

21世纪高校
计算机系列教程

汇编语言程序设计



林邦杰 陈明 编著

引进台湾原版成熟教材■

注重内容的实用性，培养学生的专业能力■

适合高校电子信息类各专业选用■

经过全国高等院校计算机基础教育研究会著名专家学者、教授的评估与审定■

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

(京)新登字 063 号

北京市版权局著作权合同登记号: 01-2003-2676 号

版 权 声 明

本书中文繁体字版由台湾知城数位科技股份有限公司出版。本书中文简体字版经台湾知城数位科技股份有限公司授权由中国铁道出版社出版。任何单位或个人未经出版者书面允许不得以任何手段复制或抄袭本书内容。

本书贴有知城数位科技激光防伪标签,无标签者不得销售。版权所有,侵权必究。

图书在版编目(CIP)数据

汇编语言程序设计/林邦杰,陈明编著. —北京:中国铁道出版社, 2003. 5

(21 世纪高校计算机应用技术系列教程)

ISBN 7-113-05303-3

I. 汇… II. ①林…②陈… III. 汇编语言-程序设计-高等学校-教材 IV. TP313

中国版本图书馆 CIP 数据核字(2003)第 042210 号

书 名: 汇编语言程序设计

作 者: 林邦杰 陈 明

出版发行: 中国铁道出版社(100054, 北京市宣武区右安门西街 8 号)

策划编辑: 严晓舟 魏 春

责任编辑: 苏 茜 吴秋淑 赵树刚

封面设计: 孙天昭

印 刷: 北京市彩桥印刷厂

开 本: 787×1092 1/16 印张: 21 字数: 489 千

版 本: 2003 年 7 月第 1 版 2003 年 7 月第 1 次印刷

印 数: 1~5000 册

书 号: ISBN 7-113-05303-3/TP·953

定 价: 27.00 元

版权所有 侵权必究

凡购买铁道版的图书,如有缺页、倒页、脱页者,请与本社计算机图书批销部调换。

汇编语言程序设计

林邦杰 陈明 编著

中国铁道出版社

2003年·北京

TP313
L13

新华书店
PDG

21 世纪高校计算机系列教材

审定委员会

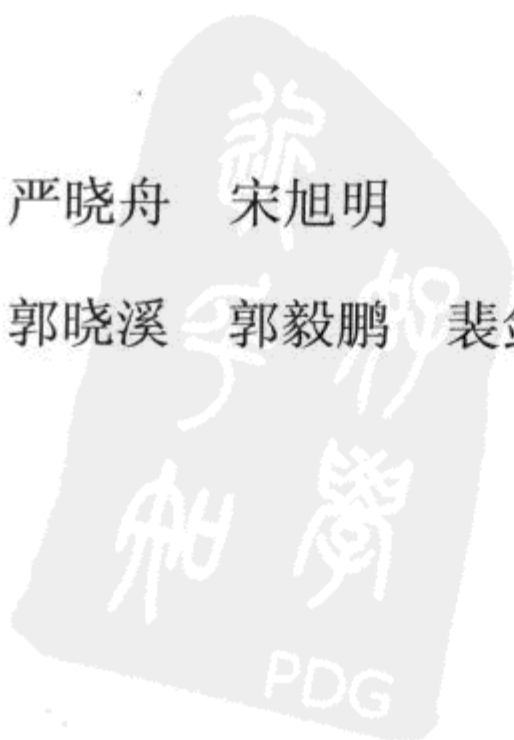
主 任：高 林

副主任：丁桂芝 李 畅

委 员：安淑芝 鲍有文 陈文博 樊月华 陈维兴
顾星海 李秀芬

项目策划：严晓舟 宋旭明

编 辑：郭晓溪 郭毅鹏 裴剑波



丛 书 序

随着人类跨入 21 世纪,以计算机和互联网为代表的信息技术的高速发展,使得计算机教育已经发展成为信息技术教育。21 世纪是信息技术高度发展,并在社会各行各业、各个层面得到广泛应用的时代,信息技术深刻地改变着人类的生活、工作和思维方式。时代要求每一个人都应当学习信息技术、应用信息技术。

随着我国社会主义市场经济和信息技术的高速发展,我国的高校计算机教育已步入从重视理论教育,走向理论与实践相结合、注重运用知识解决实际问题能力的发展阶段。此时大学计算机应用技术和高职高专教育还处于探索阶段,与之相适宜的教材建设已成为教育改革的重要方面。

我国台湾地区的高校计算机应用技术教育发展已相当成熟,在过去的二十几年中,为台湾地区经济发展培养了大批应用性人才,在职业化教育方面积累了丰富的经验;通过多年的实践,在处理理论与实践的关系以及培养实际应与操作的技术应用性人才方面都处于领先地位,也符合国际职业教育发展的主流。由于有着相同的历史文化背景,现阶段台湾的高校计算机应用技术教育教材对我们更具有借鉴作用。

因此根据教育部关于引进外文原版教材、推动我国教材改革的精神,通过多方面调查,在充分听取专家意见的基础上,中国铁道出版社以“引进,吸收,创新”为指导思想,力争走出一条新的高校计算机应用技术教育教材发展之路。作为这一决策的第一步,我们经过精心的组织策划,推出了“21 世纪台湾版高校计算机应用技术系列教材”。

我们本次引进的教材都是现阶段我国台湾地区的科技大学和技职学院正在使用的教材,这些学校是当前台湾发展高校计算机应用技术教育的主体学校,我们认为这些教材在编写过程中均采用了理论与实践相结合的方法,体现了能力本位的思想,注重在学习理论知识的基础上重点培养学生的实践能力,通过一系列实际训练,提高学生各种必备的技术应用能力,使他们一出校门便可发挥其所学专长,成为该领域的专门人才。

全国高等学校计算机基础教育研究会高林教授充分肯定了铁道出版社引进台湾版高职教材的思路,并在教材引进和编审过程中给予了全面指导,组织多位专家教授对这套教材进行了审定。专家一致认为本套教材内容新颖,易教易学,层次配套,符合高校计算机应用技术教育的特点,有利于推动我国高等职业教育的发展,建议在全国范围内推广使用。

本套教材可以作为大学计算机应用技术课程教材以及高职高专、成人高校和面向社会的培训班的教材,也可作为学习计算机应用技术的自学教材。

中国铁道出版社
2003 年 6 月



前言

汇编语言通常称为低级程序语言，只有学过汇编语言后才能了解如何控制整部计算机。在高级程序语言阶段，只能依编译器所提供的功能来设计程序。汇编语言虽然也需要编译，但是限制较少，设计的弹性较大，很多东西都要您自己动手做，因此您的成就感会更大，当然付出的代价也更高。

汇编语言给一般人的印象是艰深而难学，困难的原因有下列两点：

1. 对计算机各部分的功能不了解。
2. 操作系统只提供字符及字符串的输入输出函数，其他像整数及浮点数的输入输出都不提供，增加学习的难度。

对于第一点困难，正是我们学习汇编语言的目标，通过适当的指引以及范例的实现与说明，可以逐渐克服这一点困难。对于第二点困难就比较麻烦了，像整数及浮点数的输入输出程序必须对汇编程序设计有经验的人才能胜任，对初学者而言根本是不可能的事。

那么如何才能克服困难而学好汇编语言呢？

首先应以程序设计员的眼光来看待计算机，刚开始必须建立起程序的运行环境，如何编写程序，如何编译程序，如何运行程序以取得答案等的环境，为了这个目的，作者找了好久才在网络上找到 NASM 汇编语言，很容易就可以建立程序的运行环境，而且 NASM 是跨平台的，您可从下列网站免费下载：

<http://nasm.2y.net/>。

接着要将汇编语言当高级语言来学，要做到这一点必须取得像整数及浮点数的输入输出的程序。

本书第 3 章至第 14 章的程序，包括例题及习题，均使用 NASM 编译成 .COM 文件的格式，使用这种格式比较简单，程序代码、数据以及堆栈均在同一个分段里头，且只含单纯的机器码，因此较容易学习。第 15 章连接程序，例题及习题均使用 NASM 编译成 .obj 目标文件之后，使用免费的连接程序 `alink.exe` 将目标文件连接成可执行的 .EXE 格式。这两种格式都在 Windows 操作系统的 MSDOS 模式下执行的。因为 NASM 是跨平台的，当然也可以编译成目标程序在 Linux 等系统运行。

本书的例题及习题很多，单数习题附有解答，读者可从天勤网站 <http://www.tqbooks.net> “下载专区”中下载，同时，下载专区中还附有本书用到的宏程序、ASCII 码表等内容，读者可一并下载。

作者
2003 年 6 月



目 录

第 1 章 基本知识	1
1-1 汇编语言介绍	2
1-1-1 程序设计语言分类	2
1-1-2 汇编语言程序设计的意义	3
1-2 位及字节	4
1-3 二进制数	5
1-3-1 数字系统	5
1-3-2 补码	7
1-3-3 BCD 码	8
1-4 十六进制表示法	8
1-5 ASCII 码	9
1-6 个人计算机组成	9
1-6-1 处理器	10
1-6-2 内部存储器	11
1-6-3 段与地址	12
1-6-4 寄存器	13
1-7 硬件中断	17
课后习题	18
第 2 章 程序加载并执行	19
2-1 操作系统的组成	20
2-2 BIOS 启动程序	20
2-3 系统加载程序	21
2-4 堆栈	22
课后习题	25
第 3 章 NASM 汇编语言基础	27
3-1 源程序行格式	28
3-2 伪指令	29
3-2-1 定义含有初值的数据	29
3-2-2 定义不含初值的数据	29
3-2-3 INCBIN 伪指令	30
3-2-4 EQU 伪指令	30



3-2-5	TIMES 伪指令	30
3-3	有效地址	30
3-4	常量	31
3-4-1	数字常量	31
3-4-2	字符常量	31
3-4-3	字符串常量	31
3-4-4	浮点数常量	32
3-5	表达式	32
3-5-1	OR 运算符	32
3-5-2	XOR 运算符	32
3-5-3	AND 运算符	32
3-5-4	移位运算符	32
3-5-5	加及减运算符	33
3-5-6	乘及除运算符	33
3-5-7	单元运算符	33
3-6	临界表达式	33
3-7	局部标号	33
3-8	预处理器	34
3-8-1	%define 指令	34
3-8-2	%undef 指令	34
3-8-3	%assign 指令	34
3-8-4	多行宏	35
3-8-5	条件汇编	35
3-8-6	预处理循环	36
3-8-7	文件引用指引	37
3-8-8	标准宏	37
3-8-9	汇编语言指引	37
3-9	目标文件格式	38
3-10	NASM 汇编程序安装	38
3-11	范例	38
	课后习题	41
第 4 章 一般指令		43
4-1	源操作数与目的操作数	44
4-2	MOV 传送指令	44
4-3	XCHG 互换指令	46
4-4	有效地址送寄存器指令 LEA	46
4-5	指针送寄存器指令 LDS 及 LES 指令	46
4-6	压入 PUSH 及弹出 POP 指令	47

4-7	存储寄存器 PUSHA 及 POPA 指令	48
4-8	标志寄存器传送 PUSHF 及 POPF 指令	48
4-9	没有运算的 NOP 指令	49
	课后习题	49
第 5 章 基本输入与输出		51
5-1	软件中断 INT 指令	52
5-2	将一个字符串输出到屏幕	52
5-3	从键盘输入一个字符	53
5-4	将一个字符输出到屏幕	54
5-5	从键盘输入一个字符串	55
5-6	将一个字输出到屏幕	57
5-7	显示内存内容	58
5-8	键盘输入控制	60
5-8-1	由键盘输入字符	60
5-8-2	直接由键盘输入或输出字符	60
5-8-3	直接由键盘输入字符	60
5-8-4	直接由键盘输入字符	60
5-8-5	由键盘输入字符串	61
5-8-6	检查键盘状态	61
5-8-7	清除键盘缓冲区	61
5-8-8	从键盘缓冲区读取字符	61
5-8-9	测试键盘缓冲区是否有字符	61
5-8-10	传回控制键状态	62
5-9	屏幕输出控制	62
5-9-1	显示字符	62
5-9-2	显示字符串	62
5-9-3	设定光标位置	62
5-9-4	向上滚动屏幕	62
5-10	打印机输出控制	63
5-10-1	输出字符至打印机	63
5-10-2	打印一个字符	63
5-10-3	取得打印机状态	63
	课后习题	64
第 6 章 程序流程控制		65
6-1	标志寄存器	66
6-2	改变标志的指令	66
6-3	条件转移指令	67



6-4	比较两个整数.....	68
6-5	无条件转移指令 JMP.....	70
6-6	循环指令 LOOP.....	73
6-7	选择结构.....	76
6-8	循环结构.....	84
	课后习题.....	87

第7章 算术运算.....89

7-1	定点数与浮点数.....	90
7-2	带符号及无符号整数.....	90
7-3	加法及减法.....	91
7-4	乘法.....	96
7-5	除法.....	97
7-6	BCD 十进制数运算.....	99
7-6-1	BCD 加法.....	100
7-6-2	BCD 减法.....	103
7-6-3	BCD 乘法.....	106
7-6-4	BCD 除法.....	107
7-6-5	BCD 宏应用.....	108
7-7	综合例题.....	113
	课后习题.....	119

第8章 宏.....121

8-1	单行宏.....	122
8-1-1	%define 指令.....	122
8-1-2	%undef 指令.....	123
8-1-3	%assign 指令.....	123
8-2	多行宏.....	125
8-2-1	显示字符串宏.....	125
8-2-2	显示字符宏.....	126
8-2-3	读取字符宏.....	127
8-2-4	显示字节宏.....	130
8-2-5	读取字符串宏.....	133
8-2-6	字符串转换为数值.....	137
8-2-7	数值转换为字符串.....	139
8-2-8	数值输出至屏幕.....	142
8-3	条件汇编.....	144
8-4	预处理循环.....	145
8-5	源程序文件的包含内容.....	146

8-6 相关宏汇总	146
课后习题	150
第 9 章 过程	151
9-1 过程的定义	152
9-2 过程里的局部变量	154
9-3 传值调用	156
9-4 传址调用	160
9-5 堆栈传递参数	162
9-6 内存传递参数	163
课后习题	165
第 10 章 字符串处理	167
10-1 声明字符串	168
10-2 字符串长度	170
10-3 基本字符串指令	172
10-4 转换指令 XLATB	182
10-5 字符串宏	183
课后习题	193
第 11 章 位运算	195
11-1 位基本运算	196
11-2 位屏蔽	196
11-3 AND 指令	197
11-4 OR 指令	197
11-5 XOR 指令	198
11-6 NOT 指令	199
11-7 TEST 指令	201
11-8 改变位位置	203
11-9 左移及右移	204
11-10 算术左移及算术右移	207
11-11 循环位移	210
11-12 位移及循环位移指令总结	213
11-13 综合例题	213
课后习题	217
第 12 章 文件处理	219
12-1 输入及输出层次	220
12-2 输入及输出概念	220



12-3	标准的文件代号	221
12-4	建立一个文件代号	221
12-5	打开一个文件	222
12-6	关闭一个文件	223
12-7	从文件或设备读取数据	223
12-8	数据写入文件或设备	224
12-9	移动文件指针	231
12-10	检查并修改文件属性	233
12-11	建立新文件	234
12-12	删除文件	235
12-13	文件改名	235
12-14	建立或删除子目录	236
12-15	取得当前目录	237
12-16	改变当前目录	237
12-17	取得缺省的磁盘驱动器	238
12-18	改变缺省的磁盘驱动器	238
12-19	低级输入及输出	239
	课后习题	242
第 13 章 数据结构		243
13-1	数组声明	244
13-2	数组查找	247
13-3	使用 XLATB 指令转换	251
13-4	排序	252
13-5	队列	257
13-6	堆栈	260
13-7	链表	262
	课后习题	265
第 14 章 浮点数运算		269
14-1	80x87 协处理器的运算	270
14-2	浮点堆栈	270
14-3	状态字	272
14-4	控制字	275
14-5	数据类型	276
14-5-1	二进制整数	277
14-5-2	聚集十进制数	277
14-5-3	实数	278
14-5-4	七种数据类型值的范围	280

14-6 80x87 指令集.....	280
14-7 范例.....	282
课后习题.....	288
第 15 章 连接程序.....	291
15-1 建立 NASM 源程序.....	292
15-2 将目标文件连接成.exe 文件.....	293
15-3 显示 DOS 的 BIOS 区域数据.....	294
15-4 系统设备数据.....	296
15-5 内存容量.....	296
课后习题.....	297
附 录 NASM 汇编语言指令.....	299



1

基本知识

写汇编语言的程序 (Assembly Language Program, ALP) 必须具备有关计算机硬件 (computer's hardware) 以及指令 (instruction) 的知识。本章介绍汇编语言的基本概念、计算机硬件的基本功能, 包括: 位 (bit)、字节 (byte)、寄存器 (register)、内存 (memory)、处理器 (processor) 以及数据总线 (data bus), 至于指令则在以后的章节逐一介绍。

计算机硬件其内部主要为处理器、内存以及寄存器。寄存器其实是处理器中特殊的组成部分, 用于存储地址 (address) 及数据 (data)。计算机硬件外部主要为计算机的输入输出设备, 包括键盘 (keyboard)、显示器 (monitor)、磁盘驱动器 (disk) 以及 CD-ROM 等设备。软件 (software) 包含操作系统 (Operating System, OS)、程序 (program) 以及数据文件等, 通常存储在磁盘上。

要执行一个程序, 操作系统将程序从外部的设备, 通常为硬盘, 拷贝到内存里, 处理器执行这个程序的指令, 而寄存器负责处理所需的算术运算、数据的传送以及地址的运算等工作。





1-1 汇编语言介绍

程序设计语言指的是用来编写程序的语言。人们在相互交谈时，使用的是自然语言，如汉语、英语、俄语等。人们在同计算机打交道的时候也要使用语言，以便让计算机工作。计算机也通过语言把结果告诉给使用计算机的人。这就是所谓的“人机对话”。但是这种对话所使用的语言并不是我们平常人与人之间交流所使用的语言。人们把同计算机打交道的语言叫作程序设计语言或计算机语言。程序设计语言也是计算机系统软件的重要组成部分。进行程序设计的语言有很多，可以分为低级语言和高级语言。低级语言有机器语言和汇编语言，高级语言有 C/C++、PASCAL 等。

1-1-1 程序设计语言分类

机器语言

每一台计算机都配备有一套机器指令。每一条指令命令计算机执行一个特定的操作。我们就把这种不需任何加工，计算机就能够执行的程序叫做机器语言程序，把用二进制形式书写的的数据及计算机的指令系统称为机器语言。早期的计算机都是使用机器语言来工作的。

符号语言

机器语言编写的程序既难记，又很难懂，同时极易出错。人们经过长期的实践，不断的摸索，开始用十六进制数来表示各条指令（就是单板计算机常用的如 3E、4C、2B……机器码）。但是，仍然难以辨认和记忆。后来，人们提出用某些符号来代替这些难记的指令码，比如，一条指令，000100001010，用十六进制表示为 10A。可见简单得多了，用符号表示为 LDA,0AH。就是说，用 LD 这两个字母来表示操作码（LD:LOAD，原意是装载，装入等，符号使用了这个单词的词头、词尾），这样便很好记忆了。其他，如用 SUB（SUBTRACT，减法）、ADD（ADD，加法）等有实际意义的符号来代表对应的指令操作码。当然，这些符号可以按需要自己来规定。这比由“0”和“1”组成的机器码程序形象得多，也很容易辨认和记忆了。我们就把这些用符号写成的程序，叫符号语言程序。把用符号来表示的指令系统和数据称为符号语言。

汇编语言

符号语言已经具有很多优点，但符号语言中的每一个符号（这里称作助记符）可以是用户根据辨认或记忆的需要自己来决定的。这样，通用性不是很强。就是说，按某一个人的想法编写的符号语言程序，另外一个人是无法看懂的。汇编语言克服了上述的缺点，它是在符号语言的基础上发展起来的。它是针对一类（甚至几类）计算机，抽象出来的一种符号语言，并把这些符号加以统一规定，使得使用同类计算机的人都了解这些符号的意义，这样，使得用汇编语言编写的程序可以在这一类型的任何一台计算机上使用。这就有了极大的灵活性，当然不同类型的计算机的汇编语言也不同。它们都必须由生产厂家提供的汇编语言来编写。另外，汇编语言还增加了宏指令的功能。汇编语言是计算机工程控制中最常用的语言。

高级语言

汇编语言虽然较机器语言直观一些，但是比较繁琐难记。在 20 世纪 50 年代，人们研制出了高级程序设计语言（High-Level Programming Language）。高级语言比较接近于人类自然语言的语法习惯及数学表达形式，它与具体的计算机硬件无关，更容易被广大计算机工作者掌握和使用。利用高级语言，即使一般的计算机用户也可以编写软件，而不必懂得计算机的结构和工作原理。当然，用高级语言编写的源程序不会被机器直接执行，需要通过编译或解释程序的翻译才可变为机器语言程序。

广泛应用的高级语言有十多种，例如简单易用的 BASIC 语言、算法语言 FORTRAN、结构化语言 PASCAL、系统程序语言 C/C++ 等。用高级语言表示 2000 减 100，就是通常的数学表达形式：2000-100。

1-1-2 汇编语言程序设计的意义

高级语言简单、易学，而汇编语言复杂、难懂，是否就没有必要再采用汇编语言了呢？让我们首先比较一下汇编语言和高级语言的特点。

首先，汇编语言与处理器关系密切。每种处理器都有自己的指令系统，相应的汇编语言也各自不同。因而汇编语言程序的通用性、可移植性较差。而高级语言与具体计算机无关，高级语言程序可以在多种计算机上编译后执行。

其次，汇编语言编程涉及寄存器、内存等硬件细节，程序繁琐，调试也比较困难。而高级语言采用类似自然语言的语法，容易被掌握和使用，也不必关心诸如标志、堆栈等问题。

但是，汇编语言本质上就是机器语言，它可以直接有效地控制计算机硬件，因而可以产生运行速度快、指令序列短的高效率目标程序。而高级语言不易直接控制计算机的各种操作，编译程序产生的目标程序庞大、程序难以优化、运行速度慢。

总的来说，汇编语言的主要优点就是可以直接控制计算机硬件，可以编写在时间和空间两方面更有效的程序。这些优点使得汇编语言在程序设计中占有重要的地位。是不可取代的。

但汇编语言的缺点也是明显的，它与处理器密切相关，要求程序员熟悉计算机硬件系统，考虑许多细节问题，所以程序繁琐，调试、维护、交流和移植困难。因此，有时可以采用高级语言和汇编语言混合编程的方法，互相取长补短，更好地解决实际问题。

汇编语言主要应用场合有以下几种：

1. 程序要具有较短的运行时间，或者只能占用较小的存储容量。例如操作系统的核心程序段，实时控制系统的软件等。
2. 程序与计算机硬件密切相关，程序要直接控制硬件。例如 I/O 接口电路的初始化程序段，外部设备的底层驱动程序等。
3. 大型软件需要提高性能、优化处理的部分。例如计算机系统频繁调用的子程序、动态链接库等。
4. 没有适合的高级语言的时候。例如开发最新的处理器程序时。暂时没有支持新指令的编译程序。

汇编语言还有许多实际应用，例如分析具体系统尤其是该系统的底层软件、加密解密软



件、分析和防治计算机病毒等。

1-2 位及字节

计算机存储数据的最基本单位为位，它属于双态的，关闭（off）时它的值可视为 0 值，打开（on）时它的值可视为 1 值，一个位只能表示 0 或 1，不能表示其他更多的值，虽然如此，它的表现却会令您大吃一惊呢！

九个连续的位称为一个字节（byte），它是计算机内存里的一个存储位置（storage location），也表示外部设备的存储位置。每一个字节包括八个数据位（data bit）及一个同位检核位（parity bit）。八个数据位提供基本的二进制算术运算以及字符的表示方法。一个字节中的八个位，允许 256 种的组合，八个数据位都关闭时其值为 00000000，八个数据位全部打开时其值为 11111111，像大写的英文字母'A'，八个数据位的关闭打开组合为 01000001，像数字'1'，八个数据位的关闭打开组合为 00110001，这都在 256 种情况之中。根据规则每一个字节中打开位的总数必须是奇数，像'A'，八个数据位中打开的位数只有两个，为偶数，因此同位检核位的值必须是 1，如此总共有三个位的值是打开的，方才满足奇数的规定。又如'1'，八个数据位中打开的位数有三个，为奇数，因此同位检核位的值必须是 0，如此总共有三个位的值是打开的，才满足奇数的规定。当一个指令调用内存里的一个字节时，处理器都会检查它的同位数，若不满足奇数时，就会产生错误，处理器会认为数据已经丢失一个位，或丢失三个位，或丢失五个位，或丢失七个位，反正就是数据已经不正确了。这种情形通常是计算机硬件受损，或电路受干扰，或通讯品质不佳等因素造成的。

您一定会奇怪计算机怎么知道 01000001 这八个数据位的组合表示一个大写的英文字母'A'呢？当您从键盘输入一个 A，操作系统会在指定的内存里的字节设定其值为 01000001，您就可以对它做适当的处理，包括您可以将它传送至别的位置，加减其值，或将它以字符方式显示在显示器中等。

处理同位检核位为计算机硬件的工作，与程序设计无关，因此以后也不会再提到它。一个字节可视为八个数据位的组合，位编号从 0 至 7，第 0 个位表示在最右端，称为低位，第 7 个位表示在最左端，称为高位。

位编号 7 6 5 4 3 2 1 0

位内容 ('A') 0 1 0 0 0 0 0 1

程序不仅处理位及字节，它还处理几个连续字节所包含的数据。

word: 两个连续字节称为字。

double word: 两个连续字称为双字。

quad word: 四个连续字称为四个字。

ten bytes: 十个连续字节称为十字节。

paragraph: 十六个连续字节称为节。

Kilo Byte (KB): 1024 (2 的 10 次方) 个连续字节。

Mega Byte (MB): 1048576 (2 的 20 次方) 个连续字节。

在字里的位编号从 0 至 15，从右至左，下列的位内容，高位组为'1'，其字节合为 00110001，

低字节为'2'，其字节合为 00110010。

位编号	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
位内容 ('12')	0	0	1	1	0	0	0	1	0	0	1	1	0	0	1	0

在内存里的每一个字节都有一个独一无二的地址，内存最低字节的地址为 0，其次为 1，再其次为 2 等，若您的计算机内存总共为 1MB，那么最高字节的惟一地址为 1048575，即 2 的 20 次方再减去一。

1-3 二进制数

人一生下来，很自然地用十个手指头来计算，因此十进制一直被人用来记数。在其他的世界上，十进制不见得受欢迎。例如在章鱼的世界里，计数会使用八进制，即 0、1、2、3、4、5、6、7，逢八进位。最好的解释是章鱼有八个手指头，所以使用八进制。若在电子的世界里，因电子所表现出来的现象，很多是双态的，阴阳、开关、亮灭、01 等。因此电子喜欢使用二进制，即 0、1，逢二进位。因此不管计算机采用哪一种数字系统，最后必须转换为二进制数系统，方能被计算机处理。

位 (bit) 为二进制数 0 或 1。就像数字 (digit) 为十进制数 0 至 9 一样。很多数据均以字节表示，字符 (character) 以一个字节表示，中文字以两个或三个字节表示，整数有以两个或四个字节表示的，依数据不同而不同。

1-3-1 数字系统

一个数字系统，由 r 个有限符号所组成，即 0、1、2、...、(r-1)，逢 r 进位，称为 r 进制数字系统。r 称为该数字系统的基数 (radix)，或基址 (base)。例如十进制数系统由十个数 0、1、2、3、4、5、6、7、8、9 所组成，逢十进位，基数 r=10。又如八进制数字系统由八个数 0、1、2、3、4、5、6、7 所组成，逢八进位，基数 r=8。又如二进制数字系统由两个数 0、1 所组成，逢二进位，基数 r=2。又如十六进制数字系统由十六个数 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F 所组成，逢十六进位，基数 r=16。表 1-1 为四种数字系统的对照。

表 1-1 四种数字系统对照表

十进制	二进制	八进制	十六进制
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
8	1000	10	8
9	1001	11	9

若[^]符号表示乘方,即 $A \times A$ 表示为 A^2 , $A \times A \times A$ 表示为 A^3 ,则 r 进制整数 M 可表示如下:

$$365 = 3 \times 10^2 + 6 \times 10^1 + 5 \times 10^0$$

至于数字系统的转换可用余数定理算出,例如十进制 365 等于八进制数的什么呢? 计算如图 1-1 所示。

图 1-1 十进制 365 转换为八进制数, 计算结果为 555

$$365 = 5 \times 8^2 + 5 \times 8^1 + 5 \times 8^0$$

又如十进制数 18 转换为二进制数, 计算如图 1-2 所示。

图 1-2 十进制数 18 转换为二进制数, 计算结果为 10010

$$18 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

即二进制 10010。

十进制数 365 转换为十六进制数，计算如图 1-3 所示。

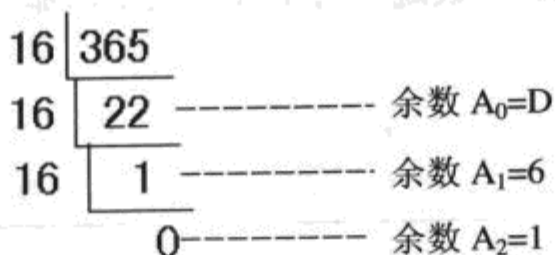


图 1-3 十进制数 365 转换为十六进制数，计算结果为 16D

因此十进制数

$$365 = 1 \times 16^2 + 6 \times 16^1 + D \times 16^0$$

即十六进制数 16D。事实上每一个十六进制数均由四位二进制数所组成，1 由 0001 所组成，6 由 0110 所组成，D 由 1101 所组成，因此十进制数 365 其二进制表示如下：

1	6	D	十六进制
0001	0110	1101	二进制

我们从右边每三个位绑在一起以一个数表示就是八进制数，也就是说十进制数 365 其八进制表示如下：

5	5	5	八进制
101	101	101	二进制

1-3-2 补码

补码用来表示负数。数轴以 0 为基点，往右一个单位为正 1，记为+1，往左一个单位为负 1，记为-1。若为十进制，如图 1-4 所示。

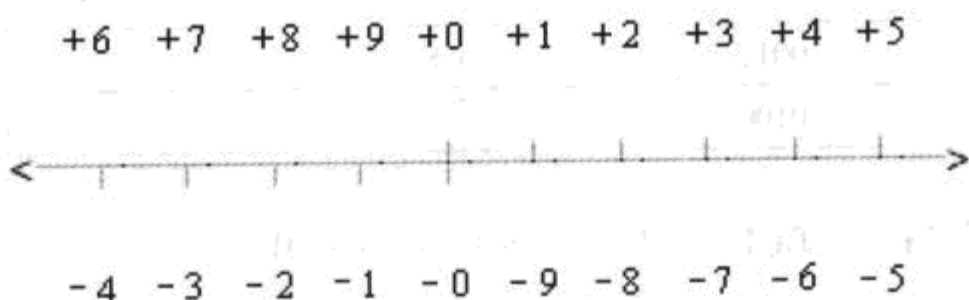


图 1-4 十进制个位数的正数及负数

个位数只有十个，即 0 至 9，但要表示正数及负数，一边一半最为公平，1 至 5 为正数，6 至 9 为负数，即+1、+2、+3、+4、+5 为正数，而+6、+7、+8、+9 为负数，分别表示-4、-3、-2、-1，这种以正数来表示负数的方法称为补码。其表示法如下：

$$C(N) = r^n - N$$

N 为 r 进制数，n 为其位数。如 N=7，r=10，n=1（因 N 为一位数），表示如下：

$$C(7) = 10^1 - 7 = 3$$

则称 3 为 7 的补码，即 C(N) 为 N 的补码。即 3 代表-7。7 代表-3。

如 $r=10$ 称为十进制补码, $r=2$ 称为补码。又如 36 的十进制补码表示如下:

$$C(36) = 10^2 - 36 = 64$$

即 36 表示-64。64 表示-36。负数既然可用补码表示，那么减法就能用加法表示。如图 1-5 所示。

$$\begin{array}{r}
 764 \\
 - 312 \\
 \hline
 \end{array}
 \qquad
 \begin{array}{r}
 764 \\
 + (-312) \\
 \hline
 \end{array}
 \qquad
 \begin{array}{r}
 764 \\
 + 688 \\
 \hline
 1452
 \end{array}$$

1 为进位，抛弃 答案为 452

图 1-5 十进制 764-312 转换为 764+(-312) 补码加法

若为二进制补码, 例如:

$$\begin{aligned} C(1011) &= 2^4 - 1011 \\ &= 10000 - 1011 \\ &= 0101 \end{aligned}$$

即 1011 的二进制补码为 0101。也就是说用 0101 来表示-1011。

1-3-3 BCD 码

BCD 码 (Binary Coded Decimal) 即二进制编码的十进制, 是每一个十进制数字以二进制四个位表示的一种编码方法。如表 1-2 所示。

表 1-2 BCD 码对照表

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

例如十进制数 365 以 BCD 码表示为 0011 0110 0101。

1-4 十六进制表示法

虽然一个字节有 256 种不同的组合，每一种组合代表一个 ASCII 码的字符，但并不是每一个字符均能显示在显示器上的，例如 Tab 字符、Enter 字符、跳页以及 Escape 字符等均不能显示在显示器上，因此计算机的设计者就提供一个速记法，将一个字节划成两半，左半部的四位称为高半字节（high nibble），右半部的四位称为低半字节（low nibble），每四个位可以用一个十六进制数表示。十六进制由 0 至 9 及 A 至 F 等十六个符号所组成，逢十六就进位，故称为十六进制。如表 1-3 所示。

表 1-3 十六进制与二进制及十进制对照表

二进制	十进制	十六进制	二进制	十进制	十六进制
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

汇编语言在许多场合都使用十六进制表示法，例如汇编后的列表文件其地址、指令、数据等均以十六进制表示法表示，在程序调试（debug）时也经常以十六进制表示法表示数据。十六进制加减法与十进制相同，只不过逢 16 才进位而已。下列五个例子中，“6+4”十进制为 10，但十六进制表示为 A。“6+8”十进制为 14，但十六进制表示为 E。“6+A”十进制为 16，但十六进制表示为 10，“6+12”十进制为 18，但十六进制表示为 12。“26+43”十进制为 69，但十六进制表示为 45。

6	6	6	6	1A
+ 4	+ 8	+ A	+ C	+ 2B
<hr/>				
A	E	10	12	45

在汇编语言中表示十六进制数只须在数字后端加上 H 或 h 就可以了。
例如您光写 45，机器会将它当成十进制数，但您若写成 45H，机器就会将它当成十六进制数了。

1-5 ASCII 码

ASCII 码（American Standard Code for Information Interchangable）为美国信息交换标准码，是国际间认可的交换码，主要用于计算机间的通讯及数据传输。ASCII 码由 8 位组成，总共可表示 256 个符号。

- 00 - 1F: 为控制字，用于代表特殊数据传输字，或控制输出输入装置的机械特性。
- 20 - 5F: 代表特殊字、数字和大写英文字母。
- 60 - 7F: 代表小写英文字母和某些特殊字。
- 80 - FF: 扩充的 ASCII 码，表示一些外国字母、数学符号、绘图符号等。

1-6 个人计算机组成

个人计算机的主要组成部分为主板（system board），它包含处理器、协处理器



(coprocessor)、主存储器 (main memory)、连接器 (connector) 以及扩展槽 (expansion slot) 等。连接器及扩展槽用于取得诸如只读存储器 (ROM)、随机访问内存 (RAM)、硬盘、CD-ROM、额外的主存储器、video 单元、键盘、鼠标、并行及串行设备、音响设备、高速缓存等的信息。处理器使用高速缓存以提高处理速度。

总线 (bus) 是一些附于主板上的信号线, 连接各个组成部分, 负责在处理器、内存及外部设备组成部分之间传递数据。例如您的程序需要从外部存储设备读取数据, 处理器决定要存放的内存地址, 并将该地址放进地址总线, 内存单元传输数据至数据总线并通知处理器数据已经准备就绪, 这时处理器才从数据总线取得数据并将它存入内存地址。

1-6-1 处理器

个人计算机的中枢就是处理器, 也称为中央处理器 (central processing unit, CPU), 对于 Intel 公司的 80x86 家族而言, 处理器负责所有指令的执行以及数据的处理。各种处理器对于处理速度、内存容量、寄存器以及数据总线等均有所差异, 下面是对于 Intel 处理器的简短说明:

8088: 具备 16 位寄存器及 8 位数据总线, 能寻址至 1 MB, 虽然寄存器可同时处理两个字节, 但数据总线一次却只能处理一个字节而已。这个处理器的程序在真实模式 (real mode) 下执行, 每次执行一个程序, 其地址通过段寄存器 (segment register) 寻址。

8086: 与 8088 类似, 数据总线一次能处理一个字, 较 8088 快速。

80286: 较 8086 快速, 能寻址至 16 MB, 其程序可以在真实模式或保护模式 (protected mode) 下执行。

80386: 具备 32 位寄存器及数据总线, 能寻址至 4096 MB, 其程序除了可以在真实模式或保护模式之下执行之外, 还可以在虚拟模式 (virtual mode) 下执行, 可将磁盘当成虚拟内存, 对于并行处理 (concurrent process) 提供较大的空间。

80486: 与 80386 类似, 具备高速缓存, 因此速度较快。

Pentium: 具备 32 位寄存器及 64 位数据总线, 具备数据及内存高速缓存, 速度较以前的处理器有所提高。

● 执行单元及总线接口单元

图 1-6 说明执行单元 (Execution Unit, EU) 及总线接口单元 (Bus Interface Unit, BIU) 操作的情形。处理器分为两个逻辑单元: 执行单元及总线接口单元, 执行单元用于执行指令, 而总线接口单元用于传递指令及数据给执行单元。执行单元包含一个算术逻辑单元 (ALU)、一个控制单元 (CU) 以及一组寄存器, 这些组成部分提供执行指令及算术逻辑运算的环境。

总线接口单元最重要的功能就是管理总线控制单元 (bus control unit)、段寄存器 (segment register) 以及指令队列 (instruction queue)。总线接口单元控制将数据传给执行单元、传给内存、传给外部设备的总线。段寄存器控制内存的寻址。总线接口单元负责将指令存入指令队列中, 提供指令给执行单元执行。

执行单元及总线接口单元各自独立执行, 总线接口单元提供指令给执行单元, 执行单元执行完指令后告诉总线接口单元, 总线接口单元又提供下一个指令给执行单元, 如此重复, 直到指令执行完毕为止。

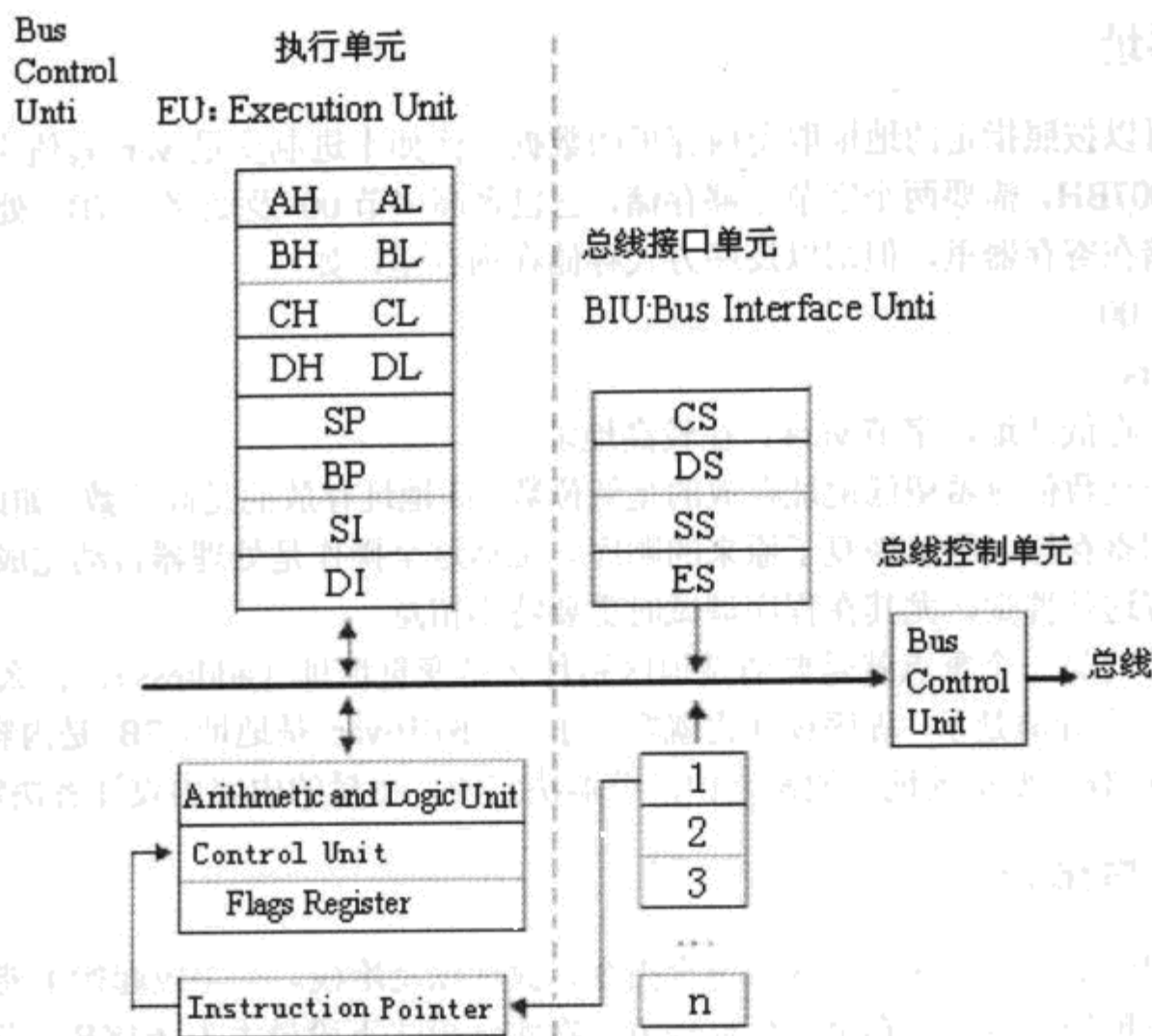


图 1-6 执行单元及总线接口单元操作情形

1-6-2 内部存储器

个人计算机提供两种类型的内部存储器，ROM 及 RAM。

ROM: Read Only Memory 表示只读存储器。

RAM: Random Access Memory 表示随机访问内存。

● ROM 部分

ROM 部分包含特殊的内存芯片，只能读出数据，不能写入。因为数据及指令已经永久烧入芯片中，不能再改变。ROM 中的基本输入输出系统 (Basic Input/Output System, BIOS) 位于 0F0000H 处，负责处理输入输出设备，例如硬盘的控制程序。在 ROM 中第 0F0000H 处的指令控制计算机的基本功能，例如开机时的自动测试以及磁盘的自动加载程序等。

● RAM 部分

RAM 部分是程序设计者最关心的，RAM 是一种可读可写的内存，它是您的程序运行时指令及数据存储的平台，当您将计算机的开关打开，ROM 里的启动程序会将操作系统部分程序加载至 RAM 中，是计算机与用户之间的一个接口，您当然可以要求它为您服务。您的程序在 RAM 里运行，通常会输出至显示器、打印机或磁盘文件。

当您关掉计算机后，RAM 的部分会被清除，但 ROM 的部分并不受影响，因为它是只读的关系。



● 数据寻址

处理器可以按照指定的地址取得内存里的数据。例如十进制变量 `var` 其值为 123，十六进制表示为 007BH，需要两个字节才够存储，它包含高字节 00 及低字节 7B，处理器将它以原来顺序存储在寄存器里，但却以反序方式存储在内存里，如：

7B 00

`var` `var+1`

字节 `var` 在低地址，字节 `var+1` 在较高地址。

处理器处理数值时希望低地址存放的是低位数，高地址存放的是高位数，如此较为方便，若将 `var` 存回寄存器，则又恢复了原来的顺序。虽然这个操作是处理器自动完成的，但程序设计者必须有这种警觉，尤其在程序调试时更要特别留意。

程序设计的另一个重点就是要清楚地区别什么是变量地址 (`address`)，什么是变量内容 (`content`)，这个弄清楚了，程序设计也就容易了。上例中 `var` 是地址，7B 是内容。`var+1` 是地址，00 是内容。变量的地址通常是由汇编器决定的，变量的内容由设计者决定。

1-6-3 段与地址

段是您程序中一个特殊的区域，包含指令、数据以及堆栈。一个段起始于能被十六除尽的地址，这个地址可以是内存里的任何地方，在实体模式下段最大为 64KB。当指令将一个地址加载到段寄存器时自动左移四个位，意思就是说保持低四位的值为零，也就保证可以被 10 除尽。

在您的程序里，您可以定义若干个段，要使用哪一个段时，只要将地址加载段寄存器就可以了。在实体模式下的三个段为指令段、数据段以及堆栈段。

指令段

包含要执行的机器指令，正常的情形第一个必须是可执行的指令，指令段寄存器 (`CS`) 存放着此段的起始地址。

数据段

包含程序所定义的数据、常量以及工作区。数据段寄存器 (`DS`) 存放着此段的起始地址。

堆栈段

包含程序所需的暂存数据工作区。堆栈段寄存器 (`SS`) 存放着此段的起始地址。

● 段地址

段寄存器只有 16 位，其内容值为段的起始地址。图 1-7 显示段寄存器 `SS`、`DS` 及 `CS` 与内存堆栈段、数据段及指令段的关系，这只是一个例子，其放置不一定是这样的顺序，80386 及其后的处理器另外具备 `FS` 及 `GS` 段寄存器，提供更多的段。一个段起始于能被十六除尽的地址，当指令将一个地址加载到段寄存器时，自动左移四个位，保持低四位的值为零。

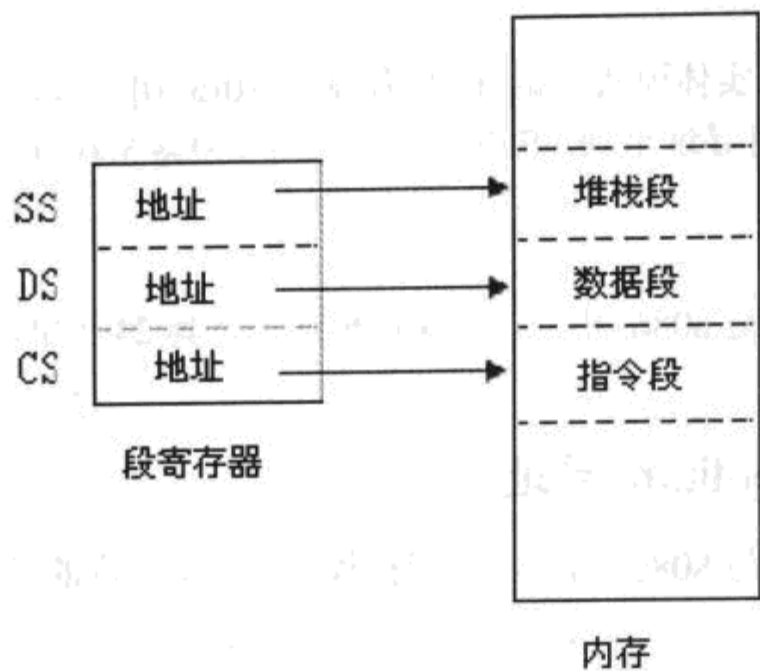


图 1-7 段寄存器与内存关系

● 段偏移

在您的程序中所有的存储位置都相对于段的起始地址，这个相对地址称为偏移（offset）。若段的起始地址为零值，那么偏移就是地址了。在实体模式下两个字节的偏移范围为 0 至 65535，即 0 至 0FFFFH。要访问段内的内存地址的内容，处理器要将段地址及段偏移结合成地址，才能取得该地址的内容。例如您有一个变量 var，它的偏移值为 0104H，这时，数据段寄存器的内容为 10E6H，那么 var 变量的绝对地址如下：

DS 内容地址	10E60H
var 偏移值	+ 0104H

var 绝对地址	10F64H

var 的实际地址为数据段寄存器 DS 的内容 10E6H 往左移四个位 10E60H 后再与 var 的偏移值相加，其结果为 var 的实际地址。这种将相对地址转换为实际地址的工作，处理器会自动执行的，您就不用担心了，虽然如此您还是要了解它的操作情形。段寄存器内容 10E6H 往左移四个位 10E60H，本书往后以方括号表示为 10E6[0]H。

1-6-4 寄存器

处理器中的寄存器用于控制正在执行中的指令，包括处理内存寻址以及提供算术运算的能力。在程序中使用寄存器只要直接使用它的名称就行了，例如 CS、DS 及 SS 等。寄存器位编号也是从 0 算起，由右往左排列。

1-6-4-1 段寄存器

一个段寄存器提供一块内存的地址，称为当前段（current segment）。个人计算机所使用的 Intel 公司处理器提供不同的寻址功能。



● 8086/8088 寻址

段寄存器为 16 位，在实体模式下执行。例如地址 10E6[0] 存入段寄存器时只取 10E6H。其寻址使用 20 位，最高可寻址至 0FFFF[0]H，相当于 1,048,560 个字节。

● 80286 寻址

在实体模式下其寻址与 8086 相同。在保护模式下，其 24 位的寻址可至 0FFFFFF[0]H 或 16MB。

● 80386/80486/Pentium 寻址

在实体模式下其寻址与 8086 相同。在保护模式下其 48 位的寻址可至 4096 MB。

● CS 寄存器

包含程序指令段 (code segment) 的起始地址，这个地址再加上指令指针 (Instruction Pointer, IP) 寄存器的偏移值，这个地址就是准备提取并执行的指令地址。

● DS 寄存器

包含程序数据段 (data segment) 的起始地址。指令使用这个地址以决定数据的位置。这个地址再加上变量的偏移值，就可取得该变量的地址及其内容值。

● SS 寄存器

允许在内存里实现一个堆栈，以便存储暂时的地址及数据。系统将堆栈的起始地址存入堆栈寄存器 SS 中。这个地址再加上堆栈指针 (Stack Pointer, SP) 的偏移值，就可取得该堆栈顶端元素的地址及其内容值。

● ES 寄存器

使用于某些字符串的操作以处理内存的寻址。额外段寄存器 (Extra Segment, ES) 常与目的变址寄存器 (Destination Index, DI) 搭配。若您的程序必须使用 ES 寄存器，您必须先赋予初值。

● FS 及 GS 寄存器

80386 以上处理器所增加的寄存器，提供更多的段选择。

1-6-4-2 指针寄存器

指针寄存器 (pointer register) 指 32 位的 EIP、ESP 和 EBP 以及 16 位的 IP、SP 及 BP。

● IP 寄存器

16 位的 IP 寄存器存储偏移值。IP 与 CS 结合 CS:IP，这时的 IP 表示相对于当前指令寄存器的指令偏移值，指到下一个要执行的指令。32 位的寄存器中称为 EIP。

● SP 寄存器

16 位的 SP 寄存器存储偏移值，与 SS 结合 SS:SP 这时的 SP 表示相对于当前堆栈寄存器的偏移值，指到堆栈顶端。32 位的寄存器中称为 ESP。

● BP 寄存器

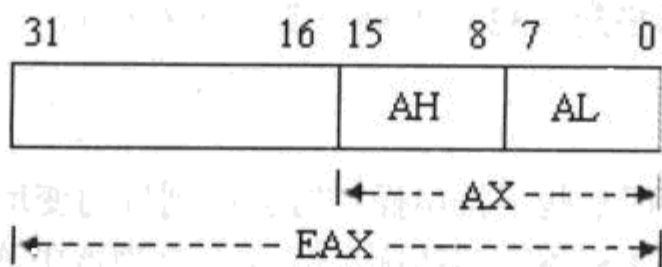
16 位的 BP 寄存器提供参数的参考地址，那是程序通过堆栈传递的参数，包括地址及数据，处理器结合 SS 的地址以及 BP 的偏移值。BP 当一个基址寄存器，也与 DI 或 SI 变址寄存器结合提供特别的寻址。80386 引进一个扩展的 BP 寄存器，称为 EBP。

1-6-4-3 通用寄存器

通用寄存器在 32 位的处理器中指 EAX、EBX、ECX 及 EDX。在 16 位的处理器中指 AX、BX、CX 及 DX。

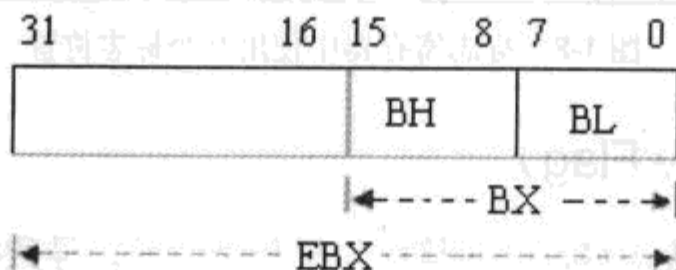
● AX 寄存器

AX 寄存器是主要的累加 (accumulator) 寄存器，用于输入输出以及大部分的算术运算，例如加减乘除等四则运算。16 位的 AX 是 32 位 EAX 的右半部分，AX 本身也包含两个部分：左半 8 位称为 AH，右半 8 位称为 AL。



● BX 寄存器

BX 寄存器是主要的基址 (base) 寄存器，用于地址的索引或计算。BX 是一个基址寄存器，也与 DI 或 SI 变址寄存器结合提供特别的寻址。16 位的 BX 是 32 位 EBX 的右半部分，BX 本身也包含两个部分：左半 8 位称为 BH，右半 8 位称为 BL。



● CX 寄存器

CX 寄存器是主要的计数 (count) 寄存器，它的内容决定循环的次数，或左移、右移的位数，您当然也可以使用 CX 当一个操作数。16 位的 CX 是 32 位 ECX 的右半部分，CX 本

AF: 辅助进位标志 (Auxiliary Carry Flag)

辅助进位标志用于十进制数的运算。运算后视情况而设定为 1 或 0。

ZF: 零标志 (Zero Flag)

任何运算其结果为 0 时, 标志值设定为 1, 否则标志值设定为 0。

SF: 符号标志 (Sign Flag)

执行的运算为具有符号者, 若结果为负数则符号标志值设定为 1, 若为正数则符号标志值设定为 0。

TF: 陷阱标志 (Trap Flag)

标志值若为 1, 则执行单步操作, 每次执行一个指令。

IF: 中断标志 (Interrupt Flag)

标志值若为 1, 则启动中断功能, 若其值为 0 则终止中断功能。

DF: 方向标志 (Direction Flag)

方向标志用于字符串运算, 标志值若为 0, 则每次增加 SI, DI 的值。标志值若为 1, 则每次减少 SI, DI 的值。

OF: 溢出标志 (Overflow Flag)

在执行具有符号的运算时, 当运算的结果数字太长, 导致指定操作数无法容纳时溢出标志值被设定为 1, 否则设为 0。

标志寄存器中 ZF、SF、OF、CF、AF、PF 这 6 个称为状态标志 (status flag), DF、IF、TF 这三个称为控制标志 (control flag)。控制标志用于控制处理器 (processor) 的操作。状态标志提供指令执行后的相关状态信息, 但控制标志控制尚未执行指令的行为。

1-7 硬件中断

某些事件 (event) 导致处理器暂停正在执行的工作, 转而处理该事件, 处理完毕后又继续刚刚暂停的工作。有些事件是正常而经常发生的, 例如从键盘输入数据产生一个事件, 导致处理器暂停正在执行的工作, 转而调用 BIOS 中从键盘输入数据的程序, 处理完毕后又继续刚刚暂停的工作。有些事件是不正常的, 例如您执行除法, 但不小心让除数为零值, 导致处理器停止执行您的程序。

另一种为软件中断 (software interrupt), 您的程序本身发出一个请求, 要将数据显示在显示器上, 导致处理器暂停正在执行的工作, 转而调用 BIOS 中将数据显示在显示器上的程序, 处理完毕后又继续刚刚暂停的工作。

1. 请将十进制数 123 转换成二进制数、八进制数以及十六进制数。
2. 请将十进制数 365.25 转换成二进制数、八进制数以及十六进制数。
3. 请将十进制数 123 以 BCD 码表示。
4. 请将十进制数-123 转换成二进制数、八进制数以及十六进制数，以补码方式表示。
5. 请将十进制数-365.25 转换成二进制数、八进制数以及十六进制数，以补码方式表示。
6. 举例说明什么是段寄存器。
7. 举例说明什么是通用寄存器。
8. 举例说明什么是变址寄存器。
9. 举例说明什么是标志寄存器。
10. 请说明标志寄存器里每一个标志的作用。



如何安装系统 1-5

2

程序加载并执行

本章说明如何将程序加载到个人计算机，并加以运行，这要靠安装于个人计算机的操作系统所提供的服务了。个人计算机的软件环境包括操作系统的功能及其组件。我们希望了解个人计算机的启动程序，开机时操作系统如何将它加载至内存里，操作系统如何将我们的程序加载至内存里并加以运行。





2-1 操作系统的组成

操作系统 (Operating System, OS) 是一般用途的且对于设备数据的访问具有独立性 (device-independent) 的一套程序, 所谓的设备包括键盘、屏幕、磁盘机及打印机等, 所谓独立性是指操作系统是操作系统, 设备数据的访问是设备数据的访问, 互相独立运作, 当用户需要设备数据的访问服务时, 操作系统就调用设备数据的访问例程 (routine) 运行过后又回到用户程序。构成个人计算机操作系统的组件包括下列的项目:

文件管理组件

操作系统维护存储在系统磁盘的文件 (file) 及目录 (directory) 中。程序可建立 (create) 及更新 (update) 文件数据, 但操作系统负责处理数据存放在磁盘的位置。

输入输出组件

程序需要从系统输入数据或将数据输出, 这要靠中断来处理了, 程序设计者可免于自己设计低级 (low-level) 的输入输出程序。

程序载入组件

当用户或程序需要运行一个程序时, 程序载入组件负责将程序从磁盘载入, 存于内存, 接着开始运行程序。

内存管理组件

当程序载入组件将程序从磁盘载入要载入的内存时, 内存管理组件配置一块足够您的程序运行的内存给程序, 包括指令、数据及堆栈空间。程序可以处理位于配置内存里的数据, 您不需要的内存可归还给内存管理组件, 内存不够使用也可以向内存管理组件申请。

中断处理组件

操作系统提供中断处理机制, 让您可以轻松访问存储于外部设备的数据。

2-2 BIOS 启动程序

当您打开个人计算机的电源开关时, 处理器进入重置 (reset) 状态, 将所有内存清除为零值, 执行内存同位测试, 将 CS 段寄存器的内容设为 0FFFF[0]H, 程序指针寄存器 IP 的偏移值设定为零值。因此第一个要执行的指令为位于 CS:IP 的指令, 它的地址为 0FFFF0H, 这个地址就是 BIOS 的进入点 (entry point)。

BIOS 在 ROM 里头提供一组例程支持对于存在设备上的数据的访问。BIOS 的例程位于 0FFFF0H, 它检查各个输入输出端口 (port), 然后对于存在的设备初始化 (initialize), 接着 BIOS 建立两个数据区:

1. 中断向量表

中断向量表的起始地址为低地址的零值, 包含 256 个四字节长的地址, 其格式为:
段地址: 段偏移

当中断发生时 BIOS 及操作系统使用中断向量表的相对应地址以执行该中断例程。

2. BIOS 数据区

BIOS 数据区的起始地址为 40[0]H, 其大小依设备的多少而定。

接着 BIOS 检查系统磁盘是否存在, 接着从系统磁盘载入启动程序, 启动程序随后从系统磁盘加载系统文件至内存, 并将控制权交给操作系统。

操作系统的一个主要任务是与 BIOS 产生互动。当需要 BIOS 提供服务时操作系统会调用 BIOS 的相关例程。

2-3 系统加载程序

一旦 BIOS 将控制权交给操作系统之后, 您就可以向操作系统要求运行程序了。可运行的程序有两种: .com 程序及 .exe 程序。

一个 .com 的程序只包含一个段, 段内包括指令、数据及堆栈, 这种程序是属于小而美的一种, 适合用于小的应用程序 (application), 或小的公用程序 (utility), 或常驻程序 (resident program)。本书绝大部分的程序均使用 .com 的格式。

.com 程序被加载运行时有下列的特点:

1. 一个 .com 程序占用一个段 (64KB) 的运行空间, PSP (Program Segment Prefix)、指令、数据及堆栈都安排在这个段内。段寄存器 CS、DS、ES 及 SS 均指到这一段的起始地址。
2. 段内安排以 PSP 在前, 指令及数据居中, 堆栈则安排在段的高地址部分。PSP 占用 256 个字节的空間, 因此第一个指令的偏移地址必须从 0100H 开始, IP 寄存器的初值即是 0100H。
3. PSP 及堆栈都由系统配置, 系统会在堆栈顶端存入两个字节, 其内容为零值, SP 就指到这个零值的数据项, 这时 SP 的内容为 0FFFEH。

从这些特点大致可以看出 .com 的程序应该如何编写:

1. 只占用 64KB 的内存。
2. 只有一个段, 指令、数据及堆栈都安排在这个段内。
3. 第一个指令必须从偏移值 0100H 开始, 可使用伪指令 ORG 0100H 完成。
4. 程序中不可另行定义堆栈, 堆栈由系统自动配置。
5. 不必给 DS 及 ES 赋初值。

.com 程序的内存配置如图 2-1 所示。

在实体模式 (real mode) 下, 一个 .exe 格式的程序分别包含指令段、数据段及堆栈段, 这种程序是较为严谨的一种, 适合用于大型的应用程序, 或结合多种目标程序 (object program) 的大型程序。

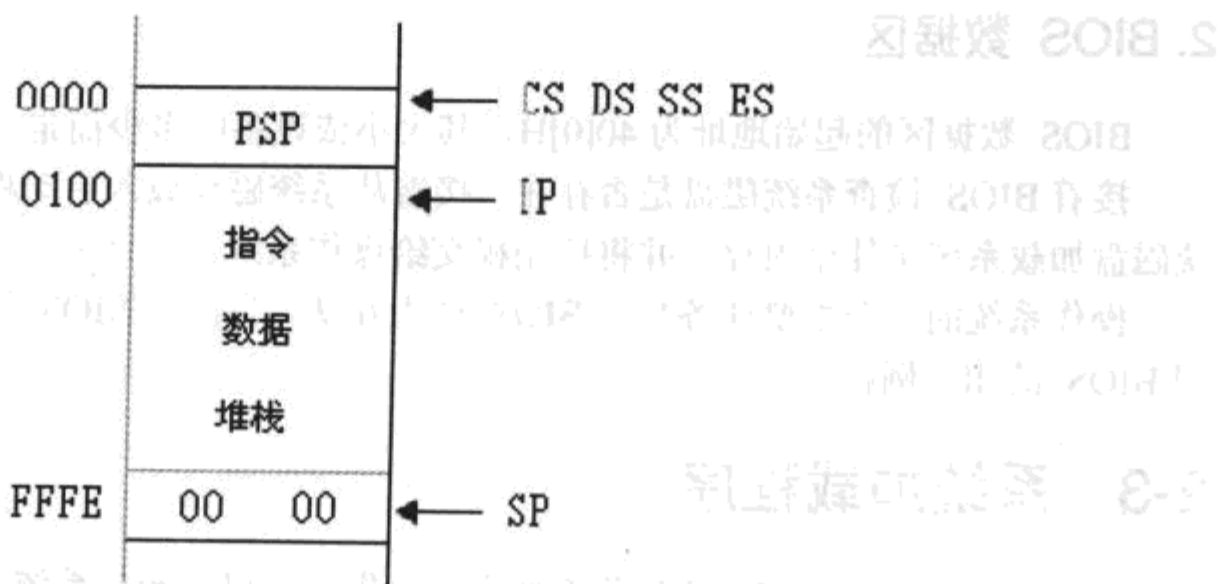


图 2-1 .com 程序的内存配置

.exe 程序被加载运行时运行下列步骤：

1. 从磁盘取得.exe 程序。
2. 在内存里建立一个占 256 字节的 PSP 区域。
3. 将.exe 程序放在 PSP 后端。
4. 将 PSP 地址存入 DS 及 ES 寄存器中。
5. 将指令段的地址存入 CS 段寄存器中，同时将 CS 内第一个要执行的指令的偏移地址存入 IP 寄存器中。
6. 将堆栈地址存入 SS 段寄存器中，同时将 SP 的值设定为堆栈长度。
7. 将控制权转给.exe 程序，执行第一个指令。

从上面的步骤您可以看出系统加载程序帮我们设定了 CS:IP 地址，也设定了 SS:SP 地址，但却将 PSP 地址存入 DS 及 ES，因此如果您想让 DS 及 ES 指到您的数据段，您必须自己设定了。

2-4 堆栈

不管是.com 还是.exe 的程序，在程序里头都需要建立一个堆栈（stack），堆栈通常有如下的用途：

1. 程序调用过程

程序调用过程（procedure）时将返回地址压入堆栈，当过程运行完毕后再取出返回地址，从而返回调用程序。

2. 调用过程的参数

程序调用过程时将参数存于堆栈，以便过程使用，过程执行的结果也可以存入堆栈供调用程序使用。

3. 存储计算的结果

程序必须使用寄存器来执行运算，您可以将寄存器的值存于堆栈，然后执行运算，最后将存于堆栈的值存回寄存器中。

对于.com 格式的程序，系统加载程序时会自动定义一个堆栈。但对于.exe 格式的程序系统加载程序时并不会自动定义一个堆栈，因此必须明确定义它。

在实体模式下，堆栈中的每一个数据项（data item）为一个字，段寄存器 SS 的内容由系统加载程序赋予初始值，指向堆栈的起始地址。

堆栈指针寄存器 SP 刚开始时其偏移值为整个堆栈的长度（字节数）。堆栈存储方法与一般的方法不同，它是从高地址开始存储，往低地址扩展。堆栈数据结构的访问都只发生在顶端，因此只要一个顶端指针就可以，这个顶端指针就是 SP。堆栈有两种操作：压入（push）及弹出（pop）。

压入

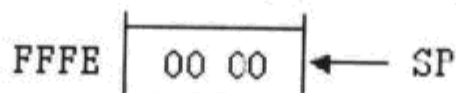
SP 减去 2 后，将要压入的字数据存入 SP 所指的数据项中。

弹出

取出 SP 所指的数据项后，SP 值加上 2。

假如 AX 的值为 1234H，BX 的值为 5678H，我们来做压入及弹出的操作。

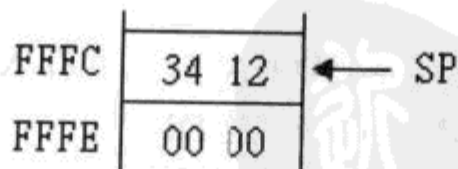
(1) 堆栈空白时如下所示。SP 指到 0FFFEH 的偏移值。



(2) 将 AX 压入堆栈顶端。

- 首先将 SP 的值减 2，指到 0FFFC 的偏移值。
- 接着将 AX 的值拷贝至 SP 所指的数据项。

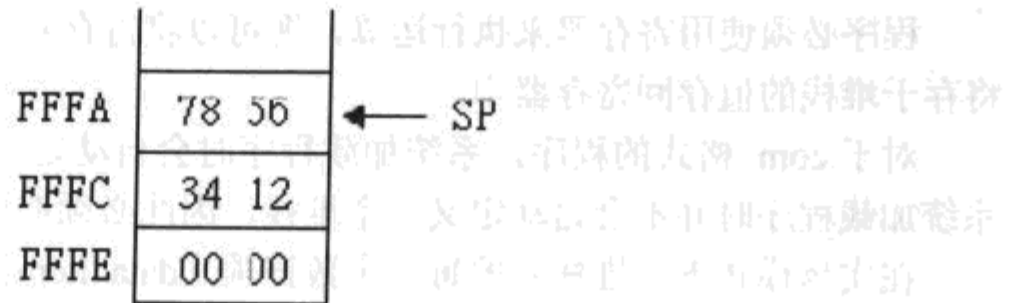
因为堆栈安装在内存里，因此低地址存储 AX 的低字节 AL 的值，高地址存储 AX 的高字节 AH 的值，所以数据项的内容为 3412H。



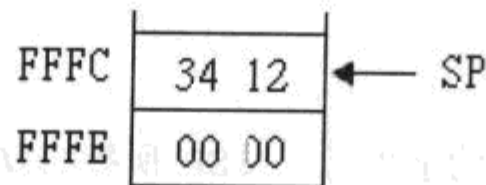
(3) 将 BX 压入堆栈顶端。

- 首先将 SP 的值减 2，指到 0FFFAH 的偏移值。
- 接着将 BX 的值拷贝至 SP 所指的数据项。

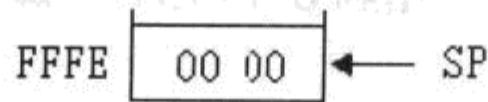
低地址存储 BX 的低字节 BL 的值，高地址存储 BX 的高字节 BH 的值，所以数据项的内容为 7856H。



- (4) 将堆栈顶端数据项弹出至 BX。
- 将堆栈顶端数据项弹出至 BX。
 - 接着将 SP 的值加 2，指到 0FFFCH 的偏移值。



- (5) 将堆栈顶端数据项弹出至 AX。
- 将堆栈顶端数据项弹出并存入 AX。
 - 接着将 SP 的值加 2，指到 0FFFEH 的偏移值。



请注意，堆栈的操作是先进后出，也可以说是后进先出，因此第一个压入的 AX 值存入堆栈顶端，第二个压入的 BX 值存入堆栈顶端，刚才的 AX 已经不是顶端了，这时您若要弹出，所弹出的是 BX 而不是 AX。有压入必须有相对应的弹出操作，否则会发生不可预知的错误。

有关堆栈操作的相关指令如下：

PUSHF 及 POPF

PUSHF 压入标志寄存器，POPF 弹出并存入标志寄存器。

PUSHA 及 POPA

PUSHA 压入所有一般用途寄存器，POPA 弹出并存入所有一般用途寄存器。一般用途寄存器指 AX、DX、BX、CX、SP、BP、SI 及 DI。

PUSHAD 及 POPAD

PUSHAD 压入所有扩展寄存器，POPAD 弹出并存入所有扩展寄存器。扩展寄存器指 EAX、ECX、EDX、EBX、ESP、EBP、ESI 及 EDI。

课后习题

1. 请说明 BIOS 的启动程序。
2. BIOS 进入点在哪里。
3. 请说明.com 程序的特点。
4. 请说明.exe 程序的特点。
5. 请说明如何使用堆栈。

新学如

PDG



1. 开发环境 2018 版
2. BIOS 设置
3. 开发环境 2018 版
4. 开发环境 2018 版
5. 开发环境 2018 版



3

NASM 汇编语言基础

NASM 汇编语言的英文名称为 NetWide Assembler, 简称 NASM, 它是专为 Intel 公司出品的 80x86 处理器 (processor) 所组成的计算机而设计的, 它是一种可移植性 (portability) 高且具有模块性 (modularity) 的一种汇编语言。NASM 支持多种目标文件格式 (object file format), 包括 Linux 操作系统的 a.out 及 ELF 格式、NetBSD/FreeBSD 格式、COFF 格式、微软公司的 16 位 OBJ 格式以及 Win32 格式等, 它也支持纯二进制文件的输出格式。

NASM 的语法类似 Intel 公司汇编语言语法, 不过较为简单易懂, 它支持 Pentium、P6 以及 MMX 机种的运算码 (opcode), 同时具备宏 (macro) 的功能。有关 NASM 的软件及参考文件可从下列网站免费下载:

<http://nasm.2y.net/>。





3-1 源程序行格式

像其他的汇编语言一样，NASM 汇编语言的源程序行 (source line) 包含 4 项组合，这 4 项分别为标号 (label)、指令 (instruction)、操作数 (operand) 以及注释 (comment)。

[标号] [指令] [操作数] [注释]

这 4 项均为可选性质，方括号表示可选。每项之间可以存放一个或多个空格。标号项由标号标识符及冒号组成。标号标识符由下列字符所组成：

_ \$ # @ ~ . ? 英文字母 数字

标号标识符第一个字符只能是英文字母、点号 (.)、下划线 (_) 或是问号 (?)。因为 NASM 对于大小写英文字母，认为是不同的标识符，因此下列三个均为不同的标号标识符。

MyLabel:

myLabel:

mylabel:

指令项包含所有机器指令 (machine instruction)，包括 Pentium 及 P6 的指令、FPU 指令、MMX 指令以及其他未公布的 (undocumented) 指令。指令前方可以伴随着 LOCK、REP、REPE/REPZ 或 REPNE/REPNZ 等。指令项除了包含真正的机器指令外，也包括 NASM 支持的伪指令 (pseudo-instruction)。

操作数项包括下列的形式：

1. 寄存器

寄存器名称，例如 AX、EBX、CL 等。

2. 有效地址

表示内存地址。

3. 常量

可为数字常量、字符常量、字符串常量或浮点数常量。

4. 表达式

表达式为寄存器、有效地址、常量以及运算符 (operator) 的结合，计算的结果应当为一个操作数。

注释项以分号开始，其后跟随着注释。

下列均为合乎语法规则的源程序：

```
label2: MOV AX, sum      ;将变量 sum 的地址存入寄存器 AX 中
        MOV BX, 12       ;将常量 12 存入寄存器 BX 中
        MUL BX           ;DX:AX=AX*BX
        CMPSB
label3:
        JMP label2
```

上例中源程序第一行包含标号项 label2:、指令项 MOV、操作数项 AX 及 sum、注释项。第二行包含指令项 MOV、操作数项 BX 及 12、以及注释项。第三行包含指令项 MUL、操作数项 BX、以及注释项。第四行包含指令项 CMPSB。第五行包含标号项 label3:。第六行包含指令项 JMP、操作数项 label2。

3-2 伪指令

伪指令 (pseudo instruction) 并非真正的机器指令, 它是指示汇编语言如何编译程序的一种指令, 它要存放在源程序的指令字段位置, 因此称它为伪指令。NASM 所支持的伪指令有:

DB、DW、DD、DQ、DT
RESB、RESW、RESQ、REST
INCBIN
EQU
TIMES

分别说明如下。

3-2-1 定义含有初值的数据

伪指令 DB 定义含有初值的字节数据 (define byte), DW 定义含有初值的字数据 (word), DD 定义含有初值的双字数据 (double word), DQ 定义含有初值的四个字数据 (define quad word), DT 定义含有初值的十个字节数据 (define ten bytes)。下列均为合乎语法的定义:

```
DB 41H ;定义字节数据, 初值为十六进制数 41
DB 31H, 32H, 33H ;初值为十六进制数 31、32、33
DB 'a', 62H ;初值为字符常量 'a' 及十六进制数 62
DB 'Hello', 13, 10, '$' ;初值为字符串常量 'Hello' 及十进制
;数 13、10 及字符常量 '$'

DW 1234H ;定义字数据, 初值为十六进制数 1234
DW 1234H, 5678H ;初值为十六进制数 1234 及 5678
DW 'a' ;初值为字符常量 'a'
DW 'ab' ;初值为字符常量 'ab'

DD 12345678H ;定义双字数据, 初值为十六进制数 12345678
DD 365.25 ;初值为十进制浮点常量 365.25
DD 3.6525E+2 ;初值为十进制浮点常量 365.25
DQ 3.6525E+2 ;定义四个字数据, 初值为十进制数 365.25
DT 3.6525E+2 ;定义十字节数据, 初值为十进制数 365.25
```

DQ 及 DT 不能接受整数数字常量及字符串常量。

3-2-2 定义不含初值的数据

RESB、RESW、RESQ 及 REST 等伪指令定义不含初值的数据, 事实上这些伪指令只保留一些内存空间而已, RES 为 reserve 的缩写, B 表示 Byte 字节, W 表示 Word 字, D 表示 Double word 双字, Q 表示 Quad word 四个字, T 表示 Ten byte 十个字节的空间。下列均为合乎语法的定义:

```
Buffer RESB 64 ;保留 64 个字节的内存空间
wordvar RESW 1 ;保留一个字的内存空间
realbuf RESQ 10 ;保留 10 个四个字的内存空间
```



3-2-3 INCBIN 伪指令

INCBIN 伪指令引用二进制文件，引用的二进制文件可以是图片文件也可以是声音文件，这对于游戏程序的设计有很大的帮助。INCBIN 伪指令的使用有下列三种格式：

```
INCBIN    "pic.dat"                ;引用全部文件
INCBIN    "pic.dat",1024           ;跳过前面 1024 个字节
INCBIN    "pic.dat",1024,512       ;跳过前面 1024 个字节
                                           ;最多引用 512 个字节
```

3-2-4 EQU 伪指令

EQU 伪指令给一个符号 (symbol) 定义一个常量，当您使用 EQU 伪指令时，源程序必须包含一个标号，这个标号就是所谓的符号。这个符号一经定义后就不能改变了。例如：

```
message    DB    'Hello, world'
msglen     EQU    $-message
```

\$ 符号表示这一源程序开始的地址，它减去 message 的地址后，得到的值为 12，它表示 message 这个字符串的长度，在这个程序中 msglen 的值永远为 12，不能改变。

3-2-5 TIMES 伪指令

TIMES 伪指令的格式如下：

TIMES 次数 指令

TIMES 伪指令将随后的“指令”执行指定的“次数”。例如：

```
zerobuf:   TIMES    64    DB    0
```

定义 64 个字节，每个字节的内容 (content) 为 0。次数可以是常量，也可以使用表达式 (expression)。

```
buffer: DB    'Hello, world'
        TIMES  64-$+buffer    DB    ' '
```

定义一个 64 个字节的空间，前面的 11 个字节的内容为“Hello,world”，其他 53 个字节的内容为空。

```
label2: TIMES  100 MOVSB
```

将 MOVSB 指令重复 100 次。TIMES 伪指令不能使用在 MACRO (宏) 里。

3-3 有效地址

有效地址 (effective address) 为指令的一个操作数，用于映射到一个内存地址。对于 NASM 来讲，有效地址的计算非常简单，计算在方括号内的表达式，结果就是一个有效地址。例如：

```
var    DB    'a',    'b',    'c',    'd'
buf    DB    '1',    '2',    '3',    '4'
MOV    AL,[var+1]
MOV    [buf],AL
```

第三行计算有效地址 `var+1`, 方括号表示该地址的内容为字符'b', 将它存入寄存器 AL 中, 接着将它存入内存 `buf` 地址, 因此 `buf` 的值改变为'b234'。

在下例中 `var` 定义为字, 第一个字的初值为'ab', 第二个字的初值为'cd', 第三行 `MOV` 指令计算有效地址 `var+2`, 方括号表示该地址的内容, 为字符'cd', 将它存入寄存器 AX 中, 接着将它存入地址为 `buf` 的内存, 因此 `buf` 的值变为'cd34'。

```
Var    DW    'ab',    'cd'
Buf    DB    '1',     '2',     '3',     '4'
        MOV  AX, [var+2]
        MOV  [buf], AX
```

上例中若将有效地址 `var+2` 改为 `var+1`, 那么 `buf` 的值变为 bc34。

3-4 常量

NASM 支持四种不同的常量: 数字常量(numeric constant)、字符常量(character constant)、字符串常量(string constant)以及浮点数常量(floating point constant)。

3-4-1 数字常量

数字常量简单地讲就是数字, NASM 支持多种基数(base)的数字常量, 在数字的后头附上 H 或 h 表示该数字为十六进制数, 附上 Q 或 q 表示该数字为八进制数, 附上 B 或 b 表示该数字为二进制数, 若没有附上则表示十进制数。NASM 也支持数字前头附上 0X 或 0x 或 \$ 表示该数字为十六进制数。数字常量的用法如下例所示:

```
MOV AX, 100           ;100 为十进制数常量
MOV AX, 0a2H          ;a2 为十六进制数字常量
MOV AX, 0xa2          ;a2 为十六进制数字常量
MOV AX, $0a2          ;a2 为十六进制数字常量
MOV AX, 177777Q       ;177777 为八进制数字常量
MOV AX, 101101101B    ;101101101 为二进制数常量
```

3-4-2 字符常量

字符常量最多包括四个字符, 可使用单引号或双引号将它们括起来。使用单引号或双引号对于 NASM 都表示同一个意思, 它的好处是当您使用单引号时, 双引号就可以当数据字符来使用, 同理当您使用双引号时, 单引号就可以当数据字符来使用。字符常量若超过一个字符时, 要注意前面的字符放在低地址, 后面的字符放在高地址, 例如您写下列的程序代码:

```
MOV AX, 'ab'
```

那么您存入寄存器 AX 的是 0x6261, 也就是'ba' 而不是'ab'。

3-4-3 字符串常量

字符串常量只适用于 `INCBIN` 指令以及与 `DB` 类似的伪指令。字符串常量看起来像字符常量, 只不过长一点而已。



```
DB 'Hello' ;字符串常量
DB 'H', 'e', 'l', 'l', 'o' ;字符常量相当于上例字符串常量
```

下列三个指令也是相当的:

```
DD 'ninechars' ;双字字符串常量
DD 'nine', 'char', 's' ;相当于三个双字字符常量
DB 'ninechars', 0,0,0 ;相当于12个字节
```

下列指令相当于双字字符串常量:

```
DW 'abcd' ;相当于双字字符串常量
```

3-4-4 浮点数常量

浮点数常量只适用于伪指令 DD、DQ 及 DT。它们以传统的方式表示出来,它的格式如下:

数字. [数字] [[E | e] [+ | -] 指数]

方括号表示可选。例如:

```
DD 365. ;浮点数常量 365.0
DD 365.25 ;浮点数常量 365.25
DD 3.6525e2 ;浮点数常量 365.25
DD 3.6525e+2 ;浮点数常量 365.25
DD 1.e-6 ;浮点数常量 0.000001
DD 3.141592653589793238462 ;浮点数常量 pi
```

3-5 表达式

NASM 的表达式语法类似 ANSI C 语言的语法。NASM 的表达式支持两个特殊的符号: \$ 及 \$\$。\$ 表示包含此表达式源程序开始的地址。而 \$\$ 表示当前的段地址 (current segment)。\$ - \$\$ 表示包含此表达式源程序开始的地址离开该段的距离 (字节数)。

NASM 所提供的算术运算符 (arithmetic operator) 依优先级由低至高说明如下:

3-5-1 OR 运算符

运算符 | 与指令 OR 的作用相同。

3-5-2 XOR 运算符

运算符 ^ 与指令 XOR 的作用相同。

3-5-3 AND 运算符

运算符 & 与指令 AND 的作用相同。

3-5-4 移位运算符

运算符 << 表示位往左移位, >> 表示位往右移位。

3-5-5 加及减运算符

加+ 运算符表示算术加法，减- 运算符表示算术减法。

3-5-6 乘及除运算符

乘* 运算符表示乘法，/ 运算符表示无符号整数除法。//运算符表示有符号整数除法，余数% 运算符表示取无符号整数除法的余数，余数%% 运算符表示取有符号整数除法的余数。

3-5-7 单元运算符

单元运算符有+、-、~及 SEG。单元运算符+ 没有作用，单元运算符- 表示将操作数的符号相反，正数变成负数，负数变成正数。单元运算符~ 表示取操作数的反码。单元运算符 SEG 表示取操作数的段地址。

3-6 临界表达式

NASM 的一个限制就是它是一个二次汇编器 (two-pass assembler)。它一直都严格地执行二次汇编，您要它去执行三或四次汇编，它就没办法了。因此在第一次汇编当中，必须能够决定所有数据及程序代码的长度（字节数），第二次汇编才能产生相对应的机器码。所以 NASM 无法处理必须在其后的运算才能决定长度的数据或程序代码。例如：

```
TIMES    (label-$) DB 0
label: DB    'Where am I?'
```

在 TIMES 的源程序并无法决定 label-\$ 的值，因此 NASM 会拒绝这样的写法。解决这个问题的最好的方法就是将 label 定义在 TIMES 的前面，例如：

```
label: DB    'Where am I?'
TIMES    ($-label) DB 0
```

在第一次汇编当中，能够决定所有数据及程序代码的长度的表达式称为临界表达式(critical expression)。TIMES 的操作数属于临界表达式，EQU 的操作数也是属于临界表达式。

3-7 局部标号

以点号“.”开头的符号 NASM 会作特殊处理，将它视为局部标号 (local label)，必须与非局部标号结合才可当成独立的标号。例如：

```
label2:                ;某些程序代码
.loop                 ;某些程序代码
    JNE .loop          ;事实上跳至 label2.loop
    RET
label3:                ;某些程序代码
.loop                 ;某些程序代码
    JNE .loop          ;事实上跳至 label3.loop
    RET
```



上例的程序代码中, 两个局部标号均命名为`.loop`, 但它的位置并不相同, 上一个局部标号`.loop` 与它的前一个非局部标号 `label2` 相结合成 `label2.loop`。

下一个局部标号`.loop` 与它的前一个非局部标号 `label3` 相结合成 `label3.loop`。因此上一个 `JNE` 指令实际上可写成 `JNE label2.loop`, 下一个 `JNE` 指令实际上可写成 `JNE label3.loop`,

3-8 预处理器

所有的预处理器指引指令 (preprocessor directive) 均以`%` 开头。包括单行宏 (single-line macro)、多行宏 (multi-line macro)、条件汇编 (conditional assembly)、预处理循环 (preprocessor loop)、引用文件 (including file)、标准宏 (standard macro) 等。

单行宏包括`%define`、`%undef` 及`%assign` 指令, 说明如下。

3-8-1 `%define` 指令

单行宏以预处理器指引指令`%define` 来定义。例如:

```
%define sum(a,b)    a+b
```

```
MOV DL,    sum(64,1) ;将 64+1 的结果存入 DL
```

上例中定义宏 `sum (a,b)`, 在 `MOV` 指令中 `a` 参数以 `64` 取代, `b` 参数以 `1` 取代, 因此 `sum (64,1)` 宏的展开为 `64 + 1`, 因此 `MOV` 的指令为:

```
MOV DL, 64+1    ;将 DL 的值以字符输出为 'A'
```

单行宏也可以定义没有参数的宏, 例如:

```
%define pi 3.14159
```

定义符号 `pi` 为 `3.14159`。

`%define` 指令所定义的宏, 其名称英文字母大小写是有区别的, 例如 `pi`、`PI`、`Pi`、`pI` 等均为不同的宏名称。您若想宏名称英文字母大小写没有区别, 您就要使用`%ifndef` 指令, 第一个字母 `i` 是 `ignore` 忽略的意思。

3-8-2 `%undef` 指令

`%undef` 指令取消`%define` 所定义的宏, 例如:

```
%define pi 3.14159    ;定义 pi 符号
```

```
...                  ;其他程序代码
```

```
%undef pi            ;取消%define 所定义的宏 pi
```

3-8-3 `%assign` 指令

`%assign` 指令定义一个单行宏, 没有参数但有一个变量值。这个数值可以用表达式方式表示, 当`%assign` 指令被执行时记一次值。`%assign` 所定义的变量可以重新定义, 例如:

```
%assign i 0          ;指定变量 i 初值为 0
```

```
%rep 3               ;循环开始 (循环三次)
```

```
    DB i              ;定义一个字节初值为变量 I 的值
```

```
    %assign i i+1      ;指定变量 i 的值为 (i+1)
```

```
%endrep          ; 循环结束
```

上例中首先指定变量 *i* 初值为 0，接着循环三次，每一次定义一个字节初值为变量 *i* 的值后，将指定变量 *i* 的值为 (*i*+1)，然后又循环，因此 `DB i` 共执行三次，第一次时 *i* 值为 0，第二次时 *i* 值为 1，第三次时 *i* 值为 2，结果相当于下列的三个指令：

```
DB 0      ;第一次循环
DB 1      ;第二次循环
DB 2      ;第三次循环
```

3-8-4 多行宏

多行宏的格式如下：

```
%MACRO 宏名称  参数个数
      宏本体
```

```
%ENDMACRO
```

例如：

```
%MACRO readchr 1          ;宏名称 readchr, 一个参数
    PUSH    AX             ;存入 AX
    MOV     AH, 01H        ;从键盘读取一个字符至寄存器 AL
    INT     21H
    MOV     [%1], AL       ;将 AL 值存入参数所指地址
    POP     AX             ;弹出 AX
%ENDMACRO                 ;宏结束
```

上例宏名称为 `readchr`，只有一个参数。首先存入 `AX`，表示要将现在的值保存在堆栈的顶端，将 `01H` 存入 `AH` 后调用 `21H` 号中断，表示要从键盘读取一个字符至寄存器 `AL`，然后将 `AL` 值存入第一个参数所指地址，最后从堆栈顶端弹出原来的 `AX` 值后才返回调用此宏的程序。这个宏因为在往后的程序中会用到，因此将它存入磁盘，文件名为：

```
\howasm\mymacro\readchr.mac.
```

调用的程序可作如下的设计：

```
ch2      DB      ' '
        ...
        readchr ch2
```

调用 `readchr` 宏从键盘读取一个字符至内存变量 `ch2` 处。例如您输入 'C' 那么 `ch2` 变量的值就变为 'C' 了。

3-8-5 条件汇编

条件汇编的格式常用的有下列两个：

格式一：

```
%ifdef 符号
```

只在符号有定义时才被汇编的程序代码。

```
%endif
```



格式二:

%if expr

只在表达式 **expr** 的值为非零时才被汇编的程序代码。

%endif

例如:

```
%ifndef READCHR_MAC
%define READCHR_MAC
%MACRO readchr 1                                ;宏名称 readchr, 一个参数
    PUSH AX                                      ;存入 AX
    MOV AH, 01H                                ;从键盘读取一个字符至寄存器 AL
    INT 21H
    MOV [%1], AL                                ;将 AL 值存入参数所指地址
    POP AX                                      ;弹出 AX
%ENDMACRO                                         ;宏结束
%endif
```

上例表示若符号 **READCHR_MAC** 还没有定义, 就定义 **READCHR_MAC**, 然后定义一个宏 **readchr**, 若 **READCHR_MAC** 已经有定义则不作任何事情。这样的安排可保证您的程序里最多保存一份 **readchr** 宏。

```
%if 0
    MOV AX, 0
```

```
%endif
```

上例中 **MOV** 指令不被汇编。

3-8-6 预处理循环

NASM 的 **TIMES** 指令虽然功能强大, 但却不能多次执行多行宏, 因此又提供一个 **%rep** 预处理循环指引。它以 **%rep** 开始, 以 **%endrep** 结束, 它的语法如下:

```
%rep    循环次数
        循环指令
```

```
%endrep
```

例如:

```
%assign i 0                                ;指定变量 i 初始值为 0
%rep 3                                       ;循环开始(循环三次)
    DW i                                    ;定义一个字初值为变量 i 的值
    %assign i i+1                           ;指定变量 i 的值为 (i+1)
%endrep                                     ;循环结束
```

上例中首先指定变量 **i** 初始值为 0, 接着循环三次, 每一次定义一个初始值为变量 **i** 的值后, 将指定变量 **i** 的值为 **(i+1)**, 然后又循环, 因此 **DW i** 共执行三次, 第一次时 **i** 值为 0, 第二次时 **i** 值为 1, 第三次时 **i** 值为 2, 结果相当于下列的三个指令:

```
DW 0                                       ;第一次循环
DW 1                                       ;第二次循环
```

3-8-7 文件引用指引

文件引用指引 (include directive) 为您提供一个方法将其他的源程序文件引用至您的程序中, 标明引用文件指引之处。其格式如下:

```
%include 引用文件名
```

例如您想将 readchr.mac 引用至您的程序中, 其指令如下:

```
%include "c:\howasm\mymacro\readchr.mac"
```

;您的其他程序指令

3-8-8 标准宏

标准宏是 NASM 已经定义完成的宏, 我们无须再定义就可直接使用。宏 `_NASM_MAJOR_` 表示 NASM 汇编语言的主要版本编号, `_NASM_MINOR_` 表示 NASM 汇编语言的次要版本编号, `_FILE_` 表示当前指令的源程序文件名, `_LINE_` 表示当前指令的行编号。

3-8-9 汇编语言指引

NASM 汇编语言的指引只有少数几个而已。NASM 缺省的汇编语言指引均以方括号括起来, 这里只介绍常用的两个汇编语言指引 BITS 及 SECTION。

BITS 汇编语言指引表示目的处理器的模式 (mode), 包括 16 位模式以及 32 位模式。大多数的情况下您并不需要很明白地指出它的模式, 对于 aout、coff、elf 以及 win32 的目标文件格式 (object format) 而言, 它们是使用 32 位的操作系统, 因此 NASM 将它们默认为 32 位的模式, 对于 obj 的目标文件格式而言, 您可自由选用 USE16 或 USE32, NASM 会选择适当的模式, 因此您也不需要很明白地指出它的模式。最需要您明白地指出它模式的就是您写 32 位的程序代码以产生纯二进制文件 (flat binary file), 那是因为 bin 的目标文件格式, 它们是使用 16 位的操作系统, 因此 NASM 将它们默认为 16 位的模式, 它们常用于编写 DOS 的 .COM 程序、.SYS 系统程序以及 DOS 的启动软件。不要只因为要使用 32 位的指令就设定为 32 位模式, 这样会产生 32 位的程序代码在 DOS 系统执行反而无法运行程序。BITS 汇编语言指引的模式定义如下:

```
[BITS 16] ;16 位模式
```

```
[BITS 32] ;32 位模式
```

SECTION 汇编语言指引标明程序代码汇编时所需的段地址, 对于某些目标文件格式, 段的名称及数目是固定的, 对于另外的目标文件格式, 段的名称及数目是要您自己设定的, 您若设定不存在的段地址就会产生错误。对于 UNIX 及 bin 目标文件格式而言, NASM 支持三个标准的段名: .text、.data 以及 .bss。 .text 是程序代码段, .data 是数据段, .bss 则是没有初始值的数据段。对于 obj 目标文件格式而言, NASM 并不认识这三个段名, 段名必须您自己去设定。SEGMENT 与 SECTION 同义。SECTION 汇编语言指引的定义如下:

```
[SECTION .text]
```

```
[SECTION .data]
```

```
[SECTION .bss]
```



3-9 目标文件格式

NASM 是一个跨平台的汇编语言，汇编后的目标文件可在支持 ANSI C 语言的平台上运行，当然那个平台的处理器必须是 Intel 公司的 80x86 系列的才行。为了达到这个目的，它支持许多的目标文件格式，您可在汇编时使用 -f 选项选定目标文件格式，为了简单起见，本小节只介绍 bin 纯二进制输出格式。

bin 格式并不产生目标文件 (object file)，它只产生与您的程序指令一样的机器码文件，这种格式用于 MS-DOS 的 .COM 执行文件以及 .SYS 设备驱动程序。bin 格式支持三个标准的段名 .text、.data 及 .bss。输出的执行文件首先引用 .text 段的指令码，然后跟随着 .data 段的数据码，段与段间以四个字节对齐 (four-byte boudary)，并不引用 .bss 段的数据。若您不标明 SECTION，那么您的程序代码默认为使用 .text 段，若您不标明 BITS，那么您的程序代码模式默认为 16 位模式。

bin 格式额外提供一个 ORG 指引，它告诉 NASM 目标程序执行时的起始地址，对于 .COM 执行文件而言，前面的 256 个字节必须保留给 PSP 程序段，您的程序代码必须从第 256 个字节开始存放，您可以使用 ORG 指令达到这个目的。

```
ORG 0100H          ;十六进制数 0100 相当十进制数 256
```

3-10 NASM 汇编程序安装

有关 NASM 的软件及参考文件可从下列网站免费下载：

<http://nasm.2y.net/>。

您可下载最新的版本。例如：

文件名：nasmw-0.98-08.exe

大小：244,152

日期：2001/11/21

经运行后自动解压缩，存入 C:\NASM 目录。

您所下载的文件：

文件名：nasmdoch.zip

大小：155,723

经解压缩后存入 C:\NASM\DOC 目录。

在自动运行文件 autoexec.bat 中加入 nasmw.exe 汇编程序的路径：

```
PATH c:\nasm;
```

这样您在任何的目录下均可运行 nasmw.exe 汇编程序。

3-11 范例

程序 hello.asm 中，每个源程序前面都附有编号，是为了说明方便才加上去的，程序本身是没有编号的，以后的程序编号也都是为了说明方便才加上去的。

程序 hello.asm 第一行是注释，说明程序文件名称为 hello.asm，程序写作的风格每个人

有所不同,本书的风格是程序第一行都标明文件名。第二行是 NASM 汇编语言指令,说明要产生 16 位的机器码。第三行也是 NASM 汇编语言指令,说明加载内存运行时要预留 256 个字节的内存存放 PSP 程序段。第四行 JMP 指令跳至 start 标号。第五行定义一个字符串,开头的地址为 msg。第六行将 msg 的地址存入寄存器 DX 中。第七行将整数常量 9 存入寄存器 AH 中。第八行调用 21H 号中断,执行 AH 寄存器内容 9 的功能,此功能会将 DX 寄存器所指的字符串显示于屏幕。第九行返回操作系统。要运行这个程序首先必须将它进行汇编:

```
howasm\ch03> nasmw hello.asm -o hello.com <Enter>
```

表示在 howasm 目录下的 ch03 子目录之下,将源程序文件 hello.asm 用汇编程序 nasmw 进行汇编,产生目标程序文件名为 hello.com,因为产生.COM 文件,因此 NASM 就产生一个 bin 格式的纯二进制机器码文件。

要运行这个程序只需输入.COM 文件名就可以了:

```
howasm\ch03> hello <Enter>
```

运行的结果:

Hello!

【程序 hello.asm】

```
1 ; ***** hello.asm *****
2 [BITS 16]
3 [ORG 0x0100]
4     JMP     start
5 msg  DB 'Hello! ',13,10,'$'
6 start:  MOV  DX, msg
7         MOV  AH, 9
8         INT  21H
9         RET
```

【运行】

```
howasm\ch03> nasmw hello.asm -o hello.com <Enter>
```

```
howasm\ch03> hello <Enter>
```

Hello!

程序 hello2.asm 与程序 hello.asm 相同,只不过将数据与程序代码分置于不同的分段而已。程序代码放置于.text 段,数据放置于.data 段。

【程序 hello2.asm】

```
; ***** hello2.asm *****
[BITS 16]
[ORG 0x0100]
[SECTION .text]
    MOV  DX, msg
    MOV  AH, 9
    INT  21H
```



```
RET
```

```
[SECTION .data]
```

```
msg DB 'Hello!',13,10,'$'
```

【运行】

```
howasm\ch03> nasmw hello2.asm -o hello2.com<Enter>
```

```
howasm\ch03> hello2 <Enter>
```

```
Hello!
```

程序 hello3.asm 与程序 hello.asm 相同,只不过将数据与程序代码同置于程序代码.text 段。

【程序 hello3.asm】

```
; ***** hello3.asm *****
```

```
ORG 0100H
```

```
JMP start
```

```
msg DB 'Hello!',13,10,'$'
```

```
start: MOV DX, msg
```

```
MOV AH, 9
```

```
INT 21H
```

```
RET
```

【运行】

```
howasm\ch03> nasmw hello3.asm -o hello3.com <Enter>
```

```
howasm\ch03> hello3 <Enter>
```

```
Hello!
```

您若想要知道 NASM 汇编后的机器码,您可使用-l 选项,输出至指定列表文件,例如您汇编 hello3.asm,指定 hello3.lst 列表文件,如下:

```
howasm\ch03> nasmw hello3.asm -l hello3.lst <Enter>
```

文件 hello3.lst 第三行地址从 0 开始产生机器码 E90900,相当于 NASM 汇编语言指令 JMP start。

【文件 hello3.lst】

```
1 ; ***** hello3.asm *****
2
3 00000000 E90900          JMP start
4 00000003 48656C6C6F210D0A24 msg DB 'Hello!',13,10,'$'
5 0000000C BA[0300]      start:  MOV DX, msg
6 0000000F B409          MOV AH, 9
7 00000011 CD21          INT 21H
8 00000013 C3            RET
```

您若想看一看 hello3.com 二进制文件的内容,您可使用 dumpfile.exe 将它以十六进制的方式显示出来,如下所示。您可以看出它就是 hello3.lst 中的机器码部分。

【文件 hello3.com】

```
E9 09 00 48 65 6C 6C 6F 21 0D 0A 24 BA 03 01 B4  
09 CD 21 C3
```

课后习题

1. 请说明数字常量后面跟随着 B、Q、H、b、q、h 各代表什么意义。
2. 请说明数字常量没跟随着 B、Q、H、b、q、h 等字符代表什么意义。
3. 请举例说明 DB、DW、DD、DQ 及 DT 的用法。
4. 请举例说明什么是局部标号。
5. 请举例说明什么是单行宏。
6. 请举例说明什么是多行宏。
7. 请举例说明什么是条件汇编。
8. 请举例说明如何使用引用文件。

【文件 hello3.com】

1. 在“文件”菜单中选择“打开”命令，打开“文件”对话框。



2. 在“文件”对话框中，选择“文件”列表中的“文件”。

3. 单击“打开”按钮，打开“文件”。

4. 在“文件”对话框中，选择“文件”列表中的“文件”。

5. 单击“打开”按钮，打开“文件”。

6. 在“文件”对话框中，选择“文件”列表中的“文件”。

7. 单击“打开”按钮，打开“文件”。

知识扩展
PDG

4

一般指令

本章说明一些一般性的汇编语言指令，包括寄存器间数据的拷贝，寄存器与内存间数据的传递，以及寄存器与堆栈间数据的传送。





4-1 源操作数与目的操作数

源操作数 (source operand) 指明数据的来源地址, 目的操作数 (object operand) 指明接收数据的地址。有些指令只有一个操作数, 例如下列的压入指令:

```
PUSH AX
```

这个操作数 AX 为源操作数, 表示要将寄存器 AX 的内容压入堆栈顶端。

```
POP AX
```

这个操作数 AX 为目的操作数, 表示要将堆栈顶端的数据项弹出后存入寄存器 AX 中。

许多指令都含有两个操作数, 操作数之间以逗号隔开, 例如下列的数据传送指令:

```
MOV AX, table ; 传送 table 偏移地址至 AX
```

表示将数据从一个操作数传送到另一个操作数, 在这种情况下, 提供数据的操作数称为源操作数, 接收数据的操作数称为目的操作数, 一般而言源操作数的数据会保持不变, 而目的操作数原来的数据会被源操作数的数据取代。

若指令需要您标明源操作数及目的操作数时, 永远是先写上目的操作数, 再写逗号, 再写源操作数。在上例中 AX 为目的操作数, table 为源操作数。

请注意这个 MOV 指令传送的是 table 的偏移地址, 而不是 table 的内容, 您若要传送 table 的内容至 AX, 那么 table 要用方括号包括起来, 如下:

```
MOV AX, [table] ; 传送 table 内容至 AX
```

下例传送寄存器 CX 内容至 AX 寄存器中, AX 为目的操作数, CX 为源操作数。CX 值不会改变, 但 AX 的值会被 CX 的值取代。

```
MOV AX, CX ; 传送 CX 内容至 AX
```

4-2 MOV 传送指令

MOV 传送指令将数据从一个地方传送到另一个地方, MOV 是 move 传送的意思, 应该叫 copy 拷贝较恰当。MOV 传送指令是汇编语言中最常用到的一个重要指令, 它的格式如下:

MOV 目的操作数, 源操作数

MOV 指令将右边的源操作数数据传送到左边的目的操作数。源操作数的数据在执行 MOV 指令后并不会改变, 但目的操作数的数据会变成源操作数的数据。目的操作数可以是寄存器或内存地址, 源操作数可为寄存器、内存地址或立即数 (immediate value)。立即数就是常量的意思。

```
total DW 0 ; total 字初值为 0
MOV AX, 0 ; 将 0 传送到 AX
MOV CX, 20H ; 将十六进制数 20 传送到 CX
MOV [total], WORD 2*33 ; 将 2*33 存入内存 total 处
```

上例中 0、20H 及 2*33 均为立即数。传送后寄存器 AX 的内容为 0 值, CX 的内容为 20H, total 的内容为 66。

```
total DW 0 ; total 字初值为 0
```

```

MOV BX, SI      ;将寄存器 SI 值传送到 BX
MOV ES, AX      ;将寄存器 AX 值传送到 ES
MOV [total], DS ;将寄存器 DS 值传送到内存 total 处

```

上例中分别将寄存器值传送到寄存器及内存地址。第一个 MOV 指令将寄存器 SI 值传送到寄存器 BX，第二个 MOV 指令将寄存器 AX 值传送到寄存器 ES，第三个 MOV 指令将寄存器 DS 值传送到内存 total 处。

```

total DW 0      ;total 字初值为 0
list  DW 1, 3, 5, 7 ;list 字初值分别为 1,3,5,7
MOV AX, [total] ;将内存 total 内容传送到 AX
MOV BX, list    ;将内存 list 偏移地址传送到 BX
MOV DX, [BX+2]  ;将内存 BX+2 处的内容传送到 DX

```

上例中分别将内存地址值或内容传送到寄存器。第一个 MOV 指令将内存 total 内容传送到 AX 寄存器，第二个 MOV 指令将内存 list 偏移地址传送到 BX，第三个 MOV 指令将内存 BX+2 处一个字的内容传送到 DX。传送后寄存器 AX 的内容为 0 值，DX 的内容为 3。

在使用 MOV 指令时有一些限制不可不知。首先目的操作数及源操作数的数据长度必须一致，例如您不能传送一个字节的数据至字里。

```

MOV DX, BL      ;错误, 传送一个字节的数据 BL 至字 DX
MOV DL, AX      ;错误, 传送一个字的数据 AX 至字节 DL

```

第二个限制是您不能使用寄存器 IP 当目的操作数或源操作数，因为 IP 所存的数据是下一个指令的地址，不能让您改变它的值。第三个限制是您不能使用寄存器 CS 当目的操作数，因为 CS 所存的数据是指令的分段起始地址，不能让您改变它的值。

```

MOV CS, 0100H   ;错误, CS 不能当目的操作数
MOV AX, IP      ;错误, IP 不能当源操作数

```

第四个限制是您不能将立即数或寄存器值传送到分段寄存器。

```

MOV SS, 0100H   ;错误, 不能将立即数传送到分段寄存器 SS
MOV ES, DS      ;错误, 不能将寄存器 DS 值传送到分段寄存器 ES
PUSH DS         ;正确, 先将 DS 寄存器值压入堆栈顶端
POP ES          ;正确, 然后再从堆栈顶端弹出并存入段寄存器

```

第五个限制是您不能将内存地址直接传送到分段寄存器，您必须先将内存地址传送到一般寄存器，例如 AX，然后再从寄存器传送到分段寄存器。

```

MOV DS, total    ;错误, 不能将内存地址直接传送到段寄存器
MOV AX, total     ;正确, 先将内存地址传送到一般寄存器
MOV DS, AX       ;正确, 然后再从寄存器传送到段寄存器

```

第六个限制是您不能将内存偏移地址或内容直接传送到另一个内存地址，您必须先将内存偏移地址传送到一般寄存器，然后再从寄存器传送到另一个内存地址，例如 AX。

```

MOV [list], [total] ;错误, 不能将内存 total 的内容
                    ;直接传送到另一个内存 list
MOV AX, [total]     ;正确, 先将内存 total 内容传送到 AX
MOV [list], AX      ;正确, 再从 AX 传送到另一个内存 list

```

4-3 XCHG 互换指令

XCHG 指令互换两个寄存器之间的数据，或一个寄存器及一个内存地址之间的数据。例如下列四个指令：

```
XCHG AX, BX           ; 互换 AX 及 BX 寄存器的内容
XCHG AX, [total]      ; 互换 AX 及内存 total 处的内容
XCHG AX, [list+2]     ; 互换 AX 及内存 [list+2] 地址的内容
XCHG AH, AL           ; 互换 AH 及 AL 的内容
```

上例中第一个 XCHG 指令互换 AX 寄存器及 BX 寄存器的数据。第二个 XCHG 指令互换 AX 寄存器及内存地址 total 处的内容。第三个 XCHG 指令互换 AX 寄存器及内存 [list+2] 地址的内容。第四个 XCHG 指令互换 AH 及 AL 寄存器的内容。

您使用 XCHG 指令时要注意三个限制。第一个限制是目的操作数和源操作数的数据长度必须一致，例如您不能互换一个字节的数据及字的数据。

```
XCHG AX, DL           ; 错误，不能互换字及字节的数据
XCHG AL, WORD [total] ; 错误，不能互换字节及字数据
```

第二个限制是您不能互换分段寄存器中的数据。

```
XCHG ES, AX           ; 错误，不能互换分段寄存器 ES 中的数据
```

第三个限制是您不能直接互换两个内存地址间的数据。

```
XCHG [total], [list]  ; 错误，不能直接互换内存间的数据
```

您必须使用一个工作寄存器，例如 AX：

```
MOV AX, [total]        ; 将 total 内容传送到 AX
XCHG AX, [list]        ; 互换 AX 及内存 list 的内容
MOV [total], AX        ; 将 AX 内容传送到 total 处
```

4-4 有效地址送寄存器指令 LEA

LEA (Load Effective Address) 有效地址送寄存器指令将源操作数的地址传送到目的寄存器，这里所说的地址事实上是指相对于段地址的偏移量。

```
LEA AX, total          ; 将 total 的偏移地址传送到 AX
```

这个指令对于 NASM 而言与下列的 MOV 指令没什么区别：

```
MOV AX, total          ; 将 total 的偏移地址传送到 AX
```

4-5 指针送寄存器指令 LDS 及 LES 指令

LDS (Load Data Segment) 指针送寄存器指令，LES (Load Extra Segment) 指针送寄存器指令，这两个指令的用途只是将一个完整的地址加载至分段地址寄存器及偏移寄存器而已，LDS 加载至 DS:DX，而 LES 则加载至 ES:DI 寄存器对 (pair)。

LDS 指令需要两个操作数，一个寄存器及一个双字的地址。例如下例中，list 变量的值分别为 1、3、5 及 7。listptr 为 list 地址的指针，也就是说它的内容是一个地址，这个内容就是 list 变量的地址，LDS 指令将 list 的分段地址存入 DS，将 list 的偏移地

址存入 DX 寄存器。

```
list      DW 1, 3, 5, 7      ;list 字初始值分别为 1,3,5,7
listptr   DD list            ;listptr 内容为 list 的地址
          LDS DX, listptr     ;listptr 的段址及偏移分别加载
                               ;段寄存器 DS 及偏移寄存器 DX
                               ;DS:DX 含 listptr 绝对地址
```

相当如下的两个指令：

```
LEA DX, list      ;将 list 的偏移值存入 DX 寄存器
MOV DS, SEG list   ;将 list 的段址存入 DS 寄存器
```

LES 指令与 LDS 指令类似，也需要两个操作数，一个寄存器及一个双字的地址。与 LDS 指令不同的是它使用的寄存器配对为 ES:DI 而已。

```
list      DW      1, 3, 5, 7
listptr   DD      list
          LES     DI, listptr
```

相当如下的两个指令：

```
LEA DI, list      ;将 list 的偏移值存入 DI 寄存器
MOV ES, SEG list   ;将 list 的段地址存入 ES 寄存器
```

您可能奇怪为什么会用到这种指令，它的主要用途为了配合 MOVSB 及 MOVSW 指令的使用。因为您在使用到这两个指令时，您事先必须将来源及目的操作数的地址加载至 DS:DX 及 ES:DI 寄存器配对中，然后才执行 MOVSB 或 MOVSW 指令。

4-6 压入 PUSH 及弹出 POP 指令

在许多情况下您会使用堆栈，一些过度性的数据可以暂时存放在堆栈结构里面，不用的时候可以将它删除。使用堆栈的另一个重要原因是程序调用时可将参数 (parameter) 通过堆栈互相传递。堆栈结构属于先进后出，或后进先出，所有的进出都发生在堆栈的顶端。将数据存入堆栈的操作称为压入 (PUSH)，将数据从堆栈顶端移出的操作称为弹出 (POP)。

PUSH 指令的格式如下：

PUSH 操作数

操作数可为寄存器、内存地址或立即数。下列是 PUSH 指令的 7 种用法：

```
PUSH AX          ;压入 16 位的寄存器 AX
PUSH EAX         ;压入 32 位的寄存器 EAX
PUSH WORD [total] ;压入 16 位的内存 total 内容
PUSH DWORD [total] ;压入 32 位的内存 total 内容
PUSH BYTE 12     ;压入 8 位的立即数 12
PUSH WORD 1234H   ;压入 16 位的立即数 1234H
PUSH DWORD 12345678H ;压入 32 位的立即数 12345678H
```

POP 指令的格式如下：

POP 操作数

操作数可为寄存器或内存地址。例如下列是 POP 指令的三种用法：

```
POP AX          ;弹出顶端数据项并存入 AX
```



POP DS ;弹出顶端数据项并存入 DS
POP WORD [total] ;弹出顶端数据项并存入内存 total 处

4-7 存储寄存器 PUSHA 及 POPA 指令

当您在设计程序时常会用到寄存器，但又想将原来寄存器的值保留起来，等到您要运行的程序指令运行完毕后，才恢复寄存器原来的值，这时您可使用一个 PUSHA 的指令就可将所有一般用途寄存器的值存储在堆栈里头，当您想恢复寄存器原来的值，这时您可使用 POPA 指令就可以了。

PUSHA 指令群的语法如下：

PUSHA
PUSHAD
PUSHAW

PUSHAW 分别存入一般用途寄存器 AX、CX、DX、BX、SP、BP、SI 及 DI，然后将堆栈指针值减去 16。PUSHAD 分别压入一般用途寄存器 EAX、ECX、EDX、EBX、ESP、EBP、ESI 及 EDI，然后将堆栈指针值减去 32。在这两种情况下，SP 或 ESP 是执行 PUSHAW 或执行 PUSHAD 指令前的值。您若单纯地使用 PUSHA 则依 BITS 所设定的为准，若设定 [BITS16] 则执行 PUSHAW，若设定 [BITS 32] 则执行 PUSHAD。

POPA 指令群的语法如下：

POPA
POPAD
POPAW

POPAW 分别弹出一般用途寄存器 DI、SI、BP、无、BX、DX、CX 及 AX，然后将堆栈指针值加上 16。POPAD 分别弹出一般用途寄存器 EDI、ESI、EBP、无、EBX、EDX、ECX 及 EAX，然后将堆栈指针值加上 32。在这两种情况下 SP 或 ESP 是执行 POPAW 或执行 POPAD 指令前的值。您若单纯地使用 POPA 则依 BITS 所设定的为准，若设定 [BITS 16] 则执行 POPAW，若设定 [BITS 32] 则执行 POPAD。

4-8 标志寄存器传送 PUSHF 及 POPF 指令

标志寄存器 (flag register) 是一个特殊的寄存器，每一个位都可当成一个数据域使用，每一栏的意义及用法已经在第 1 章介绍过。这里您只需了解如何存储它及如何恢复它的值。要存储它可使用 PUSHF 指令将标志寄存器的值存入堆栈顶端，使用 POPF 可从堆栈顶端弹出而恢复原来的值。

PUSHF 指令群的语法如下：

PUSHF
PUSHFD
PUSHFW

PUSHFW 压入 16 位标志寄存器至堆栈顶端，然后将堆栈指针值减去二。PUSHFD 压入 32 位标志寄存器至堆栈顶端，然后将堆栈指针值减去 4。您若单纯地使用 PUSHF，则依 BITS 所设定的为准，若设定 [BITS 16] 则执行 PUSHFW，若设定 [BITS 32] 则执行 PUSHFD。

POPF 指令群的语法如下:

POPF

POPFD

POPFW

POPFW 从堆栈顶端弹出并存入标志寄存器,然后将堆栈指针值加上 2。POPFD 从堆栈顶端弹出至 32 位标志寄存器,然后将堆栈指针值加上 4。您若单纯地使用 POPF 则依 BITS 所设定的为准,若设定[BITS 16] 则执行 POPFW,若设定[BITS 32] 则执行 POPFD。

4-9 没有运算的 NOP 指令

没有运算的 NOP 指令,您看到这个指令的名称可能会觉得奇怪,但这个 No Operation 指令确实没做任何运算。当您在测试一个较大的程序时,可能会碰到一些问题,这时为了测试方便,您可将有问题的指令用 NOP 取代,继续测试,等到确定问题的所在后再解决该问题。

课后习题

b DB 12, 34, 56

w DW 1234H, 5678H, 9012H

d DD 12345678H, 90abcdefH, 13579aceH

1. 请写出下列寄存器的内容。

a. MOV AL, BYTE [b]

MOV BL, BYTE [b+1]

MOV CL, BYTE [b+2]

c. MOV EAX, DWORD [d]

MOV EBX, DWORD [d+4]

MOV ECX, DWORD [d+8]

b. MOV AX, WORD [w]

MOV BX, WORD [w+2]

MOV CX, WORD [w+4]

2. 请写出下列寄存器的内容。

a. MOV AL, BYTE [w]

MOV BL, BYTE [w+1]

MOV CL, BYTE [w+2]

MOV DL, BYTE [w+3]

b. MOV AL, BYTE [d+1]

MOV BL, BYTE [d+3]

MOV CL, BYTE [d+5]

MOV DL, BYTE [d+7]

3. 请写出下列寄存器的内容。

a. MOV AX, WORD 123

MOV BH, AH

MOV BL, AL

b. MOV AX, WORD [w+2]

XCHG AH, AL

4. 请写出下列 AX 及 EAX 寄存器的内容。

a. MOV BX, w

b. MOV EBX, d



MOV SI, 2

MOV AX, WORD [BX+SI]

MOV ESI, 4

MOV EAX, DWORD [EBX+ESI]

5. 请写出下列 AX 及 EAX 寄存器的内容。

a. PUSH WORD [w+2]

b. PUSH DWORD [d]

POP AX

POP EAX

6. 若标志寄存器的值为 0C1H, 请写出 AL 寄存器的内容。

a. PUSHF

POP AX

7. 请写出标志寄存器的值。

a. PUSH WORD 83H

b. PUSH WORD [d]

POPF

POPF

8. 请写出下列 AX 及 BX 寄存器的内容。

a. MOV AX, WORD [w]

b. MOV AX, WORD [d]

MOV BX, WORD [w+2]

MOV BX, WORD [d+2]

XCHG AX, BX

XCHG AX, BX

5

基本输入与输出

计算机系统的许多事件(event)是无法预测的,例如您随时都可能按下键盘中的任一键,您有没有想过当您按下一键时会发生什么事呢?计算机必须能警觉到您按下一键,然后做适当的输入处理。您按下一键时发生一个输入事件,计算机捕捉到这个事件后,就处理这个事件,处理完后又回到事件发生时的地点,这种处理方式称为中断处理(interrupt handling),处理中断的这一段程序称为中断处理程序(interrupt handler)。本章只介绍有关键盘输入、屏幕输出及打印机输出的相关中断处理。





5-1 软件中断 INT 指令

INT 指令执行软件中断，它的格式如下：

INT 8 位立即数

8 位立即数，其值范围为 0 至 255，或以十六进制写成 00H 至 0FFH。

DOS 系统提供将近 130 个有用的中断处理功能，这些功能的代码缺省存入 AH 寄存器中，然后使用 INT 指令执行编号为 21H 的中断，就可执行存储于 AH 中的功能了。

```
MOV AH, 01H ;设定从键盘输入一个字符至 AL 寄存器的功能
```

```
INT 21H ;中断处理, 执行 AH 指定的功能
```

上例中 MOV 指令将常量 01H 拷贝至寄存器 AH，表示要从键盘输入一个字符至 AL 寄存器，只是表示而已还没有执行。您调用 INT 21H 时才执行从键盘输入一个字符至 AL 寄存器的功能。

```
MOV DL, 'A' ;将字符常量'A'拷贝至寄存器 DL
```

```
MOV AH, 02H ;设定将寄存器 DL 的内容显示在屏幕的功能
```

```
INT 21H ;中断处理, 执行 AH 指定的功能
```

上例中第一个 MOV 指令将字符常量 A 拷贝至寄存器 DL，第二个 MOV 指令将常量 02H 拷贝至寄存器 AH，表示要将寄存器 DL 的内容显示在屏幕，只是表示而已还没有执行。您调用 INT 21H 时才执行将寄存器 DL 的内容 A 显示在屏幕。

5-2 将一个字符串输出到屏幕

程序 dispmsg.asm 每行前面的编号只是为了说明方便而已，原来的程序是没有编号的，本书往后都使用编号，用来说明而已。

第一行为注释，说明源程序文件名。第二行指示保留前面 256 个字节的内存位置给 PSP，这是因为我们要产生的是 .com 二进制文件，这种文件前面必须保留给 PSP，这个操作是系统在加载 .com 文件时所该做的事，我们只是预留位置给它就可以了。第三行是一个跳跃指令，跳到标号地址为 start 处，因为第四行并非可执行的指令，因此必须跳过。第四行定义一个字符串变量 msg，共占 16 个字节，第一个字节的值为 'G'，第 16 个字节的值为 '\$'，其中 13 相当十六进制数 0dH，它是 Carriage Return 返回的意思，也就是您按 <Enter> 键所产生的 ASCII 码，10 相当十六进制数 0aH，它是 Line Feed 换行的意思，而 '\$' 表示显示字符串结束符号。第五行将您想要显示于屏幕的字符串地址存入寄存器 DX 中。第六行将功能代码 09H 存入寄存器 AH 中。第七行执行 21H 中断，将 DX 所指地址的字符串显示在屏幕上。第八行返回操作系统。

您的计算机若使用中文的窗口系统，或安装中文系统，当然可以显示中文，例如：

```
msg DB '早安!', 13, 10, '$'
```

将在屏幕上显示“早安!”。

【程序 dispmsg.asm】

```
1 ; ***** dispmsg.asm *****
```

```
2 ORG 0100H
```

```

3      JMP start
4 msg   DB 'Good Morning!', 13, 10, '$'
5 start:  MOV DX, msg
6      MOV AH, 09H
7      INT 21H
8      RET

```

【运行】

```
howasm\ch05> nasmw dispmsg.asm -o dispmsg.com <Enter>
```

```
howasm\ch05> dispmsg <Enter>
```

```
Good Morning!
```

5-3 从键盘输入一个字符

程序 `getchar.asm` 从键盘输入一个字符，再将该字符显示在屏幕上。第 4~7 行定义变量 `msg`、`msg2`、`char` 及 `crlf`，其中 `char` 为字符变量，它将存储从键盘输入的字符，其他 `msg`、`msg2` 及 `crlf` 均为字符串变量，其初值显示只为输入方便而已。第 8~10 行显示字符串 `msg`，请注意显示 `msg` 后并没有换行。第 11~12 行键入一个字符至 `AL` 寄存器。第 13 行将键入的字符从寄存器 `AL` 移至内存地址 `char`。第 14~16 行强迫换行。第 17~19 行显示 `msg2` 字符串直到 '\$' 符号为止，因此所显示的变量值有 `msg2`、`char` 以及 `crlf` 等三个。第 20 行返回操作系统。

【程序 `getchar.asm`】

```

1 ; ***** getchar.asm *****
2      ORG 0100H
3      JMP start
4 msg   DB '请输入一个字符: ', '$'
5 msg2  DB '您刚输入的字符= '
6 char  DB ' '
7 crlf  DB 13, 10, '$'
8 start:  MOV DX, msg      ;显示 msg 字符串
9      MOV AH, 09H
10     INT 21H
11     MOV AH, 01H        ;键入一个字符至 AL
12     INT 21H
13     MOV [char], AL      ;将 AL 存入 char
14     MOV DX, crlf        ;换行
15     MOV AH, 09H
16     INT 21H
17     MOV DX, msg2        ;显示 msg2 字符串
18     MOV AH, 09H
19     INT 21H
20     RET

```



【运行】

```
howasm\ch05> nasmw getchar.asm -o getchar.com <Enter>
howasm\ch05> getchar <Enter>
    请输入一个字符: 9 <Enter>
    您刚输入的字符= 9
```

5-4 将一个字符输出到屏幕

程序 `putchar.asm` 将一个字符输出到屏幕,使用 21H 中断的 02H 功能可将存储于寄存器 DL 的字符显示在屏幕上。第 3~5 行显示'A' 字符。

第 6~8 行显示 ASCII 码序号为 13 的字符,这个字符为 Carriage Return 返回字符,常简称为 CR。第 9~11 行显示 ASCII 码序号为 10 的字符,这个字符为 Line Feed 换行字符,常简称为 LF。第 12~14 行显示'B' 字符。

第 15~17 行显示 ASCII 码序号为 7 的字符,这个字符为 Beep,发出“哔”的声音。第 18 行返回操作系统。

【程序 putchar.asm】

```
1 ; ***** putchar.asm *****
2     ORG 0100H
3     MOV DL, 'A'           ;显示'A'字符
4     MOV AH, 02H
5     INT 21H
6     MOV DL, 13            ;显示 CarriageReturn 字符
7     MOV AH, 02H
8     INT 21H
9     MOV DL, 10            ;显示 LineFeed 字符
10    MOV AH, 02H
11    INT 21H
12    MOV DL, 'B'           ;显示'B'字符
13    MOV AH, 02H
14    INT 21H
15    MOV DL, 7             ;显示 Beep 字符
16    MOV AH, 02H
17    INT 21H
18    RET
```

【运行】

```
howasm\ch05> nasmw putchar.asm -o putchar.com <Enter>
howasm\ch05> putchar <Enter>
    A
    B
```

5-5 从键盘输入一个字符串

程序 `getstr.asm` 从键盘输入一个字符串至内存地址 `buf` 处,使用 21H 中断的 0aH 功能可从键盘输入一个字符串至寄存器 DX 所指的内存地址。

第 6~8 行定义三个变量 `buf`、`count` 及 `s`,事实上这三个变量在内存中是连续的,也就是说 `count` 的地址为 `buf+1`,变量 `s` 的地址为 `buf+2`,因此这三个变量也可以看成一个变量来使用,分成三个是为了说明方便而已,请注意第 8 行使用 `TIMES` 伪指令定义 81 个空白的字节,准备要存储您从键盘输入的字符串。

第 9~11 行显示 `msg` 字符串于屏幕上。

第 12~14 行调用 21H 中断的 0aH 功能,从键盘输入一个字符串至内存地址 `buf` 处,因为 `buf` 的第一个字节的初值为 80,表示最多只能输入 80 个字符的字符串,超过 80 就不能接受了,`count` 是用来存储您实际输入字符的个数,它不能超过 80,变量 `s` 用来存储您实际输入的字符,直到您按下 <Enter> 键才结束输入的操作,<Enter> 键的 ASCII 码为 0dH (13) 它会存在您输入字符串的后面,变量 `count` 的计数并不包括 0dH 这个字符。

例如您输入 `abc<Enter>`,那么 `buf`、`count` 及 `s` 的内容如下所示:

80	3	'a'	'b'	'c'	0dH
buf+0	buf+1	buf+2	buf+3	buf+4	buf+5
buf	count	s			

第 15~17 行将 0dH 改为 '\$',方便将字符串显示在屏幕上。第 18~20 行执行换行的操作。第 21~23 行将您刚输入的字符串显示在屏幕上。

【程序 `getstr.asm`】

```

1 ; ***** getstr.asm *****
2     ORG 0100H
3     JMP start
4 msg  DB '请输入一个字符串: ', '$'
5 crlf DB 13, 10, '$'
6 buf  DB 80
7 count DB 0
8 s     TIMES 81 DB ' '
9 start:  MOV DX, msg           ;显示 msg 字符串
10      MOV AH, 09H
11      INT 21H
12      MOV DX, buf           ;键入一个字符串至 buf
13      MOV AH, 0aH
14      INT 21H
15      MOV BH, 0             ;将 s 中 0dH 改成 '$'
16      MOV BL, [count]
17      MOV BYTE [BX+s], '$'
18      MOV DX, crlf          ;换行
19      MOV AH, 09H

```

```

20      INT 21H
21      MOV DX, s           ;显示 s
22      MOV AH, 09H
23      INT 21H
24      RET

```

【运行】

```

howasm\ch05> nasmw getstr.asm -o getstr.com <Enter>
howasm\ch05> getstr <Enter>
howasm\ch05> 请输入一个字符串: abc <Enter>
                abc

```

程序 `getstr.asm` 从键盘输入一个字符串至内存地址，使用 21H 中断的 0aH 功能虽然可从键盘输入一个字符串，但您要提供一个寄存器，要提供输入最大字符数，而且字符串以 <Enter> 键值 0dH 作为结束符号，但 BIOS 的 21H 中断 09H 功能所显示的字符串却以 '\$' 作为结束符号，很难配合，因此作者另外写了一个宏 `readstr`，单纯地键入字符串，并以 '\$' 作为字符串结束符号。其测试程序如程序 `readstr.asm` 所示。

【程序 readstr.asm】

```

1 ; ***** readstr.asm *****
2      ORG 0100H
3      JMP start
4 msg   DB '请输入一个字符串: ', '$'
5 msg2  DB 13, 10, '您刚输入的字符串= ', '$'
6 s     TIMES 80 DB ' '
7 crlf  DB 13,10,'$'
8 ;
9 %include "..\mymacro\readstr.mac"
10 ;
11 start:
12      MOV DX, msg           ;显示 msg 信息
13      MOV AH, 09H
14      INT 21H
15      readstr s             ;从键盘读入字符串以 '$' 结束
16      MOV DX, msg2         ;显示 msg2 信息
17      MOV AH, 09H
18      INT 21H
19      MOV DX, s             ;显示刚刚读入的 s 字符串
20      MOV AH, 09H
21      INT 21H
22      RET

```

【运行】

```

howasm\ch05> nasmw readstr.asm -o readstr.com <Enter>
howasm\ch05> readstr <Enter>

```

请输入一个字符串: 12345 <Enter>

您刚输入的字符串= 12345

5-6 将一个字输出到屏幕

文件不管是字符、字符串还是数值，在计算机的内存里面都是以二进制的方式表示的，因此如何将内存里的文件显示出来，就变成相当重要的一件事情，不然程序设计老半天，还不晓得答案是对是错。要显示内存里的字符串您可以使用 21H 中断的 09H 功能就可轻易地将它显示出来。若不是字符串而是数值的话，要显示它可没那么方便了，要显示数值的方式有二进制、八进制、十进制、十六进制等，最常用的大概是十进制和十六进制了。本小节各举一个例子分别以十进制和十六进制显示数值。

程序 dispword.asm 将一个字内容的二进制数 num 转成十进制数 numstr 后显示出来，这个转换的工作就用 itostr.mac 宏程序来处理，因为有许多指令还没有介绍过，因此这里不必说明该宏是如何设计的，本程序只是将它利用 %include 伪指令引入程序中，如第 8 行所示。第 11 行 itostr 宏将二进制数 num 转成十进制数 numstr 后，将 numstr 显示于屏幕。

【程序 dispword.asm】

```
1 ; ***** dispword.asm *****
2     ORG 0100H
3     JMP start
4 num   DW 1357
5 msg   DB '十进制数= '
6 numstr DB ' ', '$'
7 ;
8 %include "../mymacro/itostr.mac"
9 ;
10 start:
11     itostr num, numstr, '$'      ;num 转成十进制数
12     MOV DX, msg                 ;显示 msg
13     MOV AH, 09H
14     INT 21H
15     RET
```

【运行】

howasm\ch05> nasmw dispword.asm -o dispword.com <Enter>

howasm\ch05> dispword <Enter>

十进制数= 1357

程序 showcs.asm 将分段寄存器 CS 的内容以十六进制数显示在屏幕上。这个转换的工作就用 showbyte.mac 宏程序来处理，本程序只是将它利用 %include 伪指令引入程序中，如第 7 行所示。第 10 行将 CS 值传送至内存 value 处。第 11~13 行显示 csmsg 信息。第 14 行将位于内存 value+1 处的字节（高地址）值以十六进制方式显示，第 15 行将位于内存 value 处的字节（低地址）值以十六进制方式显示。



同理，您也可以显示分段寄存器 DS、ES 及 SS 的内容，您会发现它们的值均相同，我的计算机环境显示的是 3115H，如果您的计算机环境不同可能会显示别的值，但这四个分段寄存器的内容还是相同的。这是 DOS 特别为 .com 文件加载内存执行而设定的。

【程序 showcs.asm】

```
1 ; ***** showcs.asm *****
2 ORG 0100H
3 JMP start
4 value DW 123
5 csmsg DB 'CS (十六进制数) = ', '$'
6 ;
7 %include "../mymacro/showbyte.mac"
8 ;
9 start:
10 MOV [value], CS ;将 CS 存入 value 内存地址
11 MOV DX, csmsg ;显示十六进制字符串 csstr
12 MOV AH, 09H
13 INT 21H
14 showbyte value+1 ;以十六进制显示 value+1 地址的内容
15 showbyte value ;以十六进制显示 value 地址的内容
16 RET
```

【运行】

```
howasm\ch05> nasmw showcs.asm -o showcs.com <Enter>
howasm\ch05> showcs <Enter>
CS (十六进制数) = 3115
```

5-7 显示内存内容

有时您在测试程序时常需要将内存内容显示在屏幕上，从而仔细地检查。程序 dumpmem.asm 将存储在内存 mem 里每一个字节的内容以十六进制形式显示出来，存储的文件形式有字节、字、双字以及字符串等，这段程序不管文件是什么形式，均以字节为单位，一个一个显示出来，显示的操作是由宏 showbyte 执行的，这段宏的内容将在介绍算术运算指令后，再说明。程序第 18 行调用 dispchr 宏，它将空格符号 space 显示出来。程序第 16~20 行构成一个循环，将在下一章中说明。

【程序 dumpmem.asm】

```
1 ; ***** dumpmem.asm *****
2 ORG 0100H
3 JMP start
4 mem DB 12H ;字符内容 12H
5 DW 1234H ;字内容 1234H
6 DD 12345678H ;双字内容 12345678H
7 DB 'ABC123abc+' ;字符串内容 ABC123abc+
```

```

8 space DB ' ' ;空格符
9 ;
10 %include "../mymacro/showbyte.mac" ;宏 showbyte
11 %include "../mymacro/dispchr.mac" ;宏 dispchr
12 ;
13 start:
14     MOV CX, 17 ;CX=17
15     MOV SI, 0 ;SI=0
16 repeat:
17     showbyte mem+SI ;mem+SI 地址的内容以十六进制显示
18     dispchr space ;显示空格符号
19     INC SI ;SI=SI+1 指向下一个字节
20     LOOP repeat ;重复执行 17 次 (CX=17)
21     RET

```

宏 `dispchr.mac` 第 1~6 行说明该宏的用法。第 9 行定义宏的名称为 `dispchr`，只有一个参数。第 10 行将寄存器文件存入堆栈，暂时保存。

第 11 行将要显示的字符（参数）存入寄存器 `DL`。第 12 行设定显示字符功能。第 13 行调用 `21H` 服务，这时才将刚刚存入寄存器 `DL` 的内容以字符形式显示于屏幕。第 14 行我们要做的事情做完后恢复原来寄存器文件。

【宏 `dispchr.mac`】

```

1 ; ***** dispchr.mac *****
2 ;
3 ; dispchr bmem
4 ; 将 bmem 内存内容以字符形式显示在屏幕上
5 ; bmem : 字符存放内存地址
6 ;
7 %ifndef DISPCHR_MAC
8 %define DISPCHR_MAC
9 %MACRO dispchr 1 ;参数 bmem 地址
10     PUSHA ;存储原来寄存器文件
11     MOV DL, [%1] ;要显示的字符存入 DL
12     MOV AH, 02H ;设定显示字符功能
13     INT 21H ;显示存储于 DL 的字符
14     POPA ;恢复原来寄存器文件
15 %ENDMACRO
16 %endif

```

【运行】

```

howasm\ch05> nasmw dumpmem.asm -o dumpmem.com <Enter>
howasm\ch05> dumpmem <Enter>

```

```

12 34 12 78 56 34 12 41 42 43 31 32 33 61 62 63 2B

```

[注]

```

A B C 1 2 3 a b c +

```



[注] 低地址

高地址

[注] mem+0 +1 +2 +3 +4 +5 +6 +7

运行的结果请注意字中的内容，其个位数是存在低地址的字节，而高位数是存在高地址的字节。值 1234H 其低位数是 34H，存储在低地址的字节，其地址为 mem + 1。其高位数是 12H，存储在高地址的字节，其地址为 mem + 2。同样的双字其值为 12345678H，其低位数是 78H，存储在低地址的字节，其地址为 mem + 3。其高位数是 12H，存储在高地址的字节，其地址为 mem + 6。

5-8 键盘输入控制

除了 Basic Input Output System 基本输入输出系统 (BIOS) 中的 16H 中断可用来管理键盘之外，DOS 也提供 21H 的中断服务，用来处理字符的输入。说明如下：

5-8-1 由键盘输入字符

中断 21H

功能 AH = 01H

说明 由键盘输入字符，显示在屏幕上。可使用 Ctrl-C 中断。

5-8-2 直接由键盘输入或输出字符

中断 21H

功能 AH = 06H

输入参数 DL = 0ffH

输出参数 ZF = 0 时 AL=输入的字符，不显示于屏幕。

ZF = 1 时 AL=0 表示没有输入。

说明 由键盘直接输入字符，不显示于屏幕。不须经过 Ctrl-C 检查。

中断 21H

功能 AH = 06H

输入参数 DL = 0ffH 以外字符

说明 直接将 DL 内容字符输出到屏幕。

5-8-3 直接由键盘输入字符

中断 21H

功能 AH = 07H

输出参数 AL = 输入的字符，不显示于屏幕。

说明 由键盘直接输入字符，不显示于屏幕。不须经过 Ctrl-C 检查。

5-8-4 直接由键盘输入字符

中断 21H

功能 AH = 08H

输出参数 AL = 输入的字符，不显示在屏幕上。

说明 由键盘直接输入字符，不显示在屏幕上。可由 Ctrl-C 中断。

5-8-5 由键盘输入字符串

中断 21H
 功能 AH = 0aH
 输出参数 DS:DX = 输入字符串缓冲区地址。
 说明 由键盘输入字符串直到按下<Enter>键时才停止,显示在屏幕上。
 可由 Ctrl-C 中断。
 缓冲区第 0 个字节:最大字符数(含<Enter>键)。
 缓冲区第 1 个字节:实际输入字符数(不含<Enter>键)。
 缓冲区第 2 个字节起:输入的字符,含<Enter>键 0dH。

5-8-6 检查键盘状态

中断 21H
 功能 AH = 0bH
 输出参数 AL = 00H 键盘缓冲区没有字符。
 AL = 0FFH 键盘缓冲区有字符。
 说明 检查键盘状态。

5-8-7 清除键盘缓冲区

中断 21H
 功能 AH = 0cH
 输入参数 AL = 01H、06H、07H、08H 或 0aH
 输出参数 AL = 00H 键盘缓冲区已经清除。
 说明 清除键盘缓冲区。

5-8-8 从键盘缓冲区读取字符

中断 16H
 功能 AH = 00H
 输出参数 AL = ASCII 码
 AH = 扫描码
 说明 从键盘缓冲区读取字符。

5-8-9 测试键盘缓冲区是否有字符

中断 16H
 功能 AH = 01H
 输出参数 ZF = 0 有字符
 ZF = 1 没有字符
 说明 测试键盘缓冲区是否有字符。

功能 AH = 06H
 输入参数 BH = 属性
 CX = 开始行及行编号, CH 行, CL; 行
 DX = 结束行及行编号, DH 行, DL; 行
 说明 向上滚动屏幕从 CX 处至 DX 处。

5-10 打印机输出控制

打印机输出控制除了 21H 中断之外还有 BIOS 的 17H 中断, 说明如下。

5-10-1 输出字符至打印机

中断 21H
 功能 AH = 05H
 输入参数 DL = 要打印的字符
 说明 输出 DL 内容字符至打印机。可使用 Ctrl-C 中断。

5-10-2 打印一个字符

中断 17H
 功能 AH = 00H
 输入参数 AL = 要打印的字符
 DX = 打印机编号
 0 LPT1 (缺省值)
 1 LPT2
 2 LPT3
 输出参数 AH = 打印机状态
 位 0 = 1 逾时
 位 1 = 1 保留
 位 2 = 1 保留
 位 3 = 1 I/O 错误
 位 4 = 1 已选定打印机
 位 5 = 1 打印纸印完
 位 6 = 1 确认
 位 7 = 1 打印机准备就绪
 说明 通过打印机打印 AL 内容字符。

5-10-3 取得打印机状态

中断 17H
 功能 AH = 02H
 输入参数 DX = 打印机编号
 0 LPT1 (缺省值)
 1 LPT2
 2 LPT3



输出参数 AH = 打印机状态

位 0 = 1 超时

位 1 = 1 保留

位 2 = 1 保留

位 3 = 1 I/O 错误

位 4 = 1 已选定打印机

位 5 = 1 打印纸印完

位 6 = 1 确认

位 7 = 1 打印机准备就绪

说明 通过打印机打印 AL 内容字符。

课后习题

1. 设计一个程序，从键盘输入一个字符后输出到屏幕上。

2. 设计一个程序，定义字节变量 ch2 如下：

```
ch2 DB '1', '2'
```

从键盘输入第一个字符取代'2'，输入第二个字符取代'1'，然后将 ch2 内容输出到屏幕上。

3. 设计一个程序，定义字节变量 ch2 如下：

```
ch2 DB '1', '2'
```

从键盘输入第一个字符取代'2'，输入第二个字符取代'1'，但输入的字符不显示在屏幕上，一般输入密码都不会显示在屏幕上的，然后将 ch2 内容输出到屏幕上。

4. 设计一个程序，定义字节变量 buf 如下：

```
buf DB 9
```

```
len DB 0
```

```
s DB '1', '2', '3', '4', '5', '6', ' '
```

使用 21H 的 0aH 功能从键盘输入六位数字，将这六位数字显示于屏幕上。

5. 设计一个程序，定义字节变量 s 如下：

```
s TIMES 80 DB ' '
```

引入 readstr.mac 后使用 readstr 宏从键盘输入一个 80 字符以内字符串，将这字符串显示在屏幕上。

6. 设计一个程序，清除屏幕后将'OK' 显示在屏幕正中央。

7. 设计一个程序检查是否按下控制键。

8. 设计一个程序，显示字符串“I like NASM!!” 在屏幕上。

9. 设计一个程序，将显示于屏幕的字符串“I like NASM!” 存入磁盘文件 ex0509.txt。

```
howasm\exer> ex0509 >ex0509.txt <Enter>
```

10. 设计一个程序，将显示于屏幕的字符串“I like NASM!” 从打印机输出。

```
howasm\exer> ex0510 >prn <Enter>
```

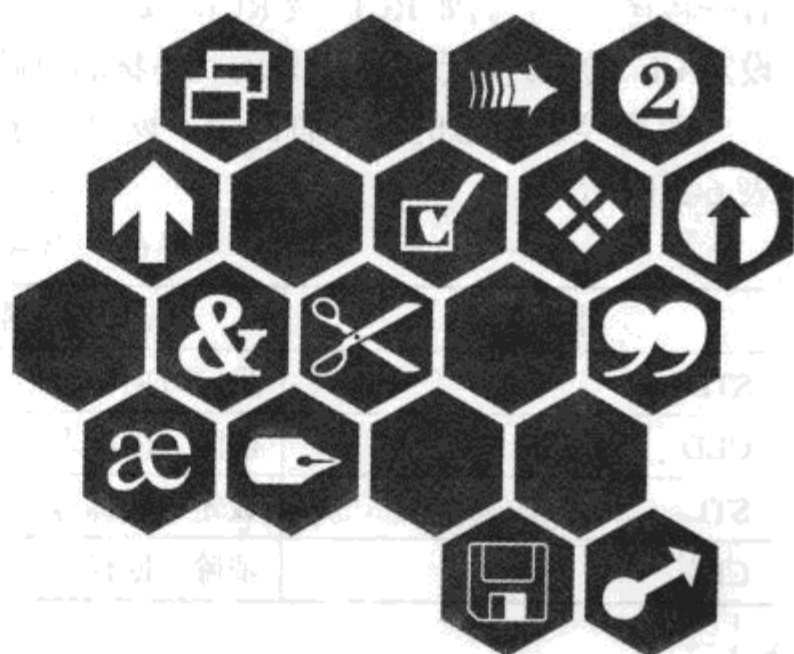
11. 设计一个程序，将字符串“I like NASM!” 直接由打印机输出。



6

程序流程控制

程序流程控制是指控制程序中指令执行的顺序。例如有些指令是按照它们出现的顺序执行，有些指令只有在指定的条件成立时才执行，有些指令则按照指定的条件成立与否而循环执行。顺序执行（sequential）、选择执行（selective）以及循环执行（iterative）等三种指令是结构化程序设计的三种基本结构。本章主要说明这三种结构及其用法。



6-1 标志寄存器

标志寄存器 (flag register), 16 位中已经使用了 9 位, 以表示指令执行的结果及机器的现况。如图 6-1 所示。

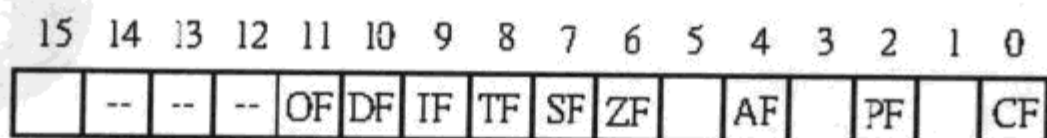


图 6-1 标志寄存器中使用 9 个标志位置

其中 CF 表示进位标志 (Carry Flag), PF 表示同位标志 (Parity Flag), AF 表示辅助进位标志 (Auxiliary Carry Flag), ZF 表示零标志 (Zero Flag), SF 表示符号标志 (Sign Flag), TF 表示单步标志 (Trap Flag), IF 表示中断标志 (Interrupt Flag), DF 表示方向标志 (Direction Flag), OF 表示溢出标志 (Overflow Flag)。

标志寄存器中 ZF、SF、OF、CF、AF 及 PF 这 6 个称为状态标志 (status flag), DF、IF 及 TF 这 3 个称为控制标志 (control flag)。控制标志用于控制处理器 (processor) 的操作。状态标志提供指令执行后的相关状态信息, 但控制标志控制尚未执行的指令行为。

6-2 改变标志的指令

通常您是不用去改变状态标志的值的，因为它主要用来让您检验而已，但有时您需要设定进位标志 CF 的值，因此处理器提供如表 6-1 所示的三个指令，用于改变状态标志 CF 的值。

表 6-1 进位标志 CF 的设定及清除

指令名称	功 能	说 明
STC	设定进位标志	设定 CF 值为 1
CLC	清除进位标志	清除 CF 值为 0
CMC	取进位标志补码	若 CF=1 则 CF=0, 若 CF=0 则 CF=1

这三个改变状态标志的指令有两个用法：第一，您可以用它记录调用一个过程（procedure）后的状态，调用执行成功与否，通常 CF 值设定为 1 时代表失败，设定为 0 时代表成功。第二，有些转移指令，例如 RCL 及 RCR，进位标志当成运算的一部分，您在执行该指令前您可能需要设定 CF 的值。对于 CF 以外的状态标志可就没有这么容易的指令，用来改变状态标志了。

改变控制标志的指令有四个，两个用于改变 DF 标志，另外两个用于改变 IF 标志。如表 6-2 所示。

表 6-2 控制标志的设定及清除

指令名称	功 能	说 明
STD	设定方向标志	设定 DF 值为 1
CLD	清除方向标志	清除 DF 值为 0
STI	设定中断标志	设定 IF 值为 1
CLI	清除中断标志	清除 IF 值为 0

每次执行字符串运算指令时，需要设定或清除 DF 方向标志的值。每次执行字符串运算指令时，最好每次都要设定或清除 DF 的值，不要认为前面已经设定或清除过了，就可以省下这个指令了，这样会增加程序调试 (debug) 的时间。一般来讲，您不需设定或清除中断标志的值，除非您要写一些与中断有关的设备驱动程序，否则请不要轻易地使用 STI 及 CLI 指令。处理器并没有提供改变控制标志 TF 值的指令。

6-3 条件转移指令

处理器提供许多可以改变程序流程的指令，其中大部分属于转移指令。通常处理器执行指令是依照程序的指令顺序，一个接一个地执行，这种指令称为顺序指令 (sequential)。转移 (jump) 指令告诉处理器转移到程序的指定标号地址 (label) 处继续执行。事实上执行的控制权已经转到该标号地址处的第一个指令，例如下列的程序：

```
L2:
    MOV AX, 0
```

若处理器执行一个转移指令跳至 L2 标号地址处，则要执行的下一个指令就是 L2 标号地址后的 MOV 指令。条件转移 (conditional jump) 是指指定的条件成立时才执行转移的操作，它的格式如下：

```
Jxxx label
```

xxx 为指定的条件代码，label 为标号地址名称。若条件成立 (其值为真 true) 则跳至 label 标号地址处继续执行，若条件不成立 (其值为假 false) 则继续执行 Jxxx 的下一个指令。

```
; if CX<>0 then CX=5
    JCXZ L2
    MOV CX, 5
```

```
L2:
```

上例中 CXZ 就是 xxx 代码的一种情形，JCXZ L2 条件转移就是说若 CX 寄存器的内容为 0 值时就转移至 L2 标号地址处继续执行，否则就执行 MOV CX, 5 指令。其流程图如图 6-2 所示。

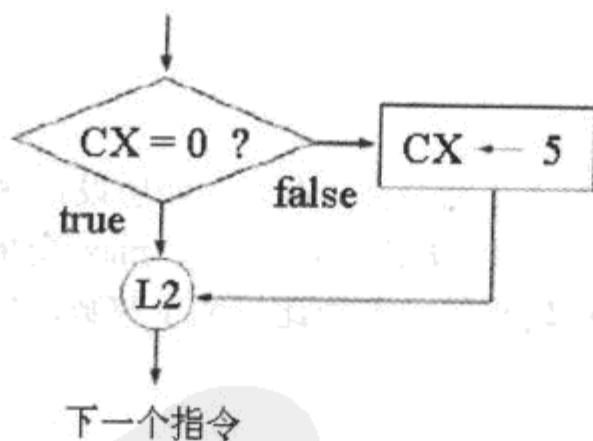


图 6-2 if CX<>0 then CX=5 流程图

条件转移指令其中有 10 个是依标志的设定与否而转移的，另外一个条件转移指令是依计数寄存器 CX 的内容是否为零而转移的。除了同位标志 PF 外的所有状态标志都有两个条



件转移指令，一个是当标志设定（标志为 1）时转移，另一个是当标志清除（标志为 0）时转移。条件转移指令如表 6-3 所示。

表 6-3 依标志及 CX 的条件转移指令表

指令名称	意 义
JZ	当 ZF=1 时转移
JNZ	当 ZF=0 时转移
JS	当 SF=1 时转移
JNS	当 SF=0 时转移
JO	当 OF=1 时转移
JNO	当 OF=0 时转移
JC	当 CF=1 时转移
JNC	当 CF=0 时转移
JP (JPE)	当 PF=1 时转移
JNP (JPO)	当 PF=0 时转移
JCXZ	当 CX=0 时转移

条件转移指令 J 其后跟随的字母意义如表 6-4 所示。

表 6-4 条件转移指令 J 跟随字母意义

字母	意 义
Z	零标志 ZF (Zero)
S	符号标志 SF (Sign)
O	溢出标志 OF (Overflow)
C	进位标志 CF (Carry)
P	同位标志 PF (Parity)
N	非 Not

6-4 比较两个整数

条件转移指令通常与比较的指令结合在一起。例如您设定变量 total 的值为 0，当它的值改变为不等于 0 时才转移，或您可以设定 AX 的值为 max 变量的值，执行一些指令后，测试 AX 的值若大于 max 的值就转移等。若要比​​较两个整数，然后依比较的结果决定是否转移，那么要做下面两件事情：

- 1. 比较两个数。
- 2. 依比较结果执行条件转移。

比较两个数您可以使用 CMP 指令，它的格式如下：

```
CMP value1 , value2
```

CMP 指令其后几乎都跟随一个条件转移指令，这里介绍几种可能的情形：

- 1. CMP AX, BX ;寄存器 AX 与寄存器 BX 比较
- 2. CMP AX, [total] ;寄存器 AX 与变量 total 比较
- 3. CMP AX, 0 ;寄存器 AX 与立即数 0 比较
- 4. CMP [total], DX ;变量 total 与寄存器 DX 比较
- 5. CMP [total], 0 ;变量 total 与立即数 0 比较

您不能直接比较两个变量，若您一定要比较两个变量，您可将其中一个拷贝至寄存器后再比较。

CMP [total], [max] ;错误，您不能直接比较两个变量
上行直接比较两个变量是错误的，应该改写如下：

```
MOV AX, [total]
CMP AX, [max]
```

除了不能直接比较两个变量外，还有一些其他的限制：

- 1. 所比较的两个数，必须同时是有符号数或无符号数，不能有符号数与无符号数混在一起。
- 2. 两个数必须长度（字节数）相同。

CMP 指令只比较两个整数，它执行时将第一个操作数的值减去第二个操作数的值，然后依结果的值设定相对应的状态标志。若想比较两个字符，可使用 CMPSB 或 CMPSW 指令。

执行 CMP 指令后，有两组的条件转移指令可供使用，这两组指令很相似，不同的只是一组针对有号整数的比较，另一组针对无符号整数的比较而已，如表 6-5 及表 6-6 表所示。

表 6-5 有符号整数比较的条件转移指令表

指令名称	意 义
JL (JNGE)	小于时转移（不大于等于）
JG (JNLE)	大于时转移（不小于等于）
JLE (JNG)	小于或等于时转移（不大于）
JGE (JNL)	大于或等于时转移（不小于）
JE	等于时转移
JNE	不等于时转移

表 6-6 无符号整数比较的条件转移指令表

指令名称	意义
JB (JNAE)	低于时转移（不高或等于）
JA (JNBE)	高于时转移（不低或等于）
JBE (JNA)	低于或等于时转移（不高于）
JAЕ (JNB)	高于或等于时转移（不低于）
JE	等于时转移
JNE	不等于时转移

当您使用条件转移指令时，有两点必须要注意：

1. 对于有符号整数的比较使用“小于” (less) 或“大于” (greater)，对于无符号整数的比较使用“低于” (below) 或“高于” (above)。
2. 执行 CMP 指令后, JL 指令表示 CMP 指令的第一个操作数值小于第二个操作数值时执行转移的操作, 执行 CMP 指令后, JB 指令表示 CMP 指令的第一个操作数值低于第二个操作数值时执行转移的操作。

下面四个例题中, 例一及例二是对于有符号数的比较, 例三及例四是对于无符号数的比较。

【例一】

```
CMP AX, 0           ;寄存器 AX 内容与 0 比较
JL L2               ;当 AX 小于 0 时转移至 L2
;当 AX >= 0 时所执行的指令
```

L2:

【例二】

```
MOV AX, [total]     ;将 total 内容拷贝至 AX
CMP AX, 0           ;寄存器 AX 内容与 0 比较
JGE L3              ;当 AX 大于或等于 0 时转移至 L3
;当 total < 0 时所执行的指令
```

L3:

【例三】

```
CMP AX, 0           ;寄存器 AX 内容与 0 比较
JB L4               ;当 AX 低于 0 时转移至 L4
;当 AX 高于或等于 0 时所执行的指令
```

L4:

【例四】

```
MOV AX, [total]     ;将 total 内容拷贝至 AX
CMP AX, 0           ;寄存器 AX 内容与 0 比较
JAE L5              ;当 AX 高于或等于 0 时转移至 L5
;当 total 低于 0 时所执行的指令
```

L5:

6-5 无条件转移指令 JMP

无条件转移指令 JMP 与条件无关, 它跳至指定的标号地址后从该标号地址处继续往下执行。它的格式如下:

```
JMP label
JMP SHORT label
```

单纯地使用 JMP label 产生的机器码为:

```
E9 rw/rd
```

E9 后面的操作数机器码占一个字或双字, 因此转移的范围可至 64KB。但使用 JMP SHORT label 则转移的范围最大为 128B 而已。

无条件转移指令 **JMP** 用于下面两种状况:

1. 跳过一些您不想执行或不能执行的指令。
2. 往回转移构成一个循环 (loop)。

程序 `jump.asm` 是一个转移的例子。第 3 行是一个无条件转移指令, 跳过不可执行的 `DB` 定义伪指令, 执行 `start` 标号地址处的可执行指令。第 15 行无条件短程转移指令跳至 `label1`, 往前只能跳过 127 个字节。第 27 行往前短程转移至 `label3`。第 34 行往前短程转移至 `endjob`。第 35 行往回无条件转移至 `label2`。

【程序 `jump.asm`】

```

1 ; ***** jump.asm *****
2     ORG 0100H
3     JMP start                ;往后长程转移
4 msg0 DB '长程转移至 start', 13, 10, '$'
5 msg1 DB '短程往前转移至 label1', 13, 10, '$'
6 msg2 DB '短程往后转移至 label2', 13, 10, '$'
7 msg3 DB '长程往前转移至 label3', 13, 10, '$'
8 msg4 DB '短程往前转移至 endjob', 13, 10, '$'
9 table TIMES 256 DB ' '
10 ;
11 start:
12     MOV DX, msg0
13     MOV AH, 09H
14     INT 21H
15     JMP SHORT label1        ;往前转移至 label1
16     TIMES 127 NOP
17 label1:
18     MOV DX, msg1
19     MOV AH, 09H
20     INT 21H
21 ;
22     MOV CX, 2
23 label2:
24     MOV DX, msg2
25     MOV AH, 09H
26     INT 21H
27     JMP SHORT label3        ;往前转移至 label3
28     TIMES 127 NOP
29 label3:
30     MOV DX, msg3
31     MOV AH, 09H
32     INT 21H
33     DEC CX
34     JCXZ endjob             ;往前短程转移至 endjob

```

; 往回长程转移至 label2

40 RET

短程往前转移至 endjob

```
33 00000203 BA[2C00]    MOV DX, msg2
```

```

... ..
36 0000020A EB7F      JMP SHORT label3
... ..
38                label3:
39 0000028B BA[4300]   MOV DX, msg3
... ..
43 00000293 E303      JCXZ endjob
44 00000295 E96BFF    JMP label2
45                endjob:
46 00000298 BA[5A00]   MOV DX, msg4
... ..
49 0000029F C3        RET

```

6-6 循环指令 LOOP

一个循环(loop)是指一系列的指令循环执行指定的次数。LOOP 指令建立一个循环,该循环的指令循环执行指定的次数。例如有一个数据表,包含 10 个数值,想将每一个数值累加至变量 sum 中,那么要循环执行累加计算 10 次。

LOOP 指令的格式如下:

LOOP label

就像条件转移指令一样, label 的范围必须在 128 个字节之内。

LOOP 指令使用 CX 寄存器当循环的计数器,开始时循环执行的次数放在 CX 中,每执行循环一次, CX 的值就自动递减一,然后测试 CX 的值是否为零,若不为零则继续下一次的循环,如此循环执行,一直到 CX 的值递减至零时为止, LOOP 的流程图如图 6-3 所示。

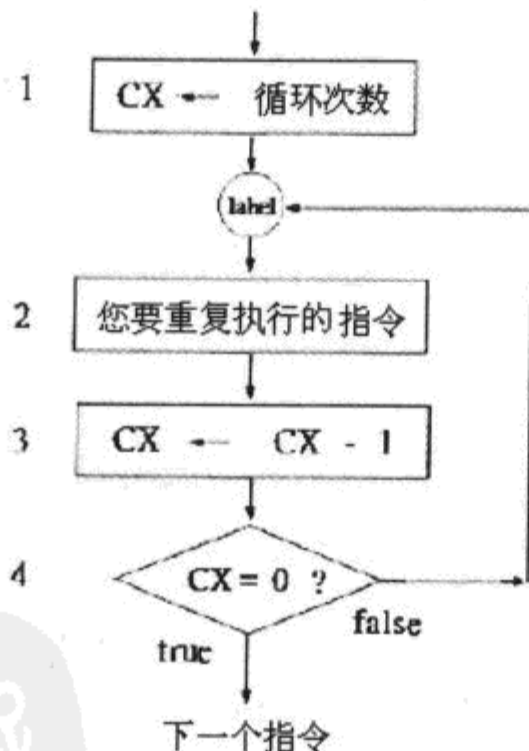


图 6-3 LOOP 循环流程图

在图 6-3 所示的 LOOP 循环流程图中,流程图处理块 1 中的循环次数您要设定,流程图处理块 2 是您要循环执行的指令,当然您要自己编写。流程图处理块 3 及 4 则隐藏在 LOOP



指令中，它会自动执行，您就不用考虑了。这个流程图写成汇编语言的程序如下：

```
MOV CX, 10 ;流程图处理块 1
label:
    ;您要循环执行的指令 ;流程图处理块 2
    LOOP label ;流程图处理块 3 及 4
```

程序 sum.asm 中第 4 行定义一个 table 数组变量，它的内容分别为 1 至 19 的奇数，每一个奇数均存放于一个字里。第 5 行定义变量 sum，要存放 table 中每一个元素值的总和。第 7 行定义变量 sumdec 是宏 itostr.mac 将 sum 二进制数转换成十进制数后所存储的地方。

第 14~18 行就是一个 LOOP 指令的循环结构，第 14 行设定循环次数，因为 table 中共有 10 个元素，因此设为 10，第 15~18 行是要执行循环的指令，将 table+SI 所指的元素值累加至 AX 寄存器后 SI 的值加 2，指到 table 的下一个元素，预备用于下一循环累加，第 18 行执行 LOOP 指令跳至 label 处继续循环。第 19 行将累加的结果存入 sum 变量中。第 21 行调用宏 itostr.mac 将 sum 二进制数转换成十进制数后存入 sumdec 变量。

第 22~24 行将 sumdec 的内容显示于屏幕。第 25 行返回操作系统。

【程序 sum.asm】

```
1 ; ***** sum.asm *****
2 ORG 0100H
3 JMP start
4 table DW 1,3,5,7,9,11,13,15,17,19
5 sum DW 0
6 msg DB 'table 元素值总和= '
7 sumdec DB ' ', 13, 10, '$'
8 ;
9 %include "../mymacro/itostr.mac"
10 ;
11 start:
12 MOV SI, 0 ;SI=0
13 MOV AX, 0 ;AX=0
14 MOV CX, 10 ;CX=10
15 label:
16 ADD AX, [SI+table] ;AX=AX+(table 第 SI 项)
17 ADD SI, 2 ;SI=SI+2
18 LOOP label ;继续循环
19 MOV [sum], AX ;sum=AX
20 ;
21 itostr sum, sumdec, '$' ;sum 转成字符串
22 MOV DX, msg ;显示 msg
23 MOV AH, 09H
24 INT 21H
25 RET
```

【运行】

```
howasm\ch06> nasmw sum.asm -o sum.com <Enter>
```

```
howasm\ch06> sum <Enter>
```

```
table 元素值总和= 100
```

LOOP 指令建立执行 CX 次的循环，但有时也需建立依比较结果而循环执行的循环，LOOPE 及 LOOPNE 指令将这两种功能结合在一起，它的格式与 LOOP 并没有两样，这两个指令在 LOOP 循环前直接放入一个比较的指令，是否需要循环不但要检查 CX 是否为 0，同时也要检查指定的条件是否满足，才决定是否继续循环。LOOPE 的条件为相等 (equal)，LOOPNE 的条件为不相等 (not equal) 时继续循环。

程序 pos.asm 第 5 行定义一个变量 pos，用于存放在 table 中第一个出现 11 元素的位置，本例中它的位置为第 6 个。第 12~18 行是 LOOPNE 的结构，第 14~17 行为您要循环执行的指令，您要处理 table 的下一个元素的同时将位置变量的值增加 1，第 17 行比较下一个元素是否为 11，若不是 11 且寄存器 CX 的值不为 0 时继续循环。table 的下一个元素值若为 11，则跳出循环，继续执行 LOOPNE 的下一个指令。请考虑假如 table 中没有您要找寻的元素 11 时，pos 的值是多少呢？请将它当一个练习题目做做看。

【程序 pos.asm】

```
1 ; ***** pos.asm *****
2      ORG 0100H
3      JMP start
4 table DW 1,3,5,7,9,11,13,15,17,19
5 pos   DW 0
6 msg   DB 'table 中元素 11 的位置= '
7 posdec DB ' ', 13, 10, '$'
8 ;
9 %include "../mymacro/itostr.mac"
10 ;
11 start:      MOV SI, 0                ;SI=0
12            MOV CX, 10               ;CX=10
13 label:
14            MOV AX, [SI+table]        ;AX=table 第 SI 元素值
15            ADD SI, 2                 ;SI=SI+2
16            INC WORD [pos]            ;pos 内容+1
17            CMP AX, 11                ;AX:11?
18            LOOPNE label              ;不相等或 CX>0 时继续循环
19 ;
20 itostr pos, posdec, '$'              ;pos 内容转成字符串
21            MOV DX, msg               ;显示 msg
22            MOV AH, 09H
23            INT 21H
24            RET
```

【运行】

```

howasm\ch06> nasmw pos.asm -o pos.com <Enter>
howasm\ch06> pos <Enter>
table 中元素 11 的位置= 6
    
```

6-7 选择结构

选择结构从一个或多个情况中选择您所要的，然后执行指定的指令。
 例如您的程序中可以检查某一个变量 ch2 值，当它的值是'a' 时就将'A' 存入 ch2 变量中。程序片段如下：

```

; if ch2='a' then ch2='A'
MOV AL, [ch2]      ;变量 ch2 内容拷贝至 AL
CMP AL, 'a'        ;AL 内容与 'a' 比较
JNE next           ;不相等时跳至 next
MOV [ch2], 'A'     ;相等时将 'A' 存入 ch2 变量中
    
```

next:

使用汇编语言时，您必须将条件的建立与检查比较的结果分开成两个指令，条件的建立您可以使用 **CMP** 指令，两个操作数比较的结果会设定相对应的状态标志，检查比较的结果可使用条件转移指令，直接测试状态标志，满足指定的条件才转移至指定的标号地址。上例中比较变量 ch2 的内容及'a' 立即数，满足相等条件时才将'A' 存入 ch2 变量中，否则跳至 next 标号地址处继续执行。这种结构称为单选的 if 结构如图 6-4 所示。

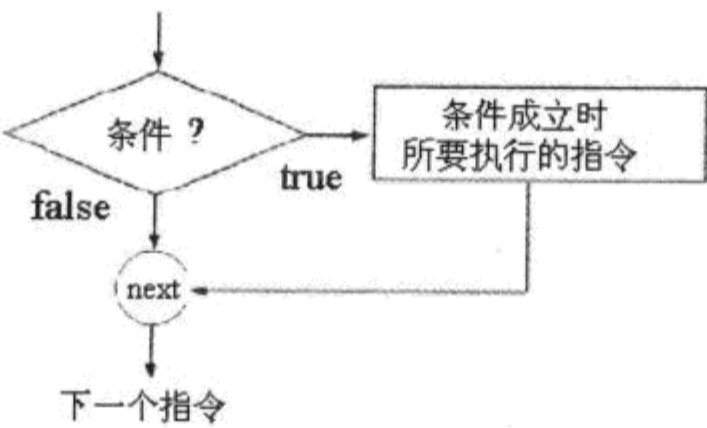


图 6-4 单选的 if 结构

在上一个例题中，条件是 **CMP AL,'a'** 指令，条件转移指令 **JNE next** 判断不为'a' 时才跳至 next，否则就执行 **MOV [ch2], 'A'** 的指令，这个指令就是当条件成立时您所执行的指令。
 在本章后面的几个例子都会用到有关字符的处理，字符在计算机内存里面是以二进制的方式存储的，个人计算机一般使用 ASCII 码以一个字节来表示一个字符，例如字符'A' 以 0100 0001 二进制的方式存储，以十六进制表示为 41，以十进制表示为 65，您要从键盘输入'A' 字符，您可以调用第 21H 号的中断，功能编号 01H，很容易输入，要输出字符，使用 02H 的功能也可轻易地将'A' 显示在屏幕上，但若若要显示 41H 或 65 可就没有那么容易了。
 程序 hex.asm 从键盘输入一字符至 AL，然后将 AL 内容字符拷贝至 ch2 变量。第 19~27 行将 ch2 字节的高四位以一个十六进制数字显示出来。

第 29~36 行将 ch2 字节的低四位以一个十六进制数字显示出来。因为到目前为止并未介绍算术运算指令的用法,因此转换成十六进制数字并没有使用到除法的运算,只用到简单的加法而已,算术运算指令将在下一章介绍。为了方便以及能够循环使用,将一个字节的内容以十六进制数字表示的这段程序代码写成一个宏 showbyte.mac。程序第 22~25 行、31~34 行都用到单选的 if 结构。

【程序 hex.asm】

```

1 ; ***** hex.asm *****
2      ORG 0100H
3      JMP start
4 msg   DB '请输入一个字符: ', '$'
5 ch2   DB ' '
6 msghex DB 13, 10, '十六进制数字= ', '$'
7 ;
8 start:      MOV DX, msg           ;显示 msg 字符串
9             MOV AH, 09H
10            INT 21H
11            MOV AH, 01H           ;从键盘输入一字符至 AL
12            INT 21H
13            MOV [ch2], AL         ;将 AL 字符拷贝至 ch2 变量
14 ;
15            MOV DX, msghex        ;显示 msghex 字符串
16            MOV AH, 09H
17            INT 21H
18 ;
19            MOV DL, [ch2]          ;显示 ch2 高位(十六进制数)
20            MOV CL, 4
21            SHR DL, CL
22            CMP DL, 10
23            JL high
24            ADD DL, 7
25 high:      ADD DL, 30H
26            MOV AH, 02H
27            INT 21H
28 ;
29            MOV DL, [ch2]          ;显示 ch2 低位(十六进制数)
30            AND DL, 0FH
31            CMP DL, 10
32            JL low
33            ADD DL, 7
34 low:      ADD DL, 30H
35            MOV AH, 02H
36            INT 21H

```



37 ;

38

RET

【运行】

```
howasm\ch06> nasmw hex.asm -o hex.com <Enter>
```

```
howasm\ch06> hex <Enter>
```

请输入一个字符: A

十六进制数字= 41

```
howasm\ch06> hex <Enter>
```

请输入一个字符: 1

十六进制数字= 31

程序 hex2.asm 第 5 行定义一个字符变量 ch2。第 8 行引入一个宏 showbyte.mac, 将字节二进制数以十六进制数字显示。

【程序 hex2.asm】

```
1 ; ***** hex2.asm *****
2          ORG 0100H
3          JMP start
4 msg      DB '请输入一个字符: ', '$'
5 ch2      DB ' '
6 msghex   DB 13, 10, '十六进制数字= ', '$'
7 ;
8 %include "../mymacro/showbyte.mac"
9 ;
10 start:   MOV DX, msg           ;显示 msg 字符串
11          MOV AH, 09H
12          INT 21H
13          MOV AH, 01H           ;从键盘输入一字符至 AL
14          INT 21H
15          MOV [ch2], AL         ;将 AL 字符拷贝至 ch2 变量
16 ;
17          MOV DX, msghex        ;显示 msghex
18          MOV AH, 09H
19          INT 21H
20          showbyte ch2          ;ch2 以十六进制显示
21          RET
```

【运行】

```
howasm\ch06> nasmw hex.asm -o hex.com <Enter>
```

```
howasm\ch06> hex <Enter>
```

请输入一个字符: A

十六进制数字= 41

程序 if2.asm 说明单选 if 结构的用法。第 9~14 行从键盘输入一个字符至寄存器 AL 后拷贝至变量 ch2。第 16~18 行换行。第 20~22 行变量 ch2 的内容与字符'a' 比较。第 22 行

当 ch2 的值不等于'a' 时跳至 next 标号地址, 将 ch2 的值显示出来, 若 ch2 的值等于'a' 时设定 ch2 的值为'A' 后才将 ch2 的值显示出来。

【程序 if2.asm】

```

1 ; ***** if2.asm *****
2      ORG 0100H
3      JMP start
4 msg   DB '请输入一个字符: ', '$'
5 newline DB 13, 10, '$'
6 msgchar DB "字符= "
7 ch2   DB ' ', '"', 13, 10, '$'
8 start:
9      MOV DX, msg           ;显示 msg 字符串
10     MOV AH, 09H
11     INT 21H
12     MOV AH, 01H           ;从键盘输入一字符至 AL
13     INT 21H
14     MOV [ch2], AL         ;将 AL 字符拷贝至 ch2 变量
15 ;
16     MOV DX, newline       ;换行
17     MOV AH, 09H
18     INT 21H
19 ;
20     MOV AL, [ch2]          ;拷贝 ch2 变量至 AL
21     CMP AL, 'a'           ;AL 内容与 'a' 比较
22     JNE next              ;不相等时跳至 next
23     MOV BYTE [ch2], 'A'   ;相等时将 ch2 转换为 'A'
24 next:
25     MOV DX, msgchar        ;以字符方式显示 ch2
26     MOV AH, 09H
27     INT 21H
28     RET

```

【运行】

```

howasm\ch06> nasmw if2.asm -o if2.com <Enter>
howasm\ch06> if2 <Enter>
howasm\ch06> 请输入一个字符: a <Enter>
           字符= 'A'
howasm\ch06> 请输入一个字符: m <Enter>
           字符= 'm'

```

有时测试条件成立时要执行这些指令, 条件不成立时要执行那些指令, 也就是说测试前程序是一条主流, 测试后不是执行这个分支的指令, 就是执行那个分支的指令, 不管是这支还是那支, 最后都要合成一个主流。

以程序 if3.asm 为例, 第 22 行比较输入的字符是否高于'D', 若是高于'D' 则跳至标号地址 above 处执行第 30~32 行的指令, 显示“notpass!(不及格!)”信息。若是低于或等于'D' 则执行第 25~27 行指令, 显示“pass!(及格!)”信息后跳至 next 处继续执行, 称为二选一的 if 结构, 流程图如图 6-5 所示。

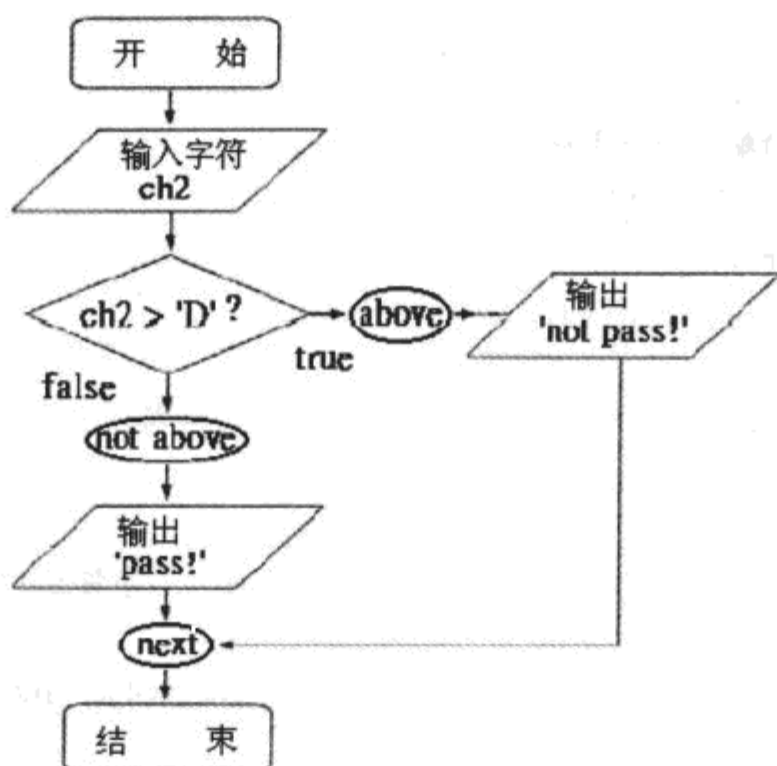


图 6-5 二选一的 if 结构

【程序 if3.asm】

```

1 ; ***** if3.asm *****
2         ORG 0100H
3         JMP start
4 msg      DB '请输入一个字符: ', '$'
5 ch2      DB ' '
6 newline  DB 13, 10, '$'
7 pass     DB '及格!', 13, 10, '$'
8 notpass  DB '不及格!', 13, 10, '$'
9 start:
10        MOV DX, msg           ;显示 msg 字符串
11        MOV AH, 09H
12        INT 21H
13        MOV AH, 01H          ;从键盘输入一字符至 AL
14        INT 21H
15        MOV [ch2], AL        ;将 AL 字符拷贝至 ch2 变量
16 ;
17        MOV DX, newline      ;换行
18        MOV AH, 09H
19        INT 21H
20 ;
21        MOV AL, [ch2]         ;拷贝 ch2 变量至 AL
  
```

```

22          CMP AL, 'D'                ;AL 内容与'D'比较
23          JA above                  ;高于'D'时跳至 above
24 notabove:
25          MOV DX, pass              ;A/B/C/D 显示 pass 字符串,及格
26          MOV AH, 09H
27          INT 21H
28          JMP next
29 above:
30          MOV DX, notpass           ;E/F 显示 notpass 字符串,不及格
31          MOV AH, 09H
32          INT 21H
33 next:
34          RET

```

【运行】

```
howasm\ch06> nasmw if3.asm -o if3.com <Enter>
```

```
howasm\ch06> if3 <Enter>
```

```
howasm\ch06> 请输入一个字符: A <Enter>
```

及格!

```
howasm\ch06> 请输入一个字符: E <Enter>
```

不及格!

有时测试的条件不只两种情形,这时您要一个个条件逐一测试,满足第一个条件就跳至第一标号地址执行,满足第二个条件就跳至第二标号地址执行,满足第三个条件就跳至第三标号地址执行等,最后合成主流。

以程序 if4.asm 为例,从键盘输入一字符至变量 ch2,然后逐一比较是否为'A',若是则显示 msgA 字符串,是否为'B',若是则显示 msgB 字符串,是否为'C',若是则显示 msgC 字符串,是否为'D',若是则显示 msgD 字符串,若都不是则显示 msgE 字符串,显然是要从五种情况中选一种来执行。

第 25 行比较输入的字符是否为'A',若是则跳至标号地址 labelA 处执行第 35~38 行的指令,显示 msgA 字符串。第 27 行比较输入的字符是否为'B',若是则跳至标号地址 labelB 处执行第 40~43 行的指令,显示 msgB 字符串。

第 29 行比较输入的字符是否为'C',若是则跳至标号地址 labelC 处执行第 45~48 行的指令,显示 msgC 字符串。第 31 行比较输入的字符是否为'D',若是则跳至标号地址 labelD 处执行第 50~53 行的指令,显示 msgD 字符串。

第 33 行是其他的情形,直接跳至标号地址 labelE 处执行第 55~57 行的指令,显示 msgE 字符串。每一种情形显示 msg? 字符串后都跳回程序的主流 next 标号地址处,它的流程如图 6-6 所示。

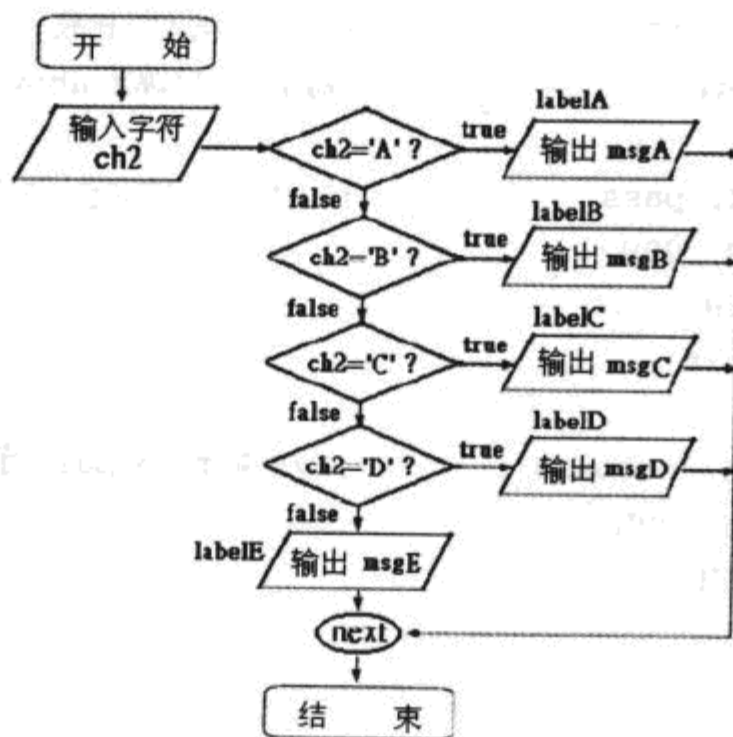


图 6-6 多选一的 if 结构

【程序 if4.asm】

```

1 ; ***** if4.asm *****
2         ORG 0100H
3         JMP start
4 msg      DB '请输入一个字符：', '$'
5 ch2      DB ' '
6 newline  DB 13, 10, '$'
7 msgA     DB '分数 90-100', 13, 10, '$'
8 msgB     DB '分数 80-89', 13, 10, '$'
9 msgC     DB '分数 70-79', 13, 10, '$'
10 msgD    DB '分数 60-69', 13, 10, '$'
11 msgE    DB '分数 0-59', 13, 10, '$'
12 start:
13         MOV DX, msg           ;显示 msg 字符串
14         MOV AH, 09H
15         INT 21H
16         MOV AH, 01H          ;从键盘输入一字符至 AL
17         INT 21H
18         MOV [ch2], AL        ;将 AL 字符拷贝至 ch2 变量
19 ;
20         MOV DX, newline      ;换行
21         MOV AH, 09H
22         INT 21H
23 ;
24         MOV AL, [ch2]         ;拷贝 ch2 变量至 AL
25         CMP AL, 'A'           ;AL 内容与 'A' 比较
26         JE labelA
  
```

```

27      CMP AL, 'B'           ;AL 内容与 'B' 比较
28      JE labelB
29      CMP AL, 'C'           ;AL 内容与 'C' 比较
30      JE labelC
31      CMP AL, 'D'           ;AL 内容与 'D' 比较
32      JE labelD
33      JMP labelE
34 labelA:
35      MOV DX, msgA           ;A 级显示 90~100 分
36      MOV AH, 09H
37      INT 21H
38      JMP next
39 labelB:
40      MOV DX, msgB           ;B 级显示 80~89 分
41      MOV AH, 09H
42      INT 21H
43      JMP next
44 labelC:
45      MOV DX, msgC           ;C 级显示 70~79 分
46      MOV AH, 09H
47      INT 21H
48      JMP next
49 labelD:
50      MOV DX, msgD           ;D 级显示 60~69 分
51      MOV AH, 09H
52      INT 21H
53      JMP next
54 labelE:
55      MOV DX, msgE           ;E 级显示 0~59 分
56      MOV AH, 09H
57      INT 21H
58 next:
59      RET

```

【运行】

```

howasm\ch06> nasmw if4.asm -o if4.com <Enter>
howasm\ch06> if4 <Enter>
howasm\ch06> 请输入一个字符: A <Enter>
          分数 90~100
howasm\ch06> 请输入一个字符: B <Enter>
          分数 80~89
howasm\ch06> 请输入一个字符: C <Enter>
          分数 70~79

```



howasm\ch06> 请输入一个字符: D <Enter>

分数 60~69

howasm\ch06> 请输入一个字符: E <Enter>

分数 0~59

6-8 循环结构

循环结构中最简单的一种是循环指定的次数, 这种结构可以使用 **LOOP** 指令, 它的结构如下:

将指定的次数存入计数寄存器 **CX**

label: 您要循环执行的指令

LOOP label

它的流程图您可参考图 6-3。它的例题如前面的 **sum.asm** 程序所示。

```
MOV SI, 0
```

```
MOV CX, 10
```

```
label:
```

```
ADD AX, [SI+table]
```

```
ADD SI, 2
```

```
LOOP label
```

首先将指定的次数 10 存入计数寄存器 **CX**。两个 **ADD** 指令为您要循环执行的指令, 一共循环执行 10 次。若您不想使用 **LOOP** 指令, 可改写如下:

```
MOV CX, 10
```

```
label:
```

```
ADD AX, [SI+table]
```

```
ADD SI, 2
```

```
DEC CX
```

```
JG label
```

将 **LOOP** 指令改成两个指令 **DEC** 及 **JG**, 将寄存器 **CX** 值减去 1 后, 若 **CX** 的值仍然大于零, 那就跳到 **label** 标号地址处继续循环。不过既然有现成的 **LOOP** 循环指令可用, 当然用 **LOOP** 指令让您的程序看起来较简洁了。

循环结构中有一种是先测试指定的条件是否成立, 若成立则执行您要循环执行的指令, 一直到指定的条件不成立为止。它的流程图如图 6-7 所示。

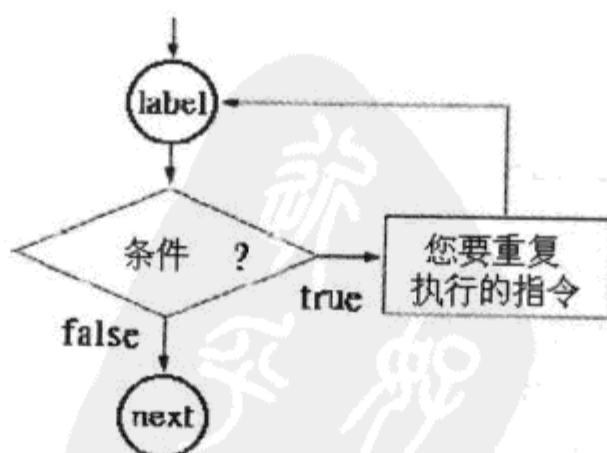


图 6-7 while 循环结构流程图

若以汇编语言的方式来表示, 如下:

```
label:  设定条件
        若条件不成立则跳出循环 next 标号地址
        您要循环执行的指令
        跳至 label 标号地址继续循环
```

next:

以程序 while2.asm 为例, 第 15~22 行构成一个 while 结构的循环, 第 16 行设定条件为存在寄存器 AL 中的输入字符是否为<Enter> 键所产生的 0dH, 即十进制数 13。第 17 行测试该条件, 若是<Enter> 键则跳出循环, next 标号地址在循环外面。若不是<Enter> 键则执行第 18~21 行的指令后, 跳至 label 标号地址继续循环, 第 18~21 行的这一段指令就是您要循环执行的指令, 它将输入的字符存入 array 数组的适当位置, 输入下一个字符至 AL 寄存器后, 再跳至 label 继续测试, 一直到您按<Enter> 键时, AL 的内容为 0dH 才跳出循环。

【程序 while2.asm】

```
1 ; ***** while2.asm *****
2         ORG 0100H
3         JMP start
4 msg     DB '请输入一个字符串: ', '$'
5 newline DB 13, 10, '$'
6 msg2    DB '您刚输入的字符串= '
7 array   TIMES 80 DB ' '
8 start:
9         MOV DX, msg                ;显示 msg 字符串
10        MOV AH, 09H
11        INT 21H
12        MOV SI, 0                  ;输入字符计数
13        MOV AH, 01H                ;从键盘输入一字符至 AL
14        INT 21H
15 label:
16        CMP AL, 0dH                ;是否为 Enter 键
17        JE next                    ;是, 跳出循环
18        MOV BYTE [array+SI], AL    ;将输入的字符存入 array
19        INC SI                      ;指向下一个字节
20        MOV AH, 01H                ;从键盘输入一字符至 AL
21        INT 21H
22        JMP label                  ;跳至 label 标号地址继续循环
23 next:
24        MOV BYTE [array+SI], '$'    ;array 以 '$' 表结束
25        MOV DX, newline             ;换行
26        MOV AH, 09H
27        INT 21H
28        MOV DX, msg2                ;显示 array 字符串
29        MOV AH, 09H
```

30 INT 21H
31 RET

【运行】

```
howasm\ch06> nasmw while2.asm -o while2.com <Enter>
howasm\ch06> while2 <Enter>
howasm\ch06> 请输入一个字符串: 12345 <Enter>
              您刚输入的字符串= 12345
```

另一种循环结构是先执行您要循环执行的指令，然后再测试指定的条件是否成立，若成立则继续执行您要循环执行的指令，一直到指定的条件不成立为止。它的流程图如图 6-8 所示。

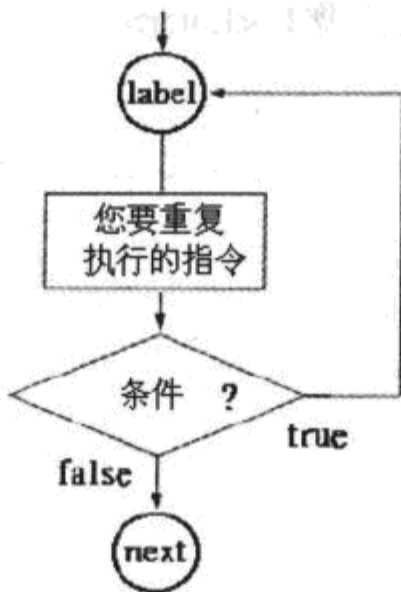


图 6-8 repeat 循环结构流程图

若以汇编语言的方式来表示，如下：

```
label: 您要循环执行的指令
       设定条件
       若条件成立则跳至 label 标号地址继续循环
```

next:

以程序 repeat2.asm 为例，第 13~19 行构成一个 repeat 循环结构，您要循环执行的指令为第 14~18 行，第 18 行设定条件 CMP AL, 0dH 比较 AL 的值是否为 <Enter> 键值。第 19 行不相等的条件成立则跳至 label 标号地址继续循环。

【程序 repeat2.asm】

```
1 ; ***** repeat2.asm *****
2           ORG 0100H
3           JMP start
4 msg       DB 'keyin a string: ', '$'
5 newline   DB 13, 10, '$'
6 msg2       DB '您刚输入的字符串= '
7 array      TIMES 80 DB ' '
8 start:
```

```

9          MOV DX, msg           ;显示 msg 字符串
10         MOV AH, 09H
11         INT 21H
12         MOV SI, -1            ;字符位置
13 label:
14         INC SI                ;指向下一个字节
15         MOV AH, 01H           ;从键盘输入一字符至 AL
16         INT 21H
17         MOV BYTE [array+SI], AL ;将输入的字符存入 array
18         CMP AL, 0dH           ;是否为<Enter>键
19         JNE label            ;不是,继续循环
20 next:
21         MOV BYTE [array+SI], '$' ;array 以 '$' 表结束
22         MOV DX, newline       ;换行
23         MOV AH, 09H
24         INT 21H
25         MOV DX, msg2          ;显示输入字符串
26         MOV AH, 09H
27         INT 21H
28         RET

```

【运行】

```

howasm\ch06> nasmw repeat2.asm -o repeat2.com <Enter>
howasm\ch06> repeat2 <Enter>
howasm\ch06> 请输入一个字符串: abcde <Enter>
               您刚输入的字符串= abcde

```

课后习题

- 分别从键盘输入两个字符存入内存变量 ch1 及 ch2 中, 请将下列比较的结果输出至屏幕。
若 $ch1 < ch2$ 则显示 “ch1 is less than ch2”
若 $ch1 = ch2$ 则显示 “ch1 is equal to ch2”
若 $ch1 > ch2$ 则显示 “ch1 is greater than ch2”
- 定义 a 为有符号字节变量, 它的初始值为 1。请将下列比较的结果输出至屏幕。
(a). a 与 0 比较
(b). a-1 后与 0 比较
(c). a-2 后与 0 比较
- 定义 a 为有符号字节变量, 它的初始值为 1。请将下列比较的结果, 将标志寄存器的符号标志输出至屏幕。
(a). a 与 0 比较

(b). a-1 后与 0 比较

(c). a-2 后与 0 比较

输出的格式如下:

a (+1) is not with sign bit

a (0) is not with sign bit

a (-1) is with sign bit

- 设计一个程序求出 $\text{sum} = 1+3+5+7+9$ 的值。
- 设计一个程序求出 $\text{sum} = 1+3+5+\dots+99$ 的值。
- 从键盘输入一个字符 ch2, 若 ch2 为 'K' 则输出 "ch2 为英文字母 K", 若 ch2 为 'y' 则输出 "ch 为英文字母 y", 若 ch2 为其他值则输出 "为 K, y 以外的字符"。
- 从键盘输入一个 'a' 至 'd' 的字符 ch2, 将其次数累加至 suma 至 sumd, 再循环输入, 若 ch2 不是 'a' 至 'd' 的字符则将其次数累加输出后停止。
- 从键盘输入一个 'a' 至 'd' 的字符 ch2, 循环输入 10 次, 次数累加至 suma 至 sumd, 若 ch2 不是 'a' 至 'd' 的字符或 10 个字符已经输入完毕, 则将其累加数输出后停止。
- 从键盘输入一个字符 ch2, 若为 'a' 则输出 "is a", 若为 'b' 则输出 "is b", 若为 '0' 则结束, 若为其他值则输出 "a, b, 0 以外其他字符" 后循环输入。

7

算术运算

算术运算是程序设计当中一个很重要的项目，因为几乎每个程序多少都会执行一些计算。在本章除了了解加减乘除的算术四则计算外，还会看到有关整数的处理。





7-1 定点数与浮点数

定点数(fixed-point number)事实上就是我们常说的整数,这个“点”是指小数点(decimal point)的意思,常将小数点固定在整数的个位数后面,小数点后面没有任何数字,常说定点数是要与浮点数相对立的关系,而浮点数是小数点后还有数字的数。下列是定点数与浮点数的几个例子:

1.	0.	-88.	365.	12345678.	-987654321.	定点数
1.2	-9.87	365.25	000246	-8642.13	3.1416	浮点数

若小数点固定在整数的个位数后面,既然是固定的位置当然不需将小数点存储在计算机的内存里,只需存储整数部分,以后就称它为整数(integer)。处理器基本上只能处理整数,若要处理浮点数必须依靠额外的数学协处理器(math coprocessor),这种处理器在 Intel 80486 以上(含)的计算机系统均已经与中央处理器同时安装在一起了,它的用法在后面的章节会详细说明。本章只处理整数的运算。

7-2 带符号及无符号整数

带符号整数(signed number)表示带有符号的整数。无符号整数(unsigned number)表示不带符号的整数,它只能表示零及正数而已。带符号整数除了表示零及正数外,它还能表示负数。如表 7-1 所示。

表 7-1 整数存储空间及值域表

类型	位	带符号整数范围	无符号整数范围
byte	8	-128 至 127	0 至 255
word	16	-32768 至 32767	0 至 65535
dword	32	-2147483648 至 2147483647	0 至 4294967295

因为无符号整数只能表示零及正数,因此正整数前面的正号就省略了,如表 7-1 中“无符号整数范围”栏所示,一个字节(byte)表示无符号整数,其范围为 0 至 255,一个字(word)表示无符号整数,其范围为 0 至 65535,一个双字(double word)表示无符号整数,其范围为 0 至 4294967295。带符号整数因为要表示负数,因此将无符号整数的范围切掉一半以表示正数,另一半表示负数。

如果在设计程序时要用到负数,就将它定义为带符号整数,若没有用到负数,就将它定义为无符号整数,例如下列定义三个整数变量 a、b 及 c:

```
a DB 7      ; 定义以字节表示的无符号整数
b DB -5     ; 定义以字节表示的带符号整数
c DB 0      ; 定义以字节表示的无符号整数
```

变量 a 与 c 均定义以字节表示的无符号整数,变量 b 定义以字节表示的带符号整数,其实处理器在执行运算时是不管正数或负数的,一概当无符号整数来处理,那它是怎么办到的呢?很简单,它只是将负数当正数来处理,那如何将负数以正数来表示呢?目前的计算机系统都采用补码法(2's complement)将负数以正数表示。例如-5 的补码表示如下:

字节	0 0 0 0 0 1 0 1	值5
反码	1 1 1 1 1 0 1 0	
+	1	
补码	1 1 1 1 1 0 1 1	值-5

若要计算 $a + b$ 的值，假如 a 为 7， b 为 -5，即 $7 + (-5)$ ，执行加法计算后产生进位，留在字节内的值为 2，就是计算的答案。

	0 0 0 0 0 1 1 1	值7
+	1 1 1 1 1 0 1 1	值-5
	1 0 0 0 0 0 0 1 0	值2

进位放弃

只要是整数的运算，不管是正数或负数，处理器都会正确的执行，就不用太在意是带符号数还是无符号数。要是执行下列的运算：

```
MOV AL, BYTE [a]
ADD AL, BYTE [b]
MOV BYTE [c], AL
```

变量 c 的内容是 2。

若要测试无符号整数是否进位，可使用 `JC` 及 `JNC` 的条件转移指令，但若要测试带符号整数是否进位，要使用 `JO` 及 `JNO` 指令。

7-3 加法及减法

加法及减法的方法较简单，加法使用 `ADD` 指令，减法使用 `SUB` 指令，它们的语法如下：

```
ADD 第一操作数, 第二操作数
SUB 第一操作数, 第二操作数
```

指令 `ADD` 将第二操作数的值加入第一操作数，其结果存于第一操作数中。指令 `SUB` 从第一操作数的值减去第二操作数的值，其结果存于第一操作数中。例如：

```
MOV AL, BYTE [a] ;将变量 a 的内容拷贝至寄存器 AL
ADD AL, BYTE [b] ;将变量 b 的内容加入寄存器 AL
MOV BYTE [c], AL ;将寄存器 AL 拷贝至变量 c
```

上面的三个指令相当于执行 $c = a + b$ 的算术运算。

```
MOV AL, BYTE [a] ;将变量 a 的内容拷贝至寄存器 AL
SUB AL, BYTE [b] ;从寄存器 AL 减去变量 b 的值
MOV BYTE [c], AL ;将寄存器 AL 拷贝至变量 c
```

上面的三个指令相当于执行 $c = a - b$ 的算术运算。



在执行加法或减法时，第一操作数与第二操作数的长度要一致，同是字节，或同是字，或同是双字。不能一个操作数属于字而另一个操作数属于字节，这种情形要将两个操作数调整成同一长度后再执行加法或减法的运算。

```
a DB 7
d DW 3
e DW 0
MOVSX AX, BYTE [a]    ;AX=a
ADD AX, WORD [d]       ;AX=AX+d
MOV WORD [e], AX       ;e=AX
```

上面的例子定义 *a* 为字节变量，其初始值为 7，定义 *d* 为字变量，其初始值为 3，定义 *e* 为字变量，其初始值为 0，现在想计算 $e = a + d$ ，因为 *a* 为字节变量，而 *d* 为字变量，其长度不同，因此先将 *a* 使用 **MOVSX** 指令拷贝至 **AX**，如此字寄存器 **AX** 的值就是 *a* 的值，这个操作已经将一个字节的值转换成一个字了，然后以字相加后存入 *e* 字变量就可以了。

MOV 后面附上 **SX**，**S** 表示 **Sign** 符号的意思，**X** 表示 **eXpansion** 扩展的意思，表示从字节扩展为字，字左边字节以 *a* 的符号位扩展。

```
MOVSX AX, BYTE [a]    ;AX=a
SUB AX, WORD [d]       ;AX=a-d
MOV WORD [e], AX       ;e=a-d
```

减法也是一样作法。另外一个方式是将 *a* 拷贝至 **AL**，将 **AH** 存入 0 值，但若 *a* 为负数时，将会得到一个错误的结果，因为计算机是以补码的方式表示负数的，将 **AH** 清除为 0 则 **AX** 所表示的却是正数，那当然会得到错误的结果，那这个问题如何补救呢？只好使用 **CBW** 指令了。

```
MOV AL, BYTE [b]      ;AL=b
CBW                    ;将 AL 高位扩展至整个 AH
ADD AX, WORD [d]       ;AX=b+d
MOV WORD [e], AX       ;e=b+d
```

变量 *b* 的值为 -5，存入 **AL** 以后其值以二进制表示如下：

AL
11111011

经过 **CBW** 指令执行后 **AX** 的值以二进制表示如下：

AH AL
11111111 11111111

它就是 -5 以字的形式存储的补码。

请注意这个 **CBW** 指令只适用于 **AH** 及 **AL** 寄存器而已。**AL** 的值若为正整数，因为正整数的符号位值为 0，若使用 **CBW** 则整个 **AH** 均为 0，不会影响其正确性。若想将字转成双字，可以使用 **CWDE** 指令将 **AX** 扩展至 32 位的寄存器 **EAX**。

加法与减法指令当中，有一个从指定的数值减去一，另一个指定的数值加上一，这两个指令为 DEC 及 INC。它的格式如下：

DEC 操作数

INC 操作数

操作数可以是寄存器或内存变量。例如：

INC AL

INC AX

INC SI

INC WORD [total]

INC BYTE [count]

加法与减法指令当中，还有一个改变符号的指令，它就是 NEG 指令。它的格式如下：

NEG 操作数

操作数可以是寄存器或内存变量。例如：

MOV AX, -7

NEG AX ;将 AX 值-7 改变符号为+7

MOV AX, 123

NEG AX ;将 AX 值 123 改变符号为-123

程序 add2byte.asm 将字节变量 a 的值与字节变量 b 的值相加，其和存入字节变量 c 中。

第 16 行使用 btostr 宏将字节变量 c 的值转换成字符串 cstr，然后将该字符串显示在屏幕上。

【程序 add2byte.asm】

```

1 ; ***** add2byte.asm *****
2      ORG 0100H
3      JMP start
4 a      DB 7
5 b      DB -5
6 c      DB 0
7 msg    DB 'c=a+b= '
8 cstr   DB ' ', 13, 10, '$'
9 ;
10 %include "../mymacro/btostr.mac"
11 ;
12 start:
13      MOV AL, [a]          ;AL=a
14      ADD AL, [b]          ;AL=a+b
15      MOV [c], AL          ;c=a+b
16      btostr c, cstr, '$'  ;将 c 值转成带符号十进制数
17      MOV DX, msg          ;显示 msg 字符串(内容 c 值)
18      MOV AH, 09H
19      INT 21H
20      RET

```



【运行】

```
howasm\ch07> nasmw add2byte.asm -o add2byte.com <Enter>
```

```
howasm\ch07> add2byte <Enter>
```

```
c=a+b= 2
```

程序 add2word.asm 将字节变量 b 的值存入 AX 后与字变量 d 的值相加, 其和存入字变量 e 中。第 16 行使用 itostr 宏将字变量 e 的值转换成字符串 estr, 然后将该字符串显示在屏幕上。

【程序 add2word.asm】

```
1 ; ***** add2word.asm *****
2      ORG 0100H
3      JMP start
4 b      DB -5
5 d      DW 3
6 e      DW 0
7 msg    DB 'e=b+d= '
8 estr   DB ' ', 13, 10, '$'
9 ;
10 %include "../mymacro/itostr.mac"
11 ;
12 start:      MOV AL, [b]          ;AL=b
13            CBW                  ;将 AL 高位扩展至整个 AH
14            ADD AX, [d]          ;AX=b+d
15            MOV [e], AX         ;e=b+d
16            itostr e, estr, '$' ;将 e 值转成十进制数
17            MOV DX, msg         ;显示 msg 字符串 (内容 e 值)
18            MOV AH, 09H
19            INT 21H
20            RET
```

【运行】

```
howasm\ch07> nasmw add2word.asm -o add2word.com <Enter>
```

```
howasm\ch07> add2word <Enter>
```

```
e=b+d= -2
```

程序 add2dw.asm 将字节变量 a 的值存入 EAX 后与双字变量 f 的值相加, 其和存入双字变量 g 中。第 18 行使用 ltostr 宏将双字变量 g 的值转换成字符串 gstr, 然后将该字符串显示在屏幕上。

【程序 add2dw.asm】

```
1 ; ***** add2dw.asm *****
2      ORG 0100H
3      JMP start
4 a      DB 7
```

```

5 f      DD 123456789
6 g      DD 0
7 msg    DB 'g=a+f= '
8 gstr   DB ' ', 13, 10, '$'
9 ;
10 %include "../mymacro/ltostr.mac"
11 ;
12 start:
13     MOV AL, [a] ;AL=a
14     CBW                ;将 AL 高位扩展至整个 AH
15     CWDE               ;将 AX 高位扩展至整个 EAX
16     ADD EAX, [f]        ;EAX=a+f
17     MOV [g], EAX        ;g=a+f
18     ltostr g, gstr, '$' ;将 g 值转成十进制数
19     MOV DX, msg         ;显示 msg 字符串(内容 e 值)
20     MOV AH, 09H
21     INT 21H
22     RET

```

【运行】

```
howasm\ch07> nasmw add2dw.asm -o add2dw.com <Enter>
```

```
howasm\ch07> add2dw <Enter>
```

```
g=a+f= 123456796
```

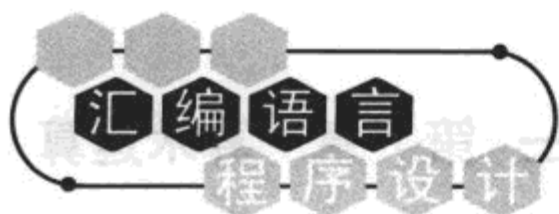
程序 sub2word.asm 将字节变量 b 的值存入 AX 后减去字变量 d 的值, 其结果存入字变量 e 中。第 16 行使用 itostr 宏将字变量 e 的值转换成字符串 estr, 然后将该字符串显示在屏幕上。

【程序 sub2word.asm】

```

1 ; ***** sub2word.asm *****
2     ORG 0100H
3     JMP start
4 b    DB -5
5 d    DW 3
6 e    DW 0
7 msg  DB 'e=b-d= '
8 estr DB ' ', 13, 10, '$'
9 ;
10 %include "../mymacro/itostr.mac"
11 ;
12 start:    MOV AL, [b] ;AL=b
13     CBW                ;将 AL 高位扩展至整个 AH
14     SUB AX, [d]         ;AX=b-d
15     MOV [e], AX        ;e=b-d
16     itostr e, estr, '$' ;将 e 值转换成十进制数

```



```
17      MOV DX, msg           ;显示 msg 字符串(内容 e 值)
18      MOV AH, 09H
19      INT 21H
20      RET
```

【运行】

```
howasm\ch07> nasmw sub2word.asm -o sub2word.com <Enter>
howasm\ch07> sub2word <Enter>
e=b+d= -8
```

7-4 乘法

处理器提供两个乘法指令，**MUL** 指令针对无符号整数的相乘，**IMUL** 指令则针对带符号整数的相乘，它的格式如下：

MUL 操作数

IMUL 操作数

操作数可以是寄存器或内存变量，8 位、16 位或 32 位均可。8 位操作数的值与 **AL** 相乘后其乘积存入 **AX** 中。16 位操作数的值与 **AX** 相乘后其乘积存入 **DX:AX** 中。32 位操作数的值与 **EAX** 相乘后其乘积存入 **EDX:EAX** 中。

程序 **mul2byte.asm** 执行两个字节相乘，其乘积存于 **AX** 中。第 12 行将字节变量 **a** 的值存入 **AL**，第 13 行将 **AL** 值乘上字节变量 **b** 的值，将乘积存入 **AX** 中，第 14 行将乘积存入字变量 **c** 中，第 15 行将字变量 **c** 值转换成十进制数，最后将结果显示在屏幕上。

【程序 mul2byte.asm】

```
1 ; ***** mul2byte.asm *****
2      ORG 0100H
3      JMP start
4 a      DB 7
5 b      DB -5
6 c      DW 0
7 msg    DB 'c=a*b= '
8 cstr    DB ' ', 13, 10, '$'
9 ;
10 %include "../mymacro/itostr.mac"
11 ;
12 start:      MOV AL, [a]           ;AL=a
13            IMUL BYTE [b]         ;AX=a*b
14            MOV [c], AX           ;c=a*b
15            itostr c, cstr, '$'   ;将 c 转换成字符串
16            MOV DX, msg           ;显示 msg 字符串(内容 c 值)
17            MOV AH, 09H
18            INT 21H
19            RET
```

【运行】

```
howasm\ch07> nasmw mul2byte.asm -o mul2byte.com <Enter>
howasm\ch07> mul2byte <Enter>
c=a*b= -35
```

程序 mul2word.asm 执行两个字相乘，其乘积存于 DX:AX 中，其中 DX 为乘积的高位字，AX 为乘积的低位字。第 12 行将字变量 a 的值存入 AX。第 13 行 AX 值乘上字变量 b 的值，将乘积存入 DX:AX 中。第 14 行将乘积低位字存入双字变量 f 低位字，第 15 行将乘积高位字存入双字变量 f 高位字。第 16 行将双字变量 f 值转换成十进制数，最后将结果显示在屏幕上。

【程序 mul2word.asm】

```
1 ; ***** mul2word.asm *****
2      ORG 0100H
3      JMP start
4 a      DW 1234
5 b      DW 5678
6 f      DD 0
7 msg    DB 'f=a*b= '
8 fstr   DB ' ', 13, 10, '$'
9 ;
10 %include "../mymacro/ltostr.mac"
11 ;
12 start: MOV AX, [a]           ;AX=a
13      MUL WORD [b]           ;DX:AX=a*b
14      MOV [f], AX            ;将乘积低位字存入 f 低位字
15      MOV [f+2], DX          ;将乘积高位字存入 f 高位字
16      ltostr f, fstr, '$'    ;将 f 值转换成十进制数
17      MOV DX, msg            ;显示 msg 字符串(内容 f 值)
18      MOV AH, 09H
19      INT 21H
20      RET
```

【运行】

```
howasm\ch07> nasmw mul2word.asm -o mul2word.com <Enter>
howasm\ch07> mul2word <Enter>
f=a*b= 7006652
```

7-5 除法

处理器提供两个除法指令，DIV 指令针对无符号整数的相除，IDIV 指令则针对带符号整数的相除，它的格式如下：

DIV 操作数

IDIV 操作数

操作数可以是寄存器或内存变量，8 位、16 位或 32 位均可。AX 的值除以 8 位操作数的



值, 其商存在 AL 上, 其余数存于 AH 中。DX:AX 的值除以 16 位操作数的值, 其商存于 AX 中, 其余数存于 DX 中。

EDX:EAX 的值除以 32 位操作数的值, 其商存于 EAX, 其余数存于 EDX 中。

程序 divbyte.asm 中双字变量 a 的值除以字节变量 b 的值, 商存入字节变量 q, 余数存入字节变量 r。q 与 r 数值使用 btostr 宏转换成 qstr 及 rstr 字符串后显示在屏幕上。

【程序 divbyte.asm】

```
1 ; ***** divbyte.asm *****
2      ORG 0100H
3      JMP start
4      a      DW 123
5      b      DB 5
6      q      DB 0
7      r      DB 0
8      msgq    DB '商数 q=123/5= '
9      qstr    DB ' ', '$'
10     msgr    DB 13, 10, '余数 r=123%5= '
11     rstr     DB ' ', '$'
12 ;
13 %include "../mymacro/btostr.mac"
14 ;
15 start: MOV AX, [a]           ;AX=a
16        DIV BYTE [b]         ;q=a/b,r=a%b
17        MOV [q], AL          ;q=a/b
18        MOV [r], AH          ;r=a%b
19        btostr q, qstr, '$'   ;将 q 值转换成十进制数
20        MOV DX, msgq         ;显示 msg 字符串(内容 q 值)
21        MOV AH, 09H
22        INT 21H
23 ;
24        btostr r, rstr, '$'   ;将 r 值转换成十进制数
25        MOV DX, msgr         ;显示 msg 字符串(内容 r 值)
26        MOV AH, 09H
27        INT 21H
28        RET
```

【运行】

```
howasm\ch07> nasmw divbyte.asm -o divbyte.com <Enter>
howasm\ch07> divbyte <Enter>
howasm\ch07> 商数 q=123/5= 24
               余数 r=123%5= 3
```

程序 divword.asm 中双字变量 a 的值除以字变量 b 的值, 商存入字变量 q, 余数存入字变量 r。q 与 r 数值使用 itostr 宏转换成 qstr 及 rstr 字符串后显示在屏幕上。

【程序 divword.asm】

```

1 ; ***** divword.asm *****
2     ORG 0100H
3     JMP start
4 a     DD 65536
5 b     DW 12345
6 q     DW 0
7 r     DW 0
8 msgq  DB '商数 q=65536/12345= '
9 qstr  DB ' ', 13, 10, '$'
10 msgr  DB 13, 10, '余数 r=65536%12345= '
11 rstr  DB ' ', 13, 10, '$'
12 ;
13 %include "../mymacro/itostr.mac"
14 ;
15 start:  MOV BX, a           ;将双字变量 a 的地址存入 BX
16         MOV AX, [BX]       ;将变量 a 的低位字内容存入 AX
17         MOV DX, [BX+2]     ;将变量 a 的高位字内容存入 DX
18         DIV WORD [b]       ;a/b
19         MOV [q], AX        ;q=a/b 的商
20         MOV [r], DX        ;r=a/b 的余数
21         itostr q, qstr, '$' ;将 q 值转换成十进制数
22         MOV DX, msgq       ;显示 msg 字符串(内容 q 值)
23         MOV AH, 09H
24         INT 21H
25 ;
26         itostr r, rstr, '$' ;将 r 值转换成十进制数
27         MOV DX, msgr       ;显示 msg 字符串(内容 r 值)
28         MOV AH, 09H
29         INT 21H
30         RET

```

【运行】

```

howasm\ch07> nasmw divword.asm -o divword.com <Enter>
howasm\ch07> divword <Enter>
howasm\ch07> 商数 q=65536/12345= 5
               余数 r=65536%12345= 3811

```

7-6 BCD 十进制数运算

带符号整数与无符号整数的运算都是针对二进制整数而言的，但除此之外还有浮点数及十进制数的运算，浮点数的运算在第14章会有详细的介绍，这里只要介绍十进制数的运算。一个十进制数指0至9，只要四个位就可表示，这种表示法称为BCD表示法。一个字



节由 8 位组成，存储两个十进制数称为压缩 (packed) BCD 表示法，存储一个十进制数称为非压缩 (unpacked) BCD 表示法。非压缩 BCD 表示法虽然较浪费内存，它处理起来较为简单，况且现代计算机内存都很庞大，建议尽量使用非压缩 BCD 表示法。

在下面两种情形下使用十进制运算较使用二进制运算占优势：

1. 在某种情况下，例如会计帐目，需要完全正确的数字。
2. 数目用来输入或输出时，BCD 表示法很容易转换成 ASCII 数字。

从表 7-2 可以看出非压缩 BCD 表示法的数字要转换成 ASCII 数字，只须加上十六进制数 30H 就可以了，反之 ASCII 数字要转换成非压缩 BCD 表示法的数字，只须减去十六进制数 30H 就可以。

表 7-2 ASCII 数字与非压缩的 BCD 表示法

ASCII 数字	十六进制	BCD 数字	十六进制
'0'	30	0	00
'1'	31	1	01
'2'	32	2	02
'3'	33	3	03
'4'	34	4	04
'5'	35	5	05
'6'	36	6	06
'7'	37	7	07
'8'	38	8	08
'9'	39	9	09

处理器并没有提供十进制运算的指令，标准的 ADD、SUB、MUL、DIV 等加减乘除运算是针对二进制整数而设的，但却提供将二进制运算的结果调整为十进制的相关指令，对于加减乘等运算先执行标准的运算后再行调整，对于除法要先调整被除数后再进行除法运算。

调整的指令永远针对存储在寄存器 AL 中的内容而调整，因此执行十进制运算时第一个操作数应该存入 AL。执行十进制运算时要注意一些限制：

1. 每次只能执行一个字节的运算。
2. 对于压缩 BCD 数字只能执行加法与减法运算。

7-6-1 BCD 加法

BCD 十进制数相加，每次相加一个字节，可将被加数一个字节先拷贝至寄存器 AL，将加数一个字节值放入 AL，然后再将 AL 值调整为 BCD 数字。

调整时若是非压缩 BCD 数则使用 AAA 指令，第一个 A 表示为 ASCII 码，第二个 A 表示 Adjust 调整的意思，第三个 A 表示 Addition 加法，因此 AAA 表示加法执行后调整为 ASCII 码的指令。

调整时若是压缩 BCD 数则使用 DAA 指令，第一个 D 表示为 Decimal 十进制数，第二个 A 表示 Adjust 调整的意思，第三个 A 表示 Addition 加法，因此 DAA 表示加法执行后

调整为十进制数的指令。

```

a DB 7
b DB 6
p1 DB 98H
p2 DB 05H
...
;非压缩 BCD 数相加 AL <-- a+b
;
    MOV AL, BYTE [a]
    ADD AL, BYTE [b]
    AAA
;
;压缩 BCD 数相加 AL <-- p1+p2
;
    MOV AL, BYTE [p1]
    ADD AL, BYTE [p2]
    DAA

```

上例是两个字节非压缩 BCD 数相加, 及压缩 BCD 数相加的例子。相加的结果均存入 AL。当 AL 的值太大(非压缩 BCD 数超过 9 或压缩 BCD 数超过 99) 时会设定进制标志(CF) 值为 1, AAA 指令同时会加一至 AH, 若 AL 值在 0~9 或 0~99 内, 则 AAA 及 DAA 指令设 CF 为 0 值。

程序 ubcdadd.asm 将两个非压缩 BCD 数 a 与 b 相加, 其结果存入 a 中。最后调用 dispbcd 宏将结果显示在屏幕上。

【程序 ubcdadd.asm】

```

1 ; ***** ubcdadd.asm *****
2     ORG 0100H
3     JMP start
4     len DW 4
5 a     DB 0, 3, 2, 1
6 b     DB 0, 4, 8, 5
7 ;
8 %include "../mymacro/dispbcd.mac"
9 ;
10 start:
11     MOV AL, BYTE [a+3]      ;最低位字节相加
12     ADD AL, BYTE [b+3]      ;二进制相加
13     AAA                     ;调整为 ASCII 数字
14     MOV BYTE [a+3], AL      ;存回相加结果
15 ;
16     MOV AL, BYTE [a+2]      ;低位字节相加
17     ADC AL, BYTE [b+2]      ;带 CF 标志值的二进制相加
18     AAA                     ;调整为 ASCII 数字

```



```
19      MOV BYTE [a+2], AL      ; 存回相加结果
20 ;
21      MOV AL, BYTE [a+1]      ; 最高位字节相加
22      ADC AL, BYTE [b+1]      ; 带 CF 标志值的二进制相加
23      MOV AH, 0               ; 只要进位
24      AAA                     ; 调整为 ASCII 数字
25      MOV BYTE [a+1], AL      ; 存回相加结果
26 ;
27      MOV BYTE [a], AH        ; 最高位字节只存入进位
28 ;
29      dispubcd a, len
30      RET
```

【运行】

```
howasm\ch07> nasmw ubcdadd.asm -o ubcdadd.com <Enter>
```

```
howasm\ch07> ubcdadd <Enter>
```

806

程序 pbcdadd.asm 将两个压缩 BCD 数 a 与 b 相加, 其结果存入 a 中。最后调用 showbyte 宏将结果以十六进制数形式显示在屏幕上。

【程序 pbcdadd.asm】

```
1 ; ***** pbcdadd.asm *****
2      ORG 0100H
3      JMP start
4 a      DB 00H, 56H, 43H, 21H
5 b      DB 00H, 45H, 68H, 54H
6 ;
7 %include "../mymacro/showbyte.mac"
8 ;
9 start:
10      MOV AL, BYTE [a+3]      ; 最低位字节相加
11      ADD AL, BYTE [b+3]      ; 二进制相加
12      DAA                     ; 调整为 ASCII 数字
13      MOV BYTE [a+3], AL      ; 存回相加结果
14 ;
15      MOV AL, BYTE [a+2]      ; 低位字节相加
16      ADC AL, BYTE [b+2]      ; 带 CF 标志值的二进制相加
17      DAA                     ; 调整为 ASCII 数字
18      MOV BYTE [a+2], AL      ; 存回相加结果
19 ;
20      MOV AL, BYTE [a+1]      ; 最高位字节相加
21      ADC AL, BYTE [b+1]      ; 带 CF 标志值的二进制相加
22      DAA                     ; 调整为 ASCII 数字
23      MOV BYTE [a+1], AL      ; 存回相加结果
```

```

24 ;
25     JC set1           ;最高位字节相加后有进位
26     JNC set0          ;最高位字节相加后没有进位
27 set1:
28     MOV BYTE [a], 1
29     JMP next
30 set0:
31     MOV BYTE [a], 0
32 next:
33     MOV SI, 0
34 loop2:
35     showbyte a+SI      ;显示 BCD 数字
36     INC SI
37     CMP SI, 4          ;共 4 位数
38     JNE loop2
39     RET

```

【运行】

```

howasm\ch07> nasmw pbcdadd.asm -o pbcdadd.com <Enter>
howasm\ch07> pbcdadd <Enter>
01021175

```

7-6-2 BCD 减法

BCD 十进制数相减，每次相减一个字节，可将被减数一个字节先拷贝至寄存器 AL 中，从 AL 减去减数一个字节，然后再将 AL 值调整为 BCD 数字。

调整时若是非压缩 BCD 数则使用 AAS 指令，第一个 A 表示为 ASCII 码，第二个 A 表示 Adjust 调整的意思，第三个 S 表示 Subtraction 减法，因此 AAS 表示减法执行后调整为 ASCII 码的指令。

调整时若是压缩 BCD 数则使用 DAS 指令，第一个 D 表示为 Decimal 十进制数，第二个 A 表示 Adjust 调整的意思，第三个 S 表示 Subtraction 减法，因此 DAS 表示减法执行后调整为十进制数的指令。

```

a    DB 7
b    DB 6
p1   DB 98H
p2   DB 05H
...
;非压缩 BCD 数相减 AL <-- a-b
;
    MOV AL, BYTE [a]
    SUB AL, BYTE [b]
    AAS
;

```



;压缩 BCD 数相减 AL <-- p1-p2

;

MOV AL, BYTE [p1]

SUB AL, BYTE [p2]

DAS

上例是两个字节非压缩 BCD 数相减, 及压缩 BCD 数相减的例子。相减的结果均存入 AL。当 AL 的值较小时, AAS 及 DAS 指令会设定进位标志 (CF) 值为 1 以表示借位, AAS 指令同时会从 AH 值减去 1, 若相减的结果 AL 值在 0~9 或 0~99 内, 则 AAS 及 DAS 指令设定 CF 为 0 值。

程序 ubcdsub.asm 将两个非压缩 BCD 数 a 与 b 相减, 其结果存入 a 中。最后调用 dispubcd 宏将结果显示在屏幕上。

【程序 ubcdsub.asm】

```
1 ; ***** ubcdsub.asm *****
2     ORG 0100H
3     JMP start
4 len  DW 4
5 a     DB 0, 5, 2, 1
6 b     DB 0, 4, 8, 4
7 ;
8 %include "../mymacro/dispubcd.mac"
9 ;
10 start:
11     MOV AL, BYTE [a+3]           ;最低位字节相减
12     SUB AL, BYTE [b+3]           ;二进制相减
13     AAS                          ;调整为 ASCII 数字
14     MOV BYTE [a+3], AL          ;存回相减结果
15 ;
16     MOV AL, BYTE [a+2]           ;低位字节相减
17     SBB AL, BYTE [b+2]           ;带 CF 标志值的二进制相减
18     AAS                          ;调整为 ASCII 数字
19     MOV BYTE [a+2], AL          ;存回相加结果
20 ;
21     MOV AL, BYTE [a+1]           ;最高位字节相减
22     SBB AL, BYTE [b+1]           ;带 CF 标志值的二进制相减
23     AAS                          ;调整为 ASCII 数字
24     MOV BYTE [a+1], AL          ;存回相减结果
25 ;
26     dispubcd a, len
27     RET
```

【运行】

howasm\ch07> nasmw ubcdsub.asm -o ubcdsub.com <Enter>

```
howasm\ch07> ubcdadd <Enter>
```

37

程序 pbcsub.asm 将两个压缩 BCD 数 a 与 b 相减,其结果存入 a 中。最后调用 showbyte 宏将结果以十六进制数的形式显示在屏幕上。

【程序 pbcsub.asm】

```
1 ; ***** pbcsub.asm *****
2     ORG 0100H
3     JMP start
4 a     DB 00H, 56H, 43H, 21H
5 b     DB 00H, 45H, 68H, 54H
6 msg   DB 'BCD 相减错误!!', 13, 10, '$'
7 ;
8 %include "../mymacro/showbyte.mac"
9 %include "../mymacro/dispstr.mac"
10 ;
11 start:
12     MOV AL, BYTE [a+3]           ;最低位字节相减
13     SUB AL, BYTE [b+3]           ;二进制相减
14     DAS                           ;调整为 ASCII 数字
15     MOV BYTE [a+3], AL           ;存回相减结果
16 ;
17     MOV AL, BYTE [a+2]           ;低位字节相减
18     SBB AL, BYTE [b+2]           ;带 CF 标志值的二进制相减
19     DAS                           ;调整为 ASCII 数字
20     MOV BYTE [a+2], AL           ;存回相减结果
21 ;
22     MOV AL, BYTE [a+1]           ;最高位字节相减
23     SBB AL, BYTE [b+1]           ;带 CF 标志值的二进制相减
24     DAS                           ;调整为 ASCII 数字
25     MOV BYTE [a+1], AL           ;存回相减结果
26 ;
27     JC suberr                     ;错误发生
28     JNC next
29 suberr:
30     dispstr msg                   ;显示错误信息
31     JMP endjob
32 next:
33     MOV SI, 0
34 loop2:
35     showbyte a+SI                 ;显示 BCD 数字
36     INC SI
37     CMP SI, 4                     ;共 4 位数
```



```
38     JNE loop2
```

```
39 endjob:
```

```
40     RET
```

【运行】

```
howasm\ch07> nasmw pbcsub.asm -o pbcsub.com <Enter>
```

```
howasm\ch07> pbcsub <Enter>
```

```
00107467
```

7-6-3 BCD 乘法

BCD 十进制数相乘，每次相乘一个字节，可将被乘数一个字节先拷贝至寄存器 AL，将乘数一个字节值与 AL 相乘，然后再将 AL 值调整为 BCD 数字。处理器只提供对非压缩 BCD 数值相乘的指令。调整时使用 AAM 指令，第一个 A 表示为 ASCII 码，第二个 A 表示 Adjust 调整的意思，第三个 M 表示 Multiply 乘法，因此 AAM 表示乘法执行后调整为 ASCII 码的指令。相乘的结果高位数存于 AH，低位数存入 AL，若乘积小于或等于 9 则 AH 为 0 值。

```
a DB 7
```

```
b DB 6
```

```
...
```

```
;非压缩 BCD 数相加 AH:AL <-- a*b
```

```
;
```

```
MOV AL, BYTE [a]
```

```
MUL BYTE [b]
```

```
AAM
```

程序 ubcdmul.asm 将两个非压缩 BCD 数 a 与 b 相乘，其结果存入 c 中。最后调用 dispubcd 宏将结果显示在屏幕上。

【程序 ubcdmul.asm】

```
1 ; ***** ubcdmul.asm *****
```

```
2     ORG 0100H
```

```
3     JMP start
```

```
4 len DW 3
```

```
5 a   DB 0, 7, 8
```

```
6 b   DB 2
```

```
7 c   TIMES 80 DB 0
```

```
8 ;
```

```
9 %include "..\mymacro\dispubcd.mac"
```

```
10 ;
```

```
11 start:
```

```
12     MOV CX, WORD [len]
```

;被乘数字节数

```
13     MOV DL, BYTE [b]
```

;乘数

```
14     MOV DI, WORD [len]
```

;乘积最低位字节索引值

```
15     DEC DI
```

```
16     MOV BYTE [c+DI], 00H
```

;乘积最低位字节 0 值

```

17      MOV SI, WORD [len]
18      DEC SI                      ;被乘数最低位字节索引值
19 begin:
20      MOV AL, [a+SI]              ;被乘数第 SI 字节
21      DEC SI                      ;被乘数下一个字节索引值
22      MUL DL                      ;相乘
23      AAM                        ;乘法调整为 BCD 数字
24      ADD AL, [c+DI]              ;加上上次的进位
25      AAA                        ;加法调整为 BCD 数字
26      MOV [c+DI], AL              ;结果存入乘积第 DI 字节
27      DEC DI                      ;乘积下一个字节索引值
28      MOV [c+DI], AH              ;进位存入第 DI 字节
29      LOOP begin                  ;继续
30 ;
31      dispubcd c, len              ;显示乘积, len 位数
32      RET

```

【运行】

```

howasm\ch07> nasmw ubcdmul.asm -o ubcdmul.com <Enter>
howasm\ch07> ubcdmul <Enter>

```

156

7-6-4 BCD 除法

BCD 十进制数相除，每次两个字节除以一个字节，与其他 BCD 运算不同的地方是除法必须先调整后执行除法运算，处理器只提供对非压缩 BCD 数值相除的指令。调整时使用 AAD 指令，第一个 A 表示为 ASCII 码，第二个 A 表示 Adjust 调整的意思，第三个 D 表示 Division 除法，因此 AAD 表示除法执行后调整为 ASCII 码的指令。调整前高位 BCD 数存于 AH，低位 BCD 数存入 AL，指令 AAD 将这两个 BCD 数转换成无符号二进制数存入 AL 中，然后将 AH 清除为 0 值。因为 DIV 指令属于二进制数除法，因此商数存于 AL 而余数存入 AH，若商数大于 9，则必须调整，最简单的方法莫如使用 AAM 指令，它将存于 AL 大于 9 的数除以 10 其十位数存入 AH，个位数存入 AL，当然这样做会把 DIV 除法运算的余数覆盖了。

```

a DB 7, 8
b DB 2
...
;非压缩 BCD 数相除 AH:AL <-- a/b
;
MOV AH, BYTE [a]      ;被除数高位数
MOV AL, BYTE [a+1]    ;被除数低位数
AAD                   ;AL=AH*10+AL, AH=0
DIV BYTE [b]          ;AL/[b]
MOV BL, AH             ;BL=余数

```



AAM

;AH=高位数, AL=低位数

程序ubcddiv.asm 将两个非压缩BCD 数a 与b 相除,其结果存入c中。最后调用dispubcd 宏将结果显示在屏幕上。

【程序 ubcddiv.asm】

```
1 ; ***** ubcddiv.asm *****
2     ORG 0100H
3     JMP start
4 len  DW 2
5 a     DB 7, 8
6 b     DB 2
7 c     TIMES 80 DB 0
8 r     DB 0
9 ;
10 %include "..\mymacro\dispubcd.mac"
11 ;
12 start:
13     MOV CX, [len]           ;被除数字节数
14     MOV DL, [b]             ;除数
15     MOV DI, c               ;商最高位字节地址
16     MOV SI, a               ;被除数最高位字节地址
17     XOR AH, AH              ;AH=0
18 begin:
19     MOV AL, [SI]             ;AL=被除数第 SI 字节
20     INC SI                   ;被除数下一个字节
21     AAD                     ;除法调整
22     DIV DL                   ;相除
23     MOV [DI], AL             ;存入商数第 DI 字节
24     INC DI                   ;商数下一个字节
25     LOOP begin              ;继续
26 ;
27     MOV [r], AL              ;余数存入 r 变量
28 ;
29     dispubcd c, len          ;显示商数, len 位数
30     RET
```

【运行】

```
howasm\ch07> nasmw ubcddiv.asm -o ubcddiv.com <Enter>
howasm\ch07> ubcddiv <Enter>
```

39

7-6-5 BCD 宏应用

下面几个宏是 BCD 数运算常用到的,使用这几个宏使得 BCD 运算的处理方便许多,

当然特殊功能的宏要靠自己动手制作了。

asc2ubcd ascmem, bytelen

将 ascmem 地址的 ASCII 数字转换成非压缩 BCD 数后存回原地址。

ascmem : ASCII 数字地址 (传址)。

bytelen : ASCII 数字字节数 (字, 传址)。

dispubcd ubcdmem, bytelen

将 ubcdmem 地址的非压缩 BCD 数字显示出来, 前边零不显示。

ubcdmem : 非压缩 BCD 数地址 (传址)。

bytelen : 非压缩 BCD 数字字节数 (字, 传址)。

pbcd2str bcdmem, dmem

将 bcdmem 地址的 10 字节压缩 BCD 数转换为 ASCII 数字, 存于 dmem 地址。

bcdmem : 10 字节带符号数 (传址)。

dmem : ASCII 数字地址 (传址)。

str2ubcd smem, endchar, ubcdmem, bytelen

将 smem 地址的 ASCII 数字转换为非压缩 BCD 数后存入 ubcdmem 地址。

smem : ASCII 数字地址 (传址)。

endchar : 字符串结束符号 (传值)。

ubcdmem : ASCII 数字转换为非压缩 BCD 数后存入的地址。

bytelen : BCD 数长度 (字, 传址)。

ubcd2asc ubcdmem, bytelen

将 ubcdmem 地址的非压缩 BCD 数转换为 ASCII 数字后存回原地址。

ubcdmem : 非压缩 BCD 数地址 (传址)。

bytelen : 非压缩 BCD 数字字节数 (字, 传址)。

ubcd2str ubcdmem, bytelen, smem

将 ubcdmem 地址的非压缩 BCD 数转换为 ASCII 数字后存回原地址。

ubcdmem : 非压缩 BCD 数地址 (传址)。

bytelen : 非压缩 BCD 数字字节数 (字, 传址)。

smem : 字符串地址 (传址)。

ubcdadd1 ubcdmem, bytelen

将 ubcdmem 地址的非压缩 BCD 数加 1 后存回原地址。

ubcdmem : 非压缩 BCD 数地址 (传址)。

bytelen : 非压缩 BCD 数字字节数 (字, 传址)。

ubcdmul2 ubcdmem, bytelen

将 ubcdmem 地址的非压缩 BCD 数乘以 2 后存回原地址。

ubcdmem : 非压缩 BCD 数地址 (传址)。

bytelen : 非压缩 BCD 数字字节数 (字, 传址)。

程序 ubcdadd1.asm 调用 ubcdadd1 宏将非压缩 BCD 变量 a 加上 1 后存回原变量, 随即调用 dispubcd 将它的值显示在屏幕上。



【程序 ubcdadd1.asm】

```
1 ; ***** ubcdadd1.asm *****
2     ORG 0100H
3     JMP start
4 len  DW 4
5 a     DB 0, 9, 9, 9
6 ;
7 %include "..\mymacro\dispubcd.mac"
8 %include "..\mymacro\ubcdadd1.mac"
9 ;
10 start:
11     ubcdadd1 a, len      ;BCD 变量 a 加 1
12     dispubcd a, len      ;显示 ASCII 数字
13     RET
```

【运行】

```
howasm\ch07> nasmw ubcdadd1.asm -o ubcdadd1.com <Enter>
howasm\ch07> ubcdadd1 <Enter>
1000
```

程序 ubcdmul2.asm 调用 readstr 宏从键盘输入一个以零开始的十进制数字,调用 str2ubcd 宏将它转成 BCD 数字,调用 ubcdmul2 宏将非压缩 BCD 变量 a 乘上 2 后存回原变量,随即调用 dispubcd 将它的值显示在屏幕上。重复输入直到直接按<Enter> 键才结束。

【程序 ubcdmul2.asm】

```
1 ; ***** ubcdmul2.asm *****
2     ORG 0100H
3     JMP start
4 len  DW 4
5 astr  TIMES 80 DB ' '
6 a     TIMES 80 DB 0
7 msg   DB '请输入一个以零开始的十进制数: ', '$'
8 ;
9 %include "..\mymacro\dispstr.mac"
10 %include "..\mymacro\readstr.mac"
11 %include "..\mymacro\str2ubcd.mac"
12 %include "..\mymacro\ubcdmul2.mac"
13 %include "..\mymacro\str2ubcd.mac"
14 %include "..\mymacro\dispubcd.mac"
15 %include "..\mymacro\newline.mac"
16 ;
17 start:
18     MOV CX, -1           ;永远循环
19 begin:
```

```

20      dispstr msg           ;显示 msg 信息
21      readstr astr          ;输入数字字符串
22      newline               ;换行
23      CMP BYTE [astr], '$'  ;Enter 键?
24      JNE next              ;否
25      JMP endjob            ;是
26 next:
27      str2ubcd astr, '$', a, len ;ASCII 数字转成 BCD
28      ubcdmul2 a, len         ;a*2
29      dispbcd a, len
30      newline
31      JMP begin              ;继续
32 endjob:
33      RET

```

【运行】

```

howasm\ch07> nasmw ubcdmul2.asm -o ubcdmul2.com <Enter>
howasm\ch07> ubcdmul2 <Enter>
      请输入一个以零开始的十进制数: 01111111 <Enter>
      222222
      请输入一个以零开始的十进制数: <Enter>

```

程序 dw2asc.asm 将一个双字变量值以 BCD 数字方式显示出来。例如双字变量的值为 101，它的计算方式如下：

$$(((0 * 2) + 1) * 2 + 0) * 2 + 1$$

初值	高位	次高位	低位

从初值为 0 开始，第一次运算为：

暂时结果(1) = 初值(0) * 2 + 高位值(1)

第二次运算为：

暂时结果(2) = 暂时结果(1) * 2 + 次高位值(0)

第三次运算为：

结果(5) = 暂时结果(2) * 2 + 低位值(1)

最后得到的结果为 5。

程序 dw2asc.asm 将一个双字变量 b，其数值为 0FFFFFFFFH，是无符号整数最大值，利用上述方法将它以 BCD 数字方式显示出来为 4294967295。

【程序 dw2asc.asm】

```

1 ; ***** dw2asc.asm *****
2      ORG 0100H
3      JMP start
4 len  DW 10
5 a     TIMES 10 DB 0

```



```
6 b      DD 0FFFFFFFFH
7 ;
8 %include "..\mymacro\ubcdadd1.mac"
9 %include "..\mymacro\ubcdmul2.mac"
10 %include "..\mymacro\dispubcd.mac"
11 ;
12 start:
13     MOV CX, 32                ;CX=双字共 32 个位
14 loop2:
15     ubcdmul2 a, len           ;BCD 变量 a 共 len 个字节×2
16     SHL DWORD [b], 1         ;b 左移一个位
17     JNC next                  ;没有 CF
18     ubcdadd1 a, len           ;a 加上 CF 的 1 值
19 next:
20     DEC CX                    ;CX=CX-1
21     JCXZ next2                ;若 CX=0 则跳至 next2
22     JMP loop2                 ;继续循环
23 next2:
24     dispubcd a, len           ;显示 BCD 的 a 数字
25     RET
```

【运行】

```
howasm\ch07> nasmw dw2asc.asm -o dw2asc.com <Enter>
```

```
howasm\ch07> dw2asc <Enter>
```

4294967295

程序 `qw2asc.asm` 将两个双字变量 `num1` 及 `num2` 相乘，其乘积高位数存于 `EDX`，低位数存入 `EAX` 寄存器里。将 `EDX` 传送至 `hinum` 双字变量，将 `EAX` 传送至 `lownum` 双字变量。然后将 `hinum` 及 `lownum` 逻辑上看成是一个 64 位的整数，利用上题 `dw2asc.asm` 的方法，以 BCD 数字方式显示出来。本例题中 `num1` 值为 12345678H，`num2` 值为 9ABCDEF0H，相乘的结果以 BCD 数字方式显示出来，为十进制数 792891155752493184。

【程序 `qw2asc.asm`】

```
1 ; ***** qw2asc.asm *****
2     ORG 0100H
3     JMP start
4 len      DW 18
5 a        TIMES 18 DB 0
6 num1     DD 12345678H
7 num2     DD 9ABCDEF0H
8 lownum   DD 0
9 hinum    DD 0
10 ;
11 %include "..\mymacro\dispubcd.mac"
```

```

12 %include "..\mymacro\ubcdadd1.mac"
13 %include "..\mymacro\ubcdmul2.mac"
14 ;
15 start:
16     MOV EAX, DWORD [num1]           ;EAX=num1 值
17     MUL DWORD [num2]                ;EDX:EAX=num1 值*num2 值
18     MOV DWORD [lownum], EAX         ;lownum 乘积低位数
19     MOV DWORD [hinum], EDX          ;hinum 乘积高位数
20 ;
21     MOV BH, 32                      ;BH=32 位
22     MOV BL, 32                      ;BL=32 位
23 loop2:
24     ubcdmul2 a, len                 ;BCD 变量 a*2
25     SHL DWORD [hinum], 1            ;hinum 左移一个位
26     JNC next2                      ;CF?否,跳至 next2
27     ubcdadd1 a, len                 ;是,a 加上 CF
28 next2:
29     DEC BH                          ;BH=BH-1
30     JZ loop3                       ;跳出 loop2 进入 loop3
31     JMP loop2                      ;继续循环
32 loop3:
33     ubcdmul2 a, len                 ;BCD 变量 a*2
34     SHL DWORD [lownum], 1           ;lownum 左移一个位
35     JNC next3                      ;CF?否,跳至 next3
36     ubcdadd1 a, len                 ;是,a 加上 CF
37 next3:
38     DEC BL                          ;BL=BL-1
39     JZ next                        ;跳出 loop3 进入 next
40     JMP loop3                      ;继续循环
41 next:
42     dispubcd a, len                 ;显示 ASCII 数字
43     RET

```

【运行】

```
howasm\ch07> nasmw qw2asc.asm -o qw2asc.com <Enter>
```

```
howasm\ch07> qw2asc <Enter>
```

```
792891155752493184
```

7-7 综合例题**【例一】**

从键盘输入三个整数 num1、num2 及 num3, 其值在-128 至 127 之间, 求其总和 sum 及平均值 avg。



程序 avgbyte.asm 第 23~27 行从键盘输入一个整数字符串 s 后, 使用 strtob 宏将它转换成带符号字节整数值 num1。第 29~33 行从键盘输入一个整数字符串 s 后, 使用 strtob 宏将它转换成带符号字节整数值 num2。第 35~39 行从键盘输入一个整数字符串 s 后, 使用 strtob 宏将它转换成带符号字节整数值 num3。第 47 行使用 itostr 宏转换成 sumstr 字符串后显示在屏幕上。第 52~59 行计算平均值 avg 后使用 btostr 宏转换成 avgstr 字符串后显示在屏幕上。

【程序 avgbyte.asm】

```
1 ; ***** avgbyte.asm *****
2         ORG 0100H
3         JMP start
4 msg     DB 13, 10, '请输入一个整数(-128..127) : ', '$'
5 s       TIMES 81 DB ' '
6 ;
7 num1    DB 0
8 num2    DB 0
9 num3    DB 0
10 avg     DB 0
11 sum     DW 0
12 ;
13 msgsum  DB 13, 10, '总和= '
14 sumstr  DB ' ', '$'
15 msgavg  DB 13, 10, '平均= '
16 avgstr  DB ' ', '$'
17 ;
18 %include "../mymacro/readstr.mac"
19 %include "../mymacro/strtob.mac"
20 %include "../mymacro/btostr.mac"
21 %include "../mymacro/itostr.mac"
22 ;
23 start:  MOV DX, msg                ;显示 msg 字符串
24         MOV AH, 09H
25         INT 21H
26         readstr s
27         strtob s, '$', num1       ;将字符串 s 转换为数值 num1
28 ;
29         MOV DX, msg                ;显示 msg 字符串
30         MOV AH, 09H
31         INT 21H
32         readstr s
33         strtob s, '$', num2       ;将字符串 s 转换为数值 num2
34 ;
35         MOV DX, msg                ;显示 msg 字符串
36         MOV AH, 09H
```

```

37      INT 21H
38      readstr s
39      strtob s, '$', num3      ;将字符串 s 转换为数值 num3
40 ;
41      MOV SX AX, [num1]      ;AX=num1
42      ADD [sum], AX          ;sum=sum+num1
43      MOV SX AX, [num2]      ;AX=num2
44      ADD [sum], AX          ;sum=sum+num2
45      MOV SX AX, [num3]      ;AX=num3
46      ADD [sum], AX          ;sum=sum+num3
47      itostr sum, sumstr, '$' ;将 sum 转换成十进制数
48      MOV DX, msgsum         ;显示 sum
49      MOV AH, 09H
50      INT 21H
51 ;
52      MOV AX, [sum]          ;AX=sum
53      MOV BL, 3              ;BL=3
54      DIV BL                 ;sum/3
55      MOV [avg], AL          ;avg=sum/3
56      btostr avg, avgstr, '$' ;将 avg 转换成十进制数
57      MOV DX, msgavg        ;显示 avg
58      MOV AH, 09H
59      INT 21H
60      RET

```

【运行】

```

howasm\ch07> nasmw avgbyte.asm -o avgbyte.com <Enter>
howasm\ch07> avgbyte <Enter>
howasm\ch07> 请输入一个整数(-128..127) : 10 <Enter>
               请输入一个整数(-128..127) : -30 <Enter>
               请输入一个整数(-128..127) : 50 <Enter>
               总和= 30
               平均= 10

```

【例二】

从键盘输入 count 个整数，每输入一个整数就将它累加至字变量 sum，输入完毕后再求其平均值然后输出。

程序 avg4num.asm 第 7 行设定 count 的值为 4，表示要从键盘输入四个整数，第 28 行使用 readstr 宏从键盘读入一个整数字符串至 s，第 29 行将它转换成数值 num，随即累加至字变量 sum，第 32~34 行本来用一个 LOOP 指令就可以的，但 LOOP 指令属于短距离的转移，而我们要重复执行的指令从第 25~31 行，其指令所占的内存已经超过 128 个字节了，因此不能使用 LOOP 指令，只好用第 32~34 行三个指令替代了。第 36~39 行将 sum 转成字符串后显示在屏幕上。第 41~48 行求得平均 avg 后转成字符串显示在屏幕上。



【程序 avg4num.asm】

```
1 ; ***** avg4num.asm *****
2     ORG 0100H
3     JMP start
4 msg  DB 13, 10, '请输入一个整数(-128..127) : ', '$'
5 s    TIMES 81 DB ' '
6 ;
7 count DB 4
8 num   DB 0
9 avg   DB 0
10 sum  DW 0
11 ;
12 msgsum DB 13, 10, '总和= '
13 sumstr DB ' ', '$'
14 msgavg DB 13, 10, '平均= '
15 avgstr DB ' ', '$'
16 ;
17 %include "../mymacro/readstr.mac"
18 %include "../mymacro/strtob.mac"
19 %include "../mymacro/btostr.mac"
20 %include "../mymacro/itostr.mac"
21 ;
22 start:
23     MOV SX CX, BYTE [count]           ;CX=count 值
24 label:
25     MOV DX, msg                       ;显示 msg 字符串
26     MOV AH, 09H
27     INT 21H
28     readstr s                         ;输入一个字符串至 s
29     strtob s, '$', num                ;将字符串 s 转换为数值 num
30     MOV SX AX, [num]                  ;AX=num
31     ADD [sum], AX                     ;sum=sum+num
32     DEC CX                             ;CX=CX-1
33     JCXZ next                         ;CX=0 时跳出
34     JMP label                         ;重复 count 次
35 next:
36     itostr sum, sumstr, '$'           ;将 sum 转换成十进制数
37     MOV DX, msgsum                   ;显示 sum
38     MOV AH, 09H
39     INT 21H
40 ;
41     MOV AX, [sum]                     ;AX=sum
42     MOV BL, BYTE [count]              ;BL=count
```

```

43          DIV BL                      ;sum/4
44          MOV [avg], AL                ;avg=sum/4
45          btostr avg, avgstr, '$'      ;将 avg 转换成十进制数
46          MOV DX, msgavg               ;显示 avg
47          MOV AH, 09H
48          INT 21H
49          RET

```

【运行】

```

howasm\ch07> nasmw avg4num.asm -o avg4num.com <Enter>
howasm\ch07> avg4num <Enter>
howasm\ch07> 请输入一个整数(-128..127) : 12 <Enter>
               请输入一个整数(-128..127) : 34 <Enter>
               请输入一个整数(-128..127) : 56 <Enter>
               请输入一个整数(-128..127) : -12 <Enter>
               总和= 90
               平均= 22

```

【例三】

从键盘输入三个整数，每输入一个整数就将它存到整数数组 `num` 中，输入完毕以后求整个数组元素总和，再求其平均值输出。

程序 `avgary.asm` 第 33 行将所输入的整数字符串转换为数值 `number` 后存入寄存器 `AL`，第 34 行将 `AL` 存入整数数组 `num` 中的第 `SI` 索引处，刚开始时 `SI` 的值为 0，因此第一个整数存入 `num` 处，接着 `SI` 值增 1，因此第二个整数存入 `num + 1` 处，接着 `SI` 值增 1，因此第三个整数存入 `num + 2` 处。构成一个整数数组。第 40~46 行计算整个数组元素的总和，并存入字变量 `sum`。第 48~51 行将 `sum` 转成字符串后显示在屏幕上。第 53~60 行求得平均 `avg` 后转换成字符串显示于屏幕。

【程序 avgary.asm】

```

1 ; ***** avgary.asm *****
2          ORG 0100H
3          JMP start
4 msg      DB 13, 10, '请输入一个整数(-128..127) : ', '$'
5 s        TIMES 81 DB ' '
6 ;
7 number   DB 0
8 num      DB 0, 0, 0
9 avg      DB 0
10 sum     DW 0
11 numptr   DW num
12 ;
13 msgsum   DB 13, 10, '总和= '
14 sumstr   DB ' ', '$'

```

```

15 msgavg DB 13, 10, '平均= '
16 avgstr DB ' ', '$'
17 ;
18 %include "../mymacro/readstr.mac"
19 %include "../mymacro/strtob.mac"
20 %include "../mymacro/btostr.mac"
21 %include "../mymacro/itostr.mac"
22 ;
23 start:
24     MOV CX, 3
25     MOV SI, 0
26 label:
27     MOV DX, msg ;显示 msg 字符串
28     MOV AH, 09H
29     INT 21H
30     readstr s ;输入一个字符串至 s
31     strtob s, '$', number ;字符串 s 转换成整数
32 ;
33     MOV AL, BYTE [number] ;AL=number
34     MOV BYTE [num+SI], AL ;存入 num 第 SI 个数
35     INC SI ;SI=SI+1
36     DEC CX ;CX=CX-1
37     JCXZ next ;CX=0 时跳出
38     JMP label ;继续循环
39 next:
40     MOV CX, 3 ;CX=3
41     MOV SI, 0 ;SI=0
42 loop3:
43     MOVSX AX, BYTE [num+SI] ;AX=num 第 SI 个数
44     ADD WORD [sum], AX ;累加至 sum
45     INC SI ;SI=SI+1
46     LOOP loop3 ;继续循环
47 ;
48     btostr sum, sumstr, '$' ;将 sum 转换成十进制数
49     MOV DX, msgsum ;显示 sum
50     MOV AH, 09H
51     INT 21H
52 ;
53     MOV AX, [sum] ;AX=sum
54     MOV BL, 3 ;BL=3
55     DIV BL ;sum/3
56     MOV [avg], AL ;avg=sum/3
57     btostr avg, avgstr, '$' ;将 avg 转换成十进制数

```

```

58      MOV DX, msgavg          ;显示 avg
59      MOV AH, 09H
60      INT 21H
61      RET

```

【运行】

```
howasm\ch07> nasmw avgary.asm -o avgary.com <Enter>
```

```
howasm\ch07> avgary <Enter>
```

```
howasm\ch07> 请输入一个整数(-128..127) : 12 <Enter>
```

```
请输入一个整数(-128..127) : 34 <Enter>
```

```
请输入一个整数(-128..127) : 56 <Enter>
```

```
总和= 102
```

```
平均= 34
```

课后习题

1. 从键盘输入两个整数到 a、b，计算 $c = a + b$ ，将 a、b、c 输出至屏幕。
2. 重复从键盘输入一个整数至 a，将 a 累积至 sum，直至输入 0 为止，最后将 sum 输出至屏幕。
3. 从键盘输入一个整数 n，若 n 为正数则输出“n 值为正整数”，否则输出“n 值不是正整数”。
4. 从键盘输入一个整数 n，若 n 为负数则输出“n 值为负整数”，否则输出“n 值不是负整数”。
5. 从键盘输入一个整数 a，请输出 a 的绝对值。
6. 从键盘输入一个整数 a，若 a 为 2 则输出“为正整数 2”，若 a 为 3 则输出“为正整数 3”，若 a 为其他值则输出“为 2，3 以外的整数”。
7. 从键盘输入一个语文成绩 a，若 $a \geq 60$ 则输出“及格”，否则输出“不及格”。
8. 某一军营有 1024 个士兵，指挥官每日派出营中一半的士兵，请问几日后军营只剩下一个士兵。
9. 某人坐出租车，车钱 a 元，拿一张一百元钞票给司机，假设司机只有十元、五元和一元面值的钞票。请问司机该找他几张十元，几个五元，几个一元。a 值从键盘输入。
10. 从键盘输入一个整数 a，若 a 为正整数则累积至 sum，再重复从键盘输入，若 a 为 0 则输出总和 sum 后停止。
11. 从键盘输入一个整数 a，若 a 为正整数则累积至 sum1，再重复从键盘输入，若 a 为负整数则累积至 sum2，再重复从键盘输入，若 a 为 0 则输出总和 sum1，sum2 后停止。
12. 重复从键盘输入一个整数，重复三次，请将循环变量的值及从键盘输入的整数值显示出来。
13. 从键盘输入一个正整数 n，计算 $f = n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1$ ，并将 n 及 f 输出。
14. 从键盘输入一个正整数 n，计算 $s = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$ 之和至小数四位，并将 n 及 s 输出。
15. 从键盘输入一个正奇数 n，计算 $f = n \times (n-2) \times (n-4) \times \dots \times 3 \times 1$ ，并将 n 及 f 输出。



16. 制作一个乘法口诀表。
17. 制作一个八进制乘法表。
18. 制作一个十六进制乘法表。
19. 从键盘输入一个整数 n ，将 n 以二进制方式输出。
20. 从键盘输入一个整数 n ，将 n 以八进制方式输出。
21. 从键盘输入一个整数 n ，将 n 以十六进制方式输出。
22. 若两个压缩 BCD 数为 01H、23H、45H 以及 67H、89H、01H，请设计一个程序将它们相加后以十进制方式显示出来。
23. 若两个压缩 BCD 数为 01H、23H、45H 以及 67H、89H、01H，请设计一个程序将它们相减后以十进制方式显示出来。
24. 若两个非压缩 BCD 数为 1H、2H、3H、4H 以及 5H、6H、7H、8H，请设计一个程序将它们相加后以十进制方式显示出来。
25. 若两个非压缩 BCD 数为 1H、2H、3H、4H 以及 5H、6H、7H、8H，请设计一个程序将它们相减后以十进制方式显示出来。
26. 若非压缩 BCD 数为 1H、2H、3H、4H，请设计一个程序将它加 1 后以十进制方式显示出来。
27. 若非压缩 BCD 数为 1H、2H、3H、4H，请设计一个程序将它乘以 2 后以十进制方式显示出来。
28. 从键盘输入两个五位数非压缩 BCD 数（前面补零），请设计一个程序将它们先后以十进制方式显示出来。
29. 重复从键盘输入五位数非压缩 BCD 数（前面补零），直到按<Enter> 键才结束。请设计一个程序将它们的值求和后以十进制方式显示出来。
30. 设计一个宏格式如下：
ubcdmul a, lena, b, lenb, c, lenc
非压缩 BCD 数 $c=a \times b$
a :被乘数地址
lena :被乘数位数（字，传址）
b :乘数地址
lenb :乘数位数（字，传址）
c :乘积地址
lenc :乘积位数（字，传址）
31. 请设计一个程序使用上题的 ubcdmul 宏，定义参数如下：
lena DW 4
a DB 0,1,2,3
lenb DW 2
b DB 1,2
lenc DW 5
c DB 0,0,0,0,0
显示 $c=a \times b$ 的值。c 的值应为 1476。

8

宏

宏(macro)可以说是许多指令的集合,它是 micro 微小的反义词,就是巨大的意思,简单的说宏就是一个巨大的指令,在程序设计上是属于函数(function)或预处理的定义文字等,但它又不全然是这些,因此就用另一个名词“宏”来称呼它。

NASM 汇编语言提供一个功能强大的宏处理器(macro processor),该宏处理器支持条件汇编(conditional assembly)、多级文件引入(multilevel file inclusion)、两种不同格式的宏写法(单行宏与多行宏)等预处理功能,属于预处理功能的命令之前都要加上%符号。





8-1 单行宏

单行宏包括%define、%undef 及%assign 指令，说明如下。

8-1-1 %define 指令

单行宏以预处理器指引指令%define 来定义。例如：

```
%define sum(a,b) a+b
```

上例中定义一个单行宏 sum(a,b)，名称为 sum，有两个参数 a 及 b，在程序中若调用 sum(a,b)，它会将 a+b 的值返回。

```
MOV DL, sum(64,1) ;将 64+1 的结果存入 DL
```

上例中使用 sum(a,b) 宏，在 MOV 指令中 a 参数以 64 取代，b 参数以 1 取代，因此 sum(64,1) 宏的展开为 64+1，因此 MOV 的指令为：

```
MOV DL, 64+1 ;将 65 存入 DL 寄存器
```

单行宏也可以定义没有参数的宏，下例定义符号 pi 为 3.14159。

```
%define pi 3.14159
```

%define 指令所定义的宏，其名称英文字母大小写是有区别的，例如 pi、PI、Pi、pI 等均为不同的宏名称。若想宏名称英文字母大小写没有区别，就要使用%ifndef 指令，第一个字母 i 是 ignore 忽略的意思。

程序 define2.asm 第 4 行定义一个单行宏 CHARA，其值为字符常量'A'。第 5 行将 CHARA 传送至 DL，相当于将'A' 传送至 DL 一样。

【程序 define2.asm】

```
1 ; ***** define2.asm *****
2     ORG 0100H
3     JMP start
4 %define CHARA 'A' ;定义单行宏 CHARA
5 start:  MOV DL, CHARA ;将 CHARA 存入 DL
6     MOV AH, 02H ;设定显示字符功能
7     INT 21H ;显示存于 DL 的字符
8     RET
```

【运行】

```
howasm\ch08> nasmw define2.asm -o define2.com <Enter>
howasm\ch08> define2 <Enter>
A
```

程序 define3.asm 第 4 行定义一个单行宏 sum，有两个参数，一个命名为 a，另一个为 b，其返回值为 a+b。这个定义只是一个结构而已，调用它并给它两个相对参数值后才会返回实际的值。例如第 5 行调用 sum 时给它两个相对参数值 64 及 1，参数值 64 取代参数 a，参数值 1 取代参数 b，它的返回值 64+1 会取代 sum(64,1)，因此下列指令：

```
MOV DL, sum(64,1)
```

就相当于下列指令：

```
MOV DL, 64+1
```

【程序 define3.asm】

```
1 ; ***** define3.asm *****
2      ORG 0100H
3      JMP start
4      %define sum(a,b) a+b      ;定义单行宏 sum
5      start: MOV DL, sum(64,1)  ;将 64+1 存入 DL
6      MOV AH, 02H              ;设定显示字符功能
7      INT 21H                  ;显示存于 DL 的字符
8      RET
```

【运行】

```
howasm\ch08> nasmw define3.asm -o define3.com <Enter>
```

```
howasm\ch08> define3 <Enter>
```

A

8-1-2 %undef 指令

%undef 指令取消%define 所定义的宏，例如：

```
%define pi 3.14159      ;定义 pi 符号
...                      ;其他程序代码
%undef pi                ;取消%define 所定义的宏 pi
```

8-1-3 %assign 指令

%assign 指令定义一个单行宏，没有参数，但定义一个变量。这个变量值可用表达式方式表示，当%assign 指令被执行时，计数一次。%assign 所定义的变量可以重新定义，如下例：

```
%assign i 0              ;指定变量 i 初值为 0
%rep 3                   ;循环开始(重复三次)
    DB i                  ;定义一个字节初值为变量 i 的值
    %assign i i+1          ;指定变量 i 的值为(i+1)
%endrep                  ;循环结束
```

上例中首先指定变量 i 初始值为 0，接着循环重复三次，每一次定义一个字节初始值为变量 i 的值后，将指定变量 i 的值设为 (i+1)，然后又循环，因此 DB i 共执行三次，第一次循环时 i 值为 0，第二次循环时 i 值为 1，第三次循环时 i 值为 2，结果相当于下列的三个指令：

```
DB 0      ;第一次循环
DB 1      ;第二次循环
DB 2      ;第三次循环
```

程序 assign2.asm 第 6 行定义指定变量 i 的初始值为 65。第 7~10 行构成一个循环，重复三次，每一次 i 的值会增 1，因此三次执行完毕后就相当于如下的定义：

```
DB 65
```



DB 66

DB 67

【程序 assign2.asm】

```
1 ; ***** assign2.asm *****
2             ORG 0100H
3             JMP start
4 begin       DB ' '
5 ;-----
6 %assign i 65
7 %rep 3
8             DB i
9             %assign i i+1
10 %endrep
11 ;-----
12            DB '$'
13 start:      MOV DX, begin+1
14            MOV AH, 09H
15            INT 21H
16            RET
```

【运行】

howasm\ch08> nasmw assign2.asm -o assign2.com <Enter>

howasm\ch08> assign2 <Enter>

ABC

若还不清楚，可制作一个列表文件检查一下。

howasm\ch08> nasmw assign2.asm -l assign2.lst <Enter>

从产生的列表文件 assign2.lst 第 11 行可看出该字节的值为十六进制数 41，它就是 ASCII 码里的'A'。第 13 行是'B'。第 15 行是'C'。

【文件 assign2.lst】

```
1 ; ***** assign2.asm *****
2             ORG 0100H
3 00000000 E90500      JMP start
4 00000003 20  begin   DB ' '
5 ;-----
6             %assign i 65
7 %rep 3
8             DB i
9             %assign i i+1
10 %endrep
11 00000004 41 <1>      DB i
12             <1>      %assign i i+1
13 00000005 42 <1>      DB i
```

```

14          <1>          %assign i i+1
15 00000006 43 <1>      DB i
16          <1>          %assign i i+1
17          ;-----
18 00000007 24          DB '$'
19 00000008 BA[0400] start: MOV DX, begin+1
20 0000000B B409        MOV AH, 09H
21 0000000D CD21        INT 21H
22 0000000F C3          RET

```

8-2 多行宏

多行宏的格式如下：

%MACRO 宏名称 参数个数

宏定义体

%ENDMACRO

8-2-1 显示字符串宏

下列 dispstr 宏属于多行宏，宏名称为 dispstr，参数个数为 1，宏定义体为位于保留字 %MACRO 及 %ENDMACRO 之间的 5 个指令。

【宏 dispstr.mac】

```

; ***** dispstr.mac *****
;
; dispstr smem
; 将 smem 内存处字符串显示于屏幕
; smem : 字符串所存放内存地址
;
%ifndef DISPSTR_MAC
%define DISPSTR_MAC
%MACRO dispstr 1          ; 参数 smem 地址
    PUSHA                ; 存储原来寄存器数据
    MOV DX, %1            ; 字符串地址存入 DX
    MOV AH, 09H           ; 设定显示字符串功能
    INT 21H               ; 显示字符串
    POPA                  ; 恢复原来寄存器数据
%ENDMACRO
%endif

```

dispstr 宏必须提供一个参数，参数的名称为 %1。首先执行 PUSH 指令，存入原来所有一般用途寄存器数据（将现在的值保存在堆栈的顶端），接着将要显示在屏幕上的字符串地址存入 DX 寄存器，这个地址被实际调用这个宏的地址参数所取代。将 09H 存入 AH 后调用 21H 号中断，表示要将存于 DX 地址处的字符串显示在屏幕上。最后从堆栈顶端弹出原来寄



存器的值后，返回调用此宏的程序。

宏使用下列的结构是预防被重复加载，重复加载时名称相同，汇编时会产生错误的。若 DISPSTR_MAC 没有定义过，则定义它，并定义一个 dispstr 宏，若 DISPSTR_MAC 已经定义过，则什么事也不做。保证程序里最多只有一个宏 dispstr。

```
%ifndef DISPSTR_MAC
#define DISPSTR_MAC
%MACRO dispstr 1
...
%ENDMACRO
#endif
```

这个 dispstr.mac 宏其路径如下：

\\howasm\\mymacro\\dispstr.mac

调用的程序可作如下的设计：

```
msg DB '所显示的信息= ', '$'
...
dispstr msg
```

将在屏幕上显示字符串“所显示的信息=”，'\$' 为字符串结束符号，并不显示出来。

8-2-2 显示字符宏

宏 dispchr 将指定内存字节的内容以字符形式显示在屏幕上。

【宏 dispchr.mac】

```
; ***** dispchr.mac *****
;
; dispchr bmem
; 将 bmem 内存内容以字符形式显示于屏幕
; bmem : 字符所存放内存地址
;
%ifndef DISPCHR_MAC
#define DISPCHR_MAC
%MACRO dispchr 1
    PUSHAD                ;参数 bmem 地址
    MOV DL, [bmem]        ;存储原来寄存器数据
    MOV AH, 02H           ;要显示的字符存入 DL
    INT 21H               ;设定显示字符功能
    POPAD                 ;显示存于 DL 的字符
                           ;恢复原来寄存器数据
%ENDMACRO
#endif
```

宏 dispchr 必须提供一个参数。首先在堆栈的顶端存入所有一般用途寄存器的值，将参数的内容存入寄存器 DL 后，将 02H 存入 AH 然后调用 21H 号中断，表示要将 DL 值显示在屏幕上，最后从堆栈顶端弹出所有原来的寄存器值后才返回调用此宏的程序。

调用的程序可作如下的设计：

```
ch2 DB 'C'
```

```
dispchr ch2
```

将在屏幕上显示字符'C'。

8-2-3 读取字符宏

宏 readchr 从键盘读取一个字符，并存放于指定的内存地址。

【宏 readchr.mac】

```
;***** readchr.mac *****
;
; readchr bmem
; 从键盘读取一个字符存放于 bmem 内存地址
; bmem : 读入字符所存放内存地址
;
%ifndef READCHR_MAC
#define READCHR_MAC
%MACRO readchr 1          ;一个参数
    PUSHA                ;保存原来寄存器值
    MOV AH, 01H           ;从键盘读取一个字符
    INT 21H               ;存入寄存器 AL
    MOV [%1], AL          ;字符存入参数地址
    POPA                  ;恢复原来寄存器值
%ENDMACRO
#endif
```

宏 readchr 只有一个参数。首先在堆栈顶端压入所有一般用途寄存器的值，将 01H 存入 AH 后调用 21H 号中断，表示要从键盘读取一个字符至寄存器 AL，然后将 AL 内容存入第一个参数所指地址，最后从堆栈顶端弹出原来的寄存器值后才返回调用此宏的程序。

调用的程序可作如下的设计：

```
ch2 DB ' '
...
readchr ch2
```

调用 readchr 宏从键盘读取一个字符至内存变量 ch2 处。例如输入 C，那么 ch2 变量的值就变为 C 了。

程序 disp.asm 用来测试这三个宏 readchr、dispchr 及 dispstr 是否正确。第 8~10 行从 mymacro 宏目录引入这三个宏文件。第 12 行调用 dispstr 宏，提供的参数地址为 msg，这时 NASM 汇编器会将 dispstr 宏展开，将宏内容的 5 个指令：

```
PUSHA
MOV DX, %1
MOV AH, 09H
INT 21H
POPA
```



取代 dispstr msg 这一行, 请参看 disp.lst 第 64-68 行。程序中若调用两次, 宏内容的五个指令就展开两次, 若调用三次, 宏内容的五个指令就展开三次, 调用愈多次就展开愈多次, 程序也就愈加庞大, 不过现代的计算机内存容量都非常巨大, 这一点可不用担心。

第 13 行调用 readchr 宏, 提供的参数地址为 char, 也是将 readchr 宏内容展开。第 14 行又调用 dispstr 宏, 提供的参数地址为 msg2, 也是将 dispstr 宏的内容再展开一次。第 15 行调用 dispchr 宏, 提供的参数地址为 char, 也是将 dispchr 宏的内容展开。请参阅 disp.lst 文件。

【程序 disp.asm】

```
1 ; ***** disp.asm *****
2          ORG 0100H
3          JMP start
4  msg DB 'keyin char: ','$'
5  msg2  DB 13,10,'disp char= ','$'
6  char  DB ' '
7 ;
8 %include "..\mymacro\readchr.mac"
9 %include "..\mymacro\dispchr.mac"
10 %include "..\mymacro\dispstr.mac"
11 ;
12 start:   dispstr msg      ;显示 msg 字符串
13         readchr char      ;读入 char 字符
14         dispstr msg2      ;显示 msg2 字符串
15         dispchr char      ;显示 char 字符
16         RET
```

文件 disp.lst 是 disp.asm 汇编时选-l 选项时产生的列表文件, 从这个列表文件可以看出每次调用一个宏就将这个宏的程序代码展开。

【文件 disp.lst】

```
1 ; ***** disp.asm *****
2          ORG 0100H
3 00000000 E91C00          JMP start
4 00000003 6B6579696E20636861- msg DB 'keyin char: ','$'
5 0000000C 723A2024
6 00000010 0D0A64697370206368- msg2  DB 13,10,'disp char= ','$'
7 00000019 61723D2024
8 0000001E 20          char DB ' '
9
10          %include "..\mymacro\readchr.mac"
11 <1> ;***** readchr.mac *****
12 <1> ;
13 <1> ; readchr bmem
14 <1> ; 从键盘读取一个字符存放于 bmem 内存地址
15 <1> ; bmem : 读入字符所存放内存地址
```

```

16      <1> ;
17      <1> %ifndef READCHR_MAC
18      <1> %define READCHR_MAC
19      <1> %MACRO readchr 1      ;一个参数
20      <1>      PUSHA            ;保存原来寄存器值
21      <1>      MOV AH, 01H      ;从键盘读取一个字符
22      <1>      INT 21H          ;存入寄存器 AL
23      <1>      MOV [%1], AL     ;字符存入参数地址
24      <1>      POPA            ;恢复原来寄存器值
25      <1> %ENDMACRO
26      <1> %endif
27      %include "..\mymacro\dispchr.mac"
28      <1> ; ***** dispchr.mac *****
29      <1> ;
30      <1> ; dispchr bmem
31      <1> ; 将 bmem 内存内容以字符形式显示在屏幕上
32      <1> ; bmem : 字符所存放内存地址
33      <1> ;
34      <1> %ifndef DISPCHR_MAC
35      <1> %define DISPCHR_MAC
36      <1> %MACRO dispchr 1      ;参数 bmem 地址
37      <1>      PUSHA            ;存储原来寄存器数据
38      <1>      MOV DL, [%1]     ;要显示的字符存入 DL
39      <1>      MOV AH, 02H      ;设定显示字符功能
40      <1>      INT 21H          ;显示存于 DL 的字符
41      <1>      POPA            ;恢复原来寄存器数据
42      <1> %ENDMACRO
43      <1> %endif
44      %include "..\mymacro\dispstr.mac"
45      <1> ; ***** dispstr.mac *****
46      <1> ;
47      <1> ; dispstr smem
48      <1> ; 将 smem 内存处字符串显示在屏幕上
49      <1> ; smem : 内存地址
50      <1> ;
51      <1> %ifndef DISPSTR_MAC
52      <1> %define DISPSTR_MAC
53      <1> %MACRO dispstr 1      ;参数 smem 地址
54      <1>      PUSHA            ;存储原来寄存器数据
55      <1>      MOV DX, %1        ;字符串地址存入 DX
56      <1>      MOV AH, 09H      ;设定显示字符串功能
57      <1>      INT 21H          ;显示字符串
58      <1>      POPA            ;恢复原来寄存器数据

```

```

59          <1> %ENDMACRO
60          <1> %endif
61          ;
62          start:  dispstr msg          ;显示 msg 字符串
63          <1> start:
64 0000001F 60          <1>          PUSHA
65 00000020 BA[0300]    <1>          MOV DX, %1
66 00000023 B409        <1>          MOV AH, 09H
67 00000025 CD21        <1>          INT 21H
68 00000027 61          <1>          POPA
69          readchr char          ;读入 char 字符
70 00000028 60          <1>          PUSHA
71 00000029 B401        <1>          MOV AH, 01H
72 0000002B CD21        <1>          INT 21H
73 0000002D A2[1E00]    <1>          MOV [%1], AL
74 00000030 61          <1>          POPA
75          dispstr msg2          ;显示 msg2 字符串
76 00000031 60          <1>          PUSHA
77 00000032 BA[1000]    <1>          MOV DX, %1
78 00000035 B409        <1>          MOV AH, 09H
79 00000037 CD21        <1>          INT 21H
80 00000039 61          <1>          POPA
81          dispchr char          ;显示 char 字符
82 0000003A 60          <1>          PUSHA
83 0000003B 8A16[1E00]  <1>          MOV DL, [%1]
84 0000003F B402        <1>          MOV AH, 02H
85 00000041 CD21        <1>          INT 21H
86 00000043 61          <1>          POPA
87 00000044 C3          RET

```

【运行】

```

howasm\ch08> nasmw disp.asm -o disp.com -l disp.lst <Enter>
howasm\ch08> disp <Enter>
keyin char: C <Enter>
disp char= C

```

8-2-4 显示字节宏

宏 `showbyte` 将指定内存字节的内容以十六进制数字的形式显示在屏幕上。它的作法是将参数值除以 16 后，其商数存于 AL 为高位数，其余数存于 AH 为低位数，将它们转换为十六进制数字即可。如表 8-1 所示。

表 8-1 十进制数转换为十六进制字符

数值	十六进制字符
0	'0'
1	'1'
2	'2'
3	'3'
4	'4'
5	'5'
6	'6'
7	'7'
8	'8'
9	'9'
10	'A'
11	'B'
12	'C'
13	'D'
14	'E'
15	'F'

数值 0 至 9 仍然转换为'0' 至'9'，但 10 要转换成'A'，11 转换成'B'，12 转换成'C'，13 转换成'D'，14 转换成'D'，15 转换成'F'。因为'0' 至'9' 的 ASCII 码其值为 48 至 57，只要将 0 至 9 的数值加上 48 就可转成'0' 至'9' 了。但'A' 至'F' 的 ASCII 码其值为 65 至 70。要将 10 转成'A' 光加上 48 是不够的（10+48=58），必须额外再加上 7 才得 65 值，也才是 ASCII 码的'A'。

宏 showbyte.mac 第 13 行将参数地址的字节内容传送至 AL。第 14 行将 AX 除以 16，其商数存于 AL，随即传送至 DL，其余数存于 AH，随即传送至 DH。第 16~23 行将高位数转换为十六进制字符并显示。第 24~31 行将低位数转换为十六进制字符并显示。

【宏 showbyte.mac】

```
1 ; ***** showbyte.mac *****
2 ;
3 ; showbyte bmem
4 ; 将 bmem 内存内容以十六进制数字显示在屏幕上
5 ; bmem : 字节内存地址
6 ;
7 %ifndef SHOWBYTE_MAC
8 %define SHOWBYTE_MAC
9 %MACRO showbyte 1                ;参数 bmem 地址
10     PUSHA                        ;存储原来寄存器数据
11     MOV BL, 16                   ;除数 BL=16
```

```

12      MOV AH, 0           ;AH=0
13      MOV AL, [%1]        ;AL=bmem 字节内含
14      DIV BL              ;AX/BL
15      MOV DH, AH          ;DH=低位数
16      MOV DL, AL          ;DL=高位数
17      CMP DL, 10          ;高位数<10?
18      JL %%less10H        ;是,跳至%%less10H
19      ADD DL, 7           ;10 至 15 值 DL=DL+7
20 %%less10H:
21      ADD DL, 30H          ;数值转换为数字
22      MOV AH, 02H         ;设定显示字符功能
23      INT 21H             ;显示字符
24      MOV DL, DH          ;DL=低位数
25      CMP DL, 10          ;低位数<10?
26      JL %%less10L        ;是,跳至%%less10L
27      ADD DL, 7           ;10 至 15 值 DL=DL+7
28 %%less10L:
29      ADD DL, 30H          ;数值转换为数字
30      MOV AH, 02H         ;设定显示字符功能
31      INT 21H             ;显示字符
32      POPA                ;恢复原来寄存器数据
33 %ENDMACRO
34 %endif

```

程序 show.asm 使用 showbyte 宏分别将字节、字以及双字的内容以十六进制字符显示在屏幕上。显示字 b 时先显示偏移地址为 b 的低字节,然后再显示偏移地址为 b+1 的高字节。显示双字 c 时也是一样,先显示偏移地址 c 的低字节,最后再显示偏移地址为 c+3 的高字节。这是在内存里的存放情形。若想先显示高字节再显示低字节当然也是可以的:

```
showbyte b+1
```

```
showbyte b
```

如此就可显示为 1234 而不是 3412 了。

【程序 show.asm】

```

1 ; ***** show.asm *****
2      ORG 0100H
3      JMP start
4 a      DB 12H
5 b      DW 1234H
6 c      DD 12345678H
7 space  DB ' '
8 ;
9 %include "..\mymacro\showbyte.mac"
10 %include "..\mymacro\dispchr.mac"
11 ;

```

```
12 start:      showbyte a      ;a 地址内含以十六进制数字显示
13            dispchr space    ;空一格
14            showbyte b       ;b 地址内含以十六进制数字显示
15            showbyte b+1     ;b+1 地址内含以十六进制数字显示
16            dispchr space    ;空一格
17            showbyte c       ;c 地址内含以十六进制数字显示
18            showbyte c+1     ;c+1 地址内含以十六进制数字显示
19            showbyte c+2     ;c+2 地址内含以十六进制数字显示
20            showbyte c+3     ;c+3 地址内含以十六进制数字显示
21            RET
```

【运行】

```
howasm\ch08> nasmw show.asm -o show.com <Enter>
howasm\ch08> show <Enter>
12 3412 78563412 [注]先显示低字节
```

8-2-5 读取字符串宏

宏 readestr 从键盘读取一个字符串，存放于指定的内存地址，调用它时必须提供两个参数，第一个参数标明读入字符个数的最大值，这个值与定义的缓冲区 buf 内存的大小有关，第二个参数标明读入字符实际个数，第三个参数标明读入字符缓冲区内存的地址。缓冲区 buf 内存的格式如下：

最大个数 实际个数 #1 字符 #2 字符.. #n 字符 <Enter>键
buf+0 buf+1 buf+2 buf+3 ... buf+(n+1) buf+(n+2)

例如在执行 readestr.asm 时调用 readestr 10, buf 表示最多输入 10 个字符，输入的字符串存入内存 buf 处，若输入 123<Enter> 那么 buf 缓冲区的内容如图 8-1 所示。

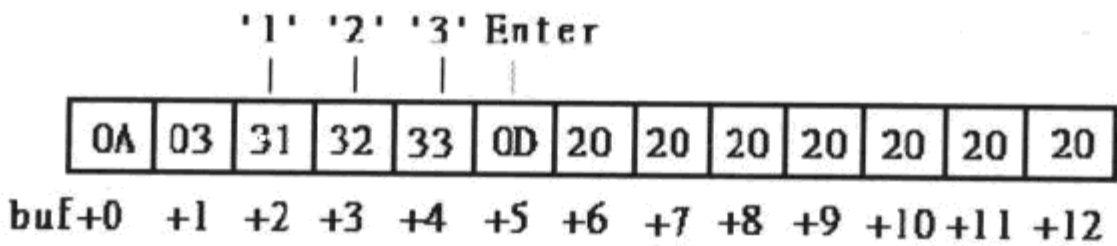


图 8-1 调用 readestr 输入 123<Enter>时缓冲区 buf 内容

【宏 readestr.mac】

```
;***** readestr.mac *****
;
; readestr limit buf
; 从键盘读取一个字符串存放于 bmem 内存地址
; limit : 读入字符最大个数
; buf : 读入字符串所存放内存地址
;
; limit    n        xx    xx    ... xx        0d
; 最大个数 实际个数 #1 字符 #2 字符... #n 字符 Enter 键
```

1998

250

```

22      showbyte buf+SI      ;显示下一个字节内容(十六进制)
23      dispchr space      ;空一格
24      INC SI              ;指向下一个字节
25      LOOP repeat        ;循环执行共 13 次
26      RET

```

【运行】

```

howasm\ch08> nasmw readstr.asm -o readstr.com <Enter>
howasm\ch08> readstr <Enter>
请输入一个字符串(以<Enter>键结束) : 123<Enter>
0A 03 31 32 33 0D 20 20 20 20 20 20 20

```

程序 readstr.asm 从键盘输入一个字符串至内存地址, 使用 21H 中断的 0aH 功能, 虽然可从键盘输入一个字符串, 但要提供一个缓冲区, 要提供输入最大字符数, 而且字符串以 <Enter> 键值 0dH 作为结束符号, 但 BIOS 的 21H 中断 09H 功能所显示的字符串却以 '\$' 作为结束符号, 很难配合, 因此另外写一个宏 readstr, 单纯地键入字符串, 并以 '\$' 作为字符串结束符号。其测试程序如 readstr.asm 所示。

宏 readstr.mac 第 14~16 行定义一个 buf 的缓冲区, 因为任何一个程序都可能调用此宏, 因此必须定义为区域性变量, 所以前面加上两个百分比符号, 成为 %%buf, 同理定义 %%n 及 %%s。第 19~21 行调用 21H 中断的 0aH 功能从键盘输入一字符串至 %%buf。第 25~27 行测试是否第一个字符为 <Enter> 键, 即 0dH, 是则直接返回一个 '\$' 符号表示空字符串, 否则将 %%s 纯字符串的部分拷贝至参数地址。第 35 行放弃 0dH 符号而改为 '\$' 字符串结束符号。

【宏 readstr.mac】

```

1 ;***** readstr.mac *****
2 ;
3 ; readstr buf
4 ; 从键盘读取一个字符串存放于 buf 内存地址
5 ; buf : 读入字符串所存放内存地址
6 ;
7 ; xx      xx      ... xx      '$'
8 ; #1 字符 #2 符 ... #n 字符 字符串结束符号
9 ;
10 %ifndef READSTR_MAC
11 %define READSTR_MAC
12 %MACRO readstr 1
13      JMP %%begin
14 %%buf DB 80
15 %%n   DB 0
16 %%s   TIMES 80 DB ' '
17 %%begin:
18      PUSHAX      ;保存原来寄存器值
19      MOV DX, %%buf ;将缓冲区地址存入 DX
20      MOV AH, 0aH  ;设定从键盘读入字符串

```

新
学
如
船



```
21          INT 21H                ;从键盘读入字符串至 buf
22          MOVZX CX, BYTE [%n]    ;CX=读入字符数
23          MOV SI, %%s            ;SI=读入字符串地址
24          MOV DI, %1             ;DI=参数地址
25          CMP BYTE [%%s], 0DH    ;是否为〈Enter〉键
26          JNE %%loop2           ;否
27          JMP %%endjob           ;是
28 %%loop2:
29          MOV AL, [SI]
30          MOV [DI], AL           ;DI 内容=SI 内容
31          INC SI                 ;SI=SI+1
32          INC DI                 ;DI=DI+1
33          LOOP %%loop2          ;继续
34 %%endjob:
35          MOV BYTE [DI], '$'     ;字符串结束符号
36          POPA                   ;恢复原来寄存器值
37 %ENDMACRO
38 %endif
```

程序 readstr.asm 用来测试 readstr 宏是否正确。第 13 行使用 dispstr 宏将 msg 显示在屏幕上。第 14 行使用 readstr 宏读入字符串至 s 变量。第 15 行使用 dispstr 宏将 crlf 显示在屏幕上,达到换行的目的。第 17 行使用 dispstr 宏将 s 显示在屏幕上。

【程序 readstr.asm】

```
1 ; ***** readstr.asm *****
2          ORG 0100H
3          JMP start
4 msg      DB '请输入一个字符串(以 Enter 键结束) : ', '$'
5 msg2     DB '刚键入的字符串= ', '$'
6 s        TIMES 80 DB ' '
7 crlf     DB 13,10,'$'
8 ;
9 %include "..\mymacro\readstr.mac"
10 %include "..\mymacro\dispstr.mac"
11 ;
12 start:
13          dispstr msg             ;显示 msg
14          readstr s               ;从键盘输入字符串
15          dispstr crlf           ;换行
16          dispstr msg2           ;显示 msg2
17          dispstr s              ;显示 s 字符串
18          RET
```

【运行】

howasm\ch08> nasmw readstr.asm -o readstr.com <Enter>

```
howasm\ch08> readstr <Enter>
```

```
请输入一个字符串(以 Enter 键结束) : 123abc <Enter>
```

```
刚键入的字符串= 123abc
```

8-2-6 字符串转换为数值

所输入的字符串若属于数字，总希望将它转换成二进制数值，方便做算术计算，因此接下来的工作就是如何将数字字符串转换成二进制数。我们先从最常用的字做起。

宏 `strtoi.mac` 将指定的数字字符串转换成带符号的二进制数，存于指定的字。第一个参数为指定的数字字符串地址，第二个参数为数字字符串结束符号常量，第三个参数为转换成带符号二进制数所存放的字地址。第 20~32 行检查数字字符串是否含有符号。第 33~44 行将数字字符串转换为数值，它的逻辑如图 8-2 所示。

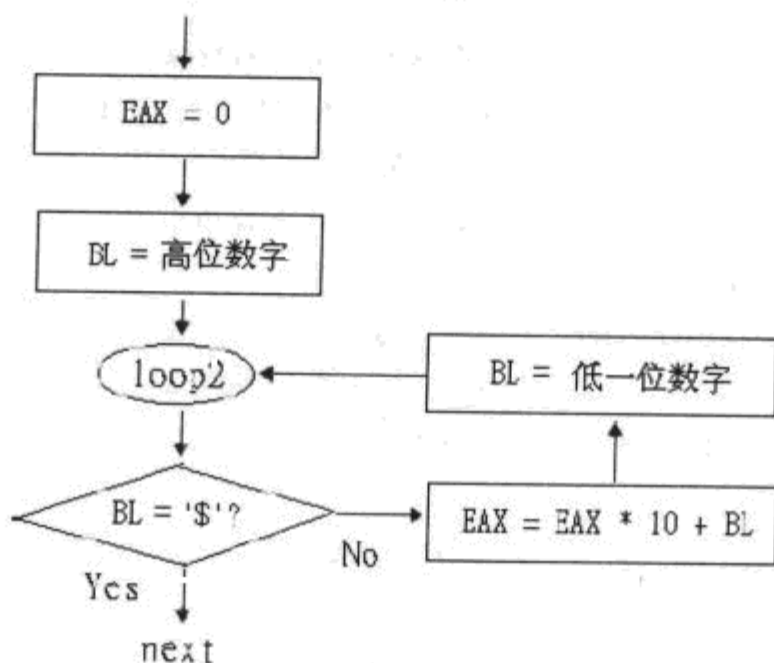


图 8-2 数字字符串转换为数值

例如数字字符串为 '123\$', 则根据图 8-2 的流程图逻辑，演算的结果如下：

$EAX = 0$

$BL = 1$

[注] 第一个数字

$EAX = 0 * 10 + BL = 1$

$BL = 2$

[注] 第二个数字

$EAX = 1 * 10 + 2 = 12$

$BL = 3$

[注] 第三个数字

$EAX = 12 * 10 + 3 = 123$

这一段逻辑就是第 33~44 行指令的依据。第 50 行将 `EAX` 中的 `AX` 部分传送至第三个参数的地址。

【宏 `strtoi.mac`】

```

1 ; ***** strtoi.mac *****
2 ;
3 ; strtoi str endchar wmem
4 ; 将数字字符串 str 转换成带符号二进制数，存于 wmem 字
5 ; 输入：

```

```

6 ; str : 字符串所存放内存地址
7 ;      xx      xx      ... xx      '$'
8 ;      #1 字符 #2 字符. ... #n 字符 字符串结束符号
9 ; endchar : 字符串结束符号常量, 0dH 或 00H 或 '$' 均可
10 ; 输出:
11 ; wmem : 转换成带符号二进制数值, 存于 wmem 字地址
12 ;
13 %ifndef STRTOI_MAC
14 %define STRTOI_MAC
15 %MACRO strtoid3 ;必须提供三个参数
16     JMP %%begin ;跳过定义
17 %%sign DB '-' ;存储符号位
18 %%begin:
19     PUSHAD ;存储原来寄存器数据
20     MOV SI, %1 ;SI=str 数字字符串地址
21     MOV AL, [SI] ;AL=第一位数字
22     CMP AL, '-' ;第一位数字是否为 '-'
23     JNE %%notm ;否
24     MOV [%%sign], AL ;将负号存入 %%sign
25     INC SI ;SI=下一个数字地址
26     JMP %%next ;继续
27 %%notm:
28     CMP AL, '+' ;第一位是否为 '+'
29     JNE %%next ;否
30     MOV [%%sign], AL ;将正号存入 %%sign
31     INC SI ;SI=下一个数字地址
32 %%next:
33     MOV EAX, 0 ;EAX=0
34 %%loop2:
35     CMP BYTE [SI], %2 ;是否为字符串结束符号
36     JE %%endjob ;是, 结束
37     MOV EBX, 10 ;EBX=10
38     MUL EBX ;EDX:EAX=EAX*10
39     MOV BL, BYTE [SI]
40     SUB BL, 30H ;下一位数字转为数值
41     MOVSX EBX, BL ;EBX=下一位数值
42     ADD EAX, EBX ;EAX=EAX+EBX
43     INC SI ;SI=下一位数字地址
44     JMP %%loop2 ;继续循环
45 %%endjob:
46     CMP BYTE [%%sign], '-' ;%%sign 是否为 '-'
47     JNE %%pos ;否, 跳至 %%pos
48     NEG EAX ;将 EAX 正数转为负数

```

```

49 %%pos:
50         MOV [%3], AX      ;返回带符号二进制数
51         POPA              ;恢复原来寄存器数据
52 %ENDMACRO
53 %endif

```

程序 `strtoi.asm` 用来测试 `strtoi` 宏是否正确。第 9 行将 `strtoi` 宏引入程序里。第 14 行将字符串 `str1` 其值 '123' 转换为 `num2`。第 15~17 行将 `num2` 以十六进制形式显示。第 18 行将字符串 `str2` 其值为 '-123' 转换为 `num2`。第 19~20 行将 `num2` 以十六进制形式显示。

【程序 strtoi.asm】

```

1 ; ***** strtoi.asm *****
2     ORG 0100H
3     JMP start
4 num2 DW 9876
5 str2 DB '-123', '$'
6 str1 DB '123', '$'
7 crlf DB 13, 10, '$'
8 ;
9 %include "..\mymacro\strtoi.mac"
10 %include "..\mymacro\dispstr.mac"
11 %include "..\mymacro\showbyte.mac"
12 ;
13 start:
14     strtoi str1, '$', num2      ;字符串 str1 转换成 num2 数值
15     showbyte num2+1            ;显示 num2 高位数(十六进制)
16     showbyte num2              ;显示 num2 高位数(十六进制)
17     dispstr crlf               ;换行
18     strtoi str2, '$', num2     ;字符串 str2 转换成 num2 数值
19     showbyte num2+1            ;显示 num2 高位数(十六进制)
20     showbyte num2              ;显示 num2 高位数(十六进制)
21     RET

```

【运行】

```

howasm\ch08> nasmw strtoi.asm -o strtoi.com <Enter>
howasm\ch08> strtoi <Enter>
007B
FF85

```

8-2-7 数值转换为字符串

宏 `itostr.mac` 将指定的带符号数转换成字符串，存于指定的地址。第一个参数为带符号二进制数所存放的地址，第二个参数为转换后字符串所存放的地址，第三个参数为字符串结束符号常量。

第 14~24 行检查数值是否为负数，负数则输出 '-', 正数或零值则输出 ''。第 25~33 行利

用除法，每次将数值除以 10 后将余数存入堆栈顶端，一直到商为零值才停止。它的逻辑如图 8-3 所示。

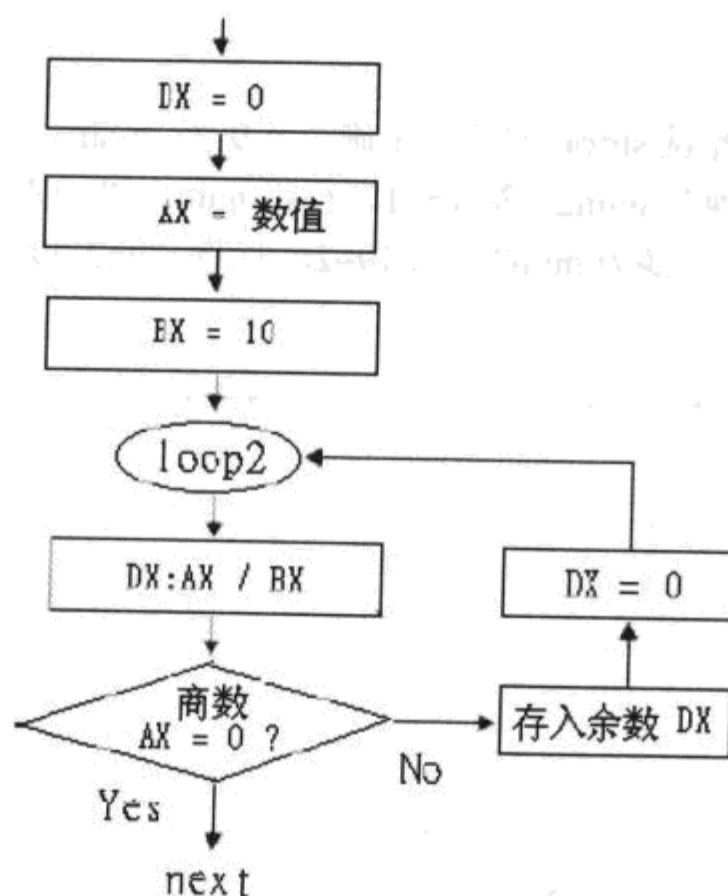


图 8-3 使用除法将数值转换成字符串

例如数值为 123，那么根据图 8-3 的流程图逻辑，演算的结果如下：

DX = 0
 AX = 123
 BX = 10
 商数 AX = 12
 余数 DX = 3 压入堆栈
 商数 AX = 1
 余数 DX = 2 压入堆栈
 商数 AX = 0
 余数 DX = 1 压入堆栈

因此最后在堆栈顶端的数据项为 1，下面是 2，在最底下的是 3，如图 8-4 所示。

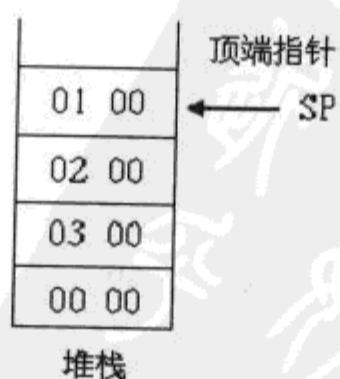


图 8-4 数值除以 10 后将余数存入堆栈顶端

第35~43行只是逐项从堆栈顶端弹出并转成字符而已。

【宏 itostr.mac】

```

1 ; ***** itostr.mac *****
2 ;
3 ; itostr wmem, dmem, endchar
4 ; 将 wmem 地址字带符号数值转换为 ASCII 数字存于 dmem 地址
5 ; wmem : 字地址
6 ; dmem : ASCII 数字地址
7 ; endchar : 字符串结束符号(常量,传值)
8 ;
9 %ifndef ITOSTR_MAC
10 %define ITOSTR_MAC
11 %MACRO itostr 3
12     PUSHA ;存储原来寄存器数据
13     MOV CX, 0 ;ASCII 数字位数计数
14     MOV AX, [%1] ;字地址内容
15     CMP AX, 0 ;AX>=0?
16     JGE %%pos ;是
17     NEG AX ;AX=-AX
18     MOV DL, '-' ;DL='-'
19     MOV [%2], DL ;ASCII 数字地址内容 '-'
20     JMP %%next
21 %%pos:
22     MOV DL, ' ' ;DL=' '
23     MOV [%2], DL ;ASCII 数字地址内容 ' '
24 %%next:
25     MOV DX, 0 ;DX=0
26     MOV BX, 10 ;BX=10
27 %%loop2:
28     DIV BX ;DX:AX/BX
29     PUSH DX ;压入余数
30     INC CX ;ASCII 数字位数增一
31     MOV DX, 0 ;DX=0
32     CMP AX, 0 ;商=0?
33     JNZ %%loop2 ;否
34 ;
35     MOV SI, %2 ;SI=ASCII 数字标号地址
36     INC SI ;下一个 ASCII 数字地址
37 %%loop3:
38     POP AX ;弹出余数至 AX
39     ADD AL, 30H ;数值转换为 ASCII 数字
40     MOV [SI], AL ;存入 SI 地址

```



```
41          INC SI          ;SI=下一个 ASCII 数字地址
42          LOOP %%loop3    ;继续
43          MOV BYTE [SI], %3 ;字符串结束符号
44          POPA             ;恢复原来寄存器数据
45 %ENDMACRO
46 %endif
```

程序 itostr.asm 用来测试 itostr 宏是否正确。第 11 行将 itostr 宏引入程序里。第 14 行将字符串 a 值 123 转换为 stra 字符串。第 15~16 行将 stra 字符串显示在屏幕上。第 18 行将数值 b 值-123 转换为 strb 字符串。第 18 行显示 strb 字符串。

【程序 itostr.asm】

```
1 ; ***** itostr.asm *****
2      ORG 0100H
3      JMP start
4 a     DW 123
5 b     DW -123
6 stra  DB ' '
7 strb  DB ' '
8 crlf  DB 13, 10, '$'
9 ;
10 %include "..\mymacro\dispstr.mac"
11 %include "..\mymacro\itostr.mac"
12 ;
13 start:
14      itostr a, stra, '$'    ;将 a 数值转换为 stra 字符串
15      dispstr stra          ;显示 stra 字符串
16      dispstr crlf          ;换行
17      itostr b, strb, '$'   ;将 b 数值转换为 strb 字符串
18      dispstr strb          ;显示 strb 字符串
19      RET
```

【运行】

```
howasm\ch08> nasmw itostr.asm -o itostr.com <Enter>
howasm\ch08> itostr <Enter>
123
-123
```

8-2-8 数值输出至屏幕

宏 dispi 将指定地址的值以 ASCII 码形式显示在屏幕上。将参数地址的值存入寄存器 AX，每次除以 10 后取其商数，压入堆栈顶端，循环执行除法直到商数等于零为止。宏 dispi 的逻辑与 itostr.mac 宏的逻辑一样，所不同的是宏 dispi 将转换的结果直接输出至屏幕而已。

【宏 dispi.mac】

```

1 ; ***** dispi.mac *****
2 ;
3 ; dispi wmem
4 ; 将 wmem 字值以 ASCII 数字显示于屏幕
5 ; wmem : 字地址
6 ;
7 %ifndef DISPI_MAC
8 %define DISPI_MAC
9 %MACRO dispi 1                ;必须提供一个参数
10     PUSH A                    ;存储原来寄存器数据
11     MOV AX, [%1]              ;参数低字值
12     CMP AX, 0                 ;与 0 比较
13     JGE %%pos                 ;正数
14     PUSH AX
15     MOV DL, '-'               ;显示 '-'
16     MOV AH, 02H
17     INT 21H
18     POP AX
19     NEG AX                    ;负数变正数
20 %%pos:
21     MOV DX, 0                 ;参数高字值
22     MOV BX, 10                ;除数为 10 (十进制)
23     MOV CX, 0                 ;整数字数计数
24 %%loop2:
25     DIV BX                    ;DX:AX/10
26     PUSH DX                   ;压入余数
27     INC CX                    ;整数字数增一
28     MOV DX, 0                 ;参数高字值
29     CMP AX, 0                 ;商数是否为零
30     JNZ %%loop2              ;否,跳至%%loop2 继续
31 %%loop4:
32     POP AX                    ;弹出余数至 AX
33     ADD AX, 0030H             ;余数调整为 ASCII 码
34     MOV DL, AL                ;DL=余数 ASCII 码
35     MOV AH, 02H               ;设定显示字符功能
36     INT 21H                  ;显示余数
37     LOOP %%loop4             ;继续处理下个余数
38     POP A                     ;恢复原来寄存器数据
39 %ENDMACRO
40 %endif

```



【程序 dispi.asm】

```
1 ; ***** dispi.asm *****
2     ORG 0100H
3     JMP start
4 a     DW 123
5 b     DW -123
6 ;
7 %include "..\mymacro\dispi.mac"
8 %include "..\mymacro\newline.mac"
9 ;
10 start:
11     dispi a           ;显示 a 数值
12     newline          ;换行
13     dispi b           ;显示 b 数值
14     RET
```

【运行】

```
howasm\ch08> nasmw dispi.asm -o dispi.com <Enter>
howasm\ch08> dispi <Enter>
123
-123
```

8-3 条件汇编

条件汇编的格式常用的有下列两种:

格式一:

%ifdef 符号

只有符号有定义时才被汇编的程序代码

%endif

格式二:

%if expr

只有表达式 expr 的值为非零时才被汇编的程序代码

%endif

例如:

%ifndef READCHR

%define READCHR

%MACRO readchr 1

PUSHA

MOV AH, 01H

INT 21H

MOV [%1], AL

POPA

%ENDMACRO

%endif

;名称 readchr, 一个参数

;保存原来寄存器值

;从键盘读取一个字符

;至寄存器 AL

;将 AL 存入参数所指地址

;恢复原来寄存器值

;宏结束

上例表示若符号 READCHR 还没有定义,就定义 READCHR,然后定义一个宏 readchr,若 READCHR 已经有定义则不作任何事情。这样的安排可保证程序里最多保存一个 readchr 宏。

```
%if 0
    MOV AX, 0
%endif
```

上例中 MOV 指令不会被汇编。

【程序 cond2.asm】

```
1 ; ***** cond2.asm *****
2      ORG 0100H
3      JMP start
4 msg   DB 'keyin a char: ', '$'
5 newline DB 13, 10, '$'
6 msg2   DB 'display= '
7 ch2    DB ' ', '$'
8 ;
9 %include "..\mymacro\readchr.mac"
10 %include "..\mymacro\dispchr.mac"
11 %include "..\mymacro\dispstr.mac"
12 ;
13 start:      dispstr msg           ;显示 msg 字符串
14             readchr ch2           ;读入一字符至 ch2
15             dispstr newline       ;换行
16             %if 0
17                 dispchr ch2        ;这个宏不被汇编(测试用)
18             %endif
19             dispstr msg2           ;显示 msg2 字符串
20             RET
```

【运行】

```
howasm\ch08> nasmw cond2.asm -o cond2.com <Enter>
```

```
howasm\ch08> cond2 <Enter>
```

```
keyin a char: A
```

```
display= A
```

8-4 预处理循环

NASM 的 TIMES 指令虽然功能强大,但却不能多次执行多行宏,因此又提供一个 %rep 预处理循环指令。它以 %rep 开始,以 %endrep 结束,它的语法如下:

```
%rep 循环次数
    循环指令
%endrep
```



例如:

```
%assign i 0      ;指定变量 i 初值为 0
%rep 3           ;循环开始(循环三次)
    DW i         ;定义一个字初值为变量 i 的值
    %assign i i+1 ;指定变量 i 的值为 (i+1)
%endrep         ;循环结束
```

上例中首先指定变量 *i* 的初值为 0, 接着循环三次, 每一次定义一个初始值为变量 *i* 的值后, 将指定变量 *i* 的值为 (*i*+1), 然后又循环, 因此 DW *i* 共执行三次, 第一次时 *i* 值为 0, 第二次时 *i* 值为 1, 第三次时 *i* 值为 2, 结果相当于下列的三个指令:

```
DW 0 ;第一次循环
DW 1 ;第二次循环
DW 2 ;第三次循环
```

8-5 源程序文件的包含内容

包含文件指引 (include directive) 提供一个方法将其他的源程序文件引入至程序中, 标明包含文件指引之处。其格式如下:

```
%include 包含文件名
```

例如想将 readchr.mac 引入至程序中, 其指令如下:

```
%include "c:\howasm\mymacro\readchr.mac"
;其他程序指令
```

8-6 相关宏汇总

本章所提到的宏, 及其相关宏的用法依序说明如下:

btostr bmem,dmem,endchar

将 bmem 地址中带符号数值转换为 ASCII 数字后存于 dmem 地址

bmem: 字节地址

dmem: ASCII 数字地址

endchar: 字符串结束符号 (常量, 传值)

dispb bmem

将 bmem 字节值以 ASCII 数字显示在屏幕上

bmem: 字节地址

dispchr bmem

将 bmem 内存以字符形式显示在屏幕上

bmem: 符所存放内存地址

dispi wmem

将 wmem 值以 ASCII 数字形式显示在屏幕上

wmem: 字地址

displ dwmem

将 dwmem 值以 ASCII 数字形式显示在屏幕上

dwmem：双字地址

dispstr smem

将 smem 内存处字符串显示在屏幕上

smem：内存地址

dispub bmem

将 bmem 值以 ASCII 数字显示在屏幕上

bmem：字节地址

dispui wmem

将 wmem 字值以 ASCII 数字显示在屏幕上

wmem：字地址

dispul dwmem

将 dwmem 双字值以 ASCII 数字显示在屏幕上

dwmem：双字地址

itostr wmem, dmem, endchar

将 wmem 地址字带符号数值转换为 ASCII 数字后存于 dmem 地址

wmem：字地址

dmem：ASCII 数字地址

endchar：字符串结束符号（常量，传值）

ltostr dwmem, dmem, endchar

将 dwmem 地址的带符号数值转换为 ASCII 数字后存于 dmem 地址

dwmem：双字地址

dmem：ASCII 数字地址

endchar：字符串结束符号（常量，传值）

newline

换行

readchr bmem

从键盘读取一个字符存放于 bmem 内存地址

bmem：读入字符所存放的内存地址

readestr limit buf

从键盘读取一个字符串并存放于 bmem 内存地址

limit：读入字符个数不能超过此值

buf：读入字符串所存放内存地址

limit n	xx	xx	...	xx	0d
最大个数	实际个数	#1 字符	#2 字符	... #n 字符	<Enter>键
buf+0	+1	+2	... +(n+1)	+3	+(n+2)

readstr buf

从键盘读取一个字符串并存放于 buf 内存地址



buf: 读入字符串所存放的内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

showbyte bmem

将 bmem 内存内容以十六进制数字显示在屏幕上

bmem: 字节内存地址

strtob str endchar bmem

将数字字符串 str 转换成无符号二进制数并存于 bmem

输入:

str: 字符串所存放的内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

endchar: 字符串结束符号常量, 0dH 或 00H 或 '\$' 均可

输出:

bmem: 转换成带符号二进制数后存于 bmem

strtoi str endchar wmem

将数字字符串 str 转换成带符号二进制数后存于 wmem 字

输入:

str: 字符串所存放内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

endchar: 字符串结束符号常量, 0dH 或 00H 或 '\$' 均可

输出:

wmem: 转换成带符号二进制数后存于 wmem 字地址

strtol str endchar dwmem

将数字字符串 str 转换成带符号二进制数并存于 dwmem 双字

输入:

str: 字符串所存放内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

endchar: 字符串结束符号常量, 0dH 或 00H 或 '\$' 均可

输出:

dwmem: 转换成带符号二进制数并存于 dwmem 双字地址

strtoub str endchar bmem

将数字字符串 str 转换成无符号二进制数并存于 bmem 字节

str: 字符串所存放内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

endchar: 字符串结束符号, 0dH 或 00H 或 '\$' 均可

bmem: 转换成无符号二进制数并存于 bmem 字节地址

strtoui str endchar wmem

将数字字符串 str 转换成无符号二进制数并存于 wmem

str: 字符串所存放的内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

endchar: 字符串结束符号, 0dH 或 00H 或 '\$' 均可

wmem: 转换成无符号二进制数并存于 wmem

strtoul str endchar dwmem

将数字字符串 str 转换成无符号二进制数并存于 dwmem

str: 字符串所存放的内存地址

XX XX ... XX '\$'

#1 字符 #2 字符 ... #n 字符 字符串结束符号

endchar: 字符串结束符号, 0dH 或 00H 或 '\$' 均可

dwmem: 转换成无符号二进制数并存于 dwmem

ubtostr bmem, dmem, endchar

将 bmem 中的数值转换为 ASCII 数字并存于 dmem

bmem : 字节地址

dmem : ASCII 数字地址

endchar: 字符串结束符号 (常量, 传值)

uitostr wmem, dmem, endchar

将 wmem 地址的数值转换为 ASCII 数字并存于 dmem 地址

wmem : 字地址

dmem : ASCII 数字地址

endcha : 字符串结束符号 (常量, 传值)

ultostr dwmem, dmem, endchar

将 dwmem 地址双字数值转换为 ASCII 数字后存于 dmem 地址

dwmem: 双字地址

dmem : ASCII 数字地址

endchar: 字符串结束符号 (常量, 传值)

1. 设计一个单行宏:

```
%define sqrsum(a,b) ((a)*(a)+(b)*(b))
```

设计一个程序 ex0801.asm 以 sqrsum (3,-4) 进行测试。

2. 使用下列宏定义 100 个数值, 其初始值分别为 1 至 100, 设计一个程序 ex0802.asm 将这 100 个值累积后输出。

```
%assign i 1
```

```
%rep 100
```

```
DW i
```

```
%assign i i+1
```

```
%endrep
```

3. 设计一个宏输入一个字变量地址参数, 判断该数值是奇数还是偶数, 若是奇数则输出变量值为 1, 若是偶数则输出变量值为 0。其格式如下:

```
%MACRO oddeven 2
```

```
...
```

```
%ENDMACRO
```

设计一个程序 ex0803.asm 测试 oddeven 宏是否正确。

4. 设计一个宏将数值以指定进制输出。第一个参数为输入数值地址。第二个参数为输入地址。

```
%MACRO dispbase 2
```

```
...
```

```
%ENDMACRO
```

设计一个程序 ex0804.asm 测试 dispbase 宏是否正确。

9

过程

过程（procedure）可以说是一小段的独立程序，这里所说的独立是指逻辑上的独立，因为是一小段程序所以也称为函数（function）、例程（routine）、子例程（subroutine）或子程序（subprogram）等，其实都表示同一个意思。本书采用大家习惯的用语——过程。





9-1 过程的定义

在 NASM 汇编语言定义一个过程很简单，它的语法如下：

过程名称：

过程程序块

RET

过程名称事实上就是标号名称，过程程序块是一个指令区块。RET 指令为返回调用过程的指令。要调用一个过程只要在 CALL 保留字后面写上过程名称就可以了。

程序 hi.asm 第 3 行及第 4 行均调用 hi 过程。第 6~15 行为 hi 过程，过程名称为 hi，过程程序块为第 7~14 行的指令，它在屏幕上显示 hi 两个英文字母，它在逻辑上是独立的，您每次调用它都会显示 hi 两个英文字母。

【程序 hi.asm】

```
1 ; ***** hi.asm *****
2     ORG 0100H
3     CALL hi      ;调用 hi 过程
4     CALL hi      ;调用 hi 过程
5     RET
6 hi:                ;hi 过程
7     PUSHA        ;保存所有寄存器值
8     MOV DL, 'H'   ;DL='H'
9     MOV AH, 02H   ;设定显示字符功能
10    INT 21H       ;显示 DL 内容字符
11    MOV DL, 'i'   ;DL='i'
12    MOV AH, 02H   ;设定显示字符功能
13    INT 21H       ;显示 DL 内容字符
14    POPA          ;恢复所有寄存器值
15    RET
```

【运行】

```
howasm\ch09> nasmw hi.asm -o hi.com -l hi.lst <Enter>
```

```
howasm\ch09> hi <Enter>
```

HiHi

汇编时使用-l选项可以产生一个列表文件，文件 hi.lst 是 hi.asm 汇编过后所产生的列表 (list) 文件，包含地址及指令的机器码 (machine code)。当程序 hi.asm 中第 3 行调用 hi 过程时，执行下面两个步骤：

1. 将下一个地址 0003 (程序第 4 行) 压入堆栈顶端。SP 值先减去 2 后再将 0003 存入 SP 所指数据项。
2. 将 hi 过程进入点的地址 (第 6 行的地址 0007) 存入 IP。

接着就依序执行第 8~14 行指令，在屏幕上显示 Hi 后，接着就执行第 15 行的 RET 指令，执行下面两个步骤：

1. 将堆栈顶端 SP 所指数据项 (0003) 存入 IP。

2. SP 值加 2。

这时 IP 的值为 0003，接着就执行第 4 行指令第二次调用 hi 过程的指令。
第一次及第二次调用 hi 过程及堆栈的情形如图 9-1 及图 9-2 所示。

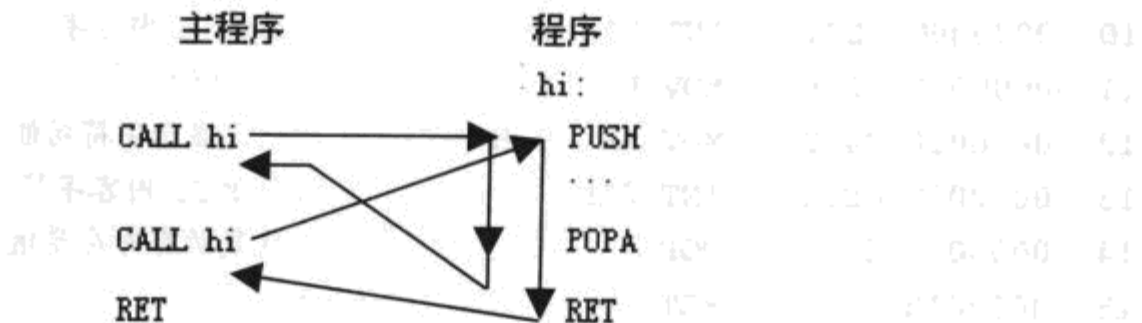


图 9-1 第一次及第二次调用 hi 过程的情形

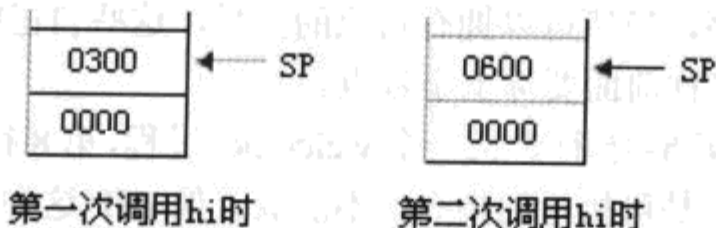


图 9-2 第一次及第二次调用 hi 过程时堆栈情形

当程序第 4 行调用 hi 过程时，执行下面两个步骤：

1. 将下一个地址 0006（程序第 5 行）压入堆栈顶端。SP 值先减去 2 后再将 0006 存入 SP 所指数据项。
2. 将 hi 过程进入点的地址（第 6 行的地址 0007）存入 IP。

接着就依序执行第 8~14 行指令，在屏幕上显示 hi 后，接着就执行第 15 行的 RET 指令，执行下面两个步骤：

1. 将堆栈顶端 SP 所指数据项（0006）存入 IP。
2. SP 值加 2。

这时 IP 的值为 0006，接着就执行第 5 行 RET 指令。

文件 hi.lst 为 hi.asm 汇编时所产生的列表文件，可以看到过程 hi 所产生的机器码只有一份，即第 7~15 行的“60B248B402CD21B269B402CD2161C3”机器码，每次调用它只是将这些机器码重新执行一次而已，不像宏，每次调用均产生一份机器码，因此使用过程调用的程序其机器码较短，使用宏调用的程序其机器码较长，但因过程调用的机制有一些额外的负担，所以优劣较难判断，不过现在的计算机内存都很大，使用宏的参数处理较简洁，所以作者比较喜欢使用宏。

【文件 hi.lst】

```
1 ; ***** hi.asm *****
2          ORG 0100H
3 00000000 E80400 CALL hi      ;调用 hi 过程
4 00000003 E80100 CALL hi      ;调用 hi 过程
5 00000006 C3      RET
```



```
6          hi:                                ;hi 过程
7 00000007 60          PUSHA                  ;保存所有寄存器值
8 00000008 B248          MOV DL, 'H'          ;DL='H'
9 0000000A B402          MOV AH, 02H          ;设定显示字符功能
10 0000000C CD21         INT 21H              ;显示 DL 内容字符
11 0000000E B269          MOV DL, 'i'          ;DL='i'
12 00000010 B402          MOV AH, 02H          ;设定显示字符功能
13 00000012 CD21         INT 21H              ;显示 DL 内容字符
14 00000014 61          POPA                  ;恢复所有寄存器值
15 00000015 C3          RET
```

9-2 过程里的局部变量

过程既然是一小段程序，当然可以拥有自己的变量，这些自己的变量称为过程里的局部变量 (local variable)，其名称前面要标上小数点。

程序 `welcome.asm` 中第 6~15 行定义一个 `welcome` 过程，第 8 行声明一个局部变量 `msg`，过程入口的第一个指令必须是可执行的指令，不能放置像 `DB` 这种伪指令，因此过程入口的第一个指令使用 `JMP` 跃过声明部分，跳到 `.begin` 标号处，在过程中的标号必须以小数点开头，表示属于区域性质。这种局部变量可以和过程名称结合而成为绝对标号，例如 `welcome.msg` 这个标号在整个程序中必须是独一无二的，否则名称重复就会产生汇编错误。

【程序 `welcome.asm`】

```
1 ; ***** welcome.asm *****
2     ORG 0100H
3     CALL welcome ;调用 welcome 过程
4     RET
5 ;-----
6 welcome: ;welcome 过程
7     JMP .begin
8 .msg  DB 'Welcome!!!', 13, 10, '$'
9 .begin:
10     PUSHA ;保存所有寄存器值
11     MOV DX, .msg ;DX=.msg 地址
12     MOV AH, 09H ;设定显示字符串功能
13     INT 21H ;显示 .msg 字符串
14     POPA ;恢复所有寄存器值
15     RET
```

【运行】

```
howasm\ch09> nasmw welcome.asm -o welcome.com <Enter>
howasm\ch09> welcome <Enter>
Welcome!!
```

程序 `welcome2.asm` 第 4 行在主程序中直接使用 `welcome` 过程里的变量 `msg`，这样做虽

然是允许的，但是作法不好。因为我们设计过程、使用过程，主要的目的是将程序模块化，一个模块设计成一个过程，一个模块最好只有一个入口和一个出口，您从主程序中直接使用过程里的变量，变成有两个以上的入口，对于将来的程序维护会增加困难的，因此这种作法并不可取。

【程序 welcome2.asm】

```

1 ; ***** welcome2.asm *****
2     ORG 0100H
3     CALL welcome           ;调用 welcome 过程
4     MOV DX, welcome.msg   ;局部变量.msg 地址
5     MOV AH, 09H           ;设定显示字符串功能
6     INT 21H               ;显示.msg 字符串
7     RET
8 ;-----
9 welcome:                   ;welcome 过程
10    JMP .begin
11    .msg DB 'Welcome!!!', 13, 10, '$'
12    .begin:
13    PUSHA                  ;保存所有寄存器值
14    MOV DX, .msg           ;DX=.msg 地址
15    MOV AH, 09H           ;设定显示字符串功能
16    INT 21H               ;显示.msg 字符串
17    POPA                   ;恢复所有寄存器值
18    RET

```

【运行】

```
howasm\ch09> nasmw welcome2.asm -o welcome2.com <Enter>
```

```
howasm\ch09> welcome2 <Enter>
```

```
Welcome!!
```

```
Welcome!!
```

为了让过程可以重复使用，可将测试好的过程存于磁盘，程序要用时才进行加载。例如 welcome 过程可存于\howasm\ch09 目录中。

【程序 welcome.pro 文件】

```

; ***** welcome.pro *****
;
; CALL welcome
; 显示 Welcome!! 信息
;
;
; welcome: ;welcome 过程
; JMP .begin
; .msg DB 'Welcome!!!', 13, 10, '$'
; .begin:
;     PUSHA ;保存所有寄存器值

```

```

MOV DX, .msg      ;DX=.msg 地址
MOV AH, 09H       ;设定显示字符串功能
INT 21H           ;显示.msg 字符串
POPA              ;恢复所有寄存器值
RET

```

程序 `welcome3.asm` 第 5 行引入同一目录中的 `welcome.pro` 过程文件。第 8 行调用 `welcome` 过程。

【程序 `welcome3.asm`】

```

1 ; ***** welcome3.asm *****
2          ORG 0100H
3          JMP start
4 ;
5 %include "welcome.pro"      ;welcome 过程文件
6 ;
7 start:
8          CALL welcome      ;调用 welcome 过程
9 RET

```

【运行】

```

howasm\ch09> nasmw welcome3.asm -o welcome3.com <Enter>
howasm\ch09> welcome3 <Enter>
Welcome!!

```

9-3 传值调用

过程间的数据可以传来传去，这些数据称为参数（parameter）。我们习惯说一个过程将一个参数传递给另一个过程。输入参数（input parameter）传递至被调用的过程，而输出参数（output parameter）则当被调用的过程返回时传递给调用者的过程。如图 9-3 所示。

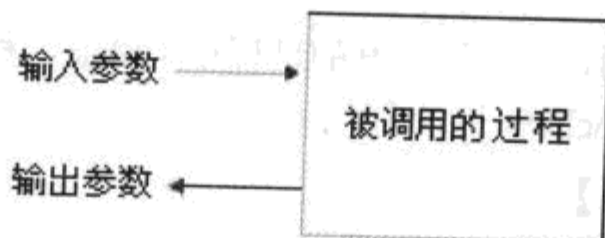


图 9-3 输入及输出参数

当所传递的参数是一个数值时，称为传值调用（call by value），当所传递的参数是一个地址时，称为传址调用（call by reference）。输入及输出参数是针对被调用的过程而言的，对调用者过程来讲不称为参数而称为自变量（argument），自变量与参数是一一对应的，自变量是真正的值，不管是数据值还是地址值，但参数是虚拟的，它的值并不固定，自变量是什么值它就接受什么值。传值调用时调用者过程的自变量并不会受到被调用过程操作的影响，但传址调用时调用者过程的自变量内容却会受到被调用过程操作的影响，因为自变量与参数均

指到同一个地址,若被调用过程更改该地址内容,地址值虽然没变,但内容却变了,返回至调用者过程的其地址的内容也就变了。

程序 byvalue.asm 第4行调用 printchr 过程,byvalue.asm 称为调用者过程,printchr 称为被调用过程,第3行 AL 寄存器存入字符常量'B',是传递给 printchr 过程的自变量,但对 printchr 过程而言 AL 寄存器称为输入参数。printchr 过程只是单纯的将 byvalue 传来的 AL 值显示在屏幕上而已。

第5行设定自变量为数值 20H,为 ASCII 码的空格符。第7行设定自变量数值为'y' 字符。

【程序 byvalue.asm】

```

1 ; ***** byvalue.asm *****
2     ORG 0100H
3     MOV AL, 'B'           ;打印一个'B'字符
4     CALL printchr
5     MOV AL, 20H           ;打印一个空格符
6     CALL printchr
7     MOV AL, 'y'           ;打印一个'y'字符
8     CALL printchr
9     RET
10 ;-----
11 printchr:
12     PUSHA                ;存储原来寄存器数据
13     MOV DL, AL            ;DL=ASCII 字符值
14     MOV AH, 02H           ;设定显示字符功能
15     INT 21H              ;显示字符
16     POPA                 ;恢复原来寄存器数据
17     RET

```

【运行】

```
howasm\ch09> nasmw byvalue.asm -o byvalue.com <Enter>
```

```
howasm\ch09> byvalue <Enter>
```

B y

将 printchr 过程存入\howasm\ch09 目录中,提供给其他程序引入使用。%ifndef 指令防止多次引入时,保证内存里只有一份 printchr 过程。

【宏 printchr.pro 文件】

```

; ***** printchr.pro *****
;
;     CALL printchr
; 将 AL 的字符显示在屏幕上
; 输入参数 AL : 字符值
;
%ifndef PRINTCHR_PRO

```



```
%define PRINTCHR_PRO
printchr:
    PUSHA                ;存储原来寄存器数据
    MOV DL, AL            ;DL=字符
    MOV AH, 02H          ;设定显示字符功能
    INT 21H              ;显示字符
    POPA                 ;恢复原来寄存器数据
    RET

%endif
```

程序 `asciitab.asm` 第 5 行引入 `printchr` 过程文件，因此就不必重复写一遍 `printchr` 过程的指令了。程序 `asciitab.asm` 打印 ASCII 码表中可打印的字符，最前面的 32 个字符为控制字符，不可打印，因此从第 33 个字符开始打印，第 33 个字符的 ASCII 码值为 32，所以第 9 行设定 CL 的值为 32。第 8 行 BL 的值为 16，表示每一行要输出 16 个 ASCII 字符，超过 16 则换行输出。正常时执行第 20~23 行的指令，显示一个空白后再显示一个 ASCII 码字符，但印满 16 字符时就执行第 15~18 行指令，输出 0dH 及 0aH，输出 0dH 字符会将打印字头返回第一个位置，输出 0aH 字符会将输出头往下移一行。第 25 行比较 CL 与 128，若相等就跳出循环。否则 CL 加一后再继续。

【程序 `asciitab.asm`】

```
1 ; ***** ascii.asm *****
2         ORG 0100H
3         JMP start
4 ;
5 %include "printchr.pro"
6 ;
7 start:
8         MOV BL, 16          ;BL=16, 每行 16 字符
9         MOV CL, 32          ;CL=32, 从可印字符开始
10 loop2:
11         MOVZX AX, CL        ;AX=CL
12         DIV BL              ;AX/BL
13         CMP AH, 0           ;商数 AH 为 0 吗?
14         JNZ next           ;否, 一行还未印满
15         MOV AL, 0dH         ;输出 CarriageReturn
16         CALL printchr       ;输出字头返回
17         MOV AL, 0aH         ;输出 LineFeed
18         CALL printchr       ;输出字头降下一行
19 next:
20         MOV AL, ' '         ;输出一个空格符
21         CALL printchr
22         MOV AL, CL          ;输出 AL 的 ASCII 码
23         CALL printchr
24         INC CL              ;CL=下一个 ASCII 码值
```

```
25      CMP CL, BYTE 128      ;CL 与 128 比较
26      JNZ loop2             ;不相等时继续
27      RET
```

【运行】

```
howasm\ch09> nasmw asciitab.asm -o asciitab.com <Enter>
howasm\ch09> asciitab <Enter>
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

程序 `gettime.asm` 第 3 行调用 `disptime` 过程，并没有提供任何的参数。程序第 25~26 行调用 21H 中断的 2cH 功能，从系统取得当前的时间。

```
MOV AH, 2cH      ;取得系统时间
INT 21H
返回来的时间存于寄存器，如下：
CH : 小时数(0-23)
CL : 分钟数(0-59)
DH : 秒数(0-59)
DL : 百分之秒数(0-99)
```

第 27 行将小时数 CH 存入 AL，然后将 AL 值当一个自变量，调用 `printub` 过程，AL 值对 `printub` 过程来讲为输入参数，随即将 AL 值以十进制 ASCII 码显示在屏幕上。其调用的过程如图 9-4 所示。

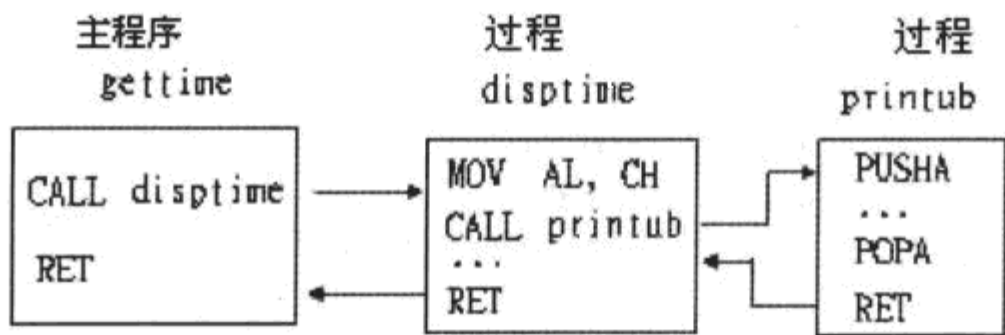
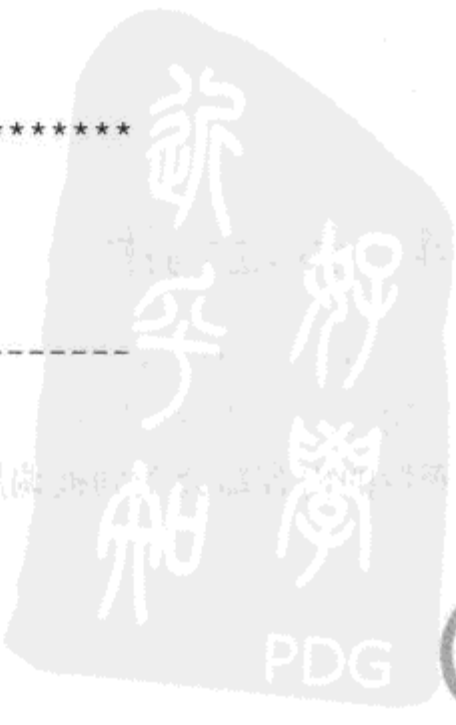


图 9-4 `gettime` 调用 `disptime` 调用 `printub` 过程

【程序 `gettime.asm`】

```
1 ; ***** gettime.asm *****
2      ORG 0100H
3      CALL disptime
4      RET
5 ;-----
6 printub:
7      PUSHA      ;存储原来寄存器数据
8      MOV AH, 0   ;AH=0
```



```

9          MOV BL, 10          ;BL=除数为10(十进制)
10         DIV BL              ;AX/10
11         OR AX, 3030H        ;转换为 ASCII 数字
12         PUSH AX
13         MOV DL, AL          ;商数,高位
14         MOV AH, 02H         ;显示
15         INT 21H
16         POP AX
17         MOV DL, AH          ;余数,低位
18         MOV AH, 02H         ;显示
19         INT 21H
20         POPA
21         RET
22 ;-----
23 disptime:
24         PUSHA               ;存储原来寄存器数据
25         MOV AH, 2cH         ;取得系统时间
26         INT 21H
27         MOV AL, CH          ;小时(00-23)
28         CALL printub        ;显示小时
29         MOV DL, ':'         ;显示冒号(:)
30         MOV AH, 02H
31         INT 21H
32         MOV AL, CL          ;分(00-59)
33         CALL printub        ;显示分
34         MOV DL, ':'         ;显示冒号(:)
35         MOV AH, 02H
36         INT 21H
37         MOV AL, DH          ;秒数(00-59)
38         CALL printub        ;显示秒数
39         POPA                ;恢复原来寄存器数据
40         RET

```

【运行】

```
howasm\ch09> nasmw gettime.asm -o gettime.com <Enter>
```

```
howasm\ch09> gettime <Enter>
```

```
08:10:36
```

9-4 传址调用

当所传递的参数是一个数值时，称为传值调用，当所传递的参数是一个地址时，就称为传址调用。传值调用时调用者的自变量并不会受到被调用过程的影响，但传址调用时调用者自变量的内容却会受到被调用过程的影响，因为自变量与参数均指到同一个地址，若被调用

过程更改该地址内容，地址值虽然没变，但内容却变了，返回至调用者的内容也就变了。

程序 byref.asm 第 4 行声明一个变量 msg，它的内容为'321\$' 字符串。第 10~11 行将 msg 字符串显示在屏幕上。第 12~13 行调用过程 printmsg，将寄存器 BX 值设为 msg 的地址，然后传递给 printmsg 过程当输入参数使用，在 printmsg 过程里头所看到的 BX 值事实上就是 msg 的地址，因此在第 20 行将'9' 传送至 BX 所指的地址，会将原来的'3' 覆盖，而变成新的'9'。执行第 22 行的 RET 指令返回 byref 主程序时，再将 msg 字符串显示出来，您将看到改变后的值'921\$'。其执行过程如图 9-5 所示。

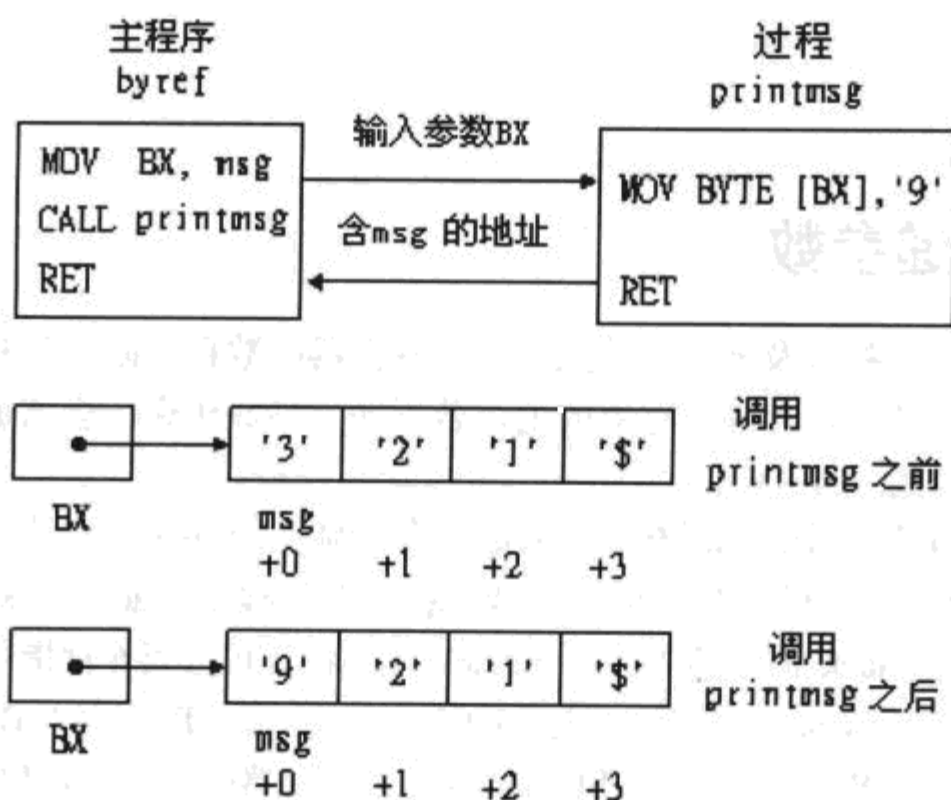


图 9-5 byref 程序的传址调用

【程序 byref.asm】

```

1 ; ***** byref.asm *****
2     ORG 0100H
3     JMP start
4     msg DB '3', '2', '1', '$'
5 ;
6 %include "..\mymacro\dispstr.mac"
7 %include "..\mymacro\newline.mac"
8 ;
9 start:
10    dispstr msg           ;调用 printmsg 过程之前
11    newline               ;换行
12    MOV BX, msg           ;msg 地址当自变量存入 BX
13    CALL printmsg         ;调用 printmsg 过程
14    dispstr msg           ;调用 printmsg 过程之后
15    newline               ;换行
16    RET

```

```

17 ;-----
18      printmsg:
19      PUSHA                ;存储原来寄存器数据
20      MOV BYTE [BX], '9'   ;将'9'传送至 BX 所指地址
21      POPA                 ;恢复原来寄存器数据
22      RET

```

【运行】

```

howasm\ch09> nasmw byref.asm -o byref.com <Enter>
howasm\ch09> byref <Enter>
321
921

```

9-5 堆栈传递参数

过程调用时若输入参数及输出参数太多，而寄存器的数目有限，可利用堆栈传递参数。但是堆栈有其固定的操作，存储寄存器的值，若再加上参数传递，会增加其复杂性，对于堆栈顶端指针的位置要特别小心。

程序 `bystack.asm` 是一个使用堆栈传递参数的简单例子，在主程序里第 3 行将自变量 `AX` 的值设为 64，相当于十六进制的 40H，然后压入堆栈，作为过程 `usestack` 的输入参数，如图 9-6 所示。当第 5 行调用 `usestack` 时先将主程序 `bystack` 的返回地址（第 6 行指令的地址）压入堆栈顶端，然后进入过程 `usestack`，执行第 11 行 `MOV` 指令后，将 `SP` 值存入 `BP`，然后执行 `PUSHA` 时存入 `AX`、`CX`、`DX`、`BX`、`SP`、`BP`、`SI`、`DI` 等八个寄存器的值，因此 `SP` 的值减少 16，如图 9-6 所示。第 13 行对于堆栈 `BP+2` 的数据项内容增加一，这是输入的参数，原来的值为 64，增加一后变为 65，因此返回主程序时，参数的值已经变成 65，以字符输出为 ASCII 码的 'A'。

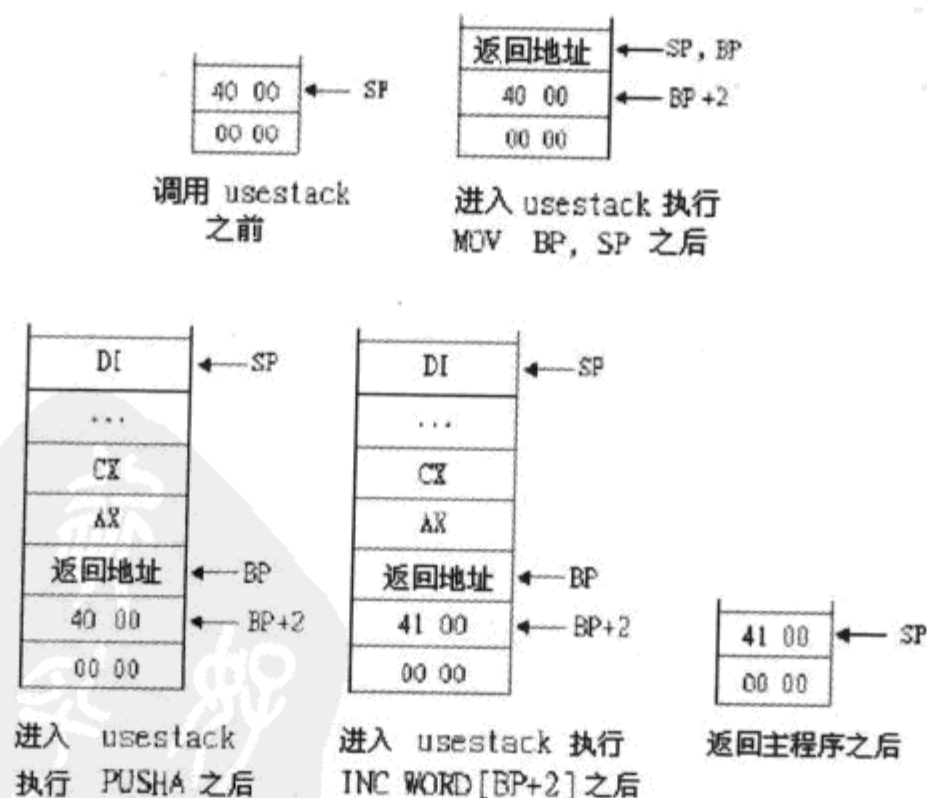


图 9-6 使用堆栈传递参数的情形

【程序 bystack.asm】

```

1 ; ***** bystack.asm *****
2      ORG 0100H
3      MOV AX, 64 ;AX=64
4      PUSH AX      ;压入堆栈顶当 usestack 输入参数
5      CALL usestack ;调用 usestack 过程
6      POP DX       ;弹出 usestack 顶端数据项至 DX
7      MOV AH, 02H  ;显示 DL 字符
8      INT 21H
9      RET
10 usestack:
11      MOV BP, SP ;BP=SP
12      PUSHA      ;存储原来寄存器数据
13      INC WORD [BP+2] ;AX=AX+1
14      POPA       ;恢复原来寄存器数据
15      RET

```

【运行】

```
howasm\ch09> nasmw bystack.asm -o bystack.com <Enter>
```

```
howasm\ch09> bystack <Enter>
```

```
A
```

9-6 内存传递参数

过程调用时若输入参数及输出参数太多，而寄存器的数目有限，除了利用堆栈传递参数外，也可以使用程序本身的内存。因为堆栈有其固定的操作，存储寄存器的值，若再加上参数传递，会增加其复杂性，对于堆栈顶端指针的位置容易弄错，导致不可预知的错误，不如使用程序本身的内存传递参数来得干净利落。

程序 bymem.asm 是一个使用程序本身内存传递参数的例子，在主程序里第 4 行声明一个五个字元素的数组 ary，其元素初值分别为 1、3、5、7、9。第 5 行声明一个字的指针变量 aryptr，它的内容为另一个变量的地址，故称为指针变量 (pointer variable)，本例题它的内容为 ary 数组的地址。第 6 行声明一个字变量 count，用于记录 ary 数组的元素个数，其初值为 5。第 7 行声明一个字变量 sum，用于存储 ary 数组各元素值的总和。count 也可以称为 aryptr+2，因为变量的名称也不过是地址的代名词而已。sum 也可以称为 aryptr+4。它们的关系如图 9-7 调用 usemem 过程之前的部分。第 12~13 行调用过程 usemem，第 12 行将 aryptr 的地址存入 BX 寄存器，因为指针变量会执行间接寻址，因此使用的寄存器要用索引寄存器，例如 BX、SI、DI 或 BP 等，本例使用 BX 索引寄存器当过程 usemem 的输入参数，第 13 行调用 usemem 过程。

在 usemem 过程中，第 18 行首先将 BX 参数的内容存入 BP，这时 BP 的内容事实上就是 ary 的地址，数组 ary 的第一个元素也可称为 BP，第二个元素也可称为 BP+2，第三个元素 BP+4，第四个元素 BP+6，第五个元素 BP+8。第 19 行将 BX+2 的内容存入 CX 中，CX

的内容事实上就是 count 的内容，表示数组 ary 元素的个数。第 20 行将 AX 清除为零值，准备累积数组 ary 每一个元素值的总和。第 21 行将 SI 索引寄存器的值置零，指到数组 ary 的第 0 个元素，其地址为 BP+SI，因为 SI 为零值，事实上就是 BP，也就是 ary 的地址。第 22~26 行构成一个循环，逐项 BP+SI 将元素值累积至 AX，每次 SI 均加 2 表示索引至下一个元素。第 27 行将累积的结果存入 BX+4 地址的字里，事实上它就是 sum 的地址。

回到主程序后，第 14 行将 sum 的内容显示出来，在调用过程 usemem 之前它的初始值为零值，但调用过程 usemem 之后它的值已经改变为 25 了，显示出来当然是 25 了。如图 9-7 所示。

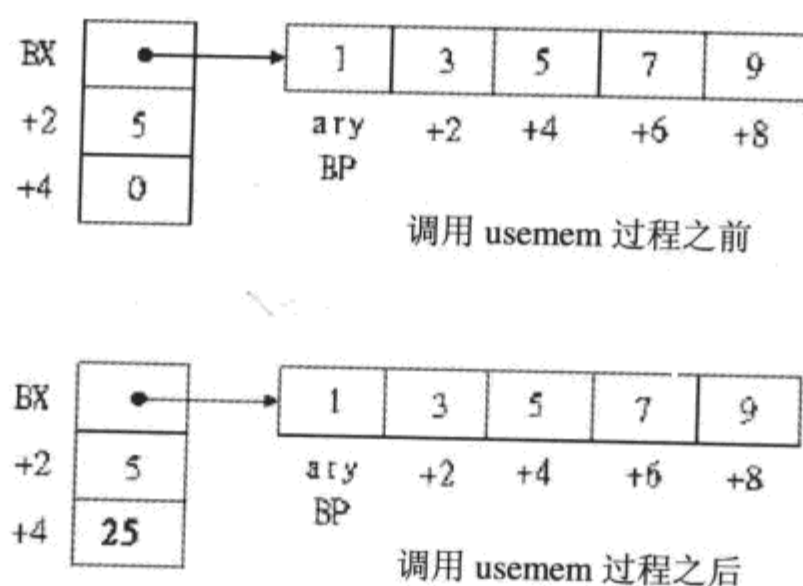


图 9-7 使用内存传递参数的情形

【程序 bymem.asm】

```

1 ; ***** bymem.asm *****
2         ORG 0100H
3         JMP start
4     ary     DW 1, 3, 5, 7, 9           ;ary 数组含五个元素
5     aryptr  DW ary                   ;ary 数组地址
6     count   DW 5                     ;ary 数组元素个数
7     sum     DW 0                     ;ary 数组元素总和
8 ;
9 %include "..\mymacro\dispui.mac"
10 ;
11 start:
12         MOV BX, aryptr               ;BX=参数起始地址
13         CALL usemem                  ;调用 usemem 过程
14         dispui sum                   ;显示数组 ary 各元素总和
15         RET
16 usemem:
17         PUSHA                        ;存储原来寄存器数据
18         MOV BP, [BX]                 ;将 ary 数组地址存入 BP
19         MOV CX, [BX+2]                ;CX=ary 数组元素个数
20         MOV AX, 0 ;AX=0
    
```

```

21      MOV SI, 0 ;SI=0
22  .loop2:
23      ADD AX, [BP+SI]      ;ary 数组第 SI 元素值累积至 AX
24      INC SI              ;SI=SI+1
25      INC SI              ;SI=SI+1, 索引至下一元素
26      LOOP .loop2         ;继续
27      MOV WORD [BX+4], AX  ;sum=AX
28      POPA                ;恢复原来寄存器数据
29      RET

```

【运行】

```
howasm\ch09> nasmw bymem.asm -o bymem.com <Enter>
```

```
howasm\ch09> bymem <Enter>
```

```
25
```

课后习题

1. 设计一个过程 `cube`, 输入参数 `AX` 为传值调用, 输出为 `AX` 的 3 次方, 其值仍然通过 `AX` 返回。设计一个程序 `ex0901.asm` 测试它。
2. 设计一个过程 `funcx`, 输入参数 `AX` 为传值调用, 输出为 `AX` 的 3 次方减去 2, 其值通过参数 `BP` 地址返回。设计一个程序 `ex0902.asm` 测试它。
3. 设计一个过程 `aryavg`, 输入参数 `BX` 为传址调用, 为字数组地址, 第二个参数 `CX` 为传值调用, 为数组元素个数, 第三个参数 `BP` 为传址调用, 为数组元素的平均值。设计一个程序 `ex0903.asm` 以下列数据测试它。

```
ary DW 123, 78, 256, 365
```

知识
能力
素质
PDG

10

字符串处理

字符串 (string) 是一个连续的字节或字, 它存储可以放入一个字节或字的任何数据。例如一个字节的字符串可以存放 ASCII 码的字符, 一个字的字符串可以存放有符号整数或无符号整数等。

80x86 处理器提供指令处理字符串, 并提供一些对字符串基本操作的重要指令。除此之外, 当然可以自己设计对字符串操作的宏或过程。





10-1 声明字符串

可以使用 DB 或 DW 伪指令声明字符串。以 DB 声明的通常为字符的字符串较多。以 DW 声明的通常在一个字里存放两个字符。

程序 string.asm 中声明两个字符串, string 字符串及 char 字符串, 字符串以 '\$' 符号结束, 为了配合中断 21H 的 09H 功能, 所显示的字符串均以 '\$' 符号结束。string 字符串存储在计算机的连续内存中, 如图 10-1 所示。

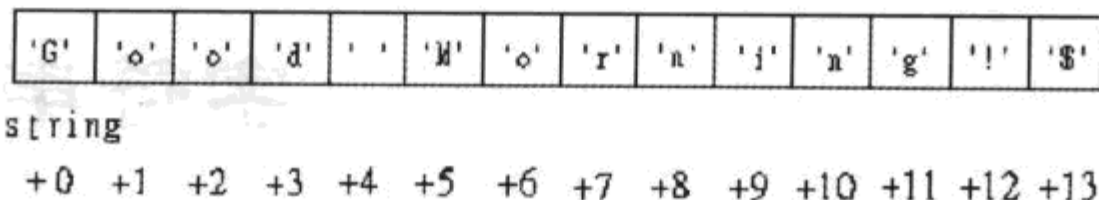


图 10-1 string 字符串存于计算机的连续内存

第 7~9 行将 string 字符串显示出来。第 11~13 行将 string+5 地址开始的字符串显示出来。第 15 行将 string+5 地址的内容存入 AL, 接着又从 AL 传送至 char 地址, 因为处理器不允许从甲地址传送至乙地址, 因此必须通过一般用途的寄存器才可以。第 17~19 行将 char 字符串显示出来。

【程序 string.asm】

```
1 ; ***** string.asm *****
2     ORG 0100H
3     JMP start
4 string    DB      'Good Morning!', '$'
5 char    DB      ' ', '$'
6 start:
7     MOV DX, string          ;DX=字符串 string 地址
8     MOV AH, 09H             ;设定显示字符串功能
9     INT 21H                 ;显示 string 字符串
10 ;
11     MOV DX, string+5        ;DX=字符串 string 地址
12     MOV AH, 09H             ;设定显示字符串功能
13     INT 21H                 ;显示 string 字符串
14 ;
15     MOV AL, BYTE [string+5] ;AL=string+5 地址字符
16     MOV BYTE [char], AL     ;char 地址内容为 AL
17     MOV DX, char            ;DX=字符串 char 地址
18     MOV AH, 09H             ;设定显示字符串功能
19     INT 21H                 ;显示 char 字符串
20     RET
```

【运行】

howasm\ch10> nasmw string.asm -o string.com <Enter>

```
howasm\ch10> string <Enter>
```

```
Good Morning!Morning!M
```

程序 wordstr.asm 中声明两个字符串, wordstr 字符串及 char 字符串, wordstr 字符串存储在计算机的连续字内存中, 如图 10-2 所示。因为是以字为单位, 所以两个字符存放在一个字里, '!' 字符只一个字符, 因此自动补充一个十六进制的 00H 字符。'\$' 也是这样。可以看出使用字声明的字符串处理上较为麻烦, 因此字符串通常均以 DB 声明。

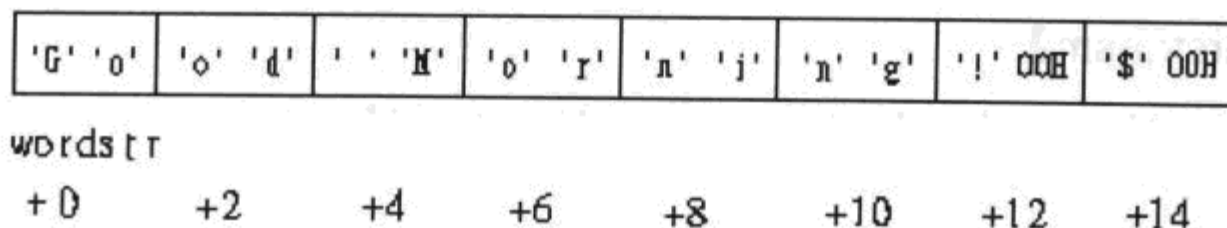


图 10-2 wordstr 字符串存于计算机的连续字内存

第 7~9 行将 wordstr 字符串显示出来。第 11~13 行将 wordstr+5 地址开始的字符串显示出来。第 15 行将 wordstr+5 地址的内容存入 AX, 接着又从 AX 传送至 char 地址。第 17~19 行将 char 字符串显示出来。执行时在 '!' 符号后面显示 'a', 事实上它是 00H 码所显示的, 属于乱码。因此有关字符的最好使用 DB 声明, 属于数值的使用 DW 声明较好。

【程序 wordstr.asm】

```
1 ; ***** wordstr.asm *****
2     ORG 0100H
3     JMP start
4 wordstr DW      'Good Morning!', '$'
5 char  DW      ' ', '$'
6 start:
7     MOV DX, wordstr      ;DX=字符串 wordstr 地址
8     MOV AH, 09H          ;设定显示字符串功能
9     INT 21H              ;显示 wordstr 字符串
10 ;
11     MOV DX, wordstr+5    ;DX=字符串 wordstr+5 地址
12     MOV AH, 09H          ;设定显示字符串功能
13     INT 21H              ;显示 wordstr+5 字符串
14 ;
15     MOV AX, WORD [wordstr+5] ;AX=wordstr+5 地址字符
16     MOV WORD [char], AX    ;char 地址内容为 AX
17     MOV DX, char          ;DX=字符串 char 地址
18     MOV AH, 09H          ;设定显示字符串功能
19     INT 21H              ;显示 char 字符串
20     RET
```

【运行】

```
howasm\ch10> nasmw wordstr.asm -o wordstr.com <Enter>
```

```
howasm\ch10> wordstr <Enter>
```

```
Good Morning!aMorning!aMo
```



若要声明一个很长的字符串，而要将每一个字符的值写出来，那是一件不容易的事。例如要声明一个 1024 个字符长度的字符串，每个字符均为 '*'。可使用 TIMES 伪指令做如下的声明：

```
starstr TIMES 1024 DB '*'
DB '$'
```

则这 1024 个星号的起始地址为 starstr，以 '\$' 结束。程序 starstr.asm 只显示 36 个星号。

【程序 starstr.asm】

```
1 ; ***** starstr.asm *****
2     ORG 0100H
3     JMP start
4 starstr TIMES 36 DB '*'
5     DB '$'
6 start:
7     MOV DX, starstr ;DX=字符串 starstr 地址
8     MOV AH, 09H ;设定显示字符串功能
9     INT 21H ;显示 starstr 字符串
10    RET
```

【运行】

```
howasm\ch10> nasmw starstr.asm -o starstr.com <Enter>
howasm\ch10> starstr <Enter>
```

10-2 字符串长度

在字符串处理中常常需要计算字符串的长度，因此设计一个 strlen 宏存放在 \howasm\mymacro 目录中，程序需要它时只要将它引入就可以了。

第 3~11 行说明这个宏的用法。第 13~14 及第 31 行的结构保证只载入一份程序代码。第 15~30 行定义 strlen 宏。%1 为第一个参数，相当于 smem。%2 为第二个参数，相当于 endchar。%3 为第三个参数，相当于 len。

【宏\howasm\mymacro\strlen.mac 文件】

```
1 ; ***** strlen.mac *****
2 ;
3 ; strlen smem, endchar, len
4 ;     计算 smem 内存处字符串长度，存入 len 字变量
5 ; smem : 内存地址
6 ; endchar : 字符串结束符号代码(字符常量)
7 ;         00H ASCIIZ 字符串与 ANSI C 字符串格式相同
8 ;         0dH 从键盘输入按<Enter> 键结束
9 ;         使用 21H 而 AH=01H 从键盘输入格式
10 ;        '$' 使用 21H 而 AH=09H 显示字符串格式
```

```

11 ; len : 字符串长度(字变量)
12 ;
13 %ifndef STRLEN_MAC
14 %define STRLEN_MAC
15 %MACRO strlen 3          ;必须提供三个参数
16     PUSH A                ;存储原来寄存器数据
17     MOV CX, 0             ;字符串长度初值
18     MOV SI, %1            ;字符串地址
19     MOV DH, %2            ;字符串结束符号常量
20 %%loop2:
21     MOV DL, [SI]          ;DL=下一个字符
22     CMP DL, DH             ;字符串结束符号?
23     JE %%end2             ;是
24     INC CX                ;长度增一
25     INC SI                ;下一个字符地址
26     JMP %%loop2           ;继续
27 %%end2:
28     MOV [%3], CX          ;长度存入第三个参数变量
29     POP A                 ;恢复原来寄存器数据
30 %ENDMACRO
31 %endif

```

程序 `strlen.asm` 用于测试字符串长度的宏 `strlen.mac` 是否正确。第 4 行声明一个字变量 `len`，用于存储字符串长度。第 8 行引入 `strlen.mac` 宏文件。第 15 行求字符串 `string` 的长度，该字符串以 '\$' 为结束符号，字符串长度存于 `len` 变量中。第 18 行显示字 `len` 的数值。

【程序 `strlen.asm`】

```

1 ; ***** strlen.asm *****
2     ORG 0100H
3     JMP start
4 len    DW    0
5 string DB    'Good Morning!', '$'
6 msg    DB    '字符串 string 的长度= ', '$'
7 ;
8 %include "..\mymacro\strlen.mac"
9 %include "..\mymacro\dispstr.mac"
10 %include "..\mymacro\dispi.mac"
11 %include "..\mymacro\newline.mac"
12 ;
13 start:
14     dispstr string          ;显示 string 字符串
15     strlen string, '$', len ;计算 string 长度存入 len
16     newline                ;换行
17     dispstr msg            ;显示 msg 字符串

```



```
18      dispi len          ;显示字整数值
19      RET
```

【运行】

```
howasm\ch10> nasmw strlen.asm -o strlen.com <Enter>
```

```
howasm\ch10> strlen <Enter>
```

字符串 string = Good Morning!

字符串 string 的长度= 13

10-3 基本字符串指令

对于字符串的基本操作，80x86 处理器提供 5 个指令，如表 10-1 所示。表 10-1 中指令后面有 SB 的表示 String Byte 字符串字节，有 SW 的表示 String Word 字符串字，有 SD 表示 String Double word。MOV 表示传送 move，CMP 表示比较 compare，SCA 表示扫描 scan，LOD 表示加载 load，STO 表示存储 store。

表 10-1 五个字符串基本操作指令

指令	功能	说明
CMPSB	比较	比较源字符串与目的字符串一个字符
CMPSW		一个字
CMPSD		一个双字
LODSB	加载	将源字符串一个指定的字符加载入 AL
LODSW		字加载入 AX
LODSD		双字加载入 EAX
MOVSB	传送	从源字符串传送一个字符至目的字符串
MOVSW		字至目的字
MOVSD		串双字至目的字符串
SCASB	扫描	扫描源字符串一个指定的字符
SCASW		字
SCASD		双字
STOSB	存储	将 AL 存储至字符串一个指定的字符地址
STOSW		AX 字地址
STOSD		EAX 双字地址

每一个字符串操作指令都需要两个操作数，但并不需要将这两个操作数表示在指令中，处理器自动定义寄存器操作数，例如 MOVSB、MOVSW 及 MOVSD 定义 SI 寄存器为源操作数的偏移值，定义 DI 寄存器为目的操作数的偏移值，表示从 SI 指定的地址将字符串拷贝至 DI 指定的地址。SI 表 Source Index 源索引，DI 表 Destination Index 目的索引。

字符串指令操作时，先要准备寄存器的偏移地址，然后再执行该指令。表 10-2 为字符串指令所使用的操作数。

表 10-2 字符串指令所使用的操作数

指令	第一个操作数	第二个操作数	搭配伪指令
MOVSb	DS:SI	ES:DI	REP
MOVSw	DS:SI	ES:DI	REP
MOVSD	DS:SI	ES:DI	REP
CMPSb	DS:SI	ES:DI	REPE, REPNE
CMPSw	DS:SI	ES:DI	REPE, REPNE
CMPSD	DS:SI	ES:DI	REPE, REPNE
SCASb	AL	ES:DI	REPE, REPNE
SCASw	AX	ES:DI	REPE, REPNE
SCASD	EAX	ES:DI	REPE, REPNE
LODSb	AL	DS:SI	
LODSw	AX	DS:SI	
LODSD	EAX	DS:SI	
STOSb	ES:DI	AL	REP
STOSw	ES:DI	AX	REP
STOSD	ES:DI	EAX	REP

SI 源索引寄存器搭配的分段寄存器定义为 DS，也可与 ES 搭配，但 DI 目的索引则一定要与分段寄存器 ES 搭配。

程序 movsb.asm 是一个字符串传送的例子，源字符串为 str1，其值为：

'Good Morning!'

目的字符串为 str2，其值为空。程序第 17 行使用 strlen 宏返回字符串长度并存入 len 变量里。第 19 行将 str1 地址存入 SI。第 20 行将 str2 地址存入 DI。第 21 行设定从左至右传送。这些都准备好了，才执行真正的传送操作，就是第 23 行的 MOVSb 指令。第 24 行指示重复执行传送的操作，共执行字符串长度次数，字符串长度若为 13 就重复传送 13 次，若字符串长度为 100 就重复传送 100 次。

【程序 movsb.asm】

```

1 ; ***** movsb.asm *****
2     ORG     0100H
3     JMP     start
4 msg1 DB     '源字符串= '
5 str1  DB     'Good Morning!', '$'
6 len   DW     0
7 msg2  DB     '目的字符串= '
8 str2  DB     ' ', '$'
9 msg3  DB     '目的字符串的长度= ', '$'
10 ;

```

```

11 %include "..\mymacro\strlen.mac"
12 %include "..\mymacro\dispi.mac"
13 %include "..\mymacro\dispstr.mac"
14 %include "..\mymacro\newline.mac"
15 ;
16 start:
17     strlen str1, '$', len    ;源字符串 str1 长度
18     MOV CX, WORD [len]      ;CX=str1 字符串长度
19     MOV SI, str1            ;源字符串 str1 地址
20     MOV DI, str2            ;目的字符串 str2 地址
21     CLD                    ;方向,从左至右
22 loop2:
23     MOVSB                    ;字节传送
24     LOOP loop2              ;重复 CX 次
25     strlen str2, '$', len    ;目的字符串 str2 长度
26     dispstr msg1             ;显示 msg 字符串(str1)
27     newline                  ;换行
28     dispstr msg2             ;显示 msg2 字符串(str2)
29     newline                  ;换行
30     dispstr msg3             ;显示 msg3 字符串
31     dispi len                ;显示 str2 字符串长度(len)
32     RET

```

【运行】

howasm\ch10> nasmw movsb.asm -o movsb.com <Enter>

howasm\ch10> movsb <Enter>

源字符串= Good Morning!

目的字符串= Good Morning!

目的字符串的长度= 13

程序 repmovsb.asm 程序主要的操作为 MOVSB, 配合重复的 REP 虚拟指令, 写成:

REP MOVSB

要执行这个重复传送指令之前要做一些准备的工作。首先将模板字符串定义出来, 本例题模板字符串 strpat 的值为 '*--*', 长度 lenpat 为 4。第 22 行声明源字符串偏移地址 SI, 其值为 strpat, 第 23 行声明目的字符串偏移地址 DI, 其值仍为 strpat, 表示来源及目的都是同一个字符串, 不过 DI 后来又增加模板字符串的长度, 因此 SI 较低, 而 DI 较高, 重复拷贝时就发生重叠的现象, 一直拷贝到 strpat 字符串结束时才停止。

【程序 repmovsb.asm】

```

1 ; ***** repmovsb.asm *****
2     ORG 0100H
3     JMP start
4 lenpat DW 4
5 len DW 44

```

```

6 strpat    DB      '*--*'
7          TIMES    40 DB ' '
8          DB '$'
9 msgpat DB      '模板字符串: ', '$'
10 msgfill DB     '填满模板之后: ', '$'
11 ;
12 %include "..\mymacro\dispstr.mac"
13 %include "..\mymacro\newline.mac"
14 ;
15 start:
16     dispstr msgpat                ;显示 msgpat 信息
17     newline                      ;换行
18     dispstr strpat                ;显示模板字符串 strpat
19     newline                      ;换行
20     MOV CX, [len]                ;CX=字符串 strpat 长度
21     SUB CX, WORD [lenpat]        ;减去模板长度
22     MOV SI, strpat               ;SI=源字符串 strpat 地址
23     MOV DI, strpat               ;DI=目的字符串 strpat 地址
24     ADD DI, WORD [lenpat]        ;跳过模板
25     CLD                         ;方向,从左至右
26     REP MOVSB                   ;重复拷贝模板
27     dispstr msgfill              ;显示 msgfill 信息
28     newline                     ;换行
29     dispstr strpat               ;显示目的字符串 strpat
30     RET

```

【运行】

```
howasm\ch10> nasmw repmovsb.asm -o repmovsb.com <Enter>
```

```
howasm\ch10> repmovsb <Enter>
```

```
模板字符串:
```

```
*--*
```

```
填满模板之后:
```

```
*-----*
```

程序 `cmpsb.asm` 对于源字符串 `str1` 及目的字符串 `str2` 执行 `CMPSB` 比较指令, 一个字符一个字符比较, 直到 `str1` 字符串结束为止。比较的结果, 到第 5 个 (从 0 算起) 字符时, 发现 'M' 与 'N' 不同, 'M' 较小, 因此输出不同的位置为 5。

【程序 cmpsb.asm】

```

1 ; ***** cmpsb.asm *****
2     ORG      0100H
3     JMP      start
4 len    DW      0
5 msg1    DB      '源字符串= '
6 str1    DB      'Good Morning!', '$'

```



```
7 msg2 DB      '目的字符串='
8 str2  DB      'Good Night!', '$'
9 pos   DW      0
10 msge DB      '相等', '$'
11 msgne DB     '不相等, 位置(从0算起) = ', '$'
12 ;
13 %include "..\mymacro\strlen.mac"
14 %include "..\mymacro\dispstr.mac"
15 %include "..\mymacro\newline.mac"
16 %include "..\mymacro\dispi.mac"
17 ;
18 start:
19     strlen    str1, '$', len ;源字符串 str1 长度
20     dispstr   msg1           ;显示 msg1 源字符串
21     newline   ;换行
22     dispstr   msg2           ;显示 msg2 目的字符串
23     newline   ;换行
24 ;
25     MOV       CX, WORD [len] ;CX=源字符串 str1 长度
26     MOV       SI, str1       ;SI=源字符串 str1 地址
27     MOV       DI, str2       ;DI=目的字符串 str2 地址
28     CLD           ;方向,从左至右
29     MOV       WORD [pos], 0  ;pos=0
30 loop2:
31     CMPSB        ;比较不相等或 CX 次
32     PUSHF        ;压入标志寄存器
33     dispi    pos ;显示刚比较字符位置
34     newline   ;换行
35     INC WORD [pos] ;位置增加一
36     POPF       ;弹出标志寄存器
37     LOOPE    loop2 ;若相等或 CX>0 则继续循环
38 ;
39     CMP CX, 0 ;字符串每一个字符均比较过?
40     JE      equal ;是,跳至 equal
41     DEC WORD [pos] ;位置值减一,从0计数
42     dispstr msgne ;显示不相等信息
43     dispi    pos ;显示 pos 数值
44     JMP next ;跳至 next
45 equal:
46     dispstr msge ;显示相等信息
47 next:
48     RET
```

【运行】

```
howasm\ch10> nasmw cmpsb.asm -o cmpsb.com <Enter>
```

```
howasm\ch10> cmpsb <Enter>
```

源字符串= Good Morning!

目的字符串= Good Night!

0

1

2

3

4

5

不相等, 位置(从0算起) = 5

程序 repcmpsb.asm 与上题 cmpsb.asm 类似, 只不过将 CMPSB 指令与 REP 相结合。

【程序 repcmpsb.asm】

```
1 ; ***** repcmpsb.asm *****
2     ORG 0100H
3     JMP     start
4 len  DW  0
5 msg1 DB  '源字符串= '
6 str1 DB  'Good Morning!', '$'
7 msg2 DB  '目的字符串= '
8 str2 DB  'Good Night!', '$'
9 pos  DW  0
10 msge DB  '相等', '$'
11 msgne DB  '不相等, 位置(从0算起) = ', '$'
12 ;
13 %include "..\mymacro\strlen.mac"
14 %include "..\mymacro\dispstr.mac"
15 %include "..\mymacro\newline.mac"
16 %include "..\mymacro\dispi.mac"
17 ;
18 start:
19     strlen      str1, '$', len      ;源字符串 str1 长度
20     dispstr     msg1                    ;显示 msg1 源字符串
21     newline
22     dispstr     msg2                    ;显示 msg2 目的字符串
23     newline
24 ;
25     MOV     CX, WORD [len]            ;CX=源字符串 str1 长度
26     MOV     SI, str1                  ;SI=源字符串 str1 地址
27     MOV     DI, str2                  ;DI=目的字符串 str2 地址
28     CLD                                ;方向,从左至右
```

```

29      MOV      WORD [pos], 0           ;pos=0
30      REPE      CMPSB                   ;比较不相等或 CX 次
31      JE        equal                  ;是,跳至 equal
32      SUB      SI, str1                 ;位置计数
33      DEC      SI                       ;位置计数减一,从 0 计数
34      MOV      WORD [pos], SI          ;pos=位置(从 0 计数)
35      dispstr   msgne                   ;显示不相等信息
36      dispi     pos                     ;显示 pos 数值
37      JMP      next                    ;跳至 next
38 equal:
39      dispstr   msgc                     ;显示相等信息
40 next:
41      RET

```

【运行】

howasm\ch10> nasmw repcmpsb.asm -o repcmpsb.com <Enter>

howasm\ch10> repcmpsb <Enter>

源字符串= Good Morning!

目的字符串= Good Night!

不相等, 位置(从 0 算起) = 5

程序 scasb.asm 在字符串 str2 中找指定的字符'N', 这个问题可使用 SCASB 指令来解决, 如程序第 26 行所示。

【程序 scasb.asm】

```

1 ; ***** scasb.asm *****
2      ORG 0100H
3      JMP start
4 len   DW 0
5 msg2  DB      '目的字符串= '
6 str2  DB      'Good Night!', '$'
7 char  DB      'N'
8 pos   DW 0
9 msgnf DB      'N 找不到!', '$'
10 msgf  DB      '找到 N, 所在位置(从 0 算起) = ', '$'
11 ;
12 %include "..\mymacro\strlen.mac"
13 %include "..\mymacro\dispi.mac"
14 %include "..\mymacro\dispstr.mac"
15 %include "..\mymacro\newline.mac"
16 ;
17 start:
18      strlen    str2, '$', len           ;目的字符串 str2 长度
19      dispstr   msg2                      ;显示目的字符串 str2
20      newline

```

```

21      MOV     AL, [char]           ;AL=搜寻字符'N'
22      MOV     CX, [len]           ;CX=目的字符串 str2 长度
23      MOV     DI, str2            ;DI=目的字符串 str2 地址
24      CLD                          ;方向,从左至右
25 loop2:
26      SCASB                        ;比较 AL 直到相等或比较 CX 次
27      PUSHF                        ;压入标志寄存器
28      dispi    pos                ;显示刚比较字符位置
29      newline                      ;换行
30      INC WORD [pos]              ;位置增加一
31      POPF                          ;弹出标志寄存器
32      LOOPNE loop2                ;若不相等或 CX>0 则继续
33 ;
34      CMP CX, 0                    ;CX 是否为 0
35      JE      notfound             ;CX=0, 跳至 notfound
36 found:
37      DEC WORD [pos]              ;位置值减一, 从 0 计数
38      dispstr msgf                 ;显示找到信息
39      dispi    pos                ;显示 pos 数值
40      JMP next                     ;跳至 next
41 notfound:
42      dispstr msgnf                ;显示找不到信息
43 next:
44      RET

```

【运行】

```
howasm\ch10> nasmw scasb.asm -o scasb.com <Enter>
```

```
howasm\ch10> scasb <Enter>
```

```
目的字符串= Good Night!
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
找到 N, 所在位置(从 0 算起) = 5
```

程序 lodsrb.asm 执行 LODSB 指令将源字符串的字符一个一个加载入寄存器 AL, 每加载一个字符之后随即执行 STOSB 指令将 AL 的值存入目的字符串 DI 所指的地址。事实上可以在加载后做适当的处理, 然后再存储至目的字符串, 本例题只是单纯的拷贝。

【程序 lodsrb.asm】

```

1 ; ***** lodsrb.asm *****
2      ORG 0100H
3      JMP start

```



```
4 len    DW    0
5 msg1   DB    '源字符串= '
6 str1   DB    'Good Morning!', '$'
7 msg2   DB    '目的字符串= '
8 str2   DB    'Good Night! ', '$'
9 pos    DW    0
10 msg3  DB    '执行 LODSB 及 STOSB 指令之后', 13, 10, '$'
11 ;
12 %include "..\mymacro\strlen.mac"
13 %include "..\mymacro\dispi.mac"
14 %include "..\mymacro\dispstr.mac"
15 %include "..\mymacro\newline.mac"
16 ;
17 start:
18     dispstr    msg1                ;显示源字符串 str1
19     newline                    ;换行
20     dispstr    msg2                ;显示目的字符串 str2
21     newline                    ;换行
22 ;
23     strlen     str1, '$', len      ;len=源字符串 str1 长度
24     MOV        CX, WORD [len]      ;CX=源字符串 str1 长度
25     MOV        SI, str1            ;源字符串 str1 地址
26     MOV        DI, str2            ;目的字符串 str2 地址
27     CLD                          ;方向,从左至右
28 loop2:
29     LODSB                    ;SI 字符地址加载 AL
30     STOSB                    ;AL 存储至 DI 字符地址
31     LOOP       loop2           ;继续循环
32 ;
33     dispstr    msg3                ;显示 msg3 信息
34     dispstr    msg2                ;显示拷贝后的字符串
35     RET
```

【运行】

howasm\ch10> nasmw lodsbs.asm -o lodsbs.com <Enter>

howasm\ch10> lodsbs <Enter>

源字符串= Good Morning!

目的字符串= Good Night!

执行 LODSB 及 STOSB 指令之后

目的字符串= Good Morning!

程序 repstosb.asm 是指令 STOSB 与 REP 的结合,将 AL 值重复存储至目的字符串地址。
本例题重复拷贝 36 个 '=' 至 str2 字符串。

【程序 repstosb.asm】

```

1 ; ***** repstosb.asm *****
2     ORG 0100H
3     JMP start
4 str2 TIMES 80 DB ' '
5     DB '$'
6 ;
7 %include "..\mymacro\dispstr.mac"
8 ;
9 start:
10    MOV AL, '=' ;AL='='
11    MOV CX, 36 ;CX=源字符串 str1 长度
12    MOV DI, str2 ;目的字符串 str2 地址
13    CLD ;方向,从左至右
14    REP STOSB ;AL 存储至 DI 字符地址
15    dispstr str2 ;显示 str2 信息
16    RET

```

【运行】

```

howasm\ch10> nasmw repstosb.asm -o repstosb.com <Enter>
howasm\ch10> repstosb <Enter>

```

程序 ucase.asm 执行 LODSB 指令后,将源字符串字符中的小写英文字母转换成大写英文字母之后,再存储至目的字符串。

【程序 ucase.asm】

```

1 ; ***** ucase.asm *****
2     ORG 0100H
3     JMP start
4 len DW 0
5 str1 DB 'Good Morning!', '$'
6 msg DB 'LODSB 后,将小写转成大写字母,再 STOSB', '$'
7 ;
8 %include "..\mymacro\strlen.mac"
9 %include "..\mymacro\dispi.mac"
10 %include "..\mymacro\dispstr.mac"
11 %include "..\mymacro\newline.mac"
12 ;
13 start:
14     strlen str1, '$', len ;len=源字符串 str1 长度
15     dispstr str1 ;显示源字符串 str1
16     newline ;换行
17 ;

```



```
18      MOV CX, WORD [len]      ;源字符串 str1 长度
19      MOV SI, str1            ;源字符串 str1 地址
20      MOV DI, str1            ;目的字符串 str1 地址
21      CLD                    ;方向,从左至右
22 loop2:
23      LODSB                   ;SI 字符地址加载 AL
24      CMP AL, 'a'             ;AL='a'?
25      JB next                 ;AL<'a',next
26      CMP AL, 'z'             ;AL='z'?
27      JA next                 ;AL>'z',next
28 a2z:
29      SUB AL, 20H             ;AL 转换成英文大写字母
30 next:
31      STOSB                   ;AL 存储至 DI 字符地址
32      LOOP loop2              ;继续循环
33 ;
34      dispstr msg             ;显示 msg 信息
35      newline ;换行
36      dispstr str1            ;显示转换后的 str1 字符串
37      RET
```

【运行】

```
howasm\ch10> nasmw ucase.asm -o ucase.com <Enter>
```

```
howasm\ch10> ucase <Enter>
```

```
Good Morning!
```

```
LODSB 后,将小写转成大写字母,再 STOSB
```

```
GOOD MORNING!
```

10-4 转换指令 XLATB

指令 XLATB 将指定的字节值转换为预先定义好的字节值。一个字节值的范围为 0 至 255, 因此要预先定义好最多 256 个字节的数据表, 这个表的偏移地址要存入 BX, 要转换的字节值要存入 AL, 而转换后的字节值会存回 AL, 覆盖原来的值。若要转换的字节值范围没那么大, 数据表可以不要那么大, 例如十六进制只需 16 个符号, 那么数据表只需 16 个符号就可以了。不过若要转换的字节值范围太大, 超过 256 个时, 指令 XLATB 就派不上用场了。

程序 xlatb.asm 第 4 行建立一个十六进制转换表 htab, 只有 16 个符号。第 11 行将 htab 的偏移地址存入 BX 中。第 5 行声明一个字节变量 num, 它的内容为 12, 第 12 行将它存入 AL 中。随即执行 XLATB 指令, 将 AL 的值 12 转换成 htab 表第 12 个 (从 0 算起) 字符, 就是 'C', 也就是说 12 相当于十六进制的 'C' 了。当然 AL 的内容必须在 0 至 15 间才可得到正确的字符, 否则会取得错误字符。

【程序 xlatb.asm】

```

1 ; ***** xlatb.asm *****
2     ORG 0100H
3     JMP start
4 htab DB '0123456789ABCDEF' ;十六进制数字表
5 num  DB 12
6 char DB ' '
7 ;
8 %include "..\mymacro\dispchr.mac"
9 ;
10 start:
11     MOV BX, htab ;BX=htab 偏移地址
12     MOV AL, BYTE [num] ;AL=索引值
13     XLATB ;AL=转换为 htab 对应字符
14     MOV BYTE [char], AL ;AL 传送至 char
15     dispchr char ;将 char 输出
16     RET

```

【运行】

```
howasm\ch10> nasmw xlatb.asm -o xlatb.com <Enter>
```

```
howasm\ch10> xlatb <Enter>
```

C

10-5 字符串宏

为了字符串操作方便，作者设计了许多宏，这里只说明使用这些宏的接口。

btostr bmem,dmem,endchar

将 bmem 地址中有符号数值转换为 ASCII 数字并存于 dmem 地址

bmem：字节地址

dmem：ASCII 数字地址

endchar：字符串结束符号（常量，传值）

ftostr dqmem,decimal_places,dmem,endchar

将 dqmem 地址中有符号数值转换为 ASCII 数字存于 dmem 地址

dqmem：四字地址

decimal_place：小数字数（常量，传值）

dmem：ASCII 数字地址

endchar：字符串结束符号（常量，传值）

itostr wmem, dmem, endchar

将 wmem 地址中有符号数值转换为 ASCII 数字并存于 dmem 地址

wmem：字地址

dmem：ASCII 数字地址



endchar: 字符串结束符号 (常量, 传值)

ltostr dwmem, dmem, endchar

将 dwmem 地址中有符号数值转换为 ASCII 数字后存于 dmem 地址

dwmem: 双字地址

dmem: ASCII 数字地址

endchar: 字符串结束符号 (常量, 传值)

strcat str1, str2, endchar

将 str2 字符串附加至 str1 字符串末端

str1: 第一字符串内存地址

str2: 第二字符串内存地址

endchar: 字符串结束符号常量

00H ASCIIZ 字符串与 ANSI C 字符串格式相同

0dH 从键盘输入并按<Enter>键结束

使用 21H, 而 AH=01H, 从键盘输入格式

'\$' 使用 21H, 而 AH=09H, 显示字符串的格式

strchr smem, endchar, searchchar, pos

将 smem 内存处字符串显示在屏幕上

smem: 内存地址

endchar: 字符串结束符号常量

searchchar: 搜索字符常量

pos: 位置从 0 算起 (字变量)

strcmp str1, str2, endchar, result

将 str1 与 str2 比较结果返回 result

str1: 第一字符串内存地址

str2: 第二字符串内存地址

endchar: 字符串结束符号常量

result: 若 str1<str2 则 result<0

若 str1=str2 则 result=0

若 str1>str2 则 result>0

result 为字

strcpy str1, str2, endchar

将 str2 字符串拷贝至 str1, 包含字符串结束符号

str1: 目的字符串内存地址

str2: 源字符串内存地址

endchar: 字符串结束符号常量

strfill str1, endchar, charconstant

将 charconstant 字符常量拷贝至 str1 字符串

str1: 字符串内存地址

endchar: 字符串结束符号常量

charconstant: 字符常量, 如 '*', '\', '\" 等。

strlen smem, endchar, len

计算 smem 内存处字符串长度并存入 len 字变量

smem: 内存地址

endchar: 字符串结束符号常量

len: 字符串长度 (字变量)

strmov str1, str2, endchar

将 str2 字符串拷贝至 str1, 不包含字符串结束符号

str1: 目的字符串内存地址

str2: 源字符串内存地址

endchar: 字符串结束符号常量

strncmp str1, str2, n, result

将 str1 与 str2 比较 n 个字符结果返回 result

str1: 第一字符串内存地址

str2: 第二字符串内存地址

n: 欲比较的字符数, n 为字

result: 若 $str1 < str2$ 则 $result < 0$

若 $str1 = str2$ 则 $result = 0$

若 $str1 > str2$ 则 $result > 0$

result 为字

strncpy str1, str2, n

将 str2 字符串 n 个字符拷贝至 str1

str1: 第一字符串内存地址

str2: 第二字符串内存地址

n: 欲拷贝的字符数

strnfill str1, endchar, n, charconstant

将 charconstant 字符常量拷贝 n 个至 str1 字符串

str1: 字符串内存地址

endchar: 字符串结束符号常量

n: 拷贝字符常量个数

charconstant: 字符常量, 如 '*', '\', '\" 等。

strtob str endchar bmem

将数字字符串 str 转换成无符号二进制数并存于 bmem 字节

str: 字符串所存放的内存地址

endchar: 字符串结束符号常量

bmem: 转换成有符号二进制数值并存于 bmem 字节地址

strtod str endchar dqmem

将数字字符串 str 转换成浮点数并存于 dqmem

str: 字符串所存放的内存地址



endchar : 字符串结束符号常量

dqmem : 转换成浮点数并存于 dqmem

strtoi str endchar wmem

将数字字符串 str 转换成有符号二进制数并存于 wmem

str : 字符串所存放的内存地址

endchar : 字符串结束符号常量

wmem : 转换成有符号二进制数并存于 wmem 字地址

strtol str endchar dqmem

将数字字符串 str 转换成有符号二进制数并存于 dqmem 双字

str : 字符串所存放内存地址

endchar : 字符串结束符号常量

dqmem : 转换成有符号二进制数并存于 dqmem 双字地址

strtoul str endchar dqmem

将数字字符串 str 转换成无符号二进制并存于 dqmem 双字

str : 字符串所存放的内存地址

endchar : 字符串结束符号常量

dqmem : 转换成无符号二进制数值存于 dqmem 双字地址

strtoub str endchar bmem

将数字字符串 str 转换成无符号二进制并存于 bmem 字节

str : 字符串所存放内存地址

endchar : 字符串结束符号常量

bmem : 转换成无符号二进制数值并存于 bmem 字节地址

strtoui str endchar wmem

将数字字符串 str 转换成无符号二进制并存于 wmem 字

str : 字符串所存放的内存地址

endchar : 字符串结束符号常量

wmem : 转换成无符号二进制数值并存于 wmem 字地址

ubtostr bmem, dmem, endchar

将 bmem 地址中的数值转换为 ASCII 数字后存于 dmem 地址

bmem : 字节地址

dmem : ASCII 数字地址

endchar : 字符串结束符号 (常量, 传值)

uitostr wmem, dmem, endchar

将 wmem 地址中的数值转换为 ASCII 数字后存于 dmem 地址

wmem : 字地址

dmem : ASCII 数字地址

endchar : 字符串结束符号 (常量, 传值)

ultostr dqmem, dmem, endchar

将 dqmem 地址中的数值转换为 ASCII 数字后存于 dmem 地址

dwmem: 双字地址

dmem: ASCII 数字地址

endchar: 字符串结束符号 (常量, 传值)

程序 strfill.asm 将字符串填满指定的符号, 本题将字符串 str2 填满 '*'。

【程序 strfill.asm】

```
1 ; ***** strfill.asm *****
2     ORG 0100H
3     JMP start
4 str2 DB      'Good Night!', '$'
5 msg2 DB      '调用 strfill 宏填满星号之后', 13, 10, '$'
6 ;
7 %include "..\mymacro\strfill.mac"
8 %include "..\mymacro\dispstr.mac"
9 %include "..\mymacro\newline.mac"
10 ;
11 start:
12     dispstr str2                ;显示字符串 str2
13     newline                    ;换行
14     strfill str2, '$', '*'     ;填满星号
15     dispstr msg2               ;显示 msg2 信息
16     dispstr str2              ;显示字符串 str2
17     RET
```

【运行】

howasm\ch10> nasmw strfill.asm -o strfill.com <Enter>

howasm\ch10> strfill <Enter>

Good Night!

调用 strfill 宏填满星号之后

程序 strnfill.asm 调用 strnfill 宏, 在 str2 字符串第 5 个字符开始填入 5 个星号。

【程序 strnfill.asm】

```
1 ; ***** strnfill.asm *****
2     ORG 0100H
3     JMP start
4 n     DW 5
5 str2 DB      'Good Night!', '$'
6 ;
7 %include "..\mymacro\strnfill.mac"
8 %include "..\mymacro\dispstr.mac"
9 %include "..\mymacro\newline.mac"
10 ;
11 start:
```



```
12      dispstr str2                ;显示字符串 str2
13      newline                    ;换行
14      strnfill    str2+5, '$', n, '*' ;填入n个星号
15      dispstr str2                ;显示字符串 str2
16      RET
```

【运行】

```
howasm\ch10> nasmw strnfill.asm -o strnfill.com <Enter>
```

```
howasm\ch10> strnfill <Enter>
```

Good Night!

Good *****!

程序 `strchr.asm` 调用 `strchr` 宏, 在 'Good Morning!' 字符串中查找 'M' 字符, 结果存于字变量 `pos` 中, 在第 5 个 (从 0 算起) 地址找到。

【程序 `strchr.asm`】

```
1 ; ***** strchr.asm *****
2      ORG 0100H
3      JMP start
4 pos   DW 0
5 str1  DB      'Good Morning!', '$'
6 ;
7 %include "..\mymacro\strchr.mac"
8 %include "..\mymacro\dispi.mac"
9 ;
10 start:
11      strchr str1, '$', 'M', pos    ;str1 中找 'M' 位置存入 pos
12      dispi  pos                    ;显示 pos 位置 (从 0 计数)
13      RET
```

【运行】

```
howasm\ch10> nasmw strchr.asm -o strchr.com <Enter>
```

```
howasm\ch10> strchr <Enter>
```

5

程序 `strcpy.asm` 调用 `strcpy` 宏, 将字符串 `str1` 拷贝至 `str2`。

【程序 `strcpy.asm`】

```
1 ; ***** strcpy.asm *****
2      ORG 0100H
3      JMP start
4 result DW 0
5 str1  DB      'Good Morning!', '$'
6 str2  DB      'Good Night!', '$'
7 ;
8 %include "..\mymacro\strcpy.mac"
9 %include "..\mymacro\dispstr.mac"
```

```

10 %include "..\mymacro\newline.mac"
11 ;
12 start:
13 dispstr str1 ;显示 str1
14 newline ;换行
15 dispstr str2 ;显示 str2
16 newline ;换行
17 strcpy str2, str1, '$' ;拷贝 str1 至 str2
18 dispstr str2 ;显示 str2
19 RET

```

【运行】

```

howasm\ch10> nasmw strcpy.asm -o strcpy.com <Enter>
howasm\ch10> strcpy <Enter>
    Good Morning!
    Good Night!
    Good Morning!

```

程序 strcat.asm 调用 strcat 宏, 将字符串 str2 附加至 str1 末端。

【程序 strcat.asm】

```

1 ; ***** strcat.asm *****
2     ORG 0100H
3     JMP start
4 result    DW 0
5 str1 DB    'Good Morning!', '$'
6 TIMES 80 DB ' '
7 str2 DB    'Good Night! ', '$'
8 ;
9 %include "..\mymacro\strcat.mac"
10 %include "..\mymacro\dispstr.mac"
11 %include "..\mymacro\newline.mac"
12 ;
13 start:
14     dispstr str1 ;显示 str1
15     newline ;换行
16     dispstr str2 ;显示 str2
17     newline ;换行
18     strcat str1, str2, '$' ;str2 附至 str1 尾端
19     dispstr str1 ;显示 str1
20     RET

```

【运行】

```

howasm\ch10> nasmw strcat.asm -o strcat.com <Enter>
howasm\ch10> strcat <Enter>

```



Good Morning!

Good Night!

Good Morning!Good Night!

程序 strncpy.asm 调用 strncpy 宏, 将字符串 str2 拷贝 n 个字符至 str1。本例题 str2 字符串变量的值为'Good Morning!', n 的值从 1 开始, 每次增加 1, 直到 str2 字符串结束为止。

【程序 strncpy.asm】

```
1 ; ***** strncpy.asm *****
2     ORG 0100H
3     JMP start
4 str2 DB      'Good Morning!', '$'
5 len  DW      0
6 n     DW      4
7 msg  DB      '执行 strncpy 宏之后,n 值每次增一', '$'
8 str1 DB      ' ', '$'
9 ;
10 %include "..\mymacro\strlen.mac"
11 %include "..\mymacro\strncpy.mac"
12 %include "..\mymacro\dispstr.mac"
13 %include "..\mymacro\newline.mac"
14 ;
15 start:
16     dispstr str2                ;显示 str2
17     newline                     ;换行
18     dispstr msg                 ;显示 msg
19     newline                     ;换行
20     strlen str2, '$', len       ;len=str2 长度
21     MOV CX, WORD [len]          ;CX=str2 长度
22     MOV WORD [n], 1             ;n=1
23 loop2:
24     strncpy str1, str2, n        ;str2 拷贝 n 个字符至 str1
25     dispstr str1                ;显示 str1
26     newline                     ;换行
27     INC WORD [n]                ;n=n+1
28     LOOP    loop2              ;继续循环
29     RET
```

【运行】

howasm\ch10> nasmw strncpy.asm -o strncpy.com <Enter>

howasm\ch10> strncpy <Enter>

Good Morning!

执行 strncpy 宏之后,n 值每次增一

G

Go

Goo
Good
Good
Good M
Good Mo
Good Mor
Good Morn
Good Morni
Good Mornin
Good Morning
Good Morning!

程序 numtostr.asm 将整数变量 numi, 其值-12345, 使用宏 itostr 转换成字符串'-12345', 长整数变量 numl, 其值 123456789, 使用宏 ultostr 转换成字符串'123456789', 浮点数变量 numf, 其值-12345678.0789, 使用宏 ftostr 转换成字符串'-12345678.0789'。

【程序 numtostr.asm】

```
1 ; ***** numtostr.asm *****
2     ORG 0100H
3     JMP start
4 numi DW -12345           ;整数
5 numl DD 123456789       ;长整数
6 numf DQ -12345678.0789  ;浮点数
7 stri TIMES 30 DB ' '    ;整数字符串
8 strl TIMES 30 DB ' '    ;长整数字符串
9 strf TIMES 30 DB ' '    ;浮点数字符串
10 ;
11 %include "..\mymacro\dispstr.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\itostr.mac"
14 %include "..\mymacro\ultostr.mac"
15 %include "..\mymacro\ftostr.mac"
16 ;
17 start:
18     FINIT                ;浮点堆栈初始化
19     ftostr numf, 4, strf, '$' ;浮点数转换成 strf 字符串
20     dispstr strf          ;显示 strf 字符串
21     newline              ;换行
22     ultostr numl, strl, '$' ;无号长整数转换成 strl 字符串
23     dispstr strl          ;显示 strl 字符串
24     newline              ;换行
25     itostr numi, stri, '$' ;有号整数转换成 stri 字符串
26     dispstr stri          ;显示 stri 字符串
27     newline              ;换行
```



28 RET

【运行】

```
howasm\ch10> nasmw numtostr.asm -o numtostr.com <Enter>
howasm\ch10> numtostr <Enter>
-12345678.0789 [注]-12345678.0789 浮点数转换成字符串
123456789 [注]123456789 无符号长整数转换成字符串
-12345 [注]-12345 有符号整数转换成字符串
```

程序 `strtonum.asm` 将整数字符串'987' 使用宏 `strtoi` 转换成数值 987, 长整数字符串'-987654321' 使用宏 `strtol` 转换成数值-987654321, 浮点数字符串'987654.0321' 使用宏 `strtod` 转换成数值 987654.0321。

【程序 `strtonum.asm`】

```
1 ; ***** strtonum.asm *****
2     ORG 0100H
3     JMP start
4     stri DB '987', '$' ;整数字符串
5     strl DB '-987654321', '$' ;长整数字符串
6     strf DB '987654.0321', '$' ;浮点数字符串
7     numi DW 0
8     numl DD 0
9     numf DQ 0.0
10 ;
11 %include "..\mymacro\dispstr.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\strtoi.mac"
14 %include "..\mymacro\strtol.mac"
15 %include "..\mymacro\strtod.mac"
16 %include "..\mymacro\dispi.mac"
17 %include "..\mymacro\displ.mac"
18 %include "..\mymacro\dispf.mac"
19 ;
20 start:
21     FINIT
22     strtoi stri, '$', numi ;整数字符串转换成数值 numi
23     dispi numi ;显示 numi
24     newline ;换行
25     strtol strl, '$', numl ;长整数字符串转换成数值 numl
26     displ numl ;显示 numl
27     newline ;换行
28     strtod strf, '$', numf ;浮点数字符串转换成数值 numf
29     dispf numf, 4 ;显示 numf
30     newline ;换行
31     RET
```

【运行】

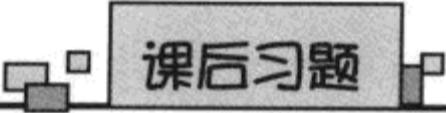
```
howasm\ch10> nasmw strtonum.asm -o strtonum.com <Enter>
```

```
howasm\ch10> strtonum <Enter>
```

```
987 [注]字符串'987'转换成数值
```

```
-987654321 [注]字符串'-987654321'转换成数值
```

```
987654.0321 [注]字符串'987654.0321'转换成数值
```


 课后习题

1. 设计一个程序，以字符数组 s 表示字符串“I like NASM!!”，然后使用 21H 中断的 02H 功能依次显示字符。
2. 设计一个程序，以字符数组 s 表示字符串“I like NASM!!”，然后使用 21H 中断的 09H 功能将整个字符串显示出来。
3. 设计一个程序，以字符数组 s 表示字符串“早安!”，然后调用 strlen 宏返回字符串 s 的长度，并显示于屏幕。
4. 设计一个程序，以字符串变量 s 表示字符串“I like NASM!!”，并将它拷贝至另一个字符串变量 s2。
5. 设计一个程序，以字符串变量 s 表示字符串“I like NASM!!”，并将它前面六个字符拷贝至另一个字符串变量 s2。
6. 设计一个程序，以字符串变量 s 表示字符串“I like ”，字符串变量 s2 表示字符串“NASM!!”，并将 s2 附加在 s1 后面。
7. 设计一个程序，以字符串变量 s1 表示字符串“abcdefgh”，字符串变量 s2 表示字符串“abcdef”，请显示 s1 与 s2 比较的结果。
8. 设计一个程序，以字符串变量 s 表示字符串“Happy birthday to you!”，从键盘输入一个字符存入字节变量 ch2，显示 ch2 是否在 s 字符串中，若找到请问在第几个位置找到（从 0 算起）。
9. 设计一个程序，将字符串 s=“365.25” 分开为两个字符串，整数字符串 s1=“365” 及小数字符串 s2=“25”。

【运行】

Downward shift: a character is shifted one position in the alphabet to the left. For example, 'b' becomes 'a', 'c' becomes 'b', and so on. 'z' becomes 'y'.
Upward shift: a character is shifted one position in the alphabet to the right. For example, 'a' becomes 'b', 'z' becomes 'a', and so on.



1. 字符串的存储：在内存中，字符串是以字符数组的形式存储的。每个字符占用一个字节的空间。字符串的结束标志是空字符 '\0'。

2. 字符串的输入输出：可以使用 scanf 和 printf 函数来输入和输出字符串。scanf 函数可以读取字符串直到遇到空格、换行符或结束符。printf 函数可以输出字符串。

3. 字符串的比较：可以使用 strcmp 函数来比较两个字符串。strcmp 函数返回一个整数，如果两个字符串相等，则返回 0；如果第一个字符串大于第二个字符串，则返回正数；如果第一个字符串小于第二个字符串，则返回负数。

4. 字符串的查找：可以使用 strstr 函数来查找一个字符串是否在另一个字符串中出现。strstr 函数返回指向子字符串的指针，如果找不到，则返回 NULL。

5. 字符串的复制：可以使用 strcpy 函数来复制一个字符串到另一个字符串中。strcpy 函数返回指向目标字符串的指针。

6. 字符串的拼接：可以使用 strcat 函数来将一个字符串追加到另一个字符串的末尾。strcat 函数返回指向目标字符串的指针。

7. 字符串的截取：可以使用 strncpy 函数来截取一个字符串的一部分。strncpy 函数返回指向目标字符串的指针。

8. 字符串的逆序：可以使用 reverse 函数来逆序一个字符串。reverse 函数返回指向目标字符串的指针。

9. 字符串的排序：可以使用 qsort 函数来对一个字符串数组进行排序。qsort 函数返回指向目标字符串数组的指针。

10. 字符串的加密解密：可以使用 Caesar 密码来加密和解密字符串。Caesar 密码是一种简单的替换密码，通过将每个字母在字母表中向前或向后移动一定的位数来实现加密和解密。

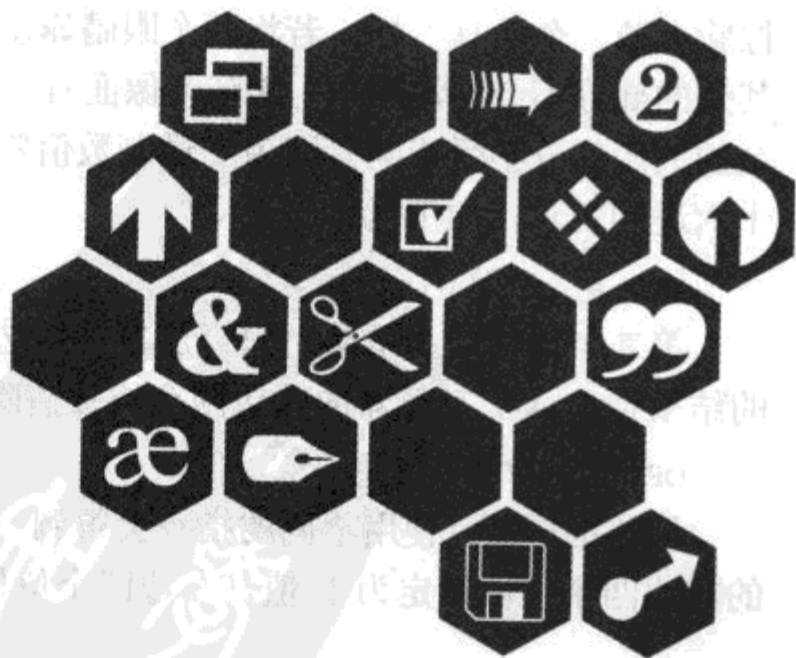


位运算

这里有两个理由说明为什么要了解位运算 (bit operation)。第一个理由是可以使用位来表示标志 (flag)，80x86 处理器提供标志寄存器用于存储若干运算的执行结果，包括状态标志 (status flag) 及控制标志 (control flag)，这些标志是以位的方式存储的，通过位运算可以设定 (set) 标志的值，也可以清除 (clear) 标志的值。第二个理由是可以精细地控制所处理的数据，我们都知道计算机内存的最小单元为位，经常需要更改数据中的某一位。所以了解位运算可以说是基本的需要了。

在本章将会学习到两种有关位运算的指令:

1. 在一个字节、字或双字里变更及测试某一位的值, 例如指令 AND、OR、XOR、NOT 及 TEST。
2. 在一个字节、字或双字里改变某一位的位置, 例如指令 SHL、SHR、SAL、SAR、ROL、ROR、RCL 及 RCR。




```
XOR AL, 00001111B
```

指令 XOR 执行的结果强迫 AL 寄存器的左边四个位值保持不变，右边四个位则取其补码。

11-3 AND 指令

AND 指令用来关闭第一操作数指定的位。第二操作数作为屏蔽，其值不变。在屏蔽里一个 0 值位会关闭第一操作数相对应的位，一个 1 值位会保留第一操作数相对应的位值不变。

```
AND AL, 00001111B
```

第二操作数为屏蔽，屏蔽其左边四个位值为 0，强迫 AL 寄存器的左边四个位被清除为 0 值，屏蔽右边四个位值为 1，让 AL 右边四个位保存原来的值。若屏蔽的值全部为 0 值，关闭第一操作数相对应的位值。如下例将 BX 寄存器的内容清除为 0 值。

```
AND BX, 0000H
```

若屏蔽的值全部为 1 值，则第一操作数相对应的位值保留原值。如下例内存变量 value 的内容保留原值。

```
AND WORD [value], 0ffffH
```

一个最常用的 AND 运算是将指定的位关闭，其余的位打开，如下例的屏蔽最右边位关闭，其余打开：

```
AND CX, 1111111111111110B
```

两个操作数相对应的位值若均为 1 值则 AND 的结果为 1，否则为 0 值。

第一操作数位值	第二操作数位值	AND 的结果
0	0	0
0	1	0
1	0	0
1	1	1

例如字符'5'的 ASCII 码为 35H，而数值 5 为 05H。要将字符'5'转换为数值 5，只须将左边四个位值清除为 0 值即可。

```
AND AL, 00001111B
```

若是字则是：

```
AND AX, 0000111100001111B
```

当然也可用十六进制表示如下：

```
AND AX, 0F0FH
```

11-4 OR 指令

OR 指令用来打开第一操作数指定的位。第二操作数作为屏蔽，其值不变。在屏蔽里一个 1 值位会打开第一操作数相对应的位，一个 0 值位会保留第一操作数相对应的位值不变。



```
OR AL, 00110000B
```

第二个操作数为屏蔽，屏蔽其右边四个位值为 0，让 AL 右边四个位保存原来的值。屏蔽左边四个位，其中低两位值为 1，高两位其值为 0，强迫 AL 寄存器的左边四个位中低两位打开，高两位保存原来的值。事实上如此操作可将 AL 的 BCD 码转换为 ASCII 码。

若屏蔽的值全部为 0 值，则保留第一操作数位值不变。如下例中 BX 寄存器的内容并不会受影响。

```
OR BX, 0000H
```

若屏蔽的值全部为 1 值，则第一操作数相对应的位值被强迫设定为 1 值。如下例内存变量 value 的内容变为 0ffffH。

```
OR WORD [value], 0ffffH
```

一个最常用的 OR 运算是将指定的位打开，其余的位关闭，如下例的屏蔽最右边位打开，其余关闭：

```
OR CX, 00000000000000001B
```

如果两个操作数相对应的位值中有一个为 1 则 OR 的结果为 1，否则为 0 值。

第一操作数位值	第二操作数位值	OR 的结果
0	0	0
0	1	1
1	0	1
1	1	1

例如字符 '5' 的 ASCII 码为 35H，而数值 5 为 05H。要将数值 5 转换为字符 '5'，只须将左边四个位的值设定为 0011 即可。

```
OR AL, 00110000B
```

若是字则是：

```
OR AX, 0011000000110000B
```

当然也可用十六进制表示如下：

```
OR AX, 3030H
```

11-5 XOR 指令

XOR 指令用来取第一操作数指定的位的补码。第二操作数作为屏蔽，其值不变。在屏蔽里一个 1 值位会将第一操作数相对应的位变为补码，一个 0 值位会使第一操作数相对应的位值保持不变。

```
XOR AL, 11110000B
```

第二个操作数为屏蔽，屏蔽其右边四个位值为 0，让 AL 右边四个位保存原来的值。其左边四个位为 1，AL 左边四个位若为 0 值则变为 1 值，若为 1 值则改变为 0 值。若 AL 值为 11001100B，则执行上述 XOR 运算后，AL 值变为 00111100B。

若屏蔽的值全部为 0 值，则第一操作数位值保存原来的值。

```
XOR BX, 0000H
```

若屏蔽的值全部为 1 值，第一操作数位值若为 0 值则变为 1 值，若为 1 值则改变为 0 值。如下例内存变量 value 的内容原来为 1010101010101010B，则经下述 XOR 运算后其值变为 0101010101010101B。

```
XOR WORD [value], 0ffffH
```

一个最常用的 XOR 运算是将指定的位以其补码取代，其余的位保持不变，如下例的屏蔽最右边位打开，其余关闭：

```
XOR CX, 00000000000000001B
```

两个操作数相对应的位值若两个均为 1 或均为 0，则 XOR 的结果为 0，否则为 1。

第一操作数位值	第二操作数位值	XOR 的结果
0	0	0
0	1	1
1	0	1
1	1	0

11-6 NOT 指令

NOT 指令用来取惟一操作数所有位的补码。操作数所有位中值为 0 的变为 1，值为 1 的变为 0。例如 AL 的值为 11110000B，经过 NOT 运算后其值变为 00001111B。

```
NOT AL
```

又如下例内存变量 value 的内容原来为 1010101010101010B，则经过 NOT 运算后其值改变为 0101010101010101B。

```
NOT WORD [value]
```

事实上 NOT 的运算相当于 XOR 的屏蔽位全部设为 1。

```
XOR AX, 1111111111111111B
```

程序 bitbyte.asm 对两个字节变量 a 及 b 执行基本的位运算 AND、OR、XOR 及 NOT，a、b 以及运算的结果 c 均使用 disp8bit 宏输出。

```
disp8bit bmem
```

将 bmem 字节值以 0 或 1 的 ASCII 数字显示在屏幕上

bmem : 字节地址

【程序 bitbyte.asm】

```
1 ; ***** bitbyte.asm *****
2     ORG 0100H
3     JMP start
4 a    DB    11010101B
5 b    DB    11101100B
6 c    DB    0
```

```

7 msga DB ' a = ', '$'
8 msgb DB ' b = ', '$'
9 msgand DB 'a AND b = ', '$'
10 msgor DB 'a OR b = ', '$'
11 msgxor DB 'a XOR b = ', '$'
12 msgnot DB ' NOT a = ', '$'
13 ;
14 %include "..\mymacro\disp8bit.mac"
15 %include "..\mymacro\dispstr.mac"
16 %include "..\mymacro\newline.mac"
17 ;
18 start:
19     dispstr msga ;显示 msga 信息
20     disp8bit a ;显示变量 a 内含每一位值
21     newline ;换行
22     dispstr msgb ;显示 msgb 信息
23     disp8bit b ;显示变量 b 内含每一位值
24     newline ;换行
25 ;
26     MOV AL, [a]
27     AND AL, BYTE [b]
28     MOV BYTE [c], AL ;c=a AND b
29     dispstr msgand ;显示 msgand 信息
30     disp8bit c ;显示变量 c 内含每一位值
31     newline ;换行
32 ;
33     MOV AL, [a]
34     OR AL, BYTE [b]
35     MOV BYTE [c], AL ;c=a OR b
36     dispstr msgor ;显示 msgor 信息
37     disp8bit c ;显示变量 c 内含每一位值
38     newline ;换行
39 ;
40     MOV AL, [a]
41     XOR AL, BYTE [b]
42     MOV BYTE [c], AL ;c=a XOR b
43     dispstr msgxor ;显示 msgxor 信息
44     disp8bit c ;显示变量 c 内含每一位值
45     newline ;换行
46 ;
47     MOV AL, [a]
48     NOT AL
49     MOV BYTE [c], AL ;c=NOT a

```

```

50 dispstr msgnot ;显示 msgnot 信息
51 disp8bit c ;显示变量 c 内含每一位值
52 newline ;换行
53 RET

```

【运行】

```

howasm\ch11> nasmw bitbyte.asm -o bitbyte.com <Enter>
howasm\ch11> bitbyte <Enter>
a          = 1 1 0 1 0 1 0 1
b          = 1 1 1 0 1 1 0 0
a AND b    = 1 1 0 0 0 1 0 0
a OR b     = 1 1 1 1 1 1 0 1
a XOR b    = 0 0 1 1 1 0 0 1
NOT a      = 0 0 1 0 1 0 1 0

```

11-7 TEST 指令

TEST 指令的格式如下：

TEST 第一操作数, 第二操作数

TEST 指令检查第一操作数, 看看某个指定的位是否已被打开, 就像 AND、OR 及 XOR 指令一样, 第一个操作数可以是寄存器或内存变量, 第二个操作数可以是寄存器、内存变量或立即数。第二操作数为屏蔽, 其值不变, 在屏蔽里一个 1 值表示第一操作数相对应的位要接受检验, 其值为 1 或 0。

```
TEST AL, 11110000B
```

表示第一操作数 AL 相对应的左边四个位要接受检验, 其值为 1 或 0, 右边四个位则被忽略。

TEST 检验的结果并不会改变任何操作数的值, 但它会更新零标志 (ZF) 的值。事实上 TEST 指令与 AND 的执行结果相同, 只不过没有将结果存回第一操作数而已。这表示若程序流程根据 TEST 的结果来决定分岔, 则在 TEST 之后马上跟着一个条件跳跃指令, 例如:

```

TEST AL, 00000001B
JNZ bitOn ;AL 最右边位其值为 1
JZ bitOff ;AL 最右边位其值为 0

```

TEST 指令最常见的用法是建立一个循环, 直到某指定的位的值为 1 或 0。下列例子从键盘读入一个字符至 char, 测试 char 字符是否为奇数, 若是则继续循环, 若不是则结束循环。

```

loop2:
    readchr char ;从键盘读入一位数字
    TEST BYTE [char], 01H ;测试是否为奇数
    JNZ loop2 ;是奇数, 继续循环
evenNum:

```

程序 testbyte.asm 从键盘输入一位数字至字节变量 char, 若为奇数则最低位的值为 1, 若为偶数则最低位的值为 0, 奇数为 '1'、'3'、'5'、'7'、'9', 偶数为 '0'、'2'、'4'、'6'、'8'。因此只需



要使用 TEST 指令测试 char 的第 0 位的值是 1 还是 0 就可判断 char 是奇数还是偶数了, 屏蔽可设为 00000001B, 其他位均关闭, 只保留最低的第 0 位测试就可以。测试的结果会设定标志寄存器里的零标志 ZF 值, 零标志 ZF 值若为 0 值表示 char 的最低位值为 1, 因此 JNZ 条件跳跃成立时表示 char 为奇数。

反之, 若零标志 ZF 值为 1, 则表示 char 的最低位值为 0, 因此 JZ 条件跳跃成立时表示 char 为偶数。

执行时输入 1, 标志寄存器的 ZF 值为 0, 因此判断为奇数。

```
testbyte <Enter>
```

请输入一位数字: 1 <Enter>

标志寄存器: 0 0 0 0 0 0 1 0

|
ZF

输入一个奇数!

执行时输入 2, 标志寄存器的 ZF 值为 1, 因此判断为偶数。

```
testbyte <Enter>
```

请输入一位数字: 2 <Enter>

标志寄存器: 0 1 0 0 0 1 1 0

|
ZF

输入一个偶数!

【程序 testbyte.asm】

```
1 ; ***** testbyte.asm *****
2     ORG 0100H
3     JMP start
4 msg   DB      '请输入一位数字: ', '$'
5 char  DB      ' '
6 flag  DW      0
7 msgflag DB    '标志寄存器: ', '$'
8 msgodd DB     '输入一个奇数!', '$'
9 msgeven DB    '输入一个偶数!', '$'
10 ;
11 %include "..\mymacro\readchr.mac"
12 %include "..\mymacro\disp8bit.mac"
13 %include "..\mymacro\dispstr.mac"
14 %include "..\mymacro\newline.mac"
15 ;
16 start:
17     dispstr msg                ;显示 msg 信息
18     readchr char               ;从键盘输入一位数字
19     newline                    ;换行
20     TEST     BYTE [char], 01H  ;测试是否为奇数
```

```

21      PUSHF
22      POP WORD [flag]      ;flag=标志寄存器值
23      dispstr msgflag      ;显示 msgflag 信息
24      disp8bit flag        ;显示低 8 位标志
25      newline              ;换行
26 ;
27      PUSH WORD [flag]
28      POPF                  ;标志寄存器值=flag 值
29      JNZ oddNum            ;是奇数
30      JZ evenNum            ;是偶数
31 oddNum:
32      dispstr msgodd        ;显示奇数信息
33      JMP next
34 evenNum:
35      dispstr msgeven        ;显示偶数信息
36 next:
37      RET

```

【运行】

```
howasm\ch11> nasmw testbyte.asm -o testbyte.com <Enter>
```

```
howasm\ch11> testbyte <Enter>
```

```
    请输入一位数字: 1 <Enter>
```

```
    标志寄存器: 0 0 0 0 0 0 1 0
```

```
    输入一个奇数!
```

```
howasm\ch11> testbyte <Enter>
```

```
    请输入一位数字: 2 <Enter>
```

```
    标志寄存器: 0 1 0 0 0 1 1 0
```

```
    输入一个偶数!
```

11-8 改变位位置

下面的章节要来讨论在字节、字和双字中改变位的位置。表 11-1 说明这些指令的意义。这些指令都有共同的指令格式:

指令名称 操作数, 移位计数

例如:

```
SHL BYTE [value], 1
```

操作数必须是寄存器或内存变量, 且必须是字节、字或双字。上例中操作数为字节, 移位计数为 1 表示内存变量 value 字节内容左移一个位。移位计数可以为立即数, 例如 1, 也可以存入寄存器 CL 中, 例如:

```
MOV CL, 2
```

```
SHL BYTE [value], CL
```

当想重复左移时, 可使用 CL 寄存器。可逐次改变 CL 的值。

表 11-1 改变位的指令

指令名称	说明
SHL	左移 (shift left)
SHR	右移 (shift right)
SAL	算术左移 (shift arithmetic left)
SAR	算术右移 (shift arithmetic right)
ROL	循环左移 (shift rotate left)
ROR	循环右移 (shift rotate right)
RCL	带进位循环左移 (shift rotate with carry left)
RCR	带进位循环右移 (shift rotate with carry right)

11-9 左移及右移

左移指令 SHL 及右移指令 SHR 所移出的位均丢失。最后移出的位的值存入进位标志 (CF) 里面。

程序 shlbyte.asm 第 4 行声明一个内存字节变量 a，其值为：

```
1 1 0 1 0 1 0 1 B
```

逐次左移，第一次左移时，将最左边位的值为 1 的移出，存入进位标志里面，a 及标志寄存器的值如 shlbyte.asm 程序输出所示。

```
SHL a, 1 = 1 0 1 0 1 0 1 0  flag reg = 1 0 0 1 0 1 1 1
                                     | |      |
                                     SF ZF      CF
```

从输出可以看出，左移一位时将 a 最左边位 1 移出而丢失，最右边的空位被补上一个 0 值。也可以看到标志寄存器的变化情形，进位标志 CF 的值为 1，这个 1 就是被左移丢失的那个位值。符号位 SF 值为 1，表示左移一位后 a 的符号位值为 1，表示负数。零标志 ZF 值为 0，表示 a 的值不为 0。若 a 的值为 0 则 ZF 的值为 1。

【程序 shlbyte.asm】

```
1 ; ***** shlbyte.asm *****
2      ORG 0100H
3      JMP start
4 a      DB      11010101B
5 flag   DW  0
6 count  DW  8
7 msga   DB      ' a =', '$'
8 msg    DB      'SHL a,1 =', '$'
9 msgflag DB      ' flag reg =', '$'
10 ;
11 %include "..\mymacro\disp8bit.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\dispstr.mac"
```

```

14 ;
15 start:
16     dispstr msga           ;显示 msga 信息
17     disp8bit  a           ;显示 a 的各位值
18     newline               ;换行
19 ;
20     MOV CX, WORD [count]   ;CX=count 的值(8)
21 loop2:
22     SHL BYTE [a], 1       ;a 左移一位
23     PUSHF
24     POP WORD [flag]
25     dispstr msg           ;显示 msg 信息
26     disp8bit  a           ;显示 a 的各位值
27     dispstr msgflag       ;显示 msgflag 信息
28     disp8bit  flag        ;显示 flag 的各位值
29     newline               ;换行
30     DEC CX
31     JZ      endjob
32     JMP loop2             ;继续
33 endjob:
34     RET

```

【运行】

```
howasm\ch11> nasmw shlbyte.asm -o shlbyte.com <Enter>
```

```
howasm\ch11> shlbyte <Enter>
```

```

a = 1 1 0 1 0 1 0 1
SHL a,1 = 1 0 1 0 1 0 1 0 flag reg = 1 0 0 1 0 1 1 1
SHL a,1 = 0 1 0 1 0 1 0 0 flag reg = 0 0 0 1 0 0 1 1
SHL a,1 = 1 0 1 0 1 0 0 0 flag reg = 1 0 0 1 0 0 1 0
SHL a,1 = 0 1 0 1 0 0 0 0 flag reg = 0 0 0 1 0 1 1 1
SHL a,1 = 1 0 1 0 0 0 0 0 flag reg = 1 0 0 1 0 1 1 0
SHL a,1 = 0 1 0 0 0 0 0 0 flag reg = 0 0 0 1 0 0 1 1
SHL a,1 = 1 0 0 0 0 0 0 0 flag reg = 1 0 0 1 0 0 1 0
SHL a,1 = 0 0 0 0 0 0 0 0 flag reg = 0 1 0 1 0 1 1 1

```

```

| | |
SF ZF CF

```

程序 shrbyte.asm 第 4 行声明一个内存字节变量 a，其值为：

```
1 1 0 1 0 1 0 1 B
```

逐次右移，第一次右移时，将最右边位的值为 1 的移出，存入进位标志里面，a 及标志寄存器的值如 shrbyte.asm 程序输出所示。

```

SHR a,1 = 0 1 1 0 1 0 1 0 flag reg = 0 0 0 1 0 1 1 1

```

```

| | |
SF ZF CF

```



从输出可以看出右移一位时, 将 a 最右边位 1 的移出而丢失, 最左边的空位被补上一个 0 值。也可以看到标志寄存器的变化情形, 进位标志 CF 的值为 1, 这个 1 就是被右移丢失的那个位值。符号位 SF 值为 0, 表示右移一位后 a 的符号位值为 0, 表示正数。零标志 ZF 值为 0, 表示 a 的值不为 0。若 a 的值为 0 则 ZF 的值为 1。

【程序 shrbyte.asm】

```
1 ; ***** shrbyte.asm *****
2     ORG 0100H
3     JMP start
4 a     DB      11010101B
5 flag  DW      0
6 count DW      8
7 msga  DB      ' a =', '$'
8 msg    DB      'SHR a,1 =', '$'
9 msgflag DB    ' flag reg =', '$'
10 ;
11 %include "..\mymacro\disp8bit.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\dispstr.mac"
14 ;
15 start:
16     dispstr msga                ;显示 msga 信息
17     disp8bit    a               ;显示 a 的各位值
18     newline                    ;换行
19 ;
20     MOV CX, WORD [count]        ;CX=count 的值(8)
21 loop2:
22     SHR BYTE [a], 1             ;a 右移一位
23     PUSHF
24     POP     WORD [flag]
25     dispstr msg                 ;显示 msg 信息
26     disp8bit    a               ;显示 a 的各位值
27     dispstr msgflag             ;显示 msgflag 信息
28     disp8bit    flag            ;显示 flag 的各位值
29     newline                    ;换行
30     DEC CX
31     JZ      endjob
32     JMP loop2                   ;继续
33 endjob:
34     RET
```

【运行】

howasm\ch11> nasmw shrbyte.asm -o shrbyte.com <Enter>

```
howasm\ch11> shrbyte <Enter>
```

```
a = 1 1 0 1 0 1 0 1
```

```
SHR a,1 = 0 1 1 0 1 0 1 0 flag reg = 0 0 0 1 0 1 1 1
```

```
SHR a,1 = 0 0 1 1 0 1 0 1 flag reg = 0 0 0 1 0 1 1 0
```

```
SHR a,1 = 0 0 0 1 1 0 1 0 flag reg = 0 0 0 1 0 0 1 1
```

```
SHR a,1 = 0 0 0 0 1 1 0 1 flag reg = 0 0 0 1 0 0 1 0
```

```
SHR a,1 = 0 0 0 0 0 1 1 0 flag reg = 0 0 0 1 0 1 1 1
```

```
SHR a,1 = 0 0 0 0 0 0 1 1 flag reg = 0 0 0 1 0 1 1 0
```

```
SHR a,1 = 0 0 0 0 0 0 0 1 flag reg = 0 0 0 1 0 0 1 1
```

```
SHR a,1 = 0 0 0 0 0 0 0 0 flag reg = 0 1 0 1 0 1 1 1
```

11-10 算术左移及算术右移

算术左移 SAL 及算术右移 SAR 指令是针对有符号整数的移位,算术右移时左边的空位以最左边的符号位补充,算术左移时右边的空位以 0 补充。

程序 salbyte.asm 第 4 行声明一个内存字节变量 a, 其值为:

```
1 1 0 1 0 1 0 1 B
```

逐次算术左移,第一次算术左移时,将最左边位的值 1 移出,存入进位标志里面, a 及标志寄存器的值如 salbyte.asm 程序输出所示。

```
SAL a,1 = 1 0 1 0 1 0 1 0 flag reg = 1 0 0 1 0 1 1 1
```

```

      | |      |
      SF ZF    CF

```

从输出可以看出算术左移一位时,将 a 最左边位 1 移出而丢失,最右边的空位被补上一个 0 值。也可以看到标志寄存器的变化情形,进位标志 CF 的值为 1,这个 1 就是被算术左移丢失的那个位值。符号位 SF 值为 1,表示算术左移一位后的 a 符号位值为 1,表示负数。零标志 ZF 值为 0,表示 a 的值不为 0。若 a 的值为 0 则 ZF 的值为 1。

【程序 salbyte.asm】

```

1 ; ***** salbyte.asm *****
2     ORG 0100H
3     JMP start
4 a     DB     11010101B
5 flag  DW     0
6 count DW     8
7 msga  DB     ' a =', '$'
8 msg    DB     'SAL a,1 =', '$'
9 msgflag DB     ' flag reg =', '$'
10 ;
11 %include "..\mymacro\disp8bit.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\dispstr.mac"
14 ;

```

```

15 start:
16     dispstr msga                ;显示 msga 信息
17     disp8bit    a                ;显示 a 的各位值
18     newline                ;换行
19 ;
20     MOV CX, WORD [count]        ;CX=count 的值(8)
21 loop2:
22     SAL BYTE [a], 1            ;a 算术左移一位
23     PUSHF
24     POP     WORD [flag]
25     dispstr msg                ;显示 msg 信息
26     disp8bit    a                ;显示 a 的各位值
27     dispstr msgflag            ;显示 msgflag 信息
28     disp8bit    flag            ;显示 flag 的各位值
29     newline                ;换行
30     DEC CX
31     JZ     endjob
32     JMP loop2                ;继续
33 endjob:
34     RET

```

【运行】

howasm\ch11> nasmw salbyte.asm -o salbyte.com <Enter>

howasm\ch11> salbyte <Enter>

```

a = 1 1 0 1 0 1 0 1
SAL a,1 = 1 0 1 0 1 0 1 0 flag reg = 1 0 0 1 0 1 1 1
SAL a,1 = 0 1 0 1 0 1 0 0 flag reg = 0 0 0 1 0 0 1 1
SAL a,1 = 1 0 1 0 1 0 0 0 flag reg = 1 0 0 1 0 0 1 0
SAL a,1 = 0 1 0 1 0 0 0 0 flag reg = 0 0 0 1 0 1 1 1
SAL a,1 = 1 0 1 0 0 0 0 0 flag reg = 1 0 0 1 0 1 1 0
SAL a,1 = 0 1 0 0 0 0 0 0 flag reg = 0 0 0 1 0 0 1 1
SAL a,1 = 1 0 0 0 0 0 0 0 flag reg = 1 0 0 1 0 0 1 0
SAL a,1 = 0 0 0 0 0 0 0 0 flag reg = 0 1 0 1 0 1 1 1

```

程序 sarbyte.asm 第 4 行声明一个内存字节变量 a，其值为：

1 1 0 1 0 1 0 1 B

逐次算术右移，第一次算术右移时，将最右边位的值 1 移出，存入进位标志里面，a 及标志寄存器的值如 sarbyte.asm 程序输出所示。

```

SAR a,1 = 1 1 1 0 1 0 1 0 flag reg = 1 0 0 1 0 0 1 1
                | |           |
                SF ZF         CF

```

从输出可以看出算术右移一位时将 a 最右边位 1 移出而丢失，最左边的空位被补上一个符号位 1 值。也可以看到标志寄存器的变化情形，进位标志 CF 的值为 1，这个 1 就是被算术右移丢失的那个位值。符号位 SF 值为 1，表示算术右移一位后的 a 符号位值为 1，表示

负数。零标志 ZF 值为 0, 表示 a 的值不为 0。若 a 的值为 0 则 ZF 的值为 1。

【程序 sarbyte.asm】

```

1 ; ***** sarbyte.asm *****
2     ORG 0100H
3     JMP start
4 a     DB      11010101B
5 flag  DW      0
6 count  DW      8
7 msga   DB      ' a =', '$'
8 msg     DB      'SAR a,1 =', '$'
9 msgflag DB      ' flag reg =', '$'
10 ;
11 %include "..\mymacro\disp8bit.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\dispstr.mac"
14 ;
15 start:
16     dispstr msga           ;显示 msga 信息
17     disp8bit a             ;显示 a 的各位值
18     newline               ;换行
19 ;
20     MOV CX, WORD [count]   ;CX=count 的值(8)
21 loop2:
22     SAR BYTE [a], 1        ;a 算术右移一位
23     PUSHF
24     POP WORD [flag]
25     dispstr msg            ;显示 msg 信息
26     disp8bit a             ;显示 a 的各位值
27     dispstr msgflag        ;显示 msgflag 信息
28     disp8bit flag          ;显示 flag 的各位值
29     newline               ;换行
30     DEC CX
31     JZ      endjob
32     JMP loop2              ;继续
33 endjob:
34     RET

```

【运行】

howasm\ch11> nasmw sarbyte.asm -o sarbyte.com <Enter>

howasm\ch11> sarbyte <Enter>

a = 1 1 0 1 0 1 0 1

SAR a,1 = 1 1 1 0 1 0 1 0 flag reg = 1 0 0 1 0 0 1 1

SAR a,1 = 1 1 1 1 0 1 0 1 flag reg = 1 0 0 1 0 1 1 0



```
SAR a,1 = 1 1 1 1 1 0 1 0 flag reg = 1 0 0 1 0 1 1 1
SAR a,1 = 1 1 1 1 1 1 0 1 flag reg = 1 0 0 1 0 0 1 0
SAR a,1 = 1 1 1 1 1 1 1 0 flag reg = 1 0 0 1 0 0 1 1
SAR a,1 = 1 1 1 1 1 1 1 1 flag reg = 1 0 0 1 0 1 1 0
SAR a,1 = 1 1 1 1 1 1 1 1 flag reg = 1 0 0 1 0 1 1 1
SAR a,1 = 1 1 1 1 1 1 1 1 flag reg = 1 0 0 1 0 1 1 1
```

11-11 循环位移

循环位移指令事实上是从移位指令扩展而来的，移位指令所移出的位丢失，循环位移只是将丢失不见的位捡回来，置于另一端而已。ROL 及 ROR 指令针对无符号整数循环位移，RCL 及 RCR 指令针对有符号整数循环位移。

程序 rolbyte.asm 第 4 行声明一个内存字节变量 a，其值为：

```
1 1 0 1 0 1 0 1 B
```

逐次循环左移，第一次循环左移时，将最左边位的值 1 移出，存入进位标志里面，并转入最右边的空位里，a 及标志寄存器的值如 rolbyte.asm 程序输出所示。

```
ROL a, 1 = 1 0 1 0 1 0 1 1 flag reg = 0 0 0 0 0 0 1 1
```

ZF

CF

从输出可以看出循环左移一位时将 a 最左边位 1 移出，存入最右边的空位。也可以看到标志寄存器的变化情形，进位标志 CF 的值为 1，这个 1 就是被循环左移丢失的那个位值。循环位移是针对无符号整数的，因此 SF 标志并无意义。零标志 ZF 值为 0，表示 a 的值不为 0。若 a 的值为 0 则 ZF 的值为 1。

【程序 rolbyte.asm】

```
1 ; ***** rolbyte.asm *****
2     ORG 0100H
3     JMP start
4 a     DB      11010101B
5 flag  DW      0
6 count DW      8
7 msga  DB      ' a =', '$'
8 msg    DB      'ROL a,1 =', '$'
9 msgflag DB      ' flag reg =', '$'
10 ;
11 %include "..\mymacro\disp8bit.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\dispstr.mac"
14 ;
15 start:
16     dispstr msga          ;显示 msga 信息
17     disp8bit      a      ;显示 a 的各位值
```

```

18      newline                                ;换行
19 ;
20      MOV CX, WORD [count]                  ;CX=count 的值(8)
21 loop2:
22      ROL BYTE [a], 1                       ;a 循环左移一位
23      PUSHF
24      POP WORD [flag]
25      dispstr msg                           ;显示 msg 信息
26      disp8bit a                           ;显示 a 的各位值
27      dispstr msgflag                       ;显示 msgflag 信息
28      disp8bit flag                        ;显示 flag 的各位值
29      newline                              ;换行
30      DEC CX
31      JZ      endjob
32      JMP loop2                             ;继续
33 endjob:
34      RET

```

【运行】

```
howasm\ch11> nasmw rolbyte.asm -o rolbyte.com <Enter>
```

```
howasm\ch11> rolbyte <Enter>
```

```

a = 1 1 0 1 0 1 0 1
ROL a,1 = 1 0 1 0 1 0 1 1 flag reg = 0 0 0 0 0 0 1 1
ROL a,1 = 0 1 0 1 0 1 1 1 flag reg = 0 0 0 0 0 0 1 1
ROL a,1 = 1 0 1 0 1 1 1 0 flag reg = 0 0 0 0 0 1 1 0
ROL a,1 = 0 1 0 1 1 1 0 1 flag reg = 0 0 0 0 0 1 1 1
ROL a,1 = 1 0 1 1 1 0 1 0 flag reg = 0 0 0 0 0 0 1 0
ROL a,1 = 0 1 1 1 0 1 0 1 flag reg = 0 0 0 0 0 1 1 1
ROL a,1 = 1 1 1 0 1 0 1 0 flag reg = 0 0 0 0 0 0 1 0
ROL a,1 = 1 1 0 1 0 1 0 1 flag reg = 0 0 0 0 0 0 1 1

```

程序 `rorbyte.asm` 第 4 行声明一个内存字节变量 `a`，其值为：

```
1 1 0 1 0 1 0 1 B
```

逐次循环右移，第一次循环右移时，将最右边位值 1 移出，存入进位标志里面，并转存入最左边的空位里，`a` 及标志寄存器的值如 `rorbyte.asm` 程序输出所示。

```

ROR a, 1 = 1 1 1 0 1 0 1 0 flag reg = 0 0 0 0 0 0 1 1
                                   |           |
                                   ZF         CF

```

从输出可以看出循环右移一位时，将 `a` 最右边位 1 移出，并存入最左边的空位。也可以看到标志寄存器的变化情形，进位标志 `CF` 的值为 1，这个 1 就是被循环右移丢失的那个位值。循环位移是针对无符号整数的，因此 `SF` 标志并无意义。零标志 `ZF` 其值为 0，表示 `a` 的值不为 0。若 `a` 的值为 0 则 `ZF` 的值为 1。



【程序 rorbyte.asm】

```
1 ; ***** rorbyte.asm *****
2     ORG 0100H
3     JMP start
4 a     DB     11010101B
5 flag  DW  0
6 count DW  8
7 msga  DB     ' a =', '$'
8 msg    DB     'ROR a,1 =', '$'
9 msgflag DB     ' flag reg =', '$'
10 ;
11 %include "..\mymacro\disp8bit.mac"
12 %include "..\mymacro\newline.mac"
13 %include "..\mymacro\dispstr.mac"
14 ;
15 start:
16     dispstr msga                ;显示 msga 信息
17     disp8bit    a              ;显示 a 的各位值
18     newline                ;换行
19 ;
20     MOV CX, WORD [count]      ;CX=count 的值(8)
21 loop2:
22     ROR BYTE [a], 1           ;a 循环右移一位
23     PUSHF
24     POP WORD [flag]
25     dispstr msg                ;显示 msg 信息
26     disp8bit    a              ;显示 a 的各位值
27     dispstr msgflag           ;显示 msgflag 信息
28     disp8bit    flag          ;显示 flag 的各位值
29     newline                ;换行
30     DEC CX
31     JZ     endjob
32     JMP loop2                ;继续
33 endjob:
34     RET
```

【运行】

howasm\ch11> nasmw rorbyte.asm -o rorbyte.com <Enter>

howasm\ch11> rorbyte <Enter>

a = 1 1 0 1 0 1 0 1

ROR a,1 = 1 1 1 0 1 0 1 0 flag reg = 0 0 0 0 0 0 1 1

ROR a,1 = 0 1 1 1 0 1 0 1 flag reg = 0 0 0 0 0 0 1 0

```
ROR a,1 = 1 0 1 1 1 0 1 0 flag reg = 0 0 0 0 0 1 1 1
ROR a,1 = 0 1 0 1 1 1 0 1 flag reg = 0 0 0 0 0 1 1 0
ROR a,1 = 1 0 1 0 1 1 1 0 flag reg = 0 0 0 0 0 0 1 1
ROR a,1 = 0 1 0 1 0 1 1 1 flag reg = 0 0 0 0 0 1 1 0
ROR a,1 = 1 0 1 0 1 0 1 1 flag reg = 0 0 0 0 0 0 1 1
ROR a,1 = 1 1 0 1 0 1 0 1 flag reg = 0 0 0 0 0 0 1 1
```

11-12 位移及循环位移指令总结

图 11-1 说明位移及循环位移指令的操作情形。

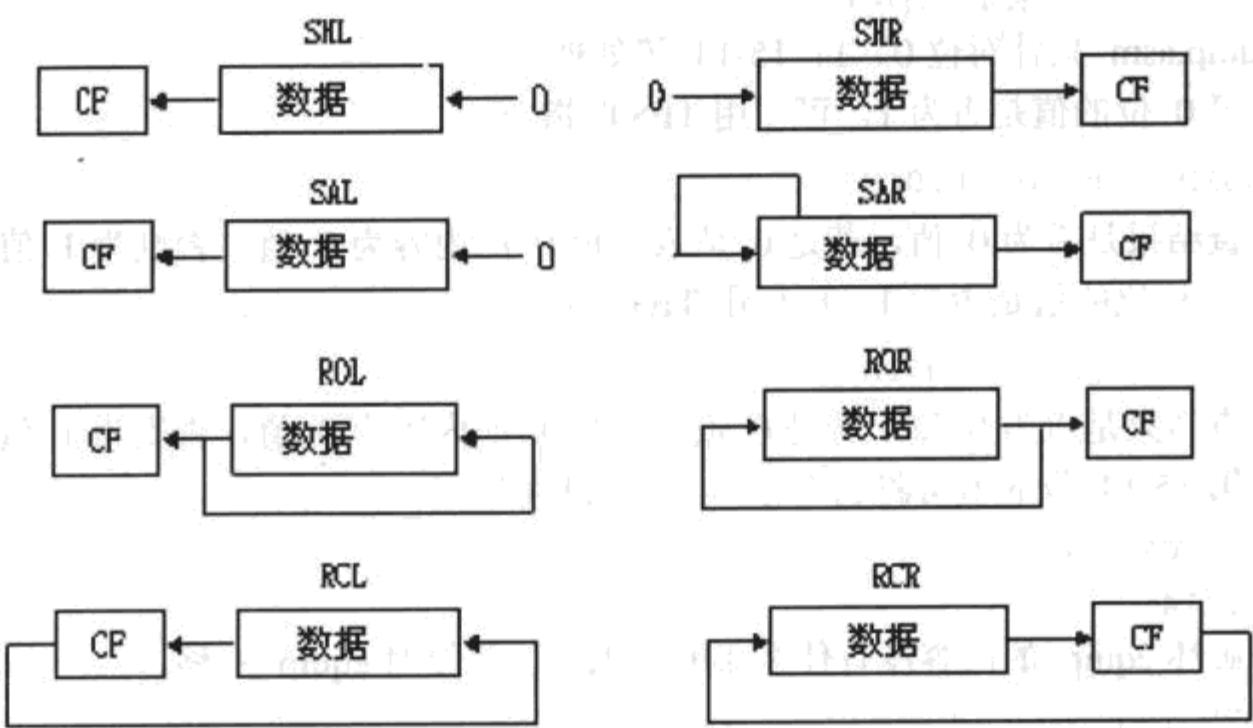


图 11-1 位移及循环位移指令的操作情形

SHL 向左位移，右边空位补 0。SHR 向右位移，左边空位补 0。SAL 向左位移，右边空位补 0。SAR 向右位移，左边空位补符号位值。ROL 循环左移，移出的位填入右边空位。ROR 循环右移，转出的位填入左边空位。RCL 带着 CF 位循环左移，转出的位填入 CF，原 CF 值填入右边空位。RCR 带着 CF 位循环右移，移出的位填入 CF，原 CF 值填入左边空位。

11-13 综合例题

程序 equip.asm 调用 11H 中断，向 AX 返回计算机的相关设备状态，随即存入字变量 equip 中，它代表的意义如下：

- 位 0 : 值 1 表示安装软盘。
- 位 1 : 值 1 表示安装协处理器。
- 位 2 : 值 1 表示安装鼠标。
- 位 3 : 保留。
- 位 5- 4 : 表示影像(video) 模式初值。
值 00 表示没有使用。

值 01 表示 40×25 彩色。
值 10 表示 80×25 彩色。
值 11 表示 80×25 黑白。

位 7-6 : 表示磁盘数目。
值 00 表示 1 个。
值 01 表示 2 个。
值 10 表示 3 个。
值 11 表示 4 个。

位 11-9 : 表示 RS232 串行端口数目。

位 3 : 保留。

位 15-14 : 表示并行端口数目。

程序 equip.asm 只针对位 0、1、15-14 等处理。

要测试第 0 位的值是否为 1, 可使用 TEST 指令:

```
TEST WORD [equip], 0001H
```

然后检查结果是否为 0 值, 若是 0 值表示位 0 的内容为 0 值, 否则为 1 值。

要测试第 1 位的值是否为 1, 可使用 TEST 指令:

```
TEST WORD [equip], 0002H
```

然后检查结果是否为 0 值, 若是 0 值表示位 1 的内容为 0 值, 否则为 1 值。

要取得第 15-14 位的值可将它右移 14 个位就可以了。

```
MOV AX, WORD [equip]
```

```
SHR AX, 14
```

若认为破坏 equip 的内容没有什么影响, 也可以直接对 equip 右移。

```
SHR WORD [equip], 14
```

【程序 equip.asm】

```
1 ; ***** equip.asm *****
2     ORG 0100H
3     JMP start
4 equip DW 0
5 numport DW 0
6 msg0 DB '有安装软盘', 13, 10, '$'
7 msg1 DB '有安装协处理器', 13, 10, '$'
8 msg15 DB '提供并行端口个数='
9 portstr DB ' '
10 ;
11 %include "..\mymacro\dispstr.mac"
12 %include "..\mymacro\dispi.mac"
13 %include "..\mymacro\itostr.mac"
14 ;
15 start:
16     INT 11H
17     MOV WORD [equip], AX
```

```

18 ;
19     TEST     WORD [equip], 0001H
20     JZ       next
21     dispstr msg0
22 next:
23     TEST     WORD [equip], 0002H
24     JZ       next2
25     dispstr msg1
26 next2:
27     MOV AX, WORD [equip]
28     SHR AX, 14
29     MOV WORD [numport], AX
30     itostr numport, portstr, '$'
31     dispstr msg15
32     RET

```

【运行】

```
howasm\ch11> nasmw equip.asm -o equip.com <Enter>
```

```
howasm\ch11> equip <Enter>
```

有安装软盘

有安装协处理器

提供并行端口个数= 3

宏 `randgen.mac` 是一个随机数产生器，产生 0 至 65535 的随机数，存于参数内存地址。第一次调用 `randgen` 宏时它以任意数 9357H 为起点，加上当前时间的秒数及百分秒数所组成的字，循环右移 4 个位后产生一个新的随机数，将这个新的随机数返回调用程序，同时将这个新的随机数存入 `%%rnd` 变量中，下回就以这个 `%%rnd` 值为起点，再产生一个新的随机数，如此循环不已。

【宏 randgen.mac】

```

1 ; ***** randgen.mac *****
2 ;
3 ; randgen    rmem
4 ; 随机数产生器，0~65535，存于 rmem 内存变量地址
5 ; rmem : 字内存变量地址
6 ;
7 %ifndef RANDGEN_MAC
8 %define RANDGEN_MAC
9 %MACRO randgen 1
10         PUSH A                ;存储一般性寄存器
11         JMP     %%begin
12 %%rnd    DW     9357H          ;任意数
13 %%begin:
14         MOV AH, 2cH

```

```

15      INT 21H                ;取得当前时间
16      MOV AX, WORD [%rnd]
17      ADD AX, DX              ;DH 秒数 DL 百分秒数
18      MOV CL, 4               ;4 个位
19      ROR AX, CL              ;循环右移 4 个位
20      MOV WORD [%rnd], AX     ;新的随机数
21      MOV WORD [%1], AX       ;返回新的随机数
22      POPA                    ;恢复一般性寄存器
23 %ENDMACRO
24 %endif

```

程序 rnd.asm 用于测试宏 randgen 是否正确。第 13 行引入 randgen.mac 宏文件。第 18 行调用 randgen 宏产生一个随机数 randnum。第 20 行测试随机数第 0 位是否为 0 值, 若是则 sum0 值增 1, 若不是则 sum1 值增 1。最后将 sum1 及 sum0 值显示出来。

【程序 rnd.asm】

```

1 ; ***** rnd.asm *****
2 ;
3      ORG 0100H
4      JMP start
5 msg1      DB      13, 10, '出现 1 的次数= ', '$'
6 msg0      DB      13, 10, '出现 0 的次数= ', '$'
7 sum1      DW      0
8 sum0      DW      0
9 randnum   DW      0
10 ;
11 %include "..\mymacro\dispui.mac"
12 %include "..\mymacro\dispstr.mac"
13 %include "..\mymacro\randgen.mac"
14 ;
15 start:
16      MOV CX, 40000            ;4 万次
17 loop2:
18      randgen randnum          ;随机数 randnum
19      MOV AX, WORD [randnum]   ;AX=随机数 randnum
20      TEST  AL, 00000001B      ;测试第 0 位
21      JZ    next0              ;0 值
22      INC WORD [sum1]          ;1 值, 累积 sum1
23      JMP  next
24 next0:
25      INC WORD [sum0]          ;0 值, 累积 sum0
26 next:
27      LOOP loop2              ;重复
28 ;

```

```

29      dispstr msg1      ;显示 msg1
30      dispui sum1      ;显示 sum1
31      dispstr msg0      ;显示 msg0
32      dispui sum0      ;显示 sum0
33      RET

```

【运行】

```
howasm\ch11> nasmw rnd.asm -o rnd.com <Enter>
```

```
howasm\ch11> rnd <Enter>
```

```
出现 1 的次数= 20017
```

```
出现 0 的次数= 19983
```

课后习题

- 假设 AX 内容为 7f79H, 有一个内存变量 a, 其值为 0e3e3H, 请将下列指令的执行结果输出。
 - AND AX, WORD [a]
 - OR AX, WORD [a]
 - XOR AX, WORD [a]
 - XOR AX, AX
 - NOT WORD [a]
- 请将上题中每执行一个指令后将标志寄存器值输出。
- 设计一个程序, 从键盘输入一个字符串, 将英文大写字母转换为小写字母后输出。以输入 “I like NASM!!” 进行测试。
- 设计一个程序, 从键盘输入一个字符串, 将英文小写字母转换为大写字母后输出。以输入 “I like NASM!!” 进行测试。
- 设计一个程序, 从键盘输入一个字符串, 将英文大写字母转换为小写字母, 小写字母转换为大写字母后输出。以输入 “I like NASM!!” 进行测试。
- 重复从键盘输入整数, 利用右移一位相当于除以 2 的特性, 判断是奇数还是偶数, 统计一下所输入的奇数及偶数个数。
- 从键盘输入整数, 每次右移一位, 显示标志寄存器的每一个标志值。
- 从键盘输入整数, 每次左移一位, 显示标志寄存器的每一个标志值。
- 从键盘输入整数, 每次循环右移一位, 显示标志寄存器的每一个标志值。
- 从键盘输入整数, 每次循环左移一位, 显示标志寄存器的每一个标志值。
- 从键盘输入整数, 每次算术右移一位, 显示标志寄存器的每一个标志值。
- 从键盘输入整数, 每次算术左移一位, 显示标志寄存器的每一个标志值。



13. 中断 21H 的 2cH 功能返回秒数 (DH) 及秒数的百分数 (DL), 将 DX 加上原来的随机数 (例如 9357H) 后右移四位产生一个新的随机数, 请实现一个程序。
14. 调用 11H 中断, 返回计算机的相关设备状态于 AX, 取得位 7-6 表示磁盘数目显示在屏幕上。
15. 使用 randgen 宏测试 10000 个随机数中奇数及偶数个数。
16. 使用 randgen 宏测试骰子投掷 10000 次中, 出现 1 至 6 点的次数。

12

文件处理

输入及输出的 (Input/Output) 简称为 I/O, 牵涉到数据在文件或设备之间传送的操作。通常数据都从键盘或磁盘文件输入, 输出至屏幕、磁盘文件或打印机。一般而言输入及输出是计算机系统较复杂的部分, 因为程序对于这些输入及输出设备必须做到精细的控制, 但是众所周知的是, 每一种输入及输出设备都有它独特的地方, 这也就是它复杂的所在。还好 DOS 操作系统提供大部分的输入及输出服务。





12-1 输入及输出层次

要了解输入及输出，可将计算机的输入及输出看成三种不同的层次。

最低的层次是，程序对于输入及输出设备，一个个字节的输入或输出，在这个层次程序无可逃避的要对输入及输出设备做精细的控制。这个层次的程序设计当然较为困难。

中等层次的输入及输出，可以使用 DOS 系统所提供的 BIOS，将低层次的输入及输出工作交给 BIOS，这样就解除了对输入及输出设备做精细控制的负担，不过仍然要维护逻辑的数据结构。

最高层次的输入及输出，可以使用 DOS 系统所提供的函数调用，这部分的服务类似其他高级语言所提供的较为成熟的输入及输出接口，在这个层次可以将文件及设备以一种标准的方式处理，也可将设备看成一种特殊的文件。

显然使用最高层次的输入及输出是最容易设计的一种，事实上计算机软体厂商也希望使用 DOS 系统所提供的函数调用，虽然设备厂商会改变，但这些输入及输出的接口是不会改变的，这样程序就可以历久而常新了。

12-2 输入及输出概念

对于输入及输出的函数调用有一些概念不可不知。

1. 要处理一个文件之前必须打开该文件

当打开一个文件时必须指明路径及文件名。打开之后 DOS 会返回一个 16 位的整数，称为文件代号 (file handle)，从此之后程序用这个文件代号就可以了。

2. 处理一个文件完毕之后必须关闭该文件

关闭文件时 DOS 会帮忙做善后的工作。必须养成习惯，用完的文件必须关闭，否则可能会造成有些数据丢失。

3. 当读写数据时必须提供一个缓冲区

读写数据时必须提供一块足够的内存以存储这些数据，这块内存称为缓冲区 (buffer)。读数据时将数据存入缓冲区，写数据时从这个缓冲区写出。

4. 使用 ASCIIZ 字符串

当标明文件名或路径名称时要使用 ASCIIZ 字符串，这种字符串以 00H 作为字符串结束符号。当从键盘输入路径时必须使用反斜线“\”作为隔开符号，不过当使用 ASCIIZ 字符串时可以使用反斜线或一般的斜线作为隔开符号。

12-3 标准的文件代号

在使用 DOS 的函数调用处理输入及输出时，只要使用文件代号，就可对文件及设备的处理方式一视同仁了。只要使用文件代号，DOS 会帮忙处理其他的细节，如此对于输出至屏幕、输出至打印机、输出至磁盘文件等操作，其处理方式都一样。

对于大部分的文件而言，当打开它时 DOS 都会返回一个文件代号，但对于常用的标准设备而言，DOS 已经缺省它们的文件代号，如表 12-1 所示。DOS 会自动打开及关闭这些文件代号，因为它们与 DOS 系统是一起的。

表 12-1 标准的文件代号

文件代号	用途	缺省设备
0000H	标准输入	CON:
0001H	标准输出	CON:
0002H	标准错误	CON:
0003H	标准辅助	AUX:
0004H	标准打印机	PRN:

表 12-1 中标准的文件代号第一个表示键盘，第二及第三个为屏幕。从键盘输入使用文件代号 0000H，要将数据显示在屏幕上使用 0001H，错误信息输出使用 0002H，这些标准输入及输出可以重新引导其他的设备或文件，但错误信息输出不能重新引导其他的设备或文件。标准辅助指第一台串行（serial）打印机，标准打印机指第一台并行（parallel）打印机。

12-4 建立一个文件代号

建立一个文件代号可以使用函数调用的 3CH 功能。
输入的参数如下：

- AH = 3CH
- CX = 文件属性
 - 0000H 正常情形，可读可写不受限制
 - 0001H 只读文件
 - 0002H 隐藏文件
 - 0004H 系统文件
 - 0008H 卷文件 (volume file)
 - 0010H 子目录 (subdirectory)
 - 0020H Archive 归档文件
- DS:DX = 文件路径地址 (ASCII 字符串地址)

输出的参数如下：

- CF = 0 建立成功
- AX = 文件代号
- CF = 1 建立失败
- AX = 2 找不到该文件



- = 3 找不到该路径
- = 4 已经打开太多文件
- = 5 不允许访问

建立结果,若进位标志 CF 值清除为 0 则表示建立成功,这时 AX 返回文件代号,若进位标志 CF 值设定为 1 则表示建立失败,这时 AX 返回错误代号。建立的程序片段如下:

```
filename DB "kdbfile.txt", 00H
handle DW 0
...
MOV CX, 0020H           ;Archive 归档文件
MOV DX, filename         ;文件路径地址
MOV AH, 3cH              ;建立一个文件代号功能
INT 21H                  ;调用函数
MOV [handle], AX         ;返回文件代号 handle
```

功能 3cH 建立一个文件并赋给文件代号,若指定的文件不存在,本功能会建立一个文件,若指定的文件已经存在,本功能会将文件数据长度更新为 0 值,接着使用 3cH 功能设定文件属性为 CX 所指定的,然后将该文件以输入输出模式打开。

12-5 打开一个文件

打开一个文件可以使用函数调用的 3DH 功能。

输入的参数如下:

AH = 3DH

AL = 00H 只读

= 01H 可写

= 02H 读写

DS:DX = 文件路径地址(ASCII字符串地址)

输出的参数如下:

CF = 0 打开成功

AX = 文件代号

CF = 1 打开失败

AX = 2 找不到该文件

= 3 找不到该路径

= 4 已经打开太多文件

= 5 不允许访问

= 12 访问错误

打开结果,若进位标志 CF 值清除为 0 则表示打开成功,这时 AX 返回文件代号,若进位标志 CF 值设定为 1 则表示打开失败,这时 AX 返回错误代号。打开的程序片段如下:

```
filename DB "kdbfile.txt", 00H
handle DW 0
...
MOV AL, 02H           ;打开 kdbfile.txt 为读写文件
MOV DX, filename
```

```

MOV AH, 3dH
INT 21H
JC error ;打开错误
MOV [handle], AX ;打开成功,文件代号为 handle

```

12-6 关闭一个文件

关闭一个文件可以使用函数调用的 3EH 功能。

输入的参数如下：

AH = 3EH

BX = 文件代号

输出的参数如下：

CF = 0 关闭成功

CF = 1 关闭失败

AX = 6 文件代号错误

关闭结果，若进位标志 CF 值清除为 0 则表示关闭成功，若进位标志 CF 值设定为 1 则表示关闭失败，这时 AX 返回错误代号。关闭的程序片段如下：

```

MOV BX, [handle]
MOV AH, 3eH
INT 21H
JC error

```

12-7 从文件或设备读取数据

从文件或设备读取数据可以使用函数调用的 3FH 功能。

输入的参数如下：

AH = 3FH

BX = 文件代号

CX = 读取的字节数

DS:DX = 缓冲区地址

输出的参数如下：

CF = 0 读取成功

AX = 实际读取的字节数

CF = 1 读取失败

AX = 5 不允许读取

6 文件代号错误

读取结果，若进位标志 CF 值清除为 0 则表示读取成功，这时 AX 为实际读取的字节数，若进位标志 CF 值设定为 1 则表示读取失败，这时 AX 返回错误代码。

从键盘读取数据的程序片段如下：

```

KEYBOARD EQU 0000H
buffer TIMES 80 DB ' '
lenbuf EQU $-buffer

```

```

byteread      DW      0
...
MOV           BX, KEYBOARD      ;读入 byteread 字节
MOV           CX, lenbuf
MOV           DX, buffer
MOV           AH, 3fH
INT           21H
JC            error
MOV           [byteread], AX

```

当从键盘读取数据时，DOS 只帮您读取一行，一行以<Enter>键结束，按下<Enter>键会产生两个字符码，就是返回字符（CR）及换行字符（LF），返回字符的 ASCII 码为 0dH，换行字符的 ASCII 码为 0cH。因此若输入：

```
abc<Enter>
```

事实上产生五个字符，即 a、b、c、0dH、0cH。输入结束时要按下<Ctrl+Z>，表示文件结束，<Ctrl+Z>表示<Ctrl> 键与<Z>键同时按下。

从磁盘文件 handle 读取数据的程序片段如下：

```

buffer      TIMES    80 DB  ' '
lenbuf      EQU      $-buffer
byteread    DW        0
handle      DW        0
...
MOV         BX, [handle]      ;从 handle 文件读入数据
MOV         CX, lenbuf
MOV         DX, buffer
MOV         AH, 3fH
INT         21H
JC          error            ;读入错误
CMP         AX, CX
JB          endOfFile        ;文件末端
MOV         [byteread],      AX ;读入数据字节数

```

12-8 数据写入文件或设备

数据写入文件或设备可以使用函数调用的 40H 功能。

输入的参数如下：

- AH = 40H
- BX = 文件代号
- CX = 写入的字节数
- DS:DX = 缓冲区地址

输出的参数如下：

- CF = 0 写入成功
- AX = 实际写入的字节数

CF = 1 写入失败

AX = 5 不允许写入

6 文件代号错误

写入结果,若进位标志 CF 值清除为 0 则表示写入成功,这时 AX 为实际写入的字节数,若进位标志 CF 值设定为 1 则表示写入失败,这时 AX 返回错误代码。

输出至屏幕的程序片段如下:

```
MOV BX, 0001H
MOV CX, lenbuf
MOV DX, buffer
MOV AH, 40H
INT 21H
JC error
```

写入磁盘文件 handle 的程序片段如下:

```
MOV BX, handle
MOV CX, lenbuf
MOV DX, buffer
MOV AH, 40H
JC error
CMP AX, CX
JNE error
```

程序 kbdfile.asm 重复从键盘输入一个字符串,并输出至 kbdfile.txt 文件,它的算法如下:

1. 建立一个相当于 kbdfile.txt 的文件代号 handle。
2. 从键盘输入一个字符串,其长度为 bytread。
3. 若输入的字符串为空字符串则跳至步骤 6。
4. 将输入的字符串写入 handle 文件。
5. 重复 2~4 步骤。
6. 关闭 handle 文件。

第 17~21 行建立一个相当于 kbdfile.txt 的文件代号 handle。第 23~30 行从键盘输入一个字符串,其长度为 bytread。第 31 行判断是否为文件尾,若为文件尾则跳至 endjob。第 33~37 行将输入的字符串写入 handle 文件。第 38 行重复输入、判断、写入的操作,直到输入文件尾 Ctrl-Z 为止。第 42~44 行关闭 handle 文件。

【程序 kbdfile.asm】

```
1 ; ***** kbdfile.asm *****
2             ORG 0100H
3             JMP start
4 KEYBOARD EQU 0000H
5 buffer      TIMES 80 DB ' '
6 lenbuf      EQU $-buffer
7 bytread     DW 0
8 filename    DB "kbdfile.txt", 00H
9 handle      DW 0
```



```
10 msg          DB      '请输入一个字符串: ', '$'
11 errmsg       DB      '从键盘输入错误!!', '$'
12 ;
13 %include "..\mymacro\dispstr.mac"
14 %include "..\mymacro\newline.mac"
15 ;
16 start:
17             MOV      CX, 0020H          ;Archive 文件
18             MOV      DX, filename       ;文件路径
19             MOV      AH, 3cH           ;建立一个文件代号功能
20             INT      21H               ;建立一个文件代号
21             MOV      [handle], AX      ;返回文件代号 handle
22 keyin:
23             dispstr    msg             ;显示 msg 输入信息
24             MOV      BX, KEYBOARD      ;键盘文件代号 KEYBOARD
25             MOV      CX, lenbuf        ;缓冲区长度
26             MOV      DX, buffer       ;缓冲区地址
27             MOV      AH, 3fH          ;键盘读入功能
28             INT      21H              ;键盘读入
29             JC       kbdErr           ;错误时
30             MOV      [byteread], AX   ;实际读入长度
31             CMP      WORD [byteread], 0 ;文件尾吗?
32             JE       endjob           ;是
33             MOV      BX, [handle]     ;文件代号
34             MOV      CX, [byteread]   ;数据长度
35             MOV      DX, buffer       ;缓冲区地址
36             MOV      AH, 40H          ;写入文件功能
37             INT      21H              ;写入文件
38             JMP      keyin            ;继续从键盘输入
39 kbdErr:
40             dispstr    errmsg          ;显示键盘错误信息
41 endjob:
42             MOV      BX, [handle]     ;文件代号
43             MOV      AH, 3eH          ;关闭文件功能
44             INT      21H              ;关闭文件
45             RET
```

【运行】

```
howasm\ch12> nasmw kbdfile.asm -o kbdfile.com <Enter>
howasm\ch12> kbdfile <Enter>
请输入一个字符串: aaa <Enter>
请输入一个字符串: bbb <Enter>
请输入一个字符串: ccc <Enter>
请输入一个字符串: <Ctrl+Z> <Enter>
```

【文件 kbdfile.txt】

aaa

bbb

ccc

程序 dumpfile.asm 将指定的文件以十六进制复制 (dump) 其内容。文件路径及名称从键盘输入。第 19~20 行使用 readzstr 宏从键盘输入一个 ASCIIZ 字符串的文件名存入 filename 变量。第 23~27 行赋予 filename 一个文件代号 handle。第 30~35 行从 handle 读入 byteread 个字节, 本例题正常情况下 byteread 的值为 1。第 36~37 行判断是否为文件尾, 若是则跳至 endjob。第 39~40 行以十六进制显示读入字符后再空一格。第 41~48 行判断是否已经输出 16 个字符, 若是则换行。第 49 行继续读下一个字符。第 51~53 行关闭 handle 文件。

执行 dumpfile 程序时输入的文件名为 kbdfile.txt, 就是我们执行 kbdfile 从键盘输入字符串所建立的文本文件, aaa<Enter> 在 kbdfile.txt 文件中表示为 6161610D0A 五个字节。bbb<Enter> 在 kbdfile.txt 文件表示为“6262620D0A”五个字节。ccc<Enter> 在 kbdfile.txt 文件中表示为“6363630D0A”五个字节。

【程序 dumpfile.asm】

```
1 ; ***** dumpfile.asm *****
```

```
2     ORG 0100H
```

```
3     JMP start
```

```
4 char DB      ' '
```

```
5 count DW 0
```

```
6 byteread DW 0
```

```
7 RW     EQU 02H
```

```
8 filename TIMES 36 DB ' '
```

```
9 handle DW 0
```

```
10 space DB      ' ', '$'
```

```
11 msg DB      '请输入路径及文件名: ', '$'
```

```
12 ;
```

```
13 %include "..\mymacro\readzstr.mac"
```

```
14 %include "..\mymacro\dispstr.mac"
```

```
15 %include "..\mymacro\newline.mac"
```

```
16 %include "..\mymacro\showbyte.mac"
```

```
17 ;
```

```
18 start:
```

```
19     dispstr      msg
```

;显示 msg 信息

```
20     readzstr      filename
```

;从键盘输入文件名 ASCIIZ 字符串

```
21     newline
```

;换行

```
22 ;
```

```
23     MOV     AL, RW
```

;打开读写文件

```
24     MOV     DX, filename
```

;输入的文件名

```
25     MOV     AH, 3dH
```

;打开文件功能

```
26     INT     21H
```

;打开文件



```
27      MOV      [handle], AX      ;返回文件代号
28 ;
29 readbyte:
30      MOV      BX, [handle]      ;从 handle 文件读入一个字符
31      MOV      CX, 1             ;一个字符
32      MOV      DX, char          ;字符存放地址
33      MOV      AH, 3fH           ;从 handle 文件读入功能
34      INT      21H               ;从 handle 文件读入
35      MOV      [byteread], AX     ;字符个数
36      CMP      WORD [byteread], 0 ;文件尾吗?
37      JE       endjob            ;是
38 ;
39      showbyte  char              ;以十六进制显示读入字符
40      dispstr   space            ;空白
41      INC      WORD [count]       ;count 值=count 值+1
42      MOV      AX, WORD [count]   ;AX=count 值
43      MOV      BL, 16             ;BL=16
44      DIV      BL                 ;AX/16
45      CMP      AH, 0              ;余数 0?
46      JNE      next              ;否
47      newline   ;换行
48 next:
49      JMP      readbyte           ;继续读下一个字符
50 endjob:
51      MOV      BX, [handle]       ;关闭输入文件
52      MOV      AH, 3eH
53      INT      21H
54      RET
```

【运行】

```
howasm\ch12> nasmw dumpfile.asm -o dumpfile.com <Enter>
```

```
howasm\ch12> dumpfile <Enter>
```

```
请输入路径及文件名: kbdfile.txt <Enter>
```

```
61 61 61 0D 0A 62 62 62 0D 0A 63 63 63 0D 0A
```

程序 `dispfile.asm` 从文件 `kbdfile.txt` 每次读入若干个字符,一直到字符值为 `0D` 及 `0A` 为止,换句话说即每次读入一行 (line),而行的结束符号为 `0D` 及 `0A`,即 `CR/LF`。输入使用 `21H` 的 `3fH` 功能,若文件已经结束,则读入的字符数为 `0`。

第 15~19 行打开 `kbdfile.txt`,作为输入文件,其文件代号为 `handle`。第 22~27 行从 `handle` 文件读入一行数据至缓冲区 `buffer`。第 28 行测试是否为文件尾。第 31~34 行将行尾附上 '\$' 号,然后将它显示在屏幕上。第 36 行读下一笔数据,重复读入、测试文件尾、显示读入数据的操作。第 38~40 行关闭 `handle` 文件。

【程序 dispfile.asm】

【程序】

```

1 ; ***** dispfile.asm *****
2     ORG 0100H
3     JMP start
4 buffer    TIMES    255 DB ' '
5 lenbuf    EQU $-buffer
6 byteread  DW    0
7 RW        EQU 02H
8 filename  DB    "kbdfile.txt", 00H
9 handle    DW    0
10 ;
11 %include "..\mymacro\dispstr.mac"
12 %include "..\mymacro\newline.mac"
13 ;
14 start:
15     MOV AL, RW                ;打开读写文件
16     MOV DX, filename          ;输入的文件名
17     MOV AH, 3dH               ;打开文件功能
18     INT 21H                   ;打开文件
19     MOV [handle], AX          ;返回文件代号
20 ;
21 readline:
22     MOV BX, [handle]           ;从 handle 文件读入数据
23     MOV CX, lenbuf             ;lenbuf 个字符
24     MOV DX, buffer             ;缓冲区地址
25     MOV AH, 3fH               ;从 handle 文件读入功能
26     INT 21H                   ;从 handle 文件读入
27     MOV [byteread], AX         ;读入字符个数
28     CMP WORD [byteread], 0     ;文件尾吗?
29     JE      endjob            ;是
30 ;
31     MOV BX, [byteread]         ;BX=读入字符个数
32     MOV BYTE [buffer+BX], '$'  ;附加一个字符串结束符号
33     dispstr buffer             ;显示读入数据
34     newline                    ;换行
35 ;
36     JMP readline              ;读下一笔数据
37 endjob:
38     MOV BX, [handle]           ;关闭输入文件
39     MOV AH, 3eH
40     INT 21H
41     RET

```



【运行】

```
howasm\ch12> nasmw dispfile.asm -o dispfile.com <Enter>
howasm\ch12> dispfile <Enter>
aaa
bbb
ccc
```

程序 copyfile.asm 从输入文件 kbdfile.txt 读入数据并存入内存的缓冲区，再从内存的缓冲区写入 diskfile.txt 输出文件。第 13~17 行打开输入文件 kbdfile.txt，其文件代号为 inhand。第 19~23 行打开输出文件 diskfile.txt，其文件代号为 outhand。第 25~30 行从 inhand 文件读入一行数据至缓冲区 buffer。第 31 行测试文件尾。第 34~38 行将数据从缓冲区写入 outhand 文件。第 40 行重复读入、测试文件尾、显示读入数据的操作。第 42~47 行是在文件尾时关闭 inhand 及 outhand 文件。

【程序 copyfile.asm】

```
1 ; ***** copyfile.asm *****
2     ORG 0100H
3     JMP start
4 buffer    TIMES    80 DB
5 lenbuf    EQU $-buffer
6 byteread  DW    0
7 RW        EQU 02H
8 infile    DB      "kbdfile.txt", 00H
9 inhand    DW    0
10 outfile  DB      "diskfile.txt", 00H
11 outhand  DW    0
12 start:
13     MOV AL, RW                ;打开读写文件
14     MOV DX, infile            ;输入的文件名
15     MOV AH, 3dH               ;打开文件功能
16     INT 21H                  ;打开文件
17     MOV [inhand], AX          ;返回文件代号
18 ;
19     MOV CX, 0020H             ;Archive 文件
20     MOV DX, outfile           ;文件路径
21     MOV AH, 3cH               ;建立一个文件代号功能
22     INT 21H                  ;建立一个文件代号
23     MOV [outhand], AX         ;返回文件代号
24 readline:
25     MOV BX, [inhand]          ;从 inhand 文件读入
26     MOV CX, lenbuf            ;lenbuf 个字符
27     MOV DX, buffer            ;缓冲区地址
28     MOV AH, 3fH               ;从 inhand 文件读入功能
```

```

29      INT 21H                ;从 handle 文件读入
30      MOV [byteread], AX     ;读入字符个数
31      CMP WORD [byteread], 0 ;文件尾吗?
32      JE      endjob        ;是
33 ;
34      MOV BX, [outhand]      ;否,写入 outhand 输出文件
35      MOV CX, [byteread]    ;字符个数
36      MOV DX, buffer        ;缓冲区地址
37      MOV AH, 40H           ;写入 outhand 输出文件功能
38      INT 21H               ;写入 outhand 输出文件
39 ;
40      JMP readline           ;读取下一笔数据
41 endjob:
42      MOV BX, [inhand]      ;关闭输入文件
43      MOV AH, 3eH
44      INT 21H
45      MOV BX, [outhand]    ;关闭输出文件
46      MOV AH, 3eH
47      INT 21H
48      RET

```

【运行】

```
howasm\ch12> nasmw copyfile.asm -o copyfile.com <Enter>
```

```
howasm\ch12> copyfile <Enter>
```

【文件 diskfile.txt】

```

aaa
bbb
ccc

```

12-9 移动文件指针

对于每一个打开过的文件，都备有一个文件指针（file pointer）。它指到下一个要处理的字节处。当一个程序从文件读取或写入文件时，DOS 会检查指针的位置以决定从何处开始操作。

当一个文件被打开后，DOS 将文件指针定位在文件的开始处，也就是第 0 个字节的位置，位置从 0 计数。每次读写之后 DOS 会更新文件指针的位置。例如文件指针在第 0 个字节的位置，读入 100 个字节后，文件指针被更新至第 100 个字节的位置，接着写入 200 个字节后，文件指针被更新至第 300 个字节的位置。

可以使用 42H 的功能将文件指针移到指定的位置。

输入的参数如下：

AH = 42H

AL = 起始位置

0 文件开始处

1 当前文件指针位置

2 文件结束处(文件尾)

BX = 文件代号

CX:DX = 移动的字节数

输出的参数如下:

CF = 0 移动成功

DX:AX = 移动后的文件指针位置

CF = 1 移动失败

AX = 1 错误的功能

6 文件代号错误

移动结果,若进位标志 CF 值清除为 0 则表示移动成功,这时 DX:AX 为移动后的文件指针位置,若进位标志 CF 值设定为 1 则表示移动失败,这时 AX 返回错误代码。

输入参数 CX:DX 构成一个 32 位的整数,输出参数 DX:AX 也是一样。长整数值范围为 0 至 4294967295。若文件长度在 65535 个字节之内,可以将 CX 设为 0 值,直接使用 DX 表示就可以了。当然移动后的位置 DX 也会设为 0 值的,单纯使用 AX 就可以。

要移到文件开始处可设 AL=0 及 CX:DX=0,要移到文件结束处(文件尾)可设 AL=2 及 CX:DX=0,要计算文件长度可将文件指针移到结束处,然后取得 DX:AX 的值。

下面例子将文件指针往前移动 100 个字节。

```
MOV AL, 1
```

```
MOV BX, [handle]
```

```
MOV CX, 0
```

```
MOV DX, 100
```

```
MOV AH, 42H
```

```
INT 21H
```

```
JC error
```

下面例子将文件指针移到文件开始处。

```
MOV AL, 0
```

```
MOV BX, [handle]
```

```
MOV CX, 0
```

```
MOV DX, 0
```

```
MOV AH, 42H
```

```
INT 21H
```

```
JC error
```

下面例子将文件指针移到文件结束处,并将文件长度拷贝至 lenfile 变量。

```
MOV AL, 2
```

```
MOV BX, [handle]
```

```
MOV CX, 0
```

```
MOV DX, 0
```

```
MOV AH, 42H
```

```
INT 21H
```

```
JC error
```

```
MOV WORD [lenfile], AX
```

12-10 检查并修改文件属性

每一个磁盘文件在磁盘的目录中都有一个入口（entry），这个入口存储这个文件的相关数据：

- 1. 文件名及延伸文件名。
- 2. 最后修改的日期及时间。
- 3. 在磁盘上这个文件开始的位置。
- 4. 文件长度。
- 5. 文件属性。

文件属性描述是否为隐藏文件、只读文件、系统文件或属于存档文件（archived）。一个文件可以具备多种属性。

检查并修改文件属性，可以使用 43H 的功能。

输入的参数如下：

- AH = 43H
- AL = 检查修改
 - 0 检查
 - 1 修改
- CX = 新的属性值
- DS:DX = 路径文件名地址 (ASCII 字符串)

输出的参数如下：

- CF = 0 检查或修改成功
- CX = 目前属性
- CF = 1 检查或修改失败
- AX = 1 功能错误
 - 2 文件找不到
 - 3 路径找不到
 - 5 不允许访问

检查或修改结果，若进位标志 CF 值清除为 0 则表示检查或修改成功，这时 CX 为检查或修改后的文件属性（见表 12-2），若进位标志 CF 值设定为 1 则表示检查或修改失败，这时 AX 返回错误代码。

表 12-2 CX 属性位的意义

位编号	值	意义
0	0	只读文件
	1	非只读文件
1	0	隐藏文件
	1	非隐藏文件
5	0	存档文件
	1	非存档文件



下面例子将文件属性改为只读、隐藏、非存档:

```
MOV AL, 1
MOV CX, 0000000000100011B
MOV DX, pathfile
MOV AH, 43H
INT 21H
JC error
```

下面例子检查文件属性是否为隐藏文件:

```
MOV AL, 0
MOV DX, pathfile
MOV AH, 43H
INT 21H
JC error
TEST CX, 0000000000000010B
JZ notHidden
```

12-11 建立新文件

建立新文件可以使用 5BH 的功能。

输入的参数如下:

AH = 5BH
CX = 新文件的属性值
DS:DX = 路径文件名地址 (ASCII 字符串)

输出的参数如下:

CF = 0 建立新文件成功
AX = 文件代号
CF = 1 建立新文件失败
AX = 2 文件找不到
3 路径找不到
4 已经打开太多文件
5 不允许访问
80 文件已经存在

建立新文件结果, 若进位标志 CF 值清除为 0 则表示建立新文件成功, 这时 AX 为建立新文件后的文件代号, 若进位标志 CF 值设定为 1 则表示建立新文件失败, 这时 AX 返回错误代码。若文件不存在则建立一个新文件并返回文件代号, 但若文件已经存在则返回错误代号。

下面例子建立一个新文件:

```
MOV CX, 0000000000100000B ;not archived file
MOV DX, pathfile
MOV AH, 5BH
INT 21H
JC error
MOV WORD [handle], AX ;file pointer
```

12-12 删除文件

删除文件可以使用 41H 的功能。

输入的参数如下：

AH = 41H

DS:DX = 路径文件名地址 (ASCII 字符串)

输出的参数如下：

CF = 0 删除文件成功

CF = 1 删除文件失败

AX = 2 文件找不到

3 路径找不到

5 不允许访问

删除文件结果，若进位标志 CF 值清除为 0 则表示删除文件成功，若进位标志 CF 值设定为 1 则表示删除文件失败，这时 AX 返回错误代码。对于非只读文件则执行删除操作，对于只读文件则将文件属性值改为 0 值。

下面例子是删除文件：

```
MOV DX, pathfile
```

```
MOV AH, 41H
```

```
INT 21H
```

```
JC error
```

12-13 文件改名

文件改名可以使用 56H 的功能。

输入的参数如下：

AH = 56H

DS:DX = 路径文件名地址 (ASCII 字符串)

ES:DI = 新路径文件名地址 (ASCII 字符串)

输出的参数如下：

CF = 0 文件改名成功

CF = 1 文件改名失败

AX = 2 文件找不到

3 路径找不到

5 不允许访问

17 不是同一个设备

文件改名的结果，若进位标志 CF 值清除为 0 则表示文件改名成功，若进位标志 CF 值设定为 1 则表示文件改名失败，这时 AX 返回错误代码。不能使用这个功能于一个已经打开的文件、隐藏文件、系统文件或一个子目录，否则将导致系统死机。

下面例子为文件改名：

```
MOV DX, oldPathfile
```

```
MOV DI, newPathfile
```



```
MOV AH, 56H
INT 21H
JC error
```

12-14 建立或删除子目录

建立或删除子目录，建立子目录可以使用 39H 的功能，删除子目录可以使用 3AH 的功能。

● 建立子目录

输入的参数如下：

```
AH = 39H
DS:DX = 路径地址(ASCII字符串)
```

输出的参数如下：

```
CF = 0 建立子目录成功
CF = 1 建立子目录失败
AX = 2 文件找不到
      3 路径找不到
      5 不允许访问
```

建立子目录结果，若进位标志 CF 值清除为 0 则表示建立子目录成功，若进位标志 CF 值设定为 1 则表示建立子目录失败，这时 AX 返回错误代码。

下面例子建立子目录：

```
MOV DX, pathname
MOV AH, 39H
INT 21H
JC error
```

● 删除子目录

输入的参数如下：

```
AH = 3AH
DS:DX = 路径地址(ASCII字符串)
```

输出的参数如下：

```
CF = 0 删除子目录成功
CF = 1 删除子目录失败
AX = 2 文件找不到
      3 路径找不到
      5 不允许访问
      16 正在使用中的子目录
```

删除子目录结果，若进位标志 CF 值清除为 0 则表示删除子目录成功，若进位标志 CF 值设定为 1 则表示删除子目录失败，这时 AX 返回错误代码。

下面例子删除子目录：

```
MOV DX, pathname
```

```
MOV AH, 3AH
INT 21H
JC error
```

12-15 取得当前目录

取得当前目录可以使用 47H 的功能。

输入的参数如下：

```
AH = 47H
DL = 磁盘驱动器号码
    0 缺省磁盘驱动器
    1 A:
    2 B:
    3 C:
    4 D:
...
```

DS:SI = 当前目录路径地址 (64 字节缓冲区)

输出的参数如下：

```
CF = 0 取得当前目录成功
DS:SI = 当前目录路径地址 (ASCII 缓冲区)
CF = 1 取得当前目录失败
AX = 15 磁盘驱动器号码错误
```

取得当前目录结果，若进位标志 CF 值清除为 0，则表示取得当前目录成功，若进位标志 CF 值设定为 1，则表示取得当前目录失败，这时 AX 返回错误代码。

下面例子取得当前目录：

```
MOV DL, drivenum
MOV SI, pathname
MOV AH, 47H
INT 21H
JC error
```

12-16 改变当前目录

改变当前目录可以使用 3BH 的功能。

输入的参数如下：

```
AH = 3BH
DS:DX = 新目录路径地址 (ASCII 缓冲区)
```

输出的参数如下：

```
CF = 0 改变当前目录成功
CF = 1 改变当前目录失败
AX = 2 文件找不到
    3 路径找不到
```



改变当前目录结果，若进位标志 CF 值清除为 0，则表示改变当前目录成功，若进位标志 CF 值设定为 1，则表示改变当前目录失败，这时 AX 返回错误代码。

下面例子改变当前目录：

```
MOV DX, pathname
MOV AH, 3BH
INT 21H
JC error
```

12-17 取得缺省的磁盘驱动器

取得缺省的磁盘驱动器可以使用 19H 的功能。
输入的参数如下：

AH = 19H

输出的参数如下：

AL = 磁盘驱动器号码

0 A:

1 B:

2 C:

3 D:

...

下面例子取得缺省磁盘驱动器：

```
MOV AH, 19H
INT 21H
```

12-18 改变缺省的磁盘驱动器

改变缺省的磁盘驱动器可以使用 0EH 的功能。
输入的参数如下：

AH = 0EH

DL = 磁盘驱动器号码

0 A:

1 B:

2 C:

3 D:

...

输出的参数如下：

AL = 磁盘驱动器总数

下面例子改变缺省磁盘驱动器：

```
MOV DL, 2 ;改变到 C:
MOV AH, 0EH
INT 21H
```

12-19 低级输入及输出

低级输入及输出直接对设备做输入及输出的操作。这是最底层的输入及输出，万不得已时才使用。最底层的输入及输出直接对输出输入端口（port）做读写的操作。每一个端口均有一个独一无二的 16 位编号，从 0000H 到 FFFFH。有些设备使用若干个端口号，每个端口号用途不同。

从一个端口号读取数据可使用 IN 指令，写入一个端口号可使用 OUT 指令，其格式如下：

IN 寄存器，端口号
OUT 端口号，寄存器
寄存器必须是 AX 或 AL。

这两个指令允许直接对设备输入或输出，每次一个字或字节。请注意必须使用 AX 或 AL。

下面例子直接从 61H 端口号输入一个字节至 AL 寄存器：

```
IN AL, 61H
```

下面例子直接输出一个存于 AL 寄存器的字节至 43H 端口号：

```
OUT 43H, AL
```

程序 beep.asm 直接控制喇叭发出“哔”声，直到按<Enter>键时才停止。第 6~8 行显示 msg 字符串信息。第 10 行从 61H 端口号输入一个字节至 AL，如图 12-1 所示为芯片 8255 的 61H 端口号，第 0 位为芯片 8523 的开关位，其值为 1 时表示发声由 8523 芯片控制，为 0 时表示发声不由 8523 芯片控制。第 1 位为喇叭的开关位，其值为 1 时表示喇叭可发声，为 0 时表示喇叭不能发声。

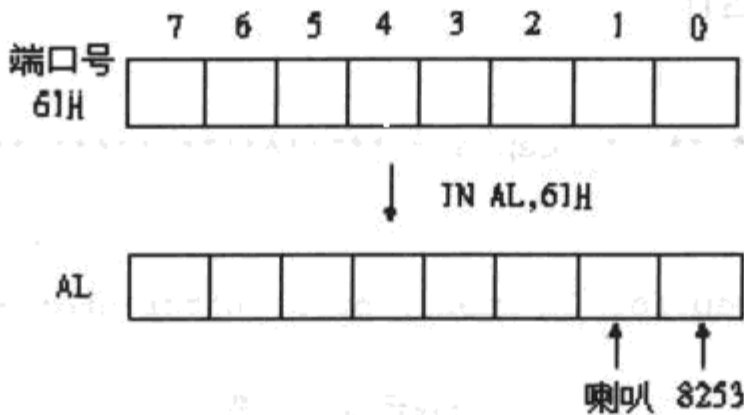


图 12-1 IN AL,61H 执行情形

第 11 行将位 0 及位 1 清除为 0 值，如图 12-2 所示。表示发声不由 8523 芯片控制，喇叭是在无声的状态。

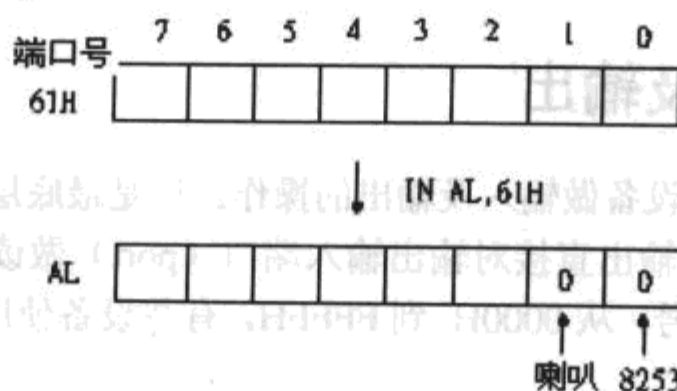


图 12-2 AL 低二位清除为 0 值

第 12~26 行构成一个循环。第 13 行每次循环均以 XOR 指令改变 AL 的值, 位 1 值逐次在 0 与 1 间切换, 位 1 为 0 值时表示喇叭不发声, 位 1 为 1 值时表示喇叭发声, 因此一个循环喇叭发声, 另一个循环喇叭不发声。

	11111100	
XOR	00000010	第1次循环时经XOR后的AL值
	11111110	
XOR	00000010	第2次循环时经XOR后的AL值
	11111100	
XOR	00000010	第3次循环时经XOR后的AL值
	11111110	
XOR	00000010	第4次循环时经XOR后的AL值
	11111100	

第 14 行将 XOR 后的 AL 值暂存入堆栈顶端，因为后面的指令会用到 AL 寄存器，怕数据被破坏，因此暂存起来。第 15 行将 AL 值存入 61H 端口号，根据 AL 的第一位值是 1 或 0 决定发声或不发声。第 16~19 行延迟 16 次。第 21~23 行测试键盘状态，是否有按键操作。第 25 行从堆栈顶端取回刚刚存储的 AL 值。第 26 行检查键盘状态，若没按键则重复 beep 循环。若有按键则程序停止运行。

【程序 beep.asm】

```

1 ; ***** beep.asm *****
2     ORG 0100H
3     JMP start
4 msg DB      'beep for 61H port, press Enter stop : ','$'
5 start:
6     MOV DX, msg                ;显示 msg 信息
7     MOV AH, 09H
8     INT 21H
9     IN  AL, 61H                ;AL=从 61H 端口号输入一个字节
10    AND AL, 11111100B          ;将位 0 及位 1 清除为 0 值
11 beep:
12    XOR AL, 00000010B          ;位 1 值逐次在 0 与 1 间切换
13    PUSH AX                    ;将 AL 值保存在堆栈顶端
14    OUT 61H, AL                ;将 AL 值输出至 61H 端口号, beep 声
15    MOV CX, 0010H              ;CX=循环次数
16 loop2:

```

```

17      NOP                      ;没事
18      LOOP    loop2           ;循环
19      MOV AH, 0BH              ;取得键盘输入状态功能
20      INT 21H                  ;AL=取得键盘输入状态
21      CMP AL, 00H              ;没输入时 AL=00H
22      POP AX                   ;取回堆栈顶端刚保存的值
23      JE      beep             ;键盘没输入
24      RET

```

【运行】

```
howasm\ch12> nasmw beep.asm -o beep.com <Enter>
```

```
howasm\ch12> beep <Enter>
```

beep for 61H port, press Enter stop :

程序 beepfreq.asm 与 beep.asm 类似, 只不过每次循环时均改变延迟的时间而已, 延迟次数 CX 与频率成反比, CX 值愈小频率愈高, 每秒振动次数愈多, CX 值愈大频率愈低, 每秒振动次数愈少。

第 16 行设定 BX 值为 7FFFH, 将 BX 拷贝至 CX, 因此开始时频率较低, 每次循环 BX 值减 1, 愈来愈频率愈高, 高到 0 时又从 7FFFH 重新开始。

【程序 beepfreq.asm】

```

1 ; ***** beepfreq.asm *****
2 ; 逐渐变换声音频率, 改变 CX 值即可改变频率
3      ORG 0100H
4      JMP start
5      msg DB      'beep for 61H port, press Enter stop : ','$'
6 start:
7      MOV DX, msg                      ;显示 msg 信息
8      MOV AH, 09H
9      INT 21H
10     IN      AL, 61H                  ;AL=从 61H 端口号输入一个字节
11     AND AL, 11111100B                ;将位 0 及位 1 清除为 0 值
12 begin:
13     MOV BX, 07FFFH
14 beep:
15     XOR AL, 00000010B                ;位 1 值逐次在 0 与 1 间切换
16     PUSH AX                          ;将 AL 值保存在堆栈顶端
17     OUT 61H, AL                      ;将 AL 值输出至 61H 端口号, beep 声
18     MOV CX, BX                      ;CX=循环次数
19 loop2:
20     NOP                              ;没事
21     LOOP    loop2                   ;循环
22     MOV AH, 0BH                      ;取得键盘输入状态功能
23     INT 21H                          ;AL=取得键盘输入状态
24     CMP AL, 0FFH                    ;有输入时 AL=0FFH
25     JE      endjob                  ;按 Enter 键结束

```



```
26      DEC BX          ;BX=BX-1
27      JZ      begin    ;BX=0 时从新设定
28      POP AX          ;取回堆栈顶端刚保存的值
29      JMP     beep     ;键盘没输入时重复 beep 声
30 endjob:
31      RET
```

【运行】

```
howasm\ch12> nasmw beepfreq.asm -o beepfreq.com <Enter>
```

```
howasm\ch12> beepfreq <Enter>
```

beep for 61H port, press Enter stop :

课后习题

1. 重复从键盘输入五位数字帐号及六位数存款，存款数前面要补零，帐号及存款之间以逗号隔开，使用 `readstr` 宏从键盘输入，每一笔数据输入完毕按<Enter>键，直接按<Enter>键表示输入已经结束，每一笔记录均存入磁盘文件 `account.txt`，输入结束时要关闭文件。使用以下数据进行测试。

```
10001,001100<Enter>
10003,003300<Enter>
10002,002200<Enter>
10005,005500<Enter>
10004,004400<Enter>
<Enter>
```

2. 将 `account.txt` 每一笔记录读入内存后，显示在屏幕上，累积所有存款金额后将总金额显示出来。
3. 因为 `account.txt` 文件的每一笔记录的长度都是固定的（连同 CR/LF 共 14 字节），因此可以使用随机的方式读取指定记录编号的记录。记录从零开始编号，帐号 10001 占第 0 个记录，帐号 10003 占第 1 个记录，帐号 10004 占第 4 个记录，五个记录编号范围为 0 至 4。

请从键盘输入一个记录编号，然后从 `account.txt` 读入该笔记录，显示在屏幕上，输入 -1 时结束程序的执行。

4. 利用文件指针移到 `account.txt` 文件尾，计算总共有几笔记录。
5. 从文件 `account.txt` 输入每一笔记录至内存后，依帐号由小到大排序后，输出至 `accounts.txt` 文件。排序后其记录如下：

```
10001,001100
10002,002200
10003,003300
10004,004400
10005,005500
```

6. 请从键盘输入一个帐号，然后从 `accounts.txt` 读入该笔记录，显示在屏幕上，输入 -1 时结束程序的运行。输入 10003 帐号测试读取是否成功。

13

数据结构

数组 (array) 在数据结构 (data structure) 中占很重要的地位。数组是由一组具有相同数据类型的元素所组成的。利用循环并配合数组的下标 (subscript), 可轻易地处理大量的数据。数组属于静态结构, 链表 (list)、队列 (queue)、堆栈 (stack) 等结构属于动态的。

在汇编语言里并没有提供数组及结构的机制, 不过我们可以试着来建立这种结构。



13-1 数组声明

数组是计算机里占用一块连续的内存，用于存储相同类型的若干个数据，每一个数据均使用同一个名称，这个共同的名称就是这个数组的名称。

要参考数组中某一个位置或元素，就以数组的名称及位置索引值表示。

```
okstr DB 'O', 'K', '$'
```

上例中 okstr 变量在内存里每一个字符分配一个字节，共有三个字符，因此占 3 个字节。第 0 个字符'O' 的内存偏移地址为 okstr+0，第 1 个字符'K' 的内存偏移地址为 okstr+1，第 2 个字符'\$' 的内存偏移地址为 okstr+2。这些都属于同一个字符类型，以第 0 个元素为代表，其他的元素以偏移的索引值为代表，因此 okstr 称为字节数组（array），也称为表格（table）。+0、+1、+2 等所加的数值称为索引值（index），或称为下标（subscript）。如图 13-1 所示。

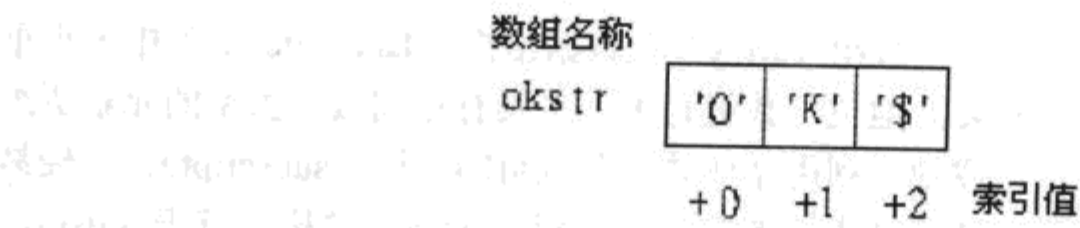


图 13-1 okstr 字节数组

- 第 0 个元素的偏移地址为 okstr+0，它的内容为 BYTE [okstr+0]。
- 第 1 个元素的偏移地址为 okstr+1，它的内容为 BYTE [okstr+1]。
- 第 2 个元素的偏移地址为 okstr+2，它的内容为 BYTE [okstr+2]。

因为很有规律，若将索引值存于索引寄存器变量 SI 中，那么偏移地址就可以表示为 okstr+SI，内容为 BYTE [okstr+SI]。SI 的值从 0 变化至 2，就可当循环变量了。例如：

```
MOV CX, 3
MOV SI, 0
loop2:
    MOV DL, BYTE [okstr+SI]
    MOV AH, 02H
    INT 21H
    INC SI
    LOOP loop2
```

程序 okstr.asm 中将每一个数组元素值逐一打印，三个元素可以这样做，30 个元素就要使用循环了。数组配合循环是天经地义的事。

【程序 okstr.asm】

```
1 ; ***** okstr.asm *****
2     ORG 0100H
3     JMP start
4 okstr DB      'O', 'K', '$'
5 start:
6     MOV DL, BYTE [okstr]
7     MOV AH, 02H
```

;okstr 的偏移地址的内容
;显示 DL 内含字符

```
8      INT 21H
9      MOV DL, BYTE [okstr+1]      ;okstr+1 的偏移地址的内容
10     MOV AH, 02H                  ;显示 DL 内含字符
11     INT 21H
12     MOV DL, BYTE [okstr+2]      ;okstr+2 的偏移地址的内容
13     MOV AH, 02H                  ;显示 DL 内含字符
14     INT 21H
15     RET
```

【运行】

```
howasm\ch13> nasmw okstr.asm -o okstr.com <Enter>
howasm\ch13> okstr <Enter>
OK$
```

程序 sum.asm 中声明一个字数组 ary 如下：

```
ary DW 18, 25, 13, 56, 44, 78, 99, 66
```

数组 ary 在内存里每一个元素分配一个字，共有 8 个元素，因此占 8 个字，也就是 16 个字节。第 0 个元素 18 的内存偏移地址为 ary+0，第 1 个元素 25 的内存偏移地址为 ary+2，因为一个字占两个字节，因此是加 2 而不是加 1。第 7 个元素 66 的内存偏移地址为 ary+14，如图 13-2 所示。

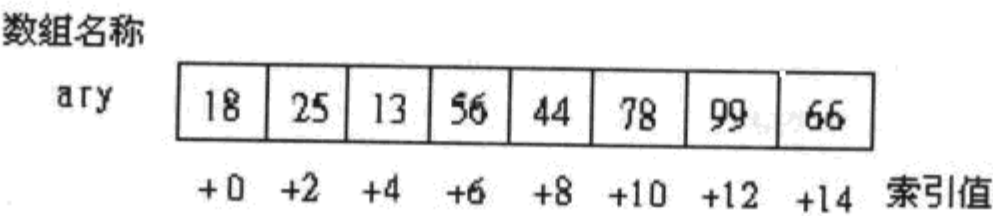


图 13-2 ary 字数组

【程序 sum.asm】

```
1 ; ***** sum.asm *****
2     ORG 0100H
3     JMP start
4 count DW 8
5 ary   DW 18, 25, 13, 56, 44, 78, 99, 66
6 msg   DB '八个整数的总和为', '$'
7 sum   DW 0
8 ;
9 %include "..\mymacro\dispi.mac"
10 %include "..\mymacro\dispstr.mac"
11 ;
12 start:
13     MOV AX, 0      ;AX=累加器初始值
14     MOV SI, 0      ;SI=数组元素索引初始值
15     MOV CX, [count] ;CX=数组元素个数
16 loop2:
```



```
17      ADD AX, WORD [ary+SI]      ;AX=AX+第 SI 元素值
18      ADD SI, 2                  ;下一个元素索引
19      LOOP    loop2              ;继续
20      MOV WORD [sum], AX          ;sum 为总和
21      dispstr msg                 ;显示 msg 信息
22      dispi   sum                 ;显示总和
23      RET
```

【运行】

```
howasm\ch13> nasmw sum.asm -o sum.com <Enter>
```

```
howasm\ch13> sum <Enter>
```

八个整数的总和为 399

程序 min.asm 在一个数组中找出最小值。第 5 行声明一个 8 个元素的字节整数数组 ary, 第 13 行设第 0 个元素为最小存储寄存器 AL, 然后逐一与数组中其他元素值比较, 若 AL 较大则与所比较元素值互换 (第 20 行), 最后存于 AL 者即为最小值。

【程序 min.asm】

```
1 ; ***** min.asm *****
2      ORG 0100H
3      JMP start
4 count DW 8
5 ary   DB      18, 25, 13, 56, 44, 78, 99, 66
6 msg   DB      '八个整数最小的是', '$'
7 min   DB      0
8 ;
9 %include "..\mymacro\dispb.mac"
10 %include "..\mymacro\dispstr.mac"
11 ;
12 start:
13      MOV AL, BYTE [ary]          ;假设第 0 个元素值为最小
14      MOV SI, 1                  ;SI=数组第 1 个元素索引
15      MOV CX, [count]            ;CX=数组元素个数
16      DEC CX
17 loop2:
18      CMP AL, BYTE [ary+SI]      ;AL:第 SI 元素值
19      JL     next                ;AL 较小
20      XCHG   AL, BYTE [ary+SI]   ;互换
21 next:
22      INC SI                      ;下一个元素索引
23      LOOP   loop2              ;继续
24      MOV BYTE [min], AL         ;min 为最小值
25      dispstr msg                 ;显示 msg 信息
26      dispb  min                 ;显示最小值
27      RET
```

【运行】

```
howasm\ch13> nasmw min.asm -o min.com <Enter>
```

```
howasm\ch13> min <Enter>
```

八个整数最小的是 13

13-2 数组查找

程序 monthtab.asm 中第 4~7 行建立一个数组, 包含 12 个元素, 每一个元素占 3 个字节, 数组名为 table, 第 0 个元素的偏移地址为 table+0, 第 1 个元素的偏移地址为 table+3, 第 n 个元素的偏移地址为 table+n*3, 自变量 lenkey 的值为 3, 表示数组 table 中每一个元素占 3 个字节。

程序从键盘输入月份数, 例如您输入 3 为字符, 将它转换为数值 3 并存入 monnum 变量中。它相对应的月份简称的偏移地址可计算如下:

$$\text{table} + (\text{monnum} - 1) * \text{lenkey}$$

即

$$\text{table} + (3 - 1) * 3$$

数组元素其偏移值为 table+6, 其内容为'Mar', 因此三月份的简称就是'Mar'。

【程序 monthtab.asm】

```
1 ; ***** monthtab.asm *****
2     ORG 0100H
3     JMP start
4 lenkey    DW 3
5 table DB   'Jan', 'Feb', 'Mar', 'Apr'
6         DB   'May', 'Jun', 'Jul', 'Aug'
7         DB   'Sep', 'Oct', 'Nov', 'Dec'
8 monnum DW 10
9 msg  DB   '请输入月份数(1-12) : ', '$'
10 monstr DB ' ', '$'
11 msg2 DB   '月份名称= '
12 monname DB ' ', '$'
13 ;
14 %include "..\mymacro\readstr.mac"
15 %include "..\mymacro\dispstr.mac"
16 %include "..\mymacro\newline.mac"
17 %include "..\mymacro\strncpy.mac"
18 %include "..\mymacro\strtoi.mac"
19 ;
20 start:
21     dispstr msg           ;显示 msg 信息
22     readstr monstr        ;键入月份数值
23     newline              ;换行
```

```

24      strtol monstr, '$', monnum      ;字符串转换数值 monnum
25 ;
26      MOV AX, [monnum]                ;AX=月份数
27      DEC AX                          ;从 0 算起
28      MUL WORD [lenkey]               ;AX=(月份数-1)*键长度
29      ADD AX, table                   ;AX=AX+table 地址
30      MOV BP, AX                      ;BP=AX
31 ;将找到的月份名称
32      stncpy monname, BP, lenkey      ;拷贝至 monname 处
33      dispstr msg2                    ;显示月份名称
34      RET
RET

```

【运行】

```

howasm\ch13> nasmw monthtab.asm -o monthtab.com <Enter>
howasm\ch13> monthtab <Enter>
      请输入月份数(1-12) : 3 <Enter>
      月份名称= Mar

```

程序 datetab.asm 中声明两个数组，一个是月份名称数组 montab，另一个是星期名称数组 weektab，做法与 monthtab.asm 程序同。程序中调用 21H 中断功能 AH=0aH，返回系统时间如下：

```

AL = 星期天数(星期日为 0 算起)
CX = 年数(1980-2099)
DH = 月份(01-12)
DL = 日数(01-31)

```

【程序 datetab.asm】

```

1 ; ***** datetab.asm *****
2 ;
3 ; 调用 21H 中断功能 AH=0aH 时返回下列值:
4 ;
5 ; AL = day of week (sunday=0) 星期天数(星期日为 0 算起)
6 ; CX = year (1980-2099) 年数
7 ; DH = month (01-12) 月份
8 ; DL = day (01-31) 日数
9 ;
10      ORG 0100H
11      JMP start
12 lenmon DW      3
13 montab  DB      'Jan', 'Feb', 'Mar', 'Apr'
14          DB      'May', 'Jun', 'Jul', 'Aug'
15          DB      'Sep', 'Oct', 'Nov', 'Dec'
16 month   DW      0
17 ;

```

```

18 lenweek DW 9
19 weektab DB 'Sunday ', 'Monday ', 'Tuesday ',
20 DB 'Wednesday', 'Thursday ', 'Friday '
21 DB 'Saturday '
22 week DW 0
23 year DW 0
24 day DW 0
25 ;
26 msg DB '现在的日期及时间= ', '$'
27 monthstr DB ' ', '$'
28 weekstr DB ' ', '$'
29 ;
30 %include "..\mymacro\dispstr.mac"
31 %include "..\mymacro\strncpy.mac"
32 %include "..\mymacro\dispi.mac"
33 ;
34 start:
35 MOV AH, 2aH ;设定取得日期功能
36 INT 21H ;取得日期
37 MOV BYTE [week], AL ;将星期数存入 week
38 MOV WORD [year], CX ;将年数存入 year
39 MOV BYTE [month], DH ;将月份数存入 month
40 MOV BYTE [day], DL ;将日数存入 day
41 ;
42 MOV AX, [month] ;AX=月份数
43 DEC AX ;从 0 算起
44 MOV BX, [lenmon] ;BX=月份表键长度
45 MUL BX ;AX=(月份数-1)*键长度
46 MOV BP, montab ;BP=montab 地址
47 ADD BP, AX ;BP=montab 地址+AX
48 ;将找到的月份名称
49 strncpy monthstr, BP, lenmon ;拷贝至 monthstr 处
50 ;
51 MOV AX, WORD [week] ;AX=星期数
52 DEC AX ;从 0 算起
53 MOV BX, WORD [lenweek] ;BX=星期表键长度
54 MUL BX ;AX=(星期数-1)*键长度
55 MOV BP, weektab ;BP=weektab 地址
56 ADD BP, AX ;BP=weektab 地址+AX
57 ;将找到的星期名称
58 strncpy weekstr, BP, lenweek ;拷贝至 weekstr 处
59 dispstr msg ;显示 msg 信息
60 dispstr weekstr ;星期名称

```



```
61      dispstr monthstr      ;月份名称
62      dispi   day            ;日数
63      dispi   year           ;年份
64      RET
```

【运行】

```
howasm\ch13> nasmw datetab.asm -o datetab.com <Enter>
howasm\ch13> datetab <Enter>
```

现在的日期及时间= Tuesday Nov 28 2001

程序 booktab.asm 中建立一个 table 数组, 每一个元素包含两列, 一列为书号, 另一列为书名, 每列都要固定长度, 其元素地址才可使用公式计算取得, 书号列长度 lenkey 为 5 个字符, 书名列长度 lendata 为 16 个字符, 因此每一个元素长度为 21 个字符。第 n 个元素 (从 0 算起) 的偏移地址为:

$table + n * 21$

【程序 booktab.asm】

```
1 ; ***** booktab.asm *****
2      ORG 0100H
3      JMP start
4 lenkey  DW  5
5 lendata DW  16
6 lentab  DW  5
7 table DB  '27020', '如何设计 JAVA 程序',
8          DB  '27009', 'JSP 程序设计',
9          DB  '29009', 'C/C++程序设计',
10         DB  '31033', '网页设计',
11         DB  '29099', '汇编程序设计'
12 ;
13 result  DW  0
14 msgerr  DB  '书号找不到!', '$'
15 msgdata DB  '书名= ',
16 data    DB  ' ', '$'
17 msgkey  DB  '请输入书号: ', '$'
18 key     DB  ' ', '$'
19 ;
20 %include "..\mymacro\readstr.mac"
21 %include "..\mymacro\dispstr.mac"
22 %include "..\mymacro\newline.mac"
23 %include "..\mymacro\strncpy.mac"
24 %include "..\mymacro\strncmp.mac"
25 ;
26 start:
27      dispstr msgkey      ;显示 msgkey 信息
```

```

28      readstr key                ;输入书号(5 位数字)
29      newline                    ;换行
30 ;
31      MOV CX, [lentab]           ;CX=表格总记录数
32      MOV BP, table              ;BP=table 地址
33 loop2:
34      strncmp key, BP, lenkey, result ;key 与记录键比较
35      CMP WORD [result], 0        ;是否相同?
36      JE      equal              ;是,跳至 equal
37      ADD BP, [lenkey]           ;BP=BP+键长度
38      ADD BP, [lendata]          ;BP=BP+数据长度
39      LOOP    loop2              ;继续循环
40 notfound:
41      dispstr msgerr              ;找不到,显示错误
42      JMP endjob                 ;结束
43 equal:
44      ADD BP, [lenkey]           ;BP=BP+键长度
45      strncpy data, BP, lendata   ;将数据拷贝至 data
46      dispstr msgdata            ;显示数据
47 endjob:
48      RET

```

【运行】

```

howasm\ch13> nasmw booktab.asm -o booktab.com <Enter>
howasm\ch13> booktab <Enter>
      请输入书号: 27020 <Enter>
      书名= 如何设计 JAVA 程序
howasm\ch13> booktab <Enter>
      请输入书号: 27999 <Enter>
      书号找不到!

```

13-3 使用 XLATB 指令转换

指令 XLATB 将指定的字节转换为预先定义好的字节。一个字节的值的范围为 0 至 255, 因此您要预先定义最多 256 个字节的数据表, 这个表的地址要存入 BX, 您要转换的字节要存入 AL, 而转换后的字节存回 AL, 覆盖原来的值。若您要转换的字节范围没那么大, 数据表可以不要那么大, 例如十六进制只需 16 个符号, 那么数据表只需 16 个符号就可以了。不过若您要转换的字节范围太大, 超过 256 个时, 指令 XLATB 就派不上用场了。

程序 hextab.asm 第 4 行声明一个 hextab 数据表, 含 16 个符号, 它的索引值从 0 至 15, hextab+0 的内容为'0', "hextab+1" 的内容为'1', "hextab+15" 的内容为'F'。您若将索引值 15 存入 AL 后, 执行 XLATB 指令, 返回来的 AL 值为'F', 也就是说 15 转换为'F'。



【程序 hextab.asm】

```
1 ; ***** hextab.asm *****
2     ORG 0100H
3     JMP start
4 hextab      DB      '0123456789ABCDEF'
5 numbyte     DB      0
6 msghex      DB      '相当于十六进制数= ', '$'
7 numchar     DB      ', '
8 msgin       DB      '请输入一个整数(0-15) : ', '$'
9 numstr      DB      ', '
10 ;
11 %include "..\mymacro\readstr.mac"
12 %include "..\mymacro\dispstr.mac"
13 %include "..\mymacro\newline.mac"
14 %include "..\mymacro\strtob.mac"
15 %include "..\mymacro\dispchr.mac"
16 ;
17 start:
18     dispstr msgin                ;显示 msgin 信息
19     readstr numstr               ;输入书号(0-15 数字)
20     newline                     ;换行
21     strtob numstr, '$', numbyte ;字符串转换为数值
22 ;
23     MOV BX, hextab              ;BX=十六进制数字表地址
24     MOV AL, BYTE [numbyte]      ;AL=十进制数值
25     AND AL, 00001111B          ;只取右边四位
26     XLATB                      ;AL=对应十六进制数字
27     MOV BYTE [numchar], AL      ;存入 numchar+1 地址处
28     dispstr msghex              ;显示 msghex 信息
29     dispchr numchar              ;显示对应十六进制数字
30     RET
```

【运行】

howasm\ch13> nasmw hextab.asm -o hextab.com <Enter>

howasm\ch13> hextab <Enter>

请输入一个整数(0-15) : 12 <Enter>

相当于十六进制数= C

13-4 排序

程序 max.asm 第 5 行声明一个数组 array，共有 8 个元素，每一个元素均为字，字变量 count 存储数组元素个数 8。现在要将 array 的元素值按照由小到大的顺序排列，排序的方法有很多，其中有一种冒泡法 (bubble sort)，将相邻两元素逐个比较，左边的元素值若较右边

的元素值大, 则左右两元素互换。

第0个与1个元素值比较, $18 < 25$, 不互换。

第1个与2个元素值比较, $25 > 13$, 互换。

18 13 25 56 44 78 99 66

第2个与3个元素值比较, $25 < 56$, 不互换。

第3个与4个元素值比较, $56 > 44$, 互换。

18 13 25 44 56 78 99 66

第4个与5个元素值比较, $56 < 78$, 不互换。

第5个与6个元素值比较, $78 < 99$, 不互换。

第6个与7个元素值比较, $99 > 66$, 互换。

18 13 25 44 56 78 66 99

最大值从左边一直往右边移动, 右边您可将它想象为水面, 气泡从深处往水面移动时, 因压力的关系, 气泡会愈来愈大。最右边的 99 为最大值。但其他的七个值并不依序排列, 不过对这七个值您若按照上次八个值的作法, 您会找出这七个值中最大者。如此一轮一轮地做七次, 就可排成顺序, 虽然看起来很笨, 不过计算机的速度太快了, 可说快能补拙了。

程序 max.asm 只做第一轮的比较, 因此只找出其中的最大者而已。第 14 行 SI 为左边元素的索引值, 第 15 行 DI 为右边元素的索引值, 第 19 行为左边元素值与右边元素值比较, 第 21~23 行为左边元素值较大时所做的互换操作。第 25~27 设定下一个左边及右边元素的索引值后继续比较。

【程序 max.asm】

```
1 ; ***** max.asm *****
2     ORG 0100H
3     JMP start
4 count DW 8
5 array DW 18, 25, 13, 56, 44, 78, 99, 66
6 msg    DB '数组中最大的元素值是', '$'
7 ;
8 %include "..\mymacro\dispi.mac"
9 %include "..\mymacro\dispstr.mac"
10 ;
11 start:
12     MOV CX, [count]           ;CX=数组元素个数
13     DEC CX                     ;从0计数
14     MOV SI, array             ;SI第0个元素偏移地址
15     MOV DI, array+2           ;DI第1个元素偏移地址
16 loop2:
17     MOV AX, WORD [SI]         ;AX=第SI个元素值
18     MOV DX, WORD [DI]         ;DX=第DI个元素值
19     CMP AX, DX                ;AX:DX
20     JL  less                  ;AX<DX
```



```
21      XCHG     AX, DX           ;AX 与 DX 互换
22      MOV WORD [SI], AX       ;第 SI 个元素值=AX
23      MOV WORD [DI], DX       ;第 DI 个元素值=DX
24 less:
25      ADD SI, 2                 ;SI=SI-2
26      ADD DI, 2                 ;DI=DI-2
27      LOOP     loop2           ;继续
28 ;
29      dispstr msg              ;显示 msg
30      dispi     SI              ;显示最大元素值
31      RET
```

【运行】

```
howasm\ch13> nasmw max.asm -o max.com <Enter>
```

```
howasm\ch13> max <Enter>
```

数组中最大的元素值是 99

程序 sortword.asm 是继续 max.asm 的排序, max.asm 只做第一轮的比较, 而 sortword 做完 count-1 轮的比较, 每一轮的比较都产生一个该轮的最大值, 但第一轮的最大值较第二轮的最大值还大, 第二轮的最大值较第三轮的最大值还大等, 最后完成冒泡法的排序。

【程序 sortword.asm】

```
1 ; ***** sortword.asm *****
2      ORG 0100H
3      JMP start
4 count DW 8
5 array DW 18, 25, 13, 56, 44, 78, 99, 66
6 ;
7 %include "..\mymacro\dispi.mac"
8 ;
9 start:
10     MOV BX, WORD [count]       ;BX=数组元素个数
11 begin:
12     MOV CX, [count]            ;CX=数组元素个数
13     DEC CX                     ;从 0 计数
14     MOV SI, array              ;SI 第 0 个元素偏移地址
15     MOV DI, array+2            ;DI 第 1 个元素偏移地址
16 loop2:
17     MOV AX, WORD [SI]          ;AX=第 SI 个元素值
18     MOV DX, WORD [DI]          ;DX=第 DI 个元素值
19     CMP AX, DX                 ;AX:DX
20     JL     less                ;AX<DX
21     XCHG     AX, DX            ;AX 与 DX 互换
22     MOV WORD [SI], AX          ;第 SI 个元素值=AX
23     MOV WORD [DI], DX          ;第 DI 个元素值=DX
```

```

24 less:
25     ADD SI, 2                      ;SI=SI-2
26     ADD DI, 2                      ;DI=DI-2
27     LOOP    loop2                  ;继续
28 ;
29     DEC BX                          ;BX=BX-1
30     CMP BX, 1                      ;BX=1?
31     JG      begin                  ;否
32 ;
33     MOV CX, WORD [count]           ;CX=数组元素个数
34     MOV BX, array                  ;BX=array 起始地址
35 loop3:
36     dispi  BX                      ;显示 BX 所指元素内容
37     ADD BX, 2                      ;BX 指向下一个元素
38     LOOP    loop3                  ;继续
39     RET

```

【运行】

```
howasm\ch13> nasmw sortword.asm -o sortword.com <Enter>
```

```
howasm\ch13> sortword <Enter>
```

```
13 18 25 44 56 66 78 99
```

程序 `sortstr.asm` 对于相等长度字符串的排序，用的还是相同的冒泡法，所不同的是在这里用字符串方式比较，但互换的并不是字符串本身，而是该字符串的地址。第 6 行声明一个字符串数组 `array`，每一个元素均为 7 个字节的字符串，共有 8 个元素，每一个元素的地址均存于 `strptr` 数组中，如图 13-3 所示。

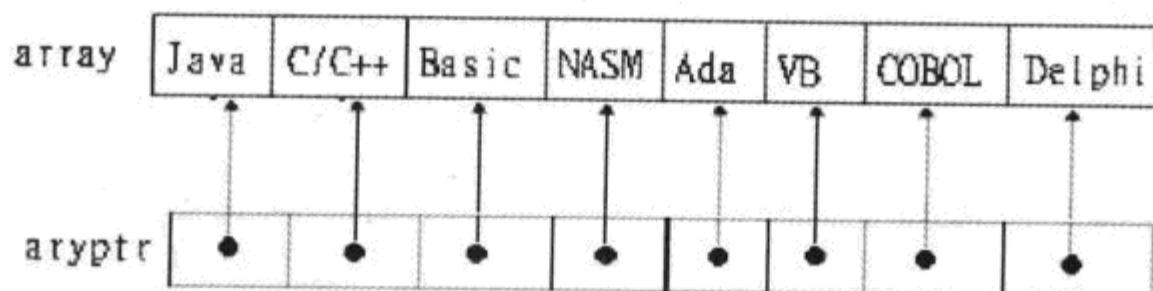


图 13-3 字符串数组与其地址指针

当第 0 个元素 `Java` 与第 1 个元素 `C/C++` 比较时，`Java` 较 `C/C++` 大，因此地址指针互换，如图 13-4 所示。

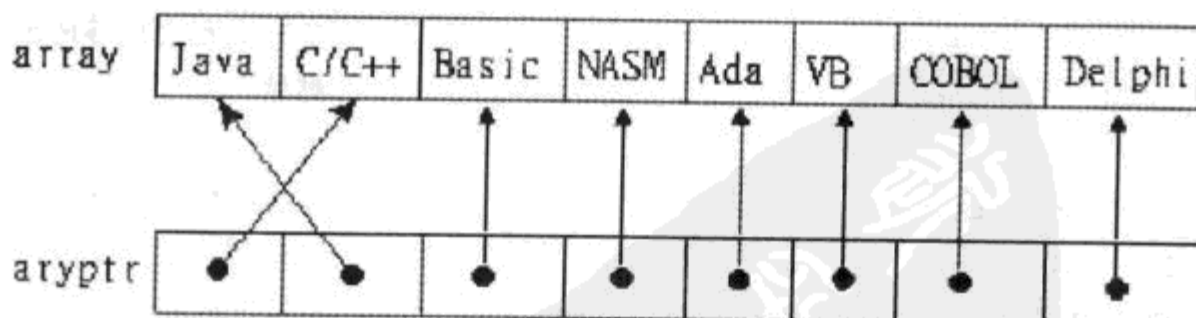


图 13-4 `Java` 与 `C/C++` 地址指针互换

最后排序的结果如图 13-5 所示。

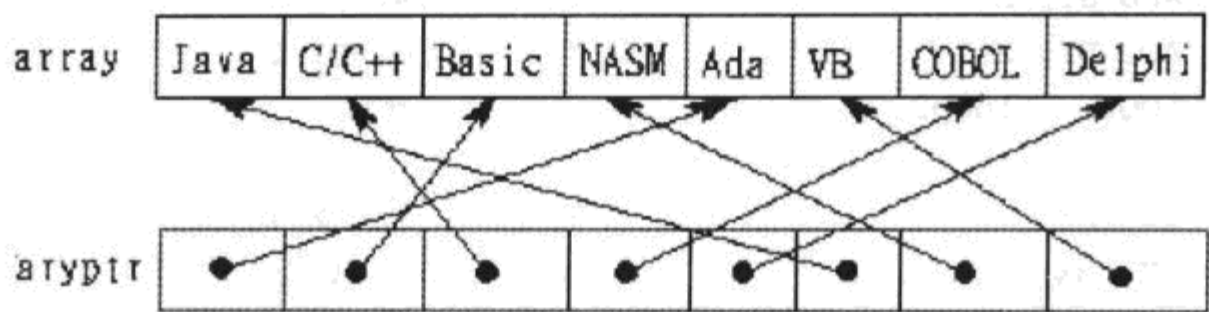


图 13-5 字符串排序最后结果的地址指针

程序第 47~54 输出 arrayptr 地址所指的字符串。arrayptr+0 所指的字符串为最小值的 Ada，arrayptr+2 所指的字符串为 Basic，arrayptr+14 所指的字符串为最大值的 VB。

【程序 sortstr.asm】

```

1 ; ***** sortstr.asm *****
2     ORG 0100H
3     JMP start
4 count DW 8
5 lenkey DW 7
6 array DB "Java ", "C/C++ ", "Basic ",
7         DB "NASM ", "Ada ", "VB ",
8         DB "COBOL ", "Delphi "
9 strptr TIMES 8 DW 0
10 result DW 0
11 ;
12 %include "..\mymacro\dispnchr.mac"
13 %include "..\mymacro\strncmp.mac"
14 ;
15 start:
16     PUSH WORD [count]                ;压入堆栈顶端
17     MOV CX, [count]                  ;CX=数组元素个数
18     MOV BX, array                    ;BX=数组偏移地址
19     MOV SI, 0                        ;SI=0 索引值
20 loop1:
21     MOV [strptr+SI], BX              ;SI 所指元素值(地址)
22     ADD SI, 2                        ;下一个元素
23     ADD BX, [lenkey]                 ;下一个元素偏移地址
24     LOOP loop1                       ;继续
25 loop3:
26     MOV CX, [count]                 ;CX=数组元素个数
27     DEC CX                           ;从 0 计数
28     MOV BX, 0                        ;BX=0
29     MOV SI, [strptr+BX]              ;BX 偏移地址所指内容
30     MOV DI, [strptr+BX+2]            ;BX+2 所指元素内容

```

```

31 loop2:
32     strncmp SI, DI, lenkey, result      ;SI 与 DI 元素值比较
33     CMP WORD [result], 0                ;SI 元素值较小?
34     JL      less                        ;是
35     MOV WORD [strptr+BX], DI            ;DI 与
36     MOV WORD [strptr+BX+2], SI          ;SI 偏移值互换
37 less:
38     ADD BX, 2                            ;BX=BX+2, 下一个
39     MOV SI, [strptr+BX]                  ;BX 偏移地址所指内容
40     MOV DI, [strptr+BX+2]                ;BX+2 所指元素内容
41     LOOP    loop2                        ;继续
42 ;
43     DEC WORD [count]                     ;count 值减 1
44     CMP WORD [count], 1                  ;count 值>1?
45     JG      loop3                        ;是
46 ;
47     POP CX                               ;取回元素个数
48     MOV BX, [strptr]                     ;BX=第 0 个元素地址
49     MOV SI, 0                             ;SI=0, 索引
50 loop4:
51     dispnchr BX, lenkey                   ;显示 BX 所指内容
52     ADD SI, 2                             ;SI=SI+2, 下一个
53     MOV BX, [strptr+SI]                   ;BX=下一个元素地址
54     LOOP    loop4                        ;继续
55     RET

```

【运行】

```
howasm\ch13> nasmw sortstr.asm -o sortstr.com <Enter>
```

```
howasm\ch13> sortstr <Enter>
```

```
Ada Basic C/C++ COBOL Delphi Java NASM VB
```

13-5 队列

数组用于存储静态数据 (static data)，队列 (queue)、链表 (list)、堆栈 (stack) 等数据结构用于动态数据 (dynamic data) 的访问。队列的插入 (insert) 发生于后端，删除 (remove) 发生于前端。堆栈的插入及删除均发生于栈顶端。链表的插入及删除可发生于前端、后端或中间。由此看来队列及堆栈均是链表的一个特例。

程序 queue.asm 逐渐插入一个数据而构成一个队列。队列每一个节点 (node) 由数据域 data 及指针域 link 组成，程序中 data 及 link 均声明为 100 个元素的数组变量，也就是说您的数据不能超过 100 个。队列的头端指针命名为 head，尾端为 tail，刚开始时 head 及 tail 均指向一个内定的头节点 (head node)，头节点的数据域并无意义，其指针域的内容为 0，表示队列结束，如图 13-6 所示。

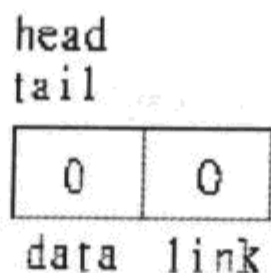


图 13-6 头节点为空的队列表示法

第 19~23 行执行队列初始化。第 25~32 行从键盘输入一个整数 numval。第 34~37 行取得一个空节点 SI，将 numval 置入 data 域，将 0 存入 link 域。第 39~41 行将 SI 节点插入队列 tail 后端，插入第一个节点数据值 33 后的队列如图 13-7 所示。

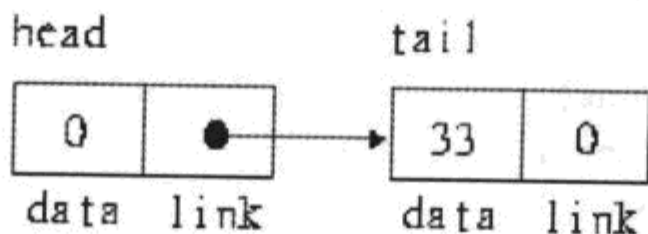


图 13-7 插入第一个节点 33 后的队列

第 42 行重复输入、建立节点、插入节点的操作，直到输入结束为止。继续插入数据值 11、55、44 后直接按〈Enter〉键结束，其队列如图 13-8 所示。

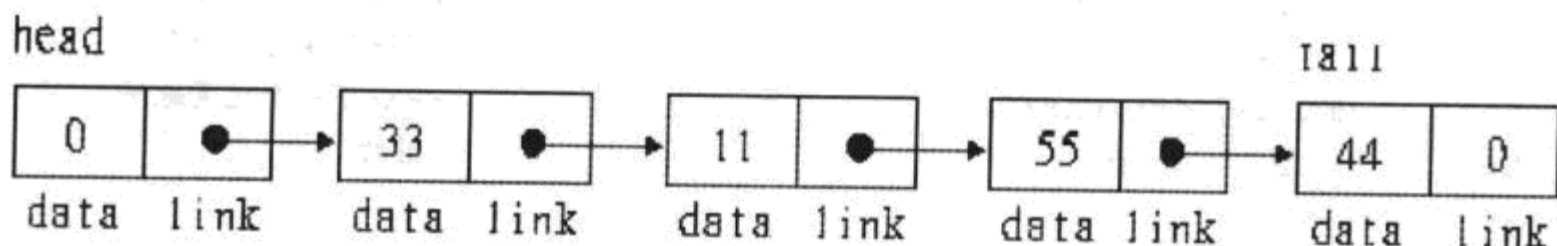


图 13-8 继续插入 11、55、44 后的队列

程序第 45~51 行从第一个数据节点 11 开始输出数据域，一直到尾节点处理后才停止。输出的顺序与输入的顺序相同，先进入队列结构就先输出，因此称为先进先出结构。

【程序 queue.asm】

```

1 ; ***** queue.asm *****
2     ORG 0100H
3     JMP start
4 data    TIMES    100 DW 0
5 link    TIMES    100 DW 0
6 head    DW 0
7 tail    DW 0
8 msg     DB      '请输入整数 (0-65535, Enter 结束) : ', '$'
9 numstr   TIMES 80 DB ' '
10 numval  DW 0
11 ;
12 %include "...mymacro\readstr.mac"
```

```

13 %include "..\mymacro\strtoi.mac"
14 %include "..\mymacro\dispi.mac"
15 %include "..\mymacro\dispstr.mac"
16 %include "..\mymacro\newline.mac"
17 ;
18 start:
19     MOV SI, 0                ;SI 为队列索引值
20     MOV WORD [head], SI      ;队列头节点索引值
21     MOV WORD [tail], SI      ;队列尾节点索引值
22     MOV WORD [data], SI      ;队列头节点数据域值
23     MOV WORD [link], SI      ;队列头节点指针域值
24 loop1:
25     dispstr msg              ;显示输入信息
26     readstr numstr           ;输入一个整数字符串
27     CMP BYTE [numstr], '$'   ;输入结束吗?
28     JNE next                 ;否
29     JMP endRead              ;是
30 next:
31     newline                  ;换行
32     strtoi numstr, '$', numval ;整数字符串转成数值
33 ;
34     ADD SI, 2                ;取得下一个可用节点
35     MOV AX, WORD [numval]
36     MOV WORD [data+SI], AX    ;SI 节点数据域值=numval
37     MOV WORD [link+SI], 0     ;SI 节点指针域值=0
38 ;
39     MOV DI, WORD [tail]      ;DI=尾节点索引值
40     MOV WORD [link+DI], SI    ;原尾节点指针域值=SI
41     MOV WORD [tail], SI      ;尾节点索引值=SI
42     JMP loop1                ;继续
43 endRead:
44     newline                  ;输入完毕,换行
45     MOV BX, WORD [head]      ;BX=头节点索引值
46     MOV BP, WORD [link+BX]   ;BP=头节点指针域值
47 loop2:
48     dispi data+BP            ;显示 BP 节点的数据域值
49     MOV BP, [link+BP]        ;BP=下一个节点索引值
50     CMP BP, 0                ;最后一个节点吗?
51     JNZ loop2                ;否
52     RET

```

【运行】

howasm\ch13> nasmw queue.asm -o queue.com <Enter>

```
howasm\ch13> queue <Enter>
```

```

请输入整数(0-65535, Enter 结束) : 33 <Enter>
请输入整数(0-65535, Enter 结束) : 11 <Enter>
请输入整数(0-65535, Enter 结束) : 55 <Enter>
请输入整数(0-65535, Enter 结束) : 44 <Enter>
请输入整数(0-65535, Enter 结束) : <Enter>
33 11 55 44

```

13-6 堆栈

堆栈的插入及删除均发生在顶端。程序 stack.asm 第 18~19 行执行堆栈初始化, SI 为堆栈索引值, top 为堆栈顶端头节点索引值, 开始时 SI 及 top 值均为 0 值, 往后若 top 值指向第 0 个元素时就表示堆栈为空。如图 13-9 所示。

第 21~28 行从键盘输入一个整数 numval。第 30~35 行取得一个空白节点 SI, 将 numval 置入 data 栏, 将 top 存入 link 栏, 并将 top 移至 SI 位置。插入第一个节点 33 后的堆栈如图 13-10 所示。



图 13-9 堆栈为空时 top 指向第 0 个元素

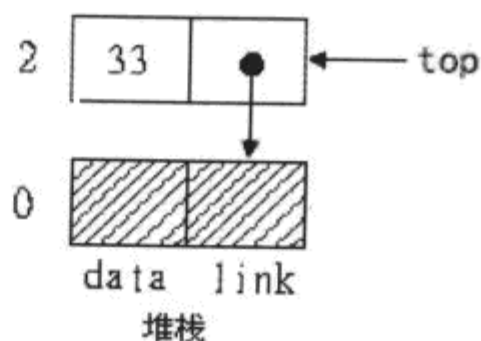


图 13-10 插入第一个节点 33 后的堆栈

第 36 行重复输入、建立节点、插入节点的操作, 直到输入结束为止。继续插入数据值 11、55、44 后直接按<Enter> 键结束, 其堆栈如图 13-11 所示。

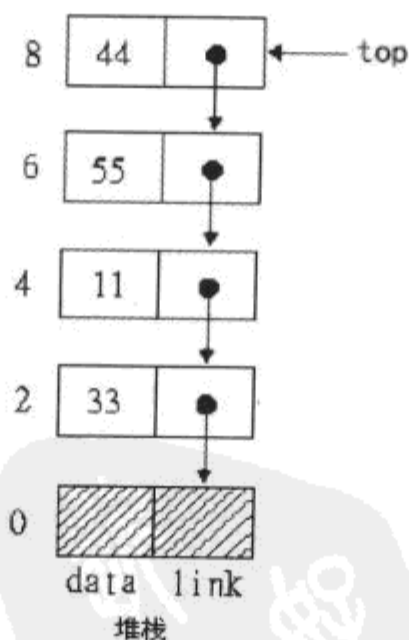


图 13-11 继续插入 11、55、44 后的堆栈

程序第 39~44 行从第一个数据节点 11 开始输出数据域, 一直到堆栈为空为止。输出的顺

序与输入的顺序相反, 先进入堆栈结构者后输出, 因此称为先进后出结构, 也称为后进先出。

【程序 stack.asm】

```

1 ; ***** stack.asm *****
2     ORG 0100H
3     JMP start
4 data TIMES 100 DW 0
5 link TIMES 100 DW 0
6 top  DW 0
7 msg  DB      '请输入整数(0-65535, Enter 结束) : ', '$'
8 numstr TIMES 80 DB ' '
9 numval DW 0
10 ;
11 %include "..\mymacro\readstr.mac"
12 %include "..\mymacro\strtoi.mac"
13 %include "..\mymacro\dispi.mac"
14 %include "..\mymacro\dispstr.mac"
15 %include "..\mymacro\newline.mac"
16 ;
17 start:
18     MOV SI, 0                                ;SI 为堆栈索引值
19     MOV WORD [top], SI                       ;堆栈顶端头节点索引值
20 loop1:
21     dispstr msg                                ;显示输入信息
22     readstr numstr                            ;输入一个整数字符串
23     CMP BYTE [numstr], '$'                   ;输入结束吗?
24     JNE next                                ;否
25     JMP     endRead                          ;是
26 next:
27     newline                                换行
28     strtoi numstr, '$', numval                ;整数字符串转换成数值
29 ;
30     ADD SI, 2                                ;取得下一个可用节点
31     MOV BX, WORD [top]
32     MOV AX, WORD [numval]
33     MOV WORD [data+SI], AX                    ;SI 节点数据域值=numval
34     MOV WORD [link+SI], BX                    ;SI 节点指针域值=top
35     MOV WORD [top], SI                        ;top=SI
36     JMP loop1                                ;继续
37 endRead:
38     newline                                ;输入完毕, 换行
39     MOV BP, WORD [top]                        ;BX=top
40 loop2:

```

```

41      dispi    data+BP      ;显示 BP 节点的数据域值
42      MOV BP, [link+BP]    ;BP=下一个节点索引值
43      CMP BP, 0            ;最后一个节点吗?
44      JNZ loop2
45      RET

```

【运行】

```

howasm\ch13> nasmw stack.asm -o stack.com <Enter>
howasm\ch13> stack <Enter>
请输入整数(0-65535, Enter 结束) : 33 <Enter>
请输入整数(0-65535, Enter 结束) : 11 <Enter>
请输入整数(0-65535, Enter 结束) : 55 <Enter>
请输入整数(0-65535, Enter 结束) : 44 <Enter>
请输入整数(0-65535, Enter 结束) : <Enter>
44 55 11 33

```

13-7 链表

数组用于存储静态数据，但链表可存储动态数据，其插入及删除可发生于前端、后端或中间。

程序 list.asm 逐渐插入一个数据而构成一个链表。链表每一个节点由数据域 data 及指针域 link 组成，程序中 data 及 link 均声明为 100 个元素的数组变量。链表的头端指针 head，尾端为 tail，刚开始时 head 及 tail 均指向一个内定的头节点，头节点的数据域存入-1，为我们排序的最小值，因为我们现在要排序的是正整数，若不是正整数，则其值要另外调整，指针域的内容为 0，表示链表的结束。如图 13-12 所示。

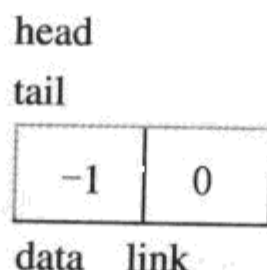


图 13-12 带有头节点链表空白时的表示法

第 19~23 行执行链表初始化。第 25~32 行从键盘输入一个整数 numval。第 34~36 行取得一个空节点 SI，将 numval 置入 data 域。第 38~51 行在链表中搜索这个节点的位置，它要比前面的节点小但要比后面的节点大，搜索的结果在 BX 节点及 BP 节点之间插入 SI 节点，插入第一个节点数据值 33 后的链表如图 13-13 所示。

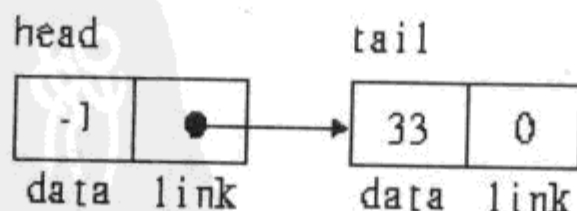


图 13-13 插入第一个节点 33 后的链表

第 52 行重复输入、建立节点、插入节点的操作,直到输入结束为止。继续插入数据值 11 后,其链表如图 13-14 所示。

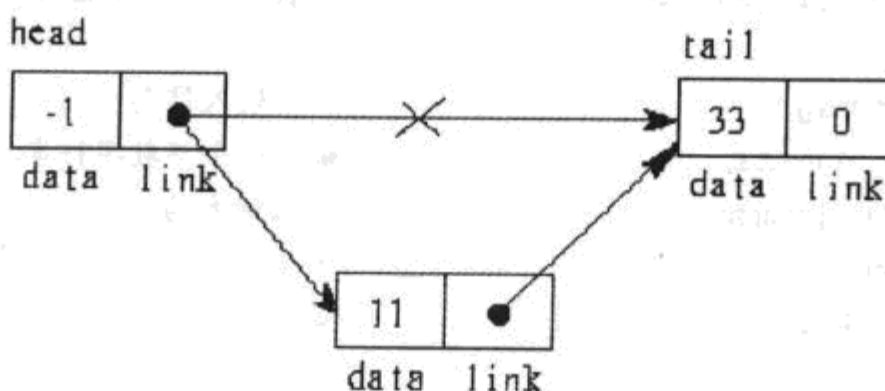


图 13-14 继续插入 11 后的链表

继续插入数据值 55、44 后直接按<Enter>键结束,其链表如图 13-15 所示。

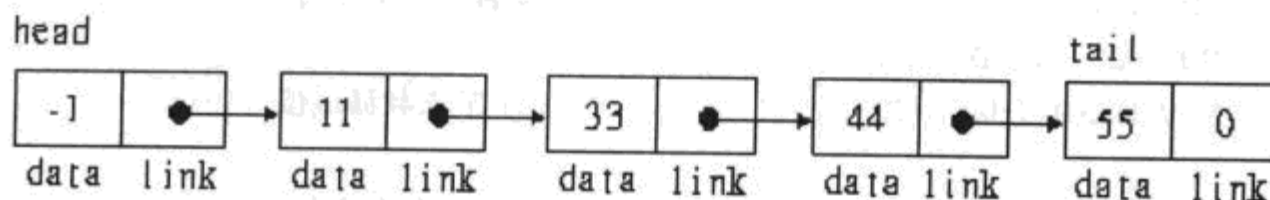


图 13-15 继续插入 54、44 后的链表

程序第 61~67 行从第一个数据节点数据值 11 开始输出,一直到尾节点处理后才停止。输出的结果依由小到大顺序排列。

【程序 list.asm】

```

1 ; ***** list.asm *****
2     ORG 0100H
3     JMP start
4 data  TIMES 100 DW 0
5 link  TIMES 100 DW 0
6 head  DW 0
7 tail  DW 0
8 msg   DB      '请输入整数(0-65535, Enter 结束) : ', '$'
9 numstr  TIMES 80 DB ' '
10 numval  DW 0
11 ;
12 %include "..\mymacro\readstr.mac"
13 %include "..\mymacro\strtoi.mac"
14 %include "..\mymacro\dispi.mac"
15 %include "..\mymacro\dispstr.mac"
16 %include "..\mymacro\newline.mac"
17 ;
18 start:
19     MOV SI, 0
20     MOV WORD [head], SI

```

;SI 为链表索引值
;链表头节点索引值

```

21      MOV WORD [tail], SI      ;链表尾节点索引值
22      MOV WORD [data], -1     ;链表头节点数据域值
23      MOV WORD [link], SI     ;链表头节点指针域值
24 loop1:
25      dispstr msg              ;显示输入信息
26      readstr numstr           ;输入一个整数字符串
27      CMP BYTE [numstr], '$'   ;输入结束吗?
28      JNE next                 ;否
29      JMP endRead              ;是
30 next:
31      newline                   ;换行
32      strtol numstr, '$', numval ;整数字符串转换成数值
33 ;
34      ADD SI, 2                 ;取得下一个可用节点
35      MOV AX, WORD [numval]
36      MOV WORD [data+SI], AX    ;SI 节点数据域值=numval
37 ;
38      MOV BP, WORD [head]       ;BP=链表头节点索引值
39      MOV BX, BP                ;BX=BP
40 comp:
41      MOV AX, [data+BP]         ;AX=BP 节点的数据
42      CMP WORD [numval], AX     ;numval<AX
43      JL      less              ;是
44      MOV BX, BP                ;BX=BP, 为 BP 的前一个
45      MOV BP, WORD [link+BP]    ;BP=下一个
46      CMP BP, 0                 ;链表结束了吗?
47      JE      endList           ;是
48      JMP comp                  ;否
49 less:
50      MOV WORD [link+SI], BP    ;SI 节点插入 BX 与 BP
51      MOV WORD [link+BX], SI    ;两节点之间
52      JMP loop1                 ;继续
53 endList:
54      MOV WORD [link+SI], 0     ;SI 节点指针域值=0
55      MOV DI, WORD [tail]       ;DI=尾节点索引值
56      MOV WORD [link+DI], SI    ;原尾节点指针域值=SI
57      MOV WORD [tail], SI      ;尾节点索引值=SI
58      JMP loop1                 ;继续
59 endRead:
60      newline                   ;输入完毕,换行
61      MOV BX, WORD [head]       ;BX=头节点索引值
62      MOV BP, WORD [link+BX]    ;BP=头节点指针域值
63 loop2:

```

```

64      dispi    data+BP                ;显示 BP 节点的数据域值
65      MOV BP, [link+BP]              ;BP=下一个节点索引值
66      CMP BP, 0                      ;最后一个节点吗?
67      JNZ loop2                      ;否
68      RET

```

【运行】

```

howasm\ch13> nasmw list.asm -o list.com <Enter>
howasm\ch13> list <Enter>
      请输入整数(0-65535, Enter 结束) : 33 <Enter>
      请输入整数(0-65535, Enter 结束) : 11 <Enter>
      请输入整数(0-65535, Enter 结束) : 55 <Enter>
      请输入整数(0-65535, Enter 结束) : 44 <Enter>
      请输入整数(0-65535, Enter 结束) : <Enter>
11 33 44 55

```

课后习题

1. 重复从键盘输入整数，构建一个整数数组 intary，输入-1 时结束输入，接着计算它们的平均值后输出至屏幕。
2. 字符串可以说是字符数组，假如有一个整数 i，它的内容为 123，现在要将它以六个位置显示在屏幕上，可能空三格后再显示 123，也可能显示 123 后再空三格，更可能显示 000123，请设计一个程序显示 000123。
3. 字符串可以说是字符数组，假如有一个整数 i，它的内容为 123，现在要将它以六个位置显示在屏幕上，前面空三格后再显示 123。
4. 将字整数 i 的内容 123 以指定位数（6）转换成字符串 istr 后显示在屏幕上，前面空格或补零以参数指定，请将它设计成宏 uitostrn，共有四个输入参数，其调用格式如下：

```
uitostrn wmem, smem, n, fillchar
```

将 wmem 字地址内容以 n 位置转换成字符串存入 smem 地址

有效数字左边补上 fillchar 字符

wmem : 字整数地址(传址)

smem : 字符串地址(传址)

n : 指定显示位数(传值)

fillchar : 填入字符(传值)

填入字符若为' '表示补空格，若为'0'表示补 0。请设计一个程序 ex1304.asm 测试它。

以 uitostrn i, istr, 6, '0' 进行测试。

5. 将双字整数 i 的内容 123456789 以指定位数（12）转换成字符串 istr 后显示在屏幕上，前面空格或补零以参数指定，请将它设计成宏 ultostrn，共有四个输入参数，其调用格式如下：

ultostrn dmem, smem, n, fillchar

将 dmem 双字地址内含以 n 位置转换成字符串存入 smem 地址

有效数字左边补上 fillchar 字符

dmem : 双字整数地址(传址)

smem : 字符串地址(传址)

n : 指定显示位数(传值)

fillchar : 填入字符(传值)

填入字符若为' ' 表示补空格, 若为'0' 表示补 0。请设计一个程序 ex1305.asm 测试它。

以 ultostrn i, istr, 12, '0' 进行测试。

6. 双字整数 i 声明如下:

i DD 45678901

将它除以 100 后, 其商 456789 存于双字变量 q 中, 其余数 01 存于双字 r 中。请将它输出, 如下所示:

456789.01

请利用 ultostrn 宏。

7. 重复从键盘输入字整数, 累积至双字变量 sum 中, 输入整数个数存于双字变量 count 中, 请输出平均值至小数二位。

8. 综合所得税每年三月份都要申报, 它的税率如下所示。

综合所得净额	税率	累进逆差
370000	0.06	0
990000	0.13	25900
1980000	0.21	105100
3720000	0.30	283300
99999999	0.40	655300

综合所得税= 综合所得净额×税率-累进逆差

例如您的综合所得净额为 500000 元, 那么您的综合所得税

$$500000 \times 0.13 - 25900 = 39100$$

请设计一个程序从键盘输入综合所得净额, 依税率表算出综合所得税, 然后将它输出至屏幕。

9. 重复从键盘输入星期天数, 输入-1 结束, 将相对应的星期名称显示在屏幕上。输入 7 表示星期日。

星期天数	1	2	3	4	5	6	7
星期名称	星期一	星期二	星期三	星期四	星期五	星期六	星期日

10. 重复从键盘输入月份数, 输入-1 结束, 将相对应的月份名称显示在屏幕上。

月份数	1	2	3	4	5	6
月份名称	一月	二月	三月	四月	五月	六月
月份数	7	8	9	10	11	12
月份名称	七月	八月	九月	十月	十一月	十二月

11. 百货公司提供下列的商品。

商品编号	商品名称	商品单价
1001	电视机	10000
1005	吹风机	300
1009	洗衣机	12000
1013	烘干机	4000

重复从键盘输入商品编号，输入-1 结束，将相对应的商品名称及单价显示在屏幕上。

12. 上题若重复从键盘输入商品编号及数量，输入-1 结束，将相对应的商品名称、单价及金额显示在屏幕上。

13. 从键盘输入一个英文元音字符，经下列规律转换，转换后的字符显示在屏幕上。

原字符 AEIOUaeiou

转换后字符 bczyxXYZCB

14. 从键盘输入一个字符串，字符串中若有下列“原字符”的，经下列规律转换，若字符串中没有下列“原字符”的，则保持不变。转换后的字符串可当加密字符串使用。请将加密后的字符串显示在屏幕上。以“ILikeYou”测试它，加密后应为“zLZkYeCB”。

原字符 AEIOUaeiou bczyxXYZCB

转换后字符 bczyxXYZCB AEIOUaeiou

以“zLZkYeCB”测试它，解密后应为“IlikeYou”。



14

浮点数运算

本章对于 80x87 数值型协处理器 (coprocessor) 的要点作一个简单的介绍, 包括浮点堆栈 (floating-point stack)、状态字 (status word)、控制字 (control word) 以及浮点指令 (floating-point instruction) 的运算。



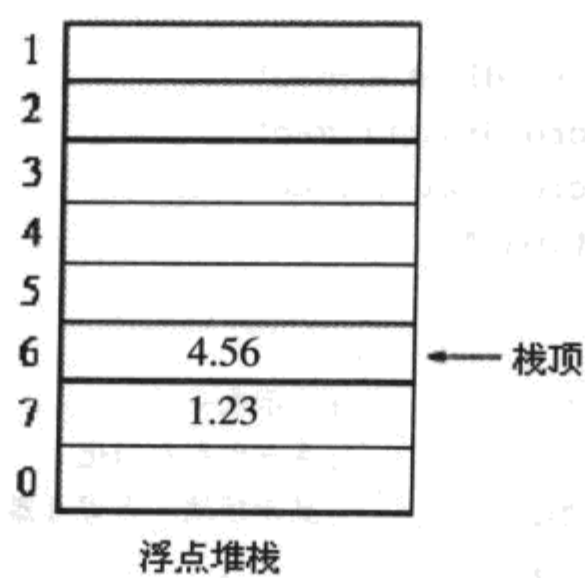


图 14-2 载入 a 及 b 后的浮点堆栈

将内存中的数值存入寄存器的操作称为载入 (load)，相反的操作，就是将寄存器的值存入内存的操作称为存储 (store)。但目前这八个浮点寄存器比较特别，它构成一个堆栈 (stack)，以堆栈的操作而言，将存于内存的数值存入堆栈顶端，这个操作称为压入 (push)，将顶端浮点寄存器的值存入内存的操作称为弹出 (pop)。弹出可使用浮点指令 FST，F 表示浮点 (floating-point) 的意思，ST 表示存储 (store)，指令后若附上 P 表示存储后还要作弹出 (pop) 的操作，例如 FSTP 指令。

```
FSTP    DWORD [a]      ;弹出顶端浮点数并存入 a 变量中
FSTP    DWORD [b]      ;弹出浮点数并入 b 变量中
```

程序 fpustack.asm 说明压入及弹出操作后，浮点堆栈顶端指针的变化情形。浮点堆栈顶端指针事实上就是浮点堆栈状态字中的 ST 栏。程序第 18 行执行 FINIT 指令初始化后，浮点堆栈状态字中的 ST 栏三位的值为 000，表示浮点堆栈顶端指针指向第 0 个浮点寄存器，其内容为空。程序第 20 行 FLD 指令压入 a 浮点数后，浮点堆栈状态字中的 ST 栏三位的值为 111，表示浮点堆栈顶端指针指向第 7 个浮点寄存器，其内容为 1.23。程序第 22 行 FLD 指令压入 b 浮点数后，浮点堆栈状态字中的 ST 栏三位的值为 110，表示浮点堆栈顶端指针指向第 6 个浮点寄存器，其内容为 4.56。程序第 24 行 FSTP 指令弹出到 a 变量后，浮点堆栈状态字中的 ST 栏三位的值为 111，表示浮点堆栈顶端指针指向第 7 个浮点寄存器，其内容为 1.23。程序第 26 行 FSTP 指令弹出到 b 变量后，浮点堆栈状态字中的 ST 栏三位的值为 000，表示浮点堆栈顶端指针指向第 0 个浮点寄存器，其内容为空。

【程序 fpustack.asm】

```
1 ; ***** fpustack.asm *****
2      ORG 0100H
3      JMP start
4 title DB ' B C3 ST C2 C1 C0'
5      DB ' IR PE UE OE ZE DE IE top', 13,10, '$'
6 space DB ' ', '$'
7 a     DD 1.23
8 b     DD 4.56
9 fpuFlag DW 0
```

10 ;

【运行】

```
howasm\ch14> nasmw fpustack.asm -o fpustack.com <Enter>
```

B	C3	ST	C2	C1	C0	IR	PE	UE	OE	ZE	DE	IE	top
0	0	0	0	0	0	0	0	0	0	0	0	0	-214748.3648
0	1	1	1	1	0	1	0	1	0	0	0	1	1.2300
0	0	1	1	0	0	0	0	1	1	0	0	1	4.5600
0	0	1	1	1	0	0	0	1	1	0	0	1	1.2300
0	0	0	0	0	0	0	0	1	1	0	0	1	-214748.3648

14-3 状态字

80x87 的状态字如图 14-3 所示, 可以反映整个协处理器的状况。状态字分为两项, 状态区 (status field) 及异常标志 (exception flag), 状态字可以使用 80x87 的指令 FSTSW 将它存至内存变量或存入 AX 寄存器, 再使用 80x86 的指令进行检查。

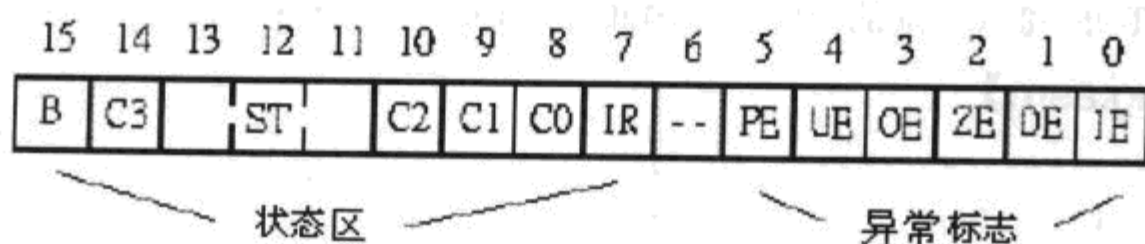


图 14-3 80x87 协处理器的状态字

B 状态区

指示 80x87 是否正在执行指令。

ST 状态区

指示当前浮点堆栈顶端是哪一个浮点寄存器。

C3 C2 C1 C0 状态区

提供浮点堆栈顶端的一些信息，例如其值是否为负数、零或正数等信息，让程序设计者可做适当的判断。

例如 FCOM、FCOMP、FCOMPP 指令将 TOS 与指定操作数比较，比较的结果设定浮点状态字中的 C0 及 C3 的值。

TOS 较小则 C0=1、C3=0，

TOS 较大则 C0=0、C3=0，

相等则 C0=0、C3=1，

若 TOS 为非数值 (NaN) 时无法比较，则 C0=1、C3=1。

IR

表示 80x87 是否要中断 80x86。

PE

精度异常 (Precision Exception)。

设为 1 时表示超出 80x87 的精度。

UE

Underflow Exception 其值设为 1 时表示必须执行 denormalize。

OE

Overflow Exception 其值设为 1 时表示溢出。

ZE

Zero Divide Exception 其值设为 1 时表示除数为 0。

DE

Denormalized Operand Exception 其值设为 1 时表示操作数必须执行 Denormalize 运算。

IE

Invalid Operation Exception 其值设为 1 时表示操作数不符合规定。

程序 fpuflag.asm 说明浮点数比较时，状态字中 C0 至 C3 各位变化的情形。

第 19 行执行 FINIT 浮点堆栈初始化的工作。第 20 行 FTST 指令将顶端寄存器值与 0 比较，刚初始化后顶端的值为空，为非数值，无法比较，因此设定 C0=1 及 C3=1。第 22~23 行压入 -1 后与 0 比较，TOS 堆栈顶端的值较小，因此设定 C0=1 及 C3=0。第 25~26 行压入 0 后与 0 比较，TOS 堆栈顶端的值相等，因此设定 C0=0 及 C3=1。第 28~29 行压入 1



后与 0 比较, TOS 堆栈顶端的值较大, 因此设定 $C0 = 0$ 及 $C3 = 0$ 。

【程序 fpuflag.asm】

```
1 ; ***** fpuflag.asm *****
2      ORG 0100H
3      JMP start
4 title DB ' B C3 ST C2 C1 C0'
5      DB ' IR PE UE OE ZE DE IE top',13,10,'$'
6 newline DB 13, 10, '$'
7 space DB ' ', '$'
8 negnum DD -1.0
9 zeronum DD 0.0
10 posnum DD 1.0
11 fpuFlag DW 0
12 ;
13 %include "..\mymacro\disptos.mac"
14 %include "..\mymacro\dispstr.mac"
15 %include "disp8bit.mac"
16 ;
17 start:
18      dispstr title           ;显示表头
19      FINIT                   ;浮点堆栈初始化
20      FTST                    ;顶端寄存器值与 0 比较
21      CALL dispTOS            ;显示状态字
22      FLD DWORD [negnum]      ;压入 negnum
23      FTST                    ;顶端寄存器值与 0 比较
24      CALL dispTOS            ;显示状态字
25      FLD DWORD [zeronum]     ;压入 zeronum
26      FTST                    ;顶端寄存器值与 0 比较
27      CALL dispTOS            ;显示状态字
28      FLD DWORD [posnum]      ;压入 posnum
29      FTST                    ;顶端寄存器值与 0 比较
30      CALL dispTOS            ;显示状态字
31      RET
32 dispTOS:
33      FSTSW WORD [fpuFlag]     ;弹出状态字
34      disp8bit fpuFlag+1       ;显示高 8 位
35      disp8bit fpuFlag         ;显示低 8 位
36      dispstr space            ;空一格
37      disptos 4                ;小数 4 位
38      dispstr newline         ;换行
39      RET
```

【运行】

```

howasm\ch14> nasmw fpuflag.asm -o fpuflag.com <Enter>
howasm\ch14> fpuflag <Enter>
B C3    ST    C2 C1 C0 IR    PE UE OE ZE DE IE  top
0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 -214748.3648
0 0 1 1 1 0 0 1 0 1 0 0 0 0 0 1 -1.0000
0 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1 .0
0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 1.0000

```

14-4 控制字

80x97 的控制字 (control word) 包含: 异常屏蔽 (exception mask)、中断屏蔽 (interrupt mask) 及控制屏蔽 (control mask) 等三项。

异常屏蔽包括:

Precision Mask (PM)

Underflow Mask (UM)

Overflow Mask (OM)

Zero-divide Mask (ZM)

Denormalized-operand Mask (DM)

Invalid-operation Mask (IM)

控制屏蔽包括:

1. 无限控制 (Infinity Control, IC 位)

其值为 0 表示投影法, 1 表示模拟法。

2. 舍入控制 (Rounding Control, RC 位)

其值 00 表示要四舍五入至整数, 01 表示不四舍五入至整数。

3. 精度控制 (Precision Control, PC 位)

包括短实数、长实数及浮点堆栈实数格式。

控制字如图 14-4 所示。

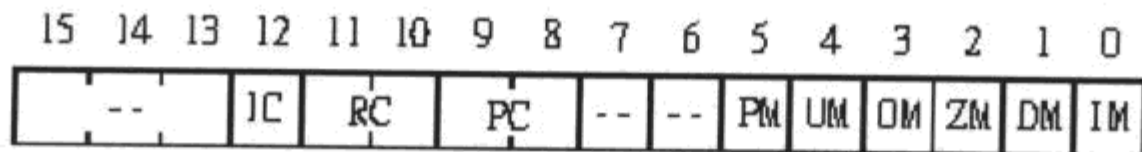


图 14-4 80x87 协处理器的控制字

程序 roundctr.asm 显示 FINIT 初始化后的控制字的每一个值, 其精度控制 PC 二位的内定值为 11B, 舍入控制 RC 二位的内定值为 00B, 内定为四舍五入, 因此将浮点数值 17.5 转换成整数变量时其值变为 18。第 24~26 行将控制字 RC 二位的值设定为 01B, 表示不用四舍五入, 因此将浮点数值 17.5 转换成整数变量时其值变为 17。

【程序 roundctr.asm】

```

1 ; ***** roundctr.asm *****
2 ;

```

```

3      ORG 0100H
4      JMP start
5 n      DD 17.5
6 f      DD 0
7 ctrlword DW 0
8 ;
9 %include "..\mymacro\disp8bit.mac"
10 %include "..\mymacro\dispul.mac"
11 %include "..\mymacro\newline.mac"
12 ;
13 start:
14      FINIT                                ;浮点堆栈初始化
15      FSTCW WORD [ctrlword]                ;取得控制字(要舍入)
16      disp8bit ctrlword+1                  ;显示高字节
17      disp8bit ctrlword                    ;显示低字节
18      newline                              ;换行
19      FLD DWORD [n]                        ;TOS=n
20      FISTP DWORD [f]                      ;f=TOS
21      dispul f                             ;显示 f 值
22      newline                              ;换行
23 ;
24      AND WORD [ctrlword], 00ffH           ;将 RC 及 PC 位清除
25      OR WORD [ctrlword], 0400H            ;设 RC=01 不舍入
26      FLDCW WORD [ctrlword]                ;设定控制字
27      disp8bit ctrlword+1                  ;显示高字节
28      disp8bit ctrlword                    ;显示低字节
29      newline                              ;换行
30      FLD DWORD [n]                        ;TOS=n
31      FISTP DWORD [f]                      ;f=TOS
32      dispul f                             ;显示 f 值
33      RET

```

【运行】

howasm\ch14> nasmw roundctr.asm -o roundctr.com <Enter>

howasm\ch14> roundctr <Enter>

0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 1 [注]原来的控制字

18 [注]17.5 四舍五入

0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 [注]调整 RC 位值后的控制字

17 [注]17.5 没四舍五入

14-5 数据类型

80x87 提供 7 种不同的数据类型，这七种不同的数据都可通过 80x86 标准的寻址模式来

访问, 每种数据类型的最高位均表示符号位, 0 表示正数, 1 表示负数。

14-5-1 二进制整数

80x87 的整数格式共有三种, 除了长度不同之外, 表示的方法均相同, 负数都用补码表示, 80x87 的字整数与 80x86 的 16 位带符号整数格式完全相同。这三种整数格式分别为字整数 (word integer)、短整数 (short integer) 以及长整数 (long integer)。

● 字整数

字整数其 16 位格式如下:

15	14	13...	0
符号	数--值--大---小		

最高第 15 位 0 表示正数, 1 表示负数。第 0 至 14 位表示数值大小。

若为负数则以补码表示。例如下列 0007H 以十六进制表示+7, 而 0fff9H 表示-7, 为补码表示法。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	位编号
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1 数值+7
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1 数值-7

字整数的范围为-32768 至+32767。

● 短整数

短整数其 32 位格式如下:

31	30	29	...	0
符号	数--值--大---小			

最高第 31 位 0 表示正数, 1 表示负数。第 0 至 30 位表示数值大小。若为负数则以补码表示。例如 00000007H 以十六进制表示+7, 而 0fffffff9H 表示-7, 为补码表示法。

● 长整数

长整数其 64 位格式如下:

63	62	61	...	0
符号	数--值--大---小			

最高第 63 位 0 表示正数, 1 表示负数。第 0 至 62 位表示数值大小。若为负数则以补码表示。例如 0000000000000007H 以十六进制表示+7, 而 0xfffffffffff9H 表示-7, 为补码表示法。

14-5-2 聚集十进制数

聚集十进制数 (packed decimal) 表示法是以四位表示一个十进制数, 因此每一个字节可以容纳两个十进制数, 其格式如下:

79	78	--	72	71	70	69	68	...	3	2	1	0
符号	没有定义		每四个位表示一个十进制数									

14-5-3 实数

实数 (real number) 为含有小数点的数, 分为短实数 (short real) 及长实数 (long real) 两种, 这两种格式均只出现在内存里, 一但压入浮点堆栈中则自动转换成临时实数 (temporay real) 格式, 此种格式仅存于浮点堆栈中, 因此也称为浮点堆栈实数 (floating-point stack real)。

● 短实数

31

30-23

22

21

20

...

3

2

1

0

符号

指-数

有

效

数

字

最高第 31 位 0 表示正数, 1 表示负数。第 30 至 23 位表示指数 (exponent), 7fH 表示零次方, 大于 7fH 表示正次方, 小于 7fH 表示负次方。第 22 至 0 位表示有效数字大小 (significant)。

例如 17.5 想要以短实数格式表示, 首先将它 normalize 规格化, 以二进制表示如下:

+ 1 0 0 0 1 . 1

然后将小数点移到最高位的右方:

+ 1 . 0 0 0 1 乘上 2 的 4 次方

从这里我们可以知道它为正数, 因此符号位为 0, 指数为 7fH+4H=83H, 有效数字的整数部分 1 为固定的值, 不必存放在内存里, 因此有效数字只含小数部分, 即 0001, 这三个部分都有了, 就可套入短实数的格式里, 如下所示:

31

30

29

28

27

26

25

24

23

22

21

20

19

18

17

16

15...2

1

0

0

1

0

0

0

0

0

1

1

.

0

0

0

1

1

0

0

0...

0

0

0

以十六进制表示为 418C0000H。符号位值 0 表示正数, 指数 10000011B 即 83H, 较 7fH 增加 4, 因此为 2 的 4 次方, 有效数字为 0.00011, 再加上 1 为 1.00011, 因此它的内容如下:

+ 1 . 0 0 0 1 1 乘上 2 的 4 次方

小数点往右移 4 位, 事实上它表示二进制+10001.1, 相当于十进制的 17.5, 它在内存里以十六进制表示为 418C0000H, 如程序 short.asm 所示。

【程序 short.asm】

```

1 ; ***** short.asm *****
2     ORG 0100H
3     JMP start
4 a     DD 17.5
5 ;
6 %include "..\mymacro\showbyte.mac"
7 ;
8 start:
9     showbyte a+3           ;最高字节
10    showbyte a+2
11    showbyte a+1
12    showbyte a             ;最低字节

```

13 RET

【运行】

```
howasm\ch14> nasmw short.asm -o short.com <Enter>
howasm\ch14> short <Enter>
418C0000
```

● 长实数

63 62-52 51 50 49 ... 3 2 1 0
符号 指-数 有 - - - - 效 - - - - 数 - - - - 字

最高第 63 位 0 表示正数, 1 表示负数。第 62 至 52 位表示指数, 3ffH 表示零次方, 大于 3ffH 表示正次方, 小于 3ffH 表示负次方。第 51 至 0 位表示有效数字大小。17.5 以长实数表示为 4031800000000000H, 如程序 long.asm 所示。

【程序 long.asm】

```
1 ; ***** long.asm *****
2     ORG 0100H
3     JMP start
4 a    DQ 17.5
5 ;
6 %include "..\mymacro\showbyte.mac"
7 ;
8 start:
9     showbyte a+7    ;最高字节
10    showbyte a+6
11    showbyte a+5
12    showbyte a+4
13    showbyte a+3
14    showbyte a+2
15    showbyte a+1
16    showbyte a      ;最低字节
17    RET
```

【运行】

```
howasm\ch14> nasmw long.asm -o long.com <Enter>
howasm\ch14> long <Enter>
4031800000000000
```

● 浮点堆栈实数

79 78-64 63 62 61 ... 3 2 1 0
符号 指-数 有 - - - - 效 - - - - 数 - - - - 字

最高第 79 位 0 表示正数, 1 表示负数。第 78 至 64 位表示指数, 3fffH 表示零次方, 大于 3fffH 表示正次方, 小于 3fffH 表示负次方。第 63 至 0 位表示有效数字大小。

14-5-4 七种数据类型值的范围

如图 14-5 所示。

数据类型	范 围	
字组整数	-32768	+32767
短整数	-2147483648	+2147483647
长整数	-922337203685477808	+922337203685477807
短实数	1.2×10^{-38}	3.4×10^{38}
长实数	2.3×10^{-308}	1.7×10^{308}
浮点堆栈实数	3.4×10^{-4932}	1.1×10^{4932}
聚集十进制数	-999999999999999999	+999999999999999999

图 14-5 七种数据类型值的范围

14-6 80x87 指令集

下面的指令对于 80387（含） 以上的数值型协处理器均适用。大部分的浮点数运算均以浮点堆栈顶端（Top of Stack, TOS）的浮点寄存器值为源操作数（source operand），因为名称太长，这个顶端元素就以 TOS 称呼它。本小节所介绍的是常用的几个指令，完整的指令请参阅附录。

F2XM1

计算 2 的 x 次方值后再减去一。x 为 TOS 值，计算结果仍存回 TOS。

FABS

取 TOS 值的绝对值后仍存回 TOS。

FADD, FADDP

将指定操作数的值加入 TOS，指令后面跟着 P，表示要将 TOS 值弹出。

FBLD, FBST, FBSTP

FBLD 将操作数地址的聚集十进制数压入 TOS，指令 FBST 将 TOS 弹出至操作数地址。

FNCHS

TOS 值改变符号。

FCOM, FCOMP, FCOMPP

TOS 与指定操作数比较，指令后面跟着两个 P，表示要将浮点堆栈弹出两次。比较的结果设定浮点状态字（FPU status word）中的 C0 及 C3 的值。TOS 较小则 C0=1、C3=0，相等则 C0=0、C3=1，TOS 较大则 C0=0、C3=0。若 TOS 为非数值（NAN）时无法比较，则 C0=1、C3=1。

FCOS, FSIN

求 TOS 弧度（radian） 值的余弦及正弦值，结果仍存回 TOS。

FDIV, FDIVP

将 TOS 除以指定操作数，结果仍存回 TOS。

FIADD

将指定操作数的整数值加入 TOS。

FICOM, FICOMP

TOS 与指定整数操作数比较, 比较的结果设定浮点状态字中的 C0 及 C3 的值。TOS 较小则 C0=1、C3=0, 相等则 C0=0、C3=1, TOS 较大则 C0=0、C3=0, 不相等则 C0=1、C3=1。

FIDIV

将 TOS 除以指定整数操作数, 结果仍存回 TOS。

FILD, FIST, FISTP

指令 FILD 将指定整数操作数压入浮点堆栈顶端。

指令 FIST 将 TOS 压入指定整数操作数。

指令 FISTP 将 TOS 弹出至指定整数操作数。

FIMUL

将 TOS 乘以指定整数操作数, 结果仍存回 TOS。

FINIT

浮点堆栈初始化, 堆栈顶端指针指向 ST0, 所有浮点寄存器内容均为空。

FISUB

从 TOS 减去指定操作数的整数值。

FLD

将指定操作数压入浮点堆栈顶端。

FLDxx

FLD1 压入常数 1。

FLDL2E 压入以 2 为底的自然对数 e。

FLDL2T 压入以 2 为底的对数 10。

FLDLG2 压入以 10 为底的对数 2。

FLDLN2 压入以 e 为底的对数 2。

FLDPI 压入圆周率。

FLDZ 压入常数 0。

FMUL, FMULP

将 TOS 乘以指定操作数, 结果仍存回 TOS。

FSQRT

求 TOS 平方根, 结果仍存回 TOS。

FST, FSTP

FST 将 TOS 压入指定操作数。

FSTP 将 TOS 弹出至指定操作数。

FSTSW

将 FPU 状态字存入指定操作数或 AX。

FSUB, FSUBP

从 TOS 减去指定操作数的值。



FTST

TOS 与 0 比较, 设定浮点状态字中的 C0 及 C3 的值。TOS 较小则 C0=1、C3=0, 相等则 C0=0、C3=1, TOS 较大则 C0=0、C3=0。若 TOS 为非数值 (NAN) 时无法比较, 则 C0=1、C3=1。

FXCH

TOS 与指定操作数值互换。

14-7 范例

程序 `dispf.asm` 显示短实数、长实数以及浮点堆栈实数格式的浮点数。其中引入 `dispd.mac` 文件以显示短实数, 引入 `dispf.mac` 文件以显示长实数, 引入 `disptos.mac` 文件以显示浮点堆栈实数。其调用格式说明如下:

`dispd dmem, decimal_places`

将 `dmem` 地址中带符号的双字数值转换为 ASCII 数字并显示在屏幕上

`dmem`: 双字地址

`decimal_place`: 小数字数 (常数, 传值)

`dispf qmem, decimal_places`

将 `qmem` 地址中带符号的四个字数值转换为 ASCII 数字并显示在屏幕上

`qmem`: 四个字地址

`decimal_place`: 小数字数 (常数, 传值)

`disptos decimal_places`

将浮点数堆栈顶端元素转换为 ASCII 数字并显示在屏幕上

`decimal_place`: 小数字数 (常数, 传值)

【程序 `dispf.asm`】

```
1 ; ***** dispf.asm *****
2     ORG 0100H
3     JMP start
4 a    DD 1234.56
5 b    DQ 1234567.89
6 ;
7 %include "..\mymacro\dispf.mac"
8 %include "..\mymacro\dispd.mac"
9 %include "..\mymacro\disptos.mac"
10 %include "..\mymacro\newline.mac"
11 ;
12 start:
13     FINIT                ;浮点堆栈初始化
14     dispd a, 2            ;显示短实数 1234.56
15     newline              ;换行
16     dispf b, 2           ;显示长实数 1234567.89
```

```

17      newline                ;换行
18      FLD DWORD [a]          ;TOS=a
19      disptos 2               ;显示 TOS 浮点堆栈实数 1234.56
20      newline                ;换行
21      FCHS                    ;TOS=-TOS
22      disptos 2               ;显示 TOS 浮点堆栈实数-1234.56
23      newline                ;换行
24      FSTP QWORD [b]          ;b=TOS
25      dispf b, 2              ;显示长实数-1234.56
26      newline                ;换行
27      RET

```

【运行】

```
howasm\ch14> nasmw dispf.asm -o dispf.com <Enter>
```

```
howasm\ch14> dispf <Enter>
```

```

1234.56          [注]原来双字变量 a 值(短实数)
1234567.89       [注]原来四个字变量 b 值(长实数)
1234.56          [注]a 值压入 TOS (浮点堆栈实数)
-1234.56         [注]TOS 改变符号
-1234.56         [注]弹出至四个字变量 b

```

程序 fiadd.asm 将两个双字整数 num1 及 num2, 使用 FILD 指令将其中的 num1 压入浮点堆栈顶端(TOS), 再使用 FIADD 指令将 num2 加上 TOS, 其和存于 TOS, 然后使用 FISTP 将 TOS 弹出并存入双字整数 sum 中, 接着将 sum 以十六进制数输出。

【程序 fiadd.asm】

```

1 ; ***** fiadd.asm *****
2      ORG 0100H
3      JMP start
4 num1 DD 12345678H
5 num2 DD 12345678H
6 sum  DD 0
7 msg  DB 'sum = ', '$'
8 ;
9 %include "..\mymacro\showbyte.mac"
10 %include "..\mymacro\dispstr.mac"
11 ;
12 start:
13      FINIT                    ;浮点堆栈初始化
14      FILD DWORD [num1]        ;TOS=整数 num1
15      FIADD DWORD [num2]       ;TOS=num1+num2
16      FISTP DWORD [sum]        ;sum=TOS
17      dispstr msg
18      showbyte sum+3           ;sum 最高字节
19      showbyte sum+2

```



```
20      showbyte sum+1
21      showbyte sum+0      ;sum 最低字节
22      RET
```

【运行】

```
howasm\ch14> nasmw fiadd.asm -o fiadd.com <Enter>
howasm\ch14> fiadd <Enter>
sum = 2468ACF0
```

程序 fadd.asm 将两个浮点数 num1 及 num2, 使用 FLD 指令将其中的 num1 压入浮点堆栈顶端(TOS), 再使用 FADD 指令将 num2 加上 TOS, 其和存于 TOS, 然后使用 FSTP 将 TOS 弹出并存入浮点数 sum 中, 接着将 sum 以短实数浮点数格式输出。

【程序 fadd.asm】

```
1 ; ***** fadd.asm *****
2      ORG 0100H
3      JMP start
4 num1 DD 12.5
5 num2 DD 67.8
6 sum  DD 1.0
7 msg  DB 'sum = ', '$'
8 ;
9 %include "..\mymacro\dispd.mac"
10 %include "..\mymacro\dispstr.mac"
11 ;
12 start:
13      FINIT                      ;浮点堆栈初始化
14      FLD DWORD [num1]           ;TOS=整数 num1
15      FADD DWORD [num2]          ;TOS=num1+num2
16      FSTP DWORD [sum]           ;sum=TOS
17      dispstr msg                ;显示 msg
18      dispd sum, 2               ;显示 sum 值
19      RET
```

【运行】

```
howasm\ch14> nasmw fadd.asm -o fadd.com <Enter>
howasm\ch14> fadd <Enter>
sum = 80.30
```

程序 fsqrt.asm 将双字整数 num, 使用 FILD 指令压入浮点堆栈顶端, 再使用 FSQRT 指令求出 num 的平方根, 存于 TOS, 然后使用 FSTP 将 TOS 弹出并存入双字浮点数 sqrt 中, 接着将 sqrt 以短实数浮点数格式输出。

【程序 fsqrt.asm】

```
1 ; ***** fsqrt.asm *****
2      ORG 0100H
```

```

3      JMP start
4 num   DD 3
5 sqrt  DD 0.0
6 msg   DB 'square root of 3 = ', '$'
7 ;
8 %include "..\mymacro\dispd.mac"
9 %include "..\mymacro\dispstr.mac"
10 ;
11 start:
12      FINIT                      ;浮点堆栈初始化
13      FILD DWORD [num]           ;TOS=整数 num
14      FSQRT                      ;TOS=TOS 的平方根
15      FSTP DWORD [sqrt]         ;sqrt=TOS
16      dispstr msg                ;显示 msg
17      dispd sqrt,6              ;显示平方根,6 位小数
18      RET

```

【运行】

```

howasm\ch14> nasmw fsqrt.asm -o fsqrt.com <Enter>
howasm\ch14> fsqrt <Enter>

```

square root of 3 = 1.732051

程序 froot.asm 求出 $x \times x + 3 \times x + 2 = 0$ 一元二次方程式的一个根, 存入长实数变量 x 中, 接着将 x 以长实数浮点数格式输出。

【程序 froot.asm】

```

1 ; ***** froot.asm *****
2      ORG 0100H
3      JMP start
4 a     DQ 1.0
5 b     DQ 3.0
6 c     DQ 2.0
7 d     DQ 0.0
8 sqr   DD 0
9 four  DQ -4.0
10 two  DQ 2.0
11 work DQ 0.0
12 msg  DB 'x*x + 3*x + 2 = 0 其一根为', '$'
13 ;
14 %include "..\mymacro\dispstr.mac"
15 %include "..\myproc\DispF.pro"
16 ;
17 start:
18      FINIT                      ;浮点堆栈初始化
19      FLD QWORD [b]              ;TOS=b

```



```
20      FMUL QWORD [b]          ;TOS=b*b
21      FSTP QWORD [d]          ;d=b*b
22      FLD QWORD [a]           ;TOS=a
23      FMUL QWORD [c]          ;TOS=a*c
24      FMUL QWORD [four]       ;TOS=4*a*c
25      FSTP QWORD [work]       ;work=4*a*c
26      FLD QWORD [d]           ;TOS=d=b*b
27      FADD QWORD [work]       ;TOS=b*b-4*a*c
28      FSQRT                   ;TOS=(b*b-4*a*c)的平方根
29      FSUB QWORD [b]          ;TOS=-b+(b*b-4*a*c)的平方根
30      FDIV QWORD [a]          ;TOS=TOS/a
31      FDIV QWORD [two]        ;TOS=TOS/2
32      dispstr msg             ;显示 msg
33      MOV AL, 2                ;小数 2 位
34      CALL DispF              ;显示 TOS 值
35      FISTP DWORD [sqr]       ;sqr=TOS
36      RET
```

【运行】

```
howasm\ch14> nasmw froot.asm -o froot.com <Enter>
```

```
howasm\ch14> froot <Enter>
```

$x^2 + 3x + 2 = 0$ 其一根为-1.0

程序 fbcd.asm 将 10 字节变量 fnum 使用 FLD 指令载入浮点堆栈顶端, 然后使用指令 FBSTP 将它存于 fbcd 10 字节变量中, 接着使用 bcdtostr 宏转成 ASCII 字符后输出。

【程序 fbcd.asm】

```
1 ; ***** fbcd.asm *****
2      ORG 0100H
3      JMP start
4 fnum  DT 123456789123456789.0
5 fbcd  DT 0.0
6 msg   DB '浮点数 123456789123456789.0 转成 BCD 数字= '
7 newline DB 13, 10, '$'
8 ascnum DB ' ', '$'
9 ;
10 %include "..\mymacro\dispstr.mac"
11 %include "..\mymacro\bcdtostr.mac"
12 ;
13 start:
14      FINIT                ;浮点堆栈初始化
15      FLD TWORD [fnum]     ;TOS=整数 fnum
16      FBSTP TWORD [fbcd]   ;TOS 转换成压缩 BCD 数
17 ;
18      dispstr msg          ;显示 msg
```

```

19      bcdtostr fbcd, ascnum    ;将压缩 BCD 数转换成 ASCII 数字
20      dispstr ascnum          ;显示 ASCII 数字
21      RET

```

【运行】

```
howasm\ch14> nasmw fbcd.asm -o fbcd.com <Enter>
```

```
howasm\ch14> fbcd <Enter>
```

浮点数 123456789123456789.0 转成 BCD 数字=

123456789123456789

程序 fsin.asm 计算 30° 的正弦、余弦及正切三角函数值后输出。

【程序 fsin.asm】

```

1 ; ***** fsin.asm *****
2      ORG 0100H
3      JMP start
4 x      DD 30.0
5 radian  DD 0.0
6 pi      DQ 0.0
7 deg180 DD 180.0
8 sinx    DD 0.0
9 cosx    DD 0.0
10 tanx   DD 0.0
11 msg    DB ' 度数 x sin(x) cos(x) tan(x) '
12 tab    DB 09H, '$'
13 ;
14 %include "..\mymacro\dispstr.mac"
15 %include "..\mymacro\newline.mac"
16 %include "..\mymacro\dispf.mac"
17 %include "..\mymacro\disptos.mac"
18 ;
19 start:
20      dispstr msg
21      newline
22      FINIT                ;浮点堆栈初始化
23      FLDPI                ;TOS=圆周率
24      FSTP DWORD [pi]      ;pi=圆周率
25      FLD DWORD [x]        ;TOS=x
26      disptos 4
27      dispstr tab          ;跳至下一位置
28      FMUL DWORD [pi]      ;TOS=x*pi
29      FDIV DWORD [deg180]   ;TOS=x*pi/180
30      FSTP DWORD [radian]  ;radian=x*pi/180
31 ;
32      FLD DWORD [radian]    ;TOS=x*pi/180

```

```

33      FSIN                      ;TOS=sin(x)值
34      disptos 4                 ;小数4位
35      dispstr tab              ;跳至下一位置
36      FSTP DWORD [sinx]        ;sinx=sin(x)值
37 ;
38      FLD DWORD [radian]       ;TOS=x*pi/180
39      FCOS                     ;TOS=cos(x)值
40      disptos 4                 ;小数4位
41      dispstr tab              ;跳至下一位置
42      FSTP DWORD [cosx]        ;cosx=cos(x)值
43 ;
44      FLD DWORD [sinx]         ;TOS=sinx
45      FDIV DWORD [cosx]        ;TOS=sinx/cosx
46      disptos 4                 ;小数4位
47      dispstr tab              ;跳至下一位置
48      FSTP DWORD [tanx]        ;tanx=tan(x)值
49      RET

```

【运行】

```
howasm\ch14> nasmw fsin.asm -o fsin.com <Enter>
```

```
howasm\ch14> fsin <Enter>
```

```

    度数 x sin(x) cos(x) tan(x)
30.0000 .5000 .8660 .5774

```

课后习题

1. 使用浮点堆栈将两个双字变量值相加后，其和显示在屏幕上。以 12345678H 及 00987654H 测试。其和为 12CCCCCH。
2. 使用浮点堆栈将两个双字变量值相加后，其和显示在屏幕上。以 1234.567 及 876.321 测试。其和为 2110.888。
3. 使用浮点堆栈将两个双字变量值相乘后，其乘积显示在屏幕上。以 1234.567 及 876.321 测试。
4. 使用浮点堆栈将两个双字变量值相除后，其商显示在屏幕上。以 1234.567 除以 876.321 测试。
5. 重复从键盘输入浮点数字字符串，直接按<Enter>键时结束输入，计算它们的平均值至小数点后两位，输出至屏幕。以：

```
12.3<Enter>
```

```
34.5<Enter>
```

```
<Enter>
```

测试。平均值为 23.40。

6. 重复从键盘输入浮点数字符串, 直接按<Enter> 键时结束输入, 计算每一个浮点数的平方根至小数后六位, 输出至屏幕。以:

2.0<Enter>

3.0<Enter>

<Enter>

测试。2 的平方根为 1.414214, 3 的平方根为 1.732051。

7. 重复从键盘输入圆半径浮点数字符串, 直接按<Enter> 键时结束输入, 计算每一个圆面积至小数四位, 输出至屏幕。圆面积 = 圆周率 \times 半径 \times 半径。以:

10.0<Enter>

20.0<Enter>

<Enter>

测试。面积各为 314.1593 及 1256.6371。

8. 已知数学函数 $f(x) = x \times x \times x - 3.0$ 在 $x=0$ 及 $x=2$ 之间有根。因为 $f(0) = -3$, $f(2) = 5$ 函数值异号, 因此其间存在某一 x 值, 满足 $f(x) = 0$, 请使用二分搜索法 (binary search) 求出 x 至小数后六位。
9. 制作一个正弦函数表, 从 0° 至 90° , 间隔 5° 。正弦函数值取至小数后六位。
10. 从键盘输入三个浮点数 a 、 b 、及 c , 求 $a \times x \times x + b \times x + c = 0$ 方程式的实数根。



15

连接程序

本章说明 NASM 汇编语言的汇编、连接以及建立.exe 可执行文件的步骤。在互联网上提供若干免费的连接程序，本章使用 NASM 主网站所推荐的连接程序 `alink.exe`，您可从下列网站免费下载：

<http://www.geocities.com/SiliconValley/Network/#alink>

您下载的文件为 `al_al.zip`，将它解压缩后，将其主要的程序 `alink.exe` 存入所建目录中的步骤如下：

1. 建立 NASM 汇编语言源程序。
2. 使用 `nasmw` 建立目标文件。
3. 使用 `alink` 将目标文件连接成一个可执行文件。





15-1 建立 NASM 源程序

程序 `hello.asm` 里包含三个段, 程序代码段、数据段及堆栈段。这三个段都使用 NASM 汇编语言的 `SEGMENT` 指令定义, 如下:

程序代码段定义为 `SEGMENT code`

数据段定义为 `SEGMENT data`

堆栈段定义为 `SEGMENT stack stack`

`code`、`data` 及 `stack` 分别为段的名称。堆栈段定义第二个 `stack` 为堆栈段定义的属性, 有些连接程序会使用这个属性判断您的程序有没有定义堆栈。

`SEGMENT` 汇编语言指令除了声明段名称之外, 其后可跟随着可选的属性, 有下列的属性可用:

1. `PRIVATE`、`PUBLIC`、`COMMON`、`STACK`
2. `ALIGN`
3. `CLASS`
4. `OVERLAY`
5. `USE16`、`USE32`
6. `FLAT`
7. `ABSOLUTE`

第 3 行声明一个 `..start` 标号, 这个标号很特别, 它是连接成可执行程序后首先要执行的指令地址。第 4~5 行在程序代码段里建立数据段的起始地址。第 6~8 行在程序代码段里建立堆栈段的起始地址以及堆栈顶端指针的地址。这些地址建立完成之后才可以做您要做的事情。第 10 行将 `msg` 的偏移地址转移到 `DX` 寄存器, 它是与 `DS` 寄存器搭配来形成绝对地址的。

第 11~12 行将 `msg` 显示在屏幕上。第 14~15 行返回操作系统。第 17~18 建立数据段。第 20~22 建立堆栈段。汇编时使用 `-f` 选项, 指明要产生一个 `obj` 文件, 文件名为 `hello.obj`。

```
nasmw -fobj hello.asm <Enter>
```

【程序 `hello.asm`】

```
1 ; ***** hello.asm *****
2 SEGMENT code ;程序代码段
3 ..start: ;开始执行地址
4     MOV AX, data
5     MOV DS, AX ;DS=data 段地址
6     MOV AX, stack
7     MOV SS, AX ;SS=stack 段地址
8     MOV SP, stacktop ;SP=堆栈顶端指针
9 ;
10    MOV DX, msg ;DX=msg 偏移地址
11    MOV AH, 9
12    INT 21H ;显示 msg 信息
13 ;
```

```

14      MOV AX, 4c00H          ;返回操作系统
15      INT 21H
16 ;-----
17 SEGMENT data                ;数据段
18 msg DB 'Hello, world!', 13, 10, '$'
19 ;-----
20 SEGMENT stack stack        ;堆栈段
21      RESB 64
22 stacktop:

```

【运行】

```
howasm\ch15> nasmw -fobj hello.asm <Enter>
```

15-2 将目标文件连接成.exe 文件

您建立目标文件之后,可以使用 alink.exe 连接程序将您需要的若干目标文件连接成一个可执行的.exe 文件。

```
alink [选项] 文件名[ [选项] 文件名] ...
```

ALINK 连接程序提供下列的选项:

- c 连接时区分英文大小写字母
- c+ 同上
- c- 连接时不区分英文大小写字母
- p 段边界填入初始值
- p+ 同上
- p- 段边界不填入初始值
- o yyy yyy 是输出文件名
- oxxx xxx 指定输出格式
 - COM = 输出 COM 文件
 - EXE = 输出 EXE 文件
 - PE = 输出 Win32 PE 文件(.EXE)
- m 产生 map 文件
- m+ 同上
- m- 不产生 map 文件
- L ddd 将 ddd 目录加入搜索目录中
- h 显示本选项说明
- H 同上
- ? 同上

例如将 hello.obj 做如下的连接:

```
howasm\ch15> alink hello.obj -oEXE hello.exe <Enter>
```

产生一个 hello.exe 可执行文件。

```
howasm\ch15> hello <Enter>
```

```
Hello, world!
```



15-3 显示 DOS 的 BIOS 区域数据

程序 `dumpbios.asm` 显示 DOS 操作系统 BIOS 数据区域的数据内容, 以十六进制显示。这个数据区域的地址从 `0040[0]H` 开始, 共 256 个字节。第 3 行声明 `code` 程序代码段, 第 43 行声明 `data` 数据段, 第 48 行声明 `stack` 堆栈段, 与上一个程序 `hello.asm` 的作法一样。第 5~7 行引入本程序使用到的宏。第 9~14 行声明执行起始地址 `..start` 以及数据段、堆栈段的起始地址及堆栈指针的地址。

第 16~39 行才是程序代码的主体, 第 16 行设定 `BL` 为 16, 表示每行要显示 16 个字节的值。第 17 行 `CX` 设定为 256 表示总共要显示 256 个字节。第 18 行设 `SI` 的值为 0, 表示索引值从 0 算起。第 19~20 行设定段寄存器 `ES` 的值为 `0040H`, 这是 DOS 操作系统 BIOS 数据区域的起始地址。`ES:SI` 地址的值就是该区域的第 0 个位值, 第 32 行使用 `showbyte` 宏以十六进制显示该字节的值。第 22~31 行判断是否已经显示 16 个字节, 若是则执行换行等操作。

【程序 `dumpbios.asm`】

```
1 ; ***** dumpbios.asm *****
2 ;
3 SEGMENT code ;程序代码段
4 ;
5 %include "../mymacro/showbyte.mac"
6 %include "../mymacro/dispchr.mac"
7 %include "../mymacro/newline.mac"
8 ;
9 ..start: ;开始执行地址
10     MOV AX, data
11     MOV DS, AX ;DS=data 段地址
12     MOV AX, stack
13     MOV SS, AX ;SS=stack 段地址
14     MOV SP, stacktop ;SP=堆栈顶端指针
15 ;
16     MOV BL, 16 ;BL=16
17     MOV CX, 256 ;CX=256
18     MOV SI, 0 ;SI=0
19     MOV AX, 0040H
20     MOV ES, AX ;ES=0040[0]H
21 repeat:
22     MOV AX, SI ;AX=SI
23     DIV BL ;AX/BL
24     CMP AH, 0 ;AH 余数为 0?
25     JNE next ;否
26     newline ;换行
```

```

27      MOV WORD [offset], SI      ;offset=偏移地址
28      showbyte offset           ;显示偏移地址
29      dispchr colon             ;显示分号
30      dispchr space            ;空一格
31 next:
32      showbyte ES:SI            ;显示 ES:SI 内容以十六进制显示
33      dispchr space            ;显示空格符
34      INC SI                    ;SI=SI+1 指向下一个字节
35 ;
36      DEC CX                    ;CX=CX-1
37      JCXZ endjob               ;CX=0?若是则跳至 endjob
38      JMP repeat                ;否,重复执行
39 endjob:
40      MOV AX, 4c00H             ;返回操作系统
41      INT 21H
42 ;-----
43      SEGMENT data               ;数据段
44      offset DW 0
45      colon DB ':'
46      space DB ' '
47 ;-----
48 SEGMENT stack stack            ;堆栈段
49      RESB 64
50 stacktop:

```

【运行】

```
howasm\ch15> nasmw -fobj dumpbios.asm <Enter>
```

```
howasm\ch15> nasmw dumpbios.obj <Enter>
```

```
howasm\ch15> dumpbios <Enter>
```

```

00: F8 03 F8 02 00 00 00 00 78 03 00 00 00 00 13 02
10: 27 C4 00 80 02 00 00 80 00 00 24 00 24 00 3E 34
20: 74 14 0D 1C 00 3B 00 3B 00 3B 00 3B 00 3B 00 3B
30: 00 3B 00 3B 00 3B 00 3B 00 3B 00 3B 20 39 00 80
40: 00 00 C0 00 00 00 00 00 00 00 03 50 00 00 10 00
50: 00 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 0E 0D 00 D4 03 09 30 A4 17 9B 84 FF AE 73 11 00
70: 00 00 00 00 00 01 00 00 14 14 14 3C 01 01 01 01
80: 1E 00 3E 00 18 10 00 60 09 11 0B 88 58 01 00 01
90: 07 07 00 00 00 00 10 00 00 00 00 00 00 00 00 00
A0: 00 00 00 D2 10 00 C0 00 A6 0C 00 C0 10 10 00 00
B0: 00 00 00 01 D5 00 44 00 00 00 84 00 00 00 00 00
C0: 00 00 00 00 00 00 0A 00 00 00 00 00 00 00 00 00

```

```

D0: 00 00 00 00 00 00 00 00 DC 04 00 00 00 00 00 00
E0: 00 00 00 00 00 00 00 C0 07 02 08 00 00 00 00 00
F0: AA 00 00 00 00 00 00 00 FC FF 4F 01 00 00 00 00
    
```

15-4 系统设备数据

在 DOS 的 BIOS 区域数据中，0040[0]H:10H 至 0040[0]H:11H 这两个字节，以后简称 10H-11H，表示系统的设备状态，您要取得这个字组可调用 INT 11H，会返回至 AX 中。它代表的意义如下：

- 位 0：值 1 表示安装软盘。
- 位 1：值 1 表示安装协处理器。
- 位 2：值 1 表示安装鼠标。
- 位 3：保留。
- 位 5~4：表示影像(video) 模式初始值。
 值 00 表示没有使用。
 值 01 表示 40×25 彩色。
 值 10 表示 80×25 彩色。
 值 11 表示 80×25 黑白。
- 位 7~6：表示磁盘数目。
 值 00 表示 1 个。
 值 01 表示 2 个。
 值 10 表示 3 个。
 值 11 表示 4 个。
- 位 11~9：表示 RS232 串行端口数目。
- 位 3：保留。
- 位 15~14：表示并行端口数目。

程序 dumpbios 运行的结果中 10H 地址的值为 27H，11H 地址的值为 C4H，其位编号及值如图 15-1 所示。

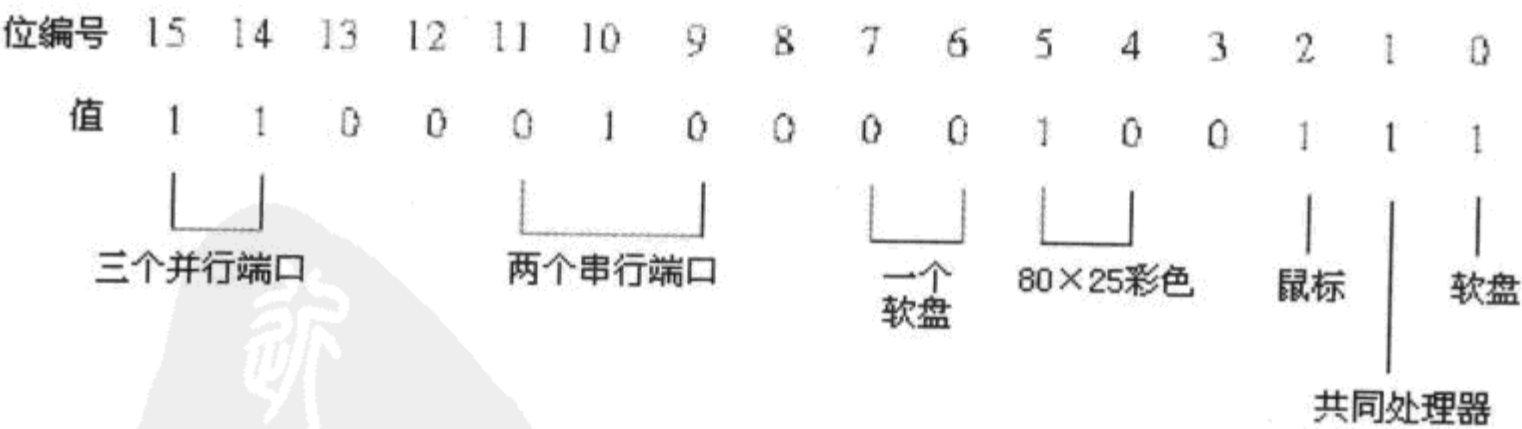
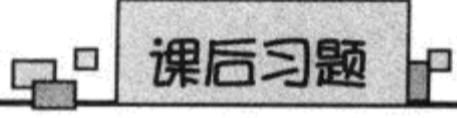


图 15-1 系统设备状态字

15-5 内存容量

在 DOS 的 BIOS 区域数据中，13H-14H 表示主板上的内存容量，15H-16H 表示扩展的

内存容量, 程序 dumpbios 运行的结果中 13H-14H 地址的值为 8002H, 15H-16H 地址的值为 0000H, 因此主机板上的内存容量为 0280H KB, 没有扩展的内存容量。

课后习题

1. 设计一个.exe 格式的程序, 调用 INT 11H, 返回系统的设备状态至 AX 中。
2. 设计一个.exe 格式的程序, 取得 DOS 的 BIOS 区域数据 13H-14H 并输出内存容量。



第 1 章 緒 言

1.1 研究の背景と意義

2. 研究の目的と範囲

2.1 研究の目的

新
平
和
学

PDG

附录

NASM 汇编语言指令

本章说明 NASM 汇编语言的汇编、连接以及建立.exe 可执行文件的步骤。在互联网上提供若干免费的连接程序，本章使用 NASM 主网站所推荐的连接程序 alink.exe，您可从下列网站免费下载：

<http://www.geocities.com/SiliconValley/Network/#alink>

您下载的文件为 al_al.zip，将它解压缩后，将其主要的程序 alink.exe 存入所建目录中的步骤如下：

1. 建立 NASM 汇编语言源程序。
2. 使用 nasmw 建立目标文件。
3. 使用 alink 将目标文件连接成一个可执行文件。





本附录只简短地说明 80x86 处理器的指令，您若要想了解详细的规格请参阅 Intel 公司的网站：

<http://www.intel.com/>。

本附录按照相类似的指令排列，并不是按照英文字母顺序排列。指令中用到的符号说明如下：

reg8	8 位寄存器
reg16	16 位寄存器
reg32	32 位寄存器
fpureg	浮点堆栈寄存器
segreg	分段寄存器
imm	一般性质立即数
imm8	8 位立即数
imm16	16 位立即数
imm32	32 位立即数
mem	一般性质内存地址
mem8	8 位内存地址
mem16	16 位内存地址
mem32	32 位内存地址
mem64	64 位内存地址
mem80	80 位内存地址
r/m8	8 位寄存器或内存地址
r/m16	16 位寄存器或内存地址
r/m32	32 位寄存器或内存地址
r/m64	64 位寄存器或内存地址

AAA, AAS, AAM, AAD

AAA 为加法的 ASCII 调整，结果在 AL。

AAS 为减法的 ASCII 调整，结果在 AL。

AAM 为乘法的 ASCII 调整，结果在 AX。

AAD 为除法的 ASCII 调整，结果在 AX。

AAA	[8086]
AAS	[8086]
AAD	[8086]
AAD imm	[8086]
AAM	[8086]
AAM imm	[8086]

ADC

带进位的整数加法。将第二操作数与第一操作数相加，并加入进位标志值，结果存于第一操作数。

ADC r/m8, reg8 [8086]

ADC r/m16, reg16	[8086]
ADC r/m32, reg32	[386]
ADC reg8, r/m8	[8086]
ADC reg16, r/m16	[8086]
ADC reg32, r/m32	[386]
ADC r/m8, imm8	[8086]
ADC r/m16, imm16	[8086]
ADC r/m32, imm32	[386]
ADC r/m16, imm8	[8086]
ADC r/m32, imm8	[386]
ADC AL, imm8	[8086]
ADC AX, imm16	[8086]
ADC EAX, imm32	[386]

ADD

执行整数加法。将第二操作数与第一操作数相加，结果存于第一操作数。

ADD r/m8, reg8	[8086]
ADD r/m16, reg16	[8086]
ADD r/m32, reg32	[386]
ADD reg8, r/m8	[8086]
ADD reg16, r/m16	[8086]
ADD reg32, r/m32	[386]
ADD r/m8, imm8	[8086]
ADD r/m16, imm16	[8086]
ADD r/m32, imm32	[386]
ADD r/m16, imm8	[8086]
ADD r/m32, imm8	[386]
ADD AL, imm8	[8086]
ADD AX, imm16	[8086]
ADD EAX, imm32	[386]

AND

执行逻辑 AND 运算。将第二操作数的值与第一操作数做 AND 运算，结果存于第一操作数。

AND r/m8, reg8	[8086]
AND r/m16, reg16	[8086]
AND r/m32, reg32	[386]
AND reg8, r/m8	[8086]
AND reg16, r/m16	[8086]
AND reg32, r/m32	[386]
AND r/m8, imm8	[8086]
AND r/m16, imm16	[8086]
AND r/m32, imm32	[386]

ADC r/m16, reg16	[8086]
ADC r/m32, reg32	[386]
ADC reg8, r/m8	[8086]
ADC reg16, r/m16	[8086]
ADC reg32, r/m32	[386]
ADC r/m8, imm8	[8086]
ADC r/m16, imm16	[8086]
ADC r/m32, imm32	[386]
ADC r/m16, imm8	[8086]
ADC r/m32, imm8	[386]
ADC AL, imm8	[8086]
ADC AX, imm16	[8086]
ADC EAX, imm32	[386]

ADD

执行整数加法。将第二操作数与第一操作数相加，结果存于第一操作数。

ADD r/m8, reg8	[8086]
ADD r/m16, reg16	[8086]
ADD r/m32, reg32	[386]
ADD reg8, r/m8	[8086]
ADD reg16, r/m16	[8086]
ADD reg32, r/m32	[386]
ADD r/m8, imm8	[8086]
ADD r/m16, imm16	[8086]
ADD r/m32, imm32	[386]
ADD r/m16, imm8	[8086]
ADD r/m32, imm8	[386]
ADD AL, imm8	[8086]
ADD AX, imm16	[8086]
ADD EAX, imm32	[386]

AND

执行逻辑 AND 运算。将第二操作数的值与第一操作数做 AND 运算，结果存于第一操作数。

AND r/m8, reg8	[8086]
AND r/m16, reg16	[8086]
AND r/m32, reg32	[386]
AND reg8, r/m8	[8086]
AND reg16, r/m16	[8086]
AND reg32, r/m32	[386]
AND r/m8, imm8	[8086]
AND r/m16, imm16	[8086]
AND r/m32, imm32	[386]

```

CMP r/m32, imm32    [386]
CMP r/m16, imm8     [8086]
CMP r/m32, imm8     [386]
CMP AL, imm8        [8086]
CMP AX, imm16       [8086]
CMP EAX, imm32      [386]

```

CMPSB, CMPSW, CMPSD

比较两个字节、字、双字，其结果存于标志之中。

```

CMPSB    [8086]
CMPSW    [8086]
CMPSD    [386]

```

DAA, DAS

DAA 为 BCD 加法的十进制调整，结果存于 AL 中。

DAS 为 BCD 减法的十进制调整，结果存于 AL 中。

```

DAA      [8086]
DAS      [8086]

```

DEC

操作数值减一。

```

DEC reg16    [8086]
DEC reg32    [386]
DEC r/m8     [8086]
DEC r/m16    [8086]
DEC r/m32    [386]

```

DIV

除以操作数的值。

操作数为 r/m8 时被除数为 AX，商为 AL，余数为 AH。

操作数为 r/m16 时被除数为 DX:AX，商为 AX，余数为 DX。

操作数为 r/m32 时被除数为 EDX:EAX，商为 EAX，余数为 EDX。

```

DIV r/m8     [8086]
DIV r/m16    [8086]
DIV r/m32    [386]

```

F2XM1

计算 2 的 x 次方后再减去 1。x 为 TOS 值，计算结果仍存回 TOS。

```

F2XM1    [8086, FPU]

```

FABS

取 TOS 值的绝对值后仍存回 TOS。

```

FABS     [8086, FPU]

```



FADD, FADDP

将指定操作数的值加入 TOS, 指令后面跟着 P, 表示要将 TOS 值弹出。

```
FADD mem32          [8086,FPU]
FADD mem64          [8086,FPU]
FADD fpureg         [8086,FPU]
FADD ST0, fpureg    [8086,FPU]
FADD TO fpureg      [8086,FPU]
FADD fpureg, ST0    [8086,FPU]
FADDP fpureg        [8086,FPU]
FADDP fpureg, ST0   [8086,FPU]
```

FBLD, FBSTP

FBLD 将操作数地址的聚集十进制数压入 TOS, 指令 FBST 将 TOS 弹出至操作数地址。

```
FBLD mem80          [8086,FPU]
FBSTP mem80         [8086,FPU]
```

FCHS

TOS 值改变符号。

```
FCHS                [8086,FPU]
```

FCLEX

清除浮点数闲置的例外。

```
FCLEX               [8086,FPU]
FNCLEX              [8086,FPU]
```

FCOM, FCOMP, FCOMPP

TOS 与指定操作数比较, 指令后面跟着两个 P, 表示要将浮点堆栈弹出两次。比较的结果设定浮点状态字 (FPU status word) 中的 C0 及 C3 的值。

TOS 较小则 C0=1、C3=0,

相等则 C0=0、C3=1,

TOS 较大则 C0=0、C3=0。

若 TOS 为非数值时无法比较, 则 C0=1、C3=1。

```
FCOM mem32          [8086,FPU]
FCOM mem64          [8086,FPU]
FCOM fpureg         [8086,FPU]
FCOM ST0, fpureg    [8086,FPU]
FCOMP mem32         [8086,FPU]
FCOMP mem64         [8086,FPU]
FCOMP fpureg        [8086,FPU]
FCOMP ST0, fpureg   [8086,FPU]
FCOMPP              [8086,FPU]
```

FCOS, FSIN

求 TOS 弧度 (radian) 的余弦及正弦值, 结果仍存回 TOS。

FCOS [386, FPU]

FSIN [386, FPU]

FDECSTP

浮点堆栈指针减一。

FDECSTP [8086, FPU]

FxDISI, FxENI

浮点数中断禁止/允许。

FDISI [8086, FPU]

FNDISI [8086, FPU]

FENI [8086, FPU]

FNENI [8086, FPU]

FDIV

将 TOS 除以指定操作数, 结果仍存回 TOS。

FDIV mem32 [8086, FPU]

FDIV mem64 [8086, FPU]

FDIV fpureg [8086, FPU]

FDIV ST0, fpureg [8086, FPU]

FDIV TO fpureg [8086, FPU]

FDIV fpureg, ST0 [8086, FPU]

FDIVR mem32 [8086, FPU]

FDIVR mem64 [8086, FPU]

FDIVR fpureg [8086, FPU]

FDIVR ST0, fpureg [8086, FPU]

FDIVR TO fpureg [8086, FPU]

FDIVR fpureg, ST0 [8086, FPU]

FDIVP fpureg [8086, FPU]

FDIVP fpureg, ST0 [8086, FPU]

FDIVRP fpureg [8086, FPU]

FDIVRP fpureg, ST0 [8086, FPU]

FFREE

清除浮点堆栈所有寄存器, 内含为空白。

FFREE fpureg [8086, FPU]

FIADD

将指定操作数的整数值加入 TOS。

FIADD mem16 [8086, FPU]

FIADD mem32 [8086, FPU]



FICOM, FICOMP

TOS 与指定整数操作数比较, 比较的结果设定浮点状态字中的 C0 及 C3 字节值。

TOS 较小则 C0=1、C3=0;

TOS 相等则 C0=0、C3=1;

TOS 较大则 C0=0、C3=0;

若 TOS 为非数值时无法比较, 则 C0=1、C3=1。

FICOM	mem16	[8086,FPU]
FICOM	mem32	[8086,FPU]
FICOMP	mem16	[8086,FPU]
FICOMP	mem32	[8086,FPU]

FIDIV, FIDIVR

将 TOS 除以指定整数操作数, 结果仍存回 TOS。

FIDIV	mem16	[8086,FPU]
FIDIV	mem32	[8086,FPU]
FIDIVR	mem16	[8086,FPU]
FIDIVR	mem32	[8086,FPU]

FILD, FIST, FISTP

指令 FILD 将指定整数操作数压入浮点堆栈顶端。

指令 FIST 将 TOS 压入指定整数操作数。

指令 FISTP 将 TOS 弹出至指定整数操作数。

FILD	mem16	[8086,FPU]
FILD	mem32	[8086,FPU]
FILD	mem64	[8086,FPU]
FIST	mem16	[8086,FPU]
FIST	mem32	[8086,FPU]
FISTP	mem16	[8086,FPU]
FISTP	mem32	[8086,FPU]
FISTP	mem64	[8086,FPU]

FIMUL

将 TOS 乘以指定整数操作数, 结果仍存回 TOS。

FIMUL	mem16	[8086,FPU]
FIMUL	mem32	[8086,FPU]

FINCSTP

浮点堆栈指针加一。

FINCSTP	[8086,FPU]
---------	------------

FINIT, FNINIT

浮点堆栈初始化, 堆栈顶端指针指向 ST0, 所有浮点寄存器内含均为空白。

FINIT	[8086,FPU]
-------	------------

FNINIT [8086, FPU]

FISUB

从 TOS 减去指定操作数的整数值。

FISUB mem16 [8086, FPU]

FISUB mem32 [8086, FPU]

FISUBR mem16 [8086, FPU]

FISUBR mem32 [8086, FPU]

FLD

将指定操作数压入浮点堆栈顶端。

FLD mem32 [8086, FPU]

FLD mem64 [8086, FPU]

FLD mem80 [8086, FPU]

FLD fpureg [8086, FPU]

FLDxx

FLD1 压入常数 1。

FLDL2E 压入以 2 为底的自然对数 e 。

FLDL2T 压入以 2 为底的对数 10。

FLDLG2 压入以 10 为底的对数 2。

FLDLN2 压入以 e 为底的对数 2。

FLDPI 压入圆周率。

FLDZ 压入常数 0。

FLD1 [8086, FPU]

FLDL2E [8086, FPU]

FLDL2T [8086, FPU]

FLDLG2 [8086, FPU]

FLDLN2 [8086, FPU]

FLDPI [8086, FPU]

FLDZ [8086, FPU]

FLDCW

将操作数值存入浮点数控制字。

FLDCW mem16 [8086, FPU]

FLDENV

将内存变量值存入浮点环境变量中。

FLDENV mem [8086, FPU]

FMUL, FMULP

将 TOS 乘以指定操作数, 结果仍存回 TOS。

FMUL mem32 [8086, FPU]

FMUL mem64 [8086, FPU]



FMUL	fpureg	[8086,FPU]
FMUL	ST0, fpureg	[8086,FPU]
FMUL	TO fpureg	[8086,FPU]
FMUL	fpureg, ST0	[8086,FPU]
FMULP	fpureg	[8086,FPU]
FMULP	fpureg, ST0	[8086,FPU]

FNOP

不执行任何运算。

FNOP	[8086,FPU]
------	------------

FPATAN, FPTAN

返回浮点反正切及正切三角函数值。

FPATAN	[8086,FPU]
FPTAN	[8086,FPU]

FPREM

返回 ST0 除以 ST1 的余数。

FPREM	[8086,FPU]
FPREM1	[386,FPU]

FRNDINT

ST0 四舍五入成整数后又存回 ST0。

FRNDINT	[8086,FPU]
---------	------------

FSAVE, FRSTOR

将整个浮点数状况存入内存变量中。

FSAVE	mem	[8036,FPU]
FNSAVE	mem	[8086,FPU]
FRSTOR	mem	[8086,FPU]

FSCALE

将 ST1 四舍五入成整数后, ST0 乘上 2 的 ST1 整数次方。

FSCALE	[8086,FPU]
--------	------------

FSQRT

求 TOS 平方根, 结果仍存回 TOS。

FSQRT	[8086,FPU]
-------	------------

FST, FSTP

FST 将 TOS 压入指定操作数。FSTP 将 TOS 弹出至指定操作数。

FST	mem32	[8086,FPU]
FST	mem64	[8086,FPU]
FST	fpureg	[8086,FPU]
FSTP	mem32	[8086,FPU]

FSTP	mem64	[8086, FPU]
FSTP	mem80	[8086, FPU]
FSTP	fpureg	[8086, FPU]

FSTCW

将 FPU 状态字存入指定操作数或 AX。

FSTCW	mem16	[8086, FPU]
FNSTCW	mem16	[8086, FPU]

FSTENV

将浮点环境变量值存入内存 mem 地址。

FSTENV	mem	[8086, FPU]
FNSTENV	mem	[8086, FPU]

FSTSW

将浮点状态字值存入内存 mem16 地址。

FSTSW	mem16	[8086, FPU]
FSTSW	AX	[286, FPU]
FNSTSW	mem16	[8086, FPU]
FNSTSW	AX	[286, FPU]

FSUB, FSUBP, FSUBR, FSUBRP

从 TOS 减去指定操作数的值。

FSUB	mem32	[8086, FPU]
FSUB	mem64	[8086, FPU]
FSUB	fpureg	[8086, FPU]
FSUB	ST0, fpureg	[8086, FPU]
FSUB	TO fpureg	[8086, FPU]
FSUB	fpureg, ST0	[8086, FPU]
FSUBR	mem32	[8086, FPU]
FSUBR	mem64	[8086, FPU]
FSUBR	fpureg	[8086, FPU]
FSUBR	ST0, fpureg	[8086, FPU]
FSUBR	TO fpureg	[8086, FPU]
FSUBR	fpureg, ST0	[8086, FPU]
FSUBP	fpureg	[8086, FPU]
FSUBP	fpureg, ST0	[8086, FPU]
FSUBRP	fpureg	[8086, FPU]
FSUBRP	fpureg, ST0	[8086, FPU]

FTST

TOS 与 0 比较, 设定浮点状态字中的 C0 及 C3 位值。TOS 较小则 C0=1、C3=0, 相等则 C0=0、C3=1, TOS 较大则 C0=0、C3=0。若 TOS 为非数值时无法比较, 则 C0=1、C3=1。

FTST	[8086, FPU]
------	-------------



FXAM

检查 ST0 的类型。

FXAM [8086, FPU]

FXCH

TOS 与指定操作数值互换。

FXCH [8086, FPU]

FXCH fpureg [8086, FPU]

FXCH fpureg, ST0 [8086, FPU]

FXCH ST0, fpureg [8086, FPU]

FXTRACT

将 ST0 值分开为指数及有效数字。指数存回 ST0, 有效数 r 存入 ST1。

FXTRACT [8086, FPU]

FYL2X, FYL2XP1

FYL2X 将 ST1 乘上以 2 为底的 ST0 对数值后存回 ST1。

FYL2X1 将 ST1 乘上以 2 为底的 ST0+1 对数值后存回 ST1。

FYL2X [8086, FPU]

FYL2XP1 [8086, FPU]

HLT

处理器暂停执行 C。

HLT [8086, FPU]

IDIV

有符号整数除法。操作数为 r/m8 时被除数为 AX, 商为 AL, 余数为 AH。

操作数为 r/m16 时被除数为 DX:AX, 商为 AX, 余数为 DX。

操作数为 r/m32 时被除数为 EDX:EAX, 商为 EAX, 余数为 EDX。

IDIV r/m8 [8086]

IDIV r/m16 [8086]

IDIV r/m32 [386]

IMUL

有符号整数乘法。操作数为 r/m8 时被乘数为 AL, 乘积为 AX。

操作数为 r/m16 时被乘数为 AX, 乘积为 DX:AX。

操作数为 r/m32 时被乘数为 EAX, 乘积为 EDX:EAX。

IMUL r/m8 [8086]

IMUL r/m16 [8086]

IMUL r/m32 [386]

IMUL reg16, r/m16 [386]

IMUL reg32, r/m32 [386]

IMUL reg16, imm8 [286]

```

IMUL reg16, imm16      [286]
IMUL reg32, imm8        [386]
IMUL reg32, imm32       [386]
IMUL reg16, r/m16, imm8 [286]
IMUL reg16, r/m16, imm16 [286]
IMUL reg32, r/m32, imm8 [386]
IMUL reg32, r/m32, imm32 [386]

```

IN

从输入端口输入至操作数。

```

IN AL, imm8      [8086]
IN AX, imm8      [8086]
IN EAX, imm8     [386]
IN AL, DX        [8086]
IN AX, DX        [8086]
IN EAX, DX       [386]

```

INC

操作数值加一。

```

INC reg16      [8086]
INC reg32      [386]
INC r/m8       [8086]
INC r/m16      [8086]
INC r/m32      [386]

```

INT

执行指定操作数的中断。

```
INT imm8 [8086]
```

INTO

溢出时执行 INT 4 的中断。

```
INTO      [8086]
```

IRET, IRETW, IRETD

从中断返回。

```

IRET      [8086]
IRETW     [8086]
IRETD     [386]

```

JCXZ, JECXZ

若 CX 或 ECX 值为 0 则转移至 imm。

```

JCXZ imm      [8086]
JECXZ imm     [386]

```



JMP

转移至指定的操作数。

```
JMP imm          [8086]
JMP SHORT imm     [8086]
JMP imm:imm16     [8086]
JMP imm:imm32     [386]
JMP FAR mem       [8086]
JMP r/m16         [8086]
JMP r/m32         [386]
```

Jcc

依条件 cc 值的真或假而转移至指定操作数。

```
Jcc imm          [8086]
Jcc NEAR imm     [386]
```

LAHF

将标志的低 8 位载入寄存器 AH 中。

```
LAHF             [8086]
```

LDS, LES, LSS

将指定的寄存器及内存偏移值载入指定的第一操作数寄存器,将内存下一个地址载入 DS 或 ES 或 SS 中。

```
LDS reg16, mem    [8086]
LDS reg32, mem    [386]
LES reg16, mem    [8086]
LES reg32, mem    [386]
LSS reg16, mem    [8086]
LSS reg32, mem    [386]
```

LEA

将内存 mem 偏移值载入第一操作数寄存器。

```
LEA reg16, mem    [8086]
LEA reg32, mem    [386]
```

LODSB, LODSW, LODSD

从 DS:SI 或 DS:ESI 载入一个字节值至 AL, 一个字至 AX, 一个双字至 EAX。

```
LODSB           [8086]
LODSW           [8086]
LODSD           [386]
```

LOOP, LOOPE, LOOPNE, LOOPZ, LOOPNZ

LOOP 为循环指令。LOOPE 及 LOOPZ 为相等或零值时循环。LOOPNE 及 LOOPNZ 为不相等或不是零值时循环。

LOOP imm	[8086]
LOOP imm, CX	[8086]
LOOP imm, ECX	[386]
LOOPE imm	[8086]
LOOPE imm, CX	[8086]
LOOPE imm, ECX	[386]
LOOPZ imm	[8086]
LOOPZ imm, CX	[8086]
LOOPZ imm, ECX	[386]
LOOPNE imm	[8086]
LOOPNE imm, CX	[8086]
LOOPNE imm, ECX	[386]
LOOPNZ imm	[8086]
LOOPNZ imm, CX	[8086]
LOOPNZ imm, ECX	[386]

MOV

将第二操作数的内容传送至第一操作数。

MOV r/m8, reg8	[8086]
MOV r/m16, reg16	[8086]
MOV r/m32, reg32	[386]
MOV reg8, r/m8	[8086]
MOV reg16, r/m16	[8086]
MOV reg32, r/m32	[386]
MOV reg8, imm8	[8086]
MOV reg16, imm16	[8086]
MOV reg32, imm32	[386]
MOV r/m8, imm8	[8086]
MOV r/m16, imm16	[8086]
MOV r/m32, imm32	[386]
MOV AL, memoffs8	[8086]
MOV AX, memoffs16	[8086]
MOV EAX, memoffs32	[386]
MOV memoffs8, AL	[8086]
MOV memoffs16, AX	[8086]
MOV memoffs32, EAX	[386]
MOV r/m16, segreg	[8086]
MOV r/m32, segreg	[386]
MOV segreg, r/m16	[8086]
MOV segreg, r/m32	[386]

MOVSB, MOVSW, MOVSD

将 DS:SI 的内容传送至 ES:DI 地址。或将 DS:ESI 的内容传送至 ES:EDI 地址。



MOVSB [8086]
MOVSW [8086]
MOVSD [386]

MOVSX, MOVZX

S 表示符号 (sign) 扩展, Z 表示零 (zero) 扩展。将第二操作数数值扩展传送至第一操作数地址。

MOVSX reg16, r/m8 [386]
MOVSX reg32, r/m8 [386]
MOVSX reg32, r/m16 [386]
MOVZX reg16, r/m8 [386]
MOVZX reg32, r/m8 [386]
MOVZX reg32, r/m16 [386]

MUL

无符号数相乘。操作数为 r/m8 时被乘数为 AL, 乘积为 AX。

操作数为 r/m16 时被乘数为 AX, 乘积为 DX:AX。

操作数为 r/m32 时被乘数为 EAX, 乘积为 EDX:EAX。

MUL r/m8 [8086]
MUL r/m16 [8086]
MUL r/m32 [386]

NEG, NOT

NEG 指令取操作数值的补码。NOT 指令则执行逻辑 NOT 运算, 取操作数值的反码, 结果存于第一操作数。

NEG r/m8 [8086]
NEG r/m16 [8086]
NEG r/m32 [386]
NOT r/m8 [8086]
NOT r/m16 [8086]
NOT r/m32 [386]

NOP

不执行任何运算。

NOP [8086]

OR

执行逻辑 OR 运算。将第二操作数与第一操作数做 OR 运算, 结果存于第一操作数。

OR r/m8, reg8 [8086]
OR r/m16, reg16 [8086]
OR r/m32, reg32 [386]
OR reg8, r/m8 [8086]
OR reg16, r/m16 [8086]
OR reg32, r/m32 [386]

```

OR r/m8, imm8      [8086]
OR r/m16, imm16     [8086]
OR r/m32, imm32     [386]
OR r/m16, imm8      [8086]
OR r/m32, imm8      [386]
OR AL, imm8          [8086]
OR AX, imm16         [8086]
OR EAX, imm32        [386]

```

OUT

将第二个操作数的值输出至第一个操作数或 DX 值所指的输入输出端口。

```

OUT imm8, AL        [8086]
OUT imm8, AX        [8086]
OUT imm8, EAX       [386]
OUT DX, AL          [8086]
OUT DX, AX          [8086]
OUT DX, EAX         [386]

```

POP

从堆栈顶端弹出至操作数 C。

```

POP reg16           [8086]
POP reg32           [386]
POP r/m16           [8086]
POP r/m32           [386]
POP DS              [8086]
POP ES              [8086]
POP SS              [8086]
POP FS              [386]
POP GS              [386]

```

POPFx

从堆栈顶端弹出并存入标志寄存器。

```

POPF                [8086]
POPFW               [8086]
POPFD               [386]

```

PUSH

从指定的操作数压入至堆栈顶端。

```

PUSH reg16          [8086]
PUSH reg32          [386]
PUSH r/m16          [8086]
PUSH r/m32          [386]
PUSH CS             [8086]
PUSH DS             [8086]

```

PUSH ES	[8086]
PUSH SS	[8086]
PUSH FS	[386]
PUSH GS	[386]
PUSH imm8	[286]
PUSH imm16	[286]
PUSH imm32	[386]

PUSHAX

将所有一般性寄存器压入堆栈。PUSHAW 压入 AX、CX、DX、BX、SP、BP、SI、DI。

PUSHAD 压入 EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI。

PUSHA 若为 16 位处理器则相当于 PUSHAW，若为 32 位处理器则相当于 PUSHAD。

PUSHA	[186]
PUSHAD	[386]
PUSHAW	[186]

RCL, RCR

带进位标志的左移及右移。第二操作数表示移动的位数。

RCL r/m8, 1	[8086]
RCL r/m8, CL	[8086]
RCL r/m8, imm8	[286]
RCL r/m16, 1	[8086]
RCL r/m16, CL	[8086]
RCL r/m16, imm8	[286]
RCL r/m32, 1	[386]
RCL r/m32, CL	[386]
RCL r/m32, imm8	[386]
RCR r/m8, 1	[8086]
RCR r/m8, CL	[8086]
RCR r/m8, imm8	[286]
RCR r/m16, 1	[8086]
RCR r/m16, CL	[8086]
RCR r/m16, imm8	[286]
RCR r/m32, 1	[386]
RCR r/m32, CL	[386]
RCR r/m32, imm8	[386]

RET, RETF, RETN

从程序返回调用的程序。

RET	[8086]
RET imm16	[8086]
RETF	[8086]
RETF imm16	[8086]

RETN [8086]

RETN imm16 [8086]

ROL, ROR

左移及右移。第二操作数值表示移动的位数。

ROL r/m8, 1 [8086]

ROL r/m8, CL [8086]

ROL r/m8, imm8 [286]

ROL r/m16, 1 [8086]

ROL r/m16, CL [8086]

ROL r/m16, imm8 [286]

ROL r/m32, 1 [386]

ROL r/m32, CL [386]

ROL r/m32, imm8 [386]

ROR r/m8, 1 [8086]

ROR r/m8, CL [8086]

ROR r/m8, imm8 [286]

ROR r/m16, 1 [8086]

ROR r/m16, CL [8086]

ROR r/m16, imm8 [286]

ROR r/m32, 1 [386]

ROR r/m32, CL [386]

ROR r/m32, imm8 [386]

SAHF

将 AH 值存入标志寄存器的低 8 位。

SAHF [8086]

SAL, SAR

算术左移及右移。第二操作数值表示移动的位数。

SAL r/m8, 1 [8086]

SAL r/m8, CL [8086]

SAL r/m8, imm8 [286]

SAL r/m16, 1 [8086]

SAL r/m16, CL [8086]

SAL r/m16, imm8 [286]

SAL r/m32, 1 [386]

SAL r/m32, CL [386]

SAL r/m32, imm8 [386]

SAR r/m8, 1 [8086]

SAR r/m8, CL [8086]

SAR r/m8, imm8 [286]

SAR r/m16, 1 [8086]

SAR r/m16, CL [8086]

SAR r/m16, imm8 [286]
 SAR r/m32, 1 [386]
 SAR r/m32, CL [386]
 SAR r/m32, imm8 [386]

SBB

带借位的减法运算。

SBB r/m8, reg8 [8086]
 SBB r/m16, reg16 [8086]
 SBB r/m32, reg32 [386]
 SBB reg8, r/m8 [8086]
 SBB reg16, r/m16 [8086]
 SBB reg32, r/m32 [386]
 SBB r/m8, imm8 [8086]
 SBB r/m16, imm16 [8086]
 SBB r/m32, imm32 [386]
 SBB r/m16, imm8 [8086]
 SBB r/m32, imm8 [386]
 SBB AL, imm8 [8086]
 SBB AX, imm16 [8086]
 SBB EAX, imm32 [386]

SCASB, SCASW, SCASD

将 AL 或 AX 或 EAX 值与 ES:DI 或 ES:EDI 的内容比较。

SCASB [8086]
 SCASW [8086]
 SCASD [386]

SHL, SHR

左移及右移。第二操作数值表示移动的位数。

SHL r/m8, 1 [8086]
 SHL r/m8, CL [8086]
 SHL r/m8, imm8 [286]
 SHL r/m16, 1 [8086]
 SHL r/m16, CL [8086]
 SHL r/m16, imm8 [286]
 SHL r/m32, 1 [386]
 SHL r/m32, CL [386]
 SHL r/m32, imm8 [386]
 SHR r/m8, 1 [8086]
 SHR r/m8, CL [8086]
 SHR r/m8, imm8 [286]
 SHR r/m16, 1 [8086]
 SHR r/m16, CL [8086]

```
SHR r/m16, imm8    [286]
SHR r/m32, 1        [386]
SHR r/m32, CL        [386]
SHR r/m32, imm8     [386]
```

STC, STD, STI

STC 将 CF 设为 1。STD 将 DF 设为 1。STI 将 IF 设为 1。

```
STC    [8086]
STD    [8086]
STI    [8086]
```

STOSB, STOSW, STOSD

STOSB 将 AL 值存入 ES:DI 或 ES:EDI 所指地址。STOSW 将 AW 值存入 ES:DI 或 ES:EDI 所指地址。STOSD 将 EAX 值存入 ES:DI 或 ES:EDI 所指地址。

```
STOSB    [8086]
STOSW    [8086]
STOSD    [386]
```

SUB

整数减法运算。第一操作数值减去第二操作数值，结果存回第一操作数。

```
SUB r/m8, reg8    [8086]
SUB r/m16, reg16   [8086]
SUB r/m32, reg32   [386]
SUB reg8, r/m8     [8086]
SUB reg16, r/m16   [8086]
SUB reg32, r/m32   [386]
SUB r/m8, imm8     [8086]
SUB r/m16, imm16   [8086]
SUB r/m32, imm32   [386]
SUB r/m16, imm8    [8086]
SUB r/m32, imm8    [386]
SUB AL, imm8       [8086]
SUB AX, imm16      [8086]
SUB EAX, imm32     [386]
```

TEST

逻辑比较运算。与 AND 相同，不过没将结果存回第一操作数。

```
TEST r/m8, reg8    [8086]
TEST r/m16, reg16   [8086]
TEST r/m32, reg32   [386]
TEST r/m8, imm8     [8086]
TEST r/m16, imm16   [8086]
TEST r/m32, imm32   [386]
TEST AL, imm8       [8086]
```



```
TEST AX, imm16      [8086]
TEST EAX, imm32      [386]
```

WAIT

等候 80x87 完成运算。

```
WAIT      [8086,FPU]
```

XCHG

第一操作数与第二操作数的值互换。

```
XCHG reg8, r/m8      [8086]
XCHG reg16, r/m16     [8086]
XCHG reg32, r/m32     [386]
XCHG r/m8, reg8       [8086]
XCHG r/m16, reg16     [8086]
XCHG r/m32, reg32     [386]
XCHG AX, reg16        [8086]
XCHG EAX, reg32       [386]
XCHG reg16, AX        [8086]
XCHG reg32, EAX       [386]
```

XLATB

查表。将 AL 值在 DS:BX 所指数组当索引值，将其相对应的内容返回给 AL 寄存器。

```
XLATB     [8086]
```

XOR

执行逻辑 XOR 运算。将第二操作数的值与第一操作数做 XOR 运算，结果存于第一操作数。

```
XOR r/m8, reg8        [8086]
XOR r/m16, reg16      [8086]
XOR r/m32, reg32      [386]
XOR reg8, r/m8        [8086]
XOR reg16, r/m16      [8086]
XOR reg32, r/m32      [386]
XOR r/m8, imm8        [8086]
XOR r/m16, imm16      [8086]
XOR r/m32, imm32      [386]
XOR r/m16, imm8       [8086]
XOR r/m32, imm8       [386]
XOR AL, imm8          [8086]
XOR AX, imm16         [8086]
XOR EAX, imm32        [386]
```

新平知

PDG

【责任编辑】 苏 茜 吴秋淑
【封面设计】 孙天昭

21世纪高校计算机系列教程

- 计算机概论
- C/C++程序设计
- 数据结构
- JAVA程序设计
- 计算机网络教程
- C语言程序设计
- 网络概论
- 网页设计实用教程
- 操作系统
- C/C++程序设计习题集
- Visual FoxPro 6.0程序设计与应用
- Visual FoxPro 6.0上机指导与习题集
- Visual Basic 6.x程序设计
- ASP与网络数据库技术
- AutoCAD 入门与应用实例
- 计算机组装维护与维修教程
- 汇编语言程序设计

ISBN 7-113-05303-3



9 787113 053031 >

ISBN7-113-05303-3/TP·953

定价: 27.00 元



中国铁道出版社

CHINA RAILWAY PUBLISHING HOUSE

中国铁道出版社·计算机图书批销部

地址: 北京市宣武区右安门西街8号

邮编: 100054

网址: <http://www.tqbooks.net>

读者热线电话: (010)63560056 (010)63560058

销售服务电话: (010)63560067 (010)63560043



PDG