 POPFD 指令，用于传送新扩展的标志。

```
PUSHFD  ;将 EFLAGS 压入堆栈,堆栈中标志位 VM 和 RF 被清 0
POPFD   ;将 EFLAGS 弹出堆栈,其中 VIP 和 VIF 被清 0, VM 不改变
```

(6)新增 LFS、LGS、LSS 3 条地址传送指令，功能与原 LDS、LES 指令类似。

2)算术运算指令

(1)有符号数乘法又提供了新形式。

①双操作数格式：双操作数格式中，乘积存储在第一个操作数中，第一个操作数必须是寄存器，第二个操作数可以是寄存器、内存操作数或立即数。

```
IMUL    r16,r16/m16/i8/i16   ;r16←r16×r16/m16/i8/i16
IMUL    r32,r32/m32/i8/i32   ;r32←r32×r32/m32/i8/i32
```

②三操作数格式：三操作数格式中，乘积存储在第一个操作数。

```
IMUL    r16,r16/m16,i8/i16   ;r16←r16/m16×i8/i16
IMUL    r32,r32/m32,i8/i32   ;r32←r32/m32×i8/i32
```

如果有效位丢失,则溢出标志和进位标志置位。使用三操作数格式时,一定要在执行完IMUL操作后检查相关操作位。

(2)新增6条符号扩展指令。

```
CWDE   ;把AX符号扩展为EAX,该指令是CBW的扩展
CDQ    ;把EAX符号扩展为EDX:EAX,该指令是CWD的扩展
MOVSX  r16,r8/m8            ;把r8/m8符号扩展并传送至r16
MOVSX   r32,r8/m8/r16/m16   ;把r8/m8/r16/m16符号扩展并传送至r32
MOVZX   r16,r8/m8           ;把r8/m8零位扩展并传送至r16
MOVZX   r32,r8/m8/r16/m16   ;把r8/m8/r16/m16零位扩展并传送至r32
```

3)位操作类指令

移位和循环移位指令从80186开始,可以用一个立即数指定移位次数。但是,实际的移位次数等于指令中指定移位次数的低5位,所以为0～31。

4)串操作指令

新增串输入INS和串输出OUTS指令。

```
INS   ;I/O串输入:存储单元ES:[(E)DI]←I/O端口[DX];(E)DI←(E)DI±1/2/4
OUTS  ;I/O串输出:I/O端口[DX]←存储单元DS:[(E)SI];(E)SI←(E)SI±1/2/4
```

其中,(DF)=0时,地址自增;(DF)=1时,地址自减。±1为字节串;±2为字串;±4为双字串。

5)控制转移类指令

为了更好地支持高级语言,引入3条新指令。

(1)建立堆栈帧指令ENTER。

```
ENTER   i16,i8    ;分配一个堆栈帧,目的操作数是一个16位常数(取值0～FFFFH),
                  ;表示堆栈空间的字节数;源操作数是一个8位常数(取值0～31),
                  ;表示允许过程嵌套的层数
```

(2)释放堆栈帧指令LEAVE。

```
LEAVE  ;撤销ENTER指令所设置的堆栈空间,一般在过程返回指令之前
```

(3)边界检测指令BOUND。

```
BOUND   r16/r32,m16/m32   ;检查寄存器的内容是否满足关系式(存储器地址)≤(寄存
                          ;器)≤(存储器地址+2),如果不能满足则产生一个5号中
                          ;断;若满足关系式,则指令不做任何操作
```

6)保护方式类指令

80286虽然也是一个16位CPU,但它引入了保护方式,设计有一些用于保护方式的指令,极大地提高了性能。相应地,这些指令同样适合于32位80x86微处理器。这些指令很多都是所谓的特权指令,通常只有系统核心程序能够使用它们。保护方式类指令这里不做介绍,感兴趣的读者可参阅其他相关书籍。

3.5.4　32 位新增指令

1)80386 新增指令

(1)双精度移位指令。此组指令有:双精度左移 SHLD 和双精度右移 SHRD。它们都是具有 3 个操作数的指令,其指令格式如下:

```
SHLD    r16/m16,r16,i8/CL    ;将 r16 的 i8/CL 位左移进入 r16/m16
SHLD    r32/m32,r32,i8/CL    ;将 r32 的 i8/CL 位左移进入 r32/m32
SHRD    r16/m16,r16,i8/CL    ;将 r16 的 i8/CL 位右移进入 r16/m16
SHRD    r32/m32,r32,i8/CL    ;将 r32 的 i8/CL 位右移进入 r32/m32
```

其中,第一操作数是一个 16 位/32 位的寄存器或存储单元;第二操作数(与前者具有相同位数)一定是寄存器;第三操作数是移动的位数,它可由 CL 或一个立即数来确定。在执行 SHLD 指令时,第一操作数向左移 n 位,其“空出”的低位由第二操作数的高 n 位来填补,但第二操作数不移动、不改变。在执行 SHRD 指令时,第一操作数向右移 n 位,其“空出”的高位由第二操作数的低 n 位来填补,第二操作数也不移动、不改变。受影响的标志位为:CF、OF、PF、SF 和 ZF(AF 无定义)。表 3-2 是几个双精度移位的例子及其执行结果。

表 3-2　双精度移位的例子及其执行结果

双精度移位指令	指令操作数的初值	指令执行后的结果
SHLD AX, BX, 1	(AX)=1234H,(BX)=8765H	(AX)=2469H
SHLD AX, BX, 3	(AX)=1234H,(BX)=8765H	(AX)=91A4H
SHRD AX, BX, 2	(AX)=1234H,(BX)=8765H	(AX)=448DH

(2)位扫描指令。此组指令有向前位扫描指令 BSF 和向后位扫描指令 BSR。其指令格式如下:

```
BSF     r16,r16/m16     ;16 位操作
BSF     r32,r32/m32     ;32 位操作
BSR     r16,r16/m16     ;16 位操作
BSR     r32,r32/m32     ;32 位操作
```

向前位扫描指令从最低位(第 0 位)开始测试源操作数中的各位,当遇到有 1 的位时,(ZF)=0,且将该位的序号存入目的操作数中;如源操作数的所有位都是 0,则(ZF)=1,且目的操作数中的值无意义。源操作数应为和目的操作数同类型的 16 或 32 位的寄存器或存储器操作数,目的操作数必须为寄存器操作数。例如,假定 EBX 中的数是 12F234E0H,则执行指令 BSF EAX,EBX 后,(EAX)=5,(ZF)=0。

向后位扫描指令从最高位(第 15 位或 31 位)开始测试源操作数中的各位,当遇到有 1 的位时,(ZF)=0,且将该位的序号存入目的操作数中;如源操作数中的所有位都是 0,则(ZF)=1,且目的操作数中的值无意义。源操作数应为和目的操作数同类型的 16 或 32 位的寄存器或存储器操作数,目的操作数必须为寄存器操作数。例如,假定 EBX 中的数是 12F234E0H,则执行指令 BSR EAX,EBX 后,(EAX)=28,(ZF)=0。

(3)位测试指令。此组指令有 BT、BTC、BTR 和 BTS 四条。其指令格式如下：

```
BT    dest,src    ;把目的操作数(dest)中由源操作数(src)指定的位送(CF)
BTC   dest,src    ;把(dest)中由(src)指定的位送(CF),然后对指定位求反
BTR   dest,src    ;把(dest)中由(src)指定的位送(CF),然后对指定位复位
BTS   dest,src    ;把(dest)中由(src)指定的位送(CF),然后对指定位置位
```

在指令中,目的操作数(dest)只能是 16 或 32 位通用寄存器或存储单元,用于指定要测试的数据;源操作数(src)必须是 8 位立即数或者是与目的操作数等长的 16 位或 32 位通用寄存器,用于指定要测试的位。如果目的操作数是寄存器,则源操作数除以 16 或 32 的余数就是要测试的位,它在 0～15 或 0～31 之间。例如：

```
MOV    EAX,12345678H        ;(EAX)=12345678H
BT     EAX,5                ;(EAX)的 D5 位=1→(CF),(EAX)=12345678H
MOV    EAX,12345678H        ;(EAX)=12345678H
BTC    EAX,10               ;(EAX)的 D10位=1→(CF),(EAX)=12345278H
MOV    EAX,12345678H        ;(EAX)=12345678H
BTR    EAX,20               ;(EAX)的 D20位=1→(CF),(EAX)=12245278H
MOV    EAX,12345678H        ;(EAX)=12345678H
BTS    EAX,34               ;(EAX)的 D2 位=0→(CF),(EAX)=1224527CH
```

(4)条件设置指令。此组指令的通用格式如下：

```
SETxx   r8/m8
```

其中,xx 是表示测试条件,共有 16 种,与条件转移指令 Jxx 中的条件意义相同,具体如表 3-3 所示;指令操作数只能是 8 位寄存器或一个字节单元。这组指令的执行不影响任何标志位。

表 3-3　条件设置指令中的 SETxx 及说明

助记符	标志位	说　明	助记符	标志位	说　明
SETZ/SETE	(ZF)=1	等于零/相等	SETC/SETB/SETNAE	(CF)=1	进位/低于/不高于等于
SETNZ/SETNE	(ZF)=0	不等于零/不相等	SETNC/SETNB/SETAE	(CF)=0	无进位/不低于/高于等于
SETS	(SF)=1	符号为负	SETBE/SETNA	(CF)=1 或(ZF)=1	低于等于/不高于
SETNS	(SF)=0	符号为正	SETNBE/SETA	(CF)=0 且(ZF)=0	不低于等于/高于
SETP/SETPE	(PF)=1	1 的个数为偶	SETL/SETNGE	(SF)≠(OF)	小于/不大于等于
SETNP/SETPO	(PF)=0	1 的个数为奇	SETNL/SETGE	(SF)=(OF)	不小于/大于等于
SETO	(OF)=1	溢出	SETLE/SETNG	(ZF)≠(OF)或(ZF)=1	小于等于/不大于
SETNO	(OF)=0	无溢出	SETNLE/SETG	(SF)=(OF)且(ZF)=0	不小于等于/大于

【例 3-34】 编写程序段,检测寄存器 EAX 中的 8 个十六进制数中有多少个 0H,并把统计结果存入(BH)中。

解　方法 1:用条件转移指令来实现。

```
XOR    BH, BH
MOV    CX, 8      ;测试寄存器 EAX 8 次
```

```
again:  TEST   AL, 0FH   ;测试低 4 位二进制是否为 0H
        JNZ    next
        INC    BH
next:   ROR    EAX, 4    ;循环向右移 4 位,为测试高 4 位做准备
        LOOP   again
```

方法 2:用条件设置字节指令来实现。

```
        XOR    BH, BH
        MOV    CX, 8     ;测试寄存器 EAX 8 次
again:  TEST   AL, 0FH   ;测试低 4 位二进制是否为 0H
        SETZ   BL        ;如果 AL 的低 4 位是 0,则 BL 置为 1,否则,BL 为 0
        ADD    BH, BL
        ROR    EAX, 4
        LOOP   again
```

(5)控制、调试和测试寄存器传送指令。32 位 80x86 CPU 中增加了一些系统控制用的寄存器,故增加了有关这些寄存器的传送指令,通常只有系统程序使用它们。其指令格式如下:

```
MOV  CRn,r32   ;控制寄存器装入:(CRn)←r32,n=0～4
MOV  r32,CRn   ;控制寄存器读取:r32←(CRn),n=0～4
MOV  DRn,r32   ;调试寄存器装入:(DRn)←r32,n=0～7
MOV  r32,DRn   ;调试寄存器读取:r32←(DRn),n=0～7
MOV  TRn,r32   ;测试寄存器装入:(TRn)←r32,n=3～7
MOV  r32,TRn   ;测试寄存器读取:r32←(TRn),n=3～7
```

其中,Pentium 以后才支持 CR4、TR3～TR5。

2)80486 新增指令

(1)字节交换指令 BSWAP。其指令格式如下:

```
BSWAP  r32
```

该指令反转 32 位(目标)寄存器的字节顺序:0～7 位与 24～31 位交换,8～15 位与 16～23 位交换。提供此指令是为了将低位在先、高位在后的值转换成高位在先、低位在后的格式,或者正相反。例如,(EAX)=0123 4567H,执行指令 BSWAP EAX,使(EAX)=6745 2301H。

注意:要交换字值(16 位寄存器)中的字节,使用 XCHG 指令。BSWAP 指令引用 16 位寄存器时,结果未定义。

(2)交换加指令 XADD。其指令格式如下:

```
XADD  r32/m32, r32
XADD  r16/m16, r16
XADD  r8/m8, r8
```

该指令交换第一个操作数(目标操作数)与第二个操作数(源操作数),然后将这两个值的和加载到目标操作数。目标操作数可以是寄存器或内存地址;源操作数是寄存器。该指令按加法指令影响相应标志位。例如:

```
XADD BL, DL
```

如执行前(BL)=12H,(DL)=02H,则执行后(BL)=14H,(DL)=12H。

(3)比较交换指令 CMPXCHG。其指令格式如下:

```
CMPXCHG  r32/m32, r32
CMPXCHG  r16/m16, r16
CMPXCHG  r8/m8, r8
```

该指令将累加器 AL/AX/EAX 中的值与目的操作数比较,如果相等,将源操作数的值装载到目的操作数,(ZF)置 1;如果不等,将目的操作数的值装载到 AL/AX/EAX,并将(ZF)清 0。该指令按比较指令影响相应标志位。例如:

```
CMPXCHG CX,DX
```

如果指令执行前(AX)=2300H,(CX)=2300H,(DX)=2400H,则指令执行后因(CX)=(AX),故(CX)=2400H,(ZF)=1;如果指令执行前(AX)=2500H,(CX)=2300H,(DX)=2400H,则指令执行后因(CX)≠(AX),故(AX)=2300H,(ZF)=0。

(4)高速缓存无效指令 INVD。其指令格式如下:

```
INVD
```

该指令使处理器的内部缓存失效(清除),并发出一个专用总线周期,指示外部缓存也进行清除,内部缓存保存的数据不写回主内存。

执行此指令之后,处理器不等待外部缓存完成清除操作,而是继续执行指令,缓存清除信号的响应由硬件负责。INVD 指令是特权指令,处理器在保护模式中运行时,程序或过程的 CPL(当前进程的权限级别)必须是 0 才能执行此指令。另外,应谨慎使用此指令,因为内部缓存及未写回主内存的数据可能会丢失。除非有特殊要求或好处,才可以清除内存而不将修改的缓存线写回内存(例如,在测试或错误恢复场合,缓存与主内存的一致性无关紧要),否则软件应使用 WBINVD 指令。

(5)回写及高速缓存无效指令 WBINVD。其指令格式如下:

```
WBINVD
```

该指令写回处理器内部缓存中所有修改的缓存线,并使这些内部缓存失效(清除)。接着,指令发出一个特殊功能的总线周期,指示外部缓存写回修改的数据并发出另一个总线周期,指示应该使外部缓存失效。执行此指令之后,处理器并不等待外部缓存完成写回与清除操作,而是继续执行指令,缓存写回与清除信号的响应由硬件负责。WBINVD 指令是特权指令,处理器在保护模式中运行时,程序或过程的 CPL 必须是 0 才能执行此指令。在缓存与主内存是否一致无关紧要的情况下,软件可以使用 INVD 指令。

(6)TLB 无效指令 INVLPG。其指令格式如下:

```
INVLPG  mem
```

该指令的功能是使源操作数指定的变换旁查缓冲器(TLB)项目失效(清除)。源操作数是内存地址。处理器确定包含该地址的页,并清除该页的 TLB 项目。INVLPG 指令是特权指令,处理器在保护模式中运行时,程序或过程的 CPL 必须是 0 才能执行此指令。INVLPG 指令通常只清除指定的页的 TLB 项目,但在某些情况下,它会清除整个 TLB。

3)Pentium 新增指令

(1)8 字节比较交换指令 CMPXCHG8B。其指令格式如下:

CMPXCHG8B　m64

该指令比较 EDX:EAX 中的 64 位值与操作数(目标操作数),如果这两个值相等,则将 ECX:EBX 中的 64 位值存储到目标操作数,否则,将目标操作数的值加载到 EDX:EAX。目标操作数是 8 字节内存位置。对于一对 EDX:EAX 与 ECX:EBX 寄存器,EDX 与 ECX 包含 64 位值的 32 个高位,EAX 与 EBX 包含 32 个低位。

(2)处理器识别指令 CPUID。其指令格式如下:

CPUID

该指令是 Intel IA32 架构下获得 CPU 信息的汇编指令,可以得到 CPU 类型、型号、制造商信息、商标信息、序列号、缓存等一系列与 CPU 相关的信息。这条指令涉及的内容极其丰富,Intel 有一个超过 100 页的文档,专门介绍该指令。

CPUID 指令使用 EAX 作为输入参数,使用 EBX、ECX、EDX 作为输出参数。如果程序可以改变 EFLAGE(扩展标志寄存器)的第 21 位,那么 CPUID 有效,否则无效。

把(EAX)=0 作为输入参数,可以得到 CPU 的制造商信息。CPUID 指令执行以后,会返回一个 12 字符的制造商信息,前 4 个字符的 ASCII 码按低位到高位放在 EBX,中间 4 个放在 EDX,最后 4 个字符放在 ECX。例如,对于 Intel 的 CPU,会返回一个“GenuineIntel”的字符串,(EBX)=756E6547H,(EDX)= 49656E69H,(ECX)= 6C65746EH。

把(EAX)=1 作为输入参数,可以得到处理器签名和功能位。CPUID 指令执行完成后,处理器签名放在 EAX 中,功能位及其他内容分别放在 EBX、ECX 和 EDX 中。通过处理器签名,可以确定 CPU 的具体型号。处理器签名中的 12 位和 13 位返回的是处理器类型(00 表示以前的 OEM 处理器,01 表示 OverDrive® 处理器,10 表示多处理器)。在 EDX 和 ECX 中返回的功能标志表明该 CPU 所支持的功能,EDX 定义可参看 Intel 相关资料。

注意:有时从处理器签名上仍然不能识别 CPU,如根据 Intel 提供的资料,Pentium II (Model 5)、Pentium II Xeon(Model 5)和 Celeron®(Model 5)的处理器签名完全一样,要区别它们只能通过检查其高速缓存(cache)的大小。如果没有高速缓存,就是 Celeron® 处理器;如果 L2 高速缓存为 1 MB 或 2 MB,则应该是 Pentium II Xeon 处理器;其他情况则应该是 Pentium II 处理器或是只有 512 KB 高速缓存的 Pentium II Xeon 处理器。

此外,有些情况下也可以使用 Brand ID(在 EBX 的 bit7:0 返回)来区分,如 Pentium III (Model 8)、Pentium III Xeon(Model 8)和 Celeron®(Model 8)3 种处理器的处理器签名是一样的,但它们的 Brand ID 不同。

把(EAX)=2 作为输入参数,可以得到处理器高速缓存描述符。执行完 CPUID 指令后,高速缓存描述符和 TLB 特性将在 EAX、EBX、ECX 和 EDX 中返回,每个寄存器中的 4 字节分别表示 4 个描述符,描述符中不同的值表示不同的含义。其中,EAX 中的最低 8 位 AL 的值表示要得到完整的高速缓存的信息,需要执行(EAX)=2 的 CPUID 指令的次数;同时,寄存器的最高位(bit 31)为 0,表示该寄存器中的描述符是有效的。

把(EAX)=3 作为输入参数,可以得到处理器序列号,只有部分 CPU 提供该功能。查看处理器是否支持处理器序列号功能,可以执行(EAX)=1 的 CPUID 指令,然后查看 EDX 的 PSN 标志位(bit 18),如果为 1,说明处理器可以返回序列号,否则不支持序列号功能或者是序列号功能被关闭了。处理器序列号一共 96 位,最高 32 位就是处理器签名,通过执行(EAX)=1 的 CPUID 指令获得,其余的 64 位在执行(EAX)=3 的 CPUID 指令后,中间 32

位在 EDX 中，最低 32 位在 ECX 中。AMD 所有的 CPU 都不提供处理器序列号功能。

除了上面介绍的功能外，EAX 还可以使用其他参数作为输入，实现其他功能，相关知识可参看 Intel 相关资料。

(3)读时间标签计数器指令 RDTSC。其指令格式如下：

```
RDTSC
```

该指令将处理器的时间标签计数器的当前值加载到 EDX:EAX 寄存器。时间标签计数器包含在 64 位 MSR 中。MSR 的高 32 位加载到 EDX 寄存器，低 32 位加载到 EAX 寄存器。处理器每时钟周期递增时间标签计数器 MSR 一次，在处理器复位时将它重设为 0。

寄存器 CR4 中的时间标签禁用(TSD)标志限制 RDTSC 的使用。清除 TSD 标志时，RDTSC 指令可以在任何特权级别执行；设置此标志时，该指令只能在特权级别 0 执行。在特权级别 0 执行时，时间标签计数器还可以使用 RDMSR 指令读取。

RDTSC 指令不是序列化指令，因此在读取计数器之前，不需等到前面的所有指令都已执行。类似地，在执行读取操作之前，后面的指令也可以开始执行。

(4)读模型专用寄存器指令 RDMSR。其指令格式如下：

```
RDMSR
```

该指令将 ECX 寄存器指定的 64 位型号专用寄存器(MSR)的内容加载到寄存器 EDX:EAX。EDX 寄存器中加载 MSR 的高 32 位，EAX 寄存器中加载低 32 位。在读取的 MSR 中，如果实现的位数小于 64，则 EDX:EAX 中未实现的位的值未定义。

此指令必须在特权级别 0 或实模式中执行，否则会生成一般保护性异常＃GP(0)。在 ECX 中指定保留的或未实现的 MSR 地址，也会导致一般保护性异常。

MSR 控制用于测试、执行跟踪、性能监视及机器检查错误。使用 RDMSR 指令之前，应该使用 CPUID 指令确定是否支持 MSR(EDX[5]=1)。

(5)写模型专用寄存器指令 WRMSR。其指令格式如下：

```
WRMSR
```

该指令功能与 RDMSR 相反。

(6)系统管理方式返回指令 RSM。其指令格式如下：

```
RSM
```

该指令将程序控制权从系统管理模式(SMM)返回给处理器收到 SMM 中断时被中断的应用程序或操作系统过程。处理器的状态从进入 SMM 时创建的转储信息中还原。在状态还原过程中，如果处理器检测到状态信息无效，则进入关闭状态。

4)Pentium Pro 新增指令

(1)条件传送指令 CMOV。其指令格式如下：

```
CMOVxx  r16,r16/m16  ;若条件 xx 成立，则 r16←r16/m16;否则，不传送
CMOVxx  r32,r32/m32  ;若条件 xx 成立，则 r32←r32/m32;否则，不传送
```

该指令首先判断条件是否满足。如果条件成立，则发生传送，将源操作数传送到目的操作数；如果条件不成立，则不进行传送，就好像没有执行该指令一样。该指令的条件 xx 同 Jxx 和 SETxx 指令，参见表 3-3。利用条件传送指令可以代替条件转移指令，从而减少程序分支，提高处理器性能。例如，一个典型的单分支程序段：

```
TEST   ECX,ECX      ;判断(ECX)是否等于 0
```

```
JNE    IH
MOV    EAX,EBX          ;(ECX)=0,则(EAX)←(EBX)
IH:    ...              ;(ECX)≠0
```

如果采用条件传送指令,则可以优化为:

```
TEST    ECX, ECX       ;CMOVEQ 有条件地将(EBX)传送到(EAX)
CMOVEQ  EAX, EBX       ;可以代替 JNE 与 MOV 指令,从而消除分支
```

(2)读性能监控计数器指令 RDPMC。其指令格式如下:

```
RDPMC
```

该指令将由 ECX 指定的 40 位性能监控计数器的当前值加载到 EDX:EAX 寄存器,其中高 8 位加载到 EDX 寄存器,低 32 位加载到 EAX 寄存器。

(3)无定义指令 UD2。其指令格式如下:

```
UD2
```

该指令将产生一个无效操作码异常,通常用于软件测试。

虽然这些新增指令功能非常强大实用,但是 MASM 在默认情况下只接受 8086 指令集。如果用户使用了 80186 及以后微处理器新增的指令,就必须使用处理器选择伪指令,如表 3-4 所示。

表 3-4　处理器选择伪指令及其功能说明

伪 指 令	功　　能	伪 指 令	功　　能
.8086	仅接受 8086 指令(默认状态)	.586	接受除特权指令外的 Pentium 指令
.186	接受 80186 指令	.586P	接受全部 Pentium 指令
.286	接受除特权指令外的 80286 指令	.686	接受除特权指令外的 Pentium Pro 指令
.286P	接受全部 80286 指令,包括特权指令	.686P	接受全部 Pentium Pro 指令
.386	接受除特权指令外的 80386 指令	.MMX	接受 MMX 指令
.386P	接受全部 80386 指令,包括特权指令	.K3D	接受 AMD 处理器的 3D 指令
.486	接受除特权指令外的 80486 指令,包括浮点指令	.XMM	接受 SSE 指令和 SSE2 指令
.486P	接受全部 80486 指令,包括特权指令和浮点指令		
.387	接受 80387 数学协处理器指令		
.No87	取消使用协处理器指令		

注:.586 和.586P 是 MASM 6.11 引入的;.686,.686P 和.MMX 是 MASM 6.12 引入的;.K3D 是 MASM 6.13 引入的;.XMM 是 MASM 6.14 引入的,MASM 6.15 才支持 SSE2 指令。

习　　题

(1)试根据以下要求写出相应的汇编语言指令。

①把 BX 寄存器和 CX 寄存器的内容相加,结果存入 CX 寄存器中。

②用寄存器 BX 和 SI 的基址变址寻址方式把存储器中的一个字节与 CL 寄存器的内容相加，并把结果送到 CL 寄存器中。

③用寄存器 BX 和位移量 0A5H 的寄存器相对寻址方式把存储器中的一个字和(AX)相加，并把结果送回存储器中。

④用位移量为 0137H 的直接寻址方式把存储器中的一个字与数 4B68H 相加，并把结果送回该存储单元中。

⑤把数 0E2H 与(BL)相加，并把结果送回 BL 中。

(2)现有(DS)＝2000H，(BX)＝0100H，(SI)＝0002H，(20100H)＝12H，(20101H)＝34H，(20102H)＝56H，(20103H)＝78H，(21200H)＝2AH，(21201H)＝4CH，(21202H)＝B7H，(21203H)＝65H，试说明下列各条指令执行完后 AX 寄存器的内容。

①MOV AX，1200H

②MOV AX，BX

③MOV AX，[1200H]

④MOV AX，[BX]

⑤MOV AX，1100[BX]

⑥MOV AX，[BX][SI]

⑦MOV AX，1100[BX][SI]

(3)设当前数据段寄存器的内容为 1400H，在数据段的偏移地址 3000H 单元内含有一个内容为 0FC20H 和 8000H 的指针，它们是一个 16 位变量的偏移地址和段地址，试写出把该变量装入 AX 的指令序列，并画图表示出来。

(4)在 ARRAY 数组中依次存储了 7 个字数据，紧接着是名为 ZIAREA 的字单元，如下：

```
ARRAY    DW 45，48，7，200，23000，55，0
ZIAREA   DW ?
```

①如果 BX 包含数组 ARRAY 的初始地址，编写指令将数据 0 传送给 ZIAREA 单元。

②如果 BX 包含数据 0 在数组中的位移量，编写指令将数据 0 传送给 ZIAREA 单元。

(5)如 TABLE 为数据段中 0035 单元的符号名，其中存放的内容为 5678H，试问以下两条指令执行完后，AX 寄存器的内容是什么？

①MOV AX，TABLE

②LEA AX，TABLE

(6)执行下列指令后，AX 寄存器中的内容是什么？

```
TABLE DW 50，40，30，20，10
ENTRY DW 4
…
MOV BX，OFFSET TABLE
ADD BX，ENTRY
MOV AX，[BX]
```

(7)下列 ASCII 码串(包括空格符)依次存储在起始地址为 CSTRING 的字节单元中：

```
CSTRING DB 'TABLE ADDRESSING'
```

编写指令将字符串中的第 1 个和第 7 个字符传送给 DX 寄存器。

(8)已知 SS 寄存器的内容是 0FFA0H，SP 寄存器的内容是 00B0H，先执行两条把 8057H 和 0F79H 分别进栈的 PUSH 指令，再执行一条 POP 指令。试画出堆栈区和 SP 的内容变化过程示意图。

(9)求出以下各十六进制数与十六进制数 62A0 之和，并根据结果设置标志位 SF、ZF、CF 和 OF 的值。

①1234

②4321

③CFA0

④9D60

(10)写出执行以下算式的指令序列，其中 A、B、C、D、E 均为存放 16 位带符号数单元的地址。

①C＊E＋(C－A)

②C＊E－(A＋6)－(D＋9)

③C(E＊A)/(B＋6)，余数放在 D 中

(11)写出对存放在 DX 和 AX 中的双字长数求补的指令序列。

(12)试编写一个程序求出双字长数的绝对值。双字长数在 A 和 A＋2 单元中，结果存放在 B 和 B＋2 单元中。

(13)写出完成以下操作的程序段，设各变量的值均为压缩 BCD 码表示的两位十进制数。

①A＊B＋(C－6)

②A(B＋C)－(D－A)

(14)假设(BX)＝0F5H，变量 VALUE 中存放的内容为 68H，确定下列各条指令单独执行后 BX 的值。

①XOR BX，VALUE

②AND BX，VALUE

③OR BX，VALUE

④XOR BX，0FFH

⑤AND BX，0

⑥TEST BX，01H

(15)试分析下面的程序段完成的功能。

```
MOV CL，04
SHL DX，CL
MOV BL，AH
SHL AX，CL
SHR BL，CL
OR  DL，BL
```

(16)试写出程序段把 DX:AX 中的双字右移 4 位。

(17)REP 前缀的作用是什么？能用指令“REP LODSB”读取 DS:SI 所指内存中的每个

字符来进行处理吗？若不能，请说明原因。

(18)编写指令序列，在字符串 LIST 中查找字符“B”，若找到，则转向 Found，否则转向 NotFound，假设该字符串含有 200 个字符。

(19)编写指令序列，将 32 位数 AX:BX 中的 8 位 BCD 与 CX:DX 中的 8 位 BCD 相加，并把所得结果存入 CX:DX 中。