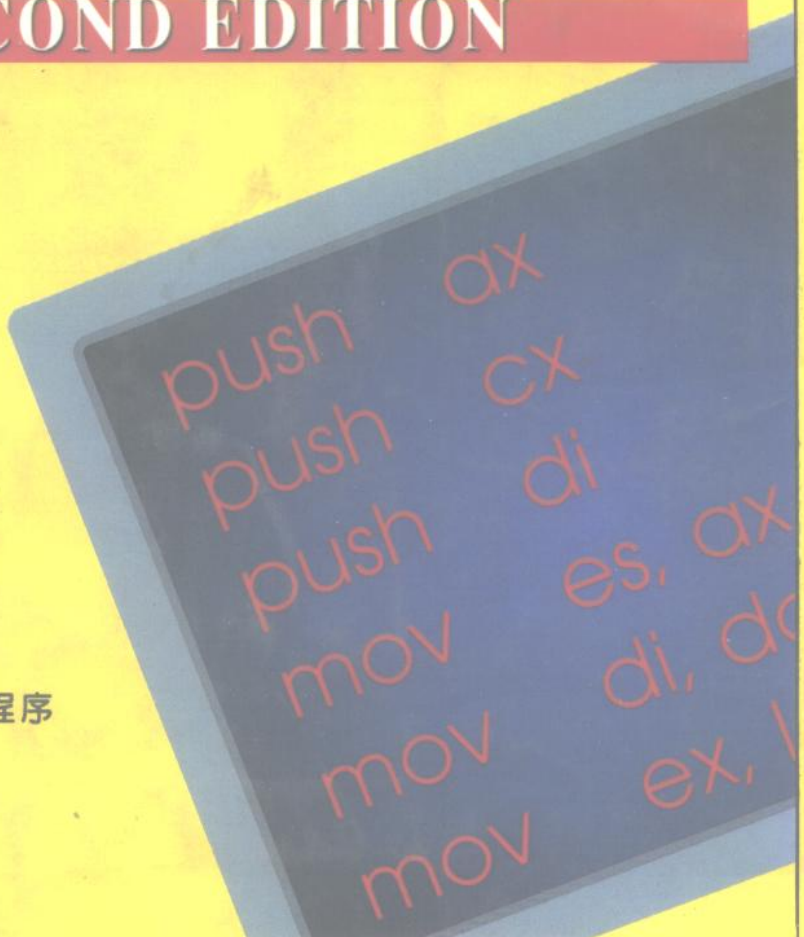
[美] Mark Goodwin 著
周予滨 田学锋 译
王 勇 校

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

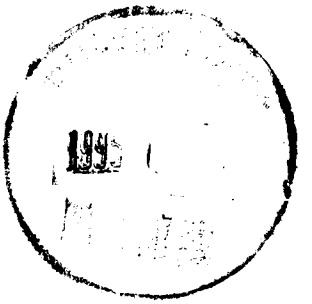
汇编语言程序设计自学教程

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
```