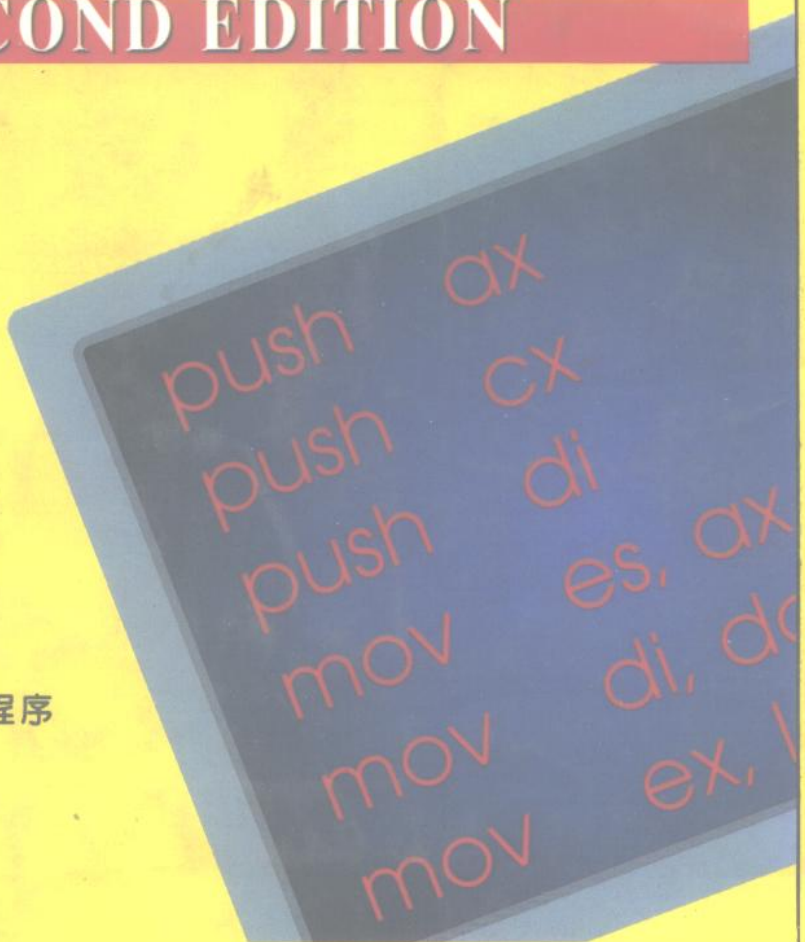
[美] Mark Goodwin 著
周予滨 田学锋 译
王 勇 校

DISK INCLUDED

汇编语言程序设计自学教程

SECOND EDITION



- 通过数以百计的汇编语言指令学习汇编语言程序设计
- 了解汇编语言子程序和高级语言的接口
- 为 80286/80386/80486 计算机创建 8088 汇编语言程序

MIS: PRESS

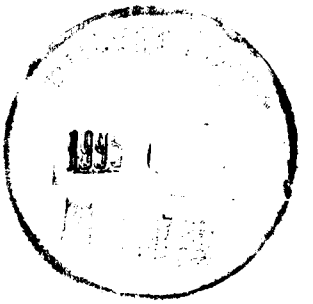
水利电力出版社

汇编语言程序设计自学教程

[美] Mark Goodwin 著

周予滨 田学锋 译

王 勇 校



水利电力出版社

1995

（京）新登字 115 号

内 容 提 要

汇编语言是重要的计算机低级语言。本书以简捷的语言风格剖析了汇编语言程序的基本结构、数据表示法、指令集、伪指令、操作符和寻址方式，介绍了汇编语言的数据结构，包括结构、记录、宏、堆栈。本书着重讲解了汇编语言程序设计的基本要素和高级的结构化程序设计方法。

本书能使读者在最短的时间内学会汇编语言程序设计，并且养成结构化编程的好习惯，为今后进一步学习奠定坚实的基础。本书既可作为广大计算机用户和高、中等学校师生的参考书，也可作为初学者的入门书。

书　　名	**汇编语言程序设计自学教程**
作　　者	[美] Mark Goodwin 著
译 校 者	周予滨 田学锋 译　王 勇 校
出版、发行	水利电力出版社（北京三里河路 6 号） 各地新华书店经售
排　　版	五环出版服务部
印　　刷	北京市朝阳区小红门印刷厂
规　　格	787×1092 mm　16 开本　11.5 印张　260 千字
版　　次	1995 年 1 月第一版　1995 年 1 月北京第一次印刷
印　　数	0001—5000 册
定　　价	**47.00** 元（含磁盘）
书　　号	ISBN 7-120-02175-3/TP・85

引 言

在计算机发展的早期，编程方法只有一种——机器语言编程。计算机先驱们很快就认识到机器语言编程很不方便，因此他们设计了一种以符号名称代表机器语言指令的编程语言。他们称这种符号表示的编程语言为汇编语言（assembly language）。继汇编语言在前进的道路上迈进了一步之后，计算机设计者又开发了诸如 Fortran 和 COBOL 等高级语言（high-level language），极大地简化了用户的编程工作。

现代计算机程序员可以使用的语言非常丰富。许多人喜欢 C 或 Pascal。然而，还有许许多多其它语言（包括汇编语言在内）被广泛地使用。虽然目前完全使用汇编语言编写的应用程序很少见，但是一些忠实的汇编语言程序员不会放弃选用汇编语言的机会。更常见的是，程序员在高级语言中融入汇编语言以加速某些与时间相关的例程。而本书的目的正在于此。本书假设读者熟悉一种高级语言，如 C 或 Pascal，并希望学习使用汇编语言增强高级语言程序的方法。虽然有优秀的优化编译器可以使用，但是，一些要求执行速度的过程只能通过精心制作的汇编语言代码来实现。另外，熟悉计算机可以帮助读者写出高效率的高级语言程序，理解计算机的实际操作。

本书讲授内容

本书传授了使用汇编语言编程的方法，书中涉及了汇编语言所有的基本特性，包括：

- 汇编语言程序的要素
- 8088 体系结构
- 数据表示法
- 伪指令和操作符
- 8088 指令集
- 寻址方式
- 汇编语言中的字符串处理
- 结构化编程技术
- 结构和记录
- 堆栈
- 过程
- 输入输出
- 条件汇编
- 等价和宏
- 用于增强 C 程序的汇编语言例程
- 用于增强 Pascal 程序的汇编语言例程

本书不讲授的内容

本书不详细解释汇编语言编程的每一个细枝末节。这些细节应从汇编软件包附带的参考手册中得到。另外，本书也不讨论大量华而不实的算法。如果读者是一位有经验的程序员，就一定知道如何利用计算机程序达到自己的目的。本书只包含一些程序框架，而不是大量的完整程序示范，每处只讲解一点汇编语言编程知识。在本书结束时，读者可把这些点滴片段联系起来组成自己的汇编语言程序。

使用本书必备

需要一台 IBM PC 或兼容机及 Turbo Assembler 或者 Microsoft Macro Assembler。

目　录

第一章　8088 汇编语言程序设计

学习汇编语言编程有助于编写用于增强高级语言程序的高效代码。本章通过讲解一个示例程序及以下内容开始汇编语言的编程学习：

- 机器语言
- 汇编语言
- 8088 汇编语言程序组成部分
- 程序汇编
- 程序执行

1.1　机　器　语　言

在开始汇编语言编程的学习之前，必须了解机器语言编程。顾名思义，机器语言编程由计算机本身的语言完成。一台诸如 PC 或兼容机的数字计算机只明白两种不同的状态：关和开。这两种数字状态通常以二进制数字 0 和 1 表示。而且，一位二进制数字习惯称之为一个位（bit）。机器语言程序的表达式就由这些位串组成。这些位串被称为位模式（bit patterns）。下行表示一个机器语言位模式：

```
101110000000010100000000
```

这些机器语言位模式代表指令、数据和指令数据的地址。上面位模式的前 8 位代表 8088 机器语言指令，后 16 位代表一个数据值。下例说明计算机是怎样看待上述模式的：

```
10111000 0000010100000000
mov   ax                 3
```

如上例所示，位模式的前 8 位命令计算机把指令后面的 16 位数字送到名叫 AX 的存储单元。这些存储单元叫寄存器（registers）。在上述指令中，机器语言指令命令 CPU 在 AX 寄存器中存放数值 3。

虽然机器语言很吸引人，但毕竟太难以理解。因为，一个机器语言程序可以由大量位模式组成，而记住各种位模式完成的功能，即使是最优秀的程序员也会感到困难。为了以更容易理解的方式表示这些机器语言指令，早期的程序设计者发明了一种使用语言表示机器语言指令的方法，这种语言就是汇编语言（assembly language）。

1.2　汇　编　语　言

在汇编语言中，机器语言指令的专用名称为助记符（mnemonics）。另外，数据可以用数字值（例如 5，16，0bfh，33，555，1）或者符号名称（例如 MAX，count，line，或

RESULT）表示。下例说明了怎样使用汇编语言表达式表示机器语言位模式的：

```
mov             ax，3
```

至少可以这么说，上述的汇编语言语句比相应的机器语言位模式好记得多。遗憾的是，计算机不知道上述汇编语言语句的含义。为把上述语句翻译成计算机能够理解的格式，必须使用汇编程序。汇编程序只是把汇编语言程序语句翻译成等价的机器语言位模式。

1.3 程序的组成部分

8088 汇编语言程序由下列四个基本部分组成：

- 代码段
- 数据段
- 标号
- 注释

1.3.1 代码段

实质上，代码段（code segments）是包含指令的程序部分，这些指令完成各种任务，诸如传送数据、控制程序流程、实现算术运算功能等等。每条汇编语言指令由两个基本部分组成：操作符（operation）和操作数（operands）。下例说明了指令 mov ax，3 的组成：

```
操作符          操作数
mov             ax，3
```

上例表明，助记符 mov 命令 CPU 传送一个数据。操作数 ax，3 告诉 CPU 要传送的数据是 3，要送到 AX 寄存器。尽管所有的汇编语言指令都需要操作符，但不一定需要操作数。例如，下列语句命令 8088 CPU 忽略所有的可屏蔽中断：

```
cli
```

1.3.2 数据段

虽然代码段是必不可少的部分，但是几乎所有的 8088 汇编语言程序都至少需要一个数据段。数据段是汇编语言程序存放数据的部分。例如，一个需要 3 个 16 位变量（a，b，c）的程序要求如下数据段：

```
a       dw      3
b       dw      4
c       dw      ?
```

上述语句中的三个 dw 是汇编伪指令（assembler directives），指示汇编程序定义三个字。对于 8088，一个字是 16 位长。因此，上面的三个语句定义所需的三个 16 位变量。16 位存储单元 a 中存放数值 3，16 位存储单元 b 中存放数值 4，而 16 位存储单元 c 中存放不定值。实质上，上述数据段中的？告诉汇编程序 16 位存储单元 c 未定义。因此，并没有把

c 假设成任何特定值。

1.3.3 标号

上述示例数据段中的符号名 a、b 和 c 是汇编语言标号 (labels) 的一个极好实例。除了用作变量名以外，汇编语言标号还可用作名称，例如段名或过程名。合法的 8088 汇编语言标号由数字、字母和字符?、.、@_、和 $ 组成。另外，名称不能以数字打头，点号 (.) 只能用作标号的第一个字符。标号的长度不限，但只有前 31 个字符有效。除非标号用于连接汇编伪指令，否则后面必须跟一个分号 (:)。

1.3.4 注释

汇编语言程序的第四部分是注释 (comment)。顾名思义，注释描述程序所做的工作而并不影响程序的实际操作。虽然注释看起来不是必需的，但它们是任何程序的一个重要组成部分。通常在以后程序员需要调试或修改程序时，利用注释会明显地减少重新读懂完成某些功能的程序所需的时间。在 8088 汇编语言程序中，编写注释只需以分号 (;) 开头即可。下例列举了一些汇编语言注释：

```
mov ax, 3 ; AX is set to 3
          ; The previous instruction sets register AX to 3
```

1.4 一个汇编语言程序范例

下面的例子列举了一个非常短小的 8088 汇编语言程序。虽然相当简单，但是它指出了组成一个汇编语言程序的要素。首先，查看下面的程序。然后逐行解释同一程序。

```
;
; first.asm - A first assembly language program
;
;
; Code segment
;
_TEXT    segment word public 'CODE'
         assume  cs:_TEXT,ds:_DATA,ss:_STACK
;
; Add two 16 bit values
;
addem    proc  far          ;Entry point from DOS
         mov   ax,_DATA     ;Point the data segment
         mov   ds,ax        ; register to the data ;
                            segment
         mov   ax,a         ;AX = a
         add   ax,b         ;AX = a + b
         mov   c,ax         ;c = a + b
```

```
        mov   ax,4c00h     ;AX = No error return
                           ; code
        int   21h          ;Return to DOS
addem   endp
_TEXT   ends
;
; Data segment
;
_DATA   segment     word public 'DATA'
a       dw    3
b       dw    4
c       dw    ?
_DATA   ends
;
; Stack segment
;
_STACK  segment     para stack 'STACK'
        db    128 dup (?)
_STACK  ends
        end   addem        ;Defines the entry point
```

1.4.1 逐行程序解释

```
;
; first.asm - A first assembly language program
;
```

是一个注释，它给出了程序名 first.asm，并简要说明了程序用途。

```
;
; Code segment
;
```

说明后面是代码段。

```
_TEXT    segment word public 'CODE'
```

表明 CODE（代码）段开始，并为该段指定标号_TEXT。

```
assume cs: _TEXT, ds: _DATA, ss: _STACK
```

assume 是一个汇编程序伪指令，它向汇编程序通报 8088 段寄存器所指向的段。在此，汇编程序“假设”8088 代码段寄存器 CS 指向_TEXT 段，数据段寄存器 DS 指向_DATA

段，堆栈段寄存器 SS 指向_STACK 段。注意，这些都是从汇编程序角度而言的假定。实际上，段寄存器可能指向指示的段，也可能指向别处。

```
;
; Add two 16-bit values
;
```

说明后继过程的用途。

```
addem proc far          ; Entry point from DOS
```

表明过程 addem 开始。而且过程是远（far）调用。

```
mov ax, _DATA           ; Point the data segment
```

把_DATA 段地址送入寄存器 AX。

```
mov ds, ax              ; register to the data
                        ; segment
```

设置 8088 数据段寄存器指向_DATA 段。由于 first.asm 是 DOS EXE 程序，因此 DOS 自动设置 8088 代码段寄存器指向_TEXT 段，而堆栈段寄存器指向程序上端的_STACK 段。然而，8088 数据段寄存器的相应值必须由程序手工设置。

```
mov ax, a               ; AX=a
```

把存放在符号名为 a 的存储器单元中的值送入寄存器 AX。

```
add ax, b ; AX=a+b
```

寄存器 AX 中的值加上存放在符号名为 b 的存储器单元中的值。注意，相加的结果保留在 AX 中。

```
mov c, ax               ; c=a+b
```

在符号名为 c 的存储器单元中存放相加运算的结果。

```
mov ax, 4c00h           ; AX=No error return code
int 21 h                ; Return to DOS
```

把程序控制还给 MS-DOS。

```
addem endp
```

告知汇编程序过程 addem 已经结束。

```
_TEXT ends
```

告知汇编程序_TEXT 段已经结束。

```
;
; Data segment
;
```

说明后继段是数据段。

```
_DATA    segment    word public    'DATA'
```

表明 DATA（数据）段开始并为该段指定标号_DATA。

```
a    dw    3
```

指示汇编程序留出一个 16 位字的空间，初始值为 3，符号名为 a。

```
b    dw    4
```

指示汇编程序留出一个 16 位字的空间，初始值为 4，符号名为 b。

```
c    dw    ?
```

指示汇编程序留出一个 16 位字的空间，无初始值，符号名为 c。

```
_DATA ends
```

告知汇编程序_DATA 段已结束。

```
;
; stack segment
;
```

说明后继段是堆栈段。

```
_STACK segment para stack 'STACK'
```

表明 STACK 段并指定标号为_STACK。实质上，堆栈段是数据段的一个特殊类型。它常用于保存在 CPU 和程序之间传送的数据。

```
db 128 dup (?)
```

指示汇编程序保留 128 个字节的不定值。

```
_STACK ends
```

告知汇编程序_STACK 段已结束。

```
end    addem    ; Defines the entry point
```

告知汇编程序到达程序结尾，紧接着 addem 标号后面的代码是 MS-DOS 进入该程序的入口点。

1.5 汇编 First 程序

虽然上述程序范例深入透视了一个 8088 汇编语言程序，但是没有说明如何创建一个程序。创建汇编语言程序的第一步是使用文本编辑器输入程序的源代码（source code）。源代码只是一个程序的文本文件。磁盘上如果有以上程序的源代码，就可以使用下列命令行利用 Microsoft Macro Assembler（MASM）汇编它：

masm first.asm;

如果使用 Turbo Assembler（TASM）汇编程序，就输入下列命令行：

tasm first.asm;

如果正确地输入了 first.asm 的源代码，汇编程序产生一个名为 first.obj 的目标模块(object modale)。目标模块是汇编源代码到机器语言的转换结果。

此程序在能够实际运行之前还需要完成最后一个步骤。汇编语言程序创建周期的最后阶段叫做链接（linking)。程序的链接过程把汇编程序生成的目标模块变成可执行文件，名叫 first. exe。为利用 Microsoft Linker 创建 first.exe，可以使用下列命令行：

link first;

另外，下列命令行使用 Turbo Linker 链接 first.exe：

tlink first;

1.5.1 运行程序

至此，已建立了该程序的 DOS 可执行形式，使用下列 DOS 命令执行它：

first

试运行 first 后，程序员可能想知道有没有出错。输入正确的命令行后，MS-DOS 系统几乎立即显示提示符。虽然这看上去不像正确的结果，但实际上该程序确实完成了预期的功能，程序员可以使用 debug 查看程序的内部工作。为此，输入下列 DOS 命令：

debug first.exe

debug 提示符出现。在提示符下输入 u。u 命令使 debug 不汇编（unassemble)，或更确切地说反汇编（disassemble）后继的 20 个字节。下面列出了 debug 产生的结果：

```
2131:0000 B83221      MOV   AX,2132
2131:0003 8ED8        MOV   DS,AX
2131:0005 A10400      MOV   AX,[0004]
2131:0008 03060600    ADD   AX,[0006]
2131:000C A30800      MOV   [0008],AX
2131:000F B8004C      MOV   AX,4C00
2131:0012 CD21        INT   21
2131:0014 0300        ADD   AX,[BX+SI]
2131:0016 0400        ADD   AL,00
2131:0018 0000        ADD   [BX+SI],AL
2131:001A 1B894       SBB   CX,[BX+DI+F646]
2131:001E A1D84       MOV   AX,[43D8]
```

反汇编列出的前七行和 first.asm 的代码段中的七行代码几乎完全一样。唯一的区别在

于源代码使用像_STACK、a、b、和c这样的符号名，而反汇编列举所使用的实际存储器单元。

debug的t命令用于查看程序实际执行预期功能的效果。t命令使debug执行下一条机器指令，显示8088寄存器的结果内容，然后显示下一条机器语言指令的反汇编形式。输入t，显示以下结果：

```
-t
AX=2132 BX=0000 CX=001A DX=0000 SP=0080 BP=0000 SI=0000 DI=0000
DS=2121 ES=2121 SS=2133 CS=2131 IP=0003 NV UP EI PL NZ NA PO NC
2131:0003 8ED8        MOV    DS,AX
```

请注意此时AX是怎样保持_DATA段地址（2132H）的。

注：实际段地址根据DOS版本以及调入内存中的TSR（终止驻留程序）而各不相同。

继续执行指令MOV DS，AX：

```
-t
AX=2132 BX=0000 CX=001A DX=0000 SP=0080 BP=0000 SI=0000 DI=0000
DS=2132 ES=2121 SS=2133 CS=2131 IP=0005 NV UP EI PL NZ NA PO NC
2131:0005 A10400          MOV    AX,[0004]
DS:0004=0003
```

此时，DS寄存器保存寄存器AX的值。执行指令MOV AX，[00004]：

```
-t
AX=0003 BX=0000 CX=001A DX=0000 SP=0080 BP=0000 SI=0000 DI=0000
DS=2132 ES=2121 SS=2133 CS=2131 IP=0008 NV UP EI PL NZ NA PO NC
2131:0008 03060600        ADD    AX,[0006]
DS:0006=0004
```

此时，AX保存a中的值。执行指令ADD AX，[0006]：

```
-t
AX=0007 BX=0000 CX=001A DX=0000 SP=0080 BP=0000 SI=0000 DI=0000
DS=2132 ES=2121 SS=2133 CS=2131 IP=000C NV UP EI PL NZ NA PO NC
2131:000C A30800        MOV        [0008],AX
DS:0008=0000
```

此时，AX等于a和b相加的和。执行指令MOV [0008]，AX：

```
-t
AX=0007 BX=0000 CX=001A DX=0000 SP=0080 BP=0000 SI=0000 DI=0000
DS=2132 ES=2121 SS=2133 CS=2131 IP=000F  NV UP EI PL NZ NA PO NC
2131:000F B8004C        MOV        AX,4C00
```

到此为，符号名 c 的地址中的数值等于 a 和 b 相加的和。但是怎样检验呢？其实很简单。debug 的 d 命令可以显示（display）任何存储器单元的值。用下列 debug 命令察看地址 0008（根据以上指令可以看出，这是 c 的地址）：

```
-d 8
2132:0000                       07 00 1B 89 46 F6 A1 D8          ....F...
2132:0010 43 8B 16 DA 43 89 46 EA-89 56 EC EB 2A 90 A1 D6  C...C.F..V..*...
2132:0020 16 0B 06 D8 16 74 1D C4-5E EE 26 8A 47 16 24 07  .....t..^.&.G.$.
2132:0030 3C 01 75 10 A1 4E 1F 89-46 F6 A1 D6 16 8B 16 D8  <.u..N..F.......
2132:0040 16 EB D2 90 89 7E F6 8D-45 01 89 46 E6 E9 38 03  .....~..E..F..8.
2132:0050 C4 5E EE 26 8A 5F 0E 2A-FF D1 E3 D1 E3 FF B7 7C  .^.&._.*.......|
2132:0060 20 FF B7 7A 20 2B C0 50-E8 85 22 52 50 E8 30 2D   ..z +.P.."RP.0-
2132:0070 40 50 8A 5E FE 2A FF D1-E3 D1 E3 FF B7 7C 20 FF  @P.^.*.......| .
2132:0080 B7 7A 20 2B C0 50 07 00                          .z +.P..
```

从清单可见，c 已被设置为 a 和 b 的和。由于正在使用 debug 执行程序，所以使用 debug 的 q（退出）命令返回 DOS。

1.6　小　　结

在本章中，读者了解了机器语言和汇编语言编程的区别，并学习了汇编语言程序的四个基本要素。另外，了解了如何创建、汇编、链接和执行一个实际的汇编语言程序。最后，使用 debug 检查了程序内部工作的结果。

第二章 8088 体系结构

理解计算机如何处理数据将有助于理解 8088 汇编语言编程底层程序的方法。本章讨论：

- 存储器单元
- 8088 寄存器组
- 计算机存储器
- 输入/输出设备

2.1 存储单元

理解计算机的构成（体系结构）的第一步是了解表示机器语言指令和数据值的基本存储单元。8088 汇编语言的特点是具有七种基本存储单元类型：位（bits）、半字节（nibbles）、字节（bytes）、字（words）、双字（doublewords）、四字（quadwords）和十字节（tenbytes）。下面，我们看一下这七种基本存储单元类型。

2.1.1 位

数字计算机以二进制数字串表示数据。这些二进制数字称为位（bits），每一位只能是 0 或 1。图 2-1 表示等于 1 的一位数字。

图 2-1 一位

虽然位是 8088 存储单元的基础，但是它们本身并不特别有用。毕竟，只具有两个不同值的存储单元不能表示各种数据。因此，8088 使用一些更适于表示各种各样数据的其它存储单元。

2.1.2 半字节

第二大类存储单元是半字节（nibbles）。半字节是四个单个位的序列。按照惯例，半字节中的位从右向左计数，最右边一位（最低有效位）是位 0，而最左边一位（最高有效位）是位 3。图 2-2 表示一个组成半字节的四位序列。

2.1.3 字节

由半字节发展而来的下一个类型是字节（byte）。一个字节由 8 位序列构成。字节中的

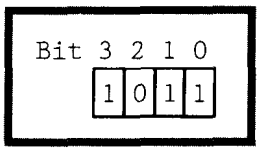


图 2-2 一个半字节

位像半字节一样从右到左计数。图 2-3 表示一个组成字节的 8 位序列。

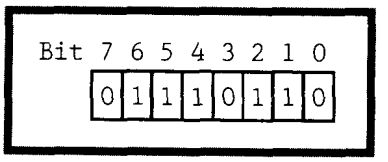


图 2-3 一个字节

2.1.4 字

由字节发展而来的下一个类型是字（word）。字由 16 位序列构成。和半字节及字节一样，字中的位也是从右到左计数。注意，最低有效字节（位 0 到 7）在存储器中存放在最高有效字节（位 8 到 15）的前面。图 2-4 表示作为 2 字节序列在存储器中存放的字。

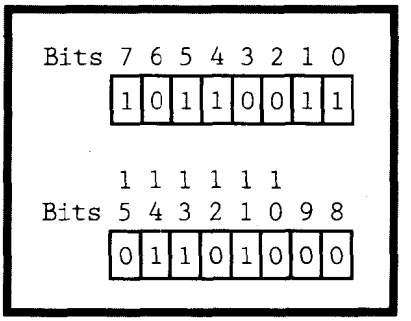


图 2-4 一个字

2.1.5 双字

由字发展而来的下一个类型是双字（doubleword）。双字由 32 位序列构成。双字的位从右到左计数。而且，最低有效字节（位 0 到 7）在存储器中放在前面，再高的有效字节（位 8 到 15）在其后存放，接下来是更高的有效字节（位 16 到 23）。最后，最高有效字节（位 24 到 31）放在最后。图 2-5 表示作为一个 4 字节组存放的双字。

2.1.6 四字

双字发展的下一步是四字（quadword），四字由 64 位序列构成。和前面的存储单元类

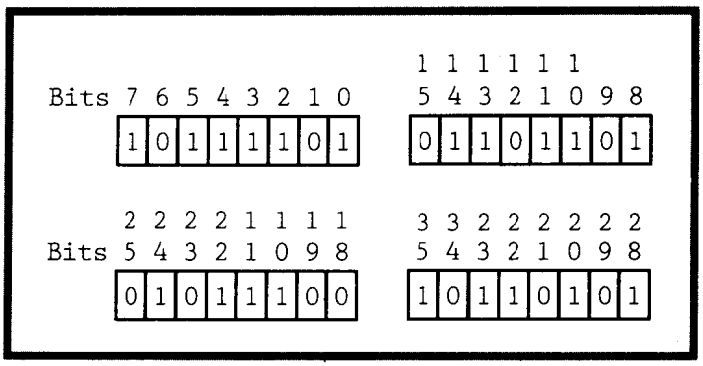


图 2-5 一个双字

型一样，四字的位从右到左计数，最低有效字节在前，最高有效字节在后。图 2-6 表示作为一个 8 字节组存放的四字。

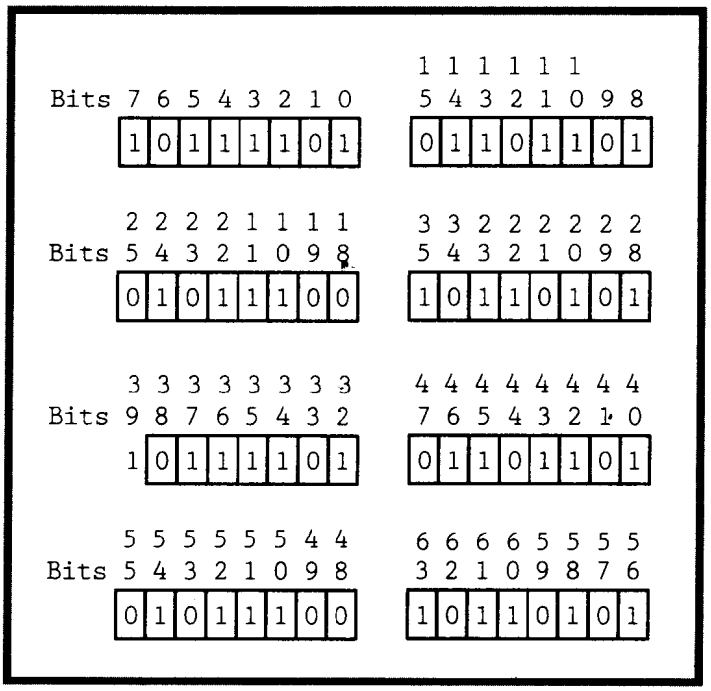


图 2-6 一个四字

2.1.7 十字节

8088 中最大的存储单元是十字节 (tenbyte)。十字节是 10 字节序列或 80 位序列。和前面的存储类型一样，十字节中的位从右到左计数，最低有效字节在前，最高有效字节在后。图 2-7 表示作为一个 10 字节组存放的十字节。

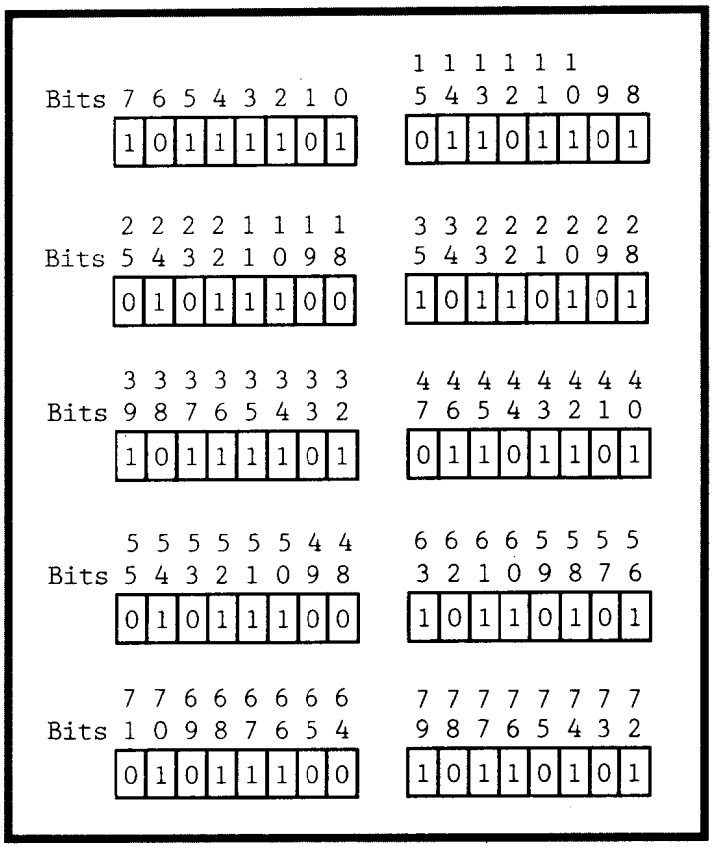


图 2-7 一个十字节

2.2 8088 寄存器组

8088 微处理器具有一些专用存储单元叫做寄存器(registers)。图 2-8 表示 8088 寄存器组。如图所示，8088 有不同的寄存器。

仔细考察这些寄存器，弄清楚它们的一般用法。

2.2.1 寄存器 AX

AX 寄存器是一个累加器，它是 16 位通用寄存器。8088 通用寄存器存放 16 位数值，向存储器或从存储器传送和提取数据，执行算术和逻辑运算。而且寄存器 AX 可以作为两个 8 位寄存器访问。组成 AX 的两个 8 位寄存器是 AH 和 AL。寄存器 AH 引用 AX 中的最高有效字节，AL 引用 AX 中的最低有效字节。寄存器 AX 常用于保存临时数据。使用寄存器 AX 可以加快许多 8088 指令（例如加法或减法）的执行速度。

2.2.2 寄存器 BX

BX 寄存器是一个基址寄存器，它是可以作为两个 8 位寄存器访问的 16 位通用寄存器。

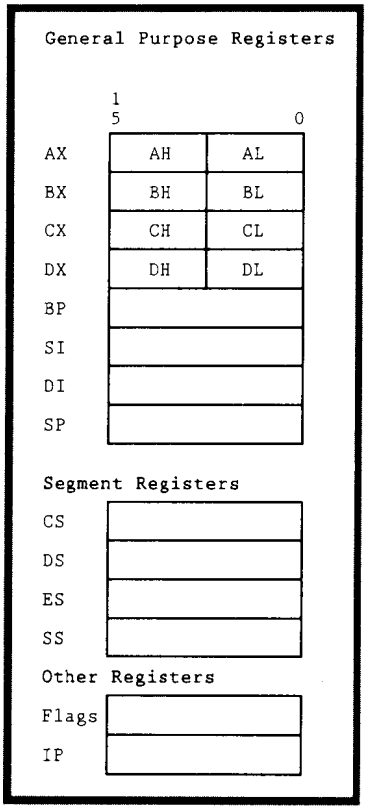


图 2-8 8088 寄存器组

组成 BX 的两个 8 位寄存器是 BH 和 BL。寄存器 BH 引用 BX 中的最高有效字节，BL 引用 BX 中的最低有效字节。寄存器 BX 主要用作指针，也可用作数据存储。

2.2.3 寄存器 CX

CX 寄存器是一个计数器。它是可以作为两个 8 位寄存器访问的 16 位通用寄存器。组成 CX 的两个 8 位寄存器是 CH 和 CL。CH 引用 CX 中的最高有效字节，CL 引用 CX 中的最低有效字节。寄存器 CX 主要用于保存循环或重复指令的次数，也可用作数据存储。

2.2.4 寄存器 DX

DX 寄存器是一个数据寄存器，它是可以作为两个 8 位寄存器访问的 16 位通用寄存器。组成 DX 的两个 8 位寄存器是 DH 和 DL。DH 引用 DX 中的最高有效字节，DL 引用 DX 中的最低有效字节。寄存器 DX 常用于双字算术运算及保存端口输入/输出操作过程中的端口号，它也用作临时数据存储。

2.2.5 寄存器 BP

BP 寄存器是基址指针。它是 16 位通用寄存器，常用于指向高级语言堆栈帧，但也可用作临时数据存储。堆栈段（寄存器 SS）假定为 BP 指向的段寄存器。

2.2.6 寄存器 SI

SI 寄存器是源变址寄存器。它是 16 位通用寄存器，主要用作指针。SI 是字符串操作中的源指针，除了作为指针外，SI 还可用作临时数据存储。

2.2.7 寄存器 DI

DI 寄存器是目的变址寄存器。它是 16 位通用寄存器，主要用作指针。DI 是字符串操作中的目的指针。除了作为指针，DI 还可用作临时数据存储。

2.2.8 寄存器 SP

SP 寄存器是堆栈指针。它是指向堆栈段（寄存器 SS）中堆栈当前地址的 16 位通用寄存器。

2.2.9 寄存器 IP

IP 寄存器是指令指针。它是指向下一个要执行指令的 16 位寄存器。

2.2.10 寄存器 CS

CS 寄存器是代码段寄存器。它是指向汇编语言程序指令和操作数所在的存储器段的 16 位寄存器。为了更好地理解代码段寄存器和其它段寄存器（DS、ES 和 SS）的功能，必须了解 8088 寻址内存的方式。8088 微处理器最多可寻址 1 兆字节存储器。

8088 变址和指针寄存器（BX、BP、DI、IP 和 SI）都是 16 位寄存器。一个 16 位寄存器只能指向 65，536（64K）存储单元。因此，8088 使用一个突破 64K 段限制的段结构。实质上，8088 结合变址或指针寄存器和相应的段寄存器共同确定正确的存储单元。段寄存器和变址或指针寄存器组合使用时，8088 以 16 乘以段寄存器中的值再加上变址或指针寄存器中的内容得到结果。这样，就可以有效地寻址 1 兆存储器（16 * 65，536＝1，048，576）。

2.2.11 寄存器 DS

DS 寄存器是数据段寄存器，它是一个 16 位寄存器并且主要用于指向汇编语言程序中拥有数据分配的段。

2.2.12 寄存器 ES

ES 寄存器是附加段寄存器，它是一些 8088 字符串操作使用的 16 位寄存器。另外，临时段地址也可以存放在寄存器 ES 中。

2.2.13 寄存器 SS

SS 寄存器是堆栈段寄存器，它是指向汇编语言程序中堆栈区的 16 位寄存器。程序和 CPU 都使用堆栈区保存临时数据值。

2.2.14 标志寄存器

标志寄存器指示一些 8088 指令的结果。标志寄存器是 16 位寄存器，其中每一位都有专门含义。图 2-9 说明了 8088 是怎样使用标志寄存器的各个位的。

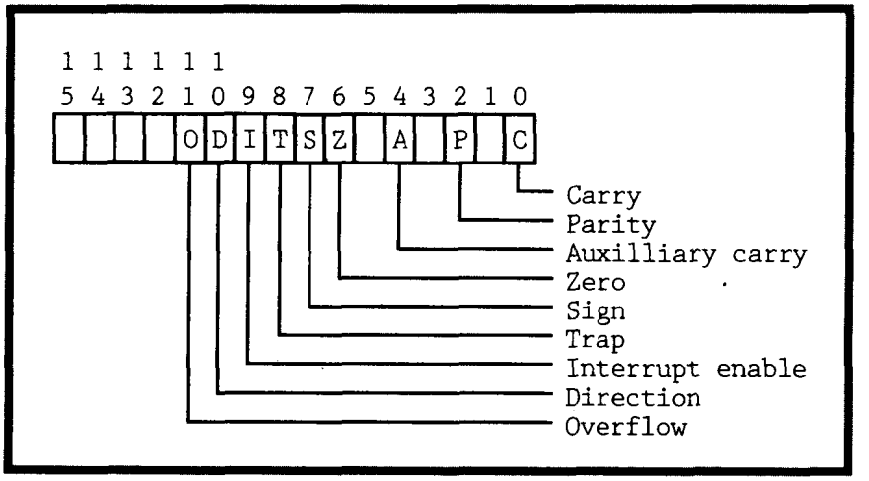


图 2-9 标志寄存器

- 进位标志（carry flag）由操作置为（1）指示进位或借位。
- 奇偶校验标志（parity flag）被置为（1）表示操作结果有偶数个位为 1。
- 辅助进位标志（auxiliary carry flag）被置为（1）表示操作的低四位产生一个进位或借位。辅助进位标志用于二进制编码的十进制运算。
- 零标志（zero flag）置为（1）表示操作结果为 0。
- 符号标志（sign flag）置为（1）表示操作结果为负值。反之，sign 标志清（0）表示正值结果。
- 一旦陷阱标志（trap flag）置为（1），在每条指令执行后产生一个单步中断。正是利用陷阱标志，调试程序才得以单步执行程序。
- 中断激活标志（interrupt enable flag）置为（1），8088 识别所有的可屏蔽中断。
- 方向标志（direction flag）置为（1），所有 8088 字符串操作以反向进行——下行方向，清除方向标志使 8088 字符串操作在存储器中以上行方向进行。
- 溢出标志（overflow flag）置为（1）指示操作结果对于目的操作数太大或太小。

2.3 计算机存储器

8088 微处理器需要与其它一些设备一起构成功能系统。这些设备中最重要的是存储器。计算机存储器有两个基本类型：随机访问存储器（random access memory，RAM）和只读存储器（read only memory，ROM）。随机访问存储器可以读写，只读存储器只能读不能写。

2.4 输 入 和 输 出

目前，计算机已有一个 CPU 和一些存储器，但是，还需要其它设备和计算机通信。而且，还需要存放和取得程序和数据的场所。满足这些通信和存储要求的设备称为输入/输出设备 (input/output devices)。输入/输出设备可分为三个主要类别：输入设备、输出设备和输入/输出设备。

2.4.1 输入设备

输入设备是用于向计算机发送数据的设备。例如键盘、鼠标、轨迹球、操纵杆、数字化仪、扫描仪和 CD-ROM 驱动器。如上所列，输入设备主要用于人们和计算机通信。

2.4.2 输出设备

输出设备是从计算机获取数据的设备。例如显示器、打印机和绘图仪。输出设备主要用于计算机和操作者的一些通信。

2.4.3 输入/输出设备

输入/输出设备是既可以向计算机发送数据又可以从计算机得到数据的设备。例如调制解调器、软盘驱动器、硬盘驱动器和光盘驱动器。虽然像调制解调器这样的设备用于与其它计算机通信，但是输入/输出设备通常用于存放和计算机程序和数据。

2.5 小 结

在本章中，我们学习了 8088 微处理器的基本存储单元类型、寄存器和段结构。另外，还学习了使计算机成为功能设备的设备：计算机存储器、输入设备、输出设备和输入/输出设备。

第三章 数据表示法

虽然计算机所使用的二进制数系统可以表示任何数字，但是以二进制数字来考虑问题很困难。因此，程序设计者使用了多种数制系统简化计算机程序中的数字表示。本章讨论了 8088 汇编语言是怎样使用下列系统的：

- 二进制数字
- 十进制数字系统
- 十六进制数字系统
- 正数和负数
- 布尔运算符
- 二进制编码的十进制
- 浮点数
- 字符和字符串

3.1 二进制数字

由于二进制数字(binary numbers)是数字计算机的语言，所以二进制的采用值得赞赏。虽然其它数字系统可能更重要，但是特定任务实质上是用二进制数字完成的。图 3-1 表明一个字节是如何用于表示二进制数字的。

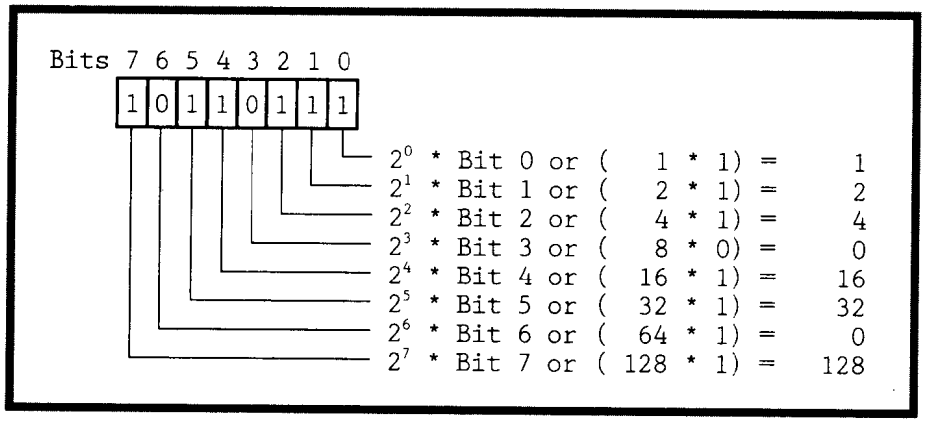


图 3-1 二进制数字系统

如图 3-1 所示，二进制数字系统中的每个数位都是 2 的幂，并且每个数位所表示的 2 的幂次都直接来自其位标号 (bit number)。

3.2 十进制数字系统

十进制数字系统(decimal number system)或基数 10 是日常使用的数字系统。实质上，十进制数字的每个数位都是 10 的幂。图 3-2 表示了一个 5 位十进制数字可以表示的数字值域

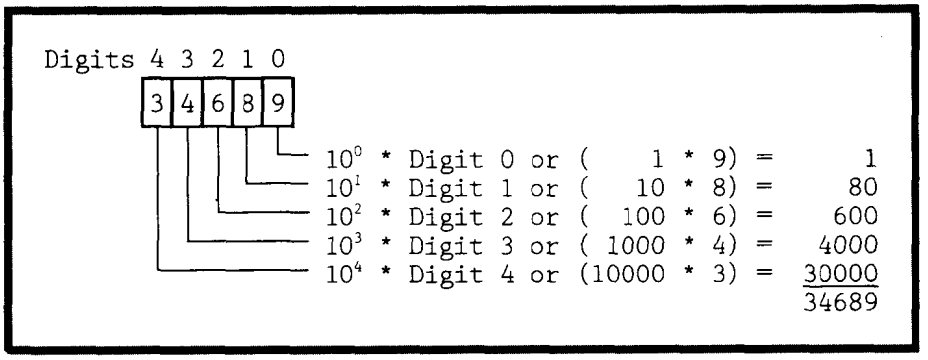


图 3-2 十进制数字系统

人们天生就理解十进制数字，而不理解二进制数字。人类能够轻松地想像基数 10。不管这是因为人类社会采用了十进制系统，还是由于十进制数字系统确实很神秘，总之，十进制数字最容易为计算机程序员接受。遗憾的是，十进制数字在表示汇编语言程序中的数据时并不总是最好的。

3.3 十六进制数字系统

十六进制数字系统(hexadecimal mumber system)或者基数 16 可能是表示汇编语言程序最广泛使用的数字系统。由于每个十六进制数位可以表示 0 到 15 的数字，所以它可以表示一个半字节数据。不要忘了，半字节是四位序列。因此，一个半字节可以包含 0000 (0) 到 1111 (15) 的值。

只用一位表示 16 个不同的值将引出一点问题。在十六进制数字中怎样表示数字 10 到 15 呢？十六进制数字系统使用数字字符 0、1、2、3、4、5、6、7、8、9 表示值 0 到 9，另外用字母字符 A、B、C、D、E、F (或 a、b、c、d、e、f) 表示值 10 到 15。表 3-1 说明了所有 16 个十六进制数字的表示：

表 3.1 所有 16 个十六进制数字的表示

数 字 值	字 符
0	0
1	1
2	2
3	3
4	4

（续表 3.1）

数 字 值	字 符
5	5
6	6
7	7
8	8
9	9
10	A 或 a
11	B 或 b
12	C 或 c
13	D 或 d
14	E 或 e
15	F 或 f

由于每个十六进制数字可以表示一个半字节，所以十六进制数字系统很适合汇编语言编程。图 3-3 说明了如何使用一个四位十六进制数字表示一个数据字。此图说明了在表示可能碰到的众多数字时，十六进制数字系统比二进制数字系统更好。而且，十六进制数字系统比十进制数字系统优越的原因是因为它能简捷地表示字节、字以及在汇编语言程序中出现的各种类型数据的值域。

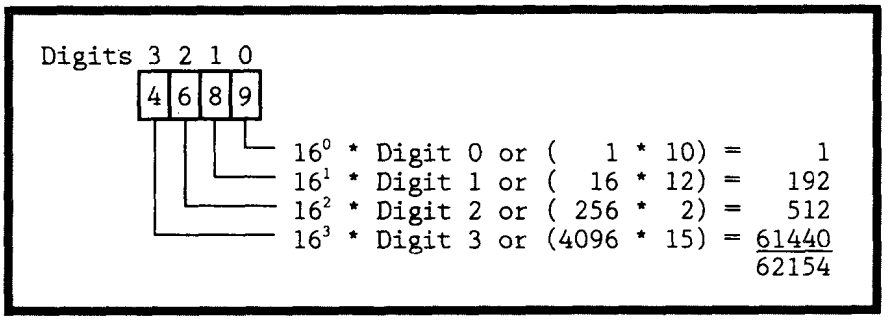


图 3-3 十六进制数字系统

3.4 正 数 和 负 数

到目前为止，读者只涉及到无符号整数值。整数是不含分数部分的数字。虽然汇编语言程序中的大多数数字不需要符号，但是很多数据类型确实需要带符号的数值。每个数据指定的符号位指示数值的正负。图 3-4 说明了在字节值中怎样实现符号位。如图所示，符号位是最高有效位，置为 0 表示正数，置为 1 表示负数。

由于符号本身要占一位，所以带符号的字节值只使用 7 位表示数值。因此，带符号的字节值只能表示 0 到 127 之间的正数。但怎样表示负值呢？首先的考虑应该是仅仅简单地

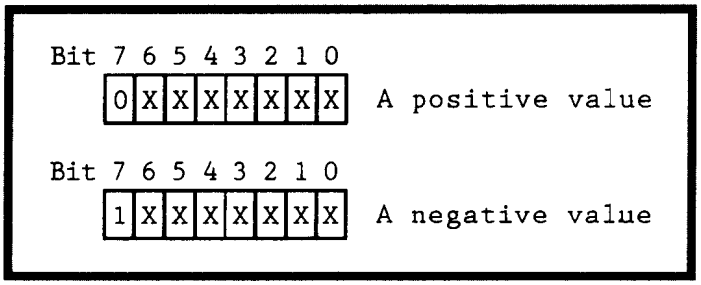


图 3-1 正数和负数

翻转全部正值位。这种位翻转称为 1 次取补（1' s complement）。遗憾的是，简单地 1 次取补不完全可行，图 3-5 说明了原因。

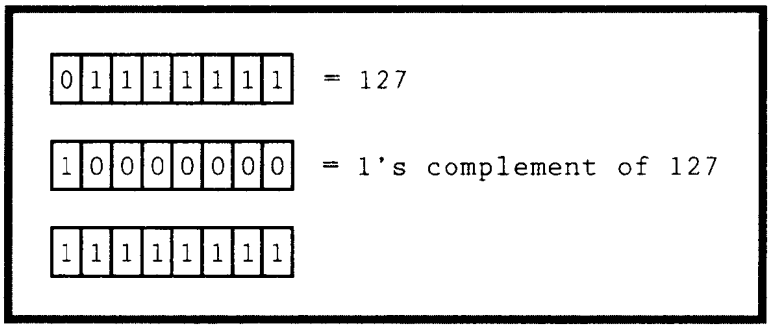


图 3-5 加 127 和 1 次取补

从中可见，使用 1 次取补计算 127 加－127，但是结果是全 1。显然这是错误结果（127 ＋（－127）＝0）。但是在结果上简单地加 1 就可以得到正确结果。图 3-6 表明上述结果是怎样加上 1 的。

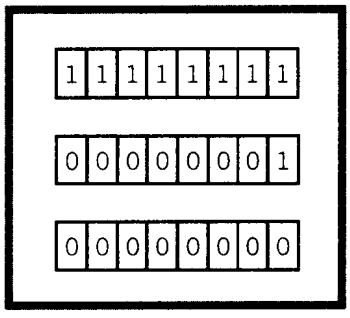


图 3-6 调整 1 次取补的结果

应用图 3-6 中给出的相同原理，可以使用 2 次取补方法形成负数。为把正数转换为负数，只需执行 1 次取补然后把结果加 1。在把负数转换为正数时，此法同样奏效。图 3-7 表明了是怎样在 127 上完成 2 次取补得到－127 的。

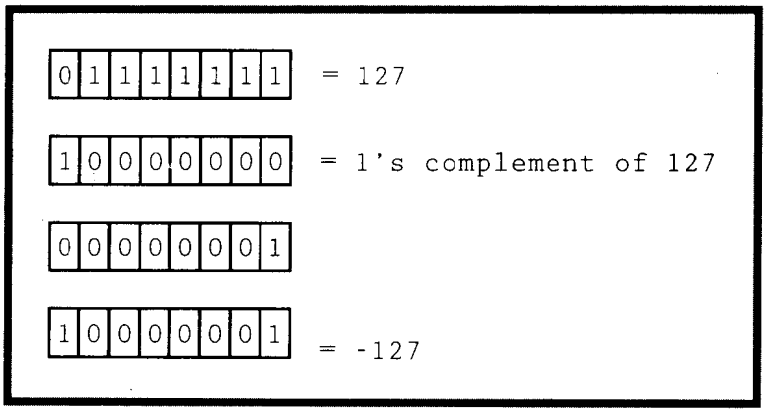


图 3-7 计算 127 的 2 次取补值

最后，图 3-8 通过把图 3-7 中的计算值加上 127，表明它确实是－127。由于这次使用了 2 次取补计算出正确值－127，所以运算得到正确结果 0。

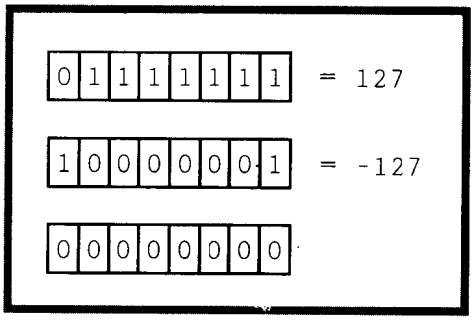


图 3-8 计算 127＋（－127）

3.5 布尔运算符

布尔运算符（boolean operators）表示值 True（真）和 False（假）。这两个逻辑值都以一个数位表示，1 表示 True，0 表示 False。对于尺寸大于一位的存储单元，布尔值通过在存储单元各位中重复相应位值表示。这样，值为 True 的字节置为 FF^{16}，而值为 False 的字节置为 00^{16}。图 3-9 表明在字节值中是如何表示 True 的。置为 True 的字应等于 $FFFF^{16}$。

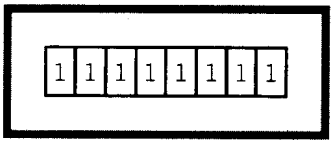


图 3-9 True 的字节表示

注意，许多高级语言使用 1 和 0 的整数值或字节值分别表示 True 和 False 的逻辑值。

在为高级语言编写的汇编语言子程序中，如果使用 8088 汇编语言的 Boolean 方法，就会出现问题。原因是高级语言和汇编语言表示布尔值的方式不同。因此，向高级语言程序传递布尔值时，必须保证该值为高级语言可以使用的形式。

3.6 以二进制编码的十进制

很多程序，尤其是商业应用程序，要求带小数点的数字。保存带小数点的数字有两种基本方法：作为二进制编码的十进制（binary-coded decimals）或者浮点数（floating-point numbers）。这两种数据类型中最简单最适用的是二进制编码的十进制。为了使用二进制编码的十进制对数字编码，每个数字位都要作为半字节存储。例如，要作为二进制编码的十进存储数字 53，就应以值 53^{16}存储字节。

由于数字以 53^{16}而不是以 53^{10}存储，所以必须小心，不要混淆二进制编码的十进制和其它数据类型。例中的二进制编码的十进制如果被错误地用作二进制数字，那么计算机就会把数字解释为 83^{10}（5 * 16+3）。为处理二进制编码的十进制，8088 具有一些只使用这种数据类型的专门指令。虽然使用二进制编码的十进制需要一点额外工作（诸如记录小数点，或使用专门的二进制编码的十进制 8088 指令），但是当高精度是实质问题时，这是存储十进制数字的最佳方法。

3.7 浮 点 数

在计算机上表示浮点数带来了一些有趣的问题：数的符号必须保存，小数点的位置必须保存，而数本身也要保存。几乎所有的计算机编程语言都有各种表示浮点数的方法。在基于 8088 的计算机上，最常用的方法是 Institute of Electrical and Electronic Engineers（IEEE）浮点格式。此格式用于大多数编译器和与 8088 相关的数字协处理器。为了更好地理解浮点数的存储方式，请参考 IEEE 的短实数格式（short real format）。

IEEE 短实数格式需要 32 位存储空间。利用短实数格式存储浮点数的第一步是使其标准化。标准化意味着浮点数的二进制表示左移或右移，直到数字的第一个 1 移到二进制小数点右边。小数部分称为尾数（mantissa），以 23 位值存储。数字移位的位数称为阶（exponent），指数在移位数基础上加 128 称为增阶。然后增阶码作为 8 位数值保存，浮点数的符号和带符号的整型量一样以单独一位保存。图 3-10 表明了 IEEE 短实数是怎样作为一个 32 位值存储的。

3.8 字 符 和 字 符 串

汇编语言程序处理的最重要数据可能是字符和字符串。程序通过显示器上的字符或硬拷贝输出与程序员通信。字符串正是串起来形成信息的字符序列。既然字符和字符串如此重要，全面理解字符在存储器中的存储就是必不可少的。

3.8.1 字符表示法

理解字符在存储器中表示的最简单方法是回忆孩时的情形。某些时候，我们以密码形

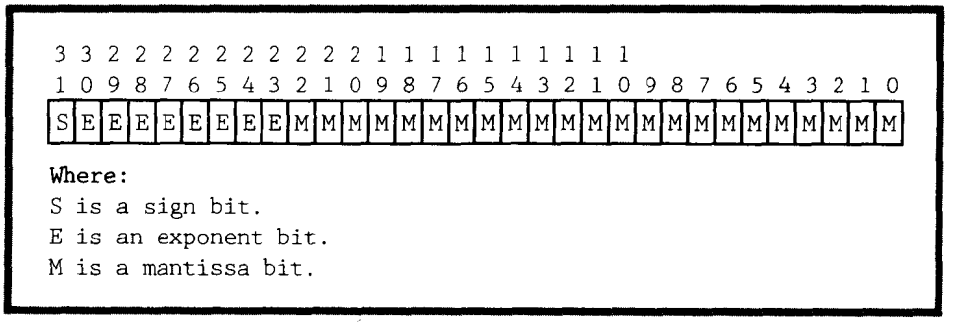


图 3-10 IEEE 的短实型浮点格式

式向朋友传递信息。这些密码通常由表示字母的指定数字组成。持密码解释的人们只需用相应的字母替代数字代码，就可以容易地解释信息。

计算机基本上运用同样的过程处理字符。每个字符赋予一个数值。当在显示器上显示字符或者在打印机上打印字符时，计算机、显示器和打印机通过查找代码表中相应的字符值正确地处理字符。

3.8.2 ASCII 字符

IBM PC 使用 American standard code for Information Interchange (ASCII，发音 as－kee)(美国标准信息交换代码表)。附录 A 列出了 ASCII 代码的完整清单。从中可见，ASCII 码只定义 0 到 127 之间的代码。ASCII 表不能寻址 128 到 255 之间的值，这些值必须由各个计算机制造商定义。对于 IBM PC，代码 128 到 255 用作特殊字符，例如画线字符或外国语言字符的 PC 扩展字符集。

由于一个字节可以表示 0 到 255 的值，因此，字符在 8088 汇编语言编程中以字节值形式存储。字符串以一组字节值存储，图 3-11 说明了字符串“8088”是如何在存储器中存储的。

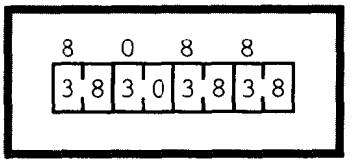


图 3-11 字符串“8088”如何在存储器中存储

3.9 小 结

在本章中，读者学习了各种数据类型在存储器中的表示。这些数据类型包括整型量、带符号整数、布尔运算符、二进制编码的十进制、浮点数、字符和字符串。同时，还学习了怎样使用二进制、十进制和十六进制数字系统表示这些数据类型。

第四章 使用数据工作

在8088汇编语言程序中，表达式使用存储器伪指令定义存储器单元，使用赋值伪指令定义为符号名分配的值，并且定义操作数。MASM和TASM都提供了一组丰富的运算符，从而极大地简化了赋值操作。利用这些运算符，可以轻松地生成复杂的表达式。本章讨论：

- 伪指令、常数和运算符的有关知识
- 如何使用这些工具生成表达式

4.1 伪 指 令

使用8088汇编语言伪指令可以生成汇编语言程序中的数据值。

4.1.1 DB伪指令

DB（定义字节）伪指令初始化一个单字节存储单元。存储字节的值域，对于无符号整数是从0到255，对于带符号整数从－128到127。下例说明了使用DB伪指令定义字节的语法：

label db initializer，initializer，initializer

其中：

- label（标号）是任选标号。
- initializer（初始值）是存储单元中的初始值。如果初始值为?，那么该存储单元具有一个未定义的初始值。定义多个初始值时，必须以逗号隔开。

下例列举了一些使用DB伪指令定义字节的汇编语言语句：

```
          .
          .
a         db    33
scores    db    98,99,100,95,92
hotkey    db    'A'
name      db    "Mark Goodwin"
array     db    10 dup (0)
          .
          .
```

尽管头两个语句具备自身解释，然而后三个需要附加解释。后三个语句的第一个是定义字符的示范。欲定义的字符只需用单引号（'）或双引号（"）括住即可。第二个汇编

语言语句说明如何定义字符串。和字符常数一样，字符串常数也要以单引号或双引号括住。上例中的最后一个语句说明如何在存储器中定义数组或缓冲区。DUP 运算符告知汇编程序在存储器中按指定次数重复复制一个或多个值。下例说明了在 8088 汇编语言程序中是如何使用 DUP 运算符的：

```
count dup initializer, initializer, initializer
```

其中：

- count（次数）是复制初始值的次数。
- initializer 是一个或多个初始值。指定多个初始值时，必须以逗号隔开。

4.1.2 DW 伪指令

DW（定义字）伪指令初始化一个单字存储单元。字存储的值域，对于无符号整数是从 0 到 65，545，对于带符号整数从－32，768 到 32767。除了用于存放整数外，字还常用于存放近指针（段偏移量）。下例说明使用 DW 伪指令定义字的语法：

```
label dw initializer, initializer, initializer
```

其中：

- label 是任选标号。
- initializer 是一个或多个初始值。定义多个初始值时，必须以逗号隔开。

下例列举了一些使用 DW 伪指令定义字的汇编语言语句：

```
        .
        .
account     dw    10000
count       dw    0ffffh
        .
        .
```

4.1.3 DD 伪指令

DD（定义双字）伪指令初始化一个双字存储单元。双字存储的值域，对于无符号整数是从 0 到 4，294，967，295，对于带符号整数是从－2，147，483，648 到 2，147，483，647。除了用于存放大整数，双字还常用于存放远指针（同时要求段地址和段偏移量的指针）。下例说明使用 DD 伪指令定义双字的语法：

```
label dd initializer, initializer, initializer
```

其中：

- label 是任选标号。
- initializer 是一个或多个初始值。定义多个初始值时，必须以逗号隔开。

4.1.4 DQ 伪指令

DQ（定义四字）伪指令初始化一个 64 位的存储单元。四字常用于存储浮点值。下例说明了使用 DQ 伪指令定义 64 位值的语法：

label dq initializer, initializer, initializer

其中：

- label 是任选标号。
- initializer 是一个或多个初始值。定义多个初始值时，必须以逗号隔开。

4.1.5 DT 伪指令

DT（定义十字节）伪指令初始化一个 80 位的存储单元。缺省情况下，DT 伪指令定义一个 10 字节长的二进制编码的十进制值。下例说明了使用 DT 伪指令定义 80 位值的语法：

label dt initializer, initializer, initializer

其中：

- label 是任选标号。
- initializer 是一个或多个初始值。定义多个初始值时，必须以逗号隔开。

4.1.6 EQU 伪指令

EQU（等价）伪指令把整数表达式的结果、定义过的符号或汇编符号赋给一个符号名。汇编程序用赋值取代符号名。下例说明了使用 EQU 伪指令为符号名赋值的语法：

label equ value

其中：

- label 是要赋值的符号名。
- value（值）是为符号名所赋的值。

下例列举了一些使用 EQU 伪指令为符号名赋值的汇编语言语句：

```
    .
    .
row     equ  [bp+2]
copy    equ  mov
length  equ  3*55
    .
    .
```

4.1.7 =伪指令

=（等号）伪指令把整数表达式的结果、定义过的符号或汇编符号赋给一个符号名。

虽然=伪指令和 EQU 伪指令看上去是一样的，但是它们有一个重要区别。使用 EQU 伪指令赋的值不能改变；可是，使用=伪指令赋的值在程序的任何位置都可以改变。下例说明了用=伪指令为符号名赋值的语法：

```
label   =   value
```

其中：

- label 是要赋值的符号名。
- value 是为符号名赋的值。

4.1.8 常数

生成表达式需要许多数值常数。汇编程序缺省地认为所有常数是十进制数字。为了正确地解释其它基数的数值常数，在数值常数后面要加上 radix（基数）定义符。表 4-1 列出了 MASM 和 TASM 提供的基数定义符：

表 4-1 基数定义符

基　数	定 义 符
二进制	B 或 b
八进制	Q、q、O 或 o
十进制	D 或 d
十六进制	H 或 h

上例涉及到八进制数字系统（octal number system）。八进制数字系统的基数是 8，曾经广泛应用于汇编语言程序。此系统目前很少使用，因此本书中不包含这一部分。

构造数值常数时必须意识到另一个需求：所有数值常数必须以 0—9 之间的某个数字打头。使用数字打头，汇编语言立刻知道这是一个数值常数，而非符号名。尽管这种需求不影响二进制、八进制或十进制数，但是许多十六进制是以字母打头的，因而有所影响。幸运的是，汇编语言通过在所有十六进制常数前加数字 0 的字符使之非常容易地编译。例如，值 FE^{16}写为 0feb，而非 feb。

4.1.9 .RADIX 伪指令

.RADIX 伪指令把缺省基数从 10 改为 2 到 16 之间的任何基数。下例说明了使用 .RADIX伪指令改变汇编程序缺省基数的语法：

```
.radix   expression
```

其中：

· expression（表达式）是新的缺省基数值。注意，表达式总认为是十进制的。

4.2 运 算 符

运算符表达汇编语言程序中数据间的关系。

4.2.1 +运算符

+运算返回两个表达式相加的结果。下例说明了使用+运算符相加两个表达式的语法：

expression+expression

其中：

· expression 是合法的表达式。

下例列举了一些使用+运算符表达式的实例：

```
.
.
a  db    33 + 16
b  db    44 + 14
.
.
```

4.2.2 −运算符

−运算返回两个表达式相减的结果。下例说明了使用−运算符相减两个表达式的语法：

expression−expression

其中：

· expression 是合法的表达式。

下例列举了一些使用−运算符的表达式：

```
.
.
a  db    55 - 33
b  db    66 - 100
.
.
```

4.2.3 *运算符

*运算返回两个表达式相乘的结果。下例说明了使用*运算符相乘两个表达式的语法：

expression * expression

其中：

· expression 是合法的表达式。

下例列举了一些使用 * 运算符的表达式：

```
.
.
a  dw    25 * 64
b  db    2 * 4
.
.
```

4.2.4 /运算符

/运算返回两个表达式相除的结果。下例说明了使用/运算符相除两个表达式的语法：

expression/expression

其中：

· expression 是合法的表达式。

下例列举了一些使用/运算符的表达式：

```
.
.
a  db    66 / 2
b  dw    44 / 10
.
.
```

4.2.5 MOD（取模）运算符

MOD 运算返回两个表达式相除的余数。下例说明了使用 MOD 运算符计算两式相除的余数的语法：

expression mod expression

其中：

· expression 是合法的表达式。

下例列举了一些使用 MOD 运算符的表达式：

```
.
.
a  db    33 mod 10
b  db    20 mod 18
.
.
```

4.2.6 NOT（取反）运算符

NOT 运算翻转表达式中各位的值。下表是 NOT 运算的真值表。如下表所示，NOT 运算把位值 1 变为位值 0，把位值 0 变为位值 1。

表 4-2 NOT 运 算

位	NOT 位
1	0
0	1

下例说明了使用 not 运算符翻转表达式各位的语法：

not expression

其中：

• expression 是合法的表达式。

下例列举了一些使用 NOT 运算符的表达式：

```
.
.
a  db    not 01110110b
b  db    not 3
.
.
```

4.2.7 AND（与）运算符

AND 运算使两个表达式进行逐位与操作。下表为 AND 运算的真值表。如表 4-3 所示，如果在两个表达式相应的位都等于 1 时，AND 运算在结果表达式中的此位返回 1，否则这一位返回 0。

表 4-3 AND 运 算

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

下例说明了两个表达式相与的语法：

expression and expression

其中：

• expression 是合法的表达式。

下例列举了一些使用 AND 运算符的表达式：

```
    .
    .
a   db    01110110b and 10100101b
b   db    0feh and 35
    .
    .
```

4.2.8 OR（或）运算符

OR 运算使两个表达式进行逐位或操作。下表是 OR 运算的真值表。如表 4-4 所示，在两个表达式相应的位有一个为 1 时，OR 运算在结果表达式中的该位返回 1，否则，返回 0。

表 4-4 OR 运 算

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

下例说明了两个表达式相或的语法：

expression or expression

其中：

• expression 是合法的表达式。

下例列举了一些使用 OR 运算符的表达式：

```
    .
    .
a   db    10001101b or 00010001b
b   db    10000111b or 13h
    .
    .
```

4.2.9 XOR（异或）运算符

XOR 运算进行两个表达式的逐位异或操作。下表是 XOR 运算的真值表。如表 4-5 所示，在两个表达式的相应位只有一个等于 1 时，XOR 运算在结果表达式中的这一位返回 1，否则这一位返回 0。

表 4-5 XOR 运 算

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

下例说明了两个表达式异或的语法：

expression xor expression

其中：

• expression 是合法的表达式。

下例列举了一些使用 XOR 运算符的表达式：

```
.
.
a  db    01110111b xor 10101010b
b  db    10101010b xor 0ffh
.
.
```

4.2.10 SHL（左移）运算符

SHL 运算根据一个表达式指定的次数，左移另一个表达式各位。注意，每次左移时最高有效位将丢失，而最低有效位置为 0。最高有效位如果是 0，左移将使表达式乘 2。下例说明了使用 SHL 运算符左移表达式的语法：

expression shl count

其中：

• expression 是合法的表达式。
• count 是表达式左移的次数。

下例列举了一些使用 SHL 运算符的表达式：

```
.
.
a  db    01001011b shl 3
b  db    3ah shl 1
.
.
```

4.2.11 SHR（右移）运算符

SHR 运算根据一个表达式指定的次数右移另一个表达式各位。注意，每次右移时最低有效将丢失，而最高有效位置 0。这将使表达式除以 2。下例说明了使用 SHR 运算符右移表达式的语法：

expression shr count

其中：

- expression 是合法的表达式。
- count 是表达式右移的次数。

下例列举了一些使用 SHR 运算符的表达式：

```
.
.
a  db    11110101 shr 4
b  db    43 shr 1
.
.
```

4.2.12 EQ（等于）运算符

EQ 运算判断两个表达式是否相等。如果相等，EQ 运算返回真值（1），否则返回（0）。下例说明了测试两个表达式是否相等的语法：

expression eq expression

其中：

- expression 是合法的表达式。

下例列举了一些使用 EQ 运算符的表达式：

```
.
.
a  db    43 eq 41
b  db    41 eq 41
.
.
```

4.2.13 NE（不等于）运算符

NE 运算判断两个表达式是否不等。如果两个表达式不等，NE 运算返回真值（1），否则返回（0）。下例说明了使用 NE 运算符测试两个表达式式的语法：

expression ne expression

其中：

• expression 是合法的表达式。

下例列举了一些使用 NE 运算符的表达式：

```
.
.
a  db    43 ne 41
b  db    41 ne 41
.
.
```

4.2.14 LT（小于）运算符

LT 运算判断一个表达式是否小于另一个表达式。第一个表达式如果小于第二个，LT 运算返回真值（1），否则返回（0）。下例说明了使用 LT 运算符测试两个表达式的语法：

expression lt expression

其中：

• expression 是合法的表达式。

下例列举了一些使用 LT 运算符的表达式：

```
.
.
a  db    43 lt 41
b  db    41 lt 55
.
.
```

4.2.15 LE（小于或等于）运算符

LE 运算判断一个表达式是否小于或等于另一个表达式。第一个表达式如果小于或等于第二个，LE 运算返回真值（1），否则返回（0）。下例说明使用 LE 运算符测试两个表达式的语法：

expression le expression

其中：

• expression 是合法的表达式

下例列举了一些使用 LE 运算符的表达式：

```
.
.
a  db    43 le 41
b  db    41 le 43
.
.
```

4.2.16 GT（大于）运算符

GT 运算判断一个表达式是否大于另一个表达式。第一个表达式如果大于第二个，GT 运算返回真值（1），否则返回（0）。下例说明了使用 GT 运算符测试两个表达式的语法：

```
expression gt expression
```

其中：

- expression 是合法的表达式。

下例列举了一些使用 GT 运算符的表达式：

```
    .
    .
a   db    43 gt 41
b   db    41 gt 55
    .
    .
```

4.2.17 GE（大于或等于）运算符

GE 运算判断一个表达式是否大于或等于另一个表达式。第一个表达式如果大于或等于第二个，GE 运算返回真值（1），否则返回（0）。下例说明了使用 GE 运算符测试两个表达式的语法：

```
expression ge expression
```

其中：

- expression 是合法的表达式。

下例列举了一些使用 GE 运算符的表达式：

```
    .
    .
a   db    43 ge 41
b   db    41 ge 55
    .
    .
```

4.2.18 SEG（段）运算符

SEG 伪指令返回操作数的段地址。下例说明了使用 SEG 运算符确定操作数段地址的语法：

```
seg expression
```

其中：

• expression 为标号、变量、段名、组名或其它存储器操作数。

下例列举了一个使用 SEG 运算符的表达式：

```
        .
        .
a       db      ?
        .
        .
        mov     ax,seg a      ;AX=a's segment address
        .
        .
```

4.2.19 OFFSET（偏移量）运算符

OFFSET 运算返回操作数在段中的存储器偏移量。存储器偏移量通常称为 near（近）指针。下例说明了使用 OFFSET 运算符确定操作数偏移地址的方法：

offset expression

其中：

• expression 为标号、变量或其它存储器操作数。

下例列举了一个使用 OFFSET 运算符的表达式：

```
        .
        .
a       db      ?
        .
        .
        mov     ax,offset a       ;AX=a's offset address
        .
        .
```

4.2.20 TYPE（类型）运算符

TYPE 运算返回一个变量中每个数据对象的字节数。另外，对近标号（同一段内标号）返回值 0ffffh，对于远标号（不在当前段内的标号）返回 0fffeh。最后，对于常数返回值 0。下例说明了是怎样使用 TYPE 运算确定数据类型的：

type expression

其中：

• expression 为变量、近标号、远标号、或常数。

下例列举了一些使用 TYPE 运算符的表达式：

```
    .
    .
a   dw    1000
b   db    100 dup (?)
    .
    .
    mov   ax,type a   ;AX = 2
    mov   bx,type b   ;BX = 1
    .
    .
```

4.2.21 LENGTH（长度）运算符

LENGTH 运算返回变量中数据对象数。下例说明了使用 LENGTH 运算符确定变量中元素个数的语法：

length variable

其中：

• variable（变量）是先前定义的变量。

下例列举了一些使用 LENGTH 运算符的表达式：

```
    .
    .
a   db    50 dup (?)
buffer  dw    100 dup (0)
c   db    10
    .
    .
    mov   ax,length a       ;AX = 50
    mov   bx,length buffer  ;BX = 100
    mov   cx,length c       ;CX = 1
    .
    .
```

4.2.22 SIZE（尺寸）运算符

SIZE 运算返回汇编程序为变量分配的字节数。实质上，字节数也可由变量的 TYPE 乘以其 LENGTH 得到。下例说明了确定变量中字节数的语法：

size variable

其中：

• variable 为先前定义的变量。

下例列举了一些使用 SIZE 运算符的表达式：

```
    .
    .
a   db    50 dup (?)
buffer  dw    100 dup (0)
c   db    10
    .
    .
    mov   ax,size a          ;AX = 50
    mov   bx,size buffer     ;BX = 200
    mov   cx,size c          ;CX = 1
    .
    .
```

4.2.23 HIGH（高位）运算符

HIGH 运算返回常数表达式的高八位。下例说明了使用 HIGH 运算符确定常数表达式高八位的语法：

high expression

其中：

- expression 是包含不变值的表达式。

下例列举了一些使用 HIGH 运算符的表达式：

```
    .
    .
a   equ   0f3e5h
    .
    .
    mov   ah,high a   ;AH = 0f3h
    .
    .
```

4.2.24 LOW（低位）运算符

LOW 运算返回常数表达式的低八位。下例说明了使用 LOW 运算符确定常数表达式低八位的语法：

low expression

其中：

• expression 是包含不变值的表达式。

下例列举了一个使用 LOW 运算符的表达式：

```
        .
        .
    a   equ   0f3e5h
        .
        .
        mov   ah,low a    ;AH = 0e5h
        .
        .
```

4.2.25 PTR 运算符

PTR 运算强制表达式为指定类型。下例说明了使用 PTR 运算符强制表达式为指定类型的语法：

```
type ptr expression
```

其中：

• type（类型）为存储器操作数的 BYTE、WORD、DWORD、FWORD、QWORD、TBYTE 或标号的 NEAR、FAR、PROC 类型。

• expression 为合法的存储器操作数或标号。

下例列举了一个使用 PTR 运算符的表达式：

```
        .
        .
    a   dw    300
        .
        .
        mov   al,byte ptr a     ;AL = a's low-order byte
        .
        .
```

4.2.26 Segment-Override 运算符

Segment-override 运算强制一个地址与指定段相关。8088 寄存器全部与缺省段相关。Segment-override 运算可以超越这些假定。下例说明了用 segment-override 运算符超越假定段的语法：

```
segment：expression
```

其中：

• segment 为 CS、DS、SS 或 ES。

• expression 为变量或标号的地址。

下例列举了一个使用 segment-override 运算符的表达式：

```
        .
        .
mov     ax,ss:[bx+8]        ;[BX + 8] is now assumed
to                          ; be in the SS segment
        .
        .
```

4.2.27 运算符优先级

截止到本节，所有的表达式都只使用一个运算符形成。因此，表达式的计算结果不存在什么问题。但是对于包含不止一个运算符的表达式呢?

含有多个相同优先级运算符的表达式严格地从左到右计算。看一下表达式 3+4+5 的计算。首先，计算子表达式 3+4 并返回 7。然后计算剩下的表达式 7+5，结果 12。

由使用不同运算符的子表达式组成的表达式利用优先级法则以确定正确的计算顺序。看一看 3+4 * 5 的计算。如果汇编程序先计算子表达式 3+4，结果为 35。如果先计算子表达式 4 * 5，结果是 23。显然，必须有一组计算表达式的法则以避免出现混乱的结果。这正是汇编程序优先级法则的作用所在。

表 4-6 列出了以上讨论的运算符优先级：

表 4-6 运 算 优 先 级

LENGTH	SIZE				
segment-override		PTR	OFFSET	SEG	
TYPE					
HIGH	LOW				
*		/	MOD	SHL	
SHR					
+		−			
EQ	NE	LT	LE		GT
GE					
NOT					
AND					
OR	XOR				
SHORT					

当汇编程序计算含有多个运算符的表达式时，先计算具有最高优先级的子表达式（由运算符在运算优先级表中的位置决定）。如果有多个子表达式含有相同优先级的运算符，汇编程序就按从左到右的顺序计算子表达式。运用这些优先级法则，可以容易地得出表达式 3+4 * 5 的结果应为 23 而不是 35。原因是，* 运算比+运算优先级高。使用圆括号可以改

变运算优先级。例如表达式（3+4）*5 的结果应为 35。

4.3 位置计数器

位置计数器（$）是返回正在汇编语句地址的特殊操作数。下例列举了一个使用位置计数器（$）的表达式：

```
    .
    .
string   db    "This is a sample string"
strlen   equ   $-string    ;strlen = the length of
string
    .
    .
```

4.4 小 结

在本章中，读者学习了数据存储伪指令和数据赋值伪指令。另外，本章还介绍了汇编语言运算符以及怎样使用它们组成表达式。最后，学习了使用位置计数器取得当前地址的方法。

第五章 8088 指 令 集

本章将讨论所有的重要 8088 指令集。这些指令可以用来构造汇编语言程序的代码段。许多最重要的 8088 操作码（opcodes）将展现给读者。这些操作码分为：

- 数据传送指令
- 算术指令
- 数据转换指令
- 位串和布尔指令
- 程序控制和重复指令
- 标志指令
- 其它指令

5.1 汇 编 语 言 指 令

多数汇编语言指令有两个基本部分：操作符（operation）和操作数（operands）。虽然每条汇编语言指令都有操作符，但是并非都有操作数。为简化指令说明，本章使用下列记号描述常用操作数：

- Reg　寄存器。
- Reg8　8 位寄存器：AL、AH、BL、BH、CL、CH、DL 或 DH。
- Reg16　16 位通用寄存器：AX、BX、CX、DX、SP、BP、SI 或 DI。
- SReg　段寄存器：CS、DS、ES 或 SS。
- Mem　存储器单元。
- Mem8　8 位存储器单元。
- Mem16　16 位存储器单元。
- Imm　立即数（immediate value）。换而言之，如整数或定义过的符号特定值。
- Imm8　8 位立即数。
- Imm16　16 位立即数。

5.2 数 据 传 送 指 令

数据传送指令完成以下任务：

- 在 8088 寄存器和计算机存储器之间传送数据
- 在寄存器和存储器之间交换数据
- 访问字节数组中的特定元素

- 在寄存器中装入近指针
- 在 DS 或 ES 寄存器及特定的 16 位寄存器中装入远指针

5.2.1 MOV 指令

8088 MOV（传送）指令在寄存器和存储器之间传送数据。下例定义了 MOV 指令的语法：

```
mov destination, source
```

其中：

- destination（目的操作数）为 Reg8、Reg16、SReg、Mem8 或 Mem16。
- source（源操作数）为 Reg8、Reg16、SReg、Mem8、Mem16 或 Imm。

CS 可以作为目的操作数。存储器单元不能同时作为目的操作数和源操作数。目的操作数如果是段寄存器，就不能使用立即数。

下例列举了一些使用 8088 MOV 指令的语句：

```
.
.
mov   ax,4  ;AX = 4
mov   si,dx ;SI = DX
.
.
```

5.2.2 XCHG 指令

8088 XCHG（交换）指令交换两个寄存器或一个寄存器和一个存储器单元的内容。下例定义了 XCHG 指令的语法：

```
xchg operand, operand
```

其中：

- operand 为 Reg8、Reg16、Mem8 或 Mem16。

注：两个操作数都可以是寄存器，但是只能有一个操作数是存储器单元。而且，要交换的数据规格必须一致，例如，Reg8<->Reg8、Reg16 <->Reg16、Reg8<->Mem8，Reg16<->Mem16。

下例列举了一个使用 XCHG 指令的汇编语言语句：

```
      .
      .
a  db    20
      .
      .
      xchg  ah,a  ;Exchange the contents of AH
```

```
                ; with a
    .
    .
```

5.2.3 LDS 指令

LDS（装入 DS）指令把一个远指针（一个段地址和一个偏移地址）传送到段寄存器 DS 和另一个 16 位寄存器中。指令完成后，段寄存器 DS 保存段地址，指定的 16 位寄存器保存偏移地址。下例定义了 LDS 指令的语法：

```
lds  Reg16, Mem32
```

其中：

- Reg16 为存放偏移地址的寄存器。
- Mem32 为存放远指针的存储器单元。

下例列举了一个使用 LDS 指令的汇编语言语句：

```
    .
    .
a   db    (?)
aptr      equ   this dword
    dw    offset a
    dw    seg a
    .
    .
    lds   si,aptr          ;DS:SI = a's address
    .
    .
```

注意，在上面的例子中，使用了 THIS 运算符，它定义标号的类型。例子中创建了一个双字变量的标号。

5.2.4 LES 指令

LES（装入 ES）指令把一个远指针（一个段地址和一个偏移地址）传送到段寄存器 ES 和另一个 16 位寄存器中。指令完成后，段寄存器 ES 保存段地址，指定的 16 位寄存器保存偏移地址。下例定义了使用 LES 指令的语法：

```
les  Reg16, Mem32
```

其中：

- Reg16 为存放偏移地址的寄存器。
- Mem32 为存放远指针的存储器单元。

下例列举了一个使用 LES 指令的汇编语言语句：

```
    .
    .
a   db    (?)
aptr      equ   this dword
    dw    offset a
    dw    segment a
    .
    .
    les   di,aptr          ;ES:DI = a's address
    .
    .
```

5.2.5 LEA 指令

LEA（装入有效地址）指令把一个偏移地址传送到一个 16 位寄存器。下例了定义使用 LEA 指令的语法：

```
lea Reg16, Mem16
```

其中：

- Reg16 为存放偏移地址的寄存器。
- Mem16 为要装入寄存器的存储器地址。

下例列举了一个使用 LEA 指令的汇编语言语句：

```
    .
    .
a   db    (?)
    .
    .
    lea   bx,a  ;BX = a's address
    .
    .
```

5.2.6 XLAT 指令

XLAT（转移）指令把一个数组元素送入寄存器 AL 中。在调用该指令之前，寄存器 BX 保存数组的起始地址，寄存器 AL 保存元素序号－1。下例定义了 XLAT 指令的语法：

```
xlat  array
```

其中：

• array（数组）为要转移的数组地址。注意，此操作数是任选的，只有在指定了段超越（segment-override）时才必不可少。

下例列举了一个使用 XLAT 指令的汇编语言语句：

```
        .
        .
a   db    3,4,6,10,11
        .
        .
        mov   al,1          ;Specify the 2nd element
        mov   bx,offset a ;BX = Array's address
        xlat                ;Move the 2nd element into
                            ; AL
        .
        .
```

5.3 算 术 指 令

8088 具有一组丰富的算术指令。8088 算术指令完成诸如加 1、减 1、取反、加、减、乘、除的运算。

5.3.1 INC 指令

INC（加 1）指令把寄存器或存储器单元中的值加 1。下例定义了 INC 指令的语法：

```
inc   operand
```

其中：

• operand 为 Reg8、Reg16、Mem8 或 Mem16。

下例列举了一些使用 INC 指令的汇编语言语句：

```
        .
        .
a   db    1
b   dw    33
        .
        .
        inc   al    ;Increment AL
        inc   a     ;Increment memory location a
        inc   cx    ;Increment CX
        inc   b     ;Increment memory location b
        .
        .
```

5.3.2 DEC 指令

DEC（减 1）指令把寄存器或存储器单元中的值减 1。下例定义了 DEC 指令的语法：

```
dec operand
```

其中：

- operand 为 Reg8、Reg16、Mem8 或 Mem16。

下例列举了一些使用 DEC 指令的汇编语言语句：

```
        .
        .
a   db    1
b   dw    33
        .
        .
        dec   al     ;Decrement AL
        dec   a      ;Decrement memory location a
        dec   cx     ;Decrement CX
        dec   b      ;Decrement b
        .
        .
```

5.3.3 NEG 指令

NEG（取反）指令翻转寄存器或存储器单元中的符号。这通过对操作数进行 2 次取补来实现。下例定义了 NEG 指令的语法：

```
neg   operand
```

其中：

- operand 为 Reg8、Reg16、Mem8 或 Mem16。

下例列举了一些使用 NEG 指令的汇编语言语句：

```
        .
        .
a   db    -3
        .
        .
        mov   ax,45 ;AX = 45
        neg   ax    ;AX = -45
        neg   a     ;a = 3
        .
        .
```

5.3.4 ADD 指令

ADD（加）指令相加两个操作数。下例定义了 ADD 指令的语法：

```
add  destination, source
```

其中：

- destination 为存放一个操作数和运算结果的寄存器或存储器单元。目的操作数可以是 Reg8、Reg16、Mem8 或 Mem16。
- source 为与目的操作数相加的寄存器、存储器单元或立即数。源操作数可以是 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数为内存地址。另外，立即数只能作为源操作数。

下例列举了一些使用 ADD 指令的汇编语言语句：

```
        .
        .
a   db    3
b   dw    45
        .
        .
        add   ax,345       ;AX = AX + 345
        add   a,35  ;a = a + 35
        add   b,cx  ;b = b + CX
        add   dx,b  ;DX = DX + b
        .
        .
```

5.3.5 ADC 指令

ADC（进位加）指令把一个操作数的值和 carry（进位）标志的值与另一个操作数相加。当上一个加法运算（ADD 或 ADC）中目的操作数上溢时，carry 标志被置 1。ADC 指令的基本用法是结合 ADD 指令相加两个大于双字节长度的数。

假设要相加两个 32 位的数。使用 ADD 指令正确地相加两数的低 16 位，使用 ADC 指令相加高 16 位和 carry 标志完成余下的运算。使用 ADC 指令取代 ADD 指令的原因是相加低 16 位的结果可能大于 16 位，于是置位 carry 标志。即使更大的数，也可以利用 ADC 指令以此方法计算高位完成正确的计算。记住，低位使用 ADD 指令计算，然后使用 ADC 指令计算其余部分。下例定义了 ADC 指令的语法：

```
adc  destination, source
```

其中：

- destination 为存放一个操作数和运算结果的寄存器或存储器单元。目的操作数可以

是 Reg8、Reg16、Mem8 或 Mem16。

- source 为与目的操作数相加的寄存器、存储器单元或立即数。源操作数可以是 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数是存储器单元。另外，立即数只能作为源操作数。

下例列举了一个使用 ADC 指令的汇编语言指令：

```
.
.
mov   ax,345            ;AX = Least significant
                        ; 16 bits
add   ax,33000    ;AX = Result's least
                  ; significant 16 bits
mov   dx,50000    ;DX = Most significant
                  ; 16 bits
adc   dx,5534           ;DX:AX = 32-bit result
.
.
```

5.3.6 SUB 指令

SUB（相减）指令相减两个操作数。下例定义了 SUB 指令的语法：

```
sub  destination, source
```

其中：

- destination 为存放被减操作数的寄存器或存储器单元。另外，目的操作数还存放运算结果。目的操作数可以是 Reg8、Reg16、Mem8 或 Mem16。
- source 为存放减数的寄存器、存储器单元或立即数。源操作数可以是 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数是存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用 SUB 指令的语句：

```
    .
    .
a   db    3
b   dw    35
    .
    .
    sub   ax,345      ;AX = AX - 345
```

```
sub   a,35  ;a = a - 35
sub   b,cx  ;b = b - cx
sub   dx,b  ;DX = DX - b
.
.
```

5.3.7 SBB 指令

SBB（借位减）指令从一个操作数中减去另一个操作数及 carry 标志值。当上一次减法运算（SUB 或 SBB）中目的操作数下溢时，carry 标志置 1。就像 ADD/ADC 指令组合可以相加大于 16 位的数值一样，SUB/SBB 指令组合可以相减大于 16 位的数值。假定必须相减两个 32 位数。用 SUB 指令相减两个数的低 16 位，用 SBB 指令相减高 16 位及 carry 标志完成余下的运算。对于大于 32 位的减法运算，只需连续使用 SBB 指令。下例定义了 SBB 指令的语法：

```
sbb  destination, source
```

其中：

- destination 为存放被减操作数和运算结果的寄存器或存储器单元。目的操作数可以是 Reg8、Reg16、Mem8 或 Mem16。
- source 为存放减数的寄存器、存储器单元、或立即数。源操作数可以是 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数是存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用 SBB 指令的语句：

```
.
.
mov   ax,10000    ;AX = Least significant
                  ; 16 bits
sub   ax,20000    ;AX = Result's least
                  ; significant 16 bits
mov   dx,50000    ;DX = Most significant
                  ; 16 bits
sbb   dx,32000    ;DX:AX = 32-bit result
.
.
```

5.3.8 MUL 指令

MUL（乘）指令相乘两个无符号字节或字，下例定义了 MUL 指令的语法：

```
mul  operand
```

其中：

- operand 在无符号字节相乘中为 Reg8 或 Mem8，在无符号字相乘中为 Reg16 或 Mem16。

注：相乘无符号字节时，寄存器 AL 中的值和操作数相乘，而结果返回到 AH 和 AL 中。相乘无符号字时，寄存器 AX 中的值与操作数相乘，而结果返回 DX 和 AX 中。

下例列举了一些使用 MUL 指令的汇编语言语句：

```
        .
        .
a       dw      45
        .
        .
        mov     ax,35000        ;AX = 35000
        mul     a               ;DX:AX = 32-bit result of
                                ; 35000 * 45
        .
        .
```

5.3.9 IMUL 指令

IMUL(带符号整数乘)指令相乘两个带符号字节或字。下例定义了 IMUL 指令的语法：

```
imul   operand
```

其中：

- operand 在带符号字节相乘中为 Reg8 或 Mem8，在带符号字相乘中为 Reg16 或 Mem16。

注：相乘带符号字节时，寄存器 AL 中的值和操作数相乘，而结果返回到 AH 和 AL 中。相乘带符号字时，寄存器 AX 中的值与操作数相乘，而结果返回 DX 和 AX 中。

下例列举了一些使用 IMUL 指令的汇编语言语句：

```
        .
        .
a       dw      -45
        .
        .
        mov     ax,35000        ;AX = 35000
        imul    a               ;DX:AX = 32-bit result of
                                ; 35000 * -45
        .
        .
```

5.3.10 DIV 指令

DIV(除)指令以一个无符号字除以一个无符号字节或以一个无符号双字除以一个无符号字。下例定义了 DIV 指令的语法：

```
div   operand
```

其中：

- operand 在无符号字除以一个无符号字节时是一个 Reg8 或 Mem8 除数，在无符号双字时是 Reg16 或 Mem16 除数。

注：当以无符号字除无符号字节时，在执行 DIV 指令之前，被除数必须置于 AX 中。运算完成后，商返回到 AL 中，而余数返回到 AH 中。

注：当以无符号双字除无符号字时，在执行 DIV 指令之前，被除数必须置于 DX 和 AX 中。DX 存放双字的高 16 位，而 AX 存放低 16 位。运算完成后，商返回到 AX 中，而余数返回到 DX 中。

注：一旦除数等于 0，8088 就放弃当前执行程序并显示除法溢出错误信息。因此，合格的结构化程序在执行除法指令之前应检查除零错误。

下例列举了一些使用 DIV 指令的汇编语言语句：

```
        .
        .
a       db    3
        .
        .
        mov   ax,35 ;AX = Dividend
        div   a     ;Divide AX by a
        .
        .
```

5.3.11 IDIV 指令

IDIV（带符号整数除）指令以一个带符号字除以一个带符号字节或以一个带符号双字除以一个带符号字。下例定义了 IDIV 指令的语法：

```
idiv   operand
```

其中：

- operand 在带符号字除以带符号字节时是 Reg8 或 Memm8 除数，在带符号双字除以带符号字时是 Reg16 或 Mem16 除数。

注：当以带符号字除带符号字节时，在执行 IDIV 指令之前，被除数必须置于 AX 中。运算完成后，商返回到 AL 中而，余数返回到 AH 中。

注：当以带符号双字除带符号字时，在执行 IDIV 指令之前，被除数必须置于 DX 和 AX 中。DX 存放双字的高 16 位，而 AX 保存双字的低 16 位。运算完成后，商返回到 AX 中，

而余数返回到 DX 中。

注：一旦除数等于 0，8088 就放弃当前执行程序并显示除法溢出错误信息。因此，合格的结构化程序在执行除法指令之前应检查除零错误。

下例列举了一些使用 IDIV 指令的汇编语言语句：

```
        .
        .
    a   db    -4
        .
        .
        mov   ax,645        ;AX = Dividend
        idiv  a             ;Divide AX by a
        .
        .
```

5.4 数据转换指令

执行算术运算时，处理的数据并非总是符合运算要求的格式。而且，8088 总是假定算术运算在二进制整数上完成。因此，8088 指令集具有确保二进制编码的十进制正确运算的数据转换指令。

5.4.1 CBW 指令

CBW（转换字节为字）指令带符号扩展 AL 中的字节为 AX 中的字。因此，对应 AL 中的正数，寄存器 AH 被置为 0，对应负数，AH 被置为 0FFH。下例定义了 CBW 指令的语法：

```
cbw
```

下例列举了一些使用 CBW 指令的汇编语言语句：

```
.
.
mov   al,3          ;AL = 3
cbw                 ;AX = 3
mov   al,-50              ;AL = -50
cbw                 ;AX = -50
.
.
```

5.4.2 CWD 指令

CWD（转换字为双字）指令带符号扩展 AX 中的字为 DX：AX 中的双字。因此，对应

AX 中的正数，寄存器 DX 被置为 0，对应负数，DX 被置为 0FFFFH。下例定义了 CWD 指令的语法：

```
cwd
```

下例列举了一些使用 CWD 指令的汇编语言语句：

```
.
.
mov   ax,-5304     ;AX = -5304
cwd                ;DX:AX = -5304
mov   ax,3499            ;AX = 3499
cwd                ;DX:AX = 3499
.
.
```

5.4.3 AAA 指令

AAA（加后调整）指令把上次加法运算的和调整为十进制数字（0 到 9）。要求上次加法运算的结果置于 AL 中。和如果大于 9，寄存器 AH 加 1，借位和辅助进位标志置位。如果和没有上溢，AAA 指令就清零两个进位标志。下例定义了 AAA 指令的语法：

```
aaa
```

下例列举了一些使用 AAA 指令的语句：

```
.
.
mov   al,2  ;AL = 2
add   al,8  ;AL = Unconverted result
aaa         ;Convert the result
.
.
```

5.4.4 AAS 指令

AAS（减后调整）指令调整上次减法运算的差为十进制数字（0 到 9）。AAS 指令要求上次减法运算的结果置于 AL 中。差如果大于 9，寄存器 AH 减 1，carry 和 auxiliary carry 标志置位。如果结果没有下溢，AAS 指令就清除两个借位标志。下例定义了 AAS 指令的语法：

```
aas
```

下例列举了一些使用 AAS 指令的汇编语言语句：

```
.
.
mov   al,4   ;AL = 4
sub   al,7   ;AL = Unconverted result
aas          ;Convert the result
.
.
```

5.4.5 AAM 指令

AAM（乘后调整）指令把 AL 中小于 100 的二进制数转换为 AX 中未压缩的二进制编码的十进制数。转换后，寄存器 AH 存放最高有效位数字，AL 存放最低有效位数字。下例定义了 AAM 指令的语法：

```
aam
```

下例列举了一些使用 AAM 指令的汇编语言语句：

```
    .
    .
a   db    7
    .
    .
    mov   al,9   ;AL = 9
    mul   a      ;AX = Unconverted result
    aam          ;Convert the result
    .
    .
```

5.4.6 AAD 指令

AAD(除前 ASCII 调整)指令把 AH 和 AL 中未压缩数字转换为 AX 中的二进制数。下例定义了 AAD 指令的语法：

```
aad
```

下例列举了一些使用 AAD 指令的汇编语言语句：

```
    .
    .
a   db    4
    .
    .
    mov   ah,5   ;AH = Most significant digit
    mov   al,3   ;AL = Least significant digit
```

```
aad          ;AX = Binary value
div   a      ;Perform the division operation
.
.
```

5.4.7 DAA 指令

DAA（加后十进制调整）指令将上次加法运算的和转换为压缩的二进制编码的十进制数。DAA 指令要求上次运算的结果置于 AL 中。转换后，AL 的高半字节存放高位数字，AL 的低半字节存放低位数字。二进制编码的数值如果大于 99H，进位和辅助进位标志置位。否则，DAA 指令清零进位标志。下例定义了 DAA 指令的语法：

```
daa
```

下例列举了一些使用 DAA 指令的汇编语言语句：

```
    .
    .
a   db    34h    ;BCD 34
    .
    .
    mov   al,43h       ;AL = BCD 43
    add   al,a         ;AL = BCD 43 + BCD 34
    daa                ;Convert the result to BCD
    .
    .
```

5.4.8 DAS 指令

DAS（减后十进制调整）指令把上次减法运算的差转换为压缩的二进制编码的十进制数值。DAS 指令要求上次运算的结果置于 AL 中。转换后，AL 中的高半字节存放二进制编码的十进制数值的高位数字，AL 中的低半字节存放低位数字。二进制编码的十制值如果大于 99H，借位和辅助借位标志置位。否则了 DAS 指令清零借位标志。下例定义了 DAS 指令的语法：

```
das
```

下例列举了一些使用 DAS 指令的汇编语言语句：

```
    .
    .
a   db    32h    ;BCD 32
    .
    .
    mov   al,56h       ;AL = BCD 56
```

```
sub   al,a  ;AL = BCD 56 - BCD 32
das         ;Convert the result to BCD
.
.
```

5.5 布 尔 指 令

很多数据类型需要真（1）和假（0）值。正如以前学习的，这些真/假数据类型叫做布尔型。8088 指令集拥有四个处理布尔型数值的指令。

5.5.1 NOT 指令

NOT（非）指令翻转指定操作数的所有位。操作数的所有 0 值变为 1 值，1 值变为 0 值。表 5-1 是 NOT 指令的真值表。

表 5-1 NOT 指 令

位	NOT 位
1	0
0	1

下例说明了 NOT 指令的语法：

```
not   operand
```

其中：

- operand 为 Reg8、Reg16、Mem8 或 Mem16。

下例列举了一些使用 NOT 指令的汇编语言语句：

```
      .
      .
a     db    32h
      .
      .
      not   a     ;Inverts a
      not   dx    ;Inverts DX
      not   ah    ;Inverts AH
      .
      .
```

5.5.2 AND 指令

AND（与）指令完成两个操作数的逻辑与运算，每次处理一位。两数的相应位如果都

等于 1，目的操作数中的相应位就置为 1，否则置为 0。表 5-2 是 AND 指令的真值表。

表 5-2 AND 指 令

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

下例说明了 AND 指令的语法：

and destination，source

其中：

- destination 为 Reg8、Reg16、Mem8 或 Mem16，存放运算结果。
- source 为 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数为存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用了 AND 指令的汇编语言语句：

```
      .
      .
a     db    34h
      .
      .
      mov   al,43h
      and   al,a  ;Perform a logical AND on the
                  ; values
      .
      .
```

5.5.3 OR 指令

OR（或）指令完成两个操作数的逻辑或运算，每次处理一位。如果两数的相应位中任何一个等于 1，那么目的操作数中的相应位置为 1，否则置为 0。表 5-3 是 OR 指令的真值表：

表 5-3 OR 指 令

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

下例说明 OR 指令的语法：

or destination，source

其中：

- destination 为 Reg8、Reg16、Mem8 或 Mem16，存放运算结果。
- source 为 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数为存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用 OR 指令的汇编语言语句：

```
        .
        .
a       db      55h
        .
        .
        mov     dl,33h
        or      dl,a    ;Perform a logical OR on the values
        .
        .
```

5.5.4 XOR 指令

XOR（异或）指令完成两个操作数的异或运算，每次处理一位。两数的相应位中如果仅有一个为 1，目的操作数中的相应位就置为 1，否则置为 0。表 5-4 是 XOR 指令的真值表。

表 5-4 XOR 指 令

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

下例定义了 XOR 指令的语法：

xor destination，source

其中：

- destination 为 Reg8、Reg16、Mem8 或 Mem16，存放运算结果。
- source 为 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数为存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用 XOR 指令的汇编语言语句：

```
        .
        .
a       db      23h
        .
        .
        mov     bl,0ffh
        xor     bl,a  ;Perform a logical XOR on the values
        .
        .
```

5.6 循环和移位指令

除了完成数值位的逻辑运算的指令之外，还有各种循环和移位的汇编语言指令。8088 的循环和移位指令不仅影响数值，还大大影响进位标志。

5.6.1 ROL 指令

ROL（循环左移）指令按指定次数循环左移操作数各位。在循环中，操作数最高位移至进位标志和最低位中。循环次数既可以是立即数 1，也可以是寄存器 CL 中指定的值。图 5-1 表明了 ROL 指令是如何循环移位字节值的：

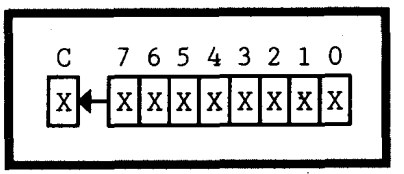


图 5-1 使用 ROL 循环移位字节值

下例定义了 ROL 指令的语法：

```
rol   operand，count
```

其中：

- operand 为循环的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。
- count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 ROL 指令的汇编语言语句：

```
        .
        .
a   db    0f3h
        .
        .
        rol   a,1    ;Rotate a once to the left
        mov   al,55h
        mov   cl,4
        rol   al,cl ;Rotate AL four times to the left
        .
        .
```

5.6.2 ROR 指令

ROR（循环右移）指令按指定次数循环右移操作数各位。在循环中，操作数的最低位移至进位标志和最高位中。循环次数既可以是立即数 1，也可以是寄存 CL 中指定的值。图 5-2表明 ROR 指令是如何循环移位一个字节值的：

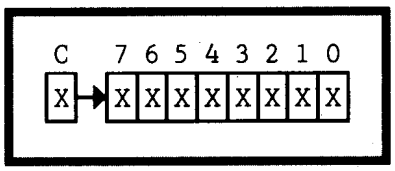


图 5-2 使用 ROR 循环移位字节值

下例定义了 ROR 指令的语法：

```
ror  operand, count
```

其中：

- operand 为循环的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。
- count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 ROR 指令的汇编语言语句：

```
        .
        .
a   db    0f3h
        .
        .
        ror   a,1    ;Rotate a once to the right
```

```
mov   al,55h
mov   cl,4
ror   al,cl ;Rotate AL four times to the right
.
.
```

5.6.3 RCL 指令

RCL（带进位循环左移）指令按指定次数循环左移操作数各位。在循环中，操作数的最高位移至进位标志中，而 carry 标志的内容移至操作数最低位中。循环次数既可以为立即

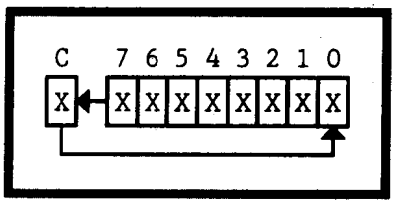


图 5-3 使用 RCL 循环移位字节值

数 1，也可以为寄存器 CL 中指定的值。图 5-3 表明了 RCL 是如何循环移位字节值的：下例定义了 RCL 指令的语法：

```
rcl   operand, count
```

其中：

- operand 为循环的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。
- count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 RCL 指令的汇编语言语句：

```
      .
      .
a     db    0edh
      .
      .
      rcl   a,1   ;Rotate a once to the left
      mov   al,34h
      mov   cl,3
      rcl   al,cl ;Rotate AL three times to the left
      .
      .
```

5.6.4 RCR 指令

RCR（带进位循环右移）指令按指定次数循环右移操作数各位。在循环中，操作数的

最低位移至进位标志中，而进位标志移至操作数最高位。循环次数既可以为立即数 1，也可以为寄存器 CL 中指定的值。图 5-4 表明了 RCR 是如何循环移位字节值的：

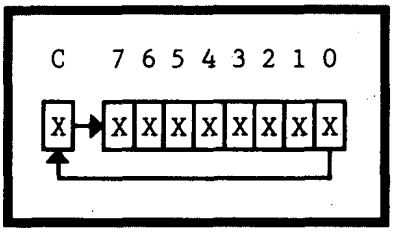


图 5-4 使用 RCR 循环移位字节值

下例定义了 RCR 指令的语法：

```
rcr  operand, count
```

其中：

- operand 为循环的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。
- count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 RCR 指令的汇编语言语句：

```
    .
    .
a   dw    0f34dh
    .
    .
    rcr   a,1   ;Rotate a once to the right
    mov   dx,5567h
    mov   cl,5
    rcr   dx,cl ;Rotate DX five times to the right
    .
    .
```

5.6.5 SAL 指令

SAL（算术左移）指令按指定次数左移操作数各位。在移位中，0 移至操作数的最低位，而操作数最高位移至进位标志。移位数既可以为立即数 1，也可以为寄存器 CL 中指定的值。图 5-5 表明了 SAL 是如何循环移位字节值的：

下例定义了 SAL 指令的语法：

```
sal operand, count
```

其中：

- operand 为移位的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。
- count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

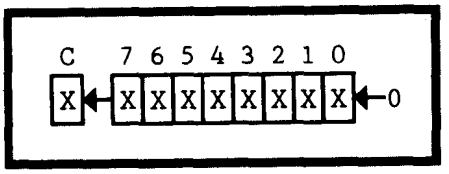


图 5-5 使用 SAL 循环移位字节值

下例列举了一些使用 SAL 指令的汇编语言语句：

```
        .
        .
a   dw    034feh
        .
        .
        sal   a,1          ;Shift a once to the left
        mov   dx,45cbh
        mov   cl,4
        sal   dx,cl        ;Shift DX four times
        .
        .
```

5.6.6 SAR 指令

SAR（算术右移）指令按指定次数右移操作数各位。在移位中，操作数的符号位（最高位）保留，而最低位移至进位标志中。移位数既可以为立即数 1，也可以为寄存器 CL 中指定的值。图 5-6 表明了 SAR 指令是如何移位字节值的：

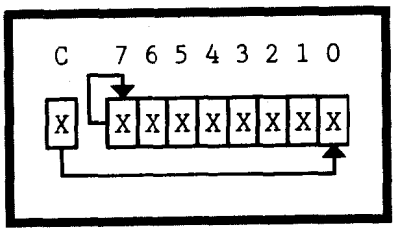


图 5-6 使用 SAR 循环移位字节值

下例定义 SAR 指令的语法：

```
sar   operand, count
```

其中：

- operand 为移位的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。

• count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 SAR 指令的汇编语言语句：

```
        .
        .
a   db    45
        .
        .
        sar   a,1    ;Shift a once to the right
        mov   ch,35h
        mov   cl,2
        sar   ch,cl ;Shift CH two times
        .
        .
```

5.6.7 SHL 指令

SHL（左移）指令按指定次数左移操作数各位。在移位中，0 移至操作数的最低位，而最高位移至进位标志中。注意 SHL 运算和 SAL 完全相同。移位数既可以是立即数 1，也可以是寄存器 CL 中指定的值。图 5-7 表明了 SHL 指令是如何移位字节值的：

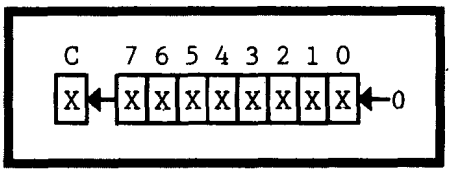


图 5-7 使用 SHL 循环移位字节值

下例定义了 SHL 指令的语法：

shl operand，count

其中：

• operand 为移位的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。

• count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 SHL 指令的汇编语言语句：

```
        .
        .
a   db    44h
        .
```

```
.
shl   a,1   ;Shift a once to the left
mov   bh,33h
mov   cl,4
shl   bh,cl ;Shift BL four times
.
.
```

5.6.8 SHR 指令

SHR（右移）指令按指定的次数右移操作数各位。在移位中，0 移至操作数的最高位，而最低位移至进位标志中。移位数既可以是立即数 1，也可以是寄存器 CL 中指定的值。图 5-8 表明了 SHR 指令是如何移位字的节值：

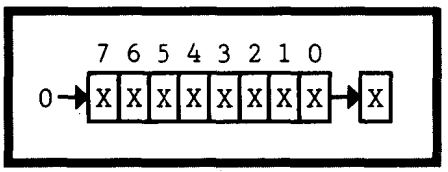


图 5-8 使用 SHR 循环移位字节值

下例定义了 SHR 指令的语法：

```
shr      operand, count
```

其中：

- operand 为移位的数值，类型为 Reg8、Reg16、Mem8 或 Mem16。
- count 为移位操作数的次数，可以为立即数 1 或寄存器 CL。

下例列举了一些使用 SHR 指令的汇编语言语句：

```
.
.
a  db    45
.
.

   shr   a,1   ;Shift a once to the right
   mov   ch,35h
   mov   cl,2
   shr   ch,cl ;Shift CH two times
   .
   .
```

5.7 比 较 指 令

和其它编程语言一样，汇编语言程序也需要依据数据的某一部分的值完成不同的运算。8088 具有两个完成比较的指令：CMP 和 TEST。

5.7.1 CMP 指令

CMP（比较）指令比较两个操作数的值。实质上，CMP 指令相减两数，但是目的操作数不受运算影响。完成比较后，8088 设置相应的处理标志。指示比较结果的标志一旦被置位，就可使用 8088 跳转指令（大多只与某一标志位有关）改变程序执行的流向。下例定义了 CMP 指令的语法：

```
cmp  destination, source
```

其中：

- destination 为 Reg8、Reg16、Mem8 或 Mem16。
- source 为 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数为存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用 CMP 指令的汇编语言语句：

```
    .
    .
a   db    34
    .
    .
    mov   ah,53
    cmp   ah,a  ;Compare the two values
    .
    .
```

5.7.2 TEST 指令

TEST（测试）指令完成两个操作数的逻辑与运算。和 AND 指令不同，TEST 指令只设置 8088 的标志而不影响目的操作数。指示运算结果的标志一旦被置位，8088 就可以利用它与跳转指令结合改变程序流向。下例定义了 TEST 指令的语法：

```
test destination, source
```

其中：

- destination 为 Reg8、Reg16、Mem8 或 Mem16。
- source 为 Reg8、Reg16、Mem8、Mem16、Imm8 或 Imm16。

注：虽然目的操作数和源操作数都可以是寄存器，但是一次只能有一个操作数为存储器单元。另外，立即数只能作为源操作数。

下例列举了一些使用 TEST 指令的汇编语言语句：

```
        .
        .
a   db    4FH
        .
        .
        mov   al,33h
        test  al,a  ;AND the two values
        .
        .
```

5.8 跳 转 指 令

和利用其它编程语言编写的程序一样，汇编语言程序并非一直从头到尾直接执行下来。几乎所有的程序都是循环往复地进行的。为此，8088 提供了各种跳转指令。这些跳转指令能使程序无条件或有条件地沿不同的路经执行。

5.8.1 JMP 指令

JMP（跳转）指令把程序执行转向计算机存储器中的任意位置。下例定义了 JMP 指令的语法：

```
jmp   operand
```

其中：

• operand 为标号，类型为 Reg16 或 Mem16。

下例列举了一些使用 JMP 指令的汇编语言语句：

```
        .
        .
        jmp   next  ;Jump to the memory location with
                    ; the label "next"
        .
        .
        .
next:
        .
        .
```

5.8.2 JA 指令

JA（若大于则跳转）指令在无符号数据比较中，如果第一个操作数大于第二个，则把程序执行转向指定位置。JA 指令和 JNBE 指令完成相同的功能。下例定义了 JA 指令的语法：

```
ja  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JA 指令的汇编语言语句：

```
    .
    .
    cmp   bx,5504     ;Jump if
    ja    next  ; BX > 5504
    .
    .
next:    .
    .
    .
```

5.8.3 JAE 指令

JAE（若大于或等于则跳转）指令在无符号数据比较时，如果第一个操作数大于或等于第二个操作数，则把程序执行转向指定位置。JAE 指令完成和 JNB 指令相同的功能。下例定义了 JAE 指令的语法：

```
jae  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JAE 指令的汇编语言语句：

```
          .
          .
          cmp   ax,578      ;Jump if
          jae   next  ; ax >= 578
          .
          .
next:     .
          .
```

5.8.4 JB 指令

JB（若小于则跳转）指令在无符号数据比较时，如果第一个操作数小于第二个操作数，就把程序执行转向指定位置。JB 指令和 JNAE 指令完成相同的功能。下例定义了 JB 指令的语法：

```
jb  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JB 指令的汇编语言语句：

```
        .
        .
        cmp   al,55 ;Jump if AL
        jb    next  ; is < 55
        .
        .
next:   .
        .
        .
```

5.8.5 JBE 指令

JBE（若小于或等于则跳转）指令在无符号数据比较时，如果第一个操作数小于或等于第二个操作数，就把程序执行转向指定位置。JBE 指令和 JNA 指令完成相同的功能。下列定义了 JBE 指令的语法：

```
jbe  label
```

其中：

label 为距当前存储器单元不超过－128 或 127 字节的标号

下例列举了一些使用 JBE 指令的汇编语言语句：

```
      .
      ..
      cmp  b1, 55h      ; Jump if
      jbe  next         ; b1 <= 55h
      .
      .
next:
      .
      .
```

5.8.6 JC 指令

JC（若进位则跳转）指令在 carry 标志被置位时把程序执行转向指定位置。下例定义了 JC 指令的语法：

```
jc  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JC 指令的汇编语言语句：

```
        .
        .
        sub   al,55
        jc    next  ;Jump if AL - 55 set the carry
                    ; flag
        .
        .
next:   .
        .
        .
```

5.8.7 JE 指令

JE（若等于则跳转）指令在比较的两个操作数相等时把程序执行转向指定位置。下例定义了 JE 指令的语法：

```
je  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JE 指令的汇编语言语句：

```
        .
        .
        cmp   bx,445      ;Jump if
        je    next  ; BX = 445
        .
        .
next:   .
        .
        .
```

5.8.8 JG 指令

JG（若大于则跳转）在比较带符号数据时，如果第一个操作数大于第二个操作数，就把程序执行转向指定位置。JG 指令和 JNLE 指令完成相同的功能。下例定义了 JG 指令的语法：

```
jg  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JG 指令的汇编语言语句：

```
        .
        .
        cmp   al,-101     ;Jump if
        jg    next        ; AL > -101
        .
        .
next:   .
        .
        .
```

5.8.9 JGE 指令

JGE（若大于或等于则跳转）指令在比较带符号数据时，如果第一个操作数大于或等于第二个操作数，就把程序执行转向指定位置。JGE 指令和 JNL 指令完成相同的功能。下例定义了 JGE 指令的语法：

```
jge  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JGE 指令的汇编语言语句：

```
        .
        .
        cmp   bl,67 ;Jump if
        jge   next  ; BL >= 67
        .
        .
next:   .
        .
        .
```

5.8.10 JL 指令

JL（若小于则跳转）指令在比较带符号数据时，如果第一个操作数小于第二个操作数，就把程序执行转向指定位置。JL 指令和 JNGE 指令完成相同的功能。下例定义了 JL 指令的语法：

```
jl  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JL 指令的汇编语言语句：

```
            .
            .
            cmp   dx,-455      ;Jump if
            jl    next         ; dx < -455
            .
            .
next:       .
            .
            .
```

5.8.11 JLE 指令

JLE（若小于或等于则跳转）指令比较在带符号数据时，如果第一个操作数小于或等于第二个操作数，就把程序执行转向指定位置。JLE 指令和 JNG 指令完成相同的功能。下例定义了 JLE 指令的语法：

```
jle  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JLE 指令的汇编语言语句：

```
            .
            .
            cmp   ax,-9999     ;Jump if
            jle   next         ; AX <= -9999
            .
            .
next:       .
            .
            .
```

5.8.12 JNA 指令

JNA（若不大于则跳转）指令在比较无符号数据时，如果第一个操作数不大于第二个操作数，就把程序执行转向指定位置。JNA 指令和 JBE 指令完成相同的功能。下例定义了 JNA 指令的语法：

```
jna label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNA 指令的汇编语言语句：

```
          .
          .
          cmp   bl,55h       ;Jump if
          jna   next         ; bl <= 55h
          .
          .
next:     .
          .
          .
```

5.8.13 JNAE 指令

JNAE（若不大于或等于则跳转）指令比较在无符号数据时，如果第一个操作数不大于或等于第二个操作数，就把程序执行转向指定位置。JNAE 指令和 JB 指令完成相同的功能。下例定义了 JNAE 指令的语法：

```
jnae  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNAE 指令的汇编语言语句：

```
          .
          .
          cmp   al,55 ;Jump if AL
          jnae  next  ; is < 55
          .
          .
next:     .
          .
          .
```

5.8.14 JNB 指令

JNB（若不小于则跳转）指令在比较无符号数据时，如果第一个操作数不小于第二个操作数，就把程序执行转向指定位置。JNB 指令和 JAE 指令完成相同的功能。下例定义了 JNB 指令的语法：

```
jnb  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNB 指令的汇编语言语句：

```
        .
        .
        cmp   ax,578      ;Jump if
        jnb   next        ; ax >= 578
        .
        .
next:   .
        .
        .
```

5.8.15 JNBE 指令

JNBE（若不小于或等于则跳转）指令在比较无符号数据时，如果第一个操作数不小于或等于第二个操作数，就把程序执行转向指定位置。JNBE 指令和 JA 指令完成相同的功能。下例定义了 JNBE 指令的语法：

```
jnbe  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNBE 指令的汇编语言语句：

```
        .
        .
        cmp   bx,5504     ;Jump if
        jnbe  next        ; BX > 5504
        .
        .
next:   .
        .
        .
```

5.8.16 JNC 指令

JNC（若无进位则跳转）指令在无进位标志置位时把程序执行转向指定位置。下例定义了 JNC 指令的语法：

```
jnc  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNC 指令的汇编语言语句：

```
          .
          .
          sub   al,55
          jnc   next       ;Jump if AL - 55 didn't set the
                           ; carry flag
          .
          .
next:     .
          .
          .
```

5.8.17 JNE 指令

JNE（若不等于则跳转）指令在数据比较时如果两个操作数不相等，就把程序执行转向指定位置。JNE 指令和 JNZ 指令完成相同的功能。下例定义了 JNE 指令的语法：

```
jne  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNE 指令的汇编语言语句：

```
          .
          .
          cmp   al,54 ;Jump if
          jne   next  ; AL <> 54
          .
          .
next:     .
          .
          .
```

5.8.18 JNG 指令

JNG（若不大于则跳转）指令在比较带符号数据时，如果第一个操作数不大于第二个操作数，就把程序转向指定位置。JNG 指令和 JLE 指令完成相同的功能。下例定义了 JNG 指令的语法：

```
jng  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNG 指令的汇编语言语句：

```
        .
        .
        cmp   ax,-9999     ;Jump if
        jng   next         ; AX <= -9999
        .
        .
next:   .
        .
        .
```

5.8.19 JNGE 指令

JNGE（若不大于或等于则跳转）指令在比较带符号数据时，如果第一个操作数不大于或等于第二个操作数，就把程序执行转向指定位置。JNGE 指令和 JL 指令完成相同的功能。下例定义了 JNGE 指令的语法：

```
jnge  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNGE 指令的汇编语言语句：

```
        .
        .
        cmp   dx,-455      ;Jump if
        jnge  next         ; dx < -455
        .
        .
next:   .
        .
        .
```

5.8.20 JNL 指令

JNL（若不小于则跳转）指令在比较带符号数据时，如果第一个操作数不小于第二个操作数，就把程序执行转向指定位置。JNL 指令和 JGE 指令完成相同的功能。下例定义了 JNL 指令的语法：

```
jnl   label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNL 指令的汇编语言语句：

```
          .
          .
          cmp   bl,67 ;Jump if
          jnl   next  ; BL >= 67
          .
          .
          .
next:     .
          .
          .
```

5.8.21 JNLE 指令

JNLE（若不小于或等于则跳转）指令在带符号数据比较中，如果第一个操作数不小于或等于第二个操作数，就把程序执行转向指定位置。JNLE 指令和 JG 指令完成相同的功能。下例定义了 JNLE 指令的语法：

```
jnle   label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNLE 指令的汇编语言语句：

```
          .
          .
          cmp   al,-101     ;Jump if
          jnle  next        ; AL > -101
          .
          .
          .
next:     .
          .
          .
```

5.8.22 JNO 指令

JNO（若不溢出则跳转）指令在 overflow（溢出）标志未置位时把程序执行转向指定存

储器单元。下例定义了 JNO 指令的语法：

```
jno  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNO 指令的汇编语言语句：

```
        .
        .
        add   al,245       ;Jump if
        jno   next         ; AL + 245 didn't overflow
        .
        .
next:   .
        .
        .
```

5.8.23 JNP 指令

JNP(若无奇偶校验则跳转)指令在上条指令结果无奇偶校验时把程序执行转向指定的存储器单元。JNP 和 JPO 指令完成相同的功能。下例定义 JNP 指令的语法：

```
jnp  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNP 指令的汇编语言语句：

```
        .
        .
        and   al,al ;Jump if AL
        jnp   next  ; doesn't have parity
        .
        .
next:   .
        .
        .
```

5.8.24 JNS 指令

JNS(若无符号置位跳转)指令在上条指令结果未设置符号标志时把程序执行转向指定的存储器单元。下例定义了 JNS 指令的语法：

```
jns  label
```

其中：

· label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNS 指令的汇编语言语句：

```
            .
            .
            and   dx,dx ;Jump if
            jns   next  ; DX is positive
            .
            .
next:       .
            .
            .
```

5.8.25 JNZ 指令

JNZ(若非零则跳转)指令在上条指令结果未置位零标志时把程序执行转向指定存的储器单元。JNZ 指令和 JNE 指令完成相同的功能。下例定义了 JNZ 指令的语法：

```
jnz  label
```

其中：

· label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JNZ 指令的汇编语言语句：

```
            .
            .
            or    bx,bx ;Jump if
            jnz   next  ; BX <> 0
            .
            .
next:       .
            .
            .
```

5.8.26 JS 指令

JS(若有符号置位则跳转)指令在上条指令结果设置符号标志时把程序执行转向指定的存储器单元。下例定义了 JS 的语法：

```
js  label
```

其中：

· label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JS 指令的汇编语言语句：

```
            .
            .
            and   ax,ax ;Jump if
            js    next  ; AX is negative
            .
            .
next:       .
            .
            .
```

5.8.27 JO 指令

JO（若溢出则跳转）指令在上条指令结果设置溢出位时把程序执行转向指定的存储器单元。下例定义了 JO 指令的语法：

```
jo  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JO 指令的汇编语言语句：

```
            .
            .
            add   ax,55678    ;Jump if
            jo    next  ; AX + 55678 overflowed
            .
            .
next:       .
            .
            .
```

5.8.28 JP 指令

JP（若有奇偶校验则跳转）指令在上条指令的结果有奇偶校验时把程序执行转向指定的存储器单元。JP 指令和 JPE 指令完成相同的功能。下例定义了 JP 指令的语法：

```
jp  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JP 指令的汇编语言语句：

```
            .
            .
            and   ax,ax ;Jump if AX
            jp    next  ; has parity
            .
            .
next:       .
            .
            .
```

5.8.29 JPE 指令

JPE(若偶校验则跳转)指令在上条指令结果有偶校验时把程序执行转向指定的存储器单元。JPE 指令和 JP 指令完成相同的功能。下例定义了 JPE 指令的语法：

```
jpe  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JPE 指令的汇编语言语句：

```
            .
            .
            and   ax,ax ;Jump if AX
            jpe   next  ; has even parity
            .
            .
next:       .
            .
            .
```

5.8.30 JPO 指令

JPO(若奇校验则跳转)指令在上条指令结果有奇校验时把程序执行转向指定的存储器单元。JPO 指令和 JNP 指令完成相同的功能。下例定义了 JPO 指令的语法：

```
jpo  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JPO 指令的汇编语言语句：

```
.
.
and   al,al ;Jump if AL
jpo   next  ; had odd parity
```

```
        .
        .
next:   .
        .
        .
```

5.8.31 JCXZ 指令

JCXZ（若 CX 为零则跳转）指令在寄存器 CX 等于 0 时把程序执行转向指定的存储器单元。下例定义了 JCXZ 指令的语法：

```
jcxz  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 JCXZ 指令的汇编语言语句：

```
        .
        .
next:   .
        .
        .
        dec   cx     ;Decrement CX
        jcxz  next   ;Jump if CX = 0
        .
        .
```

5.9 重 复 指 令

虽然使用 8088 的跳转指令可以实现程序内循环，但是还有一些 8088 指令专门用于实现循环。这些循环指令有条件或无条件地转移程序执行。

5.9.1 LOOP 指令

LOOP（循环）指令重复循环指定的存储单元。每次执行 LOOP 指令时，CX 寄存器减 1，如果 CX 不等于 0，程序执行转向指定的存储单元。下例定义了 LOOP 指令的语法：

```
loop  label
```

其中：

• label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 LOOP 指令的汇编语言语句：

```
        .
        .
next:   .
```

```
        .
        .
        loop  next  ;Loop until CX = 0
        .
        .
```

5.9.2 LOOPE 指令

LOOPE（若等于则循环）指令在上次比较结果把零标志置位并且 CX 不等于 0 时重复循环指定的存储器单元。和 LOOP 指令一样，在每次执行 LOOPE 指令时 CX 减 1。LOOPE 和 LOOPZ 完成相同的功能。下例定义了 LOOPE 指令的语法：

```
loope  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 LOOPE 指令的汇编语言语句：

```
        .
        .
next:   .
        .
        cmp   ax,55 ;Loop while AX = 55
        loope next  ; and CX <> 0
        .
        .
```

5.9.3 LOOPNE 指令

LOOPNE（若不等于则循环）指令在上次比较结果未设置零标志并且 CX 不等于 0 时重复循环指定的存储器单元。和 LOOP 指令一样，每次执行 LOOPNE 时 CX 减 1。LOOPNE 和 LOOPNZ 完成相同的功能。下例定义了 LOOPNE 指令的语法：

```
loopne  label
```

其中：

- label 为距当前存储器单元不超过－128 或 127 字节的标号。

下例列举了一些使用 LOOPNE 指令的汇编语言语句：

```
        .
        .
next:   .
        .
        .
        cmp         ax,55       ;Loop while AX <> 55
        loopne      next        ; and CX <> 0
```

5.9.4 LOOPNZ 指令

LOOPNZ(若无零标志则循环)指令在未设置零标志并且CX不等于0时重复循环指定的存储器单元。和LOOP指令一样，每次执行LOOPNZ时CX减1。LOOPNZ和LOOPNE完成相同的功能。下例定义了LOOPNZ指令的语法：

```
loopnz  label
```

其中：

- label为距当前存储器单元不超过－128或127字节的标号。

下例列举了一些使用LOOPNZ指令的汇编语言语句：

```
next:
        or          al,al       ;Loop while AL <> 0
        loopnz      next        ; and CX <> 0
```

5.9.5 LOOPZ 指令

LOOPZ（若置位零标志则循环）指令在一条指令设置零标志并且CX不等于0时重复循环指定的存储器单元。和LOOP指令一样，每次执行LOOPZ时CX减1。LOOPZ和LOOPE完成相同的功能。下例定义了LOOPZ指令的语法：

```
loopz  label
```

其中：

- label为距当前存储器单元不超过－128或127字节的标号。

下例列举了一些使用LOOPZ指令的汇编语言语句：

```
next:
        and   ax,ax ;Loop while AX = 0
        loop  next  ; and CX <> 0
```

5.10 其 它 指 令

除了前面提到的所有指令之外，8088 还有许多完成实用任务的指令。这些指令完成诸如设置标志、清除标志、翻转标志之类的任务和空操作。

5.10.1 CLC 指令

CLC（清除进位标志）指令通过置 0 清除进位标志。下例定义了 CLC 指令的语法：

```
clc
```

下例列举了一个使用 CLC 指令的汇编语言语句：

```
.
.
clc          ; Clear the carry flag
.
.
```

5.10.2 CLD 指令

CLD（清除方向标志）指令通过置 0 清除方向标志。下例定义了 CLD 指令的语法：

```
cld
```

下例列举了一个使用 CLD 指令的汇编语言语句：

```
.
.
cld          ; Clear the direction flag
.
.
```

5.10.3 CLI 指令

CLI（清除中断标志）指令通过置 0 清除中断标志。中断标志清零时，所有的可屏蔽中断都被禁止。下例定义了 CLI 指令的语法：

```
cli
```

下例列举了一个使用 CLI 指令的汇编语言语句：

```
.
.
cli          ; Disable the interrupts
.
.
```

5.10.4 CMC 指令

CMC（进位标志取反）指令翻转进位标志的值。下例定义了 CMC 指令的语法：

```
cmc
```

下例列举了一个使用 CMC 指令的汇编语言语句：

```
.
.
cmc        ; Invert the carry flag
.
.
```

5.10.5 LAHF 指令

LAHF（把标志位装入 AH）指令把

- 进位标志装入 AH 的第 0 位
- 奇偶校验标志装入 AH 第 2 位
- 把辅助进位标志装入 AH 的第 4 位
- 把零标志装入 AH 的第 6 位
- 把符号标志装入 AH 的第 7 位

下例定义了 LAHF 指令的语法：

```
lahf
```

下例列举了一个使用 LAHF 指令的汇编语言语句：

```
.
.
lahf       ; Load AH with the flags
.
.
```

5.10.6 NOP 指令

NOP（空操作）指令不执行任何功能，它常用产生一个可寻址的存储单元。下例定义了 NOP 指令的语法：

```
nop
```

下例列举了一个使用 NOP 指令的汇编语言语句：

```
    .
    .
next: nop
```

5.10.7 SAHF 指令

SAHF（存入标志位）指令将：

- AH 的第 0 位装入进位标志
- AH 的第 2 位装入奇偶校验标志
- AH 的第 4 位装入辅助进位标志
- AH 的第 6 位装入零标志
- AH 的第 7 位装入符号标志

下例定义了 SAHF 指令的语法：

```
sahf
```

下例列举了一个使用 SAHF 指令的汇编语言语句：

```
.
.
sahf            ; Store AH in the flags
.
.
```

5.10.8 STC 指令

STC（设置进位标志）指令通过置 1 设置进位标志。下例定义了 STC 指令的语法：

```
stc
```

下例列举了一个使用 STC 指令的汇编语言语句：

```
.
.
stc             ; Set the carry flag
.
.
```

5.10.9 STD 指令

STD（设置方向标志）指令通过置 1 设置方向标志。下例定义了 STD 指令的语法：

```
std
```

下例列举了一个使用 STD 指令的汇编语言语句：

```
.
```

```
.
std             ; Set the direction flag
.
.
```

5.10.10 STI 指令

STI（设置中断标志）指令通过置 1 设置中断标志。中断标志置位时，激活所有的可屏蔽中断。下例定义了 STI 的语法：

```
sti
```

下例列举了一个使用 STI 指令的汇编语言语句：

```
.
.
sti             ; Enable the interrupts
.
.
```

5.11 小　结

本章介绍了各种 8088 汇编语言指令。这些指令完成各种各样的操作，诸如在寄存器和存储单元中的数据存贮、取出数据、算术运算、无条件和有条件的程序执行转向、标志操作等等。这些是 8088 微处理器指令的大部分内容，后面的章节还将讲授一些编写高效汇编语言程序所需的其它指令。

第六章 寻 址 方 式

8088 微处理器使下列用四种基本的寻址方式访问几乎所有的数据：立即寻址方式、寄存器寻址方式、直接寻址方式和间接寻址方式。本章将讨论：

- 四种寻址方式的内容
- 怎样在 8088 汇编语言程序中使用这四种寻址方式

6.1 立即寻址方式

立即寻址方式使用立即操作数（immediate operands）访问数据。立即操作数是汇编时确定的数值常数。8088 的许多指令可以使用立即操作数。对于同时要求源操作数和目的操作数的指令，不能把立即操作数作为目的操作数，它受到逻辑上的限制。指令的目的操作数是指令存放运算结果的单元。因此，为常数赋新值是不合逻辑的。下例列举了一些使用立即寻址方式的汇编语言语句：

```
    .
    .
cr  equ   13          ;Define a carriage return constant
    .
    .
    mov   al,55       ;Loads AL with immediate value 55
    xor   bx,33h      ;XORs AX with immediate value 33h
    mov   ah,cr       ;Loads AH with immediate value cr
    sub   cx,6678     ;Subtract immediate value 6678
                      ; from CX
    .
    .
```

6.2 寄存器寻址方式

寄存器寻址方式使用寄存器操作数（register operands）访问数据。由于访问的数据是指令执行时寄存器保存的值，所以寄存器寻址方式常常称为寄存器直接寻址方式。因为任何 8088 指令都可以使用寄存器操作数，所以寄存器寻址方式可以说是最常用的 8088 寻址方式。下例列举了一些使用寄存器寻址方式的汇编语言语句：

```
        .
        .
array   dw    10 dup (?)
        .
        .
        mov   ax,bx        ;Both operands are regis-
                           ; ter operands
        sub   al,55        ;The destination is a
                           ; register operand
        mov   bx           ;The only operand is a
                           ; register operand
        mov   array,ax     ;The source is a register
                           ; operand
        .
        .
```

6.3 直接存储器寻址方式

直接存储器寻址方式使用直接存储器操作数（direct memory operands）访问数据。直接存储器操作数是在汇编时计算出的偏移地址。除非使用段超越运算符，8088 利用 DS 段寄存器计算计算机存储器中直接存储器地址的实际位置。下例列举了一些使用直接存储器寻址方式的汇编语言语句：

```
        .
        .
value   dw    100h
        .
        .
        mov   bx,value     ;Use direct memory
                           ; addressing to load BX with
                           ; 100h
        mov   value,34     ;Use direct memory
                           ; addressing to save a new
                           ; value
        .
        .
```

除了使用标号表示直接存储器地址之外，还可以使用数值常数表示它。但是使用数值常数时必须指定段。否则，汇编程序存储器地址错误地假定当前使用的是立即寻址方式。下例列举了一些在直接寻址方式中使用数值常数的汇编语言语句：

```
.
.
mov   al,ds:33h          ;Load AL with the value at
                         ; DS:33H
mov   ss:2,ax            ;Save AX at SS:02H
.
.
```

6.4 间接存储器寻址方式

间接寻址方式使用寄存器指向数据。在间接寻址方式中，寄存器很像高级语言中的指针。前面提到过，寄存器 BX 和 BP 是基址寄存器，DI 和 SI 是变址寄存器。在间接存储器寻址中只能使用这四个 16 位寄存器。这四个寄存器除了一点微小差别之外，基本上用法一致。

6.4.1 间接存储器寻址方式类型

间接存储器寻址方式有四种基本类型：

- 寄存器间接寻址方式
- 基址或变址寻址方式
- 基址变址寻址方式
- 带位移的基址变址寻址方式

6.4.1.1 寄存器间接寻址方式

在四种间接寻址方式中，最简单的是寄存器间接寻址方式。下例说明了使用寄存器间接寻址方式的语法：

[register]

其中：

- register 为 BX、BP、DI 或 SI。

如上例所示，寄存器以方括号括住。方括号是变址运算符，它告诉汇编程序寄存器中的值指向其它值，而不是用于寄存器直接寻址方式中使用的值。下例列举了一些使用寄存器间接寻址方式的汇编语言语句：

```
.
.
mov   ax,[dx]        ;Load AX with the value pointed to
                     ; by DI
add   [bx],al        ;Add al to the value pointed to by
                     ; BX
.
.
```

6.4.1.2 基址或变址寻址方式

和寄存器间接寻址方式一样，基址或变址寻址方式提供一个数值指针而不是实际数值。下例说明了使用基址或变址寻址方式的语法：

displacement [register]

或

[register+displacement]

其中：

- displacement（位移）为常数、存储器地址或两者的组合。
- register 为 BX、BP、DI 或 SI。

下例列举了一些使用基址或变址寻址方式的汇编语言语句：

```
          .
          .
table     db  100 dup (?)
          .
          .
          mov al,table[bx]        ;Load AL with the value
                                  ;pointed to by table and
                                  ; BX
          mov al,[table + bx]     ;Does the same as the
                                  ; above operation
          mov bl,6[di]            ;Load BL with the value
                                  ; pointed to by DI + 6
          mov cl,8 + [si] + table ;Load CL the value pointed
                                  ; to by SI + table + 8
          .
          .
```

6.4.1.3 基址变址寻址方式

和其它间接存储器寻址方式一样，基址变址寻址方式也提供一个数值指针而不是实际数值。下例说明了使用基址变址寻址方式的语法：

[register] [register]

或

[register+register]

其中：

- register 为 BX、BP、DI 或 SI。

注：在基址变址寻址方式中，必须使用一个基址寄存器和一个变址寄存器来访问数据。

下例列举了一些使用基址变址寻址方式的汇编语言语句：

```
.
.
mov   ax,[bx][si]        ;Load AX with the value
                         ; pointed to by BX and SI
mov   [bp][di],al        ;Save AL as the value
                         ; pointed to by BP and DI
add   dx,[bx + di]       ;Add the value pointed to
                         ; by BX and DI to DX
.
.
```

6.4.1.4 带位移的基址变址寻址方式

和其它三个间接寻址方式一样，此方式也是提供一个数值指针而不是实际数值。下例说明了使用带位移的基址变址寻址方式的语法：

displacement [register] [register]

或

[register+register+displacement]

其中：

- displacement 为常数、存储器地址或两者的组合。
- register 为 BX、BP、DI 或 SI。

注：和基址变址寻址方式一样，在带位移的基址变址寻址方式中必须使用一个基址寄存器和一个变址寄存器来访问数据。

下例列举了一些使用带位移的基址变址寻址方式的汇编语言语句：

```
        .
        .
table dp   1000 dup (?)
        .
        .
        mov al,table[bx][si]    ;Load AL with the value
                                ; pointe to by table + BX
                                ; + SI
        mov al,[table + bx + si] ;Does the same as the
                                ; above operation
        add ax,10[bp][di]       ;Add the value pointed to
                                ; by BP + DI + 10 to AX
```

6.5 小 结

本章介绍了 8088 微处理器提供的各种寻址方式。这些寻址方式以常数、寄存器和存储器中的变量和使用指针来访问数据。所有这些寻址方式和高级语言提供的寻址方式非常相似。

第七章　结构化程序设计

利用结构化编程技术，在8088汇编语言编程中可以采用许多高级语言中最重要的编程技术。结构化编程技术简化了程序实现，使程序具有更好的可读性，并且易于维护。因此，如果要成为一名真正的汇编语言程序员，彻底了解如何利用结构化编程技术编写8088汇编语言程序是必不可少的。本章讨论了如何利用三个相当简单的控制结构编写汇编语言程序的：

- 顺序
- 选择
- 重复

7.1　顺　序　控　制

顺序控制就是计算机程序逐条地执行语句。下面看几个在C语言程序中执行的赋值语句：

```
a=b+4;
c=c+10-a;
```

在顺序控制下，先执行语句a=b+4，然后执行语句c=c+10-a。现在看以8088汇编语言编写的相同赋值语句：

```
        .
        .
a       dw      ?
b       dw      ?
c       dw      ?
        .
        .
        mov     ax,b    ;AX = b
        add     ax,4    ;AX = b + 4
        mov     a;ax    ;a = b + 4
        mov     ax,c    ;AX = c
        add     ax,10   ;AX = c + 10
        sub     ax,a    ;AX = c + 10 - a
        mov     c,ax    ;c = c + 10 - a
        .
        .
```

虽然赋值语句的汇编语言形式比较长，但是这两种赋值语句以相同的顺序完成相同的

功能。从上述可见，顺序控制提供语句之间的顺序流程。如果没有顺序控制，程序将发生混乱。

7.2 选 择 控 制

选择控制结构使程序执行从顺序控制的语句顺序流程转向其它分支。在高级语言中，像goto、if…then 和 if…then…else 等语句实现选择控制结构。8088 的 JMP 指令就像高级语言中的 goto 指令一样。而且，使用条件跳转可以模拟 if…then 和 if…then…else 来控制 8088 汇编语言的结构。首先，看一下 C 的 if…then 结构：

```
if (a ! =b)
  c=3;
```

这个简单的 if…then 结构在 a 不等于 b 时将 c 赋值 3。现在看一看等价的 8088 汇编语句：

```
            .
            .
a           dw    ?
b           dw    ?
c           dw    ?
            .
            .
            mov   ax,a  ;AX = a
            cmp   ax,b  ;Is a equal to b?
            je    next  ;Jump if it is
            mov   c,3   ;c = 3
next:       .
            .
            .
```

注意，JE 指令是怎样代替 JNE 指令的。在上例的情形下，使用与测试条件相反的条件跳转通常更容易些。下面，我们再看一看程序中怎样使用 JNE 指令：

```
            .
            .
a           dw    ?
b           dw    ?
c           dw    ?
            .
            .
            mov   ax,a        ;AX = a
            cmp   ax,b        ;Is a equal to b?
            jne   nequal      ;Jump if it isn't
```

```
            jmp   next          ;Jump over the next statement
nequal:     mov   c,3           ;c = 3
next:       .
            .
            .
```

在8088汇编语言中，if…then…else控制结构的实现和if…then一样简单。我们先看C中一个简单的if…then…else控制结构：

```
if (a ! =b)
  c=3;
else
  c=4;
```

和上例一样，if…then…else语句在a不等于b时将c赋值3，否则为c赋值4。再看一看以8088汇编语言编写的上述if…then…else语句：

```
            .
            .
a           dw    ?
b           dw    ?
c           dw    ?
            .
            .
            mov   ax,a          ;AX = a
            cmp   ax,b          ;Is a equal to b?
            jne   nequal        ;Jump if it isn't
            mov   c,4           ;c = 4
            jmp   next          ;Jump
nequal:     mov   c,3           ;c = 3
next:       .
            .
            .
```

7.3 重　复　控　制

重复控制结构在条件未满足时无休止地重复执行一条或多条语句。在高级语言中，重复控制结构由for、while、repeat或do循环实现。与顺序控制和选择控制一样，在8088汇编语言程序中重复控制的实现相当容易。

7.3.1 For循环

我们先看一个C语言中的简单for循环：

```
for     (i = 0; i< 10; i++)
  a++;
```

此 for 循环将重复 10 次，每次 a 加 1。下面，我们再看一看在汇编语言程序中是如何实现相同循环的：

```
        .
        .
a       dw    ?
        .
        .
        mov   al,0  ;Initialize the counter
l1:     cmp   al,10 ;Jump if the
        jae   l2    ; loop is complete
        inc   a     ;Bump a
        inc   al    ;Bump the loop counter
        jmp   l1    ;Loop
l2:     .
        .
        .
```

虽然上述汇编语言示例也完成了任务，但它基本上是前面 C 语言示例的文本转换形式。因此，汇编语言代码可以认为是 C 编译的优化结果。然而，汇编语言编程是少数几个仅存的高于机器语言的计算机编程保留方法中的一员。下面是上述汇编语言示例的改进形式：

```
        .
        .
a       dw    ?
        .
        .
        mov   cx,10 ;Initialize the loop counter
l1:     inc   a     ;Bump a
        loop  l1    ;Loop till done
        .
        .
```

注意，此循环是怎样用三条指条代替上例中六条指令的。这种循环更加优越。不仅程序的整个长度减小了，而且循环内部指令的减少还显著地缩短了执行时间。当然，还可以更进一步，仅用一条语句 add ax，10，也可得到同样的结果。但是，这里考虑的焦点是表明 8088 汇编语言中的循环构成，而不是怎样把编译生成的代码压缩到最少。

7.3.2 While 循环

在汇编语言中实现 while 循环之前，我们先看一下 C 语言中的 while 循环：

```
while (a ! =10)
  a++;
```

可以看到，在 while 循环中 a 连续加 1，直到它等于 10 为止。现在，我们看看在 8088 汇编语言中是怎样编写同样的 while 循环代码的：

```
          .
          .
a         dw    ?
          .
          .
11:       cmp   a,10  ;Jump
          je    12    ; if done
          inc   a     ;Bump a
          jmp   11    ;Loop till done
12:       .
          .
          .
```

7.3.3 DO 循环

最后，我们看一看在汇编语言中怎样编写重复结构的代码，下面是 C 语言中的 do 循环：

```
do
  a++;
while (a ! =10)
```

和 while 循环一样，在 do 循环中 a 连续加 1，直到它等于 10 为止。下面，我们看看在汇编语言中怎样编写同样的循环代码：

```
          .
          .
a         dw    ?
          .
          .
11:       inc   a     ;Bump a
          cmp   a,10  ;Loop
          jne   11    ; until done
          .
          .
```

7.4 小　结

在本章中，我们学习了在汇编语言编程中使用顺序、选择、重复控制结构的方法。还列举了高级语言示例和等价的汇编语言示例，以加深对这些非常重要的编程技术的理解。

第八章 字 符 串

在大多数计算机程序中，最重要的数据块就是字符串，对于高级语言和汇编语言程序都是如此。因此，8088 具有一组为处理字符串而设计的专用指令。本章讨论了使用 8088 字符串指令的方法：

- 传送字符串
- 装入字符串
- 存储字符串
- 比较字符串
- 扫描字符串

8.1 MOVS、MOVSB 和 MOVSW 指令

MOVS（传送字符串）、MOVSB（传送字节串）和 MOVSW（传送字串）指令在存储单元之间中传送字符串。DS：SI 指向源串地址，ES：DI 指向目的串地址。每次执行传送指令时，DI 和 SI 根据方向标志增加或减少。方向标志如果由 CLD 指令清零，在每次字节串传送时 DI 和 SI 加 1，在字串传送时 DI 和 SI 加 2。反之，当方向标志被 STD 指令置位时，DI 和 SI 相应地减 1 或减 2。下例定义了 MOVS 指令的语法：

```
movs  destination, source
```

其中：

- destination 为传送字符串的目的地址。
- source 为字符串的当前位置。注意，可以使用段超越运算符超越缺省段地址。

注：除了指示源串和目的串的地址外，操作数还指出操作是字节传送还是字传送的。

下例定义了 MOVSB 指令的语法：

```
movsb
```

下例定义了 MOVSW 指令的语法：

```
movsw
```

下例列举了一些使用 MOVS 指令的汇编语言语句：

```
        .
        .
_DATA  segment word public 'DATA'
srcstr db      10 dup (?)
desstr db      10 dup (?)
```

```
_DATA  ends
       .
       .
_TEXT  segment word public 'CODE'
       assume  cs:_TEXT,ds:_DATA,es:_DATA
       .
       .
       mov     ax,_DATA      ;Set
       mov     ds,ax         ; both
       mov     es,ax         ; DS and ES
       lea     si,srcstr     ;DS:SI = Source string
                             ; pointer
       lea     di,desstr     ;ES:DI = Destination
                             ; pointer
       mov     cx,10         ;CX = Loop counter
       cld                   ;Clear the direction flag
11:    movs    desstr,srcstr ;Move a byte of data
       loop    11            ;Loop till the string is
                             ; moved
       .
       .
```

上述程序段简单地把一个 10 字符长的字符串从某个存储单元送至另一个存储单元。下面的程序段完成相同的任务，但它使用 MOVSB 指令替代 MOVS 指令：

```
       .
       .
_DATA  segment word public 'DATA'
srcstr db      10 dup (?)
desstr db      10 dup (?)
_DATA  ends
       .
       .
_TEXT  segment word public 'CODE'
       assume  cs:_TEXT,ds:_DATA,es:_DATA
       .
       .
       mov     ax,_DATA    ;Set
       mov     ds,ax       ; both
       mov     es,ax       ; DS and ES
       lea     si,srcstr   ;DS:SI = Source string
                           ; pointer
       lea     di,desstr   ;ES:DI = Destination
```

```
                                ; pointer
        mov     cx,10           ;CX = Loop counter
        cld                     ;Clear the direction flag
11:     movsb                   ;Move a byte of data
        loop    11              ;Loop till the string is
                                ; moved
        .
        .
```

正如所看到的，上述程序段和前面的没有多大差别。下面的程序段和前面两例类似，只是它在存储单元之间传送一个 5 字长的字符串：

```
        .
        .
_DATA   segment word public 'DATA'
srcstr  dw      5 dup (?)
desstr  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA        ;Set
        mov     ds,ax           ; both
        mov     es,ax           ; DS and ES
        lea     si,srcstr       ;DS:SI = Source string
                                ; pointer
        lea     di,desstr       ;ES:DI = Destination
                                ; pointer
        mov     cx,5            ;CX = Loop counter
        cld                     ;Clear the direction flag
11:     movsw                   ;Move a byte of data
        loop    11              ;Loop till the string is
                                ; moved
        .
        .
```

8.1.1 REP 前缀

8088 提供 REP（重复）前缀以简化字符串指令的使用。每次执行带 REP 前缀的 8088 字符串指令后，CX 寄存器减 1。如果 CX 不等于 0，重新执行字符串指令。否则，在 CX 等

于0时，字符串指令就不再重复执行。实质上，REP前缀只是取代了LOOP指令。下列程序段和前面的MOVSW示例基本一样，只是使用REP前缀代替了LOOP指令：

```
        .
        .
_DATA   segment word public 'DATA'
srcstr  dw      5 dup (?)
desstr  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ;both
        mov     es,ax       ;DS and ES
        lea     si,srcstr   ;DS:SI = Source string
                            ; pointer
        lea     di,desstr   ;ES:DI = Destination
                            ; pointer
        mov     cx,5        ;CX = Loop counter
        cld                 ;Clear the direction flag
        rep     movsw       ;Move the string
        .
        .
```

8.2 LODS、LODSB和LODSW指令

LODS（装入字符串）、LODSB（装入字节串）和LODSW（装入字串）指令在累加器（寄存器AL或AX）中装入字节值或字值。DS：SI指向该串。每次装入指令执行后，SI根据方向标志值增加或减少。方向标志如果由CLD指令清零，在字节装入时SI每次加1，在字装入时每次加2。反之，在方向标志被STD指令置位时，在字节装入时SI每次减1，在字装入时每次减2。下例定义了LODS指令的语法：

lods source

其中：

• source为字符串的当前位置。可以用段超越运算符超越当前缺省地址。

注：除了指出字符串的当前地址外，源操作数还指出了送入累加器的是字节还是字。

下例定义了LODSB指令的语法：

lodsb

下例定义了 LODSW 指令的语法：

lodsw

下例列举了一些使用 LODS 指令的汇编语言语句：

```
        .
        .
_DATA   segment word public 'DATA'
scores  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; DS
        lea     si,scores   ;DS:SI = Source pointer
        mov     cx,10       ;CX = Loop counter
        mov     bl,0        ;BL = Total
        cld                 ;Flag increment
11:     lods    scores      ;AL = Next byte
        add     bl,al       ;Adjust the total
        loop    11          ;Loop till done
                .
                .
```

上述程序段使用 LODS 指令计算 10 个字节值的和。下列程序段完成同样的任务，只是使用 LODSB 指令替代 LODS 指令：

```
        .
        .
_DATA   segment word public 'DATA'
scores  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
```

```
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; DS
        lea     si,scores   ;DS:SI = Source pointer
        mov     cx,10       ;CX = Loop counter
        mov     bl,0        ;BL = Total
        cld                 ;Flag increment
11:     lodsb               ;AL = Next byte
        add     bl,al       ;Adjust the total
        loop    11          ;Loop till done
        .
        .
```

下列程序段和上面两例类似，只是它计算五个字值的和：

```
        .
        .
_DATA   segment word public 'DATA'
scores  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ;DS
        lea     si,scores   ;DS:SI = Source pointer
        mov     cx,5        ;CX = Loop counter
        mov     bx,0        ;BX = Total
        cld                 ;Flag increment
11:     lodsw               ;AX = Next byte
        add     bx,ax       ;Adjust the total
        loop    11          ;Loop till done
        .
        .
```

8.3 STOS、STOSB 和 STOSW 指令

STOS（存储字符串）、STOSB（存储字节串）和 STOSW（存储字串）指令把字节或字值存入字符串单元中。ES：DI 指向该串，欲存储的字符串存放在相应的累加器（寄存器 AL 或 AX）中。每次执行字符串存储指令后，DI 根据方向标志值增减。如果方向标志由 CLD

指令清零，在字节存储时 DI 每次加 1，在字存储时每次加 2。反之，当方向标志被 STD 指令置位时，在字节存储时 DI 每次减 1，在字存储时每次减 2。下例定义了 STOS 指令的语法：

```
stos   destination
```

其中：

• destination 为字符串的当前位置。

注：除了指出字符串的当前地址外，目的操作数还指示了从相应累加器中移出的是字节还是字。

下例定义了 STOSB 指令的语法：

```
stosb
```

下例定义了 STOSW 指令的语法：

```
stosw
```

下例列举了一些使用 STOS 指令的汇编语言语句：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ;ES
        lea     di,buffer   ;ES:DI = Buffer pointer
        mov     cx,10       ;CX = Loop counter
        mov     al,0        ;AL = Value to be stored
        cld                 ;Flag increment
11:     stos    buffer      ;Save a 0
        loop    11          ;Loop till done
        .
        .
```

上述程序段使用 STOS 指令在 10 字节的缓冲区中填零。下面的程序段完成同样的任务，只不过使用 STOSB 指令替代了 STOS 指令：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ;ES
        lea     di,buffer   ;ES:DI = Buffer pointer
        mov     cx,5        ;CX = Loop counter
        mov     ax,0        ;AX = Value to be stored
        cld                 ;Flag increment
11:     stosw               ;Save a 0
        loop    11          ;Loop till done
        .
        .
```

下面的程序段类似于前面两例，不同之处只是在5个字的缓冲区中填零：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ;ES
        lea     di,buffer   ;ES:DI = Buffer pointer
        mov     cx,10       ;CX = Loop counter
        mov     al,0        ;AL = Value to be stored
        cld                 ;Flag increment
11:     stosb               ;Save a 0
        loop    11          ;Loop till done
        .
        .
```

8.3.1 REP 前缀

和 MOVS 类指令一样，REP 前缀也可以简化 STOS 类指令的使用。下面的程序段和 STOSW 示例相似，不过使用 REP 前缀代替了 LOOP 指令：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ;ES
        lea     di,buffer   ;ES:DI = Buffer pointer
        mov     cx,5        ;CX = Loop counter
        mov     ax,0        ;AX = Value to be stored
        cld                 ;Flag increment
        rep     stosw       ;Zero out the buffer
        .
        .
```

8.4 CMPS、CMPSB、CMPSW 指令

CMPS（比较字符串）、CMPSB（比较字节串）和 CMPSW（比较字串）指令比较字符串。DS：SI 指向比较的源串，ES：DI 指向目的串。在源串和目的串比较中，每次处理一个字节或字。每次比较后，标志相应地更改，DI 和 SI 根据方向标志值增减。如果方向标志由 CLD 指令清零，在字节串比较中，DI 和 SI 每次加 1，在字串比较中，每次加 2。反之，当方向标志被 STD 指令置位时，在字节串比较中，DI 和 SI 每次减 1，在字串比较中，每次减 2。下例定义了 CMPS 指令的语法：

cmps source，destination

其中：

- destination 为目的串地址。
- source 为源串地址。注意，用段超越运算符可以超越缺省段地址。

注：除了指出源串和目的串地址外，操作数还指示了操作为字节比较还是字比较。

下例定义了 CMPSB 指令的语法：

cmpsb

下例定义了 CMPSW 指令的语法：

cmpsw

下例列举了一些使用 CMPS 指令的汇编语言语句：

```
        .
        .
_DATA   segment word public 'DATA'
srcstr  db      10 dup (?)
desstr  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA        ;Set
        mov     ds,ax           ;both
        mov     es,ax           ;DS and ES
        lea     si,srcstr       ;DS:SI = Source string
                                ; pointer
        lea     di,desstr       ;ES:DI = Destination
                                ; string pointer
        mov     cx,10           ;CX = String length
        cld                     ;Flag increment
l1:     cmps    srcstr,desstr   ;Compare the strings
        jne     next            ;Jump if the bytes aren't
                                ; the same
        loop    l1              ;Loop till the end of the
                                ; strings
next:   .
        .
        .
```

上述程序段简单地比较了两个 10 字节字符串。下面的程序段完成同样的任务，不过使用 CMPSB 指令替代了 CMPS 指令：

```
        .
        .
_DATA   segment word public 'DATA'
srcstr  db      10 dup (?)
desstr  db      10 dup (?)
```

```
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ;both
        mov     es,ax       ;DS and ES
        lea     si,srcstr   ;DS:SI = Source string
                            ; pointer
        lea     di,desstr   ;ES:DI = Destination
                            ;string pointer
        mov     cx,10       ; CX = String length
        cld                 ;Flag increment
11:     cmpsb               ;Compare the strings
        jne     next        ;Jump if the bytes aren't
                            ; the same
        loop    11          ;Loop till the end of the
                            ; strings
next:   .
        .
        .
```

下面的程序段类似于前面两例，不同之处在于它比较两个 5 字长的字符串：

```
        .
        .
_DATA   segment word public 'DATA'
srcstr  dw      5 dup (?)
desstr  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; both
        mov     es,ax       ; DS and ES
        lea     si,srcstr   ;DS:SI = Source string
                            ; pointer
```

```
        lea     di,desstr   ;ES:DI = Destination string
                            ; pointer
        mov     cx,5        ;CX = String length
        cld                 ;Flag increment
11:     cmpsw               ;Compare the strings
        jne     next        ;Jump if the words aren't
                            ; the same
        loop    11          ;Loop till the end of the
                            ; strings
next:   .
        .
        .
```

8.4.1 REPE 和 REPNE 前缀

除了 REP 前缀外，8088 还提供 REPE（或 REPZ）和 REPNE（或 REPNZ）前缀简化 CMPS 类指令的使用。当执行带 REPE 前缀的 8088 字符串指令时，CX 减 1，并检查零标志。如果 CX 等于 0 并且在比较中置位零标志，那么字符串指令就不再重复执行；否则，就一直重复执行。当执行带 REPNE 前缀的字符串指令时，CX 减 1，并检查零标志。如果 CX 等于 0 并且在比较中未置位零标志，字符串指令停止重复执行；否则，就一直重复执行。下面的程序段除了使用 REPE 指令替代 LOOP 指令之外，和 CMPSW 示例完全一致：

```
        .
        .
_DATA   segment word public 'DATA'
srcstr  dw      5 dup (?)
desstr  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; both
        mov     es,ax       ; DS and ES
        lea     si,srcstr   ;DS:SI = Source string
                            ; pointer
        lea     di,desstr   ;ES:DI = Destination string
                            ; pointer
```

```
mov     cx,5        ;CX = String length
cld                 ;Flag increment
repe    cmpw        ;Compare the strings
.
.
```

8.5 SCAS、SCASB 和 SCASW 指令

SCAS（扫描字符串）、SCASB（扫描字节串）、和 SCASW（扫描字串）指令在字符中扫描累加器（寄存器 AL 或 AX）中指定的值。ES：DI 指向该串。串值和累加器比较时每次扫描字符串的一个字节或字。每次比较后，标志相应地更新，DI 根据方向标志值增减。如果方向标志由 CLD 指令清零，在字节串扫描时，DI 每次加 1，在字串扫描时，每次加 2。反之，当方向标志被 STD 指令置位时，在字节扫描中，DI 每次减 1。在字串扫描时，每次减 2。下例定义了 SCAS 指令的语法：

scas destination

其中：

• destination 为扫描累加器值的字符串地址。

注：除了指出目的串地址外，操作数还指示了操作为字节扫描还是字扫描。

下例定义了 SCASB 指令的语法：

scasb

下例定义了 SCASW 指令的语法：

scasw

下面的程序段是汇编语言程序使用 SCAS 指令的范例：

```
        .
        .
_DATA   segment word public 'DATA'
string  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ;ES
        lea     di,string   ;ES:DI = String pointer
```

```
        mov     cx,10       ;CX = String length
        cld                 ;Flag increment
        mov     al,0ffh     ;AL = Value to scan for
11:     scas    string      ;Compare the string byte
                            ; with AL
        je      next        ;Jump if 0FFh is found
        loop    11          ;Continue with the scan
next:   .
        .
        .
```

可以看到，上述程序段在10字节的字符串中扫描0FFH。下面的程序段完成同样的任务，不过使用SCASB指令替代了SCAS指令：

```
        .
        .
_DATA   segment word public 'DATA'
string  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ;ES
        lea     di,string   ;ES:DI = String pointer
        mov     cx,10       ;CX = String length
        cld                 ;Flag increment
        mov     al,0ffh     ;AL = Value to scan for
11:     scasb               ;Compare the string word
                            ;with AX
        je      next        ;Jump if 0FFh is found
        loop    11          ;Continue with the scan
next:   .
        .
        .
```

下面的程序段类似于前面两例，不同之处在于它使用SCASW指令扫描5字长的字符串：

```
        .
        .
_DATA   segment word public 'DATA'
```

```
string  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ; ES
        lea     di,string   ;ES:DI = String pointer
        mov     cx,5        ;CX = String length
        cld                 ;Flag increment
        mov     AX,03effh   ;AX = Value to scan for
11:     scasw               ;Compare the string byte
                            ; with AL
        je      next        ;Jump if 03eFFh is found
        loop    11          ;Continue with the scan
next:   .
        .
        .
```

8.5.1 REPE 和 REPNE 前缀

和 CMPS 类指令一样，REPE 和 REPNE 指令简化了 SCAS 指令的使用。下面的程序段类似于 SCASW 示例，只是使用 REPNE 指令替代了 LOOP 指令：

```
        .
        .
_DATA   segment word public 'DATA'
string  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA,es:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ; ES
        lea     di,string   ;ES:DI = String pointer
        mov     cx,5        ;CX = String length
        cld                 ;Flag increment
        mov     AX,03effh   ;AX = Value to scan for
```

```
repne   scasw       ;Compare the string word
                    ; with AX
.
.
```

8.6 小 结

本章介绍了字符串指令。这些指令完成诸如传送字符串、装入字符串、存储字符串、比较字符串和扫描字符串等操作。而且本章还介绍了怎样结合 8088 重复前缀和字符串指令简化程序的实现。

第九章 结 构 和 记 录

虽然程序处理中最常用的数据类型是数字和字符串，但是还有一些数据类型只能由其它数据类型组合来表示。本章将讨论：

- 结构，它表示多种数据类型
- 记录，为编写完成位操作的程序提供方便

9.1 结 构

8088 汇编语言提供了结构类型，以使程序员能够提供由其它数据类型组合生成新数据类型的结构。使用汇编语言结构的第一步是通过建立结构样板来定义结构类型：

```
name struc
field declaration
.
.
field declaration
name ends
```

其中：

- name 为结构类型名称
- field declaration 为字段说明

如上例所示，一个结构类型定义由许多字段说明组成。这些字段说明和数据说明的语法一样。下面是一个包含某人的姓氏、名字、出生月份、出生日期和出生年份的结构类型定义：

```
person   struc
fname    db    'First name goes here'
lname    db    'Last name goes here'
month    dw    ?
day      dw    ?
year     dw    ?
person   ends
```

注意，每个字段说明都有一个名字。字段名可以是在汇编语言程序中未使用过的任何合法的标识符。因此，在使用上述结构的程序中不能在其它地方把 fname 用作变量标识符。还要注意，数值或字符串的字段长度必须保证能够存放该字段中可能的最大数值或最长的

字符串。

9.1.1 结构变量

定义结构类型后，通过在程序中定义一个结构变量，即可以在程序中使用它。下例表示了定义结构变量的语法：

name strucname initial value，initial value

其中：

- name 为变量标识符。
- strucname 为结构类型名。
- initial value 为字段的初始值。

如上例所示，在结构变量说明中可，以定义结构变量字段的初始值。如果在结构变量说明中未定义初始值，就使用该字段的缺省值。下例列举了一些结构变量说明：

```
p1 person     <' John', 'Smith', 2, 4, 81 >
p2 person     <' Jane', 'Doe',, 12, 83 >
```

注意，第二个例子未给出 month（月份）的初始值。这样，由于 month 字段的缺省值为?，所以 p2's month 就没有定义。还要注意，拥有多个缺省值的字段不能用结构变量说明中的初始值替代。现在看看下列结构类型定义：

```
person   struc
fname    db    'First name goes here'
lname    db    'Last name goes here'
birthday dw    3 dup (?)
person   ends
```

可以看到，新结构类型是前面 person 结构类型的改进。然而，新类型把两个出生日期组合成一个字段。在结构变量说明中仍然可以初始化前两个字段，但是由于 birthday 值不只一个，所以它不能如此初始化。因此，birthday 字段总是被指定为新的 person 结构类型说明的变量缺省值。

9.1.2 引用结构字段

读者已学会了定义结构类型以及使用建立的结构类型说明变量的方法，现在我们讨论在汇编语言指令中如何引用字段值，下例说明了结构字段的引用：

variable. field

其中：

- variable 为变量名。
- field 为字段名。

为了更好地说明汇编语言程序中如何使用结构，可参考下面的程序段，它使用先前说

明的 person 类型结构完成各种运算：

```
                .
                .
person struc
fname           db          'First name goes here'
lname           db          'Last name goes here '
month           dw          ?
day             dw          ?
year            dw          ?
person          ends
                .
                .
_DATA           segment     word public 'DATA'
somebody        person <'John', , , , >
last   db       'Doe'
_DATA  ends
                .
                .
_TEXT           segment     word public 'CODE'
                assume cs:_TEXT,ds:_DATA,es:_DATA
                mov    ax,_DATA                ;Set
                mov    ds,ax                   ; both
                mov    es,ax                   ; DS and ES
                mov    si,offset last          ;DS:SI = The person's
                                               ; last name string
                mov    di,offset somebody.lname ;ES:DI = Destina
                                               ; tion pointer
                mov    cx,3                    ;CX = String length
                cld                            ;Flag increment
                rep    movsb                   ;Save the new last
                                               ; name
                mov    somebody.month,10       ;Set the person's
                                               ; birth month
                mov    dx,9                    ;Set the
                mov    somebody.day,dx         ;Set the person's
                                               ; birthday
                mov    somebody.year,1990      ;Set the person's
                                               ; birth year
                .
                .
```

9.2 记 录

就像在汇编语言程序中使用结构之前要求类型定义一样，在使用记录之前也必须完成类型定义。为此，建立一个样板。记录类型的样板和结构样板类似，不过它处理的是字位而不是多种数据类型。下例定义了创建记录类型样板的方法：

```
name  record  field, field
```

其中：

- name 为记录类型名。
- field 为字段说明。

如上例所示，记录类型定义需要一个或多个字段说明。这些字段说明指定字段宽和一个可选缺省值。创建记录类型时，必须牢记记录类型中位数最多为 16 位。如果记录类型使用位数少于 8 位，其值作为字节存储。如果长度大于 8 位，其值作为字存储。下例定义了说明了字段宽和一个可选缺省值的语法：

```
name: width=expression
```

其中：

- name 为字段名。
- width 为字段宽。
- expression 为字段的可选缺省值。

为了更好地说明在汇编语言程序中如何使用记录类型，请参考图 9-1，它说明了是如何在字节中存储 IBM PC 颜色代码的。

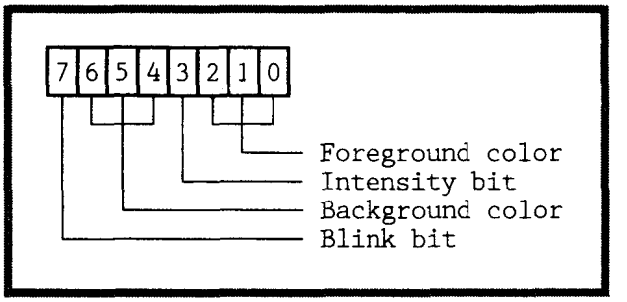


图 9-1 在一个字节值中存储的 IBM PC 颜色代码

如图 9-1 所示，IBM PC 颜色代码字节是一个记录类型的特例。下面定义了颜色代码字节的记录类型：

```
colorcode  record  blink: 1, back: 3, intensity: 1, fore: 3
```

9.2.1 记录变量

定义了记录类型后，在汇编语言程序中可以使用它定义记录变量。下例说明了定义一个记录变量的语法：

```
name   recordname      <initial value, initial value, etc>
```

其中：

- name 为变量标识符。
- recordname 为记录类型名。
- initial value 为字段的初始值。

如上例所示，在记录变量说明中，可以定义记录变量字段的初始值。如果未定义初始值，就使用缺省字段值。下例列举了一些记录变量说明：

```
c1 color    <0, 0, 1, 5 >
array       color 20 dup (< >)
c2 color    < 1, 2, 1, 7 >
```

为了更好地说明在汇编语言程序中是如何使用记录的，请参考下面的程序段，它使用记录类型 colorcode 中的操作数和变量完成各种运算：

```
            :
            :
colorcode record    blink:1,back:3,intensity:1,fore:3
            .
            .
_DATA       segment   word public 'DATA'
c1          colorcode <0,3,1,6>
_DATA       ends
            .
            .
_TEXT       segment   word public 'CODE'
            assume    cs:_TEXT,ds:_DATA
            mov       ax,_DATA          ;Set
            mov       ds,ax             ; DS
            mov       al,c1             ;Load AL with the
                                        ; color code
            mov       ah,color <1,3,1,5> ;Load AH with a
                                        ; color code
            mov       c1,ah             ;Save the new
                                        ; color code
            .
            .
```

注意，在上述程序段中，把表达式 color ＜1，3，1，5 ＞作为一个操作数。在 8088 汇编语言中，不仅可以把记录作为操作数，还有两个在表达式中专门使用记录工作的运算符：MASK 和 WIDTH。

9.2.2 MASK 运算

MASK（屏蔽）运算返回特定记录字段的位屏蔽值。屏蔽结果把该字段的位置 1，字段外的其它位置 0。另外，MASK 运算还可以返回整个记录的位屏蔽值。对于字段的屏蔽结果，记录中的所有位置 1，记录外的所有位置 0。下例定义了 MASK 运算的语法：

mask　field name or record

其中：

- field name 为指定的字段名。
- record 为定义过的记录变量。

下面的程序段说明了在汇编语言程序中是如何使用 MASK 运算符的：

```
                .
                .
colorcode       record      blink:1,back:3,inten-
sity:1,fore:3
                .
                .
_DATA           segment   word public 'DATA'
c1              colorcode <1,0,1,5>
_DATA           ends
                .
                .
_TEXT           segment   word public 'CODE'
                assume    cs:_TEXT,ds:_DATA
                .
                .
                mov       al,c1                ;AL = Color
                                               ; code
                and       al,not mask blink    ;Turn off the
                                               ; blinking
                and       al,not mask intensity ;Turn off the
                .                              ; intensity
                .
```

9.2.3 WIDTH 运算

WIDTH（宽度）运算返回字段或记录的位数，下例定义了 WIDTH 运算的语法：

width　field name or record

其中：

- field name 为指定的字段名。
- record 为定义过的记录变量。

下面的程序段说明了在汇编语言程序中是如何使用 WIDTH 运算符的：

```
            .
            .
colorcode record  blink:1,back:3,intensity:1,fore:3
            .
            .
_TEXT       segment word public 'CODE'
            assume  cs:_TEXT,ds:_DATA
            .
            .
            mov     al,width blink ;AL = Blink width
            add     al,width back  ;AL = Blink width +
                                   ; back width
            .
            .
```

9.2.4 引用记录字段

当把记录字段当作操作数时，结果返回该字段最低有效位的地址。例如，colorcode 的 fore 字段返回值 0，intensity 字段返回值 3，back 字段返回 4，blink 字段返回 7。下面的程序段说明了在汇编语言程序中是如何使用记录字段的地址的：

```
            .
            .
colorcode record    blink:1,back:3,intensity:1,fore:3
            .
            .
_DATA       segment   word public 'DATA'
c1          colorcode <1,0,1,5>
_DATA       ends
            .
            .
_TEXT       segment   word public 'CODE'
            assume    cs:_TEXT,ds:_DATA
            .
            .
            mov       al,c1         ;AL = Color code
            and       al,mask back ;Strip away all but
                                   ; the background
            mov       c1,back       ;CL = Background
```

```
                        ; position
shr         al,cl       ;Move it to the
                        ; byte's least sig-
                        ; nificant bits
.
.
```

9.3 小 结

在本章中，读者学习了8088汇编语言结构和记录。本章还介绍了如何使用结构把各种数据类型组合成一个新数据类型以及如何使用记录把位串组成字节和字。另外，本章还介绍了使用MASK和WIDTH运算处理记录的方法。

第十章 堆 栈

本章将讨论 8088 汇编语言程序设计最重要的方面——堆栈。堆栈的一个最主要用途是：临时存储 8088 微处理器和汇编语言程序的数据。本章的重点在于讲述汇编语言程序如何使用堆栈。

10.1 堆 栈

所有的汇编语言程序都需要划出一段存储器区域作为堆栈来使用。堆栈的段地址存储在段地址寄存器 SS 中。为了更好地理解堆栈，可以把它看做一摞纸。对于这一摞纸来讲，它总是从底部向上一层层码起来的。要从这堆纸中找出所需要的某张纸，也只能从这堆纸的上部一层层移去，直至遇到所需寻找的纸。

对于大部分内容而言，计算机堆栈功能非常类似于上面所提的一摞纸。主要不同之处是堆栈和纸的形成方向不同：纸是从下往上，而堆栈是从上往下。虽然堆栈的形成方向有背于常理，但在计算机里这是最有效的方式。在计算机中，堆栈的形成顺序是和计算机内存中装载程序的方式密切相关的。

在计算机向内存装载程序时，程序的可执行代码首先装载。紧接着代码段的是程序所需的数据区。由于大多数程序在运行前是不可能预知所需数据占用多大内存的，因此在数据区后紧接着放置任何东西都将是不安全的。数据区会随着程序的执行而动态地变化。因此，堆栈放在哪里呢？很简单，堆栈放在存储器的高端。从高地址向下扩建。当然，会有这种可能性发生；向上生长的数据区覆盖向下生长的堆栈。然而，在了解了汇编语言程序设计时，就可确保这种情况不会发生，在汇编程序中定义一个合理的堆栈大小，保证不和数据区发生冲突。下图说明了程序的代码段、数据段及堆栈在存储器中的分布情况。

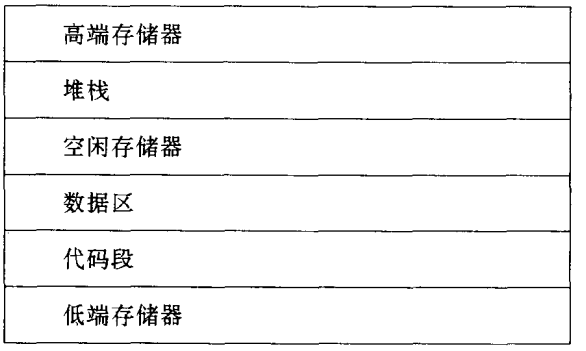


图 10-1 程序中的存储器使用

10.2 往堆栈中置数和从堆栈中取数

正如读者所知道的，在往堆栈中置入数据时，数据是按照从上至下的顺序存放的。在堆栈中，SP 寄存器总是指向放于堆栈中的最后一个数据单元，以此来保证堆栈中数据的顺序。在 8088 堆栈中，只允许存放 16 位的字。使用 8088 提供的 PUSH 指令在堆栈中置数，下面的例子说明了使用 PUSH 指令的语法：

```
push operand
```

这里：

· operand 是 AX、BX、CX、DX、SP、BP、SI、DI、ES、CS、SS、DS 或 Mem16。

通过使用 8088 提供的 POP 指令，一个 16 位字可以从堆栈的顶部（或说堆栈底部）移出。下面的例子说明了使用 POP 指令的语法：

```
pop operand
```

这里：

· operand 是 AX、BX、CX、DX、SP、BP、SI、DI、ES、CS、SS、DS 或 Mem16。

下面的程序段说明了汇编语言是如何使用 PUSH 和 POP 指令的：

```
.
.
push  ax     ;Save AX
push  bx     ;Save BX
.
.
mov   ax,33 ;AX = 33
mov   bx,45 ;BX = 45
add   ax,bx ;AX = 33 + 45
.
.
pop   bx     ;Restore BX
pop   ax     ;Restore AX
.
.
```

前面的程序段把 AX，BX 寄存器的值压入堆栈，在进行了一些操作后，再从堆栈中把值弹出。从上面的程序中可以看出，弹出堆栈的顺序和压入堆栈的顺序正好相反。记住，POP 指令总是弹出栈顶单元，因此，最后压入堆栈的数据必然第一个被弹出。

10.3 标志和堆栈

作为对堆栈中压入数据和弹出数据功能的补充，8088 提供了两个特殊指令用于向堆栈中压入标志和弹出标志。这两个指令是 PUSHF 和 POPF。下面的例子说明了使用 PUSHF 指令的语法：

```
pushf
```

下面的例子说明了使用 POPF 指令的语法：

```
popf
```

下面的程序段说明了汇编语言程序是如何使用 PUSHF 和 POPF 指令的：

```
    .
    .
    pushf          ;Save the flags
    add   ax,33 ;Add 33 to AX
    popf           ;Restore the flags to the former
values
    .
    .
```

10.4 小 结

在本章中，我们介绍了计算机如何在存储器中划出一个特定的区域作为堆栈来使用。此外，本章还介绍了把数据和标志压入堆栈和弹出堆栈的方法。

第十一章 过 程

当前，大多数高级语言都使用过程或函数来使程序的代码模块化。进一步讲，过程和函数也为节省存储器提供了方便，因为程序在执行过程中只需调用过程或函数而无需重写代码。本章将讲述：

- 怎样在汇编语言程序中像高级语言一样使用过程和函数来消除冗余的代码
- 怎样使用三种方法向汇编语言的过程传递参数

11.1 一个汇编语言过程

一个汇编语言过程是指完成特定目的的一个代码段。使用 PROC 和 ENDP 可以在汇编语言中创建过程。下面给出了使用 PROC 和 ENDP 定义一个过程的实例。

```
name    proc  type
        instruction
        .
        .
        instruction
name    endp
```

这里：

- name 是过程的名字。
- type 或者是 NEAR，或者是 FAR。若此项忽略，则为 NEAR。若过程是远调用，就必须表明过程为 FAR。注意，当使用的汇编程序只支持简化的段伪指令，而又没有用完整的段定义时，上述假定不适用。
- instruction 是一个汇编语言代码语句。

11.2 连 接

简单地定义一个过程是不能使程序工作的。必须采用一种方法使程序在执行过程中调用过程，执行完过程的代码后再回到调用处往下继续执行。尽管可以使用 JMP 来进入和退出一个过程，但使用它们把程序的各个部分链接在一起的效率很低。分支程序执行到一个过程优先使用的方法是使用 CALL 指令。下面的例子说明了使用 CALL 指令的语法：

```
call  operand
```

这里：

• operand 是一个标号，Reg16 或 Mem16。

调用有两种基本类型：near 和 far。近调用是指定义过程的代码段和调用过程的代码段处于同一代码段。对于一个近调用，它仅需要一个 16 位偏移量。在 8088 微处理器调用过程前，它把下一条要执行的指令地址压入堆栈。下面的例子说明了各种近调用：

```
.
.
call  addint              ;Call a procedure called
                          ; addint
call  near addint         ;Explicitly call addint as
                          ; a near call
call  bx                  ;Call the procedure whose
                          ; address is in BX
call  word_ptr [bx]       ;Call the procedure whose
                          ; address is pointed to by
                          ; BX
.
.
```

远调用是指对驻留在计算机内存中任意位置处过程的调用。因此，远调用既可发生在同一代码段，也可以在不同代码段。注意，任何称做远调用的过程都要在 PROC 语句中定义为 FAR。一个远调用需要段地址和偏移地址。在 8088 微处理器调用远过程前，它把 CS 寄存器和下一条要执行语句的地址压入堆栈。下面的例子说明了远调用的方法：

```
.
.
call  addint              ;Call a procedure named
                          ; addint
call  far addint          ;Explicitly call addint as
                          ; a far call
call  dword ptr [bx]      ;Call the procedure whose
                          ; address is pointed to by
                          ; BX
.
.
```

11.3 从过程返回

从一个调用过程中返回是由 RET 指令完成的。假如调用的是一个近过程，则 RET 指令从堆栈中弹出第一个字存入 IP。假如调用的是一个远过程，则 RET 指令从堆栈中弹出第

一个字存入IP，弹出第二个字存入CS。RET 指令通过过程的 PROC 伪指令中的类型说明判定相关过程是远调用还是近调用。一个 RET 语句在从堆栈中删除返回地址以后，还可以有选择地删除指定数目的字节。在下一节中将会看到，RET 语句的可选功能可以用来经堆栈向过程传递参数。下面的例子说明了 RET 语句的语法：

```
ret   bytes
```

这里：

• bytes 是一个立即数值，它是在弹出返回地址后从堆栈中删除指定的字节数。

下面的程序段演示了一个汇编语言过程，它把存放在 AX 和 BX 中的两个整数相加，然后把返回值放入 AX 中：

```
          .
          .
main      proc
          .
          mov   ax,33 ;Put the first argument in AX
          mov   bx,44 ;Put the second argument in BX
          call  addint;Figure the total
          .
main      endp
          .
          .
addint    proc
          add   ax,bx ;Figure the result
          ret         ;Return
addint    endp
          .
          .
```

11.4 参 数

正如在前面的例子中所看到的，一个过程为了完成指定的任务通常需要一个或更多的参数。过程参数的传递可以使用下述三种相当简单的方法：

• 通过寄存器
• 通过堆栈
• 通过指向寄存器或堆栈中实际参数的指针

前面的例子说明了如何通过寄存器传递参数。这种方式是传递参数最简单和最快速的方法。然而，对于复杂的程序，通过寄存器传递参数并不一定实用。

在过程调用前，把参数放入寄存器然后再将其压入堆栈，可以实现经堆栈传递参数的操作。下面的程序段是前面的 addint 过程的修订版，这里使用堆栈而不是寄存器传递参数：

```
        .
        .
main    proc
        .
        mov   ax,33        ;Put the first argument in AX
        push  ax           ;Put it on the stack
        mov   ax,44        ;Put the second argument in AX
        push  ax           ;Put it on the stack
        call  addint       ;Figure the total
        .
main    endp
        .
        .
addint  proc  near
        push  bp           ;Save BP
        mov   bp,sp        ;Point BP to the bottom of
                           ; the stack
        mov   ax,[bp + 6]  ;AX = First argument
        add   ax,[bp + 4] ;Add the second argument to AX
        pop   bp           ;Restore BP
        ret   4            ;Return and clean up the stack
addint  endp
        .
        .
```

正如从上面的例子程序所看到的，两个参数在过程调用前送入 AX 寄存器，然后再压入堆栈。一旦过程开始执行时，首先把 BP 压入堆栈来保存 BP 的内容。然后，BP 指向当前的堆栈栈顶。因为 BP 的缺省段寄存器是 SS，因此可用 BP 指向堆栈中的任意数据。图 11-1 说明了在这种使用方式下堆栈中数据的分布：

```
[BP +6]        first argument
[BP + 4]       second argument
[BP + 2]       return address
[BP]           BP
```

图 11-1 堆栈帧

正如例子所说明的，过程参数存放在堆栈帧的堆栈上，第一个参数可以通过 [BP+6] 得到，第二个可以通过 [BP+4] 得到。假如是一个远过程调用，那么必须使用 [BP+8] 和 [BP+6]，因为在此种情况下，返回地址将是 32 位而不是近调用使用的 16 位。在返回前，

过程恢复 BP 的数据。更进一步，ret 4 指令在实际返回程序执行以前从堆栈删除参数。尽管 ret4 指令可能是最快和最简单的从堆栈中删除参数的指令，但是，还可以在执行返回后由调用程序删除参数。下面的例子是 addint 程序的翻版，它在调用程序中删除存储在堆栈中的参数：

```
        .
        .
main    proc
        .
        mov     ax,33           ;Put the first argument in AX
        push    ax              ;Put it on the stack
        mov     ax,44           ;Put the second argument in AX
        push    ax              ;Put it on the stack
        call    addint          ;Figure the total
        add     sp,4            ;Remove the arguments from the
                                ; stack
        .
        .
main    endp
        .
        .
addint  proc    near
        push    bp              ;Save BP
        mov     bp,sp           ;Point BP to the bottom of the
                                ; stack
        mov     ax,[bp + 6]     ;AX = First argument
        add     ax,[bp + 4]     ;Add the second argument to AX
        pop     bp              ;Restore BP
        ret                     ;Return and clean up the stack
addint  endp
        .
        .
```

下面的 addint 程序说明了如何使用存储在堆栈中的指针来向过程传递参数：

```
        .
        .
arg1    dw      33
arg2    dw      44
        .
        .
main    proc
        .
        mov     ax,offset arg1      ;AX = Pointer to 1st
```

```
                                    ; argument
        push  ax                    ;Put it on the stack
        mov   ax,offset arg2        ;AX = Pointer to 2nd
                                    ; argument
        push  ax                    ;Put it on the stack
        call  addint                ;Figure the total
main    endp
        .
        .
addint  proc  near
        push  bp                    ;Save BP
        mov   bp,sp                 ;Point BP to the bottom of
                                    ; the stack
        push  bx                    ;Save BX
        mov   bx,[bp + 6]           ;BX = 1st argument pointer
        mov   ax,[bx]               ;AX = 1st argument
        mov   bx,[bp + 4]           ;BX = 2nd argument pointer
        add   ax,[bx]               ;Figure the result
        pop   bx                    ;Restore BX
        pop   bp                    ;Restore BP
        ret   4                     ;Return and clean up the
                                    ; stack
addint  endp
        .
        .
```

11.5 小　结

在本章中，我们介绍了如何像高级语言一样在汇编语言中使用过程和函数。并且介绍了过程的调用，以及怎样从调用的过程中返回。此外，本章还讨论了向汇编语言过程传递参数的三种方法。

第十二章 端 口

8088 微处理器通过一些端口与输入及输出设备通信。这些端口可以直接在累加器或存储器与外设间传递数据。本章将介绍经端口完成数据传输的 8088 汇编语言指令。

12.1 IN 指 令

IN（从端口输入）指令从指定的端口中获取一个字节或字的数据，并且把它置于累加器中。所取得的数据是 8 位还是 16 位由累加器使用 AL 还是 AX 决定。下面的例子说明了 IN 指令的语法：

```
in  accumulator, port
```

这里：

- accumulator 是 AL 或者是 AX。
- port 是从中获取字节或字数据端口的端口号。端口号可以在 DX 或 Imm8 中指定。因为立即数限制在 8 位，因此若端口号大于 255，就必须使用 DX 寄存器。

下面的程序段是使用 IN 指令从一个端口获取数据的汇编语言实例：

```
.
.
in      al,3        ;Fetch a byte from port 3 into AL
mov     dx,0f35h    ;DX = Port number
in      al,dx       ;Fetch a byte from DX's port
inc     dx          ;Bump the port number
int     ax,dx       ;Fetch a word from DX's port
.
.
```

12.2 OUT 指 令

OUT（向端口输出）指令把存储在累加器中的值输出到指定的端口。输出数据的位数由累加器使用 AL 还是 AX 决定。下面的例子定义了使用 OUT 指令的语法：

```
out port, accumulator
```

这里：

- port 是要发送数据的目的端口号。端口号既可由寄存器 DX 来说明也可由 Imm8 说

明。由于立即数限制在 8 位，因此当端口号大于 255 时需使用 DX 寄存器。

· accumulator 是 AL 或者是 AX。

下面是使用 OUT 指令的汇编语言程序段：

```
.
.
out   5,ax          ;Send the value in AX to
                    ; port 5
mov   dx,03ffh      ;DX = Port number
out   dx,ax         ;Send a word out DX's port
dec   dx            ;Decrement the port number
out   dx,al         ;Send a byte out DX's port
.
.
```

12.3 INS、INSB 和 INSW 指令

INS（从端口输入字符串）、INSB（从端口输入字节串）和 INSW（从端口输入字串）指令从端口取得字节或字并把它存储在一个串单元中。串由 ES：DI 所指向，并且存入的值取自 DX 指定的端口。每当指令执行一次，DI 依据方向标志位的值增加或减少。当方向标志位通过 CLD 指令清零时，对应字节串，DI 寄存器加 1，对应字串，DI 寄存器加 2。相反，当方向标志位通过 STD 指令置位时，DI 寄存器对应字节串将减 1，对应字串将减 2。下面的例子定义了 INS 指令的语法：

```
ins   destination
```

这里：

· destination 是字符串的位置。

注：除了指明串地址以外，目的操作数还指出了端口取得的是字节还是字。

下面的例子定义了 INSB 指令的语法：

```
insb
```

下面的例子定义了 INSW 指令的语法：

```
insw
```

下面的程序段使用了汇编语言的 INS 指令：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  db      10 dup (?)
```

```
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA        ;Set
        mov     es,ax           ; ES
        lea     di,buffer       ;ES:DI = Buffer pointer
        mov     dx,0dfeh        ;DX = Port number
        mov     cx,10           ;CX = Loop counter
        cld                     ;Flag increment
11:     ins     buffer          ;Get a byte from the port
                                ; and store it
        loop                    11 ;Loop till done
        .
        .
```

在前面的程序段中，使用 INS 指令从一个端口取得 10 字节串，然后把它存储在正确的缓冲区内。下面的程序段完成同样的任务，但用 INSB 指令替代了 INS 指令：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ; ES
        lea     di,buffer   :ES:DI = Buffer pointer
        mov     dx,0dfeh    ;DX = Port number
        mov     cx,10       ;CX = Loop counter
        cld                 ;Flag increment
11:     insb                ;Get a byte from the port
                            ; and store it
        loop    11          ;Loop till done
        .
        .
```

下面的程序段类似于前面的两个例子，不同的是从指定的端口取得 5 字串。

```
        .
        .
_DATA   segment word public 'DATA'
buffer  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     es,ax       ; ES
        lea     di,buffer   ;ES:DI = Buffer pointer
        mov     dx,0dfeh    ;DX = Port number
        mov     cx,5        ;CX = Loop counter
        cld                 ;Flag increment
l1:     insw                ;Get a word from the port
                            ; and store it
        loop    l1          ;Loop till done
        .
        .
```

12.4 REP 前缀

像许多 8088 字符串指令一样，REP 前缀能简化 INS 指令族的执行。下面的程序段和 INSW 的例子很相似，不同之处是使用 REP 前缀替代了 LOOP 指令：

```
        .
        .
_DATA   segment word public 'DATA'
buffer  dw      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
```

```
mov     ax,_DATA    ;Set
mov     es,ax       ;ES
lea     di,buffer   ;ES:DI = Buffer pointer
mov     dx,0dfeh    ;DX = Port number
mov     cx,5        ;CX = Loop counter
cld                 ;Flag increment
rep     insw        ;Get the string from the
                    ; port
.
.
```

12.5 OUTS、OUTSB 和 OUTSW 指令

OUTS（向端口输出字符串）、OUTSB（向端口输出字节串）和 OUTSW（向端口输出字符串）指令从字符串中向端口发送一个字节或者字。字符串由 DS:SI 指向，端口号由 DX 寄存器指定。每一个指令执行后，SI 根据方向标志位增加或者减少。若方向标志位通过 CLD 指令清零，则 SI 寄存器对字节串每次加 1，对字串每次加 2。相反，若方向标志位通过 STD 指令置位，则 SI 寄存器对应字节串每次减 1，对字串每次减 2。下面的例子定义了使用 OUTS 指令的语法：

```
outs   source
```

这里：

- source 是字符串的位置。

注：除了指明字符串的存储地址以外，source 操作数还指明了每次向指定端口是发送一个字节还是一个字。

下面的例子定义了 OUTSB 指令的语法：

```
outsb
```

下面的例子定义了 OUTSW 指令的语法：

```
outsw
```

下面的例子是在汇编语言中使用 OUTS 指令的实例：

```
        .
        .
_DATA   segment word public 'DATA'
string  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
```

```
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; DS
        lea     si,string   ;DS:SI = String pointer
        mov     dx,0feh     ;DX = Port number
        mov     cx,10       ;CX = Loop count
        cld                 ;Flag increment
11:     outs    string      ;Send a byte to the port
        loop    11          ;Loop till done
        .
        .
```

前面的程序段使用 OUTS 指令向指定的端口发送 10 个字节的字符串。下面的程序完成同样的功能，不同的是使用 OUTSB 指令替代了 OUTS 指令：

```
        .
        .
_DATA   segment word public 'DATA'
string  db      10 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; DS
        lea     si,string   ;DS:SI = String pointer
        mov     dx,0feh     ;DX = Port number
        mov     cx,10       ;CX = Loop count
        cld                 ;Flag increment
11:     outsb               ;Send a byte to the port
        loop    11          :Loop till done
        .
        .
```

下面的程序段和前面两个例子类似，只是它向指定端口发送 5 个字的字符串：

```
        .
        .
_DATA   segment word public 'DATA'
string  db      5 dup (?)
```

```
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; DS
        lea     si,string   ;DS:SI = String pointer
        mov     dx,0feh     ;DX = Port number
        mov     cx,5        ;CX = Loop count
        cld                 ;Flag increment
11:     outsw               ;Send a word to the port
        loop    11          ;Loop till done
        .
        .
```

12.6 REP 前 缀

像 INS 族指令和其它 8088 字符串指令一样，REP 前缀能简化 OUTS 族指令的执行，下面的程序段和 OUTSW 的例子类似，只是用 REP 前缀替代了 LOOP 指令：

```
        .
        .
_DATA   segment word public 'DATA'
string  db      5 dup (?)
_DATA   ends
        .
        .
_TEXT   segment word public 'CODE'
        assume  cs:_TEXT,ds:_DATA
        .
        .
        mov     ax,_DATA    ;Set
        mov     ds,ax       ; DS
        lea     si,string   ;DS:SI = String pointer
        mov     dx,0feh     ;DX = Port number
        mov     cx,5        ;CX = Loop count
        cld                 ;Flag increment
        rep     outsw       ;Send a word to the port
        .
        .
```

12.7 小 结

本章介绍了 8088 微处理器通过端口在计算机及外设间通讯的方法。更进一步，本章广泛讲解了 8088 微处理器通过端口进行数据传输的各种指令。此外，在本章中，读者还学会了通过 8088 端口传输一个字节、一个字、甚至一个字符串的方法。

第十三章 中 断

有时，硬件设备要求CPU对特定的事件做出即时响应。这时，硬件设备产生中断以使8088微处理器挂起正在执行的程序响应中断。本章将讨论：

- 8088微处理器如何处理中断
- 如何使用8088汇编语言处理中断

13.1 8088 中 断

中断使8088挂起正在执行的程序。一旦程序被挂起，8088将把一些关键寄存器的值保存在堆栈中，然后调用预定义的中断处理程序。当从中断返回时，将从堆栈中弹出原来保存的寄存器值，从中断处恢复程序的执行。假如中断激活标志置位，8088微处理器通过调用和中断相联系的中断处理程序处理中断。中断处理程序的地址可以从中断描述表中查到。8088的中断描述表在存储器的0000：0000H处。

每个中断处理程序需要一个32位的远指针指向它所在的存储器区域。合法的的8088中断号的范围是0到255。因此，中断描述表的长度为1024字节（256个中断*4字节）长。

在CPU跳转到中断处理程序地址前，它首先把标志寄存器、代码段（CS）和指令指针（IP）压入堆栈。然后，8088清除陷阱和中断激活允许标志。在8088跳转到中断处理程序后，在遇到IRET指令以前，一直执行中断例程IRET指令，使8088从堆栈中弹出指令指针、代码段和标志寄存器。

除了硬件中断以外，中断也可由汇编语言指令INT产生。INT中断指令将执行指定中断号所对应的中断处理程序。在个人计算机上，软件中断将完成调用函数，如ROM BIOS例程或DOS例程。例如，中断21H完成主要的DOS中断例程。下面的例子定义了INT指令的语法：

```
int   number
```

这里：

- number是被调用的中断号。它必须在0到255之间。

下面的程序段说明了如何使用INT指令：

```
.
.
mov   dx,offset string ;DX = String pointer
mov   ah,09h           ;AH = DOS function call
int   21h              ;Display the string
.
.
```

8088 拥有一个 INTO（溢出中断）指令，用于调用中断处理程序。它和 INT 指令类似，INTO 指令在溢出标志置位时调用中断 04H。若溢出标志没有置位，那么忽略 INTO 指令。下面的例子定义了 INTO 指令的语法：

```
into
```

下面的程序段说明了 INTO 指令的使用：

```
.
.
add   ax,0fff3h   ;Add 0FFF3H to AX
into              ;Generate an interrupt if
            ; the result overflowed
.
.
```

13.2 中断处理程序

中断处理程序很像其他的汇编语言过程。不同的是，中断处理过程必须被定义为远（FAR）过程，必须以 IRET 指令结束而不是 RET 指令结束。下面的例子说明了如何构造中断处理程序：

```
name     proc  far
         .
         .
         .
         iret
name     endp
```

这里：

- name 是中断过程的名字。

下面的程序段演示了一个汇编语言中断处理程序的执行实例：

```
         .
         .
_DATA    segment word public 'DATA'
string   db      "Overflow error", 13, 10, "$"
old4     dd      ?
_DATA    ends
         .
         .
_TEXT    segment word public 'CODE'
         assume  cs:_TEXT,ds:_DATA
main     proc
```

```
        mov     ax,_DATA            ;Set
        mov     ds,ax               ; DS
        mov     ah,35h              ;AH = Get int vector
                                    ; function code
        mov     al,4                ;AL = Interrupt number
        int     21h                 ;Get the current INT 4
                                    ; address
        mov     word ptr old4[2],es ;Save the segment
                                    ;address
        mov     word ptr old4[0],bx ;Save the offset
        push    ds                  ;Save DS
        mov     ax,cs               ;Set DS with
        mov     ds,ax               ; the segment address
        mov     dx,offset overflow  ;DX = Offset address
        mov     ah,25h              ;AH = Set int vector
                                    ; function code
        mov     al,4                ;AL = Interrupt number
        int     21h                 ;Set the new interrupt
                                    ; address
        .
        .
        add     ax,0fff3h           ;Add 0FFF3H to AX
        into                        ;Call INT 4 if overflow
        .
        .
        lds     dx,old4             ;DS:DX = Old INT 4 ad-
                                    ; dress
        mov     ah,25h              ;AH = Set int vector
                                    ; function code
        mov     al,4                ;AL = Interrupt number
        int     21h                 ;Restore the old inter-
                                    ; rupt
        mov     ah,4ch              ;AH = DOS exit program
                                    ; function code
        mov     al,0                ;AL = Return code
        int     21h                 ;Return to DOS
main    endp
        .
        .
overflow proc far
        push ax                     ;Save the
        push dx                     ;AH = Display string
```

```
                                    ; function code
        mov     dx,offset string ;DX = Error message
                                    ; address
        int     21h              ;Display the error mes-
                                    ; sage
        pop     dx               ;Restore
        pop     ax               ; the registers
        iret                     ;Return
overflow endp
        .
        .
        end     main
```

13.3 激活和关闭中断

最后一个不易理解的中断指令是关闭和激活中断。CLI(清除中断标志)指令关闭中断，STI（设置中断标志）指令激活中断。注意，当中断被 CLI 指令关闭后，任何中断仅在执行了一条 STI 指令后发生。可是，由同一设备引起的多个中断就丢失了。如果长时间关闭中断，那么可能对与时间相关的中断产生问题。下面的例子定义了 CLI 指令的语法：

```
cli
```

下面的例子定义了 STI 指令的语法：

```
sti
```

13.4 小　　结

本章介绍了当中断发生时，CPU 是如何临时中断一个程序的执行以及执行与中断相关联的中断处理程序的。更进一步，本章还介绍了如何编制中断处理程序的软调用及中断处理例程是如何执行的。最后，本章介绍了关闭和激活中断指令。

第十四章 条 件 汇 编

像高级语言拥有条件编译伪指令一样，8088 汇编语言有条件汇编伪指令。条件汇编伪指令可以从一个源代码文件中生成多个程序版本。本章介绍如何 8088 条件汇编伪指令的使用：

- IF...ENDIF
- IF...ELSE...ENDIF
- IFDEF...ENDIF
- IFNDEF...ENDIF

14.1 IF...ENDIF 条件伪指令

最简单的 8088 条件汇编指令是 IF...ENDIF 组合伪指令。下面的例子说明了 IF...ENDIF 伪指令的使用方法。例子说明只有当指定的表达式的值为真时，IF 和 ENDIF 之间的汇编语句才会被汇编。若表达式的值为假，IF 和 ENDIF 之间的汇编语句将不会被汇编。

```
if          expression
statement
.
.
statement
endif
```

这里：

- expression 是一个表达式，返回真值或者返回假值。
- statement 是一个汇编语言语句。

下面的程序段说明了 IF...ENDIF 条件语句的使用：

```
.
.
if    version eq 2
mov   ax,2          ;This statement will be
                    ; assembled if
                    ; version is equal to 2
endif
.
.
```

14.2 IF...ELSE...ENDIF 条件伪指令

除了允许使用 IF...ENDIF 条件伪指令以外，8088 汇编语言也允许在其间使用 ELSE 语句。若 IF 的表达式返回真，则在 IF 和 ELSE 之间的语句将被汇编。否则，若 IF 的表达式返回假，则在 ELSE 和 ENDIF 之间的语言将被汇编。下面的例子说明了使用 IF...ELSE...ENDIF 语句的格式：

```
if          expression

statement
.
.
statement
else
statement
.
.
statement
endif
```

这里：

- expression 是表达式，返回真值或者返回假值。
- statement 是汇编语言语句。

下面的例子说明了汇编语言程序中 IF...ELSE...ENDIF 条件伪指令的使用：

```
.
.
if    version eq 2
mov   ax,2          ;This statement will be
                    ; assembled if
                    ; version is equal to 2
else
mov   ax,3          ;This statement will be
                    ; assembled if
                    ; version isn't equal to 2
endif
.
.
```

14.3 IFDEF...ENDIF 条件伪指令

只有当指定的标号、变量或符号被定义后，IFDEF...ENDIF 条件伪指令才汇编一组汇编语言语句。若指定了名字定义，则位于 IFDEF 和 ENDIF 语句之间的汇编语言语句将被汇编。注意，ELSE 语句像在 IF...ENDIF 语句中的情况一样，也可与 IFDEF 和 ENDIF 结合使用。下面的例子说明了使用 IFDEF...ENDIF 的格式：

```
ifdef       name
statement
.
.
statement
endif
```

这里：

- name 是预定义的标号、变量和符号。
- statement 是汇编语言语句。

下面的例子说明了如何在汇编语句程序中使用 IFDEF...ENDIF 条件指令。

```
    .
    .
    ifdef MSC
_addint  proc  far
    endif
    .
    .
    ifdef MSC
_addint  endp
    endif
    .
    .
```

14.4 IFNDEF...ENDIF 条件伪指令

只有当指定的标号、变量或符号不被定义时，IFNDEF...ENDIF 条件伪指令才汇编一组汇编语言语句。若指定的名字未被定义，则位于 IFNDEF 和 ENDIF 之间的所有语句才会被汇编。注意，ELSE 语句像在 IF...ENDIF 和 IFDEF...ENDIF 中的情况一样，也可以与 IFNDEF 和 ENDIF 结合使用。下面的例子演示了使用 IFNDEF...ENDIF 指令的格式：

```
ifndef          name
statement
.
.
statement
endif
```

这里：

- name 是一个未被定义的标号、变量或符号。
- statement 是一个汇编语言指令。

下面的程序段演示了是如何在汇编语言程序中使用 IFNDEF...ENDIF 条件语句的。

```
            .
            .
            ifndef      MSC
addint      proc        far
            endif
            .
            .
            ifndef      MSC
addint      endp
            endif
            .
            .
```

14.5 小　结

在本章中，我们学习了使用条件汇编伪指令控制程序的汇编过程。尽管这些条件汇编伪指令非常简单，但读者可以利用它们把一个源代码文件汇编成不同的程序版本。

第十五章 等 价 与 宏

尽管过程在中等或非常大的代码段中工作正常，但是调用小程序或从小程序返回的负担有时应引起注意。幸运的是，可以使用等价与宏来有效处理整个程序中不断重复的小段代码。本章将讨论以下内容：

- 不可重复定义的数值等价
- 可重复定义的数值等价
- 字符串等价
- 宏
- 局部标号
- 重复块
- 退出宏
- 宏操作

15.1 不可重复定义的数值等价

必须了解的第一种汇编语言等价是不可重复定义的数值等价，该等价允许给一个数字常数赋予一个符号名称。一旦赋予了一个数值，那么符号名称在汇编时就被对应数值所取代。正如其名称含义一样，不可重复定义的数值等价一旦赋予值后就不可再赋新值。下面给出使用 EQU 指令定义一个不可重复定义数值等价的实例：

```
name  equ  expression
```

这里：

- name 是一个符号名。
- expression 是一个合法的的数值表达式。以下一段程序是使用汇编语言的 EQU 指令定义不可重复定义等价的例子。

```
          .
          .
string    db    '123456789'
strlen    equ   9
          .
          .
          mov   bx,offset string ;BX = String pointer
          mov   cx,strlen        ;CX = String length
          call  disp_string      ;Display the string
          .
          .
```

15.2 可重复定义的数值等价

除提供不可重复定义的数值等价外，8088 汇编语言也可定义可重复定义的数值等价。和不可重复定义的数值等价一样，可重复定义的数值等价也是将一个数值赋与一个符号名。然而可重复定义数值等价可以在程序另外部分被赋于一个新值。一旦被赋予一个数值后，符号在汇编时被相应数值取代。以下显示了＝号是如何定义一个可重复定义的等价的：

name = expression

这里：

- name 是符号名。
- expression 是一个合法的的数值表达式。

下面一段程序给出了使用＝指令的汇编语句定义一个可重复定义等价的例子。

```
          .
          .
row       =     1
column    =     1
          .
          .
          mov   al,row       ;AL = Row number
          mov   ah,column    ;AL = Column number
          .
          .
row       =     3
column    =     45
          mov   al,row       ;AL = Row number
          mov   ah,column    ;AL = Column number
          .
          .
```

15.3 字 符 串 等 价

可以使用 8088 汇编语言中的字符串等价语句给一个符号名赋予一个字符常量。字符串等价有时也称为文本宏。一旦赋予了一个字符串值，当汇编程序在碰到该符号名时使用所赋的值替代它。和不可重复定义的数值等价一样，字符串等价使用 EQU 指令。但字符串等价是可重复定义的，如同可重复定义的数值等价一样。以下程序给出了使用 EQU 语句定义一个字符串等价的语法：

```
name  equ  <expression>
```

这里：

- name 是符号名。
- expression 是合法的的字符串表达式。

注意，<and>符号是可选择的，应该经常使用它们以防止汇编程序混淆字符串常量和已定义的数值常量。

以下是一段使用 EQU 语句的汇编语言程序，程序给出了一个字符串等价的实例。

```
        .
        .
row     equ   <[bp + 6]>
column  equ   <[bp + 8]>
        .
        .
        mov   ax,row      ;AX = Row value
        mov   bx,column   ;BX = Column value
        .
        .
```

15.4 宏

尽管等价是非常有用的编程工具，但是汇编语言中的宏在实现程序中不断重复小段代码时更为有力。使用 MACRO 与 ENDM 伪指令来创建宏。下面的例子给出了使用 MACRO 与 ENDM 定义宏的格式：

```
name    macro      parameter, parameter, etc.
        statements
        .
        .
        statements
        endm
```

这里：

- name 是宏的名字。
- parameters 是一个可替代参数。
- statements 是一个合法的的汇编语句。

以下例子是一个宏定义：

```
addints  macro int1,int2
         mov   ax,int1
         add   ax,int2
         endm
```

在上面的宏例子中，只是简单地将第一个参数放入 AX 中并且和第二个参数值相加。以下例子给出了在汇编程序中使用宏的语法。

```
name  argument, argument, etc。
```

这里：

- name 是已定义宏的名字。
- arguments 是一个或多个用来替代宏参数的变元。

注：如果传递的参数多于宏定义的参数，则多余的参数被忽略。如果传递的参数少于宏定义的参数，则以空字符串替代没有对应变元的参数。

以下的宏调用使用了前面已定义的 addints 宏定义：

```
addints  3, 4
```

当汇编程序碰到该宏调用时，用以下语句超越它：

```
mov   ax, 3
add   ax, 4
```

15.5 局 部 标 号

因为许多例程使用标号使执行转入新的路径，所以最好在宏内部使用标号。然而，在一个程序中可多次使用宏，这就很容易产生标号重复定义的错误。所以，宏应具有某种机制以允许标号重复定义。可以在宏内部使用局部标号来重复定义一个标号。一旦定义后，汇编程序将程序中宏的每一个事例中的局部标号替换成一个唯一的名字。以下程序定义了使用 LOCAL 语句创建局部标号的语法。

```
local  name, name, name
```

这里：

- name 是众多局部标号名中的一个。

以下例子是定义局部标号的汇编程序：

```
movstring macro  src_seg,src_off,dest_seg,dest_off,len
          local  ll
          push   ax
          push   cx
```

```
        push    di
        push    si
        push    ds
        push    es
        mov     ax,src_seg
        mov     ds,ax
        mov     si,src_off
        mov     ax,dest_seg
        mov     es,ax
        mov     di,dest_off
        mov     cx,len
        cld
l1:     movsb
        loop    l1
        pop     es
        pop     ds
        pop     si
        pop     di
        pop     cx
        pop     ax
        endm
```

15.6 重 复 块

8088 汇编语言还支持一种特殊形式的宏，称做重复块。与正常的汇编语言宏不同，一个重复块每次在程序中使用时都必须定义。8088 汇编语言中的三种不同形式的重复块分别以 REPT、IRP、IRPC 指令定义。

15.6.1 REPT 重复块

REPT 语句创建一个按指定次数重复的重复块。以下例子给出了定义 REPT 重复块的格式：

```
rept        expression
statement
.
.
statement
endm
```

这里：

- expression 是一个数值表达式，用来定义汇编程序重复该块或语句的次数。
- statement 是一个合法的汇编语句。

以下一段程序是 REPT 重复块的例子：

```
        .
        .
    n   =     0
        rept  10
        db    n
    n   =     n + 1
        endm
        .
        .
```

以上程序给出了使用 REPT 重复块创建 0～9 数值的 10 个 DB 语句，以下是汇编程序依此创建的汇编语句：

```
        .
        .
    n   =     0
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        db    n
    n   =     n + 1
        .
        .
```

15.6.2 IRP 重复块

IRP 指令建立的重复块对尖括号(<>)内指定的每个变元都重复一次。以下例子给出了定义 IRP 重复块的格式。如例子中所示，一个 IRP 重复块需要一个参数，该参数在重复块语句中依次被变元表中指定的值取代。

```
irp         parameter,<argument,argument>
statement
.
.
statement
endm
```

这里：

- parameter 是将被变元取代的参数。
- argument 是汇编程序用来替代重复块参数的变量。
- statement 是一个合法的语句。

以下程序段是一个 IRP 重复块的例子：

```
.
.
irp   n,<0,1,2,3,4,5,6,7,8,9>
db    n
endm
.
.
```

正如所看到的，以上重复块完成了与前面 REPT 重复块例子相似的功能。但是汇编程序解释该 IRP 重复块与解释 REPT 重复块稍有不同。下例说明了汇编程序是如何解释上述 IRP 块的。

```
.
.
db    0
db    1
db    2
db    3
db    4
db    5
db    6
db    7
```

```
db    8
db    9
.
.
```

15.6.3 IRPC 重复块

IRPC 指令建立一个重复字符串变量中每一个字符的重复块。以下例子给出了定义一个 IRPC 重复块的格式。如例中所示，一个 IRPC 重复块和 IRP 重复块一样需要一个参数。IRPC 的参数在重复块语句中被字符串变量中的字符取代。

```
irpc  parameter,argument
statement
.
.
statement
endm
```

这里：

- parameter 是变量字符将要取代的参数。
- argument 是一个合法的字符串。
- statement 是一个合法的汇编语句。

以下程序是 IRPC 重复块的一个例子：

```
.
.
irpc  n,0123456789
db    n
endm
.
.
```

以上程序中的 IRPC 重复块和前述 IRP 重复块例子的功能相似。下面语句给出了汇编程序是如何解释该 IPRC 块的。

```
.
.
db    0
db    1
db    2
db    3
db    4
```

```
db    5
db    6
db    7
db    8
db    9
.
.
```

15.7 退 出 宏

在某些情况下，需要在宏未执行完时从宏退出。8088 汇编语言提供了 EXITM 伪指令完成从宏代码中退出 。重要的是，EXITM 伪指令指示汇编程序对宏不再进行汇编。以下例子给了出在宏中使用 EXITM 以保证一个重复块不会死循环的实例。

```
makebytes     macro n
cnt           =     0
              rept  n
              if    cnt gt 255
              exitm
              endif
              db    cnt
cnt           =     cnt + 1
              endm
              endm
```

上面的宏也是一个很好的宏嵌套例子——一个宏定义在另一个宏内。注意，上例的 EXITM 语句只是终止进一步汇编 REPT 重复块而不是 makebytes 宏。尽管如此，由于 makebytes 只是实现 REPT 重复块的简单外壳，因此，EXITM 语句实际上终止了对这两个宏的汇编。

15.8 & 操 作 符

& 操作符可以对二义性的参数引用进行强制参数替代。通过给一个宏参数加上 & 操作符前缀，汇编程序不管它出现在宏定义的任何地方都将替代该参数的相应变元。尽管 & 操作符不是必须的，但加在其它字符之前或之后，或者参数在加引号的字符串之中时，使汇编程序仍可进行参数替代。以下例子定义了使用 & 操作符进行替代的语法：

```
&parameter
```

这里：

- parameter 是已定义的参数。

下面一段程序给出了在宏定义和宏调用中使用 & 操作符的实例：

```
          .
          .
message   macro       n
msg&n     db          'This is message no. &n'
          endm
          .
          .
          message     2
          .
          .
```

因为 & 操作符强制完成参数替代，上面的 message 2 宏调用将被汇编成：

```
msg5   db    'This is message no. 5'
```

15.9 <> 操 作 符

<>操作符指示汇编程序将其括起的文本看作文本字符串。该操作符按宏变元传送像 1、2、3、4 这样的字符串时非常方便。当使用<>操作符将它们像<1、2、3、4>这样括起来后，汇编程序将它们作为一个变元而不是 4 个单独变元来对待。以下例子定义了使用<>操作符的语法：

```
<text>
```

这里：

• text 是一个文本字符串。

以下程序段说明了在宏调用中使用<>操作符将一个多义的文本字符串作为单个变元处理的用法：

```
          .
          .
message   macro       string
          db          string,0
          endm
          .
          .
          message     <1,2,3,4>
          .
          .
```

因为< >操作符强制汇编程序将变元作为单个字符串处理，所以上面 message<1，2，3，4>宏调用被汇编成：

```
db  1, 2, 3, 4, 0
```

15.10 ! 操 作 符

!操作符指示汇编程序将下一个字符作为文本字符而不是一个符号处理。!操作符在澄清一个多义性字符是如何汇编时非常有用。以下例子定义了使用! 操作符的语法：

```
! character
```

这里：

• character 是一个文本字符。

以下程序段给出了在宏调用中! 操作符的用法以说明如何汇编一个多义性字符：

```
        .
        .
message macro       string
        db          '&string',0
        endm
        .
        .
        message     <76 !> 75>
        .
        .
```

因为!操作符强制汇编程序将随后的字符作为文本字符处理，所以上面的 message< 76! > 75>宏调用汇编结果如下：

```
db   '76>75', 0
```

15.11 % 操 作 符

%操作符告诉汇编程序按一个表达式来求变元的值，并以所得结果替代相应参数。%操作符使宏调用传递一个符号名的值而不是符号名本身。以下程序定义了%操作符的语法：

```
%text
```

这里：

• text 是求值符号。

以下程序段说明了如何在宏调用中使用%操作符按一个表达式而不是一个符号求一个

变元的值。

```
          .
          .
message   macro       name,exp
          db          '&name = &exp',0
          endm
          .
          .
msg       equ         <Hi>
          .
          .
          message     msg,%msg
          .
          .
```

因为%操作符强制汇编程序将一个变元作为表达式对待，所以上面的 message msg,%msg 宏调用汇编成如下形式：

```
db 'msg=Hi', 0
```

15.12 宏 注 释

尽管可以在宏定义中使用；注释语句，但是，汇编程序在大多数情况下将它们包含在每次宏调用中。使用;; 来替代；可以在宏定义中使用注释语句，并且在宏调用中并不包括该注释语句。以下例子定义了使用宏注释的语法：

```
;; text
```

这里：

- text 是将作为注释的文本。

下面程序段给出了一个宏定义是如何使用宏注释的：

```
movstring  macro src_seg,src_off,dest_seg,dest_off,len
           local l1
           push  ax          ;;Save
           push  cx          ;; the
           push  di          ;; registers
           push  si
           push  ds
           push  es
           mov   ax,src_seg  ;;Set
```

```
        mov   ds,ax        ;; DS
        mov   si,src_off   ;;DS:SI = Source pointer
        mov   ax,dest_seg  ;;Set
        mov   es,ax        ;; ES
        mov   di,dest_off  ;;ES:DI = Destination pointer
        mov   cx,len       ;;CX = Number of byte to move
        cld                ;;Flag increment
l1:     movsb              ;;Move a byte
        loop  l1           ;;Loop till done
        pop   es           ;;Restore
        pop   ds           ;; the
        pop   si           ;; registers
        pop   di
        pop   cx
        pop   ax
        endm
```

15.13 小 结

在本章中，读者了解到了使用等价与宏可以非常简单地实现在程序中重复出现的汇编语言小程序。本章还介绍了三种用来创建汇编语句重复块的特殊宏。最后，本章介绍了一些宏操作符，并且解释了如何编写汇编注释。

第十六章 汇编语言与C和C++的接口

现在，应用程序很少完全使用汇编语言编写。实际上，大多数程序员选择的语言是C语言和C++语言。尽管当今的C和C++编译器可以产生非常高效的代码，但有时仍需使用汇编例程来改进代码质量。本章将介绍汇编语言例程与高级C语言程序相结合的方法。其内容分为如下几个方面：

- 函数和变量名
- 参数传递
- 返回调用程序
- 局部变量空间

16.1 函数和变量名

选择C（在本章以后部分中，.所有针对C的描述同样适用于C++）函数或变量名是一项非常简单、直接的工作。例如，一个C函数用来查找一个整数数组的最大值，并返回查找的结果。该函数可被命名为search _array。假设search _array函数名也可用做一个有相似功能的汇编语言过程名，这也是合乎逻辑的。尽管少数几个C编译器认为search _array作为函数名是合法的的，但大多数C编译器不把它当做一个合法的函数名。

最常用的命名规范要求所有函数及变量名以一个_（下划线）开头。一些C编译器要求函数和变量名以_结束，该下划线是当C编译器编译程序时加到函数和变量名上的。因此，依赖于C编译器，一个汇编过程search _array可以被命名为search _array，_search _array甚至search _array _。尽管这么多不同的命名规范看起来非常不方便，但是可以通过条件汇编伪指令，使汇编语言的例程名可以被不同种类的C编译器有效处理。

除了遵守C编译器的命名规范外，在C程序调用汇编函数或引用变量以前，这些函数和变量名必须设成全局的。因此，所有全局汇编语言的函数名以及变量名都被说明为公共的（public）。通过使用public汇编语言伪指令，连接程序可以正确地将汇编语言过程和变量与任何使用它们的C程序连接。下面的例子定义了使用public伪指令说明全局函数和变量名的语法：

```
public  name，name，etc
```

上式中：

- name是一个过程或变量名。

除了C程序需要访问一个汇编语言过程或变量外，反之亦然。当然，汇编程序通过使用EXTRN伪指令将C的函数和变量名说明为外部名称而确知它们的存在。下例定义了了EXTRN指令的语法。

```
extrn   name, name, etc.
```

上式中：

- name 是外部 C 函数和变量名。

16.2 参 数 传 递

C 编译器将参数传给汇编过程的最常用方法是将它们送入堆栈桢中的堆栈上。注意，越来越多的 C 编译器能够通过 CPU 寄存器向每个汇编过程传递参数。通过 CPU 寄存器传递参数是完成参数传递的最快方法。尽管如此，通过堆栈帧传递参数是各种编译器的通用标准。因为兼容性的原因，大多数情况下都使用堆栈传递参数。特殊情况下，当速度是压倒一切的需求时，通过寄存器传递参数就必要了。

在汇编语言过程的入口处，堆栈桢由一个返回地址（近调用是双字节，远调用是四字节），以及从第一至最后一个参数构成。以下例子显示了 search _ array 程序的堆栈桢。

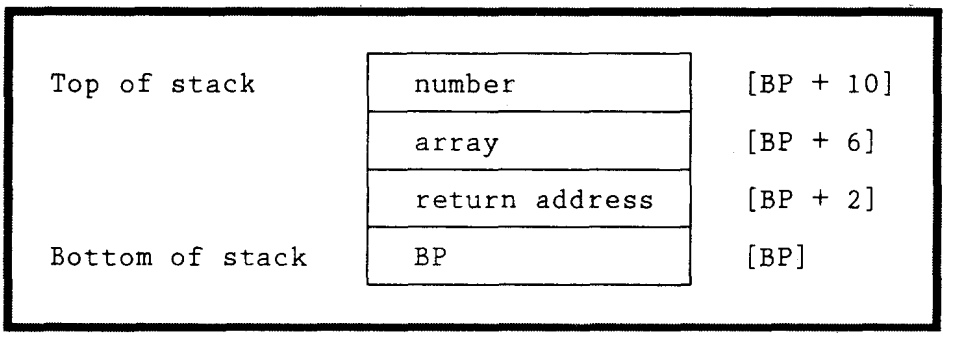


图 16-1　一个 search _ array 堆栈帧

该堆栈桢假设 search _ array 使用 int far search _ array 的一个 C 函数原型（int far * array, int number）；因为 search _ array 说明为远函数，所以 C 编译器将四字节的返回地址放入堆栈底端。另外，将数组说明为一个远指针意味着 C 编译器将一个 32 位远指针放入堆栈，而不是放入 16 位偏移量。当数组说明为一个近指针时，堆栈中存放 16 位偏移量。尽管在小模式下将函数和指针说明为远的形式并不是必需的，但大多数程序员在汇编过程中都使用远指针，这可以避免为每种内存模式创建不同版本的汇编例程。

通过在 C 头文件中将所有东西都说明为 far，C 编译器就可确知如何正确调用汇编过程。不幸的是，并非所有 C 编译器都支持混和内存模式编程。因此，依据所用的 C 编译器为每个内存模式编写一个独有版本的汇编例程是必要的。

一旦程序执行从 C 程序转入汇编过程，汇编过程必须将 BP 寄存器指向堆栈底部以引用传来的参数。以下一小段程序显示了前面提及的 search _ array 程序是如何完成这个任务的。

```
array     equ     < [bp+6] >
```

```
number  equ     < [bp+10] >
        push    bp              ; Set BP
        mov     bp, sp          ; Point it to the stack
```

BP 指向堆栈帧底部后，指向数组的远指针可以使用偏移量 [bp+6] 引用。此外，被查找的单元个数可以通过偏移量 [bp+10] 来引用。注意，这些偏移量通过文本等价命名。通过命名偏移量，可以大大减小发生错误的可能性。因为记住名字 array 比记住 [bp+6] 更容易。借助 BP 偏移量来访问参数，可以像如下所示的实例继续编写汇编例程。

```
        push  ds            ;Save DS
        lds   si,array      ;DS:SI = Array pointer
        mov   cx,number     ;CX = Number of elements
        mov   ax,0          ;AX = Starting value
l1:     jcxz  l3            ;Jump if done
        cmp   ax,[si]       ;Jump if AX
        jge   l2            ; is >= this element
        mov   ax,[si]       ;Put it in AX
l2:     inc   si            ;Bump the
        inc   si            ; pointer
        dec   cx            ;Decrement the count
        jmp   l1            ;Loop till done
l3:     pop   ds            ;Restore
```

16.3 返回调用程序

在汇编过程完成了其功能后，必须将结果返回调用它的 C 程序。大多数 C 编译器通过将返回值放入一个 CPU 寄存器或一个 CPU 寄存器组中将该值返回给调用程序。整数基本上总是放在 AX 寄存器返回，即然 search _array 过程的结果总是存于 AX 寄存器中，因此不必再另外采取措施将该值传回调用它的 C 程序了。但是，若结果放在另外一个寄存器中或某个存储器单元，那么在执行返回到调用程序之前，必须将它移至 AX 寄存器中。

除了准备返回值外，汇编过程在返回到调用 C 程序前还必须清空堆栈。因为寄存器 BP 的值被压入了堆栈，所以必须使用一条 pop bp 指令取回它。BP 寄存器的值从堆栈中取回后，堆栈就恢复到了其入口条件下。因此，汇编过程通过执行一条 ret 指令返回到调用它的 C 程序。C 程序在开始其下一个任务之前将传递至堆栈中的参数逐个删除。下面的例子说明了 search _array 过程余下的代码。

```
pop bp          ; the registers
ret             ; Return
```

下面是 search _array 汇编过程完整的清单。

```
        ;
        ; search.asm - Assembly language search integer array
        routine
        ;
        SEARCH_TEXT segment     para public 'CODE'
                    assume      cs:SEARCH_TEXT
                    public      _search_array
        ;
        ; Search array for highest element
        ;
        _search_array           proc  far
        array       equ         <[bp + 6]>
        number      equ         <[bp + 10]>
                    push        bp          ;Set BP
                    mov         bp,sp       ;Point it to the stack
                    push        ds          ;Save DS
                    lds         si,array    ;DS:SI = Array pointer
                    mov         cx,number   ;CX = Number of
                                            ; elements
                    mov         ax,0        ;AX = Starting value
    11:             jcxz        13          ;Jump if done
                    cmp         ax,[si]     ;Jump if AX
                    jge         12          ; is >= this element
                    mov         ax,[si]     ;Put it in AX
    12:             inc         si          ;Bump the
                    inc         si          ; pointer
                    dec         cx          ;Decrement the count
                    jmp         11          ;Loop till done
    13:             pop         ds          ;Restore
                    pop         bp          ; the registers
                    ret                     ;Return
        _search_array   endp
        SEARCH_TEXT     ends
                        end
```

以下的一小段C程序说明了在一个实际的程序中是如何使用search _array汇编过程的。其中最值得注意的是，search _array汇编过程被C过程调用之前是如何定义其函数原型的。通过在函数原型中定义search _array，该C程序可以正确调用汇编过程。

```
/*
   asmdemo.c - Assembly language interfacing demo
```

```
*/
#include <stdio.h>
#include <stdlib.h>
int far search_array(int far *array, int number);
int test_array[10] = { 2, 3, 55, 66, -2, 3, 4, 5, 9, 34 };
void main(void)
{
   int n;
   n = search_array(test_array, 10);
   printf("The highest element in test_array is %d.\n",n);
   exit(0);
}
```

16.4 局部变量空间

尽管上面的 search _array 过程没有为其局部变量要求堆栈空间，但是，许多汇编过程需要这样做。局部变量空间的分配是通过从堆栈指针中减去所需的字节数来完成的。假设有一个汇编过程需要为 2 个整数 row、col 分配局部变量空间，以下汇编过程可以分配其所需的空间。

```
.
.
push  bp     ;Save BP
mov   bp,sp  ;Point it to the stack
sub   sp,4   ;Adjust stack for local variables
.
.
```

局部变量空间分配后，局部变量可以通过寄存器 BP 的负偏移来引用。因而 row 和 col 可以通过 [bp－2] 和 [bp－4] 引用。具体选择哪一个位置来存放变量并不重要，但位置一旦确定，则必须保持它为常量。

因为堆栈指针在分配局部变量空间后已被移动了，所以汇编过程必须在恢复 BP 寄存器之前回收局部变量空间。回收局部变量空间是通过 mov sp，bp 指令来完成的。在分配局部变量空间之前，寄存器 BP 和 SP 是指向同一个存储器位置的，因此将 BP 寄存器的值赋与 SP 寄存器实际上从堆栈中消除了局部变量空间。下面一段程序显示了汇编过程在返回调用程序前是如何回收它的局部存储器空间的。

```
.
.
mov   sp,bp ;Restore the stack pointer
```

```
pop   bp     ;Restore BP
ret          ;Return to the calling program
.
.
```

不需要分配局部存储器空间的汇编过程在堆栈桢指针被 mov bp，sp 指令设置后需要保存一些必要的寄存器。在汇编过程退出时，必须在 BP 寄存器复原前恢复这些保存的寄存器。注意，search _array 汇编过程正确地保存和恢复了 DS 寄存器。需要局部变量存储器空间的汇编过程则应在分配了局部变量存储器空间后保存所需的寄存器。同样，所有保存的寄存器必须在汇编过程回收局部变量空间之前恢复所保存的寄存器。如果先回收存储器，那么寄存器的内容就会丢失，导致程序执行出错。

使用汇编过程最后需要加以注意的是，大多数 C 编译器规定某些 CPU 寄存器不可被汇编过程修改。因此，汇编过程中使用的不可修改内容的寄存器在汇编过程开始时必须存入堆栈中，在汇编过程返回调用程序前必须恢复。

16.5 小　结

在本章中，读者学会了连接汇编和 C 程序的许多重要技术。如何为汇编过程选择正确名称是很重要的。另外，本章进述了参数传递、返回值给调用程序以及局部变量空间等问题。最后，本章讲解了保存和恢复某些不可修改 CPU 寄存器的重要性。

第十七章 汇编语言与 Pascal 的接口

尽管 C 和 C++是最专业化的软件开发语言，但 Turbo Pascal 和 Borland Pascal（在这里均称之为 Turbo Pascal）也是许多编程爱好者使用的处于第二位的编程语言。Turbo Pascal 可以产生和汇编语言相媲美的小 EXE 文件。然而，在执行速度方面，Turbo Pascal 程序既没有具有很好的优化能力的 C/C++编译器产生的代码速度快，也没有手工编制的汇编语言程序的执行速度快。因此，在对时间要求严格的情况下，在 Turbo Pascal 程序中加入一些使用汇编语言编写的过程是一条很有效的途径。本章讲述把汇编语言和高级 Pascal 程序结合使用的方法。其内容包括：

- 函数名和变量名
- 参数传递
- 返回调用程序
- 局部变量空间

17.1 函数名和变量名

选择一个 Pascal 函数和变量名是一项简单明了的工作。例如，扫描一个整型数组并返回其最大值的函数可以被命名为 SearchArray。不像其它高级语言那样需要修改函数和变量名，在 pascal 语言中可以直接使用 SearchArray 作为汇编语言的名字。

除了要考虑编译器的命名规范外，Pascal 程序在调用汇编函数或引用其变量时，函数和变量名必须是全局的。因此，所有全局的汇编语言函数和变量名必须说明为公共的。通过使用公共的汇编语言伪指令，连接程序能正确地把汇编语言过程和变量同调用它的 Pascal 程序连接在一起。下面的例子说明了使用 public 语句把过程或变量说明为全局的语法：

```
public  name，name，etc
```

这里：

- name 是一个过程或变量名。

除了 Pascal 程序可以调用汇编语言的过程或变量以外，反之亦然。在这种情况下，汇编语言调用的任何 Pascal 函数名或引用的任何变量名必须使用 extrn 伪指令说明为外部名称。下面的例子定义了 extrn 伪指令的语法：

```
extrn  name，name，etc.
```

这里：

- name 是外部的 Pascal 函数或变量名。

17.2 参 数 传 递

Turbo Pascal 通过堆栈帧中的堆栈向汇编语言程序传递参数。在汇编语言过程入口处，堆栈帧由一个返回地址（Turbo Pascal 调用汇编语言过程是长调用，因此使用 4 字节）以及后继的从最后一个至第一个的参数构成。请注意，Pascal 在堆栈中存放参数的顺序是反向的，这是其独有的方法。下面的例子说明了 SearchArray 过程的堆栈帧：

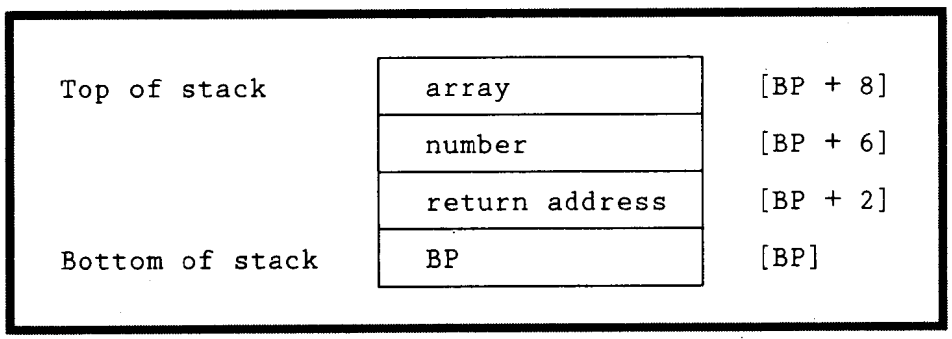


图 17-1　一个 SearchArray 堆栈帧

该堆栈帧假定了 SearchArray 使用了下面的 Pascal 函数原型：

```
type
   IntArray = array[1..10] of Integer;
   IntArrayPtr = ^IntArray;
function SearchArray(iarray : IntArrayPtr; number
: Integer) :
   Integer ; far; External;
```

因为 SearchArray 说明为 far，Turbo Pascal 在堆栈的底部留下 4 字节作为返回地址。因为 Turbo Pascal 首先把第一个参数压栈，所以它在栈顶放置一个 32 位远指针并且该指针指向整数数组的第一个元素。在返回地址和数组指针之间，Turbo Pascal 在堆栈上放置了表示数组单元个数的 16 位整型值。

一旦 Pascal 程序执行到汇编语言过程，汇编语言必须设置 BP 指针指向堆栈底部来引用传递过来的参数。下面的程序段演示了 SearchArray 过程是如何完成上述任务的。

```
array    equ   <8[bp]>
number   equ   <6[bp]>
         push  bp          ;Set BP
         mov   bp,sp       ;Point it to the stack
               .
```

当 BP 指针指向堆栈底部时，数组指针通过偏移量 8 [BP] 进行引用，要搜寻的数组单元个数由偏移量 6 [BP] 引用。请注意在程序中是如何使用文本等价给编移量赋名的。通过

命名偏移量，可以减少错误发生的可能性。毕竟，记住 array 名比记住 8[BP] 要容易得多。通过 BP 的偏移量访问参数，可以按下列方法继续编码汇编语言例程：

```
        push    ds          ;Save DS
        lds     si,array    ;DS:SI = Array pointer
        mov     cx,number   ;CX = Number of
                            ; elements
        mov     ax,0        ;AX = Starting value
11:     jcxz    13          ;Jump if done
        cmp     ax,[si]     ;Jump if AX
        jge     12          ; is >= this element
        mov     ax,[si]     ;Put it in AX
12:     inc     si          ;Bump the
        inc     si          ; pointer
        dec     cx          ;Decrement the count
        jmp     11          ;Loop till done
13:     pop     ds          ;Restore
```

17.3 返回调用程序

现在，汇编语言过程已经完成了其功能，它必须连同结果返回调用它的 Turbo Pascal 程序。对于 Turbo Pascal 来讲，调用程序的值位于一个 CPU 寄存器或一个 CPU 寄存器组中，整型值置于 AX 寄存器中。

因为 SearchArray 过程的返回值已经被放入 AX 寄存器中了，所以无需进一步向 Turbo Pascal 程序返回值。假如，若返回的结果置于别的寄存器或某个存储单元中，它在返回调用程序之前必须将其移入 AX 寄存器。

在汇编语言过程返回前，除了要准备返回值，它还必须把堆栈清空。因为 BP 指针被压入堆栈中，所以，必须使用 pop bp 指令将其弹出。在从堆栈中弹出 BP 寄存器后，堆栈恢复到了它的初始条件。因而，汇编语言过程通过执行 ret 指令返回调用它的 Turbo Pascal 程序。

假如，SearchArray 函数的参数通过堆栈来传递，汇编语言过程必须指定参数在堆栈上占据的字节数作为 ret 指令的操作数。在 SearchArray 函数中，必须通过 ret6 指令返回调用它的 Turbo Pascal 程序并且从堆栈中删除堆栈帧。下面的例子说明了 SearchArray 剩余过程的代码。

```
pop bp      ; the registers
ret 6       ; Return and clean up the stack
```

下面的例子列出了 SearchArray 汇编语言过程的全部代码清单：

```
;
; psearch.asm - Assembly language search integer
array routine
;
DATA        segment word public
DATA        ends
CODE        segment byte public
            assume  cs:CODE, ds:DATA
            public  SearchArray
;
; Search array for highest element
;
SearchArray proc    far
array       equ     <8[bp]>
number      equ     <6[bp]>
            push    bp          ;Set BP
            mov     bp,sp       ;Point it to the stack
            push    ds          ;Save DS
            lds     si,array    ;DS:SI = Array pointer
            mov     cx,number   ;CX = Number of
                                ; elements
            mov     ax,0        ;AX = Starting value
11:         jcxz    13          ;Jump if done
            cmp     ax,[si]     ;Jump if AX
            jge     12          ; is >= this element
            mov     ax,[si]     ;Put it in AX
12:         inc     si          ;Bump the
            inc     si          ; pointer
            dec     cx          ;Decrement the count
            jmp     11          ;Loop till done
13:         pop     ds          ;Restore
            pop     bp          ; the registers
            ret     6           ;Return and clean up
the stack
SearchArray endp
CODE        ends
            end
```

下面一小段 Turbo Pascal 程序演示了汇编语言过程 SearchArray 的使用。在此程序中，需要注意两点。为了使汇编语言过程和汇编语言过程连接在一起，Turbo Pascal 程序必须使 $L 编译指令指定汇编语言过程的目标模块名称。另外，在 Turbo Pascal 程序中，必须说明汇编语言过程的函数原型，以便 Turbo Pascal 程序能正确地调用它。

```
{
   asmdemo.pas - Assembly language interfacing demo
}
{$L PSEARCH.OBJ}
type
   IntArray = array[1..10] of Integer;
   IntArrayPtr = ^IntArray;
function SearchArray(iarray : IntArrayPtr; number
: Integer) :
   Integer ; far; External;
const
   testarray : IntArray = ( 2, 3, 55, 66, -2, 3, 4,
5, 9, 34 );
var
   n : Integer;
begin
   n := SearchArray(@testarray, 10);
  Writeln('The highest element in testarray is ', n);
end.
```

17.4 局部变量空间

尽管 SearchArray 过程不需要为局部变量保留堆栈空间，但是许多其它的汇编语言过程将需要这样做。局部变量空间通过把堆栈指针减去所需的字节数来得到。假设汇编语言过程需要两个整数 row 和 col 的局部变量空间。下面的汇编语言代码能够分配所需的空间。

```
push  bp     ;Save BP
mov   bp,sp ;Point it to the stack
sub   sp,4   ;Adjust stack for local variables
```

在局部变量空间分配后，局部变量可以通过 BP 指针的负偏移量来引用。因此，row 和 col 能够通过－2 [bp] 和－4 [bp] 引用。事实上，局部变量的位置在何处是无关紧要的，关键是一旦为其分配了存储器，它必须保持不变。

因为在分配了局部变量空间后，堆栈指针发生了变化，因此，在恢复 BP 寄存器前，汇编语言函数必须释放所分配的局部变量空间。释放所分配的局部变量空间可以通过 mov sp，bp 指令来完成。记住，在分配局部变量空间前，SP 和 BP 指向同一存储单元。因此，把 BP 寄存器的值赋给 SP 寄存器将从堆栈中清除局部变量空间。下面的程序段说明了汇编语言过程在返回它的调用程序前是如何释放所分配的局部变量空间的。

```
mov       sp, bp   ; Restore the stack pointer
pop       bp       ; Restore BP
```

```
ret                    ; Return to the calling program
```

不需要局部变量空间分配的汇编语言过程在使用 mov bp，sp 指令设置堆栈指针之后，应保存必要的寄存器。在汇编语言过程退出期间，在恢复 BP 指针之前，应恢复保存的寄存器。请注意，SearchArray 过程正确地保存和恢复了 DS 寄存器。汇编语言过程若需要为局部变量分配空间，在分配前就不必保存所需寄存器的值，而在分配了局部变量空间后，就需要保存空间。相应地，在汇编语言过程释放了所分配的存储器空间后，就应相应地恢复寄存器的值。假若首先释放了局部变量空间，寄存器中的内容将丢失并且导致程序执行错误。

对汇编语言过程来说，还有一点需要注意，Turbo Pascal 需要汇编语言过程保存 BP、SP、SS 和 DS 寄存器的值。因此，任何在汇编语言过程中需要用到的、不可修改的寄存器都需先在堆栈中保存起来，在汇编语言过程执行完后，再从堆栈中恢复。

17.5 小　　结

本章介绍了汇编语言过程与 Turbo Pascal 程序接口的许多重要技术。我们学习了如下内容：参数传递，向调用程序返回值和局部变量空间。最后，本章解释了保存和恢复某个不修改的 CPU 寄存器的重要性。

附录 A ASCII 代码表

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	[sp]	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	—	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

[General Information]
书名=汇编语言程序设计自学教程
作者=（美）Mark Goodwin
页数=176
SS号=10203529
出版日期=1995年1月第1版