第3章 chapter 3

汇编语言指令集

指令是CPU操作的基本单位，指令集是所有指令构成的集合。计算机程序由指令构成，软件技术的发展增加了计算机程序的复杂性，从而推动了指令集的发展。现代计算机系统的指令集通常由完成基本操作的基本指令集和支撑新兴软硬件技术的高级指令集构成。本章讲述基于80x86系统结构的汇编语言指令集，在讲述与指令相关的概念和知识的基础上，重点讲解汇编语言基本指令集，并对汇编语言高级指令集进行介绍。

第3章 导语

第3章 课件

3.1 概 述

3.1.1 指令集体系结构、机器指令和符号指令

指令集概述

在计算机技术中，指令是指在某种计算机结构中定义的单个CPU操作，每条指令执行一个特定的操作，如将两个数相加。在这种计算机结构中，CPU支持的所有指令构成的集合称为指令集。

将指令编码成为二进制格式的序列，称为机器指令。通常CPU只能识别和执行机器指令。由于在不同的计算机结构中，CPU设计不相同，从而使得相同的指令被编码成为的机器指令不相同。人们将指令集和指令集编码（即指令集对应的机器指令集）称为CPU的指令集体系结构（Instruction Set Architecture，ISA）。各个CPU厂商设计的ISA可能各不相同，如Intel公司设计的IA32系统结构，俗称80x86。本章将重点讲述80x86指令集，包括汇编语言基本指令集和高级指令集。

最常见的指令集体系结构包括精简指令集计算机（Reduced Instruction Set Computing，RISC）和复杂指令集计算机（Complex Instruction Set Computing，CISC）。

RISC 体系中常见的 CPU 是 IBM 公司的 PowerPC、Oracle 公司的 SPARC、ARM 公司的 ARM 等，对应的主要操作系统是 UNIX 以及 iOS、Android 等移动操作系统。CISC 体系中常见的 CPU 是 80x86 架构，如 Intel 公司的 Pentium 等，AMD 公司的 Athlon 等以及 Cyrix 的部分处理器，对应的主要操作系统是 Microsoft Windows 以及 Linux。除特别说明以外，本书中的汇编语言程序的结构和运行均基于 80x86 结构下 Microsoft Windows 操作系统。

早期计算机的程序设计大多直接使用机器指令。程序员设计好程序后，将构成程序的机器指令对应的二进制序列通过打孔卡片或打孔纸带的方式输入计算机中，程序在计算机中运行完毕后得到处理结果。这一过程中的程序的书写和输入过程很烦琐，容易出错。在 20 世纪 70 年代微型计算机发展的早期阶段，虽然指令的输入方法改进为通过拨码开关输入，但无法降低直接采用机器指令编写程序带来的难度，直到后来符号指令的诞生才解决了这一问题。

符号指令定义了一套特定的助记符和书写格式，把指令表示成为字符串形式的序列。与机器指令相比，符号指令更容易记忆、书写和阅读，输入也不容易出错。目前在编写汇编语言程序时一般都采用符号指令。在指令集体系结构中，每条指令都有对应的机器指令和符号指令，如表 3.1 所示。

表 3.1 IA32(80x86)中的指令、机器指令和符号指令

指 令	机器指令，括号内为等值的十六进制序列	符 号 指 令
1234H→AX	10111000001101000010010(B83412)	MOV AX,1234H
AX+BX→AX	000000111000011(03C3)	ADD AX,BX

表 3.1 中分别列举了两条指令和它们对应的机器指令以及符号指令。第一条指令表示把数值 1234H 传送到 AX 寄存器中，符号指令“MOV AX, 1234H”采用英文符号 MOV 作为助记符，传送的数据 1234H 和传送的目的地 AX 书写在助记符的后面，与机器指令对应的 10111000001101000010010 二进制序列比较，符号指令更容易编写和输入。同样，用符号指令“ADD AX,BX”将 AX 寄存器和 BX 寄存器中的数相加，结果放入 AX 寄存器中。

【例 3.1】 用机器指令和符号指令分别编写功能相同的程序，在文本屏幕上输出十进制数 1 和 2 相加的结果 3。

机器指令程序(二进制格式)：

```
1011001000000001
10000000110000100000010
10000000110000100110000
1011010000000010
1100110100100001
1011010001001100
1100110100100001
```

符号指令程序：

```
CODE SEGMENT
ASSUME CS:CODE
START:MOV DL,1
      ADD DL,2
      ADD DL,30H
      MOV AH,02H
      INT 21H
      MOV AH,4CH
      INT 21H
CODE ENDS
END START
```

从例 3.1 中可以看到，与直接使用机器指令相比，使用符号指令不容易出错并且便于阅读。除非有特别需求，通常人们都使用符号指令来编写汇编语言源程序。在源代码程序编译为机器代码程序的过程中，通过汇编工具程序将符号指令转换为对应的机器指令。

3.1.2 符号指令的书写格式

在 80x86 指令集中，符号指令的书写格式定义如下：

标号：	操作码助记符	空格	操作数助记符（多个操作数之间用“,”隔开）	;注释

格式说明如下。

（1）符号指令的核心部分是操作码和操作数。操作码表示指令的功能，操作数表示指令的操作对象，分别采用操作码助记符和操作数助记符表示。例如，指令“ADD AX, BX”的操作码助记符为 ADD，表示这是一条加法指令，AX 和 BX 中存放的数为加法运算的操作数。操作码和操作数之间必须用空格符或 Tab 符隔开。有的指令的操作数是隐含操作数，不显式地在指令中表示，此时指令书写时只包含操作码助记符这个部分，例如对 C 标志置 1 的指令 STC。在 Microsoft Windows 操作系统中编写汇编程序时，对指令的操作码助记符以及操作数助记符的大小写并未加以严格限制，可以选择大写或小写字母书写。

（2）有一个操作数的指令称为单操作数指令，有两个及两个以上操作数的指令称为多操作数指令，多个操作数助记符之间需要用“,”隔开。有两个操作数时，位于“,”左边的操作数称为目标操作数，位于“,”右边的操作数称为源操作数。例如，在指令“ADD AX, BX”中，AX 中存放的数为目标操作数，BX 中存放的数为源操作数。另外，多字节操作数在存储器中的存放顺序依据小端法（Little Endian）规则，即低位字节存放在低地址单元，高位字节存放在高地址单元。

（3）标号表示该条指令的符号地址。当该条指令被作为分支或循环等指令的转移目标或作为程序开始执行的首条语句时，需要设置标号，其他情况下则可以忽略。标号和操作码助记符之间需要用“:”隔开。标号的命名规则：开头的符号需要是字母或下画

线，其余符号可以是字母、数字和下画线等，全部符号构成标号名，其长度应不超过 31 个字符。特别注意：指令的操作码助记符、伪指令助记符、CPU 中的寄存器名称等系统保留字不能作为标号的名称。

(4) 注释是在程序设计中对指令等信息的附加说明，可以忽略不写。使用“;”引起一行注释。注释仅用于在源程序中为程序员提供附加信息，在程序编译成为机器代码时将会被编译器忽略，不会出现在目标机器代码中，也不会被 CPU 执行。

(5) 一条符号指令对应的机器指令一般由若干字节构成，在存储器中连续存放。指令在存储器中占用的字节数称为指令长度，长度为一字节的指令称为单字节指令，长度为一字节以上的指令称为多字节指令。其中，第一字节所在存储器单元的地址称为指令地址。例如，符号指令“ADD AX，BX”对应的机器指令为 0000001111000011，在存储器中占用 2B，其指令长度为 2B，假设首字节单元在存储器中的物理地址为 12345H，则该条指令的物理地址为 12345H。

符号指令举例：

```
NEXT:  MOV CH, 00H      ;将立即数 00H 送入 CH 寄存器
ADD    AX, BX           ;AX 寄存器中的数与 BX 寄存器中的数相加后和送入 AX 寄存器中
INC    BYTE PTR [BX]    ;把 BX 间接寻址的内存单元中的数加 1
```

3.2 操 作 数


操作数

指令由操作码和操作数两部分构成。操作码表示指令的功能，操作数表示指令的操作对象，包括输入数据(状态)和输出数据(状态)。在计算机硬件中，这些数据(状态)被存放在 3 个区域：CPU 的寄存器、计算机的存储器以及计算机接口电路中的端口。掌握操作数在计算机硬件系统中的存放位置和存放方法是学习汇编语言指令的基础。本书第 2 章对 32 位微处理器的寄存器组织进行了概要介绍，本节讲述基本体系结构寄存器(包括通用寄存器、段寄存器、指令指针寄存器和标志寄存器，参见 2.3.3 节)和操作数的关系。其他用于存放操作数的硬件，即计算机的存储器和计算机接口电路中的端口，将在第 6 章和第 7 章中分别讲述。

3.2.1 通用寄存器中的操作数

32 位微处理器一共有 8 个 32 位的通用寄存器。指令的操作数存放在这些寄存器中时称为寄存器操作数。寄存器操作数的字长由寄存器的名称决定。以 EAX 为例，如图 3.1 所示，EAX 是一个长度为 32 位的寄存器，可以存储 32 位字长的操作数。当使用 AX 作为寄存器名称时，只能使用 32 位寄存器的低 16 位部分，此时 AX 是一个 16 位寄存器，可以存储 16 位字长的操作数。而使用 AH 作为寄存器名称时，只能使用 AX 的高 8 位部分，此时 AH 是一个 8 位寄存器。同样，使用 AL 作为寄存器名称时，只能使用 AX 的低 8 位部分，此时 AL 也是一个 8 位寄存器。AH 和 AL 都可以存储 8 位字长的操作

数。通常从 D_0 开始由右向左依次编号标识寄存器中的二进制位，最右边的位称为最低位，最左边的位称为最高位。例如，EAX 的最低位为 D_0，最高位为 D_{31}。AH 的最低位为 D_8，最高位为 D_{15}。同时把 AH 称为 AX 的高位字节单元，AL 称为 AX 的低位字节单元，AX 称为 EAX 的低位字单元。

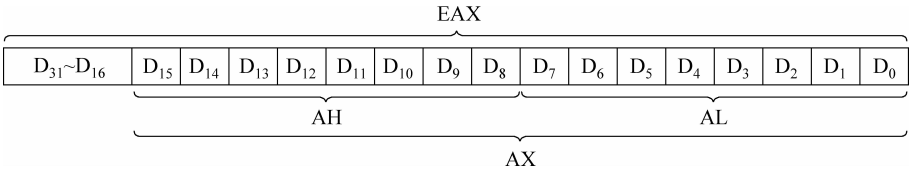


图 3.1　EAX 寄存器不同字长的逻辑结构

EBX、ECX、EDX 的寄存器名称和操作数字长的对应方法与 EAX 相同。但 ESP、EBP、EDI、ESI 这 4 个寄存器的逻辑结构只有 32 位和 16 位字长的两种类型，16 位的逻辑结构名称分别为 SP、BP、DI 和 SI，因此存放的操作数只有 32 位和 16 位两种字长。

此外，当操作数位于存储器中时，通用寄存器中的一些特定寄存器被用于存放寻找操作数所需的基地址或偏移地址，如 BX、SI、DI 等，将在 3.3 节寻址方式中讲述。

3.2.2　段寄存器和指令指针寄存器

32 位微处理器中一共有 6 个 16 位的段寄存器和 1 个 32 位的指令指针寄存器。代码段寄存器 CS 以及指令指针寄存器 IP 分别存放 CPU 将要取出的指令的段基址和偏移地址。

实模式下对存储器采用的管理方式是段式管理。程序中的代码一般被放在代码段中，当前 CPU 将要读取的指令，其存储单元的逻辑地址中的段基址存放在代码段寄存器 CS 中，其偏移地址则存放在指令指针寄存器 IP 中。

【例 3.2】　实模式下，CPU 将要读取的指令在存储器中的逻辑地址为 1000H：2345H，分析 CPU 读取指令的过程。

如图 3.2 所示，指令“ADD AX，BX”是 CPU 将要取出的指令，其逻辑地址是 1000H：2345H。此时 CPU 中的代码段寄存器 CS 中段基址为 1000H，指令指针寄存器 IP 中偏移地址为 2345H，通过地址合成器计算出指令的物理地址为 12345H，CPU 将该地址通过地址总线发送给存储器并同时发出读控制信号，存储器则通过数据总线将指令发送给 CPU 进行指令译码。一条指令读取完成后，IP 中的偏移地址将自动增加，例中 IP 值将变为 2347H，对应下一条将要取出的指令“SUB CX，DX”。从上述过程中可以看到，代码段寄存器 CS 以及指令指针寄存器 IP 决定了 CPU 将要取出的指令。修改 CS 和 IP 的值则可以改变 CPU 取出的指令，在程序设计中可以实现程序控制转移，例如分支和循环。汇编语言指令集中的转移以及调用指令可以修改 CS 和 IP 的值，实现程序的控制转移。

当操作数存放在存储器中时，数据段 DS、堆栈段 SS 以及附加段（ES、FS 和 GS）用于存放 CPU 取出该操作数所需的段基址。访问存储器操作数的过程与从代码段中取出指令相似，但偏移地址的获取更复杂，根据不同的寻址方式有不同的偏移地址计算方法。

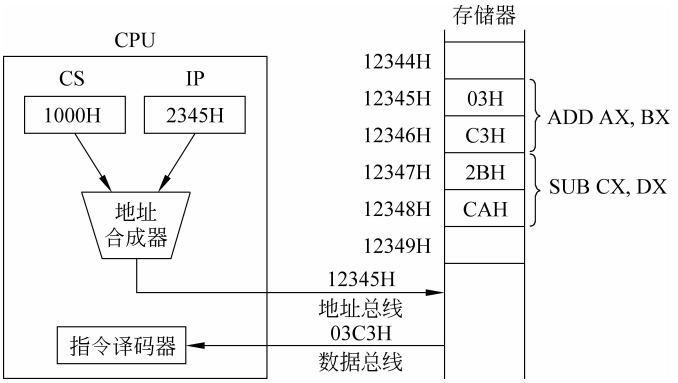


图 3.2 CPU 从存储器中读取指令的过程示意图

具体方法将在 3.3 节寻址方式中讲述。

在保护模式下,每个应用程序的整个 4GB 线性地址空间都作为一个段。代码段和数据段/堆栈段的空间是统一的,都是 00000000H～FFFFFFFFH。在这个 4GB 的地址空间中,一部分用来存放程序,一部分作为数据区,一部分作为堆栈,另外还有一部分被系统使用。这些部分的地址区域是不重合的。

同时,在保护模式下,操作系统不仅已经预先为要运行的用户应用程序的代码段、数据段和堆栈段设置好描述符,规定这些段的段基址都为 0,段界限都为 FFFFFFFFH。而且程序开始执行时,CS、DS、ES、SS 中存放的选择子已经指向正确的描述符,程序员不需要给这些段寄存器赋值。在整个程序运行期间,程序员也不应该修改这些段寄存器的值。因为,操作系统为了保证系统的安全性,是不允许用户对描述符表和页表等进行写操作的。所以,使用高版本汇编编写保护模式应用程序时,不需要创建段描述符,也不需要创建段描述符表,与实模式的汇编相比,对内存数据的访问更加方便。

3.2.3 标志寄存器

指令的操作对象除了数据外还包括状态。在大多数情况下,使用标志寄存器中的标志位来存储状态。标志位分为两种类型:状态标志和控制标志。状态标志用于作为某些指令操作的前提状态以及指令操作完成后的结果状态。控制标志可以设定 CPU 的某些功能,例如中断;或者设定指令的操作功能,例如串操作指令。控制标志值可以用相应指令进行设定。

80x86 的标志寄存器(EFLAGS)是一个 32 位的寄存器,实际只使用到 15 个二进制位,一共有 14 个标志。16 位 CPU 中的标志寄存器只有 16 位,也称为程序状态字寄存器(PSW),有效的标志位共有 9 个(包括 O、D、I、T、S、Z、A、P、C)。为了保持兼容,80x86 的标志寄存器 EFLAGS 除了新增的 5 个标志外,其他标志的位置和名称不变,如图 3.3 所示。

	AC	VM	R		NT	IOPL	O	D	I	T	S	Z		A		P		C
D_{31}~D_{19}	D_{18}	D_{17}	D_{16}	D_{15}	D_{14}	$D_{13}$$D_{12}$	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

图 3.3 80x86 标志寄存器 EFLAGS

标志寄存器中的各个标志位都有确定的功能，各个标志位的功能如下。

1. C 标志（进位/借位标志）

加法/减法指令执行后，如果最高位产生进位/借位，则 C 标志置 1，否则 C 标志置 0。例如，字节加法运算，D_7 位产生进位，则 C 标志被设置为 1。除此以外，无符号数乘法指令、求补指令、移位指令和一部分逻辑运算指令在执行后对 C 标志也会产生影响。根据需要，也可以使用处理机控制指令 CLC 和 STC，将 C 标志直接设定为 0 和 1。

2. A 标志（辅助进位/借位标志）

A 标志也称为半进位/借位标志。在进行加法/减法运算时，如果 D_3 位向 D_4 位有进位/借位，则 A 标志置 1，否则 A 标志置 0。特别注意：当加法/减法运算为字以及双字运算时，A 标志的设置仅与最低一字节中的 D_3 位向 D_4 位的进位/借位有关。

3. S 标志（符号标志）

S 标志记录运算结果的最高位的位值。例如，字运算后 D_{15} 位为 1，则 S 标志被设置为 1，否则 S 标志被设置为 0。当参加运算的数为有符号数时，S 标志就是运算结果的符号位。

4. Z 标志（全零标志）

Z 标志表示运算结果是否为全零。当运算结果为全零时，Z 标志被设置为 1，否则 Z 标志被设置为 0。

5. P 标志（奇偶标志）

P 标志的设置依据运算结果的最低一字节中 1 的个数进行，当 1 的个数为偶数时，P 标志被设置为 1，否则 P 标志被设置为 0。特别注意：当运算结果是字以及双字时，P 标志仅与结果中的最低一字节有关。例如，字运算结果为 1101000010101010，P 标志被设置为 1。P 标志经常被用于在数据读写或通信传输中进行差错检测。

6. O 标志（溢出标志）

CPU 根据判溢电路对 O 标志进行设置，O 标志置 1 表示运算溢出，否则运算不溢出。

1）溢出和判断溢出的方法

运算结果超出目标寄存器或存储单元所能表示的数值的范围称为运算溢出。CPU 根据判溢电路产生的输出对 O 标志置 1 或置 0，称为 CPU 判溢。实际的运算操作数有无符号数和有符号数两种情形，是否溢出与操作数类型有关，需要在 CPU 判溢给出 O 标志以及运算给出 C 标志的基础上，结合操作数类型进行判溢，这种判溢方法称为程序员判溢。

2）CPU 判溢

CPU 判溢依据运算器中的判溢电路进行，如图 3.4 所示。

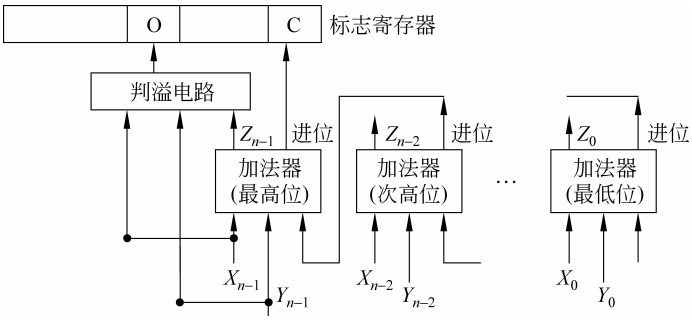


图 3.4 CPU 运算器中的判溢电路

CPU 的运算器可以完成两个字长为 n 的补码数的加法运算，设$[X]_{补}$为被加数，$[Y]_{补}$为加数，运算后得到和为$[Z]_{补}$。

$$[X]_{补} = X_{n-1}X_{n-2}\cdots X_1X_0,\quad [Y]_{补} = Y_{n-1}Y_{n-2}\cdots Y_1Y_0,$$
$$[Z]_{补} = [X]_{补} + [Y]_{补} = Z_{n-1}Z_{n-2}\cdots Z_1Z_0$$

加法运算产生的进位存放在 C 标志中。判溢电路的输出存放在 O 标志中，判溢电路的输入分别为被加数 X_{n-1}、加数 Y_{n-1}以及和的最高位 Z_{n-1}，输出的 O 标志为 CPU 判溢的结果，O 标志置 1 表示溢出，否则不溢出。其逻辑表达式为

$$O = X_{n-1}\cdot Y_{n-1}\cdot \overline{Z_{n-1}} + \overline{X_{n-1}}\cdot \overline{Y_{n-1}}\cdot Z_{n-1}$$

O 标志和 C 标志一起作为程序员判溢的基础。

3）程序员判溢

依据操作数类型的不同，程序员判溢时，产生溢出有下面两种情形。

（1）字长为 n 位的两个无符号数相加，结果大于 2^n-1。

（2）字长为 n 位的两个有符号数相加，结果大于 $2^{n-1}-1$ 或小于 -2^{n-1}。

程序员判溢需要使用 O 标志和 C 标志并结合操作数类型进行。

（1）参加运算的操作数是无符号数，则检测运算后的 C 标志，C 标志为 1 表示溢出，否则不溢出。

（2）参加运算的操作数是有符号数，则检测运算后的 O 标志，O 标志为 1 表示溢出，否则不溢出。

【例 3.3】 将两个补码数$[X]_{补}$和$[Y]_{补}$相加，$[X]_{补}=(11001000)_2$，$[Y]_{补}=(11001000)_2$，计算加法运算的和$[Z]_{补}$，并给出运算完成后 C、A、S、Z、P、O 6 个状态标志的值，假设$[X]_{补}$和$[Y]_{补}$均为无符号数，判断是否溢出；若$[X]_{补}$和$[Y]_{补}$均为有符号数，判断是否溢出。

列出计算竖式如下：

$$\begin{array}{rr} [X]_{补} & 11001000 \\ +[Y]_{补} & +)\ 11001000 \\ \hline [Z]_{补} & 1\ 10010000 \end{array}$$

加法运算得到的和$[Z]_{补}=(10010000)_2$

依据 6 个状态的设置方法，分别有 C=1，A=1，S=1，Z=0，P=1，O=1·1·0+

0·0·1=0，O=0，表示 CPU 判溢的结果是没有溢出。

下面进行程序员判溢，如果$[X]_{补}$和$[Y]_{补}$均为无符号数，C=1，运算有溢出；如果$[X]_{补}$和$[Y]_{补}$均为有符号数，O=0，运算没有溢出。

7. D 标志（方向标志）

D 标志用于在串操作指令中控制字符串指针的调整方向，D=0 时，为增址型调整，即指针由低位地址向高位地址移动；D=1 时，为减址型调整，即指针由高位地址向低位地址移动。在执行串操作指令前，使用处理机控制指令 CLD 将 D 标志设置为 0，或使用 STD 将 D 标志设置为 1。

8. I 标志（中断允许标志）

I 标志用于控制 CPU 是否响应来自引脚 INTR 的可屏蔽中断请求。I 标志为 0 时，CPU 不响应可屏蔽中断请求；I 标志为 1 时，CPU 响应可屏蔽中断请求。使用处理机控制指令 CLI 和 STI 设置 I 标志，CLI 将 I 标志设置为 0，STI 将 I 标志设置为 1。

9. T 标志（陷阱标志）

T 标志用于控制 CPU 是否以单步方式执行指令。T 标志为 0，CPU 以连续方式执行指令；T 标志为 1，CPU 以单步方式执行指令，即每执行一条指令后产生一次单步中断，自动调用中断类型号为 1 的单步中断服务子程序。T 标志默认值为 0，需要启动单步操作时，可以通过逻辑运算指令将标志寄存器中的 T 标志位设置为 1。

10. IOPL 标志（I/O 特权级标志）

IOPL 标志有 00～11 四个值，分别对应 0～3 四个 I/O 特权级，其中 0 为最高级，3 为最低级。IOPL 标志用于在保护模式下的输入输出操作，只有在当前的特权级低于或等于 IOPL 设定的级别时，执行 I/O 指令才能保证不发生异常，否则将引发一个保护异常。

11. NT 标志（任务嵌套标志）

NT 标志用于设定任务执行是否可以嵌套，该标志仅在保护模式下可用。中断返回指令 IRET 执行根据 NT 标志的值分为两种情形：当 NT 标志为 0，表示当前任务内的返回，即使用在堆栈中保存的 EFLAGS、EIP 以及 CS 寄存器的值，执行正常的 IRET 中断返回；当 NT 标志为 1，表示是嵌套任务的返回，将通过任务切换实现中断返回。

12. R 标志（恢复标志）

R 标志表示是否接受调试故障，与调试寄存器配合使用。R 标志为 0 时表示接受调试故障；R 标志为 1 时表示拒绝调试故障。在一条指令正常执行完毕后，CPU 将 R 标志设置为 0。当 CPU 响应断点异常中断时，R 标志设置为 1，然后标志寄存器压栈，转入断点处理程序，此时即便遇到调试故障也不产生异常中断，断点处理程序结束后返回断点指令。

13. VM 标志(虚拟标志)

VM 标志表示 CPU 是否处于虚拟 8086 模式。当 CPU 处于保护模式时,VM 被设置为 1,则 CPU 转换为虚拟 8086 模式,否则 CPU 仍处于一般的保护模式。

14. AC 标志(对准检查标志)

AC 标志设定是否进行对准检查。当 AC 标志设置为 0 时,不进行对准检查;当 AC 标志设置为 1,且 CR0 寄存器的 AM 位也为 1 时,进行字、双字和四字的对准检查,要访问的内存操作数没有按照边界对准时,将引发异常中断,即访问字操作数应该从偶地址开始,访问双字操作数应该从 4 的整数倍地址开始,访问四字操作数应该从 8 的整数倍地址开始,否则就是没有对准,发生越界。

以上 14 个标志中,C、A、S、Z、P、O 是 6 个最常用的状态标志,它们记录当前指令执行完毕后输出的状态。这些状态多被用于转移指令,作为程序控制转移的前提条件。常用的控制标志有 D 标志(用于串操作指令)和 I 标志(用于控制对可屏蔽中断的响应)。

3.3 寻址方式


寻址方式

指令操作数存放于计算机硬件中的 CPU 寄存器、存储器和计算机接口电路中的端口,此外有的操作数被直接包含在指令中(严格意义上,直接包含在指令中的操作数作为指令的一个组成部分,在读入 CPU 前也被存放在存储器中,但这种情况比较特别,所以在寻址方式中单独作为操作数的一种存放位置),因此操作数一共有 4 种存放位置。

(1) 操作数作为本条指令的一部分,直接包含在指令中,这种操作数称为立即数。

(2) 操作数存放在 CPU 的寄存器中,这种操作数称为寄存器操作数。

(3) 操作数存放在计算机的存储器中,这种操作数称为存储器操作数或内存操作数。

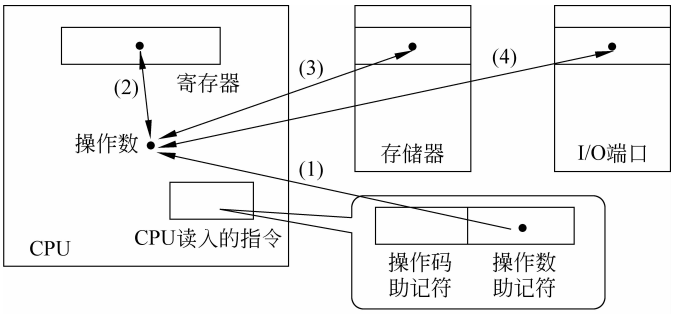
(4) 操作数存放在计算机接口电路的端口中,这种操作数称为 I/O 端口操作数。

寻址方式就是在指令中,使用特定的助记符或助记符表达式(地址表达式),告知 CPU 如何计算出操作数的地址,从而正确地存取操作数,以便进行后继的指令操作。通俗地讲,寻址方式就是 CPU 寻找到操作数的方式。

对应操作数的 4 种存放位置,如图 3.5 所示,4 种寻址方式分别为立即寻址、寄存器寻址、存储器操作数寻址和 I/O 端口操作数寻址。在 PC 系列机中,I/O 端口操作数被存放在独立的 I/O 空间中,对应寻址方式将在第 7 章中讲述,本章只讲述前 3 种操作数的寻址方式。

3.3.1 立即寻址

立即寻址:操作数包含在指令中,是指令的一个组成部分,CPU 读入该条指令后也



(1) 立即寻址; (2) 寄存器寻址; (3) 存储器操作数寻址; (4) I/O端口操作数寻址

图 3.5　操作数的 4 种存放位置和对应的寻址方式

就取得了操作数。采用立即寻址方式的操作数称为立即数。

下列指令都是将立即数传送到目标寄存器中，源操作数的寻址方式都是立即寻址。

```
MOV AL, 10101111B        ;指令执行后(AL)=AFH
MOV BX, 1234H            ;指令执行后(BX)=1234H
MOV CX, 0A234H           ;指令执行后(CX)=A234H
MOV AH,-2                ;指令执行后(AH)=FEH
MOV BL, 'A'              ;指令执行后(BL)=41H,这条指令等同于 MOV BL, 41H
MOV AX, 11*12            ;指令执行后(AX)=0084H,这条指令等同于 MOV AX, 132
```

说明：

(1) 实际编程中，立即寻址方式常用于给寄存器或内存操作数赋值，并且只能用于源操作数，不能用于目标操作数。

(2) 立即数的书写方法有 3 种。

① 立即数可以用不同进制的数值表示，以二进制形式给出时，在数值的后面加上 B 作为后缀；以十六进制形式给出时，在数值的后面加上 H 作为后缀，并且当开头为十六进制符号 A～F 时，需要在前面加上数字 0 作为前缀；以十进制形式给出时，在数值的后面加上 D 作为后缀(不加后缀也默认为十进制)；以八进制形式给出时，在数值的后面加上 Q 作为后缀。汇编程序在汇编时，不同进制的立即数一律汇编成为等值的二进制数，负数则用其补码表示。

② 立即数可以是用单引号括起来的字符，汇编后成为对应的 ASCII 码。

③ 立即数可以是用＋、－、*、/以及括号表示的算术表达式，汇编程序在生成机器指令时，将按照算术运算规则计算出的结果作为实际操作数。

3.3.2　寄存器寻址

寄存器寻址即将操作数存放在寄存器中，寄存器的名称在指令中的操作数中给出。在寄存器中存取操作数，可以获得较快的访问速度。

下列指令的源操作数或目标操作数的寻址方式为寄存器寻址。

```
MOV EAX, 12345678H   ;执行后(EAX)=12345678H,目标操作数为寄存器寻址
```

```
ADD CH, CL          ;将 CH 和 CL 中的数相加的和放入 CH,源、目两个操作数均为寄存器寻址
INC SI              ;将 SI 中的数加 1,操作数为寄存器寻址
```

3.3.3 存储器操作数寻址

存储器操作数寻址也称为内存操作数寻址,操作数存放在存储器中。正确地使用这种寻址方式需要理解 CPU 在存储器中存取数据的过程。

物理地址是存储器单元在物理空间中的编号,逻辑地址是存储器单元在分段式管理中的逻辑编号。由于程序被装载入存储器中的位置由操作系统在程序载入的时候决定,在编写程序时无法确定指令中使用的存储单元在程序载入存储器后的物理地址,只能使用逻辑地址来描述指令中用到的存储单元。CPU 在分析指令时使用内部的段页式管理部件将指令中的逻辑地址转换为对应的物理地址,再通过总线系统访问实际的物理存储单元。

区别于一维编号的物理地址,逻辑地址采用二维编号方式,在汇编语言中书写格式为

段寄存器名称:偏移地址表达式

其中,段寄存器名称也称为段超越前缀,表示存放操作数的存储单元所在的逻辑段。常用的逻辑段:代码段存放当前正在运行的程序的机器指令,段寄存器为 CS;数据段存放当前程序中使用的数据,段寄存器为 DS;堆栈段存放需要具有先进后出特性的数据,段寄存器为 SS;附加段也用于存放当前程序中使用的数据,段寄存器为 ES。段寄存器中存放逻辑段的段基址,在实模式下,段基址是用逻辑段段首单元的物理地址除以 16 后得到的商值。段基址描述了逻辑段在存储空间中的位置,用 16 位二进制数表示,范围为 0000H～FFFFH;偏移地址表达式给出偏移地址,也称为偏移量,表示存放操作数的存储单元在所处逻辑段中的位置,即该存储单元相对逻辑段段首单元的地址偏移量,用 16 位二进制数表示,范围为 0000H～FFFFH。CPU 将逻辑地址中的段基址乘以 16 后加上偏移地址就得到了操作数的物理地址,从对应的物理存储单元中存取操作数。

根据不同的应用场景,逻辑地址中的偏移地址表达式有 5 种不同的格式,分别对应 5 种存储器操作数寻址方法:直接寻址、寄存器间接寻址、基址寻址、变址寻址和基址加变址寻址。

1. 直接寻址

直接寻址方式的操作数的逻辑地址有两种书写格式。

1) 段寄存器名称:[偏移地址]

偏移地址表达式直接给出存储单元的偏移地址值。这种格式允许操作数存放在不同的逻辑段,段寄存器名称不可以省略,否则将出现寻址错误。在编写程序时,存储单元的偏移地址可以通过计算存储单元相对逻辑段首的偏移量得到,但手工计算比较烦琐容易出错,一般情况下不建议使用这种书写格式。

例如:

```
ADD AL, DS:[45H]      ;取出数据段中偏移地址为 0045H 的存储单元中的数据与 AL 相加
MOV AX, ES:[1000H]    ;取出附加段中偏移地址为 1000H 的存储单元中的数据送入 AX
```

2）段寄存器名称：变量名

汇编语言中可以使用伪指令为存储单元命名，即存储单元的变量名，也称为符号地址。汇编器将自动计算出该存储单元的偏移地址，这样在书写源程序时不再需要手工计算存储单元的偏移地址，简化了程序设计，直接寻址方式多采用这种书写格式。由于变量名中本身蕴涵了其所在逻辑段的名称，因此在书写逻辑地址时，段寄存器名称可以省略不写。例如：

```
MOV AX, DS:BUF        ;将数据段中名为 BUF 的存储单元中的数据送入 AX
```

这条指令也可以省略段寄存器名称，写为

```
MOV AX, BUF
```

实际编程中，直接寻址适用于存取单个内存操作数。

2. 寄存器间接寻址

寄存器间接寻址方式也称为间接寻址或间址，其逻辑地址的书写格式为

```
段寄存器名称:[间址寄存器]
```

偏移地址表达式给出的间址寄存器用于存放操作数的偏移地址。注意：只有一些特别指定的通用寄存器能够作为间址寄存器使用，如表 3.2 所示。

表 3.2　间址寄存器和约定访问的逻辑段

间址寄存器	约定访问的逻辑段	寻址位数/位
BP	堆栈段	16
BX、SI、DI	数据段	
EBP、ESP	堆栈段	32
EAX、EBX、ECX、EDX、ESI、EDI	数据段	

约定访问的逻辑段的含义：当间址访问的操作数位于的逻辑段就是间址寄存器约定访问的逻辑段时，逻辑地址中的段寄存器的名称可以省略不写，否则必须写出段寄存器名称。

例如：

```
MOV BX, BUF 单元的偏移地址
MOV AH, DS:[BX]          ;用 BX 间址取出数据段中 BUF 单元的操作数，将其送入 AH
```

操作数符合逻辑段约定访问规则，也可改写为

```
MOV AH, [BX]
```

实际编程中，寄存器间接寻址适合用于存取按一定规律连续存放在存储器中的多个

数据，如数据表格或数组。在间址寄存器中预先存放初始的存储单元的偏移地址，在存取数据过程中按照一定规律对间址寄存器中的偏移地址进行增量或减量运算，从而实现对多个数据的访问。将在第 4 章分支循环程序设计中学习这种设计方法。

3. 基址寻址

基址寻址方式的逻辑地址书写格式为

段寄存器名称：[基址寄存器＋位移量]

或

段寄存器名称：位移量[基址寄存器]

偏移地址表达式由基址寄存器和位移量的和构成。需要注意：只有一些特别指定的通用寄存器能够作为基址寄存器使用，如表 3.3 所示。

表 3.3 基址寄存器和约定访问的逻辑段

基址寄存器	约定访问的逻辑段	寻址位数/位
BP	堆栈段	16
BX	数据段	
EBP、ESP	堆栈段	32
EAX、EBX、ECX、EDX、ESI、EDI	数据段	

与间接寻址一样，在使用基址寻址时，也需要遵守逻辑段的约定访问规则。

例如：

```
MOV BX,BUF 单元的偏移地址
MOV AH, DS:[BX+3]          ;基址寻址取出数据段 BUF+3 单元中的操作数送入 AH
```

这条指令也可改写为

```
MOV AH, [BX+3]
```

实际编程中，与间接寻址相似，基址寻址也适合用于存取按一定规律连续存放在存储器中的多个数据，但由于增加了位移量参数，可以指定在数据表中访问的起始点，使用更加灵活。

4. 变址寻址

变址寻址方式的逻辑地址书写格式有两种。

(1) 有比例因子的变址寻址，逻辑地址书写格式为

段寄存器名称：[比例因子 * 变址寄存器＋位移量]

或

段寄存器名称：位移量[比例因子 * 变址寄存器]

偏移地址表达式由“比例因子＊变址寄存器＋位移量”构成。需要注意：只有一些特别指定的通用寄存器能够作为变址寄存器使用，如表 3.4 所示。

表 3.4　有比例因子变址寄存器和约定访问的逻辑段

变址寄存器	约定访问的逻辑段	寻址位数/位
EBP	堆栈段	32
EAX、EBX、ECX、EDX、ESI、EDI	数据段	

注意：比例因子只能是 1、2、4、8 中的一个数。

例如：

```
MOV EBX,BUF 单元的偏移地址
MOV AH, DS:[4 * EBX+3]    ;变址寻址取出数据段 4 * EBX+3 单元中的操作数送入 AH
```

这条指令也可改写为

```
MOV AH, [4 * EBX+3]
```

（2）无比例因子的变址寻址，逻辑地址书写格式为

```
段寄存器名称:[变址寄存器+位移量]
```

或

```
段寄存器名称: 位移量[变址寄存器]
```

在无比例因子的情形下，能够作为变址寄存器使用的通用寄存器如表 3.5 所示。

表 3.5　无比例因子变址寄存器和约定访问的逻辑段

变址寄存器	约定访问的逻辑段	寻址位数/位
SI、DI	数据段	16

注意：无比例因子的变址寻址只能使用 SI 或 DI 作为变址寄存器。

例如：

```
MOV SI,BUF 单元的偏移地址
MOV AH, SS:[SI+3]          ;变址寻址取出堆栈段 BUF+3 单元中的操作数送入 AH
```

实际编程中，与基址寻址相似，变址寻址也适合用于存取按一定规律连续存放在存储器中的多个数据，特别是带比例因子的寻址方式，可以在存取数据时跨越比较大的地址范围。

5. 基址加变址寻址

基址加变址寻址方式也称为基加变寻址方式，是基址寻址和变址寻址两种寻址方式的结合，根据是否带有比例因子，基址加变址寻址方式的逻辑地址也有两种书写格式。

(1) 有比例因子的基址加变址寻址,逻辑地址书写格式为

段寄存器名称:[基址寄存器+比例因子*变址寄存器+位移量]

或

段寄存器名称:位移量[基址寄存器][比例因子*变址寄存器]

有比例因子基址加变址寻址方式中的基址寄存器和变址寄存器分别与基址寻址方式中的基址寄存器以及有比例因子的变址寻址方式中的变址寄存器相同,并且都是32位的寄存器;在访问约定的逻辑段时,段寄存器名称也可以省略。

(2) 无比例因子的基址加变址寻址,逻辑地址书写格式为

段寄存器名称:[基址寄存器+变址寄存器+位移量]

或

段寄存器名称:位移量[基址寄存器][变址寄存器]

无比例因子基址加变址寻址方式中的基址寄存器和变址寄存器分别与基址寻址方式中的基址寄存器以及无比例因子的变址寻址方式中的变址寄存器相同,并且都是16位的寄存器;在访问约定的逻辑段时,段寄存器名称也可以省略。

实际编程中,基址加变址寻址方式由于有基地址和变址地址两个参数,特别适合表示二维下标,对二维数组进行访问。使用了位移量的基址加变址寻址方式,常用于对结构体数据进行访问,此时用基地址定位结构体,用位移量定位结构体中的数据项,用变址地址定位数据项中的每个元素。

3.3.4 寻址方式小结

(1) 寻址方式是CPU在执行指令时正确地找到操作数的方式。特别注意:寻址方式是针对指令中的操作数,并非针对指令,当指令有多个操作数时,各个操作数可能对应不同的寻址方式。例如:

```
MOV AX, 1234H    ;源操作数1234H是立即寻址方式,位于AX中的目标操作数是寄存器寻址
                 ;方式
```

(2) 操作数在指令中时是立即寻址方式。立即寻址方式只能用于源操作数,常用于给目标寄存器或存储器单元赋值。

(3) 操作数在寄存器中时是寄存器寻址方式。对比存储器操作数寻址,寄存器寻址方式能够得到较快的数据存取速度。

(4) 操作数在存储器单元中时是存储器操作数寻址方式。存储器操作数寻址方式在使用时需要注意两个问题。

① 逻辑地址表达式在约定访问逻辑段时出现的可以省略不写段寄存器名称的现象也称为段超越前缀省略。当地址表达式使用的寄存器符合段超越前缀省略的条件时,段寄存器名称可以写出,也可以不写出,但不写出是默认的汇编源程序的书写风格,可以让

程序代码简洁。但同时需要注意：在不满足段超越前缀省略的条件时，省略不写段寄存器会使指令出错。

② 5 种不同的存储器操作数寻址方式有各自的特点和应用场景：直接寻址方式用于访问单个内存操作数；寄存器间接寻址、基址寻址和变址寻址用于访问具有连续排列规律的多个内存操作数；基址加变址寻址多用于访问二维数据或数据结构体。实际编程中应该根据程序中的数据结构的特点正确选择相应的寻址方式。

3.4 汇编语言语法

3.4.1 汇编语言语句类型和格式

语句是汇编语言进行汇编和执行的单位。汇编语言源程序包括的语句类型为指令性语句和指示性语句。指令性语句即为通常所说的符号指令。指示性语句包括伪指令以及宏指令。本章主要讲解符号指令和伪指令。宏指令作为比较特殊的指示性语句，将在第 4 章讲述。

汇编语言语法

符号指令是指令的一种符号串格式的表现形式，与指令的二进制格式，即机器指令直接对应。汇编源程序中的符号指令将由汇编工具转换为对应的机器指令供 CPU 读取并解析执行；伪指令是为汇编工具提供汇编和连接信息的指令。这些信息将帮助汇编工具完成将汇编源程序编译成为目标机器程序的任务，如表 3.6 所示。

表 3.6 符号指令和伪指令

语句类型	符号指令（机器指令）	伪 指 令
执行者	CPU	汇编工具
功能	完成某个特定的 CPU 操作，例如，把两个寄存器中的数相加	为汇编工具提供信息，例如，计算出某个逻辑段的段基址

汇编语言语句的通用格式定义如下：

名字项	操作项	空格	操作数项（多个操作数之间用“,”隔开）	;注释

（1）当汇编语言语句类型为指令性语句时，格式为

标号：	操作码助记符	空格	操作数助记符（多个操作数之间用“,”隔开）	;注释

格式说明：

① 指令性语句的格式就是符号指令的书写格式，参见 3.1.2 节。

② 通用格式的名字项在指令性语句中对应符号指令的标号（加上“：”），操作项对应符号指令的操作码助记符，操作数项对应符号指令的操作数助记符。

(2) 当汇编语言语句类型为指示性语句时,格式为

变量	伪指令助记符	空格	操作数项(多个操作数之间用“,”隔开)	;注释

格式说明:

① 通用格式的名字项在指示性语句中对应变量,特别注意:变量名后没有“:”。

② 通用格式的操作项在指示性语句中对应伪指令助记符,其操作数个数和类型取决于不同的伪指令。

3.4.2 名字项

汇编语言语句的名字项有两种类型:标号和变量,也称为符号地址。标号和变量采用相同的命名规则:在命名中可以使用字母 A～Z(a～z)、数字 0～9 以及专用符号?、.、@、_、$。

注意:名字必须以除数字以外的字母或符号开头,符号“.”不能用于除开头以外的其他位置,名字长度不能超过 31 个字符,名字不能使用系统保留字,如指令助记符、寄存器名称等。

1. 标号

标号定义在代码段中,通常作为指令的转移目标。直接定义的标号后面必须加上“:”。也可以使用伪指令 LABEL 以及 EQU 定义标号。使用 PROC 定义的标号也可作为子程序名称或中断服务子程序名称。标号具有 3 个属性。

(1) 段属性:表示标号所在的代码段的段基址,该段基址存放在代码段寄存器 CS 中。

(2) 偏移属性:表示在代码段中,标号所在位置相对于段首的偏移地址。

(3) 类型属性:指出标号是段内转移的目标地址还是段间转移的目标地址。前者类型属性用 NEAR 表示,此时转移指令和标号处于同一个代码段;后者类型属性用 FAR 表示,此时转移指令和标号处于不同的代码段。类型属性不写出时默认为 NEAR。

在 DOS 汇编中,标号的作用域是整个程序,在整个程序中是唯一的。但在 Win32 汇编使用的高版本汇编中,标号的作用域是当前的子程序。在同一个子程序中的标号不能同名,但在不同的子程序中可以有相同名称的标号。

2. 变量

在 DOS 汇编中,变量定义在数据段、堆栈段以及附加段中,表示逻辑段中的存储单元。与标号相同,变量也具有 3 个属性。

(1) 段属性:表示存储单元所在逻辑段的段基址,该段基址存放在对应的逻辑段寄存器中,如 DS、SS 或 ES。

(2) 偏移属性:表示存储单元相对于段首单元的偏移地址。

(3) 类型属性:变量的类型属性有字节型、字型、双字型等。类型属性由数据定义伪

指令在定义变量的时候确定。例如，使用 DB 定义的变量的类型属性为字节型，使用 DW 定义的变量的类型属性为字型，使用 DD 定义的变量的类型属性为双字型。

需要注意：在一个源程序中，同一个名称的标号或变量只能出现一次，不能重名，否则汇编时将出现错误。

与 DOS 汇编相比，高版本汇编中变量的类型很多，如表 3.7 所示。

表 3.7　变量的类型

名　　称	表 示 方 式	缩　　写	长度/B
字节	BYTE	DB	1
字	WORD	DW	2
双字(DOUBLEWORD)	DWORD	DD	4
三字(FARWORD)	FWORD	DF	6
四字(QUADWORD)	QWORD	DQ	8
十字节(TENBYTE)	TBYTE	DT	10
有符号字节(SIGNBYTE)	SBYTE		1
有符号字(SIGNWORD)	SWORD		2
有符号双字(SIGNDWORD)	SDWORD		4
单精度浮点数	REAL4		4
双精度浮点数	REAL8		8
十字节浮点数	REAL10		10

根据变量的作用域可分为全局变量和局部变量。

1）全局变量

与 DOS 汇编相同，可用数据定义伪指令在.DATA 或.DATA? 段定义全局变量。全局变量的作用域是整个程序。

2）局部变量

局部变量的作用域是当前的子程序。定义的格式：

```
LOCAL 变量 1[重复数量 1][:类型],变量 2[重复数量 2][:类型]……
```

LOCAL 伪指令必须紧跟在子程序定义伪指令 PROC 之后，其他指令之前。如果定义的变量是 DWORD 类型，可以省略类型。局部变量不能和全局变量同名。例如：

```
LOCAL VAR1:WORD          //定义了一个字局部变量
LOCAL VAR2               //定义了一个双字局部变量
LOCAL VAR3[10]:BYTE      //定义了有 10 字节长的局部变量
```

在程序中使用局部变量，例如：

```
N1   PROC
```

```
    LOCAL VAR1:DWORD,VAR2:WORD
    LOCAL VAR3:BYTE
    MOV EAX,VAR1
    MOV AX,VAR2
    MOV AL,VAR3
    RET
N1  ENDP
```

局部变量被存放在堆栈空间。CPU 进入子程序时就会根据该子程序里面的局部变量所需的空间大小,在堆栈中留出相应大小的空间供局部变量使用。因此,局部变量无法初始化,只能在程序里用指令给它们赋值。

3.4.3 操作数项

汇编语言的语句中的操作数项包含两种类型的表达式:数值表达式和地址表达式。

1. 数值表达式

将标号、变量以及常量用数值运算符连接起来构成数值表达式,数值表达式汇编后的结果为数值。

1) 标号和变量

标号和变量除了作为名字项外,在汇编语句中也可以作为操作数项使用。

2) 常量

常量包含立即数、字符串常数和符号常数 3 种类型。

(1) 立即数。

立即数可以是不同的进制表示的数值,但需要加上相应的后缀。在汇编后,立即数均被转换为二进制数,负数则用其补码表示。

十进制立即数加上后缀 D(也可省略),例如 123D、123。

二进制立即数加上后缀 B,例如 10101010B。

十六进制立即数加上后缀 H,并且以 A～F 开头时必须加上前缀 0,例如 45H、0F004H。

八进制立即数加上后缀 Q,例如 112Q。

(2) 字符串常数。

字符串常数是用单引号括起来的一个或多个字符,在汇编后,字符串常数中的字符被转换为对应的 ASCII 码。例如,字符串'ABC',在汇编后转换为 414243H;字符串'12',在汇编后转换为 3132H。

(3) 符号常数。

符号常数是使用符号定义伪指令 EQU 或=定义的常数。将程序中经常使用到的数值定义为符号常数可以让源程序简洁,符号常数在汇编后被替换为原有的常数数值。例如:

```
NUM EQU 15                    ;NUM 定义为数值为 15 的符号常数
```