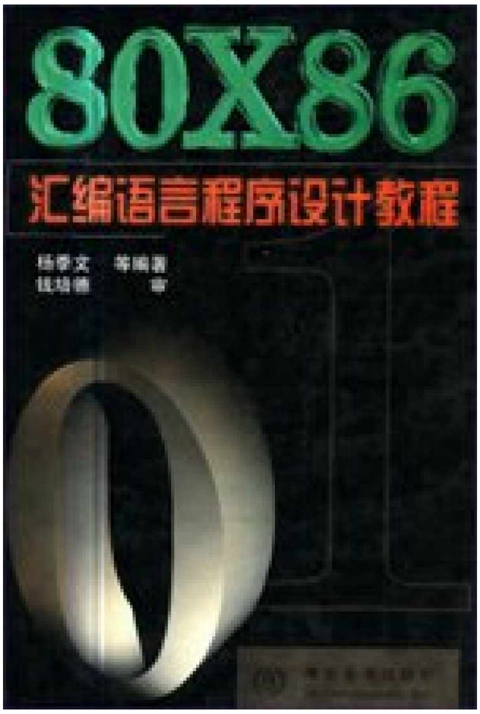

80X86
汇编语言程序设计教程
杨季文 等编著
钱培德 审

80x86汇编语言程序设计教程

杨季文等 编著
钱培德 审

清华大学出版社

(京)新登字 158 号

内 容 提 要

本书分为三部分。第一部分是基础部分，以 8086/8088 为背景，以 DOS 和 PC 兼容机为软硬件平台，以 MASM 和 TASM 为汇编器，介绍汇编语言的有关概念，讲解汇编语言程序设计技术。第二部分是提高部分，以 80386 为背景，以新一代微处理器 Pentium 为目标，细致和通俗地介绍了保护方式下的有关概念，系统和详细地讲解了保护方式下的编程技术，真实和生动地展示了保护方式下的编程细节。第三部分是上机实验指导。

本书的第一部分适合初学者，可作为学习汇编语言程序设计的教材。本书的第二部分适合已基本掌握 8086/8088 汇编语言的程序员，可作为学习保护方式编程技术的教材或参考书，也可作为其他人员了解高档微处理器和保护方式编程技术的参考书，还可作为程序员透彻地了解 Windows 程序设计技术的参考书。

图书在版编目(CIP)数据

80x86 汇编语言程序设计教程/杨季文等编著. —北京：清华大学出版社，1998.4
ISBN 7-302-02901-6

Ⅰ.80… Ⅱ.杨… Ⅲ.汇编语言-程序设计-教材 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(98)第 07169 号

出版：清华大学出版社(北京清华大学校内，邮编 100084)
因特网地址：www.tup.tsinghua.edu.cn
印刷：丰华印刷厂
发行：新华书店总店北京科技发行所
开本：787×1092 1/16 **印张**：38 **字数**：902 千字
版次：1998 年 6 月第 1 版 1998 年 6 月第 1 次印刷
书号：ISBN 7-302-02901-6/TP·1535
印数：0001～6000
定价：39.00 元

前　言

汇编语言面向机器，只有它能够为程序员提供最直接操纵机器硬件系统的途径，利用它可以编写出在“时间”和“空间”两个方面最具效率的程序。

“汇编语言程序设计”是计算机各专业的一门重要基础课程，是必修的核心课程之一，是“操作系统”和“微机原理与接口技术”等其他核心课程必要的先修课。该课程对于训练学生掌握程序设计技术，熟悉上机操作和程序调试技术都有重要作用。此外，“汇编语言程序设计”也是其他相关专业的必修或选修课。

目前，国内最广泛使用的 PC 系列机（包括兼容机），都以 Intel 的 80x86 系列微处理器或者兼容的微处理器为 CPU。在 Intel 的 80x86 家族中，16 位的 8086/8088 是基础，实现了以分段方式管理存储器；32 位的 80386 是高档微处理器的里程碑，实现了支持多任务的保护工作方式；基于 MMX 技术的 Pentium 是新一代的微处理器，实现了对多媒体处理的支持。本书以 8086/8088 为基础，以 80386 为重点，面向 Pentium 等新一代微处理器，讲解汇编语言程序设计的一般概念、基本技术和常用技巧，介绍宏和模块化程序设计的技术方法，讲解保护方式编程的相关概念、编程技术及实现细节。

本书分三个部分，共 12 章。第一部分是基础部分。第 1 章介绍汇编语言的特点和其他基本概念。第 2、第 3 章以 8086/8088 为背景，简要介绍 8086/8088 寻址方式、指令系统和汇编语言的常用伪指令语句后，讲解如何利用汇编语言实现程序的基本结构。第 4 章详细讲解了子程序的设计和如何调用 DOS 提供的子程序。第 5 章以 PC 及其兼容机为硬件平台，介绍输入/输出和中断等概念，讲解如何利用汇编语言编写 BIOS 程序和调用 BIOS 程序。第 6 章以 DOS 为软件平台，讲解如何利用汇编语言编写小型应用程序。第 7 章以 MASM 和 TASM 为汇编器，介绍宏和条件汇编等汇编语言的高级技术。第 8 章介绍模块化程序设计技术以及与高级语言的混合编程。第二部分是提高部分。第 9 章介绍实方式下的 80386 及其编程。第 10 章讲解保护方式下的 80386 及其编程，该章内容十分丰富。第 11 章介绍 80486 和 Pentium 程序设计基础。第三部分是上机实验指导，安排为第 12 章，应在上机实验前先阅读了解该章内容。

本书的第一部分适合初学者，可作为学习汇编语言程序设计的教材。本书的第二部分适合已基本掌握 8086/8088 汇编语言的程序员，可作为学习保护方式编程技术的教材或参考书，也可作为其他人员了解高档微处理器和保护方式编程技术细节的参考书，还可作为程序员透彻地了解 Windows 程序设计技术的参考书。

杨季文编写第 1 章、第 3 章、第 4 章至第 7 章、第 9 章至第 11 章，朱巧明编写第 2 章，吕强编写第 8 章，曹培培编写第 12 章。由杨季文最后统一定稿。

本书从初稿到定稿的全过程都始终得到了指导老师钱培德教授的热情关心和大力支持，承蒙他审阅了全书，特在此表示衷心感谢。本书在编著过程中得到了同事鲁征山、陈时

飚、李培峰和李廷彦等同志的帮助，还得到了赵雷和朱楠灏等同志的帮助，在此表示感谢。刘文杰和许晨等同志在讲课时使用过本书的初稿，并提出了宝贵意见，在此表示感谢。

书中不妥和谬误之处难免，恳请读者批评指正。

编 者

1998 年 2 月

目 录

第一部分 基础部分

第二部分 提高部分

第三部分 上机实验指导

第一部分　基础部分

第1章　绪　论

本章先介绍汇编语言的一些基本概念，然后介绍数据的表示和类型，最后简单介绍了Intel的x86家族历代微处理。

1.1　汇编语言概述

尽管在使用汇编语言进行程序设计之前，完全理解汇编语言的特点有困难，但了解汇编语言的特点对学习汇编语言程序设计是有益的。本节先说明汇编语言的内容，再介绍汇编语言的特点和使用汇编语言的场合。

1.1.1　汇编语言

1. 机器语言

CPU能直接识别并遵照执行的指令称为机器指令。机器指令在形式上表现为二进制编码。机器指令一般由操作码和操作数两部分构成，操作码在前，操作数在后。操作码指出要进行的操作或运算，如加、减、传送等。操作数指出参与操作或运算的对象，也指出操作或运算结果存放的位置，如CPU的寄存器、存储单元和数据等。

机器指令与CPU有着密切的关系。通常，CPU的种类不同，对应的机器指令也就不同。不同型号CPU的指令集往往有较大的差异。但同一个系列CPU的指令集常常具有良好的向上兼容性，也即下一代CPU的指令集是上一代CPU指令集的超集。例如，Intel 80386指令集包含了8086指令集。

机器语言是用二进制编码的机器指令的集合及一组使用机器指令的规则。它是CPU能直接识别的唯一语言。只有用机器语言描述的程序，CPU才能直接执行。用机器语言描述的程序称为目的程序或目标程序。

为了阅读和书写方便，常用十六进制形式或八进制形式表示二进制编码。例如，我们用Intel 8086指令写一个两数相加的程序片段。具体要求是把偏移2200H存储单元中的数与偏移2201H存储单元中的数相加，将它们的和送入偏移2202H存储单元。完成这一工作的程序片段包含三条机器指令，用十六进制形式表示如下：

```
A0 00 20
02 06 01 20
```

A2 02 20

几乎没有人能直接看出该程序片段的功能，原因是程序员难以掌握机器语言。因此，程序员难以用机器语言写程序，更难写出健壮的程序；用机器语言编制出的程序也不易为人们理解、记忆和交流。所以，只是在早期或不得已时才用机器语言写程序，现在几乎没有人用机器语言写程序了。机器语言有如下缺点：机器语言不能用人们熟悉的形式来描述计算机要执行的任务；用机器语言编写程序十分繁难，极易出错；一旦有错，也很难发现，也即调试困难。

2. 汇编语言

为了克服机器语言的上述缺点，人们采用便于记忆、并能描述指令功能的符号来表示指令的操作码。这些符号被称为指令助记符。助记符一般是说明指令功能的英语词汇或者词汇的缩写。同时也用符号表示操作数，如 CPU 的寄存器、存储单元地址等。

用指令助记符、地址符号等符号表示的指令称为汇编格式指令。

汇编语言是汇编格式指令、伪指令的集合及其表示、使用这些指令的一组规则。伪指令的概念留待以后介绍。用汇编语言书写的程序称为汇编语言程序，或称为汇编语言源程序，或简称为源程序。

利用汇编语言，上述两数相加的程序片段可表示如下：

```
MOV    AL,VAR1
ADD    AL,VAR2
MOV    VAR3,AL
```

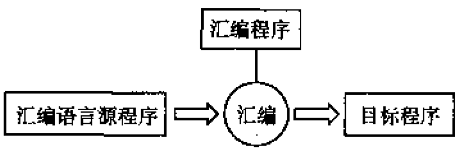


图 1.1 汇编过程示意图

显然，汇编格式指令比二进制编码的机器指令要容易掌握得多，用汇编语言编写的程序要比用机器语言编写的程序容易理解、调试和维护

3. 汇编程序

由于 CPU 能直接识别的唯一语言是机器语言，所以用汇编语言编写的源程序必须被翻译成用机器语言表示的目标程序后才能由 CPU 执行。把汇编语言源程序翻译成目标程序的过程称为汇编。完成汇编任务的程序叫做汇编程序。汇编过程如图 1.1 所示。

1.1.2 汇编语言的特点

由于汇编语言使用指令助记符和符号地址，所以它要比机器语言容易掌握得多。与高级语言相比较，汇编语言有如下特点。

1. 汇编语言与机器关系密切

因为汇编格式指令是机器指令的符号表示，所以汇编格式指令与机器有着密切的关系，因此汇编语言也与机器有着密切的关系，确切地说汇编语言与机器所带的 CPU 有着十分密切的关系。对于各种不同类型的 CPU，要使用各种不同的汇编语言。于是，对于各种不同类型的 CPU，也就有各种不同的汇编程序。

由于汇编语言与机器关系十分密切，所以汇编语言源程序与高级语言源程序相比，它

的通用性和可移植性要差得多。但通过汇编语言可最直接和最有效地控制机器，这常常是大多数高级语言难以做到的。

2. 汇编语言程序效率高

用汇编语言编写的源程序在汇编后所得的目标程序效率高。这种目标程序的高效率反映在时间和空间两个方面：其一是运行速度快；其二是目标程序短。在采用相同算法的前提下，任何高级语言程序在这两方面的效率都不如汇编语言程序，许多情况下更是远远不及。

汇编语言程序能获得“时空”高效率的主要原因是：构成汇编语言主体的汇编格式指令是机器指令的符号表示，每一条汇编格式指令都是所对应的某条机器指令的“化身”；另一个重要原因是汇编语言程序能直接并充分利用机器硬件系统的许多特性。高级语言程序在上述两点上要逊色得多。

3. 编写汇编语言源程序繁琐

编写汇编语言源程序要比编写高级语言源程序繁琐得多。汇编语言是面向机器的语言，高级语言是面向过程或面向目标、对象的语言。如下两点突出表现了汇编语言的这一特性：

作为机器指令符号化的每一条汇编格式指令所能完成的操作极为有限。例如，Z80 指令集中没有乘法指令，8086 指令集中没有能够同时完成两次算术运算的指令。

程序员在利用汇编语言编写程序时，必须考虑包括寄存器、存储单元和寻址方式在内的几乎所有细节问题。例如：指令执行对标志的影响，堆栈设置的位置等。在使用高级语言编写程序时，程序员不会遇到这些琐碎却重要的问题。

4. 汇编语言程序调试困难

调试汇编语言程序往往要比调试高级语言程序困难。汇编格式指令的功能有限和程序员要注意太多的细节问题是造成这种困难的两个客观原因；汇编语言提供给了程序员最大的“舞台”，而程序员往往为了追求“时空”上的高效率而不顾程序的结构，这是造成调试困难的主观原因。

1.1.3 恰当地使用汇编语言

1. 汇编语言的优缺点

为了恰当地使用汇编语言，我们先明确一下它的优缺点。

汇编语言的主要优点是利用它可能编写出在“时空”两个方面最有效率的程序。另外，通过它可最直接和最有效地操纵机器硬件系统。

汇编语言的主要缺点是它面向机器，与机器关系密切，它要求程序员比较熟悉机器硬件系统，要考虑许多细节问题，最终导致程序员编写程序繁琐；调试程序困难；维护、交流和移植程序更困难。

正是由于汇编语言与机器关系密切，才使汇编语言具有其他高级语言所不具备的上述优点和缺点。为了利用汇编语言的优点，必须付出相应的代价。但汇编语言的每一个优点常常闪耀出诱人的光芒，使人们勇敢地面对它的缺点。

2. 使用汇编语言的场合

根据汇编语言的优缺点，我们要恰当地使用汇编语言，即尽可能地“扬长避短”。是否利用汇编语言编写程序，要看具体的应用场合，要充分考虑到软件的开发时间和软件的质量等诸多方面的因素。我们认为下列应用场合，可考虑使用汇编语言编写程序。

(1) 对软件的执行时间或存储容量有较高要求的场合。例如：系统程序的关键核心，智能化仪器仪表的控制系统，实时控制系统等。

(2) 需要提高大型软件性能的场合。通常把大型软件中执行频率高的子程序(过程)用汇编语言编写，然后把它们与其他程序一起连接。

(3) 软件与硬件关系密切，软件要有直接和有效控制硬件的场合。如设备驱动程序等。

(4) 没有合适的高级语言的场合。

3. 适度地追求“时空”效率

在用汇编语言编写程序时，追求“时空”效率要适度。在编写汇编语言程序时，我们要尽量利用最恰当的指令，以便节约一个字节或节省几个机器周期。但时至今日，计算机硬件系统的整体性能已极大地提高，所以，除非不得已，不要为节约少量字节或机器周期而影响程序的结构性、健壮性和可读性等。要在确保汇编语言程序上述性能良好的前提下追求“时空”性能。

1.2 数据的表示和类型

熟悉数据在计算机内的表示形式是掌握汇编语言程序设计的关键之一。本节简单介绍数据的表示形式和类型。

计算机中存储信息的最小单位称为位，在绝大多数系统中它只能表示两种状态。这两种状态可分别代表 0 和 1。计算机系统内部采用二进制表示数值数据，也采用二进制编码表示非数值数据和指令，其主要原因就在于此。

1.2.1 数值数据的表示

所谓数值数据就是数。这里仅介绍定点整数的有关内容。

1. 数的二进制表示

尽管日常生活中大多采用十进制计数，但在计算机内，数却大多采用二进制表示。某个二进制数 $b_nb_{n-1}\cdots b_2b_1b_0$ 所表示的数值用十进制数来衡量时，可利用如下按权相加的方法计算得到：

$$b_n2^n+b_{n-1}2^{n-1}+\cdots+b_22^2+b_12^1+b_02^0$$

在书写时，为了与十进制数相区别，通常在二进制数后加一个字母 B。

2. 有符号数的补码表示

为了方便地表示负数和容易地实现减法操作，有符号数采用补码形式表示。所以，有符号数二进制表示的最高位是符号位，0 表示正数，1 表示负数。正数数值的补码形式用二进制表示。为得到一个负数数值的补码形式，方法可以是先得出该负数所对应正数的二进

制形式，然后使正数的每一个二进制位变反，最后再将变反的结果加 1。

3. 符号扩展

常常需要把一个 n 位二进制数扩展成 m 位二进制数(m>n)。当要扩展的数是无符号数时，只要在最高位前扩展(m－n)个 0。如果要扩展的数是有符号数，并且采用补码形式表示，那么，就要进行符号位的扩展。

例如，21 的 8 位二进制和 16 位的二进制补码如下：

00010101　　8 位

0000000000010101　　16 位

例如，－3 的 8 位二进制补码和 16 位二进制补码如下：

11111101　　8 位

1111111111111101　　16 位

4. 数值数据的表示范围

n 位二进制数能够表示的无符号整数的范围是：

$0 \leqslant I \leqslant 2^n - 1$

采用补码形式表示有符号数。那么 n 位二进制数能够表示的有符号整数的范围是：

$-2^{(n-1)} \leqslant I \leqslant +2^{(n-1)} - 1$

所以，如果 n 是 8，那么能够表示的无符号整数的范围是 0～255，能够表示的有符号整数的范围是－128～＋127；如果 n 是 16，那么能够表示的无符号整数的范围是 0～65535，能够表示的有符号整数的范围是－32768～＋32767。

5. BCD 码

虽然二进制数实现容易，并且二进制运算规律简单，但不符合人们的使用习惯，书写阅读都不方便。所以在计算机输入输出时通常还是采用十进制来表示数，这就需要实现十进制与二进制间的转换。为了转换方便，常采用二进制编码的十进制，简称为 BCD 码(Binary Coded Decimal)。

BCD 码就是用 4 位二进制数编码表示 1 位十进制数。表示的方法可有多种，常用的是 8421 BCD 码，它的表示规则如表 1.1 所示。从表 1.1 可见，8421 BCD 码最自然和最简单。例如，十进制数 1996 用 8421 BCD 码表示成 0001 1001 1001 0110，每组 4 位二进制数之间是二进制的，但组与组之间是十进制的。和十进制数 1996 等值的二进制数是 11111001100。

表 1.1　十进制数字的 8421 BCD 码

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

6. 十六进制表示

由于二进制数的基数太小，所以书写和阅读都不够方便。而十六进制数的基数 16 等于 2 的 4 次幂，于是二进制数与十六进制数之间能方便地转换，也即 4 位二进制数对应 1 位十六进制数，或者 1 位十六进制数对应 4 位二进制数。因此，人们常常把二进制数改写成十六进制数，在汇编语言程序设计过程中尤其如此。

在书写时，为了区别于十进制数和二进制数，通常在十六进制数后加一个字母 H。

1.2.2 非数值数据的表示

计算机除了处理数值数据外，还要处理大量的非数值数据，如文字信息和图表信息等，为此必须对非数值数据进行编码，这样不仅计算机能够方便地处理和存储它们，而且还可以赋予它们数值数据的某些属性。

1. ASCII 码

美国信息交换标准码（American Standard Code for Information Interchange），简称为 ASCII 码，是目前国际上比较通用的字符二进制编码，微型计算机中也普遍采用它作为字符的编码。它是 7 位二进制编码，表 1.2 列出了 ASCII 码。

表 1.2 ASCII 码表

高位 / 低位		000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
0000	0	NUL	DEL	SP	0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0001	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	'	<	L	\	l	|
1101	D	CR	GS	−	=	M	]	m	}
1110	E	SO	RS	.	>	N	∧	n	~
1111	F	SI	US	/	?	O	_	o	DEL

从表 1.2 可看到，它对 94 个常用的一般符号进行了编码，其中包括 26 个英文字母的大小写符号、10 个数字符号和 32 个其他符号。空格也作为一个符号，其编码是 20H，它界

于一般符号和 32 个控制符之间。所有这些一般符号和可控制符统称为字符。从表 1.2 还可看到，数字符号的编码、大写字母符号的编码和小写字母符号的编码分别是连续的，所以只要记住数字符号的编码从 30H 开始、大写字母符号的编码从 41H 开始和小写字母的编码从 61H 开始，那么就可推出其他数字符号和字母符号的编码。

由于 ASCII 码只使用了 7 位二进制进行编码，故最多表示 128 个字符。这往往不能满足使用要求。为此在 IBM PC 系列及其兼容机上，使用扩展的 ASCII 码。扩展的 ASCII 码使用 8 位二进制进行编码，故可表示 256 个字符。另外，在该扩展的 ASCII 码中，控制符所对应的编码同时也表示其他图形符号。请参见附录。

2. 变形国标码

有了 ASCII 码，计算机就能处理数字、字母等字符，但还不能处理汉字符。为了使计算机能够处理汉字信息，就必须对汉字进行编码。我国在 1981 年 5 月对六千多个常用汉字制定了交换码的国家标准，即 GB2312—80《信息交换用汉字编码字符集——基本集》。该标准规定了汉字信息交换用的基本汉字符和一般图形字符，它们共计 7445 个，其中汉字分成两级共计 6763 个。该标准同时也给定了它们的二进制编码，即国标码。

国标码是 16 位编码，高 8 位表示汉字符的区号，低 8 位表示汉字符的位号。实际上，为了给汉字符编码，该标准把代码表分成 94 个区，每个区有 94 个位。区号和位号都从 21H 开始。一级汉字安排在 30H 区至 57H 区，二级汉字安排在 58H 区至 77H 区。例如，"啊"字的国标码是 3021H。国标码为汉字的输入提供了一种标准输入方式。

目前在计算机中文平台中普遍采用的汉字编码是变形国标码。变形国标码是 16 位二进制编码，顾名思义它是国标码的变形。用得最多的变形方法是把国标码的第 15 位和第 7 位均置成 1，由于国标码中第 15 位和第 7 位都是 0，所以这种变形方法实际上就是在国标码上加 8080H。

尽管 16 位的变形国标码与两个扩展的 ASCII 码的组合有冲突，但它在相关系统模块的支持下，有效地实现了汉字在计算机内的表示。

1.2.3 基本数据类型

计算机存取的以二进制位表示的信息位数一般是 8 的倍数，它们有专门的名称。

1. 字节

一个字节由 8 个二进制位组成。字节的最低位一般称为第 0 位，最高位称为第 7 位，如图 1.2 所示。

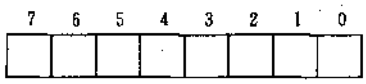


图 1.2 一个字节 8 个位

通常，硬件存储器的每一存储单元就由 8 个连续的位组成，也即可用于存储一个字节的信息。

如用一个字节来表示一个无符号数，那么表示范围是 0～255；如表示有符号数，则表示范围是 −128～+127。一个字节足以表示一个 ASCII 字符，也可以表示一个扩展的 ASCII 字符。

另外，一个字节可分成 2 个 4 位的位组，称为半字节。

2. 字

2 个字节(即 16 个二进制位)组成一个字，如图 1.3 所示。字的最低位称为第 0 位，最

高位称为第 15 位。字的低 8 位称为低字节，高 8 位称为高字节。由于一个字由 16 个二进制位组成，所以用一个字来表示无符号数，则表示范围是 0～65535；如表示有符号数，则表示范围是－32768～＋32767 间的有符号数。一个字还可表示一个变形国标码。

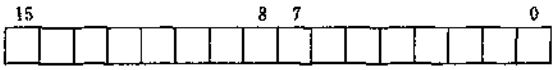


图 1.3　一个字含 16 个位

注意，有时候字是涉及处理器一次能够处理的信息量的一个术语，字长是衡量处理器品质的一个重要指标。

3. 双字

就和听起来一样，双字由 2 个字组成，也即包含 32 个二进制位。低 16 位称为低字，高 16 位称为高字。双字能表示的数的范围更大。

4. 四字

四字就是由四个字组成，包含 64 个二进制位。如果双字还不能表达所需要的数值精度，那么四字也许就能解决问题了。

5. 十字节

十字节就和它的名称一样，由 10 个字节组成，含 80 个二进制位。可用于存储非常大的数或表示较多的信息。

6. 字符串

字符串是指由字符构成的一个线性数组。通常每个字符用一个字节表示，但有时每个字符也可用一个字或一个双字来表示。

1.3　Intel 系列 CPU 简介

汇编语言与 CPU 关系密切。本节从汇编语言程序设计的角度对 Intel 的 80x86 系列 CPU 作一简单介绍。

1.3.1　8 位微处理器

1971 年 Intel 开发出了第一代微处理器 4004，它是一个 4 位的微处理器，自身含有计算和逻辑功能。它由 2250 个 MOS 晶体管构成，每秒内能够执行约 6 万次操作。含有一个累加器，16 个用作暂存数据的寄存器。可寻址 640 字节的内存。指令集含有 45 条指令。4004 作为一般处理器来讲，功能还不够强，只能作为计算器的核心来使用。但它是一种新思想的第一代产物。

1972 年 Intel 公司推出了第一块 8 位微处理器 8008。它由约 3300 个 MOS 晶体管构成。由于无论是指令执行的数据还是译码数据，以及操作数都能按 8 位处理，所以它比 4004 要快，每秒内执行的操作可超过 8 万次。它含有 7 个 8 位寄存器，可寻址 16K 的内存。具有 48 条指令组成的指令集，但与 4004 的指令集不兼容。

1974 年 Intel 公司又推出了为多种应用而设计的 8 位微处理器 8080，它是 Intel 的第

二代微处理器,也是第一个通用的微处理器。它的功能相当强,足以作为微计算机的核心。它由6000多个晶体管构成,每秒能执行约60万次操作。寻址能力达到64K。8080的指令集包含了8008的指令集,从而获得与8008指令集的兼容性,此外还增加了20多条指令。8080为Intel公司成为当今CPU的霸主打下了坚实的基础。

1976年Intel公司公布了8080的变种8080A,此后还公布了作为8080A增强型的8085。

1.3.2 16位微处理器

1978年Intel公司率先推出了第三代微处理器即16位微处理器8086。有两个关键的结构概念使微处理器设计定型且从8086开始施行,这两个概念是存储器分段和指令译码表。Intel的x86家族也由此开始。

1. Intel 8086

8086内部分成如图1.4所示的两部分:总线接口部件BIU(Bus Interface Unit)和执行部件EU(Execution Unit)。

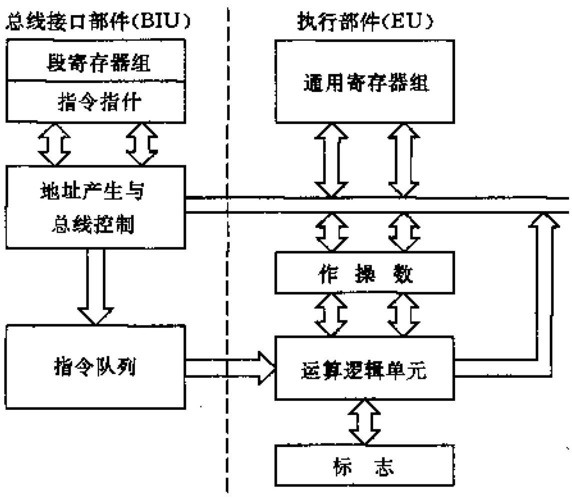


图1.4 8086的两个组成部分

总线接口部件BIU包括一组段寄存器、一个指令指示器、指令队列(长6个字节)、地址产生器和总线控制器等。BIU根据执行部件EU的请求,完成CPU与存储器或I/O设备之间的数据传送。在EU执行指令的过程中,BIU根据需要从存储器中预先取出一些指令,保存到指令队列中。如果EU执行一条转移指令,使程序发生转移,那么存放在指令队列中的预先取得的指令就不再有用,BIU会根据EU的指示从新的地址重新开始取指令。

执行部件EU包括一个算术逻辑单元(ALU)、一组通用寄存器和标志寄存器等,它们均是16位的。EU负责指令的执行,并进行算术逻辑运算等。EU从BIU中的指令队列中取得指令。当指令要求将数据存放到存储器或输出到外部设备,或者要从存储器或外部设备读取数据时,EU就向BIU发出请求,BIU根据EU发来的请求完成这些操作。

由于 EU 和 BIU 分开，8086 的取指令过程和执行指令的过程在很大程度上是重叠的，即两个部件是并行工作的。图 1.5 是执行顺序示意图。8086 的这种结构，大大减少了等待取指令所需的时间，提高了 CPU 的效率。与先前的 8080 和 8085 相比，8086 的这种结构一方面可以提高整体的执行速度，另一方面又降低了对与之相配的存储器的存取速度的要求。

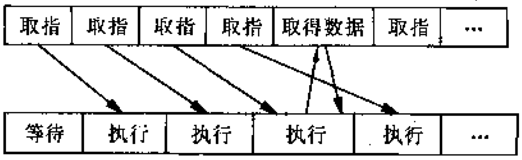


图 1.5　8086 指令的执行顺序

8086 的功能足够强。它具有 20 条地址线，故寻址范围可达到 1M。它具有 16 条数据线，能在一个总线周期内存取在偶地址开始的字操作数。它能执行整套 8080/8085 指令，并且还增加了包括乘除法指令在内的许多条新指令。由于处理速度的提高、运算能力的增强和内部部件的并行工作这三个方面的原因，使得它的处理能力大大地超过了 8 位微处理器。

2. Intel 8088

在 8086 推出之时，8 位机已使用了一段时间，许多价格合理的外部接口或设备都是 8 位结构，为了方便地与 8 位外部接口或设备相连，1979 年 Intel 公司又推出了 8088。8088 是 8086 的 8 位版，它具有与 8086 相同的内部结构，包括 EU 和 BIU 两部件、16 位的寄存器等。所不同的是 8088 对外只有 8 根数据线，总是按字节存取内存单元。8088 也称为准 16 位微处理器。

IBM PC 和 PC/XT 及其同档次的兼容机都采用 8088 作为 CPU。

从汇编语言程序设计的角度看，8088 与 8086 几乎没有什么区别。除了非常特别的程序外，适用于其中一个 CPU 的程序，可以不加修改地在另一个 CPU 上执行。

3. Intel 80186

1981 年 Intel 公司推出了 80186。除了 8086 所具有的特性外，80186 还集成若干通用系统所需的部件，包括一个片选逻辑部件，两个独立的高速直接存储器访问通道，三个可编程时钟，一个可编程中断控制器和一个时钟发生器等，这些部件使得 80186 功能更强。

80186 指令集包括了从早期的 8080 开始的所有指令，并且还增加了十余条新的指令，以改造现存的编码或产生最佳的 80186 编码。

从汇编语言程序设计的角度看，80186 只是比 8086 多了几条指令。

4. Intel 80286

在 1982 年 2 月 Intel 公司还推出了一种超级 16 位微处理器即 80286。它比 8086/8088 和 80186 在速度和性能上都有较大的提高。它具有 24 根地址线，可寻址的最大物理空间达 16M，它具有大批量数据处理、存储保护和多道程序处理能力，支持迫切需要的虚拟存储系统，因此它已可成为多任务、多用户系统的核心。

80286 有四个独立的处理部件，分别是执行部件 EU、总线部件 BU、指令部件 IU 和地址部件 AU。这四个部件能同时并行工作，与 8086 相比，80286 效率更高。

80286 可按两种模式运行：一种是实方式，另一种是保护方式。初始状态是实方式。

在实方式下，80286 的操作与 80186 极为相似，它提供的指令集包含了 80186 的指令集，此外还提供了几条特殊指令，它们是为实现从实方式转到保护方式服务的。总之，从汇编语言程序设计的角度看，除了多识别几条指令外，实模式下的 80286 相当于一个快速的 8086。

只有在保护模式下 80286 才能发挥其全部功能。实现寻址 16M 字节的物理地址空间，而在实方式下仍只能寻址最低端的 1M 字节空间。利用存储保护实现操作系统和任务的分离，在实方式下没有此能力。支持每个任务多达 1024M 字节的虚拟存储空间，同样在实方式下没有此能力。在保护模式下，80286 还提供多条仅供操作系统使用的特权指令。

IBM PC/AT 采用 80286 作为 CPU。

1.3.3 32 位微处理器

第四代微处理器是 32 位微处理器。Intel 80x86 家族的 32 位微处理器始于 80386。

1. Intel 80386

1985 年 10 月 Intel 公司推出了 32 位微处理器 80386。它不仅是微处理器发展进程中的里程碑，而且现在看来也是 80x86 家族中担负过“发扬光大”之重任的成员。

80386 兼容先前的 8086/8088、80186 和 80286。

80386 全面支持 32 位数据类型和 32 位操作。通用寄存器等从先前的 16 位扩展到 32 位。数据传送和算术逻辑运算等各种操作从先前的 8 或者 16 位扩展到 8、16 或者 32 位。80386 拥有 32 根数据线，存储器存取操作也从先前的 1 个或者 2 个连续字节(16 位)扩展到 1 个、2 个或者 4 个连续字节(32 位)。

80386 还增加了若干条包括位操作在内的新指令。这些新指令可使得某些任务更容易实现。

80386 支持实方式和保护方式两种运行模式。在实方式下，80386 相当于一个可进行 32 位处理的快速的 8086。只有在保护方式下，80386 才能真正发挥其全部强大的功能。

80386 支持 32 位物理地址，在保护方式下，可寻址的物理地址空间高达 4G。与 8086 和 80286 等相比，这个数字是巨大的。在此基础上，80386 有效地支持虚拟存储器和多任务。与 80286 相比，80386 在支持虚拟存储器时，还提供了可选的分页机制，这是很重要的改进。

80386 在保护方式下会支持称为虚拟 8086 方式的运行模式。虚拟 8086 方式可更有效地执行 8086/8088 代码。

所有这些功能，为我们进入 32 位时代做好了准备。实现这些功能的 80386 基本上由六个可并行处理的部件构成，如图 1.6 所示。

这些部件的并行处理对提高 80386 的效率是有益的。总线接口部件(Bus Interface Unit)为 80386 和它的环境之间提供接口。它接收来自执行部件或代码预取部件的存取传递请求，并按优先级选择这些请求。同时，它产生或完成当前总线周期的信号，这些信号包

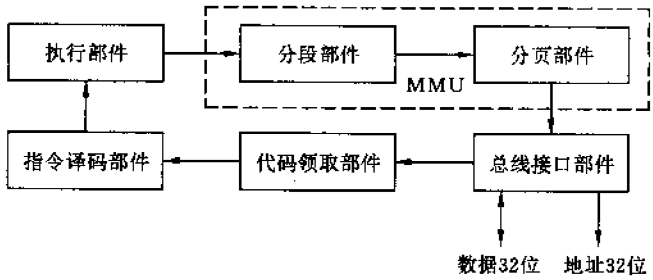


图 1.6 构成 80386 的各主要部件

括访问存储器和 I/O 的地址、数据和控制输出。代码预取部件(Code Prefetch Unit)执行预取代码功能。当 BIU 不占用总线周期来执行一条指令时,它指示 BIU 顺着指令字节流顺序提取代码。这些被预取的指令存放在 16 字节的预取代码队列中,等待指令译码部件的处理。指令译码部件(Instruction Decode Unit)从预取代码队列中取出指令并将它们转换成微代码。被译码的指令按 FIFO 方式存放在三个代码的指令队列中,等待执行部件的处理。执行部件(Execution Unit)执行指令队列中的指令,并与完成该指令要求的所有其他部件实现通信。它含有八个用于地址计算和数据操作的 32 位通用寄存器,还含有一个 64 位桶式移位器,用于加速移位、循环、乘法和除法操作。分段部件(Segment Unit)和分页部件(Paging Unit)构成存储器管理部件 MMU。分段部件把逻辑地址(虚拟地址)转换成线性地址,并有效地实现多种存储器保护措施。分页部件把线性地址转换成物理地址,并更有效地支持虚拟存储器的实现。分页是可选的,当不启用分页部件时,线性地址就直接作为物理地址。

2. Intel 80486

1989 年 4 月 Intel 公司推出了 80486,它是 80x86 家族中继 80386 之后又一种功能更强大的 32 位微处理器。它兼容先前的 8086/8088、80186、80286 和 80386。

简单地说,80486 是在微处理器 80386 的基础上集成数值协处理器 80387 和超高速缓存而构成的。图 1.7 是 80486 的结构示意图。图中的微处理器部分相当于 80386,但它采用"流水线"的方式执行指令,从而总体效益更好。所谓"流水线"方式,是指把指令处理

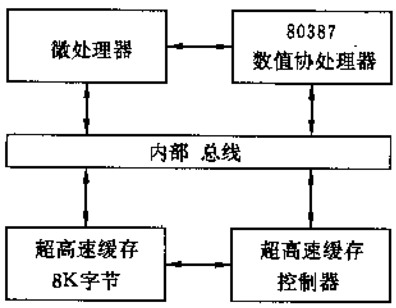


图 1.7 80486 的结构示意

分隔成若干个阶段，每个阶段都有独立的部件来处理，当一条指令的某个处理阶段完成后，它就进入到下一处理阶段，而独立的处理部件就可立即处理下一条指令。超高速缓存的容量为 8K 字节，利用它可在片上存储常用的数据和指令，以减少对外部总线的访问。数值协处理器是专为快速地进行浮点运算而设计的，称为浮点部件。集成在片内的浮点部件可更加有效地协助 80486 进行浮点数值运算。

1.3.4 Pentium 和 Pentium Pro

Intel 把其第五代微处理器命名为 Pentium(奔腾)，把其第六代微处理器命名为 Pentium Pro(高能奔腾)。

1. Pentium

1993 年 3 月 Intel 公司推出了接替 80486 的新一代微处理器 Pentium。它的性能比 80486 又有较大幅度的提高，但它兼容先前的 8086/8088、80186、80286、80386 和 80486。

Pentium 支持的数据总线位数达到 64 位；支持的物理地址位数是 32 位；内部寄存器仍是 32 位。Pentium 采用超标量体系结构，拥有两条“流水线”，称为“U”流水线和“V”流水线。“U”流水线和“V”流水线都能执行整数指令，“U”流水线还能执行浮点指令。这样可实现在每个时钟周期内最多可执行两条指令。Pentium 还开始支持动态分支预测。Pentium 内置的浮点部件是在 80486 浮点部件的基础上完全重新设计，运算性能大大提高。Pentium 有两个独立的超高速缓存，即一个指令超高速缓存和一个数据超高速缓存，容量分别是 8K 字节。所以，即使 Pentium 以与 80486 相同的频率工作，整数运算性能仍可提高一倍，浮点性能可提高 5 倍。

2. Pentium Pro

1995 年 11 月 Intel 公司推出了更新一代微处理器 Pentium Pro。它似乎是为进一步加快 32 位代码的运行而设计的，但它兼容先前的 8086/8088、80186、80286、80386、80486 和 Pentium。

Pentium Pro 支持的数据总线位数是 64 位；支持的物理地址位数达到 36 位；内部寄存器是 32 位。除了像 Pentium 那样具有两个独立的容量为 8K 字节的 L1 级缓存分别作为指令超高速缓存和数据超高速缓存外，Pentium Pro 还集成了一个 256K 或 512K 字节的 L2 级高速缓存。L2 高速缓存能以处理器的全速运行。

Intel 在 Pentium Pro 上实现了一种卓绝的设计：既是超标量的又是超流水线的，既能支持乱序执行又能支持寄存器重命名，既开发了分支预测又开发了推测执行。所有这些统称为“动态执行”。Pentium Pro 的超流水线技术扩展了原先的基本流水线概念，进一步划细了基本流水线的各处理阶段。超标量指有多条流水线，Pentium Pro 具有三发超标量模式，在每个时钟周期内最多可执行三条指令。乱序执行是指不必按程序中指定的顺序执行对应每一条指令的内部 RISC 型操作，这有利于提高流水线的效率。寄存器重命名是指把对体系结构寄存器的引用转换成对物理寄存器的引用，有助于减轻指令间的伪相关。分支预测是指对一个分支是否真正改变程序流而进行的猜测；推测执行是指允许提前执行那些由于分支原因导致不一定总被执行的指令，这些都有利于提高流水线的效率。上述“动态执行”使得 Pentium Pro 在大多数情况下处理指令比 Pentium 的效率更高。

尽管 Pentium Pro 的"动态执行"十分复杂，但对汇编语言程序员而言，Pentium Pro 与 Pentium 几乎无区别，相反它使得软件优化对于提高其性能显得并不太重要。

3. 发展趋势

表 1.3 列出了 Intel 的 80x86 家族前几代微处理器的性能，从中可见微处理器技术发展迅猛，每种新型号的速度都比前一种旧型号更快，因此能更好地满足最终用户各种各样的要求。

表 1.3 各种 Intel CPU 性能比较

	8086	286	386	486	Pentium	Pentium Pro
晶体管数（万只）	2.9	13.4	27.5	160	330	550（不计 L2）
工作频率（MHz）	4.7～10	6～12	16～33	20～100	60～200	150～200
寄存器位数	16	16	32	32	32	32
数据总线位数	16	16	32	32	64	64
地址总线位数	20	24	32	32	32	36
MIPS	<1	1～2.5	3～12	16～75	100～200	100～200

1997 年初推出的基于 MMX 技术的 Pentium 处理器使微处理器性能又上了一个台阶。MMX 是指多媒体扩展（MultiMedia eXtension），它是自 80386 出现以来 Intel 的 80x86 家族体系结构的最大的改进和增强。MMX 技术具有一套基本的、通用目的的整数指令，可以比较容易地满足各种多媒体应用程序及多媒体通信程序的需要。重点的技术包括单指令多数据技术（SIMD）、57 条新指令、8 个 64 位宽的 MMX 寄存器和 4 种新的数据类型。基于 MMX 技术的处理器具有足够的能力完成高速通信或带有多媒体任务的应用程序。MMX 技术使程序员可设计更多、更丰富、更令人惊奇的应用程序。

1.4 习 题

题 1.1 与机器语言相比，汇编语言有何特点？与高级语言相比，汇编语言有何特点？

题 1.2 汇编语言有何优缺点？

题 1.3 汇编程序的作用是什么？汇编程序与编译程序有何异同？

题 1.4 哪些场合需要使用汇编语言？

题 1.5 在计算机系统中，如何表示西文字符和汉字符？

题 1.6 什么是 BCD 码？

题 1.7 说明字节、字和双字之间的关系。

题 1.8 到目前为止，Intel 的 80x86 家族有哪些成员？这些成员有何特征？

第 2 章 8086/8088 寻址方式和指令系统

从汇编语言程序设计的角度看,8086/8088、80186 和实方式下的 80286 没有多大差异,而且 Intel 的 80x86 指令系统向上兼容。本章介绍 8086/8088 寻址方式和指令系统。

2.1 8086/8088 寄存器组

程序员可使用的 8086/8088 寄存器有通用寄存器、段寄存器和标志寄存器。这些寄存器有某些特定的用途,最常用的是通用寄存器。

2.1.1 8086/8088 CPU 寄存器组

8086/8088 包括四个 16 位数据寄存器,两个 16 位指针寄存器,两个 16 位变址寄存器,一个 16 位指令指针,四个 16 位段寄存器,一个 16 位标志寄存器。这 14 个 16 位寄存器分成四组,它们的名称和分组情况如图 2.1 所示。

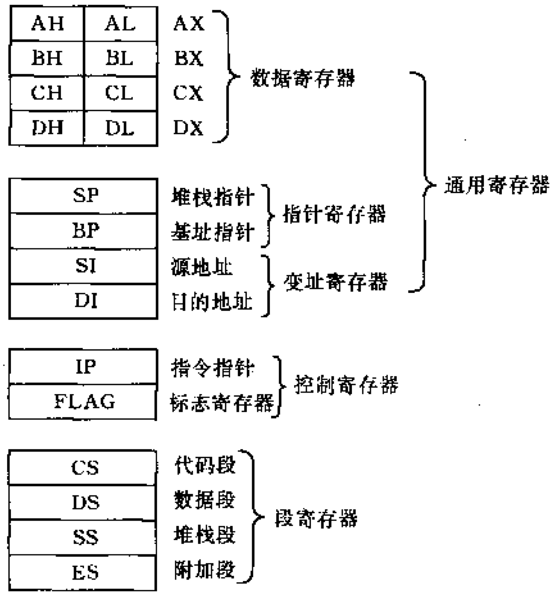


图 2.1 8086/8088 CPU 寄存器分组

1. 通用寄存器

数据寄存器、指针寄存器和变址寄存器统称为通用寄存器。这样称呼的理由是,这些寄存器除了各自规定的专门用途外,它们均可用于传送和暂存数据,可以保存算术逻辑运

算中的操作数和运算结果。

各通用寄存器的专门用途列于表 2.1 中。汇编语言程序员对这些用途必须充分注意，以便正确和合理地使用这些通用寄存器。

表 2.1 通用寄存器的专门用途

寄 存 器	用 途
AX	字乘法，字除法，字 I/O
AL	字节乘法，字节除法，字节 I/O，十进制算术运算
AH	字节乘法，字节除法
BX	存储器指针
CX	串操作或循环控制中的计数器
CL	移位计数器
DX	字乘法，字除法，间接 I/O
SI	存储器指针（串操作中的源指针）
DI	存储器指针（串操作中的目的指针）
BP	存储器指针（存取堆栈的指针）
SP	堆栈指针

（1）数据寄存器

数据寄存器主要用来保存操作数或运算结果等信息，它们的存在节省了为存取操作数所需占用总线和访问存储器的时间。

四个 16 位的数据寄存器可分解成八个独立的 8 位寄存器，这八个 8 位的寄存器有各自的名称，均可独立存取。如图 2.1 所示，AX 寄存器分解为 AH 寄存器和 AL 寄存器；BX 寄存器分解为 BH 寄存器和 BL 寄存器；CX 寄存器分解为 CH 寄存器和 CL 寄存器；DX 寄存器分解为 DH 寄存器和 DL 寄存器。名称中的字母 H 表示高，字母 L 表示低。AH 寄存器就是 AX 寄存器的高 8 位，AL 寄存器就是 AX 寄存器的低 8 位。AH 寄存器和 AL 寄存器的合并就是 AX 寄存器。其他寄存器类推。程序员在设计 8086/8088 程序时，要充分利用数据寄存器的上述双重性，恰当地进行合分，以便有效地处理字和字节信息。

AX 和 AL 寄存器又称为累加器（Accumulator）。一般通过累加器进行的操作所花的时间可能最少，此外累加器还有许多专门的用途，所以累加器使用得最普遍。

BX 寄存器称为基（Base）地址寄存器。它是四个数据寄存器中唯一可作为存储器指针使用的寄存器。

CX 寄存器称为计数（Count）寄存器。在字符串操作和循环操作时，用它来控制重复循环操作次数。在移位操作时，CL 寄存器用于保存移位的位数。

DX 寄存器称为数据（Data）寄存器。在进行 32 位的乘除法操作时，用它存放被除数的高 16 位或余数。它也用于存放 I/O 端口地址。

（2）变址和指针寄存器

变址和指针寄存器主要用于存放某个存储单元地址的偏移，或某组存储单元开始地

址的偏移,即作为存储器(短)指针使用。作为通用寄存器,它们也可以保存16位算术逻辑运算中的操作数和运算结果,有时运算结果就是需要的存储单元地址的偏移。注意,16位的变址寄存器和指针寄存器不能分解成8位寄存器使用。利用变址寄存器和指针寄存器不仅能够有效地缩短机器指令的长度,而且能够实现多种存储器操作数的寻址,从而方便地实现对多种类型数据的操作。

SI和DI寄存器称为变址寄存器。在字符串操作中,规定由SI给出源指针,由DI给出目的指针,所以SI也称为源变址(Source Index)寄存器,DI也称为目的变址(Destination Index)寄存器。当然,SI和DI也可作为一般存储器指针使用。

BP和SP寄存器称为指针寄存器。BP主要用于给出堆栈中数据区基址的偏移,从而方便地实现直接存取堆栈中的数据,所以BP也称为基指针(Base Pointer)寄存器。正常情况下,SP只作为堆栈指针(Stack Pointer)使用,即保存堆栈栈顶地址的偏移。堆栈是一片存储区域,我们以后再介绍堆栈操作和堆栈的作用。

2. 段寄存器

8086/8088 CPU依赖其内部的四个段寄存器实现寻址1M字节物理地址空间。8086/8088把1M字节地址空间分成若干逻辑段,当前使用段的段值存放在段寄存器中。由段值和段内偏移形成20位地址,在2.2节介绍形成20位地址的具体方法。

8086/8088 CPU的四个段寄存器均是16位的,分别称为代码段(Code Segment)寄存器CS,数据段(Data Segment)寄存器DS,堆栈段(Stack Segment)寄存器SS,附加段(Extra Segment)寄存器ES。由于8086/8088有这四个段寄存器,所以有四个当前使用段可直接存取,这四个当前段分别称为代码段、数据段、堆栈段和附加段。在2.2节介绍如何使用这四个段寄存器和访问这四个段。

3. 指令指针

8086/8088 CPU中的指令指针IP(Instruction Pointer)也是16位的,它类似于8080/8085中的程序计数器PC(Program Counter)。指令指针IP给出接着要执行的指令在代码段中的偏移。实际上接着要执行的指令已被预取到指令预取队列,除非发生转移。在理解IP的功能时,可不考虑指令预取队列。

2.1.2 标志寄存器

8086/8088 CPU中有一个16位的标志寄存器,包含了9个标志,主要用于反映处理器的状态和运算结果的某些特征。各标志在标志寄存器中的位置如下所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

有些指令的执行会影响部分标志,而有些指令的执行不会影响标志;反过来,有些指令的执行受某些标志的影响,有些指令的执行不受标志的影响。所以,程序员要充分注意指令与标志的关系。

9个标志可分成两组,第一组6个标志主要受加减运算和逻辑运算结果的影响,称为运算结果标志,第二组标志不受运算结果的影响,称为状态控制标志。

1. 运算结果标志

(1) 进位标志 CF(Carry Flag)

进位标志CF主要用于反映运算是否产生进位或借位。如果运算结果的最高位(字操作时的第15位或字节操作时的第7位)产生一个进位或借位,则CF被置1,否则CF被清0。在进行多字节数的加减运算时,要使用到该标志;在比较无符号数的大小时,要使用到该标志。

移位指令也把操作数的最高位或最低位移入CF。移位指令和CF的配合,可实现操作数之间的位传送。

CF也常作为子程序的出口参数之一。

8086/8088提供专门的改变CF值的指令。

(2) 零标志 ZF(Zero Flag)

零标志ZF用于反映运算结果是否为0。如果运算结果为0,则ZF被置1,否则ZF被清0。在判断运算结果是否为0时,要使用到该标志。

(3) 符号标志 SF(Sign Flag)

符号标志SF用于反映运算结果的符号位。SF与运算结果的最高位相同,如果运算结果的最高位为1,则SF被置1,否则SF被清0。由于在8086/8088系统中,有符号数采用补码的形式表示,所以SF反映了运算结果的符号。如果运算结果为正,则SF被清0,否则SF被置1。

(4) 溢出标志 OF(Overflow Flag)

溢出标志OF用于反映有符号数加减运算是否引起溢出。如运算结果超出了8位或16位有符号数的表示范围,即在字节运算时大于127或小于－128,在字运算时大于32767或小于－32768,称为溢出。如果溢出,则OF被置1,否则OF被清0。

要特别注意,溢出标志与进位标志是两个不同性质的标志,不能混淆。

(5) 奇偶标志 PF(Parity Flag)

奇偶标志PF用于反映运算结果中"1"的个数。如果"1"的个数为偶数,则OF被置1,否则OF被清0。利用PF可进行奇偶校验检查,或产生奇偶效验位。在串行通信中,为了提高传送的可靠性,常采用奇偶校验。

(6) 辅助进位标志 AF(Auxiliary Carry Flag)

在字节操作时,如发生低半字节向高半字节进位或借位;在字操作时,如发生低字节向高字节进位或借位,则辅助进位标志AF被置1,否则AF被清0。十进制算术运算调整指令自动根据该标志产生相应的调整动作。

2. 状态控制标志

状态控制标志控制处理器的操作,要通过专门的指令才能使状态控制标志发生变化。

(1) 方向标志 DF(Direction Flag)

方向标志决定着串操作指令执行时有关指针寄存器调整方向。当DF为1时,串操作指令按减方式改变有关的存储器指针值;当DF为0时,串操作指令按加方式改变有关的存储器指针值。8086/8088提供的专门用于设置方向标志DF的指令是STD,专门用于清DF的指令是CLD。

(2) 中断允许标志 IF(Interrupt－enable Flag)

中断允许标志决定着 CPU 是否响应外部可屏蔽中断请求。当 IF 为 1 时,CPU 能够响应外部的可屏蔽中断请求;当 IF 为 0 时,则不响应外部的可屏蔽中断请求。但此标志的状态对于外部的非屏蔽中断请求,或内部产生的中断不起作用。8086/8088 提供的专门用于设置中断允许标志 IF 的指令是 STI,专门用于清 IF 的指令是 CLI。

(3) 追踪标志 TF(Trap Flag)

当追踪标志 TF 被置 1 后,CPU 进入单步方式。所谓单步方式是指在一条指令执行后,产生一个单步中断。这主要用于程序的调试。8086/8088 没有专门设置和清除 TF 标志的指令,要通过其他方法设置或清除 TF。

2.2 存储器分段和地址的形成

从 8086 开始采用分段的方法管理存储器。只有充分理解存储器分段的概念和存储器逻辑地址和物理地址的关系,才能熟练地使用 8086/8088 汇编语言。

2.2.1 存储单元的地址和内容

在以 8086 或 8088 为 CPU 的系统中(如 IBM PC 兼容机),以字节为单位线性地组织存储器。为了标识和存取每一个存储单元,给每一个存储单元规定一个编号,也就是存储单元地址。存储单元地址用二进制数表示,从 0 开始,顺序地每次加 1。存储单元的地址是无符号数,n 位二进制数总共能够表示 2^n 个存储单元的地址。为了书写方便,存储单元地址常采用十六进制数表示。

一个存储单元中存放的信息称为该存储单元的内容。图2.2示意了存储器中部分存储单元存放信息的情况。从图 2.2 可看到,地址为 56780H 的字节存储单元中的内容是 34H,而地址为 56781H 的字节存储单元中的内容是 12H。

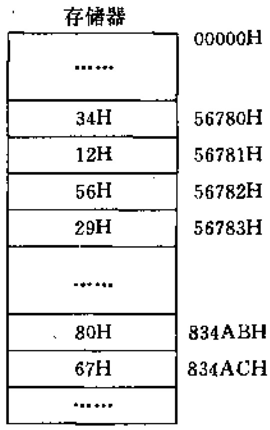


图 2.2 "高高低低"原则存储示意图

一个字存放到存储器要占用连续的两个字节单元。系统规定,当把一个字存放到存储器时,其低字节存放在地址较低的字节单元中,其高字节存放在地址较高的字节单元中。这样二个连续的字节单元就构成了一个字单元,字单元的地址采用它的低地址表示。例如,图 2.2 中地址为 56780H 的字单元的内容是 1234H,而地址为 834ABH 的字单元的内容是 6780H。

上述存储原则称为"高高低低"原则。在以字节方式存取字时需要特别注意该原则,当以字方式存取字时,处理器自动采用该原则。

四个连续的字节单元就构成了一个双字单元,双字单元的地址就是最低字节单元的地址。一个双字存放到存储器时也按照"高高低低"原则存储,也即高字在高地址字中,低字在低地址字中,也就是最高字节在最高地址字节单元中,最低字节在最低地址字节单元中。如图 2.2 所示,地址为 56780H 的双字单元中存放的内容是 29561234H。

2.2.2 存储器的分段

8086/8088 CPU 有 20 根地址线，可直接寻址的物理地址空间为 1M 字节($=2^{20}$)。系统存储器由以字节为单位的存储单元组成，存储单元的物理地址长 20 位，范围是 00000H 至 FFFFFH。尽管 8086/8088 内部的 ALU 每次最多进行 16 位运算，存放存储单元地址偏移的指针寄存器(如 P、SP 以及 BP、SI、DI 和 BX)都是 16 位，但 8086/8088 通过对存储器分段和使用段寄存器的方法有效地实现了寻址 1M 字节物理空间。

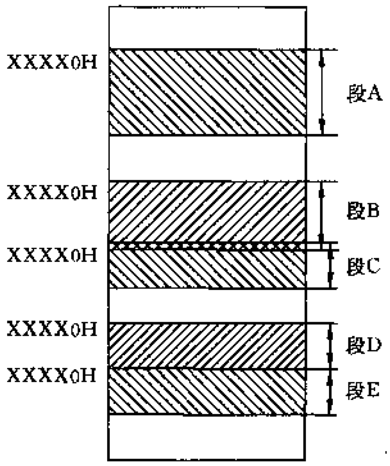


图 2.3 逻辑段的划分

根据需要把 1M 字节地址空间划分成若干逻辑段。每个逻辑段必须满足如下两个条件：第一，逻辑段的开始地址必须是 16 的倍数；第二，逻辑段的最大长度为 64K。按照这两个条件，1M 字节地址空间最多可划分成 64K 个逻辑段，最少也要划分成 16 个逻辑段。第一个条件与段寄存器长 16 位有关；第二个条件与指针寄存器长 16 位相关。

逻辑段与逻辑段可以相连，也可以不相连，还可以部分重叠。图 2.3 给出了若干逻辑段的划分情况。在图 2.3 中，段 B 与段 C 部分重叠，段 E 与段 D 相连。

这种存储器分段的方法不仅有利于实现寻址 1M 字节空间，而且也十分有利于对 1M 字节存储空间的管理。对实现程序的重定位和浮动，对实现代码数据的隔离，对充分利用存储空间，这种方法都有益。

2.2.3 物理地址的形成

由于段的起始地址必须是 16 的倍数，所以段起始地址有如下形式：

bbbbbbbbbbbbbbbb0000

用 16 进制可表示成 XXXX0。这种 20 位的段起始地址，可压缩表示成 16 位的 XXXX 形式。我们把 20 位段起始地址的高 16 位 XXXX 称为段值。显然，段起始地址等于段值乘 16 (即左移 4 位)。

要访问的某一个存储单元总是属于某个段。我们把存储单元的地址与所在段的起始地址的差称为段内偏移，简称为偏移。在一个段内，通过偏移可指定要访问的存储单元，或者说要访问的存储单元可由偏移来指定。在整个 1M 地址空间中，存储单元的物理地址等于段起始地址加上偏移。

于是，存储单元的逻辑地址由段值和偏移两部分组成，用如下形式表示：

段值：偏移

根据逻辑地址可方便地得到存储单元的物理地址，计算公式如下：

物理地址 ＝ 段值×16＋偏移

通过移位和算术加可容易地实现上述公式，图 2.4 是物理地址产生的示意图。例如，用 16 进制表示的逻辑地址 1234：3456H 所对应的存储单元的物理地址为 15796H。

由于段可以重叠，所以一个物理地址可用多个逻辑地址表示。图 2.5 是这样的一个例子，其中存储单元的物理地址是 12345H，标出的两个重叠段的段值分别是 1002H 和 1233H，在对应段内的偏移分别是 2325H 和 0015H。

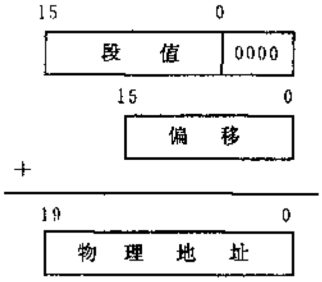


图 2.4 物理地址产生示意图

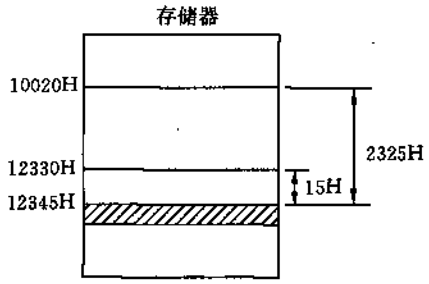


图 2.5 一个物理地址可对应多个逻辑地址

采用段值和偏移构成逻辑地址后，段值由段寄存器给出，偏移可由指令指针 IP、堆栈指针 SP 和其他可作为存储器指针使用的寄存器(SI、DI、BX 和 BP)给出，偏移还可直接用 16 位数给出。指令中不使用物理地址，而是使用逻辑地址，由总线接口单元 BIU 按需要根据段值和偏移自动形成 20 位物理地址。

2.2.4 段寄存器的引用

由于 8086/8088CPU 有四个段寄存器，可保存四个段值，所以可同时使用四个段，但这四个段有所分工。每当需要产生一个 20 位的物理地址时，BIU 会自动引用一个段寄存器，且左移 4 位再与一个 16 位的偏移相加。图 2.6 给出了一个同时使用四个段的例子。

在取指令的时候，自动引用代码段寄存器 CS，再加上由 IP 所给出的 16 位偏移，得到要取指令的物理地址。

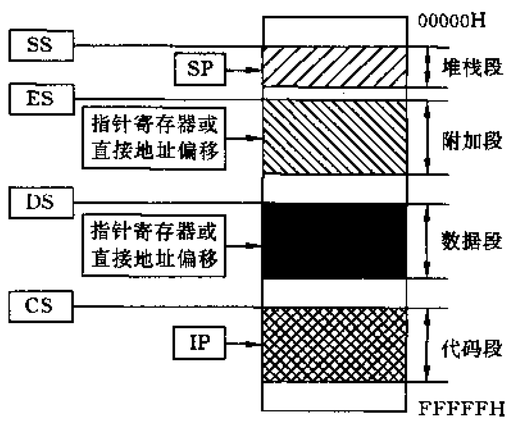


图 2.6 段和段寄存器的引用

当涉及到一个堆栈操作时，则自动引用堆栈段寄存器SS，再加上由SP所给出的16位偏移，得到堆栈操作所需的物理地址。当偏移涉及BP寄存器时，缺省引用的段寄存器也为堆栈段寄存器SS。

在存取一个普通存储器操作数时，则自动选择数据段寄存器DS或附加段寄存器ES，再加上16位偏移，得到存储器操作数的物理地址。此时的16位偏移，可以是包含在指令中的直接地址，也可以是某一个16位存储器指针寄存器的值，也可以是指令中的偏移再加上存储器指针寄存器中的值，这取决于指令的寻址方式。除了串操作时目的段选择附加段寄存器ES外，缺省选择数据段寄存器DS。

在不改变段寄存器值的情况下，寻址的最大范围是64K字节。若某个程序使用的总的存储长度(包括代码、堆栈和数据区)不超过64K字节，则整个程序可以合用一个64K字节的段。若有一个程序，它的代码长度、堆栈长度和数据区长度均不超过64K字节，则可在程序开始时分别给DS和SS等段寄存器赋值，在程序的其他地方就可不再考虑这些段寄存器所含的段值，程序就能正常地运行。假如某个程序的数据区长度超过64K字节，那么就要在两个或多个数据段中存取数据。如果出现这种情况，只要在从存取一个数据段改变到存取另一个数据段时，改变数据段寄存器内的段值就可以了。

由于BIU能根据需要自动选择段寄存器，所以通常情况下在指令中不指明所需要的段寄存器。取指令和堆栈操作所引用的段寄存器分别规定为CS和SS，是不可变的；串操作中目的段的段寄存器规定为ES也是不可变的。但是，在存取一般存储器操作数时，段寄存器可以不一定是DS；当偏移涉及BP寄存器时，段寄存器也不是非要为SS。8086/8088允许使用段超越前缀，改变上述两种情况下所使用的段寄存器，也即用段超越前缀直接明确指定引用的段寄存器。表2.2列出了段寄存器的引用规定，其中“可选用的段寄存器”栏就列出了可作为段超越前缀改变的段寄存器，另外，有效地址EA(Effective Address)就是指段内偏移。

表2.2 段寄存器的引用规定

访问存储器涉及的方式	正常使用的段寄存器	可选用的段寄存器	偏　移
取指令	CS	无	IP
堆栈操作	SS	无	SP
一般数据存取(下列情况除外)	DS	CS、ES、SS	有效地址
源数据串	DS	CS、ES、SS	SI
目的数据串	ES	无	DI
BP作为指针寄存器使用	SS	CS、DS、ES	有效地址

2.3 8086/8088的寻址方式

表示指令中操作数所在的方法称为寻址方式。8086/8088有七种基本的寻址方式：立即寻址、寄存器寻址、直接寻址、寄存器间接寻址、寄存器相对寻址、基址加变址寻址、相

对基址加变址寻址。

直接寻址、寄存器间接寻址、寄存器相对寻址、基址加变址寻址和相对基址加变址寻址，这五种寻址方式属于存储器寻址，用于说明操作数所在存储单元的地址。由于总线接口单元BIU能根据需要自动引用段寄存器得到段值，所以这五种方式也就是确定存放操作数的存储单元有效地址EA的方法。有效地址EA是一个16位的无符号数，在利用这五种方法计算有效地址时，所得的结果认为是一个无符号数。

除了这些基本的寻址方式外，还有固定寻址和I/O端口寻址等。

2.3.1 立即寻址方式

操作数就包含在指令中，它作为指令的一部分，跟在操作码后存放在代码段。这种操作数称为立即数。

立即数可以是8位，也可以是16位。如果立即数是16位，那么按“高高低低”原则存放，即高位字节在高地址存储单元，低位字节在低地址存储单元。

指令“MOV AX, 1234H”的存储和执行情况如图2.7所示。

这种寻址方式主要用于给寄存器或存储单元赋初值的场合。

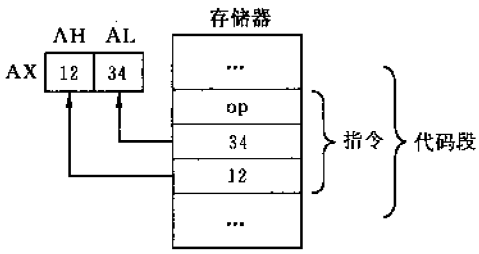


图2.7 立即寻址方式示意图

2.3.2 寄存器寻址方式

操作数在CPU内部的寄存器中，指令中指定寄存器号。对于16位操作数，寄存器可以是AX、BX、CX、DX、SI、DI、SP和BP等；对于8位操作数，寄存器可以是AL、AH、BL、BH、CL、CH、DL和DH。

例如，指令“MOV SI,AX”和指令“MOV AL,DH”中的源操作数和目的操作数均是寄存器寻址。再如，图2.7所示指令中，目的操作数采用寄存器寻址。

由于操作数在寄存器中，不需要通过访问存储器来取得操作数，所以采用这种寻址方式的指令执行速度较快。

2.3.3 直接寻址方式

操作数在存储器中，指令直接包含有操作数的有效地址。操作数一般存放在数据段，所以操作数的地址由DS加上指令中直接给出的16位偏移得到。如果采用段超越前缀，则操作数也可含在数据段外的其他段中。

设数据段寄存器 DS 的内容是 5000H，地址为 51234H 字存储单元中的内容是 6789H，那么在执行指令“MOV　AX，[1234H]”后寄存器 AX 的内容是 6789H。图 2.8 是此指令的存储和执行情况。为方便，本章常用(reg)表示寄存器 reg 的内容。于是该例的假设用(DS)＝5000H 表示，执行结果用(AX)＝6789H 表示。

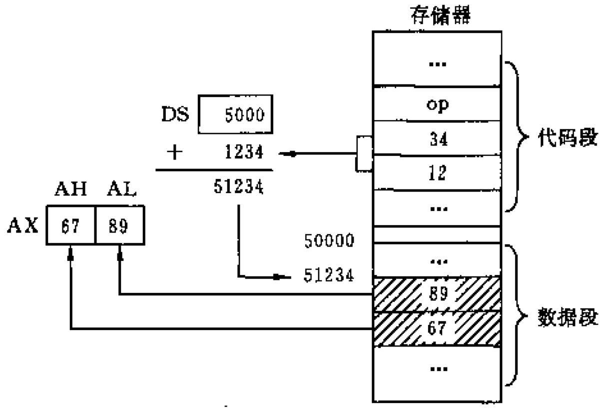


图 2.8　直接寻址方式示意图

下面指令中目标操作数采用直接寻址，并且使用了段超越前缀：

MOV　ES：[5678H]，BL　　　　　；引用的段寄存器是 ES

这种寻址方式常用于处理单个存储器变量的情况。它可实现在 64K 字节的段内寻找操作数。直接寻址的操作数通常是程序使用的变量。

注意立即寻址和直接寻址书写表示方法上的不同，直接寻址的地址要放在方括号中。在源程序中，往往用变量名表示。

2.3.4　寄存器间接寻址方式

操作数在存储器中，操作数有效地址在 SI、DI、BX、BP 这四个寄存器之一中。在一般情况(即不使用段超越前缀明确指定段寄存器)下，如果有效地址在 SI、DI 和 BX 中，则以 DS 段寄存器之内容为段值；如果有效地址在 BP 中，则以 SS 段寄存器之内容为段值。

例如，　MOV　AX，[SI]

假设，　(DS)＝5000H，(SI)＝1234H

那么，存取的物理存储单元地址是 51234H。再设该字存储单元的内容是 6789H，那么在执行该指令后，(AX)＝6789H。图 2.9 反映该指令的存储和执行情况。

下面指令中源操作数采用寄存器间接寻址，并且使用了段超越前缀：

MOV　DL，CS：[BX]　　　　　；引用的段寄存器是 CS

下面指令中目的操作数采用寄存器间接寻址，由于使用 BP 作为指针寄存器，所以缺省的段寄存器是 SS：

```
MOV   [BP],CX                      ;引用的段寄存器是 BP
```

这种寻址方式可以用于表格处理，在处理完表中的一项后，只要修改指针寄存器的内容就可以方便地处理表中的另一项。

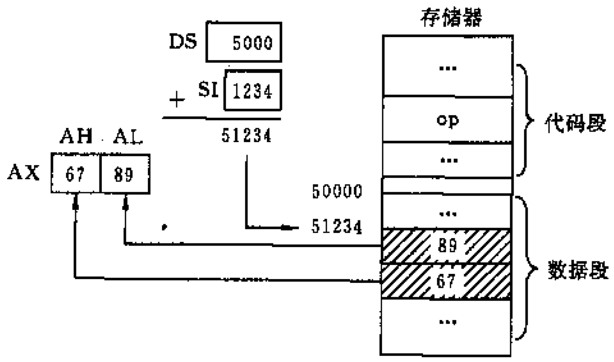


图 2.9 寄存器间接寻址方式示意图

请注意在书写表示寄存器间接寻址时，寄存器名一定要放在方括号中。下面两条指令的目的操作数的寻址方式完全不同：

```
MOV   [SI],AX             ;目的操作数寄存器间接寻址
MOV   SI,AX               ;目的操作数寄存器寻址
```

2.3.5 寄存器相对寻址方式

操作数在存储器中，操作数的有效地址是一个基址寄存器(BX、BP)或变址寄存器的(SI、DI)内容加上指令中给定的 8 位或 16 位位移量之和。即：

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

在一般情况(即不使用段超越前缀明确指定段寄存器)下，如果 SI、DI 或 BX 之内容作为有效地址的一部分，那么引用的段寄存器是 DS；如果 BP 之内容作为有效地址的一部分，那么引用的段寄存器是 SS。

在指令中给定的 8 位或 16 位位移量采用补码形式表示。在计算有效地址时，如位移量是 8 位，则被带符号扩展成 16 位。当所得的有效地址超过 FFFFH，则取其 64K 的模。

例如，MOV AX，[DI+1223H]
假设，(DS)=5000H，(DI)=3678H

那么，存取的物理存储单元地址是 5489BH。再设该字存储单元的内容是 55AAH，那么在执行该指令后，(AX)=55AAH。图 2.10 反映该指令的存储和执行情况。

下面指令中，源操作数采用寄存器相对寻址，引用的段寄存器是 SS：

MOV　BX,[BP−4]

下面指令中,目的操作数采用寄存器相对寻址,引用的段寄存器是 ES:

MOV　ES:[BX+5],AL

这种寻址方式同样可用于表格处理,表格的首地址可设置为指令中的位移量,利用修改基址或变址寄存器的内容来存取表格中的项值。所以,这种方式很有利于实现高级语言中对结构或记录等数据类型所实施的操作。

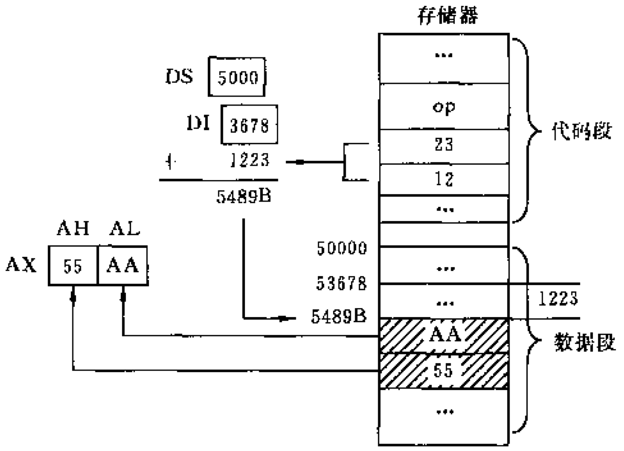


图 2.10　寄存器相对寻址方式示意图

请注意书写时基址或变址寄存器名一定要放在方括号中,而位移可不写在方括号中。下面两条指令源操作数的寻址方式是相同的,表示的形式等价:

MOV　AX,[SI+3]
MOV　AX,3[SI]

2.3.6　基址加变址寻址方式

操作数在存储器中,操作数的有效地址由基址寄存器之一的内容与变址寄存器之一的内容相加得到。即:

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

在一般情况(即不使用段超越前缀明确指定段寄存器)下,如果 BP 之内容作为有效地址的一部分,则以 SS 之内容为段值,否则以 DS 之内容为段值。

当所得的有效地址超过 FFFFH 时,就取其 64K 的模。

例如,　MOV　AX,[BX+DI]
假设,　(DS)=5000H,(BX)=1223H,(DI)=54H

那么,存取的物理存储单元地址是 51277H。再设该字存储单元的内容是 168H,那么

在执行该指令后，(AX)＝168H。图 2.11 反映该指令的执行情况。

下面指令中，源操作数采用基址加变址寻址，通过增加段超越前缀来引用段寄存器 ES：

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

下面指令中，目的操作数采用基址加变址寻址，引用的段寄存器是 DS：

```
MOV    DS:[BP+SI],AL
```

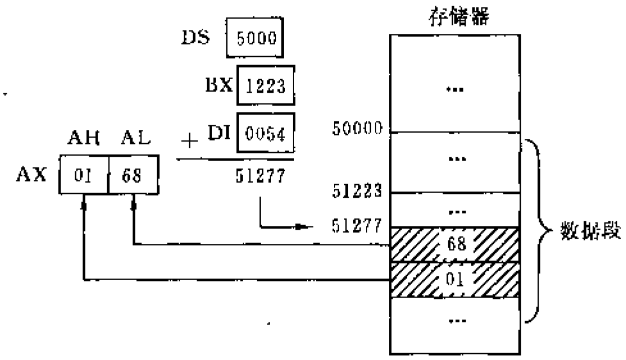


图 2.11　基址加变址寻址方式示意图

这种寻址方式适用于数组或表格处理。用基址寄存器存放数组首地址，而用变址寄存器来定位数组中的各元素，或反之。由于两个寄存器都可改变，所以能更加灵活地访问数组或表格中的元素。

下面的二种表示方法是等价的：

```
MOV    AX,[BX+DI]
MOV    AX,[DI][BX]
```

2.3.7　相对基址加变址寻址方式

操作数在存储器中，操作数的有效地址由基址寄存器之一的内容与变址寄存器之一的内容及指令中给定的 8 位或 16 位位移量相加得到。也即：

$$EA=\begin{Bmatrix}(BX)\\(BP)\end{Bmatrix}+\begin{Bmatrix}(SI)\\(DI)\end{Bmatrix}+\begin{Bmatrix}8\text{ 位}\\16\text{ 位}\end{Bmatrix}\text{位移量}$$

在一般情况（即不使用段超越前缀指令明确指定段寄存器）下，如果 BP 之内容作为有效地址的一部分，则以 SS 段寄存器之内容为段值，否则以 DS 段寄存器之内容为段值。

在指令中给定的 8 位或 16 位位移量采用补码形式表示。在计算有效地址时，如果位移量是 8 位，那么被带符号扩展成 16 位。当所得的有效地址超过 FFFFH 时，就取其 64K 的模。

例如、 MOV　AX,　[BX+DI−2]

假设， (DS)＝5000H，(BX)＝1223H，(DI)＝54H

那么，存取的物理存储单元地址是51275H。再设该字存储单元的内容是7654H，那么在执行该指令后，(AX)＝7654H。图2.12反映该指令的存储和执行情况，位移用补码表示。

尽管相对基址加变址这种寻址方式最复杂，但也是最灵活。

相对基址加变址这种寻址方式的表示方法多种多样，下面四种表示方法均是等价的：

```
MOV    AX,[BX+DI+1234H]
MOV    AX,1234H[BX+DI]
MOV    AX,1234H[BX][DI]
MOV    AX,1234H[DI][BX]
```

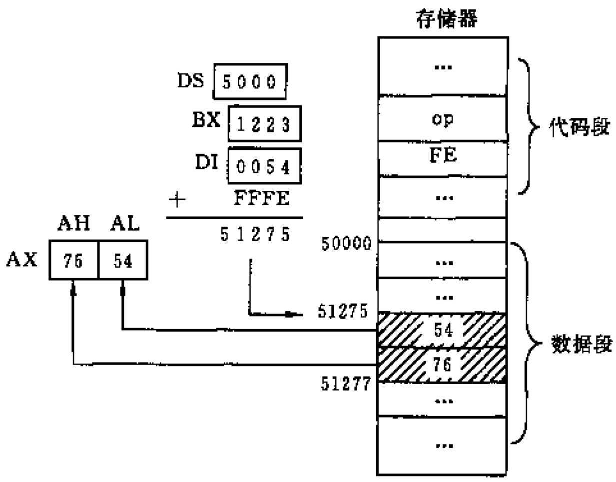


图 2.12 相对基址加变址寻址方式示意图

2.4 8086/8088 指令系统

本节详细介绍8086/8088指令集中的大部分常用指令，剩下的部分指令分散在有关章节中介绍。

2.4.1 指令集说明

1. 分组

与早先的8位微处理器相比，8086/8088的指令系统丰富，而且指令的功能也强。大多数指令既能处理字数据，又能处理字节数据；算术运算和逻辑运算不局限于累加器，存储器操作数也可直接参加算术逻辑运算。

8086/8088的指令系统可分为如下六个功能组：

(1) 数据传送；

(2) 算术运算；

(3) 逻辑运算；

(4) 串操作；

(5) 程序控制；

(6) 处理器控制。

2. 指令表示格式

为了方便地介绍指令系统中的指令，我们先介绍汇编语言中指令语句的一般格式。在汇编语言中，指令语句可由四部分组成，一般格式如下：

[标号:] 指令助记符 [操作数1 [，操作数2]] [;注释]

指令是否带有操作数，完全取决于指令本身，有的指令无操作数，有的指令只有一个操作数，有的指令需要两个操作数。标号的使用取决于程序的需要，是否写上注释由程序员决定。请注意：标号只被汇编程序识别，它与指令本身无关；由分号引导的注释则纯粹是为了理解和阅读程序的需要，汇编程序将其全部忽略，绝对不影响指令。

3. 其他说明

对于每一条指令，程序员要注意以下几点：

(1) 指令的功能；

(2) 适用于指令的操作数寻址方式；

(3) 指令对标志的影响；

(4) 指令的长度和执行时间。

2.4.2 数据传送指令

数据传送指令组又可分为：传送指令，交换指令，地址传送指令，堆栈操作指令，标志传送指令，查表指令，输入输出指令。查表指令和输入输出指令在有关章节介绍。

除了 SAHF 和 POPF 指令外，这组指令对各标志没有影响。

1. 传送指令

传送指令是使用得最频繁的指令，其格式如下：

```
MOV   DST,SRC
```

此指令把一个字节或一个字从源操作数 SRC 送至目的操作数 DST。

源操作数可以是累加器、寄存器、存储单元以及立即数，而目的操作数可以是累加器、寄存器和存储单元。传送不改变源操作数。

MOV 指令可实现的传送方向如图 2.13 所示。

具体地说，数据传送指令能实现下列传送功能：

(1) CPU 内部寄存器之间的数据传送。例如：

```
MOV   AH,AL
MOV   DL,BH
MOV   BP,SP
MOV   AX,CS
MOV   DS,BX
```

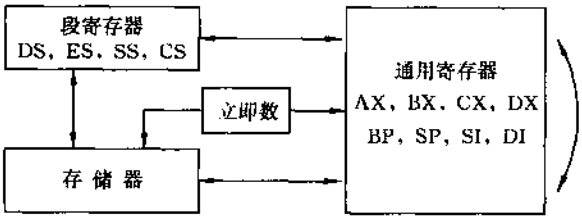


图 2.13 MOV 指令数据传送方向示意图

就寄存器之间传送而言,下列情况是例外:源和目的不能同时是段寄存器;代码段寄存器 CS 不能作为目的;指令指针 IP 既不能作为源,也不能作为目的。注意,这种例外永远存在。

(2) 立即数送至通用寄存器或存储单元(各种存储器寻址方式)。例如:

```
MOV   AL,3
MOV   SI,-5
MOV   VARB,-1            ;VARB 是变量名,代表一个存储单元
MOV   VARW,3456H         ;VARW 是一个字变量
MOV   [SI],6543H
```

注意,立即数不能直接传送到段寄存器,立即数永远不能作为目的操作数。

(3) 寄存器与存储器间的数据传送。例如:

```
MOV   AX,VARW          ;VARW 是一个字变量,存储器操作为直接寻址
MOV   BH,[DI]          ;存储器操作数为寄存器间接寻址
MOV   DI,ES:[SI+3]     ;存储器操作数为相对变址寻址,使用段超越前缀
MOV   BP,[BX+SI+3]     ;存储器操作数为相对基址加变址寻址
MOV   VARB,DL          ;VARB 是一个字节变量
MOV   [BP],AX          ;使用 SS 段寄存器
MOV   DS:[BP],DL       ;使用段超越前缀
MOV   VARW,DS          ;VARW 是一个字变量
MOV   ES,VARW
```

对存储器操作数而言,可采用各种存储器寻址方式,这一点对其他指令也一直成立。关于 MOV 指令,除了前面的例外,还要遵守下列规定:

① 源操作数和目的操作数类型要一致。即同时为字节或字,不能一个是字节,另一个是字。

② 除了串操作指令外,源操作数和目的操作数不能同时是存储器操作数。

这些例外和规定不仅适用于 MOV 指令,也同样适用于所有涉及到操作数的指令。

如果要在两个存储单元间传送数据,那么可利用通用寄存器过渡的方法进行,例如:

```
MOV   AX,VARW1         ;把字变量 VARW1 的内容送到字变量 VARW2
MOV   VARW2,AX
```

这种利用通用寄存器过渡的方法，也适用于段寄存器间的数据传送。例如：

```
MOV    AX,CS     ;把 CS 的内容送到 DS
MOV    DS,AX
```

2. 交换指令

利用交换指令可方便地实现通用寄存器与通用寄存器或存储单元间的数据交换，交换指令的格式如下：

```
XCHG   OPRD1,OPRD2
```

此指令把操作数 OPRD1 的内容与操作数 OPRD2 的内容交换。操作数同时是字节或字。例如：

```
XCHG   AL,AH
XCHG   SI,BX
```

OPRD1 和 OPRD2 可以是通用寄存器和存储单元。但不包括段寄存器，也不能同时是存储单元，还不能有立即数，这符合已介绍过的例外和规定。可采用各种存储器寻址方式来指定存储单元。例如：

```
XCHG   [SI+3],AL
XCHG   [DI+BP+3],BX
```

3. 地址传送指令

8086/8088 有如下三条地址传送指令。

(1) 指令 LEA(Load Effective Address)

指令 LEA 称为传送有效地址指令，其格式如下：

```
LEA    REG,  OPRD
```

该指令把操作数 OPRD 的有效地址传送到操作数 REG。操作数 OPRD 必须是一个存储器操作数，操作数 REG 必须是一个 16 位的通用寄存器。例如：

```
LEA    AX,BUFFER         ;BUFFER 是变量名
LEA    DX,[BX+3]
LEA    SI,[BP+DI+4]
```

请注意，LEA 指令与把存储单元中的数据传送到寄存器的 MOV 指令有本质上的区别。假设变量 BUFFER 的偏移是 1234H，该字变量的值为 5678H，那么在执行完指令“LEA　AX,BUFFER”后，AX 寄存器中的值为 1234H，而不是 5678H，在执行完指令“MOV　AX,BUFFER”后，AX 寄存器中的值为 5678H，而不是 1234H。

(2) 指令 LDS(Load pointer into DS)

段值和段内偏移构成 32 位的地址指针。该指令传送 32 位地址指针，其格式如下：

```
LDS    REG,  OPRD
```

该指令把操作数 OPRD 中所含的一个 32 位地址指针的段值部分送到数据段寄存器

DS,把偏移部分送到指令给出的通用寄存器 REG。操作数 OPRD 必须是一个 32 位的存储器操作数,操作数 REG 可以是一个 16 位的通用寄存器,但实际使用的往往是变址寄存器或指针寄存器。例如:

```
LDS    DI,[BX]
LDS    SI,FARPOINTER        ;FARPOINTER 是一个双字变量
```

假设双字变量 FARPOINTER 包含的 32 位地址指针的段值为 5678H,偏移为 1234H,那么在执行指令"LDS SI,FARPOINTER"后,段寄存器 DS 的值为 5678H,寄存器 SI 的值为 1234H。32 位地址指针的偏移部分存储在双字变量的低地址字中,段值部分存储在高地址字中。图 2.14 是该指令的执行示意图。

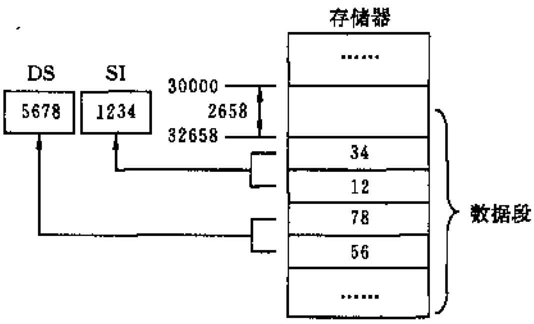


图 2.14 LDS 指令执行示意图

(3) 指令 LES(Load pointer into ES)

LES 指令也传送 32 位地址指针,其格式如下:

```
LES   REG, OPRD
```

该指令把操作数 OPRD 中所含的 32 位地址指针的段值部分送到附加段寄存器 ES,把偏移部分送到指令给出的通用寄存器 REG。其他说明同指令 LDS。

2.4.3 堆栈操作指令

在 8086/8088 系统中,堆栈是一段 RAM 区域。称为栈底的一端地址较大,称为栈顶的一端地址较小。堆栈的段值在堆栈段寄存器 SS 中,堆栈指针寄存器 SP 始终指向栈顶。只要重新设置 SS 和 SP 的初值(例如用 MOV 指令),就可以改变堆栈的位置。堆栈的深度由 SP 的初值决定。

堆栈操作始终遵守"后进先出"的原则,所有数据的存入和取出都在栈顶进行。在 8086/8088 系统中,进出堆栈的数据均以字为单位。

我们先列出堆栈的如下主要用途,每种用途的具体使用情况在以后的章节中陆续作介绍:

(1) 现场和返回地址的保护;

(2) 寄存器内容的保护;

(3) 传递参数；

(4) 存储局部变量。

堆栈操作指令分为两种：进栈指令 PUSH 和出栈指令 POP。

1. 进栈指令 PUSH

进栈指令把 16 位数据压入堆栈，其格式如下：

```
PUSH  SRC
```

该指令把源操作数 SRC 压入堆栈。它先把堆栈指针寄存器 SP 的值减 2，然后把源操作数 SRC 送入由 SP 所指的栈顶。图 2.15(a)和(b)示意指令“PUSH　AX”执行前后堆栈的变化情况，假设 AX＝8A9BH。随着压入堆栈的数据增多，堆栈也逐步扩展。SP 值随着压栈而减小，但每次操作完，SP 总是指向栈顶。当把一个 16 位数据压入堆栈时，总是遵守“高高低低”的存储原则。

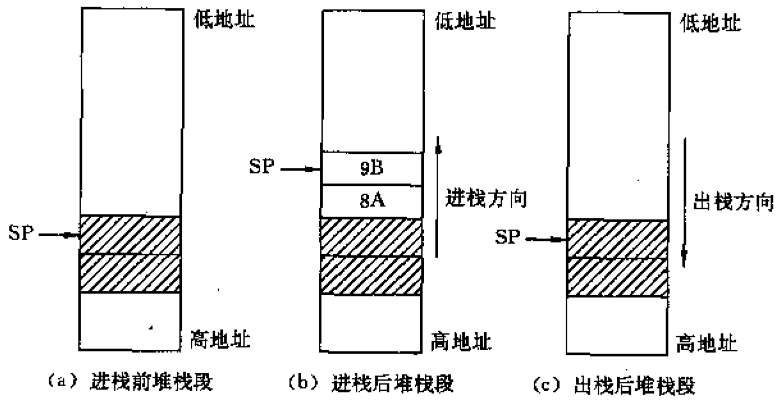


图 2.15　进栈和出栈操作示意图

源操作数 SRC 可以是通用寄存器和段寄存器，也可以是字存储单元。例如：

```
PUSH  SI
PUSH  DS
PUSH  VARW        ;VARW 是字变量
PUSH  [SI]
```

2. 出栈指令 POP

出栈指令从堆栈弹出 16 位数据，其格式如下：

```
POP  DST
```

该指令从栈顶弹出一个字数据到目的操作数 DST。它先把堆栈指针寄存器 SP 所指的字数据送至目的操作数 DST，然后 SP 值加 2，使其仍指向栈顶。图 2.15(b)和(c)示意执行指令“POP　AX”前后的堆栈变化情况。随着弹出堆栈的数据增多，堆栈也逐步收缩。SP 值随着弹出操作而增大，但每次操作完，SP 总是指向栈顶。

目的操作数 DST 可以是通用寄存器和段寄存器(但 CS 例外)，也可以是字存储单

元。例如：

```
POP  [SI]
POP  VARW          ;VARW 是字变量
POP  ES
POP  SI
```

下面的程序片段说明堆栈的一种用途，临时保存寄存器的内容：

```
PUSH  DS          ;保护 DS
PUSH  CS
POP   DS          ;使 DS 的内容与 CS 的内容相同
……                ;其他操作
POP   DS          ;恢复 DS
```

2.4.4 标志操作指令

8086/8088 指令集中，有一部分指令是专门对标志寄存器或标志位进行的，包括四条标志寄存器传送指令和七条专门用于设置或清除某些标志位的指令。

1. 标志传送指令

标志传送指令属于数据传送指令组。

(1) 指令 LAHF(Load AH with Flags)

指令 LAHF 采用固定寻址方式，指令格式如下：

```
LAHF
```

该条指令把标志寄存器的低 8 位(包括符号标志 SF、零标志 ZF、辅助进位标志 AF、奇偶标志 PF 和进位标志 CF)传送到寄存器 AH 的指定位，即相应地传送至寄存器 AH 的位 7、6、4、2 和 0，其他的位(位 5、3 和 1)的内容无定义，如图 2.16 所示。

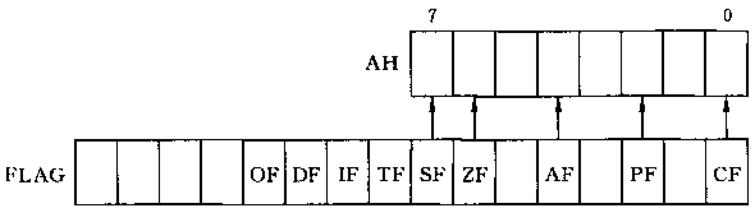


图 2.16 LAHF 指令示意图

这条指令本身不影响这些标志和其他标志。

(2) 指令 SAHF(Store AH into Flags)

指令 SAHF 采用固定寻址方式，其格式如下：

```
SAHF
```

该条指令与指令 LAHF 刚好相反，把寄存器 AH 的指定位送至标志寄存器低 8 位的

SF、ZF、AF、PF 和 CF 标志位。因而这些标志的内容就要受到影响，并取决于 AH 中相应位的状态。但这条指令不影响溢出标志 OF、方向标志 DF、中断允许标志 IF 和追踪标志 TF，也即不影响标志寄存器的高位字节。例如：

```
MOV    AH,  0C1H
SAHF                    ;CF=1,PF=0,AF=0,ZF=1,SF=1
```

(3) 指令 PUSHF

指令 PUSHF 的格式如下：

```
PUSHF
```

该条指令把标志寄存器的内容压入堆栈，即先把堆栈指针寄存器 SP 的值减 2，然后把标志寄存器的内容送入由 SP 所指的栈顶。

这条指令不影响标志。

(4) 指令 POPF

指令 POPF 的格式如下：

```
POPF
```

该条指令把当前堆栈顶的一个字传送到标志寄存器，同时相应地修改堆栈指针，即把堆栈指针寄存器 SP 的值加 2。

在执行该指令后，标志寄存器各位会发生相应变化。

这条指令和 PUSHF 指令一起可以保存和恢复标志寄存器的内容，即保存和恢复各标志的值。另外，这两条指令也可以用来改变追踪标志 TF。在 8086/8088 指令系统中，没有专门设置和清除 TF 标志的指令，为了改变 TF 标志，可先用 PUSHF 指令将标志压入堆栈，然后设法改变栈顶字单元中的第 8 位(把整个标志寄存器看成是一个字)，再用 POPF 指令把该字弹回到标志寄存器，这样其余的标志不受影响，而只有 TF 标志按需要改变了。

2. 标志位操作指令

标志位操作指令属于处理器控制指令组，它们仅对指令规定的标志产生指令规定的影响，对其他标志没有影响。

(1) 清进位标志指令 CLC(CLear Carry flag)

清进位标志指令的格式如下：

```
CLC
```

该条指令使进位标志为 0。

(2) 置进位标志指令 STC(SeT Carry flag)

置进位标志指令的格式如下：

```
STC
```

该条指令使进位标志为 1。

(3) 进位标志取反指令 CMC(CoMplement Carry flag)

进位标志取反指令的格式如下：

CMC

该条指令使进位标志取反。如 CF 为 1，则使 CF 为 0；如 CF 为 0，则 CF 为 1。

(4) 清方向标志 CLD(CLear Direction flag)

清方向标志指令的格式如下：

CLD

该条指令使方向标志 DF 为 0。从而在执行串操作指令时，使地址按递增方式变化。

(5) 置方向标志 STD(SeT Direction flag)

置方向标志指令的格式如下：

STD

该条指令使方向标志 DF 为 1。从而在执行串操作指令时，使地址按递减方式变化。

(6) 清中断允许标志 CLI(CLear Interrupt enable flag)

清中断允许标志指令的格式如下：

CLI

该条指令使中断允许标志 IF 为 0，于是 CPU 就不响应来自外部装置的可屏蔽中断。但对不可屏蔽中断和内部中断都没有影响。

(7) 置中断允许标志 STI(SeT Interrupt enable flag)

置中断允许标志指令的格式如下：

STI

该条指令使中断允许标志 IF 为 1，则 CPU 可以响应可屏蔽中断。

2.4.5 加减运算指令

8086/8088 提供加、减、乘和除四种基本算术运算操作。这些操作都可用于字节或字的运算，也可以用于无符号数的运算或有符号数的运算。有符号数用补码表示。加减运算指令不再分为无符号数运算指令和有符号数运算指令，而乘除运算指令还分为无符号数运算指令和有符号数运算指令。另外，8086/8088 还提供了各种十进制算术运算调整指令。

关于加减运算指令，有如下几点通用说明，请予以注意：

① 加减运算指令对无符号数和有符号数的处理一视同仁。既作为无符号数而影响标志 CF 和 AF，也作为有符号数影响标志 OF 和 SF，当然总会影响标志 ZF。加减运算指令也要影响标志 PF。有些指令稍有例外。

② 可参与加减运算的操作数如图 2.17 所示。总是只有通用寄存器或存储单元可用于存放运算结果。如果参与运算的操作数有两个，则最多只能有一个是存储器操作数。

③ 如果参与运算的操作数有两个，则它们的类型必须一致，即同时为字节，或同时为字。

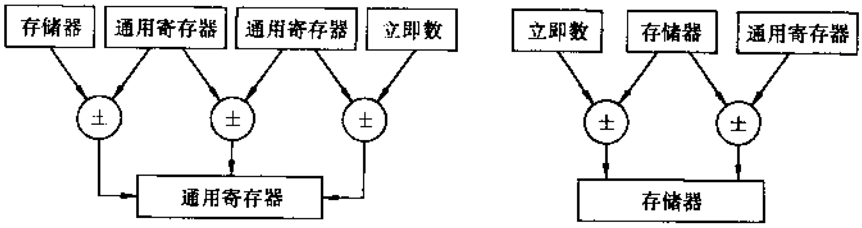


图 2.17　参与加减运算的操作数

④ 存储器操作数可采用 2.3 节中介绍的四种存储器操作数寻址方式。

1. 加法指令

(1) 普通加法指令 ADD(ADDiton)

普通加法指令的格式如下：

```
ADD   OPRD1, OPRD2
```

这条指令完成两个操作数相加，结果送至目的操作数 OPRD1，即：

```
OPRD1←OPRD1 + OPRD2
```

例如：

```
ADD   AL,5
ADD   AL,AH
ADD   DI,DI
ADD   BL,VARB            ;VARB 是字节变量
ADD   VARW,SI            ;VARW 是字变量
ADD   [BX+SI-3],AX
```

我们用下面的程序片段说明加法指令及其对标志的影响，同时说明 8 位数据寄存器与 16 位数据寄存器间的关系。安排的注释用于说明对应指令执行完后受影响的寄存器和标志位的变化，为了便于说明，采用 16 进制的形式表示数据。

```
MOV   AX,7896H           ;AX=7896H,即 AH=78H,AL=96H
                         ;各标志位保持不变
ADD   AL,AH              ;AL=0EH,AH=78H,即 AX=780EH
                         ;CF=1,ZF=0,SF=0,OF=0,AF=0,PF=0
ADD   AH,AL              ;AH=86H,AL=0EH,即 AX=860EH
                         ;CF=0,ZF=0,SF=1,OF=1,AF=1,PF=0
ADD   AL,0F2H            ;AL=00H,AH=86H,即 AX=8600H
                         ;CF=1,ZF=1,SF=0,OF=0,AF=1,PF=1
ADD   AX,1234H           ;AX=9834H,即 AH=98,AL=34H
                         ;CF=0,ZF=0,SF=1,OF=0,AF=0,PF=0
```

(2) 带进位加指令 ADC(ADd with Carry)

带进位加指令的格式如下：

```
ADC    OPRD1,OPRD2
```

这条指令与 ADD 指令类似，完成两个操作数相加，但还要把进位标志 CF 的现行值加上去，把结果送至目的操作数 OPRD1，即：

OPRD1⇐OPRD1 + OPRD2 + CF

例如：

```
ADC    AL,[SI]
ADC    DX,AX
ADC    DX,VARW           ;VARW 是字变量
```

ADC 指令主要用于多字节运算中。尽管在 8086/8088 中可以进行 16 位运算，但 16 位二进制数能表达的整数的范围还是很有限的，为了扩大数的范围，仍然需要多字节运算。例如，有两个四字节的数相加，加法要分两次进行，先进行低两字节相加，然后再做高两字节相加。在高两字节相加时，要把低两字节相加以后可能出现的进位考虑进去，用 ADC 指令实现这点很方便。

下面的程序片段实现两个四字节数相加，注意传送指令不影响标志：

```
MOV    AX,FIRST1         ;FIRST1 是存放第一个数低两字节的变量
ADD    AX,SECOND1        ;SECOND1 是存放第二个数低两字节的变量
MOV    THIRD1,AX         ;保存低两字节相加的结果到 THIRD1 变量中
MOV    AX,FIRST2         ;FIRST2 是存放第一个数高两字节的变量
ADC    AX,SECOND2        ;SECOND2 是存放第二个数高两字节的变量
MOV    THIRD2,AX         ;保存结果的高两字节到 THIRD2 变量中
```

(3) 加 1 指令 INC(INCrement)

加 1 指令的格式如下：

```
INC    OPRD
```

这条指令完成对操作数 OPRD 加 1，然后把结果送回 OPRD，即：

OPRD⇐OPRD + 1

例如：

```
INC    AL
INC    VARB   ;VARB 是字节变量
```

操作数 DST 可以是通用寄存器，也可以是存储单元。这条指令执行的结果影响标志 ZF、SF、OF、PF 和 AF，但它不影响 CF。

该指令主要用于调整地址指针和计数器。

例，假设有 100 个 16 位无符号数存放在从 1234：5678H 开始的内存中，现需要求它们的和。设把 32 位的和保存在 DX(高位)和 AX 寄存器中。

下面的程序片段能实现上述功能：

```
        ……
        MOV   AX,1234H
        MOV   DS,AX          ;置数据段寄存器值
        MOV   SI,5678H       ;置指针初值
        MOV   AX,0           ;清 32 位累加和
        MOV   DX,AX
        MOV   CX,100         ;置数据个数计数器
NEXT:   ADD   AX,[SI]        ;求和
        ADC   DX,0           ;加上可能的进位
        INC   SI             ;调整指针
        INC   SI
        DEC   CX             ;计数器减 1
        JNZ   NEXT           ;如果不为 0,那么就继续累加下一个数据
        ……
```

2. 减法指令

(1) 普通减法指令 SUB(SUBtraction)

普通减法指令的格式如下：

```
SUB  OPRD1, OPRD2
```

这条指令完成两个操作数相减，从 OPRD1 中减去 OPRD2，结果送到目标操作数 OPRD1 中，即：

$$OPRD1 \Leftarrow OPRD1 - OPRD2$$

例如：

```
SUB  AH,12
SUB  BX,BP
SUB  AL,[BX]
SUB  BX,VARW        ;VARW 是字变量
SUB  [BP-2],AX
```

我们用下面的程序片段说明减法指令及其对标志的影响，同时再次说明 8 位数据寄存器与 16 位数据寄存器间的关系。安排的注释用于说明对应指令执行完受影响的寄存器和标志位的变化，为了便于说明，还采用 16 进制的形式表示数据。

```
MOV   BX,9048H      ;BX=9048H,即 BH=90H,BL=48H
SUB   BH,BL         ;BH=48H,BL=48H,即 BX=4848H
                    ;CF=0,ZF=0,SF=0,OF=1,AF=1,PF=1
SUB   BL,BH         ;BL=00H,BH=48H,即 BX=4800H
                    ;CF=0,ZF=1,SF=0,OF=0,AF=0,PF=1
SUB   BL,5          ;BL=FBH,BH=48H,即 BX=48FBH
```

```
                              ;CF=1,ZF=0,SF=1,OF=0,AF=1,PF=0
SUB    BX,8F34H               ;BX=B9C7H,即 BH=B9H,BL=C7H
                              ;CF=1,ZF=0,SF=1,OF=1,AF=0,PF=0
```

(2) 带进(借)位减指令 SBB(SuBtract with Borrow)

带借位指令的格式如下：

```
SBB    OPRD1, OPRD2
```

这条指令与 SUB 指令类似，在操作数 OPRD1 减去操作数 OPRD2 的同时还要减借位(进位)标志 CF 的现行值，即：

OPRD1⇐OPRD1 − OPRD2 − CF

例如：

```
SBBAL,  DL
SBBDX,  AX
```

该指令主要用于多字节数相减的场合。

(3) 减 1 指令 DEC(DECrement)

减 1 指令的格式如下：

```
DEC    OPRD
```

这条指令把操作数 OPRD 减 1，并把结果送回 OPRD，即：

OPRD⇐OPRD − 1

例如：

```
DEC    BX
DEC    VARB           ;VARB 是字节变量
```

操作数 OPRD 可以是通用寄存器，也可以是存储单元。在相减时，把操作数作为一个无符号数对待。这条指令执行的结果影响标志 ZF、SF、OF、PF 和 AF，但它不影响 CF。

该指令主要用于调整地址指针和计数器。

(4) 取补指令 NEG(NEGate)

取补指令的格式如下：

```
NEG    OPRD
```

这条指令对操作数取补，就是用零减去操作数 OPRD，再把结果送回 OPRD，也即：

OPRD⇐ 0 − OPRD

例如：

```
NEG    AL
NEG    VARW[SI]       ;有效地址是变量 VARW 的位移加 SI 的值
```

如在字节操作时对－128 取补，或在字操作时对－32768 取补，则操作数没有变化，但

OF 被置位。操作数可以是通用寄存器,也可以是存储单元。此指令的执行结果影响 CF、ZF、SF、OF、AF 和 PF,一般总使 CF 为 1,除非操作数为 0。

(5) 比较指令 CMP(CoMPare)

比较指令的格式如下:

```
CMP    OPRD1,OPRD2
```

这条指令完成操作数 OPRD1 减去操作数 OPRD2,运算结果不送到 OPRD1,但影响标志 CF、ZF、SF、OF、AF 和 PF。例如:

```
CMP    SI,DI
CMP    CL,5
CMP    DX,[BP-4]
```

比较指令主要用于比较两个数的关系,是否相等,谁大谁小。在执行了比较指令后,可根据 ZF 是否置位,判断两者是否相等;如果两者是无符号数,则可根据 CF 判断大小;如果两者是有符号数,则要根据 SF 和 OF 判断大小。

例,设有两个 64 位数按“高高低低”原则存放同一个段的两个缓冲区 DATA1 和 DATA2 中,现需要计算 DATA1－DATA2。下面的程序片段计算 DATA1－DATA2,结果存放在 DATA1 中,可能发生的借位保留在 CF 中:

```
        ......
        MOV    CX,4              ;64 位分成 4 个字
        SUB    BX,BX             ;清指针,同时清 CF
NEXT:   MOV    AX,DATA2[BX]      ;取减数
        SBB    DATA1[BX],AX      ;带借位减
        INC    BX                ;调整指针
        INC    BX
        DEC    CX                ;是否已处理完 4 个字?
        JNZ    NEXT              ;没完继续
        ......
```

2.4.6 乘除运算指令

8086/8088 除了提供加减运算指令外,还提供乘除运算指令。乘除运算指令分为无符号数运算指令和有符号数运算指令,这点与加减运算指令不同。乘除运算指令对标志位的影响有些特别,不像加减运算指令对标志位的影响那样自然。

1. 乘法指令

在乘法指令中,一个操作数总是隐含在寄存器 AL(8 位数相乘)或者 AX(16 位数相乘)中,另一个操作数可以采用除立即数方式以外的任一种寻址方式。

(1) 无符号数乘法指令 MUL(MULtiply)

无符号指令的格式如下:

```
MUL    OPRD
```

如果 OPRD 是字节操作数，则把 AL 中的无符号数与 OPRD 相乘，16 位结果送到 AX 中；如果 OPRD 是字操作数，则把 AX 中的无符号数与 OPRD 相乘，32 位结果送到 DX 和 AX 对中，DX 含高 16 位，AX 含低 16 位。所以由操作数 OPRD 决定是字节相乘，还是字相乘。例如：

```
MUL    BL
MUL    AX
MUL    VARW              ;VARW 是字变量
```

如果乘积结果的高半部分（字节相乘时为 AH，在字相乘时为 DX）不等于零，则标志 CF=1，OF=1；否则 CF=0，OF=0。所以如果 CF=1 和 OF=1 表示在 AH 或 DX 中含有结果的有效数。该指令对其他标志位无定义。

(2) 有符号数乘法指令 IMUL(sIgned MULtiply)

有符号数乘指令的格式如下：

```
IMUL   OPRD
```

这条指令把被乘数和乘数均作为有符号数，此外与指令 MUL 完全类似。例如：

```
IMUL   CL
IMUL   DX
IMUL   VARW              ;VARW 是字变量
```

如果乘积结果的高半部分（字节相乘时为 AH，在字相乘时为 DX）不是低半部分的符号扩展，则标志 CF=1，OF=1；否则 CF=0，OF=0。所以如果 CF=1 和 OF=1 表示在 AH 或 DX 中含有结果的有效数。该指令对其他标志位无定义。

2. 除法指令

在除法指令中，被除数总是在隐含在寄存器 AX(除数是 8 位)或者 DX 和 AX(除数是 16 位)中，另一个操作数可以采用除立即数方式外的任一种寻址方式。

(1) 无符号数除法指令 DIV(DIVision)

无符号数除法指令的格式如下：

```
DIV    OPRD
```

如果 OPRD 是字节操作数，则把 AX 中的无符号数除以 OPRD，8 位的商送到 AL 中，8 位的余数送到 AH；如果 OPRD 是字操作数，则把 DX(高 16 位)和 AX 中的无符号数除以 OPRD，16 位的商送到 AX，16 位的余数送到 DX 中。所以由操作数 OPRD 决定是字节除，还是字除。例如：

```
DIV    BL
DIV    SI
DIV    VARW              ;VARW 是字变量
```

注意：如果除数为 0，或者在 8 位数除时商超过 8 位，或者在 16 位除时商超过 16 位，则认为是除溢出，引起 0 号中断。

除法指令对标志位的影响无定义。

(2) 有符号数除法指令 IDIV(sIgned DIVision)

有符号数除法指令的格式如下：

```
IDIV    OPRD
```

这条指令把被除数和除数均作为有符号数,此外与指令 DIV 完全类似。

例如：

```
IDIV    CX
IDIV    VARW            ;VARW 是字变量
```

当除数为 0,或者商太大(字节除时超过 127,字除时超过 32767),或者商太小(字节除时小于－127,字除时小于－32767)时,则引起 0 号中断。

3. 符号扩展指令

由于除法指令隐含使用字被除数或双字被除数,所以当被除数为字节,或者除数和被除数均为字时,需要在除操作前扩展被除数。为此 8086/8088 专门提供了符号扩展指令。

(1) 字节转换为字指令 CBW(Convert Byte to Word)

字节转换为字指令的格式如下：

```
CBW
```

这条指令把寄存器 AL 中的符号扩展到寄存器 AH。即若 AL 的最高有效位为 0,则 AH＝0;若 AL 的最高有效位为 1,则 AH＝0FFH。例如：

```
MOV     AX,3487H        ;AX=3487H,即 AH=34H, AL=87H
CBW                     ;AH=0FFH,AL=87H, 即 AX=0FF87H
```

这条指令能在两个字节相除以前,产生一个字长度的被除数。这条指令不影响各标志位。

(2) 字转换为双字指令 CWD(Convert Word to Double word)

字转换为双字指令的格式如下：

```
CWD
```

这条指令把寄存器 AX 中的符号扩展到寄存器 DX。即若(AX)的最高有效位为 0,则 DX＝0;若 AX 的最高有效位为 1,则 DX＝0FFFFH。

例如：

```
MOV     AX,4567H        ;AX=4567H
CWD                     ;AX=4567H,  DX=0
```

这条指令能在两个字相除以前,产生一个双字长度的被除数。该指令不影响各标志位。

注意：在无符号数除之前,不宜用 CBW 或 CWD 指令扩展符号位,一般采用 XOR 指令清高 8 位或高 16 位。

例，计算如下表达式的值：

$$(X * Y + Z - 1024)/75$$

假设其中的 X、Y 和 Z 均为 16 位带符号数，分别存放在名为 XXX、YYY 和 ZZZ 的变量单元中。再假设计算结果的商保存在 AX 中，余数保存在 DX 中。下面的程序片段能够满足要求：

```
……
MOV   AX,XXX
IMUL  YYY                 计算 X * Y
MOV   CX,AX
MOV   BX,DX               ;积保存到 BX : CX 中
MOV   AX,ZZZ
CWD                       ;把 ZZZ 扩展成 32 位
ADD   AX,CX               ;再计算和
ADC   DX,BX
SUB   AX,1024             ;再计算差
SBB   DX,0
MOV   CX,75
IDIV  CX                  ;最后计算商和余数
……
```

2.4.7 逻辑运算和移位指令

这组指令包括逻辑运算、移位和循环移位指令三部分。逻辑运算指令除指令 NOT 外，均有两个操作数。移位和循环移位指令只有一个操作数。关于这组指令有如下几点通用说明，请予以注意：

① 如果指令有两个操作数，那么这两个操作数也可如图 2.17 所示结合。但最多只能有一个为存储器操作数。

② 只有通用寄存器或存储器操作数可作为目的操作数，用于存放运算结果。

③ 如果只有一个操作数，则该操作数既是源又是目的。

④ 操作数可以是字节，也可以是字。但如果有两个操作数，则它们的类型必须一致，即同时为字节，或同时为字。

⑤ 对于存储器操作数可采用 2.3 节中介绍的四种存储器操作数寻址方式。

1. 逻辑运算指令

(1) 否操作指令 NOT

否操作指令的格式如下：

```
NOT    OPRD
```

这条指令把操作数 OPRD 取反，然后送回 OPRD。

例如：

```
NOT    AX
```

```
NOT     VARB            ;VARB是字节变量
```

操作数OPRD可以是通用寄存器，也可以是存储器操作数。此指令对标志没有影响。

(2) 与操作指令AND

与操作指令的格式如下：

```
AND     OPRD1, OPRD2
```

这条指令对两个操作数进行按位的逻辑“与”运算，结果送到目的操作数OPRD1。

例如：

```
AND  DH, DH
AND  AX,ES:[SI]
```

该指令执行以后，标志CF=0，标志OF=0，标志PF、ZF、SF反映运算结果，标志AF未定义。

某个操作数自己与自己相“与”，则值不变，但可使进位标志CF清0。与操作指令主要用在使一个操作数中的若干位维持不变，而另外若干位清为0的场合。把要维持不变的这些位与“1”相“与”，而把要清为0的这些位与“0”相“与”就能达到这样的目的。

例如：

```
MOV     AL,34H          ;AL=34H
AND     AL,0FH          ;AL=04H
```

(3) 或操作指令OR

或操作指令的格式如下：

```
OR      OPRD1,OPRD2
```

这条指令对两个操作数进行按位的逻辑“或”运算，结果送到目的操作数OPRD1。

例如：

```
OR      AX,8080H
OR      CL,AL
OR      [BX-3],AX
```

OR指令执行以后，标志CF=0，标志OF=0，标志PF、ZF、SF反映运算结果，标志AF未定义。

某个操作数自己与自己相“或”，则值不变，但可使进位标志CF清0。或操作指令主要用在使一个操作数中的若干位维持不变，而另外若干位置为1的场合。把要维持不变的这些位与“0”相“或”，而把要置为1的这些位与“1”相“或”就能达到这样的目的。

例如：

```
MOV    AL,41H          ;AL=01000001B,B表示二进制
OR     AL,20H          ;AL=01100001B
```

(4) 异或操作指令XOR

异或指令的格式如下：

```
XOR   OPRD1, OPRD2
```

这条指令对两个操作数进行按位的逻辑“异或”运算，结果送到目的操作数 OPRD1。该指令执行以后，标志 CF＝0，标志 OF＝0，标志 PF、ZF、SF 反映运算结果，标志 AF 未定义。

某个操作数自己与自己相“异或”，则结果为 0，并可使进位标志 CF 清 0。例如：

```
XOR    DX,DX          ;DX=0,  CF=0
```

异或操作指令主要用在使一个操作数中的若干位维持不变，而另外若干位置取反的场合。把要维持不变的这些位与“0”相“异或”，而把要取反的这些位与“1”相“异或”就能达到这样的目的。例如：

```
MOV    AL,34H        ;AL=00110100B,符号 B 表示二进制
XOR    AL,0FH        ;AL=00111011B
```

(5) 测试指令 TEST

测试指令的格式如下：

```
TEST  OPRD1, OPRD2
```

这条指令和指令 AND 类似，也把两个操作数进行按位“与”，但结果不送到操作数 OPRD1，仅仅影响标志。该指令执行以后，标志 ZF、PF 和 SF 反映运算结果，标志 CF 和 OF 被清 0。

该指令通常用于检测某些位是否为 1，但又不希望改变原操作数值的场合。例如，要检查 AL 中的位 6 或位 2 是否有一位为 1，可使用如下的指令：

```
TEST   AL,01000100B;          符号 B 表示二进制
```

如果位 6 和位 2 全为 0，那么在执行上面的指令后，ZF 被置 1，否则 ZF 被清 0。

2. 一般移位指令

8086/8088 有三条一般移位指令：算术左移/逻辑左移指令，算术右移指令，逻辑右移指令。这三条指令的一般格式如下：

```
SAL   OPRD,m          ;算术左移指令(同逻辑左移指令)
SHL   OPRD,m          ;逻辑左移指令
SAR   OPRD,m          ;算术右移指令
SHR   OPRD,m          ;逻辑右移指令
```

其中，m 是移位位数，或为 1 或为 CL，当要移多个位时，移位位数需存放在 CL 寄存器中。操作数 OPRD 可以是通用寄存器，也可以是存储器操作数。

标志 PF、SF 和 ZF 反映移位后的结果。标志 OF 也受影响，但标志 AF 未定义。

(1) 算术左移或逻辑左移指令 SAL/SHL(Shift Arithmetic Left 或 SHift logic Left)

算术左移和逻辑左移进行相同的动作，尽管为了方便提供有两个助记符，但只有一条机器指令。具体格式如下：

```
SAL    OPRD,m        或者
SHL    OPRD,m
```

算术左移 SAL/逻辑左移 SHL 指令把操作数 OPRD 左移 m 位，每移动一位，右边用 0 补足一位，移出的最高位进入标志位 CF。如图 2.18(a)所示。

下面的程序片段用于说明该指令的使用及其对标志位的影响，安排的注释给出了指令执行完后的操作数值和受影响的标志变化情况。

```
MOV    AL,8CH     ;AL=8CH
SHL    AL,1       ;AL=18H,CF=1,PF=1,ZF=0,SF=0,OF=1
MOV    CL,6       ;CL=6
SHL    AL,CL      ;AL=0,CF=0,PF=1,ZF=1,SF=0,OF=0
```

只要左移以后的结果未超出一个字节或一个字的表达范围，那么每左移一次，原操作数每一位的权增加了一倍，也即相当于原数乘 2。下面的程序片段实现把寄存器 AL 中的内容(设为无符号数)乘 10，结果存放在 AX 中。

```
XOR    AH,AH      ;(AH)=0
SHL    AX,1       ;2X
MOV    BX,AX      ;暂存 2X
SHL    AX,1       ;4X
SHL    AX,1       ;8X
ADD    AX,BX      ;8X+2X
```

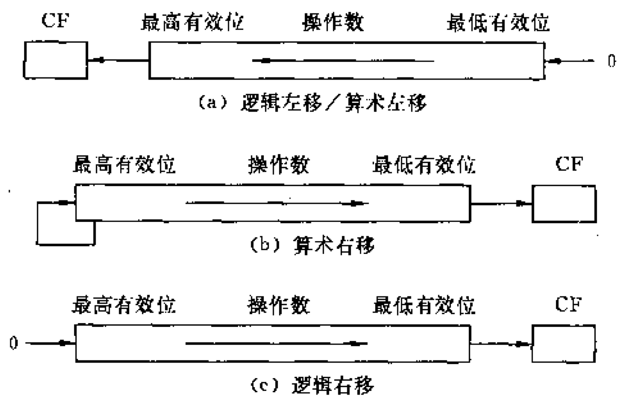


图 2.18 移位指令示意图

(2) 算术右移指令 SAR(Shift Arithmetic Right)

算术右移指令的格式如下：

```
SAR    OPRD,m
```

该指令使操作数右移 m 位，同时每移一位，左边的符号位保持不变，移出的最低位进入标志位 CF。如图 2.18(b)所示。

例如：

```
SAR    AL,1
SAR    BX,CL
```

对于有符号数和无符号数而言,算术右移一位相当于除以 2。

(3) 逻辑右移指令 SHR(SHift logic Right)

逻辑右移指令的格式如下:

```
SHR    OPRD,m
```

该指令使操作数右移 m 位,同时每移一位,左边用 0 补足,移出的最低位进入标志位 CF。如图 2.18(c)所示。例如:

```
SHR    BL,1
SHR    AX,CL
```

对于无符号数而言,逻辑右移一位相当于除以 2。

在汇编语言程序设计中,经常需要对以位为单位的数据进行合并和分解处理。一般通过移位指令和逻辑运算指令进行这种数据的合并和分解处理。

例,假设 DATA1 和 DATA2 各长 4 位,分别存放在 AL 寄存器的低 4 位和高 4 位中,现要把它们分别存放到 BL 寄存器和 BH 寄存器的低 4 位中。

下面的程序片段能实现上述要求:

```
……
MOV     BL,AL
AND     BL,0FH        ;得 DATA1
MOV     BH,AL         ;得 DATA2
MOV     CL,4
SHR     BH,CL
……
```

3. 循环移位指令

8086/8088 有四条循环移位指令:左循环移位指令 ROL(ROtate Left),右循环移位指令 ROR(ROtate Right),带进位左循环移位指令 RCL(Rotate Left through CF),带进位右循环移位指令 RCR(Rotate Right through CF)。这些指令可以一次只移一位,也可以一次移多位。如移多位,那么移位次数存放在 CL 寄存器中。

这些指令的格式如下:

```
ROL    OPRD,m
ROR    OPRD,m
RCL    OPRD,m
RCR    OPRD,m
```

其中,m 是移位次数,或为 1 或为 CL。操作数 OPRD 可以是通用寄存器,也可以是存储器操作数。

前两条循环指令没有把进位标志位 CF 包含在循环的环中;后两条循环指令把进位

标志 CF 包含在循环的环中，即作为整个循环的一部分。四条循环指令的操作如图 2.19 所示。

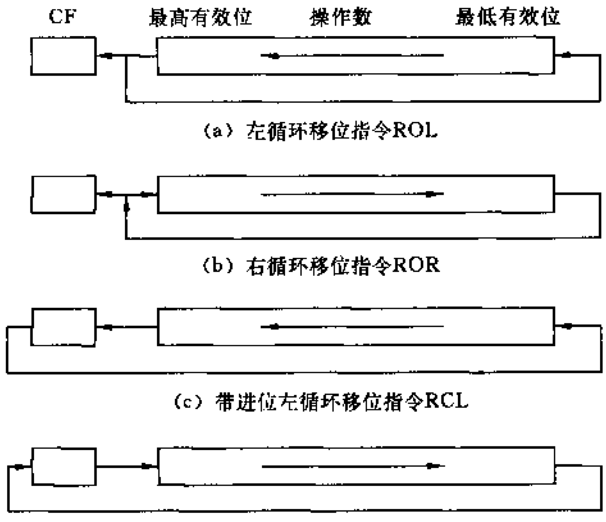


图 2.19 循环移位指令示意图

这些指令只影响标志 CF 和 OF。

左循环移位指令 ROL，它每移位一次，操作数左移，其最高位移入最低位，同时最高位也移入进位标志 CF。

右循环移位指令 ROR，它每移位一次，操作数右移，其最低位移入最高位，同时最低位也移入进位标志 CF。

带进位左循环移位指令 RCL，它每移位一次，操作数左移，其最高位移入进位标志 CF，CF 移入最低位。

带进位右循环移位指令 RCR，它每移位一次，操作数右移，其最低位移入进位标志 CF，CF 移入最高位。

对于不带进位的循环移位指令而言，如果操作数是 8 位，那么在移位 8 次后，操作数就能复原；如果操作是 16 位，那么在移位 16 次后，操作数就能复原。对于带进位的循环移位指令而言，如果操作数是 8 位，那么在移位 9 次后，操作就能复原；如果操作是 16 位，那么在移位 17 次后，操作就能复原。例如：

```
MOV    CL,9
RCR    AL,CL
```

通过带进位循环移位指令和其他移位指令的结合，可以实现两个或多个操作数的重新结合。

例，下面的程序片段实现把 AL 的高 4 位与低 4 位交换：

```
ROL  AL,1
```

```
ROL   AL,1
ROL   AL,1
ROL   AL,1
```

例,下面的程序片段实现把 AL 的最低位送入 BL 的最低位,仍保持 AL 不变:

```
ROR     BL,1
ROR     AL,1
RCL     BL,1
ROL     AL,1
```

例,设 DATA1 存放在 AL 的低 4 位,DATA2 存放在 AH 的低 4 位,DATA3 存放在 SI 的低 4 位,DATA4 存放在 SI 的高 4 位。现要把这四个数据合并为 16 位,并存放到 DX 寄存器中。存放要求如下所示。

DH		DL	
DATA1	DATA2	DATA3	DATA4

实现上述功能的程序片段如下:

```
……
;把 DATA1 送到 DH 的高 4 位,即 DX 的高 4 位
MOV     DH,AL
MOV     CL,4
SHL     DH,CL
;把 DATA2 送到 DH 的低 4 位,即 DX 的位 11 至位 8
AND     AH,0FH
OR      DH,AH
;把 DATA4 送到 DL 的低 4 位,即 DX 的低 4 位,同时 DATA3 送到 AL 的高 4 位
MOV     AX,SI
SHL     AX,1
RCL     DL,1
SHL     AX,1
RCL     DL,1
SHL     AX,1
RCL     DL,1
SHL     AX,1
RCL     DL,1
;把 DATA3 送到 DL 的高 4 位,即 DX 的位 7 至位 4
AND     DL,0FH
OR      DL,AL
…
```

下面的程序片段,也能实现上述功能,请比较之:

```
……
```

```
;把 DATA1 与 DATA2 合并,存放到 DH
MOV     CL,4
ROL     AL,CL
AND     AX,0FF0H
MOV     DH,AH
OR      DH,AL
;把 DATA3 与 DATA4 合并,存放到 DL
MOV     AX,SI
ROR     AX,CL
MOV     DL,AH
……
```

2.4.8 转移指令

8086/8088 提供了大量用于控制程序流程的指令,按功能可分成如下四类:

(1) 无条件转移指令和条件转移指令;

(2) 循环指令;

(3) 过程调用和过程返回指令;

(4) 软中断指令和中断返回指令。

由于程序代码可分为多个段,所以根据转移时是否重置代码段寄存器 CS 的内容,它们又可分为段内转移和段间转移两大类。段内转移是指仅重新设置指令指针 IP 的转移,由于没有重置 CS,所以转移后继续执行的指令仍在同一个代码段中。条件转移指令和循环指令只能实现段内转移。段间转移是指不仅重新设置 IP,而且重新设置代码段寄存器 CS 的转移,由于重置 CS,所以转移后继续执行的指令在另一个段中。软中断指令和中断返回指令总是段间转移。无条件转移指令和过程调用及返回指令既可以是段内转移,也可以是段间转移。段内转移也称为近转移,而段间转移也称为远转移。

对无条件转移指令和过程调用指令而言,按确定转移目的地址的方式还可分为直接转移和间接转移两种。

下面介绍无条件转移指令、条件转移指令和循环指令。这些指令均不影响标志。

1. 无条件转移指令

(1) 无条件段内直接转移指令

无条件段内直接转移指令的使用格式如下:

```
JMP     标号
```

这条指令使控制无条件地转移到标号地址处。例如:

```
NEXT:   MOV     AX,CX
        ……
        JMP     NEXT            ;转 NEXT 处
        ……
        JMP     OVER            ;转 OVER 处
```

```
        ......
OVER:   MOV    AX,1
```

无条件段内直接转移指令对应的机器指令格式如下，由操作码和地址差值构成。

指令操作码	地址差

其中的地址差是程序中该无条件转移指令的下一条指令的开始地址到转移目标地址（标号所指定指令的开始地址）的差值，由汇编程序在汇编时计算得出。因此，在执行无条件段内转移指令时，实际的动作是把指令中的地址差加到指令指针 IP 上，使 IP 之内容为目标地址，从而达到转移的目的。图 2.20 是无条件段内转移指令的存储和执行示意图。请注意，指令中的地址差值由汇编程序计算得出。

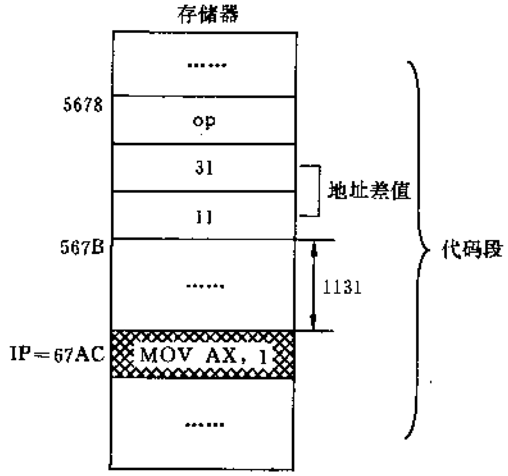


图 2.20 无条件段内转移指令的存储和执行示意图

段内无条件直接转移指令中的地址差可用一个字节表示，也可用一个字表示。如果地址差只要用一个字节表示，就称为短转移；如果地址差要用一个字表示，就称为近转移。一个字节表示的地址差的范围是－128 至＋127，所以，如果以转移指令本身为基准，那么短转移的范围则在－126 至＋129 之间。一个字表示的地址差的范围是 0 至 65535，当 IP 与地址差之和超过 65535 时，那么便在段内反饶（即取 65536 的模），所以，近转移的范围是整个段。

如果当汇编程序汇编到该转移指令时能够正确地计算出地址差，那么汇编程序就根据地址差的大小，决定使用一个字节表示地址差，还是使用一个字表示地址差。例如，上例中的“JMP NEXT”指令。如果当汇编程序汇编到该指令时还不能计算出地址差，那么汇编程序就按两字节地址差汇编此转移指令。例如，上例中的“JMP OVER”指令。对于后一种情况，如果程序员在写程序时能估计出用一字节就可表示地址差，那么可在标号前加一个汇编程序操作符 SHORT，例如：

```
JMP    SHORT OVER
```

这样汇编程序就按一字节的地址差汇编此转移指令。当实际的地址差无法用一个字节表示时，汇编程序会发出汇编出现错误的提示信息。

这种利用目标地址与当前转移指令本身地址之间的差值记录转移目标地址的转移方式也称为相对转移。相对转移有利于程序的浮动。

(2) 无条件段内间接转移指令

无条件段内间接转移指令的格式如下：

```
JMP    OPRD
```

这条指令使控制无条件地转移到由操作数 OPRD 的内容给定的目标地址处。操作数 OPRD 可以是通用寄存器，也可以是字存储单元。例如：

```
JMP    CX                         ;CX 寄存器的内容送 IP
JMP    WORD  PTR  [1234H]         ;字存储单元[1234H]的内容送 IP
```

图 2.21 给出了上述指令的存储和执行示意图。其中假设当前数据段偏移 1234H 处字单元的内容是 5678H。

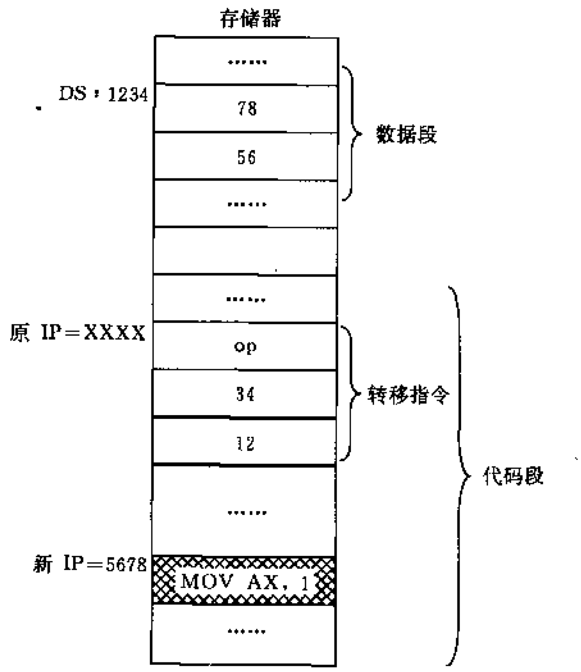


图 2.21 无条件段内间接转移指令示意图

(3) 无条件段间直接转移指令

无条件段间直接转移指令的使用格式如下：

```
JMP    FAR  PTR    标号
```

这条指令使控制无条件地转移到标号所对应的地址处。标号前的符号“FAR PTR”

向汇编程序说明这是段间转移。只有当标号具有远属性，且标号处的指令已先被汇编的情况下，才可省去远属性的说明“FAR PTR”。

例如：

```
JMP   FAR   PTR   EXIT        ;EXIT 是定义在另一个代码段中的标号
```

无条件段间直接转移指令的机器指令格式如下，由操作码及包括段值和偏移的地址构成。

指令操作码	目标地址偏移	目标地址段值

无条件段间直接转移指令的具体动作是把指令中包含的目标地址的段值和偏移分别置入 CS 和 IP。

这种在指令中直接包含转移目标地址的转移方式称为绝对转移。

(4) 无条件段间间接转移指令

无条件段间间接转移指令的格式如下：

```
JMP    OPRD
```

这条指令使控制无条件地转移到由操作数 OPRD 的内容给定的目标地址处。操作数 OPRD 必须是双字存储单元。例如：

```
JMP    DWORD  PTR  [1234H]      ;双字存储单元的低字内容送 IP
                                ;双字存储单元的高字内容送 CS
```

2. 条件转移指令

8086/8088 提供了大量的条件转移指令，它们根据某标志位或某些标志位的逻辑运算来判别条件是否成立。如果条件成立，则转移，否则继续顺序执行。

所有条件转移都只是段内转移。

条件转移也采用相对转移方式。即通过在 IP 上加一个地址差的方法实现转移。但条件转移指令中只用一个字节表示地址差，所以，如果以条件转移指令本身作为基准，那么条件转移的范围在－126 至＋129 之间。如果条件转移的目标超出此范围，那么必须借助于无条件转移指令。

条件转移指令不影响标志。

条件转移指令的格式列于表 2.3 中，有些条件转移指令有两个助记符，还有些条件转移指令有三个助记符。使用多个助记符的目的是便于记忆和使用。

表 2.3 条件转移指令

指令格式		转移条件	转移说明	其他说明
JZ	标号	ZF＝1	等于 0 转移	单个标志
JE	标号	ZF＝1	或者，相等转移	
JNZ	标号	ZF＝0	不等于 0 转移	单个标志
JNE	标号	ZF＝0	或者，不相等转移	

续表

指令格式	转移条件	转移说明	其他说明
JS　标号	SF=1	为负转移	单个标志
JNS　标号	SF=0	为正转移	单个标志
JO　标号	OF=1	溢出转移	单个标志
JNO　标号	OF=0	不溢出转移	单个标志
JP　标号 JPE　标号	PF=1 PF=1	偶转移	单个标志
JNP　标号 JPO　标号	PF=0 PF=0	奇转移	单个标志
JB　标号 JNAE　标号 JC　标号	CF=1 CF=1 CF=1	低于转移 或者,不高于等于转移 或者,进位标志被置转移	单个标志 无符号数
JNB　标号 JAE　标号 JNC　标号	CF=0 CF=0 CF=0	不低于转移 或者,高于等于转移 或者,进位标志被清转移	单个标志 无符号数
JBE　标号 JNA　标号	(CF 或 ZF)=1 (CF 或 ZF)=1	低于等于转移 或者,不高于转移	两个标志 无符号数
JNBE　标号 JA　标号	(CF 或 ZF)=0 (CF 或 ZF)=0	不低于等于转移 或者,高于转移	两个标志 无符号数
JL　标号 JNGE　标号	(SF 异或 OF)=1 (SF 异或 OF)=1	小于转移 或者,不大于等于转移	两个标志 有符号数
JNL　标号 JGE　标号	(SF 异或 OF)=0 (SF 异或 OF)=0	不小于转移 或者,大于等于转移	两个标志 有符号数
JLE　标号 JNG　标号	((SF 异或 OF)或 ZF)=1 ((SF 异或 OF)或 ZF)=1	小于等于转移 不大于转移	三个标志 有符号数
JNLE　标号 JG　标号	((SF 异或 OF)或 ZF)=1 ((SF 异或 OF)或 ZF)=1	不小于等于转移 大于转移	三个标志 有符号数

条件转移指令是用得最多的转移指令。通常,在条件转移指令前,总有用于条件判别的有关指令。

下面的程序片段测试 AX 的低四位是否全是 0,如果均是 0,那么使 CX=0,否则使 CX=-1。

```
        MOV     CX,-1           ;先使 CX=-1
        TEST    AX,0FH          ;测试 AX 的低 4 位
        JNZ     NZERO           ;不全为 0 则转移
        MOV     CX,0            ;全为 0 时使 CX=0
NZERO:  ……
```

从表 2.3 中可见,无符号数之间大小比较后的条件转移指令和有符号数之间的大小比较后的条件转移指令有很大不同。有符号数间的次序关系称为大于(G)、等于(E)和小

于(L)；无符号数间的次序关系称为高于(A)、等于(E)和低于(B)。所以，在使用时要注意区分它们，不能混淆。

下面的程序片段实现两个无符号数(设在 AX 和 BX 中)的比较，把较大的数存放到 AX 中，把较小的数存放在 BX 中：

```
      CMP     AX,BX
      JAE     OK           ;无符号数比较大小转移
      XCHG    AX,BX
OK:……
```

如果要比较的两个数是有符号数，则可用下面的程序片段：

```
      CMP     AX,BX
      JGE     OK           ;有符号数比较大小转移
      XCHG    AX,BX
OK: ……
```

从表 2.3 中可见，无符号数之间大小比较后的条件转移指令和有符号数之间的大小比较后的条件转移指令测试的标志完全不同。

不论无符号数还是有符号数，两数是否相等可由 ZF 标志反映。

当两个无符号数相减时，CF 位的情况说明了是否有借位。因此进位标志 CF 反映两个无符号数比较后的大小关系，所以用于无符号数比较后的条件转移指令(如 JB 和 JAE 等)检测标志 CF，以判别条件是否成立。但进位标志 CF 不能反映两个有符号数比较后的大小关系。

两个有符号数比较后的大小关系由符号标志 SF 和溢出标志 OF 一起来反映。所以用于有符号数比较后的条件转移指令(如 JL 和 JGE 等)检测标志 SF 和 OF，以判别条件是否成立。

设要比较的两个不相等的有符号数 a 和 b 分别存放在寄存器 AX 和 BX 中，执行指令“CMP　AX，　BX”后，标志 SF 及 OF 的设置情况和两数的大小情况如下：

当没有溢出(OF＝0)时，若 SF＝0，则 a＞b

　　　　　　　　　　　若 SF＝1，则 a＜b

当产生溢出(OF＝1)时，若 SF＝0，则 a＜b

　　　　　　　　　　　若 SF＝1，则 a＞b

据此可推断出表 2.3 中用于有符号数比较后的条件转移指令所测试的条件。

3. 循环指令

利用条件转移指令和无条件转移指令可以实现循环，但为了更加方便于循环的实现，8086/8088 还提供了四条用于实现循环的循环指令。

循环指令类似于条件转移指令，不仅属于段内转移，而且也采用相对转移的方式，即通过在 IP 上加一个地址差的方式实现转移。循环指令中也只用一个字节表示地址差，所以，如果以循环指令本身作为基准，那么循环转移的范围在－126 至＋129 之间。

循环指令不影响各标志。

(1) 计数循环指令 LOOP

计数循环指令的格式如下：

```
LOOP   标号
```

这条指令使寄存器 CX 的值减 1，如果结果不等于 0，则转移到标号，否则顺序执行 LOOP 指令后的指令。该指令类似于如下的两条指令：

```
DEC  CX
JNZ  标号
```

通常在利用 LOOP 指令构成循环时，先要设置好计数器 CX 的初值，即循环次数。由于首先进行 CX 寄存器减 1 操作，再判结果是否为 0，所以最多可循环 65536 次。

如下程序片段实现把从偏移 1000H 开始的 512 个字节的数据复制到从偏移 3000H 开始的缓冲区中(假设在当前数据段中进行移动)：

```
        MOV   SI,1000H      ;置源指针
        MOV   DI,3000H      ;置目标指针
        MOV   CX,512        ;置计数初值
NEXT:   MOV   AL,[SI]
        INC   SI
        MOV   [DI],AL
        INC   DI
        LOOP  NEXT          ;控制循环
  ……
```

(2) 等于/全零循环指令 LOOPE/LOOPZ

等于/全零循环指令有两个助记符，格式如下：

```
LOOPE   标号
```

或者

```
LOOPZ   标号
```

这条指令使寄存器 CX 的值减 1，如果结果不等于 0，并且零标志 ZF 等于 1，那么则转移到标号，否则顺序执行。注意指令本身实施的寄存器 CX 减 1 操作不影响标志。

如下的程序片段在字符串中查找第一个非'A'字符。设字符串长度已保存在 CX 中，并且 DS：DI 指向字符串。如果找到，那么使 BX 指向该非'A'字符，如果找不到，那么使 BX=0FFFFH。

```
        ……
        MOV     AL,'A'
        DEC     DI
NEXT:   INC     DI
        CMP     AL,[DI]
        LOOPE   NEXT
```

```
        MOV     BX,DI
        JNE     OK
        MOV     BX,-1
OK:     ……
```

(3) 不等于/非零循环指令 LOOPNE/LOOPNZ

不等于/非零循环指令有两个助记符，格式如下：

```
LOOPNE  标号
LOOPNZ  标号
```

这条指令使寄存器 CX 的值减 1，如果结果不等于 0，并且零标志 ZF 等于 0，那么则转移到标号，否则顺序执行。注意指令本身实施的寄存器 CX 减 1 操作不影响标志。

(4) 跳转指令 JCXZ

跳转指令也可以认为是条件转移指令。跳转指令的格式如下：

```
JCXZ  标号
```

该指令实现当寄存器 CX 的值等于 0 时转移到标号，否则顺序执行。

通常该指令用在循环开始前，以便在循环次数为 0 时，跳过循环体。

例如：

```
        ……
        JCXZ  OK            ;如果循环计数为 0,就跳过循环
NEXT:   ……                  ;循环体
        ……
        LOOP  NEXT          ;根据计数控制循环
OK:     ……
```

2.5 习 题

题 2.1 8086/8088 通用寄存器的通用性表现在何处？8 个通用寄存器各自有何专门的用途？哪些寄存器可作为存储器寻址方式的指针寄存器？

题 2.2 从程序员的角度看，8086/8088 有多少个可访问的 16 位寄存器？有多少个可访问的 8 位寄存器？

题 2.3 寄存器 AX 与寄存器 AH 和 AL 的关系如何？请写出如下程序片段中每条指令执行后寄存器 AX 的内容：

```
MOV     AX,1234H
MOV     AL,98H
MOV     AH,76H
ADD     AL,81H
SUB     AL,35H
ADD     AL,AH
```

```
ADC    AH,AL
ADD    AX,0D2H
SUB    AX,0FFH
```

题 2.4 8086/8088 标志寄存器中定义了哪些标志？这些标志可分为哪两类？如何改变这些标志的状态？

题 2.5 请说说标志 CF 和标志 OF 的差异。

题 2.6 8086/8088 如何寻址 1M 字节的存储器物理地址空间？在划分段时必须满足的两个条件是什么？最多可把 1M 字节空间划分成几个段？最少可把 1M 字节地址空间划分成几个段？

题 2.7 在 8086/8088 上运行的程序某一时刻最多可访问几个段？程序最多可具有多少个段？程序至少有几个段？

题 2.8 存储单元的逻辑地址如何表示？存储单元的 20 位物理地址如何构成？

题 2.9 当段重叠时，一个存储单元的地址可表示成多个逻辑地址。请问物理地址 12345H 可表示多少个不同的逻辑地址？偏移最大的逻辑地址是什么？偏移最小的逻辑地址是什么？

题 2.10 为什么称 CS 为代码段寄存器？为什么称 SS 为堆栈段寄存器？

题 2.11 请举例说明何为段前缀超越。什么场合下要使用段前缀超越？

题 2.12 8086/8088 的基本寻址方式可分为哪三类？它们说明了什么？

题 2.13 存储器寻址方式可分为哪几种？何为存储单元的有效地址？

题 2.14 什么场合下缺省的段寄存器是 SS？为什么要这样安排？

题 2.15 请说明如下指令中源操作数的寻址方式，并作相互比较：

```
MOV    BX,[1234H]
MOV    BX,1234H
MOV    DX,BX
MOV    DX,[BX]
MOV    DX,[BX+1234H]
MOV    DX,[BX+DI]
MOV    DX,[BX+DI+1234H]
```

题 2.16 8086/8088 提供了灵活多样的寻址方式。如何恰当地选择寻址方式？

题 2.17 设想一下这些寻址方式如何支持高级语言的多种数据结构。

题 2.18 为什么目标操作数不能采用立即寻址方式？

题 2.19 处理器内的通用寄存器是否越多越好？通用寄存器不够用怎么办？

题 2.20 哪些存储器寻址方式可能导致有效地址超出 64K 的范围？8086/8088 如何处理这种情况？

题 2.21 什么情况下根据段值和偏移确定的存储单元地址会超出 1M？8086/8088 如何处理这种情况？

题 2.22 8086/8088 的指令集可分为哪 6 个子集？

题 2.23 8086/8088 指令集合中，最长的指令有几个字节？最短的指令有几个字节？

请举例说明?

题 2.24 8086/8088 的算术逻辑运算指令最多一次处理多少 2 进制位?当欲处理的数据长度超出该范围怎么办?

题 2.25 如何实现使数据段与代码段相同?

题 2.26 通常情况下源操作数和目的操作数不能同时是存储器操作数。请给出把存储器操作数甲送到存储器操作数乙的两种方法。

题 2.27 请用一条指令实现把 BX 的内容加上 123 并把和送到寄存器 AX。

题 2.28 堆栈有哪些用途?请举例说明。

题 2.29 在本章介绍的 8086/8088 指令中,哪些指令把寄存器 SP 作为指针使用?8086/8088 指令集中,哪些指令把寄存器 SP 作为指针使用?

题 2.30 请说说标志 CF 的用途。请至少给出使标志 CF 清 0 的三种方法。

题 2.31 请写出如下程序片段中每条算术运算指令执行后标志 CF、ZF、SF、OF、PF 和 AF 的状态:

```
MOV     AL,89H
ADD     AL,AL
ADD     AL,9DH
CMP     AL,0BCH
SUB     AL,AL
DEC     AL
INC     AL
```

题 2.32 什么是除法溢出?如何解决 16 位被除数 8 位除数可能产生的溢出?

题 2.33 请写出如下程序片段中每条逻辑运算指令执行后标志 ZF、SF 和 PF 的状态:

```
MOV     AL,45H
AND     AL,0FH
OR      AL,0C3H
XOR     AL,AL
```

题 2.34 "MOV AX,0"可使寄存器 AX 清 0。另外再请写出三条可使寄存器 AX 清 0 的指令。

题 2.35 请写出如下程序片段中每条移位指令执行后标志 CF、ZF、SF 和 PF 的状态:

```
MOV     AL,84H
SAR     AL,1
SHR     AL,1
ROR     AL,1
RCL     AL,1
SHL     AL,1
ROL     AL,1
```

题 2.36 8086/8088 中，哪些指令把寄存器 CX 作为计数器使用？哪些指令把寄存器 BX 作为基指针寄存器使用？

题 2.37 请不用条件转移指令 JG、JGE、JL 和 JLE 等指令实现如下程序片段的功能：

```
        CMP     AL,BL
        JGE     OK
        XCHG    AL,BL
OK:     …………
```

题 2.38 段间转移和段内转移的本质区别是什么？8086/8088 哪些指令可实现段间转移？

题 2.39 8086/8088 的条件转移指令的转移范围有多大？如何实现超出范围的条件转移？

题 2.40 相对转移和绝对转移的区别是什么？相对转移有何优点？

题 2.41 请指出下列指令的错误所在：

(1) MOV CX,DL	(2) XCHG [SI],3
(3) POP CS	(4) MOV IP,AX
(5) SUB [SI],[DI]	(6) PUSH DH
(7) OR BL,DX	(8) AND AX,DS
(9) MUL 16	(10) AND 7FFFH,AX
(11) DIV 256	(12) ROL CX,BL
(13) MOV ES,1234H	(14) MOV CS,AX
(15) SUB DL,CF	(16) ADC AX,AL
(17) MOV AL,300	(18) JDXZ NEXT

题 2.42 请指出如下指令哪些是错误的，并说明原因：

(1) MOV [SP],AX	(2) PUSH CS
(3) JMP BX+100H	(4) JMP CX
(5) ADD AL,[SI+DI]	(6) SUB [BP+DI−1000],AL
(7) ADD BH,[BL−3]	(8) ADD [BX],BX
(9) MOV AX,BX+DI	(10) LEA AX,[BX+DI]
(11) XCHG ES:[BP],AL	(12) XCHG [BP],ES

题 2.43 下列程序片段完成什么功能，可否有更简单的方法实现同样的功能：

```
XCHG    AX,[SI]
XCHG    AX,[DI]
XCHG    AX,[SI]
```

题 2.44 请比较如下指令片段：

```
(1) LDS SI,[BX]          (2) MOV   SI,[BX]
                             MOV   DS,[BX+2]
```

(3) MOV　DS,[BX+2]

MOV　BX,[BX]

题 2.45　TC 或 BC 的编译模式与存储器分段管理有什么关系？

题 2.46　TC 或 BC 中使用的长指针的实质是什么？

第3章　汇编语言及其程序设计初步

汇编语言不仅仅是由汇编格式指令构成的指令语句，它还包括丰富的伪指令语句及其他内容。本章先简单介绍汇编语言，然后介绍如何进行顺序、分支和循环等程序设计。

3.1　汇编语言的语句

汇编语言源程序由汇编语言语句组成。尽管与高级语言的语句相比，汇编语言语句比较简单，但它有两类完全不同的语句。本节介绍语句格式及其主要组成部分——表达式的表示。

3.1.1　语句的种类和格式

1. 语句的种类

汇编语言有两种类型的语句，一种是指令语句，另一种是伪指令语句。这两种语句截然不同。汇编程序在对源程序进行汇编时，把指令语句翻译成机器指令，也就是说，指令语句有着与其对应的机器指令。伪指令语句没有与其对应的机器指令，只是指示汇编程序如何汇编源程序，包括符号的定义、变量的定义、段的定义等。

在宏汇编语言中，还有一种特殊的语句，称为宏指令语句。利用宏定义伪指令，可以把一个程序片段定义为一宏指令。当宏指令作为语句出现时，该语句就称为宏指令语句。所以，在宏汇编语言中，除了指令语句和伪指令语句外，还有宏指令语句。我们在第7章中再介绍宏指令语句。

2. 语句的格式

指令语句和伪指令语句的格式是相似的，都由四部分组成。

指令语句的格式如下：

[标号:]　指令助记符　[操作数[,操作数]]　[;注释]

我们在2.4.1中已对指令语句的格式作过简要说明。其中操作数可以是常数(数值表达式)操作数、寄存器操作数(寄存器名)或者存储器操作数(地址表达式)。

伪指令语句的格式如下：

[名字]　伪指令定义符　[参数,...,参数]　[;注释]

伪指令定义符规定了伪指令的功能。一般伪指令语句都有参数，用于说明伪指令的操作对象，参数的类型和个数随着伪指令的不同而不同。有时参数是常数(数值表达式)，有时参数是一般的符号，有时是具有特殊意义的符号。伪指令语句中的名字有时是必需的，有时是可省的，这也与具体的伪指令有关。在汇编语言源程序中，名字与标号很容易区分，名字后没有冒号，而标号后一定有冒号。

汇编程序忽略由分号开始至行尾的注释。为了阅读和理解程序的方便，程序员要恰当地使用注释，通过注释来说明语句或程序的功能。有时整行都可作为注释，只要该行以分号引导。

通常一个语句写一行。语句的各组成部分间要有分隔符。标号后的冒号是现成的分隔符，注释引导符分号也是现成的分隔符。此外，空格和制表符是最常用的分隔符，且多个空格或多个制表符的作用与一个空格或制表符的作用相同。空格和制表符被作为分隔符（除非作为字符串中的字符）而被忽略，所以常通过在语句行中加入空格和制表符的方法使上下语句行的各部分对齐，以方便阅读。尽管对齐不是必需的，但肯定有助于阅读。参数之间常用逗号作分隔符，但有时也用空格或制表符作分隔符。

标号和名字一般最多由 31 个字母、数字及规定的特殊字符（? @ _ $）等组成，并且不能用数字开头。一般情况下，汇编程序不区分标号和名字中的字母的大小写，除非要求汇编程序进行区分。值得指出的是，标号和名字要尽量起得有意义，这会大大有助于程序的阅读和理解。另外，标号和名字不能是汇编语言的保留字。汇编语言中的保留字主要是指令助记符、伪指令定义符和寄存器名，还有一些其他的特殊保留字。顺便说一下，汇编程序也不区分保留字中字母的大小写。

3.1.2 数值表达式

在汇编语言中，不仅有各种类型的运算符，还有许多操作符。通过运算符、操作符及括号把常数和符号连起来，就得到表达式。表达式又分为数值表达式和地址表达式。上述指令语句中的操作数和伪指令语句中的参数在许多场合下只是数值表达式。所谓数值表达式是指在汇编过程中能够由汇编程序计算出数值的表达式。所以组成数值表达式的各部分必须在汇编时就能完全确定。

标号和变量可作为数值表达式中的符号，由符号说明伪指令语句或符号定义伪指令语句说明或定义的符号，也可成为数值表达式中的符号。下面先介绍常数和运算符。

1. 常数

常数有多种类型和表示方式，常用的类型和表示方式如下：

(1) 十进制常数

由若干个 0 到 9 的数字组成的序列，可以用字母 D 结尾。例如：1024，2048D。通常情况下，常数用十进制表示，所以在表示十进制常数时一般不加后缀字母 D。但汇编语言提供改变基数的伪指令.RADIX。

(2) 十六进制常数

由若干个 0 至 9 的数字或字母 A 至 F 所组成的序列，必须以字母 H 结尾。为了避免与普通符号（如标号、名字和保留字）相混淆，十六进制数必须以数字开头。所以，凡是以字母 A 至 F 开头的十六进制数，必须在前面加一个 0。在汇编语言中，十六进制数用得较普遍。例如：

```
    OR      AX,8080H
    AND     BL,0F0H
```

(3) 二进制常数

由若干个 0 和 1 组成的序列，必须以字母 B 结尾。在汇编语言程序设计中，有时用二进制数较方便。例如：

```
TEST    BL,00110100B
OR      AL,11001010B
```

(4) 八进制常数

由若干个 0 至 7 的数字组成的序列，必须以字母 Q 结尾。例如：127Q，377Q。

(5) 字符串常数(串常数)

一个字符串常数是用引号括起来的一个或多个字符。串常数的值是包括在引号中的字符的 ASCII 代码值。例如，'A' 的值是 41H，而 'ab' 的值是 6162H。因此，串常数与整常数有时可以交替使用。例如：

```
CMP     AL,'A'
MOV     VARW,'ab'
CMP     AX,"AB"
```

2. 算术运算符

算术运算符包括正(+)、负(－)、加(+)、减(－)、乘(*)、除(/)和模(MOD)，这些算术运算符的意义与高级语言中同样运算符的意义相似。例如：

```
ADD     AX,100*4+2
SUB     CX,100H/2
MOV     AL,-3
```

3. 关系运算符

关系运算符包括相等(EQ)、不等(NE)、小于(LT)、大于(GT)、小于或等于(LE)、大于或等于(GE)。运算结果总是一个数字值。若关系不成立，则结果为 0，若关系成立，则结果为 0FFFFH。例如：

```
MOV     AX,1234H  GT  1024H
MOV     BX,1234H+5  LT  1024H
```

汇编后，目标程序中对应上述语句的指令如下：

```
MOV     AX,0FFFFH
MOV     BX,0
```

4. 逻辑运算符

逻辑运算符包括按位操作的“与”(AND)、“或”(OR)、“异或”(XOR)和“非”(NOT)，另外，还有左移位(SHL)和右移位(SHR)。逻辑运算的结果是数值。例如：

```
MOV     AX,1  SHL  3
ADD     CX,1024  SHR  4
OR      AL,3  AND  47H
AND     BL,NOT ( 7  OR  54H)
```

汇编后，目标程序中对应上述语句的指令如下：

```
MOV    AX,8
ADD    CX,40H
OR     AL,3
AND    BL,0A8H
```

请注意逻辑运算符与指令助记符的区别，表达式中的逻辑运算是由汇编程序在汇编时完成的。

5. 在数值表达式中使用的操作符

汇编语言中还有如下操作符可用在数值表达式中：HIGH、LOW、LENGTH、SIZE、OFFSET、SEG、TYPE、WIDTH 和 MASK 等。下面先介绍 HIGH 和 LOW，其他的操作符在以后的章节中再作介绍。

(1) HIGH

使用格式如下：

```
HIGH  数值表达式
```

结果是数值表达式值的高 8 位。

(2) LOW 使用格式如下：

```
LOW  表达式
```

结果是数值表达式值的低 8 位。

例如：

```
MOV    AX,HIGH  (1234H+5)
MOV    AX,HIGH  1234H +5
MOV    AX,LOW  1234H-3
```

汇编后，目标程序中对应上述语句的指令如下：

```
MOV    AX,12H
MOV    AX,17H                 ;HIGH 优先级高于加(+)
MOV    AX,31H
```

6. 运算符和操作符的优先级

汇编语言中各种运算符和操作符的优先级按高到低排列如下：

(1) 圆括号，尖括号，方括号，圆点符，LENGTH，SIZE，WIDTH，MASK。其中，尖括号使用于记录中，圆点符使用于结构中。

(2) PTR，OFFSET，SEG，TYPE，THIS，冒号。其中，冒号用于表示段超越前缀。

(3) *，/，MOD，SHL，SHR。

(4) HIGH，LOW。

(5) +，-。

(6) EQ，NE，LT，LE，GT，GE。

(7) NOT。

(8) AND。

(9) OR,XOR。

(10) SHORT。

3.1.3 地址表达式

地址表达式所表示的是存储器操作数的地址。单个的标号、变量(对应直接寻址方式)和有方括号括起的基址或变址寄存器(对应寄存器间接寻址)是地址表达式的特例。在2.3节中介绍的寄存器相对寻址、基址加变址寻址和相对基址加变址寻址等寻址方式的各种表示均属于地址表达式。

在一个存储器地址上加或减一个数字量,结果仍为存储器地址。例如:

```
MOV    AX,VARW+4
```

如VARW是变量,那么"VARW+4"表示以变量VARW的偏移加4为偏移的存储单元,而不是变量VARW的内容加4。实际上,在汇编时无法确定变量VARW的值。

在表示变址寻址方式时,下面的表示方法是等价的(其中VARW是变量或是符号常量):

```
[VARW+BX]
VARW[BX]
```

在表示基址加变址寻址方式时,下面的表示方法是等价的(其中VARW是变量或是符号常量):

```
VARW[BX+DI]
[VARW+BX+DI]
VARW[BX][DI]
VARW[DI][BX]
```

3.2 变量和标号

变量和标号分别代表存储单元。变量表示的存储单元中存放数值;标号表示的存储单元中存放指令代码。标号的定义很简单。本节介绍如何定义变量,以及变量和标号的属性。

3.2.1 数据定义语句

通过数据定义语句可为数据项分配存储单元,并根据需要设置其初值。还可用符号代表数据项,此时符号就与分配的存储单元相联系。代表数据项的符号本身称为变量名,与之相对应的存储单元用于存放变量,所以常常就把这样的存储单元称为变量。

1. 数据定义语句

数据定义语句是最常用的伪指令语句。一般格式如下:

```
[变量名]  数据定义符  表达式[,表达式,...,表达式]  ;注释
```

例如：

```
VARB    DB    3
VARW    DW    -12345
        DB    1
```

变量名是可选的，如果使用变量名，那么它就直接代表该语句所定义若干数据项中的第一个数据项。各表达式间用逗号分隔。例如：

```
BUFF    DB100,  3+4,  5*6
```

(1) 定义字节数据项

每一字节数据项只占用一个字节存储单元。定义字节数据项的数据定义符是DB。例如：

```
COUNT   DB    100
        DB    0DH,0AH,'$'
TABLE   DB    0,1,4,9,16
```

上面的数据定义语句被汇编后所对应的存储区域分配情况如图 3.1(a)所示，图中的数字值用 16 进制表示。从图 3.1(a)可见，由引号括起的字符对应其 ASCII 码值。下面是存取上述有关变量或数据项的指令举例：

```
DEC     COUNT
MOV     AL,TABLE
MOV     TABLE+2,BL      ;TABLE+2 是从 TABLE 开始的第 3 个字节
```

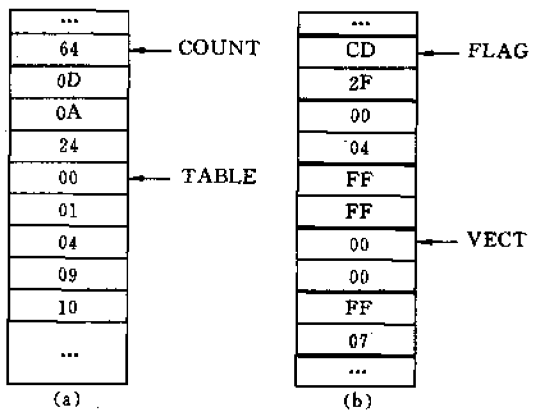


图 3.1 定义字节和字数据项示意图

(2) 定义字数据项

每一字数据项占用两个字节存储单元。定义字数据项的数据定义符是DW。例如：

```
FLAG    DW    2FCDH,1024,-1
VECT    DW    0
```

DW　　2047

上面的数据定义语句被汇编后所对应的存储区域分配情况如图 3.1(b)所示，图中的数字值用 16 进制表示。－1 用补码表示为 0FFFFH。下面是存取上述有关变量或数据项的指令举例：

```
MOV     BX,VECT
TEST    FLAG,1234H
OR      AX,FLAG+2        ;FLAG+2 是从 FLAG 开始的第二个字变量
```

(3) 定义双字数据项

每一双字数据项要占用四个字节存储单元。定义双字数据项的数据定义符是 DD。例如：

```
VECTOR  DD   4
FARPTR  DD   12345678H,0
```

数据定义语句中的表达式一般是数值表达式，汇编程序在计算出数值后就作为对应数据项的初值。所以，结果数值的大小必须适合对应变量或数据项的范围。

(4) 定义没有初值的数据项

如果数据定义语句中的表达式单单是一个问号(?)，那么表示不预置对应变量的初值，而仅仅是给变量分配存储单元。例如：

```
INBUFF DB    5,?,?,8,?
VARW   DW    ?
OLDV   DD    ?
```

(5) 定义字符串

定义字节数据伪指令 DB 也可方便地用于定义字符串。字符串要用引号括起来，单引号和双引号皆可，只要配对。DB 伪指令把字符串中的各个字符的 ASCII 码值依次存放在相应的字节存储单元中。例如：

```
MESS1 DB'HELLO! '
```

上述语句与如下语句起相同的作用：

```
MESS1 DB'H','E','L','L','O','! '
```

显然，用一对引号把字符串括起，要比把每一个字符用引号括起方便得多。再如：

```
MESS2 DB "How are you ?", 0DH,0AH,24H
```

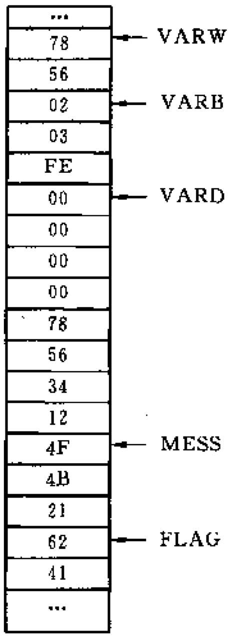


图 3.2　数据定义语句定义变量的存储格式

图 3.2 给出了某个程序中的下列数据定义语句所定义的变量或数据项使用的存储单元和存放格式。

```
VARW   DW   5678H
VARB   DB   2,3,-2
VARD   DD   0,12345678H
MESS   DB   'OK! '
FLAG   DW   'Ab'
```

从图 3.2 可见，字变量 FLAG 的初值 'Ab' 被解释为 4162H，所以，其高 8 位 41H 存放在高地址字节中。请注意"高高低低"原则。

(6) 定义其他类型数据项

利用数据定义语句还可定义 8 字节数据项和 10 字节数据项。定义 8 字节数据项的数据定义符是 DQ，定义 10 字节数据项的数据定义符是 DT。例如：

```
DT   0
DQ   ?
```

2. 重复操作符 DUP

有时需要定义数组，有时还需要定义数据缓冲区。为此，汇编语言提供了在数据定义语句中使用的重复操作符 DUP。例如：

```
BUFFER DB    8 DUP (0)
```

上述伪指令语句就定义了由 8 个字节组成的缓冲区，每个字节的初值为 0。这样的缓冲区也可理解成由 8 项构成的数组，每项一个字节，初值为 0。上述数据定义语句与如下的数据定义语句起到相同的作用：

```
BUFFER DB    0,0,0,0,0,0,0,0
```

重复操作符 DUP 的一般使用格式如下：

```
count  DUP  ( 表达式 [,表达式,...] )
```

上式作为特殊的表达式只能使用在数据定义语句中。其中，count 是重复次数，要重复的内容含在括号内，如有多个表达式，则表达式间用逗号分隔。表达式中还可再使用重复操作符 DUP，但有一定的嵌套层次限制。

例如：

```
BUFFER1  DB   5, 0, 5 DUP (?)
BUFFER2  DW   1024 DUP (0)
BUFFER3  DB   256 DUP ('ABCDE')
DATA     DW    1, 5 DUP (1, 2, 4 DUP(0) )
```

3.2.2 变量和标号

1. 变量和标号的属性

变量表示存储单元，这种存储单元中存放数值；标号也表示存储单元，这种存储单元中存放机器指令代码。所以，变量和标号均表示存储器操作数，都具有如下三种属性：

(1) 段值，变量或标号对应存储单元所在段的段值。

(2) 偏移,变量或标号对应存储单元的起始地址的段内偏移。

(3) 类型,变量的类型主要是字节(BYTE)、字(WORD)和双字(DWORD);标号的类型主要是近(NEAR)和远(FAR),近表示段内标号,远表示段间标号。

在汇编语言程序设计中,变量和标号的这三个属性很重要,为此,汇编语言提供专门的析值操作符和类型操作符,以便于对变量和标号的这三个属性进行有关操作处理。下面就简单介绍这些操作符。

2. 析值操作符

析值操作符也称为数值回送操作符,原因是这些操作符把一些特征或存储器地址的一部分作为数值回送。五个析值操作符的简单使用格式如下:

SEG	变量名或标号
OFFSET	变量名或标号
TYPE	变量名或标号
LENGTH	变量名
SIZE	变量名

这些操作符都使用在数值表达式中。为了说明这些操作符的作用,我们设在某个程序中有如下数据定义片段:

```
VARW   DW   1234H,5678H
VARB   DB   3,4
VARD   DD   12345678H
BUFF   DB   10 DUP (?)
MESS   DB   'HELLO'
```

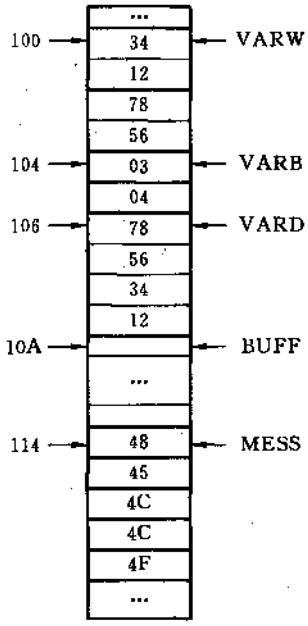


图 3.3 数据定义分配示意

设变量 VARW 从偏移 100H 开始,对应的存储情况如图 3.3 所示。

(1) 操作符 SEG 能返回变量所在段的段值,例如:

```
MOV    AX, SEG VARW          ;把变量 VARW 所在段的段值送 AX
MOV    DS,AX                 ;再送到数据段寄存器 DS
```

(2) 操作符 OFFSET 返回变量或者标号的偏移,例如:

```
MOV    BX,OFFSET  VARW       ;把 VARW 的偏移(100H)送 BX
ADD    DI,OFFSET  VARW+2     ;在汇编时计算出 OFFSET VARW +2 = 102H
MOV    SI,OFFSET  VARB       ;把 VARB 的偏移(104H)送 SI
```

请注意上一条指令与如下指令有质的不同:

```
LEA    SI, VARB
```

利用操作符 OFFSET 只能取得用数据定义伪指令定义的变量的有效地址,而不能取得一般操作数的有效地址。实际上,OFFSET 只是汇编语言提供的操作符,它的返回值是在汇编时由汇编程序计算出来的。

(3) 操作符 TYPE 返回变量或标号的类型,类型用数值表示,常见类型和对应的数值

规定如下：

字节(BYTE)变量	1
字(WORD)变量	2
双字(DWORD)变量	4
近(NEAR)标号	−1
远(FAR)标号	−2

由上述表示关系可见，变量的类型值是对应类型的变量项所占用的字节数，而标号的类型值却没有实际的物理意义。

(4) 操作符 LENGTH 返回利用 DUP 定义的数组中元素的个数，即重复操作符 DUP 前的 count 值。如果变量定义语句中没有使用 DUP，则总返回 1。如果嵌套使用了 DUP，则只返回最外层的重复数。例如：

```
MOV    CX,LENGTH  VARW    ;1 送 CX
MOV    CX,LENGTH  BUFF    ;10 送 CX
MOV    CX,LENGTH  MESS    ;1 送 CX
```

(5) 操作符 SIZE 返回用 DUP 定义的数组占用的字节数，可按下式计算：

SIZE 变量 = (LENGTH 变量) * (TYPE 变量)

例如：

```
MOV    CX,SIZE  VARW      ;2 送 CX
MOV    CX,SIZE  BUFF      ;10 送 CX
MOV    CX,SIZE  MESS      ;1 送 CX
```

3. 属性操作符

为了提高访问变量、标号和一般存储器操作数的灵活性，汇编语言还提供了属性操作符 PTR 和 THIS，以达到按指定属性访问的目的。

(1) 操作符 PTR

我们先看一个例子。汇编程序在汇编指令"MOV [SI],1"时，将发出警告提示信息或出错提示信息，其原因是汇编程序不能确定指针寄存器所指的存储器操作数的类型，即要访问的存储器操作数是字节类型还是字类型。

程序员要在源程序中明确指明，要访问的存储器操作数是字节类型还是字类型。这可利用 PTR 操作符来指明，例如：

```
MOV    WORD  PTR  [SI],1      ;指明字类型
MOV    BYTE  PTR  [SI],1      ;指明字节类型
```

PTR 是最常用的合成操作符，用在地址表达式前，用于指定或临时改变变量和标号的类型。一般格式如下：

类型 PTR 地址表达式

其中，类型可以是 BYTE、WORD、DWORD、NEAR 和 FAR。它指示汇编程序无论地址表达式所表示的单元类型是什么，当前均以 PTR 前面的类型为准。请注意，PTR 操作

符并不分配存储单元，而只是临时性地强制指定变量或标号的类型。

于是，利用 PTR 便可访问一个字变量的高字节和低字节，也可把两个字节变量当作一个字变量来访问。例如：

```
VARW   DW     1234H
VARB   DB     1
       DB     3
VARD   DD     12345678H
       ……
       MOV    AX,WORD  PTR  VARB        ;访问由 VARB 开始的一个字
       MOV    AL,BYTE  PTR  VARW        ;访问字变量 VARW 的低字节
       MOV    BYTE PTR  VARW+1,AL       ;访问字变量 VARW 的高字节
       MOV    DX,WORD  PTR  VARD        ;访问双字变量 VARD 的低字
       MOV    WORD  PTR  VARD+2,ES      ;访问双字变量 VARD 的高字
       MOV    WORD PTR ES:[DI+2],0
```

再如：

```
JMP    FAR  PTR  OK                     ;OK 是标号
JMP    DWORD  PTR  OLDVECT              ;OLDVECT 是变量
```

(2) 操作符 THIS

操作符 THIS 的一般格式如下：

THIS　类型

其中类型可以是 BYTE、WORD、DWORD、NEAR 和 FAR 等。它返回一个具有指定类型的存储器操作数，但决不为该存储器操作数分配存储单元，所返回存储器操作数地址的段值和偏移就是下一个将分配的存储单元的段值和偏移。

与操作符 PTR 相比，有相似之处，即都能指定操作数类型。所不同的是，操作符 THIS 并不直接作用于其他的变量或标号，而操作符 PTR 则不然。

操作符 THIS 一般使用在符号定义语句中，从而定义一个具有类型、段值和偏移三属性的表示存储器操作数的符号。例如：

```
MY_BYTE   EQU   THIS  BYTE       ;EQU 是符号定义语句的定义符
MY_WORD   DW    ?
```

如果在源程序中安排上述两条伪指令语句，那么符号 MY_BYTE 就表示一个字节变量，它的段值和偏移与紧随其后的字变量 MY_WORD 相同。所以，对字节变量 MY_BYTE 的访问实际上就是对字变量 MY_WORD 低字节的访问。

3.3　常用伪指令语句和源程序组织

本节介绍常用的伪指令语句，并给出汇编语言源程序的组织形式。其他伪指令语句在以后各章节需要处给出。

3.3.1 符号定义语句

通过符号定义语句，可把常数、表达式等用符号来表示。恰当地使用符号定义语句，不仅可大大方便程序的书写和阅读，对程序的调试和修改也很有利。

1. 等价语句 EQU

等价语句的一般格式如下：

符号名　EQU　表达式

(1) 用符号来代表常数或数值表达式。在这种情况下，汇编程序计算出表达式的值，符号就代表计算结果。例如：

```
COUNT       EQU     100           ;符号 COUNT 就代表常数 100
BUFF_LEN    EQU     4 * COUNT     ;COUNT 是已定义的符号常数
LTX         EQU     1
RDX         EQU     LTX + 50
......
INBUFFER    DB      COUNT,?,COUNT DUP (?)
......
```

(2) 用符号表示一个字符串。可用一简短的符号表示一复杂的字符串，以后当汇编程序遇到所定义的符号时，就用字符串代替之。例如：

```
HELLO EQU  "How  are  you !"
```

(3) 重新定义关键字或指令助记符。也即给汇编语言的关键字或指令助记符起一个别的名称。例如：

```
MOVE   EQU    MOV
COUNT  EQU    CX
```

在安排了上述语句后，就可用 MOVE 代替指令助记符 MOV，用 COUNT 代表寄存器 CX。当然指令助记符 MOV 和寄存器名 CX 还照可使用。例如：

```
MOVE   AX,CX                ;MOV   AX,CX
MOV    COUNT,100            ;MOV   CX,100
```

(4) 定义存储器操作数符号。所定义的存储器操作数符号具有类型、段值和偏移属性。例如：

```
VARW    EQU     THIS   WORD     ;VARW 的类型是字，段值和偏移与紧
VARB    DB      2 DUP (0)       ;接的下一单元 VARB 相同。
FLAG    DW      ?
FLAG1   EQU     BYTE   PTR   FLAG
FLAG2   EQU     BYTE   PTR   FLAG +1
```

在这之后，就可使用这些符号，例如：

```
MOV     AX,VARW
MOV     AL,FLAG1                ;相当于 MOV AL,BYTE PTR FLAG
MOV     FLAG2,AL                ;相当于 MOV BYTE PTR FLAG+1,AL
```

需要注意：第一，等价语句不另外给符号分配存储单元；第二，等价语句定义的符号不能与其它符号相同，也不能被重新定义，否则汇编程序会认为出现符号重新定义错误。

2. 等号语句(=)

汇编语言还专门提供等号语句来定义符号常数，即用符号表示一个常数。等号语句的一般格式如下：

```
符号名= 数值表达式
```

例如：

```
XX = 10
YY= 20+300/4
```

数值表达式应该可以计算出数值，所以表达式中一般不能含有向前引用的符号名称。用等号语句定义的符号可被重新定义。例如：

```
ABCD  = 1
ABCD  = 100
ABCD  = 2*ABCD+1
```

3. 定义符号名语句

定义符号名语句的一般格式如下：

```
符号名   LABEL   类型
```

其中类型可以是 BYTE、WORD、DWORD、NEAR 和 FAR 等。该语句的功能是定义有符号名指定的符号，使该符号的段属性和偏移属性与下一个紧接着的存储单元的段属性和偏移属性相同，使该符号的类型为参数所规定的类型。例如：

```
BUFFER LABEL   WORD
BUFF   DB      100  DUP(0)
```

BUFFER 的类型是 WORD，段属性和偏移属性与 BUFF 相同。再如：

```
QUIT    LABEL FAR
EXIT:   MOV     AH,4CH
```

这样指令“MOV　AH,4CH”就有了两个标号 QUIT 和 EXIT，但它们的类型不同。

3.3.2 段定义语句

为了与存储器的分段结构相对应，汇编语言的源程序也由若干个段组成。段定义语句就是用来按段组织程序和利用存储器的。

1. 段开始和结束语句

汇编语言源程序中的段以段开始语句开始，以段结束语句结束。段定义的一般格式

如下：

```
段名  SEGMENT  [定位类型] [组合类型] ['类别']
  ⋮
段名  ENDS
```

段开始语句的定义符是SEGMENT，其中的定位类型、组合类型和类别都是可省的，暂不介绍它们的用法和用途(请见第8章)。段结束语句的定义符是ENDS。段开始语句中的段名与段结束语句中的段名要相同，从而保持配对。段名的命名方法与一般符号的命名方法相同。

一个简单的数据段如下所示：

```
DSEG    SEGMENT
MESS    DB'HELLO',0DH,0AH,'$'
DSEG    ENDS
```

一个简单的代码段如下所示：

```
CSEG    SEGMENT
        MOV     AX,DSEG           ;把数据段DSEG的段值送AX
        MOV     DS,AX             ;再送DS寄存器
        MOV     DX,OFFSET  MESS
        MOV     AH,9
        INT     21H               ;DOS系统功能调用
        MOV     AH,4CH
        INT     21H               ;DOS系统功能调用
CSEG    ENDS
```

当段名作为符号被引用时，表示对应段的段值。指令"INT 21H"是软中断指令，在上述代码中起调用DOS系统功能的作用。

由于指令语句和数据定义语句都要占用存储单元，所以它们一定要安排在某个程序段内。

2. 段使用设定语句

汇编程序根据段开始语句和段结束语句判断出源程序的段划分，为了有效地产生目标代码，汇编程序还要了解各程序段与段寄存器间的对应关系。段寄存器与程序段的对应关系由段使用设定语句说明。

段使用设定语句的简单格式如下：

```
ASSUME  段寄存器名:段名[,段寄存器名:段名,....]
```

段寄存器名可以是CS、DS、SS和ES。段名就是段开始语句和段结束语句中规定的段名。例如，下面的ASSUME语句告诉汇编程序，从现在开始CS寄存器对应CSEG段，DS寄存器对应DSEG段。

```
ASSUME   CS:CSEG,DS:DSEG
```

ASSUME 伪指令中的段名域也可以是一个特别的关键字 NOTHING，它表示某个段寄存器不再与任何段有对应关系。

在一条 ASSUME 语句中可建立多个段寄存器与段的关系，只要用逗号分隔。在源程序中可使用多条 ASSUME 语句，通常在代码段的一开始就使用 ASSUME 语句，确定段寄存器与段的对应关系，以后可根据需要再使用 ASSUME 语句改变已建立的对应关系。

例如：

```
DSEG1   SEGMENT                      ;定义一个数据段，段名为 DSEG1
VARW    DW    12
        ......
DSEG1   ENDS
;
DSEG2   SEGMENT                      ;定义另一个数据段，段名为 DSEG2
XXX     DW    0
YYY     DW    0
        ......
DSEG2   ENDS
;
CSEG    SEGMENT                      ;定义一个代码段
        ASSUME  CS:CSEG, DS:DSEG1, ES:DSEG2
        MOV     AX,DSEG1
        MOV     DS,AX
        MOV     AX,DSEG2
        MOV     ES,AX
        ......
        MOV     AX,VARW
        MOV     XXX,AX
        ......
        ASSUME    DS:DSEG2,ES:NOTHING
        MOV     AX,DSEG2
        MOV     DS,AX
        ......
        MOV     AX,XXX
        MOV     YYY,AX
        ......
CSEG    ENDS
```

段使用设定语句是伪指令语句，它不能设置段寄存器的值，所以在上述程序中还需要通过指令语句来给数据段寄存器和附加段寄存器赋值。

变量 VARW 在 DSEG1 段中定义，变量 XXX 在 DSEG2 段中定义，而第一条 ASSUME 语句表示数据段寄存器 DS 对应 DSEG1，附加段寄存器 ES 对应 DSEG2，因此

汇编程序在汇编“MOV XXX,AX”时,将自动加上段超越前缀,即成为“MOV ES:XXX,AX”。由于第二条ASSUME语句把DS与DSEG2对应,而变量XXX和YYY均在DSEG2段中,所以汇编程序在汇编“MOV AX,XXX”和“MOV YYY,AX”时就不再加上段超越前缀。假设没有第二条ASSUME语句,那么汇编程序也要为上述两条指令加上段超越前缀ES:。如果在第二条ASSUME语句后,安排指令“MOV AX,VARW”,汇编程序将发出无法访问变量VARW的出错提示信息,原因是第二条ASSUME语句解除了DS寄存器与DSEG1段的对应关系,DSEG1段不再与任何段寄存器对应。再如果把第二条ASSUME语句改变为“ASSUME DS:DSEG2,ES:DSEG1”,结果会是什么?

我们可以在有关的指令中明确加上段超越前缀,从而改变ASSUME语句对有关指令的影响。例如:

```
MOV    AX,ES:XXX
MOV    DS:YYY,AX
```

3. ORG 语句

汇编程序在对源程序汇编的过程中,使用地址计数器来保存当前正在汇编的指令或者变量的地址偏移。通常地址计数器的值逐步递增,但程序员可利用ORG语句调整地址计数器的当前值。

ORG语句的一般格式如下:

```
ORG    数值表达式
```

汇编程序在汇编到该伪指令语句后,使地址计数器的值调整成数值表达式的结果值。如数值表达式的值是n,那么ORG伪指令语句使下一个字节的地址成为n。例如:

```
TESTSEG    SEGMENT
           ORG      100H
BEGIN:     MOV      AX,1234H
           ......
           ORG      500H
VAR        DW       1234
           ......
TESTS      ENDS
```

标号BEGIN的偏移等于100H,变量VAR的偏移等于500H。

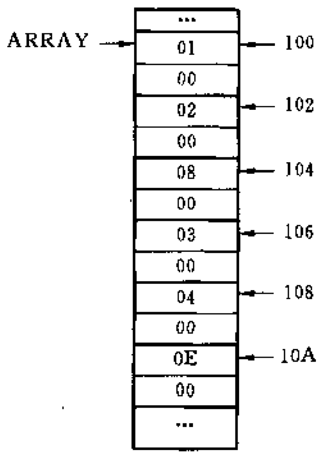


图 3.4 数组 ARRAY 的存储分配图

另外,汇编语言用符号“$”表示地址计数器的值。允许程序员在指令和伪指令中直接用符号$引用地址计数器的当前值。例如下面的语句表示跳过8个字节的存储区:

```
ORG    $+8
```

当$用在指令中时,它表示本条指令第一字节的地址偏移。例如:下面的指令表示转

移到距当前指令第一字节后 6 字节处：

```
JMP    $+6
```

当 $ 用在伪指令的参数中时，它表示的是地址计数器的当前值。例如：

```
ARRAY  DW    1,2,$+4,3,4,$+4
```

设在汇编时 ARRAY 分配的地址偏移是 100H，那么汇编后相应的存储情况如图 3.4 所示。从图 3.4 中可见，由于 $ 的值在不断变化，所以数组中两个数值表达式 $+4 得到的结果是不同的。

3.3.3 汇编语言源程序的组织

1. 一个完整的源程序

我们先看一个简单又完整的源程序。

```
;程序名：T3-1.ASM
;功  能：显示信息"HELLO"
;
SSEG    SEGMENT PARA  STACK  ;堆栈段
        DW   256 DUP (?)
SSEG    ENDS
;
DSEG    SEGMENT              ;数据段
MESS    DB'HELLO',0DH,0AH,'$'
DSEG    ENDS
;
CSEG    SEGMENT              ;代码段
        ASSUME  CS:CSEG, DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX        ;设置数据段寄存器
        MOV     DX,OFFSET  MESS
        MOV     AH,9
        INT     21H          ;显示信息 HELLO
        MOV     AH,4CH
        INT     21H          ;返回 DOS
CSEG    ENDS
        END     START
```

源程序 T3-1.ASM 含有三个段，即堆栈段、数据段和代码段。数据段含有程序要使用到的数据，代码段一般含有程序的代码。堆栈段作为堆栈使用，它由堆栈段定义语句说明，在第 8 章节中再对此作解释。

在经过汇编和连接处理后，可得到一个可执行程序。它运行时，在屏幕上显示字符串信息"HELLO"。操作系统(DOS)在把它装入运行时，将给上述逻辑上的三个段，分配三个

相应的物理段。在 DOS 把控制权转到该程序时，将设置妥代码段寄存器 CS 和指令指针寄存器 IP。

2. 源程序的组织

汇编语言源程序的主体是若干个段，少到一个段，多至几十个段，一般格式如下：

```
NAME1  SEGMENT
       ......
       ......
NAME1  ENDS
;
NAME2  SEGMENT
       ......
       ......
NAME2  ENDS
       ......
       ......
NAMEn  SEGMENT
       ......
       ......
NAMEn  ENDS
       END     标号
```

通常情况下，代码和数据分别在代码段和数据段中，但有时代码和数据可以合并在同一个段中。一个完整的汇编语言源程序至少含有一个代码段，但一个汇编语言源程序模块却可以只有数据段。目前，我们总把数据段安排在程序的前面，把代码段安排在程序的后面，但不是非要这样安排的。

此外，一个完整的程序还应该带有自己的堆栈段。但操作系统(DOS)在装载没有堆栈段的程序时，会指定一个堆栈段。由于堆栈段的安排比较固定，而且我们在全教程中所举的例子程序均较小，总可利用 DOS 安排的堆栈，所以为了简单，在以后所举各例子中均省去堆栈段。尽管连接程序在连接这种没有堆栈段的目标模块时，会发出一条警告信息，但可忽略它。

指令语句和数据定义伪指令语句应安排在段内。部分伪指令语句可安排在段外，例如，符号定义语句一般安排在源程序的开始处。在源程序的最后还要有源程序结束语句。

3. 源程序结束语句

源程序结束语句的一般格式如下：

```
END  [标号]
```

该语句告诉汇编程序，源程序到此为止。汇编程序在遇到该语句后，就不再对其后的任何语句进行汇编，所以，源程序结束语句往往是源程序的最后一条语句。

END 语句可带有一个已在程序中定义过的标号，这表示程序要从标号所对应的指令开始执行，也就是说，标号给定了程序的启动地址。

如果源程序是一个独立的程序，那么 END 语句应带有标号，从而指定程序的启动地址。如果源程序仅是一个模块，且不是主模块，则 END 语句不应带有标号。我们在第 8 章中介绍模块和主模块的概念。

3.4 顺序程序设计

CPU 在执行顺序程序片段时，按照指令的先后次序执行指令，因此在顺序程序片段中，指令的先后次序是至关重要的。在具体的顺序程序片段中，有些指令语句可以前后颠倒，有些则不行。此外，还要注意顺序的优化，做到充分利用前面的处理结果，尽量避免重复操作。本节介绍顺序程序设计的基本方法。

3.4.1 顺序程序举例

例 1：设 X 和 Y 均为 16 位无符号数，写一个求表达式 16X+Y 值的程序。

由于表达式中的 X 和 Y 是 16 位数，表达式的结果可能要超出 16 位，所以定义两个字变量用于保存 X 和 Y，另外用一个 32 位的双字变量来保存结果。数据段可定义如下：

```
;程序名：T3-2.ASM
;功  能：计算 16X+Y
DSEG    SEGMENT
XXX     DW      1234H           ;设 X 为 1234H
YYY     DW      5678H           ;设 Y 为 5678H
ZZZ     DD      ?               ;用于保存结果
DSEG    ENDS
```

CPU 对寄存器操作数的运算操作要比对存储器操作数的运算操作快得多，所以应尽量利用寄存器进行运算操作。用 DX 和 AX 寄存器保存中间结果，DX 保存高 16 位，AX 保存低 16 位。结合上述数据段，程序的代码段如下：

```
CSEG    SEGMENT
        ASSUME  CS：CSEG， DS：DSEG
START： MOV     AX,DSEG             ;(1)设置数据段寄存器值
        MOV     DS,AX
        MOV     AX,XXX              ;(2)把 X 送 AX 并扩展到 32 位
        XOR     DX,DX
        ADD     AX,AX               ;(3)计算 X * 16
        ADC     DX,DX
        ADD     AX,AX               ;X * 4
        ADC     DX,DX
        ADD     AX,AX               ;X * 8
        ADC     DX,DX
        ADD     AX,AX               ;X * 16
        ADC     DX,DX
```

```
        ADD     AX,YYY                  ;(4)在 X * 16 的结果上再加 Y
        ADC     DX,0
        MOV     WORD  PTR  ZZZ,AX       ;(5)保存结果
        MOV     WORD  PTR  ZZZ+2,DX
        MOV     AH,4CH                  ;(6)返回 DOS
        INT     21H
CSEG    ENDS
        END     START
```

上面的代码分为六步，必须依次执行。第(2)步和第(5)步中的两条指令语句可以分别颠倒，其它各步中的指令语句均不能颠倒。

第(3)步是计算 X * 16，采用了四次自身相加的方法达到乘 16 的目的。也可采用移位的方法达到乘 16 的目的，这样就不能简单地把 16 位扩展到 32 位，第(2)步和第(3)步可合并如下：

```
MOV     AX,XXX                  ;X 送 AX
MOV     DX,AX                   ;X 送 DX
MOV     CL,4
SHL     AX,CL                   ;AX 左移 4 位等价于乘 16
MOV     CL,12
SHR     DX,CL                   ;等价于得 AX 的高 4 位
```

还有一个更简单的方法计算 X * 16，那就是采用乘法指令。对应于第(2)步和第(3)步的代码可简化如下：

```
MOV     AX,XXX
MOV     DX,16
MUL     DX                      ;AX * DX->DX : AX
```

显然，代码长度越来越短，但这并不意味着执行速度越来越快，因为执行一条乘除法指令所花的时间较多。当乘数是 2 的倍数时，往往可用移位指令实现乘运算；当乘数较小时，往往可采用相加的方法实现乘运算；在对执行速度要求不大的情况下，可直接采用乘法指令实现乘运算。

例 2：写一个把压缩存放的 BCD 码，转换为对应十进制数字 ASCII 码的程序。

所谓压缩存放是指一个字节中存放两个 BCD 码，即低 4 位存放一个 BCD 码，高 4 位存放一个 BCD 码。如果仅仅是低 4 位存放一个 BCD 码，就称为非压缩存放。8421 BCD 码与对应十进制数字 ASCII 码的关系很简单，在非压缩 BCD 码上加 30H，就得对应十进制数字的 ASCII 码。源程序如下：

```
;程序名：T3-3.ASM
;功  能：压缩 BCD 码转换成 ASCII 码
DSEG    SEGMENT
BCD     DB86H                   ;假设的压缩 BCD 码
ASCII   DB2  DUP  (0)           ;存放 ASCII 码
```

```
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:DSEG
START:  MOV     AX,DSEG         ;(1)设置数据段寄存器值
        MOV     DS,AX
                                ;(2)把存放在低4位的BCD码转换为对应十进制
                                    数字的ASCII码
        MOV     AL,BCD
        AND     AL,0FH
        ADD     AL,30H
        MOV     ASCII+1,AL
                                ;(3)把存放在高4位的BCD码转换为对应十进制
                                    数字的ASCII码
        MOV     AL,BCD
        MOV     CL,4
        SHR     AL,CL
        ADD     AL,30H
        MOV     ASCII,AL
        MOV     AH,4CH          ;(4)返回DOS
        INT     21H
CSEG    ENDS
        END     START
```

上述代码段中的第(2)步和第(3)步可以颠倒，它们分别独立地把存放在低位和高位的 BCD 码转换为对应十进制数的 ASCII 码。

3.4.2 简单查表法代码转换

在汇编语言程序设计中，代码转换是经常的事。上面的把 BCD 码转换成 ASCII 码就是一例。对于各种不同代码之间的转换，往往要采用各种不同的方法，以便获得最佳效率。查表是实现代码转换的一种方法，下面介绍的简单查表方法是一种计算查表方法，适用于代码集合较小且转换关系复杂的场合。

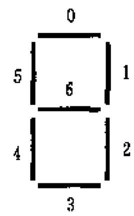


图 3.5 七段显示数码管示意图

例 3：写一个把 16 进制数字码转换为对应七段代码的程序。

利用如图 3.5 所示的七段显示数码管，能较好地显示 16 进制数字(0,...,9,A,b,C,d,E,F)。七段数码管的每一段对应一个二进制位，如果我们设 0 表示对应段亮，1 表示对应段暗，那么数字码 0 对应以二进制形式表示的代码 1000000，数字码 1 对应以二进形式表示的代码 1111001，如此，数字码 F 对应以二进制形式表示的代码 0001110。这种用于表示七段数码管亮暗的代码就称为七段代码。

显然，16 进制数字码与七段代码间的关系难以表示成一个简单的算术表达式，所以，

利用表的方法实现代码转换较合适。源程序如下：

```
;程序名：T3-4.ASM
;功  能：16进制数字码到七段代码的转换
DSEG    SEGMENT
TAB     DB 1000000B， 1111001B， 0100100B， 0110000B   ;七段代码表
        DB 0011001B， 0010010B， 0000010B， 1111000B
        DB 0000000B， 0010000B， 0001000B， 0000011B
        DB 1000110B， 0100001B， 0000110B， 0001110B
XCODE   DB 8                            ;假设的16进制数字码
YCODE   DB ?                            ;存放对应的七段代码
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  CS：CSEG， DS：DSEG
START： MOV     AX,DSEG
        MOV     DS,AX
        MOV     BL,XCODE        ;取16进制数字码
        AND     BL,0FH          ;保证在0至F之间
        XOR     BH,BH           ;表内偏移用16位表示以便寻址
        MOV     AL,TAB[BX]      ;取得对应的代码
        MOV     YCODE,AL        ;保存
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

利用查表的方法实现代码转换的关键是表的组织。上述程序中按16进制数字码的大小组织七段代码表，这就便于查找。可以说，实际上没有进行真正的查找，只是根据16进制数字码，确定对应代码在表中的位置而已。这种代码转换方法简明快捷。

8086/8088CPU还专门提供一条查表指令XLAT（也称为换码指令），以方便实现上述这种类型的查表。

查表指令XLAT的格式如下：

```
XLAT
```

该指令把寄存器BX的内容作为表（每项一字节）的首地址，把寄存器AL的值作为下标，取出表项内容再送AL寄存器。也就是把寄存器AL中的内容转换成表中对应项的值，此即所谓的换码。使用此指令前，应先把表的首地址送BX寄存器。表最大为256项。此指令属于数据传送指令组。

利用XLAT指令，上述程序T3-4.ASM的转换部分的代码可改写如下：

```
MOV     BX,OFFSET  TAB
MOV     AL,XCODE
```

```
AND     AL,0FH
XLAT
```

3.4.3 查表法求函数值

有许多数学函数的求值计算用汇编语言实现较为困难,除非利用数学协处理器。然而,上述这种表的组织形式和查表的方法,能够适用于直接获得某些数学函数的值。

例4:设X是一个1~10之间的整数,写一个求函数Y=LG(X)值的程序。

把1~10这10个数的对数值组织成一张表,那么程序运行时的计算工作就大大简化,甚至可以说没有具体的计算。由于1~10的以10为底的对数在0~1的范围之间,为了表示的方便和考虑一定的精度,所以把这些对数值放大10000倍,这样每个对数值就用一个字表示。源程序如下:

```
;程序名:T3-5.ASM
;功  能:求1到10的对数值
DSEG    SEGMENT
VALUE     DB 4                    ;假设的X
ANSWER    DW ?                    ;存放X的对数值
TAB       DW 0,3010,4771,6021,6990,7782,8451,9031,9542,10000     ;对数表
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME    CS:CSEG, DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     AL,VLAUE          ;取X值
        XOR     AH,AH             ;把X值用16位表示
        DEC     AX                ;得表项号(从1开始)
        ADD     AX,AX             ;得表项在表内的地址
        MOV     BX,AX             ;表内地址送基址寄存器
        MOV     AX,TAB[BX]        ;获得对数
        MOV     ANSWER,AX         ;保存
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

由于1的对数值是第一项,所以首先要自变量值X减1;又由于每个对数值用一个字表示,所以在计算表项在表内的地址时要乘2。上述程序没有考虑X超出范围的情况。

使用查表法求函数值有两个优点:(1)程序比较容易;(2)能够得到十进制或十六进制(或任何其他)格式的高精度函数值。其缺点也许不那么明显:(1)函数值必须事先安排好,因而有许多限制;(2)函数值的精度和准确性由程序员控制,而不是由数学函数决定,当数

据表的项较多时，难免有误差。

3.5 分支程序设计

几乎所有的程序都不是从头顺序地执行到尾，而是在处理中经常存在着判断，并根据某种条件的判定结果而转向不同的处理。这样程序就不再是简单地顺序执行，而是分成两个或多个分支。本节介绍分支程序设计的基本方法。

3.5.1 分支程序举例

程序分支的两种基本结构如图 3.6 所示，这两种结构分别对应高级语言中的 if 语句和 if-else 语句。在汇编语言中，一般利用条件测试指令和条件转移指令等实现简单的分支。

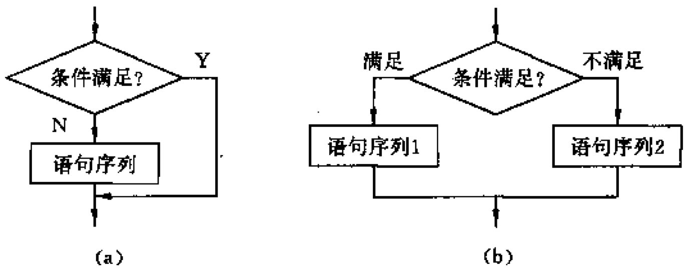


图 3.6 分支结构示意图

例 1：设有三个单字节无符号数存放在 BUFFER 开始的缓冲区中，写一个能将它们按大到小重新排列的程序。

设数据段就只有三个要排序是数据，定义如下：

```
;程序名：T3-6.ASM
;功  能：实现三个无符号数的由大到小的排序
DATAS   SEGMENT
BUFFER  DB 87,234,123
DATAS   ENDS
```

有多种方法可实现三个数的排序，我们采用交换法，先得到三个数中的最大数，然后再得到剩下两个数的最大数。为了方便，先把要排序的三个数取到三个寄存器中。源程序代码段如下所示：

```
CODES   SEGMENT
        ASSUME  CS:CODES,  DS:DATAS
START:  MOV     AX,DATAS
        MOV     DS,AX
        MOV     SI,OFFSET  BUFFER
```

```
        MOV     AL,[SI]           ;(1)把三个数取到寄存器中
        MOV     BL,[SI+1]
        MOV     CL,[SI+2]
        CMP     AL,BL             ;(2)排序
        JAE     NEXT1
        XCHG    AL,BL
NEXT1:  CMP     AL,CL
        JAE     NEXT2
        XCHG    AL,CL
NEXT2:  CMP     BL,CL
        JAE     NEXT3
        XCHG    BL,CL
NEXT3:  MOV     [SI],AL           ;(4)按大到小依次存回缓冲区
        MOV     [SI+1],BL
        MOV     [SI+2],CL
        MOV     AH,4CH
        INT     21H
CODES   ENDS
        END     START
```

在上述的排序片段中，含有三个分支，每个分支都是图 3.6(a)的结构。

另外，上述程序先把三个要排序的数取到三个寄存器中。当然也可以不这样做，而是直接在内存中交换排序。请把下面的程序片段与上述程序中的(2)、(3)和(4)步作一比较，孰优孰劣？

```
        MOV     SI,OFFSET  BUFFER
        MOV     AL,[SI]
        CMP     AL,[SI+1]
        JAE     NEXT1
        XCHG    AL,[SI+1]
        MOV     [SI],AL
NEXT1:  CMP     AL,[SI+2]
        JAE     NEXT2
        XCHG    AL,[SI+2]
        MOV     [SI],AL
NEXT2:  MOV     AL,[SI+1]
        CMP     AL,[SI+2]
        JAE     NEXT3
        XCHG    AL,[SI+2]
        MOV     [SI+1],AL
NEXT3:
```

例 2：写一个实现把一位十六进制数转换为对应 ASCII 码的程序。

十六进制数码与对应 ASCII 码的关系如下所示：

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
30H	31H	32H	33H	34H	35H	36H	37H	38H	39H	41H	42H	43H	44H	45H	46H

这种对应关系可表示为一个分段函数：

$$Y=\begin{cases} X+30H & (0<=X<=9) \\ X+37H & (0AH<=X<=0FH) \end{cases}$$

所以，程序要根据十六进制数码值是否超过 9 而进行分支。源程序如下：

```
;程序名：T3-7.ASM
;功  能：十六进制数到 ASCII 码的转换
DATA    SEGMENT
XX      DB 4                          ;假设的十六进制数码
ASCII   DB ?                          ;存放对应的 ASCII 码
DATA    ENDS
 ;
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     AL,XX
        AND     AL,0FH                ;确保在 0 至 F 之间
        CMP     AL,9                  ;(1)
        JA      LAB1                  ;(2)超过 9 转移
        ADD     AL,30H                ;(3)
        JMP     LAB2                  ;(4)
LAB1:   ADD     AL,37H                ;(5)
LAB2:   MOV     ASCII,AL              ;(6)
        MOV     AH,4CH
        INT     21H
CODE    ENDS
        END     START
```

上述程序中进行转换工作的指令(1)－(6)符合图 3.6(b)的结构，指令(3)的无条件转移语句(指令)很重要，如果没有这条指令，分支的结构就简化成为图 3.6(a)。

由于上述程序分支的一边稍作变形后，可包含分支的另一边，所以进行代码转换工作可作如下优化，从而使处理既简单又高效：

```
        ADD     AL,30H
        CMP     AL,39H
```

```
        JBE     LAB2
        ADD     AL,7
LAB2:
```

一般情况下,如果分支结构同图 3.6(b),且有一边很简单时,可考虑把它改变为图 3.6(a)的结构。具体的方法是,在判断之前先假设是简单的情况。

例 3: 写一个实现把一位十六进制数所对应的 ASCII 码转换为十六进制数的程序。如果要转换的 ASCII 码没有对应的十六进制数码,则转换为特殊值负 1。

考虑到 A--F 也能用小写字母表示,转换关系可用如下函数描述:

$$Y=\begin{cases} -1 & (X<'0') \\ X-'0' & ('0'<=X<='9') \\ -1 & ('9'<X<'A') \\ X-'A'+10 & ('A'<=X<='F') \\ -1 & ('F'<X<'a') \\ X-'a'+10 & ('a'<=X<='f') \\ -1 & (X>'f') \end{cases}$$

实现转换功能的源程序如下,其中对分支结构作了些优化处理:

```
;程序名: T3-8.ASM
;功  能: ASCII 码转换为十六进制数
DATA    SEGMENT
XX      DB  ?                   ;存放十六进制数
ASCII   DB  'a'                 ;假设的 ASCII 码
DATA    ENDS
 ;
CODE    SEGMENT
        ASSUME  CS:CODE, DS:DATA
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     AL,ASCII
        CMP     AL,'0'
LAB:    JB      LAB5            ;小于 '0' 转范围外处理
        MOV     AH,AL           ;设在 '0'—'9' 之间
        SUB     AH,'0'          ;转换处理
        CMP     AL,'9'
        JBE     LAB6            ;确在 '0'—'9' 之间转保存处
        CMP     AL,'A'
        JB      LAB5            ;小于 'A' 转范围外处理
        MOV     AH,AL           ;设在 'A'—'F' 之间
        SUB     AH,'A'-10
        CMP     AL,'F'
```

```
        JBE     LAB6            ;确在 'A'－'F' 之间转保存处
        CMP     AL,'a'
        JB      LAB5            ;小于 'a' 转范围外处理
        MOV     AH,AL           ;设在 'a'－'f' 之间
        SUB     AH,'a'－10
        CMP     AL,'f'
        JBE     LAB6            ;确在 'a'－'f' 之间转保存处
        ;
LAB5:   MOV     AH,－1          ;范围外处理
LAB6:   MOV     XX,AH           ;保存转换结果
        MOV     AH,4CH
        INT     21H
CODE    ENDS
        END     START
```

由于大写字母的 ASCII 码值与对应小写字母的 ASCII 码值间相差 20H，或者说大写字母 ASCII 码的位 5 为 0，而小写字母 ASCII 码的位 5 为 1，所以指令“AND AL，0DFH”能把 AL 中的小写字母的 ASCII 码值转换为对应大写字母的 ASCII 码，因此上述程序中进行代码转换的工作可进一步优化为：

```
        CMP     AL,'0'
LAB:    JB      LAB5            ;小于 '0' 转范围外处理
        MOV     AH,AL           ;设在 '0'－'9' 之间
        SUB     AH,'0'          ;转换处理
        CMP     AL,'9'
        JBE     LAB6            ;确在 '0'－'9' 之间转保存处
        AND     AL,11011111B    ;如为小写字母，则转换为大写字母
        CMP     AL,'A'
        JB      LAB5            ;小于 'A' 转范围外处理
        MOV     AH,AL           ;设在 'A'－'F' 之间
        SUB     AH,'A'－10
        CMP     AL,'F'
        JBE     LAB6            ;确在 'A'－'F' 之间转保存处
LAB5:   …
```

如果不考虑待转换的字符可能不是十六进制数码的情况，则程序要简单得多。通过指令“OR AL，20H”把 AL 寄存器中的大写字母 ASCII 码值转换为对应的小写字母的 ASCII 码值，如果 AL 中是数字符的 ASCII 码，也不会受影响。相应的代码如下：

```
        OR      AL,20H
        SUB     AL,'0'
        CMP     AL,9
        JB      LAB4
        SUB     AL,'a'－'0'－10
```

```
LAB4:    MOV      XX,AL
```

3.5.2 利用地址表实现多向分支

当要根据某个变量的值,进行多种不同处理时,就产生了多向分支。多向分支的结构如图 3.7 所示。在高级语言中,常用 switch 语句等实现多向分支。在汇编语言中,如何实现多向分支呢?

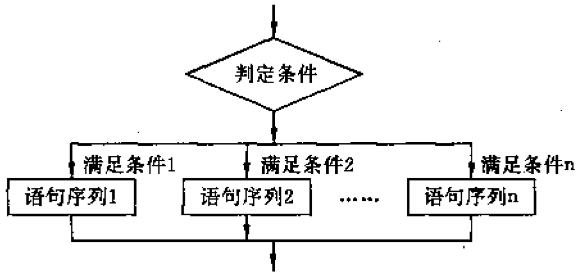


图 3.7 多向分支结构示意图

任何复杂的多向分支总可分解成多个简单分支。图 3.8 给出了根据 X 的值是否为 1～4,而进行 5 种不同处理的流程图片段。

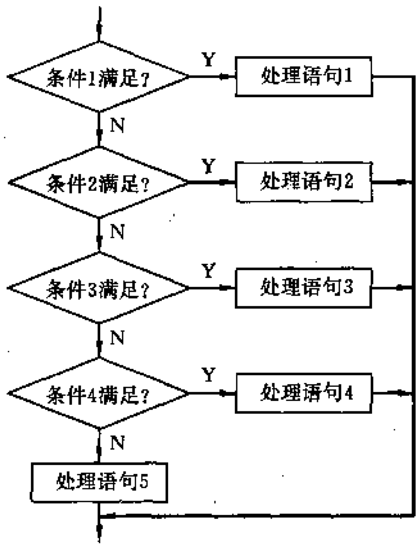


图 3.8 一种实现多向分支的流程图片段

用汇编语言实现这种多向分支的源程序结构如下所示:

```
        ……
        CMP     AH,1            ;设 X 在 AH 寄存器中
        JZ      YES_1           ;(*)
```

```
          JMP     NOT_1
YES_1:    ......
          ......
          JMP     OK
NOT_1:    CMP     AH,2
          JZ      YES_2           ;(*)
          JMP     NOT_2
YES_2:    ......
          ......
          JMP     OK
NOT_2:    CMP     AH,3
          JNZ     NOT_3           ;(**)
YES_3:    ......
          ......
          JMP     OK
NOT_3:    CMP     AH,4
          JNZ     NOT_4           ;(**)
YES_4:    ......
          ......
          JMP     SHORT  OK       ;(***)
NOT_4:    ......
          ......
OK:       ......
```

在上述说明多向分支程序结构的(*)语句处,无条件转移指令和条件转移指令配合实现分支(远距离条件转移);在(**)语句处,假设地址差能用一字节表示,故没有使用无条件转移指令;在(***)处也假设地址差能用一字节表示,故使用了SHORT。在实际的程序中,要根据具体情况选择使用合适的指令。

这种程序结构显得繁琐,如果要用它实现5路以上的多向分支,则更加复杂。在汇编语言中,可使用地址表实现多向分支。当多向分支在5路以上时,用地址表实现起来既方便又高效。

设程序MBRANCH每次只接收一个单键命令"A"至"H",然后根据命令进行相应的处理。如果接受到的输入不是规定的命令字母,则不处理。

为了利用入口地址表(也称散转表)实现多向分支,事先必须安排一张入口地址表。如果各处理程序均在同一代码段内,则入口地址只要用偏移表示,所以入口地址表的每一项只用一个字。MBRANCH的地址表可如下组织:

```
DSEG    SEGMENT
        ......                                  ;定义的其他数据
COMTAB  DW  COMA,COMB,COMC,COMD
        DW  COME,COMF,COMG,COMH                 ;入口地址表
        ......                                  ;定义的其他数据
```

```
DSEG    ENDS
```

有了上述地址表后，多向分支的实现是方便的。采用地址表实现多向分支的 MBRANCH 源程序的有关代码如下所示：

```
        ……
        MOV     AH,1
        INT     21H             ;接收键盘命令，命令代码在 AL 中
        ;
        AND     AL,11011111B    ;小写转大写
        CMP     AL,'A'          ;判是否起"A"--"H"
        JB      OK
        CMP     AL,'H'
        JA      OK
        ;
        SUB     AL,'A'          ;* * 把命令字母键转换成序号(从 0 开始)
        XOR     AH,AH           ;* *
        ADD     AX,AX           ;* * 计算入口地址表内的地址
        MOV     BX,AX           ;* *
        JMP     COMTAB[BX]      ;* * 转对应命令处理程序
        ;
OK:     ……
        MOV     AH,4CH          ;结束
        INT     21H
        ;
COMA:   ……                      ;命令 A 处理程序
        ……
        JMP     OK
        ;
COMB:   ……                      ;命令 B 处理程序
        ……
        JMP     OK
        ;
COMC:   ……                      ;命令 C 处理程序
        ……
        JMP     OK
        ;
        …………                    ;其他命令处理程序
…………
        ;
COMH:   ……                      ;命令 H 处理程序
        ……
        JMP     OK
```

上述代码中真正实现多向分支的部分是注释行带(* *)的指令。

通过地址表实现多向分支的关键是，根据分支各路的条件确定对应处理程序的入口地址在地址表中的位置，或者说对应处理程序的编号。在程序 MBRANCH 中，代表命令的字母是连续的，所以很容易由命令字母得出对应处理程序的编号。如果命令字母不连续，则稍稍复杂些。

3.6 循环程序设计

当要重复某些操作时，就应考虑使用循环。循环通常由四部分组成：(1)初始化部分；(2)循环体部分；(3)调整部分；(4)控制部分。各部分之间的关系如图 3.9 所示。图 3.9(a)是先执行后判断的结构，图 3.9(b)是先判断后执行的结构。有时这四部分可以简化，形成互相包含交叉的情况，不一定能明确分成四部分。有多种方法可实现循环的控制，常用的有计数控制法和条件控制法等。本节介绍循环程序设计的基本方法。

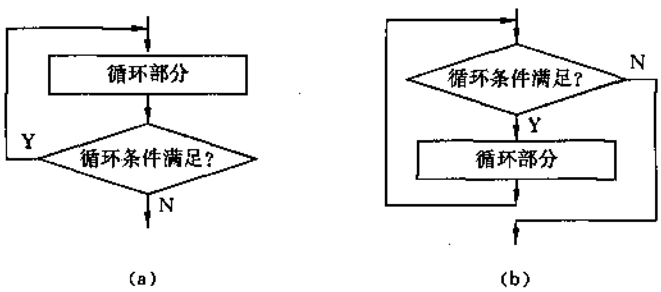


图 3.9 循环结构示意图

3.6.1 循环程序举例

在第 2 章中介绍的若干数据求和等程序片段均采用了循环结构，下面再举几例，说明实现循环的基本方法。

例 1：求内存中从地址 0040：0000H 开始的 1024 个字的字检验和。

所谓字检验和是指，结果只用字表示，忽略可能产生的进位。在数据传输时，为了保证数据传输的正确性，一般要对传输的数据进行某种检查，常用的简单检查方法是检查数据的按字节或字累计的和。

程序采用如图 3.9(a)所示的循环结构，这个循环的四部分俱全。在初始化部分设置存储器指针和循环计数器等；循环体部分只要实现累加求和，并不要考虑进位；存储器指针的调整作为循环调整部分的内容；采用计数法控制循环。源程序如下：

```
;程序名: T3-9.ASM
;功  能: 说明根据计数法控制循环
DSEG    SEGMENT
SUM     DW  ?                        ;存放检验和
```

```
DSEG    ENDS
CSEG    SEGMENT
        ASSUME CS：CSEG
        ;(1)初始化部分
START:  MOV     AX,40H
        MOV     DS,AX           ;设置数据段寄存器值为 0040H
        MOV     SI,0            ;设置偏移为 0
        MOV     CX,1024         ;设置循环计数器
        XOR     AX,AX           ;检验和清 0
        ;(2)循环体部分
AGAIN:  ADD     AX,[SI]         ;求和
        ;(3)调整部分
        INC     SI              ;修改存储器地址指针
        INC     SI
        ;(4)循环控制部分
        LOOP    AGAIN           ;根据 CX 倒计数
        ;
        ASSUME DS：DSEG
        MOV     BX,DSEG
        MOV     DS,BX
        MOV     SUM,AX          ;保存检验和
        MOV     AH,4CH          ;返回 DOS
        INT     21H
CSEG    ENDS
        END     START
```

在初始化部分设置数据段寄存器 DS 之值为 40H，这与先前的程序稍有不同，实际上，根据要求被处理的数据已明确在 40H 段。在得到检验和之后，重新设置数据段寄存器 DS 之值为程序中定义的 DSEG 段的段值，以便保存检验和。

例 2：不利用乘法指令实现乘法运算。

为了简单，设乘数和被乘数均是单字节无符号整数。举本例的目的是为了说明设计循环程序的基本方法，并非提倡不利用乘法指令。另一方面确有不能使用乘法指令的场合，例如 Z80 的指令集中无乘法指令，那么只有用其他方法实现乘法运算了。

下面的程序采用累加的方法实现乘法，也就是把被乘数累加乘数次得到积。由于设被乘数和乘数均是单字节无符号整数，所以积只用一个字表示。源程序如下：

```
;程序名：T3-10.ASM
;功  能：(略)
DSEG    SEGMENT
XXX     DB      234             ;假设的被乘数
YYY     DB      125             ;假设的乘数
ZZZ     DW      ?               ;存放积
```

```
DSEG    ENDS
CSEG    SEGMENT
        ASSUME  CS：CSEG， DS：DSEG
START： MOV     AX,DSEG
        MOV     DS,AX
        ;(1)初始化部分
        MOV     AL,XXX
        XOR     AH,AH           ;为了累加方便,用16位表示被乘数
        MOV     CL,YYY
        XOR     CH,CH           ;用CX存放循环计数(乘数)
        XOR     DX,DX           ;清累加器
        JCXZ    OK              ;如果乘数为0,则不必循环
        ;(2)循环体部分
MULTI： ADD     DX,AX
        ;(3)循环控制部分
        LOOP    MULTI
        ;
OK：    MOV     ZZZ,DX          ;保存积
        MOV     AH,4CH          ;返回DOS
        INT     21H
CSEG    ENDS
        END     START
```

在上述程序的循环中，只有循环的三部分，而缺少循环调整部分。为了使循环体简单，所以把被乘数扩展成16位，从而可直接采用16位的累加。如果只采用8位累加，则循环体可修改如下：

```
        ADD     DL,AL
        ADC     DH,0
```

采用累加的方法实现乘法运算虽然简单，但当乘数稍大时将花较多的时间。一般可采用移位相加的方法实现乘法运算，移位相加的方法类似于手算。图3.10是一种移位相加法实现乘法运算的流程图，其中被乘数左移，乘数右移。图3.10符合图3.9(a)所示的循环结构。

实现图3.10所示流程图的源程序片段如下所示：

```
        ……
        MOV     AL,XXX
        XOR     AH,AH           ;取被乘数,并扩展到16位
        MOV     BL,YYY          ;取乘数
        XOR     DX,DX           ;积清0
        MOV     CX,8            ;设置循环计数器(单字节无符号,故8次循环)
        ;
```

```
MULTI: SHR    BL,1          ;乘数右移 1 位
       JNC    NEXT          ;判是否要把被乘数加到积
       ADD    DX,AX         ;加到积上
NEXT:  ADD    AX,AX         ;被乘数左移 1 位
       LOOP   MULTI         ;循环控制
;
       MOV    ZZZ,DX        ;保存积
       ……
```

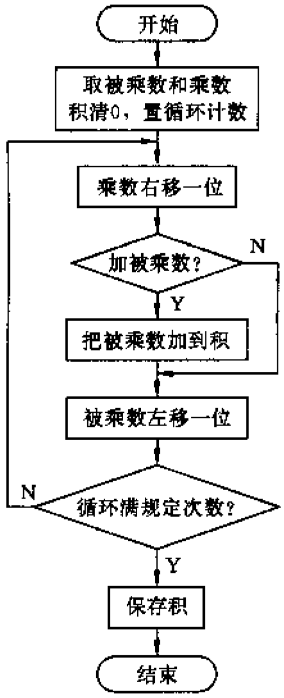


图 3.10 移位相加实现乘法的流程图

例 3：把 16 位二进制数转换为 5 位十进制数。为了简单，设二进制数是无符号的，采用 8421BCD 码表示十进制数。

16 位二进制数能表示的最大十进制数只有 5 位，即最高位是“万位”。有多种转换方法。这里采用的方法是：先把二进制数除以 10000，得到的商即为十进制数的“万位”；再用余数除以 1000，得到的商为十进制数的“千位”；按照这样的顺序，分别用每次的余数除以 100、10、1 得到的商分别为十进制数的“百位”、“十位”和“个位”。每次除后得到的商就是 8421BCD 码。

我们用一个循环次数确定为 5 的循环来实现转换。在每次循环中，先获取新的除数；然后进行除法；保存商；由于使用 16 位除数，还要把除后的余数调整为 32 位作为新的被除数。源程序如下：

```
;程序名：T3-11.ASM
;功  能：(略)
DSEG    SEGMENT
DATA    DW      23456                   ;假设的二进制数
BUFFER  DB      5  DUP  (0)             ;准备存放十进制数
JM      DW      10000,1000,100,10,1     ;5个除数
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:DSEG
START:  MOV     AX,SEG  DATA
        MOV     DS,AX
        MOV     DI,OFFSET JM            ;置指向除数的指针初值
        MOV     SI,OFFSET BUFFER        ;置缓冲区指针初值
        MOV     CX,5                    ;置循环次数
        MOV     AX,DATA                 ;取二进制数
        XOR     DX,DX                   ;扩展为32位
NEXT:   MOV     BX,[DI]                 ;取除数
        ADD     DI,2
        DIV     BX                      ;进行除
        MOV     [SI],AL                 ;保存商
        INC     SI
        MOV     AX,DX                   ;调整余数为新的被除数
        XOR     DX,DX
        LOOP    NEXT                    ;计数循环控制
        MOV     AX,4C00H                ;返回DOS
        INT     21H
CSEG    ENDS
        END     START
```

如果要把各位十进制数字用对应的 ASCII 码表示，那么只需在除法指令和保存商指令之间加入指令“ADD AL,30H”，该指令实现把 AL 寄存器中的 BCD 码转换为对应的 ASCII 码。

例 4：写一个把字符串中的所有大写字母改为小写的程序。设字符串以 0 结尾。

图 3.11 是实现的流程图，具有图 3.9(b)所示的结构。它的主体是一个循环次数不确定(字符串长不确定)，根据是否到达字符串尾这个条件来控制的循环。源程序如下，数据段中的字符串起示例作用。

```
;程序名：T3-12.ASM
;功  能：说明根据某个条件控制的循环
DSEG    SEGMENT
STRING  DB  'HOW are yoU ! ',0      ;假设的字符串
DSEG    ENDS
```

```
;
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     SI,OFFSET STRING    ;取字符串开始地址
AGAIN:  MOV     AL,[SI]             ;取一字符
        OR      AL,AL               ;是否到字符串尾?
        JZ      OK                  ;到字符串尾,转
        CMP     AL,'A'              ;否则,判是否为大写字母
        JB      NEXT                ;否,转继续
        CMP     AL,'Z'
        JA      NEXT                ;否,转继续
        OR      AL,20H              ;是大写字母,改为小写字母
        MOV     [SI],AL             ;送回到字符串中
NEXT:   INC     SI                  ;调整指针
        JMP     AGAIN               ;继续
OK:     MOV     AX,4C00H            ;结束
        INT     21H
CSEG    ENDS
        END     START
```

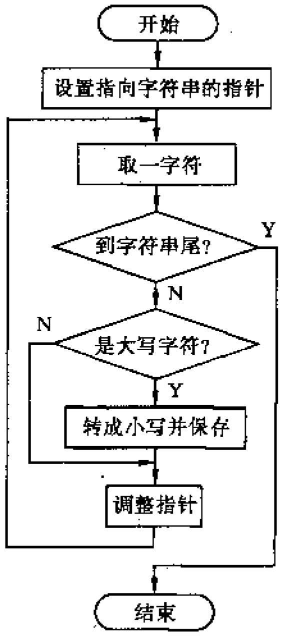


图 3.11 字符串中的字母小写化的程序流程图

上述的循环采用先判后执行的结构。此外,为了更有效,可作如下变形:

```
        MOV     SI,OFFSET STRING    ;取字符串开始地址
AGAIN:  MOV     AL,[SI]             ;取一字符
```

```
INC     SI              ;调整指针
OR      AL,AL           ;是否到字符串尾?
JZ      OK              ;到字符串尾,转
CMP     AL,'A'          ;否则,判是否为大写字母
JB      AGAIN           ;(*)
CMP     AL,'Z'
JA      AGAIN           ;(*)
ADD     AL,20H          ;是大写字母,改为小写字母
MOV     [SI-1],AL       ;送回到字符串中
JMP     AGAIN           ;继续
```

例 5:写一个程序判定从地址 0040:0000H 开始的 2048 个内存字节单元中是否有字符 'A'。如有则把第一个(按地址由小到大为序)含此指定字符的存储单元的地址偏移送到 0000:03FEH 单元中;如没有则把特征值 0FFFFH 送上述指定单元。

图 3.12 是流程图,含有一个由计数和条件双重控制的一个循环,也就是最多循环 N 次,在 N 次循环过程中,如特定条件满足,则提前结束循环。

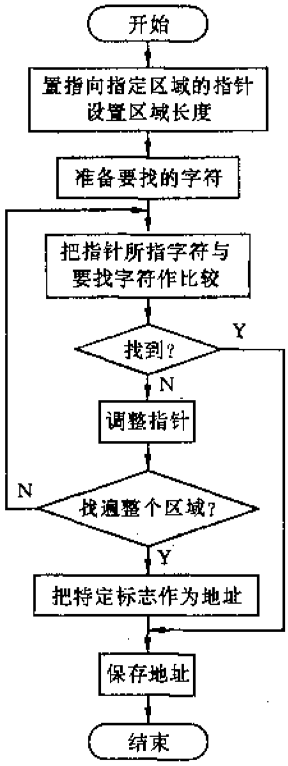


图 3.12　在规定内存范围内找指定字符的流程图

实现上述流程的源程序不需要数据段。为了使程序便于阅读和修改,在程序开始定义了若干符号常量。

```
程序名： T3-13.ASM
功  能： 说明由计数和条件双重控制的循环
;常量定义
SEGADDR    = 40H                  ;开始地址段值
OFFADDR    = 0                    ;开始地址偏移
COUNT      = 2048                 ;长度(计数)
KEYCHAR     = 'A'                 ;指定字符
SEGRESU     = 0                   ;结果保存单元段值
OFFRESU     = 3FEH                ;结果保存单元偏移
;代码段
CSEG    SEGMENT
        ASSUME  CS:CSEG
START:  MOV     AX,SEGADDR        ;初始化
        MOV     DS,AX
        MOV     SI,OFFADDR
        MOV     CX,COUNT
        MOV     AL,KEYCHAR
NEXT:   CMP     AL,[SI]           ;找
        JZ      OK                ;找到转
        INC     SI                ;调整指针
        LOOP    NEXT              ;继续下一次
        MOV     SI,0FFFFH         ;没有找到时的特征值
OK:     MOV     AX,SEGRESU
        MOV     DS,AX
        MOV     BX,OFFRESU
        MOV     [BX],SI           ;结果送指定单元
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

例 6：设缓冲区 DATA 中有一组单字节有符号数，以 0 为结束标志。写一个程序实现如下功能：把前 5 个正数依次送入缓冲区 PDATA，把前 5 个负数依次送入缓冲区 MDATA；如正数或负数不足 5 个，则用 0 补足。

在把正数和负数送入对应缓冲区前，用 0 填正数和负数缓冲区，从而实现"如正数或负数不足 5 个，则用 0 补足"的要求。这通过一个循环次数确定的循环完成。然后，获取前 5 个正数和负数，这也用一个循环结构来实现。第二个循环的结束条件是：遇到结束标志，或者获取的正数个数和负数个数均已为 5。图 3.13 给出了流程图。

我们用变址寄存器 SI 和 DI 作为已获取的正数和负数的计数器，这样做不仅可方便地把获得的正数和负数送入相应的缓冲区，而且有利于统计已取得的正数和负数的个数。源程序如下所示。

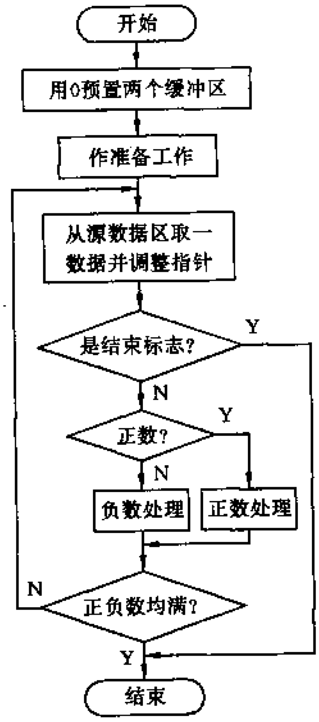


图 3.13 获取前若干个正数和负数的流程图

```
;程序名: T3-14.ASM
;功  能: (略)
MAX_COUNT    =    5
DSEG    SEGMENT
DATA    DB 3,-4,5,6,-7,8,-9,-10,-1,-32,-123,27,58,44,-12,0
PDATA   DB MAX_COUNT  DUP  (?)   ;存放正数
MDATA   DB MAX_COUNT  DUP  (?)   ;存放负数
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     CX,MAX_COUNT     ;清存放正数和负数的缓冲区
        MOV     SI,OFFSET  PDATA
        MOV     DI,OFFSET  MDATA
        MOV     AL,0
NEXT1:  MOV     [SI],AL
        MOV     [DI],AL
        INC     SI
```

```
        INC     DI
        LOOP    NEXT1
        ;获取前若干个正数和负数
        MOV     BX,OFFSET  DATA  ;置指针初值
        XOR     SI,SI
        XOR     DI,DI
NEXT2:  MOV     AL,[BX]          ;取一个数据
        INC     BX
        CMP     AL,0             ;与 0 比较
        JZ      OVER             ;是结束标志,就结束
        JG      PLUS             ;为正数,则转
        CMP     DI,MAX_COUNT     ;负数处理
        JAE     CONT
        MOV     MDATA[DI],AL
        INC     DI
        JMP     SHORT CONT
PLUS:   CMP     SI,MAX_COUNT     ;正数处理
        JAE     CONT
        MOV     PDATA[SI],AL
        INC     SI
CONT:   MOV     AX,SI            ;判是否已获得足够的正数和负数
        ADD     AX,DI
        CMP     AX,MAX_COUNT+MAX_COUNT
        JB      NEXT2
OVER:   MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

3.6.2 多重循环程序举例

所谓多重循环就是循环之中还有循环。

例 7：设 BUFFER 缓冲区中有 10 个单字节无符号整数，写一个程序将它们由小到大排序。

有各种各样的排序算法，这里为了方便地说明二重循环，采用"简单选择"法，图 3.14 是流程图。

源程序如下，其中 SI 相当于外层循环控制变量 I，DI 相当于内存循环控制变量 J，为了使 I 从 1 开始递增，排序数组开始地址先减 1 后再存入 BX 寄存器。

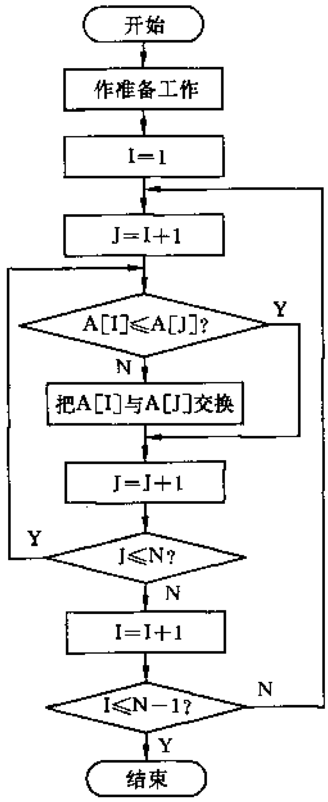


图 3.14　排序算法流程图

```
;程序名：T3-15.ASM
;功  能：说明二重循环的实现
DSEG    SEGMENT
BUFFER DB 23, 12, 45, 32, 127, 3, 9, 58, 81, 72  ;假设的 10 个数据
N       EQU    10                                ;定义符号 N 为常数 10
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME CS:CSEG,DS:DSEG
START:  MOV    AX,DSEG
        MOV    DS,AX
        MOV    BX,OFFSET  BUFFER -1   ;设置缓冲区开始地址
        MOV    SI,1                   ;I=1
FORI:   MOV    DI,SI
        INC    DI                     ;J=I+1
FORJ:   MOV    AL,[BX+SI]
        CMP    AL,[BX+DI]             ;A[I]与 A[J]比较
        JBE    NEXTJ                  ;A[I]小于等于 A[J]转
```

```
        XCHG    AL,[BX+DI]              ;A[I]与 A[J]交换
        MOV     [BX+SI],AL
NEXTJ:  INC     DI                      ;J=J+1
        CMP     DI,N
        JBE     FORJ                    ;J<=N 时转
NEXTI:  INC     SI                      ;I=I+1
        CMP     SI,N - 1
        JBE     FORI                    ;I<=N-1 时转
        ;
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

读者能否在保持算法的前提下，对上述程序作些优化工作。

例 8：设字符串 1 在数据段 1 中，字符串 2 在数据段 2 中，写一程序判别字符串 2 是否是字符串 1 的子字符串。如是子字符串，则把数据段 2 中的 FLAG 单元置 1，否则将其清 0。设字符串以 0 结尾。

判别一个字符串是否是另一字符串的子字符串的方法很多，我们选取实现较简单的一种算法，源程序如下。

```
;程序名：T3-16.ASM
;功  能：(略)
DSEG1   SEGMENT
STRM    DB "THIS IS A STRING!",0        ;假设的在数据段 1 中的字符串
DSEG1   ENDS
;
DSEG2   SEGMENT
STRS    DB "STRING",0                   ;假设的在数据段 2 中的字符串
FLAG    DB ?
DSEG2   ENDS
;
CODE    SEGMENT
        ASSUME  CS:CODE, DS:DSEG1, ES:DSEG2
START:  MOV     AX,DSEG1
        MOV     DS,AX                   ;数据段 1 段值置 DS
        MOV     AX,DSEG2
        MOV     ES,AX                   ;数据段 2 段值置 ES
        ;
        MOV     DI,OFFSET STRS          ;测字符串 2 的长度
        MOV     BX,DI                   ;保存字符串 2 首地址
        XOR     CX,CX                   ;清计数器
```

```
        DEC     DI
WHILE1: INC     DI                          ;调整指针
        INC     CX                          ;计数先加 1
        CMP     BYTE  PTR  ES: [DI],0       ;字符串 2 是否结束
        JNZ     WHILE1
        DEC     CX                          ;得字符串 2 长度
        MOV     DX,CX                       ;保存
        ;
        MOV     SI,OFFSET STRM              ;取字符串 1 首地址
        MOV     BP,SI
FORI:   MOV     CX,DX                       ;置要比较的字符个数
        MOV     DI,BX                       ;置首地址
FORJ:   MOV     AL,ES: [DI]
        CMP     [SI],AL                     ;比较一字节
        JNZ     NEXTI                       ;不等,从字符串 1 的下一个字符开始
NEXTJ:  INC     DI
        INC     SI
        LOOP    FORJ                        ;继续下一字符的比较
        MOV     FLAG,1                      ;置是子字符串标志
        JMP     OVER
NEXTI:  CMP     BYTE  PTR  [SI],0           ;判字符串 1 是否结束
        JZ      NOTF                        ;是,转结束
        INC     BP
        MOV     SI,BP
        JMP     FORI
NOTF:   MOV     FLAG,0                      ;置非子字符串标志
        ;
OVER:   MOV     AH,4CH                      ;程序结束
        INT     21H
CODE    ENDS
        END     START
```

另请注意上面程序中 ASSUME 语句和给 FLAG 单元赋值的语句。

3.7 习　　题

题 3.1　伪指令语句与指令语句的本质区别是什么？伪指令的主要作用是什么？

题 3.2　汇编语言中的表达式与高级语言中的表达式有何相同点和不同点？

题 3.3　汇编语言中数值表达式与地址表达式有何区别？

题 3.4　汇编语言中的变量和标号有何异同之处？

题 3.5 请计算如下各数值表达式的值：

(1) 23H AND 45H OR 67H　　(2) 1234H / 16 +10H

(3) NOT (65535 XOR 1234H)　　(4) 1024 MOD 7 + 3

(5) LOW 1234 OR HIGH 5678H　　(6) 23H SHL 4

(7) "Eb" GE 4562H XOR −1　　(8) 1234H SHR 6

(9) 'a' AND (NOT ('a' − 'A'))　　(10) 'H' OR 00100000B

(11) 76543Q LT 32768 XOR 76543　　(12) 3645H AND 0FF00H

题 3.6 请计算如下程序片段中各地址表达式的值，设 BX＝1000H，SI＝2000H，DI＝3000H，BP＝4000H：

(1) [BX+100H]　　(2) [DI][BP]

(3) 2000H[SI]　　(4) 10H[BX][SI]

(5) [BP−128]　　(6) [BX][DI−2]

题 3.7 设在某个程序中有如下片段，请写出每条传送指令执行后寄存器 AX 的内容：

```
        ORG   100H
VARW    DW    1234H,5678H
VARB    DB    3,4
VARD    DD    12345678H
BUFF    DB    10  DUP (?)
MESS    DB    'HELLO'
BEGIN:  MOV   AX,OFFSET  VARB  +  OFFSET  MESS
        MOV   AX,TYPE  BUFF  +  TYPE  MESS  +  TYPE  VARD
        MOV   AX,SIZE  VARW  +  SIZE  BUFF  +  SIZE  MESS
        MOV   AX,LENGTH  VARW  +  LENGTH  VARD
        MOV   AX,LENGTH  BUFF  +  SIZE  VARW
        MOV   AX,TYPE  BEGIN
        MOV   AX,OFFSET  BEGIN
```

题 3.8 设如下两条指令中的符号 ABCD 是变量名，请说明这两条指令的异同。

```
MOV     AX,OFFSET  ABCD
LEA     AX,ABCD
```

题 3.9 请指出如下指令中的不明确之处，并使其明确：

(1) MOV ES:[BP],5　　(2) ADD CS:[1000H],10H

(3) DEC SS:[BX−8]　　(4) JMP CS:[SI+1000H]

(5) MUL [BX+DI+2]　　(6) DIV [BP−4]

题 3.10 设在某个程序中有如下片段，请改正其中有错误的指令语句：

```
VARW    DW      1234H,5678H
VARB    DB      3,4
```

```
VARD    DD      12345678H
        ......
        MOV     AX,VARB
        MOV     VARD,BX
        MOV     VARD+2,ES
        MOV     CL,VARW+3
        LES     DI,VARW
```

题 3.11 请举例说明伪指令 ASSUME 的作用。

题 3.12 设在某个程序片段中有如下语句，请说明各符号的属性：

```
SYMB1   LABEL   BYTE
SYMB2   EQU     THIS   BYTE
SYMB3   DW      ?
SYMB4   EQU     BYTE   PTR   SYMB3
```

题 3.13 为什么说汇编语言中的等价语句 EQU 可理解为简单的宏定义？请举例说明。

题 3.14 设在某个程序片段中有如下语句，请说明各符号所表示的值：

```
SYMB1   =   10
SYMB2   =   SYMB1 * 2
SYMB1   =   SYMB1+SYMB2 + 4
SYMB3  EQU  SYMB1
```

题 3.15 请改写 3.3.3 的程序 T3-1.ASM，使其只有一个段。

题 3.16 请说明指令"JMP　　$ +2"指令的机器码中的地址差值是多少。

题 3.17 源程序是否一定要以 END 语句结束？程序是否一定从代码段的偏移 0 开始执行？如果不是，那么如何指定？

题 3.18 利用查表的方法实现代码转换有何特点？利用查表的方法求函数值有何特点？

题 3.19 利用地址表实现多向分支有何特点？请举例说明。

题 3.20 请举例说明如何避免条件转移超出转移范围。

题 3.21 请写一个程序片段统计寄存器 AX 中置 1 位的个数。

题 3.22 设一个 32 位有符号数存放在 DX：AX 中，请写一个求其补码的程序片段。

题 3.23 写一个程序片段实现如下功能：依次重复寄存器 AL 中的每一位，得到 16 位的结果存放到 DX 寄存器中。

题 3.24 写一个程序片段实现如下功能：依次重复四次寄存器 AL 中的每一位，得到 32 位的结果存放到 DX：AX 寄存器中。

题 3.25 写一个程序片段实现如下功能：把寄存器 AL 和 BL 中的位依次交叉，得到的 16 位结果存放到 DX 寄存器中。

题 3.26 写一个优化的程序片段，实现把字符串中的小写字母变换为对应的大写字母。设字符串以 0 结尾。

题 3.27 写一个优化的程序片段，统计字符串的长度。设字符串以 0 结尾。

题 3.28 写一个程序片段，滤去某个字符串中的空格符号(ASCII 码 20H)。设字符串以 0 结尾。

题 3.29 请写一个把两个字符串合并的示例程序。

题 3.30 请写一个可把某个字变量的值转换为对应二进制数 ASCII 码串的示例程序。

题 3.31 请写一个可把某个十进制数 ASCII 码串转换为对应非压缩 BCD 和压缩 BCD 的示例程序。

题 3.32 请写一个可把某个十进制数 ASCII 码串转换为对应二进制数的示例程序。

题 3.33 请写一个可把某个十六进制数 ASCII 码串转换为对应二进制数的示例程序。

题 3.34 请写一个实现数据块移动的示例程序。

题 3.35 请编一个程序求从地址 F000：0000H 开始的 64K 字节内存区域的检验和，并转换为十六进制数的 ASCII 码串。

题 3.36 设已在地址 F000：0000H 开始的内存区域安排了 100 个字节的无符号 8 位二进制数。请编写一个程序求它们的和，并转换为对应十进制数的 ASCII 码串。

题 3.37 设已在地址 F000：0000H 开始的内存区域安排了 1024 个 16 位有符号数。请编写一个程序统计其中的正数、负数和零的个数，并分别转换为对应十进制数的 ASCII 码串。

题 3.38 设从地址 F000：0000H 开始的内存区域是缓冲区，存放了一组单字节的正数或负数，以 0 结尾。请编写一个程序确定其中最大的正数和最小的负数。

题 3.39 设从地址 F000：0000H 开始的 1K 字节内存区域是缓冲区。请写一个可收集该区域内所有子串"OK"开始地址的程序。

题 3.40 请优化 3.6.2 节例 7 所示排序程序。

题 3.41 请考虑 TC 或 BC 中 for 语句和 while 语句的实现方法。

题 3.42 请考虑 TC 或 BC 中 switch 语句的实现方法。

第4章　子程序设计和DOS功能调用

本章先介绍如何设计汇编语言子程序，然后把DOS功能调用视为子程序，介绍如何利用DOS系统功能调用。最后介绍子程序的递归和重入概念。

4.1　子程序设计

如果某个程序片段将反复在程序中出现，就把它设计成子程序。这样能有效地缩短程序长度、节约存储空间。如果某个程序片段具有通用性，可供许多程序共享，就把它设计成子程序。这样能大大减轻程序设计的工作量，例如标准函数程序。此外，当某个程序片段的功能相对独立时，也可把它设计成子程序，这样便于模块化，也便于程序的阅读、调试和修改。

在80x86系列汇编语言中，子程序常常以过程的形式出现。

4.1.1　过程调用和返回指令

过程调用指令和过程返回指令属于程序控制指令这一组。通常，过程调用指令用于由主程序转子程序，过程返回指令用于由子程序返回主程序。由于程序的代码可分为多个段，所以，像无条件转移指令一样，过程调用指令有段内调用和段间调用之分，与之相对应，过程返回指令也有段内返回和段间返回之分。把段内调用和段内返回称为近调用和近返回，把段间调用和段间返回称为远调用和远返回。在汇编语言中，过程也有远近类型之分。

1. 过程调用指令

过程调用指令首先把子程序的返回地址(即CALL指令的下一条指令的地址)压入堆栈，以便执行完子程序后返回调用程序(主程序)继续往下执行。然后转移到子程序的入口地址去执行子程序。按照转移目标是否在同一段来分，调用指令分为段内调用和段间调用；按照获得转移目标地址的方式来分，调用指令分为直接调用和间接调用。下面介绍这四种调用指令，在汇编语言中，均用指令助记符CALL表示。

过程调用指令不影响标志。

(1) 段内直接调用

段内直接调用指令用于调用当前段内的子程序，格式如下：

```
CALL    过程名
```

例如：

```
CALL    SUB1                  ;SUB1是近过程
CALL    TOASCII               ;TOASCII是近过程
```

该指令进行的具体操作分解如下：

```
SP     ⇐  SP-2
[SP]   ⇐  IP
IP     ⇐  IP+disp
```

段内直接调用指令只把返回地址的偏移部分压入堆栈保存，堆栈变化如图 4.1 所示。实际上，转移发生在同一段内，代码段寄存器 CS 的内容不发生变化。然后把返回地址与子程序入口地址的差值(disp)加到指令指针 IP 上，使 IP 之内容为目标地址偏移，从而达到转移的目的。与无条件段内直接转移指令相似，段内直接调用指令的转移是相对转移，指令由操作码和地址差构成。地址差(disp)等于目标地址到 CALL 指令下一条指令开始地址的差，汇编程序在汇编时计算出 disp。在段内直接调用指令中，总用一个字表示 disp，所以转移范围可达－32768～＋32767。

(2) 段内间接调用

段内间接调用指令也用于调用当前段内的子程序，格式如下：

```
CALL    OPRD
```

OPRD 是 16 位通用寄存器或字存储器操作数。该指令进行的具体操作分解如下：

```
SP     ⇐  SP-2
[SP]   ⇐  IP
IP     ⇐  (OPRD)
```

该指令只把返回地址的偏移部分压入堆栈保存，堆栈变化如图 4.1 所示。如 OPRD 是 16 位通用寄存器操作数，则把寄存器之内容送 IP；如 OPRD 是字存储器操作数，则把字存储单元之内容送 IP。

例如：

```
CALL    BX
CALL    WORD  PTR  [BX]
CALL    VARW                 ;VARW 是字变量
```

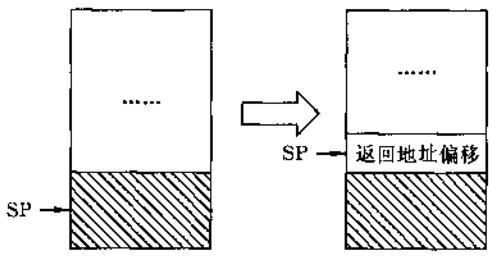


图 4.1 执行段内调用指令时的堆栈变化

(3) 段间直接调用

段间直接调用指令用于调用其它代码段中的子程序。格式如下：

```
CALL      过程名
```

该指令先把返回地址的段值压入堆栈，再把返回地址的偏移压入堆栈，达到保存返回地址的目的，堆栈的变化如图 4.2 所示。然后把过程的入口地址的段值和偏移分别送入 CS 和 IP，达到转移的目的。该指令进行的具体操作分解如下：

```
SP     ⇐   SP-2
[SP]   ⇐   CS
SP     ⇐   SP-2
[SP]   ⇐   IP
IP     ⇐   过程入口地址的偏移
CS     ⇐   过程入口地址的段值
```

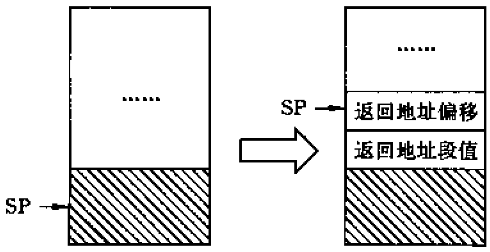
段间直接调用指令与无条件段间直接转移指令相似，机器指令中含有转移目标地址。

例如：

```
CALL      FAR  PTR  SUBRO      ;设 SUBRO 是远过程
CALL      SUBF                 ;设 SUBF 是远过程
```

汇编程序 MASM 能根据过程名所指定的被调用过程的类型决定采用段内直接调用指令还是段间直接调用指令。如果先调用后定义，那么为了调用远过程须在过程名前叫上类型说明符号“FAR　PTR”。

汇编程序 TASM 根据由过程名所指定的被调用过程与调用指令是否在同一段内决定采用段内直接调用指令还是段间直接调用指令。



(a) 段间调用前的堆栈　　(b) 段间调用后的堆栈

图 4.2　执行段间调用指令时的堆栈变化

(4) 段间间接调用

段间间接调用指令也用于调用其它代码段中的子程序。格式如下：

```
CALL  OPRD
```

OPRD 是双字存储器操作数。该指令进行的具体操作可分解如下：

```
SP     ⇐   SP-2
[SP]   ⇐   CS
SP     ⇐   SP-2
[SP]   ⇐   IP
IP     ⇐   OPRD之低字值
CS     ⇐   OPRD之高字值
```

该指令把返回地址的段值和偏移分别压入堆栈保存(堆栈变化如图 4.2 所示),然后把双字存储器操作数的低字送 IP,把双字存储器操作数的高字送 CS,从而实现远转移。

例如:

```
CALL      DWORD  PTR  [BX]
CALL      VARD                   ;设 VARD 是双字变量
```

对于间接调用,如果操作数是 16 位操作数,则汇编成段内间接调用,如果操作数是 32 位操作数,则汇编成段间间接调用。如果发生调用指令语句在先,有关变量定义伪指令语句在后的情况,则需要在调用语句中用 PTR 等操作符加以说明。

2. 过程返回指令

过程返回指令把子程序的返回地址从堆栈弹出到 IP 或 CS 和 IP,从而返回调用程序(主程序)继续往下执行。弹出的子程序返回地址一般应该就是由调用指令压入堆栈的返回地址。

过程返回指令不影响标志。

(1) 段内返回指令

段内返回指令用于近过程的返回,格式如下:

```
RET
```

该指令完成的具体操作如下所示:

```
IP     ⇐   [SP]
SP     ⇐   SP+2
```

该指令只从堆栈弹出一个字,送到指令指针 IP。堆栈变化过程如图 4.1(b)到图 4.1(a)所示。它与段内调用指令相对应,使用在近过程中。实际上,段内调用指令调用近过程时,仅把返回地址的偏移压入堆栈。

(2) 段间返回指令

段间返回指令用于远过程的返回,格式如下:

```
RET
```

该指令完成的具体操作如下所示:

```
IP     ⇐   [SP]
SP     ⇐   SP+2
CS     ⇐   [SP]
SP     ⇐   SP+2
```

该指令从堆栈弹出两个字，分别送到指令指针 IP 和代码段寄存器 CS。堆栈变化过程如图 4.2(b)到图 4.2(a)所示。它与段间调用指令相对应，使用在远过程中。实际上，段间调用指令调用远过程时，把返回地址的段值和偏移都压入堆栈。

尽管段内返回指令的助记符与段间返回指令的助记符是相同的，但它们的机器指令码是不同的。汇编程序 MASM 能根据 RET 指令所在过程的类型决定采用段内返回指令还是采用段间返回指令，只有当返回指令语句出现在远类型过程中时，MASM 才把它汇编成段间返回指令。

汇编程序 TASM 除了具有上述功能外，还提供了段间返回指令的专门助记符 RETF。例如：

```
RETF
```

无论 RETF 出现在远过程中还是近过程中，汇编程序 TASM 总把它汇编成段间返回指令。

(3) 段内带立即数返回指令

段内带立即数返回指令的格式如下：

```
RET    表达式
```

汇编程序把表达式的结果 data 取整。

该指令完成的具体操作如下所示：

```
IP    ⇐    [SP]
SP    ⇐    SP+2
SP    ⇐    SP+data
```

先从堆栈弹出一个字作为返回地址，再额外根据 data 修改堆栈指针。所以该指令不仅能实现段内返回，而且能同时再修改堆栈指针。由于对堆栈的正常操作均是以字为单位，所以，表达式的结果一般应是偶数。例如：

```
RET        4
```

设在执行上面的指令前 SP=B67EH，在执行后 SP=B684H。

(4) 段间带立即数返回指令

段间带立即数返回指令的格式如下：

```
RET        表达式
```

该指令先从堆栈弹出两个字，分别送到指令指针 IP 和代码段寄存器 CS，再额外修改堆栈指针，即把表达式的结果加到 SP。所以该指令不仅能实现段间返回，而且能同时再修改堆栈指针。例如：

```
RETF       4
```

设在执行上面的指令前 SP=B67EH，则在执行后 SP=B686H。

4.1.2 过程定义语句

利用过程定义伪指令语句，可把程序片段说明为具有近类型或远类型的过程，并且能给过程起一个名字。过程定义语句的格式如下：

```
过程名        PROC  [NEAR  |  FAR]
                .
                .
                .
过程名        ENDP
```

过程的类型在过程定义开始语句 PROC 中指定，过程可以被指定为近(NEAR)类型，也可以被指定为远(FAR)类型。如果不指定，则通常默认为近类型。如前所述，使用段内调用指令还是段间调用指令调用一个过程，由被调用过程的类型决定。因此，如果一个过程要被别的程序段调用，就应该把它说明为 FAR 类型。如果某个过程具有 NEAR 类型，那么该过程仅能被其所在段调用。

定义一个过程的开始语句 PROC 和结束语句 ENDP 前的过程名称必须一致，从而保持配对。过程名称的命名与普通标号的命名方法相同。像普通标号一样，过程名具有段值、偏移和类型这三个属性。过程名的段值和偏移是对应过程入口(过程定义开始伪指令语句后的指令语句)的段值和偏移，过程名的类型就是过程的类型。

把一位十六进制数转换为对应 ASCII 码的过程可定义如下：

```
;设欲转换的十六进制数码在 AL 的低 4 位
    ;转换得到的 ASCII 码在 AL 中
    HTOASC   PROC   NEAR
             AND    AL,0FH
             ADD    AL,30H
             CMP    AL,39H
             JBE    HTOASC1
             ADD    AL,7
    HTOASC1: RET
    HTOASC   ENDP
```

为了能返回调用程序，一般说来，在一个过程中至少要有一条返回指令，也可含多条返回指令。返回指令语句是过程的出口。但返回指令语句不一定非要安排在过程的最后。

为了说明在一个过程中使用多条 RET 指令，现将上述的"把一位十六进制数转换为对应 ASCII 码的过程"改写如下：

```
HTOASC      PROC
            AND    AL,0FH
            CMP    AL,9
            JBE    HTOASC1
            ADD    AL,37H
```

```
            RET
HTOASC1:ADD        AL,30H
            RET
HTOASC          ENDP
```

还可把上述过程作如下改变,其中 RET 语句不在过程的最后:

```
HTOASC          PROC
            AND     AL,0FH
            CMP     AL,9
            JBE     HTOASC1
            ADD     AL,37H
HTOASC2:RET
HTOASC1:ADD        AL,30H
            JMP     HTOASC2
HTOASC          ENDP
```

4.1.3 子程序举例

例 1:写一个把用 ASCII 码表示的两位十进制数转换为对应二进制数的子程序。

转换算法是:设 x 是十位数,y 是个位数,计算 10x+y。子程序源代码如下:

```
;入口参数:DH=十位数 ASCII 码,DL=个位数 ASCII 码
;出口参数:AL=对应二进制数
SUBR        PROC
            MOV     AL,DH
            AND     AL,0FH
            MOV     AH,10
            MUL     AH
            MOV     AH,DL
            AND     AH,0FH
            ADD     AL,AH
            RET
SUBR        ENDP
```

上述子程序 SUBR 没有判别作为入口参数提供的 DH 和 DL 寄存器中的内容是否是十进制数的 ASCII 码。

例 2:写一个把 16 位二进制数转换为 4 位十六进制数 ASCII 码的子程序。

转换方法是:把 16 位二进制数向左循环移位四次,使高 4 位成为低四位,析出低四位,调用子程序 HTOASC 转换出 1 位十六进制数的 ASCII 码,重复四次便完成转换。子程序源代码如下:

```
;入口参数:      DX=欲转换的二进制数
;               DS:BX=存放转换所得 ASCII 码串的缓冲区首地址
;出口参数:      十六进制数 ASCII 码串按高位到低位依次存放在指定的缓冲区中
```

```
HTASCS          PROC
                MOV     CX,4
HTASCS1:ROL             DX,1                    ;循环左移4位
                ROL     DX,1                    ;高四位成为低四位
                ROL     DX,1
                ROL     DX,1
                MOV     AL,DL                   ;复制出低四位
                CALL    HTOASC                  ;转换得ASCII码
                MOV     [BX],AL                 ;保存
                INC     BX                      ;调整缓冲区指针
                LOOP    HTASCS1                 ;重复四次
                RET
HTASCS          ENDP
```

利用上述子程序HTASCS按十六进制数形式显示地址为F000：0000H的字单元内容的一个程序如下：

```
;程序名:     T4-1.ASM
;功  能:     (略)
DSEG        SEGMENT                             ;数据段
BUFF        DB 4  DUP  (0)                      ;存放4位十六进制数的ASCII码
            DB 'H',  0DH,  0AH,  '$'            ;形成以$结尾的串
DSEG        ENDS
;
CSEG        SEGMENT                             ;代码段
            ASSUME   CS:CSEG,  DS:DSEG
START:      MOV     AX,DSEG
            MOV     DS,AX
            MOV     AX,0F000H
            MOV     ES,AX
            MOV     DX,ES:[0]                   ;取指定内存单元的内容
            MOV     BX,OFFSET  BUFF             ;准备入口参数
            CALL    HTASCS                      ;转换为4位十六进制数的ASCII码
            ;
            MOV     DX,OFFSET  BUFF
            MOV     AH,9                        ;调用9H号功能显示之
            INT     21H
            ;
            MOV     AX,4C00H                    ;程序正常结束
            INT     21H
            ;
HTASCS          PROC
;内容略
```

```
HTASCS         ENDP
;
HTOASC         PROC
;内容略
HTOASC         ENDP
CSEG      ENDS
          END     START
```

例3：写一个把16位二进制数转换为5位十进制数ASCII码的子程序。为了简单，设二进制数是无符号的。

第三章中的程序T3-11.ASM实现"把16位二进制数转换为5位十进制的BCD码"，把BCD码转换为对应ASCII码是容易的，所以可把T3-11.ASM改写成一个子程序。但我们采用另一个算法把16位二进制数转换为5位十进制数的BCD码。该算法是：把16位二进制数除以10，余数是"个位"数的BCD码；把商再除以10，余数就是"十位"数的BCD码；如此，可依次得"百位"、"千位"和"万位"数的BCD码。这可利用一个循环来控制。

```
;入口参数：    AX=欲转换的二进制数
;              DS：BX=存放转换所得ASCII码串的缓冲区首地址
;出口参数：    十进制数ASCII码串按万位到个位的序依次存放在指定的缓冲区中
BTOASC         PROC
          MOV     SI,5              ;置循环次数
          MOV     CX,10             ;置除数10
BTOASC1:XOR       DX,DX             ;把被除数扩展成32位
          DIV     CX                ;除操作
          ADD     DL,30H            ;余数为BCD码，转换为ASCII码
          DEC     SI                ;调整循环计数器
          MOV     [BX][SI],DL       ;保存所得ASCII码
          OR      SI,SI             ;判是否转换结束
          JNZ     BTOASC1           ;否，继续
          RET
BTOASC         ENDP
```

在上面的子程序中，寄存器SI既作计数器使用又作变址指针使用。

4.1.4 子程序说明信息

为了能正确地使用子程序，在给出子程序代码时还要给出子程序的说明信息。子程序说明信息一般由如下几部分组成，每一部分内容应简明确切：

(1) 子程序名。

(2) 功能描述。

(3) 入口和出口参数。

(4) 所用的寄存器和存储单元。

(5) 使用的算法和重要的性能指标。

(6) 其他调用注意事项和说明信息。

(7) 调用实例。

子程序说明信息至少应该包含前三部分内容。例如：

```
;子程序名：   AHTOASC
;功    能：   把 8 位二进制数转换为 2 位十六进制数的 ASCII 码
;入口参数：   AL＝欲转换的 8 位二进制数
;出口参数：   AH＝十六进制数高位的 ASCII 码
;             AL＝十六进制数低位的 ASCII 码
;其他说明：   (1)近过程
;             (2)除 AX 寄存器外，不影响其他寄存器
;             (3)调用了 HTOASC 实现十六进制数到 ASCII 码的转换
```

在看了上述关于子程序 AHTOASC 的说明信息后，即使不熟悉或不了解这个子程序本身，也就能够方便地调用它了。

子程序 AHTOASC 的源代码如下：

```
AHTOASC    PROC
           MOV    AH,AL
           SHR    AL,1
           SHR    AL,1
           SHR    AL,1
           SHR    AL,1
           CALL   HTOASC
           XCHG   AH,AL
           CALL   HTOASC
           RET
AHTOASC    ENDP
```

4.1.5 寄存器的保护与恢复

子程序为了完成其功能，通常要使用一些寄存器存放内容，有时还要使用一些存储单元存放内容。也就是说，在子程序运行时通常会破坏一些寄存器或存储单元的原有内容。所以，如果不采取措施，那么在调用子程序后，主程序就无法再使用存放在这些寄存器或存储单元中的原有内容了，这常常会导致主程序的错误。为此，要对有关寄存器或存储单元的内容进行保护与恢复。

寄存器的保护与恢复有两种方法：

(1) 把需要保护的寄存器的内容，在主程序中压入堆栈和弹出堆栈。这种方法的优点是，在每次调用子程序时，只要把主程序所关心的寄存器压入堆栈。但缺点是：在主程序中使用压入和弹出堆栈的操作会使主程序不易理解；如果要多次调用子程序时，会很累赘，而且常常会忘了把某个寄存器压入堆栈。

(2) 在子程序一开始就把在子程序中要改变的寄存器内容压入堆栈，而在返回之前再恢复这些寄存器的内容。这种方法的优点是：在主程序中可方便地调用子程序，而无需考虑要把哪些寄存器压入堆栈；只需要在子程序中写一次压入和弹出堆栈指令组即可。这种方法是常用的方法。

下面是重写的 BTOASC 子程序，它保护了所有被改变的寄存器内容，主程序在调用它时，不必担心各寄存器的内容受影响。

```
        ;子程序说明信息略
        BTOASC      PROC
                    PUSH    AX
                    PUSH    CX
                    PUSH    DX
                    PUSH    SI
                    MOV     SI,5                ;置循环次数
                    MOV     CX,10               ;置除数 10
        BTOASC1:XOR         DX,DX               ;把被除数扩展成 32 位
                    DIV     CX                  ;除操作
                    ADD     DL,30H              ;余数为 BCD 码，转换为 ASCII 码
                    DEC     SI                  ;调整循环计数器
                    MOV     [BX][SI],DL         ;保存所得 ASCII 码
                    OR      SI,SI               ;判是否转换结束
                    JNZ     BTOASC1             ;否，继续
                    POP     SI
                    POP     DX
                    POP     CX
                    POP     AX
                    RET
        BTOASC      ENDP
```

还有几点说明，请注意：

(1) 上述子程序 BTOASC 实际上还破坏了标志寄存器中的部分标志，可用 PUSHF 指令和 POPF 指令保护和恢复标志寄存器。但一般不在子程序中保护和恢复标志寄存器。

(2) 在利用堆栈进行寄存器的保护和恢复时，要注意堆栈的先进后出特性。

(3) 有时为了简单，并不保护含有入口参数的寄存器。是否要保护入口参数寄存器，可以根据实际情况事先约定。在 BTOASC 中，含有入口参数的 AX 寄存器得到了保护。

可像寄存器的保护和恢复那样，保护和恢复有关存储单元的内容，即在子程序开始时把有关存储单元的内容压入堆栈，在子程序返回前恢复它们。在子程序中应尽量避免把普通存储单元作为临时变量使用，可以利用堆栈元素作为临时变量使用。

4.2　主程序与子程序间的参数传递

主程序在调用子程序时，往往要向子程序传递一些参数；同样，子程序运行后也经常要把一些结果参数传回给主程序。主程序与子程序之间的这种信息传递称为参数传递。我们把由主程序传给子程序的参数称为子程序的入口参数，把由子程序传给主程序的参数称为子程序的出口参数。一般子程序既有入口参数，又有出口参数。但有的子程序只有入口参数，而没有出口参数；有的子程序只有出口参数，而没有入口参数。

有多种参数传递的方法：寄存器传递法、约定内存单元传递法、堆栈传递法和 CALL 后续区传递法等。主程序与子程序间传递参数的方法是根据具体情况而事先约定好的。有时可能同时采用多种方法。

4.2.1　利用寄存器传递参数

利用寄存器传递参数就是把参数放在约定的寄存器中。这种方法的优点是实现简单和调用方便。但由于寄存器的个数是有限的，且寄存器往往还要存放其他数据，所以只适用于要传递的参数较少的情况。

在 4.1 节中的子程序 HTOASC 和子程序 AHTOASC 等就是利用寄存器来传递参数的，现再举几例。

例 1：写一个把大写字母改为小写字母的子程序。

在 ASCII 码表中，大写字母的 ASCII 码比对应小写字母的 ASCII 码小 20H(即 'a'-'A')，所以在确定字符是大写字母后，把它转换为对应的小写字母是容易的。

```
;子程序名:  UPTOLW
;功    能:  把大写字母转换为小写字母
;入口参数:  AL=字符 ASCII 码
;出口参数:  AL=字符 ASCII 码
;说    明:  如果字符是大写字母，则转换为小写字母，其他字符不变
UPTOLW      PROC
      PUSHF                                ;保护各标志
            CMP     AL,'A'
            JB      UPTOLW1
            CMP     AL,'Z'
            JA      UPTOLW1
            ADD     AL,'a'-'A'
UPTOLW1:POPF                               ;恢复各标志
            RET
UPTOLW      ENDP
```

上面的子程序还保护了标志寄存器，从而使得在执行该子程序后，各标志不受影响。是否要保护标志可视具体情况而事先约定。有时还利用某些标志作为出入口参数。进位标志 CF 常常用作为出口参数。

例 2：写一个判别字符是否为数字符的子程序。并利用该子程序把一个字符串中的所有数字符删除。

子程序 ISDECM 及其说明如下：

```
;子程序名： ISDECM
;功    能：  判别一个字符是否为数字符
;入口参数：  AL=字符
;出口参数：  CF 为 0 表示字符是数字符，否则字符是非数字符
ISDECM      PROC
          CMP      AL,'0'
          JB       ISDECM1
          CMP      AL,'9'
          JA       ISDECM1
          CLC
          RET
ISDECM1:STC
          RET
ISDECM      ENDP
```

对上述子程序可作些优化工作，改写过的 ISDECM 如下所示：

```
ISDECM      PROC
          CMP      AL,'0'
          JB       ISDECM1
          CMP      AL,'9'+1
          CMC                                    ;把 CF 标志取反
ISDECM1:RET
ISDECM      ENDP
```

改写过的子程序 ISDECM 利用了 CMP 指令对标志的影响。使用该子程序把字符串中的数字符删除的源程序如下，子程序位于主程序之前：

```
;程序名：    T4-2.ASM
;功  能：    (略)
DSEG        SEGMENT
STRING        DB    'AB=C950=asd',0             ;假设的字符串
DSEG        ENDS
;
CSEG        SEGMENT
            ASSUME CS : CSEG,  DS : DSEG
ISDECM        PROC
            ;(略)
ISDECM    ENDP
;
```

```
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     SI,OFFSET  STRING    ;置取指针
        MOV     DI,SI                ;置存指针
NEXT:   MOV     AL,[SI]              ;取一个字符
        INC     SI
        OR      AL,AL                ;是否到字符串尾?
        JZ      OK                   ;是,转
        CALL    ISDECM               ;否,判是否为数字符
        JNC     NEXT                 ;是,不保存而处理下一个字符
        MOV     [DI],AL              ;否,保存
        INC     DI
        JMP     NEXT                 ;处理下一个字符
OK:     MOV     [DI],AL
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

4.2.2 利用约定存储单元传递参数

在传递参数较多的情况下,可利用约定的内存变量来传递参数。这种方法的优点是子程序要处理的数据或送出的结果都有独立的存储单元,编写程序时不易出错。但这种方法要占用一定的存储单元,此外还有其他缺点,我们在下文中说明。

例 3:写一个实现 32 位数相加的子程序。

```
;子程序名:   MADD
;功    能:   32 位数相加
;入口参数:   DATA1 和 DATA2 缓冲区中分别存放要相加的 32 数
;出口参数:   DATA3 缓冲区中存放结果
;说    明: (1)32 位数据的存放次序采用"高高低低"的原则
;          (2)可能产生的进位存放在 DATA3 开始的第 5 字节中
MADD    PROC
        PUSH    AX
        PUSH    CX                          ;保护寄存器
        PUSH    SI
        MOV     CX,2
        XOR     SI,SI
MADD1:  MOV     AX,WORD  PTR  DATA1[SI]
        ADC     AX,WORD  PTR  DATA2[SI]    ;16 位相加
        MOV     WORD  PTR  DATA3[SI],AX
        INC     SI
```

```
            INC     SI
            LOOP    MADD1
            MOV     AL,0                        ;处理进位位
            ADC     AL,0
            MOV     BYTE  PTR  DATA3+4,AL
            POP     SI
            POP     CX                          ;恢复寄存器
            POP     AX
            RET
MADD        ENDP
```

利用约定的存储单元传递参数，通用性较差。上述子程序 MADD 的通用性就较差。为了传递较多的参数，又要保持良好的通用性，通常把参数组织成一张参数表，存放在某个存储区，然后把这个存储区的首地址传送给子程序。既可利用寄存器传递首地址，也可利用堆栈方法传递首地址。

例 4：设计一个把以 ASCII 码表示的十进制数字串转换为二进制数的子程序。设表示的十进制数不大于 65535。

假设要转换的十进制数字串存放在由 DS：BX 所指的缓冲区中，该缓冲区的第一字节含有十进制数字串的长度。缓冲区结构如图 4.3 所示。

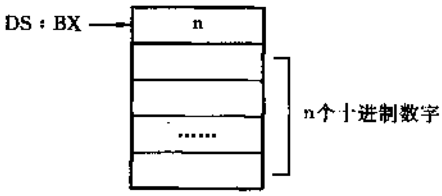


图 4.3　例 4 缓冲区结构示意图

设十进制数字串中各位对应的 BCD 码是 d_n、d_{n-1}、…、d_2、d_1，那么它所表示的二进制数可由下式计算出：

$$Y=((((0*10+d_n)*10+d_{n-1})*10+\cdots)*10+d_2)*10+d_1$$

可通过迭代的方法进行上式的计算，迭代公式如下，Y 的初值为 0：

$$Y=Y*10+d_i \qquad (i=n,n-1,\cdots 1)$$

所以，当十进制数字串中数字的个数为 n 时，那么只需进行 n 次迭代计算。

由于假设数字串表示的十进制数不超过 65535，所以用 16 位的 AX 寄存器存放迭代式中的 Y，也用 AX 寄存器返回转换后得到的二进制数。子程序源代码如下所示：

```
;子程序名：  DTOBIN
;功    能：  把用 ASCII 码表示的十进制数字串转换为二进制数
;入口参数：  DS：BX=缓冲区首地址(缓冲区结构如图 4.3)
;出口参数：  AX=转换得到的二进制数
DTOBIN          PROC
            PUSH    BX
            PUSH    CX
            PUSH    DX
```

```
            XOR     AX,AX               ;设置初值0
            MOV     CL,[BX]
            INC     BX
            XOR     CH,CH               ;CX=n
            JCXZ    DTOBIN2
DTOBIN1:MOV     DX,10
            MUL     DX                  ;Y * 10
            MOV     DL,[BX]             ;取下一个数字符
            INC     BX
            AND     DL,0FH              ;转成BCD码
            XOR     DH,DH
            ADD     AX,DX               ;Y * 10+di
            LOOP    DTOBIN1
DTOBIN2:POP     DX
            POP     CX
            POP     BX
            RET
DTOBIN      ENDP
```

这个子程序有两点不足：(1) 没有检查数字串中是否有非十进制数字符存在，(2) 不适用于数字串表示的十进制数超过值65535的情况。

4.2.3 利用堆栈传递参数

如果使用堆栈传递入口参数，那么主程序在调用子程序之前，把需要传递的参数依次压入堆栈，子程序从堆栈中取入口参数；如果使用堆栈传递出口参数，那么子程序在返回前，把需要返回的参数存入堆栈，主程序在堆栈中取出口参数。

利用堆栈传递参数可以不占用寄存器，也无需使用额外的存储单元。但由于参数和子程序的返回地址混杂在一起，有时还要考虑保护寄存器，所以较为复杂。通常利用堆栈传递入口参数，而利用寄存器传递出口参数。

例5：写一个测量字符串长度的子程序。设字符串以0为结束标志。

利用堆栈来传递入口参数：字符串的起始地址(设包括段值和偏移)；利用寄存器传递出口参数：字符串的长度。设子程序STRLEN是一个近过程，那么主程序在调用它时，只把返回地址的偏移压入堆栈。

```
        ;子程序名：  STRLEN
        ;功    能：  测量字符串的长度
        ;入口参数：  字符串起始地址的段值和偏移在堆栈中，见图4.4(a)
        ;出口参数：  AX=字符串长度
STRLEN      PROC
            PUSH    BP
            MOV     BP,SP               ;为取参数作准备
            PUSH    DS                  ;保护寄存器
```

```
              PUSH    SI                     ;执行此指令后的堆栈如图 4.4(c)
              MOV     DS,[BP+6]              ;取入口参数
              MOV     SI,[BP+4]
              MOV     AL,0                   ;置要找的结束标志值
STRLEN1:      CMP     AL,[SI]                ;判是否到字符串尾
              JZ      STRLEN2                ;到尾转
              INC     SI
              JMP     STRLEN1                ;继续
STRLEN2:      MOV     AX,SI
              SUB     AX,[BP+4]              ;计算字符串长度
              POP     SI
              POP     DS                     ;恢复寄存器
              POP     BP
              RET
STRLEN        ENDP
```

子程序借助指针寄存器BP,采用寄存器相对寻址方式取得入口参数,位移量与返回地址占用的字节数、为保护寄存器BP而使用的字节数、入口参数压入堆栈的次序有关。另外,请注意当BP寄存器作为指针使用时,隐含使用的段寄存器是SS,这正是堆栈段。

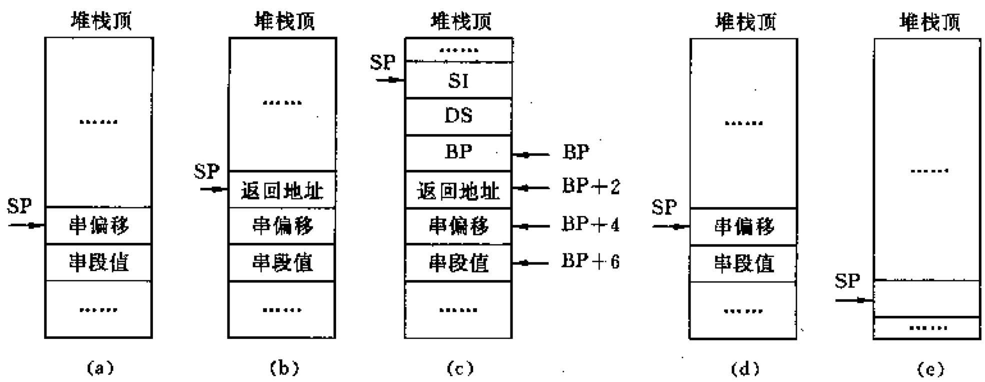


图 4.4 STRLEN 子程序利用堆栈传递入口参数时的堆栈变化

主程序调用这个子程序的代码片段如下所示,它说明主程序与子程序一定要事先约定,密切配合:

```
;……
MOV     AX,SEG  STRMESS           ;把入口参数压入堆栈
PUSH    AX
MOV     AX,OFFSET  STRMESS
PUSH    AX
CALL    STRLEN                    ;调用子程序
;                                 ;执行调用指令后堆栈如图 4.4(b)
```

```
ADD     SP,4            ;从子程序返回后堆栈如图 4.4(d)
;                       ;执行此指令后堆栈如图 4.4(e)
MOV     LEN,AX          ;保存字符串长度值
;……
```

主程序在调用子程序 STRLEN 之前把要测量长度的字符串起始地址的段值和偏移作为入口参数压入堆栈，然后就马上调用子程序 STRLEN，这样堆栈就如图 4.4(b)所示。从子程序返回后堆栈如图 4.4(d)所示，所以需要废除仍在堆栈中的入口参数，本例中利用了指令“ADD SP,4”来平衡堆栈，使堆栈恢复到如图 4.5(e)所示。

另一种废除堆栈中入口参数的方法是使用带立即数返回指令。这可在子程序返回时就自动平衡堆栈，主程序就不需要再进行堆栈的平衡工作。对于上述子程序，可采用如下的返回指令：

```
RET     4               ;此立即数与入口参数量有关
```

如果这样的话，上面主程序调用子程序的片段就不需要调整堆栈指针的指令“ADDSP,4”。

4.2.4 利用 CALL 后续区传递参数

CALL 后续区是指位于 CALL 指令后的存储区域。主程序在调用子程序之前，把入口参数存入 CALL 指令后的存储单元中，子程序根据保存在堆栈中的返回地址找到入口参数，这种传递参数的方法称为 CALL 后续区传递参数法。利用 CALL 后续区传递参数的子程序必须修改返回地址。由于这种方法把数据和代码混在一起，所以在 x86 系列汇编语言程序中使用得不多。

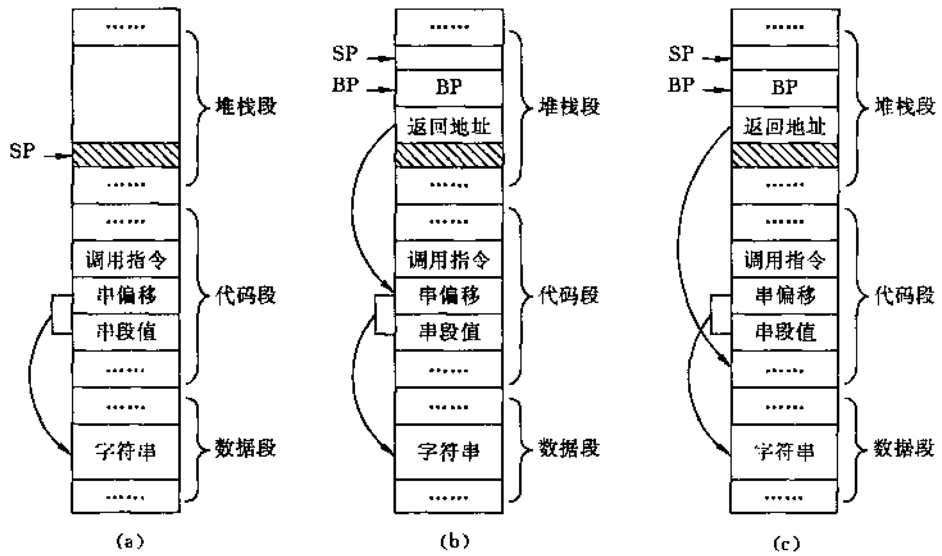


图 4.5 例 6 有关代码、数据和堆栈段的示意图

例6：写一个把字符串中的大写字母改为小写字母的子程序(近过程)。设字符串以 0 为结束标志。用 CALL 后续区传递字符串起始地址的段值和偏移。

```
;子程序名:    STRLWR
;功    能:    把字符串中的所有大写字母改为小写字母
;入口参数:    字符串起始地址的段值和偏移在 CALL 后续区(见图 4.5(a))。
;出口参数:    无
STRLWR      PROC
            PUSH    BP
            MOV     BP,SP
            PUSH    AX
            PUSH    SI                      ;保护寄存器
            PUSH    DS                      ;
            MOV     SI,[BP+2]               ;从堆栈中取得返回地址,见图 4.5(b)
            MOV     DS,CS:[SI+2]            ;取入口参数(字符串段值)
            MOV     SI,CS:[SI]              ;取入口参数(字符串偏移)
STRLWR1:    MOV     AL,[SI]
            CMP     AL,0
            JZ      STRLWR3
            CMP     AL,'A'
            JB      STRLWR2                 ;把字符串中的大写字母改为小写
            CMP     AL,'Z'
            JA      STRLWR2
            ADD     AL,'a'-'A'              ;ADD  AL,20H
            MOV     [SI],AL
STRLWR2:    INC     SI
            JMP     STRLWR1
STRLWR3:    ADD     WORD  PTR  [BP+2],4     ;修改返回地址,见图 4.5(c)
            POP     DS                      ;恢复寄存器
            POP     SI
            POP     AX
            POP     BP
            RET
STRLWR      ENDP
```

调用上述子程序的源程序片段如下所示：

```
            ;……
            CALL    STRLWR
            DW      OFFSET   STRMESS
            DW  SEG   DSEG
CONT:       ……
```

图 4.5 是有关代码段、数据段和堆栈段的示意图，其中(a)是调用之前的情形，(b)是

调用和保护有关寄存器后的情形，(c)是修改返回地址后的情形。

4.3 DOS 功能调用及应用

尽管 DOS 正在逐步消亡，但目前在 DOS 平台上进行 x86 汇编语言程序设计的实习还是最有效的途径。了解 DOS 功能调用，不仅有益于程序设计锻炼，也便于在 DOS 平台上进行汇编语言程序设计的实习。

4.3.1 DOS 功能调用概述

1. 什么是 DOS 功能调用

MS-DOS(PC-DOS)内包含了许多涉及设备驱动和文件管理等方面的子程序，DOS 的各种命令就是通过适当地调用这些子程序实现的。为了方便程序员使用，把这些子程序编写成相对独立的程序模块而且编上号。程序员利用汇编语言可方便地调用这些子程序。这些子程序被精心编写，而且经过了大量的各种应用范围的实践考验。程序员调用这些子程序可减少对系统硬件环境的考虑和依赖，从而一方面可大大精简应用程序的编写，另一方面可使程序有良好的通用性。这些编了号的可由程序员调用的子程序就称为 DOS 的功能调用或称为系统调用。一般认为 DOS 的各种命令是操作员与 DOS 的接口，而功能调用则是程序员与 DOS 的接口。

DOS 功能调用主要包括三方面的子程序：设备驱动(基本 I/O)、文件管理和其他(包括内存管理、置取时间、置取中断向量、终止程序等)。随着 DOS 版本的升级，这种称为 DOS 功能调用的子程序数量也不断增加，功能更加完备，使用也更加方便。

2. 调用方法

可按如下方法调用 DOS 功能调用：

(1) 根据需调用的功能调用准备入口参数。有部分功能调用是不需要入口参数的，但大部分功能调用需要入口参数，在调用前应按要求准备好入口参数。

(2) 把功能调用号送 AH 寄存器。

(3) 发软中断指令“INT　21H”。

程序员不必关心有关子程序在何处，也不必关心它是如何实现其功能的。

例如：调用 2 号功能调用，使喇叭发出“嘟”的一声。2 号功能调用的功能是在屏幕上显示一个字符，入口参数是 DL 寄存器为要显示字符的 ASCII 码。当要显示字符的 ASCII 码为 07H 时，并不在屏幕上显示字符，而是使喇叭发出“嘟”的一声。程序片段如下：

```
MOV     DL,07H          ;准备入口参数
MOV     AH,2            ;置功能调用号
INT     21H             ;实施调用
```

大部分功能调用都有出口参数，在调用后，可根据有关功能调用的说明取得出口参数。部分功能调用没有出口参数，如 2 号功能调用，调用它后，只是在屏幕上显示相应的字符，或发出“嘟”的一声。

还有个别功能调用很特殊，调用它后就不再返回。例如 4CH 号功能调用，其功能就是

结束程序的运行而返回DOS。我们已在多个程序中使用了这个功能调用。4CH号功能调用有一个存放在AL寄存器中的入口参数，该入口参数是程序的结束码，其值的大小不影响程序的结束。例如：

```
MOV     AL,0        ;置退出码
MOV     AH,4CH      ;置功能调用号
INT     21H         ;实施调用
```

4.3.2 基本I/O功能调用

1. 带回显键盘输入(1号功能调用)

功　能：从标准输入设备上读一字符，并将该字符回显在标准输出设备上。通常情况下，标准输入设备就是键盘，标准输出设备就是屏幕。如果键盘无字符可读，则一直等待到有字符可读(即按键)。

入口参数：无。

出口参数：AL＝读到字符的代码(ASCII码)。

说　明：如果读到的字符是Ctrl+C或Ctrl+Break，则结束程序。

2. 不带回显键盘输入(8号功能调用)

除读到的输入字符不在屏幕上显示外，同1号功能调用。

3. 直接键盘输入(7号功能调用)

功　能：从标准输入上读一字符。通常情况下，标准输入就是键盘。如果键盘无字符可读，则一直等待到有字符可读(即按键)。

入口参数：无。

出口参数：AL＝读到字符的代码。

说　明：(1) 不检查读到的字符是否是Ctrl+C或Ctrl+Break。

(2) 不回显读到的字符。

4. 显示输出(2号功能调用)

功　能：向标准输出设备写一字符。通常情况下，标准输出设备就是屏幕。

入口参数：DL＝要输出的字符(ASCII码)。

出口参数：无。

说　明：在显示输出时检查是否按Ctrl+C或Ctrl+Break键，如是则结束程序。

5. 直接控制台输入输出(6号功能调用)

功　能：直接控制台输入输出。通常情况下，控制台输入就是键盘输入，控制台输出就是屏幕输出。

入口参数：若DL＝0FFH，表示输入；否则表示输出，DL＝输出字符代码。

出口参数：输入时，ZF＝1表示无字符可读；ZF＝0表示读到字符，AL＝输入字符代码；输出时，无。

说　明：(1) 在输入时，如无字符可读，并不等待。

(2) 不检查Ctrl+C或Ctrl+Break键。

(3) 在读到字符时也不回显。

(4) 在输入时，如果 AL=0，表示用户曾按过一个扩展键，在下一次调用该功能时返回扩展键的扫描码。

(5) 在输出时，不解释制表符等特殊控制符。

6. 显示字符串(9 号功能调用)

功　　能：在标准输出上显示一个字符串。通常情况下，标准输出就是屏幕。

入口参数：DS：DX=需要输出字符串的首地址，字符串以字符'$'为结束标志。

出口参数：无。

说　　明：在显示输出时检查是否按 Ctrl+C 或 Ctrl+Break 键，如是则结束程序。

7. 输入字符串(0AH 号功能调用)

功　　能：从标准输入上读一个字符串。通常情况下，标准输入就是键盘。

入口参数：DS：DX=缓冲区首地址。

出口参数：接收到的输入字符串在缓冲区中。

说　　明：(1) 缓冲区第一字节置为缓冲区最大容量，可认为是入口参数；缓冲区第二字节存放实际读入的字符数(不包括回车符)，可认为是出口参数的一部分；第三字节开始存放接受的字符串。

(2) 字符串以回车键结束，回车符是接受到的字符串的最后一个字符。

(3) 如果输入的字符数超过缓冲区所能容纳的最大字符数，则随后的输入字符被丢弃并且响铃，直到遇回车键为止。

(4) 如果在输入时按 Ctrl+C 或 Ctrl+Break 键，则结束程序。

8. 取键盘输入状态(0BH 号功能调用)

功　　能：判别在标准输入设备上是否有字符可读。

入口参数：无。

出口参数：AL=0，表示无字符可读；

AL=0FFH，表示有字符可读。

说　　明：检查是否按 Ctrl+C 或 Ctrl+Break 键，如遇这种键，则程序结束。

9. 清除输入缓冲区后再输入(0CH 号功能调用)

功　　能：清除输入缓冲区，然后再执行某个输入功能。

入口参数：AL=清除输入缓冲区后要执行的功能号。

出口参数：决定于清除输入缓冲区后执行的功能。

说　　明：清除缓冲区后执行的功能应是 01H、06H、07H、08H 或 0AH，如不是这样，则不输入。例如：若 AL=0，则在清除输入缓冲区后，没有进一步的处理。

10. 打印输出(5 号功能调用)

功　　能：向连接在第一个并行口上的打印机输出一字符。

入口参数：DL=要打印的字符(ASCII 码)。

出口参数：无。

说　　明：打印机可能不立即打印出指定的字符。

4.3.3 应用举例

例 1：写一个程序，它用二进制数形式显示所按键的 ASCII 码。

首先利用 1 号功能调用接受一个字符，然后通过移位的方法从高到低依次把其 ASCII 码值的各位析出，再转换成 ASCII 码，利用 2 号功能调用显示输出。源程序如下所示：它还含有一个形成回车换行（光标移到下一行首）的子程序。

```
;程序名:   T4-3.ASM
;功  能:   (略)
CSEG      SEGMENT
          ASSUME   CS:CSEG,   DS:CSEG
START:    MOV     AH,1               ;读一个键
          INT     21H
          CALL    NEWLINE            ;回车换行
          MOV     BL,AL
          MOV     CX,8               ;8 位
NEXT:     SHL     BL,1               ;依次析出高位
          MOV     DL,30H
          ADC     DL,0               ;转换得 ASCII 码
          MOV     AH,2
          INT     21H                ;显示之
          LOOP    NEXT               ;循环 8 次
          MOV     DL,'B'
          MOV     AH,2               ;显示二进制数表示符
          INT     21H
          MOV     AH,4CH             ;正常结束
          INT     21H
;
;子程序名:NEWLINE
;功    能:形成回车和换行(光标移到下一行首)
;入口参数:无
;出口参数:无
;说    明:通过显示回车符形成回车,通过显示换行符形成换行
NEWLINE  PROC
          PUSH   AX
          PUSH   DX
          MOV    DL,0DH              ;回车符的 ASCII 码是 0DH
          MOV    AH,2                ;显示回车符
          INT    21H
          MOV    DL,0AH              ;换行符的 ASCII 码是 0AH
          MOV    AH,2                ;显示换行符
          INT    21H
```

```
        POP     DX
        POP     AX
        RET
NEWLINE ENDP
CSEG    ENDS
        END     START
```

例 2：写一个程序，它先接受一个字符串，然后显示其中数字符的个数、英文字母的个数和字符串的长度。

先利用 0AH 号功能调用接受一个字符串，然后分别统计其中数字符的个数和英文字母的个数，最后用十进制数的形式显示它们。整个字符串的长度可从 0AH 号功能调用的出口参数中取得。源程序如下所示：

```
;程序名：  T4-4.ASM
;功  能：  (略)
MLENGTH = 128                                ;缓冲区长度
;
DSEG     SEGMENT                             ;数据段
BUFF     DB  MLENGTH                         ;符合 0AH 号功能调用所需的缓冲区
         DB ?                                ;实际键入的字符数
         DB MLENGTH  DUP  (0)
MESS0    DB 'Please  input：$'
MESS1    DB 'Length  =   $'
MESS2    DB 'X  =  $'
MESS3    DB 'Y  =  $'
DSEG     ENDS
;
CSEG     SEGMENT                             ;代码段
         ASSUME  CS：CSEG,  DS：DSEG
START：  MOV     AX,DSEG
         MOV     DS,AX                       ;置 DS
         MOV     DX,OFFSET  MESS0            ;显示提示信息
         CALL    DISPMESS
         MOV     DX,OFFSET  BUFF
         MOV     AH,10                       ;接受一个字符串
         INT     21H
         CALL    NEWLINE
         MOV     BH,0                        ;清数字符计数器
         MOV     BL,0                        ;清字母符计数器
         MOV     CL,BUFF+1                   ;取字符串长度
         MOV     CH,0
         JCXZ    COK                         ;若字符串长度等于 0,不统计
```

```
        MOV     SI,OFFSET   BUFF+2  ;指向字符串首
AGAIN:  MOV     AL,[SI]             ;取一个字符
        INC     SI
        CMP     AL,'0'              ;判是否是数字符
        JB      NEXT
        CMP     AL,'9'
        JA      NODEC
        INC     BH                  ;数字符计数加1
        JMP     SHORT   NEXT
NODEC:  OR      AL,20H              ;转小写
        CMP     AL,'a'              ;判是否是字母符
        JB      NEXT
        CMP     AL,'z'
        JA      NEXT
        INC     BL                  ;字母符计数加1
NEXT:   LOOP    AGAIN               ;下一个
;
COK:    MOV     DX,OFFSET   MESS1
        CALL    DISPMESS
        MOV     AL,BUFF+1           ;取字符串长度
        XOR     AH,AH
        CALL    DISPAL              ;显示字符串长度
        CALL    NEWLINE
        ;
        MOV     DX,OFFSET   MESS2
        CALL    DISPMESS
        MOV     AL,BH
        XOR     AH,AH
        CALL    DISPAL              ;显示数字符个数
        CALL    NEWLINE
        ;
        MOV     DX,OFFSET   MESS3
        CALL    DISPMESS
        MOV     AL,BL
        XOR     AH,AH
        CALL    DISPAL              ;显示字母符个数
        CALL    NEWLINE
        ;
        MOV     AX,4C00H            ;程序正常结束
        INT     21H
;
;子程序名:DISPAL
```

```
;功    能:用十进制数的形式显示8位二进制数
;入口参数:AL=8位二进制数
;出口参数:无
DISPAL      PROC
            MOV     CX,3              ;8位二进制数最多转换成3位十进制数
            MOV     DL,10
DISP1:      DIV     DL
            XCHG    AH,AL             ;使AL=余数,AH=商
            ADD     AL,'0'            ;得ASCII码
            PUSH    AX                ;压入堆栈
            XCHG    AH,AL
            MOV     AH,0
            LOOP    DISP1             ;继续
            MOV     CX,3
DISP2:      POP     DX                ;弹出一位
            CALL    ECHOCH            ;显示之
            LOOP    DISP2             ;继续
            RET
DISPAL      ENDP
;
;显示由DX所指的提示信息,其他子程序说明信息略
DISPMESS    PROC
            MOV     AH,9
            INT     21H
            RET
DISPMESS    ENDP
;
;显示DL中的字符,其他子程序说明信息略
ECHOCH      PROC
            MOV     AH,2
            INT     21H
            RET
ECHOCH      ENDP
;
;略去子程序NEWLINE,该子程序列于源程序T4-3.ASM中
CSEG        ENDS
            END     START
```

例3:写一个显示指定内存单元内容的程序。具体要求是:允许用户按十六进制数的形式输入指定内存单元的段值和偏移,然后用十六进制数形式显示指定字节单元的内容。

该程序可分成如下几步:(1)接收段值和偏移;(2)把指定字节单元中的内容转换成2位十六进制数的ASCII码,边转换边显示。

为了接收段值和偏移，设计子程序 GETADR。子程序 GETADR 接收用户输入的十六进制数串，并转换为二进制数。根据功能划分，又派生出子程序 GETSTR 和子程序 HTOBIN。子程序 GETSTR 接收一个最大长度为 4 的十六进制数串；子程序 HTOBIN 负责把这个数串转换为二进制数。所以，子程序 GETADR 调用 GETSTR 和 HTOBIN 实现其功能。在两次调用 GETADR 后，可分别得段值和偏移。子程序 GETADR 没有入口参数，只有在寄存器中提供的出口参数。

子程序 GETSTR 较为复杂。它并没有调用 0AH 号功能接收字符串，而是循环调用 8 号功能接收单个字符。这样处理后，如果用户按错键，就可使喇叭发出"嘟"的一声，作为提示。为此，它要检查用户按的键是否是十六进制数码键，要处理退格键，还要控制接收的字符数不超过规定等。子程序 GETSTR 的实现流程如图 4.6 所示。它通过约定的缓冲区传递参数。

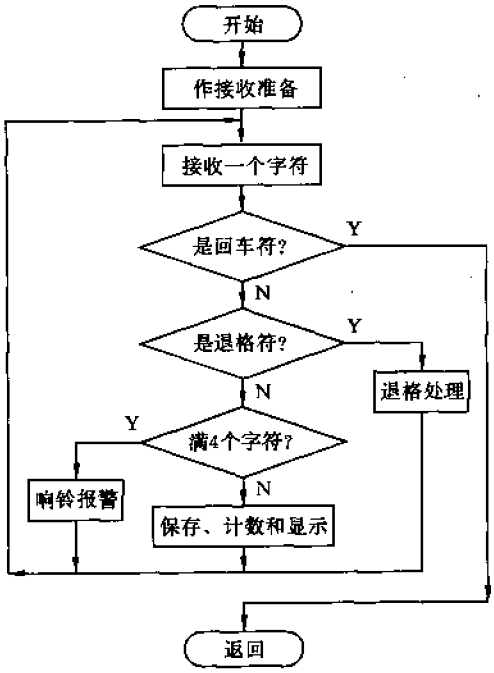


图 4.6 子程序 GETSTR 的流程图

子程序 HTOBIN 实现把 4 位十六进制数的 ASCII 码转换为一个二进制数。它采用"X=16 * X+Yi"的迭代公式，进行四次迭代计算。其中 X 的初值为 0，Yi 是根据十六进制数符的 ASCII 码转换得出的值。

```
;程序名:T4-5.ASM
;功  能:用十六进制数的形式显示指定内存字节单元的内容
;符号常量定义
CR = 0DH                                    ;回车符
LF = 0AH                                    ;换行符
```

```
BACKSPACE  =  08H                          ;退格符
BELLCH  =  07H                             ;响铃符
;数据段
DSEG      SEGMENT
SEGOFF        DD      ?                    ;存放指定单元的段值和偏移
MESS1         DB      'SEGMENT:$'
MESS2         DB      'OFFSET:$'
BUFFER        DB      5  DUP  (0)          ;缓冲区
DSEG      ENDS
;代码段
CSEG      SEGMENT
          ASSUME  CS:CSEG,  DS:DSEG
START:    MOV     AX,DSEG
          MOV     DS,AX                    ;设置数据段寄存器
          ;
          MOV     DX,OFFSET  MESS1
          CALL    DISPMESS                 ;显示提示信息
          CALL    GETADR                   ;接收段值
          MOV     WORD  PTR  SEGOFF+2,AX        ;保存
          MOV     DX,OFFSET  MESS2
          CALL    DISPMESS                 ;显示提示信息
          CALL    GETADR                   ;接收偏移
          MOV     WORD  PTR  SEGOFF,AX     ;保存
          ;
          LES     DI,SEGOFF                ;把段值和偏移送有关寄存器
          MOV     AL,ES:[DI]               ;取字节值
          CALL    SHOWAL                   ;转换并且显示
          ;
          MOV     AX,4C00H                 ;程序结束
          INT     21H
;
;子程序说明信息略
SHOWAL    PROC
          PUSH    AX
          MOV     CL,4
          ROL     AL,CL
          CALL    HTOASC
          CALL    PUTCH
          POP     AX
          CALL    HTOASC
          CALL    PUTCH
          RET
```

```
        SHOWAL    ENDP
;
;子程序说明信息略
GETADR   PROC
GETADR1: CALL   GETSTR              ;接收一个字符串
         CMP    BUFFER,CR           ;字符串是否空
         JNZ    GETADR2             ;不空,转
         CALL   BELL                ;字符串无内容则响铃
         JMP    GETADR1             ;重新接收
GETADR2: MOV    DX,OFFSET  BUFFER
         CALL   HTOBIN              ;转换出段值
         CALL   NEWLINE             ;另起一行
         RET
GETADR   ENDP
;
;子程序说明信息略
GETSTR   PROC
         MOV    DI,OFFSET  BUFFER   ;置缓冲区首地址
         MOV    BX,0                ;清接收字符数计数器
GETSTR1: CALL   GETCH               ;得一个字符
         CMP    AL,CR               ;是否是回车键
         JZ     GETSTR5             ;是,转
         CMP    AL,BACKSPACE        ;否,是否是退格键
         JNZ    GETSTR4             ;否,转
         CMP    BX,0                ;是,判是否有字符可擦除
         JZ     GETSTR2             ;没有字符可擦除,响铃
         DEC    BX                  ;有字符可擦,计数减
         CALL   PUTCH               ;光标回移
         MOV    AL,20H
         CALL   PUTCH               ;显示空格擦原字符
         MOV    AL,BACKSPACE
         CALL   PUTCH               ;光标再回移
         JMP    GETSTR1             ;继续接收
GETSTR2: CALL   BELL
         JMP    GETSTR1
GETSTR4: CMP    BX,4                ;一般键处理
         JZ     GETSTR2             ;如已接收四个字符,响铃
         CALL   ISHEX               ;判是否为十六进制数码符
         JC     GETSTR2             ;否,响铃
         MOV    [BX][DI],AL         ;是,依次保存
         INC    BX                  ;计数加
         CALL   PUTCH               ;显示之
```

```
        JMP     GETSTR1             ;继续接收
GETSTR5: MOV    [BX][DI],AL         ;保存回车符
        RET
GETSTR      ENDP
;
;子程序说明信息略
HTOBIN  PROC
        PUSH    CX
        PUSH    DX                  ;保护寄存器
        PUSH    SI
        MOV     SI,DX               ;置指针
        XOR     DX,DX               ;值清 0
        MOV     CH,4                ;置循环计数初值
        MOV     CL,4                ;置移位位数
HTOBIN1: MOV    AL,[SI]             ;取一位十六进制数
        INC     SI
        CMP     AL,CR               ;是否是回车符
        JZ      HTOBIN2             ;是,转返回
        CALL    ATOBIN              ;十六进制数码符转换成值
        SHL     DX,CL               ;X * 16+Yi
        OR      DL,AL
        DEC     CH                  ;循环控制
        JNZ     HTOBIN1
HTOBIN2: MOV    AX,DX               ;置出口参数
        POP     SI
        POP     DX                  ;恢复寄存器
        POP     CX
        RET
HTOBIN  ENDP
;
;子程序说明信息略
ISHEX   PROC
        CMP     AL,'0'
        JB      ISHEX2
        CMP     AL,'9'+1
        JB      ISHEX1
        CMP     AL,'A'
        JB      ISHEX2
        CMP     AL,'F'+1
        JB      ISHEX1
        CMP     AL,'a'
        JB      ISHEX2
```

```
          CMP     AL,'f'+1
ISHEX1:   CMC
ISHEX2:   RET
ISHEX     ENDP
;
;子程序说明信息略
ATOBIN    PROC
          SUB     AL,30H
          CMP     AL,9
          JBE     ATOBIN1
          SUB     AL,7
          CMP     AL,15
          JBE     ATOBIN1
          SUB     AL,20H
ATOBIN1:  RET
ATOBIN    ENDP
;
;子程序说明信息略
PUTCH     PROC
          PUSH    DX
          MOV     DL,AL
          MOV     AH,2
          INT     21H
          POP     DX
          RET
PUTCH     ENDP
;
;子程序说明信息略
BELL      PROC
          MOV     AL,BELLCH
          CALL    PUTCH
          RET
BELL      ENDP
;
;子程序说明信息略
GETCH     PROC
          MOV     AH,8                    ;接受一个字符但不显示
          INT     21H
          RET
GETCH     ENDP
;
;略去子程序 HTOASC,该子程序列于 4.1 节中
```

```
;略去子程序 DISPMESS,该子程序列于源程序 T4-4.ASM 中
;略去子程序 NEWLINE,该子程序列于源程序 T4-3.ASM 中
CSEG        ENDS
            END     START
```

4.4 磁盘文件管理及应用

DOS 磁盘文件管理功能调用是 DOS 功能调用的重要组成部分,不仅有助于汇编语言程序设计练习,也有助于对磁盘文件管理系统的理解。

4.4.1 DOS 磁盘文件管理功能调用

在下面介绍的 DOS 磁盘文件管理功能调用中,用于表示文件名的 ASCII 字符串必须以 ASCII 码值 0 结尾(不是数字符号 0),这样的字符串通常称为 ASCIIZ 串。文件名可以是包含盘符和路径的文件标识。如没有盘符,那么认为是当前盘,如路径不是从根目录开始,那么就认为从当前目录开始。

这些功能调用均利用标志 CF 表示调用是否成功,如果不成功,那么 AX 含有错误代码。常见的错误代码有:

01	无效的功能号
02	文件未找到
03	路径未找到
04	同时打开文件太多
05	拒绝存取
06	无效的文件号(柄)

1. 建立文件(3CH 号功能调用)

功　　能:建立文件(创建新的,或刷新老的文件)。

入口参数:DS:DX=代表文件名的字符串的首地址。

　　　　　CX=文件属性。

出口参数:CF=0　表示成功,AX=文件号(柄)。

　　　　　CF=1　表示失败,AX=错误代码。

说　　明:(1) 可指定的文件属性如下:

00H	普通
01H	只读
02H	隐含
04H	系统

　　　　　(2) 创建文件成功后,文件长度定为 0。

2. 打开文件(3DH 号功能调用)

功　　能:打开文件。

入口参数:DS：DX＝代表文件名的字符串的首地址。

AL＝存取方式。

出口参数:CF＝0　表示成功,AX＝文件号(柄)。

CF＝1　表示失败,AX＝错误代码。

说　　明:(1) 存取方式规定如下：

00H　　只读方式

01H　　只写方式

02H　　读写方式

(2) 打开文件成功后,文件指针定位于开始的第一个字节(偏移 0)处。

3. 读文件(3FH 号功能调用)

功　　能:读文件。

入口参数:BX＝文件号(柄)。

CX＝读入字节数。

DS：DX＝准备存放所读数据的缓冲区的首地址。

出口参数:CF＝0　表示成功,AX＝实际读到的字节数。

CF＝1　表示失败,AX＝错误代码。

说　　明:(1) 通常情况下,实际读到的字节数与欲读入的字节数相同,除非不够读。

(2) 缓冲区应保证能容下所读到的数据。

(3) 文件应以读或读写方式打开。

(4) 读文件后,文件指针将定位到读出字节之后的第一个字节处。

4. 写文件(40H 号功能调用)

功　　能:写文件。

入口参数:BX＝文件号(柄)；

CX＝写盘字节数；

DS：DX＝存放写数据的缓冲区的首地址。

出口参数:CF＝0　表示成功,AX＝实际写出的字节数。

CF＝1　表示失败;AX＝错误代码。

说　　明:(1) 通常情况下,实际写出的字节数与欲写盘的字节数相同,除非磁盘满。

(2) 文件应以写或读写方式打开。

(3) 写文件后,文件指针将定位到写入字节之后的第一个字节处。

5. 关闭文件(3EH 号功能调用)

功　　能:关闭文件。

入口参数:BX＝文件号(柄)。

出口参数:CF＝0　表示成功。

CF＝1　表示失败。

说　　明:文件号是打开该文件时系统所给定的文件号。

6. 移动文件读写指针(42H 号功能调用)

功　　能:移动文件(读写)指针。

入口参数：BX＝文件号(柄)。

CX：DX＝移动位移量。

AL＝移动方式。

出口参数：CF＝0　表示成功，此时，DX：AX＝移动后文件指针值。

CF＝1　表示失败，此时，(AX)＝1　表示无效的移动方式，(AX)＝6　表示无效的文件号。

说　　明：(1) 文件指针值(双字)是以文件首字节为0计算的。

(2) 移动方式和表示的意义如下：

00H	移动后文件指针值＝0(文件头)＋移动位移量
01H	＝当前文件指针值＋移动位移量
02H	＝文件长(文件尾)＋移动位移量

(3) 在第一种移动方式中，移动位移量总是正的。

(4)在后两种移动方式中，移动位移量可正可负。

(5)该子功能不考虑文件指针是否超出文件范围。

7. 删除文件(41H号功能调用)

功　　能：删除文件。

入口参数：DS：DX＝代表文件名的字符串首地址。

出口参数：CF＝0　表示成功；

CF＝1　表示失败，AX＝错误代码。

说　　明：只能删除一个普通文件。

4.4.2　应用举例

例1：写一个显示文本文件内容的程序。文本文件固定为当前目录下的TEST.TXT文件。

具体算法是：先打开文件；然后顺序读文件，每次读一个字符，把所读字符在屏幕上显示出来，如此循环直到文件结束；最后关闭文件。图4.7是流程图。考虑到TEST.TXT是文本文件，所以认为ASCII码值为1AH的字符就是文件结束符。也就是说，在读到ASCII码值为1AH的字符，就认为文件结束。

设计一个子程序READCH，它每次从文件中顺序读一个字符。这个子程序通过进位标志CF来反映是否正确地读到字符，如果读时发生错误，则CF置位，否则CF被清。调用它的程序应通过CF判别读文件操作是否成功。考虑到万一文本文件没有文件结束符的情况，所以该子程序还判别是否的确已读到文件尾(如果实际读到的字符数为0就意味着文件结束)，当这种情况发生时，就返回一个文件结束符。源程序如下：

```
;程序名:T4-6.ASM
;功  能:显示当前目录下的文本文件 TEST.TXT 内容
;符号常量定义
EOF     =   1AH                          ;文件结束符 ASCII 码
;数据段
```

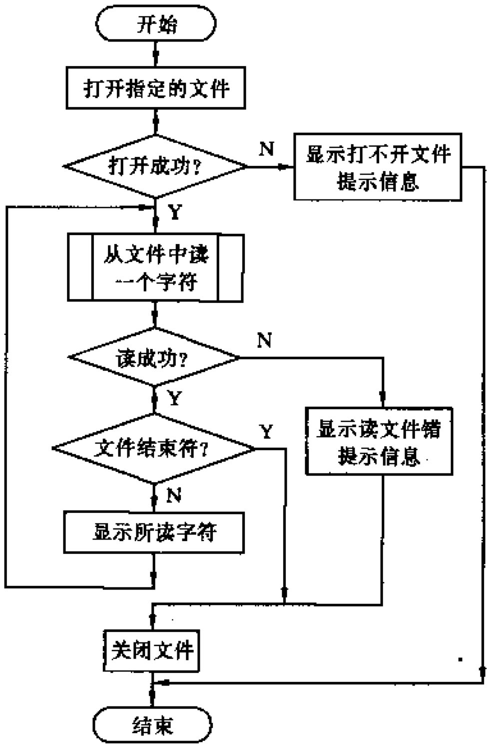


图 4.7 程序 T4-6.ASM 的流程图

```
DSEG     SEGMENT
FNAME    DB 'TEST.TXT',0                    ;文件名
ERROR1   DB 'File  not  found',  07H,0      ;提示信息
ERROR2   DB 'Reading  error',  07H,0
BUFFER   DB ?                               ;1字节的缓冲区
DSEG     ENDS
;代码段
CSEG     SEGMENT
         ASSUME  CS:CSEG,  DS:DSEG
START:   MOV     AX,DSEG
         MOV     DS,AX                      ;置数据段寄存器
         ;
         MOV     DX,OFFSET  FNAME
         MOV     AX,3D00H                   ;为读打开指定文件
         INT     21H
         JNC     OPEN_OK                    ;打开成功,转
         ;
         MOV     SI,OFFSET  ERROR1
         CALL    DMESS                      ;显示打开不成功提示信息
```

```
        JMP     OVER
        ;
OPEN_OK:MOV     BX,AX                   ;保存文件柄
CONT:   CALL    READCH                  ;从文件中读一个字符
        JC      READERR                 ;如读出错,则转
        CMP     AL,EOF                  ;读到文件结束符吗?
        JZ      TYPE_OK                 ;是,转
        CALL    PUTCH                   ;显示所读字符
        JMP     CONT                    ;继续
        ;
READERR:MOV SI,OFFSET   ERROR2
        CALL    DMESS                   ;显示读出错提示信息
        ;
TYPE_OK:MOV     AH,3EH                  ;关闭文件
        INT     21H
OVER:   MOV     AH,4CH                  ;程序结束
        INT     21H
;
;子程序说明信息略
READCH      PROC
        MOV     CX,1                    ;置读字节数
        MOV     DX,OFFSET   BUFFER      ;置读缓冲区地址
        MOV     AH,3FH                  ;置功能调用号
        INT     21H                     ;读
        JC      READCH2                 ;读出错,转
        CMP     AX,CX                   ;判文件是否结束
        MOV     AL,EOF                  ;设文件已结束,置文件结束符
        JB      READCH1                 ;文件确已结束,转
        MOV     AL,BUFFER               ;文件未结束,取所读字符
READCH1:CLC
READCH2:RET
READCH      ENDP
;
;子程序名: DMESS
;功    能: 显示一个以 0 为结束符的字符串
;入口参数: SI=字符串首地址
;出口参数: 无
DMESS       PROC
DMESS1: MOV     DL,[SI]
        INC     SI
        OR      DL,DL
        JZ      DMESS2
```

```
          MOV     AH,2
          INT     21H
          JMP     DMESS1
DMESS2:   RET
DMESS     ENDP
;
PUTCH     PROC
;同 T4-5.ASM 中的子程序 PUTCH
PUTCH     ENDP
CSEG      ENDS
          END     START
```

上述程序每次只从文件中读一个字节数据，读者可把它改为每次从文件中读若干字节数据。如果文件 TEST.TXT 的中间带有文件结束符(1AH)，则在结束符后的数据就不会在屏幕上显示出来，请读者修改上述程序，使其不考虑文件结束符。

例 2：写一个能把键盘上输入的全部字符(直到 CTRL+Z 键，值 1AH)存入某个文件的程序。为简单起见，文件固定为当前盘根目录下的 TEST.TXT，如果它已存在，则更新它。

具体算法是：先建立指定文件；然后读键盘，把所读字符顺序写入文件，如此循环直到读到文件结束符(1AH)；关闭文件。源程序如下：

```
;程序名:T4-7.ASM
;功  能:把键盘上输入的字符全部存入文件 TEST.TXT
;常量定义
EOF       =      1AH                          ;文件结束符的 ASCII 码
;数据段
DSEG          SEGMENT
FNAME         DB '\TEST.TXT',0                ;文件名
ERRMESS1      DB 'Can not  create  file',07H,'$'
ERRMESS2      DB 'Writing error',07H,'$'      ;提示信息
BUFFER        DB ?                            ;1 字节缓冲区
DSEG          ENDS
;代码段
CSEG      SEGMENT
          ASSUME  CS:CSEG,  DS:DSEG
START:    MOV     AX,DSEG
          MOV     DS,AX                       ;置数据段寄存器
          ;
          MOV     DX,OFFSET FNAME             ;建立文件
          MOV     CX,0                        ;普通文件
          MOV     AH,3CH
          INT     21H
```

```
        JNC     CREA_OK                     ;建立成功,转
        ;
        MOV     DX,OFFSET  ERRMESS1         ;显示不能建立提示信息
        CALL    DISPMESS
        JMP     OVER
        ;
CREA_OK:MOV     BX,AX                       ;保存文件柄
CONT:   CALL    GETCHAR                     ;接收一个键
        PUSH    AX
        CALL    WRITECH                     ;向文件写所读字符
        POP     AX
        JC      WERROR                      ;写出错,转
        CMP     AL,EOF                      ;读到文件结束符吗?
        JNZ     CONT                        ;不是,继续
        JMP     CLOSEF                      ;遇文件结束符,转结束
        ;
WERROR:MOV      DX,OFFSET  ERRMESS2         ;显示写出错提示信息
        CALL    DISPMESS
        ;
CLOSEF: MOV     AH,3EH                      ;关闭文件
        INT     21H
OVER:   MOV     AX,4C00H                    ;程序结束
        INT     21H
;子程序说明信息略
WRITECH     PROC
        MOV     BUFFER,AL                   ;把要写的一字节送入缓冲区
        MOV     DX,OFFSET BUFFER            ;置缓冲区地址
        MOV     CX,1                        ;置写的字节数
        MOV     AH,40H                      ;置功能号
        INT     21H                         ;写
        RET
WRITECH     ENDP
;
;子程序说明信息略
GETCHAR     PROC
        MOV     AH,1
        INT     21H
        RET
GETCHAR     ENDP
;
DISPMESS    PROC
;同 T4-4.ASM 中的 DISPMESS
```

```
DISPMESS     ENDP
CSEG      ENDS
          END     START
```

上述程序每次只向文件写一个字节，读者可把它改成每次向文件写若干字节。它通过1号功能调用读键盘，然后就把所读的字符写入文件，如果实际运行这个程序，也许读者会发现在按退格键和回车键时有些特殊异常，如何解决就作为练习留给读者。

例 3：写一个程序把文件 2 拼接到文件 1 上。文件 1 固定为当前目录下的 TEST1，文件 2 固定为当前目录下的 TEST2。

具体算法是：为写打开文件 TEST1，为读打开文件 TEST2；把文件 TEST1 的读写指针移到尾；读 TEST2 的一块到缓冲区，写这一个块到 TEST1，如此循环，直到 TEST2 结束；最后关闭两个文件。源程序如下：

```
;程序名:T4-8.ASM
;功  能:把文件 TEST2 拼接到文件 TEST1 之后
;符号常量定义
BUFFLEN    =    512
;数据段
DSEG       SEGMENT
FNAME1     DB 'TEST1',0                     ;文件名 1
FNAME2     DB 'TEST2',0                     ;文件名 2
HANDLE1    DW 0                             ;存放文件 1 的文件柄
HANDLE2    DW 0                             ;存放文件 2 的文件柄
ERRMESS1   DB 'Can not  open file',07H,'$'
ERRMESS2   DB 'Reading  error',07h,'$'
ERRMESS3   DB 'Writing  error',07H,'$'
BUFFER     DB BUFFLEN  DUP  (0)             ;缓冲区
DSEG       ENDS
;代码段
CSEG    SEGMENT
        ASSUME  CS:CSEG,  DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX                       ;置数据段寄存器
        ;
        MOV     DX,OFFSET  FNAME1           ;为写打开文件 1
        MOV     AX,3D01H
        INT     21H
        JNC     OPENOK1                     ;成功,转
        ;
ERR1:   MOV     DX,OFFSET  ERRMESS1         ;显示打开文件不成功提示信息
        CALL    DISPMESS
        JMP     OVER                        ;转结束
```

```
        ;
OPENOK1:MOV    HANDLE1,AX              ;保存文件1的柄
        MOV    DX,OFFSET  FNAME2       ;为读打开文件2
        MOV    AX,3D00H
        INT    21H
        JNC    OPENOK2                 ;成功,转
        ;
        MOV    BX,HANDLE1              ;如文件2打开不成功
        MOV    AH,3EH                  ;则关闭文件1
        INT    21H
        JMP    ERR1                    ;再显示提示信息
        ;
OPENOK2:MOV    HANDLE2,AX              ;保存文件2的柄
        ;
        MOV    BX,HANDLE1
        XOR    CX,CX
        XOR    DX,DX
        MOV    AX,4202H                ;移动文件1的指针到文件尾
        INT    21H
        ;
CONT:   MOV    DX,OFFSET  BUFFER       ;读文件2
        MOV    CX,BUFFLEN
        MOV    BX,HANDLE2
        MOV    AH,3FH
        INT    21H
        JC     RERR                    ;读出错,转
        OR     AX,AX                   ;文件2读完了?
        JZ     COPYOK                  ;是,转结束
        MOV    CX,AX                   ;写到文件2的长度等于读出的长度
        MOV    BX,HANDLE1
        MOV    AH,40H                  ;写到文件2
        INT    21H
        JNC    CONT                    ;写正确,继续
        ;
WERR:   MOV    DX,OFFSET  ERRMESS3
        CALL   DISPMESS                ;显示写出错提示信息
        JMP    SHORT  COPYOK
        ;
RERR:   MOV    DX,OFFSET  ERRMESS2
        CALL   DISPMESS                ;显示读出错提示信息
        ;
COPYOK:MOV     BX,HANDLE1              ;关闭文件
```

```
        MOV     AH,3EH
        INT     21H
        MOV     BX,HANDLE2
        MOV     AH,3EH
        INT     21H
        ;
OVER:   MOV     AH,4CH                  ;程序结束
        INT     21H
        ;
DISPMESS    PROC
;同 T4-4.ASM 中的 DISPMESS
DISPMESS    ENDP
CSEG    ENDS
        END     START
```

在上述程序中，文件名是固定的，读者可把它改成文件名由用户输入。读者也可把它扩展为把两个文件合并成第三个文件。这些都留给读者作为练习。

4.5 子程序的递归和重入

子程序是否可递归和是否可重入是子程序的两个重要特性，下面简单介绍这两个基本概念。

4.5.1 递归子程序

如果一个子程序直接调用它自身，这种调用称为直接递归调用；如果一个子程序间接调用它自身，这种调用称为间接递归调用。具有递归调用的子程序就称为递归子程序。递归是嵌套的特殊情形。

递归子程序必须采用寄存器或堆栈传递参数。递归的深度受堆栈空间的限制。

下面的子程序 FACT 采用递归算法实现求阶乘，是递归子程序。

```
;子程序名：FACT
;功    能：计算 n!
;入口参数：(AX)=n
;出口参数：(AX)=n!
;说    明：(1) 采用递归算法实现求阶乘；
;          (2)n 不能超过 8。
FACT    PROC
        PUSH    DX
        MOV     DX,AX
        CMP     AX,0                    ;n 为 0?
        JZ      DONE                    ;是，转
        DEC     AX                      ;否，则 n-1
```

```
        CALL    FACT                ;求(n-1)!
        MUL     DX                  ;n*(n-1)!
        POP     DX
        RET
DONE:   MOV     AX,1                ;0! =1
        POP     DX
        RET
FACT    ENDP
```

该子程序限制入口参数 n 大小的主要原因,是只采用 16 位表示阶乘值。

4.5.2 可重入子程序

子程序的重入是指子程序在中断后被重新调用。子程序的重入不同于子程序的递归,重入是被动行为,而递归是主动行为,重入前的调用和重入调用往往是不相干的,而递归调用前后却是密切相关的。

我们把能够重入的子程序称为可重入子程序。在设计可重入子程序时,必须注意如下几点:

(1) 不能利用约定存储单元传递参数。

下面的子程序 NRENT 的功能与子程序 HTOASC 相同,实现把一位 16 进制数转换为对应的 ASCII 码,但利用存储单元传递参数。

```
;入口参数:欲转换的一位 16 进制数在变量 HVAR 中
;出口参数:对应的 ASCII 码在变量 RESULT 中
NRENT   PROC
        PUSH    AX              ;(1)
        MOV     AL,HVAR         ;(2)
        AND     AL,0FH          ;(3)
        ADD     AL,30H          ;(4)
        CMP     AL,39H          ;(5)
        JBE     HOK             ;(6)
        ADD     AL,7            ;(7)
HOK:    MOV     RESULT,AL       ;(8)
        POP     AX              ;(9)
        RET                     ;(10)
NRENT   ENDP
```

如果在执行该子程序的第(1)或(2)条指令前被中断,并且中断处理程序重新调用该子程序,那么入口参数就会被破坏,所以原调用的返回就不正确。如果在执行该子程序的第(8)条指令后被中断,并且中断处理程序重新调用该子程序,那么第一次调用的出口参数就会被破坏。所以上述子程序是不可重入的。

(2) 不能使用约定的存储单元保存中间值。

原因是约定存储单元保存的中间值在重入时可能被破坏。如果子程序要使用临时变

量保存中间值，那么临时变量须安排在堆栈中。下面的例子说明如何把临时变量安排在堆栈中。

```
;子程序名: SLEN
;功    能: 测字符串的长度
;入口参数: (DS:SI)=字符串首地址
;出口参数: (AX)=字符串长度
;说    明: 字符串以0结尾
SLEN     PROC   NEAR
         PUSH   BP
         MOV    BP,SP
         SUB    SP,2
         MOV    WORD  PTR  [BP-2],0
SLEN1:   MOV    AL,[SI]
         INC    SI
         OR     AL,AL
         JZ     SLEN2
         INC     WORD    PTR
[BP-2]
         JMP    SLEN1
SLEN2:   MOV    AX,[BP-2]
         MOV    SP,BP
         POP    BP
         RET
SLEN     ENDP
```

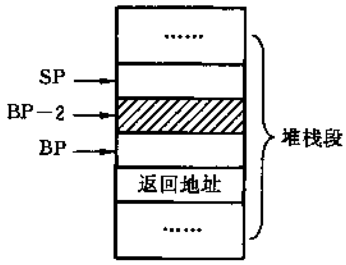


图 4.8 调用子程序 LEN 后的堆栈

调用上述子程序后的堆栈如图 4.8 所示，BP-2 所指单元为计数器使用的临时单元。

4.6 习 题

题 4.1 把怎样的程序片段设计成子程序或者过程？

题 4.2 子程序说明信息应包含哪些内容？举例说明之。

题 4.3 请把过程调用指令 CALL 与无条件转移指令 JMP 作一番比较，说明它们的异同。

题 4.4 是否可用段内调用指令调用远过程？如果可以，请举例说明。

题 4.5 是否可用过程返回指令 RET 调用某个过程？如果可能，请写出实现的程序片段。

题 4.6 请对保护和恢复寄存器的两种方法作比较。

题 4.7 在汇编语言中，主程序与子程序之间如何传递参数？请举例说明每种方法，并对这些方法作比较。

题 4.8 如果利用堆栈传递参数，那么有两种平衡堆栈的方法，请比较这两种方法。

题 4.9 子程序的递归和重入有何异同？

题 4.10 对于可重入的子程序，为什么不能利用固定的存储单元保存中间值？为什么不能利用约定存储单元传递参数？

题 4.11 编写一个求 32 位补码的子程序。通过寄存器传递出入口参数。

题 4.12 编写一个利用查表的方法实现把 1 位十六进制数转换为对应 ASCII 码的子程序。出入口参数传递的方法自定。

题 4.13 按要求分别编写实现如下功能的子程序：把由十进制数 ASCII 码组成的字符串转换为对应的数值。过程 SUBA 通过寄存器传递入口参数，通过寄存器传递出口参数。过程 SUBB 通过堆栈传递入口参数，通过寄存器传递出口参数。过程 SUBC 通过堆栈传递入口参数，通过堆栈传递出口参数。作为入口参数的字符串首地址由段值和偏移构成，其他参数或要求自定。

题 4.14 按要求分别编写实现如下功能的子程序：把 16 位二进制数转换为对应十进制数 ASCII 码串。作为入口参数的二进制数是有符号的，采用补码形式表示。通过堆栈传递入口参数。过程 SUBA 是近过程，堆栈由主程序平衡。过程 SUBB 是近过程，堆栈由子程序平衡。过程 SUBC 是远过程，堆栈由主程序平衡。过程 SUBD 是远过程，堆栈由子程序平衡。

题 4.15 编写具有如下功能的子程序：把 32 位无符号二进制数转换为对应十进制数 ASCII 码串。作为入口参数的指示存放 ASCII 码串缓冲区首地址由段值和偏移两部分构成，其他具体要求与习题 4.14 相同。

题 4.16 写一个程序在屏幕上依次循环显示 10 个数字符号，每行显示 13 个。最初所显示的两行如下所示：

```
0 1 2 3 4 5 6 7 8 9 0 1 2
3 4 5 6 7 8 9 0 1 2 3 4 5
```

题 4.17 写一个程序实现习题 4.16 的功能，但在按回车键时，结束程序。

题 4.18 写一个程序把从键盘上接收到的小写字母用大写字母显示出来，其他字符原样显示。按回车键结束程序。

题 4.19 写一个程序实现如下功能：先从键盘上输入一个字符串，然后再在另一行按相反顺序显示该字符串。

题 4.20 写一个程序显示所按字符键对应的 ASCII 码。先用 2 位十六进制数显示 ASCII 码；再用 3 位八进制数显示 ASCII 码。

题 4.21 写一个程序实现如下功能：先从键盘上输入一个字符串，然后显示该字符串中非数字字符或字母字符的个数。

题 4.22 写一个程序实现如下功能：先从键盘上输入一个字符串，然后在下一行显示滤去字母符号后的字符串；最后在另一行显示大小写字母翻转的字符串。

题 4.23 写一个程序实现如下功能：先从键盘上输入一个较长的字符串和一个较短的字符串，然后判断较短的字符串是否是较长字符串的子串，最后显示提示信息说明判断结果。

题 4.24 设 $A_1=0, A_2=1$，当 $n>=3$ 时，$A_n=A_{n-1}+3*A_{n-2}$。请编写一个求 A_n 的子程序，要求采用递归算法。

题 4.25 请编写一个利用选择法实现排序的子程序。要求该子程序具有可重入性，并且至少使用一个安排在堆栈中的变量。

题 4.26 写一个程序在屏幕上（或在打印机上）列出小于 65535 的素数（质数）。

题 4.27 请写一个程序实现如下功能：把从内存单元 F000：0000H 开始的 200 个字节作为无符号整数，求它们的和，并用十进制数在屏幕上显示出来；把该区域作为 100 个无符号 16 位的字，求它们的和，并用十进制数在屏幕上显示出来。

题 4.28 请写一个程序实现如下功能：把内存单元 F000：0000H 开始的 1024 个字节作为有符号数，分别统计其中的正数、负数和 0 的个数，并显示。

题 4.29 请写一个程序实现如下功能：把指定开始地址的内存区域作为存放 16 位字数组的缓冲区，依次顺序显示其值。具体要求是：开始地址由键盘输入；每此在一行中以多种进制形式显示一个字单元的内容，行首标上用十六进制表示的存储单元的段值和偏移。

题 4.30 请编写一个能够按 DEBUG 的 D 命令格式显示内存单元内容的小型工具程序。

题 4.31 请编写一个能够在最低端的 640K 内存区域内搜索指定信息的小型工具程序。使用说明自定。

题 4.32 请编写一个实现多种数制（十六进制、十进制、八进制和二进制）转换的小型工具程序。适用于小范围内的整数，其他要求和说明自定。

题 4.33 请编写一个可实现两个在 0－65535 范围内的整数进行加、减、乘或除运算的小型工具程序。使用说明自定。

题 4.34 写一个能够复制文件的程序。源文件标识和目标文件标识由键盘输入。

题 4.35 改写 4.4.2 节的例 1、例 2 和例 3，使它们更完善。

题 4.36 写一个程序实现如下功能：把内存区域最低端的 1K 字节存放到文件 MEM.DAT 中。

题 4.37 写一个程序实现如下功能：把内存区域最低端的 1K 字节作为 256 个双字，依次把每个双字转换为对应的 8 字节十六进制 ASCII 码串，顺序存放到文件 MEM.TXT 中，每存放一个 8 字节 ASCII 码串，再存放回车和换行符（0DH 和 0AH）。

题 4.38 写一个能够显示当前工作目录下文件 TEST.TXT 长度的程序。改进程序使其可显示任一指定文件的长度，文件标识由键盘输入。

题 4.39 写一个程序统计当前工作目录下文本文件 TEST.TXT 中的各十进制数字符和各英文字母的个数。

题 4.40 写一个能够比较文本文件的程序。设文本文件由若干行组成，文本行以回车符 0DH 和换行符 0AH 结束。比较依行为单位进行。如果对应的两行不等，那么就显示这两行，并标上行号

题 4.41 请问 BC 或 TC 如何实现函数参数的传递？如何安排局部变量？

题 4.42 请问 Turbo Pascal 如何实现函数参数的传递？如何安排局部变量？

第5章　输入输出与中断

通常用汇编语言编写与硬件关系密切的程序。输入输出和中断与硬件关系密切，所以相关的程序往往用汇编语言编写。本章介绍输入输出与中断的基本概念，结合实例说明如何用汇编语言编写输入输出程序和中断处理程序。

5.1　输入和输出的基本概念

每种输入输出设备都要通过一个硬件接口或控制器和 CPU 相连。例如，打印机通过打印接口与系统相连；显示器通过显示控制器和系统相连。从程序设计的角度看，接口由一组寄存器组成，是完成输入输出的桥梁。程序利用 I/O 指令，存取接口上的寄存器，获得外部设备的状态信息，控制外部设备的动作，从而实现输入输出。

本章所说的输入和输出是站在处理器或主机立场上而言的，也即输入是指输入到处理器或主机，输出是指从处理器或主机输出。

5.1.1　I/O 端口地址和 I/O 指令

1. I/O 端口地址

为了存取接口上的寄存器，系统给这些寄存器分配专门的存取地址，这样的地址称为 I/O 端口地址。

在某些微型机上，I/O 端口地址和存储单元地址统一编址。这相当于把 I/O 接口(设备)视为一个或几个存储单元，利用存取内存单元的指令就可存取接口上的寄存器。但这会减少原本就有限的一部分存储空间，同时由于访问内存的指令一般超过 2 字节，从而延长了外部设备与处理器进行数据交换的时间。

在以 Intel 的 80x86 家族处理器为 CPU 的系统中，I/O 端口地址和存储单元的地址是各自独立的，分占两个不同的地址空间。8086/8088 提供的 I/O 端口地址空间达 64K，因而可接 64K 个 8 位端口，或可接 32K 个 16 位端口。但实际上，PC 及其兼容机一般只使用 0 到 3FFH 之间的 I/O 端口地址，只占整个 I/O 端口地址空间的很小一部分。

2. I/O 指令

由于 8086/8088 的 I/O 端口地址和内存单元地址是独立的，所以要用专门的 I/O 指令来存取端口上的寄存器，也就是说要用专门的 I/O 指令进行输入输出。

I/O 指令属于数据传送指令组。

(1) 输入指令

输入指令的一般格式如下：

```
IN    累加器,端口地址
```

输入指令从一个输入端口读取一个字节或一个字，传送至 AL(若是一个字节)或 AX(若是一个字)。端口地址可采用直接方式表示，也可采用间接方式表示。当采用直接方式表示端口地址时，端口地址仅为 8 位，即 0～255；当采用间接方式表示端口地址时，端口地址存放在 DX 寄存器中，端口地址可为 16 位。所以输入指令有如下四种具体格式：

```
IN    AL, PORT     ;AL ⇐ (PORT)
IN    AX, PORT     ;AX ⇐ (PORT+1 : PORT)
IN    AL, DX       ;AL ⇐ (DX)
IN    AX, DX       ;AX ⇐ (DX+1 : DX)
```

前两种格式是直接端口寻址，端口地址 PORT 是一个 8 位的立即数，例如：

```
IN    AL,21H
```

后两种是间接端口寻址，端口地址在寄存器 DX 中。当端口地址超过 255 时，只能采用 DX 间接端口寻址。例如：

```
MOV   DX,2FCH
IN    AX,DX
```

注意：当从端口 n 输入一个字时，相当于同时从端口 n 和 n+1 分别读取一个字节。如果上述两条指令连续执行，相当于从端口 2FCH 输入一个字节送 AL，从 2FDH 输入一个字节送 AH。

(2) 输出指令

输出指令的一般格式如下：

```
OUT   端口地址,累加器
```

输出指令将 AL 中的一个字节，或在 AX 中的一个字，输出到指定端口。像输入指令一样，端口地址可采用直接方式表示，也可采用间接方式表示。当采用直接方式表示端口地址时，端口地址仅为 8 位，即 0～255；当采用间接方式表示端口地址时，端口地址存放在 DX 寄存器中，端口地址可为 16 位。所以输出指令也有如下四种具体格式：

```
OUT   PORT,AL     ;(PORT) ⇐ AL
OUT   PORT,AX     ;(PORT+1 : PORT) ⇐ AX
OUT   DX,AL       ;(DX)⇐ AL
OUT   DX,AX       ;(DX+1 : DX) ⇐ AX
```

注意：当向端口 n 输出一个字时，相当于向端口 n 输出 AL 中的内容和向端口 n+1 输出 AH 中的内容。例如，下面的程序片段向 2FCH 端口输出 23H，向 2FDH 端口输出 45H：

```
MOV   AX,4523H
MOV   DX,2FCH
OUT   DX,AX
```

5.1.2 数据传送方式

1. CPU 与外设之间交换的信息

CPU 与外设之间交换的信息包括数据、控制和状态信息。尽管这三种信息具有不同

性质，但它们都通过 IN 和 OUT 指令在数据总线上进行传送，所以通常采用分配不同端口的方法将它们加以区别。

数据是 CPU 和外设真正要交换的信息。数据通常为 8 位或 16 位，可分为各种不同类型。不同的外设要传送的数据类型也是不同的。

控制信息输出到 I/O 接口，告诉接口和设备要做什么工作。

从接口输入的状态信息表示 I/O 设备当前的状态。在输入数据前，通常要先取得表示设备是否已准备好的状态信息；在输出数据前，往往要先取得表示设备是否忙的状态信息。

2. 数据传送方式

系统中数据传送的方式主要有：

(1) 无条件传送方式

在不需要查询外设的状态，即已知外设已准备好或不忙时，可以直接使用 IN 或 OUT 指令实现数据传送。这种方式软件实现简单，只要在指令中指明端口地址，就可选通指定外设进行输入输出。

无条件传送方式是方便的，但要求外设工作速度能与 CPU 同步，否则就可能出错。例如，在外设还没有准备好的情况下，就用 IN 指令得到的数据就可能是不正确的数据。

(2) 查询方式

查询传送方式适用于 CPU 与外设不同步的情况。输入之前，查询外设数据是否已准备好，若数据已准备好，则输入；否则继续查询，直到数据准备好。输出之前，查询外设是否"忙"，若不"忙"，则输出；否则继续查询，直到不"忙"。也就是说，要等待到外设准备好时才能输入或输出数据，而通常外设速度远远慢于 CPU 速度，于是查询过程就将花费大量的时间。

(3) 中断方式

为了提高 CPU 的效率，可采用中断方式。当外设准备好时，外设向 CPU 发出中断请求，CPU 转入中断处理程序，完成输入输出工作。

(4) 直接存储器传送(DMA)方式

由于高速 I/O 设备(如磁盘机等)准备数据的时间短，要求传送速度快等特点，所以一般采用直接存储器传送方式，即高速设备与内存储器直接交换数据。这种方式传送数据是成组进行的。其过程是：先把数据在高速外设中存放的起始位置、数据在内存储器中存放的起始地址、传送数据长度等参数输出到连接高速外设的接口(控制器)，然后启动高速外设，设备准备开始直接传送数据。当高速外设直接传送准备好后，向处理机发送一个直接传送的请求信号，处理机以最短时间批准进行直接传送，并让出总线控制权，高速外设在其控制器控制下交换数据。数据交换完毕后，由高速外设发出"完成中断请求"，并交回总线控制权。处理机响应上述中断，由对应的中断处理程序对高速外设进行控制或对已经传送的数据进行处理，中断返回后，原程序继续运行。

5.1.3 存取 RT/CMOS RAM

1. 关于 RT/CMOS RAM

在 IBM PC/AT 及其兼容机上均安装有一个 RT/COMS RAM 芯片，它是互补金属

氧化物半导体随机存取存储器，不仅可长期保存系统配置状况，而且记录包括世纪、年、月、日和时分秒在内的实时钟(Real_Time Clock)信息。

RT/CMOS RAM 作为一个 I/O 接口芯片，系统分配的 I/O 端口地址区为 70H 至 7FH，通过 IN 和 OUT 指令可对其进行存取。它共提供 64 个字节 RAM 单元，分配使用情况如表 5.1 所示。前 14 个字节用于实时钟，剩下的 50 个字节用于系统配置。

表 5.1 RT/CMOS RAM 内部信息布局

位移	用　途	位移	用　途
0	秒	10	软盘驱动器类型
1	报警秒	11	保留
2	分	12	硬盘驱动器类型
3	报警分	13	保留
4	时	14	设备标志
5	报警时	15	常规 RAM 容量低字节
6	星期	16	常规 RAM 容量高字节
7	日	17	扩展 RAM 容量低字节
8	月	18	扩展 RAM 容量高字节
9	年	19-1A	硬驱类型扩展字节
A-D	状态寄存器 A 至 D	1B-2D	保留
E	诊断状态	2E-2F	配置信息字节累加和
F	停止状态	30-3F	其它(含世纪信息)

2. 存取 RT/CMOS RAM

在存取 RT/CMOS RAM 芯片内部的 64 个字节内容时，往往要分两步进行。即先把要存取单元的地址送端口 70H，然后再存取端口 71H。还需注意，14 个记录实时钟信息的单元(位移 0 置 0DH)的地址就是表中位移，其他单元的地址是表 5.1 所示位移上加 80H。

(1) 读操作代码片段如下：

```
MOV   AL,n         ;n 是要访问单元地址
OUT   70H,AL       ;把要访问单元的地址送地址端口
JMP   $+2          ;延时
IN    AL,71H       ;从数据端口取访问单元的内容
```

(2) 写操作代码片段如下：

```
MOV   AL,n         ;n 是要访问单元地址
OUT   70H,AL       ;把要访问单元的地址送地址端口
JMP   $+2          ;延时
```

```
MOV   AL,m          ;m是要输出数据
OUT   71H,AL        ;把数据从数据端口输出
```

需要指出，在对同一个I/O设备或端口相继发送I/O指令时，为确保I/O设备或端口有足够的电路恢复时间，一般在I/O指令之间使用一条转移指令"JMP $+2"。该指令的动作是转移到下一条指令执行，其意义是延时，以满足I/O端口的需要。

3. CMOS RAM累加和检查

在对系统配置时，要对CMOS RAM的位移10H～2DH的系统配置信息按字节求累加和，其值存放在位移2E～2FH的单元中。低字节存放在2EH单元中，高字节存放在2FH单元中。在系统加电自检时，将对CMOS RAM的位移10H～2DH单元的内容求和，并与保存的累加和比较。若两者不等，则置诊断状态字节的第6位，表明累加和错。这往往会导致要求用户重新进行系统配置。

下面的程序片段进行累加和检查，在累加和错时设置诊断状态字节的位6。

```
CMOS_PORT     EQU   70H              ;CMOS端口地址
CMOS_BEGIN    EQU   90H              ;求和开始地址
CMOS_END      EQU   ADH              ;求和结束地址
CHECK_SUM     EQU   AEH              ;累加和存放开始地址
DIAG_STATUS   EQU   8EH              ;诊断状态字节地址
BAD_CKSUM     EQU   40H              ;累加和检查错标志位
        ;
        SUB   BX,BX                  ;累加和清0
        MOV   CL,CMOS_BEGIN          ;指向开始地址
        MOV   CH,CMOS_END+1          ;指向结束地址后一单元
        SUB   AH,AH
        ;
CMOS2:  MOV   AL,CL
        OUT   CMOS_PORT,AL
        JMP   $+2
        IN    AL,CMOS_PORT+1         ;取一字节
        ADD   BX,AX                  ;求累加和
        INC   CL                     ;指向下一字节
        CMP   CH,CL                  ;是否到结束地址
        JNZ   CMOS2                  ;未到，继续求和
        ;
        MOV   AL,CHECK_SUM+1
        OUT   CMOS_PORT,AL
        JMP   $+2
        IN    AL,CMOS_PORT+1         ;取原累加和高字节
        MOV   AH,AL                  ;保存到AH
        MOV   AL,CHECK_SUM
        OUT   CMOS_PORT,AL
```

```
        JMP     $+2
        IN      AL,CMOS_PORT+1          ;取原累加和低字节
        CMP     AX,BX                   ;比较
        JZ      CMOS4                   ;等,转
        ;
        MOV     AL,DIAG STATU
        OUT     CMOS_PORT,AL
        JMP     $+2
        IN      AL,CMOS_PORT+1          ;取诊断状态字节
        MOV     AH,AL                   ;送 AH
        OR      AH,BAD_CKSUM            ;或上累加和检查错标志
        MOV     AL,DIAG_STATUS
        OUT     CMOS_PORT,AL
        JMP     $+2
        MOV     AL,AH
        OUT     CMOS_PORT+1,AL          ;再送回诊断字节
CMOS4:  ……
```

5.2 查询方式传送数据

本节在介绍查询传送方式概念后,举例说明查询传送方式的实现。

5.2.1 查询传送方式

查询方式的基本思想是由 CPU 主动地通过输入输出指令查询指定的外部设备的当前状态,若设备就绪,则立即与设备进行数据交换,否则循环查询。具体地说,在输入之前,要查询外设的数据是否已准备好,直到外设把数据准备好后才输入;在输出之前,要查询外设是否"忙",直到外设不"忙"后才输出。查询传送方式适用于 CPU 与外设不同步的情况。查询方式输入输出的示意流程如图 5.1 所示。

为了采用查询方式输入或输出,相应的外设(或接口)不仅要有数据寄存器,而且还要有状态寄存器,有些外设还有控制寄存器。数据寄存器用来存放要传送的数据,状态寄存器用来存放表示设备所处状态的信息。通常,在状态寄存器中有一个"就绪(Ready)"位或一个"忙(Busy)"位来反映外设是否已准备好。

在实际应用中,为防止设备因某种原因发生故障而无法就绪或空闲,从而导致 CPU 在无限循环之中,通常都设计一个等待超时值,其值随设备而定。一旦设备在规定时间内还无法就绪或空闲,也中止循环查询过程。如此,图 5.1 所示的流程图修改为图 5.2 所示的流程图。大多数情况下,等待超时值用查询次数表示,每查询一次,查询次数减一,如果查询次数减到 0,那么查询等待也就结束。

有时系统中同时有几个设备要求输入输出数据,那么对每个设备都可编写一段执行输入输出数据的程序,然后轮流查询这些设备的状态寄存器中的就绪位,当某一设备准备

好允许输入或输出数据时，就调用这个设备的I/O程序完成数据传送，否则依次查询下一设备是否准备好。

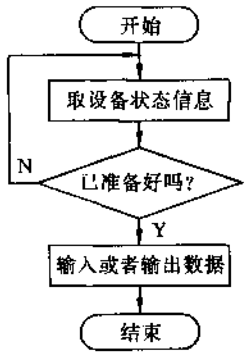


图 5.1 查询方式输入输出

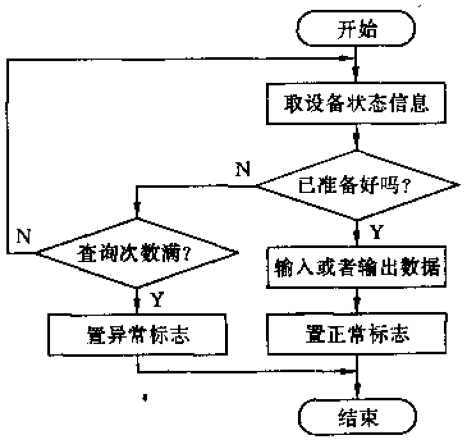


图 5.2 查询方式的一般流程图

查询方式的优点是：软硬件实现比较简单；当同时查询多个外设时，可以由程序安排查询的先后次序。缺点是浪费了CPU原本可执行大量指令的时间。

5.2.2 读实时钟

RT/CMOS RAM的状态寄存器A的位7是计时更新标志位，为1表示实时钟正在计时，为0表示实时钟信息可用于读出。所以，在读实时钟前，要判别该标志位是否为0。

下面的程序片段读实时种，并把读到的秒、分和时数保存到相应的变量中。

```
CMOS_PORT     EQU    70H              ;CMOS端口地址
CMOS_REGA     EQU    0AH              ;状态寄存器A地址
UPDATE_F      EQU    80H              ;更新标志位
CMOS_SEC      EQU    00H              ;秒单元地址
CMOS_MIN      EQU    02H              ;分单元地址
CMOS_HOUR     EQU    04H              ;时单元地址
          ;
          ......
SECOND    DB    ?                     ;秒数保存单元
MINUTE    DB    ?                     ;分数保存单元
HOUR      DB    ?                     ;时数保存单元
          ......
        ;
        MOV     AX,SEG SECOND         ;设置数据段寄存器值
        MOV     DS,AX
        ;
UIP:    MOV     AL,CMOS_REGA          ;判是否可读实时钟
```

```
OUT     CMOS_PORT,AL          ;准备读状态寄存器A
JMP     $+2
IN      AL,CMOS_PORT+1        ;读状态寄存器A
TEST    AL,UPDATE_F           ;测更新标志
JNZ     UIP                   ;如不可读则继续测试
;
MOV     AL,CMOS_SEC
OUT     CMOS_PORT,AL
JMP     $+2
IN      AL,CMOS_PORT+1        ;读秒值
MOV     SECOND,AL             ;保存之
MOV     AL,CMOS_MIN
OUT     CMOS_PORT,AL
JMP     $+2
IN      AL,CMOS_PORT+1        ;读分值
MOV     MINUTE,AL             ;保存之
MOV     AL,CMOS_HOUR
OUT     CMOS_PORT,AL
JMP     $+2
IN      AL,CMOS_PORT+1        ;读时值
MOV     HOUR,AL               ;保存之
……
```

把更新标志位理解为状态寄存器中的"就绪"位，上面的程序片段采用查询方式检测是否就绪，但没有限制检测次数，符合如图5.1所示的流程。

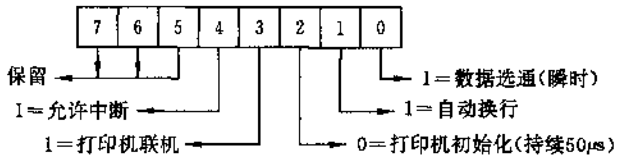
5.2.3 查询方式打印输出

在IBM PC系列及其兼容机上，打印机通过打印接口(打印适配卡)连入系统。打印接口的功能是传递打印命令和数据到打印机并返回打印机状态，为此它包含数据寄存器、状态寄存器和控制寄存器。打印接口的状态寄存器和控制寄存器各位的定义如图5.3所示。

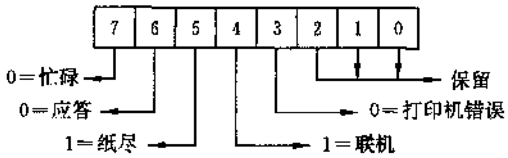
上述三个寄存器有各自的端口地址，并且三个端口地址是连续的。设数据寄存器端口地址是378H，那么，状态寄存器端口地址是379H，控制寄存器端口地址是37AH。在系统加电初始化期间，数据寄存器端口地址被保存到BIOS数据区。

在确定查询次数和取得数据寄存器端口地址后，利用查询方式打印一个字符的流程如图5.4所示。首先输出打印数据，但此时打印机并未接受，随后读取状态信息，判打印机是否忙碌，一直等到打印机不忙碌，才向打印机发出选通命令。下面的子程序实现了上述流程。

```
;子程序名：PRINT
;功    能：打印一个字符
;入口参数：DX=数据寄存器端口地址
;          BL=超时参数
```



(a) 控制寄存器各位定义



(b) 状态寄存器各位定义

图 5.3 状态寄存器和控制寄存器各位的定义

```
;           AL=打印字符的代码
;出口参数: AH=打印机状态,各位意义如下:
;           位 0:1 表示超时,即超过规定的查询次数
;           位 1 和位 2:不用
;           位 3:1 表示出错
;           位 4:1 表示联机
;           位 5:1 表示无纸
;           位 6:1 表示应答
;           位 7:0 表示忙碌
PRINT  PROC
       PUSH  DX
       PUSH  AX
       OUT   DX,AL      ;输出打印数据
       INC   DX         ;使 DX 含状态寄
                         存器端口地址
WAIT:  XOR   CX,CX      ;1 个超时参数单位
                         表示查询 65536 次
WAIT1: IN    AL,DX      ;读取状态信息
       MOV   AH,AL      ;保存到 AH
       TEST  AL,80H     ;测是否忙碌
       JNZ   NEXT       ;不忙碌,则转
       LOOP  WAIT1      ;继续查询
       DEC   BL         ;超时参数减 1
       JNZ   WAIT       ;未超时,继续查询
       AND   AH,0F8H    ;已超时,去掉状态信息中的无用位
       OR    AH,1       ;置超时标志
```

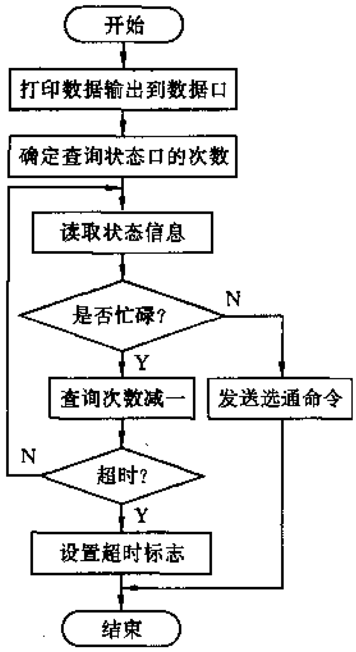


图 5.4 查询方式打印一个字符的流程

```
        JMP     EXIT      ;转结束
NEXT:   INC     DX        ;不忙碌,使 DX 含控制寄存器端口地址
        MOV     AL,0DH    ;准备选通命令
        OUT     DX,AL     ;选通
        MOV     AL,0CH    ;准备复位选通命令
        JMP     $+2
        OUT     DX,AL     ;复位选通位
        AND     AH,0F8H   ;去掉状态信息中的不用位
EXIT:   XOR     AH,48H    ;使返回的状态消息中有关位符合要求
        POP     DX
        MOV     AL,DL     ;恢复 AL 寄存器值
        POP     DX
        RET
PRINT   ENDP
```

在上面的子程序中,使用了一个超时参数,一个超时参数单位表示查询 65536 次,超时参数应根据系统具体情况而定。在发出选通命令和复位选通之间使用了一条转移指令,该指令起延时作用,保证有足够的选通时间。另外,为了使 AH 寄存器含有规定的出口参数,要取反应答位和错误位的值,所以在子程序结束前,用异或指令取反这两位。

5.3 中 断

本节以 PC 系统为背景介绍中断的基本概念和 8086/8088 响应中断的过程。

5.3.1 中断和中断传送方式

1. 中断和中断源

中断是一种使 CPU 挂起正在执行的程序而转去处理特殊事件的操作。这些引起中断的事件称为中断源。它们可能是来自外设的输入输出请求,例如,由按键引起的键盘中断,由串行口接收到信息引起的串行口中断等;也可能是计算机的一些异常事件或其他的内部原因,例如:除数为 0。

2. 中断传送方式

中断传送方式的具体过程是:当 CPU 需要输入或输出数据时,先作一些必要的准备工作(有时包括启动外部设备),然后继续执行程序;当外设完成一个数据的输入或输出后,则向 CPU 发出中断请求,CPU 就挂起正在执行的程序,转去执行输入或输出操作,在完成输入或输出操作后,返回原程序继续执行。

中断传送方式是 CPU 和外部设备进行输入输出的有效方式,一直被大多数计算机所采用,它可以避免因反复查询外部设备的状态而浪费时间,从而提高 CPU 的效率。不过,每中断一次,只传送一次数据,数据传送的效率并不高,所以,中断传送方式一般用于低速外设。另外,与查询方式相比,中断传送方式实现比较复杂,对硬件的条件也较多。

5.3.2 中断向量表

1. 中断向量表

IBM PC 系列及其兼容机共能支持 256 种类型的中断，系统给每一种中断都安排一个中断类型号(简称为中断号)，中断类型号依次为 0～0FFH。例如，属于外部中断的定时器中断类型号为 08 和键盘中断类型号为 09，属于内部中断的除法出错中断类型号为 0 等等。

每种类型的中断都由相应的中断处理程序来处理，为了使系统在响应中断后，CPU 能快速地转入对应的中断处理程序，系统用一张表来保存这些中断处理程序的入口地址，该表就称为中断向量表。中断向量表的每一项保存一个中断处理程序的入口地址，它相当于一个指向中断处理程序的指针，所以就称它为中断向量。中断向量也依次编号为 0～0FFH，n 号中断向量就保存处理中断类型为 n 的中断处理程序的入口地址。所以，一般不再区分中断类型号和中断向量号。

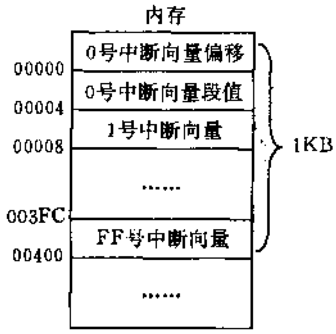


图 5.5 中断向量表

中断向量表如图 5.5 所示，它被安排在内存最低端的 1K 字节空间中。其中每个中断向量占用四个字节，前(低地址)两字节保存中断处理程序入口地址的偏移，后(高地址)两字节保存中断处理程序入口地址的段值，所以含有 256 个中断向量的中断向量表需要占用 1K 字节内存空间。

按照上述中断向量表的结构和存放位置，根据中断向量号(或类型号)可方便地计算出中断向量所在单元的地址。设中断向量号为 n，则中断向量所在单元的开始地址是4 * n

2. 中断向量号的分配

在系统中，某个中断类型号分配给哪个中断，即某个中断向量含有哪个中断处理程序的入口地址有一些规定和约定，应用程序不能违反规定，不宜不遵守约定。

在 IBM PC 系列及其兼容机上，除保留给用户使用的 60H～68H 和 F1H～FFH 中断向量号外，可以认为其他中断向量号已被分配。表 5.2 列出了部分中断向量号的分配情况。

表 5.2 部分中断向量的分配

向量号	使　用	向量号	使　用
0	除法出错	4	溢出
1	单步	5	打印屏幕
2	非屏蔽中断	6	保留
3	断点	7	保留
8	定时器	0C	串行通信接口 1
9	键盘	0D	硬盘(并行口)
0A	保留(从中断控制器)	0E	软盘
0B	串行通信接口 2	0F	打印机

续表

向量号	使　用	向量号	使　用
10	视频显示	17	打印输出
11	设备配置	18	ROM BASIC
12	存储容量	19	系统自举
13	磁盘 I/O	1A	时钟管理
14	串行 I/O	1B	Ctrl+Break 键处理
15	扩充的 BIOS	1C	定时处理
16	键盘输入	1D--1F	参数指针
20--2F	DOS 使用	30--3F	为 DOS 保留

顺便说一下，中断向量不一定非要指向中断处理程序，也可作为指向一组数据的指针。例如，1DH 号中断向量就指向显示器参数，1EH 号中断向量指向软盘基数。当然，如果中断向量 m 没有指向中断处理程序，那么就不应发生类型为 m 的中断。

3. 设置和获取中断向量

在系统程序或应用程序由于某种需要而提供新的中断处理程序时，就要设置对应的中断向量，使其指向新的中断处理程序。

下面的程序片段直接设置 n 号中断向量，假设对应中断处理程序的入口标号是 INTHAND：

```
......
MOV    AX,0
MOV    DS,AX
MOV    BX,n*4                                ;准备设置n号中断向量
CLI                                          ;关中断
MOV    WORD PTR [BX],OFFSET INTHAND          ;置偏移
MOV    WORD PTR [BX+2],SEG INTHAND           ;置段值
STI
......
```

在上面的程序片段中，使用了关中断指令 CLI，目的是保证真正用于设置中断向量的两条传送指令能够连续执行。在执行完前一条传送指令后，n 号中断向量就暂时被破坏，既不指向原中断处理程序，也不指向新的中断处理程序，如果此时发生类型为 n 的中断，那么就不能正确地转到中断处理程序执行，这是最糟糕的事了。如果能确定当前是关中断状态，当然就不再需要使用该关中断指令，也不需要随后的开中断指令。另外，如果能肯定在设置 n 号中断向量的过程中不发生类型为 n 的中断，那么可不考虑是否为关中断状态，这种情况只有在对应的中断处理程序仅供应用程序自己使用时才有可能。

实际上，总是尽量避免采用上述直接设置中断向量的方法(有时是必须的)，而是利用 DOS 提供的 25H 号系统功能调用来设置中断向量，这可避免考虑许多细节。

25H 号系统功能调用是设置中断向量，其入口参数如下：

AL＝中断向量(类型)号

DS＝中断处理程序入口地址的段值

DX＝中断处理程序入口地址的偏移

下面的程序片段设置 n 号中断向量，假设对应中断处理程序入口标号是 INTHAND：

```
……
MOV    AX,SEG  INTHAND
MOV    DS,AX
MOV    DX,OFFSET  INTHAND
MOV    AH,25H
MOV    AL,n
INT    21H
……
```

有时需要取得中断向量。例如：在应用程序要用自己的中断处理程序代替系统中原有的某个中断处理程序时，先要保存原中断向量，待应用程序结束时再恢复原中断向量。

下面的程序片段直接从中断向量表中取得 n 号中断向量，并且保存到双字变量 OLDVECTOR 中：

```
……
XOR    AX,AX
MOV    ES,AX
MOV    AX,ES:[n*4]
MOV    WORD PTR OLDVECTOR,AX
MOV    AX,ES:[n*4+2]
MOV    WORD  PTR  OLDVECTOR+2,AX
……
```

与利用 DOS 功能调用设置中断向量一样，实际上一般都利用 DOS 提供的 35H 号系统功能调用取得中断向量。35H 号系统调用的功能是获取中断向量，其出入口参数如下：

入口参数：

AL＝中断向量(类型)号

出口参数：

ES＝中断处理程序入口地址的段值

BX＝中断处理程序入口地址的偏移

下面的程序片段取得 n 号中断向量，并将其保存到双字变量 OLDVECTOR 中：

```
……
MOV    AH,35H
MOV    AL,N
INT    21H
MOV    WORD  PTR  OLDVECTOR,ES
MOV    WORD  PTR  OLDVECTOR,BX
```

……

5.3.3 中断响应过程

1. 中断响应过程

通常 CPU 在执行完每一条指令后均要检测是否有中断请求，在有中断请求且满足一定条件时就响应中断，这个过程如图 5.6 所示。

在中断响应的过程中，由硬件自动完成如下工作：

(1) 取得中断类型号；

(2) 把标志寄存器内容压入堆栈；

(3) 禁止外部中断和单步中断(使 IF 和 TF 标志位为 0)；

(4) 把下一条要执行指令的地址(中断返回地址)压入堆栈(CS 和 IP 内容压入堆栈)；

(5) 根据中断类型号从中断向量表中取中断处理程序入口地址；

(6) 转入中断处理程序。

在 CPU 响应中断转入中断处理程序时，堆栈如图 5.7 所示。中断处理程序在最后从堆栈中弹出返回地址和原标志值结束中断，返回被中断程序。

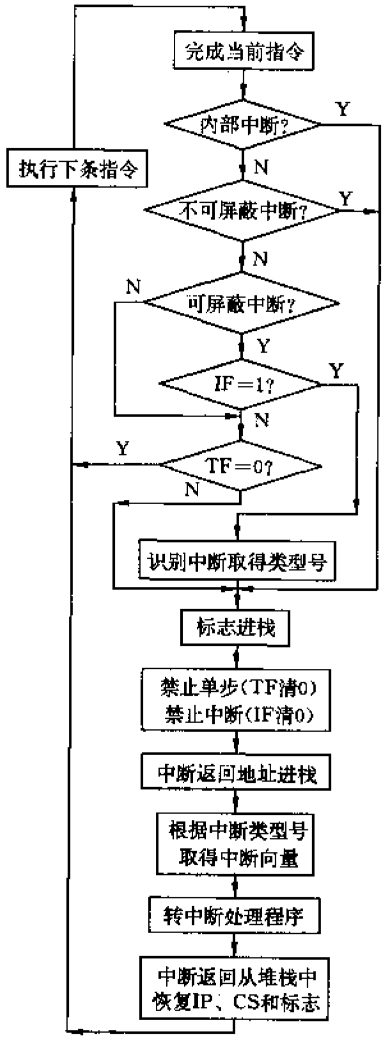


图 5.6　中断响应过程

2. 中断返回指令

中断处理程序利用中断返回指令从堆栈中弹出返回地址和原标志值。中断返回指令的格式如下：

IRET

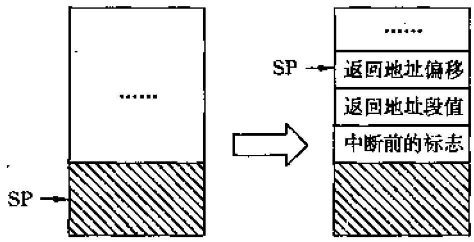
该指令的功能是从中断返回。具体操作如下所示：

IP ⇐ [SP]
SP ⇐ SP+2
CS ⇐ [SP]
SP ⇐ SP+2
FLAGS ⇐[SP]

在执行中断返回指令 IRET 时的堆栈变化如图 5.7(b)到(a)。

5.3.4 外部中断

由发生在 CPU 外部的某个事件引起的中断称为外部中断。如输入输出设备和协处理器等引起的中断就是外部中断。外部中断以完全随机的方式中断现行程序。8086/8088



(a) 中断前的堆栈　　(b) 中断后的堆栈

图 5.7　响应中断时的堆栈

有两条外部中断请求线:INTR 接受可屏蔽中断请求,NMI 接受非屏蔽中断请求。

1. 可屏蔽中断

在 IBM PC 系列及其兼容机中,键盘和硬盘等外设的中断请求都通过中断控制器 8259A 传给可屏蔽中断请求线 INTR,如图 5.8 所示,中断控制器 8259A 共能接收 8 个独立的中断请求信号 IRQ0 至 IRQ7。在 AT 机上,有两个中断控制器 8259A,一主一从,从 8259A 连接到主 8259A 的 IRQ2 上,这样 AT 系统就可接收 15 个独立的中断请求信号。

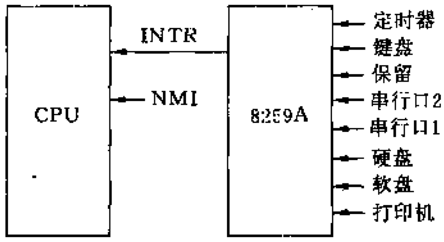


图 5.8　可屏蔽外部中断

中断控制器 8259A 在控制外设中断方面起着重要的作用。如果接收到一个中断请求信号,并且满足一定的条件,那么它就把中断请求信号传到 CPU 的可屏蔽中断请求线 INTR,使 CPU 感知到有外部中断请求;同时也把相应的中断类型号送给 CPU,使 CPU 在响应中断时可根据中断类型号取得中断向量,转相应的中断处理程序。

中断控制器 8259A 是可编程的,也就是说可由程序设置它如何控制中断。在机器系统加电初始化期间,已对 8259A 进行过初始化。在初始化时规定了在传出中断请求 IRQ0 至 IRQ7 时,送出的对应中断类型号分别是 08H～0FH,请参见表 5.2。例如,设传出中断请求 IRQ1,即传出键盘中断请求,那么送出的中断类型号为 9,所以键盘中断的中断类型号为 9,键盘中断处理程序的入口地址存放在 9 号中断向量中。

从普通汇编语言程序设计者的角度看,中断控制器 8259A 包含两个寄存器:中断屏蔽寄存器和中断命令寄存器,它们决定了传出一个中断请求信号的条件。中断屏蔽寄存器的 I/O 端口地址是 21H,它的 8 位对应控制 8 个外部设备,通过设置这个寄存器的某位为 0 或为 1 来允许或禁止相应外部设备中断。当第 i 位为 0 时,表示允许传出来自 IRQi 的中断请求信号,当第 i 位为 1 时,表示禁止传出来自 IRQi 的中断请求信号。中断屏蔽寄存器的内容称为中断屏蔽字。在 PC 系列及其兼容机上,中断屏蔽寄存器各位与对应外设的关系如图 5.9 所示。

例如:为了使中断控制器 8259A 只传出来自键盘的中断请求信号,可设置中断屏蔽字 11111101B,程序片段如下:

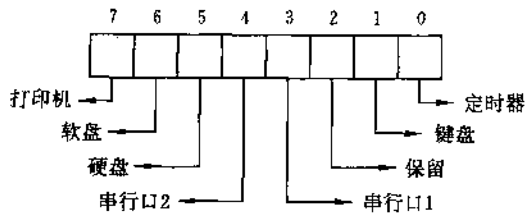


图 5.9 中断屏蔽寄存器

```
MOV    AL,11111101B
OUT    21H,AL
```

例如：下面的程序片段使中断屏蔽寄存器的位 4 为 0，从而允许传出来自串行通信口 1 的中断请求信号：

```
IN     AL,21H
AND    AL,11101111B
OUT    21H,AL
```

从图 5.6 可知，尽管中断控制器把外设的中断请求信号由 INTR 传给 CPU，但 CPU 是否响应还取决于中断允许标志 IF。如果 IF 为 0，则 CPU 仍不响应由 INTR 传入的中断请求；只有在 IF 为 1 时，CPU 才响应由 INTR 传入的中断请求。所以，由 INTR 传入的外部中断请求称为可屏蔽的外部中断请求，由此引起的中断称为可屏蔽中断。由于外设的中断请求均由 INTR 传给 CPU，所以，当 IF 为 0 时，CPU 不响应所有外设中断请求，当 IF 为 1 时，才响应外设中断请求。CPU 响应外设中断请求称为开中断(IF=1)，反之称为关中断(IF=0)。CPU 在响应中断时会自动关中断，从而避免在中断过程中再响应其他外设中断。当然，程序员也可根据需要在程序中安排关中断指令 CLI 和开中断指令 STI。

综上所述，在 IBM PC 系列及其兼容机中，所有的外设中断均是可屏蔽中断；CPU 响应某个外设中断请求的两个必要条件是：中断屏蔽寄存器中的相应位为 0 和处于开中断状态。通过对这两个必要条件的控制，可使 CPU 响应某些外设中断请求，而不响应另外一些外设中断请求。

2. 非屏蔽外部中断

从图 5.6 可知，当收到从 NMI 传来的中断请求信号时，不论是否处于开中断状态，CPU 总会响应。所以，由 NMI 传入的外部中断请求称为非屏蔽外部中断请求，由此而引起的中断称为非屏蔽中断。不可屏蔽中断请求由电源掉电、存储器出错或者总线奇偶校验错等紧急故障产生，要求 CPU 及时处理。在 IBM PC 系列及其兼容机上，非屏蔽中断的中断类型号规定为 2。CPU 在响应非屏蔽中断请求时，总是转入由 2 号中断向量所指定的中断处理程序。

5.3.5 内部中断

由发生在 CPU 内部的某个事件引起的中断称为内部中断。由于内部中断是 CPU 在

执行某些指令时产生，所以也称为软件中断。其特点是：不需要外部硬件的支持；不受中断允许标志 IF 的控制。

1. 中断指令 INT 引起的中断

中断指令的一般格式如下：

```
INT    n
```

其中，n 是一个 0～0FFH 的立即数。CPU 在执行上面的中断指令后，便产生一个类型号为 n 的中断，从而转入对应的中断处理程序。

例如，为了调用 DOS 系统功能，就在程序中安排如下的中断指令：

```
INT    21H
```

当 CPU 执行该指令后，就产生一个类型为 21H 的中断，从而转入对应的中断处理程序，也即转入 DOS 系统功能服务程序。执行中断指令后的堆栈如图 5.7 所示。

值得指出的是，程序员根据需要在程序中安排中断指令，所以它不会真正随机产生，而完全受程序控制。

2. CPU 遇到特殊情况引起的中断

(1) 除法错中断

在执行除法指令时，如果 CPU 发现除数为 0 或者商超过了规定的范围，那么就产生一个除法错中断，中断类型号规定为 0。

例如，在执行下面的程序片段时，会产生一个 0 号类型的中断：

```
MOV    AX,1234
MOV    CL,3
DIV    CL              ;商超过 255(AL 容纳不下)
```

为了避免产生 0 号类型的中断，可改写上述程序片段如下：

```
MOV    AX,1234
MOV    CL,3
XOR    DX,DX
XOR    CH,CH
DIV    CX
```

(2) 溢出中断

8086/8088 提供一条专门检测运算溢出的指令，该指令的格式如下：

```
INTO
```

在溢出标志 OF 置 1 时，如果执行该指令，则产生溢出中断。溢出中断的类型号规定为 4。如果溢出标志 OF 为 0，则执行该指令后并不产生溢出中断。

3. 用于程序调试的中断

(1) 单步中断

如单步标志 TF 为 1，则在每条指令执行后产生一个单步中断，中断类型号规定为 1。产生单步中断后，CPU 就执行单步中断处理程序。由于 CPU 在响应中断时，已把 TF 置

为 0，所以，不会以单步方式执行单步中断处理程序。通常，由调试工具（如 DEBUG 等）把 TF 置 1，在执行完一条被调试程序的指令后，就转入单步中断处理程序，一般情况下，单步中断处理程序报告各寄存器的当前内容，程序员可据此调试程序。

（2）断点中断

8086/8088 提供一条特殊的中断指令“INT 3”，调试工具（如 DEBUG 等）可用它替换断点处的代码，当 CPU 执行这条中断指令后，就产生类型号为 3 的中断。这种中断称为断点中断。通常情况下，断点中断处理程序恢复被替换的代码，并报告各寄存器的当前内容，程序员可据此调试程序。所以说中断指令“INT 3”特殊是因为它只有一个字节长，其他的中断指令长 2 字节。

5.3.6 中断优先级和中断嵌套

1. 中断优先级

系统中有多个中断源，当多个中断源同时向 CPU 请求中断时，CPU 按系统设计时规定的优先级响应中断请求。在 IBM PC 系列及其兼容机系统中，如图 5.6 所示，规定的优先级如下：

优先级最高　　内部中断（除法错，INTO，INT）
　　|　　　　非屏蔽中断（NMI）
　　↓　　　　可屏蔽中断（INTR）
　最低　　　　单步中断

如图 5.8 所示，外设的中断请求都通过中断控制器 8259A 传给 CPU 的 INTR 引线。在对 8259A 初始化时规定了 8 个优先级，在正常的优先级方式下，优先级次序如下：

IRQ0，IRQ1，IRQ2，IRQ3，IRQ4，IRQ5，IRQ6，IRQ7

但在必要的情况下，通过设置中断控制器 8259A 中的中断命令寄存器的有关位可改变上述优先级次序。中断命令寄存器的 I/O 端口地址是 20H，其各位的定义如图 5.10 所示。其中的 L2～L0 三位指定 IRQ0～IRQ7 中具有最低优先级的中断请求，R 位和 SL 位控制 IRQ0～IRQ7 的中断优先级的次序。当 R 位和 SL 位全为 0 时，表示采用正常优先级方式，即上述优先级次序。在一般情况下，总采用这种正常优先级方式。在以后的章节中，如无特别说明，外设中断的优先级次序是上述正常优先级次序。

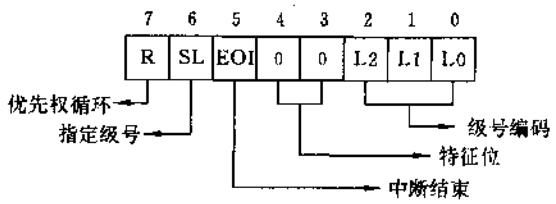


图 5.10　中断命令寄存器

中断命令寄存器中 EOI 位是中断结束位，当把它置为 1 时，表示当前中断处理结束。在对中断控制器 8259A 初始化时规定，在 CPU 响应某个外设的中断请求后，中断控制器

8259A 不再传出中断级相同或较低的外设中断请求，直到 8259A 接收到中断结束命令为止。例如：CPU 在响应来自 IRQ1 的 9 号键盘中断后，8259A 就不再传出来自 IRQ1～IRQ7 的外设中断请求，直到通知 8259A 键盘中断已结束为止。所以，在外设中断处理程序结束时，要通知 8259A 中断已结束，以便使 8259A 传出中断级相同或较低的外设中断请求，从而使 CPU 响应它们。下面程序片段通知 8259A 当前中断结束：

```
MOV    AL,20H
OUT    20H,AL
```

注意，通知中断控制器 8259A 当前中断结束，并非中断返回。只有在执行了中断返回指令后，才返回被中断程序。

在对 8259A 初始化时作上述规定的理由是，使 CPU 在响应外设中断请求后，只要开中断，那么就可响应优先级高的外设中断请求，而不会响应优先级相同或低的外设中断请求。

2. 中断嵌套

CPU 在执行中断处理程序时，又发生中断，这种情况称为中断嵌套。

在中断处理过程中，发生内部中断，引起中断嵌套是经常的事。例如：CPU 在执行中断处理程序时，遇到软中断指令，就会引起中断嵌套。

在中断处理过程中，发生非屏蔽中断，也会引起中断嵌套。

由于 CPU 在响应中断的过程中，已自动关中断，所以，CPU 也就不会再自动响应可屏蔽中断。如果需要在中断处理过程的某些时候响应可屏蔽中断，那么可在中断处理程序中安排开中断指令，CPU 在执行开中断指令后，就处于开中断状态，也就可以响应可屏蔽中断了，直到再关中断。所以，如果在中断处理程序中使用了开中断指令，也就可能会发生可屏蔽中断引起的中断嵌套。

8086/8088 没有限制中断嵌套的深度（层次），但客观上受到堆栈容量的限制。

5.3.7 中断处理程序的设计

CPU 在响应中断后，自动根据中断类型，取中断向量，并转入中断处理程序，所以，具体的处理工作由中断处理程序完成。不同的中断处理，由不同的中断处理程序完成。对应外设中断的外设中断处理程序和对应指令中断的软中断处理程序有些区别，下面对它们的设计分别作些原则性的介绍。

1. 外设中断处理程序

在开中断的情况下，外设中断的发生是随机的，在设计外设中断处理程序时必须充分注意到这一点。外设中断处理程序的主要步骤如下：

（1）必须保护现场。这里的现场可理解为中断发生时 CPU 各内部寄存器的内容。CPU 在响应中断时，已把各标志和返回地址压入堆栈，所以要保护的现场主要是指通用寄存器的内容和除代码段寄存器外的其他三个段寄存器的内容。因为中断的发生是随机的，所以凡是中断处理程序中要重新赋值的各寄存器的原有内容必须先预保护。保护的一般方法是把它们压入堆栈。

(2) 尽快完成中断处理。外设中断处理必须尽快完成,所以外设中断处理必须追求速度上的高效率。因为在进行外设中断处理时,往往不再响应其它外设的中断请求,因此必须快,以免影响对其他外设的中断请求。

(3) 恢复现场。在中断处理完成后,依次恢复被保护寄存器的原有内容。

(4) 通知中断控制器中断已结束。如果应用需要,也可提早通知中断控制器中断结束,这样做必须考虑到外设中断的嵌套。

(5) 利用 IRET 指令实现中断返回。

此外,应及时开中断。除非必要,中断处理程序应尽早开中断,以便 CPU 响应具有更高优先级的中断请求。

2. 软中断处理程序

由中断指令引起的软件中断尽管是不可屏蔽的,但它不会随机发生,只有在 CPU 执行了中断指令后,才会发生。所以,中断指令类似于子程序调用指令,相应的软中断处理程序在很大程度上类似于子程序,但并不等同于子程序。软中断处理程序的主要步骤如下:

(1) 考虑切换堆栈。由于软中断处理程序往往在开中断状态下执行,并且可能较复杂(要占用大量的堆栈空间),所以应该考虑切换堆栈。切换堆栈对实现中断嵌套等均较为有利。

(2) 及时开中断。开中断后,CPU 就可响应可屏蔽的外设中断请求,或者说使外设中断请求可及时得到处理。但要注意,如果该软中断程序要被外设中断处理程序“调用”,则是否要开中断或者何时开中断应另外考虑。

(3) 应该保护现场。应该保护中断处理程序要重新赋值的寄存器原有内容,这样在使用软中断指令时,可不必考虑有关寄存器内容的保护问题。

(4) 完成中断处理。但不必过分追求速度上的高效率,除非它是被外设中断处理程序“调用”的。

(5) 恢复现场。依次恢复被保护寄存器的原内容。

(6) 堆栈切换。如果在开始时切换了堆栈,那么也要再重新切换回原堆栈。

(7) 一般利用 IRET 指令实现中断返回。

5.4 基本输入输出系统 BIOS

本节在介绍 BIOS 基本概念的基础上,介绍键盘输入、显示和打印输出三方面的 BIOS 及部分细节。

5.4.1 基本输入输出系统 BIOS 概述

固化在 ROM 中的基本输入输出系统 BIOS(Basic Input/Output System)包含了主要 I/O 设备的处理程序和许多常用例行程序,它们一般以中断处理程序的形式存在。例如:负责显示输出的显示 I/O 程序作为 10H 号中断处理程序存在,负责打印输出的打印 I/O 程序作为 17H 号中断处理程序存在,而负责键盘输入的键盘 I/O 程序作为 16H 号中断处理程序存在。再如,获取内存容量的例行程序就作为 12H 号中断处理程序存在。BIOS

直接建立在硬件基础上。

磁盘操作系统DOS(Disk Operating System)建立在BIOS的基础上,通过BIOS操纵控制硬件。例如,DOS调用BIOS显示I/O程序完成显示输出,调用打印I/O程序完成打印输出,调用键盘I/O程序完成键盘输入。尽管DOS和BIOS都提供某些相同的功能,但它们之间的层次关系是明显的。

应用程序、DOS、BIOS和外设接口之间的关系如图5.11所示。

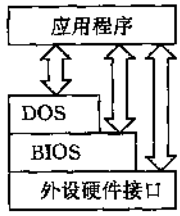


图 5.11 应用程序、DOS、BIOS和硬件接口间的关系

通常应用程序应该调用DOS提供的系统功能完成输入输出或其他操作。这样做不仅实现容易,而且对硬件的依赖性最少。但有时DOS不提供某种服务,例如,取打印机状态信息,那么就不能调用DOS实现了。

应用程序可以通过BIOS进行输入输出或完成其他功能。在下列三种场合可考虑调用BIOS:一是需要利用BIOS提供而DOS不提供的某个功能的场合;二是不能利用DOS功能调用的场合;三是出于某种原因需要绕过DOS的场合。由于BIOS提供的设备处理程序和常用例行程序都以中断处理程序的形式存在,所以应用程序调用BIOS较为方便。但BIOS毕竟比DOS更靠近硬件。

应用程序也可以直接操纵外设接口来控制外设,从而获得速度上最高的效率,但这样的应用程序不仅复杂而且与硬件关系十分密切,此外,还需要程序员对硬件性能比较了解熟悉。所以,应用程序一般不直接与硬件发生关系。

值得指出的是,有时应用程序需要扩充或替换ROM BIOS中的某些处理程序或例行程序,那么这些新的BIOS程序原则上不能调用DOS提供的功能。

5.4.2 键盘输入

1. 键盘中断处理程序

当用户按键时,键盘接口会得到一个代表被按键的键盘扫描码,同时产生一个中断请求。从图5.8和图5.9可知,如果键盘中断是允许的(中断屏蔽字中的位1为0),并且CPU处于开中断状态(IF=1),那么CPU通常就会响应中断请求。由于键盘中断的中断类型号安排为9,所以CPU响应键盘中断,就是转入9号中断处理程序。我们把9号中断处理程序称为键盘中断处理程序,它属于外设中断处理程序这一类。

键盘中断处理程序首先从键盘接口取得代表被按键的扫描码,然后根据扫描码判定用户所按的键并作相应的处理,最后通知中断控制器中断结束并实现中断返回。我们把键盘上的键简单地分成五种类型:字符键(字母、数字和符号等),功能键(如F1和PgUp等),控制键(Ctrl、Alt和左右Shift),双态键(如Num Lock和Caps Lock等),特殊请求键(如Print screen等)。键盘中断处理程序对五种键的基本处理方法如下:

如果用户按的是双态键,那么就设置有关标志,在AT以上档次的系统上还要改变LED指示器状态。如果用户按的是控制键,那么就设置有关标志。如果用户按的是功能键,那么就根据键盘扫描码和是否按下某些控制键(如Alt)确定系统扫描码,把系统扫描

码和一个全 0 字节一起存入键盘缓冲区。如果用户按的是字符键，那么就根据键盘扫描码和是否按下某些控制键(如 Ctrl)确定系统扫描码，并且得出对应的 ASCII 码，把系统扫描码和 ASCII 码一起存入键盘缓冲区。如果用户按的是特殊请求键，那么就产生一个相对应的动作，例如用户按 Print screen 键，那么就调用 5H 号中断处理程序打印屏幕。

2. 键盘缓冲区

键盘缓冲区是一个先进先出的环形队列，结构和占用的内存区域如下：

```
BUFF_HEAD    DW   ?                 ;0040:001AH
BUFF_TAIL    DW   ?                 ;0040:001CH
KB_BUFFER    DW   16   DUP  (?)     ;0040:001EH~003DH
```

BUFF_HEAD 和 BUFF_TAIL 是缓冲区的头指针和尾指针，当这两个指针相等时，表示缓冲区为空。由于缓冲区本身长 16 个字，而存放一个键的扫描码和对应的 ASCII 码需要占用一个字，所以键盘缓冲区可实际存放 15 个键的扫描码和 ASCII 码。键盘中断处理程序把所按字符键或功能键的扫描码和对应的 ASCII 码(如为功能键，对应的 ASCII 码理解为 0)依次存入键盘缓冲区。如缓冲区已满，则不再存入，而是发出"嘟"的一声。

顺便说一下，键盘中断处理程序根据控制键和双态键建立的标志在内存单元 0040:0017H 字单元中。

3. 键盘 I/O 程序

尽管系统程序和应用程序可从键盘缓冲区中取得用户所按键的代码，但除非特殊情况，一般不宜直接存取键盘缓冲区，而应调用 BIOS 提供的键盘 I/O 程序。

键盘 I/O 程序以 16H 号中断处理程序的形式存在，它属于软中断处理程序这一类。它的主要功能是进行键盘输入。一般情况下，系统程序和应用程序的键盘输入最后都是调用它完成的。简单的键盘 I/O 程序从键盘缓冲区中取出所按键的 ASCII 码和扫描码返回给调用者。键盘中断程序、键盘缓冲区和键盘 I/O 程序之间的关系如图 5.12 所示。

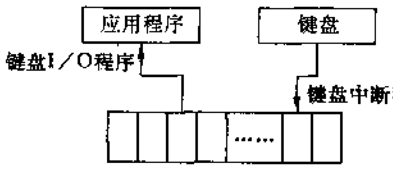


图 5.12　键盘 I/O 程序与键盘缓冲区的关系

4. 键盘 I/O 程序的功能和调用方法

键盘 I/O 程序提供的主要功能列于表 5.3，每一个功能有一个编号。在调用键盘 I/O 程序时，把功能编号置入 AH 寄存器，然后发出中断指令"INT 16H"。调用返回后，从有关寄存器中取得出口参数。除保存出口参数的寄存器外，其他寄存器内容保持不变。

我们把控制键和双态键统称为变换键，调用键盘 I/O 程序的 2 号功能可获得各变换键的状态。变换键状态字节各位的定义如图 5.13 所示，其中高四位记录双态键的变换情况，每按一下双态键，则对应的位值取反；低四位反映控制键是否正被按下，按着某个控制键时，对应的位为 1。

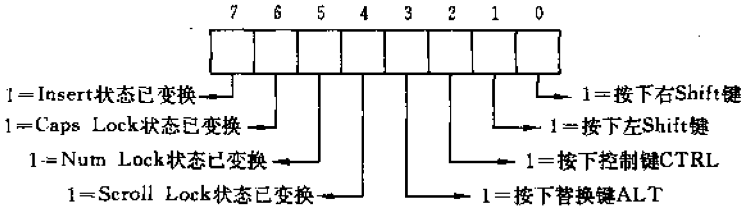


图 5.13 变换键状态字节各位定义

表 5.3 16H 号中断处理程序的基本功能

功　　能	出 口 参 数	说　　明
AH=0 从键盘读一个字符	AL=字符的 ASCII 码 AH=字符的扫描码	如果无字符可读(键盘缓冲区空),则等待;字符也包括功能键,对应 ASCII 码为 0
AH=1 判键盘是否有键可读	ZF=1 表示无键可读 ZF=0 表示有键可读	不等待,立即返回 AL=字符的 ASCII 码 AH=字符的扫描码
AH=2 取变换键当前状态	AL=变换键状态字节	
AH=10H 从键盘读一个字符	同 0 号功能	所不同的是它不删除扩展的键 在早期的系统中没有此功能
AH=11H 判键盘是否有键可读	同 1 号功能	所不同的是它不删除扩展的键 在早期的系统中没有此功能

注:所列基本功能无入口参数。

下面的程序片段从键盘读一个字符:

```
MOV   AH,0
INT   16H
```

如果键盘缓冲区中有字符,那么中断处理就会极快结束,即调用就会极快返回,读到的字符是调用发出之前用户按下的字符。如果键盘缓冲区空,那么要等待用户按键后调用才会返回,读到的字符是调用发出之后按下的字符。如果程序员出于某种理由,要从键盘取得在调用发出之后用户按下的字符,那么就要先清除键盘缓冲区。下面的程序片段先清除键盘缓冲区,然后再从键盘读一个字符:

```
        ......
AGAIN:  MOV   AH,1
        INT   16H          ;判缓冲区空?
        JZ    NEXT         ;空,转
        MOV   AH,0
        INT   16H          ;从键盘缓冲区取走一字符
```

```
        JMP     AGAIN       ;继续
NEXT:   MOV     AH,0
        INT     16H         ;等待键盘输入
        ……
```

当然,程序员也可通过直接修改键盘缓冲区头指针的方法清除键盘缓冲区,但我们不鼓励这样做。

例1: 写一个程序完成如下功能:读键盘,并把所按键盘显示出来,在检测到按下SHIFT键后,就结束运行。

调用键盘I/O程序的2号功能取得变换键状态字节,进而判断是否按下了SHIFT键。在调用0号功能读键盘之前,先调用1号功能判键盘是否有键可读,否则会导致不能及时检测到用户按下的SHIFT。源程序如下:

```
;程序名:T5-1.ASM
;功  能:(略)
;常量定义
L_SHIFT = 00000010B
R_SHIFT = 00000001B
;代码段
CSEG    SEGMENT
        ASSUME  CS:CSEG
START:  MOV     AH,2                    ;取变换键状态字节
        INT     16H
        TEST    AL,L_SHIFT+R_SHIFT      ;判是否按下SHIFT键
        JNZ     OVER                    ;按下,转
        MOV     AH,1
        INT     16H                     ;是否有键可读
        JZ      START                   ;没有,转
        MOV     AH,0                    ;读键
        INT     16H
        MOV     DL,AL                   ;显示所读键
        MOV     AH,6
        INT     21H
        JMP     START                   ;继续
OVER:   MOV     AH,4CH
        INT     21H                     ;结束
CSEG    ENDS
        END     START
```

5.4.3 显示输出

显示器通过显示适配卡与系统相连,显示适配卡是显示输出的接口。早先的显示适配

卡是 CGA 和 EGA 等,目前常用的显示适配卡是 VGA 和 TVGA 等。它们都支持两类显示方式:文本显示方式和图形显示方式,每一类显示方式还含有多种显示模式。

1. 文本显示方式

所谓文本显示方式是指以字符为单位显示的方式。字符通常是指字母、数字、普通符号(如运算符号)和一些特殊符号(如菱形块和矩形块)。

通常 0~3 号显示模式为文本显示方式,它们之间的区别是每屏可显示的字符数和可使用的颜色数目不同。用得最普遍的是 3 号显示模式,我们就以 3 号显示模式为代表作介绍。

在 3 号文本显示模式下,显示器的屏幕被划分成 80 列 25 行,所以每一屏最多可显示 2000(80×25)个字符。我们用行号和位号组成的坐标来定位屏幕上的每个可显示位置,左上角的坐标规定为(0,0),向右增加列号,向下增加行号,这样右下角的坐标便是(79,24)。如图 5.15 所示。

2. 显示属性

屏幕上显示的字符取决于字符代码及字符属性。这里的属性是指显示属性,它规定字符显示 时的特性。在单色显示时,属性定义了闪烁、反相和高亮度等显示特性。在彩色显示时,属性还定义了前景色和背景色。图 5.14 给出了彩色显示时属性字节各位的定义。

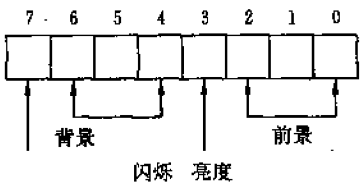


图 5.14 属性字节各位的定义

在属性字节中,RGB 分别表示红、绿、蓝,I 表示亮度,BL 表示闪烁。位 0~位 3 组合 16 种前景颜色,位 4~位 6 组合 8 种背景颜色。亮度和闪烁只能用于前景。当 I 位为 1 时,表示高亮度,当 I 位为 0 时,表示普通亮度。当 BL 位为 1 时,表示闪烁,当 BL 位为 0 时,表示不闪烁。表 5.4 给出了彩色文本模式下 IRGB 组合成的通常颜色。前景颜色和背景颜色一起确定字符的显示效果,表 5.5 列出了几种典型的属性值。当前景和背景相同时,字符就看不出了。

表 5.4 彩色文本模式下的颜色组合

色号	I	R	G	B	颜色	色号	I	R	G	B	颜色
0	0	0	0	0	黑	8	1	0	0	0	亮灰
1	0	0	0	1	蓝	9	1	0	0	1	亮蓝
2	0	0	1	0	绿	10	1	0	1	0	亮绿
3	0	0	1	1	青(深蓝)	11	1	0	1	1	亮青
4	0	1	0	0	红	12	1	1	0	0	亮红
5	0	1	0	1	品红	13	1	1	0	1	亮品红
6	0	1	1	0	棕	14	1	1	1	0	黄
7	0	1	1	1	白	15	1	1	1	1	亮白

表 5.5　属性字节的典型值

效　果	BL	R	G	B	I	R	G	B	16 进制值
黑底蓝字	0	0	0	0	0	0	0	1	01
黑底红字	0	0	0	0	0	1	0	0	04
黑底白字	0	0	0	0	0	1	1	1	07
黑底黄字	0	0	0	0	1	1	1	0	0E
黑底亮白字	0	0	0	0	1	1	1	1	0F
白底黑字	0	1	1	1	0	0	0	0	70
白底红字	0	1	1	1	0	1	0	0	74
黑底灰白闪烁字	1	0	0	0	0	1	1	1	87
白底红闪烁字	1	1	1	1	0	1	0	0	F4

3. 显示存储区

显示适配卡带有显示存储器，用于存放屏幕上显示文本的代码及属性或图形信息。显示存储器作为系统存储器的一部分，可用访问普通内存的方法访问显示存储器。通常，为显示存储器安排的存储地址空间的段值是 B800H 或 B000H，对应的内存区域就称为显示存储区。我们假设段值是 B800H。

在 3 号文本显示模式下，屏幕上的每一个显示位置依次对应显示存储区中的两个字节单元，这种对应关系如图 5.15 所示。

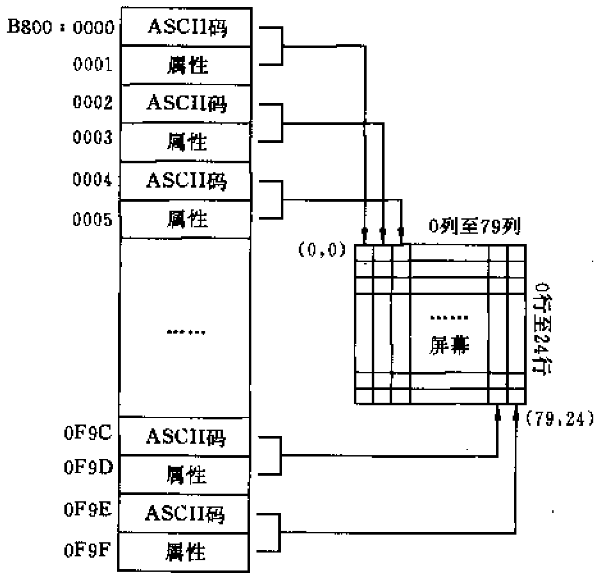


图 5.15　显示存储区与显示位置的对应关系

为了在屏幕上某个位置显示字符，只需把要显示字符的代码及其属性填到显示存储区中的对应存储单元即可。下面的程序片段实现在屏幕的左上角以黑底白字显示字符“A”：

```
......
MOV     AX,B800H
MOV     DS,AX
MOV     BX,0
MOV     AL,'A'
MOV     AH,07H
MOV     [BX],AX
......
```

为了了解屏幕上某个显示位置所显示的字符是什么，或显示的颜色是什么，那么只要从显示存储区中的对应存储单元中取出字符的代码和属性即可。下面的程序片段取得屏幕右下角所显示字符的代码及属性：

```
......
MOV     AX,B800H
MOV     DS,AX
MOV     BX,(80*24+79)*2
MOV     AX,[BX]
......
```

这种直接存取显示存储器进行显示的方法称为直接写屏。

例 2： 采用直接写屏法在屏幕上用多种属性显示字符串"HELLO"。

先用一种属性在屏幕上显示指定信息，然后在用户按一键后再换一种属性显示，如按 ESC 键，则结束。其中的显示子程序 ECHO 采用了直接写屏方法实现显示。源程序如下：

```
;源程序名：T5-2.ASM
;功    能：(略)
;常量定义
ROW       = 5                              ;显示信息的行号
COLUM     =10                              ;列号
ESCKEY    =1BH                             ;ESC 键的 ASCII 码值
;数据段
DSEG     SEGMENT
MESS     DB 'HELLO'                        ;显示信息
MESS_LEN = $ - OFFSET MESS                 ;显示信息长度
COLORB     DB 07H,17H,0FH,70H,74H          ;颜色
COLORE     LABEL BYTE
DSEG       ENDS
;代码段
CSEG     SEGMENT
         ASSUME    CS:CSEG,    DS:DSEG
START:   MOV    AX,DSEG
         MOV    DS,AX                           ;设置数据段段值
```

```
        MOV     DI,OFFSET COLORB -1     ;颜色指针初值
NEXTC:  INC     DI                      ;调整颜色指针
        CMP     DI,OFFSET COLORE        ;是否超过指定的最后一种颜色
        JNZ     NEXTE                   ;否
        MOV     DI,OFFSET COLORB        ;是,重新指定第一种颜色
NEXTE:  MOV     BL,[DI]                 ;取颜色
        MOV     SI,OFFSET MESS          ;取显示信息指针
        MOV     CX,MESS_LEN             ;取显示信息长度
        MOV     DH,ROW                  ;置显示开始行号
        MOV     DL,COLUM                ;置显示开始列号
        CALL    ECHO                    ;显示
        MOV     AH,0
        INT     16H
        CMP     AL,ESCKEY               ;是否为 ESC 键
        JNZ     NEXTC                   ;不是,继续
        MOV     AX,4C00H                ;结束
        INT     21H
;
;子程序名:ECHO
;功    能:直接写屏显示字符串
;入口参数:DS:SI=字符串首地址
;          CX=字符串长度;BL=属性
;          DH=显示开始行号;DL=显示开始列号
;出口参数:无
ECHO    PROC
        MOV     AX,0B800H
        MOV     ES,AX                   ;设置显示段段值
        MOV     AL,80                   ;计算显示开始位置偏移
        MUL     DH                      ;偏移=(行号*80+列号)*2
        XOR     DH,DH
        ADD     AX,DX
        ADD     AX,AX
        XCHG    AX,BX
        MOV     AH,AL                   ;属性值保存到 AH 寄存器
        JCXZ    ECHO2                   ;显示信息长度是否为 0
ECHO1:  MOV     AL,[SI]                 ;取一要显示字符代码
        INC     SI                      ;调整指针
        MOV     ES:[BX],AX              ;送显示存储区,即显示
        INC     BX                      ;准备显示下一个字符
        INC     BX
        LOOP    ECHO1                   ;循环显示
ECHO2:  RET                             ;返回
```

```
ECHO    ENDP
CSEG    ENDS
        END  START
```

4. 显示 I/O 程序的功能和调用方法

利用直接写屏方法，程序可实现快速显示。但为了实现直接写屏，必须了解显示存储器占用存储空间的具体细节和显示存储区与屏幕显示位置的对应关系，并且最终的程序也与显示适配卡相关。所以，除非追求显示速度，一般不采用直接写屏方法，而是调用 BIOS 提供的显示 I/O 程序。

BIOS 提供的显示 I/O 程序作为 10H 号中断处理程序存在。

显示 I/O 程序的主要功能列于表 5.6，每一个功能有一个编号。在调用显示 I/O 程序的某个功能时，应根据要求设置好入口参数，把功能编号置入 AH 寄存器，然后发出中断指令“INT 10H”。调用返回后，从有关寄存器中取得出口参数。除保存出口参数的寄存器外，其他寄存器内容保持不变。

现就显示页号作些说明。为了支持屏幕上显示 2000 个字符，需要的显示存储器容量约为 4KB。如果显示存储器的容量为 32KB，那么显示存储器可存放 8 屏显示内容。为此，把显示存储器再分成若干段，称为显示页。调用显示 I/O 程序的 5 号功能，可选择当前显示页。通常，总是使用第 0 页。

表 5.6　10H 号中断处理程序的基本功能

功　能	入口参数	出口参数	说　明
AH=0 设置显示模式	AL=显示模式代号		可设置的显示模式与显示适配卡有关
AH=1 置光标类型	CH 低 4 位=光标开始线 CL 低 4 位=光标结束线		文本模式下光标大小；当第 4 位为 1 时光标不显示
AH=2 置光标位置	BH=显示页号 DH=行号 DL=列号		左上角坐标是(0,0)
AH=3 读光标位置	BH=显示页号	CH=光标开始行 CL=光标结束行 DH=行号 DL=列号	CH 和 CL 含光标类型 左上角坐标是(0,0)
AH=5 选择当前显示页	AL=新页号		
AH=6 向上滚屏	AL=上滚行数 BH=填空白行的属性 CH=窗口左上角行号 CL=窗口左上角列号 DH=窗口右下角行号 DL=窗口右下角列号		①当滚动行数为 0 时表示清除整个窗口 ②仅影响当前显示页
AH=7 向下滚屏	AL=下滚行数 其他参数同 AH=6		同 AH=6

续表

功　能	入口参数	出口参数	说　明
AH＝8 读取光标位置处的字符和属性	BH＝显示页号	AH＝属性 AL＝字符代码	
AH＝9 将字符和属性写到光标位置处	BH＝显示页号 AL＝字符代码 BL＝属性 CX＝字符重复次数		光标不移动
AH＝0AH 将字符写到光标位置处	BH＝显示页号 AL＝字符代码 CX＝字符重复次数		①光标不移动 ②不带属性
AH＝0EH TTY 方式显示	BH＝显示页号 AL＝字符代码		光标处显示字符并后移光标；解释回车、换行、退格和响铃等控制符
AH＝0FH 取当前显示模式		(AL)＝显示模式号 (AH)＝最大列数 (BH)＝当前页号	
AH＝13H 写字符串(在 AT 及其以上系统上才有此功能)	AL＝写模式 BH＝显示页号 BL＝属性 CX＝串中的字符数 DH＝写串的起始行号 DL＝写串的起始列号 ES：BP＝要写串首地址		①解释回车等控制符 ②写模式仅低 2 位： 位 0：0 表示移动光标，1 表示不动光标 位 1：0 表示串中字符代码和属性交替，1 表示串中只含字符代码

5. 举例

调用显示 I/O 程序的 5 号功能，可选择当前显示页。下面的程序片段选择第 0 页作为当前显示页：

```
……
MOV    AL,0
MOV    AH,5
INT    10H
……
```

如果要知道当前显示页号，则可调用显示 I/O 程序的 0FH 号功能，同时可知道当前显示模式和该模式下的最大显示列数，下面的程序片段调用 0FH 号功能：

```
……
MOV    AH,0FH
INT    10H
……
```

下面的程序片段调用显示 I/O 程序的 9 号功能在当前光标位置处显示指定属性的字

符，但不移动光标：

```
……
MOV    BH,0          ;第 0 页
MOV    BL,47H        ;红底白字
MOV    CX,1          ;1 个
MOV    AL,'A'        ;字符为 A
MOV    AH,9
INT    10H
……
```

在窗口滚屏时，如果滚屏行数为 0，就表示清除整个窗口。如果把整个屏幕作为窗口，那么就可实现清屏。下面的程序假设屏幕为 80 列，它先清除屏幕，然后把光标定到左上角：

```
……
MOV    CH,0          ;置左上角坐标
MOV    CL,0
MOV    DH,24         ;置右下角坐标
MOV    DL,79
MOV    BH,07         ;清除区的填充属性(黑底白字)
MOV    AL,0          ;清整个窗口
MOV    AH,6
INT    10H           ;实现清屏
MOV    BH,0          ;设第 0 页
MOV    DH,0          ;置光标定位坐标
MOV    DL,0
MOV    AH,2
INT    10H           ;定位光标
……
```

例 3： 写一个程序完成如下功能：在屏幕中间部位开出一个窗口，随后接收用户按键，并把按键字符显示在窗口的最底行，当窗口底行显示满时，窗口内容就向上滚动一行；用户按 Ctrl+C 键时，结束运行。

调用显示 I/O 程序可方便地实现该程序。源程序如下：

```
;程序名：T5-3.ASM
;功  能：(略)
;常量定义
WINWIDTH    =   40                    ;窗口宽度
WINTOP      =   8                     ;窗口左上角行号
WINLEFT     =   20                    ;窗口左上角列号
WINBOTTOM   =   17                    ;窗口右下角行号
WINRIGHT    =   WINLEFT+WINWIDTH-1    ;窗口右下角列号
```

```
COLOR        =   74H                     ;属性值
PAGEN        =   0                       ;显示页号
CTRL_C       =   03H                     ;结束符 ASCII 码
;代码段
CSEG    SEGMENT
    ASSUME    CS:CSEG
START:  MOV     AL,PAGEN                 ;选择显示页
        MOV     AH,5
        INT     10H
        ;
        MOV     CH,WINTOP                ;清规定窗口
        MOV     CL,WINLEFT
        MOV     DH,WINBOTTOM
        MOV     DL,WINRIGHT
        MOV     BH,COLOR
        MOV     AL,0
        MOV     AH,6
        INT     10H
        ;
        MOV     BH,PAGEN                 ;定位光标到窗口左下角
        MOV     DH,WINBOTTOM
        MOV     DL,WINLEFT
        MOV     AH,2
        INT     10H
        ;
NEXT:   MOV     AH,0                     ;接受一个键
        INT     16H
        CMP     AL,CTRL_C                ;判是否是结束键
        JZ      EXIT                     ;是,转
        ;
        MOV     BH,PAGEN                 ;在当前光标位置显示所按键
        MOV     CX,1                     ;但没有移动光标
        MOV     AH,0AH
        INT     10H
        ;
        INC     DL                       ;光标列数加 1,准备向右移动光标
        CMP     DL,WINRIGHT+1            ;判是否越出窗口右边界
        JNZ     SETCUR                   ;不,转
        MOV     CH,WINTOP                ;是,窗口内容上滚一行
        MOV     CL,WINLEFT               ;空出窗口的最底行
        MOV     DH,WINBOTTOM
        MOV     DL,WINRIGHT
```

```
          MOV     BH,COLOR
          MOV     AL,1
          MOV     AH,6
          INT     10H
          MOV     DL,WINLEFT              ;光标要回到最左面
SETCUR:   MOV     BH,PAGEN                ;置光标(光标后移)
          MOV     AH,2
          INT     10H
          JMP     NEXT                    ;继续
EXIT:     MOV     AX,4C00H
          INT     21H                     ;结束
CSEG      ENDS
          END     START
```

例4: 调用显示I/O程序来实现程序T5-2.ASM中的显示子程序ECHO。

方法之一：调用I/O程序中的13H号功能直接显示字符串。源程序代码如下：

```
;子程序名：ECHOA
;功    能：调用显示I/O程序的13H号功能显示字符串
;入口参数：DS：SI=字符串首地址
;          CX=字符串长度;BL=属性
;          DH=显示开始行号;DL=显示开始列号
;出口参数：无
ECHOA   PROC
        PUSH    ES
        PUSH    BP          ;保护有关寄存器内容
        PUSH    DS
        POP     ES
        MOV     BP,SI       ;满足13H号功能入口参数要求
        MOV     BH,0        ;指定显示页
        MOV     AL,0        ;采用0号显示方式(不移动光标
        MOV     AH,13H      ;字符串中不含属性)
        INT     10H
        POP     BP
        POP     ES          ;恢复有关寄存器内容
        RET
ECHOA   ENDP
```

该子程序与ECHO稍有不同：第一是13H号功能解释回车和换行等控制码，所以如果要显示的字符串中含有这样的控制码，那么显示效果就会有差异；第二是当显示超出最后一行的最后一列时，13H功能要引起滚屏。

方法之二：先调用显示I/O程序的2号功能把光标定到指定位置，然后利用显示I/O程序的TTY显示功能逐个显示字符串中的字符。但由于TTY方式显示不含属性，所以

先调用 9 号功能把指定属性写到显示字符串的位置处。源程序代码如下：

```
;子程序名:ECHOB
;其他说明信息略
ECHO    PROC
        JCXZ    ECHO2      ;如果字符串长度为0,则结束
        MOV     BH,0
        MOV     AH,2       ;设置光标位置
        INT     10H
        MOV     AL,20H     ;用指定属性写一串空格
        MOV     AH,9
        INT     10H
        MOV     AH,0EH
ECHO1:  MOV     AL,[SI]
        INC     SI
        INT     10H        ;逐个显示字符
        LOOP    ECHO1
ECHO2:  RET
ECHO    ENDP
```

该子程序于 ECHO 也稍有不同：第一 TTY 方式显示也解释控制符，所以如果要显示的字符串中含有控制符，则显示效果就不一样；第二是当 TTY 方式在屏幕的右下角显示一个字符后要上滚屏幕；第三是字符串显示完后，光标定位在字符串之后。

方法之三：先读取当前光标位置且保存；定位光标到指定位置；调用显示 I/O 程序的 9 号功能逐个显示字符，每显示一个字符后，把光标向后移一个位置；最后把光标回到原位。注意，在把光标向后移一个位置时，要判别是否超越屏幕右边界和是否已到达屏幕的右下角。请读者作为练习完成子程序 ECHOC。

5.4.4 打印输出

1. 打印 I/O 程序的功能和调用方法

BIOS 提供的打印 I/O 程序作为 17H 号中断处理程序存在。

打印 I/O 程序的主要功能列于表 5.7，每一个功能有一个编号。系统可连接多台打印机，用打印机号选择打印机，打印机号为 0、1 和 2。调用返回的出口参数只是打印机状态字节。打印机状态字节各位的定义类似与图 5.3 给出的打印接口中状态寄存器的定义，有两点区别：(1)第 0 位定义成超时标志位，当打印机在规定时间内仍处于忙，即不能接受打印数据时，就置超时标志。(2)第 6 位(应答位)和第 3 位(错误位)都规定状态有效为 1(与状态寄存器的定义相反)。

在调用打印 I/O 程序的某个功能时，应根据要求设置好入口参数，把功能编号置入 AH 寄存器，然后发出中断指令“INT 17H”。除保存出口参数的寄存器外，其他寄存器内容保持不变。通常系统只连接一台打印机，打印机号为 0。

表 5.7　17H 号中断处理程序的基本功能

功　能	入口参数	出口参数	说　明
AH＝0 打印一个字符	AL＝字符代码 DX＝打印机号	AH＝状态字节	字符包括打印命令控制符
AH＝1 初始化打印机	DX＝打印机号	AH＝状态字节	初始化时清打印机内的缓冲区
AH＝2 取打印机状态	DX＝打印机号	AH＝状态字节	

2. 举例

例 5： 写一个在 0 号打印机上打印屏幕内容的程序。

程序流程如图 5.16 所示，现结合流程作些说明。为了把屏幕上的内容打印出来，先要获得屏幕上的显示内容，这可利用显示 I/O 程序提供的 8 号功能实现。以行为单位从左到右打印屏幕上的内容，从顶行开始到底行结束，这样就形成一个二重循环。在把一行屏幕内容输出到打印机后，追加输出一个回车符和一个换行符，使打印机实施打印动作。在输出一个字符到打印机后，还判打印机是否正常工作，判打印机是否正常工作的依据是测打印机状态字节中的超时标志、IO 错误标志和纸尽标志。由于要移动光标，所以在移动前先保存光标位置，最后恢复原光标位置。

```
;程序名：T5-4.ASM
;功　能：打印屏幕内容
;常量定义
TIME_OUT    =    00000001B
IO_ERROR    =    00001000B
OUT_OF_P    =    00100000B
FLAG        =    TIME_OUT + IO_ERROR
                 + OUT_OF_P
;代码段
CSEG    SEGMENT
        MOV     AH,0FH
        INT     10H          ;取当前显示页
                              号和最大列数
        MOV     CL,AH        ;最大列数保存
                              到 CL
        MOV     CH,25        ;行数送 CH
        PUSH    CX
        MOV     AH,3         ;取当前光标位置
        INT     10H
```

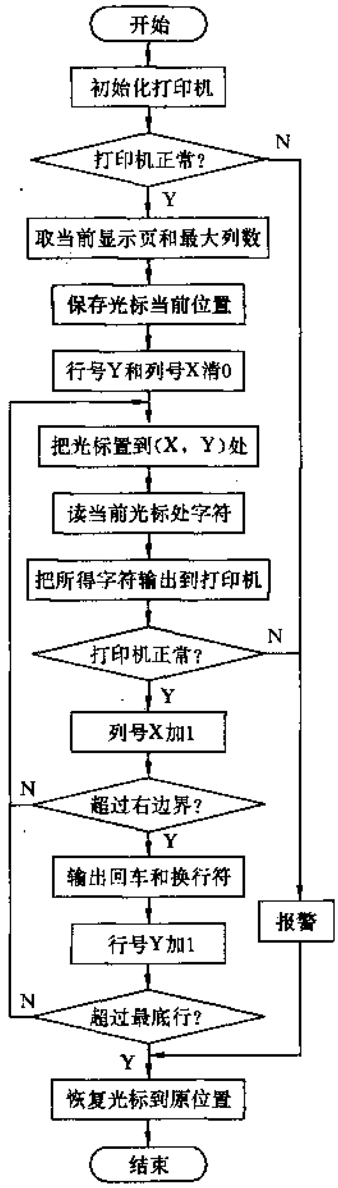


图 5.16　打印屏幕程序的流程

```
        POP     CX
        PUSH    DX              ;保存当前光标
        ;
        XOR     DX,DX           ;准备从左上角开始
PRI1:   MOV     AH,2            ;置光标
        INT     10H
        MOV     AH,8            ;取当前光标处字符
        INT     10H
        OR      AL,AL           ;字符有效?
        JNZ     PRI2            ;是,转
        MOV     AL,"            ;否,作为空格处理
PRI2:   PUSH    DX
        XOR     DX,DX
        XOR     AH,AH           ;所取字符送打印机
        INT     17H
        POP     DX
        TEST    AH,FLAG         ;打印机 OK?
        JNZ     ERR1            ;否,转
        INC     DL              ;是,光标列加 1
        CMP     CL,DL           ;到右边界?
        JNZ     PRI1            ;否,继续下一列
        PUSH    DX
        XOR     DX,DX           ;是,发出回车和换行控制符
        MOV     AX,0DH
        INT     17H
        MOV     AX,0AH
        INT     17H
        POP     DX
        XOR     DL,DL           ;光标准备到下一行左边
        INC     DH              ;为此,行号加 1
        CMP     CH,DH           ;最后一行完?
        JNZ     PRI1            ;否,继续下一行
        POP     DX              ;恢复保存的原光标位置
        MOV     AH,2            ;把光标置回原处
        INT     10H
        JMP     SHORT EXIT      ;转结束
        ;
ERR1:   POP     DX              ;恢复保存的原光标位置
        MOV     AH,2
        INT     10H             ;把光标置回原处
ERR2:   MOV     AL,7
        MOV     AH,0EH
```

```
        INT     10H         ;打印机故障时,发出三声"嘟"
        INT     10H
        INT     10H
        ;
EXIT:   MOV     AH,4CH      ;结束
        INT     21H
CSEG    ENDS
        END     START
```

BIOS 提供的屏幕打印程序的实现方法和上述程序是相同的,只是作为 5H 号中断处理程序而存在。

5.5 软中断处理程序举例

本节以打印 I/O 程序和时钟显示程序的实现为例介绍软中断处理程序的设计。

5.5.1 打印 I/O 程序

我们在 5.4.4 节介绍了由 BIOS 提供的打印 I/O 程序的功能和调用方法,现在介绍打印 I/O 程序的源程序,把它作为 BIOS 中断处理程序设计的例子。

每个打印接口有三个寄存器:数据寄存器、状态寄存器和控制寄存器,它们的端口地址都是连续的,在系统加电初始化期间,数据寄存器端口地址被依次保存到 BIOS 数据区,从 40H 段的偏移 8 处开始。

尽管通过中断控制器 8259A 可实现中断方式打印输出,但 BIOS 提供的打印 I/O 程序却采用在 5.2.3 节中介绍的查询方式实现打印输出。查询时使用的超时参数也在 BIOS 数据区中,从 40H 段的偏移 78H 开始。

BIOS 提供的打印 I/O 程序作为 17H 号中断处理程序存在。实现流程如图 5.17 所示。作为 BIOS 软中断处理程序,没有自己的堆栈,而是使用主程序的堆栈。首先开中断,以便使 CPU 及时响应外设中断请求。然后保护要使用到的各寄存器内容。再从 BIOS 数据区中取出指定打印机的数据寄存器端口地址,如端口地址值为 0,表示没有安装指定的打印接口。接下来根据功能号,分情况处理。源程序如下所示:

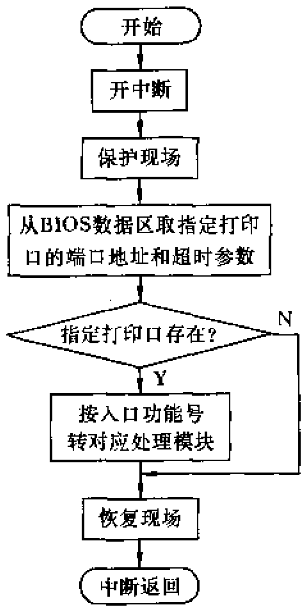


图 5.17 打印 I/O 程序的流程

```
CODE    SEGMENT     PUBLIC
        ASSUME      CS:CODE, DS:DATA
;常量说明
PRINT_TIM_OUT   =   78H         ;超时参数存放单元开始偏移
```

```
PRINTER_BASE       =   8           ;端口地址存放单元开始地址偏移
BIOS_DATA_SEG      =   40H         ;BIOS 数据段的段值
;代码部分
PRINTER_IO         PROC     FAR
         STI                                    ;开中断
         PUSH      DS
         PUSH      DX                           ;保护现场
         PUSH      SI
         PUSH      CX
         PUSH      BX
         MOV       BX,BIOS_DATA_SEG
         MOV       DS,BX                        ;置 BIOS 数据段段值
         MOV       SI,DX
         MOV       BL,PRINT_TIM_OUT[SI]         ;取超时测试基本单位数
         SHL       SI,1
         MOV       DX,PRINTER_BASE[SI]          ;取数据寄存器端口地址
         OR        DX,DX                        ;判系统是否有对应接口
         JZ        B1                           ;无,转结束
         OR        AH,AH                        ;0 号功能?
         JZ        B2                           ;是,转
         DEC       AH                           ;1 号功能?
         JZ        B8                           ;是,转
         DEC       AH                           ;2 号功能?
         JZ        B5                           ;是,转
B1:      POP       BX
         POP       CX                           ;恢复现场
         POP       SI
         POP       DX
         POP       DS
         IRET                                   ;中断返回
;0 号功能处理
B2:      PUSH      AX
         OUT       DX,AL                        ;输出打印数据
         INC       DX                           ;DX=状态寄存器端口地址
         PUSH      BX
         SUB       BH,BH
         RCL       BX,1                         ;确定超时参数单位数
         RCL       BX,1
B3:      SUB       CX,CX                        ;每一单位测 65536 次
B3_1:    IN        AL,DX                        ;读状态寄存器
         MOV       AH,AL
         TEST      AL,80H                       ;是否"忙碌"?
```

```
        JNZ     B4                      ;否,转
        LOOP    B3_1                    ;是,继续测
        DEC     BX                      ;单位数完?
        JNZ     B3                      ;否,转
        POP     BX                      ;是
        OR      AH,1                    ;置超时标志位
        AND     AH,0F9H                 ;去掉无定义位
        JMP     SHORT B7                ;转,中断返回
B4:     POP     BX
        MOV     AL,0DH                  ;选通,即真正输出到打印机
        INC     DX
        OUT     DX,AL
        MOV     AL,0CH
        JMP     $+2
        OUT     DX,AL
        POP     AX
;2 号功能处理
B5:     PUSH    AX
B6:     MOV     DX,PRINTER_BASE[SI]
        INC     DX                      ;DX=状态寄存器端口地址
        IN      AL,DX                   ;读状态寄存器
        MOV     AH,AL
        AND     AH,0F8H                 ;去掉无定义位和清超时标志位
B7:     POP     DX
        MOV     AL,DL
        XOR     AH,48H                  ;第 6 和第 3 位取反(符合出口约定)
        JMP     B1                      ;转,中断返回
;1 号功能处理
B8:     PUSH    AX
        INC     DX
        INC     DX                      ;DX=控制寄存器端口地址
        MOV     AL,8
        OUT     DX,AL                   ;输出初始化命令
        MOV     AX,10000
B9:     DEC     AX                      ;等待一段时间
        JNZ     B9
        MOV     AL,0CH                  ;输出正常控制命令
        OUT     DX,AL
        JMP     B6                      ;转取状态字节和中断返回
PRINTER_IO      ENDP
CODE    ENDS
        END
```

上述 17H 号中断处理程序直接操纵控制打印接口，没有再调用其他程序。

5.5.2 时钟显示程序

在系统加电初始化其间，把系统定时器初始化为每隔约 55 毫秒发出一次中断请求。据图 5.9，CPU 在响应定时中断请求后转入 8H 号中断处理程序。BIOS 提供的 8H 号中断处理程序中有一条中断指令“INT 1CH”，所以每秒要调用到约 18.2 次 1CH 号中断处理程序。而 BIOS 的 1CH 号中断处理程序实际上并没有做任何工作，只有一条中断返回指令。这样安排的目的是为应用程序留下一个软接口，应用程序只要提供新的 1CH 号中断处理程序，就可能实现某些周期性的工作。

下面介绍的时钟显示程序就是利用这个软接口，实现时钟显示。

在新的 1CH 号中断处理程序中安排一个计数器，记录调用它的次数，当计数满 18 次后，就在屏幕的右上角显示当前的时间(时分秒)，清计数器。这样约每秒显示一次当前时间。当前时间的获取是调用 1AH 号中断处理程序的 2 号功能完成的，该功能在 CH、CL 和 DH 寄存器中返回时、分和秒的 BCD 码。在把 BCD 码转换为对应十进制数的 ASCII 码后，调用显示 I/O 程序完成显示。

主程序首先保存原 1CH 号中断向量，然后设置新的 1CH 号中断向量，在完成主程序的其他工作后，再恢复原 1CH 号中断向量。在主程序设置新的 1CH 号中断向量后，时钟就开始工作。作为例子，这里的主程序实际上并没有进行实质性的工作。

```
;程序名：T5-5.ASM
;功  能：(略)
;中断处理程序常量定义
COUNT_VAL  =  18                            ;间隔“嘀答”数
DPAGE      =  0                             ;显示页号
ROW        =  0                             ;显示时钟的行号
COLUMN     =  80 - BUFF_LEN                 ;显示时钟的开始列号
COLOR      =  07H                           ;显示时钟的属性值
;代码
CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG
;1CH 号中断处理程序使用的变量
COUNT   DW    COUNT_VAL                     ;“嘀答”计数
HHHH    DB ?,?,':'                          ;时
MMMM    DB ?,?,':'                          ;分
SSSS    DB ?,?                              ;秒
BUFF_LEN = $-OFFSET HHHH                    ;BUFF_LEN 为显示信息长度
CURSOR     DW ?                             ;原光标位置
;1CH 号中断处理程序代码
NEW1CH:
    CMP      CS:COUNT,0                     ;是否已到显示时候?
```

```
        JZ      NEXT                     ;是,转
        DEC     CS:COUNT                 ;否
        IRET                             ;中断返回
NEXT:
        MOV     CS:COUNT,COUNT_VAL       ;重置间隔数初值
        STI                              ;开中断
        PUSH    DS
        PUSH    ES
        PUSH    AX
        PUSH    BX
        PUSH    CX                       ;保护现场
        PUSH    DX
        PUSH    SI
        PUSH    BP
        PUSH    CS
        POP     DS                       ;置数据段寄存器
        PUSH    DS
        POP     ES                       ;置代码段寄存器
        CALL    GET_T                    ;取时间
        MOV     BH,DPAGE
        MOV     AH,3                     ;取原光标位置
        INT     10H
        MOV     CURSOR,DX                ;保存原光标位置
        MOV     BP,OFFSET HHHH
        MOV     BH,DPAGE
        MOV     DH,ROW
        MOV     DL,COLUMN
        MOV     BL,COLOR
        MOV     CX,MESS_LEN
        MOV     AL,0
        MOV     AH,13H                   ;显示时钟
        INT     10H
        MOV     BH,DPAGE                 ;恢复原光标
        MOV     DX,CURSOR
        MOV     AH,2
        INT     10H
        POP     BP
        POP     SI
        POP     DX                       ;恢复现场
        POP     CX
        POP     BX
        POP     AX
```

```
    POP    ES
    POP    DS
    IRET                                 ;中断返回
    ;
;子程序说明信息略
GET_T  PROC
       MOV    AH,2                       ;取时间信息
       INT    1AH
       MOV    AL,CH                      ;把时数转为可显示形式
       CALL   TTASC
       XCHG   AH,AL
       MOV    WORD PTR HHHH,AX           ;保存
       MOV    AL,CL                      ;把分数转为可显示形式
       CALL   TTASC
       XCHG   AH,AL
       MOV    WORD PTR MMMM,AX           ;保存
       MOV    AL,DH                      ;把秒数转为可显示形式
       CALL   TTASC
       XCHG   AH,AL
       MOV    WORD PTR SSSS,AX           ;保存
       RET
GET_T  ENDP
;
;子程序名：TTASC
;功    能：把两位压缩的BCD码转换为对应的ASCII码
;入口参数：AL=压缩BCD码
;出口参数：AH=高位BCD码所对应的ASCII码
;          AL=低位BCD码所对应的ASCII码
TTASC PROC
      MOV   AH,AL
      AND   AL,0FH
      SHR   AH,1
      SHR   AH,1
      SHR   AH,1
      SHR   AH,1
      ADD   AX,3030H
      RET
TTASC ENDP
;==============================================
;初始化部分代码和变量
OLD1CH DD    ?                           ;原中断向量保存单元
START: PUSH  CS
```

```
        POP   DS
        MOV   AX,351CH                      ;取1CH号中断向量
        INT   21H
        MOV   WORD PTR OLD1CH,BX            ;保存
        MOV   WORD PTR OLD1CH+2,ES
        MOV   DX,OFFSET NEW1CH
        MOV   AX,251CH                      ;置新的1CH号中断向量
        INT   21H
;
;……                                         ;其它工作
;……
        MOV   AH,0                          ;假设其它工作是等待按键
        INT   16H
        LDS   DX,OLD1CH                     ;取保存的原1CH号中断向量
        MOV   AX,251CH
        INT   21H                           ;恢复原1CH号中断向量
        MOV   AH,4CH                        ;结束
        INT   21H
CSEG    ENDS
        END   START
```

上述1CH号中断处理程序没有调用DOS系统功能显示时间，而是调用显示I/O程序显示时间。另外，它除了管理计数器外，还使用了若干变量。

5.6 习　题

题5.1 什么是I/O端口地址？8086/8088的I/O端口地址空间有多大？

题5.2 请说明指令“IN AX,DX”和如下程序片段的异同：

```
IN    AL,DX
INC   DX
IN    AL,DX
MOV   AH,AL
```

题5.3 请说明指令“OUT 20H,AL”和如下程序片段的异同：

```
MOV   DX,20H
OUT   DX,AL
```

题5.4 CPU与外设之间交换的信息可分为哪几类？如何区分它们？

题5.5 微机系统常采用哪些方式实现输入输出？

题5.6 简述查询传输方式的优缺点。请画出一般查询方式的实现流程图。

题5.7 简述中断传输方式及其优缺点。

题5.8 什么是中断？什么是中断源？

题5.9 中断向量表的作用是什么？中断向量表有多大？安排在哪里？

题 5.10 请简述中断响应的过程。

题 5.11 中断返回指令 IRET 与如下两组指令有何不同？

(1) RETF 2　　　　(2) RETF

　　　　　　　　　　POPF

题 5.12 如何不使用软中断指令 INT 调用 16H 号中断处理程序。

题 5.13 外部中断与内部中断有何异同？举例说明内部中断和外部中断。

题 5.14 可屏蔽中断与不可屏蔽中断(非屏蔽中断)有何异同？

题 5.15 在什么条件下才会响应键盘中断？

题 5.16 PC 系统如何实现中断优先级和中断嵌套？

题 5.17 设计中断处理程序时应该遵循哪些原则？PC 系统中的中断处理程序通常应该含有哪些步骤？

题 5.18 基本输入/输出系统 BIOS 主要含有哪些内容？

题 5.19 应用程序、操作系统、BIOS 和外设接口之间的相互关系如何？

题 5.20 有哪些方法可在屏幕的左上角显示 AB 两个字符？请比较这些方法。

题 5.21 简述键盘中断处理程序、键盘 I/O 程序和键盘缓冲区三者之间的关系。这样安排有何优点？

题 5.22 软中断处理程序与硬中断处理程序有何异同？

题 5.23 请写一个程序显示打印接口的当前状态。

题 5.24 从 RT/COMS RAM 中可获得系统当前日期和时间。写一个程序显示 RT/CMOS RAM 中的系统当前日期和时间。

题 5.25 请写一个程序打印一份中断向量表。

题 5.26 请写一个能够显示指定向量号的中断向量的程序。

题 5.27 写一个程序采用十六进制数的形式显示所按键的扫描码及对应的 ASCII 码。当连续两次按回车键时终止程序。

题 5.28 写一个清屏程序。

题 5.29 写一个程序采用直接填显示缓冲区的方法在屏幕上循环显示 26 个大写字母。当按任一键后终止程序，通过调用 BIOS 键盘管理模块的 1 号功能判别是否有键按下。

题 5.30 写一个程序在屏幕上循环显示 26 个大写字母，每行显示 10 个，逐行变换显示的颜色。当按下 ALT+F1 键时终止程序。

题 5.31 写一个程序把屏幕上显示的大写字母全部变换为对应的小写字母。

题 5.32 写一个程序统计当前屏幕上显示的字母个数。

题 5.33 写一个程序判别屏幕上是否显示字符串“AB”。在屏幕的最底行显示提示信息，按任意键后终止程序。

题 5.34 通过调整 BIOS 中的键盘中断处理程序，可使到所按的大写字母全部变换为对应的小写字母。写一个测试程序验证上述方法。

题 5.35 通过调整 BIOS 中的键盘 I/O 程序，可使到所按的大写字母全部变换为对应的小写字母。写一个测试程序验证上述方法。

题 5.36 通过调整 BIOS 中的显示 I/O 程序是否可使屏幕上只显示小写字母？为什么？编写程序测试之。

题 5.37 请写一个程序把当前目录下的 TEST. TXT 文件打印输出。

题 5.38 写一个程序在屏幕中央显示系统当前时间，当同时按下左右 SHIFT 键时清屏，结束程序。

题 5.39 请谈谈汇编语言与机器系统的关系。

第6章　简单应用程序的设计

8086/8088指令集含有专门的字符串操作指令，利用它们可有效地进行字符串操作。指令集还有十进制数算术运算调整指令，利用它们可快速地实现十进制数算术运算。本章先介绍这些指令，然后结合若干特殊情况处理程序说明简单应用程序的设计方法。

6.1　字符串处理

字符串是字符的一个序列。对字符串的操作处理包括复制、检索、插入、删除和替换等。为了便于对字符串进行有效的处理，8086/8088提供专门用于处理字符串的指令，我们称之为字符串操作指令，简称为串操作指令。本节先介绍串操作指令及与串操作指令密切相关的重复前缀，然后举例说明如何利用它们进行字符串处理。

6.1.1　字符串操作指令

1. 一般说明

8086/8088共有五种基本的串操作指令。每种基本的串操作指令包括两条指令，一条适用于以字节为单元的字符串，另一条适用于以字为单元的字符串。

在字符串操作指令中，由变址寄存器SI指向源操作数(串)，由变址寄存器DI指向目的操作数(串)。规定源串存放在当前数据段中，目的串存放在当前附加段中，也即在涉及源操作数时，引用数据段寄存器DS，在涉及目的操作数时，引用附加段寄存器ES。换句话说，DS：SI指向源串，ES：DI指向目的串。

串操作指令执行时会自动调整作为指针使用的寄存器SI或DI之值。如串操作的单元是字节，则调整值为1；如串操作的单元是字，则调整值为2。此外，字符串操作的方向(处理字符串中单元的次序)由标志寄存器中的方向标志DF控制。当方向标志DF复位(为0)时，按递增方式调整寄存器SI或DI值；当方向标志DF置位(为1)时，按递减方式调整寄存器SI或DI之值。

2. 字符串装入指令(LOAD String)

字符串装入指令的格式如下：

```
LODSB        ;装入字节(Byte)
LODSW        ;装入字(Word)
```

字符串装入指令只是把字符串中的一个字符装入到累加器中。字节装入指令LODSB把寄存器SI所指向的一个字节数据装入到累加器AL中，然后根据方向标志DF复位或置位使SI之值增1或减1。它类似下面的两条指令：

```
MOV    AL,[SI]
```

```
INC     SI      或      DEC     SI
```

字装入指令 LODSW 把寄存器 SI 所指向的一个字数据装入到累加器 AX 中，然后根据方向标志 DF 复位或置位使 SI 之值增 2 或减 2。类似于如下的两条指令：

```
MOV     AX,[SI]
ADD     SI,2      或      SUB     SI,2
```

字符串装入指令的源操作是存储操作数，所以引用数据段寄存器 DS。

字符串装入指令不影响标志。

下面的子程序使用了 LODSB 指令。此外，该子程序算法也较好，所以它的效率较高。请与 3.6 节中的例 4 作比较。

```
;子程序名：STRLWR
;功    能：把字符串中的大写字母转化为小写(字符串以 0 结尾)
;入口参数：DS：SI＝字符串首地址的段值：偏移
;出口参数：无
STRLWR  PROC
        PUSH    SI
        CLD                         ;清方向标志(以便按增值方式调整指针)
        JMP     SHORT STRLWR2
STRLWR1:SUB     AL,'A'
        CMP     AL,'Z'-'A'
        JA      STRLWR2
        ADD     AL,'a'
        MOV [SI-1],AL               ;注意指针已被调整
STRLWR2:LODSB                       ;取一字符，同时调整指针
        AND     AL,AL
        JNZ     STRLWR1
        POP     SI
        RET
STRLWR  ENDP
```

在汇编语言中，两条字符串装入指令的格式可统一为如下一种格式：

```
LODS    OPRD
```

汇编程序根据操作数的类型决定使用字节装入指令还是字装入指令。也即，如果操作数的类型为字节，则采用 LODSB 指令；如果操作数的类型为字，则采用 LODSW 指令。请注意，操作数 OPRD 不影响指针寄存器 SI 之值，所以在使用上述格式的串装入指令时，仍必须先给 SI 赋合适的值。例如：

```
        ......
MESS  DB        'HELLO',0
TAB   DW        123, 43, 332, 44,-1
        ......
```

```
MOV     SI,OFFSET  MESS
LODS    MESS                ;LODSB
......
MOV     SI,OFFSET  TAB
LODS    TAB                 ;LODSW
......
```

3. 字符串存储指令(STOre String)

字符串存储指令的格式如下:

```
STOSB    ;存储字节
STOSW    ;存储字
```

字符串存储指令只是把累加器的值存到字符串中,即替换字符串中一个字符。

字节存储指令 STOSB 把累加器 AL 的内容送到寄存器 DI 所指向的存储单元中,然后根据方向标志 DF 复位或置位使 DI 之值增 1 或减 1。它类似下面的两条指令:

```
MOV      ES:[DI],AL
INC      DI        或      DEC    DI
```

字装入指令 STOSW 把累加器 AX 的内容送到寄存器 DI 所指向的存储单元中,然后根据方向标志 DF 复位或置位使 DI 之值增 2 或减 2。类似于如下的两条指令:

```
MOV      ES:[DI],AX
ADD      DI,2      或      SUB    DI,2
```

字符串存储指令的源操作是累加器 AL 或 AX,目的操作是存储操作数,所以引用当前附加段寄存器 ES。

字符串存储指令不影响标志。

在汇编语言中,两条字符串存储指令的格式可统一为如下一种格式:

```
STOS     OPRD
```

汇编程序根据操作数 OPRD 的类型决定使用字节存储指令还是字存储指令。操作数 OPRD 不影响指针寄存器 DI 之值。

例如:如下程序片段把当前数据段中偏移 1000H 开始的 100 个字节的数据传送到从偏移 2000H 开始的单元中。

```
        CLD                 ;清方向标志(以便按增值方式调整指针)
        PUSH    DS          ;由于在当前数据段中传送数据
        POP     ES          ;所以使 ES 等于 DS
        MOV     SI,1000H    ;置源串指针初值
        MOV     DI,2000H    ;置目的串指针初值
        MOV     CX,100      ;置循环次数
NEXT:   LODSB               ;取一字节数据
        STOSB               ;存一字节数据
```

```
        LOOP    NEXT        ;循环CX次
```

如果方向标志已清，则清方向标志的指令可省；如果当前附加段和当前数据已是重叠的，则也就无需再给ES赋值。

4. 字符串传送指令(MOVe String)

字符串传送指令的格式如下：

```
    MOVSB       ;字节传送
    MOVSW       ;字传送
```

字节传送指令MOVSB把寄存器SI所指向的一个字节数据传送到由寄存器DI所指向的存储单元中，然后根据方向标志DF复位或置位使SI和DI之值分别增1或减1。字传送指令MOVSW把寄存器SI所指向的一个字数据传送到由寄存器DI所指向的存储单元中，然后根据方向标志DF复位或置位使SI和DI之值分别增2或减2。注意，根据DS和SI计算源操作数地址，根据ES和DI计算目的操作数地址。

字符串传送指令不影响标志。

该指令的源操作数和目的操作均在存储器中，它与下面的字符串比较指令一起属于特殊情况。

在汇编语言中，两条字符串传送指令的格式可统一为如下一种格式：

```
    MOVS        OPRD1,OPRD2
```

两个操作数的类型应该一致。汇编程序根据操作数的类型决定使用字节传送指令还是字传送指令。也即，如果操作数的类型为字节，则采用MOVSB指令；如果操作数的类型为字，则采用MOVSW指令。请注意，操作数OPRD1或OPRD2可起到方便阅读程序的作用，但不影响寄存器SI和DI之值，所以在使用上述格式的串传送指令时，仍必须先给SI和DI赋合适的值。

上面我们利用了字符串装入指令和字符串存储指令的结合实现数据块的移动，现在利用字符串传送指令实现数据块的移动。假设要求同上，程序片段如下，请作比较。

```
        CLD                     ;清方向标志
        ……                      ;其他指令同上
        MOV     CX,100          ;置循环次数
NEXT:   MOVSB                   ;每次传送一字节数据
        LOOP    NEXT
```

现在循环体中只有一条串传送指令，执行速度可明显提高。在这个程序片段中，把100个字节的数据当作以字节为单元的字符串，所以利用了字节传送指令。如果把这100个字节的数据当作以字为单元的字符串，那么这个字符串也就只有50个单元了，于是循环次数可减少一半，执行速度还会提高。改写后的程序片段如下：

```
        CLD                     ;清方向标志
        ……                      ;其他指令同上
        MOV     CX,100/2        ;置循环次数
```

```
NEXT: MOVSW              ;每次传送一字节数据
      LOOP    NEXT
```

5. 字符串扫描指令(SCAn String)

字符串比较指令的格式如下:

```
SCASB          ;串字节扫描
SCASW          ;串字扫描
```

串字节扫描指令SCASB把累加器AL的内容与由寄存器DI所指向一个字节数据采用相减方式比较,相减结果反映到各有关标志位(AF,CF,OF,PF,SF和ZF),但不影响两个操作数,然后根据方向标志DF复位或置位使DI之值增1或减1。串字扫描指令SCASW把累加器AX的内容与由寄存器DI所指向的一个字数据比较,结果影响标志,然后DI之值增2或减2。

下面的程序片段判断AL中的字符是否为十六进制数符:

```
         ......
STRING   DB  '0123456789ABCDEFabcdef'
STRINGL  EQU    $  -  STRING
         ......                      ;把要判断的字符送AL
         CLD
         MOV  DX,SEG STRING
         MOV  ES,DX
         MOV  CX,STRINGL
         MOV  DI,OFFSET STRING
NEXT:    SCASB
         LOOPNZ   NEXT
         JNZ   NOT_FOUND
FOUND:   ......
         ......
NOT_FOUND:
         ......
```

在汇编语言中,两条字符串比较指令的格式可统一为如下一种格式:

```
SCAS      OPRD
```

汇编程序根据操作数的类型决定使用串字节扫描指令还是串字扫描指令。

6. 字符串比较指令(CoMPare String)

字符串比较指令的格式如下:

```
CMPSB          ;串字节比较
CMPSW          ;串字比较
```

串字节比较指令CMPS把寄存器SI所指向的一个字节数据与由寄存器DI所指向一个字节数据采用相减方式比较,相减结果反映到各有关标志位(AF,CF,OF,PF,SF和

ZF)，但不影响两个操作数，然后根据方向标志 DF 复位或置位使 SI 和 DI 之值分别增 1 或减 1。串字比较指令 CMPSW 把寄存器 SI 所指向的一个字数据与由寄存器 DI 所指向的一个字数据比较，结果影响标志，然后按调整值 2 调整 SI 和 DI 之值。

在汇编语言中，两条字符串比较指令的格式可统一为如下一种格式：

```
CMPS OPRD1,OPRD2
```

两个操作数的类型应该一致。汇编程序根据操作数的类型决定使用串字节比较指令还是串字比较指令。请注意，OPRD1 或 OPRD2 不影响寄存器 SI 和 DI 之值和段寄存器 DS 和 ES 之值。

6.1.2 重复前缀

由于串操作指令每次只能对字符串中的一个字符进行处理，所以使用了一个循环，以便完成对整个字符串的处理。为了进一步提高效率，8086/8088 还提供了重复指令前缀。重复前缀可加在串操作指令之前，达到重复执行其后的串操作指令的目的。

1. 重复前缀 REP

REP 用作为一个串操作指令的前缀，它重复其后的串操作指令动作。每一次重复都先判断 CX 是否为 0，如为 0 就结束重复，否则 CX 的值减 1，重复其后的串操作指令。所以当 CX 值为 0 时，就不执行其后的字符串操作指令。

它类似于 LOOP 指令，但 LOOP 指令是先把 CX 的值减 1，后再判是否为 0。

注意，在重复过程中的 CX 减 1 操作，不影响各标志。

重复前缀 REP 主要用在串传送指令 MOVS 和串存储指令 STOS 之前。值得指出的是，一般不在 LODSB 或 LODSW 指令之前使用任何重复前缀。

使用重复前缀 REP，可进一步改写前面的移动数据块的程序片段如下，请作比较：

```
CLD                     ;如果已清方向标志，则这条指令可省
......                  ;其他指令同上
MOV     CX,50
REP     MOVSW           ;重复执行(CX)次
```

在下面的子程序中，重复前缀 REP 与串存储操作指令配合，实现用指定的字符填充指定的缓冲区。

```
;子程序名：FILLB
;功    能：用指定字符充填指定缓冲区
;入口参数：ES：DI＝缓冲区首地址
;          CX＝缓冲区长度，AL＝充填字符
;出口参数：无
FILLB   PROC
        PUSH    AX
        PUSH    DI
        JCXZ    FILLB_1         ;CX 值为 0 时直接跳过(可省)
```

```
        CLD
        SHR    CX,1          ;字节数转成字数
        MOV    AH,AL         ;使 AH 与 AL 相同
        REP    STOSW         ;按字充填
        JNC    FILLB_1       ;如果缓冲区长度为偶数,则转
        STOSB                ;补缓冲区长度为奇数时的一字节
FILLB_1: POP   DI
        POP    AX
        RET
FILLB   ENDP
```

在上面的子程序中,先按字充填缓冲区,然后再处理可能出现的"零头",这与重复CX次字节充填相比,可获得更高的效率。注意,字符串存储指令STOSW不影响标志。

2. 重复前缀 REPZ/REPE

REPZ与REPE是一个前缀的两个助记符,下面的介绍以REPZ为代表。

REPZ用作为一个串操作指令的前缀,它重复其后的串操作指令动作。每重复一次,CX的值减1,重复一直进行到CX为0或串操作指令使零标志ZF为0时止。重复结束条件的检查是在重复开始之前进行的。

注意,在重复过程中的CX值减1操作,不影响标志。

重复前缀REPZ主要用在字符串比较指令CMPS和字符串扫描指令SCAS之前。由于串传送指令MOVS和串存储指令STOS都不影响标志,所以在这些串操作指令前使用前缀REP和前缀REPZ的效果一样。

在下面的子程序中,重复前缀REPZ与串比较指令CMPSB配合,实现两个字符串的比较。重复前缀REPZ与CMPSB的配合表示当相同时继续比较。

```
;子程序名:STRCMP
;功    能:比较字符串是否相同
;入口参数:DS:SI=字符串1首地址的段值:偏移
;         ES:DI=字符串2首地址的段值:偏移
;出口参数:AX=0表示两字符串相同,否则表示字符串不同
;说    明:设字符串均以0为结束标志
STRCMP  PROC
        CLD
        PUSH   DI
        XOR    AL,AL        ;先测一个字符串的长度
        MOV    CX,0FFFFH
NEXT:   SCASB
        JNZ    NEXT
        NOT    CX           ;至此CX含字符串2的长度(包括结束标志)
        POP    DI
        REPZ   CMPSB        ;两个串比较(包括结束标志在内)
        MOV    AL,[SI-1]
```

```
        MOV     BL,ES:[DI-1]
        XOR     AH,AH       ;如两个字符串相同,则 AL 应等于 BL
        MOV     BH,AH
        SUB     AX,BX
        RET
STRCMP  ENDP
```

在上面子程序中最后使用的比较方法稍繁,但 CF 反映了两个字符串的字典序。

3. 重复前缀 REPNZ/REPNE

REPNZ 与 REPNE 是一个前缀的两个助记符,下面的介绍以 REPNZ 为代表。

REPNZ 用作为一个串操作指令的前缀。与 REPZ 类似,所不同的是重复一直进行到 CX 为 0 或串操作指令使零标志 ZF 为 1 时止。

重复前缀 REPNZ 主要用在字符串扫描指令 SCAS 之前。重复前缀 REPNZ 与 SCASB 指令配合,表示当不等时继续扫描,一直搜索到字符串结束。如果搜索到,则 ZF 标志为 1,CX 之值可能为 0;如果没有搜索到,则 ZF 标志为 0,CX 之值一定为 0。

上述的判断 AL 中的字符是否为十六进制数符的程序片段改写如下:

```
      ……                 ;同上
NEXT: REPNZ SCASB
      JNZ   NOT_FOUND
      ……                 ;同上
NOT_FOUND:
      ……
```

下面的子程序测字符串的长度,设字符串以 0 结尾。它巧妙地利用了重复前缀 REPNZ 和字符串扫描指令 SCASB。

```
;子程序名:STRLEN
;功    能:测字符串长度
;入口参数:ES:DI=字符串首地址的段值:偏移
;出口参数:AX=字符串长度
;说    明:字符串以 0 结尾;字符串长度不包括结尾标志
STRLEN  PROC
        PUSH    CX
        PUSH    DI
        CLD
        XOR     AL,AL       ;使 AL 含结束标志值
        MOV     CX,0FFFFH   ;取字符串长度极值
        REPNZ   SCASB       ;搜索结束标志 0
        MOV     AX,CX
        NOT     AX          ;得字符串包括结束标志在内的长度
        DEC     AX          ;减结束标志 1 字节
        POP     DI
```

```
        POP     CX
        RET
STRLEN  ENDP
```

如果重复前缀 REPZ 与 SCASB 相配合，则表示当相等时继续搜索，直到第一个不等时为止(当然 CX 之值决定了最终搜索的次数)。

4. 说明

重复的字符串处理操作过程可被中断。CPU 在处理字符串的下一个字符之前识别中断。如果发生中断，那么在中断处理返回以后，重复过程再从中断点继续执行下去。但应注意，如指令前还有其他前缀(段超越前缀或锁定前缀)的话，中断返回时其他的前缀就不再有效。因为 CPU 在中断时，只能"记住"一个前缀，即字符串操作指令前的重复前缀。如字符串操作指令必须使用一个以上的前缀，则可在此之前禁止中断。

6.1.3 字符串操作举例

下面再举几例来说明字符串操作指令和重复前缀的使用，同时说明如何进行字符串操作。

例 1：写一个判别字符是否在字符串中出现的子程序。设字符串以 0 结尾。

串扫描指令可用于在字符串中搜索指定的字符，从而判别字符是否属于字符串。下面的子程序并没有利用串扫描指令，代码虽长，自有其独到之处，请注意。

```
;子程序名：STRCHR
;功    能：判字符是否属于字符串
;入口参数：DS：SI=搜索字符串首地址的段值：偏移
;          AL=字符代码
;出口参数：CF=0 表示字符在字符串中，AX=字符首次出现处的偏移
;          CF=1 表示字符不在字符串中
STRCHR   PROC
         PUSH    BX
         PUSH    SI
         CLD
         MOV     BL,AL          ;字符复制到 BL 寄存器
         TEST    SI,1           ;判地址是否为偶
         JZ      STRCHR1        ;是，转
         LODSB                  ;取第一个字符，比较之
         CMP     AL,BL
         JZ      STRCHR3
         AND     AL,AL
         JZ      STRCHR2
STRCHR1: LODSW                  ;取一个字
         CMP     AL,BL          ;比较低字节
         JZ      STRCHR4
```

```
        AND     AL,AL
        JZ      STRCHR2
        CMP     AH,BL           ;比较高字节
        JZ      STRCHR3
        AND     AH,AH
        JNZ     STRCHR1
STRCHR2: STC
        JMP     SHORT STRCHR5
STRCHR3: INC    SI
STRCHR4: LEA    AX,[SI-2]
STRCHR5: POP    SI
        POP     BX
        RET
STRCHR  ENDP
```

上面的子程序对从奇地址开始存放的字符串的第一个字符作了特别处理。在随后的循环处理中，字符串便总从偶地址开始，每次取一个字，即两个字符，再逐个字符比较。为什么要从偶地址开始取一个字？较好的理由留给读者思考。

例 2：写一个在字符串 1 后追加字符串 2 的子程序。设字符串均以 0 结尾。

该子程序的实现流程如图 6.1 所示。现再作几点说明：(1)要传送的字符串 2 包括其结束标志；(2)字符串 2 的传送以字传送为主，考虑了从偶地址开始进行字的传送；(3)最后处理可能遗留的一字节

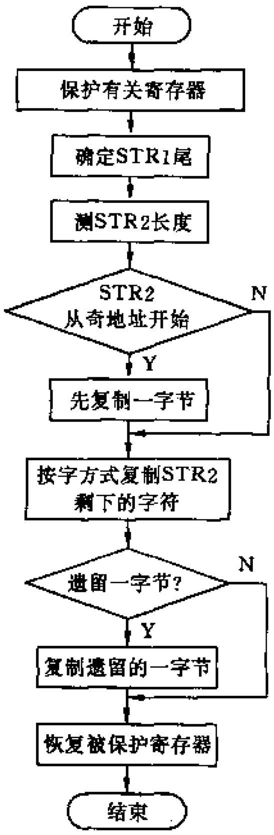


图 6.1 字符串拼接子程序的流程

```
;子程序名：STRCAT
;功    能：在字符串 1 末追加字符串 2
;入口参数：DS：SI=字符串 1 起始地址的段值：偏移
;          DS：DI=字符串 2 起始地址的段值：偏移
;出口参数：无
;说    明：不考虑在字符串 1 后是否留有足够的空间
STRCAT  PROC
        PUSH    ES
        PUSH    AX
        PUSH    CX
        PUSH    SI
        PUSH    DI
        CLD
        PUSH    DS
        POP     ES              ;使 ES 同 DS
```

```
        PUSH    DI
        MOV     DI,SI
        XOR     AL,AL
        MOV     CX,0FFFFH
        REPNZ   SCASB           ;确定字符串1的尾
        LEA     SI,[DI-1]       ;SI指向字符串1的结束标志
        POP     DI
        MOV     CX,0FFFFH
        REPNZ   SCASB           ;测字符串2的长度
        NOT     CX              ;CX为字符串2包括结束标志的长度
        SUB     DI,CX           ;DI再次指向字符串2的首
        XCHG    SI,DI           ;为拼接作准备
        TEST    SI,1            ;字符串2是否从奇地址开始?
        JZ      STRCAT1
        MOVSB                   ;特别处理第一字节
        DEC     CX
STRCAT1: SHR    CX,1            ;移动数据块长度除2
        REPZ    MOVSW           ;字移动
        JNC     STRCAT2
        MOVSB                   ;补字移动时遗留的一字节
STRCAT2: POP    DI
        POP     SI
        POP     CX
        POP     AX
        POP     ES
        RET
STRCAT  ENDP
```

例3：写一个程序，它先接收一个字符串，然后抽去其中的空格，最后按相反的顺序显示它。

程序如下所示，请注意删除空格的方法和方向标志的变化。

```
;程序名：T6-1.ASM
;功  能：接收一个字符串，去掉其中的空格后按相反的顺序显示它
;符号常量的定义
MAXLEN  =  64                           ;字符串最大长度
SPACE   =  ''                           ;空格
CR      =  0DH                          ;回车符
LF      =  0AH                          ;换行符
;数据段的定义
DSEG    SEGMENT
BUFFER DB  MAXLEN+1,0,MAXLEN+1  DUP  (0)
```

```
STRING  DB  MAXLEN+3  DUP  (0)
DSEG    ENDS
;代码
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:DSEG,ES:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     ES,AX
        MOV     DX,OFFSET BUFFER
        MOV     AH,10
        INT     21H                     ;接收一个字符串
        XOR     CH,CH
        MOV     CL,BUFFER+1
        JCXZ    OK                      ;字符串长度为 0,则结束
        ;
        CLD
        MOV     SI,OFFSET BUFFER +2
        MOV     DI,OFFSET STRING
        XOR     AL,AL
        STOSB                           ;先存入结束符
        MOV     AL,SPACE
PP1:    XCHG    SI,DI
        REPZ    SCASB                   ;去掉空格
        XCHG    SI,DI
        JCXZ    PP3                     ;如已到尾,转
        DEC     SI                      ;恢复被 REPZ SCASB 指令扫过的字符
        INC     CX                      ;及计数
PP2:    CMP     BYTE PTR [SI],SPACE     ;欲传字符为空格?
        JZ      PP1                     ;是,去掉空格
        MOVSB                           ;传一字符
        LOOP    PP2                     ;下一个
        ;
PP3:    MOV     AL,CR
        STOSB                           ;存入回车符
        MOV     AL,LF
        MOV     [DI],AL                 ;存入换行符
        STD                             ;置方向标志
        MOV     SI,DI
PP4:    LODSB                           ;按相反顺序取字符
        OR      AL,AL
        JZ      OK                      ;结束,转
        MOV     DL,AL
        MOV     AH,2
        INT     21H                     ;显示字符
        JMP     PP4
        ;
```

```
OK:     MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

例 4：写一个判字符串 2 是否为字符串 1 子串的子程序。具体要求如下：(1)子程序是一个远过程；(2)指向字符串的指针是远指针(即包括段值)；(3)通过堆栈传递两个分别指向字符串 1 和字符串 2 的远指针；(4)由 DX：AX 返回指向字符串 2 在字符串 1 中首次出现处的指针，如字符串 2 不是字符串 1 的子串，则返回空指针；(5)字符串均以 0 为结束符。

第 3 章中的程序 T3-16. ASM 完成类似的工作，现在利用串操作指令来实现它，图 6. 2是实现的流程图，请比较。图 6. 3 是调用进入该子程序后的堆栈。

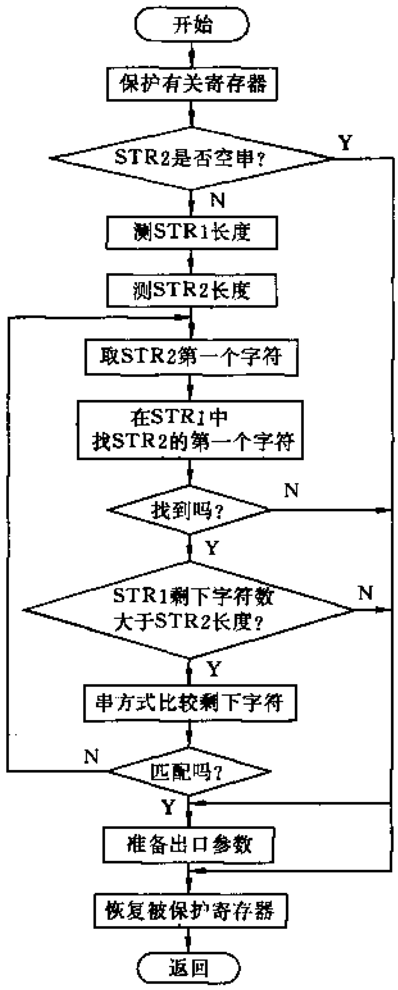


图 6. 2　子程序 STRSTR 的流程图

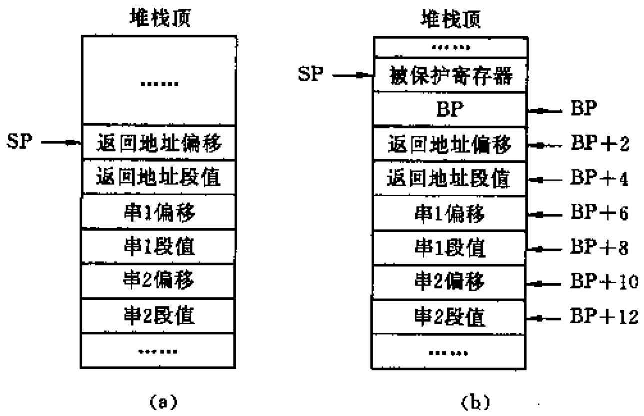


图 6.3　进入 STRSTR 子程序后的堆栈示意图

```
;子程序名：STRSTR
;功    能：判字符串 2 是否为字符串 1 的子串
;入口参数：指向字符串的远指针(见调用方法)
;出口参数：DX：AX 返回指向字符串 2 在字符串 1 中首次出现处的指针
;说    明：调用方法如下：
          (1) 压入字符串 2 的远指针
          (2) 压入字符串 1 的远指针
          (3) CALL   FAR  PTR   STRSTR
STRSTR     PROC    FAR
           PUSH    BP                      ;此前的堆栈如图 6.3(a)所示
           MOV     BP,SP
           PUSH    DS
           PUSH    ES
           PUSH    BX                      ;保护有关寄存器
           PUSH    CX
           PUSH    SI
           PUSH    DI                      ;此时的堆栈如图 6.3(b)所示
           ;
           LES     BX,[BP+10]              ;取 STR2 指针
           CMP     BYTE PTR ES：[BX],0     ;判 STR2 是否为空串
           JNZ     STRSTR1                 ;否
           MOV     DX,[BP+8]               ;STR2 为空串时
           MOV     AX,[BP+6]               ;返回 STR1 指针
           JMP     SHORT STRSTR6
           ;
STRSTR1：  CLD
           LES     DI,[BP+6]               ;取 STR1 指针
           PUSH    ES
```

```
            MOV     BX,DI               ;STR1 偏移送 BX 寄存器
            XOR     AX,AX
            MOV     CX,0FFFFH
            REPNZ   SCASB               ;测 STR1 长度
            NOT     CX
            MOV     DX,CX               ;使 DX 含 STR1 长度(含结束标志)
            ;
            LES     DI,[BP+10]          ;取 STR2 指针
            PUSH    ES
            MOV     BP,DI               ;STR2 偏移送 BP 寄存器
            XOR     AX,AX
            MOV     CX,0FFFFH
            REPNZ   SCASB               ;测 STR2 长度
            NOT     CX
            DEC     CX                  ;CX 为 STR2 长度
            POP     DS                  ;此时,DS:BP 指向 STR2
            POP     ES                  ;ES:BX 指向 STR1
            ;
STRSTR2:    MOV     SI,BP               ;DS:SI 指向 STR2
            LODSB                       ;取 STR2 的第一个字符
            MOV     DI,BX
            XCHG    CX,DX               ;使 CX 含 STR1 长度,DX 含 STR2 长度
            REPNZ   SCASB               ;在 STR1 中搜索 STR2 的第一个字符
            MOV     BX,DI
            JNZ     STRSTR3             ;找到?
            CMP     CX,DX               ;找到,STR1 剩下的字符数比 STR2 长?
            JNB     STRSTR4             ;是转
STRSTR3:    XOR     BX,BX               ;找不到处理
            MOV     ES,BX
            MOV     BX,1
            JMP     SHORT STRSTR5
            ;
STRSTR4:    XCHG    CX,DX               ;使 CX 含 STR2 长度,DX 含 STR1 长度
            MOV     AX,CX
            DEC     CX
            REPZ    CMPSB               ;判 STR1 中是否有 STR2 后的其它字符
            MOV     CX,AX
            JNZ     STRSTR2             ;没有,转继续找
            ;
STRSTR5:    MOV     AX,BX               ;找到!
            DEC     AX                  ;准备返回值
            MOV     DX,ES
```

```
            ;
STRSTR6:    POP     DI
            POP     SI
            POP     CX
            POP     BX                  ;恢复有关寄存器
            POP     ES
            POP     DS
            POP     BP
            RET                         ;RETF
STRSTR ENDP
```

6.2 十进制数算术运算调整指令及应用

在 2.4 节中介绍的算术运算指令都是对二进制数进行操作，为了方便地进行十进制数的算术运算，8086/8088 提供了各种调整指令。本节介绍这些调整指令和举例说明它们的应用。

8086/8088 的十进制数算术运算调整指令所认可的十进制数是以 8421BCD 码(见表 1.1)表示的，它分为未组合(或非压缩)的和组合(或压缩)的两种。组合的 BCD 码是指一字节含两位 BCD 码；未组合的 BCD 码是指一字节含一位 BCD 码，字节的高四位无意义。

数字的 ASCII 码是一种非组合的 BCD 码。因为数字的 ASCII 码的低四位是对应的 8421BCD 码。

6.2.1 组合 BCD 码的算术运算调整指令

1. 组合的 BCD 码加法调整指令 DAA(Decimal Adjust for Addition)

组合的 BCD 码加法调整指令的格式如下：

```
DAA
```

这条指令对在 AL 中的和(由两个组合的 BCD 码相加后的结果)进行调整，产生一个组合的 BCD 码。调整方法如下：

(1) 如 AL 中的低 4 位在 A～F 之间，或 AF 为 1，则 AL←(AL)+6，且 AF 位置 1；

(2) 如 AL 中的高 4 位在 A～F 之间，或 CF 为 1，则 AL←(AL)+60H，且 CF 位置 1。

该指令影响标志 AF、CF、PF、SF 和 ZF，但不影响标志 OF。

下面是为了说明该指令而写的一个程序片段，每条指令执行后的结果作为注释给出。第一条指令使 AL 含表示两位十进制数 34 的组合 BCD 码；第二条指令进行加操作，因 ADD 是二进制数相加，所以结果为 7BH，但作为十进制数 34 加 47 的结果应为 81。第三条指令进行调整，得正确结果 81。第五条指令又把由第四条指令相加的结果进行调整，得结果 68(百位进入 CF)。第七条指令把由第六条指令相加的结果进行调整，得结果 48(百位进入 CF)。

```
MOV    AL,34H
ADD    AL,47H          ;AL=7BH,  AF=0,  CF=0
DAA                    ;AL=81H,  AF=1,  CF=0
ADC    AL,87H          ;AL=08H,  AF=0,  CF=1
DAA                    ;AL=68H,  AF=0,  CF=1
ADC    AL,79H          ;AL=E2H,  AF=1,  CF=0
DAA                    ;AL=48H,  AF=1,  CF=1
```

2. 组合的 BCD 码减法调整指令 DAS(Decimal Adjust for Subtraction)

组合的 BCD 码减法调整指令的格式如下：

```
DAS
```

这条指令对在 AL 中的差(由两个组合的 BCD 码相减后的结果)进行调整，产生一个组合的 BCD 码。调整方法如下：

(1) 如 AL 中的低 4 位在 A～F 之间，或 AF 为 1，则 AL←(AL)－6，且 AF 位置 1；

(2) 如 AL 中的高 4 位在 A～F 之间，或 CF 为 1，则 AL←(AL)－60H，且 CF 位置 1。

该指令影响标志 AF，CF，PF，SF 和 ZF，但不影响标志 OF。

下面是为了说明该指令而写的一个程序片段，每条指令执行后的结果作为注释给出。第一条指令使 AL 含表示两位十进制数 35 的组合 BCD 码；第二条指令进行减操作，因 SUB 是二进制数相减，所以结果为 1EH，但作为十进制数 45 减 27 的结果应为 18。第三条指令进行调整，得正确结果 18。第五条指令又把由第四条指令相减的结果进行调整，得结果 69(百位上的借位在 CF 中)。

```
MOV    AL,45H
SUB    AL,27H          ;AL=1EH,  AF=1,  CF=0
DAS                    ;AL=18H,  AF=1,  CF=0
SBB    AL,49H          ;AL=CFH,  AF=1,  CF=1
DAS                    ;AL=69H, AF=1, CF=1
```

6.2.2 未组合 BCD 码的算术运算调整指令

1. 未组合的 BCD 码加法调整指令 AAA(ASCII Adjust for Addition)

未组合的 BCD 码加法调整指令的格式如下：

```
AAA
```

这条指令对在 AL 中的和(由两个未组合的 BCD 码相加后的结果)进行调整，产生一个未组合的 BCD 码。调整方法如下：

(1) 如 AL 中的低 4 位在 0～9 之间，且 AF 为 0，则转(3)；

(2) 如 AL 中的低 4 位在 A～F 之间，或 AF 为 1，则 AL←(AL)＋6，AH←(AH)＋1，且 AF 位置 1；

(3) 清除 AL 的高 4 位；

(4) AF 位的值送 CF 位。

该指令影响标志 AF 和 CF，对其他标志均无定义。

下面是为了说明该指令而写的一个程序片段，每条指令执行后的结果作为注释给出，请注意比较：

```
MOV    AX,7
ADD    AL,6        ;AL=0DH,  AH=00H,  AF=0,  CF=0
AAA                ;AL=03H,  AH=01H,  AF=1,  CF=1
ADC    AL,5        ;AL=09H,  AH=01H,  AF=0,  CF=0
AAA                ;AL=09H,  AH=01H,  AF=0,  CF=0
ADD    AL,39H      ;AL=42H,  AH=01H,  AF=1,  CF=0
AAA                ;AL=08H,  AH=02H,  AF=1,  CF=1
```

2. 未组合的 BCD 码减法调整指令 AAS(ASCII Adjust for Subtraction)

未组合的 BCD 码减法调整指令的格式如下：

```
DAS
```

这条指令对在 AL 中的差(由两个未组合的 BCD 码相减后的结果)进行调整，产生一个未组合的 BCD 码。调整方法如下：

(1) 如 AL 中的低 4 位在 0～9 之间，且 AF 为 0，则转(3)；

(2) 如 AL 中的低 4 位在 A～F 之间，或 AF 为 1，则 AL←(AL)－6，AH←(AH)－1，且 AF 位置 1；

(3) 清除 AL 的高 4 位；

(4) AF 位的值送 CF 位。

该指令影响标志 AF 和 CF，对其他标志均无定义。

下面是为了说明该指令而写的一个程序片段，每条指令执行后的结果作为注释给出，请注意比较：

```
MOV    AL,34H
SUB    AL,09H      ;AL=2BH,  AF=1,  CF=0
AAS                ;AL=05H,  AF=1,  CF=1
```

3. 未组合的 BCD 码乘法调整指令 AAM(ASCII Adjust for Multiplication)

未组合的 BCD 码乘法调整指令的格式如下：

```
AAM
```

这条指令对在 AL 中的积(由两个组合的 BCD 码相乘的结果)进行调整，产生两个未组合的 BCD 码。调整方法如下：

把 AL 中的值除以 10，商放在 AH 中，余数放在 AL 中。

该指令影响标志 SF，ZF 和 PF，对其他标志无影响。

下面是为了说明该指令而写的一个程序片段，每条指令执行后的结果作为注释给出，请注意比较：

```
MOV     AL,03H
MOV     BL,04H
MUL     BL          ;AL=0CH,  AH=00H
AAM                 ;AL=02H,  AH=01H
```

4. 未组合的 BCD 码除法调整指令 AAD(ASCII Adjust for Division)

未组合的 BCD 码除法调整指令的格式如下：

```
AAD
```

该指令和其他调整指令的使用次序上不同，其他调整指令均安排在有关算术运算指令后，而这条指令应安排在除运算指令之前。它的功能是：把存放在寄存器 AH(高位十进制数)及存放在寄存器 AL 中的两位非组合 BCD 码，调整为一个二进制数，存放在寄存器 AL 中。调整的方法如下：

AL⇐AH * 10+(AL)

AH⇐0

由于采用上述调整方法，存放在 AL 和 AH 中的非组合 BCD 的高四位应为 0。

该指令影响标志 SF,ZF 和 PF，对其他标志无影响。

下面是为了说明该指令而写的一个程序片段，每条指令执行后的结果作为注释给出，请注意比较：

```
MOV     AH,04H
MOV     AL,03H
MOV     BL,08H
AAD                 ;AL=2BH,  AH=00H
DIV     BL          ;AL=05H,  AH=03H
```

6.2.3 应用举例

例 1：设在缓冲区 DATA 中存放着 12 个组合的 BCD 码，求它们的和，把结果存放到缓冲区 SUM 中。

有关的程序片段如下：

```
        ......
NUM1    DB  23H,45H,67H,89H,32H,93H,36H,12H,66H,78H,43H,99H
RESULT  DB  2     DUP  (0)
        ......
        MOV     AX,SEG NUM1
        MOV     DS,AX
        MOV     BX,OFFSET DATA
        MOV     CX,10                   ;准备循环
        XOR     AL,AL
        XOR     AH,AH
```

```
NEXT:   ADD     AL,[BX]                 ;加
        DAA                             ;调整
        ADC     AH,0                    ;考虑进位
        XCHG    AH,AL
        DAA                             ;调整
        XCHG    AH,AL
        INC     BX                      ;修改指针
        LOOP    NEXT                    ;下一个
        XCHG    AH,AL                   ;准备高位低地址存放
        MOV     WORD PTR RESULT,AX
        ……
```

例 2：利用 DAA 指令改写把一位十六进制数转换为对应的 ASCII 码符的子程序 HTOASC。

下面的子程序巧妙地利用了加法调整指令 DAA，使得在子程序中没有条件转移指令。

```
;子程序名：HTOASC
;功    能：把一位十六进制数转换为对应的 ASCII 码
;入口参数：AL 的低 4 位为要转换的十六进制数
;出口参数：AL 含对应的 ASCII 码
HTOASC  PROC
        AND     AL,0FH
        ADD     AL,90H
        DAA
        ADC     AL,40H
        DAA
        RET
HTOASC  ENDP
```

请读者仔细考虑上述子程序。选几个十六进制数试试。

例 3：写一个能实现两个十进制数的加法运算处理的程序。设每个十进制数最多 10 位。

如果不采用十进制数算术运算调整指令，那么在接收了以 ASCII 码串表示的十进制数后，要把它转换为二进制数。在对二进制数进行运算后，还要把结果转换为十进制数的 ASCII 码。当要处理的十进制数位数较多时，这种转换较麻烦。现采用十进制数算术运算调整指令完成它。

该程序分为如下四步：(1)接收按十进制表示的被加数，并作适当的处理；(2)接收按十进制表示的加数，也作适当的处理；(3)进行加法处理；(4)显示结果。为此，设计三个子程序。它们分别是：子程序 GETNUM 接收按十进制数表示的数串并作适当的处理；子程序 ADDITION 进行加法处理；子程序 DISPNUM 显示结果。

在子程序 ADDITION 中，使用非组合 BCD 码加调整指令 AAA，所以十位的被加数

和加数均保持非组合 BCD 码串形式，产生的 11 位和也是非组合 BCD 码串。子程序 GETNUM 通过 DOS 的 0AH 号系统功能调用，接收一个字符串，然后检查用户输入是否确实输入了一个十进制数，最后形成一个十位的非组合 BCD 码串(不足用 0 补足)。子程序 DISPNUM 比较容易，先跳过结果中可能存在的前导的 0，然后把非组合的 BCD 码转换为 ASCII 码后显示之。

```
;程序名：T6-2.ASM
;功  能：完成两个由用户输入的 10 位十进制数的加法运算
;常数定义
MAXLEN    =      10                       ;最多位数
BUFFLEN   =      MAXLEN+1                 ;缓冲区长度
;数据段
DSEG      SEGMENT
BUFF1     DB     BUFFLEN, 0, BUFFLEN  DUP  (?)     ;存放被加数
NUM1      EQU    BUFF1 + 2
BUFF2     DB     BUFFLEN, 0, BUFFLEN  DUP  (?)     ;存放加数
NUM2      EQU    BUFF2 + 2
RESULT    DB     BUFFLEN DUP (?),24H      ;存放和
DIGITL    DB     '0123456789'             ;有效的十进制数字符
DIGLEN    EQU    $ - DIGITL
MESS      DB 'Invalid number! ',0DH,0AH,24H
DSEG      ENDS
;代码段
CSEG      SEGMENT
          ASSUME CS : CSEG,DS : CSEG,ES : CSEG
START:    MOV    AX,DSEG
          MOV    DS,AX                    ;置 DS 和 ES
          MOV    ES,AX
          MOV    DX,OFFSET BUFF1
          CALL   GETNUM                   ;(1)接收被加数
          JC     OVER                     ;不合法时，处理
          MOV    DX,OFFSET BUFF2
          CALL   GETNUM                   ;(2)接收加数
          JC     OVER                     ;不合法时，处理
          MOV    SI,OFFSET NUM1
          MOV    DI,OFFSET NUM2
          MOV    BX,OFFSET RESULT
          MOV    CX,MAXLEN
          CALL   ADDITION                 ;(3)加运算
          MOV    DX,OFFSET RESULT
          CALL   DISPNUM                  ;(4)显示结果
          JMP    SHORT OK
```

```
OVER:     MOV    DX,OFFSET MESS        ;出错处理
          MOV    AH,9
          INT    21H
OK:       MOV    AH,4CH
          INT    21H
          ;

;子程序名: GETNUM
;功    能: 接收一个十进制数字串,且扩展成 10 位
;入口参数: DX=缓冲区偏移
;出口参数: CF=0,表示成功;CF=1,表示不成功
GETNUM    PROC
          MOV    AH,10                 ;接收一个字符串
          INT    21H
          CALL   NEWLINE               ;产生回车和换行
          CALL   ISDNUM                ;判是否为十进制数字串
          JC     GETNUM2               ;不是,转
          MOV    SI,DX
          INC    SI
          MOV    CL,[SI]               ;取输入的数字串长度
          XOR    CH,CH
          MOV    AX,MAXLEN
          STD
          MOV    DI,SI
          ADD    DI,AX
          ADD    SI,CX
          SUB    AX,CX
          REP    MOVSB                 ;数字串向高地址移让出低地址
          MOV    CX,AX
          JCXZ   GETNUM1
          XOR    AL,AL                 ;低地址的高位用 0 补足
          REP    STOSB
GETNUM1:  CLD
          CLC
GETNUM2:  RET
GETNUM    ENDP

;子程序名: ADDITION
;功    能: 多位非组合 BCD 码数加
;入口参数: SI=代表被加数的非组合 BCD 码串开始地址偏移
;          DI=代表加数的非组合 BCD 码串开始地址偏移
;          CX=BCD 码串长度(字节数)
```

```
;           BX=存放结果的缓冲区开始地址偏移
;出口参数：结果缓冲区含结果
;说    明：在非组合的BCD码中，十进制数的高位在低地址
ADDITION  PROC
          STD                        ;准备从在高地址的低位开始处理
          ADD    BX,CX               ;BX指向结果缓冲区最后一字节
          ADD    SI,CX
          ADD    DI,CX
          DEC    SI                  ;SI指向被加数串最后一字节
          DEC    DI                  ;DI指向加数串最后一字节
          XCHG   DI,BX               ;BX指向加数串,DI指向结果串
          INC    BX                  ;循环前的指针预调整
          CLC
ADDP1:    DEC    BX
          LODSB                      ;取一字节被加数
          ADC    AL,[BX]             ;加上加数(带上低位的进位)
          AAA                        ;调整
          STOSB                      ;保存结果
          LOOP   ADDP1               ;循环处理下一个
          MOV    AL,0
          ADC    AL,0                ;考虑最后一次进位
          STOSB                      ;保存之
          CLD
          RET
ADDITION  ENDP

;子程序名：DISPNUM
;功    能：显示结果
;入口参数：DX=结果缓冲区开始地址偏移
;出口参数：无
DISPNUM   PROC
          MOV    DI,DX
          MOV    AL,0
          MOV    CX,MAXLEN
          REPZ   SCASB               ;跳过前导的0
          DEC    DI
          MOV    DX,DI
          MOV    SI,DI
          INC    CX
DISPNU2:  LODSB                      ;把非组合BCD码串转换成ASCII码串
          ADD    AL,30H
          STOSB
```

```
        LOOP    DISPNU2
        MOV     AH,9                    ;显示结果
        INT     21H
        RET
DISPNUM ENDP

;子程序名：ISDNUM
;功    能：判一个利用 DOS 的 0AH 号功能调用输入的字符串是否为数字符串
;入口参数：DX=缓冲区开始地址偏移
;出口参数：CF=0,表示是;CF=1,表示否
ISDNUM  PROC
        MOV     SI,DX
        LODSB
        LODSB                           ;AL=字符串长度
        MOV     CL,AL
        XOR     CH,CH
        JCXZ    ISDNUM2                 ;空串认为非数字串
ISDNUM1: LODSB                          ;取一个字符
        CALL    ISDECM                  ;判该字符是否为数字符
        JNZ     ISDNUM2                 ;不是,转
        LOOP    ISDNUM1                 ;是,下一个
        RET
ISDNUM2: STC
        RET
ISDNUM  ENDP

;子程序名：ISDECM
;功    能：判一个字符是否为十进制数字符
;入口参数：AL=字符
;出口参数：ZF=1,表示是;ZF=0,表示否
ISDECM  PROC
        PUSH    CX
        MOV     DI,OFFSET DIGITL
        MOV     CX,DIGLEN
        REPNZ   SCASB
        POP     CX
        RET
ISDECM  ENDP

;子程序说明信息略
NEWLINE         PROC
                ;该子程序的代码同 T4-3.ASM 中同名子程序
```

```
NEWLINE         ENDP
CSEG      ENDS
          END      START
```

6.3 DOS 程序段前缀和特殊情况处理程序

本节把几个特殊情况处理程序作为简单应用程序的实例作介绍，其中也说明了软中断处理程序的设计方法。

6.3.1 DOS 程序段前缀 PSP

1. 程序段前缀 PSP

程序段前缀(Program Segment Prefix)是 DOS 加载一个外部命令或应用程序(EXE 或 COM 类型)时，在程序段之前设置的一个具有 256 字节的信息区。

PSP 含有许多可用信息，其中常用信息的安排如表 6.1 所示。

表 6.1 程序段前缀 PSP 的常用信息安排

偏移	内容及意义	偏移	内容及意义
00H	程序终止退出命令 INT 20H	16H	保留
02H	可用的内存空间高端段值	2CH	环境块段值
04H	保留	2EH	保留
05H	DOS 系统功能远调用指令	5CH	格式化的未打开的文件控制块 1
0AH	程序结束处理中断向量原内容	6CH	格式化的未打开的文件控制块 2
0EH	Ctrl+C 键处理中断向量原内容	80H	命令行参数长度(字节数)
12H	严重错误处理中断向量原内容	81H	命令行参数

当 DOS 把控制权转给外部命令或应用程序之时，数据段寄存器 DS 和附加段寄存器 ES 均指向其 PSP，即均含有 PSP 的段值，并不指向程序的数据段和附加段。这样应用程序可方便地使用到 PSP 中的有关信息。

2. 终止程序的另一途径

利用 DOS 的 4CH 号系统功能调用能终止程序，把控制权转交回 DOS，这是我们现在常用的方法。但早先常利用 DOS 提供的 20H 号中断处理程序来终止程序。

通过 20H 号中断处理程序终止程序有一个条件，即进入 20H 号中断处理程序之前，代码段寄存器 CS 必须含有 PSP 的段值。由于对 EXE 类型的应用程序而言，其代码段与 PSP 不是同一个段，所以不能简单地直接利用指令"INT 20H"来终止程序。DOS 注意到了这一点，在 PSP 的偏移 0 处，安排了一条"INT 20H"指令(其机器码为 CD 20)。于是，应用程序只要设法转到 PSP 的偏移 0 处，就能实现程序的终止。下面是一个例子，它是程序 T3-1. ASM 的改写。

```
;程序名:T6-3.ASM
;功  能:显示信息"HELLO"
SSEG    SEGMENT  PARA  STACK          ;堆栈段
        DW    256  DUP (?)
SSEG    ENDS
DSEG    SEGMENT                       ;数据段
MESS    DB    'HELLO',0DH,0AH,'$'
DSEG    ENDS
CSEG    SEGMENT                       ;代码段
        ASSUME  CS:CSEG, DS:DSEG
MAIN    PROC     FAR
START:  PUSH     DS                   ;把 PSP 的段值压入堆栈
        XOR      AX,AX
        PUSH     AX                   ;把 0000H(偏移)压入堆栈
        ;
        MOV      AX,DSEG
        MOV      DS,AX
        MOV      DX,OFFSET MESS
        MOV      AH,9
        INT      21H
        RET                           ;转 PSP 的偏移 0 处执行
MAIN    ENDP
CSEG    ENDS
        END      START
```

在标号 START 开始处的三条指令把 PSP 的段值和值 0 压入堆栈,最后的返回指令 RET 从堆栈弹出程序开始时压入堆栈的 PSP 段值和偏移 0 到 CS 和 IP,随后 CPU 就执行位于 PSP 首的指令"INT 20H",此时的 CS 含 PSP 段值,程序终止,控制转回 DOS。细心的读者也许发现,T6-3.ASM 与 T6-1.ASM 相比,还多出一对过程定义伪指令语句,即定义了一个名为 MAIN 的远过程,这样做的目的是告诉汇编程序,把为了终止程序返回 DOS 而设的 RET 指令汇编成远返回指令。如果利用 TASM 汇编程序,则完全可不定义过程 MAIN,而使用远返回指令的助记符 RETF。

但我们不鼓励采用上述方法终止程序,而是推荐通过 DOS 系统功能调用 4CH 终止程序。

3. 应用程序取得命令行参数

DOS 加载一个外部命令或应用程序时,允许在被加载的程序名之后,输入多达 127 个字符(包括最后的回车符)的参数,并把这些参数送到 PSP 的非格式化参数区,即 PSP 中从偏移 80H 开始的区域。

应用程序可从 PSP 中获得命令行参数。PSP 的偏移 80H 处含命令行参数的长度(字节数),从 PSP 的偏移 81H 开始存放命令行参数。命令行参数通常以空格符引导,至回车符(0DH)结束。注意,命令行中的重定向符和管道符及有关信息不作为命令行参数送到 PSP。

例 1：写一个显示命令行参数的程序。

先从 PSP 中把命令行参数传到程序定义的缓冲区中，然后再显示。源程序如下所示，数据段和代码段相重。

```
;程序名：T6-4.ASM
;功  能：显示命令行参数
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:CSEG
BUFFER DB      128  DUP (?)            ;用于存放命令行参数的缓冲区
START: CLD
        MOV     SI,80H
        LODSB                          ;取得命令行参数长度
        MOV     CL,AL
        XOR     CH,CH                  ;CX 含命令行参数字节数
        PUSH    CS                     ;该程序中数据和代码在一个段中
        POP     ES
        MOV     DI,OFFSET BUFFER
        PUSH    CX
        REP     MOVSB                  ;传命令行参数
        POP     CX
        ;
        PUSH    ES
        POP     DS                     ;置数据段寄存器
        MOV     SI,OFFSET BUFFER
        MOV     AH,2
        JCXZ    OVER
NEXT:   LODSB                          ;显示命令行参数
        MOV     DL,AL
        INT     21H
        LOOP    NEXT
OVER:   MOV     AX,4C00H               ;程序结束
        INT     21H
CSEG    ENDS
        END     START
```

例 2：写一个显示文本文件内容的程序。文件名作为命令行参数给出。

程序 T4-6.ASM 显示当前盘当前目录下的文本文件 TEST.TXT 的内容，只要对它稍作修改，即把由用户输入的命令行参数传送到文件名缓冲区就可。此外，当没有命令行参数，或命令行参数仅是空格等符号时，给出有关提示信息；为了简单，没有具体分析命令行参数，仅仅去掉前导空格符后，就把余下的参数作为文件标识符处理。源程序如下：

```
;程序名：T6-5.ASM
;功  能：显示文本文件的内容
```

```
;符号常量定义
LENOFID      =     128                        ;文件标识符最大长度
SPACE        =     ''
TAB          =     09H
EOF          =     1AH                        ;文件结束符的 ASCII 码
;数据段
DSEG         SEGMENT
FNAME        DB  LENOFID  DUP  (?)            ;准备存放文件名串
ERROR        DB  'Required parameter missing',0
             ......                           ;同 T4-6.ASM 对应部分
DSEG         ENDS
;代码段
CSEG         SEGMENT
             ASSUME  CS:CSEG,  DS:DSEG
START:       CLD
             MOV     SI,80H
             LODSB                            ;取命令行参数长度
             OR      AL,AL                    ;是否有命令行参数
             JNZ     GETFID1                  ;有
FIDERR:      MOV     AX,SEG ERROR0            ;没有命令行参数处理
             MOV     DS,AX
             MOV     SI,OFFSET ERROR0
             CALL    DMESS
             JMP     OVER
GETFID1:     MOV     CL,AL
             XOR     CH,CH                    ;CX 含命令行参数长度
GETFID2:     LODSB                            ;取参数一字节
             CMP     AL,SPACE                 ;为空格?
             JZ      GETFID3                  ;是,跳过
             CMP     AL,TAB                   ;为制表符?
             JNZ     GETFID4                  ;不是,表示已去掉前导空格
GETFID3:     LOOP    GETFID2                  ;跳过前导的空格和制表符
             JMP     FIDERR                   ;命令行参数没有其他符号,转
GETFID4:     DEC     SI
             MOV     AX,SEG  FNAME
             MOV     ES,AX
             MOV     DI,OFFSET  FNAME         ;把剩下的命令行参数送
             REP     MOVSB                    ;文件标识符区
             XOR     AL,AL
             STOSB                            ;再补一个 0,形成 ASCIIZ 串
             ;
             MOV     AX,DSEG
```

```
        MOV     DS,AX                   ;置数据段寄存器
        ;
        ……                              ;同 T4-6.ASM 对应部分
CSEG    ENDS
        END     START
```

6.3.2 对 Ctrl +C 键和 Ctrl+Break 键的处理

1. Ctrl +C 键的处理程序

先看下面的程序 T6-6.ASM。它的功能是在屏幕上显示用户所按字符，直到用户按 ESC 键为止。

```
;程序名：T6-6.ASM
;功  能：(略)
CR      =   0DH                         ;常量定义
LF      =   0AH
ESCAPE  =   1BH
;
CSEG    SEGMENT                         ;代码段
        ASSUME  CS:CSEG, DS:CSEG
START:  PUSH    CS
        POP     DS
CONT:   MOV     AH,8                    ;用不回显方式，接收一个字符
        INT     21H
        CMP     AL,ESCAPE               ;是否为 ESC 键?
        JZ      SHORT XIT               ;是，结束
        MOV     DL,AL
        MOV     AH,2                    ;否，显示所按字符
        INT     21H
        CMP     DL,CR                   ;是否为回车键?
        JNZ     CONT                    ;否，继续
        MOV     AH,2
        INT     21H
        MOV     DL,LF
        MOV     AH,2                    ;是，再显示换行符(形成回车和换行)
        INT     21H
        JMP     CONT                    ;继续
XIT:    MOV     AH,4CH                  ;结束
        INT     21H
CSEG    ENDS
        END     START
```

就上述程序而言，按 ESC 键能终止程序运行，这是符合程序功能要求的。但按 Ctrl+

C 键却能中止程序的运行。

当应用程序利用 DOS 系统功能调用进行字符输入输出(如键盘输入和显示输出等)时,DOS 通常要检测 Ctrl+C 键(对应 ASCII 码为 03H),请参见 4.3 节中给出的常用 DOS 功能调用。如检测,那么 DOS 在遇到 Ctrl+C 键后,就先显示符号"^C",并产生 INT 23H。缺省的 23H 号中断处理程序是中止程序的运行。DOS 提供的这一功能便于用户随机地中止一个执行错误或不必继续执行的程序。上述程序调用 DOS 的 8 号功能输入和 2 号功能输出,其间 DOS 要检测 Ctrl+C 键,所以按 Ctrl+C 键能中止上述程序的运行。

DOS 为应用程序改变这种处理方法作了准备。应用程序只要改变 23H 号中断处理程序,就可基本控制住对 Ctrl+C 键的处理。为了改变 23H 号中断处理程序,应用程序得提供一个新的 23H 号中断处理程序,然后修改 23H 号中断向量,使其指向新的 23H 号中断处理程序。由于 DOS 在设置 PSP 时,已把当时的 23H 号中断向量(即 Ctrl+C 键处理中断向量)保存到 PSP 中,且在程序终止时再自动从 PSP 中取出并恢复。所以,应用程序在修改 23H 号中断向量后,可不必恢复它。

下面的程序 T6-6A.ASM 就增加了 23H 号中断处理程序,该处理程序极其简单,只有一条中断返回指令 IRET,也即不作任何处理。所以在运行程序 T6-6A 时,按 Ctrl+C 键就不能再中止程序的运行。只要应用程序需要,可把按 Ctrl+C 键后要进行的工作安排在 23H 号中断处理程序中。

```
;程序名:T6-6A.ASM
;功  能:(略)
……                                      ;常量定义同 T6-6.ASM
CSEG     SEGMENT                        ;代码段
         ASSUME  CS:CSEG, DS:CSEG
;新的 23H 号中断处理程序
NEW23H:IRET                             ;中断返回
;-------------------------
;主程序
START:   PUSH   CS
         POP    DS
         MOV    DX,OFFSET  NEW23H       ;置 23H 号中断向量
         MOV    AX,2523H                ;使其指向新的处理程序
         INT    21H
         ;
CONT:    ……                            ;其他代码同 T6-6.ASM
         ……
         CSEG   ENDS
                END START
```

尽管按 Ctrl+C 键不再能中止 T6-6A 的运行,但屏幕上却会显示出符号"^C"。如果应用程序不在乎由于按 Ctrl+C 键带来的符号,那么上述处理就可接受。如果应用程序

"讨厌"由 Ctrl+C 键带来的符号，那么如下几种处理方法也许可满足需要：(1)应用程序使用不检测 Ctrl+C 键的 DOS 功能调用进行字符输入输出；(2)应用程序不利用 DOS 功能调用进行字符输入输出。就上述程序 T6-6.ASM 这样的应用程序而言，这两种方法都是可行的。对一般应用程序而言，这两种方法不完全有效。首先，应用程序不利用 DOS 功能调用进行字符输入输出，就要利用 BIOS 进行字符输入输出或直接进行输入输出，有时这是麻烦的。其次，在大多数 DOS 系统功能调用(不仅仅是输入输出功能调用)其间，DOS 要查看 Ctrl+Break 键是否被按下，如发现 Ctrl+Break 键被按，则也会显示符号"^C"和产生 INT 23H。下面先介绍对 Ctrl+Break 键的处理，然后再看一个基本解决问题的例子。

2. 对 Ctrl+Break 键的处理

键盘中断处理程序(9H 号中断处理程序)发现 Ctrl+Break 键被按时，将产生 INT 1BH。在 DOS 自举时，由 DOS 提供的 1BH 号中断处理程序将在约定的内存单元中设置一个标志，然后结束。DOS 通过该标志检测 Ctrl+Break 键是否被按下，如果发现被按下过，则象处理 Ctrl+C 那样显示符号"^C"和产生 INT 23H。

如果应用程序要自己处理 Ctrl+Break 键，则可通过提供新的 1BH 号中断处理程序的方法来实现。所以，如果应用程序要使得 Ctrl+Break 键不干扰程序的运行，只要使 1BH 号中断处理程序不设置与 DOS 约定的内存单元。但要注意，DOS 并不自动保存和恢复 1BH 号中断向量，所以如果应用程序要提供新的 1BH 号中断处理程序，那么在修改 1BH 号中断向量之前，先要保存原 1BH 号中断向量，在程序结束前恢复它。

下面的程序 T6-6B.ASM 是对 T6-6A.ASM 的修改，提供了新的 1BH 号中断处理程序。作为例子，新的 1BH 号中断处理程序只显示信息"**BREAK**"，然后就返回，所以在运行 T6-6B.ASM 时，按 Ctrl+Break 键不会出现符号"^C"。

```
;程序名： T6-6B.ASM
......                                          ;常量定义同 T6-6.ASM
CSEG        SEGMENT                             ;代码段
            ASSUME  CS:CSEG,  DS:CSEG
;变量和数据定义
OLD1BH      DD  ?                               ;保存原 1BH 号中断向量
VPAGE       DB  ?                               ;当前显示页
MESS        DB  '**BREAK**',0
;
;新的 1BH 号中断处理程序
NEW1BH:     PUSH    DS
            PUSH    AX                          ;保护现场
            PUSH    BX
            PUSH    SI
            PUSH    CS
            POP     DS
            CLD
```

```
            MOV     SI,OFFSET MESS
            MOV     BH,VPAGE                ;准备显示信息 * * BREAK * *
            MOV     AH,0EH
BRKNEXT:    LODSB                           ;取一字节信息
            OR      AL,AL                   ;结束?
            JZ      SHORT BRKEXIT           ;是
            INT     10H                     ;否则,显示
            JMP     BRKNEXT                 ;继续
BRKEXIT:    POP     SI
            POP     BX
            POP     AX                      ;恢复现场
            POP     DS
            IRET                            ;中断返回
;新的 23H 号中断处理程序
NEW23H:      IRET                           ;中断返回
;-------------------------
;主程序
START:      PUSH    CS
            POP     DS
            MOV     AH,0FH                  ;取显示状态信息
            INT     10H
            MOV     VPAGE,BH                ;保存当前显示页号
            ;
            MOV     AX,351BH
            INT     21H                     ;取原 1BH 中断向量并保存
            MOV     WORD PTR OLD1BH,BX
            MOV     WORD PTR OLD1BH+2,ES
            ;
            MOV     DX,OFFSET NEW23H        ;置 23H 号中断向量
            MOV     AX,2523H                ;使其指向新的处理程序
            INT     21H
            ;
            MOV     DX,OFFSET NEW1BH        ;置 1BH 号中断向量
            MOV     AX,251BH                ;使其指向新的处理程序
            INT     21H
            ;
CONT:       ……                              ;其他代码同 T6-6.ASM
            ……
XIT:        LDS     DX,OLD1BH
            MOV     AX,251BH                ;恢复原 1BH 号中断向量
            INT     21H
            ;
```

```
        MOV     AH,4CH                      ;程序正常终止
        INT     21H
CSEG    ENDS
        END     START
```

有一点值得指出，如果在 T6-6B. ASM 中不提供新的 23H 号中断处理程序，可能会导致麻烦。因为如果按 Ctrl+C 键将中止程序的运行，而原 1BH 号中断处理程序得不到恢复。

3. 一个能控制住 Ctrl+C 键和 Ctrl+Break 键的例子

在上述程序 T6-6B 中，控制住了 Ctrl+Break 键。在其运行时，按 Ctrl+C 键也不中止程序的运行，但仍会出现符号"^C"。现在我们修改键盘管理程序(16H 号中断处理程序)，使其不返回 Ctrl+C 键(即 ASCII 码 03H)。

在下面的程序中，提供了新的 16H 号中断处理程序，它"吃掉"了 Ctrl+C 键，同时也滤掉了 Ctrl+2 键(它类似于 Ctrl+C，其扫描码为 03H)。这样，DOS 就不可能检测到 Ctrl+C键了。新的 1BH 号中断处理程序仅是一条中断返回指令。所以，我们就不再提供新的 23H 号中断处理程序。

```
;程序名：T6-6C.ASM
......                                      ;常量定义同 T6-6.ASM
CSEG    SEGMENT                             ;代码段
        ASSUME  CS:CSEG, DS:CSEG            ;变量和数据定义
OLD1BH  DD  ?                               ;保存原 1BH 号中断向量
OLD16H  DD  ?                               ;保存原 16H 号中断向量
;
NEWKEY    PROC  FAR
;新的 16H 号中断处理程序
NEW16H: CMP     AH,10H
        JZ      PKEY
        CMP     AH,11H
        JZ      PKEY2
        CMP     AH,1
        JZ      PKEY2
        OR      AH,AH
        JZ      PKEY
        JMP     DWORD  PTR  CS:OLD16H  ;转原 16H 号中断处理程序
        ;
PKEY:   PUSH    AX                          ;0 号和 10H 号功能
PKEY1:  POP     AX
        PUSH    AX
        PUSHF
        CALL    DWORD PTR CS:OLD16H     ;调原 16H 号中断处理程序
        CMP     AL,3
```

```
        JZ      PKEY1                       ;如是 Ctrl+C,则"吃掉"
        CMP     AX,0300H
        JZ      PKEY1                       ;如是 Ctrl+2,也"吃掉"
        ADD     SP,2                        ;堆栈平衡
        IRET
        ;
PKEY2:  PUSH    AX                          ;01H 和 11H 号功能
PKEY3:  POP     AX
        PUSH    AX
        PUSHF
        CALL    DWORD PTR CS:OLD16H         ;调原 16H 号中断处理程序
        JZ      PKEY6                       ;没有字符,转
        CMP     AL,3
        JZ      PKEY4                       ;有字符 Ctrl+2,转
        CMP     AX,0300H
        JNZ     PKEY5
        ;
PKEY4:  XOR     AH,AH                       ;如遇字符 Ctrl+C 和 Ctrl+2 键
        PUSHF                               ;则读出它,然后继续
        CALL    DWORD PTR CS:OLD16H
        JMP     PKEY3
        ;
PKEY5:  ADD     SP,2                        ;有字符时返回,平衡堆栈
        CMP     AX,0300H                    ;使 ZF 为 0
        RET     2                           ;带标志返回
        ;
PKEY6:  ADD     SP,2                        ;无字符时返回,平衡堆栈
        CMP     AX,AX                       ;置 ZF 为 1
        RET     2                           ;带标志返回
NEWKEY  ENDP                                ;新的 1BH 号中断处理程序
NEW1BH: IRET                                ;中断返回
;------------------------
;主程序
START:  PUSH    CS
        POP     DS
        MOV     AX,3516H
        INT     21H                         ;取 16H 号中断向量保存
        MOV     WORD PTR OLD16H,BX
        MOV     WORD PTR OLD16H+2,ES
        ;
        MOV     AX,351BH                    ;取 1B 号中断向量保存
        INT     21H
```

```
        MOV     WORD PTR OLD1BH,BX
        MOV     WORD PTR OLD1BH+2,ES
;
        MOV     DX,OFFSET NEW16H        ;置新的 16H 号中断向量
        MOV     AX,2516H
        INT     21H
        MOV     DX,OFFSET NEW1BH        ;置新的 1BH 号中断向量
        MOV     AX,251BH
        INT     21H
;
CONT:   ……                              ;同 T6-6.ASM
;
XIT:    LDS     DX,OLD1BH
        MOV     AX,251BH                ;恢复 1BH 号原中断向量
        INT     21H
        LDS     DX,CS:OLD16H
        MOV     AX,2516H                ;恢复 16H 号原中断向量
        INT     21H
        MOV     AH,4CH                  ;程序正常终止
        INT     21H
CSEG    ENDS
        END     START
```

6.4 TSR 程序设计举例

TSR(Terminate and Stay Resident)意为结束并驻留。TSR 程序是一种特殊的 DOS 应用程序，不同于结束即退出的一般 DOS 应用程序。TSR 程序装入内存并初次运行后，程序的大部分仍驻留内存，被某种条件激活后又投入运行。它能及时地处理许多暂驻程序不能处理的事件，并可为单任务操作系统 DOS 增添一定的多任务处理能力。

编写 TSR 程序时要考虑许多方面的问题，诸如 TSR 的激活条件和资源使用的冲突等，对这些问题的讨论超出了本教程的范围，所以本节仅举两个例子说明如何编写简单的 TSR 程序。

6.4.1 驻留的时钟显示程序

通常 TSR 程序由驻留内存部分和初始化部分组成。把 TSR 程序装入内存时，初次运行的是初始化部分。初始化程序的主要功能是，对驻留部分完成必要的初始化工作；使驻留部分保留在内存中。

下面的时钟显示程序是对在 5.5.2 节中介绍的时钟显示程序的扩充。初始化后驻留在内存的程序由时钟中断激活。

```
;程序名：T6-7.ASM
```

```
;功  能：在内存中驻留显示时钟的程序
;1CH 号中断处理程序使用的常数说明与 T5-5.ASM 对应部分相同
CSEG SEGMENT
      ASSUME  CS:CSEG,  DS:CSEG
;1CH 号中断处理程序使用的变量说明与 T5-5.ASM 对应部分相同
......
OLD1CH     DD  ?                                   ;保存原中断向量的变量
;1CH 号中断处理程序代码
NEW1CH: CMP   CS:COUNT,0                           ;计数为 0?
        JZ    NEXT                                 ;为 0 转,显示时钟
        DEC   CS:COUNT                             ;计数减 1,转原 1CH 中断处理程序
        JMP   DWORD  PTR  CS:OLD1CH
        ;
NEXT:   MOV   CS:COUNT,COUNT_VAL                   ;重置计数值
        STI                                        ;开中断
        ;
        PUSH  DS
        ......                                     ;(与 T5-5.ASM 对应部分相同)
        POP   DS
        JMP   DWORD  PTR  CS:OLD1CH                ;代替 IRET
        ;
;子程序 GET_T
        ......                                     ;(与 T5-5.ASM 对应部分相同)
;子程序 TTASC
        ......                                     ;(与 T5-5.ASM 对应部分相同)
;----------------------------------
;初始化部分代码和变量
START:  PUSH  CS
        POP   DS
        MOV   AX,351CH                             ;(与 T5-5.ASM 对应部分相同)
        INT   21H
        MOV   WORD  PTR  OLD1CH,BX
        MOV   WORD  PTR  OLD1CH+2,ES
        MOV   DX,OFFSET  NEW1CH
        MOV   AX,251CH
        INT   21H
                                                   ;计算驻留节数并驻留退出
        MOV   DX,OFFSET  START                     ;取欲驻留部分代码和数据的字节数
        ADD   DX,15                                ;考虑字节数不是 16 倍数的情况
        MOV   CL,4
        SHR   DX,CL                                ;转换成节数
        ADD   DX,10H                               ;加上 PSP 的长度
```

```
        MOV   AH,31H
        INT   21H                     ;结束并驻留
CSEG    ENDS
        END   START
```

把上面的程序与T5-5.ASM相比较可知，其中的1CH号中断处理程序基本相同；初始化部分包含了驻留退出的代码，把1CH号中断处理程序驻留在内存中，此外还把保存原1CH号中断向量的双字变量OLD1C移到驻留区。

通过DOS的31H号功能调用进行驻留退出。该功能调用的主要入口参数是含在DX中的驻留节数(1节等于16字节)，驻留的内容从程序段前缀开始计算，所以在计算驻留节数时，除了计算要驻留的数据和代码的长度外，还需要加上PSP的10H节。

把DOS的31H号功能调用与4CH号功能调用相比，所不同的是它在交出控制权时没有全部交出占用的内存资源，而是根据要求(由入口参数规定)保留了部分。

6.4.2 热键激活的TSR程序

有多种方式或方法激活驻留的程序，键盘激活是常见的一种方法。例如可驻留一个保存屏幕画面的程序，然后当屏幕上出现需要保存的画面时，按一个约定的激活键来激活它。这样的键称为热键。

下面的程序T6-8.ASM是一个简单的热键激活TSR程序的例子。热键设定为CTRL+F8，在驻留后，每按一次CTRL+F8键，就在屏幕约定的位置显示一字符串。

```
;程序名：T6-8.ASM
;功  能：简单的热键激活TSR程序
;常量说明
BUFF_HEAD     =   1AH                ;键盘缓冲区头指针保存单元偏移
BUFF_TAIL     =   1CH                ;键盘缓冲区尾指针保存单元偏移
BUFF_START    =   1EH                ;键盘缓冲区开始偏移
BUFF_END      =   3EH                ;键盘缓冲区结束偏移
CTRL_F8       =   6500H              ;激活键扫描码
ROW           =   3                  ;行号
COLUMN        =   0                  ;列号
PAGEN         =   0                  ;显示页号
;代码
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
;新键盘中断处理程序
OLD9H DD ?                           ;原9号中断向量保存单元
MESS  DB   'Hello! '                 ;显示信息
MESSLEN   EQU   $ - MESS
NEW9H: PUSHF
       CALL    CS:OLD9H              ;调用原中断处理程序
```

```
        ;
        STI                             ;开中断
        PUSH    DS
        PUSH    AX                      ;保护现场
        PUSH    BX
        ;
        MOV     AX,40H
        MOV     DS,AX
        MOV     BX,DS:[BUFF_HEAD]
        CMP     BX,DS:[BUFF_TAIL]       ;键盘缓冲区空?
        JZ      IOVER                   ;是,结束
        MOV     AX,DS:[BX]              ;取所按键的代码
        CMP     AX,CTRL_F8              ;是否为激活键?
        JZ      YES                     ;是,转处理
        ;
IOVER:  POP     BX                      ;结束处理
        POP     AX                      ;恢复现场
        POP     DS
        IRET                            ;中断返回
        ;
YES:    INC     BX
        INC     BX                      ;调整键盘缓冲区头指针(取走激活键)
        CMP     BX,BUFF_END             ;指针到缓冲区末?
        JNZ     YES1                    ;否,转
        MOV     BX,BUFF_START           ;是,指向头
YES1:   MOV     DS:[BUFF_HEAD],BX       ;保存
        ;
        PUSH    CX
        PUSH    DX                      ;再保护部分现场
        PUSH    BP
        PUSH    ES
        MOV     AX,CS
        MOV     ES,AX                   ;数据段同代码段
        MOV     BP,OFFSET MESS
        MOV     CX,MESSLEN
        MOV     DH,ROW
        MOV     DL,COLUMN
        MOV     BH,PAGEN
        MOV     BL,07H
        MOV     AL,0                    ;显示后不移动光标,串中不含属性
        MOV     AH,13H
        INT     10H
```

```
        POP     ES
        POP     BP                          ;恢复部分现场
        POP     DX
        POP     CX
        JMP     IOVER                       ;转结束
;------------------------
;初始化代码
INIT:   PUSH    CS
        POP     DS
        MOV     AX,3509H                    ;取 9 号中断向量
        INT     21H                         ;并保存
        MOV     WORD  PTR  OLD9H,BX
        MOV     WORD  PTR  OLD9H+2,ES
        ;
        MOV     DX,OFFSET  NEW9H            ;置新的 9 号中断向量
        MOV     AX,2509H
        INT     21H
        ;
        MOV     DX,OFFSET  INIT  +15
        MOV     CL,4                        ;计算驻留节数
        SHR     DX,CL
        ADD     DX,10H                      ;加上 PSP 的节数
        MOV     AL,0
        MOV     AH,31H                      ;驻留退出
        INT     21H
CSEG    ENDS
        END     INIT
```

上面程序的初始化部分先保存了 9 号中断向量(原键盘中断处理程序的入口地址)，然后设置新的 9 号中断向量，使其指向新的键盘中断处理程序，最后驻留结束。这样每当按键，就会运行新的键盘中断处理程序。新的键盘中断处理程序先调用老的键盘中断处理程序完成按键处理工作，然后通过检查键盘缓冲区，判断是否按了约定的热键 CTRL＋F8，当判断出按了 CTRL＋F8 后，就显示一预定的提示信息。

6.5 习　　题

题 6.1　请说明标志 DF 的作用。如何设置 DF？

题 6.2　在 8086/8088 的指令集中，哪些指令属于两个操作数都可以是存储器操作数这种例外情况？

题 6.3　请说明指令“LODSB”与如下程序片段的异同：

```
MOV     AL,[SI]
```

```
        INC     SI
```

题 6.4 请说明指令“STSOW”与如下程序片段的异同：

```
        MOV     [DI],AX
        ADD     DI,2
```

题 6.5 请说明指令“SCASB”与如下程序片段的异同：

```
        CMP     AL,ES:[DI]
        INC     DI
```

题 6.6 请写一个程序片段代替指令“REP MOVSW”。

题 6.7 请写一个程序片段代替指令“REPNZ CMPSB”。

题 6.8 请考察重复前缀 REP 与重复前缀 REPZ/REPE 的机器码。

题 6.9 编写一个实现数据块移动的近过程。通过寄存器传递参数。

题 6.10 编写一个实现字符串拷贝的近过程。通过堆栈传递作为参数的源和目标字符串首地址的偏移。

题 6.11 编写一个把字符串中的小写字母转换为大写(字符串以 0 结尾)的远过程。通过堆栈传递包括段值和偏移在内的字符串首地址。

题 6.12 改写 6.1.3 中的例 1 和例 2,通过堆栈传递入口参数。

题 6.13 编写一个把字符串 2 插入到字符串 1 指定位置的远过程。

题 6.14 编写一个截取字符串某子串的近过程。

题 6.15 编写一个去掉字符串前导空格的近过程。编写一个去掉字符串尾部空格的近过程。编写一个去掉字符串前导空格和尾部空格的近过程。

题 6.16 利用字符串操作指令写一个清屏的过程。只考虑字符显示方式。

题 6.17 十进制数算术运算指令调整指令有何用途？它们调整的主要依据是什么？

题 6.18 请编写一个小型的可实现整数四则运算的实用程序。

题 6.19 请编写一个类似于 DOS 内部命令 TYPE 的小型实用程序。通过命令行选项,可以指示是否进行分屏显示。

题 6.20 请编写一个类似于 DOS 内部命令 COPY 的小型实用程序。

题 6.21 请编写一个类似于 DOS 内部命令 TYPE 的小型实用程序。但不能利用 Ctrl+C 键等终止程序。

题 6.22 通过编写一个 TRS 程序的方法,稍稍改变 BIOS 的键盘中断处理程序,使得 DOS 平台上的应用程序无法获得大写字母 A。当按 A 键时,应用程序只能获得小写字母 a。

题 6.23 通过编写一个 TRS 程序的方法,改变 BIOS 的键盘 I/O 程序,使得 DOS 平台上的应用程序无法获得大写字母。当按大写字母键时,应用程序只能获得对应的小写字母。

题 6.24 通过编写一个 TRS 程序的方法,改变 BIOS 的显示 I/O 程序,使得通过 BIOS 显示 I/O 程序实现显示的应用程序无法显示大写字母。当显示大写字母时,只显示出对应的小写字母。

题 6.25 通过编写一个 TRS 程序的方法,改变 BIOS 的打印 I/O 程序,使得小写字

母的输出变为大写字母的输出，当打印输出 ESC＋A 时，恢复正常。

题 6.26 通过编写一个 TRS 程序的方法，使得屏幕上不出现小写字母。所有欲显示的小写字母均被替换成对应的大写字母。只考虑字符显示方式。

题 6.27 通过编写一个 TRS 程序的方法，当按下 Ctrl＋F1 键时，把屏幕上的显示信息保存到文件 SCREEN 中。

题 6.28 通过编写一个 TRS 程序的方法，使得系统报时。

第 7 章　高级汇编语言技术

尽管汇编语言作为低级语言不可能象高级语言那样具有丰富的数据类型和方便灵活的表达方式，但汇编语言仍力求提供这方面的功能。本章介绍汇编语言的一些高级技术：结构、宏和条件语句，利用它们可编写出更具适应性的汇编语言源程序。

7.1　结构和记录

为了使程序员能更方便、更有效地对数据进行组织和描述，宏汇编语言除了提供定义简单数据变量的伪指令(如 DB 和 DW 等)外，还提供了用于说明复杂数据类型的伪指令，利用这些伪指令能够说明复杂的数据类型，从而定义复杂的数据变量。本节描述结构和记录并说明如何使用它们。

7.1.1　结构

1. 结构类型的说明

在描述结构型数据或使用结构型变量之前，需要说明结构类型。用伪指令 STRUC 和 ENDS 把一系列数据定义语句括起来就说明了一个结构类型，一般格式如下：

```
结构名    STRUC
          数据定义语句序列
结构名    ENDS
```

例如，下列语句说明了一个名为 PERSON 的结构类型：

```
PERSON      STRUC
ID          DW      ?
SCORE       DB      0
PNAME       DB      'ABCDEFGH'
PERSON      ENDS
```

组成结构的变量称为结构的字段，相应的变量名称为字段名。一个结构中可以含有任意数目的字段，并且各字段可以有不同的长度(基本单位是字节)，还可以独立地存取。结构中的字段也可以没有字段名。

结构中的字段名代表了从结构的开始到相应字段的偏移。在上面定义的结构 PERSON 中，字段 ID、SCORE 和 FNAME 分别有偏移值 0、2 和 3。

在说明结构类型时，可以给字段赋初值，也可以不赋初值。在上面定义的结构 PERSON 中，给字段 SCORE 赋缺省初值 0，没有给字段 ID 赋初值。如果字段是一个字符串，那么要确保其初值有足够的长度以适应可能最长的字符串。

再如，下列语句说明了一个名为 MESST 的结构类型：

```
MESST       STRUC
MBUFF       DB  100  DUP (?)
CRLF        DB  0DH,0AH
ENDMARK     DB  24H
MESST       ENDS
```

结构 MESST 中的字段 MBUFF 和 CRLF 均含有多个值，字段 MBUFF、CRLF 和 ENDMARK 分别有偏移值 0、100 和 102。

在说明结构类型时，结构名必须是唯一的，各字段名也应该是唯一的。

注意，在说明结构类型时不进行任何存储分配，只有在定义结构变量时才进行存储分配。这与高级语言中的数据类型定义相似。

还请注意，标记一个结构类型结束的伪指令与标记一个段结束的伪指令有相同的助记符 ENDS，汇编程序通过上下文理解 ENDS 的含义，所以要确保每一 SEGMENT 伪指令和每一 STRUC 伪指令有各自对应的 ENDS 伪指令。

2. 结构变量的定义

在说明了结构类型后，就可定义相应的结构变量。结构变量定义的一般格式如下：

```
[变量名]    结构名    <[字段值表]>
```

其中，变量名就是当前定义的结构变量的名称，结构变量名也可以省略，如果省略，那么就不能直接通过符号名访问该结构变量。结构名是在说明结构类型时所用的名字。字段值表用来给结构变量的各字段赋初始值，其中各字段值的排列顺序及类型应与结构定义时的各字段相一致，中间以逗号分隔。如果某个字段采用在说明结构时所给定的缺省初值，那么可简单地用逗号表示；如果结构变量的所有字段均如此，那么可省去字段值表，但仍必须保留一对尖括号。

例如，设已说明了上述结构 PERSON，那么可定义如下结构变量：

```
STUDENT1    PERSON    <103,88,'WANG'>      ;三个字段都重赋初值
STUDENT2    PERSON    <104, ,'LIMING'>     ;字段 SCORE 仍用缺省初值
STUDENT     PERSON    < >                  ;三个字段均用缺省初值
            PERSON    99  DUP (<>)         ;定义 99 个结构变量，初值不变
```

注意，对宏汇编程序 MASM 而言，如果某个字段有多值，那么在定义结构变量时，就不能给该字段重赋初值。如上面说明的结构 MESST，不能给其 MBUFF 字段和 CRLF 字段重赋初值。下面定义结构变量的语句是正确的：

```
MESS1     MESST < >
MESS2     MESST <, , 0>
```

3. 结构变量及其字段的访问

通过结构变量名可直接存取结构变量。若要存取结构变量中的某一字段，则可采用如下形式：

结构变量名.结构字段名

在上述形式中，结构变量名与结构字段名之间用点号分隔，并且结构字段名所代表的字段必须是对应结构所具有的字段。这种形式表示的变量的地址偏移值是结构变量地址（起始地址）的偏移值与相应字段偏移值之和。

下面的程序片段说明了对结构字段的访问：

```
          ......
DATE      STRUC                                  ;说明结构类型
YEAR      DW            ?
MONTH     DB            ?
DAY       DB            ?
DATE      ENDS
          ......
YESTERDAY               DATE <1995,7,17>         ;定义结构变量
TODAY                   DATE <1995,7,18>
TOMORROW                DATE <1995,7,19>
          ......
          MOV           AL,YESTERDAY.DAY         ;访问结构变量
          MOV           AH,TODAY.MONTH
          MOV           TOMORROW.YEAR,DX
          ......
```

上述形式所对应的寻址方式是直接寻址。另一种存取结构变量中某一字段的方法是把结构变量地址的偏移先存入某个基址或变址寄存器，然后用“[寄存器名]”代替结构变量名。这种方法所对应的寻址方式是相对基址或变址寻址。例如：

```
          ......
          MOV           BX,OFFSET  YESTERDAY
          MOV           AL,[BX].MONTH
          ......
```

例 1：数据文件 SCORE.DAT 中依次存放着 30 个学生的成绩记录，文件（成绩）记录具有如下字段：

学号	整数	2 字节
姓名	字符串	8 字节
语文成绩	整数	1 字节
数学成绩	整数	1 字节
外语成绩	整数	1 字节

写一个程序计算三门课程的总分，把学号和总分依次写到文件 SCORE.SUM 中。SCORE.SUM 文件的记录有两个字段，第一个字段是学号，第二个字段是总分（用 2 字节表示）。

实现流程是：(1)打开文件 SCORE.DAT；(2)循环处理每个学生的成绩，把学号和总

分依次存入预定义的缓冲区(称为总分表)中;(3)关闭文件 SCORE. DAT;(4)建立文件 SCORE. SUM;(5)把缓冲区中的内容写入文件 SCORE. SUM;(6)关闭文件 SCORE. SUM。

为了突出结构的应用,下面列出的程序没有考虑磁盘文件读写出错等情况。

```
;程序名:T7-1.ASM
;功  能:(略)
;常量定义
COUNT   =    30                     ;设学生数为 30
;对应原始成绩的结构 SCORE 的定义
SCORE   STRUC
NO      DW   ?                      ;学号
SNAME   DB   8 DUP (' ')            ;姓名
CHN     DB   0                      ;语文成绩
MATH    DB   0                      ;数学成绩
ENG     DB   0                      ;外语成绩
SCORE   ENDS
;对应学号和总分的结构 ITEM 的定义
ITEM    STRUC
NOS     DW     0                    ;学号
SUM     DW     0                    ;总分
ITEM    ENDS
;数据段
DSEG      SEGMENT
BUFFER    SCORE < >                          ;存放原始成绩的缓冲区
STABLE    ITEM   COUNT DUP ( < > )           ;预留总分表
FNAME1    DB     'SCORE.DAT',0               ;文件名
FNAME2    DB     'SCORE.SUM',0
DSEG      ENDS
;代码段
CSEG    SEGMENT
        ASSUME CS : CSEG, DS : DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     DX,OFFSET FNAME1
        MOV     AX,3D00H                ;打开文件 SCORE.DAT
        INT     21H
        MOV     BX,AX
        MOV     DI,COUNT                ;置循环计数器初值
        MOV     SI,OFFSET STABLE        ;置学号总分缓冲区指针初值
READ:   MOV     DX,OFFSET BUFFER        ;读一个学生的原始成绩
        MOV     CX,TYPE SCORE
```

```
        MOV     AH,3FH
        INT     21H
        MOV     AL,BUFFER.CHN           ;统计总分
        XOR     AH,AH
        ADD     AL,BUFFER.MATH
        ADC     AH,0
        ADD     AL,BUFFER.ENG
        ADC     AH,0
        MOV     [SI].SUM,AX             ;把总分保存到总分表的当前项
        MOV     AX,BUFFER.NO
        MOV     [SI].NOS,AX             ;把学号保存到总分表的当前项
        ADD     SI,TYPE ITEM            ;调整当前总分表当前项
        DEC     DI                      ;循环计数控制
        JNZ     READ
        MOV     AH,3EH                  ;关闭文件 SCORE.DAT
        INT     21H
        MOV     DX,OFFSET FNAME2
        MOV     CX,0
        MOV     AH,3CH                  ;建立文件 SCORE.SUM
        INT     21H
        MOV     BX,AX
        MOV     DX,OFFSET STABLE        ;写出学号和总分
        MOV     CX,(TYPE ITEM) * COUNT
        MOV     AH,40H
        INT     21H
        MOV     AH,3EH                  ;关闭文件 SCORE.SUM
        INT     21H
        MOV     AX,4C00H
        INT     21H
CSEG    ENDS
        END     START
```

在程序 T7-1. ASM 中,我们定义两个结构 SCORE 和 ITEM,用来描述文件记录的字段组成。利用 SCORE 定义了存放一个原始成绩记录的缓冲区,这样可方便地引用各字段。还利用 ITEM 定义了一张总分表,然后借助指向总分表当前项的指针方便地访问当前项的学号和总分字段。此外,程序 T7-1. ASM 还使用了 TYPE 得到结构的字节数。

例 2:写一个求字符串长度的子程序。子程序的调用说明如下:

```
;子程序名: STRLEN
;功    能: 测字符串长度
;入口参数: 字符串首地址的段值和偏移在堆栈顶
;出口参数: AX=字符串长度
```

```
;说    明：(1)字符串以 0 结尾；字符串长度不包括结尾标志。
;           (2)本过程是一个远过程。
```

为了方便地表达堆栈参数，先定义如下结构：

```
PARM    STRUC
BPREG   DW    ?      ;对应 BP 寄存器保存单元
RETADR  DD    ?      ;对应返回地址
STROFF  DW    ?      ;对应入口参数中的偏移
STRSEG  DW    ?      ;对应入口参数中的段值
PARM    ENDS
```

只定义上述结构 PARM 类型，但不定义结构变量。当子程序为了利用寄存器 BP 访问堆栈中的参数，而把寄存器 BP 压入堆栈后，可以认为一个这样的结构变量出现在当时的堆栈顶端。然后使寄存器 BP 指向此结构变量，于是就能方便地表达要存取的位于堆栈中的参数。子程序源代码如下：

```
STRLEN   PROC  FAR
         PUSH  BP
         MOV   BP,SP
         PUSH  DS
         PUSH  SI
         MOV   DS,[BP].STRSEG   ;取字符串首地址的段值
         MOV   SI,[BP].STROFF   ;取字符串首地址的偏移
         XOR   AL,AL
STRLEN1: CMP   BYTE  PTR  [SI],AL
         JZ    STRLEN2
         INC   SI
         JMP   STRLEN1
STRLEN2: MOV   AX,SI
         SUB   AX,[BP].STROFF
         POP   SI
         POP   DS
         POP   BP
         RET
STRLEN   ENDP
```

该子程序与 4.2.3 节中的例 5 功能相同，实现算法也相同，所不同的是该子程序是远过程。请比较利用结构类型和结构字段前后代码的易读性和可修改性。

7.1.2 记录

记录类型为按二进制位存取数据或信息提供了方便。

1. 记录类型的说明

在描述记录型数据或使用记录型变量之前，需要说明记录类型。伪指令 RECORD 用

于说明记录类型，一般格式如下：

记录名　RECORD　字段　[,字段...]

记录名标识说明的记录类型；字段表示构成记录的字段的名字、宽度和缺省初值。每一字段的格式如下：

字段名:宽度 [=表达式]

字段名是记录中字段的名字。宽度表示相应的字段所占的位数，宽度必须是常数，宽度最大为 16 位。表达式的值将作为相应字段的缺省初值，如果缺省初值相对于它的宽度太大，则汇编时将产生错误提示信息。如果某个字段没有缺省初值，那么缺省初值被置为 0。

一个记录可以含有多个字段，字段间用逗号分隔。但在一般情况下各字段的宽度之和不超过 16。例如：

```
COLOR  RECORD  BLINK : 1, BACK : 3, INTENSE : 1, FORE : 3
```

上述记录类型 COLOR 含四个字段(BLINK、BACK、INTENSE 和 FORE)，各字段均没有赋缺省初值，它们的宽度分别是 1、3、1 和 3。这四个字段所占总宽度正好是 8 位，所以也称为字节记录类型。这四个字段的具体意义请参见图 5.14。

注意，在说明记录类型时，不实际分配存储单元。

如果一个记录中所有说明字段的总宽度大于 8，那么汇编程序会给对应的记录变量分配两字节，否则仅给对应的记录变量分配一字节。第一个字段放在记录左边的较高有效位，随后说明的字段放在右边的一些后续位上，如果说明的字段总宽度不正好是 8 位或 16 位，那么向右对齐，记录高端未说明的位置为 0。例如：

```
ABCD   RECORD  AA : 5=12, BB : 3=6, CC : 4=3
```

上述记录类型 ABCD 含三个字段，这三个字段所占各位如图 7.1 所示。

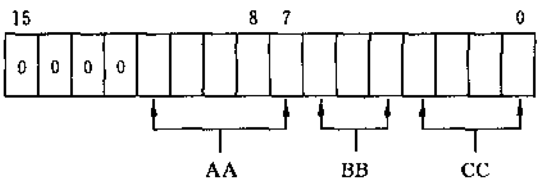


图 7.1　记录类型 ABCD 的各字段

2. 记录变量的定义

记录变量是其位分成一或多个字段的字节或字变量。定义记录变量的一般格式如下：

[变量名]　记录名　<[字段值表]>

其中，变量名就是当前定义的记录变量的名称，记录变量名可以省略，如果省略，那么就不能直接通过符号名访问该记录变量。记录名是在说明记录类型时所用的名字。字段值表用来给记录变量的各字段赋初始值，各字段值的排列顺序及大小应与记录说明时的

各字段相一致，中间以逗号分隔。如果某个字段采用在说明记录时所给定的缺省初值，那么可简单地用逗号表示；如果记录变量的所有字段均如此，那么可省去字段值表，但仍必须保留一对尖括号。

例如：设已定义了上述记录类型 COLOR，那么可定义如下结构变量：

```
WARNING     COLOR <1,0,1,4>          ;该字节的值是 8CH
            COLOR <,3,,110B>         ;该字节的值是 36H
COLORLST    COLOR 32  DUP (<>)       ;32 个字节
```

当字段有 7 位宽时，可定义为一字符。

注意，如果字段值太大，则有汇编错误提示信息。

3. 记录专用操作符

操作符 WIDTH 和 MASK 仅与记录一起使用，得到已说明记录的不同方面的常数值。

(1) 操作符 WIDTH

操作符 WIDTH 返回记录或记录中字段的以位为单位的宽度。一般格式如下：

```
     WIDTH    记录名
或者 WIDTH    记录字段名
```

设记录 COLOR 如上面的说明，那么：

```
SUB    AL,WIDTH COLOR        ;SUB AL,8
MOV    DH,WIDTH BACK         ;MOV DH,3
ADD    BH,WIDTH INTENSE      ;ADD BH,1
```

(2) 操作符 MASK

一般格式如下：

```
     MASK    记录名
或者 MASK    记录字段名
```

操作符 MASK 返回踊个 8 位或 16 位二进制数，这个二进制数中相应于指定字段的各位为 1，其余各位为 0。如果记录是字节记录类型，那么就是一个 8 位二进制数，如果记录是字记录类型，那么就是一个 16 位二进制数。设记录 COLOR 和 ABCD 如上面的说明，那么：

```
MOV    AL,MASK   BLINK     ;MOV    AL,10000000B
OR     AH,MASK   FORE      ;OR     AH,00000111B
AND    DX,MASK   ABCD      ;AND    DX,0FFFH
```

(3) 记录字段

记录字段名作为一个特殊的操作符，它不带操作数，直接返回该字段移到所在记录的最右端所需移动的位数。

设记录 COLOR 如上面的说明，那么：

```
MOV     CL,BLINK                 ;MOV CL,7
MOV     CL,INTENSE               ;MOV CL,3
```

4. 记录及其字段的访问

由于 8086/8088 没有位操作指令，记录类型和记录操作符能够也只能够提供访问记录中字段的便利。

下面的程序片段说明如何访问记录及其字段：

```
        ......
COLOR RECORD BLINK : 1, BACK : 3, INTENSE : 1, FORE : 3   ;说明记录类型
        ......
CHAR    DB      'A'
ATTR    COLOR   <0,0,1,7>                       ;定义记录变量
        ......
        MOV     BP,1 SHL  WIDTH  BACK           ;置循环计数器
NEXT:   MOV     AH,9                            ;在当前光标位置显示字符
        MOV     BH,0
        MOV     AL,CHAR
        MOV     BL,ATTR
        MOV     CX,1
        INT     10H
        MOV     AL,ATTR                         ;取显示属性(记录变量)
        MOV     AH,AL
        AND     AL,NOT MASK BACK                ;析出除背景的其他位
        MOV     CL,BACK
        SHR     AH,CL                           ;把背景字段移至右端
        INC     AH                              ;调整背景色
        SHL     AH,CL                           ;再向左移到原位
        AND     AH,MASK BACK                    ;屏蔽除背景位的其他位
        OR      AH,AL                           ;和其他位原值合并
        MOV     ATTR,AH                         ;保存属性
        MOV     AH,0
        INT     16H
        DEC     BP
        JNZ     NEXT
        ......
```

该程序片段在当前光标位置处循环显示各种不同背景的显示效果，按一次键，则换一种背景色。

7.2 宏

宏是宏汇编语言的主要特征之一。在汇编语言源程序中，若某程序片段需要多次使

用，为了避免重复书写，那么可把它定义为一条宏指令。在写源程序时，程序员用宏指令来表示程序片段；在汇编时，汇编程序用对应的程序片段代替宏指令。

7.2.1 宏指令的定义和使用

宏指令在使用之前要先定义。宏定义的一般格式如下：

```
宏指令名    MACRO    [形式参数表]
            .
            .
            ENDM
```

其中，MACRO 和 ENDM 是一对伪指令，在宏定义中，它们必须成对出现，表示宏定义的开始和宏定义的结束。MACRO 和 ENDM 之间的内容称为宏定义体，可以是由指令、伪指令和宏指令构成的程序片段。宏指令名由用户指定，适用一般标号命名规则。可选的形式参数表可由若干参数组成，各形式参数间用逗号分隔。

例如：我们把将 AL 寄存器内的低 4 位转换为对应十六进制数 ASCII 码的程序片段定义为一个宏：

```
HTOASC    AMCRO
          AND   AL,0FH
          ADD   AL,90H
          DAA
          ADC   AL,40H
          DAA
          ENDM
```

再如，我们把通过 DOS 的 1 功能调用从键盘读一个字符的程序片段定义为一个宏：

```
GETTCH    MACRO
          MOV   AH,1
          INT   21H
          ENDM
```

在定义宏指令后，就可使用宏指令来表示对应的程序片段，这称为宏调用。宏调用的一般格式如下：

```
宏指令名    [实参数表]
```

其中，实参数表中的实参数应该与宏定义时的形式参数表中的形式参数相对应。

例如，下面的程序片段调用了刚定义的两个宏：

```
          ......
          GETCH
          MOV   AH,AL
          SHR   AL,1
          SHR   AL,1
```

```
        SHR    AL,1
        SHR    AL,1
        HTOASC
        XCHG   AH,AL
        HTOASC
        ......
```

在对源程序汇编时，汇编程序把源程序中的宏指令替换成对应的宏定义体，这称为宏展开或宏扩展。上述程序片段在汇编时得到的指令如下：

```
        ......
1       MOV    AH,1
1       INT    21H
        MOV    AH,AL
        SHR    AL,1
        SHR    AL,1
        SHR    AL,1
        SHR    AL,1
1       AND    AL,0FH
1       ADD    AL,90H
1       DAA
1       ADC    AL,40H
1       DAA
        XCHG   AH,AL
1       AND    AL,0FH
1       ADD    AL,90H
1       DAA
1       ADC    AL,40H
1       DAA
        ......
```

注意，宏展开所得指令的行首标有符号"1"，以示区别。

7.2.2 宏指令的用途

1. 缩短源代码

若在源程序中要多次使用到某个程序片段，那么就可以把此程序片段定义为一条宏指令。此后，在需要这个程序片段之处安排一条对应的宏指令就行了，由汇编程序在汇编时产生对应的代码。这不仅能有效地缩短源代码，而且能减少编辑汇编语言源程序过程中的错误。

例如，我们把使光标另起一行的程序片段写成如下的一个宏：

```
CRLF    MACRO
        XOR     BH,BH
```

```
        MOV   AH,14
        MOV   AL,0DH
        INT   10H
        MOV   AL,0AH
        INT   10H
        ENDM
```

2. 扩充指令集

CPU 的指令集是确定的，但利用宏能在汇编语言中在形式上对指令集进行扩充。扩充后的指令集是机器指令集与宏指令集的并集。这不仅能方便源程序的编写，而且便于阅读理解源程序。

在写子程序和中断处理程序时，为了保护现场常常需要把 8 个通用寄存器全部压入堆栈，但 8086/8088 指令集中没有把 8 个通用寄存器全部压栈的指令。为此，我们可定义一条宏指令 PUSHA，由它实现把 8 个通用寄存器全部压入堆栈的功能：

```
PUSHA   MACRO
        PUSH  AX
        PUSH  CX
        PUSH  DX
        PUSH  BX
        PUSH  SP
        PUSH  BP
        PUSH  SI
        PUSH  DI
        ENDM
```

在定义了上述宏指令 PUSHA 后，每当需要把 8 个通用寄存器全部压栈时，就可以使用 PUSHA 指令了。与宏指令 PUSHA 相对应，我们还可定义宏指令 POPA 实现把由 PUSHA 压栈的 8 个通用寄存器依次退栈的功能。

3. 改变某些指令助记符的意义

宏指令名可以与指令助记符或伪操作名相同，在这种情况下，宏指令的优先级最高，而同名的指令或伪操作就失效了。利用宏指令的这一特点，可以改变指令助记符的意义。

例如，在定义如下宏指令后，助记符 LODSB 所表示指令的意义就变化了：

```
LODSB   MACRO
        MOV   AH,[SI]
        INC   SI
        ENDM
```

汇编程序 TASM 在遇到上述宏定义时，会发出一条警告信息；而汇编程序 MASM 则不给出任何警告信息。

7.2.3 宏指令中参数的使用

宏指令可以不带参数，如上面定义的宏指令 GETCH 和 PUSHA 等。但往往带参数的宏指令更具灵活性。下面介绍宏指令中参数的使用。

1. 宏指令的参数很灵活

(1) 宏指令的参数可以是常数、寄存器和存储单元，还可以是表达式。

例 1：在逻辑左移指令 SHL 的基础上定义一条宏指令 SHLN，它能实现指定次数的左移。

```
SHLN    MACRO REG,NUM
        PUSH  CX
        MOV   CL,NUM
        SHL   REG,CL
        POP   CX
        ENDM
```

此后，可有如下格式的各种宏调用：

```
        SHLN  BL,5
        SHLN  SI,9
        SHLN  AX,CL
```

在汇编时，宏指令"SHLN BL,5"扩展成如下的代码：

```
1       PUSH  CX
1       MOV   CL,5
1       SHL   BL,CL
1       POP   CX
```

(2) 宏指令的参数可以是操作码。

例 2：下面的宏指令 MANDM 有三个参数，第一个参数 OPR 作为操作符使用在宏体的指令中：

```
MANDM    MACRO  OPR,X,Y
        MOV   AX,X
        OPR   AX,Y
        MOV   X,AX
        ENDM
```

调用宏 MANDM 及其宏扩展如下所示：

```
MANDM    MOV  [BX],[SI]
1     MOV    AX,[BX]
1     MOV    AX,[SI]
1     MOV    [BX],AX
MANDM    ADD  [BX],ES:[1234H]
```

```
1   MOV    AX,[BX]
1   ADD    AX,ES:[1234H]
1   MOV    [BX],AX
```

2. 宏调用参数个数可以与定义时不一致

一般说来，宏调用时使用的实参个数应该与宏定义时的形参个数一致，但汇编程序并不要求它们必须相等。若实参个数多于形参个数，那么多余的实参被忽略；若实参的个数少于形参的个数，那么多余的形参用“空”代替。另外必须注意，宏展开后即实参取代形参后，所得的语句必须是有效的，否则汇编程序将会指示出错。

设已定义了上述的宏MANDM。宏调用“MANDM SUB,VAR1,VAR2,VAR3”展开后的语句如下：

```
1   MOV    AX,VAR1
1   SUB    AX,VAR2
1   MOV    VAR1,AX
```

显然，多余的实参VAR3被忽略。宏调用“MANDM MOV,VAR1”展开后的语句如下：

```
1   MOV    AX,VAR1
1   MOV    AX,
1   MOV    VAR1,AX
```

汇编程序将指示语句“MOV AX,”有错。另外，如果没有定义VAR1或VAR2，那么汇编程序也会给出相应的出错提示信息。

7.2.4 特殊的宏运算符

为了方便宏的定义和调用，汇编程序还支持如表7.1所示的特殊运算符，它们适用于宏的定义或调用中，还适用于重复块。

表7.1 特殊的宏运算符

运算符	使用格式	定　义
&	&参数	强迫替换运算符
< >	<字符串>	字符串原样传递运算符
!	!字符	文字字符运算符
%	%表达式	表达式运算符
;;	;;注释	宏注释

1. 强迫替换运算符&

在宏定义中，若参数在其它字符的紧前或紧后，或者参数出现在带引号的字符串中时，就必须使用该运算符，以区分参数。

例1：在下面定义的宏指令JUMP中，参数CON作为操作码的部分，请注意分隔符“&”的使用。

```
JUMP    MACRO    CON,LAB
```

```
        J&CON       LAB
        ENDM
```

调用宏 JUMP 及其宏扩展如下所示：

```
JUMP    NZ,HERE
1       JNZ       HERE
        JUMP      Z,THERE
1       JZ        THERE
```

例 2：下面定义的宏 MSGGEN 中，两个参数合并成标号，一个参数用在字符串中。

```
MSGGEN     MACRO LAB,NUM,XYZ
           LAB&NUM   DB 'HELLO MR. &XYZ',0DH,0AH,24H
           ENDM
```

调用宏 MSGGEN 及其宏扩展如下所示：

```
     MSGGEN    MSG,1,TAYLOR
1    MSG1      DB 'HELLO MR. TAYLOR',0DH,0AH,24H
```

2. 字符串原样传递运算符<>

字符串原样传递运算符是一对尖括号，在宏调用、重复块和条件汇编中，由它括起的内容作为一个字符串。在宏调用时，若实参包含逗号或空格等间隔符，则必须使用该运算符，以保证实参的完整。若实参是某个有特殊意义的字符，为了使它只表示字符本身，也可使用该运算符，把它括起来。

例 3：定义如下的宏：

```
DFMESS     MACRO       MESS
           DB '&MESS',0DH,0AH,0          ;;定义的字符串以 0 结尾
           ENDM
```

调用宏 DFMESS 及宏扩展如下所示：

```
     DFMESS <This is a example>
1    DB  'This is a example',0DH,0AH,0
```

如果不使用该运算符，则情况如下所示：

```
     DFMESS This is a example
1    DB 'This',0DH,0AH,0
```

3. 文字字符运算符!

该运算符使其后的一个字符只作为一般字符。在宏调用时，如果实参中含有一些特殊字符，为了使这些特殊字符作为一般字符来处理，那么就必须在其前冠上该运算符。

例 4：利用上述宏 DFMESS 定义字符串“Can not enter > 99”。由于字符串含有特殊符号“>”，为避免它与上述字符串原样传递运算符相混，则必须在其前冠该运算符。相应的宏调用及宏扩展如下所示：

```
        DFMESS <Can not enter ! > 99>
1       DB 'Can not enter > 99',0DH,0AH,0
```

4. 表达式运算符%

在宏调用时，使用该运算符能把其后表达式的结果作为实参替换，而非表达式自身。

例 5：调用上述宏 DFMESS，宏调用和宏扩展如下所示：

```
        DFMESS      %(12+3-4)          ;使用表达式运算符%
1           DB      '11',0DH,0AH,0
        DFMESS      12+3-4             ;未使用表达式运算符%
1           DB      '12+3-4',0DH,0AH,0
```

5. 宏注释

在宏定义中，如果注释以两个分号引导，那么宏扩展时该注释不出现。

7.2.5 宏与子程序的区别

采用宏和子程序这两种方法均能达到简化源程序的目的。但是，这两者之间存在质的不同。我们从宏调用与子程序调用之间的差异来说明这两者的区别。

(1) 宏调用是通过宏指令名进行的，在汇编时，由汇编程序把宏展开，有多少次宏调用，就有相应次的宏扩展，因此并不简化目标程序。子程序调用是在程序执行期间执行 CALL 指令进行的，子程序的代码只在目标程序中出现一次，所以目标程序也得到相应的简化。

(2) 宏调用时的参数由汇编程序通过实参替换形参的方式实现传递，所以参数很灵活。子程序调用时的参数须通过寄存器、堆栈或约定的内存单元传递。

(3) 宏调用是在汇编时完成，所以不需要额外的时间开销。子程序调用和子程序返回均需要时间，且还涉及堆栈。

总之，当程序片段不长，速度是主要矛盾时，通常采用宏指令的方法简化源程序；当程序片段较长，额外操作所附加的时间就不明显，而节约存储空间是主要矛盾时，通常采用子程序的方法简化源程序和目标程序。

7.2.6 与宏有关的伪指令

1. 局部变量说明伪指令 LOCAL

在宏定义体中可使用标号。例如：

```
HTOASC  MACRO
        AND  AL,0FH        ;;屏蔽高4位
        CMP  AL,9
        JBE  ISDECM        ;;不大于9时转
        ADD  AL,7
ISDECM: ADD  AL,30H
        ENDM
```

如果在程序中多次调用上述宏 HTOASC，汇编时将出现标号重复定义错误。原因是每次展开宏 HTOASC 都得到一个标号 ISDECM。为此，汇编程序提供伪指令 LOCAL，供程序员说明宏的局部标号。

伪指令 LOCAL 的一般格式如下：

```
LOCAL    标号表
```

标号表由标号构成，标号间用逗号分隔。汇编程序在每次展开宏时，总把由 LOCAL 伪指令说明的标号用唯一的符号(?? 0000 至?? FFFF)代替，从而避免标号重定义错误。

为了允许程序多次调用宏 HTOASC，应利用 LOCAL 说明标号 ISDECM，如下：

```
HTOASC   MACRO
         LOCAL  ISDECM
         AND    AL,0FH
         CMP    AL,9
         JBE    ISDECM
         ADD    AL,7
ISDECM:  ADD    AL,30H
         ENDM
```

如果在程序中有如下宏调用：

```
……
HTOASC
……
HTOASC
……
```

那么，在汇编时就得到如下宏扩展：

```
                ……
1               AND  AL,0FH
1               CMP  AL,9
1               JBE  ?? 0000
1               ADD  AL,7
1 ?? 0000:ADD  AL,30H
                ……
1               AND  AL,0FH
1               CMP  AL,9
1               JBE  ?? 0001
1               ADD  AL,7
1 ?? 0001:ADD  AL,30H
                ……
```

必须注意，LOCAL 伪指令用在宏定义体内，而且它必须是宏定义伪指令 MACRO 后的第一条语句，在 MACRO 和 LOCAL 伪指令之间还不允许有注释和分号标志。

2. 清除宏定义的伪指令 PURGE

伪指令 PURGE 的作用是告诉汇编程序取消某些宏。其一般格式如下：

```
PURGE 宏名表
```

宏名表由宏名构成，宏名之间用逗号分隔。汇编程序在遇到 PURGE 伪指令后，就取消由宏名表所列出的宏定义，此后不再扩展这些宏。

如果利用宏定义的方法改变了某些指令助记符的意义，那么在用 PURGE 伪指令取消有关宏后，就恢复了有关指令助记符的原始意义。

例如：我们先定义如下宏：

```
LODSB   MACRO
        MOV  AH,[SI]
        INC  SI
        ENDM
```

程序中使用如下指令：

```
        ……
        LODSB
        ……
        PURGE   LODSB
        ……
        LODSB
```

那么，在汇编时就得如下实际代码：

```
        ……
1       MOV  AH,[SI]
1       INC  SI
        ……
        LODSB
```

3. 中止宏扩展的伪指令 EXITM

伪指令 EXITM 通知汇编程序结束当前宏调用的扩展。一般格式如下：

```
EXITM
```

当遇到伪指令 EXITM 时，汇编程序立即退出宏，在宏中剩下的语句不被扩展。如果在一嵌套的宏内遇到伪指令 EXITM，则退出到外层宏。

伪指令 EXITM 通常与条件伪指令一起使用，以便在规定的条件下跳过宏内的最后的语句。

7.2.7 宏定义的嵌套

宏定义的嵌套有两种情况：宏定义体中含宏调用；宏定义体中含宏定义。下面分别举例说明。

1. 宏定义体中调用宏

宏汇编语言允许在宏定义体中使用宏调用，其限制条件仍是：必须先定义后调用。

如下宏 WHTOASC 的定义体内就调用了宏 HTOASC(见 7.2.1)：

```
WHTOASC    MACRO
           MOV     AH,AL
           SHR     AL,1
           SHR     AL,1
           SHR     AL,1
           SHR     AL,1
           HTOASC
           XCHG    AH,AL
           HTOASC
           ENDM
```

如果在程序中调用了上述宏，那么在汇编时可得如下的宏展开：

```
1          MOV     AH,AL
1          SHR     AL,1
1          SHR     AL,1
1          SHR     AL,1
1          SHR     AL,1
2          AND     AL,0FH
2          ADD     AL,90H
2          DAA
2          ADC     AL,40H
2          DAA
1          XCHG    AH,AL
2          AND     AL,0FH
2          ADD     AL,90H
2          DAA
2          ADC     AL,40H
2          DAA
```

注意，指令前的符号表示该指令是宏展开的结果。

2. 宏定义体中定义宏指令

宏定义体中还可含有宏定义，但只有在调用了外层的宏后，才能调用内层的宏。原因是只有在调用了外层的宏后，内层的宏定义才有效。

下面的宏 DEFMAC 含有一个宏定义，并且外层宏的参数 MACNAME 是内层的宏指令名：

```
DEFMAC     MACRO     MACNAME, OPERATOR
           MACNAME     MACRO X,Y,Z
               PUSH    AX
```

```
          MOV       AX,X
          OPERATOR  AX,Y
          MOV       Z,AX
          POP       AX
          ENDM
          ENDM
```

下面的三次宏调用就生成三条宏指令 ADDITION、SUBTRACT 和 LOGOR：

```
DEFMAC    ADDITION,ADD
;
DEFMAC    SUBTRACT,SUB
;
DEFMAC    LOGOR,OR
```

此后，就可调用这三个宏了。如下的宏调用：

```
ADDITION VAR1,VAR2,RESULT
```

可得以下宏扩展：

```
1      PUSH   AX
1      MOV    AX,VAR1
1      ADD    AX,VAR2
1      MOV    RESULT,AX
1      POP    AX
```

7.3 重复汇编

有时程序中会连续地重复完全相同或几乎相同的一组语句，当出现这种情况时，可考虑用重复伪指令定义的重复块，以简化源程序。

重复块是允许建立重复语句块的宏的一种特殊形式。它们与宏的不同之处在于它们没有被命名，并因而不能被调用。但象宏一样，它们可以有参数，且在汇编过程中参数可被实自变量代替；宏运算符、用伪指令 LOCAL 说明的符号等可用在重复块中；重复块总是由伪指令 ENDM 结束。

本节介绍由伪指令 REPT、IRP 和 IRPC 定义的三种重复块，它们之间的不同点在于如何规定重复的次数。

7.3.1 伪指令 REPT

伪指令 REPT 用于创建重复块，重复块的重复次数由一数值表达式给定。一般格式如下：

```
REPT    表达式
需重复的语句组
```

```
ENDM
```

宏汇编程序将把“需重复的语句组”连续地重复汇编由表达式值所决定的次数。表达式必须可求出数值常数(16 位的无符号数)。任何有效的汇编程序语句均可以安排在“需重复的语句组”中。

例如:下面的语句实现把字符 A 到 Z 的 ASCII 码填入数组 TABLE 中:

```
CHAR  =  'A'
TABLE LABEL   BYTE
      REPT    26          ;重复块开始,规定重复次数
      DB      CHAR        ;需重复的语句 1
CHAR  =  CHAR + 1         ;需重复的语句 2
      ENDM                ;重复块结束
```

实际的汇编结果与对如下指令的汇编结果相同:

```
TABLE LABEL   BYTE
      DB      'A'
      DB      'A'+1
      DB      'A'+2
      ......
      DB      'A'+24
      DB      'A'+25
```

7.3.2 伪指令 IRP

伪指令 IRP 用于创建重复块,重复次数和每次重复时使用的实参由实参数列表决定。一般格式如下:

```
IRP  形式参数,<实参 1,实参 2,...,实参 n>
需重复的语句组
ENDM
```

实参的个数规定了重复的次数。宏汇编程序将把“需要重复的语句组”连续地重复汇编规定的次数,并在每次重复时依次用相应位置的实参数代替“需重复语句组”中的形式参数。实参数列表应放在一对尖括号内,若有多个实参数,则各实参数间用逗号分隔。

例如:下面的重复块实现把 0~9 的平方值存入数组 QUART 中:

```
QUART  LABEL    BYTE
         IRP    X,<0,1,2,3,4,5,6,7,8,9>
         DB     X*X
         ENDM
```

再如:下面的重复块实现把若干寄存器值压入堆栈:

```
IRP      REG,<AX,BX,CX,DX>
PUSH     REG
```

```
ENDM
```

实际的汇编结果与对如下指令的汇编结果相同：

```
PUSH    AX
PUSH    BX
PUSH    CX
PUSH    DX
```

7.3.3 伪指令 IRPC

伪指令 IRPC 与伪指令 IRP 相似，但实参数列表是一个字符串，一般格式如下：

```
IRPC    形式参数,字符串
需重复的语句组
ENDM
```

字符串的长度规定了重复的次数。宏汇编程序将把"需要重复的语句组"连续地重复汇编规定的次数，并在每次重复时依次用"字符串"中的一个字符作为实参数代替"需重复语句组"中的形式参数。如果字符串含有空格、逗号等分隔符，那么字符串需用一对尖括号括起来。

例如：下面的重复块也实现把从 2 开始的 10 个偶数存入字数组 TABLE 中：

```
TABLE   LABEL   BYTE
        IRPC      X,0123456789
        DW        (X+1)*2
        ENDM
```

再如，下面的重复块实现把 AX、BX、CX 和 DX 四个寄存器依次压入堆栈：

```
IRPC    REG,    ABCD
        PUSH    REG&X
        ENDM
```

注意，上述重复块中使用了分隔宏参数的强迫替换运算符"&"。

7.4 条件汇编

条件汇编语句提供根据某种条件决定是否汇编某段源程序的功能。在源程序中使用条件汇编语句的主要目的是：(1)通过在汇编前或汇编时改变某种条件，从而方便地产生功能不同的程序；(2)增强宏定义能力，使得宏的适用范围更广；(3)改善汇编效率。

尽管条件汇编语句在形式上与高级语言中的条件语句相似，但本质上却是完全不同的。条件汇编语句是说明性语句，是由伪指令构成，它的功能由汇编程序实现；一般高级语言的条件语句是执行语句，它的功能由目标程序实现。

7.4.1 条件汇编伪指令

条件汇编语句的一般格式如下：

```
IFxxxx   条件表达式
     语句组 1
[ ELSE
     语句组 2 ]
ENDIF
```

IFxxxx 是条件伪指令助记符的一般形式，其中 xxxx 表示构成条件伪指令助记符的其他字符。完整的条件伪指令助记符如下：

```
IF      IFE     IFDEF    IFNDEF    IF1    IF2
IFB     IFNB    IFIDN    IFDIF
```

一定要在条件语句的最后安排伪指令 ENDIF。条件表达式用于表示条件，不同的条件汇编指令，条件表达式的形式也有所不同。语句组可含有任意正确的语句，包括其他的条件语句。伪指令 ELSE 及语句组 2 是可选的。

条件汇编语句的一般意义为：如果条件伪指令要求的条件满足，那么汇编语句组 1，否则不汇编语句组 1；在含有 ELSE 伪指令的情况下，如果条件不满足，则汇编语句组 2。

由于在上述形式中的语句组 1 或语句组 2 可再含有条件汇编语句，所以就可能形成条件汇编语句的嵌套。汇编语言允许的嵌套层数足以满足一般应用需要。一个嵌套的 ELSE 伪指令总是与最近但又没有 ELSE 的 IFxxxx 伪指令相配。

1. 伪指令 IF 和 IFE

伪指令 IF 的一般格式如下：

```
IF       表达式
```

如果表达式的值不等于 0，则条件满足，即条件为真。表达式不能包含向前引用，其结果应为一常数值。

伪指令 IFE 的一般格式如下：

```
IFE       表达式
```

如果表达式的值等于 0，则条件满足，即条件为真。表达式不能包含向前引用，其结果应为一常数值。IFE 伪指令的条件与 IF 伪指令的条件相反。

例 1：在下面的条件语句中，如果 MFLAG 值不为 0，即条件满足，那么就汇编语句组 1，否则汇编语句组 2：

```
if     MFLAG
MOV    AH,0          ;语句组 1
INT    16H           ;当 MFLAG 值不为 0 时，汇编此语句组
else
MOV    AH,1          ;语句组 2
```

```
INT      21H          ;当 MFLAG 值为 0 时,汇编此语句组
endif
```

为了便于表示条件,条件表达式有时是关系表达式或逻辑表达式。

例 2: 在下面的条件语句中,条件表达式是一个关系表达式,根据关系表达式的求值方法,如果 PORT 值为 0,则该关系表达式的值为 1,不为 0,所以条件满足:

```
if    PORT  EQ  0
PORTADDR  =  3F8H
IVECTN    =  0BH      ;条件满足时汇编此语句组
IMASKV    =  11110111B
endif
```

例 3: 如下定义的宏 SHIFTL 使用了重复块和结束宏扩展伪指令 EXITM:

```
SHIFTL  MACRO OP,N
COUNT   =     0
        REPT  N
        SHL   OP,1
COUNT   =     COUNT+1
        if    COUNT  GE  N
        EXITM
        endif
        ENDM
        INC   OP
        ENDM
```

调用该宏和扩展的情况如下:

```
     SHIFTL  AX,1
2    SHL     AX,1
1    INC     AX
     ;
     SHIFTL  BX,3
2    SHL     BX,1
2    SHL     BX,1
2    SHL     BX,1
1    INC     BX
```

请注意伪指令 EXITM 的作用,它只中止一层扩展或重复。

2. 伪指令 IFDEF 和 IFNDEF

伪指令 IFDEF 的一般格式如下:

```
IFDEF   符号
```

如果符号已定义或被说明成外部符号,则条件满足,即条件为真。

伪指令 IFNDEF 的一般格式如下：

```
IFNDEF    符号
```

如果符号未定义或未被说明成外部符号，则条件满足，即条件为真。伪指令 IFDEF 的条件与伪指令 IFNDEF 的条件相反。

例如：在下面的条件语句中，如果已先定义符号 MLARGE，则条件满足，那么过程 AXINC 被定义为远过程，否则过程 AXINC 被定义成近过程：

```
        ifdef   MLARGE
AXINC   PROC    FAR              ;若已定义 MLARGE 则汇编此语句
        else
AXINCS  PROC    NEAR             ;若未定义 MLAGER 则汇编此语句
        endif
        INC     AX               ;不受影响
        RET                      ;不受影响
AXINC   ENDP
```

符号可以在源程序中定义，也可以在汇编命令行中定义。如何在汇编命令行中定义符号，请参见有关汇编器使用手册。

3. 伪指令 IF1 和 IF2

伪指令 IF1 的格式如下：

```
IF1
```

若是第一趟扫描则条件为真。

伪指令 IF2 的格式如下：

```
IF2
```

若是第二趟扫描则条件为真。

7.4.2 条件汇编与宏结合

1. 宏中使用条件汇编

条件汇编与宏相结合，能大大扩大宏的使用范围。

例如：如下定义的宏 ADDNUM 有两个参数，在对宏调用扩展时，能根据不同的参数扩展成不同的指令：

```
ADDNUM    MACRO REG,NUM                          ;宏定义
          if    (NUM GT 2)  OR  (NUM LE 0)
          ADD   REG,NUM
          else
          INC   REG
          if    NUM EQ  2
          INC   REG
```

```
        endif
        endif
        ENDM
```

宏调用举例和相应的扩展情况如下：

```
        ADDNUM    AX,4
1       ADD       AX,4
        ;
        ADDNUM    AX,2
1       INC       AX
1       INC       AX
        ;
        ADDNUM  AX,1
1       INC       AX
        ;
        ADDNUM    AX,0
1       ADD       AX,0
```

汇编语言还提供了专门用于测试宏参数的条件汇编伪指令，下面就介绍这些伪指令。

1. 伪指令 IFB 和 IFNB

伪指令 IFB 一般使用在宏定义内，格式如下：

```
IFB    <参数>
```

如果在宏调用时没有使用实参来代替该形参，那么条件满足。注意，参数应该用尖扩号括起。

伪指令 IFNB 一般使用在宏定义内，格式如下：

```
IFNB    <参数>
```

如果在宏调用时使用实参来代替该形参，那么条件满足。注意，参数应该用尖扩号括起。伪指令 IFNB 的条件与伪指令 IFB 的条件相反。

例如：如下定义的宏 PRINT，若指定显示信息时，则显示之，否则显示缺省信息：

```
PRINT   MACRO MSG
        ifb     <MSG>
        MOV     SI,DEFAULTMSG
        else
        MOV     SI,MSG
        endif
        CALL    SHOWIT
        ENDM
```

2. 伪指令 IFIDN 和 IFDIF

伪指令 IFIDN 一般使用在宏定义内，格式如下：

```
IFIDN     <参数 1>,<参数 2>
IFIDNI    <参数 1>,<参数 2>
```

如果字符串参数 1 与字符串参数 2 相等，则条件满足。参数 1 或参数 2 可能是宏定义中的形参，如果是形参，比较之前先由相应的实参所代替。字符串是按字符逐个比较的，格式一对大小写有区别，格式二忽略大小写区别。注意，参数应用尖括号括起。

伪指令 IFDIF 一般使用在宏定义内，格式如下：

```
IFDIF     <参数 1>,<参数 2>
IFDIFI    <参数 1>,<参数 2>
```

如果字符串参数 1 与字符串参数 2 不等，则条件满足。其他说明同上。

例如：如下定义的宏 RDWR 的第二个参数就决定了读写方式：

```
RDWR  MACRO BUFF,RWMODE
      LEA     DX,BUFF
      ifidni  <RWMODE>,<READ>
      CALL    READIT
      endif
      ifidni  <RWMODE>,<WRITE>
      CALL    WRITEIT
      endif
      ENDM
```

宏调用举例和相应的扩展如下：

```
       RDWR    MESS,Write
   1   LEA     DX,MESS
   1   CALL    WRITEIT
       ;
       RDWR    MESS,Read
   1   LEA     DX,MESS
   1   CALL    READIT
```

例如：如下定义的宏 GETCH 就有加强的功能：

```
GETCH  MACRO CHAR
      MOV     AH,1
      INT     21H
      ifnb    <CHAR>
      ifdifi  <CHAR>,<AL>
      MOV     CHAR,AL
      endif
      endif
      ENDM
```

7.5 源程序的结合

为了便于编辑源程序和对程序进行修改或维护，汇编程序允许把源程序存放在多个文本文件中，在汇编时结合到一起，同时参加汇编。本节介绍源程序的结合方法和应用。

7.5.1 源程序的结合

存放在若干文本文件中的源程序的结合是利用伪指令 INCLUDE 完成的。它的一般格式如下：

```
INCLUDE    文件名
```

伪指令 INCLUDE 指示汇编程序将指定的文本文件从本行起加入汇编，直到该文本文件的最后一行汇编完后，继续汇编随后的语句。

文件名可带有盘符和路径，采用 DOS 有关规则表示。若文件名没有盘符或路径，则首先在由汇编命令行参数/I 所指定的目录中寻找该文件，然后再在当前目录中寻找该文件。对于 MASM 而言，最后还会在由环境变量 INCLUDE 所指定的目录中寻找该文件。对于 TASM 而言，若文件名没有扩展名，则假设扩展名为 ASM。

下列程序的功能是：接受一个字符串，然后按小写和大写形式重新显示字符串。整个源程序存放在三个文本文件中。

```
;程序名：T7-2.ASM
;功  能：(略)
;
        INCLUDE  DATA.ASM              ;结合含数据部分的文件 DATA.ASM
;
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        CLD
        MOV     DX,OFFSET MESS1        ;显示提示信息
        MOV     AH,9
        INT     21H
        MOV     AH,10                  ;接受字符串
        MOV     DX,OFFSET BUFFER
        INT     21H
        CALL    NEWLINE                ;另起一行
        MOV     BL,BUFFER+1
        XOR     BH,BH
        MOV     BUFFER[BX+2],0         ;把接收到的字符串标为 0 结尾
        MOV     SI,OFFSET STRBEG
```

```
        CALL    STRLWR              ;转换成小写字符串
        MOV     SI,OFFSET STRBEG
        CALL    DISPMES             ;显示之
        CALL    NEWLINE             ;另起一行
        MOV     SI,OFFSET STRBEG
        CALL    STRUPR              ;转换成大写字符串
        MOV     SI,OFFSET STRBEG
        CALL    DISPMES             ;显示之
        CALL    NEWLINE             ;另起一行
EXIT:   MOV     AH,4CH              ;结束
        INT     21H
;
NEWLINE  PROC
         ;另起一行,代码略。
NEWLINE  ENDP
;
;子程序名:DISPMES
;功    能:显示以0结尾的字符串
;入口参数:SI=字符串首地址偏移
DISPMES  PROC
DISPME1: LODSB
         OR      AL,AL
         JZ      DISPME2
         MOV     DL,AL
         MOV     AH,2
         INT     21H
         JMP     DISPME1
DISPME2:  RET
DISPMES   ENDP
         ;
         INCLUDE  STRING.ASM        ;结合含子程序的文件STRING.ASM
         ;
CSEG     ENDS
         END     START
```

文本文件DATA.ASM含有数据部分,其内容如下所示:

```
;文件名:DATA.ASM
;内  容:程序T7-2.ASM的一部分
STRLEN  =      128
DSEG           SEGMENT
BUFFER         DB       STRLEN,0,STRLEN  DUP  (0)
STRBEG         =        BUFFER+2
```

```
MESS1          DB      "Please input:$"
DSEG           ENDS
```

文本文件STRING.ASM含有两个子程序STRLWR和STRUPR,它们分别实现把字符串转换成小写和大写,其内容如下所示:

```
;文件名:STRING.ASM
;内  容:程序T7-2.ASM的一部分
;
;子程序名:STRLWR
;功    能:把字符串转换为小写
;入口参数:SI=字符串起始地址偏移
;其他说明信息略
STRLWR     PROC
           JMP  STRLWR2
STRLWR1:   SUB  AL,'A'
           CMP  AL,'Z'-'A'
           JA   STRLWR2
           ADD  AL,'a'
           MOV  [SI-01],AL
STRLWR2:   LODSB
           AND  AL,AL
           JNZ  STRLWR1
           RET
STRLWR     ENDP
;
;子程序名:STRUPR
;功    能:把字符串转换成大写
;入口参数:SI=字符串起始地址偏移
;其他说明信息略
STRUPR     PROC
           JMP  STRUPR2
STRUPR1:   SUB  AL,'a'
           CMP  AL,'z'-'a'
           JA   STRUPR2
           ADD  AL,'A'
           MOV  [SI-01],AL
STRUPR2:   LODSB
           AND  AL,AL
           JNZ  STRUPR1
           RET
STRUPR     ENDP
```

7.5.2 宏库的使用

通常程序员会把一组有价值和经常使用的宏定义集中存放在一个文本文件中，这样的文本文件称为宏库。有了宏库后，只要在源程序首安排结合宏库的伪指令 INCLUDE，便能方便地调用宏库中的宏。这样做不仅节省编辑源程序的时间，而且能够减少错误。

例如：设已建立宏库 DOSBIO.MAC，其内容如下：

```
;接受一个字符串
GETSTR    MACRO MBUFF
    MOV   DX,MBUFF
    MOV   AH,10
    INT   21H
    ENDM
;
;显示一个字符串
DISPSTR   MACRO MBUFF
    MOV   DX,MBUFF
    MOV   AH,9
    INT   21H
    ENDM
;
;取得一个字符
GETCH     MACRO CHAR
    MOV   AH,1
    INT   21H
    IFNB  <CHAR>
    IFDIFI <CHAR>,<AL>
    MOV   CHAR,AL
    ENDIF
    ENDIF
    ENDM
;
;显示一个字符
ECHOCH    MACRO  CHAR
    IFNB  <CHAR>
    IFDIFI <CHAR>,<DL>
    MOV   DL,CHAR
    ENDIF
ENDIF
    MOV   AH,2
    INT   21H
    ENDM
```

现在利用上述宏库 DOSBIO. MAC 改写程序 T7-2. ASM 如下：

```
;程序名：T7-3. ASM
;功  能：(略)
;
        INCLUDE     DOSBIO. MAC         ;结合宏库 DOSBIO. MAC
;
        INCLUDE     DATA. ASM           ;含数据部分的文件 DATA. ASM
;
CSEG    SEGMENT
        ASSUME  CS : CSEG,DS : DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        CLD
        DISPSTR  <OFFSET MESS1>         ;调用宏 DISPSTR
        GETSTR   <OFFSET BUFFER>        ;调用宏 GETSTR
        CALL    NEWLINE
        ;
        MOV     BL,BUFFER+1
        ;;与程序 T7-2 中相应内容相同
EXIT:   MOV     AH,4CH
        INT     21H
;
NEWLINE     PROC
        ECHOCH      0DH                 ;调用宏 ECHOCH
        ECHOCH      0AH                 ;调用宏 ECHOCH
        RET
NEWLINE     ENDP
;
DISPMES     PROC
DISPME1:  LODSB
        OR      AL,AL
        JZ      DISPME2
        ECHOCH      AL                  ;调用宏 ECHOCH
        JMP     DISPME1
DISPME2:  RET
DISPMES     ENDP
        ;
        INCLUDE     STRING. ASM         ;结合文件 STRING. ASM
        ;
CSEG    ENDS
        END     START
```

对上述程序有如下两点补充说明：

(1) 由于 MASM 采用两遍扫描，并且只在第一遍扫描时登记宏定义，所以可使用如下条件汇编伪指令，通知汇编程序在第一遍扫描时加入宏库，而在第二遍扫描时，不加入宏库：

```
IF1
INCLUDE    DOSBIO.MAC
ENDIF
```

这样既加快了第二遍扫描的速度，又能避免在汇编列表清单中含有宏定义部分。但必须注意，如果结合的不是宏库，则不能只在第一遍扫描时结合，而在第二遍扫描时不结合。

(2) 宏库中可含有多个宏定义，一个程序不一定调用宏库中定义的全部宏。例如，上述程序结合了宏库 DOSBIO.MAC，但没有调用其中的宏 GETCH。对于不使用的宏，可以用伪指令 PURGE 清除。清除操作如下所示：

```
INCLUDE    DOSBIO.MAC
PURGE      GETCH
```

注意，上述清除操作对宏库没有影响。

7.6 习　　题

题 7.1 结构类型和记录类型有什么用途？

题 7.2 如下结构类型中各字段的相对偏移是多少？

```
PERSON     STRUC
ID         DB     8  DUP  (?)
PCODE      DW     0
ADDRESS    DB     20  DUP  (?)
PERSON     ENDS
```

题 7.3 如何访问结构变量中的字段？请举例说明。

题 7.4 如何访问记录变量中的字段？请举例说明。

题 7.5 完善 7.1.1 节中的例 1，使其能够按成绩排序，并在屏幕上显示输出。

题 7.6 比较宏与子程序，它们有何异同？它们的本质区别是什么？

题 7.7 简述宏指令的用途？就每种用途，分别举例说明。

题 7.8 宏汇编语言有哪三种类型的语句？它们各有何特征？

题 7.9 宏指令中的参数有何用途？宏调用如何传递参数？如何使得宏参数是字符串或标号的一部分？

题 7.10 编写一个可把所有段寄存器和通用寄存器压入堆栈的宏。

题 7.11 编写一个利用 BIOS 显示 I/O 程序实现回车换行的宏。

题 7.12 请编写一个定义堆栈段的宏。

题 7.13 请编写一个在堆栈中定义若干局部变量的宏。可通过基于 BP 寄存器的相

对寻址访问这些局部变量。

题 7.14 请编写一个撤消堆栈中局部变量的宏。该宏与习题 7.13 所定义的宏相对应。

题 7.15 编写一个把 1 位十六进制数字 ASCII 码符转换为对应二进制数的宏。

题 7.16 请编写一个清除键盘缓冲区的宏。

题 7.17 什么是宏定义的嵌套？请举例说明。

题 7.18 宏定义的嵌套有何用途？请举例说明。

题 7.19 请编写一个用 2 位十六进制数显示 AL 内容的宏。该宏调用已定义的宏 HTOASC 和宏 ECHO。宏 HTOASC 把 1 位十六进制数转换成对应的 ASCII 码，宏 ECHO 显示字符。

题 7.20 请编写一个可定义各种移位宏指令的宏。所定义的宏指令所移位的次数不限于 1 或者 CL，而可以是常数或者其他 8 位寄存器。

题 7.21 请利用重复汇编的方法定义一个缓冲区。缓冲区有 100 双字构成，每个双字的高字部分的初值依次是 2，4，6，……，200，低字部分的初值总是 0。

题 7.22 请利用重复汇编的方法实现把通用寄存器和段寄存器依次推入堆栈实现保护。

题 7.23 设程序中有 8 个标号，分别为 NEXT1，NEXT2，……，NEXT8。请利用重复汇编的方法定义由上述 8 个标号构成的散转表，每项由段值和偏移构成。

题 7.24 请利用重复汇编的另一种方法实现习题 7.23 之要求。

题 7.25 把习题题 7.13 和习题 7.14 所要求定义的宏结合成一个宏。

题 7.26 条件汇编有何用途？请举例说明。

题 7.27 请编写一个可实现把操作数 1 和操作数 2 相加之和送操作数 1 的宏。操作数 1 及操作数 2 可能都是存储器单元，也可能都是寄存器。所定义的宏应能区分这些情况，并分别对待。

题 7.28 请编写一个可实现把操作数 1 和操作数 2 相乘之积送操作数 3 的宏。该宏必须通用、灵活和高效。

题 7.29 请编写一个能够根据某个符号值采用不同方法显示字符的宏。

题 7.30 请编写一个定义若干符号常量的源程序级文本文件。该文件内使用条件伪指令和特定标识符号，以便于被 INCLUDE 伪命令包含。

题 7.31 什么是宏库？有何用途？如何方便地利用它。

第 8 章　模块化程序设计技术

把一个程序分成具有多个明确任务的程序模块，分别编制、调试后再把它们连接在一起，形成一个完整的程序，这样的程序设计方法称为模块化程序设计。

模块化程序设计有如下优点：(1)单个的程序模块易于编写、调试及修改；(2)若干程序员可以并行工作，工作进度可加快；(3)若干反复使用和验证过的程序模块可被利用；(4)程序的易读性好；(5)程序的修改可局部化。

模块化程序设计的主要步骤是：(1)正确地描述整个程序需要完成什么样的工作；(2)把完整的任务划分成多个具有明确功能的程序模块，并明确各模块之间的相互关系；(3)根据各模块的具体功能和地位，选择合适的程序设计语言，编写程序并初步调试；(4)把各模块分别编译或汇编成目标模块，并连接到一起，经过调试形成一个完整的程序；(5)整理文档资料。

虽然模块化程序设计的关键是模块的划分，但本章只介绍如何利用汇编语言编写符合要求的程序模块。

8.1　段的完整定义

一个复杂的程序通常有若干模块组成。源模块可用汇编语言编写，也可用高级语言编写。每个源模块被单独汇编或编译成目标(OBJ)模块，最后由连接程序(LINKER)把各目标模块连接成一个完整的可执行的程序。

由于 8086/8088 采用分段的形式访问内存，所以一个模块往往又含有多个段。一个程序的若干模块之间存在的联系必定体现在模块间的段与段的联系上。

连接程序如何把若干模块的多个段恰当地组合到一起呢？如何沟通有关段之间的联系呢？实际上，汇编语言中的段定义伪指令等指示汇编程序把合适的连接信息写入到目标模块中，连接程序再根据目标模块中的连接信息进行连接操作。

在新版的汇编语言中，有两种方法定义段：完整的段定义和简化的段定义。本节介绍段的完整定义。

8.1.1　完整的段定义

完整的段定义提供了彻底控制段的机制，该机制可使得各模块的各个段严格按要求组合和衔接。

1. 一般格式

完整段定义的一般格式如下：

```
段名  SEGMENT  [定位类型]   [组合类型]   ['类别']
……
```

```
语句
……
段名　ENDS
```

段开始语句 SEGMENT 中的可选项"定位类型"、"组合类型"和"'类别'"通知汇编程序和连接程序如何建立和组合段。应当按顺序说明这些可选项，但不需要给出所有可选项，如果不给出某个可选项，那么汇编程序使用该可选项的缺省值。在前面各章节所列程序中的段虽然均采用完整的定义，却都没有给出这些可选项。

段名可以是唯一的，也可以与程序中其他的段名相同。在同一模块中，如果已用相同的段名定义过段，那么当前这个段就被视为前一个同名段的继续，即同一个段。

对一个模块中的同名段而言，后续同名段的定义伪指令 SEGMENT 中的可选项取值应该与前一个同名段相同，或者不再给定可选项值而默认与前一个同名段相同。

例 1：如下程序 T8-1. ASM 中含有两个名为 DSEG 的段和两个名为 CSEG 的段：

```
;程序名：T8-1. ASM
;功  能：(略)
DSEG    SEGMENT                 ;定义数据段 DSEG
MESS    DB  'HEL'
DSEG    ENDS
;
CSEG    SEGMENT                 ;定义代码段 CSEG
        ASSUME CS：CSEG,DS：DSEG
START： MOV   AX,DSEG
        MOV   DS,AX
        MOV   DX,OFFSET MESS
        MOV   AH,9
        INT   21H
CSEG    ENDS
;
DSEG    SEGMENT                 ;数据段 DSEG 的继续
        DB  'LO',0DH,0AH,'$'
DSEG    ENDS
;
CSEG    SEGMENT                 ;代码段 CSEG 的继续
        MOV   AX,4C00H
        INT   21H
CSEG    ENDS
        END   START
```

由于后一同名段被视为前一同名段的继续，所以汇编后只有 DSEG 和 CSEG 两个段，类似于程序 T3-1. ASM 的汇编结果。

下面介绍段定义伪指令中可选项的作用和所取值的意义。

2. 定位类型

定位类型表示出当前段对起始地址的要求，从而指示连接程序如何衔接相邻两段。可选择的定位类型及所表示的起始地址列于表 8.1。

表 8.1 定位类型

定位类型	起始地址(二进制表示)	含　义
BYTE	XXXX XXXX XXXX XXXX XXXX	使用下一个可用字节地址
WORD	XXXX XXXX XXXX XXXX XXX0	使用下一个可用字地址
DWORD	XXXX XXXX XXXX XXXX XX00	使用下一个可用双字地址
PARA	XXXX XXXX XXXX XXXX 0000	使用下一个可用节地址
PAGE	XXXX XXXX XXXX 0000 0000	使用下一个可用页地址

一般情况下(80386 以下)缺省的定位类型是 PARA，即段起始地址位于可用的第一个节(每节为 16 个字节)的边界处。定位类型 BYTE 使得当前段紧接前一段，前后两段间没有空闲单元，所以是最节约的定位类型。定位类型 WORD 使得段从偶地址开始，不仅较为节约，而且有利于把数据单元定位在偶地址。定位类型 DWORD 常用于 80386 的 32 位段。一页等于 256 字节，所以定位类型 PAGE 可能导致最大的段间隔。

例 2：如下程序 T8-2.ASM 的两个段的定位类型均是 PARA：

```
;程序名：T8-2.ASM
;功  能：(略)
DSEG    SEGMENT   PARA   COMMON              ;类型为 PARA
MESS    DB  'HELLO! ',0DH,0AH,'$'
DSEG    ENDS
;
CSEG    SEGMENT   PARA   PUBLIC              ;定位类型为 PARA
        ASSUME   CS:CSEG,DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        MOV     DX,OFFSET MESS
        MOV     AH,9
        INT     21H
        MOV     AX,4C00H
        INT     21H
CSEG    ENDS
        END     START
```

最后的可执行程序中两个段的衔接情况图 8.1(a)所示。如果把段 CSEG 的定位类型改为 WORD，衔接情况如图 8.1(b)所示。如果把段 CSEG 的定位类型改为 BYTE，衔接情况如图 8.1(c)所示。

3. 组合类型

不同模块的同名段的组合，为更有效更便利地使用存储器提供了方便。组合类型就是

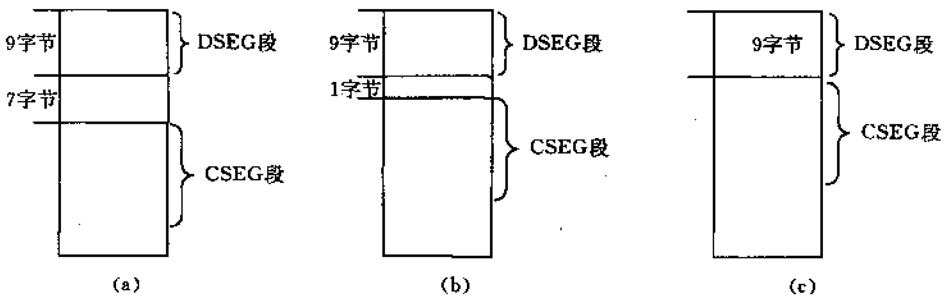


图 8.1　定位类型的选择

用于通知连接程序，如何把不同模块内段名相同的段组合到一起。有如下组合类型：

(1) PUBLIC

组合类型 PUBLIC 表示当前段与其它模块中组合类型为 PUBLIC 的同名段组合成一个段。组合的先后顺序取决于启动 LINK 程序时目标模块名排列的次序。由于组合后受同一个段地址控制，所以组合时后续段的起始地址都作相应调整。但组合时仍遵照定位类型进行衔接，即同名段间可能有间隔。

(2) COMMON

组合类型 COMMON 表示当前段与其它模块中的同名段重叠，即起始地址相同。最终段的长度等于它们中最长的段的长度。由于段覆盖，所以前一同名段中的初始化数据可能被后续同名段中的初始化数据所覆盖。

(3) STACK

组合类型 STACK 表示当前段是堆栈段，组合情况与 PUBLIC 相同。

(4) MEMORY

组合类型 MEMORY 与组合类型 PUBLIC 相同，为兼容而设。

(5) AT 表达式

它表示当前段应按绝对地址定位，其段地址即为表达式之值。一般 AT 段不包含代码和初始化数据，它仅用于表示已在内存中的代码或数据的地址样板，如显示缓冲区或其它由硬件定义的绝对存储单元。LINK 程序不对 AT 段生成任何代码或数据。

(6) PRIVATE

组合类型 PRIVATE 表示不与其它段组合。宏汇编程序 MASM 不识别此关键字。

若段定义伪指令 SEGMENT 语句中没有给出组合类型，就表示不与其它段组合。

例 3：某个程序的第一个源程序模块是例 2 所给出的 T8-2. ASM，第二个源程序模块如下所示：

```
;程序(模块)名：T8-3.ASM
;功       能：(略)
DSEG    SEGMENT    PARA    COMMON
        DB  'OK'
DSEG    ENDS
```

```
;
CSEG    SEGMENT   PARA    PUBLIC
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END
```

先把这两个源程序模块分别汇编，最后用 LINK 程序连接，命令行如下：

```
LINK    T8-2  +  T8-3
```

模块 2 中的 DSEG 段和模块 1 中的 DSEG 段重叠，模块 2 中的 CSEG 段和模块 1 中的 CSEG 段合并成一个段。最后得到的可执行程序的内容如下所示：

```
0000:0000  4F 4B 4C 4C 4F 0D 0A 24   ;两个段重叠后的结果 OKLLO.. $
0000:0008  00 00 00 00 00 00 00 00   ;CSEG 的定位类型是 PARA 导致的间隔
0001:0000  MOV AX,26E2               ;模块 T8-2 中的 CSEG 段
0001:0003  MOV DS,AX
0001:0005  MOV DX,0000
0001:0008  MOV AH,09
0001:000A  INT  21
0001:000C  MOV AX,4C00
0001:000F  INT  21
0001:0011  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ;衔接间隔
0001:0020  MOV AH,4C                 ;模块 T8-3 中的 CSEG 段
0001:0022  INT  21
```

4. 类别

类别用于表示段的分类。LINK 程序总是使类别相同的段相邻。实际上只有类别相同的同名段才根据组合类型进行组合。

类别是一个由程序员指定的字符串，但必须用单引号括起。如果一个段没有给出类别，那么这个段的类别就为空。例 3 中两个模块的四个段均没有给出类别，它们的类别均为空。

例 4：设某个程序的模块甲如下所示：

```
;模块甲(MODULE1)
DSEG    SEGMENT   PARA   PUBLIC 'DATA'
        ......
DSEG    ENDS
;
CSEG    SEGMENT   PARA   PUBLIC 'CODE'
START:  ......
CSEG    ENDS
        END     START
```

再设该程序的模块乙如下所示：

```
;模块乙(名为 MODULE2)
DSEG    SEGMENT   PARA   PUBLIC 'XYZ'
        ……
DSEG    ENDS
;
ASEG    SEGMENT   PARA   PUBLIC 'CODE'
        ……
ASEG    ENDS
        END
```

再设该程序的模块丙如下所示：

```
;模块丙(名为 MODELE3)
ESEG    SEGMENT   PARA   PUBLIC 'DATA'
        ……
ESEG    ENDS
        END
```

用如下命令连接它们的目标模块：

```
LINK    MODULE1 + MODULE2 + MODULE3
```

在得到的可执行程序中，各段的排列次序是：模块甲的 DSEG、模块丙的 ESEG、模块甲的 CSEG、模块乙的 ASEG 和模块乙的 DSEG。

8.1.2 关于堆栈段的说明

一个完整的汇编语言源程序一般应该含有一个堆栈段，只有 COM 型程序例外。

当把某个段的组合类型指定为 STACK 时，这个段就被指定为堆栈段了。也就是说，组合类型 STACK 不仅是某种组合类型，而且能够表示当前段是堆栈段。当然，如果在程序的其他模块中也有组合类型为 STACK 的同名段，那么连接时将以接续的方式组合到一起，这样会构成一个存储空间更大的堆栈。

例 5：为程序 T8-1. ASM 增加一个大小为 1024 字节的堆栈段。

```
;程序名：T8-1A. ASM
;功  能：(略)
SSEG    SEGMENT   PARA   STACK          ;定义堆栈段
        DB  1024  DUP  (?)
SSEG    ENDS
;
DSEG    SEGMENT   PARA   COMMON
MESS    DB 'HELLO',0DH,0AH,'$'
DSEG    ENDS
;
```

```
CSEG    SEGMENT   PARA   PUBLIC
        ;内容与程序 T8-1 中的代码段内容相同
CSEG    ENDS
        END    START
```

LINK 程序会把组合类型为 STACK 的段的有关信息写入可执行程序文件中。于是在执行该程序时，操作系统的装入程序就能根据这些信息自动设置寄存器 SS 和 SP，从而构成物理堆栈。设置的 SS 值是组合类型为 STACK 的段的段值，设置的 SP 值是堆栈段的大小，即 SS:SP 指向堆栈尾。

如果在说明堆栈段时不指明组合类型 STACK，虽然在主观上想让它用作堆栈段，但没有得到汇编程序和连接程序的承认，所以必须在代码段中安排传送指令来设置寄存器 SS 和 SP。

例如：设源程序中有如下准备用于堆栈的段：

```
SSEG    SEGMENT   PARA
        DB  1024  DUP  (?)
STOP    LABEL   WORD
SSEG    ENDS
```

那么在代码段中可用如下指令设置堆栈：

```
CLI
MOV    AX,SSEG
MOV    SS,AX
MOV    SP,OFFSET STOP
STI
```

由于硬件中断和程序使用同一个堆栈，所以在切换堆栈时要关中断。

允许一个汇编语言源程序不含有堆栈段。如果 LINK 程序没有发现堆栈段，那么它会发出警告信息，并把第一段的段值作为堆栈的段值，堆栈空间设定为 64KB。我们在先前章节中给出的程序几乎都没有说明堆栈段，所以这些程序都使用这个缺省的堆栈段。如果认为这个缺省的堆栈段是可行的话，那么在程序中可不必说明堆栈段，也不必理会 LINK 程序给出的警告信息。

无论在程序中是否说明堆栈段，只要需要，都可通过重置寄存器 SS 和 SP 来切换堆栈，从而建立合适的新堆栈。

8.1.3 段组的说明和使用

先看下面的程序 T8-4.ASM。它含有两个数据段和一个代码段。为了说明问题，把数据段和代码段都作了简化。

```
;程序名：T8-4.ASM
;功  能：(略)
DSEG1  SEGMENT    PUBLIC            ;数据段 1
VAR1   DB  ?
```

```
DSEG1  ENDS
;
DSEG2  SEGMENT    PUBLIC              ;数据段 2
VAR2   DB  ?
DSEG2  ENDS
;
CSEG   SEGMENT    PARA     PUBLIC ;代码段
       ASSUME   CS:CSEG,DS:DSEG1;使 CS 对应段 CSEG
                                  ;使 DS 对应段 DSEG1
START: MOV     AX,DSEG1
       MOV     DS,AX              ;置 DS 寄存器
       MOV     BL,VAR1
       ;……
       ASSUME   DS:DSEG2          ;使 DS 对应段 DSEG2
       MOV     AX,DSEG2
       MOV     DS,AX              ;重置 DS 寄存器
       MOV     VAR2,BL
       ;……
       MOV     AH,4CH
       INT     21H
CSEG   ENDS
       END     START
```

在上述程序 T8-4 中，为了访问变量 VAR1，设置了数据段寄存器 DS，为了访问变量 VAR2 而重置了 DS。如果要频繁地交叉访问段 DSEG1 和段 DSEG2 中的数据，那么不仅很麻烦，而且程序也会变得冗长。改进的方法是使附加段寄存器 ES 对应段 DSEG2，并把 DSEG2 段的段值置入 ES，通过段超越前缀"ES:"来实现对段 DSEG2 中数据的访问。但改进的方法仍会使目标代码有冗余。

好的方法是把 DSEG1 和 DSEG2 段作为一个段来处理。段组就是为实现此目的服务的。若程序员要在源代码的各独立段中安排几种数据类型，且要在执行时能通过一个独立的、公用的段寄存器访问它们，就可使用段组。

伪指令 GROUP 用于把源程序模块中若干不同名的段集合成一个组，并赋予一个组名。它的一般格式如下：

    组名 GROUP 段名[,段名……]

其中，段名与段名间用逗号间隔，段名也可用由表达式"SEG 变量"或者表达式"SEG 标号"代替。

例 6：利用段组，改写程序 T8-4.ASM。

改写的程序如下：

```
;程序名：T8-4G.ASM
;功  能：(略)
```

```
DS1S2   GROUP DSEG1,DSEG2             ;说明段组
;
DSEG1   SEGMENT  PUBLIC               ;数据段 1
VAR1    DB  ?
DSEG1   ENDS
;
DSEG2   SEGMENT  PUBLIC               ;数据段 2
VAR2    DB  ?
DSEG2   ENDS
;
CSEG    SEGMENT  PARA  PUBLIC         ;代码段
        ASSUME CS:CSEG,DS:DS1S2       ;使 DS 对应组 DS1S2
START:  MOV     AX,DS1S2
        MOV     DS,AX                 ;置 DS 寄存器
        MOV     BL,VAR1
        ;……
        MOV     VAR2,BL
        ;……
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     START
```

组名表示组，也代表组的起始地址。组名的使用与段名的使用类似。上述程序中的指令“MOV AX,DS1S2”是把组 DS1S2 的起始地址的段值送 AX 寄存器。段组名也可用在 ASSUME 语句中，表示使某个段寄存器与某个组相对应。

在定义段组后，段组内各段所定义的所有标号和变量除与定义它们的段起始点相关外还与组的起始点相关。如果在 ASSUME 伪指令中使段寄存器与段组对应，那么有关标号或变量的地址就相对于段组起点计算；如果在 ASSUME 伪指令中使段寄存器与组内某个段对应，那么有关标号或变量就相对于该段的起点计算。所以在使用段组后，程序员要谨慎地使用 ASSUME 伪指令，并保证具体置入段寄存器的值与之相适应。

例 7：如下程序 T8-5. ASM 说明了如何把变量作为组的成员访问和把变量仅作为段内的成员访问。

```
;程序名：T8-5. ASM
;功  能：(略)
DGROUP  GROUP CSEG,DSEG                    ;说明段组
;
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:DGROUP         ;使 DS 与组对应
START:  MOV     AX,DGROUP
        MOV     DS,AX
```

```
        MOV     BL,VAR1                    ;作为组内成员访问
        MOV     VAR2,BL
        ;
        ASSUME  DS:DSEG                    ;使 DS 与段 DSEG 对应
        MOV     AX,DSEG
        MOV     DS,AX
        MOV     BH,VAR1                    ;仅作为段内成员访问
        MOV     VAR2,BH
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
;
DSEG    SEGMENT                            ;数据段 DSEG
VAR1    DB  'A'
VAR2    DB  'B'
DSEG    ENDS
        END     START
```

在把连接后所得的可执行程序 T8-5 装入内存时的映象如下(设开始段值为 26E2H):

```
26E2:0000  MOV  AX,26E2       ;CSEG 段
     0003  MOV  DS,AX
     0005  MOV  BL,[0020]     ;MOV AL,VAR1
     0009  MOV  [0021],BL     ;MOV VAR2,AL
     000D  MOV  AX,26E4
     0010  MOV  DS,AX
     0012  MOV  BH,[0000]     ;MOV BH,VAR1
     0016  MOV  [0001],BH     ;MOV VAR2,BH
     001A  MOV  AH,4C
     001C  INT  21
     001D  00  00  00         ;段之间隔
26E2:0020  41  42             ;DSEG 段(即 26E4:0)
```

如果要用运算符 OFFSET 得到在段组内某个段中定义的标号或变量相对于段起始点的偏移,那么必须在标号或变量前再加上组名。例如:

```
    MOV     DX,OFFSET  DGROUP VAR1
```

否则,只能得到相对于所在段起始点的偏移。

段组并不直接影响连接时段的次序,组内各段不必连续,不属于这组的某个段可以夹在组内的两个段之间。由于要通过一个段寄存器访问组内各段,所以连接后组内所有段仍必须保证在 64KB 以内。

在纯汇编语言程序中,尽管程序员可以按自己的愿望使用段组,但使用段组并没有太

多的必要。

8.2 段的简化定义

完整的段定义使得程序员可以完全控制段，但较为复杂。新版汇编语言提供了段的简化定义方法，从而使程序员能方便地定义段。无论是编写独立的汇编语言程序，还是编写供高级语言程序调用的函数，简化的段定义伪指令几乎总使程序设计更容易。本节介绍段的简化定义。

8.2.1 存储模型说明伪指令

在程序中使用段简化定义伪指令之前，必须先使用存储模型说明伪指令描述程序采用的存储模型。存储模型说明伪指令的简单格式如下：

```
.MODEL    存储模型
```

注意该伪指令以符号点“.”引导。例如，为了说明采用SMALL存储模型，只要在源程序首使用如下伪指令：

```
.MODEL    SMALL
```

常用的存储模型有：

(1) SMALL

全部数据限制在单个64KB段内；全部代码也限制在单个64KB段内。这是独立的汇编语言程序最常用的模型。在这种存储模型下，数据段寄存器可保持不变，所有转移均可认为是段内转移。

(2) MEDIUM

全部数据限制在单个64KB段内；但代码可大于64KB。在这种存储模型下，数据段寄存器可保持不变，但会出现段间转移的情形。

(3) COMPACT

全部代码限制在单个64KB段内；数据总量可大于64KB，但一个数组不能大于64KB。

(4) LARGE

代码可超过64KB；数据也可超过64KB，但一个数组不能大于64KB。

(5) HUGE

代码可超过64KB；数据也可超过64KB，并且一个数组也能大于64KB。

独立的汇编语言程序可选用任一种存储模型，对大多数完全用汇编语言编写的程序来说，小(SMALL)模型就足够了。

8.2.2 简化的段定义伪指令

1. 简化的段定义伪指令

简化的段定义伪指令均以符号点引导。下面介绍常用的简化段定义伪指令：

(1) 定义代码段的伪指令

定义代码段的伪指令如下，它表示一个代码段的开始：

```
.CODE
```

例 1：写一个使系统喇叭发出“嘟”一声的程序。

```
;程序名：T8-6.ASM
;功  能：(略)
        .MODEL   SMALL        ;说明采用小模型
        .CODE                 ;说明代码段开始
START:  MOV    DL,7
        MOV    AH,2
        INT    21H
        MOV    AX,4C00H
        INT    21H
        END    START          ;结束代码段
```

上述程序就一个代码段，没有数据段和堆栈段。伪指令.CODE说明代码段的开始，伪指令END说明段结束。

简化的段定义伪指令说明一个段的开始，同时也表示上一个段的结束。伪指令END说明最后一个段的结束。

(2) 定义堆栈段的伪指令

定义堆栈段的伪指令一般格式如下，它表示一个堆栈段的开始：

```
.STACK    [大小]
```

可选的“大小”说明堆栈的字节数，若没有指定堆栈大小，则采用缺省值1024。

如下的伪指令就表示定义一个2KB的堆栈。

```
.STACK    2048
```

通常只有在编写纯粹的汇编语言程序时才需要定义堆栈。

(3) 定义数据段的伪指令

定义(初始化)数据段的伪指令如下，它表示数据段的开始：

```
.DATA
```

例如：如下伪指令定义了一个数据段：

```
        .DATA
VAR3    DB  5
IARRAY  DW     50  DUP(0)
MESS    DB 'HELLO',0DH,0AH,'$'
```

例 2：利用简化的段定义伪指令改写8.1.2节的程序T8-1A.ASM。

```
;程序名：T8-1B.ASM
;功  能：(略)
```

```
        .MODEL    SMALL        ;说明存储模型
        .STACK    1024         ;定义堆栈段
        .DATA                  ;说明数据段开始
MESS    DB  'HELLO',0DH,0AH,'$'
        .CODE                  ;说明代码段开始
START:  MOV    AX,DGROUP       ;把段组的起始段值
        MOV    DS,AX           ;置入数据段寄存器 DS
        MOV    DX,OFFSET MESS
        MOV    AH,9
        INT    21H
        MOV    AX,4C00H
        INT    21H
        END    START
```

在一个源程序模块中可定义多个由伪指令.DATA 开始的数据段，这如同在一个源程序模块中定义多个同名的数据段。

此外，还有伪指令.DATA?和伪指令.CONST，它们分别用于说明未初始化数据段的开始和常数数据段的开始。在编写纯粹的汇编语言程序时，一般不使用这两条伪指令，因为在由伪指令.DATA 说明的数据段中也可以定义未初始化的数据和常数数据。除非为了遵守高级语言的约定，可能使用这两条伪指令。

宏汇编程序自动把可能存在的由.DATA 说明的数据段、由.CONST 说明的常数数据段、由.DATA? 说明的未初始化数据段和由.STACK 说明的堆栈段集合成一个段组。那么如何定义一个不属于这个段组的独立数据段呢？如何定义一个较大的数据段呢？这可利用伪指令.FARDATA 来实现。

(4) 定义远程(独立)数据段的伪指令

定义独立数据段伪指令的一般格式如下，它表示一个独立数据段的开始：

```
.FARDATA  [名字]
```

“名字”是可选的，如果使用，则就成为该数据段的段名。

例如：如下伪指令定义了一个独立的数据段：

```
        .FARDATA
NEWPTR  DD  0
BUFF    DB  1024  DUP (?)
```

此外，还有伪指令.FARDATA? 用于说明未初始化的独立数据段。在编写纯粹汇编语言程序时，无需使用伪指令.FARDATA?，因为在由伪指令.FARDATA 说明的独立数据段中也可定义未初始化数据。

2. 缺省段名

在使用简化的段定义伪指令说明各段后，程序员一般不需要知道这些段的段名和它们的定位类型、组合类型等。但如果想把简化的段定义伪指令与标准的段定义伪指令混合使用，那么就需要知道这些内容了。表 8.2 列出了在小(SMALL)内存模型情况下，各段的

段名等信息。

表 8.2 小型模式下的段名和类型

伪指令	段 名	定位类型	组合类型	类 别	组名
.CODE	TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	
.DATA	DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	BSS	WORD	PUBLIC	'BSS'	DGROUP
STACK	STACK	PARA	STACK	'STACK'	DGROUP

如果在中型、大型或巨型模式下，由伪指令.CODE 说明的代码段的段名在字符串"_TEXT"之前还会加上模块名(源文件名)。例如，设模块名为 ABC，则代码段名就成为 ABC_TEXT。因此，如果使用中大型内存模式，那么模块源文件名不要以数字开头。

如果在使用伪指令.FARDATA 说明一个独立的数据段时加了名字，那么此名字就成为该独立数据段的段名。

8.2.3 存储模型说明伪指令的隐含动作

1. 隐含的段组和段设定

存储模型说明伪指令.MODEL 除了说明程序采用的存储模型外，还起着相当于如下语句的作用：

```
DGROUP        GROUP DATA, CONST, BSS, STACK
ASSUME     CS : _TEXT, DS : DGROUP, SS : DGROUP
```

它指示汇编程序把可能有的段_DATA、段 CONST、段_BSS 和段 STACK 集合成一个名为 DGROUP 的段组，同时指示汇编程序把数据段寄存器 DS 和堆栈段寄存器 SS 与段组 DGROUP 对应，使代码段寄存器 CS 与代码段对应。注意，在中型、大型模式下，代码段的段名不再是_TEXT。

由于伪指令.MODEL 的上述隐含动作，所以在使用伪指令.MODEL 后，可以直接引用段组 DGROUP，而且多数情况下也可以不使用伪指令 ASSUME，上面的程序 T8-1B.ASM 就是一例。

但在少数情况下，程序仍需要安排 ASSUME 语句来指示段寄存器与段的对应关系。例如：下列代码设置 DS，使它依次对应.DATA 段、.CODE 段、.FARDATA 段，最后又对应.DATA 段：

```
......
.DATA
......
.FARDATA
......
```

```
.CODE
MOV     AX,@DATA
MOV     DS,AX
ASSUME  DS:@DATA
……
MOV     AX,@CODE
MOV     DS,AX
ASSUME  DS:@CODE
……
MOV     AX,@FARDATA
MOV     DS,AX
ASSUME  DS:@FARDATA
……
MOV     AX,@DATA
MOV     DS,AX
ASSUME  CS:@DATA
……
```

2. 有关的预定义符

在上述程序片段中使用的符号@CODE 等是汇编程序提供的若干预定义符。它们类似于用伪指令 EQU 所定义的符号。与简化的段定义伪指令相关的一些预定义符号有：

(1) 符号@CODE 表示代码段的段名。

(2) 符号@DATA 表示由.DATA 段和.STACK 段等集合而成段组的组名。

(3) 符号@FARDATA 表示独立数据段的段名。

8.3 模块间的通信

一个程序的若干模块在功能上是有联系的，不仅程序的运行次序可能要从一个模块转到另一个模块，而且程序处理数据和变量也会涉及不同的模块。如何实现这种联系呢？具体地说，模块甲如何调用模块乙内的过程？模块乙如何访问模块甲内的数据？本节介绍这方面的内容。

8.3.1 伪指令 PUBLIC 和伪指令 EXTRN

由于各模块被单独汇编，所以，如果模块甲要按符号名调用或访问在其他模块内定义的某个过程或变量，那么，模块甲必须告诉汇编程序此指定符号名(标识符)在别的模块内定义，否则，在汇编模块甲时，汇编程序会给出"符号未定义"这类汇编出错信息。另一方面，如果在模块乙内定义的过程或变量准备供其他模块调用或访问，那么，模块乙也必须通知汇编程序，否则，汇编程序不会把相应的标识符保存到目标程序中去，最终导致连接失败。

伪指令 EXTRN 和伪指令 PUBLIC 就是分别用于通知汇编程序上述两种信息。

1. 伪指令 PUBLIC

伪指令 PUBLIC 用于声明在当前模块内定义的某些标识符是公共标识符，即可供其他模块使用的标识符。它的一般格式如下所示：

```
PUBLIC    标识符 [,标识符,...]
```

上述语句中位于助记符 PUBLIC 之后的“标识符”就是要声明的公共标识符。一条 PUBLIC 语句可声明多个这样的标识符，标识符间用逗号分隔。一个源程序模块内可使用多条 PUBLIC 语句。数据变量名和程序标号(包括过程名)均可声明为公用标识符。

例 1：如下源程序模块中声明 VAR1、VAR2 和 DELAY 为可供其它模块使用的公共标识符：

```
        .MODEL  SMALL
        PUBLIC  VAR1,VAR2       ;声明 VAR1 和 VAR2 是公共标识符
        PUBLIC  DELAY           ;声明 DELAY 为公共标识符
        .DATA                   ;数据段开始
VAR1    DW      ?
VAR2    DB      ?
VAR3    DB      5  DUP  (0)
        .CODE                   ;代码段开始
DELAY   PROC
LAB1:   RET
DELAY   ENDP
        END
```

由于没有声明 VAR3 和 LAB1 为公共标识符，所以其它模块不能使用这两个标识符。

2. 伪指令 EXTRN

伪指令 EXTRN 用于声明当前模块使用的哪些标识符在其他模块内定义。它的一般格式如下所示：

```
EXTRN  标识符:类型  [,标识符:类型, ...]
```

上述语句中位于助记符 EXTRN 后的每一项“标识符：类型”声明一个在其他模块内定义的标识符。汇编程序为了产生合适的代码或保留恰当的存储单元，要求在声明标识符的同时指出其类型属性，“标识符”和“类型”之间用冒号分隔。类型可以是 NEAR、FAR 或者 BYTE、WORD、DWORD 等标识符类型属性。

一条 EXTRN 语句可声明多个这样的标识符，每项之间用逗号分隔。一个源程序模块内可使用多条 EXTRN 语句。

例 2：下面的语句声明 VAR1、VAR2 和 DELAY 为在其它模块定义的标识符：

```
EXTRN  DELAY:NEAR
EXTRN  VAR1:WORD, VAR2:BYTE
```

注意：把 EXTRN 伪指令安排在段的里面与段的外面是有区别的。如果 EXTRN 伪

指令出现在某个段内，表示所声明的标识符虽在其它模块内，但却在同一个段内。如果EXTRN伪指令出现在段外，那么表示不知道所声明的标识符在哪一个段内被定义。

3. 声明一致性

各模块内的PUBLIC语句和EXTRN语句必须互相呼应，互相一致。凡是由PUBLIC语句声明的标识符，应该是其它模块的EXTRN语句中用到的标识符；反之，凡是由EXTRN语句声明的标识符必须在将要连接在一起的其它模块的PUBLIC语句中找到，而且所指明的类型必须一致。如果不遵守这些原则，就不能正确连接成功。

8.3.2 模块间的转移

模块间的转移是指从一个模块的某个代码段转移到另一个模块的某个代码段。这种转移通常是以过程调用及返回的形式出现，例如：模块甲调用定义在模块乙内的某个过程，但有时这种转移也直接采用转移指令的形式。

若两个模块的涉及转移的代码段在连接后不能组合为一个代码段，那么发生在这两个代码段之间的转移必须是段间转移，所以模块间的转移就成为远调用或远转移；否则模块间的转移可以是近调用或近转移，当然仍采用远调用或远转移也完全是可以的。

由于近调用或近转移的效率比远调用或远转移的效率高，所以一般总是乐意采用近调用或近转移。但是，这并非总做得到，因为分布在不同源程序模块中的两个代码段在连接时能被组合为一个段是有条件的，那就是它们的段名及其类别必须相同，而且段组合类型也应为PUBLIC。在实际编程时，不同的模块往往由不同的人员完成，所以很难做到段同名。为了避免考虑不同模块中的代码是否能组合成一个段，反而常常采用远调用或远转移。

例3：演示程序T8-7.ASM有如下三个模块组成，主模块中代码段的段名是CSEG，而两个从模块中代码段的段名是TEXT：

```
;程序名：T8-7.ASM
;功  能：演示模块间的转移
        EXTRN SUB1:FAR                          ;声明SUB1是在其它模块内定义
CSEG    SEGMENT   PARA   PUBLIC 'CODE'
        ASSUME    CS:CSEG
START:  CALL      FAR  PTR  SUB1                ;调用在T8-7MA中定义的过程SUB1
        MOV       AX,4C00H
        INT       21H
CSEG    ENDS
        END       START
;
;模块名：T8-7MA.ASM
;功  能：作为程序T8-7的一个模块
        PUBLIC  SUB1                            ;声明SUB1是公共标识符
        EXTRN   SUB2:NEAR                       ;声明SUB2是在其它模块内定义
TEXT    SEGMENT   PARA   PUBLIC 'CODE'
```

```
        ASSUME    CS:TEXT
SUB1    PROC      FAR                        ;定义远过程 SUB1
        MOV       DL,'*'
        MOV       AH,2                       ;显示符号"*"
        INT       21H
        CALL      SUB2                       ;调用在 T8-MA 中定义的过程 SUB2
        RET
SUB1    ENDP
TEXT    ENDS
        END
;
;模块名:T8-7MB.ASM
;功  能:作为程序 T8-7 的一个模块
        PUBLIC  SUB2                         ;声明 SUB2 是公共标识符
TEXT    SEGMENT   PARA   PUBLIC   'CODE'
        ASSUME    CS:TEXT
SUB2    PROC      NEAR                       ;定义近过程 SUB2
        MOV       DL,'+'
        MOV       AH,2                       ;显示符号"+"
        INT       21H
        RET
SUB2    ENDP
TEXT    ENDS
        END
```

由于模块 T8-7MA 中的代码段与主模块 T8-7 中的代码段不同名,所以连接时不能组合成一个段,因此过程 SUB2 被定义成远过程,在主模块 T8-7 中也相应地声明为 FAR 类型。由于,模块 T8-7MA 和模块 T8-7MB 中的两个代码段的段名和段类别相同,而且组合类型为 PUBLIC,所以,在连接时它们能被组合成一个段。再由于,只有过程 SUB1 调用过程 SUB2,即只有段内调用,因此过程 SUB2 才被定义为近过程,在模块 T8-7MA 中也相应地声明为 NEAR 类型。可用如下命令把它们的三个目标模块连接到一起:

```
TLINK  T8-7 + T8-7MA + T8-7MB
```

如果主模块也要调用过程 SUB2,那么应该把 SUB2 也定义为远过程。

采用简化的段定义可避免考虑段名是否相同,把有关的问题留给汇编程序解决。

例 4:利用简化的段定义改写程序 T8-7.ASM 的三个模块。

```
;程序名:T8-7A.ASM
;功  能:(略)
        EXTRN     SUB1:NEAR
        .MODEL    SMALL
        .CODE
```

```
START: CALL    SUB1
       MOV     AX,4C00H
       INT     21H
       END     START
;
;程序名: T8-7AMA.ASM
       PUBLIC  SUB1
       EXTRN   SUB2:NEAR
       .MODEL     SMALL
       .CODE
SUB1   PROC
       ......                            ;其它代码略
       CALL    SUB2
       RET
SUB1   ENDP
       END
;
;模块名: T8-7AMB.ASM
       PUBLIC  SUB2
       .MODEL   SMALL
       .CODE
SUB2   PROC
       ......                            ;代码略
SUB2   ENDP
       END
```

由于三个模块均是 SMALL 模型，所以连接后的代码在一个段内，因此 SUB1 和 SUB2 均被作为近过程对待。

8.3.3 模块间的信息传递

模块间的信息传递主要表现为模块间过程调用时的参数传递。在第 4 章介绍的过程调用参数传递原则和方法依然有效。少量参数可利用寄存器传递或利用堆栈传递，大量参数可先组织在一个缓冲区中，然后利用寄存器或堆栈传递相应的指针。

如果要利用约定的存储单元传递参数，情形稍稍复杂些，需要把它们声明为公共标识符。

例 5：写一个显示 DOS 版本号的程序。

为了展示模块间信息的传递，我们把程序分成两个模块。主模块有一个数据段和一个代码段，从模块只含有两个子程序。源程序如下所示：

```
;程序名: T8-8.ASM
;功  能: 演示模块间的利用寄存器和约定存储单元传递信息
DSEG    SEGMENT     PUBLIC  'DATA'
```

```
MESS    DB  'DOS Version is '
MESS1   DB  ?
        DB  '.'
MESS2   DB  2 DUP (?)
        DB  0DH,0AH,'$'
VERM    DB  0
VERN    DB  0
DSEG    ENDS
        ;声明 VERM 和 VERN 是公共标识符
        PUBLIC  VERM,VERN
        ;声明 GETVER 和 TODASC 在其它模块定义
        EXTRN  GETVER:FAR,TODASC:FAR
        ;
CSEG    SEGMENT  PUBLIC 'CODE'
        ASSUME  CS:CSEG,DS:DSEG
START:  MOV     AX,DSEG
        MOV     DS,AX
        ;
        CALL    GETVER                  ;取得 DOS 版本号;
        ;
        MOV     AL,VERM
        MOV     BX,LENGTH MESS1
        MOV     SI,OFFSET MESS1
        CALL    TODASC                  ;把主版本号转换成可显示形式
        MOV     AL,VERN
        MOV     BX,LENGTH MESS2
        MOV     SI,OFFSET MESS2
        CALL    TODASC                  ;把次版本号转换成可显示形式
        MOV     DX,OFFSET MESS
        MOV     AH,9
        INT     21H                     ;显示版本信息
        MOV     AX,4C00H
        INT     21H
CSEG    ENDS
        END     START
;
;模块名:T8-8MA.ASM
;功  能:作为程序 T8-8.ASM 的模块
        ;声明 GETVER 和 TODASC 为公共标识符
        PUBLIC  GETVER, TODASC
        ;声明 VERM 和 VERN 在其它模块定义
        EXTRN VERM:BYTE,VERN:BYTE
```

```
        ;
FUNC    SEGMENT   PUBLIC  'CODE'        ;定义代码段
        ASSUME    CS:FUNC
;子程序名:GETVER
;功    能:获取 DOS 版本号
;入口参数:无
;出口参数:在其它模块的 VERM 单元中存放主版本号
;         在其它模块的 VERN 单元中存放次版本号
;说    明:远过程
GETVER PROC     FAR
        MOV     AH,30H                  ;30HDOS 功能调用是
        INT     21H                     ;取 DOS 版本号
        MOV     VERM,AL                 ;AL 含主版本号
        MOV     VERN,AH                 ;AH 含次版本号
        RET
GETVER ENDP
;
;子程序名:TODASC
;功    能:把一个 8 位二进制数转换成相应十进制数的 ASCII 码串
;入口参数:AL=欲转换二进制数;BX=十进制数的最少位数
;         DS:SI=存放 ASCII 码串的缓冲区首地址
;出口参数:ASCII 码串在相应缓冲区中
;说    明:远过程
TODASC    PROC  FAR
        MOV     CL,10
TOASC1: XOR     AH,AH
        DIV     CL
        ADD     AH,30H
        MOV     [SI+BX-1],AH
        DEC     BX
        JNZ     TOASC1
        RET
TODASC    ENDP
FUNC      ENDS
          END
```

子程序 GETVER 把 DOS 的版本号直接填入在主模块中约定的单元 VERM 和 VERN 中。子程序 TODASC 的入口参数由寄存器传递,转换得到的十进制数 ASCII 码串直接写到主模块的指定缓冲区中。这两个子程序均被定义为远过程。

正确设置数据段或附加段寄存器是模块间正确传递信息的保证。在访问定义在其他模块的变量前,必须保证已设置好相应的段寄存器。例如:在调用 GETVER 之前必须正确设置数据段寄存器 DS,因为子程序 GETVER 在访问约定的变量时,认为数据段寄存

器已设置好。如有必要还可动态地改变段寄存器内容。

模块间传递信息的另一个方法是利用段覆盖，这个方法只适用于模块间传递信息。具体方法是：在两个模块中都定义一个同名同类别数据段，规定段组合类型是 COMMON；把要传递的数据(变量)安排在这两个数据段的相同位置上。由于这两个在不同模块中的数据段同名同类别，且组合类型是 COMMON，所以连接时它们就发生重叠。

例 6：写一个显示当前系统日期的程序。

为了简单化，显示的日期只含月和日。主模块有一个代码段和一个数据段，从模块也有一个代码段和数据段。源程序如下：

```
;程序名：T8-9.ASM
;功  能：演示利用段覆盖方法在模块间传递信息
EXTRN  GETDATE : FAR                    ;声明 GETDATE 在其它模块定义
;
DSEG   SEGMENT  COMMON                  ;定义一个具有 COMMON 类型的数据段
MESS   DB  'Current date is '
MESS1  DB  2  DUP (?)
       DB  '-'
MESS2  DB  2  DUP (?)
       DB  0DH,0AH,24H
DSEG   ENDS
;
CSEG   SEGMENT  PUBLIC                  ;代码段
       ASSUME  CS : CSEG,DS : DSEG
START: MOV     AX,DSEG
       MOV     DS,AX                    ;置数据段寄存器
       CALL    GETDATE                  ;调用 GETDATE 取日期
       MOV     DX,OFFSET  MESS
       MOV     AH,9                     ;显示日期信息
       INT     21H
       MOV     AX,4C00H
       INT     21H
CSEG   ENDS
       END     START
;
;模块名：T8-9MA.ASM
;功  能：作为 T8-9.ASM 的一部分
PUBLIC GETDATE                          ;声明 GETDATE 为公共标识符
;
DSEG   SEGMENT COMMON                   ;定义一个具有 COMMON 类型的数据段
MESS   DB  'Current date is '
MESS1  DB  2 DUP (?)
       DB  '-'
```

```
MESS2   DB   2 DUP (?)                     ;这部分数据与模块 T8-8.ASM 中完全相同
        DB   0DH,0AH,24H
YEAR    DW   ?                             ;这部分变量是另外加上的
MONTH   DB   ?
DAY     DB   ?
DSEG    ENDS
;
CSEG    SEGMENT  BYTE  PUBLIC              ;定义代码段
        ASSUME   CS:CSEG,DS:DSEG
;子程序名:GETDATE
;功    能:取得系统当前日期并把月日数转换成相应的十进制数 ASCII 串
;入口参数:无
;出口参数:ASCII 串填入约定缓冲区
;说    明:远过程
GETDATE       PROC FAR
              MOV  AH,2AH                  ;2AH 号 DOS 功能调用是
              INT  21H                     ;取系统当前日期
              MOV  YEAR,CX                 ;保存年数
              MOV  MONTH,DH                ;保存月数
              MOV  DAY,DL                  ;保存日数
              MOV  AL,MONTH
              MOV  BX,LENGTH MESS1         ;把月数转换成十进制数 ASCII 串
              MOV  SI,OFFSET MESS1
              CALL TODASC
              MOV  AL,DAY
              MOV  BX,LENGTH MESS2         ;把日数转换成十进制数 ASCII 串
              MOV  SI,OFFSET MESS2
              CALL TODASC
              RET
GETDATE       ENDP
;
TODASC      PROC     NEAR
    ;由于该子程序仅供当前段调用,所以规定为 NEAR 类型
    ;其他部分与 T8-8MA.ASM 中所列相同
TODASC      ENDP
CSEG    ENDS
        END
```

模块 T8-9MA 中的数据段比模块 T8-9.ASM 中的数据段多了若干变量,在段覆盖时,以最长的段为段的最后实际长度。但必须注意,要传递的数据变量必须安排在相同的位置。由于模块 T8-9MA 中含有要访问的数据段,所以过程 GETDATE 能够随便地访问想要访问的对象。

8.4 子程序库

子程序库能帮助程序员快速地编写出正确的程序，本节介绍如何建立子程序库和利用子程序库。

8.4.1 子程序库

把频繁使用的一组子程序的源代码集中存放在某个文件中，再通过 INCLUDE 伪指令把它与完成某个任务的源程序相结合，这样就能方便地利用这些子程序，而无需重新编写或编辑它们。这种方法能提高编写程序的效率，但有以下不足：其一，当前源程序中的标号或变量名等可能与被结合的子程序文件中的标号等发生冲突(即符号重新定义)；其二，由于是源程序结合，所以每次汇编都包括对子程序文件的汇编，增加了汇编时间。

采用模块化程序设计方法，把包含常用子程序的源程序文件改写成一个源程序模块，然后单独汇编它，于是就可形成一个常用子程序目标文件。完成某个具体任务的程序只要把所需调用子程序声明为在其他模块内定义，那么就能通过汇编，最后再与这个常用子程序目标文件相连接就能得到可执行程序。这个方法能克服上述源程序结合方法的不足，从而进一步提高编写程序的效率。由于被连接的每一目标文件的全部代码都会成为最终可执行程序的一部分，所以这个方法也有一个缺点：当前未使用到的但却属于常用子程序目标文件的子程序都会出现在最终的可执行程序中。库能克服这个缺点。

子程序库是子程序模块的集合。库文件中存放着子程序的名称，子程序的目标代码，以及连接过程所必需的重定位信息。当目标文件与库相连接时，LINK 程序只把目标文件所需要的子程序从库中找出来，并嵌入到最终的可执行程序中去，而不是把库内的全部子程序统统嵌入到可执行程序。所以，库与目标文件不同，子程序库能克服子程序目标文件的缺点。

8.4.2 建立子程序库

为了给调用者提供方便，库中的子程序应该提供统一的调用方法，所以需要遵守如下约定：

(1) 参数传递方法保持统一。

(2) 过程类型保持相同，即都为远过程或都为近过程。请特别注意，如果过程类型选择 NEAR，那么必须保证在连接时调用者所在段能与子程序所在段组合成一个段，为此，调用者所在段的段名和类别应该与子程序所在段的段名和类别相同，且组合类型同为 PUBLIC。

(3) 采用一致的寄存器保护措施和可能需要的堆栈平衡措施。

(4) 子程序名称规范。

建立子程序库的一般步骤如下：

(1) 确定库所含子程序的范围，即库准备包含哪些子程序。

(2) 确定参数传递方法。

(3) 确定子程序类型，还确定子程序所在段的段名、定位类型、组合类型和类别。

(4) 确定寄存器保护措施等其他内容。

(5) 利用专门的库管理工具程序，把经过调试的子程序目标模块逐一加入到库中。

下面我们来建立一个子程序库，它包含若干数制转换子程序目标模块。为了方便地使库中各子程序目标模块所在的段相同，在编写各子程序模块源程序时，采用简化的段定义，并把存储模型定为 SMALL；但子程序类型规定为 FAR，即均为远过程，于是，调用模块只要把欲调用的子程序声明为在其它模块内定义，且类型为 FAR 即可，而无需考虑是否能与被调用的子程序模块组合成同一个段。此外，还作如下约定：采用寄存器传递出入口参数；各子程序除了可能破坏 AX、BX、CX 和 DX 四个寄存器的内容外，保护其他寄存器的内容。

例 1：编写一个把二进制数转换为对应十进制数 ASCII 码串的子程序，并把它添加到名为 BDHL.LIB 的库中。

源程序如下所示：

```
;源文件名：T8L1.ASM
;功    能：(略)
            PUBLIC   BDASCS              ;声明 BDASCS 为公共标识符
            .MODEL   SMALL
            ;
            .CODE                        ;代码段开始
;子程序名：BDASCS
;功    能：(略)
;入口参数：AX=欲转换的二进制数
;          DS：DX=缓冲区首地址
;出口参数：(略)
;说    明：(1) 远过程
;          (2) 缓冲区至少长 5 个字节
BDASCS      PROC     FAR
            PUSH     SI
            MOV      SI,DX
            MOV      CX,5
            MOV      BX,10
@@1:        XOR      DX,DX
            DIV      BX
            ADD      DL,30H
            MOV      [SI+4],DL
            DEC      SI
            LOOP     @@1
            POP      SI
            RET
BDASCS      ENDP
```

```
        END
```

设源程序存放在文件 T8L1. ASM 中，可利用如下命令汇编成目标模块且添加到库 BDHL. LIB 中去：

```
    TASM    T8L1
    TLIB    BDHL.LIB +T8L1.OBJ
```

例 2：编写一个把二进制数转换为对应十六进制数 ASCII 码串的子程序，并把它添加到名为 BDHL. LIB 的库中。

源程序如下所示：

```
;源文件名：T8L2.ASM
;功    能：(略)
PUBLIC  BHASCS              ;声明 BHASCS 是公共标识符
        .MODEL  SMALL
        .CODE               ;代码段开始
;先定义一个内部使用的过程(子程序说明信息等略)
HTOASC  PROC    NEAR
        AND     AL,0FH
        ADD     AL,90H
        DAA
        ADC     AL,40H
        DAA
        RET
HTOASC  ENDP
;
;子程序名：BHASCS
;功    能：(略)
;入口参数：AX=欲转换的二进制数
;       DS:DX=缓冲区首地址
;出口参数：(略)
;说    明：(1) 远过程
;          (2) 缓冲区至少长 4 个字节
BHASCS  PROC    FAR
        PUSH    DI
        PUSH    ES
        CLD
        PUSH    DS
        POP     ES
        MOV     DI,DX
        MOV     CX,404H
@@1:    ROL     AX,CL
        MOV     DX,AX
```

```
        CALL    HTOASC
        STOSB
        MOV     AX,DX
        DEC     CH
        JNZ     @@1
        POP     ES
        POP     DI
        RET
BHASCS  ENDP
        END
```

子程序 BHASCS 调用 HTOASC 实现把一位十六进制数转换为对应 ASCII 码，但由于没有把标识符 HTOASC 声明为公共标识符，所以它不能供其它程序调用。由于子程序 HTOASC 与子程序 BHASCS 在同一个模块，且只供 BHASCS 调用，所以采用类型 NEAR。子程序 HTOASC 的代码始终伴随着子程序 BHASCS，当某个程序需要连入子程序 BHASCS 时，子程序 HTOASC 也被连入。

设源程序存放在文件 T8L2. ASM 中，可利用如下命令把它汇编成目标模块且添加到库中去：

```
TASM    T8L2
TLIB    BDHL + T8L2
```

8.4.3 使用举例

下面我们举例说明如何利用子程序库。

例 3：写一个显示 16H 号中断向量的程序。

使用简单的段定义方式，存储模型定为 SMALL。实现算法是：取有关中断向量；再调用已建立的库 BDHL. LIB 中的子程序 BHASCS 把向量值转换为对应十六进制数的 ASCII 码串；最后显示 ASCII 码串。源程序如下所示：

```
;程序名：T8-10.ASM
;功  能：(略)
VECTOR = 16H
        .MODEL  SMALL
        .STACK  1024
        .DATA
MESS    LABEL BYTE
MESS1   DB  4  DUP (0)
        DB ':'
MESS2   DB   4  DUP (0)
        DB  0DH,0AH,24H
        .CODE
        EXTRN      BHASCS:FAR              ;类型为 FAR,但认为在相同段内
```

```
START: MOV     AX,@DATA
       MOV     DS,AX                    ;设置数据段寄存器
       MOV     AH,35H
       MOV     AL,VECTOR
       INT     21H                      ;返回中断向量于 ES:BX 中
       PUSH    BX                       ;保存中断向量中的偏移部分
       MOV     AX,ES                    ;先转换中断向量的段值部分
       MOV     DX,OFFSET MESS1
       CALL    FAR PTR BHASCS           ;转换
       POP     AX                       ;中断向量的偏移部分送 AX
       MOV     DX,OFFSET                MESS2
       CALL    FAR PTR BHASCS           ;转换
       MOV     DX,OFFSET MESS
       MOV     AH,9                     ;显示中断向量
       NT      21H
       MOV     AX,4C00H
       INT     21H
       END     START
```

例 4：写一个显示系统常规内存量的程序。

系统常规内存量存放在内存单元 40:13H 的字单元中，以 KB 为单位。

实现算法是：先从 40:13H 单元中取得内存量；再调用库 BDHL.LIB 中的子程序 BDASCS 把以二进制数形式表示的内存量转换为对应十进制数的 ASCII 码串；最后显示之。程序采用完整的段定义方式。源程序如下所示：

```
;程序名：T8-11.ASM
;功  能：(略)
        EXTRN    BDASCS:FAR                ;认为 BDASCS 不在相同段内
SSEG    SEGMENT  STACK 'STACK'             ;定义堆栈段
        DB  400H  DUP (0)
SSEG    ENDS
;
CSEG    SEGMENT  PUBLIC                    ;数据代码合为一段
MESS    DB   'Total = '
MESS1   DB   5 DUP (0)
        DB   'KB',0DH,0AH,24H
        ASSUME  CS : CSEG,DS : CSEG        ;段寄存器使用设定
START:  PUSH     CS
        POP      DS
        MOV      AX,40H
        MOV      ES,AX
        MOV      AX,ES:[13H]               ;取常规内存量
```

```
        MOV     DX,OFFSET MESS1
        CALL    BDASCS              ;转换
        MOV     DX,OFFSET MESS
        MOV     AH,9                ;显示
        INT     21H
        MOV     AX,4C00H
        INT     21H
CSEG    ENDS
        END     START
```

8.5 编写供 Turbo C 调用的函数

将C语言和汇编语言混合使用的传统方式是,先用C语言和汇编语言编写出独立的模块,然后编译C语言模块并汇编汇编语言模块,最后再将得到的目标模块连接到一起。普通情况下,这种汇编语言模块由若干被频繁调用而左右程序运行效率的子程序所组成。在C语言中,习惯上把子程序称为函数,所以,我们说这种汇编语言模块由若干汇编函数组成。

为了使C模块能够调用到汇编模块中的函数,在编写汇编模块时,必须注意两个方面的内容。第一,汇编模块必须能够恰当地与C模块连接到一起,并且其中的汇编函数名等要符合C语言的约定;第二,汇编函数必须能恰当地处理C风格的函数调用,包括访问传递过来的参数、返回值及遵守C函数所要求的寄存器保护规则。

8.5.1 汇编格式的编译结果

为了能够编写出可供 Turbo C 调用的函数,应了解 Turbo C 模块与汇编模块的接口机制,而从以汇编形式给出的编译结果中可方便地了解这种机制。设有如下C程序:

```
/* 程序名:TC8.C      */
int     Sum(int,int,int);         /*说明函数 Sum 的调用格式*/
int     xxx=5;                    /*已初始化的变量*/
int     yyy;                      /*未初始化的变量*/

main()                            /*主函数*/
{
    yyy=Sum(1,xxx,3);
    printf("%d\n",yyy);
}

    int Sum(int i,int j,int m)    /* 函数 Sum */
{
    return(i+j+m);
}
```

用下面的命令要求 Turbo C 按 SMALL 模式编译 TC8. C，并以汇编格式输出编译结果：

```
TCC  -ms  -S  TC8.C
```

以汇编格式输出的编译结果保存在文件 TC8. ASM 中，尽管该文件比较冗长，但阅读和理解它对编写供 Turbo C 调用的汇编函数是有帮助的，而且对学习 C 语言也是有益的。TC8. ASM 的主要内容如下所示(已删去空段和注释等次要内容)：

```
_TEXT    segment  byte  public  'CODE'          ;代码段
DGROUP      group    _DATA,   _BSS
         assume   cs : _TEXT, ds : DGROUP, ss : DGROUP
_TEXT    ends

_DATA    segment  word  public  'DATA'          ;已初始化的数据段
_xxx     label    word
         dw       5
_DATA    ends

_TEXT    segment  byte  public  'CODE'          ;代码段
_main    proc     near
         mov      ax,3
         push     ax                            ;为调用 Sum 压入第三个参数
         push     word ptr DGROUP:_xxx          ;压入第二个参数
         mov      ax,1
         push     ax                            ;压入第一个参数
         call     near ptr _Sum                 ;调用 Sum
         add      sp,6                          ;废除压入堆栈的三个参数
         mov      word ptr DGROUP:_yyy,ax       ;把和送变量 yyy
         push     word ptr DGROUP:_yyy          ;为调用 printf 压入第二个参数
         mov      ax,offset DGROUP:s@
         push     ax                            ;压入第一个参数
         call     near ptr _printf              ;调用 printf
         pop      cx                            ;废除压入堆栈的两个参数
         pop      cx
@1:      ret
_main    endp

_Sum     proc     near                          ;函数 Sum
         push     bp
         mov      bp,sp
         mov      ax,word  ptr  [bp+4]          ;访问第一个参数
         add      ax,word  ptr  [bp+6]          ;访问第二个参数
         add      ax,word  ptr  [bp+8]          ;访问第三个参数
```

```
        jmp     short  @2
@2:     pop     bp
        ret
_Sum    endp
_TEXT   ends

_BSS    segment word  public  'BSS'            ;未初始化数据段
_yyy    label   word
        db  2  dup (?)
_BSS    ends

DATA    segment word  public  'DATA'           ;数据段
s@      label   byte                           ;也即字符串"%d\n"
        db      37
        db      100
        db      10
        db      0
DATA    ends

TEXT    segment byte   public  'CODE'          ;代码段
        extrn   _printf:near                   ;printf 是 Turbo C 的库函数
TEXT    ends

        public  _yyy                           ;声明公共标识符
        public  _xxx
        public  _main
        public  _Sum
        end
```

从上面以汇编格式给出的编译结果中，可以看到函数 Sum 除包含一条多余的跳转指令外，已足够精练。但为了方便地说明如何编写供 Turbo C 调用的函数，我们仍然假设希望把上述 C 函数 Sum 改写成汇编函数。相应地，C 程序 TC8.C 改写如下：

```
/* 程序名:CA8.C     */
extern    Sum(int,int,int);        /* 声明函数 Sum 在其他模块内定义 */
int     xxx=5;                     /* 已初始化的变量 */
int     yyy;                       /* 未初始化的变量 */

main()                             /* 主函数 */
{
    yyy=Sum(1,xxx,3);
    printf("%d\n",yyy);
}
```

程序 CA8. C 与 TC8. C 的区别是不再定义函数 Sum，而声明它在其他模块内定义。下面就介绍如何编写包含函数 Sum 的汇编模块。

8.5.2 汇编模块应该遵守的约定

1. 关于内存模式和段的约定

为了使得汇编模块能够恰当地与 Turbo C 模块连接到一起，汇编模块必须采用与 Turbo C 模块一致的内存模式，同时必须遵守与 Turbo C 兼容的段命名约定。

如果采用完整的段定义形式，那么汇编模块中所需段的定义应该按照 TC8. ASM 中相应段的定义形式来编写。这似乎有点麻烦。

幸运的是，利用简化的段定义方式，能轻松地实现内存模式的一致和兼容的段命名。因为采用简化的段定义方式后，汇编程序就自动完成这方面的全部工作。

根据表 8.1 所列内容，只要用伪指令. MODEL 说明内存模式为 SMALL，然后用伪指令. CODE 定义的代码段以及用伪指令. DATA 定义的数据段等均与 TC8. ASM 中相应段的定义一致。换句话说，在汇编模块中安排伪指令“. MODEL SMALL”，那么汇编后所得到的目标模块就能够与 Turbo C 按 SMALL 模式编译后所得的目标模块有效地连接。事实上，不仅仅在 SMALL 模式下如此，在其他内存模式下也如此。

伪指令. MODEL 通知汇编程序，用简化的段定义伪指令创建的段与选定的内存模式兼容，并控制用 PROC 伪指令创建的过程的隐含类型。另一方面，由伪指令. MODEL 定义的内存模式与具有同样类型的 Turbo C 模式是相互兼容的。所以，简化的段定义伪指令. CODE、. DATA、. DATA?、. FARDATA、. FARDATA? 及. CONST 等产生的段与 Turbo C 相应的段兼容。

2. 关于函数名的约定

一般情况下，Turbo C 希望所有的外部标号均以下划线“_”开头。Turbo C 自动地给函数名及全局变量名（包括外部变量名）加上下划线，在上述 TC8. ASM 中可清楚地看到这一点。所以，如果汇编模块中定义的函数准备提供给 Turbo C 调用，那么函数名必须以下划线开头。顺便指出，如果汇编模块中定义的变量也准备供 Turbo C 访问，那么也需以下划线开头。

通常情况下，在处理符号名时，汇编程序对字母的大小写并不敏感，所以不区别对待大写字母和小写字母，而皆以大写字母对待。因为 C 语言区别对待大小写字母，所以在编写准备与 C 模块相连接的汇编模块时，应该注意符号名的大小写，以便保持一致。而且，要通知汇编程序对大小写区别对待，至少对于 C 模块和汇编模块所共享的那些符号而言应该如此。汇编程序的命令行可选项/ml 和/mx 可以做到这一点。

汇编程序的命令行可选项/ml 使得汇编对所有符号均按大小写区别对待。命令行可选项/mx 使得汇编只对公共标识符和外部标识符等按大小写区别对待。

至此，我们可编写出如下格式的汇编模块：

```
.MODEL  SMALL
.CODE
PUBLIC  _Sum
```

```
_Sum  PROC
      ……
      ……
_Sum  ENDP
      END
```

8.5.3 参数传递和寄存器保护

下面介绍用汇编语言编写的函数如何与 Turbo C 交流信息，这包括三个方面的内容：获取由调用者提供的入口参数、把可能的处理结果值返回给调用者以及寄存器的保护。换一个角度看，这三个方面的内容也就是 Turbo C 调用函数的一般方法。

1. 获取入口参数

我们在 4.2.3 节中介绍了如何利用堆栈传递参数及访问堆栈中的参数。在知道了参数类型（占用堆栈空间的字节数）和次序后，就能够通过 BP 寄存器方便地访问堆栈中的参数。

Turbo C 通过堆栈将参数传递给函数。调用函数之前，Turbo C 先将要传给函数的参数压入堆栈，最先压入最右边的参数，最后压入最左边的参数。

在 TC8.C 中的 C 语句"yyy=Sum(1,xxx,3);"被编译成如下汇编指令：

```
mov   ax,3                          ;L1
push  ax                            ;L2(先压入最右边的参数)
push  word  ptr  DGROUP:_xxx        ;L3(然后压入变量 xxx 的值)
mov   ax,1                          ;L4
push  ax                            ;L5(最压入最左边的参数)
call  near  ptr  _Sum               ;L6
add   sp,6                          ;L7(废除压入堆栈的三个参数)
mov   word  ptr  DGROUP:_yyy,ax     ;L8
```

从中可清楚地看到，先压入最右边的参数 3，再压入变量 xxx 的值，最后压入最左边的参数 1。

从函数返回时，先前压入堆栈的参数仍然保留在堆栈中，但这些参数已没有任何用途。所以，在每次调用函数之后，Turbo C 立即调整堆栈指针，使之指向压入参数前所指的位置，这样就放弃了堆栈中的参数。在上面的例子中，3 个双字节的参数共占 6 个字节的堆栈空间，所以在调用 Sum 之后，第 L7 条指令将堆栈指针加 6 以删除这些参数。这里很重要的一点就是，堆栈中的参数由调用者负责删除。

那么，如何访问堆栈中的参数呢？先来看看 TC8.C 中的函数 Sum 是如何访问堆栈中的参数的：

```
push  bp                        ;S1
mov   bp,sp                     ;S2
mov   ax,word  ptr  [bp+4]      ;S3(访问第一个参数)
add   ax,word  ptr  [bp+6]      ;S4(访问第二个参数)
```

```
add    ax,word  ptr  [bp+8]      ;S5(访问第三个参数)
```

在执行完上面的第 L6 条指令后，堆栈顶如图 8.2(a)所示，在把参数依次压入堆栈后进行函数调用时，返回地址也被压入堆栈。为了通过 BP 寄存器访问堆栈中的参数，先保护 BP 寄存器。在执行完上面的第 S1 和第 S2 条指令后，堆栈顶如图 8.2(b)所示。

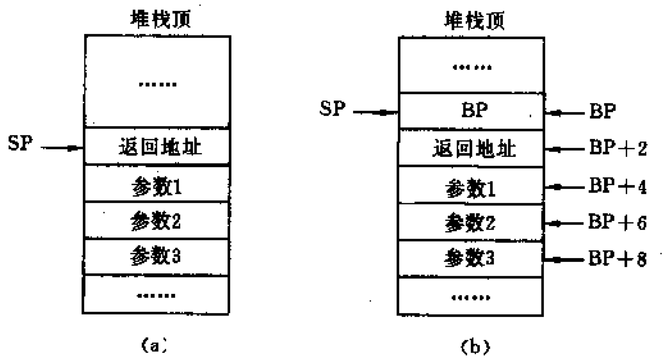


图 8.2　堆栈内容变化示意图

直接用汇编语言编写的函数 Sum 当然也能够按此方法访问堆栈中的参数。

但并非所有情形都如此简单。首先，相对于 BP 以常偏移量访问参数的做法并不理想，这不仅仅因为不易于阅读和理解，而且容易搞错偏移量，特别是如果增加要传递的参数，那么偏移量又得重新调整。更为严重的是：如果采用远调用，那么保存的返回地址要占用 4 个字节的堆栈空间；此外，压入堆栈的参数所占用堆栈空间的字节数与参数类型有关，例如：如果参数是一个长整数或长指针，则将占用 4 字节的堆栈空间。为此，TASM 还提供了一条伪指令 ARG 来帮助程序员处理好访问堆栈中参数的问题。

2. 返回值

象普通 C 函数一样，供 Turbo C 调用的汇编函数也可利用寄存器返回值。通常情况下，8 或 16 位的值通过 AX 寄存器返回，32 位的值通过 DX:AX 寄存器对返回，其中高 16 位在 DX 寄存器中。所以，可通过 AX 返回短指针，通过 DX:AX 返回长指针。在上述 TC8.ASM 中，可清楚地看到函数 Sum 的返回值由寄存器 AX 传递出来。

现在，我们可编写出如下的含有函数 Sum 的汇编模块：

```
;模块名：A8.ASM
;内  容：含一个供 CA8.C 调用的函数 Sum
        .MODEL  SMALL
        .CODE
        PUBLIC _Sum
PARM1   EQU     [BP+4]
PARM2   EQU     [BP+6]
PARM3   EQU     [BP+8]
_Sum    PROC
        PUSH    BP
```

```
        MOV     BP,SP
        MOV     AX,PARM1        ;访问第一个参数
        ADD     AX,PARM2        ;访问第二个参数
        ADD     AX,PARM3        ;访问第三个参数
        POP     BP
        RET     ;由 AX 寄存器返回结果
_Sum    ENDP
        END
```

通过下面的命令能够完成对 CA8.C 的编译、对 A8.ASM 的汇编，最后再连接到一起：

```
TCC   -ms   CA8  A8.ASM
```

通过下面的三条命令能够分别编译、汇编和连接：

```
TCC    -ms  -c  CA8
TASM   /ml  A8
TLINK  c0s  CA8  A8,  CA8,  ,cs
```

上述最后连接命令中的 c0s 和 cs 分别是 Turbo C 的 SMALL 模式下的启动代码目标模块文件和函数库文件。

必须注意，上面的汇编模块 A8.ASM 只能与按 SMALL 模式编译的 C 模块相连接。

3. 保护寄存器

Turbo C 要求供它调用的汇编语言函数必须保护好寄存器 BP、SP、CS、DS 和 SS 的内容。尽管在汇编语言函数中可以改变这些寄存器的内容，但当返回时，它们的值必须与调用前相同。可以随意地改变寄存器 AX、BX、CX、DX 和 ES 及标志寄存器的内容。

寄存器 SI 和 DI 是特殊情况，因为 Turbo C 将其用作寄存器变量。如果在调用汇编语言函数的 C 模块中启动了寄存器变量，那么在汇编语言函数中就必须保护寄存器 SI 和 DI；但若没有启动寄存器变量，就不必保护这两个寄存器。一个稳妥的做法是，象保护寄存器 BP 它们那样，总是保护寄存器 SI 和 DI。

8.5.4 举例

例 1：写一个求若干 16 位有符号数之和的汇编语言函数。它有两个入口参数，其一是数组的元素个数，其二是指向数组的指针。它的返回值是一个 32 位有符号数。在 C 模块中说明的原型格式是：

```
extern    long    niadd(int * ,int);
```

适用于 SMALL 模式编译的 C 模块相连的汇编模块如下所示：

```
;模块名：A86.ASM
;功  能：(略)
PARM          STRUC
```

```
REGBP       DW    ?
RETADDR     DW    ?
POINTER     DW    ?
COUNT       DW    ?
PARM        ENDS
;
            .MODEL    SMALL
            .CODE
PUBLIC  _niadd
_niadd      PROC
            PUSH    BP
            MOV     BP,SP
            PUSH    SI
            PUSH    DI
            CLD
            MOV     SI,[BP].POINTER
            MOV     CX,[BP].COUNT
            XOR     BX,BX
            MOV     DI,BX
@@1:        LODSW
            CWD
            ADD     BX,AX
            ADC     DI,DX
            LOOP    @@1
            MOV     DX,DI
            MOV     AX,BX
            POP     DI
            POP     SI
            POP     BP
            RET
_niadd      ENDP
            END
```

源程序中说明了一个结构数据类型，它能够反映在保护寄存器 BP 之后堆栈顶的内容，但并没有真正地分配结构变量，在获取入口参数时，可以设想在堆栈顶分配了这样的一个结构变量，这样可方便表达堆栈中的参数。另外，作为入口参数给出的缓冲区首地址只含有偏移，而没有段值，这是因为这个函数只准备提供给按 SMALL 模式编译的 C 模块调用，所以认为在调用该函数之前数据段寄存器已设置好，而无需由函数再设置。

调用上述函数的一个 Turbo C 程序如下所示：

```
extern    long    niadd(int * ,int);
```

```
int   buffer[6]={12345,10000,-20000,23456,-2345,-56};

main()
{
    long    x;
    x=niadd(buffer,6);
    printf("x=%ld\n",x);
}
```

例 2：改写上面的 A86.ASM，使其具有良好的通用性，并把所得模块添加到相应的 Turbo C 函数库中去。

汇编模块 A86.ASM 中的函数 niadd 只能供以 SMALL 模式编译的 C 模块调用，其原因除了伪指令.MODEL 说明采用 SMALL 模式外，更主要的还在于只考虑了近调用和短指针，例如：在 LARGE 模式下，C 模块将以远调用方式调用函数 niadd，并且传递给函数的指针也是长指针。

改写后的模块如下所示：

```
;程序名：A86A.ASM
;功  能：(略)
     .MODEL SMALL
     .CODE
     PUBLIC _niadd
PARM        STRUC
REGBP       DW  ?
            IF  @CODESIZE   EQ  0
RETADDR     DW ?
            ELSE
RETADDR     DD ?
            ENDIF
            IF  @DATASIZE   EQ  0
POINTER     DW ?
            ELSE
POINTER     DD ?
            ENDIF
COUNT       DW ?
PARM        ENDS
;
_niadd  PROC
        PUSH    BP
        MOV     BP,SP
        PUSH    SI
        PUSH    DI
```

```
        CLD
        IF      @DATASIZE   EQ  0
        MOV     SI,[BP].POINTER
        ELSE
        PUSH    DS
        LDS     SI,[BP].POINTER
        ENDIF
        MOV     CX,[BP].COUNT
        XOR     BX,BX
        MOV     DI,BX
@@1:    LODSW
        CWD
        ADD     BX,AX
        ADC     DI,DX
        LOOP    @@1
        MOV     DX,DI
        MOV     AX,BX
        IF      @DATASIZE   NE  0
        POP     DS
        ENDIF
        POP     DI
        POP     SI
        POP     BP
        RET
_niadd  ENDP
        END
```

上述汇编模块中使用了预定义符号@CODESIZE 和@DATASIZE，它们的值决定了条件汇编语句中的条件是否满足。当利用.MODEL 伪指令把内存模式设定为SMALL 或 COMPACT 时，预定义符号@CODESIZE 的值为 0；当内存模式为 MEDIUM、LARGE 或 HUGE 时，预定义符号@CODESIZE 的值为 1。当内存模式为 SMALL 或 MEDIUM 时，预定义符号@DATASIZE 的值为 0；当内存模式为 COMPACT 或 LARGE 时，@DATASIZE 的值为 1；当内存模式为 HUGE 时，@DATASIZE 的值为 2。

另外，在使用.MODEL 伪指令设定内存模式后，缺省的过程类型会根据内存模式的不同而改变，在中、大或巨模式下，缺省过程类型是 FAR，而不再一直 NEAR。

汇编程序模块 A86A.ASM 考虑了各种内存模式的情况，所以较为通用。如果要使它与按某种模式编译的 C 模块相连接，那么只要把由伪指令.MODEL 指定的内存模式修改为这种希望的内存模式，再重新汇编即可。同样在重新汇编后，也就可把它添加到相应模式的库函数中。

利用库管理工具 TLIB，可以直接把汇编后所得的目标模块添加到 Turbo C 的相应函数库中去。如指定的内存模式为 SMALL，那么可通过如下命令把目标模块添加到

SMALL 模式的库 CS.LIB 中：

```
TASM   /mx   A86A
TLIB   CS.LIB + A86A.OBJ
```

如果欲把它添加到 LARGE 模式的库 CL.LIB 中，则可先把伪指令.MODEL 指定的模式修改为 LARGE，然后重新汇编它，最后发出如下命令：

```
TLIB   CL.LIB + A86A.OBJ
```

8.6 习　　题

题 8.1　在完整的段定义语句中，有三个可选项：定位类型、组合类型和类别。请简述它们的意义和作用。

题 8.2　请编写简单的测试程序进一步理解上述可选项的意义。

题 8.3　请编写一个代码、数据和堆栈同段的演示程序。

题 8.4　请举例说明段组的使用方法。

题 8.5　简化的段定义有何特点？请举例说明。

题 8.6　请简述汇编语言程序设计时，模块间的通信方法。

题 8.7　如何实现模块之间的转移？请举例说明。

题 8.8　何为子程序库？请简述子程序库的作用。

题 8.9　如何建立子程序库？

题 8.10　请谈谈连接程序的功能。

题 8.11　请用 BC 或 TC 编写一个显示内存单元内容的小型实用程序。显示过程用汇编语言编写。

题 8.12　请编写一个求三角函数值的小型实用程序。利用 BC 或 TC 的库函数获得三角函数值。

第二部分 提 高 部 分

第 9 章 80386 程序设计基础

80386 是 x86 微处理器家族发展中的里程碑，它不仅兼容先前的 8086/8088 和 80286 等微处理器，而且也为后来的 80486、Pentium 和 Pentium Pro 等微处理器奠定了基础。80386 支持实方式和保护方式两种运行模式。在实方式下，80386 相当于一可进行 32 位处理的快速 8086，原先为 8086/8088 设计的程序几乎都可适用于 80386。当开机或者经硬件 RESET 线重新初始化时，80386 处于实方式下。本章介绍 80386 寄存器、寻址方式和指令集，这些内容是进行 80386 程序设计的基础，对实方式和保护方式都有效。

9.1 80386 寄存器

80386 寄存器的宽度大多是 32 位，可分为如下几组：通用寄存器、段寄存器、指令指针及标志寄存器、系统地址寄存器、控制寄存器、调试寄存器和测试寄存器。应用程序主要使用前三组寄存器，只有系统程序才会使用各种寄存器。这些寄存器是 x86 系列微处理器

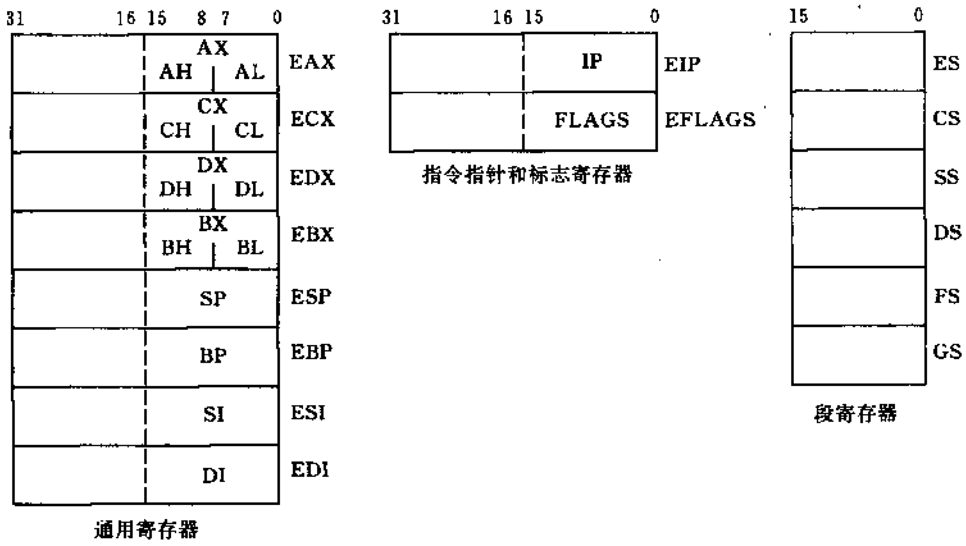


图 9.1 80386 的部分寄存器

先前成员(8086/8088、80186 和 80286)寄存器的超集,所以 80386 包含了先前微处理器的全部 16 位寄存器。8086/8088 没有系统地址寄存器和控制寄存器等寄存器。

9.1.1 通用寄存器

80386 有 8 个 32 位通用寄存器,这 8 个寄存器分别定名为 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI。它们是原先的 16 位通用寄存器的扩展,请参见图 9.1。这些通用寄存器的低 16 位可以作为 16 位的寄存器独立存取,并把它们分别定名为 AX、CX、DX、BX、SP、BP、SI 和 DI,它们也就是 x86 系列微处理器先前成员的 8 个 16 位通用寄存器。在存取这些 16 位的寄存器时,相应的 32 位通用寄存器的高 16 位不受影响。与先前的微处理器一样,AX、BX、CX 和 DX 这 4 个 16 位的数据寄存器的高 8 位和低 8 位可以被独立存取,分别命名为 AH、AL、BH、BL、CH、CL、DH 和 DL。在存取这些 8 位寄存器时,相应的 16 位寄存器的其它位不受影响,相应的通用寄存器的其它位也不受影响。由此可见,80386 在扩展先前微处理器寄存器组时,为执行 8086 和 80286 等微处理器代码提供了兼容的寄存器组。

这些 32 位通用寄存器不仅可以传送数据、暂存数据、保存算术或逻辑运算的结果,而且还可以在基址和变址寻址时,存放地址。例如:

```
MOV    EAX,12345678H
MOV    [EBX],EAX
ADD    EAX,[EBX+ESI+1]
MOV    AL,[ECX+EDI+1234]
SUB    CX,[EAX-12]
```

在先前的微处理器中,只有 BX、BP、SI 和 DI 可以在基址和变址寻址时存放地址,而现在 80386 的 8 个 32 位通用寄存器都可以作为指针寄存器使用,所以说这些 32 位通用寄存器更具有通用性。

9.1.2 段寄存器

80386 有 6 个 16 位段寄存器,分别定名为 CS、SS、DS、ES、FS 和 GS。在实方式下,代码段寄存器 CS、堆栈段寄存器 SS、数据段寄存器 DS 和附加段寄存器 ES 的功能与先前微处理器中对应段寄存器的功能相同。FS 和 GS 是 80386 新增加的段寄存器。因此 80386 上运行的程序可同时访问多达 6 个段。

在实方式下,内存单元的逻辑地址仍是"段值:偏移"形式。为了访问一个给定内存段中的数据,可直接把相应的段值装入某个段寄存器中。例如:

```
MOV    AX,SEG BUFFER
MOV    FS,AX
MOV    AX,FS:[BX]
```

在保护方式下,情况要复杂得多,装入段寄存器的不再是段值,而是称为选择子的某个值。有关内容在第 10 章介绍。

9.1.3 指令指针和标志寄存器

80386的指令指针和标志寄存器分别是先前微处理器的指令指针IP和标志寄存器FLAG的32位扩展。

1. 指令指针寄存器

80386的指令指针寄存器扩展到32位，记为EIP。EIP的低16位是16位的指令指针IP，它与先前微处理器中的IP相同。IP寄存器提供了用于执行8086和80286代码的指令指针。由于实方式下段的最大范围是64K，所以EIP中的高16位必须是0，仍相当于只有低16位的IP起作用。

2. 标志寄存器

80386的标志寄存器也扩展到32位，记为EFLAGS，如图9.2所示。与8086/8088的16位标志寄存器相比，增加了4个控制标志，其他标志位的位置和意义均与8086/8088相同。下面简单介绍这4个控制标志，它们在实方式下不发挥作用。

(1) IO特权标志IOPL(I/O Privilege Level)

IO特权标志有2位宽，也称为IO特权级字段。IOPL字段指定了要求执行I/O指令的特权级。如果当前的特权级别在数值上小于或等于IOPL，那么I/O指令可执行，否则发生一个保护异常。在80286的16位标志寄存器中，已含有该标志。

(2) 嵌套任务标志NT(Nested Task)

嵌套任务标志控制中断返回指令IRET的执行。如果NT=0，用堆栈中保存的值恢复EFLAGS、CS和EIP执行常规的从中断返回的动作。如果NT=1，通过任务转换实现中断返回。在80286的16位标志寄存器中，已含有该标志。

(3) 重启动标志RF(Restart Flag)

重启动标志控制是否接受调试故障。RF=0接受，RF=1忽略。在成功地完成每一条指令后，处理器把RF清0。而当接收到一个非调试故障时，处理器把RF置1。

(4) 虚拟8086方式标志VM(Virtual 8086 Mode)

如果该标志置为1，处理器将在虚拟的8086方式下工作，如果清0，处理器工作在一般的保护方式下。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

图9.2 80386标志寄存器

9.2 80386存储器寻址

80386支持先前微处理器的各种寻址方式。在立即寻址方式和寄存器寻址方式中，操作数可达32位宽。在存储器(内存)寻址方式中，不仅操作数可达32位，而且寻址范围和

方式更加灵活。本节介绍 80386 存储器寻址方式。

9.2.1 存储器寻址基本概念

80386 继续采用分段的方法管理主存储器。存储器的逻辑地址由段基地址(段起始地址)和段内偏移两部分表示,存储单元的地址由段基地址加上段内偏移所得。段寄存器指示段基地址,各种寻址方式决定段内偏移。

在实方式下,段基地址仍然是 16 的倍数,段的最大长度仍然是 64K。段寄存器内所含的仍然是段基地址对应的段值,存储单元的物理地址仍然是段寄存器内的段值乘上 16 加上段内偏移。所以,尽管 80386 有 32 根地址线,可直接寻址物理地址空间达到 4G 字节,但在实方式下仍然与 8086/8088 相似。

在保护方式下,段基地址可以长 32 位,无须是 16 的倍数,段的最大长度可达 4G。段寄存器内所含的是指示段基地址的选择子,存储单元的地址是段基地址加上段内偏移,但不再是段寄存器之值乘 16 加上偏移,这与 8086/8088 完全不同。

段寄存器指示段基地址。在实方式下,段寄存器含段值,直接指示段基地址;在保护方式下,段寄存器含选择子,间接指示段基地址。每次对存储器的访问或是隐含地、或是显式地、或是默认地指定了某个段寄存器。由于 80386 有 6 个段寄存器,所以在某一时刻程序可访问 6 个段,而不再是先前的 4 个段。

由段寄存器 CS 所指定的段称为当前代码段。80386 在取指令时,自动引用代码段寄存器 CS。80386 的指令指针寄存器是 EIP。通常可以认为 CS 和 EIP 指示下一条要执行的指令。在实方式下,由于段的最大长度不超过 64K,所以 EIP 的高 16 位为 0,相当于 IP。

由段寄存器 SS 所指定的段称为当前堆栈段。80386 在访问堆栈时,总是引用堆栈段寄存器 SS。在实方式下,80386 把 ESP 的低 16 位 SP 作为指向栈顶的指针,可以认为堆栈顶由 SS 和 SP 指定。在保护方式下,32 位堆栈段的堆栈指针是 ESP,16 位堆栈段的堆栈指针是 SP。如果要访问存储在堆栈中的数据,也可以通过引用 SS 段寄存器进行。

DS 寄存器是主要的数据段寄存器,对于访问除堆栈外的数据段它是一个默认的段寄存器。在以 BP 或 EBP 或 ESP 作为基址寄存器访问堆栈时,默认的段寄存器是 SS。某些字符串操作指令总是使用 ES 段寄存器作为目标操作数的段寄存器。此外,尽管 CS、SS、ES、FS 和 GS 都可作为访问数据时引用的段寄存器,但必须显式地在指令中指定,它们也即成为段超越前缀,这使得指令在长度和执行时间上的开销稍大一些。

例如:

```
MOV    EAX,[SI]          ;默认段寄存器 DS
MOV    [BP+2],EAX        ;默认段寄存器 SS
MOV    AL,FS:[BX]        ;显式指定段寄存器 FS
MOV    GS:[BP],DX        ;显式指定段寄存器 GS
```

一般说来,使 DS 含有最经常访问的数据段的段值,而用 ES、FS 和 GS 含有那些不经常使用的数据段的段值。

当然,32 位数据仍按“高高低低”原则存取。指令“MOV EAX,[1234H]”的传送示意如图 9.3 所示。

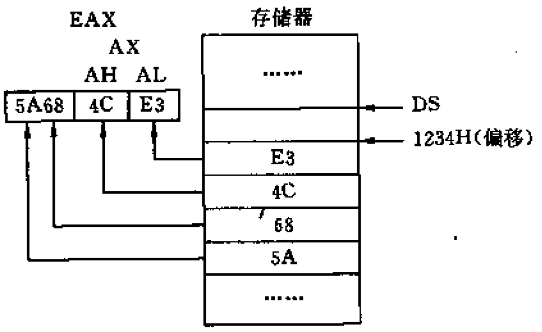


图 9.3　存取 32 位数据示意图

9.2.2　灵活的存储器寻址方式

各种存储器寻址方式表示的是有效地址，也即段内偏移。在实方式下，段内偏移不能超过 64K；在保护方式下，段内偏移可以超过 64K。80386 既支持各种先前 16 位偏移的寻址方式，又增加了灵活的 32 位偏移的寻址方式。

80386 支持先前微处理器所支持的各种存储器寻址方式。先前微处理器的存储器寻址方式可分为五种，其中相对基址加变址寻址方式最复杂。相对基址加变址寻址方式中的 16 位有效地址（偏移）由三部分相加构成：一个基址寄存器（只能是 BX 或 BP 寄存器）、一个变址寄存器（只能是 SI 或 DI 寄存器）和一个常数偏移量（最大 16 位）。在这三部分中如果没有常数偏移量，那么就成为基址加变址寻址；如果没有基址寄存器或者变址寄存器，那么就成为寄存器相对寻址；如果只有基址寄存器或者只有变址寄存器，没有相对偏移量，那么就成为寄存器间接寻址；如果既没有基址寄存器也没有变址寄存器，只剩下常数偏移量，那么就成为直接寻址。80386 支持上述各种 16 位偏移的存储器寻址方式。

80386 还支持 32 位偏移的存储器寻址方式。80386 允许内存地址的偏移可以由三部分内容相加构成：一个 32 位基址寄存器，一个可乘上比例因子 1、2、4 或 8 的 32 位变址寄存器，及一个 8 位或 32 位的常数偏移量。并且这三部分可省去任意的两部分。例如：

```
MOV    AL,ES:[5678H]
MOV    CX,[EBX]
MOV    EDX,[EBX+EDI+1234H]
MOV    AX,[EBX+ESI*4]
MOV    ESI,[EBX*8+100H]
MOV    BH,ES:[EBX+EDI*8+6]      ;显式指定段寄存器 ES
```

如果含变址寄存器，那么变址寄存器中的值先按给定的比例因子放大，再加上偏移。

在这些寻址方式中，8 个 32 位通用寄存器都可作为基址寄存器使用，除了 ESP 寄存器外，其他 7 个通用寄存器都可作为变址寄存器使用。例如：

```
MOV    AL,[ECX]
MOV    BX,[EAX-4]
```

```
MOV     [EDX+EDI],CX
MOV     [EBX+EAX*2],DH
MOV     ESI,[EAX+ECX*8+1234H]
```

在所有寻址方式中，对数据的访问所默认引用的段寄存器取决于所选择的基址寄存器。如果基址寄存器是 ESP 或者 EBP，那么默认的段寄存器从通常的 DS 改为 SS。对于别的基址寄存器的选择，包括没有基址寄存器的情况，DS 仍然是默认的段寄存器。如 9.2.1 节中所述，访问由非默认的段寄存器指定的某个段的数据，要使用一额外的指令字节来指定所要的段寄存器。当 EBP 作为变址寄存器使用（ESP 不能作为变址寄存器使用）时，不影响默认段寄存器的选择。默认段寄存器的选择只受所选的基址寄存器所影响。例如：

```
MOV     AL,[EBX+EBP*2]          ;默认的段寄存器是 DS
MOV     AL,[EBX+EBP]            ;默认的段寄存器是 DS
MOV     AL,[EBP+EBX]            ;默认的段寄存器是 SS
MOV     AL,GS:[EBP*2]           ;显式指定段寄存器 GS
MOV     EAX,[ESP]               ;默认的段寄存器是 SS
MOV     AL,CS:[ESP+2]           ;显式指定段寄存器 CS
MOV     [ESP+EBP*2],ECX         ;默认的段寄存器是 SS
MOV     AL,DS:[ESP+EDI+12]      ;显式指定段寄存器 DS
```

80386 支持的 32 位偏移的存储器寻址方式可归纳如下：

$$\begin{Bmatrix} \text{无} \\ EAX \\ ECX \\ EDX \\ EBX \\ ESP \\ EBP \\ ESI \\ EDI \end{Bmatrix} + \begin{Bmatrix} \text{无} \\ EAX \\ ECX \\ EDX \\ EBX \\ \cdots \\ EBP \\ ESI \\ EDI \end{Bmatrix} \times \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{Bmatrix} + \begin{Bmatrix} \text{无} \\ 8\text{位} \\ 32\text{位} \end{Bmatrix}$$

要特别说明的是，在实方式下，也可使用上述 32 位偏移的存储器寻址方式，但所得偏移不应超过 0FFFFH，而且操作数的最高字节单元的地址偏移也不能超过 0FFFFH。原因是实方式下段的长度是 64K。但有一种特殊情况，可使得段超过 64K，在这种特殊情况下，就要使用 32 位的偏移来访问超过偏移 64K 部分的存储单元。

如果某一存储器操作数的地址是该操作数尺寸（长度）的倍数，那么称该操作数是对齐的。当段基地址是 16 的倍数时，存储器操作数是否对齐就取决于偏移。例如：如果一个字的偏移量是 2 的倍数，那么该字为对齐的；再如：一个双字的偏移量是 4 的倍数，那么该双字是对齐的。在 80386 中，如果存储器操作数是对齐的，那么访问它就比较快。当然，不对齐的操作数也能访问，但所费时间可能多一点点。程序员应该尽量使操作数对齐，以提高访问存储器操作数的速度。

9.2.3 支持各种数据结构

80386 支持的"基地址＋变址＋位移量"寻址方式能进一步满足高级语言支持的数据结构的需要。标量变量、记录、数组、记录的数组和数组的记录等数据结构可方便地利用 80386 的这种寻址方式实现。对 FORTRAN 而言，这些数据结构可作为 Static 存储来分配。对 Pascal 或者 C 而言，这些数据结构可以动态地分配在一个程序的堆栈或者堆中。基地址和变址寄存器为寻址方式提供两个动态的成分，而位移量提供静态的成分。用在一数据段中的一个常数位移量可以简单地寻址静态分配的数据。用一相对于 ESP 或者 EBP 寄存器的常数位移量可以寻址分配在堆栈中的数据。表 9.1 给出了高级语言的需要和 80386 支持的寻址方式之间的关系。

表 9.1 数据结构和 80386 寻址方式

存储类型	结构类型	地址方式分类	例子
静态	标量	位移	DS：[1000]
	数组	变址＋位移	DS：[ESI * 4＋1004]
	记录	位移	
	记录的数组	变址＋位移	
	数组的记录	变址＋位移	
堆栈	标量	基地址＋位移	
	数组	基地址＋变址＋位移	
	记录	基地址＋位移	
	记录的数组	基地址＋变址＋位移	
	数组的记录	基地址＋变址＋位移	SS：[ESP＋EDI * 2＋100]
堆	标量	基地址	
	数组	基地址＋变址	
	记录	基地址＋位移	
	记录的数组	基地址＋变址＋位移	
	数组的记录	基地址＋变址＋位移	

为了简化寄存器的分配，可以用 8 个通用寄存器中的一个作为基地址寄存器，以及用除 ESP 外的一个变址寄存器。变址寄存器的值可以直接地被使用(比例因子为 1)，或者按 2、4 或 8 的倍率用于 16 位、32 位、64 位变量变址，而不需要计算位移的指令或使用一额外的寄存器。

9.3 80386 指令集

80386 的指令集包含了 8086/8088、80186 和 80286 指令集。可分为如下大类：数据传送指令、算术运算指令、逻辑运算和移位指令、控制转移指令、串操作指令、高级语言支持指令、条件字节设置指令、位操作指令、处理器控制指令和保护方式指令。高级语言支持指令始于 80186。保护方式指令始于 80286。条件字节设置指令和位操作指令等是 80386 新

增的。

80386是32位处理器，80386指令的操作数长度可以是8位、16位或者32位。80386认为，32位操作数是对16位操作数的扩展。80386既支持16位的存储器操作数有效地址，又支持32位的存储器操作数有效地址。80386认为，32位存储器操作数有效地址是对16位存储器操作数有效地址的扩展。所以，80386支持的32位操作数的指令往往就是对相应支持16位操作数指令的扩展；80386的32位存储器操作数有效地址寻址方式往往就是对16位存储器操作数有效地址寻址方式的扩展。

本节在第2章介绍的8086/8088指令集基础上介绍80386指令集，有关保护方式的指令在第10.8节中介绍。

9.3.1 数据传送指令

数据传送指令实现在寄存器、内存单元或I/O端口之间传送数据和地址。80386的数据传送指令仍分成四种：通用数据传送指令、累加器专用传送指令、地址传送指令和标志传送指令。

1. 通用传送指令组

80386的通用传送指令组含有如下十条指令：数值传送指令MOV、符号扩展指令MOVSX、零扩展指令MOVZX、交换指令XCHG、进栈指令PUSH、PUSHA、PUSHAD、退栈指令POP、POPA和POPAD。

(1) 数值传送指令MOV

数值传送指令MOV的格式、功能和使用注意点都与8086/8088的MOV指令相同。传送的数据可以是8位、16位或32位。例如：

```
MOV   EAX,12345678H
MOV   ESI,EDI
MOV   [BX+SI+1],EBX
MOV   CL,[EAX+EBX+3]
MOV   FS,AX
MOV   ES:[ECX+EDX*4],DX
MOV   GS:DWORD PTR [1234H],4321H
```

(2) 符号扩展指令MOVSX和零扩展指令MOVZX(始于80386)

符号扩展指令的格式如下：

```
MOVSX   DST,SRC
```

该指令的功能是把源操作数SRC的内容送到目的操作数DST，目的操作数空出的位用源操作数的符号位填补。

零扩展指令的格式如下：

```
MOVZX   DST,SRC
```

该指令的功能是把源操作数SRC的内容送到目的操作数DST，目的操作数空出的位用零填补。

符号扩展指令和零扩展指令中的目的操作数DST必须是16位或32位寄存器，源操作数SRC可以是8位或16位寄存器，也可以是8位或16位存储器操作数。如果源操作数和目的操作数都是字，那么就相当于MOV指令。

这两条指令不影响各标志。

例如：

```
MOV     DL,92H
MOVSX   AX,DL                    ;92H扩展成FF92H送AX
MOVSX   EBX,DL                   ;92H扩展成FFFFFF92H送EBX
MOVZX   CX,DL                    ;92H扩展成0092H送DX
MOVZX   ESI,DL                   ;92H扩展成00000092H送ESI
MOV     WORD PTR [BX],1234H
MOVSX   ESI,WORD PTR [BX]        ;1234H扩展成00001234H送ESI
MOVZX   EDI,WORD PTR [BX]        ;1234H扩展成00001234H送EDI
```

符号扩展指令MOVSX和零扩展指令MOVZX是80386新增的指令。使用MOVSX可以对有符号数进行扩展，显然MOVSX指令要比CBW指令和CWD指令的功能强。使用MOVZX可以方便地对无符号数进行扩展。

(3) 交换指令XCHG

交换指令XCHG的格式、功能和使用注意点都与8086/8088的XCHG指令相同。交换的数据可以是8位、16位或32位。例如：

```
XCHG    ESI,EDI
XCHG    [EBX+ESI*2+1],BX
XCHG    CL,[EAX+EBX+3]
```

(4) 进栈指令PUSH

进栈指令PUSH的格式没变，但功能增强了。从80186开始，压入堆栈的操作数还可以是立即数。从80386开始操作数长度还可以达32位，当然如果操作数长度是32位，那么堆栈指针减4。

例如：

```
PUSH    EAX
PUSH    DWORD PTR [BX]
PUSH    DWORD PTR [EAX]
PUSH    FS
PUSH    1234H                    ;16位立即数
PUSH    12345678H                ;32位立即数
```

在调用通过堆栈传递入口参数的子程序时，把立即数直接压入堆栈的操作能方便地把常量作为参数传递给子程序。例如：

```
PUSH    0F000H                   ;压入立即数
PUSH    0                        ;压入立即数
```

```
CALL    ECHOBD              ;调用子程序
ADD     SP,4                ;平衡堆栈
```

有一点要注意：当用 PUSH 指令把堆栈指针 SP 或 ESP 压入堆栈时，80386/80286 的处理方式不同于 8086/8088。8086/8088 是将 SP 减 2 后的值进栈，而 80386/80286 是将进栈操作前的 SP(或 ESP)值进栈。

(5) 出栈指令 POP

出栈指令 POP 的格式、功能和使用注意点都没变，除允许弹出 32 位操作数外。例如：

```
POP     EAX
POP     WORD PTR    [ECX]
POP     DWORD PTR   [BX]
```

(6) 16 位通用寄存器全进栈指令 PUSHA 和全出栈指令 POPA(始于 80186)

PUSHA 指令和 POPA 指令提供了压入或弹出 8 个 16 位通用寄存器的有效手段，它们的一般格式如下：

```
PUSHA
POPA
```

PUSHA 指令将所有 8 个通用寄存器(16 位)内容压入堆栈，其顺序是：AX、CX、DX、BX、SP、BP、SI、DI，然后堆栈指针寄存器 SP 之值减 16，所以 SP 进栈的内容是 PUSHA 执行之前的值。

POPA 指令从堆栈弹出内容以 PUSHA 相反的顺序送到这些通用寄存器，从而恢复 PUSHA 之前的寄存器内容。但堆栈指针寄存器 ESP 之值不是由堆栈弹出，而是通过增加 16 来恢复。

这两条指令不影响标志。这两条指令都没有显式的操作数。

在中断处理程序和子程序中，利用这两条指令能快速地进行现场保护和恢复。例如：

```
SUBX        PROC
            PUSHA
            ……
            ……
            POPA
            RET
SUBX        ENDP
```

尽管 PUSHA 指令比 8 个独立的 PUSH 指令快，但它比 3 或 4 个独立的 PUSH 指令要慢，同时还将使 SP 减 16，所以，如果只需要保存部分寄存器，那么仍以使用 PUSH 指令为妥。对 POPA 和 POP 指令而言也是如此。

(7) 32 位通用寄存器全进栈指令 PUSHAD 和全出栈指令 POPAD(始于 80386)

PUSHAD 指令和 POPAD 指令提供了压入或弹出 8 个 32 位通用寄存器的有效手段，它们的一般格式如下：

```
PUSHAD
```

```
POPAD
```

PUSHAD 指令将所有 8 个通用寄存器(32 位)内容压入堆栈，其顺序是：EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI，然后堆栈指针寄存器 ESP 之值减 32，所以 ESP 进栈的内容是 PUSHAD 执行之前的值。

POPAD 指令从堆栈弹出内容以 PUSHAD 相反的顺序送到这些通用寄存器，从而恢复 PUSHAD 之前的寄存器内容。但堆栈指针寄存器 SP 之值不是由堆栈弹出，而是通过增加 32 来恢复。

这两条指令不影响各标志。这两条指令都没有显式的操作数。

这两条指令是 PUSHA 和 POPA 指令的扩展。

2. 地址传送指令组

(1) 装入有效地址指令 LEA

装入有效地址指令的格式和功能没变。源操作数仍然必须是存储器操作数，目的操作数是 16 位或者 32 位通用寄存器。当目的操作数是 16 位通用寄存器时，那么只装入有效地址的低 16 位。这符合实方式下的实际应用需要。

例如：

```
MOV     EBX,12345678H             ;置 EBX 之值
MOV     ECX,87654321H             ;置 ECX 之值
LEA     ESI,[EBX+ECX+1234H]       ;执行后 ESI=9999ABCDH
LEA     DX,[EBX+ECX+1234H]        ;执行后 DX=0ABCDH
LEA     EDI,[BX-3]                ;执行后 EDI=00005675H
```

利用该指令还可以进行简单的算术运算，例如：

```
LEA     ECX,[EAX+EDX*2+1234H] ;ECX=EAX+EDX*2+1234H
LEA     EDX,[EBX*8]               ;DX=EBX*8
```

(2) 装入指针指令组

装入指针指令组有 5 条指令，它们的格式如下：

```
LDS     REG,OPRD
LES     REG,OPRD
LFS     REG,OPRD                  ;始于 80386
LGS     REG,OPRD                  ;始于 80386
LSS     REG,OPRD                  ;始于 80386
```

这些指令的功能将源操作数 OPRD 所指内存单元的 4 个或 6 个相继字节单元的内容送到指令助记符给定的段寄存器和目的操作数 REG 中。目的操作数必须是 16 位或 32 位(始于 80386)通用寄存器，源操作数是存储器操作数。如果目的操作数是 16 位通用寄存器，那么源操作数 OPRD 含 32 位指针。如果目的操作数是 32 位通用寄存器，那么源操作数 OPRD 含 48 位指针，这适合于使用 32 位偏移的场合，可用于一次装载 48 位全指针。

这些指令不影响各标志。

用 LDS、LES、LFS 和 LGS 指令装入完整的指针是很方便的。例如：

```
STRING DB      'HELLO'
PPSTR  DD      STRING
FPSTR  DF      STRING
WPSTR  DW      STRING
       DW      SEG STRING
       ……
       LFS     DI,PPSTR
       LGS     ESI,FPSTR
       LES     DI,DWORD  PTR  WPSTR
```

用 LSS 指令装载堆栈指针是简单安全的方法，例如：

```
LSS    SP,SSPTR          ;SSPTR 是含有堆栈指针的双字
```

它确保在一条指令中使 SS 和 SP 都被重置。

3. 标志传送指令组

80386 的标志传送指令组含有如下 6 条指令：LAHF、SAHF、PUSHF、PUSHFD、POPF 和 POPFD。

指令 LAHF、SAHF、PUSHF 和 POPF 指令格式和功能等均与 8086/8088 相同。

32 位标志寄存器进栈和出栈指令的格式如下，它们始于 80386：

```
PUSHFD
POPFD
```

PUSHFD 指令将整个标志寄存器的内容压入堆栈；POPFD 指令将栈顶的一个双字弹出到 32 位的标志寄存器中。这两条指令是 PUSHF 和 POPF 指令的扩展。

PUSHFD 指令不影响各标志；POPFD 指令将影响各标志。

4. 累加器专用传送指令组

80386 累加器专用指令组含有如下指令：IN、OUT 和 XLAT。

输入指令 IN 的格式和功能与 8086/8088 相同，但可以通过累加器 EAX 输入一个双字。例如：

```
IN     EAX,DX        ;从 DX 规定的端口输入一个双字
IN     EAX,20H       ;从 20H 口输入一个双字
```

输出指令 OUT 的格式和功能与 8086/8088 相同，但可以通过累加器 EAX 输出一个双字。例如：

```
OUT    DX,EAX        ;输出一个双字到 DX 规定的端口
OUT    20H,EAX       ;输出一个双字到 20H 口
```

表转换指令 XLAT 的格式和功能与 8086/8088 相同。但从 80386 开始存放基值的寄存器可以是 EBX。也就是说，扩展的 XLAT 指令以 EBX 为存放基值的寄存器，非扩展的 XLAT 指令以 BX 为存放基值的寄存器。

9.3.2 算术运算指令

80386 算术运算指令的操作数可以扩展到 32 位，此外与 8086/8088 相比还增强了有符号数乘法指令的功能。

1. 加法和减法指令组

加法和减法指令组含有如下 8 条指令：ADD、ADC、INC、SUB、SBB、DEC、CMP 和 NEG。

除了这些指令的操作数可以扩展到 32 位外，其他均与 8086/8088 相同。例如：

```
ADD    EAX,ESI
ADC    EAX,DWORD  PTR  [BX]
INC    EBX
SUB    ESI,4
SBB    DWORD  PTR  [EDI],DX
DEC    EDI
CMP    EAX,EDX
NEG    ECX
```

2. 乘法和除法指令组

乘法和除法指令组含有 4 条指令：MUL、DIV、IMUL 和 IDIV。

(1) 无符号数乘法和除法指令

无符号数乘法和除法指令的格式没变。指令中只给出一个操作数，自动根据给出的操作数确定另一个操作数。当指令中给出的源操作数为字节或字时，它们与 8086/8088 的情形相同。

在源操作数为双字的情况下，乘法指令 MUL 默认的另一个操作数是 EAX，其功能是把 EAX 内容乘上源操作数内容所得积送入 EDX：EAX 中，若结果的高 32 位(在 EDX 中)为 0，那么标志 CF 和 OF 被清 0，否则被置 1；除法指令 DIV 默认的被除数是 EDX：EAX，其功能是把指令中给出的操作数作为除数，所得的商送 EAX，余数送 EDX。

例如：

```
MUL     DWORD  PTR  [BX+3]
DIV     EBX
```

(2) 有符号数乘法和除法指令

原有的有符号数乘法指令 IMUL 和除法指令 IDIV 继续保持，但操作数可扩展到 32 位。当操作数为 32 位时的情形与无符号数乘法、除法指令相同。

从 80186 开始，还提供了新形式的有符号数乘法指令。一般格式如下：

```
IMUL    DST,SRC
IMUL    DST,SRC1,SRC2
```

上述第一种格式是将目的操作数 DST 与源操作数 SRC 相乘，结果送到目的操作数 DST 中；第二种格式是将 SRC1 与 SRC2 相乘，结果送目的操作数 DST 中。

其中，目的操作数只能是16位通用寄存器或32位通用寄存器；第一格式中的源操作数SRC的长度必须与目的操作数的长度相同(8位立即数除外)，可以是通用寄存器、存储单元或立即数；第二格式中的源操作数SRC1只能是通用寄存器或存储单元，并且长度必须与目的操作数的长度相同，源操作数SRC2只能是立即数。例如：

```
IMUL    EAX,10
IMUL    AX,BX,12
IMUL    DX,[SI],-2
IMUL    EAX,DWORD  PTR  [EBX+ESI*2+3456H],5
```

实际上，第一种格式是第二种格式的特殊情形。例如：

```
IMUL    AX,7
IMUL    AX,AX,7
```

对于这两种新增的乘法指令，由于存放积的目的操作数的长度与被乘数(或乘数)的长度相同，因此积有可能溢出。如果积溢出，那么高位部分将被丢掉，而置标志CF和OF为1来表示溢出；否则清标志CF和OF。所以，在这样的乘法指令后可安排检测OF的条件转移指令，用于处理积溢出的情况。

由于存放积的目的操作数的长度与乘数的长度相同，而有符号数或无符号数的乘积的低位部分是相同的。所以，这种新形式的乘法指令对无符号数和有符号数的处理是相同的。这也是只用一条IMUL指令就可以代表新形式乘法指令的原因。

在80186和80286中，没有32位通用寄存器可使用，此外第一种格式中的源操作数只能是立即数。

3. 符号扩展指令组

80386的符号扩展指令组有如下4条指令：CBW、CWD、CWDE和CDQ。

符号扩展指令CBW和CWD的功能均无变化。

符号扩展指令CWDE和CDQ是80386新增的指令。它们的格式如下：

```
CWDE
CDQ
```

指令CWDE将16位寄存器AX的符号位扩展到32位寄存器EAX的高16位中。该指令是指令CBW的扩展。

指令CDQ将寄存器EAX的符号位扩展到EDX的所有位。该指令是指令CWD的扩展。

这些指令均不影响各标志。

4. 十进制调整指令组

十进制调整指令组含有如下6条指令：DAA、DAS、AAA、AAS、AAM和AAD。这些指令的功能与8086/8088相同，均无变化。

9.3.3 逻辑运算和移位指令

80386的逻辑运算和移位指令包括逻辑运算指令、一般移位指令、循环移位指令和双

精度移位指令。

1. 逻辑运算指令组

逻辑运算指令组含有如下 5 条指令:NOT、AND、OR、XOR 和 TEST。

除了这些指令的操作数可以扩展到 32 位外,其他均与 8086/8088 相同。例如:

```
NOT     EAX
AND     EDX,0FF00FF00H
XOR     EAX,EAX
OR      ESI,[BX]
TEST    EAX,ES:[1234H]
```

2. 一般移位指令组

一般移位指令组含有如下 3 条指令:SAL/SHL、SAR 和 SHR。算术左移指令 SAL 和逻辑左移指令 SHL 是相同的。

从 80386 开始,操作数可扩展到 32 位。尽管这些指令的格式没变,但移位位数的表达增强了。从 80186 开始,移位指令中的移位位数不仅可以是 1 或者 CL,也可以是一个 8 位立即数。

例如:

```
SHL     AL,4
SHR     EAX,12
SAR     WORD  PTR  [SI],3
```

从 80386 开始,实际移位位数等于指令中指定的移位位数的低 5 位,所以移位位数的变化范围是 0 至 31。

CF 总是保留着目的操作数最后被移出去的位的值。但对逻辑左移指令 SHL 而言,当移位位数大于等于被移位的操作数长度时,CF 被置 0。

在移位位数仅是 1 的情况下,当移位前后的目的操作数的符号位相同时,那么溢出标志位 OF 被置 0。

这些移位指令还会影响标志 ZF、SF 和 PF,但对标志 AF 无定义。

如下的程序片段实现把 DX 的内容乘 16 再加上 AX 的内容,结果保存在 DL:AX 中,设它们为无符号数:

```
PUSH    DX
SHL     DX,4
ADD     AX,DX
POP     DX
PUSHF
SHR     DX,12
POPF
ADC     DL,0
```

如下的指令片段利用算术右移指令 SAR 实现除数为 2 的 n(设为 5)次方的除法,为

了保证所得的商与利用 IDIV 指令后所得的商相同，在被除数是负数的情况下，先把负数加 2^n-1：

```
        OR    EAX,EAX
        JGE   NOADJ
        LEA   EAX,[EAX+31]
NOADJ:  SAR   EAX,5
```

3. 循环移位指令组

循环移位指令组含有如下 4 条指令：ROL、ROR、RCL 和 RCR。

从 80386 开始，操作数可扩展到 32 位。与一般移位指令一样，从 80186 开始，循环移位指令中的移位位数也可以是一个 8 位的立即数。

例如：

```
ROL   AH,4
ROR   ESI,12
RCL   AX,3
RCR   ECX,4
```

从 80386 开始，对循环指令 ROL 和 ROR 而言，实际移位的位数将根据被移位的操作数的长度取 8、16 或 32 位的模；对带进位的循环移位指令 RCL 和 RCR 而言，移位位数先取指令中规定的移位位数的低 5 位，再根据被移位的操作数的长度取 9、17 或 32 位的模。

循环移位指令影响 CF 和 OF。CF 总是保留着操作数最后被移出去的位的值。在移位位数仅是 1 的情况下，当移位前后的目的操作数的符号位相同时，那么 OF 被置 0。

例：设要将寄存器 AX 的每一位依次重复一次，所得 32 位数保存在寄存器 EAX 中。如下程序片段实现这一要求：

```
        MOV   CX,16
        MOV   BX,AX
NEXT:   SHR   AX,1
        RCR   EDX,1
        SHR   BX,1
        RCR   EDX,1
        LOOP  NEXT
        MOV   EAX,EDX
```

4. 双精度移位指令组(始于 80386)

始于 80386 的双精度移位指令组含有两条指令：SHLD 和 SHRD。

双精度移位指令的一般格式如下：

```
SHLD   OPRD1,OPRD2,m
SHRD   OPRD1,OPRD2,m
```

其中，操作数 OPRD1 可以是 16 位通用寄存器、16 位存储单元、32 位通用寄存器或

者 32 位存储单元;操作数 OPRD2 的长度必须与操作数 OPRD1 的长度一致,并且只能是 16 位通用寄存器或者 32 位通用寄存器;m 是移位位数,或者为 8 位立即数,或者为 CL。

双精度左移指令 SHLD 的功能是把操作数 OPRD1 左移指定的 m 位,空出的位用操作数 OPRD2 高端的 m 位填补,(类似基于 OPRD1:OPRD2 的左移),但操作数 OPRD2 的内容不变,最后移出的位保留在进位标志 CF 中。如果只移一位,当进位标志和最后的符号位不一致时,置溢出标志 OF,否则清 OF。

例如:

```
MOV     AX,8321H
MOV     DX,5678H
SHLD    AX,DX,1        ;AX=0642H,DX=5678H,CF=1,OF=1
SHLD    AX,DX,2        ;AX=1909H,DX=5678H,CF=0,OF=0
```

双精度右移指令 SHRD 的功能是把操作数 OPRD1 右移指定的 m 位,空出的位用操作数 OPRD2 低端的 m 位填补,(类似基于 OPRD1:OPRD2 的右移),但操作数 OPRD2 的内容不变,最后移出的位保留在进位标志 CF 中。当移位位数为 1 时,OF 标志受影响。否则清 OF。

例如:

```
MOV     EAX,01234867H
MOV     EDX,5ABCDEF9H
SHRD    EAX,EDX,4        ;EAX=90123486H,CF=0,OF=1
SHRD    EAX,EDX,8        ;EAX=F9901234H,CF=1,OF=0
```

双精度移位指令还影响标志 ZF、SF 和 PF,对 AF 无定义。

双精度移位指令的实际移位的位数取指令中规定的移位位数 m 的 32 位的低 5 位,所以移位位数的变化范围是 0~31。如果移位的位数是 0,那么双精度移位指令就相当于空操作指令 NOP。如果移位位数超过被移位操作数 OPRD1 的长度,那么操作数 OPRD1 和各标志均无定义。

如下一条指令可实现把 EAX 中的 32 位数,保存到寄存器对 DX:AX 中:

```
SHLD    EDX,EAX,16
```

9.3.4 控制转移指令

控制转移指令可分为如下四组:转移指令、循环指令、过程调用和返回指令、中断调用和中断返回指令。这些控制转移指令的非扩展形式的功能保持与 8086/8088 相同。由于保护方式下 80386 的段长可超过 64K,所以,这些控制转移指令扩展后涉及的段内偏移可达 32 位。在采用 32 位表示段内偏移时,段间转移的目的地址采用 48 位全指针形式表示。但在实方式下,最大的段长仍是 64K,因此,即使段内偏移用 32 位表示,实际的偏移值也禁止超过 64K。

1. 转移指令组

(1) 无条件转移指令

无条件转移指令 JMP 在分为段内直接、段内间接、段间直接和段间间接四类的同时，还具有扩展形式。扩展的无条件转移指令的转移目的地址偏移采用 32 位表示，段间转移目的地址采用 48 位全指针形式表示。

在实方式下，无条件转移指令 JMP 的功能几乎没有提高。尽管 80386 的无条件转移指令允许把 32 位的段内偏移送到 EIP，但在实方式下段最大 64K，段内偏移不能超过 64K，所以不需要使用 32 位的段内偏移。例如：

```
JMP    EAX          ;有效，但实方式下 EAX 不能超过 64K
```

在保护方式下，段内无条件转移指令的转移方法未变，但段间无条件转移指令的执行细节较复杂，请参见 10.6 节。

(2) 条件转移指令

80386 的条件转移指令(除 JCXZ 和 JECXZ 指令外)允许用多字节来表示转移目的地偏移与当前偏移之间的差，所以转移范围可超出－128～＋127。例如：

```
     JNZ    OK
     ……                    ;超过 127 字节
OK:  ……
```

这一点比 8086/8088 的条件转移指令的功能强，它使得程序员可不必考虑条件转移的范围。但在向前引用标号时，如果程序员能够预计到所引用的标号在 127 的范围之内，那么，在标号前加上汇编语言操作符 SHORT 可使汇编程序产生只有一字节地址差的条件转移指令。例如：

```
     CMP    EDX,ECX
     JBE    SHORT OK
     XCHG   EDX,ECX
OK:
```

在 80386 中，当寄存器 CX 的值为 0 时，转移的指令 JCXZ 可以被扩展到 JECXZ，例如：

```
JECXZ    OK
```

它表示当 32 位寄存器 ECX 为 0 时，转移到标号 OK 处。但必须注意，指令 JCXZ 和 JECXZ 与其他条件转移指令不同，仍只能用一字节表示地址差值，所以转移范围仍是-128～127。

2. 循环指令组

循环指令组含有如下三条指令：LOOP、LOOPZ/LOOPE 和 LOOPNZ/LOOPNE。

这三条循环指令的非扩展形式保持原功能。它们的扩展形式使用 ECX 作为计数器，即从 CX 扩展到 ECX。汇编程序 TASM 支持使用助记符 LOOPD、LOOPZD/LOOPDE 和 LOOPDNZ/LOOPDNE，以便明确说明使用 ECX 作为计数器；同样，可使用助记符 LOOPW、LOOPWZ/LOOPWE 和 LOOPWNZ/LOOPWNE，以便明确说明使用 CX 作为计数器。

这三条循环指令的转移范围仍是－128～＋127。

如下过程 SORT 实现了“冒泡”法排序。

```
;过程名称: SORT
;功    能: 将缓冲区中的双字有符号数按小到大排序
;入口参数: ESI=缓冲区首地址偏移,ECX=缓冲区中待排序数据的个数(≥2)
;出口参数: 缓冲区中已排序好的数据
SORT        PROC
DEC         ECX
OUTLOOP:    MOV     EDX,0
INNERLOOP:  CMP     EDX,ECX
            JAE     SHORT BOTTOM
            MOV     EAX,[ESI+EDX*4+4]
            CMP     [ESI+EDX*4],EAX
            JGE     SHORT NOSWAP
            XCHG    [ESI+EDX*4],EAX
            MOV     [ESI+EDX*4+4],EAX
NOSWAP:     INC     EDX
            JMP     INNERLOOP
BOTTOM:     LOOP    OUTLOOP
            RET
SORT        ENDP
```

3. 过程调用和返回指令组

过程调用指令 CALL 在分为段内直接、段内间接、段间直接和段间间接四类的同时，还具有扩展形式。扩展的调用指令的转移目的地址偏移采用 32 位表示。对于扩展的段间调用指令，转移目的地址采用 48 位全指针形式表示，而且在把返回地址的 CS 压入堆栈时扩展成高 16 位为 0 的双字，这样会压入堆栈 2 个双字。

过程返回指令 RET 在分为段内返回和段间返回的同时，还分别具有扩展形式。扩展的过程返回指令要从堆栈弹出双字作为返回地址的偏移。如果是扩展的段间返回指令，执行时要从堆栈弹出包含 48 位返回地址全指针的 2 个双字。

在实方式下主要使用过程调用指令 CALL 和过程返回指令 RET 的非扩展形式，它们与 8086/8088 的 CALL 指令和 RET 指令相同。

在保护方式下，段内过程调用指令和返回指令的转移方法未变，但段间过程调用和返回指令的执行细节较复杂，请参见 10.6 节。

下列过程 FACT 的功能是计算 n!，它通过调用递归过程_FACT 实现其功能。

```
;过程名称: FACT
;功    能: 计算 n!
;入口参数: EAX=n
;出口参数: EAX=n!
;说    明: 如果溢出,那么 EAX=-1
```

```
FACT    PROC
        PUSH    BX
        PUSH    ECX
        MOV     BL,0                ;BL=1 表示溢出
        MOV     ECX,EAX
        CMP     ECX,0               ;负数无意义
        JL      SHORT FACT1
        CALL    _FACT               ;调用递归过程
        CMP     BL,1                ;溢出?
        JNZ     SHORT FACT2
FACT1:  MOV     EAX,-1              ;溢出处理
FACT2:  POP     ECX
        POP     BX
        RET
FACT    ENDP
;
;过程名称: _FACT
;入口参数: ECX=n,BL=0
;出口参数: EAX=n!,BL=1 表示溢出
_FACT   PROC
        CMP     ECX,0
        JZ      SHORT _FACT2
        PUSH    ECX                 ;保存 n
        DEC     ECX
        CALL    _FACT               ;计算(n-1)!
        POP     ECX                 ;n
        IMUL    EAX,ECX             ;n! =nX(n-1)!
        JNO     SHORT _FACT1        ;是否溢出?
        MOV     BL,1                ;是
_FACT1: RET
_FACT2: MOV     EAX,1               ;0! =1
        RET
_FACT   ENDP
```

在 4.5 节已有一个计算 n! 的递归过程，请把这里的递归过程与之作比较。

4. 中断调用和中断返回指令组

在实方式下，中断调用指令 INT 的功能与 8086/8088 的 INT 指令相同。

在保护方式下，中断调用指令 INT 把扩展的标志寄存器 EFLAG、CS 和 EIP 压入堆栈，也即压入堆栈 3 个双字，在压入 CS 时也扩展到 32 位，高 16 位为 0。具体执行细节较复杂，在 10.7 节中说明。

中断返回指令 IRET 有非扩展和扩展两种形式。在实方式下，总是使用其非扩展形式，其功能与 8086/8088 的 IRET 指令相同。在保护方式下，应该使用其扩展形式，以与保

护方式下的中断调用指令相对应。具体执行细节较复杂，在10.7节中说明。

在实方式下，溢出中断调用指令INTO的功能与8086/8088相同。在保护方式下，该指令的执行细节较复杂，请参见10.7节。

9.3.5 串操作指令

从80386开始，串操作的基本单位在字节和字的基础上增加了双字。从80186开始，在8086/8088的5条基本串操作指令的基础上，增加了串输入操作指令INS和串输出操作指令OUTS。

1. 基本串操作指令

对应于字节和字为元素的基本串操作指令没变化。对应于双字为元素的基本串操作指令格式如下：

```
LODSD                ;串装入指令
STOSD                ;串存储指令
MOVSD                ;串传送指令
SCANSD               ;串扫描指令
CMPSD                ;串比较指令
```

其中，LODSD、STOSD和SCANSD指令使用累加器EAX；在DF=0时，每次执行串操作后相应指针加4，在DF=1时，每次串操作后相应指针减4。

这些以双字为元素的基本串操作指令的功能和使用方法与以字节或字为元素的基本串操作指令一样。它们分别是对应以字为元素的串操作指令的扩展。

在不使用32位指针的情况下，串操作中的源指针是DS：SI，目的指针是ES：DI；在使用32位指针（地址扩展）的情况下，源指针是DS：ESI，目的指针是ES：EDI。此外，可以通过段前缀超越的方法改变源串采用的段寄存器，但不能改变目的串的段寄存器。在实方式下，通常使用16位指针。

下面介绍一个简单的位串传送过程SBITBLT。该过程能够把长度为32的倍数的位串传送到指定存储单元开始的缓冲区中，源串可以从双字中的任意位开始，但目标串必须对齐字节的边界。图9.4是源位串和目标位串的示意图。

```
;过程名称：SBITBLT
;功    能：简单位串传送
;入口参数：DS：ESI=源位串开始单元地址偏移
;          ES：EDI=目标串开始单元地址偏移
;          EBX=要传送的位串长度（以双字为单位）
;          ECX=要传送的位串在源串的第一个双字中的位偏移量
;出口参数：无
;说    明：(1)位串长度必须是双字的倍数
;          (2)目标位串必须从目标单元的第一个字节的边界处开始
SBITBLT   PROC
          CLD
```

```
            MOV     EDX,DWORD PTR [ESI]
            ADD     ESI,4
BITLOOP:    LODSD
            SHRD    EDX,EAX,CL
            XCHG    EAX,EDX
            STOSD
            DEC     EBX
            JNZ     BITLOOP
            RET
SBITBLT     ENDP
```

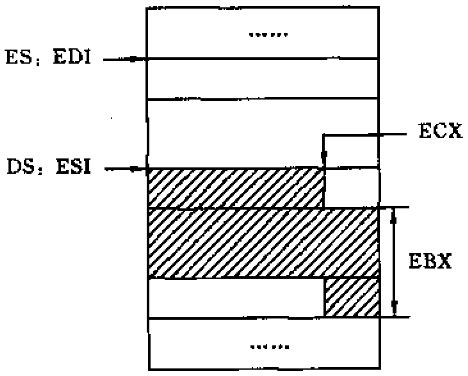


图 9.4　源位串和目标位串示意图

2. 重复前缀

重复前缀 REP、REPZ/REPE 和 REPNZ/REPNE，在仍采用 16 位地址偏移指针的情况下以 CX 作为重复计数器，在采用 32 位地址偏移的扩展情况下以 ECX 作为重复计数器。由于在实方式下通常采用 16 位指针，所以一般仍以 CX 作为计数器。

例如：在 80386 实方式下执行的如下程序片段将把长度为(CX)字节的数据块从由 DS : SI 所指向的源区传到由 ES : DI 所指向的目的区：

```
ROR     ECX,2
REP     MOVSD                   ;使用 CX 为计数器，每次传双字
ROL     ECX,1
REP     MOVSW                   ;传可能余下的字
ROL     ECX,1
REP     MOVSB                   ;传可能余下的字节
```

3. 串输入指令

串输入指令的格式如下：

```
INSB            ;输入字节(Byte)
INSW            ;输入字(Word)
```

```
INSD            ;输入双字(Dword),始于 80386
```

串输入指令从由 DX 给出端口地址的端口读入一字符,并送入由 ES:DI(或 EDI)所指的目的串中,同时根据方向标志 DF 和字符类型调整 DI(或 EDI)。

INSB 指令对应的字符类型是字节,INSW 指令对应的字符类型是字,INSD 指令对应的字符类型是双字。根据输入字符的上述类型,当 DF=0 时,对目的指针的调整值依次分别是 1、2 或 4;当 DF=1 时,对目的指针的调整值依次分别是-1、-2 或-4。

在汇编语言中,三条串输入指令的格式可统一为如下一种格式:

```
INS    DSTS,DX
```

汇编程序根据目的串 DSTS 类型决定使用字节输入指令、字输入指令或双字输入指令。也即,如果类型为字节,则采用 INSB 指令;如果类型为字,则采用 INSW 指令;如果类型是双字,则采用 INSD 指令。请注意,目的串 DSTS 并不影响实际使用指针 ES:DI(或 EDI)及其值,所以在使用上述格式的串输入指令时,仍必须先给 ES:DI(或 EDI)赋合适的值。

串输入指令不影响标志。

在串输入指令前,可使用重复前缀 REP,以便连续输入,但必须注意端口的数据准备情况。

4. 串输出指令

串输出指令的格式如下

```
OUTSB           ;输出字节(Byte)
OUTSW           ;输出字(Word)
OUTSD           ;输出双字(Dword),始于 80386
```

串输出指令把由 DS:SI(或 ESI)所指的源串中的一个字符,输出到由 DX 给出的端口,同时根据方向标志 DF 和字符类型调整 SI(或 ESI)。

OUTSB 指令对应的字符类型是字节,OUTSW 指令对应的字符类型是字,OUTSD 指令对应的字符类型是双字。根据输入字符的上述类型,当 DF=0 时,对源指针的调整值依次分别是 1、2 或 4;当 DF=1 时,对源指针的调整值依次分别是-1、-2 或-4。

在汇编语言中,三条串输出指令的格式可统一为如下一种格式:

```
OUTS DX,SRCS
```

汇编程序根据源串 SRCS 类型决定使用字节输出指令、字输出指令或双字输出指令。也即,如果类型为字节,则采用 OUTSB 指令;如果类型为字,则采用 OUTSW 指令;如果类型是双字,则采用 OUTSD 指令。请注意,指令中给出的源串并不影响实际使用指针 DS:SI(或 ESI)及其值,所以在使用上述格式的串输出指令时,仍必须先给 DS:SI(或 ESI)赋合适的值。

串输出指令不影响标志。

在串输出指令前,可使用重复前缀 REP,以便连续输出,但必须注意端口的数据接收处理情况。

9.3.6 高级语言支持指令

高级语言支持指令始于 80186，它们用于简化支持高级语言的某些特征。共有 3 条这样的指令，它们是：BOUND、ENTER 和 LEAVE。

1. 建立与释放堆栈框架指令

在 C 和 PASCAL 等高级语言中，函数或过程不仅通过堆栈传递入口参数，而且它们的局部变量也被安排在堆栈中，为了方便地获取入口参数和准确地存取局部变量，就要建立合适的堆栈框架。

对照比较如下的 C 函数和对应的汇编语言过程，就可看到堆栈框架的建立和使用情况。

为了说明情况而设计的一个简单的 C 函数 sum 如下：

```
sum(int  x,  int  y)          /* 求两数之和函数 */
{                             /* x 和 y 是入口参数 */
    int  sum;                 /* 定义一个局部变量 */
    sum=x+y;                  /* 求和 */
    return(sum);              /* 返回 */
}
```

对应上面函数 sum 的汇编语言过程如下：

```
_sum   proc   near
       push   bp
       mov    bp,sp                  ;建立堆栈框架
       sub    sp,2                   ;堆栈框架如图 9.5 所示
       ;
       mov    ax,word ptr [bp+4]     ;取参数 x
       add    ax,word ptr [bp+6]     ;加参数 y
       mov    word ptr [bp-2],ax     ;保存 x+y 之和到局部变量 sum
       ;
       mov    ax,word ptr [bp-2]     ;取返回参数
       ;
       mov    sp,bp                  ;释放堆栈框架
       pop    bp
       ret                           ;返回
_sum   endp
```

如果利用始于 80186 的建立和释放堆栈框架指令 ENTER 和 LEAVE，那么上面的汇编语言过程可优化如下：

```
_sum   proc     near
       enter    2,0                  ;建立堆栈框架
       ;
```

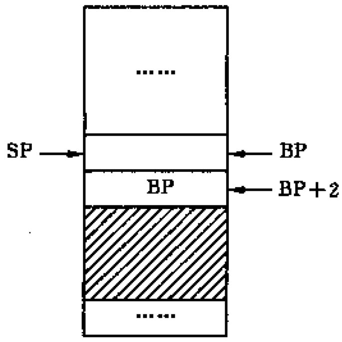


图 9.5 函数 sum 建立堆栈框架后的堆栈示意

```
        mov     ax,word ptr [bp+4]
        add     ax,word ptr [bp+6]
        mov     word ptr [bp-2],ax
        mov     ax,word ptr [bp-2]
        ;
        leave                               ;释放堆栈框架
        ret
_sum    endp
```

(1) 建立堆栈框架指令 ENTER

建立堆栈框架指令 ENTER 的一般格式如下:

```
ENTER    CNT1,CNT2
```

其中,操作数 CNT1 是 16 位立即数,表示框架的大小,也即子程序需要安排在堆栈中的局部变量所需的字节数;操作数 CNT2 是 8 位立即数,表示子程序嵌套级别,也即需要从外层框架复制到当前框架的指针数。

ENTER 指令有非扩展和扩展两种形式。汇编程序 TASM 支持使用助记符 ENTERW 和 ENTERD,以便明确说明使用该指令的非扩展形式和扩展形式。在实方式下,通常使用非扩展形式。

操作数 CNT2 为 0 时,非扩展形式的 ENTER 指令建立堆栈框架时所完成的操作如下:

① BP 进栈,即保存原堆栈框架指针

② BP⇐SP

③ SP⇐SP－16 位立即数(CNT/)

例:如下指令建立 BP 指示的堆栈框架,堆栈顶有 4 字节的局部变量:

```
ENTER    4,0
```

在 80386 的实方式下,使用该指令的扩展形式,也可建立由 EBP 指示的堆栈框架。上述操作步骤改变为:

① EBP 进栈,即保存原堆栈框架指针

② EBP⇐ESP

③ SP⇐SP－16 位立即数(CNT/)

只调整 SP 的原因是实方式下不使用 ESP 作为堆栈指针。

在 80386 的保护方式下使用该指令的扩展形式，那么 BP 和 SP 寄存器分别扩展为 EBP 和 ESP。

ENTER 指令不影响标志。

(2) 释放堆栈框架指令 LEAVE

释放堆栈框架指令 LEAVE 的一般格式如下：

```
LEAVE
```

指令 LEAVE 的功能与指令 ENTER 相反，释放当前子程序(过程)在堆栈中的局部变量，使 BP 和 SP 恢复成最近一次的 ENTER 指令被执行前的值，具体操作如下：

① SP⇐BP

② BP 退栈

LEAVE 指令也有非扩展和扩展两种形式。汇编程序 TASM 支持使用助记符 LEAVEW 和 LEAVED，以便明确说明使用该指令的非扩展形式和扩展形式。

在 80386 的实方式下，使用该指令的扩展形式，那么上述操作步骤改变为：

①SP⇐BP

②EBP 退栈

在 80386 的保护方式下使用该指令的扩展形式，那么 BP 和 SP 寄存器分别扩展为 EBP 和 ESP。

LEAVE 指令不影响标志。

注意，指令 LEAVE 只负责释放堆栈框架，不实现返回，所以在过程中 LEAVE 指令后还应安排相应的 RET 指令。

下面的过程利用指令 ENTER 和 LEAVE 建立和释放堆栈框架。该过程利用迭代算法计算 n!，出入口参数与在 9.3.4 节中介绍的过程 FACT 相同，请作比较。

```
ERROR  EQU    SS:BYTE  PTR  [BP-1]
;
FACT   PROC
       ENTER  4,0           ;建立堆栈框架
       MOV    ERROR,0
       CMP    EAX,0
       JL     FACT4
       PUSH   ECX
       MOV    ECX,EAX
       MOV    EAX,1
       JE     FACT3
FACT1: IMUL   EAX,ECX
       JNO    FACT2
```

```
          MOV     ERROR,1
FACT2:    LOOPD   FACT1          ;使用 ECX 作为计数器
FACT3:    POP     ECX
          CMP     ERROR,1
          JZ      FACT4
          LEAVE                  ;释放堆栈框架
          RET
FACT4:    MOV     EAX,-1
          LEAVE                  ;释放堆栈框架
          RET
FACT      ENDP
```

2. 检查数组下标界限指令 BOUND

检查数组下标界限指令 BOUND 的一般格式如下:

```
BOUND    OPRD1,OPRD2
```

在 80386 之前,用于给出待检查数组下标的操作数 OPRD1 是 16 位寄存器,用于给出数组下标上下界限的操作数 OPRD2 是 32 位存储器操作数,其中低字含起始下标,高字含结尾下标;从 80386 开始,OPRD1 还可以是 32 位寄存器,此时 OPRD2 只能是 64 位存储器操作数,其中低双字含起始下标,高双字含结尾下标。

该指令检查由 OPRD1 给出的有符号数是否在由操作数 OPRD2 给出的数组界限之内。如果被检查的下标不在数组允许的范围之内,那么产生类型号为 5 的异常(中断)。

指令 BOUND 不影响标志。

例如:

```
NUM = 100
ARRAY DB NUM DUP (0)
STV    DW     0
EDV    DW     NUM-1
       ......
       BOUND  SI,DWORD  PTR  STV      ;SI 含下标
       MOV    AL,ARRAY[SI]
       ......
```

9.3.7 条件字节设置指令

从 80386 开始新增加了一组条件字节设置指令。这些指令根据一些标志位设置某个字节的内容为 1 或 0。

条件字节设置指令的一般格式如下:

```
SETcc    OPRD
```

其中,cc 是指令助记符的一部分,用于表示条件,这些条件与条件转移指令中的条件相同;操作数 OPRD 只能是 8 位寄存器或者存储单元,用于存放测试的结果。

这些指令的功能是测试指令中规定的条件，若条件为“真”，那么将目的操作数 OPRD 置成 1，否则置成 0。

表 9.2 条件字节设置指令

指令格式	功能说明	测试条件	
SETZ SETE	OPRD OPRD	等于 0 或者相等时，置 OPRD 为 1	ZF=1
SETNZ SETNE	OPRD OPRD	不等于 0 或者不相等时，置 OPRD 为 1	ZF=0
SETS	OPRD	为负置 OPRD 为 1	SF=1
SETNS	OPRD	为正置 OPRD 为 1	SF=0
SETO	OPRD	溢出置 OPRD 为 1	OF=1
SETNO	OPRD	不溢出置 OPRD 为 1	OF=0
SETP SETPE	OPRD OPRD	偶置 OPRD 为 1	PF=1
SETNP SETPO	OPRD OPRD	奇置 OPRD 为 1	PF=0
SETB SETNAE SETC	OPRD OPRD OPRD	低于置、不高于等于或者进位时，置 OPRD 为 1	CF=1
SETNB SETAE SETNC	OPRD OPRD OPRD	不低于、高于等于或者进位时，置 OPRD 为 1	CF=0
SETBE SETNA	OPRD OPRD	低于等于或者不高于时，置 OPRD 为 1	(CF 或 ZF)=1
SETNBE SETA	OPRD OPRD	不低于等于或者高于时，置 OPRD 为 1	(CF 或 ZF)=0
SETL SETNGE	OPRD OPRD	小于或者不大于等于时，置 OPRD 为 1	(SF 异或 OF)=1
SETNL SETGE	OPRD OPRD	不小于或者大于等于时，置 OPRD 为 1	(SF 异或 OF)=0
SETLE SETNG	OPRD OPRD	小于等于或者不大于时，置 OPRD 为 1	((SF 异或 OF)或 ZF)=1
SETNLE SETG	OPRD OPRD	不小于等于或者大于时，置 OPRD 为 1	((SF 异或 OF)或 ZF)=1

例如：

```
SETO    AL      ;当标志 OF 置位时，把 AL 置 1，否则 AL 清 0
SETNC   CH      ;当标志 CF 清时，把 CH 置 1，否则 CH 清 0
```

条件字节设置指令与条件转移指令一样，并且测试条件的方法也相同，只是在条件满足时，设置某个字节而已。这些条件字节设置指令共有 16 条，列于表 9.2，为了便于记忆和使用，有些指令有多个助记符。

这些条件字节设置指令也可像条件转移指令那样分为三类。

这些条件字节设置指令不影响各标志。

例：如下程序片段测试含于寄存器 EAX 中的八位 16 进制数是否有一位为 0(BH=0 表示没有一位是 0)：

```
        MOV     BH,0
        MOV     CX,8
NEXT:   TEST    AL,0FH
        SETZ    BL
        OR      BH,BL
        ROR     EAX,4
        LOOP    NEXT
```

例：如下程序片段统计 DS：SI 所指向的单字节数据缓冲区中正数和负数的个数(设缓冲区以 0 结尾)：

```
        CLD
        XOR     DX,DX           ;计数器清 0
NEXT:   LODSB
        CMP     AL,0
        JZ      SHORT OVER
        SETG    BL              ;正数时 BL=1,否则 BL=0
        SETL    BH              ;负数时 BH=1,否则 BH=0
        ADD     DL,BL
        ADD     DH,BH
        JMP     NEXT
OVER:
```

9.3.8 位操作指令

从 80386 开始增加了位操作指令。这些位操作指令可以直接对一个二进制位进行测试、设置和扫描等操作。利用这些指令可以更有效地进行位操作。

位操作指令可分为位扫描指令组和位测试及设置指令组。

1. 位测试及设置指令组

位测试和设置指令组含有如下 4 条指令：位测试(Bit Test)指令 BT、位测试并取反(Bit Test and Complement)指令 BTC、位测试并复位(Bit Test and Reset)指令 BTR 和位测试并置位(Bit Test and Set)指令 BTS。

这 4 条位测试和设置指令的格式如下：

```
BT      OPRD1,OPRD2
```

```
BTC    OPRD1,OPRD2
BTR    OPRD1,OPRD2
BTS    OPRD1,OPRD2
```

其中,操作数 OPRD1 可以是 16 位或 32 位通用寄存器和 16 位或 32 位存储单元,用于指定要测试的内容;操作数 OPRD2 必须是 8 位立即数或者与操作数 OPRD1 长度相等的通用寄存器,用于指定要测试的位。

设操作数 OPRD2 除以操作数 OPRD1 的长度后所得商是 disp,所得余数是 offset。那么这些指令要测试的位是根据如下方法确定的:如果操作数 OPRD1 是寄存器,offset 是寄存器操作数 OPRD1 中要测试位的位号;如果操作数 OPRD1 是存储单元,存储器操作数 OPRD1 的偏移与 disp 相加之和是实际测试存储单元的偏移,offset 是该存储单元中要测试位的位号。操作数 OPRD2 取符号整数值,所以当 OPRD2 为 16 位时,可访问(-32K)至(32K-1)范围内的位串,当 OPRD2 是 32 位时,可访问(-2G)至(2G-1)范围内的位串。

位测试指令 BT 的功能是把被测试位的值送标志位 CF。

位测试并取反指令 BTC 的功能是把被测试位的值送标志 CF,并且把被测试位取反。

位测试并复位指令 BTR 的功能是把被测试位的值送标志 CF,并且把被测试位复位,也即清 0。

位测试并置位指令 BTS 的功能是把被测试位的值送标志 CF,并且把被测试位置位,也即置 1。

其它标志 CF、SF、OF、AF 和 PF 无定义。

例如:

```
MOV     BX,4567H
MOV     ECX,3
BT      BX,CX          ;CF=0,BX=4567H
BTC     BX,3           ;CF=0,BX=456FH
BTR     BX,CX          ;CF=1,BX=4567H
BTS     EBX,ECX        ;CF=0,BX=456FH
```

假设数据段有如下变量:

```
IMAGEW    DW          1234H,5678H
IMAGED    DD          12345678H
```

代码段有如下指令(设已置妥 DS):

```
BT        IMAGEW,4        ;CF=1,[IMAGEW]=1234H
MOV       CX,22
BTC       IMAGEW,CX       ;CF=1,[IMAGEW+2]=5638H
BTR       IMAGED,6        ;CF=1,[IMAGED]=12345638H
MOVZX     EAX,CX
BTS       IMAGED,EAX      ;CF=0,[IMAGED]=12745678H
```

要特别指出，在这些位测试指令中，如果用于指定测试位号的操作数 OPRD2 是立即数，那么其值不应超过被测试操作数 OPRD1 的长度，否则将产生未定义的位偏移量。这个规则允许规定在一个寄存器内的任何位移量，而且将存储器位串中的立即数位移量限制在规定存储单元字或双字之内。但汇编程序可以支持对于内存位串的更大的立即数位偏移量，汇编程序可将该立即数位移量的低 5 位(对于 32 位操作数)或低 4 位(对于 16 位操作数)作为机器指令中的操作数 OPRD2，将该立即数位移量的相应高位右移后加到内存位串开始单元的偏移上，作为机器指令中的操作数 OPRD1。

例：如下程序片段把寄存器 AL 的位 0、2、4、6 依次重复一次，所得的 8 位数保存在寄存器 AL 中：

```
        MOV    DL,0
        MOV    CX,4
        MOV    BX,0
NEXT:   BT     AX,BX         ;依次测 0、2、4、6 位
        SETC   AH            ;根据被测位值，置 AH
        OR     DL,AH
        ROR    DL,1
        OR     DL,AH
        ROR    DL,1
        INC    BX
        INC    BX
        LOOP   NEXT
        MOV    AL,DL
```

2. 位扫描指令组

位扫描指令组含有如下 2 条指令：顺向位扫描(Bit Scan Forward)指令 BSF 和逆向位扫描(Bit Scan Reverse)指令 BSR。

这两条位扫描指令的格式如下：

```
BSF   OPRD1,OPRD2
BSR   OPRD1,OPRD2
```

其中，操作数 OPRD1 和 OPRD2 可以是 16 或 32 位通用寄存器和 16 位或 32 位存储单元；但操作数 OPRD1 和 OPRD2 的位数(长度)必须相等。

顺向位扫描指令 BSF 的功能是从右向左(位 0～位 15 或位 31)扫描字或者双字操作数 OPRD2 中第一个含“1”的位，并把扫描到的第一个含“1”的位的位号送操作数OPRD1。

逆向位扫描指令 BSR 的功能是从左向右(位 15 或位 31～位 0)扫描字或者双字操作数 OPRD2 中第一个含“1”的位，并把扫描到的第一个含“1”的位的位号送操作数OPRD1。

如果字或双字操作数 OPRD2 等于 0，那么零标志 ZF 被置 1，操作数 OPRD1 的值不确定；否则零标志 ZF 被清 0。

其它标志 CF、SF、OF、AF 和 PF 无定义。

例如：

```
MOV     EBX,12345678H
BSR     EAX,EBX                   ;EAX=1CH,ZF=0
BSF     DX,AX                     ;DX=2,ZF=0
BSF     CX,DX                     ;CX=1,ZF=0
```

例：如下程序片段处理 AX 中的 16 位图形信息，仅保留可能有的最右和最左各一位为“1”的位：

```
        XOR     DX,DX
        BSF     CX,AX
        JZ      SHORT DONE
        BTS     DX,CX
        BSR     CX,AX
        JZ      SHORT DONE
        BTS     DX,CX
DONE:   MOV     AX,DX
```

9.3.9 处理器控制指令

处理器控制指令用于设置标志、空操作和与外部事件同步等。这里介绍的处理器控制指令都始于 8086/8088。

1. 设置标志指令组

设置进位标志 CF 的指令 CLC、STC 和 CMC 保持与原先相同。

设置方向标志 DF 的指令 CLD 和 STD 保持与原先相同。

设置中断允许标志 IF 的指令 CLI 和 STI 的功能在实方式下保持与原先相同。在保护方式下它们是 I/O 敏感指令，请参见 10.9 节。

2. 空操作指令

空操作指令 NOP 的一般格式如下：

```
NOP
```

空操作指令的功能是什么都不做。该指令就一个字节的操作码。利用该指令可“填补程序中的空白区”，使代码保持连续。

3. 外同步指令和前缀

(1) 等待指令 WAIT

等待指令 WAIT 的一般格式如下：

```
WAIT
```

该指令的功能是等待直到 BUSY 引脚为高。BUSY 由数值协处理器控制，所以该指令的功能是等待数值协处理器，以便与它同步。该指令也能够检查数值协处理器是否故障，请参见 10.7.2 节。

(2) 封锁前缀 LOCK

封锁前缀 LOCK 可以锁定其后指令的目的操作数确定的存储单元，这是通过使

LOCK 信号在指令执行期间一直保持有效而实现的。在多处理器环境中，使用这种方法可以保证指令执行时独占共享内存。

只有下列指令才能用封锁前缀 LOCK，并且目的操作数是存储器操作数：

```
XCHG
ADD,ADC,INC,SUB,SBB,DEC,NEG
OR,AND,XOR,NOT
BT,BTS,BTR,BTC
```

例如：

```
LOCK   BTC [EBX],EAX LOCK   XCHG [BX],AX
```

9.4 实方式下的程序设计

在实方式下，80386 相当于一个可进行 32 位处理的快速 8086。为 80386 编写的程序可利用 32 位的通用寄存器，可使用新增的指令，可采用扩展的寻址方式。段的最大长度是 64K，但不像真正的 8086，当所存取的存储单元的地址偏移超过 0FFFFH 时，不会引起模 64K 的地址反绕，而是导致段越界异常。所以，在实方式下运行的程序访问的存储单元的地址偏移不能超过 0FFFFH，转移的目的地址偏移也不能超过 0FFFFH。

9.4.1 说明

1. 说明处理器类型的伪指令

在缺省情况下，MASM 和 TASM 仅识别 8086/8088 的指令。为了让 MASM 或 TASM 识别由 80186、80286 和 80386 等新增的指令或功能增强的指令，须告诉汇编程序处理器的类型。汇编语言提供了如下说明处理器类型的伪指令，这些伪指令均以句点引导：

```
.8086          ;只支持对 8086 指令的汇编
.186           ;支持对 80186 指令的汇编
.286           ;支持对非特权 80286 指令的汇编
.286C          ;支持对非特权 80286 指令的汇编
.286P          ;支持对 80286 所有指令的汇编
.386           ;支持对 80386 非特权指令的汇编
.386C          ;支持对 80386 非特权指令的汇编
.386P          ;支持对 80386 所有指令的汇编
```

汇编程序在遇到说明处理器类型的伪指令后，就识别并汇编相应的指令。在一个源程序中，可根据需要安排多条说明处理器类型的伪指令，以便更改对处理器类型的说明。对 TASM 而言，这些伪指令可以安排在源程序中的任何位置。但对 MASM 而言，上述说明处理器类型的伪指令必须安排在段外。

只有在使用说明处理器类型是 80386 的伪指令后，汇编程序才识别表示 32 位寄存器的符号和表示始于 80386 的指令的助记符。

如果执行目标程序的 CPU 是 8088 或 8086，那么就不能指示汇编程序按其他处理器类型（80186、80286 和 80386 等）来汇编源程序；类似地，如果执行目标程序的 CPU 是 80286，那么就不能指示汇编程序按 80386 处理器来汇编源程序。

2. 关于段属性类型的说明

在实方式下，80386 保持与原先的 8086/8088 兼容，所以段的最大长度仍是 64K，这样的段称为 16 位段。但在保护方式下，段长度可达 4G，这样的段称为 32 位段。为了兼容，在保护方式下，也可使用 16 位段。

在 8.1.1 节中介绍了段的定义，段定义语句带有可选的定位类型、组合类型和类别。此外，还有可选的段属性类型说明，用于指示是 16 位段，还是 32 位段。完整段定义的一般格式如下：

```
段名  SEGMENT [定位类型] [组合类型] ['类别'] [属性类型]
```

属性类型说明符号是“USE16”和“USE32”。USE16 表示 16 位段，USE32 表示 32 位段。在使用“.386”等伪指令指示处理器类型 80386 后，缺省的属性类型是 USE32；如果没有指示处理器类型 80386，那么缺省的属性类型是 USE16。

例：如下语句说明一个 32 位段：

```
CSEG   SEGMENT   PARA   USE32
       ......
       ......
CSEG   ENDS
```

例：如下语句说明一个 16 位段：

```
CSEG   SEGMEN   PARA   USE16
       ......
       ......
       ......
CSEG   ENDS
```

注意，在实方式下运行的程序，只能使用 16 位段；此外，总是使用 16 位段的堆栈，也总是使用 SP 作为堆栈指针。

3. 操作数和地址长度前缀

尽管在实方式下只能使用 16 位段，但可使用 32 位操作数，也可使用以 32 位形式表示的存储单元地址，这是利用操作数长度前缀 66H 和存储器地址长度前缀 67H 来表示的。

在 16 位代码段中，正常操作数的长度是 16 位或 8 位。在指令前加上操作数长度前缀 66H 后，操作数长度就成为 32 位或 8 位，也即原来表示 16 位操作数的代码成为表示 32 位操作数的代码。一般情况下，不在源程序中直接使用操作数长度前缀，而是直接使用 32 位操作数，操作数长度前缀由汇编程序在汇编时自动加上。

请注意比较如下在16位代码段中的汇编格式指令和对应的机器码(注释部分)：

```
        .386
TEST16  SEGMENT   PARA   USE16
        ......
        MOV       EAX,EBX     ;66H,8BH,C3H
        MOV       AX,BX       ;8BH,C3H
        MOV       AL,BL       ;8AH,C3H
        ......
TEST16  ENDS
```

32位代码段的情况刚好相反。在32位代码段中,正常操作数长度是32位或8位。在指令前加上操作数长度前缀66H后,操作数长度就成为16位或8位。不在32位代码的源程序中直接使用操作数长度前缀66H表示使用16位操作数,而是直接使用16位操作数,操作数长度前缀由汇编程序在汇编时自动加上。

请注意比较如下在32位代码段中的汇编格式指令和对应的机器码(注释部分)：

```
        .386
TEST32  SEGMENT   PARA   USE32
        ......
        MOV       EAX,EBX     ;8BH,C3H
        MOV       AX,BX       ;66H,8BH,C3H
        ......
TEST32  ENDS
```

通过存储器地址长度前缀67H区分32位存储器地址和16位存储器地址的方法与上述通过操作数长度前缀66H区分32位操作数和16位操作数的方法类似。在源程序中可根据需要使用32位地址,或者16位地址。汇编程序在汇编源程序时,对于16位的代码段,在使用32位存储器地址的指令前加上前缀67H;对于32为代码段,在使用16位存储器地址的指令前加上前缀67H。

在一条指令前可能既有操作数长度前缀66H,又有存储器地址长度前缀67H。

9.4.2 实例

下面举两个例子,介绍如何编写在80386实方式下运行的程序,侧重于80386指令的应用,而不是算法的优化。

例1：写一个程序,以十进制、十六进制和二进制数三种形式显示双字存储单元F000:1234H的内容。

如下程序实现上述功能,并且在以三种形式显示时,都滤去了前导的0。

```
;程序名：T9-1.ASM
;功  能：(略)
        .386                        ;支持对80386非特权指令的汇编
CSEG    SEGMENT  USE16              ;16位段
```

```
        ASSUME CS:CSEG
BEGIN:  MOV     AX,0F000H
        MOV     FS,AX
        MOV     EAX,FS:[1234H]
        ;以十进制数形式显示
        CALL    TODEC
        CALL    NEWLINE
        ;以十六进制数形式显示
        CALL    TOHEX
        MOV     AL,'H'
        CALL    ECHO
        CALL    NEWLINE
        ;以二进制数形式显示
        MOV     EAX,FS:[1234H]
        CALL    TOBIN
        MOV     AL,'B'
        CALL    ECHO
        CALL    NEWLINE
        ;结束
        MOV     AH,4CH
        INT     21H
;子程序名:TODEC
;功    能:以十进制数形式显示 32 位值
;入口参数:EAX=要显示的值
;出口参数:无
TODEC   PROC    NEAR
        PUSHAD
        MOV     EBX,10
        XOR     CX,CX
DEC1:   XOR     EDX,EDX
        DIV     EBX
        PUSH    DX
        INC     CX
        OR      EAX,EAX
        JNZ     DEC1
DEC2:   POP     AX
        CALL    TOASC
        CALL    ECHO
        LOOP    DEC2
        POPAD
        RET
TODEC   ENDP
```

```
;子程序名：TOBIN
;功    能：以二进制数形式显示 32 位值
;入口参数：EAX＝要显示的值
;出口参数：无
TOBIN   PROC    NEAR
        PUSH    EAX
        PUSH    ECX
        PUSH    EDX
        BSR     EDX,EAX
        JNZ     BIN1
        XOR     DX,DX
BIN1:   MOV     CL,31
        SUB     CL,DL
        SHL     EAX,CL
        MOV     CX,DX
        INC     CX
        MOV     EDX,EAX
BIN2:   ROL     EDX,1
        MOV     AL,'0'
        ADC     AL,0
        CALL    ECHO
        LOOP    BIN2
        POP     EDX
        POP     ECX
        POP     EAX
        RET
TOBIN   ENDP
;子程序名:TOHEX
;功    能：以十六进制数形式显示 32 位值
;入口参数：EAX＝要显示的值
;出口参数：无
TOHEX   PROC    NEAR
COUNTB    =    8
        ENTER   COUNTB,0
        MOVZX   EBP,BP
        MOV     ECX,COUNTB
        MOV     EDX,EAX
HEX1:   MOV     AL,DL
        AND     AL,0FH
        MOV     [EBP-COUNTB+ECX-1],AL
        ROR     EDX,4
        LOOP    HEX1
```

```
        MOV     CX,COUNTB
        XOR     EBX,EBX
HEX2:   CMP     BYTE PTR [EBP-COUNTB+EBX],0
        JNZ     HEX3
        INC     EBX
        LOOP    HEX2
        DEC     EBX
        MOV     CX,1
HEX3:   MOV     AL,[EBP-COUNTB+EBX]
        INC     EBX
        CALL    TOASC
        CALL    ECHO
        LOOP    HEX3
        LEAVE
        RET
TOHEX   ENDP
;子程序名：TOASC
;功    能：把一位十六进制数转换为对应的 ASCII 码
;入口参数：AL=要显示的值
;出口参数：AL=ASCII 码
TOASC   PROC    NEAR
        AND     AL,0FH
        ADD     AL,'0'
        CMP     AL,'9'
        SETA    DL
        MOVZX   DX,DL
        IMUL    DX,7
        ADD     AL,DL
TOASC1: RET
TOASC   ENDP
;回车换行子程序
NEWLINE     PROC    NEAR
;内容略
NEWLINE     ENDP
;显示字符子程序
ECHO    PROC    NEAR
;内容略
ECHO    ENDP
CSEG    ENDS
        END     BEGIN
```

例 2：写一个程序，把以十进制数、十六进制数或二进制数形式输入的两个无符号整

数相乘，用十进制数显示乘积。

如下程序实现上述功能，输入的两个整数最大以32位表示，超过认为溢出。当输入的数无效时，显示符号"x"，允许重新输入。十六进制数以符号H结尾，二进制数以符号B结尾，十进制数也可以符号D结尾。

```
;程序名：T9-2.ASM
;功  能：(略)
        .386                        ;支持对80386非特权指令的汇编
CSEG    SEGMENT    USE16            ;16位段
        ASSUME CS：CSEG
;子程序名：LTODEC
;功    能：以十进制数形式显示一个64位无符号整数
;入口参数：EDX：EAX=64位二进制数
;出口参数：无
LTODEC  PROC    NEAR
        XOR     CX,CX
LTDEC1：MOV     EBX,10
        CALL    DIVX
        PUSH    BX
        INC     CX
        MOV     EBX,EDX
        OR      EBX,EAX
        JNZ     LTDEC1
LTDEC2：POP     AX
        AND     AL,0FH
        ADD     AL,'0'
        CALL    ECHO
        LOOP    LTDEC2
        RET
LTODEC  ENDP
;子程序名：DIVX
;功    能：64位数除以32位数，商用64位表示
;入口参数：EDX：EAX=被除数
          EBX=除数
;出口参数：EDX：EAX=商
          EBX=余数
DIVX    PROC    NEAR
        PUSH    ECX
        PUSH    ESI
        MOV     CX,64
        XOR     ESI,ESI
DIVX1： SHL     EAX,1
```

```
            RCL     EDX,1
            RCL     ESI,1
            JC      SHORT DIVX2
            CMP     ESI,EBX
            JB      SHORT DIVX3
DIVX2:      SUB     ESI,EBX
            BTS     AX,0
DIVX3:      LOOP    DIVX1
            MOV     EBX,ESI
            POP     ESI
            POP     ECX
            RET
DIVX        ENDP
;子程序名:GETVAL
;功   能:接收一个最大为 32 表示的无符号整数(可用三种进制表示)
;入口参数:无
;出口参数:EAX=接收到的无符号整数
            BL=0    表示没有接收到数
               1    表示接收到的数符合要求
              -1    表示接收到的数无效
GETVAL      PROC    NEAR
COUNTL      =       36
            ENTER   COUNTL,0
            PUSH    ECX
            PUSH    EDX
            PUSH    DS
            PUSH    ES
            ;置段寄存器
            MOV     AX,SS
            MOV     DS,AX
            MOV     ES,AX
            ;接收一个字符串
            LEA     EDX,[BP-COUNTL]
            MOV     BYTE PTR [EDX],COUNTL-2
            MOV     AH,10
            INT     21H
            CALL    NEWLINE
            ;取字符串长度
            INC     EDX
            MOV     CL,[EDX]
            XOR     CH,CH
            MOVZX   ECX,CX
```

```
        ;去掉前导空格
        INC     ECX
GVAL1:  DEC     ECX
        INC     EDX
        CMP     BYTE PTR [EDX],' '
        JZ      GVAL1
        MOV     BL,0
        JECXZ   GVAL5                   ;输入的字符串空
        ;去掉尾部空格
GVAL2:  CMP     BYTE PTR [EDX+ECX-1],' '
        LOOPZ   GVAL2
        SETNZ   AL
        ADD     CL,AL
        JECXZ   GVAL5
        ;处理可能的十进制数串
        MOV     BL,-1
        MOV     AL,[EDX+ECX-1]
        CMP     AL,'0'
        JB      SHORT GVAL5
        CMP     AL,'9'
        JA      SHORT GVAL3
GVAL2A: CALL    DSTOV
        JMP     SHORT GVAL5
GVAL3:  BTR     AX,5
        CMP     AL,'D'
        JNZ     GVAL3A
        DEC     ECX
        JMP     GVAL2A
        ;处理可能的十六进制数串
GVAL3A: CMP AL,'H'
        JNZ     GVAL4
        DEC     ECX
        CALL    HSTOV
        JMP     SHORT GVAL5
        ;处理可能的二进制数串
GVAL4:  CMP     AL,'B'
        JNZ     SHORT GVAL5
        DEC     ECX
        CALL    BSTOV
GVAL5:  POP     ES
        POP     DS
        POP     EDX
```

```
        POP     ECX
        LEAVE
        RET
GETVAL  ENDP
;子程序名：DSTOV
;功    能：把一个十进制数串转化为对应的数值
;入口参数：EDX=字符串开始地址偏移
        CX=字符串长度
;出口参数：EAX=接收到的无符号整数
        BL=0    表示没有接收到数
           1    表示接收到的数符合要求
          -1    表示接收到的数无效
DSTOV   PROC    NEAR
        PUSH    ESI
        MOV     ESI,EDX
        MOV     BL,0
        JCXZ    DTV3
        MOV     BL,-1
        XOR     EAX,EAX
        MOV     EDX,10
        PUSH    EDX
DTV1:   MOV     DL,[ESI]
        INC     ESI
        CMP     DL,'0'
        JB      SHORT DTV2
        CMP     DL,'9'
        JA      SHORT DTV2
        AND     DL,0FH
        PUSH    EDX
        MUL     DWORD  PTR  [ESP+4]
        OR      DL,DL
        POP     EDX
        JNZ     SHORT DTV2
        ADD     EAX,EDX
        JC      DTV2
        LOOP    DTV1
        MOV     BL,1
DTV2:   POP     EDX
DTV3:   POP     ESI
        RET
DSTOV   ENDP
;子程序名：HSTOV
```

```
;功    能：把一个十六进制数串转化为对应的数值
;入口参数：EDX＝字符串开始地址偏移
          CX＝字符串长度
;出口参数：EAX＝接收到的无符号整数(其他略)
HSTOV     PROC      NEAR
          PUSH      ESI
          MOV       BL,0
          JCXZ      HTV4
          MOV       BL,-1
          XOR       EAX,EAX
          XOR       ESI,ESI
HTV1:     MOV       AL,[EDX]
          INC       EDX
          BTS       AX,5
          CMP       AL,'0'
          JB        SHORT HTV2
          CMP       AL,'9'
          JA        HTV2
          AND       AL,0FH
          JMP       SHORT HTV3
HTV2:     CMP       AL,'a'
          JB        SHORT   HTV4
          CMP       AL,'f'
          JA        SHORT HTV4
          SUB       AL,'a'-10
HTV3:     TEST      ESI,0F0000000H
          JNZ       SHORT   HTV4
          SHL       ESI,4
          ADD       ESI,EAX
          LOOP      HTV1
          MOV       BL,1
HTV4:     MOV       EAX,ESI
          POP       ESI
          RET
HSTOV     ENDP
;子程序名：BSTOV
;功    能：把一个二进制数串转化为对应的数值
;入口参数：EDX＝字符串开始地址偏移
          CX＝字符串长度
;出口参数：EAX＝接收到的无符号整数(其他略)
BSTOV     PROC      NEAR
          PUSH      ESI
```

```
            MOV     BL,0
            JCXZ    BTV2
            MOV     BL,-1
            XOR     EAX,EAX
            XOR     ESI,ESI
            XCHG    EDX,ESI
BTV1:       MOV     AL,[ESI]
            INC     ESI
            CMP     AL,'0'
            JB      SHORT BTV2
            CMP     AL,'1'
            JA      SHORT BTV2
            AND     AL,0FH
            BT      EDX,31
            JC      SHORT BTV2
            SHL     EDX,1
            ADD     EDX,EAX
            LOOP    BTV1
            MOV     BL,1
BTV2:       MOV     EAX,EDX
            POP     ESI
            RET
BSTOV       ENDP
;
TOASC       PROC    NEAR
            ;内容略
TOASC       ENDP
;
NEWLINE PROC        NEAR
            ;内容略
NEWLINE ENDP
;
ECHO        PROC    NEAR
            ;内容略
ECHO        ENDP
CSEG        ENDS
;
CSEG        SEGMENT   USE16             ;16 位代码段
            ASSUME CS : CSEG
BEGIN:      MOV     CX,2
@@1:        MOV     AL,':'              ;显示冒号,提示输入
            CALL    ECHO
```

```
        CALL    GETVAL          ;接收字符串
        CMP     BL,0
        JZ      OVER            ;空串结束
        CMP     BL,1
        JZ      SHORT @@2       ;正常,下一个
        MOV     AL,'x'
        CALL    ECHO            ;无效,显示叉号
        CALL    NEWLINE
        JMP     @@1
@@2:    PUSH    EAX             ;保存到堆栈
        LOOP    @@1
        MOV     AL,'='
        CALL    ECHO            ;显示等于号,表示乘积
        POP     EAX
        POP     EDX             ;弹出两个数
        MUL     EDX             ;求乘积
        CALL    LTODEC          ;显示乘积
        CALL    NEWLINE
        CALL    NEWLINE
        JMP     BEGIN
OVER:   MOV     AH,4CH          ;结束
        INT     21H
CSEG    ENDS
        END     BEGIN
```

9.5 习　　题

题 9.1 与 8086/8088 比较,实方式下的 80386 功能在哪些方面大有提高?

题 9.2 80386 的 32 位通用寄存器与 16 位通用寄存器之间的关系如何?

题 9.3 请列出 80386 的段寄存器。实方式下的 80386 同时可对多少个段进行操作?

题 9.4 80386 可寻址的物理地址空间有多大?实方式下每个段可多大?如何得到存储器 32 位物理地址。

题 9.5 80386 的寻址方式有何扩展?

题 9.6 请给出表 9.1 所列各种情形的使用例子。

题 9.7 设寄存器 EAX 含有一个不太大的无符号数,请给出两种用一条指令实现把 EAX 内容乘 9 的方法。

题 9.8 写一个程序测试一下,在实方式下如果偏移超过 64K 会得到什么结果?

题 9.9 写一个程序用十六进制数形式在屏幕上显示从 10000H 处开始的 32 个双字的值。

题 9.10 如何通知汇编程序识别 80386 指令?

题 9.11 如何通知汇编程序形成 16 位段和 32 位段？

题 9.12 什么是操作数长度前缀和地址长度前缀？在什么情况下要使用这两个前缀？请举例说明之。

题 9.13 指令 NOP 有何用途？请举例说明。

题 9.14 设 AL 是有符号数，请用两种方法把 AL 扩展到 EAX。设 AL 含有无符号数，请用两种方法把 AL 扩展到 EAX。

题 9.15 请比较如下指令：

```
(1) MOVSX  EAX,AX        (2) CWDE
    XOR    EDX,EDX           CDQ
```

题 9.16 从 80186 开始，PUSH 指令的操作数可以是立即数。什么情况下这很有用？请举例说明。

题 9.17 请比较指令 PUSHAD 与如下程序片段的异同：

```
PUSH    EAX
PUSH    ECX
PUSH    EDX
PUSH    EBX
PUSH    ESP
PUSH    EBP
PUSH    ESI
PUSH    EDI
```

题 9.18 如下指令片段的功能是什么？

```
PUSHFD
PUSH    0
PUSH    CS
PUSH    DWORD   PTR   1234H
IRETD
```

题 9.19 如何用一条指令重新设置 80386 的堆栈指针。

题 9.20 请编写一个既适合于 8086/8088 又适合于 80386 的宏，该宏可实现两个 32 位存储器操作数相加，结果存入第三个 32 位存储器操作数。

题 9.21 请编写一个既适合于 8086/8088 又适合于 80386 的宏，该宏求存放在 DX：AX 中的 32 位数的补码。

题 9.22 请编写一个既适合于 8086/8088 又适合于 80386 的宏，调用该宏可分别再定义实现由常数指定移位位数的各种移位指令。

题 9.23 80386 条件转移指令的转移范围可有多大？

题 9.24 请编写一个适用于 80386 的较优化的数据块移动过程。

题 9.25 请画出指令"ENTER 8,0"执行前后的堆栈变化示意图。

题 9.26 请写一个程序测试 BOUND 指令的功能。

题 9.27 请举例说明条件字节设置指令的用途。

题 9.28 80386 的位操作指令说明了什么？

题 9.29 请利用 SETO 指令，改写 9.3.6 中的子程序 FACT。

题 9.30 请优化本章中的排序过程 SORT。

题 9.31 请用 80386 指令改写先前各章的相应习题。

第 10 章　保护方式下的 80386 及其编程

80386 有两种工作方式：实方式和保护方式。本章介绍保护方式下的 80386 及相关的程序设计内容。第 9 章介绍的 80386 寄存器、寻址方式和指令等基本概念，除特别说明外在保护方式下仍然保持。

在以 80386 为处理器的硬件平台和 DOS 为操作系统的软件平台上，可以调试运行本章列出的实例。请用 TASM 或者 MASM 汇编这些实例，用 TLINK 连接(部分实例需要带 32 位连接选项"/3")。在调试运行这些实例时，不要安装使用扩展内存的驱动程序，以避免发生冲突。

10.1　保护方式简述

尽管实方式下的 80386 的功能要大大超过其先前处理器(8086/8088、80186、80286)的功能，但只有在保护方式下，80386 才能真正发挥作用。在保护方式下，全部 32 条地址线有效，可寻址高达 4G 字节的物理地址空间；扩充的存储器分段管理机制和可选的存储器分页管理机制，不仅为存储器共享和保护提供了硬件支持，而且为实现虚拟存储器提供了硬件支持；支持多任务，能够快速地进行任务切换和保护任务环境；4 个特权级和完善的特权检查机制，既能实现资源共享又能保证代码及数据的安全和保密、及任务的隔离；支持虚拟 8086 方式，便于执行 8086 程序。

10.1.1　存储管理机制

为了对存储器中的程序及数据实现保护和共享提供硬件支持，为了对实现虚拟存储器提供硬件支持，在保护方式下，80386 不仅采用扩充的存储器分段管理机制，而且还提供可选的存储器分页管理机制。这些存储管理机制由80386的存储器管理部件MMU实现。

1. 背景

80386 有 32 根地址线，在保护方式下，它们都能发挥作用，所以可寻址的物理地址空间高达 4G 字节。在以 80386 及其以上处理器为 CPU 的 PC 兼容机系统中，把地址在 1M 以下的内存称为常规内存，把地址在 1M 以上的内存称为扩展内存。

80386 还要对实现虚拟存储器提供支持。虽然与 8086 可寻址的 1M 字节物理地址空间相比，80386 可寻址的物理地址空间可谓很大，但实际的微机系统不可能安装如此大的物理内存。所以，为了运行大型程序和真正实现多任务，必须采用虚拟存储器。虚拟存储器是一种软硬件结合的技术，用于提供比在计算机系统中实际可以使用的物理主存储器大得多的存储器空间。这样，程序员在编写程序时，不用考虑计算机中物理存储器的实际容量。

80386 还要对存放在存储器中的代码及数据的共享和保护提供支持。任务甲和任务

乙并存，任务甲和任务乙必须隔离，以免相互影响。但它们又可能要共享部分代码和数据。所以，80386既要支持任务隔离，又要支持可共享代码和数据的共享，还要支持特权保护。

2. 地址空间和地址转换

保护方式下的虚拟存储器由大小可变的存储块构成，这样的存储块称为段。80386采用称为描述符的数据来描述段的位置、大小和使用情况。虚拟存储器的地址（逻辑地址）由指示描述符的选择子和段内偏移两部分构成，这样的地址集合称为虚拟地址空间。80386支持的虚拟地址空间可达64T字节。程序员编写程序时使用的存储地址空间是虚拟地址空间，所以，他们可认为有足够大的存储空间可供使用。

显然，只有在物理存储器中的程序才能运行，只有在物理存储器中的数据才能访问。因此，虚拟地址空间必须映射到物理地址空间，二维的虚拟地址必须转换成一维的物理地址。由于物理地址空间远小于虚拟地址空间，所以只有虚拟地址空间中的部分可以映射到物理地址空间。由于物理存储器的大小要远小于物理地址空间，所以只有上述部分中的部分才能真正映射到物理存储器。

每一个任务有一个虚拟地址空间。为了避免多个并行任务的多个虚拟地址空间直接映射到同一个物理地址空间，采用线性地址空间隔离虚拟地址空间和物理地址空间。线性地址空间由一维的线性地址构成，线性地址空间和物理地址空间对等。线性地址32位长，线性地址空间容量为4G字节。

80386分两步实现虚拟地址空间到物理地址空间的映射，也就是分两步实现虚拟地址到物理地址的转换，但第二步是可选的。图10.1是地址映射转换的示意。

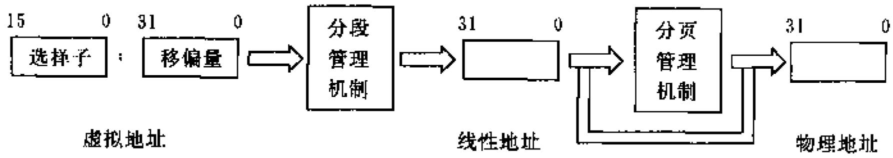


图10.1 地址映射转换示意

通过描述符表和描述符，分段管理机制实现虚拟地址空间到线性地址空间的映射，实现把二维的虚拟地址转换为一维的线性地址。这一步总是存在的。

分页管理机制把线性地址空间和物理地址空间分别划分为大小相同的块，这样的块称之为页。通过在线性地址空间的页与物理地址空间的页之间建立的映射表，分页管理机制实现线性地址空间到物理地址空间的映射，实现线性地址到物理地址的转换。分页管理机制是可选的，在不采用分页管理机制时，线性地址空间就直接等同于物理地址空间，线性地址就直接等于物理地址。

分段管理机制所使用的可变大小的块，使分段管理机制比较适宜处理复杂系统的逻辑分段。存储块的大小可以根据适当的逻辑含义进行定义，而不用考虑固定大小的页所强加的人为限制。每个段可作为独立的单位处理，以简化段的保护及共享。分页机制使用的固定大小的块最适合于管理物理存储器，无论是管理内存还是外存都同样有效。分页管理机制能够有效地支持实现虚拟存储器。

10.1.2 保护机制

为了支持多任务，对各任务实施保护是必需的。从 80286 开始，处理器就具备了保护机制。保护机制能有效地实现不同任务之间的保护和同一任务内的保护。

1. 不同任务之间的保护

保护的一个重要方面是应用程序之间的保护。通过把每个任务放置在不同的虚拟地址空间的方法来实现任务与任务的隔离，达到应用程序之间保护的目的。虚拟地址到物理地址的映射函数在每个任务中进行定义，随着任务切换，映射函数也切换。任务 A 的虚拟地址空间映射到物理地址空间的某个区域，而任务 B 的虚拟地址空间映射到物理地址空间的另外区域，彼此独立，互不相干。因此，两个不同的任务，尽管虚拟存储单元地址相同，但实际的物理存储单元地址可以不同。

2. 同一任务内的保护

在一个任务之内，定义有四种执行特权级别，用于限制对任务中的段进行访问。按照包含在段中的数据的重要性和代码的可信任程度，给段指定特权级别。把最高的特权级别分配给最重要的数据段和最可信任的代码段。具有最高特权级别的数据，只能由最可信任的代码访问。给不重要的数据段和一般代码段分配较低的特权级别。具有最低特权级别的数据，可被具有任何特权级别的代码访问。

特权级别用数字 0～3 表示，数字 0 表示最高特权级别，而数字 3 表示最低特权级别。数字较大的级别具有较低的特权。为了避免模糊和混淆，在比较特权级别时，不使用“大于”或“小于”这样的术语，而使用“里面”或“内层”这样的术语表示较高特权级，级别的数字较小；使用“外面”或“外层”这样的术语表示较低特权级别，级别的数字较大。0 级为最内层的特权级别，3 级为最外层的特权级别，按这样的表示方法，四种特权级的层次关系如图 10.2 所示。

每个存储器段都与一个特权级别相联系。特权级别限制是指，只有足够级别的程序，才可对相应的段进行访问。在任何时候，一个任务总是在 4 个特权级之一下运行，任务在特定时刻的特权级称为当前特权级(Current Privilege Level)，标记为 CPL。每当一个程序试图访问一个段时，就把 CPL 与要访问的段的特权级进行比较，以决定是否允许这一访问。对给定 CPL 执行的程序，允许访问同一级别或外层级别的数据段。如图 10.2 所示，CodeK 可访问同级的数据段 DataK，也可访问外层的 DataOS、DataAP1 及 DataAP2 等。如试图访问内层级别的数据段则是非法的，并引起异常。如图 10.2 所示，CodeOS 可访问同级的 DataOS，也可访问外层的 DataAP1 和 DataAP2 等，但不能访问内层的 DataK。

虽然应用程序都在最外层，但由于各个不同的应用程序存储在不同的虚拟地址空间中，所以各应用程序被隔离保护。如图 10.2 所示，最外层的 CodeAP1 只能访问 DataAP1，不可能访问同级的另一个应用程序的 DataAP2；同样，CodeAP2 只能访问 DataAP2，不可能访问 DataAP1。

这实际上是组合保护。应用程序 1 和操作系统构成任务 A，应用程序 2 和操作系统构成任务 B。操作系统被任务 A 和任务 B 共享，在任务 A 和任务 B 的两个不同的虚拟地址空间中，操作系统占用虚拟地址空间相同的部分。

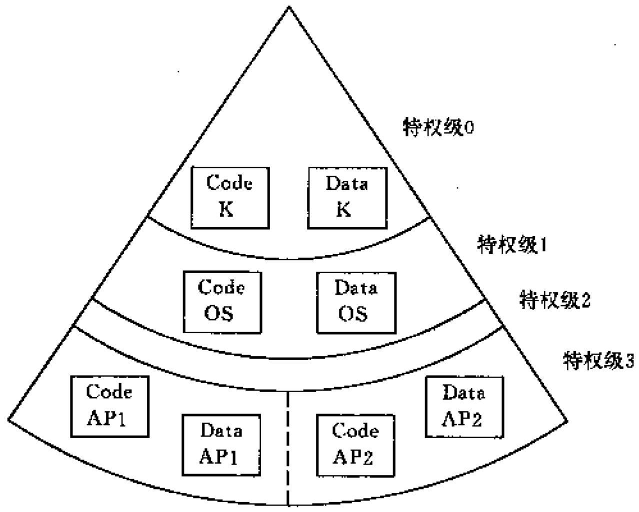


图 10.2　四层特权级别

特权级的典型用法是，把操作系统的核心放在 0 级，操作系统的其余部分放在 1 级，而应用程序放在 3 级，留下 2 级供中间软件使用。对特权级进行这样的安排，使得在 0 级的操作系统核心有权访问任务中的所有存储段；在 1 级的操作系统其余部分有权访问除 0 级外的所有存储段；而在 3 级的应用程序只能访问程序本身的存储段，这些存储段也是在 3 级。

10.2　分段管理机制

本节介绍保护方式下的段定义以及由段选择子和段内偏移构成的二维虚拟地址如何被转换为一维线性地址。

10.2.1　段定义和虚拟地址到线性地址转换

段是实现虚拟地址到线性地址转换机制的基础。在保护方式下，每个段由如下三个参数进行定义：段基地址(Base Address)、段界限(Limit)和段属性(Attributes)。

段基地址规定线性地址空间中段的开始地址。在 80386 保护方式下，段基地址长 32 位。因为基地址长度与寻址地址的长度相同，所以任何一个段都可以从 32 位线性地址空间中的任何一个字节开始，而不像实方式下规定段的边界必须被 16 整除。

段界限规定段的大小。在 80386 保护方式下，段界限用 20 位表示，而且段界限可以是以字节为单位或以 4K 字节为单位。段属性中有一位对此进行定义，把该位称为粒度位，用符号 G 标记。G=0 表示段界限以字节为单位，于是 20 位的界限可表示的范围是 1 字节至 1M 字节，增量为 1 字节；G=1 表示段界限以 4K 字节为单位，于是 20 位的界限可表示的范围是 4K 字节至 4G 字节，增量为 4K 字节。当段界限以 4K 字节为单位时，实际的段界限 LIMIT 可通过下面的公式从 20 位段界限 Limit 计算出来：

$$LIMIT = Limit * 4K + 0FFFH$$
$$= (Limit << 12) + 0FFFH$$

所以，当粒度位为 1 时，段的界限实际上就扩展成为 32 位。由此可见，在 80386 保护方式下，段的长度可大大超过 64K 字节。

基地址和界限定义了段所映射的线性地址的范围。基地址 Base 是线性地址对应于段内偏移为 0 的虚拟地址，段内偏移为 x 的虚拟地址对应 Base+x 的线性地址。段内从偏移 0 到 Limit 范围内的虚拟地址对应于从 Base 到 Base+Limit 范围内的线性地址。

图 10.3 表示一个段如何从虚拟地址空间定位到线性地址空间。图中 BaseA 等代表段基地址，LimitA 等代表段界限。另外，段 C 接在段 A 之后，也即 BaseC＝BaseA＋LimitA。

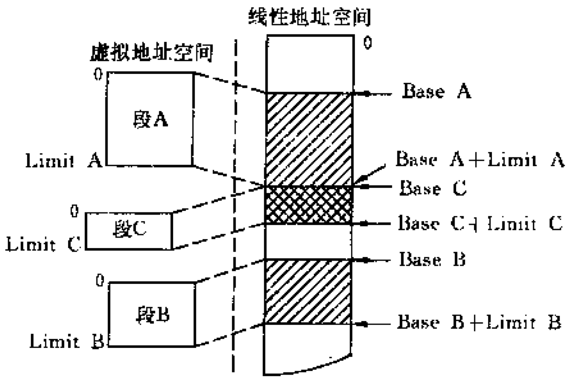


图 10.3 虚拟地址到线性地址转换示意图

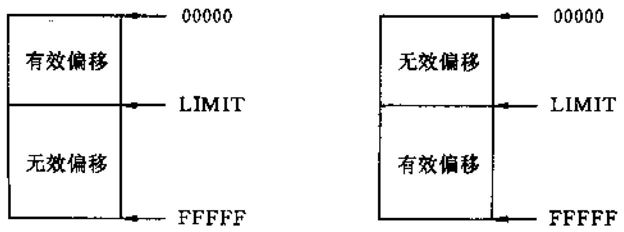
例如：设段 A 的基地址等于 00012345H，段界限等于 5678H，并且段界限以字节为单位(G＝0)，那么段 A 对应线性地址空间中从 00012345H～000179BDH 的区域。如果段界限以 4K 字节为单位(G＝1)，那么段 A 对应线性地址空间中从 00012345H～0568B344H (＝00012345H＋5678000H＋0FFFH)的区域。

通过增加段界限，可以使段的容量得到扩展。这对于那些要在内存中扩展容量的普通数据段很有效，但对堆栈段情况就不是这样。因为堆栈底在高地址端，随着压栈操作，堆栈向低地址方向扩展。为了适应普通数据段和堆栈数据段在两个相反方向上的扩展，数据段的段属性中安排一扩展方向位，标记为 ED。ED＝0 表示向高扩展，ED＝1 表示向低扩展。

数据段的扩展方向和段界限一起决定了数据段内偏移的有效范围。如图 10.4 所示。当段最大为 1M 字节时，在自然的向高扩展段内，从 0～Limit 的偏移是合法有效的偏移，而从 Limit＋1～1M－1 的偏移是非法无效的偏移；在向低扩展段内，情形刚好相反，从 0～Limit 的偏移是非法无效的偏移，而从 Limit＋1～1M－1 的偏移是合法有效的偏移，注意边界值 Limit 对应地址的有效性。当段最大为 4G 字节时，情形类似。

注意，只有在数据段的段属性中才有扩展方向属性位 ED，也就是说只有数据段(堆栈段作为特殊的数据段)才有向高扩展和向低扩展之分，其它段都是自然的向高扩展。

在每次把虚拟地址转换为线性地址的过程中，要对偏移进行检查。如果偏移不在有效



(a) 向高扩展(ED=0)　　(b) 向低扩展(ED=1)

图 10.4　数据段内偏移的有效范围

的范围内，那么就引起异常。

段属性规定段的主要特性。例如上面已提到的段粒度 G 就是段属性的一部分。在对段进行各种访问时，将对访问是否合法进行检查，主要依据是段属性。例如：如果向一个只读段进行写入操作，那么不仅不写入，而且会引起异常。在下面会详细说明各段属性位的定义和作用。

10.2.2　存储段描述符

用于表示上述定义段的三个参数的数据称为描述符。每个描述符长 8 个字节。在保护方式下，每一个段都有一个相应的描述符来描述。

按描述符所描述的对象来划分，描述符可分为如下三类：存储段描述符、系统段描述符、门描述符(控制描述符)。下面先介绍存储段描述符。

1. 存储段描述符

存储段是存放可由程序直接进行访问的代码和数据的段。存储段描述符描述存储段，所以存储段描述符也被称为代码和数据段描述符。

存储段描述符的格式如图 10.5 所示。图中上面一排是对描述符 8 个字节的使用的说明，最低地址字节(假设地址为 m)在最右边，其余字节依次向左，直到最高字节，地址为 m+7；下一排是对属性域各位的说明。

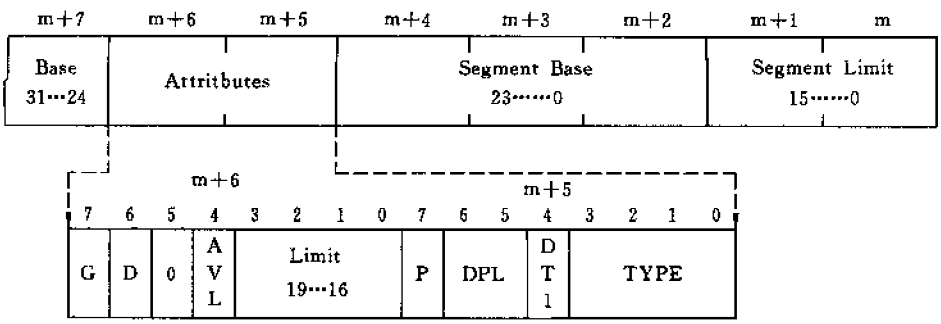


图 10.5　存储段描述符格式

从图 10.5 可知，长 32 位的段基地址(段开始地址)被安排在描述符的两个域中，其

位 0～位 23 安排在描述符内的第 2～第 4 字节中，其位 24～位 31 被安排在描述符内的第 7 字节中。长 20 位的段界限也被安排在描述符的两个域中，其位 0～位 15 被安排在描述符内的第 0～第 1 字节中；其位 16～位 19 被安排在描述符内的第 6 字节的低 4 位中。

使用两个域存放段基地址和存放段界限的原因与 80286 有关。在 80286 保护方式下，段基地址只有 24 位长，而段界限只有 16 位长。80286 存储段描述符尽管也是 8 字节长，但实际只使用低 6 字节，而高 2 字节必须置为 0。80386 存储段描述符这样的安排，可使得 80286 的存储段描述符的格式在 80386 下继续有效。

80386 描述符中的段属性也被安排在两个域中。下面对其定义及意义作说明。

(1) P 位称为存在(Present)位。P＝1 表示描述符对地址转换是有效的，或者表示该描述符所描述的段存在；P＝0 表示描述符对地址转换无效，并且使用该描述符会引起异常。

(2) DPL 表示描述符特权级(Descriptor Privilege Level)，共 2 位。它规定了所描述段的特权级，用于特权检查，以决定对该段能否进行访问。

(3) DT 位说明描述符的类型。对于存储段描述符而言，DT＝1，以区别于系统段描述符和门描述符(DT＝0)。

(4) TYPE 说明存储段描述符所描述的存储段的具体属性。

其中的位 0 指示描述符是否被访问(Accessed)，用符号 A 标记。A＝0 表示尚未被访问，A＝1 表示段已被访问。当把描述符的相应选择子装入到段寄存器时，80386 把该位置为 1，表明描述符已被访问。操作系统可测试访问位，以确定描述符是否被访问过。

其中的位 3 指示所描述的段是代码段还是数据段，用符号 E 标记。E＝0 表示段是不可执行段，也就是数据段，相应的描述符也就是数据段(包括堆栈段)描述符。E＝1 表示段是可执行段，也就是代码段，相应的描述符也就是代码段描述符。

在数据段描述符中(E＝0 的情况)，TYPE 中的位 1 指示所描述的数据段是否可写，用 W 标记。W＝0 表示对应的数据段不可写，只能读，W＝1 表示对应的数据段可写。TYPE 中的位 2 就是 ED 位，指示所描述的数据段的扩展方向。ED＝0 表示数据段向高扩展，也即段内偏移必须小于等于段界限。ED＝1 表示数据段向低扩展，也即段内偏移必须大于段界限。

在代码段描述符中(E＝1 的情况)，TYPE 中的位 1 指示所描述的代码段是否可读，用符号 R 标记。R＝0 表示对应的代码段不可读，只能执行，R＝1 表示对应的代码段可读可执行。TYPE 中的位 2 指示所描述的代码段是否是一致码段，用 C 代表。C＝0 表示对应的代码段不是一致码段(普通代码段)，C＝1 表示对应的代码段是一致码段。

存储段描述符中的 TYPE 所说明的存储段的属性可归纳为表 10.1。

在 80286 的存储段描述符中，位于描述符内第 5 字节的段属性各位的意义与上述说明相同，确切地说是 80386 为了与 80286 兼容而保持了原有定义。下面说明的属性位是 80386 在 80286 基础上的扩充的属性位。

(5) G 位就是段界限粒度(Granularity)位。G＝0 表示界限粒度为字节，G＝1 表示界限粒度是 4K 字节。注意，界限粒度只对段界限有效，对段基地址无效，段基地址总是以字节为单位。

表 10.1　存储段描述符类型

类型	说明	类型	说明
0	只读	8	只执行
1	只读、已访问	9	只执行、已访问
2	读/写	A	执行/读
3	读/写、已访问	B	执行/读、已访问
4	只读、向低扩展	C	只执行、一致码段
5	只读、向低扩展、已访问	D	只执行、一致码段、已访问
6	读/写、向低扩展	E	执行/读、一致码段
7	读/写、向低扩展、已访问	F	执行/读、一致码段、已访问

(6) D 位是一个很特殊的位，在描述可执行段、向低扩展数据段或者由 SS 寄存器寻址的段(通常就是堆栈段)的三种描述符中的意义各不相同。

在描述可执行段的描述符中，D 位决定了指令使用的地址及操作数所默认的大小。D=1 表示默认情况下指令使用 32 位地址及 32 位或 8 位操作数，这样的代码段也称为 32 位代码段；D=0 表示默认情况下使用 16 位地址及 16 位或 8 位的操作数，这样的代码段也称为 16 位代码段，它与 80286 兼容。就象在第 9 章中所述，可使用地址大小前缀和操作数大小前缀分别改变默认的地址或操作数的大小。

在向低扩展数据段的描述符中，D 位决定段的上部边界。D=1 表示段的上部界限为 4G；D=0 表示段的上部界限为 64K，这是为了与 80286 兼容。

在描述由 SS 寄存器寻址的段描述符中，D 位决定隐式的堆栈访问指令(如 PUSH 和 POP 指令)使用何种堆栈指针寄存器。D=1 表示使用 32 位堆栈指针寄存器 ESP；D=0 表示使用 16 位堆栈指针寄存器 SP，这与 80286 兼容。

(7) AVL 位是软件可利用位。80386 对该位的使用未做规定，Intel 公司也保证今后开发生产的处理器只要与 80386 兼容，就不会对该位的使用做任何定义或规定。

此外，描述符内第六字节中的位 5 须置成 0，可理解成是为以后的处理器保留的。

2. 存储段描述符的结构类型表示

根据如图 10.5 给出的存储段描述符的结构，可定义如下的描述符结构类型：

```
DESCRIPTOR   STRUC
LIMITL       DW     0        ;段界限低 16 位
BASEL        DW     0        ;基地址低 16 位
BASEM        DB     0        ;基地址中间 8 位
ATTRIBUTES   DW     0        ;段属性(含段界限的高 4 位)
BASEH        DB     0        ;基地址高 8 位
DESCRIPTOR   ENDS
```

利用结构类型 DESCRIPTOR 能方便地在程序中说明存储段描述符。

例如：如下描述符 DATAS 描述一个可读写的有效(已存在)的数据段，基地址是 100000H，以字节为单位的界限是 0FFFFH，描述符特权级 DPL=3。

```
DATAS    DESCRIPTOR <0FFFFH,0,10H,0F2H,0>
```

再如：如下描述符 CODEA 描述一个只可执行的有效的 32 位代码段，基地址是

12345678H，以4K字节为单位的界限值是10H(以字节为单位的界限是10FFFH)，描述符特权级DPL=0。

```
CODEA      DESCRIPTOR <10H,5678H,34H,0C098H,12H>
```

10.2.3 全局和局部描述符表

一个任务会涉及多个段，每个段需要一个描述符来描述，为了便于组织管理，80386把描述符组织成线性表。由描述符组成的线性表称为描述符表。在80386中有三种类型的描述符表：全局描述符表GDT(Global Descriptor Table)、局部描述符表LDT(Local Descriptor Table)和中断描述符表IDT(Interrupt Descriptor Table)。在整个系统中，全局描述符表GDT和中断描述符表IDT只有一张，局部描述符表可以有若干张，每个任务可以有一张。

例如：下列描述符表有6个描述符构成：

```
DESCTAB   LABEL   BYTE
DESC1     DESCRIPTOR    <1234H,5678H,34H,92H,0>
DESC2     DESCRIPTOR    <1234H,5678H,34H,93H,0>
DESC3     DESCRIPTOR    <5678H,1234H,56H,98H,0>
DESC4     DESCRIPTOR    <5678H,1234H,56H,99H,0>
DESC5     DESCRIPTOR    <0FFFFH,0,10H,16H,0>
DESC6     DESCRIPTOR    <0FFFFH,0,10H,90H,0>
```

每个描述符表本身形成一个特殊的数据段。这样的特殊数据段最多可以含有8K(8096)个描述符。

在10.7.3节中介绍中断描述符表IDT。

每个任务的局部描述符表LDT含有该任务自己的代码段、数据段和堆栈段的描述符，也包含该任务所使用的一些门描述符，如任务门和调用门描述符等。随着任务的切换，系统当前的局部描述符表LDT也随之切换。

全局描述符表GDT含有每一个任务都可能或可以访问的段的描述符，通常包含描述操作系统所使用的代码段、数据段和堆栈段的描述符，也包含多种特殊数据段描述符，如各个用于描述任务LDT的特殊数据段等。在任务切换时，并不切换GDT。

通过LDT可以使各任务私有的各个段与其他任务相隔离，从而达到受保护的目的。通过GDT可以使各任务都需要使用的段能够被共享。图10.6给出了任务A和任务B所涉及的有关段既隔离受保护，又合用共享的情况。通过任务A的局部描述符表LdtA和任务B的局部描述符表LdtB，把任务A所私有的代码段CodeA及数据段DataA与任务B所私有的代码段CodeB和数据段DataB及DataB2隔离，但任务A和任务B通过全局描述符表GDT共享代码段CodeK及CodeOS和数据段DataK及CodeOS。

一个任务可使用的整个虚拟地址空间分为相等的两半，一半空间的描述符在全局描述符表中，另一半空间的描述符在局部描述符表中。由于全局和局部描述符表都可以包含多达8096个描述符，而每个描述符所描述的段最大可达4G字节，因此最大的虚拟地址空间可为：

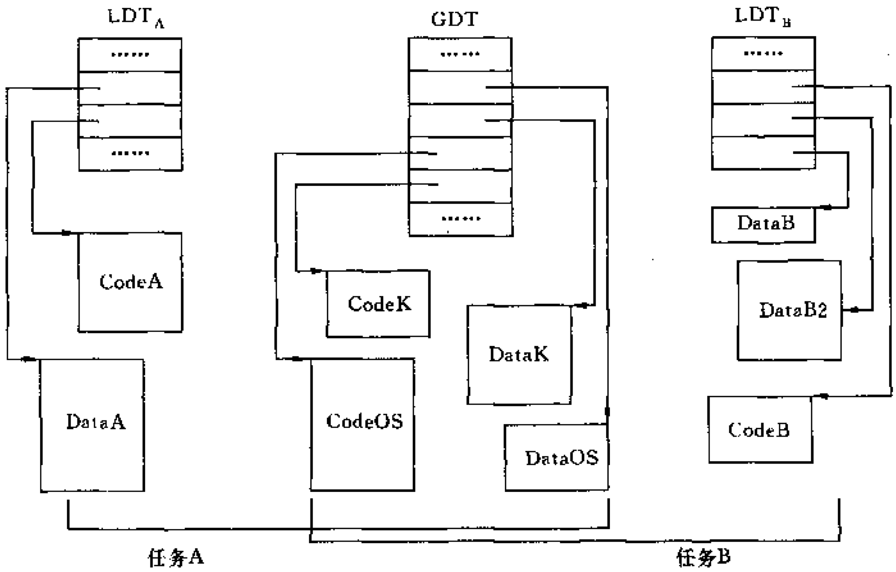


图 10.6　全局和局部描述符表的分工

4GB * 8096 * 2　=　64MMB　=　64T(字节)

10.2.4　段选择子

在实方式下，逻辑地址空间中存储单元的地址有段值和段内偏移两部分组成。在保护方式下，虚拟地址空间(相当于逻辑地址空间)中存储单元的地址有段选择子和段内偏移两部分组成。与实方式相比，段选择子替代了段值。

段选择子长16位，其格式如图10.7所示。从图中可见，段选择子的高13位是描述符索引(Index)。所谓描述符索引是指描述符在描述符表中的序号。段选择子的第2位是引用描述符表指示位，标记为TI(Table Indicator)，TI=0指示从全局描述符表GDT中读取描述符；TI=1指示从局部描述符表LDT中读取描述符。

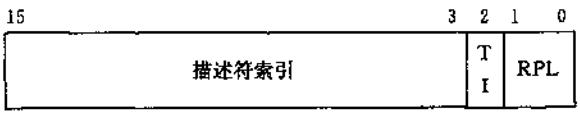


图 10.7　段选择子格式

选择子确定描述符，描述符确定段基地址，段基地址与偏移之和就是线性地址。所以，虚拟地址空间中的由段选择子和偏移两部分构成的二维虚拟地址，就是这样确定了线性地址空间中的一维线性地址。

选择子的最低两位是请求特权级RPL(Requested Privilege Level)，用于特权检查。

例如：假设某个选择子的内容是0030H。根据图10.7所示选择子的格式可知：Index=6，TI=0，RRP=0，所以它指定全局描述符表中的第6个描述符，请求特权级是0。

再如，假设某个选择子的Index=4，TI=1，RPL=3，那么该选择子的内容是27H。

由于选择子中的描述符索引字段用 13 位表示，所以可区分 8096 个描述符。这也就是描述符表最多含有 8096 个描述符的原因。由于每个描述符长 8 字节，按照图 10.7 所示选择子格式，屏蔽选择子低 3 位后所得的值就是选择子所指定的描述符在描述符表中的偏移，这可认为是安排选择子高 13 位为描述符索引的原因。

有一个特殊的选择子称为空(Null)选择子，它的 Index＝0，TI＝0，而 RPL 字段可以为任意值。空选择子有特定的用途，当用空选择子进行存储器访问时会引起异常。空选择子是特别定义的，它不对应于全局描述符表 GDT 中的第 0 个描述符，因此 GDT 中的第 0 个描述符总不会被处理器访问，一般把它置成全 0。但当 TI＝1 时，Index 为 0 的选择子不是空选择子，它指定了当前任务局部描述符表 LDT 中的第 0 个描述符。

10.2.5 段描述符高速缓冲寄存器

在实方式下，段寄存器含有段值，为访问存储器形成物理地址时，处理器引用相应的某个段寄存器得段值。在保护方式下，段寄存器含有段选择子，如上所述，为访问存储器形成线性地址时，处理器要使用选择子所指定的描述符中的基地址等信息。为了避免在每次存储器访问时，都要访问描述符表而获得对应段描述符，从 80286 开始每个段寄存器都配有一个高速缓冲寄存器，称之为段描述符高速缓冲寄存器或称为描述符投影寄存器，对程序员而言它是不可见的。每当把一个选择子装入到某个段寄存器时，处理器自动从描述符表中取出相应的描述符，把描述符中的信息保存到对应的高速缓冲寄存器中。此后在对该段访问时，处理器都使用对应高速缓冲寄存器中的描述符信息，而不用再从描述符表中取描述符。

各段描述符高速缓冲寄存器之内容如表 10.2 所列。 其中，32 位段基地址直接取自描述符，32 位段界限取自描述符中的段界限，并转换成字节为单位。其它十个特性根据描述符中的属性而定，“Y”表示“是”，“N”表示“否”，“r”表示必须可读，“w”表示必须可写，“p”表示必须存在，“d”表示根据描述符中属性而定。

表 10.2 段描述符高速缓冲寄存器之内容

段寄存器	段基地址	段界限	其他段属性									
			存在性	特权级	已存取	粒度	扩展方向	可读性	可写性	可执行	堆栈大小	一致特权
CS	32 位基地址	32 位段界限	p	d	d	d	d	d	N	Y	—	d
SS	32 位基地址	32 位段界限	p	d	d	d	d	r	w	N	d	—
DS	32 位基地址	32 位段界限	p	d	d	d	d	d	d	N	—	—
ES	32 位基地址	32 位段界限	p	d	d	d	d	d	d	N	—	—
FS	32 位基地址	32 位段界限	p	d	d	d	d	d	d	N	—	—
GS	32 位基地址	32 位段界限	p	d	d	d	d	d	d	N	—	—

段描述符高速缓冲寄存器在处理器内，所以可对其进行快速访问。绝大多数情况下，对存储器的访问是在对应选择子装入到段寄存器之后进行的，所以，使用段描述符高速缓冲寄存器可以得到很好的执行性能。

把选择子装入段寄存器和通过段描述符高速缓冲寄存器实现由虚拟地址到线性地址的转换情形如图 10.8 所示。当不采用分页管理机制时，线性地址就是物理地址。

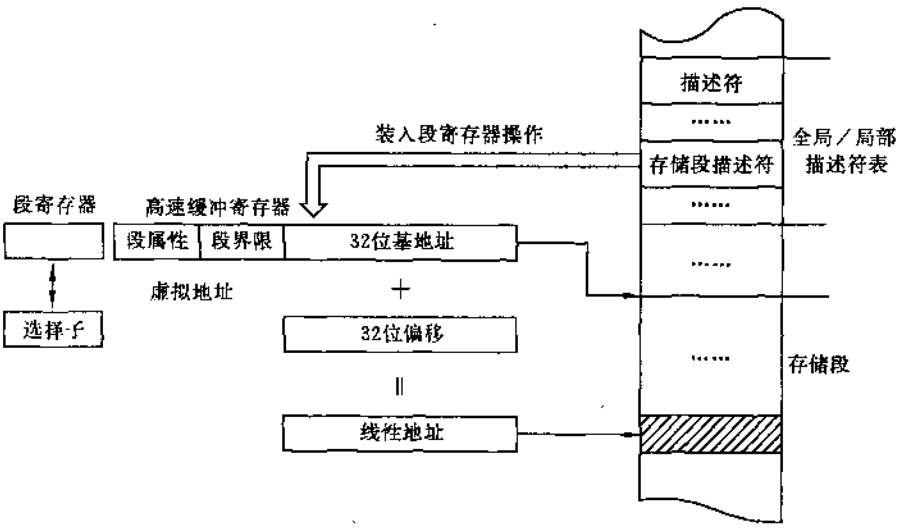


图 10.8　高速缓冲寄存器用于虚拟地址到线性地址转换

设虚拟地址是 10:12345678H，由选择子 10H 所指定的 GDT 中的存储段描述符所含的基地址是 87654321H，那么该虚拟地址转换出的线性地址是 99999999H。如不采用分页管理机制，那么物理地址就是 99999999H。

段描述符高速缓冲寄存器之内保存的描述符信息将一直保持到重新把选择子装载到段寄存器时再更新。程序员尽管不可见段描述符高速缓冲寄存器，但必须注意到它的存在和它的上述更新时机。例如，在改变了描述符表中的某个当前段的描述符后，也要更新对应的段描述符高速缓冲寄存器的内容，这可通过重新装载段寄存器实现。

10.3　80386 控制寄存器和系统地址寄存器

80386 的控制寄存器和系统地址寄存器如图 10.9 所示，它们用于控制工作方式，控制分段管理机制及分页管理机制的实施。

10.3.1　控制寄存器

从图 10.9 可见，80386 有四个 32 位的控制寄存器，分别命名为 CR0、CR1、CR2 和 CR3。但 CR1 被保留，供今后开发的处理器使用，在 80386 中不能使用 CR1，否则将引起无效指令操作异常。CR0 包含指示处理器工作方式的控制位，包含启用和禁用分页管理机制及的控制位，包含控制浮点协处理器操作的控制位。CR2 及 CR3 由分页管理机制使用。CR0 中的位 5～位 30 及 CR3 中的位 0 至位 11 是保留位，这些位不能是随意值，必须为 0。

控制寄存器 CR0 的低 16 位等同于 80286 的机器状态字 MSW。

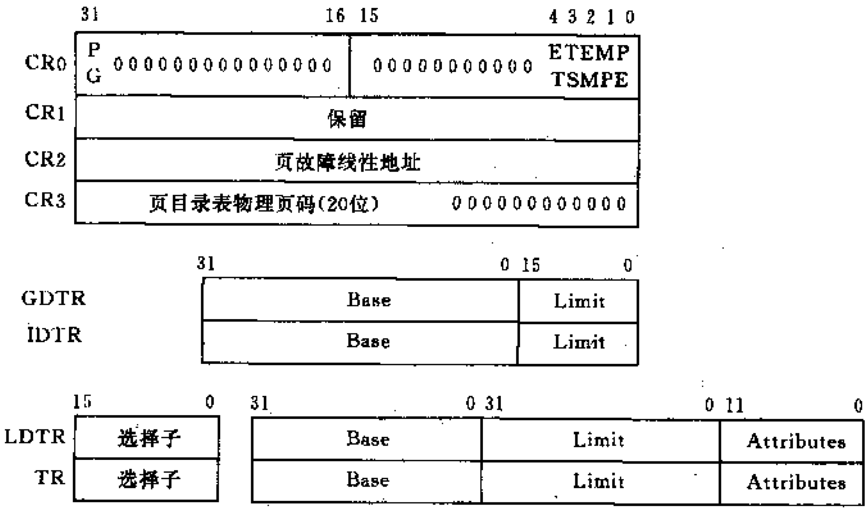


图 10.9　80386 的控制寄存器和系统地址寄存器

1. 保护控制位

控制寄存器 CR0 中的位 0 用 PE 标记，位 31 用 PG 标记，这两个位控制分段和分页管理机制的操作，所以把它们称为保护控制位。PE 控制分段管理机制。PE=0，处理器运行于实方式；PE=1，处理器运行于保护方式。PG 控制分页管理机制。PG=0，禁用分页管理机制，此时分段管理机制产生的线性地址直接作为物理地址使用；PG=1，启用分页管理机制，此时线性地址经过分页管理机制转换成物理地址。请参见图 10.1。关于分页管理机制的具体介绍在 10.9 节中进行。

表 10.3 列出了通过使用 PE 和 PG 位选择的处理器工作方式。由于只有在保护方式下才可启用分页机制，所以尽管两个位分别为 0 和 1，共可以有四种组合，但只有三种组合方式有效。PE=0 且 PG=1 是无效的组合，因此，用 PG 位为 1 且 PE 位为 0 的值装入 CR0 寄存器将引起通用保护异常。

表 10.3　PG/PE 位与处理器执行方式

PG	PE	处理器工作方式
0	0	实方式
0	1	保护方式，禁用分页机制
1	0	非法组合
1	1	保护方式，启用分页机制

2. 协处理器控制位

控制寄存器 CR0 中的位 1～位 4 分别标记为 MP(算术存在位)、EM(模拟位)、TS(任务切换位)和 ET(扩展类型位)，它们控制浮点协处理器的操作。

当处理器复位时，ET 位被初始化，以指示系统中数字协处理器的类型。如果系统中存在 80387 协处理器，那么 ET 位被置 1；如果系统中存在 80287 协处理器或者不存在协

处理器，那么 ET 位被清 0。

EM 位控制浮点指令的执行是用软件模拟，还是由硬件执行。EM＝0 时，硬件控制浮点指令传送到协处理器；EM＝1 时，浮点指令由软件模拟。

TS 位用于加快任务的切换，通过在必要时才进行协处理器切换的方法实现这一目的。每当进行任务切换时，处理器把 TS 置 1。TS＝1 时，浮点指令将产生设备不可使用(DNA)异常。MP 位控制 WAIT 指令在 TS＝1 时，是否产生 DNA 异常。MP＝1 和 TS＝1 时，WAIT 产生异常；MP＝0 时，WAIT 指令忽略 TS 条件。

3. CR2 和 CR3

控制寄存器 CR2 和 CR3 由分页管理机制使用。

CR2 用于发生页异常时报告出错信息。当发生页异常时，处理器把引起页异常的线性地址保存于 CR2 中。操作系统中的页异常处理程序可以检查 CR2 的内容，从而查出线性空间中的哪一页引起本次异常。

CR3 用于保存页目录表的起始物理地址。由于目录是页对齐的，所以仅高 20 位有效，低 12 位保留未用。向 CR3 中装入新的值时，低 12 位必须为 0；但从 CR3 中取值时，低 12 位被忽略。每当用 MOV 指令重置 CR3 值时，会导致分页高速缓冲区内容无效。在实方式下也可设置 CR3，以便进行分页机制的初始化。在任务切换时，CR3 要被改变，但是如果新任务中 CR3 的值与原任务中 CR3 的值相同，那么处理器不刷新分页高速缓冲寄存器，以便当任务共享页表时有较快的执行速度。

10.3.2 系统地址寄存器

全局描述符表 GDT、局部描述符表 LDT 和中断描述符表 IDT 等是保护方式下非常重要的特殊段，它们包含有对段机制所用的重要表格。为了方便快速地定位这些段，处理器采用一些特殊的寄存器保存这些段的基地址和界限。我们把这些特殊的寄存器称为系统地址寄存器。

1. 全局描述符表寄存器 GDTR

如图 10.9 所示，GDTR 长 48 位，其中高 32 位含基地址，低 16 位含界限。由于 GDT 不能由 GDT 本身之内的描述符进行描述定义，所以处理器采用 GDTR 为 GDT 这一特殊的系统段提供一个伪描述符。GDTR 给定了 GDT，如图 10.10 所示。

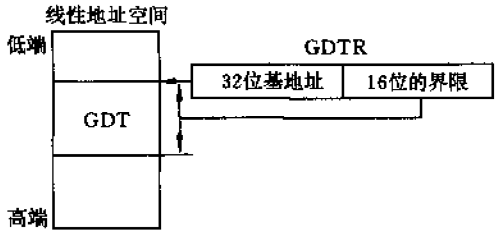


图 10.10 GDTR 给定 GDT

GDTR 中的段界限以字节为单位。由于段选择子中只有 13 位作为描述符索引，而每个描述符长 8 个字节，所以用 16 位表示 GDT 的界限足够。通常，对于含有 N 个描述符的

描述符表的段界限应设置为 8 * N－1。

利用结构类型可定义伪描述符如下：

```
PDESC   STRUC
LIMIT    DW     0
BASE     DD     0
PDESC    ENDS
```

2. 局部描述符表寄存器 LDTR

局部描述符表寄存器 LDTR 规定当前任务使用的局部描述符表 LDT。如图 10.9 所示，LDTR 类似于段寄存器，由程序员可见的一个 16 位的寄存器和程序员不可见的高速缓冲寄存器组成。实际上，每个任务的局部描述符表 LDT 作为系统的一个特殊段，由一个描述符描述，而用于描述 LDT 的描述符存放在 GDT 中。在初始化或任务切换过程中，把指示描述对应任务 LDT 的描述符的选择子装入 LDTR，处理器根据装入 LDTR 可见部分的选择子，从 GDT 中取出对应的描述符，并把 LDT 的基地址和界限等信息保存到 LDTR 的不可见的高速缓冲寄存器中。随后对 LDT 的访问，就可根据保存在高速缓冲寄存器中的有关信息进行合法性检查。

LDTR 寄存器包含当前任务的 LDT 的选择子。所以，装入到 LDTR 的选择子必须确定一个位于 GDT 的类型为 LDT 的系统段描述符，也即选择子中的 TI 位必须是 0，而且描述符中的类型字段所表示的类型必须是 LDT。

可以用一个空选择子装入 LDTR，这表示当前任务没有 LDT。在这种情况下，所有装入到段寄存器的选择子都必须指示 GDT 中的描述符，也即当前任务涉及的段均由 GDT 中的描述符来描述。如果再把一个 TI 为 1 的选择子装入到段寄存器，将引起异常。

3. 中断描述符表寄存器 IDTR

中断描述符表寄存器 IDTR 指向中断描述符表 IDT。如图 10.9 所示，IDTR 长 48 位，其中 32 位的基地址规定 IDT 的基地址，16 位的界限规定 IDT 的段界限。由于 80386 只支持 256 个中断/异常，所以 IDT 表最大长度是 2K，以字节为单位的段界限为 7FFH。IDTR 指示 IDT 表的方式与 GDTR 指示 GDT 表的方式相同。

4. 任务状态段寄存器 TR

任务状态段寄存器 TR 包含指示描述当前任务的任务状态段的描述符选择子，从而规定了当前任务的状态段。任务状态段的格式在 10.5.2 节说明。如图 10.9 所示，TR 也有程序员可见和不见两部分。当把任务状态段的选择子装入到 TR 可见部分时，处理器自动把选择子所索引的描述符中的段基地址等信息保存到不可见的高速缓冲寄存器中。在此之后，对当前任务状态段的访问可快速方便地进行。装入到 TR 的选择子不能为空，必须索引位于 GDT 中的描述符，且描述符的类型必须是 TSS。

10.4 实方式与保护方式切换实例

本节介绍两个实现实方式与保护方式切换的实例，通过它们说明如何实现实方式与保护方式的切换，也说明保护方式下的 80386 及其编程。

10.4.1 演示实方式和保护方式切换的实例(实例一)

实例一的逻辑功能是，以十六进制数的形式显示从内存地址 110000H 开始的 256 个字节的值。本实例指定该内存区域的目的仅仅是想说明切换到保护方式的必要性，因为在实方式下不能访问该指定内存区域，只有在保护方式下才能访问到该指定区域。

本实例的具体实现步骤是：(1)作切换到保护方式的准备；(2)切换到保护方式；(3)把指定内存区域的内容传送到位于常规内存的缓冲区中；(4)切换回实方式；(5)显示缓冲区内容。

1. 实例一源程序

实例一的源程序如下所示：

```
;程序名：T10-1.ASM
;功  能：演示实方式和保护方式切换
;16 位偏移的段间直接转移指令的宏定义
JUMP    MACRO selector,offsetv
        DB      0EAH                        ;操作码
        DW      offsetv                     ;16 位偏移
        DW      selector                    ;段值或者选择子
        ENDM
;字符显示宏指令的定义
ECHOCH      MACRO ascii
        MOV     AH,  2
        MOV     DL,  ascii
        INT     21H
        ENDM
;存储段描述符结构类型的定义
DESCRIPTOR    STRUC
LIMITL        DW      0                     ;段界限(0～15)
BASEL         DW      0                     ;段基地址(0～15)
BASEM         DB      0                     ;段基地址(16～23)
ATTRIBUTES    DW      0                     ;段属性
BASEH         DB      0                     ;段基地址(24～31)
DESCRIPTOR    ENDS
;伪描述符结构类型的定义
PDESC         STRUC
LIMIT         DW      0                     ;16 界限
BASE          DD      0                     ;基地址
PDESC         ENDS
;常量定义
ATDW    =     92H                           ;存在的可读写数据段属性值
ATCE    =     98H                           ;存在的只执行代码段属性值
```

```
;
                .386P
;------------------------------------------------------------------------
;数据段
DSEG    SEGMENT    USE16                       ;16 位段
GDT             LABEL BYTE                     ;全局描述符表 GDT
DUMMY           DESCRIPTOR    <>               ;空描述符
CODE            DESCRIPTOR <0FFFFH,,,ATCE,>
CODE_SEL        = CODE - GDT                   ;代码段描述的选择子
DATAS           DESCRIPTOR <0FFFFH,0H,11H,ATDW,0>
DATAS_SEL       = DATAS - GDT                  ;源数据段描述符的选择子
DATAD           DESCRIPTOR <0FFFFH,,,ATDW,>
DATAD_SEL       = DATAD - GDT                  ;目标数据段描述符的选择子
GDTLEN          = $-GDT
;
VGDTR           PDESC <GDTLEN-1,>              ;伪描述符
;
BUFFERLEN         =     256                    ;缓冲区字节长度
BUFFER          DB BUFFERLEN   DUP (0)         ;缓冲区
DSEG            ENDS
;------------------------------------------------------------------------
;代码段
CSEG    SEGMENT    USE16                       ;16 位段
        ASSUME     CS:CSEG,    DS:DSEG
START:
        MOV     AX, DSEG
        MOV     DS, AX
        ;准备要加载到 GDTR 的伪描述符
        MOV     BX,16
        MUL     BX                             ;计算并设置 GDT 基地址
        ADD     AX,OFFSET   GDT                ;界限已在定义时设置妥当
        ADC     DX,0
        MOV     WORD  PTR   VGDTR.BASE,     AX
        MOV     WORD  PTR   VGDTR.BASE+2,   DX
        ;设置代码段描述符
        MOV     AX, CS
        MUL     BX
        MOV     CODE.BASEL,AX                  ;代码段开始偏移为 0
        MOV     CODE.BASEM,DL                  ;代码段界限已在定义时设置妥当
        MOV     CODE.BASEH,DH
        ;设置目标数据段描述符
        MOV     AX,DS
```

```
        MUL     BX                              ;计算并设置目标数据段基地址
        ADD     AX,OFFSET  BUFFER
        ADC     DX,0
        MOV     DATAD.BASEL,AX
        MOV     DATAD.BASEM,DL
        MOV     DATAD.BASEH,DH
        ;加载 GDTR
        LGDT    QWORD  PTR  VGDTR
        ;
        CLI                                     ;关中断
        CALL    ENABLEA20                       ;打开地址线 A20
        ;切换到保护方式
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX
        ;清指令预取队列,并真正进入保护方式
        JUMP    <CODE_SEL>,<OFFSET VIRTUAL>
        ;
VIRTUAL:;现在开始在保护方式下
        MOV     AX,DATAS-SEL
        MOV     DS,AX                           ;加载源数据段描述符
        MOV     AX,DATAD-SEL
        MOV     ES,AX                           ;加载目标数据段描述符
        CLD
        XOR     SI,SI                           ;设置指针初值
        XOR     DI,DI
        MOV     CX,BUFFERLEN/4                  ;设置 4 字节为单位的缓冲区长度
        REPZ    MOVSD                           ;传送
        ;切换回实方式
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH;
        MOV     CR0,EAX
        ;清指令预取队列,进入实方式
        JUMP    <SEG  REAL>,<OFFSET  REAL>
        ;
REAL:   ;现在又回到实方式
        CALL    DISABLEA20                      ;关闭地址线 A20
        STI                                     ;开中断
        ;
        MOV     AX,DSEG                         ;重置数据段寄存器
        MOV     DS,AX
        MOV     SI,OFFSET  BUFFER
```

```
        CLD                                         ;显示缓冲区内容
        MOV     BP,BUFFERLEN/16
NEXTLINE:
        MOV     CX,16
NEXTCH:
        LODSB
        PUSH    AX
        SHR     AL,4
        CALL    TOASCII
        ECHOCH      AL
        POP     AX
        CALL    TOASCII
        ECHOCH      AL
        ECHOCH      ' '
        LOOP    NEXTCH
        ECHOCH      0DH
        ECHOCH      0AH
        DEC     BP
        JNZ     NEXTLINE
;
        MOV     AX,4C00H                            ;结束
        INT     21H
;
TOASCII     PROC
;把 AL 低 4 位的十六进制数转换成对应的 ASCII,保存在 AL
TOASCII     ENDP
;
EA20    PROC
;打开地址线 A20,见下面的说明
EA20    ENDP
;
DA20    PROC
;关闭地址线 A20,见下面的说明
DA20    ENDP
CSEG    ENDS
        END     START
```

2. 关于实现步骤的注释

在源程序中首定义了两条宏指令,一条是段间直接转移宏指令,另一条是显示字符宏指令。此外还定义了描述符和伪描述符的结构类型。下面对各实现步骤作些说明。

(1) 切换到保护方式的准备工作

在从实方式切换到保护方式之前,必须作必要的准备。准备工作的内容根据实际应用

而定。最起码的准备工作是建立合适的全局描述符表，并使GDTR指向该GDT。因为在切换到保护方式之时，至少要把代码段的选择子装载到CS，所以GDT中至少要含有代码段的描述符。

从本实例源程序可见，全局描述符表GDT仅有四个描述符：第一个是空描述符；第二个是代码段描述符；第三和第四个是数据段描述符。本实例各描述符中的段界限是在定义时预置的，并且除伪描述符VGDTR中的界限按GDT的实际长度设置外，各使用的存储段描述符的界限都规定为0FFFFH。另外，描述符中的段属性也根据所描述段的类型被预置，98H表示存在的只可执行代码段，92H表示存在的可读写数据段。从属性值可知，这三个存储段都是16位段。

由于在切换到保护方式后，就要引用GDT，所以在切换到保护方式前须装载GDTR。本实例使用如下指令装载GDTR：

```
LGDT    QWORD  PTR  VGDTR
```

该指令的功能是把存储器中的伪描述符VGDTR装入到全局描述符表寄存器GDTR。伪描述符VGDTR的结构如前述结构类型PDESC所示，低字是以字节为单位的界限，高双字是基地址。在10.8节中对LGDT指令作详细说明。

(2) 由实方式切换到保护方式

在做好准备后，从实方式切换到保护方式并不繁难。原则上只要把控制寄存器CR0中的PE位置1就可。本实例采用如下三条指令设置PE位：

```
MOV     EAX,CR0         ;把CR0复制到EAX
OR      EAX,1           ;把对应的PE位置1
MOV     CR0,EAX         ;使CR0的PE位为1
```

实际情况要比这复杂些。在执行上面的三条指令后，处理器转入保护方式，但CS中的内容还是实方式下代码段的段值，而不是保护方式下代码段的选择子，所以在取指令之前得把代码段选择子装入CS。为此，紧接着这三条指令，安排一条如下所示的段间转移指令：

```
JUMP    <CODE_SEL>,<OFFSET  VIRTUAL>
```

这条段间转移指令在实方式下被预取，在保护方式下被执行。利用这条段间转移指令可把保护方式下代码段的选择子装入CS，同时也刷新指令预取队列。从此真正进入保护方式。

(3) 由保护方式切换到实方式

在80386上，从保护方式切换到实方式的过程类似于从实方式切换到保护方式。原则上只要把控制寄存器CR0中的PE位清0就可。实际上，在此之后也要安排一条段间转移指令，一方面清指令预取队列，另一方面把实方式下代码段的段值送CS。这条段间转移指令在保护方式下被预取，在实方式下被执行。

(4) 传送

传送是在保护方式下进行的。首先，把源数据段和目标数据段描述符的选择子装入

DS 和 ES 寄存器，这两个描述符已在实方式下设置好，把选择子装入段寄存器就意味着把包括段基地址在内的段信息装入段描述符高速缓冲寄存器。然后，设置指针寄存器 SI 和 DI 的初值，也设置计数器 CX 初值。根据预置的段属性，在保护方式下，代码段也仅是 16 位段，串操作指令只使用 16 位的 SI、DI 和 CX 等寄存器。最后利用串操作指令实施传送。

(5) 显示缓冲区内容

由于缓冲区在常规内存中，所以在实方式下根据要求按十六进制显示其内容是容易的。

3. 内存映象

源程序中没有把 GDT 单独作为一个段对待，但在进入保护方式后，它是一个独立的段。从对代码段和源数据段描述符所赋的基地址和界限可见，代码段和数据段有部分覆盖。尽管这样做不利于代码和数据的安全，但如果需要，这样做是可行的。本实例运行时的内存映象如图 10.11 所示。

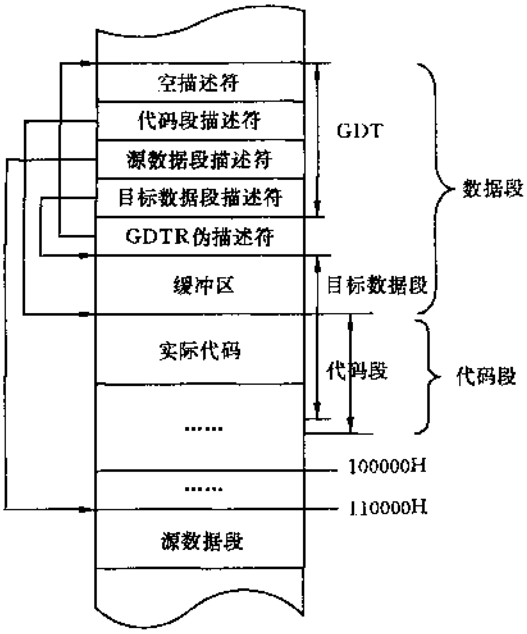


图 10.11 实例一的内存映象

4. 特别说明

作为第一个实方式和保护方式切换的例子，本实例作了大量的简单化处理。

通常由实方式切换到保护方式的准备工作还应包含建立中断描述符表。但本实例没有建立中断描述符表。为此，要求整个过程在关中断的情况下进行；要求不使用软中断指令；假设不发生任何异常。否则会导致系统崩溃。

本实例没有使用局部描述符表，所以在进入保护方式后没有设置局部描述符表寄存器 LDTR。为此，在保护方式下使用的段选择子都指定 GDT 中的描述符。

本实例没有定义保护方式下的堆栈段，GDT 中没有堆栈段描述符，在保护方式下没有设置 SS，所以在保护方式下没有涉及堆栈操作的指令。

本实例各描述符特权级 DPL 和各选择子请求特权级 RPL 均是 0，在保护方式下执行时的当前特权级 CPL 也是 0。

本实例没有采用分页管理机制，也即 CR0 中的 PG 位为 0，线性地址就是存储单元的物理地址。

5. 打开和关闭地址线 A20

PC 及其兼容机的第 20 根地址线较特殊，计算机系统中一般安排一个“门”控制该地址线是否有效。为了访问地址在 1M 以上的存储单元，应先打开控制地址线 A20 的“门”。这种设置与实方式下只使用最低端的 1M 字节存储空间有关，与处理器是否工作在实方式和保护方式无关，即使在关闭地址线 A20 时，也可进入保护方式。

如何打开和关闭地址线 A20 与计算机系统的具体设置有关。如下的两个过程，在一般的 PC 兼容机上都是可行的。

```
;打开地址线 A20
EA20    PROC
        PUSH    AX
        IN      AL,92H
        OR      AL,2
        OUT     92H,AL
        POP     AX
        RET
EA20    ENDP
;
;关闭地址线 A20
DA20    PROC
        PUSH    AX
        IN      AL,92H
        AND     AL,0FDH     ;0FDH = NOT 20H
        OUT     92H,AL
        POP     AX
        RET
DA20    ENDP
```

10.4.2 演示 32 位代码段和 16 位代码段切换的实例(实例二)

实例二的逻辑功能是，以十六进制数和 ASCII 字符两种形式显示从内存地址 100000H 开始的 16 个字节的内容。

从功能上看本实例类似于实例一，但在实现方法上却有了改变，它更能反映出实方式和保护方式切换的情况。具体实现步骤是：(1)作切换到保护方式的准备；(2)切换到保护方式的一个 32 位代码段；(3)把指定内存区域的内容以字节为单位，转换成对应十六进制

数的 ASCII 码，并直接填入显示缓冲区实现显示；(4)再变换到保护方式下的一个 16 位代码段；(5)把指定内存区域的内容直接作为 ASCII 码填入显示缓冲区实现显示；(6)切换回实方式。

1. 实例二源程序

实例二的源程序如下所示：

```
;程序名：T10-2.ASM
;功  能：演示实方式和保护方式切换
;说  明：该程序使用 TASM 汇编
;16 位偏移的段间转移指令的宏定义
JUMP16      MACRO selector,offsetv
            DB      0EAH                         ;操作码
            DW      offsetv                      ;16 位偏移
            DW      selector                     ;段值或者选择子
            ENDM
;32 位偏移的段间转移指令的宏定义
JUMP32      MACRO selector,offsetv
            DB      0EAH                         ;操作码
            DW      offsetv                      ;32 位偏移
            DW      0
            DW      selector                     ;选择子
            ENDM
;存储段描述符结构类型的定义
DESCRIPTOR      STRUC
LIMITL          DW      0                        ;段界限(0~15)
BASEL           DW      0                        ;段基地址(0~15)
BASEM           DB      0                        ;段基地址(16~23)
ATTRIBUTES      DW      0                        ;段属性
BASEH           DB      0                        ;段基地址(24~31)
DESCRIPTOR      ENDS
;伪描述符结构类型的定义
PDESC       STRUC
LIMIT       DW      0                            ;16 界限
BASE        DD      0                            ;基地址
PDESC       ENDS
;常量定义
ATDR    = 90H                                    ;存在的只读数据段属性值
ATDW    = 92H                                    ;存在的可读写数据段属性值
ATDWA = 93H                                      ;存在的已访问可读写数据段属性值
ATCE    = 98H                                    ;存在的只执行代码段属性值
ATCE32 = 4098H                                   ;存在的只执行 32 位代码段属性值
DATALEN =   16
```

```
;
          .386P
;------------------------------------
;数据段
DSEG        SEGMENT   USE16              ;16 位段
GDT         LABEL     BYTE               ;全局描述符表
DUMMY       DESCRIPTOR  <>               ;空描述符
CODE32_SEL    =    08H                   ;32 位代码段描述符选择子
CODE32      DESCRIPTOR  <CODE32LEN-1,,,ATCE32,>
CODE16_SEL    =    10H                   ;16 位代码段描述符选择子
CODE16      DESCRIPTOR  <0FFFFH,,,ATCE,>
DATAS_SEL     =    18H                   ;源数据段描述符选择子
DATAS       DESCRIPTOR  <DATALEN-1,0FFF0H,0FH,ATDR,0>
DATAD_SEL     =    20H                   ;目标数据段描述符选择子
DATAD       DESCRIPTOR  <DATALEN * 8-1,80A0H,0BH,ATDW,0>
STACKS_SEL    =    28H                   ;堆栈段描述符选择子
STACKS      DESCRIPTOR  <0FFFFH,,,ATDWA,>
NORMAL_SEL    =    30H                   ;规范段描述符选择子
NORMAL      DESCRIPTOR  <0FFFFH,0,0,ATDW,0>
GDTLEN    =   $ -GDT
;
VGDTR       PDESC <GDTLEN-1,>            ;GDT 伪描述符
VARSS       DW     ?                     ;用于保存 SS 的变量
DSEG        ENDS
;------------------------------------
;实方式下的代码段
CSEG1   SEGMENT    USE16     'REAL'
        ASSUME     CS:CSEG1,DS:DSEG
START:
        MOV       AX,DSEG
        MOV       DS,AX
        ;
        MOV       BX,16                        ;设置 GDT 的基地址
        MUL       BX
        ADD       AX,OFFSET  GDT
        ADC       DX,0
        MOV       WORD  PTR  VGDTR.BASE,AX
        MOV       WORD  PTR  VGDTR.BASE+2,DX
        ;
        MOV       AX,CSEG2                     ;设置 32 位代码段的基地址
        MUL       BX
        MOV       CODE32.BASEL,AX
```

```
        MOV     CODE32.BASEM,DL
        MOV     CODE32.BASEH,DH
        ;
        MOV     AX,CSEG3                    ;设置16位代码段的基地址
        MUL     BX
        MOV     CODE16.BASEL,AX
        MOV     CODE16.BASEM,DL
        MOV     CODE16.BASEH,DH
        ;
        MOV     AX,SS                       ;设置堆栈段的基地址
        MUL     BX
        MOV     STACKS.BASEL,AX
        MOV     STACKS.BASEM,DL
        MOV     STACKS.BASEH,DH
        MOV     VARSS,SS                    ;保存实方式下的SS
        ;
        LGDT    QWORD  PTR  VGDTR           ;装载GDTR
        ;
        CLI
        CALL    EA20                        ;打开地址线A20
        ;
        MOV     EAX,CR0                     ;切换到保护方式
        OR      EAX,1
        MOV     CR0,EAX
        ;进入32位代码段
        JUMP16  CODE32.SEL,<OFFSET SPM32>
        ;
TOREAL:;现在又回到实方式
        MOV     AX,DSEG
        MOV     DS,AX
        MOV     SS,VARSS                    ;恢复实方式下的SS
        CALL    DA20                        ;关闭地址线A20
        STI
        MOV     AH,4CH                      ;结束
INT     21H
;
EA20    PROC
        ;打开地址线A20
EA20    ENDP
;
DA20    PROC
        ;关闭地址线A20
```

```
DA20    ENDP
CSEG1   ENDS
;----------------------------------
;32 位代码段
CSEG2   SEGMENT     USE32   'PM32'
        ASSUME       CS : CSEG2
SPM32:
        MOV     AX,STACKS_SEL
        MOV     SS,AX                       ;装载堆栈段寄存器 SS
        MOV     AX,DATAS_SEL
        MOV     DS,AX                       ;装载源数据段寄存器 DS
        MOV     AX,DATAD_SEL
        MOV     ES,AX                       ;装载目标数据段寄存器 ES
        ;
        XOR     ESI,ESI                     ;设置指针和计数器
        XOR     EDI,EDI
        MOV     ECX,DATALEN
        CLD
NEXT:
        LODSB                               ;取一字节
        PUSH    AX
        CALL    TOASCII                     ;低 4 位转换成 ASCII
        MOV     AH,7                        ;显示属性为黑底白字
        SHL     EAX,16
        POP     AX
        SHR     AL,4
        CALL    TOASCII                     ;高 4 位转换成 ASCII
        MOV     AH,7
        STOSD                               ;显示
        MOV     AL,' '
        STOSW                               ;显示空格
        LOOP    NEXT
        ;变换到 16 位代码段
        JUMP32 CODE16_SEL,<OFFSET SPM16>
;
TOASCII     PROC
        ;把 AL 低位转换成对应 ASCII 码
TOASCII     ENDP
CODE32LEN   =     $
CSEG2   ENDS
;----------------------------------
;16 位代码段
```

```
CSEG3   SEGMENT    USE16 'PM16'
        ASSUME     CS : CSEG3
SPM16:
        XOR        SI,SI                          ;设置指针和计数器
        MOV        DI,DATALEN * 3 * 2
        MOV        AH,7
        MOV        CX,DATALEN
AGAIN:
        LODSB                                     ;把指定区域内容直接
        STOSW                                     ;作为 ASCII 码显示
        LOOP       AGAIN
        ;
        MOV        AX,NORMAL_SEL
        MOV        DS,AX                          ;把 NORMAL 段选择子装入 DS 和
ES
        MOV        ES,AX
        ;
        MOV        EAX,CR0                        ;切换到实方式
        AND        EAX,0FFFFFFFEH
        MOV        CR0,EAX
        ;切换回到实方式
        JMP        FAR    PTR   TOREAL
CSEG3   ENDS
        END        START
```

2. 关于实现步骤的注释

(1) 切换到保护方式的准备工作

建立全局描述符表,这里的全局描述符表含有两个 16 位数据段的描述符、一个 16 位代码段的描述符和一个 16 位堆栈段的描述符。此外,GDT 还有一个 32 位代码段的描述符,描述 32 位代码段,该描述符的属性字段中的 D 为是 1。

(2) 工作方式切换

由实方式切换到保护方式 32 位代码段的方法与切换到保护方式 16 位代码段的方法相同。由保护方式 16 位代码段切换回实方式的方法与实例一相似。

在保护方式下,通过如下直接段间转移指令从 32 位代码段切换到 16 位代码段:

```
JUMP32    CODE16_SEL,<OFFSET  SPM16>
```

从该宏指令的定义可知,该转移指令含 48 位指针,其高 16 位是 16 位代码段的选择子,低 32 位是 16 位代码段入口偏移。该指令在 32 位方式下预取,在 16 位方式下执行。由于在 32 位方式下,所以要使用 48 位指针。

(3) 显示指定区域的内容

在本实例中,采用直接写显示缓冲区的方法实现显示。假设显示缓冲区的开始物理地

址是 B8000H,3 号显示方式下,在屏幕的第 2 行上进行显示。

3. 特别说明

尽管本实例没有自己专用的堆栈段,但还是在原堆栈的基础上建立了堆栈段,所以在保护方式下使用了涉及堆栈操作的指令。

本实例仍作了大量的简单化处理。如:没有建立 IDT 和 LDT 等,各特权级均是 0。也没有采用分页管理机制。

从本实例的 GDT 中可见,两个数据段的界限都是根据实际大小而设置的。从源程序代码段 CSEG3 可见,在切换到实方式之前,把一个指向似乎没有用的数据段描述符 NORMAL 的选择子装载到 DS 和 ES。这是为什么呢?

表 10.4 实方式下段描述符高速缓冲寄存器之内容

段寄存器	段基地址	段界限(固定)	其他段属性(固定)									
			存在性	特权级	已存取	粒度	扩展方向	可读性	可写性	可执行	堆栈大小	一致特权
CS	当前 CSx16	0000FFFFH	Y	0	Y	B	U	Y	Y	Y	—	N
SS	当前 SSx16	0000FFFFH	Y	0	Y	B	U	Y	Y	N	W	—
DS	当前 DSx16	0000FFFFH	Y	0	Y	B	U	Y	Y	N	—	—
ES	当前 ESx16	0000FFFFH	Y	0	Y	B	U	Y	Y	N	—	—
FS	当前 FSx16	0000FFFFH	Y	0	Y	B	U	Y	Y	N	—	—
GS	当前 GSx16	0000FFFFH	Y	0	Y	B	U	Y	Y	N	—	—

在 10.2.5 节中已介绍过每个段寄存器都配有段描述符高速缓冲寄存器,这些高速缓冲寄存器在实方式下仍发挥作用,只是内容上与保护方式下有所不同。如表 10.4 所示,其中"Y"表示"是";"N"表示"否";"B"表示字节;"U"表示向高扩展段;"W"表示字方式操作堆栈。段基地址仍是 32 位,其值是相应段寄存器值(段值)乘 16,在把段值装载到段寄存器时被刷新。由于其值是 16 位段值乘上 16,所以在实方式下基地址实际有效位只有 20 位。每个段的 32 位段界限都固定为 0FFFFH,段属性的许多位也是固定的。所谓固定是指在实方式下必须是表 10.4 中所列值。但在实方式下,不可设置这些属性值,只能继续沿用保护方式下所设置的值。因此,在准备结束保护方式回到实方式之前,要通过加载一个合适的描述符的选择子到有关段寄存器,以使得对应段描述符高速缓冲寄存器中含有合适的段界限和属性。本实例 GDT 中的段描述符 NORMAL 就是这样的一个描述符,在返回实方式之前把对应选择子 NORMAL_SEL 加载到 DS 和 ES 就是为此目的。由于 SS 段描述符中的内容已符合实方式的需要,所以尽管在也改变了 SS 后,但没有重新加载 SS。16 位代码段描述符中的内容也符合实方式的需要,所以在通过 16 位代码段返回时,CS 段描述符高速缓冲寄存器中的内容也是符合要求的。顺便说一下,实例一的描述符都是符合实方式要求的。

4. 关于 32 位代码段程序设计的说明

在 32 位代码段中,缺省的操作数大小是 32 位,缺省的存储单元地址大小也是 32 位。

由于串操作指令使用的指针寄存器是ESI和EDI，LOOP指令使用的计数器是ECX，所以，在代码段CSEG2中，为了使用串操作指令，对ESI和EDI等寄存器赋初值。请比较代码段CSEG3中的相关片段和实例一中的相关片段，它们是16位代码段。

10.5 任务状态段和控制门

每个任务有一个任务状态段TSS，用于保存任务的有关信息，在任务内变换特权级和任务切换时，要使用这些信息。为了控制任务内发生特权级变换的转移，为了控制任务切换，一般要通过控制门进行这些转移。本节介绍任务状态段和控制门。

10.5.1 系统段描述符

系统段是为实现存储管理机制所使用的一种特别的段。在80386中，有两种系统段：任务状态段TSS和局部描述符表LDT段。用于描述系统段的描述符称为系统段描述符，也称为特殊段描述符。

1. 系统段描述符的一般格式

系统段描述符的一般格式如图10.12所示。与图10.5所示的存储段描述符相比，它们很相似，区分的标志是属性字节中的描述符类型位DT的值。DT=1表示存储段，DT=0表示系统段。系统段描述符中的段基地址和段界限字段与存储段描述符中意义完全相同；属性中的G位、AVL位、P位和DPL字段的作用也完全相同。存储段描述符属性中的D位在系统段描述符中不使用，现用符号X表示。系统段描述符的类型字段TYPE仍是4位，其编码及表示的类型列于表10.5，其含义与存储段描述符的类型却完全不同。

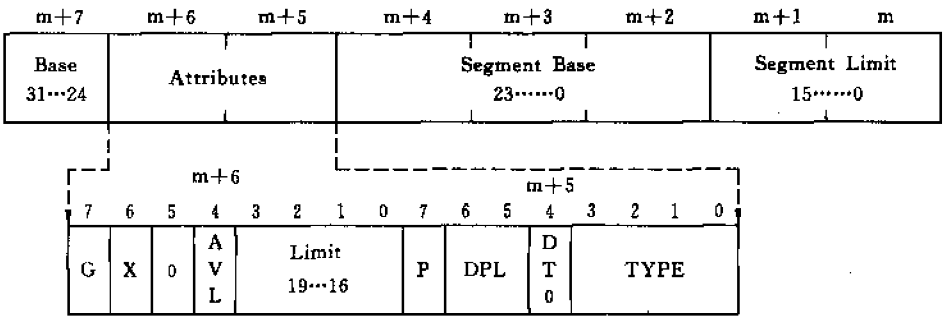


图10.12 系统段描述符格式

从表10.5可见，只有类型编码为2、1、3、9和B的描述符才是真正的系统段描述符，它们用于描述系统段LDT和任务状态段TSS，其它类型的描述符是门描述符。

利用在10.2节中定义的存储段描述符结构类型DESCRIPTOR仍能方便地在程序中说明系统段描述符。

表 10.5　系统段和门描述符类型字段的编码及含义

类型编码	说　明	类型编码	说　明
0	未定义	8	未定义
1	可用 286TSS	9	可用 386TSS
2	LDT	A	未定义
3	忙的 286TSS	B	忙的 386TSS
4	286 调用门	C	386 调用门
5	任务门	D	未定义
6	286 中断门	E	386 中断门
7	286 陷阱门	F	386 陷阱门

2. LDT 段描述符

LDT 段描述符描述任务的局部描述符表段。例如：如下描述符 LDTABLE 描述一个局部描述符表段，基地址是 654321H，以字节为单位的界限是 1FH，描述符特权级是 0。

```
LDTABLE    DESCRIPTOR  <1FH,4321H,65H,82H,0>
```

LDT 段描述符必须安排在全局描述符表中才有效。在装载 LDTR 寄存器时，描述符中的 LDT 段基地址和段界限等信息被装入如图 10.9 所示的 LDTR 高速缓冲寄存器中。

3. 任务状态段描述符

任务状态段 TSS 用于保存任务的各种状态信息。任务状态段描述符描述某个任务状态段。TSS 描述符分为 286TSS 和 386TSS 两类。TSS 描述符规定了任务状态段的基地址和任务状态段的大小。例如：如下描述符 TEMPTASKS 描述一个可用的 386 任务状态段，基地址是 123456H，以字节为单位的界限是 104，描述符特权级是 0。

```
MONTASKS    DESCRIPTOR    <104,3456H,12H,89H,0>
```

在装载任务状态段寄存器 TR 时，描述符中的 TSS 段基地址和段界限等信息被装入如图 10.9 所示的 TR 高速缓冲寄存器中。在任务切换或执行 LTR 指令时，要装载 TR 寄存器。

TSS 描述符中的类型规定：TSS 要么为“忙”，要么为“可用”。如果一个任务是当前正执行的任务，或者是用 TSS 中的链接字段沿挂起任务链接到当前任务上的任务，那么该任务是“忙”的任务；否则该任务为“可用”任务。

利用段间转移指令 JMP 和段间调用指令 CALL，直接通过 TSS 描述可实现任务切换。

10.5.2 门描述符

除存储段描述符和系统段描述符外，还有一类门描述符。门描述符并不描述某种内存段，而是描述控制转移的入口点。这种描述符好比一个通向另一代码段的门。通过这种门，可实现任务内特权级的变换和任务间的切换。所以，这种门描述符也称为控制门。

1. 门描述符的一般格式

门描述符的一般格式如图 10.13 所示。门描述符只有位于描述符内偏移 5 的类型字节与系统段描述符保持一致，也由该字节标识门描述符和系统段描述符。该字节内的 P 和 DPL 的意义与其他描述符中的意义相同。其它字节主要用于存放一个 48 位的全指针(16 位的选择子和 32 位的偏移量)。

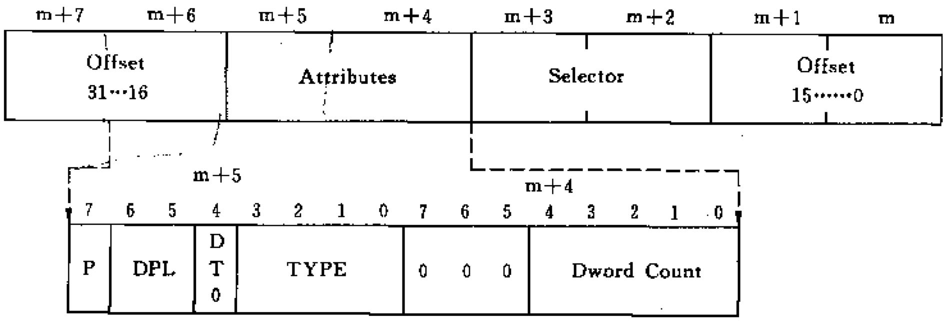


图 10.13 门描述符格式

根据如图 10.13 给出的门描述符的结构，可定义如下的门描述符结构类型：

```
GATE      STRUC        ;门结构类型定义
OFFSETL   DW    0      ;32 位偏移的低 16 位
SELECTOR  DW    0      ;选择子
DCOUNT    DB    0      ;双字计数字段
GTYPE     DB    0      ;类型
OFFSETH   DW    0      ;32 位偏移的高 16 位
GATE ENDS
```

利用门描述符结构类型 GATE 能方便地在程序中说明门描述符。

例如，如下门描述符 SUBRG 描述一个 386 调用门，门内的选择子是 10H，入口偏移是 123456H，门描述符特权级是 3，双字计数是 0。

```
SUBRG   GATE <3456H,10H,0,8CH+60H,0012H>
```

从表 10.5 可见，门描述符又可分为：任务门、调用门、中断门和陷阱门，并且除任务门外，其他门描述符还各分成 286 和 386 两种。

2. 调用门

调用门描述某个子程序的入口。调用门内的选择子必须指向代码段描述符，调用门内的偏移是对应代码段内的偏移。利用段间调用指令 CALL，通过调用门可实现任务内从外层特权级变换到内层特权级。

在图 10.13 所示门描述符内偏移 4 字节的位 0 至位 4 是双字计数字段，该字段只在调用门描述符中有效，在其它门描述符中无效。主程序通常通过堆栈把入口参数传递给子程序，如果在利用调用门调用子程序时引起特权级的转换和堆栈的改变，那么就需要将外层堆栈中的参数复制到内层堆栈。该双字计数字段就是用于说明这种情况发生时，要复制的双字参数的数量。

3. 任务门

任务门指示任务。任务门内的选择子必须指向 GDT 中的任务状态段 TSS 描述符，门中的偏移无意义。任务的入口点保存在 TSS 中。利用段间转移指令 JMP 和段间调用指令 CALL，通过任务门可实现任务切换。

4. 中断门和陷阱门

中断门和陷阱门描述中断/异常处理程序的入口点。中断门和陷阱门内的选择子必须指向代码段描述符，门内的偏移就是对应代码段的入口点偏移。中断门和陷阱门只有在中断描述符表 IDT 中才有效。关于中断门和陷阱门的区别在 10.7 节中介绍。

10.5.3 任务状态段

任务状态段(Task State Segment)是保存一个任务重要信息的特殊段。任务状态段描述符用于描述这样的系统段。任务状态段寄存器 TR 的可见部分含有当前任务的任务状态段描述符的选择子，TR 的不可见部分含有当前任务状态段的段基地址和段界限等信息。

TSS 在任务切换过程中起着重要作用，通过它实现任务的挂起和恢复。所谓任务切换是指，挂起当前正在执行的任务，恢复另一个任务的执行。在任务切换过程中，首先，处理器中各寄存器的当前值被自动地保存到 TR 所指定的 TSS 中；然后，下一任务的 TSS 的选择子被装入 TR；最后从 TR 所指定的 TSS 中取出各寄存器的值送到处理器的各寄存器中。由此可见，通过在 TSS 中保存任务现场各寄存器状态的完整映象，实现任务的切换。

任务状态段 TSS 的基本格式如图 10.14 所示。从中可见，TSS 的基本格式有 104 字节组成。这 104 字节的基本格式是不可改变的，但在此之外系统软件还可定义若干附加信息。基本的 104 字节可分为链接字段区域、内存堆栈指针区域、地址映射寄存器区域、寄存器保存区域和其它字段等五个区域。

1. 寄存器保存区域

寄存器保存区域位于 TSS 内偏移 20H 至 5FH 处，用于保存通用寄存器、段寄存器、指令指针和标志寄存器。当 TSS 对应的任务正在执行时，保存区域是未定义的；在当前任务被切换出时，这些寄存器的当前值就保存在该区域。当下次切换回原任务时，再从保存区域恢复出这些寄存器的值，从而使处理器恢复成该任务换出前的状态，最终使任务能够恢复执行。

从图 10.14 可见，各通用寄存器对应一个 32 位的双字，指令指针和标志寄存器各对应一个 32 位的双字；各段寄存器也对应一个 32 位的双字，段寄存器中的选择子只有 16 位，安排在双字的低 16 位，高 16 位空着未用。

2. 内层堆栈指针区域

为了有效地实现保护，一个任务在不同的特权级下使用不同的堆栈。例如，当从外层特权级 3 变换到内层特权级 0 时，任务使用的堆栈也同时从 3 级堆栈变换到 0 级堆栈；当从内层特权级 0 变换到外层特权级 3 时，任务使用的堆栈也同时从 0 级堆栈变换到 3 级堆栈。所以，一个任务可能具有四个堆栈，对应四个特权级。四个堆栈需要四个堆栈指针。

<table>
<tr><td>31</td><td colspan="2">0</td><td></td></tr>
<tr><td>0000000000000000</td><td colspan="2">链接字段</td><td>0</td></tr>
<tr><td colspan="3">ESP0</td><td>4</td></tr>
<tr><td>0000000000000000</td><td colspan="2">SS0</td><td>8</td></tr>
<tr><td colspan="3">ESP1</td><td>0CH</td></tr>
<tr><td>0000000000000000</td><td colspan="2">SS1</td><td>10H</td></tr>
<tr><td colspan="3">ESP2</td><td>14H</td></tr>
<tr><td>0000000000000000</td><td colspan="2">SS2</td><td>18H</td></tr>
<tr><td colspan="3">CR3</td><td>1CH</td></tr>
<tr><td colspan="3">EIP</td><td>20H</td></tr>
<tr><td colspan="3">EFLAGS</td><td>24H</td></tr>
<tr><td colspan="3">EAX</td><td>28H</td></tr>
<tr><td colspan="3">ECX</td><td>2CH</td></tr>
<tr><td colspan="3">EDX</td><td>30H</td></tr>
<tr><td colspan="3">EBX</td><td>34H</td></tr>
<tr><td colspan="3">ESP</td><td>38H</td></tr>
<tr><td colspan="3">EBP</td><td>3CH</td></tr>
<tr><td colspan="3">ESI</td><td>40H</td></tr>
<tr><td colspan="3">EDI</td><td>44H</td></tr>
<tr><td>0000000000000000</td><td colspan="2">ES</td><td>48H</td></tr>
<tr><td>0000000000000000</td><td colspan="2">CS</td><td>4CH</td></tr>
<tr><td>0000000000000000</td><td colspan="2">SS</td><td>50H</td></tr>
<tr><td>0000000000000000</td><td colspan="2">DS</td><td>54H</td></tr>
<tr><td>0000000000000000</td><td colspan="2">FS</td><td>58H</td></tr>
<tr><td>0000000000000000</td><td colspan="2">GS</td><td>5CH</td></tr>
<tr><td>0000000000000000</td><td colspan="2">LDT</td><td>60H</td></tr>
<tr><td>I/O许可位图偏移</td><td>000000000000000</td><td>T</td><td>64H</td></tr>
</table>

图 10.14 任务状态段基本格式

TSS 的内层堆栈指针区域中有三个堆栈指针，它们都是 48 位的全指针(16 位的选择子和 32 位的偏移)，分别指向 0 级、1 级和 2 级堆栈的栈顶，依次存放在 TSS 中偏移为 4、12 及 20 开始的位置。当发生向内层转移时，则把适当的堆栈指针装入到 SS 及 ESP 寄存器以变换到内层的堆栈，外层堆栈的指针保存在内层堆栈中。没有指向 3 级堆栈的指针，因为 3 级是在最外层，所以任何一个向内层的转移都不可能转移到 3 级。

但是，当特权级由内层向外层变换时，并不把内层堆栈的指针保存到 TSS 的内层堆栈指针区域。这表明向内层转移时，总是把内层堆栈认为是一个空栈。因此，不允许发生同级内层转移的递归，一旦发生向某级内层转移，那么返回到外层的正常途径是相匹配的向外层返回。

3. 地址映射寄存器区域

由虚拟地址空间到线性地址空间的映射由 GDT 和 LDT 确定，与特定任务相关的部分由 LDT 确定，而 LDT 又由 LDTR 确定。如果采用分页机制，那么由线性地址空间到物理地址空间的映射由包含页目录表起始物理地址的控制寄存器 CR3 确定。所以，与特定

任务相关的虚拟地址空间到物理地址空间的映射由 LDTR 和 CR3 确定。显然，随着任务的切换，地址映射关系也要切换。

TSS 的地址映射寄存器区域由位于偏移 1CH 处的双字字段(CR3)和位于偏移 60H 处的字字段(LDT)组成。在任务切换时，处理器自动从轮到执行的任务的 TSS 中取出这两个字段，分别装入到寄存器 CR3 和寄存器 LDTR。这样就改变了虚拟地址空间到物理地址空间的映射。

但是，在任务切换时，处理器并不把换出任务当时的寄存器 CR3 和 LDTR 的内容保存到 TSS 中的地址映射寄存器区域。因此，如果程序改变了 LDTR 或 CR3，那么必须把新值保存到 TSS 中的地址映射寄存器区域相应字段中。

4. 链接字段

链接字段安排在 TSS 内偏移 0 开始的双字中，其高 16 位未用。在起链接作用时，低 16 位保存前一任务的 TSS 描述符的选择子。

如果当前的任务由段间调用指令 CALL 或者中断/异常而激活，那么链接字段保存被挂起任务的 TSS 的选择子，并且标志寄存器 EFLAG 中的 NT 位被置 1，使链接字段有效。在返回时，由于 NT 位为 1，中断返回指令 IRET 将使得控制沿着链接字段所指恢复到链上的前一个任务。

5. 其它字段

为了实现输入/输出保护，要使用 I/O 许可位图。任务使用的 I/O 许可位图也存放在 TSS 中，作为 TSS 的扩展部分。在 TSS 内偏移 66H 处的字用于存放 I/O 许可位图在 TSS 内的开始偏移。关于 I/O 许可位图的作用在 10.9 节中介绍。

在 TSS 内偏移 64H 处的字是为任务提供的特别属性。在 80386 中，只定义了一种属性，即调试陷阱。该属性是字的最低位，用 T 表示。该字的其他位被保留，必须被置成 0。在发生任务切换时，如果进入任务的 T 位为 1，那么在任务切换完成之后，新任务的第一条指令执行之前产生调试陷阱。

6. 用结构类型定义 TSS

根据如图 10.14 给出的任务状态段 TSS 的结构，可定义如下的 TSS 结构类型：

```
TASKSS   STRUC
         DW      ?,0      ;链接字
         DD      ?        ;0 级堆栈指针
         DW      ?,0
         DD      ?        ;1 级堆栈指针
         DW      ?,0
         DD      ?        ;2 级堆栈指针
         DW      ?,0
         DD      ?        ;CR3
         DD      ?        ;EIP
         DW      ?,?      ;EFLAGS
         DD      ?        ;EAX
         DD      ?        ;ECX
```

```
        DD      ?       ;EDX
        DD      ?       ;EBX
        DD      ?       ;ESP
        DD      ?       ;EBP
        DD      ?       ;ESI
        DD      ?       ;EDI
        DW      ?,0     ;ES
        DW      ?,0     ;CS
        DW      ?,0     ;SS
        DW      ?,0     ;DS
        DW      ?,0     ;FS
        DW      ?,0     ;GS
        DW      ?,0     ;LDT
        DW      0       ;TSS的特别属性字
        DW      $+2     ;指向I/O许可位图区的指针
        DB      0FFH    ;I/O许可位图结束字节
TASKSS  ENDS
```

10.6 控制转移

控制转移基本上可分为两大类:同一任务内的控制转移和任务间的控制转移(任务切换)。同一任务内的控制转移又分为:段内转移、特权级不变的段间转移和特权级变换的段间转移。段内转移与实方式下相似,不涉及特权级变换和任务切换。只有段间转移才涉及特权级变换和任务切换。本节介绍保护方式下的控制转移,重点是任务内的特权级变换和任务间的切换。

10.6.1 任务内无特权级变换的转移

各种段内转移与实方式下相似,当然不涉及特权级变换和任务切换。只有各种形式的段间转移才涉及特权级变换和任务切换。

1. 段间转移指令

与实方式下一样,指令 JMP、CALL 和 RET 都具有段间转移的功能,指令 INT 和 IRET 总是段间转移。此外,中断/异常也将引起段间转移。有时把这些具有段间转移功能的指令统称为段间转移指令。

在保护方式下,段间转移的目标位置由选择子和偏移构成的地址表示,常把它称为目标地址指针。在32位代码段中,上述指针内的偏移使用32位表示,这样的指针也称为48位全指针。在实例二的32位代码段内就使用了48位全指针。在16位代码段中,上述指针内的偏移只使用16位表示。

与实方式下相似,段间转移指令 JMP 和段间调用指令 CALL 还可分为段间直接转移和段间间接转移两类。如果指令 JMP 和 CALL 在指令中直接含有目标地址指针,那么

就是段间直接转移；如果指令中含有指向包含目标地址指针的门描述符或 TSS 描述符的指针，那么就是段间间接转移，这种指针只有选择子部分有效，指示调用门、任务门或 TSS 描述符，而偏移部分不起作用。实际上，当段间转移指令 JMP 和段间调用指令 CALL 所含指针的选择子部分指示代码段描述符，那么就是段间直接转移，偏移部分表示目标代码段的入口点；当选择子部分指示门描述符或 TSS 描述符时，就是段间间接转移。

2. 向目标代码段转移的步骤

处理器在执行上述段间转移指令向目标代码段实施转移的过程中，一般至少要经过如下步骤：

(1) 判别目标地址指针内的选择子指示的描述符是否为空描述符。空描述符是 GDT 中的第 0 个描述符，是一个特殊的描述符。目标代码段描述符不能为空描述符，也即选择子的高 14 位不能为 0。

(2) 从全局或者局部描述符表内读出目标代码段描述符。由选择子内的 TI 位，确定使用全局描述符表还是局部描述表。

(3) 根据情况，检测描述符类型是否正确；调整 RPL。

(4) 把目标代码段描述符内的有关内容装载到 CS 高速缓冲寄存器。

(5) 判别目标地址指针内的偏移是否越出代码段。目标地址指针内的偏移必须不超过目标代码段界限。

(6) 装载 CS 段寄存器和指令指针寄存器 EIP；CPL 存入 CS 内选择子的 RPL 字段。

上述步骤只是对转移过程的简单说明，实际的动作还要复杂。在把目标代码段描述符内的有关内容装载到 CS 高速缓冲寄存器时，还要进行如下保护检测，其中的 DPL 表示目标代码段描述符特权级：

(1) 对于非一致代码段，要求 CPL＝DPL，RPL＜＝DPL；对于一致代码段，要求 CPL ＞＝DPL。

(2) 代码段必须存在，即描述符中的 P 位必须为 1。

通常描述符特权级 DPL 规定了对应段的特权级。如果描述符描述的是数据段，那么 DPL 就规定了访问该数据段的最外层特权级；如果描述符描述的是代码段，那么 DPL 就规定了执行该代码段所需要的 CPL。但从上述装载 CS 高速缓冲寄存器时进行的保护检测可见，对于一致代码段，却要求 CPL＞＝DPL，也就是说，一致代码段描述符中的 DPL，规定可以转移到一致的代码段的最内层特权级。于是，3 级的程序可以转移到任何一致的代码段，而 0 级的程序只允许转移到 DPL 等于 0 的一致代码段。一致代码段描述符内 DPL 的这种解释，正好与正常的 DPL 的解释相反。

一致的可执行段是一种特别的存储段。这种存储段，为在多个特权级执行的程序，提供对子例程的共享支持，而不要求改变特权级。例如，通过把数值库例程放在一致的代码段中，可以使不同级执行的程序共享数值库例程。这样，任何特权级的程序可以使用段间调用指令，调用库中的例程，并在调用者所具有的特权级执行该例程。

3. 任务内无特权级变换的转移

所谓任务内无特权级变换的转移指：在转移到新的代码段时，当前特权级 CPL 保持不变。利用段间转移指令 JMP、段间调用指令 CALL 和段间返回指令 RET 可实现任务内

无特权级变换的转移。利用 INT 指令和 IRET 指令也可实现任务内无特权级变换的转移。

(1) 利用段间直接转移指令 JMP 或 CALL

在执行段间转移指令 JMP 时,如果指令内所含指针指示一个代码段,那么就直接开始上述向目标代码段转移的步骤;在执行段间调用指令 CALL 时,如果指令内所含指针指示一个代码段,那么就把返回地址指针压入堆栈,然后就直接开始上述向目标代码段转移的步骤。顺利通过这几步(不调整 RPL),就完成了任务内无特权级变换的转移。

由此可见,利用段间直接转移指令 JMP 或调用指令 CALL 可方便地进行任务内无特权级变换的转移,但不能进行任务内特权级变换的转移。

(2) 利用段间返回指令 RET

在执行段间返回指令 RET 时,如果从堆栈中弹出的目标地址指针指示一个代码段,并且选择子符合 RPL=CPL 的条件,那么就开始上述向目标代码段转移的步骤。顺利通过这几步,就完成了任务内无特权级变换的转移。

通常情况下,段间返回指令 RET 与段间调用指令 CALL 对应。在利用段间调用指令 CALL 以任务内无特权级变换的方式转移到某个子程序后,在子程序内利用段间返回指令 RET 以任务内无特权级变换的方式返回主程序。由于调用时无特权级变换,所以返回时也无特权级变换,如果真是如此,那么必定能够满足条件 RPL=CPL。

(3) 利用调用门和其他途径

如何利用调用门实现任务内无特权级变换的转移在 10.6.3 节中介绍。其他实现任务内无特权级变换的途径在 10.7 节中介绍。

4. 装载数据段和堆栈段寄存器时的特权检测

上面简单地说明了把选择子装入代码段寄存器 CS 时为实现保护而进行的检测,下面也简单地说明在把选择子装入数据段寄存器和堆栈段寄存器时要进行的检测。

在把选择子装入数据段寄存器 DS、ES、FS 或 GS 时,要进行如下检测:

(1) 选择子不能为空;

(2) 选择子指定的描述符必须是数据段描述符、可读可执行的代码段或一致的可读可执行代码段;

(3) 对于数据段和可读可执行代码段,要求 CPL<=DPL,RPL<=DPL;

(4) 对应段必须存在。

在把选择子装入堆栈段寄存器 SS 时要进行如下检测:

(1) 选择子不能为空;

(2) 选择子指定的描述符必须是可读可写数据段描述符;

(3) 要求 CPL=RPL=DPL;

(4) 对应段必须存在。

10.6.2 演示任务内无特权级变换转移的实例(实例三)

在实例二中,32 位代码段到 16 位代码段的转移就是任务内无特权级变换转移的例子。

下面再给出一个用于演示任务内无特权级变换转移的实例。该实例使用了段间转移指令 JMP、段间调用指令 CALL 和段间返回 RET 指令实现同一任务内相同特权级转移。该实例还建立并使用了局部描述符表 LDT。

1. 实现步骤和源程序

实现步骤如下：(1)实方式下初始化，包括对 GDT 和演示任务 LDT 的初始化，装载 GDTR；(2)从实方式切换到保护方式，处于 0 特权级；(3)装载 LDTR，并设置堆栈；(4)利用段间转移指令 JMP 实现从代码段 K 到同级代码段 L 的转移；(5)利用段间 CALL 指令调用同级代码段 C 中的子程序 D 显示字符串信息；(6)利用段间 CALL 指令调用同级代码段 C 中的子程序 H 把十六进制数转换成对应的 ASCII 码；(7)再利用段间 CALL 指令调用同级代码段 C 中的子程序 D 显示字符串信息；(8)利用段间转移指令 JMP 实现从代码段 L 到代码段 K 的转移；(9)从保护方式切换到实方式；(10)在实方式下结束。

该实例的逻辑功能是用十六进制数的形式显示代码段 L 的段界限值。源程序如下：

```
;程序名：T10-3.ASM
;演示任务内无特权级变换的转移
;
        INCLUDE     386SCD.ASM          ;文件 386SCD.ASM 含有关结构、
                                        ;宏指令和符号常量的定义
        .386P
;------------------------------------------------------------
;全局描述符表
GDTSEG      SEGMENT     PARA    USE16   'GDT'
GDT         LABEL   BYTE
DUMMY       DESCRIPTOR      <>          ;空描述符
;规范数据段描述符
NORMAL      DESCRIPTOR      <0FFFFH,0,0,ATDW,0>
NORMAL_SEL  =   NORMAL_GDT
;代码段 K 的描述符
CODEK       DESCRIPTOR      <0FFFFH,,,ATCE,>
CODEK_SEL   =   CODEK_GDT
;局部描述符表段的描述符
LDTABLE     DESCRIPTOR      <LDTLEN_1,,,ATLDT,>
LDT_SEL     =   LDTABLE_GDT
GDTLEN      =   $
GDTSEG      ENDS
;------------------------------------------------------------
;演示任务局部描述符表
LDTSEG      SEGMENT     PARA    USE16   'LDT'
LDT         LABEL   BYTE
;代码段 L 描述符
CODEL       DESCRIPTOR      <CODELLEN_1,CODELSEG,,ATCE,>
```

```
CODEL_SEL      =  (CODEL_LDT) + TIL
;代码段 C 描述符
CODEC       DESCRIPTOR     <CODECLEN_1,CODECSEG,,ATCE,>
CODEC_SEL     =   (CODEC_LDT) + TIL
;显示缓冲区段描述符
VIDEOBUFF DESCRIPTOR     <0FFFFH,0,0,0F00H+ATDW,0>
VIDEO_SEL     =   (VIDEOBUFF_LDT) +TIL
;演示任务 LDT 别名段描述符(DPL=3)
TOLDT       DESCRIPTOR     <LDTLEN_1,LDTSEG,,ATDR+DPL3,>
TOLDT_SEL     =   (TOLDT_LDT) + TIL
;显示信息缓冲区段描述符(DPL=3)
MDATA       DESCRIPTOR     <MDATALEN_1,MDATASEG,,ATDW+DPL3,>
MDATA_SEL     =   (MDATA_LDT) + TIL + RPL3
;堆栈段描述符
STACKS      DESCRIPTOR     <TOPOFS_1,STACKSEG,,ATDWA,>
STACK_SEL =   (STACKS_LDT) + TIL
LDNUM       =   ($_LDT)/(SIZE DESCRIPTOR)     ;LDT 含描述符个数
LDTLEN      =   $_LDT                         ;LDT 字节长
LDTSEG      ENDS
;-----------------------------------------------------------
;显示信息缓冲区段
MDATASEG SEGMENT   PARA   USE16   'MDATA'
MESSAGE   DB  'Value=',0
BUFFER    DB  80  DUP (0)
MDATALEN =   $
MDATASEG ENDS
;-----------------------------------------------------------
;演示任务堆栈段
STACKSEG  SEGMENT   PARA   USE16   'STACK'
          DW   512   DUP (0)
TOPOFS     =   $
STACKSEG  ENDS
;-----------------------------------------------------------
;演示任务代码段 C(含子程序 D 和子程序 H)
CODECSEG  SEGMENT   PARA   USE16   'CODEC'
        ASSUME   CS:CODECSEG
;显示信息子程序 D
;入口参数:FS:SI 指向要显示字符串,字符串以 0 结束
;          ES:EDI 指向显示缓冲区
DISPMESS  PROC      FAR
          MOV      AH,7
DISP1:    MOV      AL,FS:[SI]
```

```
        INC     SI
        OR      AL,AL
        JZ      DISP2
        MOV     ES:[EDI],AX
        INC     EDI
        INC     EDI
        JMP     DISP1
DISP2:  RET
DISPMESS    ENDP
;
;子程序 H,把一位十六进制数转换成对应字符的 ASCII 码
HTOASC    PROC    FAR
        AND     AL,0FH
        ADD     AL,90H
        DAA
        ADC     AL,40H
        DAA
        RET
HTOASC    ENDP
CODECLEN  =  $
CODECSEG  ENDS
;----------------------------------------------------------------
;演示任务代码段 L
CODELSEG  SEGMENT  PARA  USE16  'CODEL'
        ASSUME      CS:CODELSEG
VIRTUAL2:
        MOV     AX,VIDEO_SEL
        MOV     ES,AX                   ;设置显示缓冲区指针
        MOV     EDI,0B8000H
        ;
        MOV     AX,MDATA_SEL
        MOV     FS,AX                   ;设置提示信息缓冲区指针
        MOV     SI,OFFSET   MESSAGE
        ;
        CALL16  CODEC_SEL,DISPMESS      ;显示提示信息
        ;
        MOV     AX,TOLDT_SEL            ;把演示任务的 LDT 的
        MOV     GS,AX                   ;别名段的描述符选择子装入 GS
        ;
        MOV     DX,GS:CODEL.LIMIT       ;取代码段 L 的段界限值
        MOV     SI,OFFSET   BUFFER      ;并转成对应可显示字符串
        MOV     CX,4
```

```
VIR:    ROL     DX,4
        MOV     AL,DL
        CALL16  CODEC_SEL,HTOASC     ;转换出 ASCII 码
        MOV     FS:[SI],AL
        INC     SI
        LOOP    VIR
        MOV     WORD  PTR  FS:[SI],'H'
        ;
        MOV     SI,OFFSET  BUFFER
        CALL16  CODEC_SEL,DISPMESS  ;显示转换出的字符串
        ;
        JUMP16  CODEK_SEL,VIRTUAL3  ;跳转到代码段 K
CODELLEN    =   $
CODELSEG    ENDS
;------------------------------------------------------------
;演示任务代码段 K
CODEKSEG    SEGMENT   PARA   USE16   'CODEK'
        ASSUME  CS:CODEKSEG
VIRTUAL1:
        MOV     AX,LDT_SEL
        LLDT    AX                   ;装载局部描述符表寄存器 LDTR
        ;
        MOV     AX,STACK_SEL
        MOV     SS,AX                ;建立演示任务堆栈
        MOV     SP,OFFSET TOPOFS
        ;
        JUMP16  CODEL_SEL,VIRTUAL2  ;跳转到代码段 L
VIRTUAL3:
        MOV     AX,NORMAL_SEL        ;准备返回实方式
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     SS,AX
        ;
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH
        MOV     CR0,EAX              ;返回实方式
        JUMP16  <SEG  REAL>,<OFFSET  REAL>
CODEKLEN    =   $
CODEKSEG    ENDS
;============================================================
;实方式数据段
```

```
RDATASEG  SEGMENT   PARA   USE 16
VGDTR         PDESC   <GDTLEN-1,>          ;GDT 伪描述符
SPVAR         DW        ?                    ;保存实方式下堆栈指针
SSVAR         DW        ?
RDATASEG   ENDS
;------------------------------------------------------------
;实方式代码段
RCODESEG    SEGMENT   PARA   USE16
        ASSUME   CS:RCODESEG
START:
        ASSUME   DS:GDTSEG
        MOV      AX,GDTSEG
        MOV      DS,AX
        ;初始化全局描述符表
        MOV      BX,16
        MOV      AX,CODEKSEG
        MUL      BX
        MOV      CODEK.BASEL,AX          ;设置代码段 K 基地址
        MOV      CODEK.BASEM,DL
        MOV      CODEK.BASEH,DH
        MOV      AX,LDTSEG               ;设置演示任务 LDT 基地址
        MUL      BX
        MOV      LDTABLE.BASEL,AX
        MOV      LDTABLE.BASEM,DL
        MOV      LDTABLE.BASEH,DH
        ;设置 GDT 伪描述符
        ASSUME     DS:RDATASEG
        MOV      AX,RDATASEG
        MOV      DS,AX
        MOV      AX,GDTSEG
        MUL      BX
        MOV      WORD   PTR   VGDTR.BASE,AX
        MOV      WORD   PTR   VGDTR.BASE+2,DX
        ;初始化演示任务 LDT
        CLD
        CALL     INIT_MLDT
        ;保存实方式堆栈指针
        MOV      SSVAR,SS
        MOV      SPVAR,SP
        ;装载 GDTR
        LGDT     QWORD   PTR   VGDTR
        CLI
```

```
        ;切换到保护方式
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX                     ;置 PE=1
        JUMP16  <CODEK_SEL>,<OFFSET VIRTUAL1>
REAL: ;又回到实方式
        LSS     SP,DWORD  PTR  SPVAR        ;恢复实方式堆栈指针
        STI
        MOV     AX,4C00H                    ;结束
        INT     21H
;------------------------------------------------------------
;初始化演示任务 LDT 的子程序
INIT_MLDT  PROC
        PUSH    DS
        MOV     AX,LDTSEG
        MOV     DS,AX
        MOV     CX,LDNUM
        MOV     SI,OFFSET  LDT
INITL:  MOV     AX,[SI].BASEL
        MOVZX   EAX,AX
        SHL     EAX,4
        SHLD    EDX,EAX,16
        MOV     [SI].BASEL,AX
        MOV     [SI].BASEM,DL
        MOV     [SI].BASEH,DH
        ADD     SI,SIZE DESCRIPTOR
        LOOP    INITL
        POP     DS
        RET
INIT_MLDT  ENDP
RCODESEG   ENDS
           END    START
```

2. 被包含文件 386SCD. ASM

为了节省篇幅,把有关结构类型的定义、宏指令的定义和符号常量的定义等语句集中在一个文件中,以便供每个实例程序使用。

```
;文件名: 386SCD.ASM
;内  容: 符号常量等的定义
;------------------------------------------------------------
;存储段描述符/系统段描述符结构类型的定义
DESCRIPTOR  STRUC
LIMITL      DW     0        ;段界限(0~15)
```

```
BASEL           DW      0       ;段基地址(0～15)
BASEM           DB      0       ;段基地址(16～23)
ATTRIBUTES      DW      0       ;段属性
BASEH           DB      0       ;段基地址(24～31)
DESCRIPTOR      ENDS
;-----------------------------------------------------------------
;门描述符结构类型的定义
GATE            STRUC           ;门结构类型定义
OFFSETL         DW      0       ;32 位偏移的低 16 位
SELECTOR        DW      0       ;选择子
DCOUNT          DB      0       ;双字计数字段
GTYPE           DB      0       ;类型
OFFSETH         DW      0       ;32 位偏移的高 16 位
GATE            ENDS
;-----------------------------------------------------------------
;伪描述符结构类型的定义
PDESC           STRUC
LIMIT           DW      0       ;16 界限
BASE            DD      0       ;基地址
PDESC           ENDS
;-----------------------------------------------------------------
;任务状态段 TSS 结构类型的定义
TASKSS          STRUC
TRLINK          DW      ?,0     ;链接字
TRESP0          DD      ?       ;0 级堆栈指针
TRSS0           DW      ?,0
TRESP1          DD      ?       ;1 级堆栈指针
TRSS1           DW      ?,0
TRESP2          DD      ?       ;2 级堆栈指针
TRSS2           DW      ?,0
TRCR3           DD      ?       ;CR3
TREIP           DD      ?       ;EIP
TREFLAG         DW      ?,?     ;EFLAGS
TREAX           DD      ?       ;EAX
TRECX           DD      ?       ;ECX
TREDX           DD      ?       ;EDX
TREBX           DD      ?       ;EBX
TRESP           DD      ?       ;ESP
TREBP           DD      ?       ;EBP
TRESI           DD      ?       ;ESI
TREDI           DD      ?       ;EDI
TRES            DW      ?,0     ;ES
```

```
TRCS            DW      ?,0     ;CS
TRSS            DW      ?,0     ;SS
TRDS            DW      ?,0     ;DS
TRFS            DW      ?,0     ;FS
TRGS            DW      ?,0     ;GS
TRLDT           DW      ?,0     ;LDT
TRFLAG          DW      0       ;TSS的特别属性字
TRIOMAP         DW      $+2     ;指向I/O许可位图区的指针
TASKSS          ENDS
;------------------------------------------------------------
;存储段描述符类型值说明
ATDR            =       90H     ;存在的只读数据段类型值
ATDW            =       92H     ;存在的可读写数据段类型值
ATDWA           =       93H     ;存在的已访问可读写数据段类型值
ATCE            =       98H     ;存在的只执行代码段类型值
ATCER           =       9AH     ;存在的可执行可读代码段类型值
ATCCO           =       9CH     ;存在的只执行一致代码段类型值
ATCCOR          =       9EH     ;存在的可执行可读一致代码段类型值
;系统段描述符和门描述符类型值说明
ATLDT           =       82H     ;局部描述符表段类型值
ATTASKGAT       =       85H     ;任务门类型值
AT386TSS        =       89H     ;386TSS类型值
AT386CGAT       =       8CH     ;386调用门类型值
AT386IGAT       =       8EH     ;386中断门类型值
AT386TGAT       =       8FH     ;386陷阱门类型值
;------------------------------------------------------------
;DPL和RPL值说明
DPL1            =       20H     ;DPL=1
DPL2            =       40H     ;DPL=2
DPL3            =       60H     ;DPL=3
RPL1            =       01H     ;RPL=1
RPL2            =       02H     ;RPL=2
RPL3            =       03H     ;RPL=3
IOPL1           =       1000H   ;IOPL=1
IOPL2           =       2000H   ;IOPL=2
IOPL3           =       3000H   ;IOPL=3
;------------------------------------------------------------
;其他常量值说明
D32             =       4000H   ;32位代码段标志
TIL             =       04H     ;TI=1(描述符表标志)
VMFL            =       0002H   ;VMF=1
IFL             =       0200H   ;IF=1
```

```
;---------------------------------------------------------------
;32位偏移的段间转移宏指令
JUMP32          MACRO selector,offsetv
                DB      0EAH     ;操作码
                DW      offsetv  ;32位偏移
                DW      0
                DW      selector ;选择子
                ENDM
;---------------------------------------------------------------
;32位偏移的段间调用宏指令
CALL32          MACRO selector,offsetv
                DB      09AH     ;操作码
                DW      offsetv  ;32位偏移
                DW      0
                DW      selector ;选择子
                ENDM
;---------------------------------------------------------------
;16位偏移的段间转移宏指令
JUMP16          MACRO selector,offsetv
                DB      0EAH     ;操作码
                DW      offsetv  ;16位偏移
                DW      selector ;段值/选择子
                ENDM
;---------------------------------------------------------------
;16位偏移的段间调用宏指令
CALL16          MACRO selector,offsetv
                DB      9AH      ;操作码
                DW      offsetv  ;16位偏移
                DW      selector ;段值/选择子
                ENDM
```

3. 关于实例三的说明

有些步骤的实现方法已在前面的实例中做过介绍，下面就任务内无特权级变换的转移和使用局部描述符LDT等作些说明：

(1) 实方式下初始化LDT

演示任务使用了局部描述符表LDT，该LDT在实方式下初始化。为了简便，LDT中各描述的界限和属性值在定义时预置，利用一个子程序设置各段的段基地址。为了方便，在定义时把各段的段值安排在相应描述符的段基地址低16位字段中。由于实例中各段在实方式下定位，所以把段值乘16就是对应的段基地址。

(2) 装载LDTR寄存器

在使用LDT之前，还要装载局部描述符表寄存器LDTR。本实例中的如下两条指令用于装载LDTR：

```
MOV     AX,LDT_SEL
LLDT    AX
```

LLDT 指令是专门用于装载 LDTR 的指令。该指令的操作数是对应 LDT 段描述符的选择子。根据该选择子，处理器从 GDT 中取出相应的 LDT 段描述符，在进行合法性等检查后，LDT 段描述符的基地址和界限等信息被装入 LDTR 的高速缓冲寄存器中。由于要引用 GDT，所以不能在实方式下装载 LDTR。在 10.8 节中对 LLDT 指令作详细说明。

(3) 利用段间转移指令 JMP 实现任务内无特权级变换的转移

在本实例中进入保护方式后，特权级是 0。通过如下段间直接转移指令实现从代码段 K 到代码段 L 的转移：

```
JUMP16    CODEL_SEL,VIRTUAL2
```

其中，选择子 CODEL_SEL 是对应代码段 L 的描述符的选择子。该描述符在 LDT 中，所以选择子中的描述符表指示位 TI 是 1。描述符特权级是 0，表示对应代码段的特权级是 0，选择子中的请求特权级 RPL 也是 0。目标代码段不是一致代码段，所以在 CPL＝DPL，RPL＜＝DPL 的情况下，顺利进行相同特权级的转移：目标代码段的选择子 CODEL_SEL 被装入 CS，对应描述符中的信息被装入高速缓冲寄存器，偏移 VIRTUAL2 被装入指令指针寄存器。由于是 16 位代码段，所以偏移用 16 位表示。

类似地，通过如下段间直接转移指令实现从代码段 L 转移到代码段 K：

```
JUMP16    CODEK_SEL,VIRTUAL3
```

其中，选择子 CODEK_SEL 是对应代码段 K 的描述符的选择子。由于描述符在 GDT 中，所以选择子中的 TI 位是 0。

(4) 利用段间调用指令 CALL 实现任务内无特权级变换的转移

在代码段 L 中，通过段间直接调用指令 CALL 调用了在代码段 C 中的两个子程序，这些调用都是无特权级变换的转移。

例如：利用如下指令调用了显示字符串子程序 DISPMESS：

```
CALL16    CODEC_SEL,DISPMESS
```

其中，CODEC_SEL 是代码段 C 的选择子，DISPMESS 表示子程序入口。描述代码段 C 的描述符在 LDT 中，描述符特权级 DPL 是 0，所以使用的选择子 CODE_SEL 的请求特权级 RPL 是 0，描述符表指示位 TI 是 1。目标代码段 C 不是一致代码段，所以在 CPL＝DPL，RPL＜＝DPL 的情况下，顺利进行相同特权级的转移：当前 CS 和 IP 压入堆栈，目标代码段的选择子 CODEC_SEL 被装入 CS，对应描述符中的信息被装入高速缓冲寄存器，16 位偏移 DISPMESS 被装入指令指针。由于是 16 位段，所以偏移用 16 位表示，压入堆栈的是字而不是双字。

(5) 段间返回指令 RET 实现任务内无特权级变换的转移

段间返回指令 RET 从堆栈弹出返回地址(由选择子和偏移构成)。弹出选择子内的 RPL＝CPL，并且对应 DPL＝CPL，RPL＜＝DPL 是当然的，所以能顺利进行相同特权级转移。

4. 别名技术

在上述实例三中，使用了两个描述符来描述演示任务的 LDT 段。描述符 LDTABLE 被安排在 GDT 中，它是系统段描述，把段 LDTSEG 描述成演示任务的局部描述符表 LDT。描述符 TOLDT 被安排在 LDT 中，它是数据段描述符，把段 LDTSEG 描述成一个普通的数据段。描述符 LDTABLE 被装载到 LDTR，描述符 TOLDT 被装载到某个数据段寄存器。为什么要这样处理呢？根据实例三的功能要求，需要访问演示任务的局部描述符表 LDT 段，以取得代码段 L 的段界限值，这需要通过某个段寄存器进行，但不能把系统段描述选择子装载到段寄存器，所以采用两个描述符来描述段 LDTSEG。

这种为了满足对同一个段实施不同方式操作的需要，而用多个描述符加以描述的技术称为别名技术。这好比一个演员在一部戏中扮演多个角色，在不同的情景下，使用不同的称呼。在保护方式程序设计中，常常要采用别名技术。例如：用两个具有不同类型值的描述符来描述同一个段。再如，用两个具有不同 DPL 的描述符来描述符同一个段。

10.6.3 任务内不同特权级的变换

在一个任务之内，可以存在四种特权级，所以常常会发生不同特权级之间的变换。例如：外层的应用程序调用内层操作系统的例程，以获得必要的诸如存储器分配等系统服务；内层操作系统的例程完成后，返回到外层应用程序。

在同一任务内，实现特权级从外层到内层变换的普通途径是：使用段间调用指令 CALL，通过调用门进行转移；实现特权级从内层到外层变换的普通途径是：使用段间返回指令 RET。注意，不能利用 JMP 指令实现任务内不同特权级的变换。

1. 通过调用门的转移

当段间转移指令 JMP 和段间调用指令 CALL 所含指针的选择子指示调用门描述符时，就可实现通过调用门的转移。

调用门描述调用转移的入口点，从图 10.13 可见，它包含目标地址的段及偏移量的 48 位全指针。在执行通过调用门的段间转移指令 JMP 或段间调用指令 CALL 时，指令所含指针内的选择子用于确定调用门，而偏移被丢弃；把调用门内的 48 位全指针，作为目标地址指针进行转移。

处理器采用与访问数据段相同的特权级规则控制对门描述符的访问。门描述符的 DPL 规定了访问门的最外层特权级，只有在相同级或者更内层级的程序才可以访问门。同时，还要求指示门的选择子的 RPL 必须满足 RPL<=DPL 的条件。

调用门是门描述符的一种。所以，在取出调用门内的 48 位全指针，把它作为目标地址指针向目标代码段转移之前，要进行特权级检测。必须符合 CPL<=DPL 并且 RPL<=DPL 的条件。检测通过后，开始在 10.6.1 节中所述的转移步骤。其中还要检测目标描述符是否是描述符代码段，调用门内的选择子指示的描述符必须是代码段描述符。此外，在装载高速缓冲寄存器之前调整 RPL=0，也即调用门中选择子的 RPL 被忽略。

如 10.6.1 节中所述，在装载 CS 高速缓冲寄存器时，还要对目标代码段描述符进行保护检测。检测过程中的 DPL 不再是调用门的 DPL，而是调用门内选择子所指示的目标代码段描述符的 DPL。段间调用指令 CALL 和段间转移指令 JMP 所做的检测不一样。

对于使用调用门的段间转移指令 JMP,检测条件与段间直接转移相同。由于已置 RPL=0,所以可认为 RPL<=DPL 的条件总能满足。所以,对于普通的非一致代码段,当 CPL=DPL 时,发生无特权级变换的转移;对于一致代码段,在满足 CPL>=DPL 时也发生无特权级变换的转移;其他情形就引起异常。

对于使用调用门的段间调用指令 CALL,情形就不同了。由于已置 RPL=0,所以可认为 RPL<=DPL 的条件总能满足。对于一致代码段,在满足 CPL>=DPL 时发生无特权级变换的转移。对于非一致代码段,当 CPL=DPL 时,仍发生无特权级变换的转移;当 CPL>DPL 时,就发生向内层特权级变换的转移,使 CPL 保持等于 DPL,同时切换到对应的内层堆栈。

综上所述,使用段间调用指令 CALL,通过调用门可以实现从外层程序调用进入内层程序;通过调用门也可实现无特权级变换的转移。

当然,CALL 指令在最后把目标代码段的指针装入 CS 和 EIP 之前,要把原 CS 和 EIP,即返回地址保存到堆栈。如无特权级变换,堆栈保持不变,返回地址就保存在原堆栈中;如变换特权级,那么返回地址保存在内层堆栈中。

2. 堆栈切换

在使用 CALL 指令,通过调用门向内层转移时,不仅特权级发生变换,控制转移到一个新的代码段,而且也切换到内层的堆栈段。从图 10.14 所示的任务状态段 TSS 的格式可见,TSS 中包含有指向 0 级、1 级和 2 级堆栈的指针。在特权级发生向内层变换时,根据特权级使用 TSS 中相应的堆栈指针对 SS 及 ESP 寄存器进行初始化,建立起一个空栈。

在建立起内层堆栈后,先把外层堆栈的指针 SS 及 ESP 寄存器的值压入内层堆栈,以使得相应的向外层返回可恢复原来的外层堆栈。然后,从外层堆栈复制以双字为单位的调用参数到内层堆栈,调用门中的 DCOUNT 字段值决定了复制参数的量。这些被复制的参数是主程序通过堆栈传递给子程序的实参,在调用之前被压入外层堆栈。通过复制堆栈中的参数,使内层的子程序不需要考虑堆栈的切换,而容易地访问主程序传递过来的实参。最后,调用的返回地址被压入堆栈,以便在调用结束时返回。图 10.15 给出了在向内层变

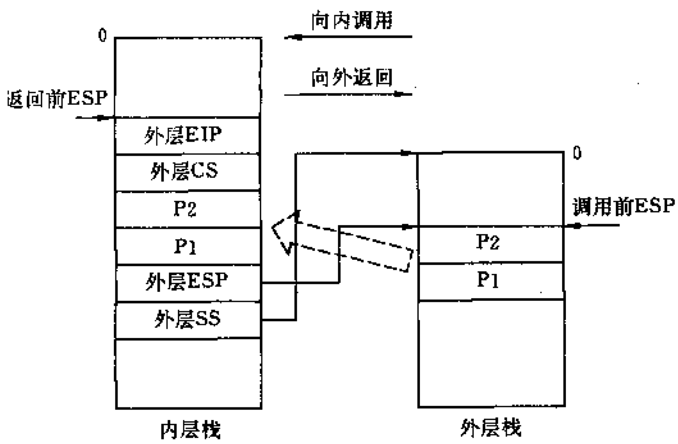


图 10.15 向内层变换时堆栈切换示意图

换时，建立内层堆栈，并从外层堆栈复制 2 个双字参数到内层堆栈的示意图。图中每项是双字，可见的段寄存器内的选择子被扩展成 32 位，高 16 位为 0。

无论是否通过调用门，只要不发生特权级变换，就不会切换堆栈。

3. 向外层返回

与使用 CALL 指令通过调用门向内层变换相反，使用 RET 指令实现向外层返回。段间返回指令 RET 从堆栈中弹出返回地址，并且可以采用调整 ESP 的方法，跳过相应的在调用之前压入堆栈的参数。返回地址的选择子指示要返回的代码段描述符，从而确定返回的代码段。选择子的 RPL 确定返回后的特权级，而不是对应描述符的 DPL，这是因为，段间返回指令 RET 可能使控制返回到一致代码段，而一致代码段可以在 DPL 规定的特权级以外的特权级执行。

RET 指令先从堆栈弹出返回地址。如果返回地址的选择子的 RPL 规定相对于 CPL 更外层的级，那么就引起向外层返回。其次，为向外层返回，跳过内层堆栈中的参数，再从内层堆栈中弹出指向外层堆栈的指针，并装入到 SS 及 ESP，以恢复外层堆栈。再次，调整 ESP，跳过在相应的调用之前压入到外层堆栈的参数。然后，检查数据段寄存器 DS、ES、FS 及 GS，以保证寻址的段在外层是可访问的，如果段寄存器寻址的段在外层是不可访问的，那么装入一个空选择子，以避免在返回时发生保护空洞。最后，返回(外层)继续执行。上述五步是对带立即数段间返回指令而言的，立即数规定了堆栈中要跳过的参数的字节数。对无立即数段间返回指令而言，缺少第二和第三步，请参见下面的实例四。如果 RET 指令不需要向外层返回，那么就只有开始和最后的两步。

10.6.4 演示任务内特权级变换的实例(实例四)

下面给出一个演示任务内特权级变换的实例。该实例演示在任务内通过调用门从外层特权级变换到内层特权级；也演示通过段间返回指令从内层特权级变换到外层特权级；还演示通过调用门的无特权级变换的转移。实例使用了任务状态段 TSS，这是因为任务内特权级变换时要使用的内层堆栈指针存放在 TSS 中。

1. 实现流程

实例四的主要实现步骤如图 10.16 所示。在图的右边标出了特权级变换的分界情况。由于在任务内发生特权级变换时要切换堆栈，而内层堆栈的指针存放在当前任务的 TSS 中，所以在进入保护方式后设置任务状态段寄存器 TR。由于演示任务使用了局部描述符表，所以设置 LDTR。从实方式切换到保护方式下的 16 位临时代码段，CPL＝0。在临时代码段通过任务门转移到 32 位过渡代码段，不发生特权级变换，CPL＝0。为了演示外层程序通过调用门调用内层程序，要使 CPL＞0。实例先通过段间返回指令 RET 从特权级 0 变换到特权级 3 的演示代码段。在特权级 3 下，通过调用门调用 1 级的子程序。随着执行段间 RET，又返回到 3 级的演示代码段。在 3 级演示代码段通过调用门转移到 0 级的过渡代码段，再转 0 级的临时代码段，最后切换回实方式。

2. 源程序组织和清单

实例四有如下部分组成：

(1) 全局描述符表 GDT。GDT 含有演示任务的 TSS 段描述符和 LDT 段描述符，此

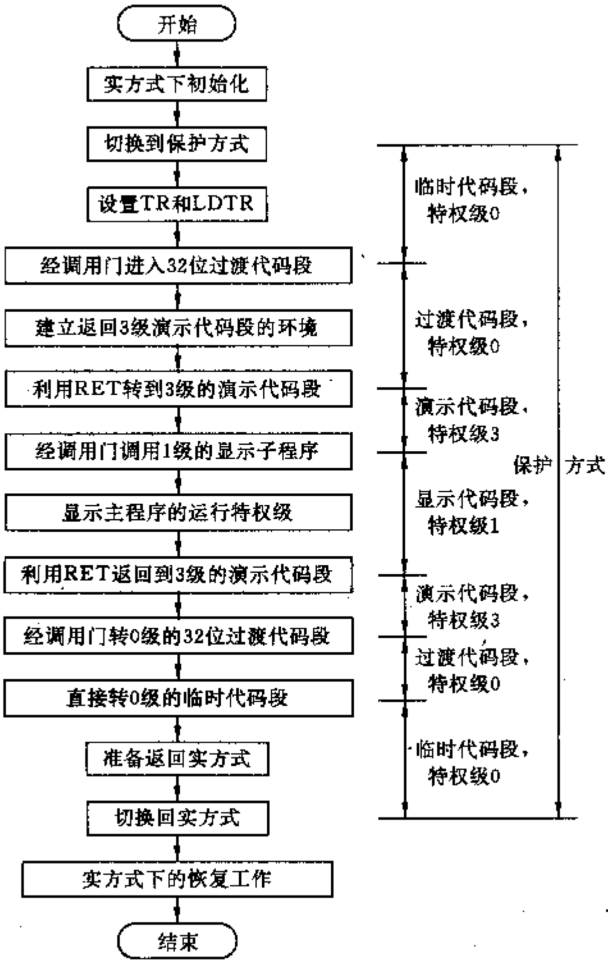


图 10.16　实例四的实现流程

外还含临时代码段描述符、规范数据段描述符和视频缓冲区段描述符。

(2) 演示任务的 LDT 段。它含有除临时代码段外的其他代码段的描述符和演示任务各级堆栈段描述符，还含有 3 个调用门。

(3) 演示任务的 TSS 段。

(4) 演示任务的 0 级、1 级和 3 级堆栈段。

(5) 显示子程序段。32 位代码段，特权级 1。

(6) 演示代码段。32 位代码段，特权级 3。

(7) 过渡代码段。32 位段，特权级 0。

(8) 临时代码段。16 位段，特权级 0。

(9) 实方式下的数据和代码段。

该实例的逻辑功能是显示演示程序代码段执行时的当前特权级 CPL。源程序清单如下：

```
;程序名:T10-4.ASM
;演示在任务内如何进行特权级变换
;
        INCLUDE     386SCD.ASM      ;参见实例三
;
        .386P
;---------------------------------------------------------------
;全局描述符表GDT
GDTSEGSEGMENT   PARA    USE16
GDT         LABEL   BYTE
DUMMY       DESCRIPTOR      <>
NORMAL      DESCRIPTOR      <0FFFFH,0,0,ATDW,0>
NORMAL_SEL  =       NORMAL-GDT
EFFGDT      LABEL   BYTE
;演示任务状态段TSS描述符
DEMOTSS     DESCRIPTOR  <DemoTSSLEN-1,DemoTSSSEG,,AT386TSS,>
DemoTSS_SEL =       DEMOTSS-GDT
;演示任务LDT段描述符
DEMOLDTD DESCRIPTOR     <DemoLDTLEN-1,DemoLDTSEG,,ATLDT,>
DemoLDT_SEL =       DEMOLDTD-GDT
;临时代码段描述符
TEMPCODE DESCRIPTOR     <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL =      TEMPCODE-GDT
;视频缓冲区段描述符(DPL=3)
VIDEOBUFF DESCRIPTOR    <0FFFFH,0,0,0F00H+ATDW+DPL3,0>
Video_SEL   =       VIDEOBUFF-GDT
GDNUM       =   ($-EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN      =   $-GDT
GDTSEG  ENDS
;---------------------------------------------------------------
;演示任务局部描述符表LDT
DemoLDTSEG  SEGMENT     PARA    USE16
DemoLDT     LABEL       BYTE
;0级堆栈段描述符(32位段)
DEMOSTACK0  DESCRIPTOR      <DemoStack0LEN-1,DemoStack0SEG
                            ,,ATDW+D32,>
DemoStack0_SEL    =   (DemoStack0-DEMOLDT) + TIL
;1级堆栈段描述符(DPL=1)
DEMOSTACK1  DESCRIPTOR      <DemoStack1LEN-1,DemoStack1SEG
                            ,,ATDW+D32+DPL1,>
DemoStack1_SEL    =   (DemoStack1-DEMOLDT) + TIL + RPL1
;3级堆栈段描述符(DPL=3)
```

```
DEMOSTACK3  DESCRIPTOR      <DemoStack3LEN-1,DemoStack3SEG,,
                             ATDW+DPL3,>
DemoStack3_SEL    =   (DemoStack3-DEMOLDT) + TIL + RPL3
;演示代码段描述符(32 位段,DPL=3)
DEMOCODE    DESCRIPTOR      <DemoCodeLEN-1,DemoCodeSEG,,
                             ATCE+D32+DPL3,>
DemoCode_SEL      =   (DEMOCODE-DEMOLDT) + TIL + RPL3
;过渡代码段描述符(32 位段)
T32CODE     DESCRIPTOR      <T32CodeLEN-1,T32CodeSEG,,ATCE+D32,>
T32Code_SEL     =   (T32CODE-DEMOLDT) + TIL
;显示子程序代码段描述符(32 位段,DPL=1)
ECHOSUBR    DESCRIPTOR      <EchoSUBRLEN-1,EchoSUBRSEG,,
                             ATCER+D32+DPL1,>
Echo_SEL1     =   (ECHOSUBR-DEMOLDT) + TIL + RPL1
Echo_SEL3     =   (ECHOSUBR-DEMOLDT) + TIL + RPL3
DemoLDNUM     =   ($ -DEMOLDT)/(SIZE DESCRIPTOR)
;指向过渡代码段内 T32Begin 点的调用门(DPL=0)
TOT32GATEA  GATE  <T32Begin,T32Code-SEL,0,AT386CGAT,0>
ToT32A_SEL    =   (TOT32GATEA-DemoLDT) + TIL
;指向过渡代码段内 T32End 点的调用门(DPL=3)
TOT32GATEB  GATE  <T32End,T32Code-SEL,0,AT386CGAT+DPL3,0>
ToT32B_SEL    =   (TOT32GATEB-DemoLDT) + TIL
;指向显示子程序的调用门(DPL=3)
TOECHOGATE  GATE  <EchoSUB,Echo-SEL3,0,AT386CGAT+DPL3,0>
ToEcho_SEL    =   (TOECHOGATE-DemoLDT) + TIL
DemoLDTLEN    =   $ -DemoLDT
DemoLDTSEG    ENDS
;-------------------------------------------------------------
;演示任务的 TSS 段
DemoTSSSEG  SEGMENT   PARA   USE 16
        DD      0                   ;BACK
        DW      DemoStack0LEN,0     ;0 级堆栈指针
        DW      DemoStack0_SEL,0    ;初始化
        DW      DemoStack1LEN,0     ;1 级堆栈指针
        DW      DemoStack1_SEL,0    ;初始化
        DD      ?                   ;2 级堆栈指针
        DW      ?,0                 ;未初始化
        DD      0                   ;CR3
        DD      ?                   ;EIP
        DD      ?                   ;EFLAGS
        DD      ?                   ;EAX
        DD      ?                   ;ECX
```

```
        DD      ?                   ;EDX
        DD      ?                   ;EBX
        DD      ?                   ;ESP
        DD      ?                   ;EBP
        DD      ?                   ;ESI
        DD      ?                   ;EDI
        DW      ?,0                 ;ES
        DW      ?,0                 ;CS
        DW      ?,0                 ;SS
        DW      ?,0                 ;DS
        DW      ?,0                 ;FS
        DW      ?,0                 ;GS
        DW      DemoLDT_SEL,0       ;LDT
        DW      0
        DW      $+2                 ;指向I/O许可位图
        DB      0FFH                ;I/O许可位图结束标志
DemoTSSLEN=$
DemoTSSSEG  ENDS
;---------------------------------------------------------------
;演示任务0级堆栈段(32位段)
DemoStack0SEG   SEGMENT   PARA   USE32
DemoStack0LEN   =   512
        DB      DemoStack0LEN   DUP (0)
DemoStack0SEG   ENDS
;---------------------------------------------------------------
;演示任务1级堆栈段(16位段)
DemoStack1SEG   SEGMENT   PARA   USE32
DemoStack1LEN   =   512
        DB      DemoStack1LEN   DUP (0)
DemoStack1SEG   ENDS
;---------------------------------------------------------------
;演示任务3级堆栈段(16位段)
DemoStack3SEG   SEGMENT   PARA   USE16
DemoStack3LEN   =   512
        DB      DemoStack3LEN   DUP (0)
DemoStack3SEG   ENDS
;---------------------------------------------------------------
;演示任务显示子程序代码段(32位段,1级)
EchoSUBRSEG     SEGMENT   PARA   USE32
MESSAGE  DB     'CPL=',0
        ASSUME  CS:EchoSUBRSEG
;显示调用程序的执行特权级
```

```
EchoSUB      PROC    FAR
        CLD
        PUSH    EBP
        MOV     EBP,ESP
        MOV     AX,Echo_SEL1                ;子程序代码段是可读段
        MOV     DS,AX                       ;采用 RPL=1 的选择子
        MOV     AX,Video_SEL
        MOV     ES,AX                       ;视频缓冲区段基地址 00000000H
        MOV     EDI,0B8000H                 ;视频缓冲区段 B8000H 开始
        MOV     ESI,OFFSET  MESSAGE
        MOV     AH,17H                      ;置显示属性
EchoSUB1:
        LODSB
        OR      AL,AL
        JZ      EchoSUB2
        STOSW                               ;显示字符串
        JMP     EchoSUB1
EchoSUB2:
        MOV     EAX,[EBP+8]                 ;从堆栈中取调用程序的选择子
        AND     AL,3                        ;调用程序的 CPL 在 CS 的 RPL 字段
        ADD     AL,'0'
        MOV     AH,17H
        STOSW                               ;显示之
        POP     EBP
        RETF                                ;返回
EchoSUB     ENDP
EchoSUBRLEN    =    $
EchoSUBRSEG    ENDS
;-------------------------------------------------------------
;演示任务的演示代码段(32 位段,3 级)
DemoCodeSEG    SEGMENT   PARA   USE32
        ASSUME   CS:DemoCodeSEG
DemoBegin:
;显示当前特权级(变换到 1 级)
        CALL32  ToEcho_SEL,0
         ;转到过渡代码段(变换到 0 级)
        CALL32  ToT32B_SEL,0
DemoCodeLEN    =    $
DemoCodeSEG    ENDS
;-------------------------------------------------------------
;演示任务的过渡代码段(32 位段,0 级)
T32CODESEG     SEGMENT   PARA   USE32
```

```
        ASSUME    CS : T32CODESEG
T32Begin:
        ;建立 0 级堆栈
        MOV     AX,DemoStack0_SEL
        MOV     SS,AX
        MOV     ESP,DemoStack0LEN
        ;压入 3 级堆栈指针
        PUSH    DWORD  PTR  DemoStack3_SEL
        PUSH    DWORD  PTR  DemoStack3LEN
        ;压入入口点
        PUSH    DWORD  PTR  DemoCode_SEL
        PUSH    OFFSET  DemoBegin
        ;利用 RET 实现转 3 级的演示代码段
        RETF
T32End: ;转临时代码段
        JUMP32      TempCode_SEL,<OFFSET ToReal>
T32CodeLEN      =   $
T32CODESEG      ENDS
;---------------------------------------------------------------
;临时代码段(16 位段,0 级)
TempCodeSEG     SEGMENT   PARA   USE16
        ASSUME   CS : TempCodeSEG
Virtual: ;装载 TR
        MOV     AX,DemoTSS_SEL
        LTR     AX
        ;装载 LDTR
        MOV     BX,DemoLDT_SEL
        LLDT    BX
        ;通过调用门转过渡段
        JUMP16  ToT32A_SEL,0
ToReal: ;准备切换回实方式
        MOV     AX,Normal_SEL
        MOV     DS,AX
        MOV     ES,AX                       ;把规范段描述符
        MOV     FS,AX                       ;装入各数据段寄存器
        MOV     GS,AX
        MOV     SS,AX
        ;
        MOV     EAX,CR0
        AND     AX,0FFFEH
        MOV     CR0,EAX                     ;返回实方式
        JUMP16      <SEG REAL>,<OFFSET REAL>
```

```
TempCodeLEN      = $
TempCodeSEG      ENDS
;==================================
;实方式下的数据段
RDataSEG     SEGMENT   PARA   USE16
VGDTR        PDESC   <GDTLEN-1,>
SPVAR        DW      ?
SSVAR        DW      ?
RDataSEG     ENDS
;----------------------------------------------------------------
;实方式下的代码段
RCodeSEG     SEGMENT   PARA   USE16
        ASSUME    CS:RCodeSEG,DS:RDataSEG
Start:
        MOV     AX,RDataSEG
        MOV     DS,AX
        CLD
        CALL    INIT_GDT                    初始化 GDT
        ;
        MOV     AX,DemoLDTSEG
        MOV     FS,AX
        MOV     SI,OFFSET DemoLDT
        MOV     CX,DemoLDNUM
        CALL    INIT_LDT                    ;初始化 LDT
        ;
        MOV     SSVAR,SS
        MOV     SPVAR,SP
        ;装载 GDTR 和切换到保护方式
        LGDT    QWORD   PTR   VGDTR
        CLI
        MOV     EAX,CR0
        OR      AX,1
        MOV     CR0,EAX
        JUMP16  TempCode_SEL,<OFFSET Virtual>
Real:
        MOV     AX,RDATASEG
        MOV     DS,AX
        LSS     SP,DWORD PTR SPVAR
        STI
        MOV     AX,4C00H
        INT     21H
;----------------------------------------------------------------
```

```
；初始化全局描述符表的子程序
；(1)把定义时预置的段值转换成 32 位段基地址并置入描述符内相应字段
；(2)初始化为 GDTR 准备的伪描述符
INIT_GDT      PROC    NEAR
        PUSH    DS
        MOV     AX,GDTSEG
        MOV     DS,AX
        MOV     CX,GDNUM                     ;GDNUM 是初始化的描述符个数
        MOV     SI,OFFSET EFFGDT             ;EFFGDT 是开始偏移
INITG:  MOV     AX,[SI].BASEL                ;取出预置的段值
        MOVZX   EAX,AX                       ;扩展到 32 位
        SHL     EAX,4
        SHLD    EDX,EAX,16                   ;分解到 2 个 16 位寄存器
        MOV     [SI].BASEL,AX
        MOV     [SI].BASEM,DL                ;置入描述符相应字段
        MOV     [SI].BASEH,DH
        ADD     SI,SIZE DESCRIPTOR           ;调整到下一描述符
        LOOP    INITG
        POP     DS
        ;
        MOV     BX,16                        ;初始化为 GDTR 准备的伪描述符
        MOV     AX,GDTSEG
        MUL     BX
        MOV     WORD  PTR  VGDTR.BASE,AX
        MOV     WORD  PTR  VGDTR.BASE+2,DX
        RET
INIT_GDT     ENDP
;
；初始化演示任务局部描述符表的子程序
；把定义时预置的段值转换成 32 位段基地址并置入描述符内相应字段
；入口参数:FS：SI=第一个要初始化的描述符
；            CX=要初始化的描述符个数
INIT_LDT     PROC
ILDT:   MOV     AX,FS：[SI].BASEL
        MOVZX   EAX,AX
        SHL     EAX,4
        SHLD    EDX,EAX,16
        MOV     FS：[SI].BASEL,AX
        MOV     FS：[SI].BASEM,DL
        MOV     FS：[SI].BASEH,DH
        ADD     SI,SIZE DESCRIPTOR
        LOOP    ILDT
```

```
RET
INIT_LDT     ENDP
RCodeSEG     ENDS
END   Start
```

3. 关于实例四的说明

程序中部分片段的背景和实现方法已在前面的实例中做过介绍，下面主要就如何实现任务内特权级变换作些说明：

(1) 通过段间返回指令实现特权级变换

实例在两处使用段间返回指令实现任务内的特权级变换。一处是在 0 级的过渡代码段中用段间 RET 指令，从特权级 0 变换到特权级 3 的演示代码段。该处 RET 指令并不对应 CALL 指令。实例从实方式切换到保护方式后 CPL＝0。为了演示如何通过调用门调用内层程序，要设法使 CPL＞0。为此，实例先建立一个已发生从外层到内层变换的环境，也即按如图 10.15 所示，在当前堆栈(0 级堆栈)中放入外层堆栈的指针和外层演示程序的入口指针，形成一个如图 10.17 所示的 0 级堆栈，无需传递参数。然后，执行段间返回指令 RET，从堆栈中弹出 3 级演示代码的选择子，RPL＝3，而当时 CPL＝0，所以导致向外层变换特权级，就从 0 级的过渡代码段变换到 3 级的演示代码段，同时切换到 3 级堆栈。

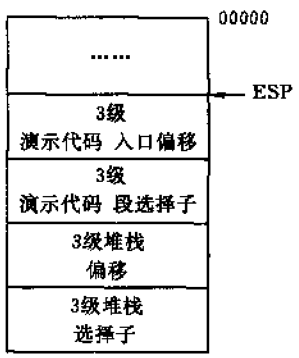


图 10.17 实例中执行 RET 时的 0/1 级堆栈

另一处是从 1 级的显示子程序 EchoSUB 返回到 3 级的演示程序段。这处的 RET 指令与演示程序中使用的通过调用门的段间调用指令 CALL 相对应，执行段间返回指令 RET 时的 1 级堆栈也如图 10.17 所示，其中的返回地址指针和外层堆栈指针由指令 CALL 压入。

(2) 通过调用门实现特权级变换

实例在两处使用了段间调用指令，通过调用门实现特权级的变换。一处是 3 级演示代码通过调用门 TOECHOGATE 调用 1 级的显示子程序。调用门 TOECHOGATE 自身 DPL＝3，只有这样，3 级的演示代码才能够使用该调用门。由于调用门内的选择子 Echo_SEL3 所指示的显示子程序代码段描述符 DPL＝1，而当时 CPL＝3，所以引起从外层特权级向内层特权级的变换，使 CPL＝1。同时形成如图 10.17 所示的 1 级堆栈。虽然调用门内的选择子 Echo_SEL3 的 RPL＝3，大于目标代码段描述符的 DPL，但没有关系，因为在通过调用门转移时，门内指示目标代码段的选择子 RPL 总被当作 0 对待。

另一处是 3 级演示代码还通过调用门 TOT32GATEB 调用了 0 级的过渡代码。该处使用的调用门描述符 DPL 也等于 3。由于调用门内的选择子 T32Code_SEL 所指示的过渡代码段描述符 DPL＝0，而当时 CPL＝3，所以引起从 3 特权级向 0 特权级的变换，使 CPL＝0。同时形成如图 10.17 所示的 0 级堆栈。但该处的调用实际上是“有去无回”的，调用的目的是转移到 0 级的过渡代码，准备返回到实方式。由于从 3 级的演示代码到 0 级的过渡代码要发生特权级变换，所以不能使用转移指令 JMP，必须使用调用指令 CALL。

(3) 通过调用门实现无特权级变换的转移

在临时代码段中,使用调用门 TOT32GATEA 转移到过渡代码段。尽管调用门内的选择子 T32Code_SEL 所指示的过渡代码段描述符 DPL＝0,但当时 CPL＝0,所以不发生特权级变换。正是这个原因,才可以使用段间转移指令 JMP。

(4) 子程序 EchoSUB 的实现

子程序 EchoSUB 的功能是显示调用程序执行时的特权级。调用程序的执行特权级在代码段寄存器 CS 内选择子的 RPL 字段,在调用 EchoSUB 时,CS 寄存器内容被压入堆栈。子程序从堆栈取得调用程序的代码段选择子,再从中分离出 RPL 就可得调用程序的执行特权级。

(5) 装载任务状态段寄存器 TR

在任务内发生特权级变换时堆栈也随着自动切换,外层堆栈指针保存在内层堆栈中,而内层堆栈指针存放在当前任务的 TSS 中。所以,在从外层向内层变换时,要访问 TSS。实例在进入保护方式下的临时代码段后,通过如下两条指令,装载任务状态段寄存器 TR,使其指向已预置好的演示任务的 TSS:

```
MOV     AX,DemoTSS_SEL
LTR     AX
```

LTR 指令是专门用于装载任务状态段寄存器 TR 的指令。该指令的操作数是对应 TSS 段描述符的选择子。LTR 指令从 GDT 中取出相应的 TSS 段描述符,把 TSS 段描述符的基地址和界限等信息被装入 TR 的高速缓冲寄存器中。在 10.8 节中对 LTR 指令的一般格式作说明。

10.6.5 任务切换

利用段间转移指令 JMP 或者段间调用指令 CALL,通过任务门或者直接通过任务状态段,可以切换到别的任务。此外,在中断/异常或者执行 IRET 指令时也可能发生任务切换。

1. 直接通过 TSS 进行任务切换

当段间转移指令 JMP 或段间调用指令 CALL 所含指针的选择子指示一个可用任务状态段 TSS 描述符时,正常情况下就发生从当前任务到由该可用 TSS 对应任务(目标任务)的切换。目标任务的入口点由目标任务 TSS 内的 CS 和 EIP 字段所规定的指针确定。这样的 JMP 或 CALL 指令内的偏移被丢弃。

处理器采用与访问数据段相同的特权级规则控制对 TSS 段描述符的访问。TSS 段描述符的 DPL 规定了访问该描述符的最外层特权级,只有在相同级或者更内层级的程序才可以访问它。同时,还要求指示它的选择子的 RPL 必须满足 RPL＜＝DPL 的条件。当这些条件满足时,就开始任务切换。

2. 通过任务门进行任务切换

任务门内的选择子指示某个任务的 TSS 描述符。当段间转移指令 JMP 或段间调用指令 CALL 所含指针的选择子指示一个任务门时,正常情况下就发生任务切换,也即从

当前任务切换到由任务门内的选择子所指示的 TSS 描述符对应的任务(目标任务)。这样的 JMP 或 CALL 指令内的偏移被丢弃;任务门内的偏移也无意义。

处理器采用与访问数据段相同的特权级规则控制对任务门的访问。任务门的 DPL 规定了访问该门的最外层特权级,只有在相同级或者更内层级的程序才可以访问它。同时,还要求指示任务门的选择子的 RPL 必须满足 RPL<=DPL 的条件。在这些条件满足时,再检查任务门内的选择子,要求该选择子指示 GDT 中的可用 TSS 描述符。在检查通过后,就开始任务切换。

3. 任务切换过程

根据指示目标任务 TSS 描述符的选择子进行任务切换的一般过程如下:

第一,测试目标任务状态段的界限。TSS 用于保存任务的各种状态信息,不同的任务,TSS 中可以有数量不等的其他信息,但根据图 10.14 所示的任务状态段基本格式,TSS 的界限应大于或等于 103。

第二,把寄存器现场保存到当前任务的 TSS。把通用寄存器、段寄存器、EIP 及 EFLAGS 的当前值保存到当前 TSS 中。保存的 EIP 的值是返回地址,指向引起任务切换指令的下一条指令。但不把 LDTR 和 CR3 内容保存到 TSS 中。

第三,把指示目标任务 TSS 的选择子装入 TR。同时,把对应 TSS 描述符装入 TR 高速缓冲寄存器中。此后,当前任务改称为原任务,目标任务改称为当前任务。

第四,基本恢复当前任务(目标任务)的寄存器现场。根据保存在 TSS 中的内容,恢复各通用寄存器、段寄存器、EFLAGS 及 EIP。在装入段寄存器的过程中,为了能正确地处理可能发生的异常,只把对应选择子装入各段寄存器。还装载 CR3 寄存器。

第五,进行链接处理。如果需要链接,那么将指向原任务 TSS 的选择子写入当前任务 TSS 的链接字字段,把当前任务 TSS 描述符类型改为"忙",并将标志寄存器 EFLAGS 中的 NT 位置 1,表示是嵌套任务。如果需要解链,那么把原任务 TSS 描述符类型改为"可用"。如果无链接处理,那么将原任务 TSS 描述符类型置为"可用",当前任务 TSS 描述符类型置为"忙"。由 JMP 指令引起的任务切换不实施链接/解链处理;由 CALL 指令、中断、IRET 指令引起的任务切换要实施链接/解链处理。

第六,把 CR0 中的 TS 标志置为 1。这表示已发生过任务切换,在当前任务使用协处理器指令时,产生自陷。由自陷处理程序完成有关协处理器现场的保存和恢复。这有利于快速地进行任务切换。

第七,把 TSS 中的 CS 选择子的 RPL 作为当前任务特权级设置为 CPL。任务切换可以在一个任务的任何特权级发生,并可切换到另一任务的任何特权级。

第八,装载 LDTR 寄存器。一个任务可以有自己的 LDT,也可以没有。当任务没有 LDT 时,TSS 中 LDT 选择子为 0。如果 TSS 中 LDT 选择子非空,则从 GDT 中读出对应 LDT 描述符,在经过测试后,把所读 LDT 描述符装入 LDTR 高速缓冲寄存器。如果,LDT 选择子为空,则将 LDT 的存在位置成 0,表明任务不使用 LDT。

第九,装载代码段寄存器 CS、堆栈段寄存器 SS 和各数据段寄存器及其它的高速缓冲寄存器。

第十,把调试寄存器 DR7 中的局部启用位设置为 0,以清除局部于原任务的各个断

点和方式。

4. 关于任务状态和嵌套的说明

在段间转移指令 JMP 引起任务切换时，不实施链接，不导致任务的嵌套。它要求目标任务是可用的任务。切换过程中把原任务置为"可用"，目标任务置为"忙"。

在段间调用指令 CALL 引起任务切换时，实施链接，导致任务的嵌套。它要求目标任务是可用的任务。在切换过程中把目标任务置为"忙"，原任务仍保持"忙"；标志寄存器 EFLAGS 中的 NT 位被置 1，表示是嵌套任务。

在由中断/异常引起任务切换时，实施链接，导致任务的嵌套。要求目标任务是可用的任务。在切换过程中把目标任务置为"忙"，原任务仍保持"忙"；标志寄存器 EFLAG 中的 NT 位被置 1，表示是嵌套任务。

在执行 IRET 时引起任务切换，那么实施解链。要求目标任务是忙的任务。在切换过程中把原任务置为"可用"，目标任务仍保持"忙"。

关于中断/异常如何引起任务切换和指令 IRET 如何考虑任务切换的内容在 10.7 节中介绍。

10.6.6 演示任务切换的实例(实例五)

下面给出一个用于演示任务切换的实例。该实例的逻辑功能是在任务切换后显示原任务的挂起点(EIP)值。该实例演示内容包括：直接通过 TSS 段的任务切换，通过任务门的任务切换，任务内特权级的变换及参数传递。

1. 实现流程

为了达到演示任务切换和特权级变换的目的，实例五在保护方式下涉及到两个任务，一个任务称为临时任务，另一个任务称为演示任务。演示任务的功能是演示通过调用门实现特权级的变换和堆栈间参数的自动复制。临时任务和演示任务配合展示任务切换。

实例五的主要实现步骤如图 10.18 所示。在图的右边标出了任务切换和特权级变换的分界情况。在任务切换时，把原任务的现场保存到 TR 所指示的 TSS 内，然后再把指向目标任务 TSS 描述符的选择子装入 TR，所以，在从临时任务切换到演示任务之前，要把指向临时任务 TSS 描述符的选择子装入 TR。通过把演示任务的 TSS 初始化成恢复点在特权级为 2 的代码段，使得在从临时任务切换到演示任务后，当前特权级 CPL=2。

2. 源程序组织和清单

实例五有如下部分组成：

(1) 全局描述符表 GDT。GDT 含有演示任务 TSS 描述符和 LDT 段描述符，还含有临时任务 TSS 描述符和临时任务的代码段描述符，此外，还含有子程序代码段描述符、规范数据段描述符和视频缓冲区段描述符。

(2) 演示任务的 TSS 段。已根据演示要求初始化。

(3) 演示任务的 LDT 段。它含有演示任务的 0 级和 2 级堆栈段描述符、代码段和数据段描述符、分别以数据段方式描述 LDT 和临时任务 TSS 的数据段描述符、以及指向子程序的调用门和指向临时任务的任务门。

(4) 演示任务的 0 级和 2 级堆栈段。32 位段，特权级分别为 0 和 2。

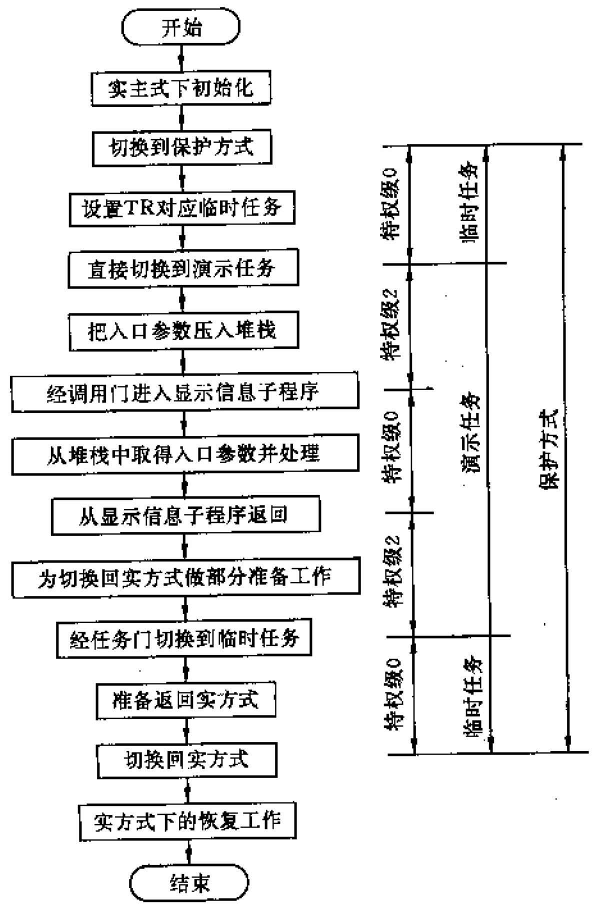


图 10.18 实例五的主要实现步骤

(5) 演示任务数据段。32 位段,特权级 3。
(6) 子程序代码段。32 位代码段,特权级 0。
(7) 演示任务代码段。32 位代码段,特权级 2。
(8) 临时任务的 TSS 段。未初始化。
(9) 临时任务代码段。16 位段,特权级 0。
(10) 实方式下的数据和代码段。

该实例的逻辑功能是在任务切换后显示原任务的挂起点值。源程序清单如下:

```
;程序名:T10-5.ASM
;功  能:演示任务切换和任务内特权级变换
;
        INCLUDE     386SCD.ASM      ;参见实例三
;
        .386P
;------------------------------------------------
```

```
;全局描述符表
GDTSEG      SEGMENT    PARA   USE16
    GDT     LABEL      BYTE
DUMMY       DESCRIPTOR     < >           ;空描述符
NORMAL      DESCRIPTOR     <0FFFFH,0,0,ATDW,0>
Normal_SEL  =   NORMAL_GDT
EFFGDT      LABEL      BYTE
;演示任务的任务状态段描述符
DEMOTSS     DESCRIPTOR     <DemoTSSLEN_1,DemoTSSSEG,,AT386TSS,>
DemoTSS_SEL      =    DEMOTSS_GDT
;演示任务的局部描述符表段描述符
DEMOLDTAB        DESCRIPTOR  <DemoLDTLEN_1,DemoLDTSEG,,ATLDT,>
DemoLDT_SEL      =    DEMOLDTAB_GDT
;临时任务的任务状态段描述符
TEMPTSS     DESCRIPTOR   <TempTSSLEN_1,TempTSSSEG,,AT386TSS+DPL2,>
TempTSS_SEL      =    TEMPTSS_GDT
;临时任务代码段
TEMPCODE   DESCRIPTOR    <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL     =    TEMPCODE_GDT
;子程序代码段描述符
SUBR         DESCRIPTOR     <SUBRLEN_1,SUBRSEG,,ATCE+D32,>
SUBR_SEL     =    SUBR_GDT + RPL3
;显示缓冲区段描述符
VIDEOBUFF  DESCRIPTOR     <0FFFFH,0,0,0F00H+ATDW+DPL3,0>
Video_SEL    =    VIDEOBUFF_GDT
GDNUM        =    ($_EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN       =    $_GDT
GDTSEG     ENDS
;------------------------------------------------------------
;演示任务的任务状态段
DemoTSSSEG      SEGMENT      PARA     USE 16
        DD      0                           ;链接字
        DD      DemoStack0LEN               ;0 级堆栈指针
        DW      DemoStack0_SEL,0
        DD      0                           ;1 级堆栈指针(实例不使用)
        DW      ?,0
        DD      DemoStack2LEN               ;2 级堆栈指针
        DW      DemoStack2_SEL,0
        DD      0                           ;CR3
        DW      DemoBegin,0                 ;EIP
        DD      0                           ;EFLAGS
        DD      0                           ;EAX
```

```
        DD      0                   ;ECX
        DD      0                   ;EDX
        DD      0                   ;EBX
        DD      DemoStack2LEN       ;ESP
        DD      0                   ;EBP
        DD      0                   ;ESI
        DD      0B8000H             ;EDI
        DW      Video_SEL,0         ;ES
        DW      DemoCode_SEL,0      ;CS
        DW      DemoStack2_SEL,0    ;SS
        DW      DemoData_SEL,0      ;DS
        DW      ToDLDT_SEL,0        ;FS
        DW      ToTTSS_SEL,0        ;GS
        DW      DemoLDT_SEL,0       ;LDTR
        DW      0
        DW      $+2                 ;I/O 许可位图指针
        DB      0FFH                ;I/O 许可位图结束字节
DemoTSSLEN      =   $
DemoTSSSEG      ENDS
;------------------------------------------------------------------
;演示任务的局部描述符表 LDT
DemoLDTSEG      SEGMENT     PARA    USE16
DemoLDT         LABEL   BYTE
;0 级堆栈段描述符(32 位段)
DEMOSTACK0      DESCRIPTOR      <DemoStack0LEN_1,DemoStack0SEG,,
                                            ATDW+D32,>
DemoStack0_SEL  =   (DEMOSTACK0_DemoLDT) + TIL
;1 级堆栈段描述符(32 位段,DPL=2)
DEMOSTACK2      DESCRIPTOR      <DemoStack2LEN_1,DemoStack2SEG,,
                                            ATDW+D32+DPL2,>
DemoStack2_SEL  =   (DEMOSTACK2_DemoLDT) + TIL + RPL2
;演示代码段描述符(32 位段,DPL=2)
DEMOCODE        DESCRIPTOR      <DemoCodeLEN_1,DemoCodeSEG,,
                                            ATCE+D32+DPL2,>
DemoCode_SEL    =   (DEMOCODE_DemoLDT) + TIL + RPL2
;演示数据段描述符(32 位段,DPL=3)
DEMODATA        DESCRIPTOR      <DemoDataLEN_1,DemoDataSEG,,
                                            ATDW+D32+DPL3,>
DemoData_SEL    =   (DEMODATA_DemoLDT) + TIL
;把 LDT 作为普通数据段描述的描述符(DPL=2)
TODLDT          DESCRIPTOR      <DemoLDTLEN_1,DemoLDTSEG,,
                                            ATDW+DPL2,>
```

```
ToDLDT_SEL      =    (TODLDT_DemoLDT) + TIL
;把 TSS 作为普通数据段描述的描述符(DPL=2)
TOTTSS          DESCRIPTOR     <TempTSSLEN_1,TempTSSSEG,,
                                                  ATDW+DPL2,>
ToTTSS_SEL      =    (TOTTSS_DemoLDT) + TIL
DemoLDNUM       =    ($_DemoLDT)/(SIZE DESCRIPTOR)
;指向子程序 SUBRB 的调用门(DPL=3)
TOSUBR          GATE    <SUBRB,SUBR_SEL,0,AT386CGAT+DPL3,0>
ToSUBR_SEL      =    (TOSUBR_DemoLDT) + TIL + RPL2
;指向临时任务 Temp 的任务门(DPL=3)
TOTEMPT         GATE   <0,TempTSS_SEL,0,ATTASKGAT+DPL3,0>
ToTempT_SEL     =    (TOTEMPT_DemoLDT) +TIL
DemoLDTLEN      =    $_DemoLDT
DemoLDTSEG      ENDS
;-----------------------------------------------------------
;演示任务的 0 级堆栈(32 位段)
DemoStack0SEG    SEGMENT   PARA   USE32
DemoStack0LEN     =  1024
                 DB   DemoStack0LEN   DUP (0)
DemoStack0SEG    ENDS
;-----------------------------------------------------------
;演示任务的 2 级堆栈(32 位段)
DemoStack2SEG    SEGMENT   PARA   USE32
DemoStack2LEN =  512
                 DB   DemoStack2LEN   DUP (0)
DemoStack2SEG    ENDS
;-----------------------------------------------------------
;演示任务的数据段(32 位段)
DemoDataSEG      SEGMENT   PARA   USE32
Message          DB 'Value=',0
DemoDataLEN      =   $
DemoDataSEG      ENDS
;-----------------------------------------------------------
;子程序段(32 位段)
SUBRSEG SEGMENT   PARA   USE32
          ASSUME  CS:SUBRSEG
SUBRB    PROC     FAR
         PUSH     EBP
         MOV      EBP,ESP
         PUSHAD                                 ;保护现场
          ;从堆栈(0 级)中取提示信息串偏移
         MOV      EAX,[EBP+12]
```

```
          MOV     ESI,EAX
          MOV     AH,7
          JMP     SHORT SUBR2
SUBR1:    STOSW
SUBR2:    LODSB
          OR      AL,AL
          JNZ     SUBR1
          ;从堆栈(0 级)中取显示值
          MOV     EDX,[EBP+16]
          MOV     ECX,8
SUBR3:    ROL     EDX,4
          MOV     AL,DL
          CALL    HTOASC
          STOSW
          LOOP    SUBR3
          POPAD                         ;恢复现场
          POP     EBP
          RET     8
SUBRB     ENDP
;
HTOASC    PROC
          AND     AL,0FH
          ADD     AL,90H
          DAA
          ADC     AL,40H
          DAA
          RET
HTOASC    ENDP
SUBRLEN   =  $
SUBRSEG   ENDS
;-----------------------------------------------------------------
;演示任务的代码段(32 位段)
DemoCodeSEG    SEGMENT   PARA   USE32
          ASSUME   CS:DemoCodeSEG
DemoBegin:
          ;把要复制的参数个数置入调用门
          MOV     FS:ToSUBR.DCOUNT,2
          ;向堆栈(2 级)中压入参数
          PUSH  DWORD  PTR  GS:TempTask.TREIP
          PUSH  OFFSET  Message
          ;通过调用门调用子程序 SUBRB
          CALL32  ToSUBR_SEL,0
```

```
         ;把指向规范数据段描述符的选择子填入临时任务 TSS
         ASSUME    DS:TempTSSSEG
         PUSH      GS
         POP       DS
         MOV       AX,Normal_SEL
         MOV       TempTask.TRDS,AX
         MOV       TempTask.TRES,AX
         MOV       TempTask.TRFS,AX
         MOV       TempTask.TRGS,AX
         MOV       TempTask.TRSS,AX
         ;通过任务切换到临时任务
         JUMP32    ToTempT_SEL,0
DemoCodeLEN       =    $
DemoCodeSEG       ENDS
;----------------------------------------------------------------
;临时任务的任务状态段
TempTSSSEG        SEGMENT   PARA   USE16
TempTask          TASKSS    <>
                  DB   0FFH
TempTSSLEN        =    $
TempTSSSEG        ENDS
;----------------------------------------------------------------
;临时任务的代码段
TempCodeSEG     SEGMENT   PARA   USE16
         ASSUME   CS:TempCodeSEG Virtual:
                  Virtual                    ;装载 TR
         MOV      BX,TempTSS_SEL
         LTR      BX
         ;直接切换到演示任务
         JUMP16   DemoTSS_SEL,0
ToReal:  ;准备返回实方式
         CLTS                                ;清任务切换标志
         MOV      EAX,CR0
         AND      EAX,0FFFFFFFEH
         MOV      CR0,EAX                    ;返回实方式
         JUMP16   <SEG  Real>,<OFFSET  Real>
TempCodeLEN       =    $
TempCodeSEG       ENDS
;================================================================
;实方式数据段
RDataSEG    SEGMENT   PARA   USE16
VGDTR       PDESC <GDTLEN-1,>
```

```
SPVAR       DW   ?
SSVAR       DW   ?
RDataSEG    ENDS
;---------------------------------------------------------------
;实方式代码段
RCodeSEG    SEGMENT   PARA   USE16
            ASSUME  CS:RCodeSEG, DS:RDataSEG, ES:RDataSEG
Start:
        MOV     AX,RDataSEG
        MOV     DS,AX
        CLD
        CALL    INIT_GDT                ;初始化 GDT
        MOV     AX,DemoLDTSEG
        MOV     FS,AX
        MOV     CX,DemoLDNUM
        MOV     SI,OFFSET DemoLDT
        CALL    INIT_LDT                ;初始化演示任务 LDT
        MOV     SSVAR,SS
        MOV     SPVAR,SP                ;保存实方式下的堆栈指针
        ;装载 GDTR
        LGDT    QWORD  PTR  VGDTR
        CLI
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX                 ;切换到保护方式
        JUMP16  <TempCode_SEL>,<OFFSET Virtual>
Real:
        ;又回到实方式
        MOV     AX,RDataSEG
        MOV     DS,AX
        LSS     SP,DWORD  PTR  SPVAR
        STI
        MOV     AX,4C00H
        INT     21H
;---------------------------------------------
INIT_GDT    PROC
;参见实例四
INIT_GDT    ENDP
;初始化 LDT 的子程序
INIT_LDT    PROC
;参见实例四
INIT_LDT    ENDP
```

```
RCodeSEG     ENDS
             END     Start
```

3. 关于实例五的说明

程序中部分片段的背景和实现方法已在前面的实例中做过介绍，下面主要就任务切换和通过调用门实现任务内特权级变换时参数的复制等情形作些说明：

(1) 从临时任务直接通过 TSS 切换到演示任务

在从实方式切换到保护方式后，就认为进入了临时任务。但 TR 并没有指向临时任务的 TSS。在从临时任务切换到演示任务时，要把临时任务的现场保存到临时任务的 TSS，这就要求 TR 指向临时任务的 TSS。所以，首先要使用 LTR 指令把指向临时任务 TSS 描述符的选择子装入 TR。在利用 LTR 指令显式地装载 TR 时，并不引用 TSS 的内容，所以临时任务的 TSS 几乎没有初始化。理由是这不是真正的任务切换。

临时任务采用段间转移指令 JMP，直接指向演示任务的 TSS，切换到演示任务。在执行切换到演示任务的段间转移指令 JMP 时，CPL＝0，JMP 指令中所含选择子内的 RPL＝0，演示任务 TSS 的描述符 DPL＝0，并且是一个可用的 TSS，所以顺利进行从临时任务到演示任务的切换。切换过程包括：把临时任务的执行现场保存到临时任务的 TSS 中；从演示任务的 TSS 中恢复演示任务的现场；把演示任务的 LDT 描述符选择子装载到 LDTR 等。从源程序可见，初始化后的演示任务 TSS 中 CS 字段存放的选择子是 DemoCode_SEL，对应的描述符在演示任务的 LDT 中，并且 DPL＝2，它描述了代码段 DemoCode；挂起点是 DemoBegin，所以在切换到演示任务后从该点开始执行，并且 CPL＝2。由于使用 JMP 指令进行任务切换，所以不实施任务链接。

(2) 从演示任务通过任务门切换到临时任务

演示任务采用段间转移指令 JMP，通过任务门 TOTEMPT 切换到临时任务。在执行切换到临时任务的段间转移指令 JMP 时，CPL＝2，JMP 指令中所含选择子 ToTempT_SEL 内的 RPL＝0，它指示的任务门 DPL＝3，所以可以访问该任务门。任务门内的选择子 TempTSS_SEL 指示临时任务 TSS，并且此时的临时任务 TSS 是可用的，所以可顺利进行任务切换。演示任务的现场保存到演示任务的 TSS；临时任务的现场从临时任务的 TSS 恢复。

临时任务的挂起点是临时任务代码段内的 ToReal 点，所以恢复后的临时任务从该点开始，CS 含临时任务代码段选择子。但由于在演示任务内“强硬”地改变了临时任务 TSS 内的 SS 和 DS 等字段，所以在恢复到临时任务时，SS 和 DS 等段寄存器内已含规范数据段的选择子，而非挂起时的原有值。注意，这种做法不被提倡，但在这里却充分地展示如何从 TSS 恢复任务。

(3) 演示任务内的特权级变换和堆栈传递参数

演示任务采用段间调用指令 CALL，通过调用门 TOSUBR 调用子程序 SUBRB。执行段间调用指令 CALL 时的 CPL＝2，指令所含指向调用门的选择子 RPL＝2，调用门的 DPL＝3，所以对调用门的访问是允许的；尽管调用门内的选择子 RPL＝3，但由于它所指示的子程序代码段描述符 DPL＝0，所以在调用过程中就发生了从特权级 2 到特权级 0 的变换，同时堆栈也被切换。

演示代码通过堆栈传递了两个参数给子程序 SUBRB。在把参数压入堆栈时,CPL=2,使用的也是对应特权级 2 的堆栈。通过调用门进入子程序后,CPL=0,使用 0 级堆栈。为此,把调用门 TOSUBR 中的 DCOUNT 字段设置为 2,表示在特权级向内层变换时,需从外层堆栈依次复制 2 个双字参数到内层堆栈。随着特权级变换,堆栈也跟着变换,如图 10.15 所示。这种在堆栈切换的同时复制所需参数的做法,保证了子程序方便地访问堆栈中的参数,而无需考虑是哪个堆栈。

随着从子程序 SUBRB 的返回,CPL=0 变换为 CPL=2,堆栈也回到 2 级堆栈。由于再进入 0 级堆栈时,总是从空开始,所以在返回前不是非要保持内层堆栈平衡的。但 2 级堆栈中的 2 个双字参数需要废除。从源程序可见,这是采用带立即数的段间返回指令实现的,在返回的同时自动废除外层堆栈中的参数。

(4) 别名技术的应用

在 10.6.2 节对实例三作说明时,已介绍过别名技术。实例五也有两处应用了别名技术。

为了把调用门 TOSUBR 中的 DCOUNT 字段设置成 2,使用一个数据段描述符 TODLDT 描述调用门所在演示任务 LDT 段,该描述符把演示任务的 LDT 段描述成数据段。请读者考虑源程序如何把指向该数据段描述符的选择子装载到 FS 寄存器。

还有一处是把临时任务的 TSS 视作为普通数据段。在从演示任务切换到临时任务之前,把指向描述规范数据段的描述符 NORMAL 的选择子 Normal_SEL 填到临时任务 TSS 中的各数据段寄存器(包括堆栈段寄存器)字段,于是在切换到临时任务时,作为恢复临时任务的现场,该选择子就被装到 DS 等数据段寄存器,对应的描述符 NORMAL 内的信息也就被装入到对应的高速缓冲寄存器中,达到为从临时任务切换到实方式作准备的目的。

10.7 80386 的中断和异常

第 5 章介绍了中断的基本概念和 8086/8088 处理中断的有关内容。80386 除了保持 8086/8088 的相关功能外,还增强了中断处理功能,并引入“异常”的概念。本节在第 5 章的基础上介绍 80386 的中断和异常。

10.7.1 80386 的中断和异常

8086/8088 把中断分为内部中断和外部中断两大类。为了支持多任务和虚拟存储器等功能,80386 把外部中断称为“中断”,把内部中断称为“异常”。与 8086/8088 一样,80386 通常在两条指令之间响应中断或异常;80386 最多处理 256 种中断或异常。

1. 中断

对 80386 而言,中断是由异步的外部事件引起的。外部事件及中断响应与正执行的指令没有关系。通常,中断用于指示 I/O 设备的一次操作已完成。与 8086/8088 一样,80386 有两根引脚 INTR 和 NMI 接受外部中断请求信号。INTR 接受可屏蔽中断请求。NMI 接受不可屏蔽中断请求。在 80386 中,标志寄存器 EFLAGS 中的 IF 标志决定是否屏蔽可屏

蔽中断请求。

外部硬件在通过 INTR 发出中断请求信号的同时，还要向处理器给出一个 8 位的中断向量号。处理器在响应可屏蔽中断请求时，读取这个由外部硬件给出的中断向量号。处理器对这个中断向量号并没有规定。但在具体的微机系统中，系统必须通过软件和硬件的配合设置，使得给出的这个中断向量号不仅与外部中断源对应，而且要避免中断向量号使用冲突情况的出现。可编程中断控制器芯片 8259A 可配合 80386 工作，能够根据设置向处理器提供上述这个中断向量号和还能处理中断请求的优先级。每个 8259A 芯片可以支持 8 路中断请求信号，如果使用 9 个 8259A 芯片，就可使 80386 在单个引脚 INTR 上接受多达 64 个中断源的中断请求信号。

处理器不屏蔽来自 NMI 的中断请求。处理器在响应 NMI 中断时，不从外部硬件接收中断向量号。与 8086/8088 一样，在 80386 中，不可屏蔽中断所对应的中断向量号固定为 2。为了避免不可屏蔽中断的嵌套，每当接受一个 NMI 中断，处理器就在内部屏蔽了再次响应 NMI，这一屏蔽过程直到执行中断返回指令 IRET 后才结束。所以，NMI 处理程序应以 IRET 指令结束。

2. 异常

异常是 80386 在执行指令期间检测到不正常的或非法的条件所引起的。异常与正执行的指令有直接的联系。例如：执行除法指令时，除数等于 0。再如，执行指令时发现特权级不正确。当发生这些情况时，指令就不能成功完成。软中断指令“INT n”和“INTO”也归类于异常而不称为中断，这是因为执行这些指令产生异常事件。

80386 识别多种不同类别的异常，并赋予每一种类别不同的中断向量号。异常发生后，处理器就象响应中断那样处理异常。也即，根据中断向量号，转相应的中断处理程序。把这种中断处理程序称为异常处理程序可能更合适。

根据引起异常的程序是否可被恢复和恢复点不同，把异常进一步分类为故障(Fault)、陷阱(Trap)和中止(Abort)。我们把对应的异常处理程序分别称为故障处理程序、陷阱处理程序和中止处理程序。

故障是在引起异常的指令之前，把异常情况通知给系统的一种异常。80386 认为故障是可排除的。当控制转移到故障处理程序时，所保存的断点 CS 及 EIP 的值指向引起故障的指令。这样，在故障处理程序把故障排除后，执行 IRET 返回到引起故障的程序继续执行时，刚才引起故障的指令可重新得到执行。这种重新执行，不需要操作系统软件的额外参与。故障的发现可能在指令开始执行之前，也可能在指令执行期间。如果在执行指令期间检测到故障，那么中止故障指令，并把指令的源操作数恢复为指令开始执行之前的值。这可保证故障指令的重新执行得到正确的结果。例如，在一条指令的执行期间，如果发现段不存在，那么停止该指令的执行，并通知系统产生段故障，对应的段故障处理程序可通过加载该段的方法来排除故障，之后，原指令就可成功执行，至少不再发生段不存在故障。

陷阱是在引起异常的指令之后，把异常情况通知给系统的一种异常。当控制转移到异常处理程序时，所保存的断点 CS 及 EIP 的值指向引起陷阱的指令的下一条要执行指令。下一条要执行的指令，不一定就是下一条指令。因此，陷阱处理程序并不是总能根据保存的断点，反推确定出产生异常的指令。在转入陷阱处理程序时，引起陷阱的指令应正常完

成，它有可能改变了寄存器或存储单元。软中断指令、单步异常是陷阱的例子。

中止是在系统出现严重情况时，通知系统的一种异常。引起中止的指令是无法确定的。产生中止时正执行的程序不能被恢复执行。系统接收中止后，处理程序要重新建立各种系统表格，并可能需要重新启动操作系统。硬件故障和系统表中出现非法值或不一致值是中止的例子。

3. 优先级

在一条指令执行期间，如检测到不止一个中断或异常，那么按表 10.6 所列优先级通知系统。把优先级最高的中断或异常通知系统，其他优先级较低的异常被废弃，而优先级较低的中断则保持悬挂。请读者考虑，为什么优先级较低的异常可被废弃？

表 10.6　80386 响应中断/异常的优先级

中断/异常类型	优先级	中断/异常类型	优先级
调试故障	最高	调试陷阱	↓
其他故障	↓	NMI 中断	↓
陷阱指令 INT　n 和 INTO	↓	INTR 中断	最低

10.7.2　异常类型

象中断分为多种类型一样，异常也分为多种类型。

1. 80386 识别的异常

80386 识别的多种不同类别的异常及赋予的对应中断向量号列于表 10.7。某些异常还以出错码的形式提供一些附加的信息传递给异常处理程序，出错代码列中的“无”表示没有出错代码，“有”表示有出错代码。

表 10.7　异常一览表

向量号	异常名称	异常类型	出错代码	相关指令
0	除法出错	故障	无	DIV，IDIV
1	调试异常	故障/陷阱	无	任何指令
3	单字节 INT3	陷阱	无	INT 3
4	溢出	陷阱	无	INTO
5	边界检查	故障	无	BOUND
6	非法操作码	故障	无	非法指令编码或操作数
7	设备不可用	故障	无	浮点指令或 WAIT
8	双重故障	中止	有	任何指令
9	协处理器段越界	中止	无	访问存储器的浮点指令
0A	无效 TSS 异常	故障	有	JMP、CALL、IRET、中断
0B	段不存在异常	故障	有	装载段寄存器的任何指令
0C	堆栈段异常	故障	有	装载 SS 寄存器的任何指令 对 SS 寻址的段访问的任何指令

续表

向量号	异常名称	异常类型	出错代码	相关指令
0D	通用保护异常	故障	有	任何特权指令 任何访问存储器的指令
0E	页异常	故障	有	任何访问存储器的指令
10	协处理器出错	故障	无	浮点指令或 WAIT
00~FF	软中断	陷阱	无	INT n

把表 10.7 与表 5.2 比较，就会发现某些中断向量号(如 08－10H)的分配发生了冲突。表 5.2 所列的中断向量号的分配基于 PC 微机系统，使用的 CPU 是 8088。表 10.7 所列中断向量号的分配是 80386 所规定的。实际上，Intel 在宣布 8086/8088 时，保留了这些发生冲突的中断向量号。尽管发生这样的冲突，但以 80386 为 CPU 的微机系统仍可保持与以 8088 为 CPU 的微机系统的兼容，原因是在 80386 的实方式下，几乎不会发生那些中断向量号与外部硬件在提出中断请求时所提供的中断向量号存在冲突的异常。

2. 故障类异常

当发生故障，控制转移到故障处理程序时，所保存的断点 CS 及 EIP 的值指向引起故障的指令，以便在排除故障后恢复执行。

(1) 除法出错故障(异常 0)

除法出错是一种故障。当执行 DIV 指令或 IDIV 指令时，如果除数等于 0，或者商太大，以至于存放商的操作数容纳不下，那么产生这一故障。除法出错故障不提供出错码。

(2) 边界检查故障(异常 5)

如果 BOUND 指令发现被测试的值超出了指令中给定的范围，那么发生边界检查故障。边界检查故障不提供出错码。

(3) 非法操作码故障(异常 6)

如果 80386 不能把 CS 及 EIP 所指存储单元处的位模式识别为某条指令的部分，那么就发生非法操作码故障。当出现如下情况时，发生这样的故障：①操作码字段的内容不是一个合法的 80386 指令的代码；②要求使用存储器操作数的场合，使用了寄存器操作数；③不能被加锁的指令使用了 LOCK 前缀。非法操作码故障不提供出错码。

(4) 设备不可用故障(异常 7)

设备不可用故障支持 80387 数字协处理器。在没有 80387 协处理器硬件的系统中，可用该异常的处理程序代替协处理器的软件模拟器。在发生任务切换时，使得只有在新任务使用浮点指令时，才进行 80387 寄存器状态的切换。设备不可用故障不提供出错码。该故障在下列情况下产生：①在执行浮点指令时，控制寄存器 CR0 中的 EM 位或 TS 为 1；②在执行 WAIT 指令时，控制寄存器 CR0 中 TS 位及 EM 位都为 1。

(5) 段不存在故障(异常 0BH)

处理器在把描述符装入非 SS 段寄存器的高速缓冲器时，如果发现描述符其他方面有效，而 P 位为 0(表示对应段不存在)，那么就发生段不存在故障。有关 SS 段的情形纳入堆栈段故障。在进入故障处理程序时，保存的 CS 及 EIP 指向发生故障的指令；或者该故

障作为任务切换的一部分发生时，指向任务的第一条指令。

段不存在故障提供一个包含引起该异常的段选择子的出错码。出错码的格式如图 10.19 所示。16 位出错码的主要成分是选择子，高 13 位是选择子的索引部分，TI 位是描述符表指示位。

图 10.19 所示出错码格式是段异常时出错码的一般格式。从图中可见出错码中不含选择子的 RPL，而由 IDT 位和 EXT 位代替。当处理某一异常或外部中断时，又发生了某种异常，那么 EXT 位置 1。当从中断描述符表 IDT 中读出表项并产生异常时，IDT 位置 1，这只在中断或异常的处理期间才会发生。当没有选择子时，构成出错码选择子部分的值为 0。

(6) 堆栈段故障(异常 0CH)

当处理器检测到用 SS 寄存器进行寻址的与段有关的某种问题时，就发生堆栈段故障。在进入故障处理程序时，保存的 CS 及 EIP 指向发生故障的指令；或者该故障作为任务切换的一部分发生时，指向任务的第一条指令。

堆栈段故障提供一个出错码，出错码的一般格式也如图 10.19 所示。

15　　　　　　　　　　　　3	2	1	0
选择子的索引部分	TI	IDT	EXT

图 10.19　段异常出错码格式

具体地说，当出现下列三种情况时，将引起堆栈段故障：

① 在堆栈操作时，偏移超出段界限所规定的范围。这种情况下的出错码是 0。例如，PUSH 操作时，堆栈溢出。

② 在由特权级变换所引起的对内层堆栈的操作时，偏移超出段界限所规定的范围。这种情况下的出错码包含有内层堆栈的选择子。

③ 装入到 SS 寄存器(高速缓冲寄存器)的描述符中的存在位为 0。这种情况下的出错码包含有对应的选择子。

上述第一种情况是容易辨别的。第二和第三种情况的辨别要通过判断出错码所含选择子所指示的描述符中的存在位进行。如果存在位为 1，那么是第二种情况；否则是第三种情况。

(7) 无效 TSS 故障(异常 0AH)

当正从任务状态段 TSS 装入选择子时，如果发生除了不存在故障以外的段异常时，就发生无效 TSS 故障。在进入故障处理程序时，保存的 CS 及 EIP 指向发生故障的指令；或者该故障作为任务切换的一部分发生时，指向任务的第一条指令。

无效 TSS 故障提供一个出错码，出错码的格式也如图 10.19 所示，其中选择子部分是指向引起故障的 TSS 的选择子。

一些引起无效 TSS 故障的原因如下：

TSS 描述符中的段限长小于 103；

无效的 LDT 描述符，或者 LDT 未出现；

堆栈段不是一个可写段；

堆栈段选择子索引的描述符超出描述符表界限；

堆栈段 DPL 与新的 CPL 不匹配；

堆栈段选择子 RPL 不等于 CPL；

代码段选择子索引的描述符超出描述符表界限；

代码段选择子不指向代码段；

非一致代码段的 DPL 不等于新的 CPL；

一致代码段 DPL 大于新的 CPL；

对应 DS、ES、FS 或 GS 的选择子指向一个不可读段；

对应 DS、ES、FS 或 GS 的选择子索引的描述符超出描述符表界限。

(8) 通用保护故障(异常 0DH)

除了明确列出的段异常外，其他的段异常都被视作为通用保护故障。在进入故障处理程序时，保存的 CS 及 EIP 指向发生故障的指令；或者该故障作为任务切换的一部分发生时，指向任务的第一条指令。

通用保护故障提供一个出错码，出错码的一般格式也如图 10.19 所示。

根据处理程序可能进行的响应，通用保护故障可分为如下两类：

① 违反保护方式，但程序无须终止的异常。这类故障提供的出错码是 0。这种异常在应用程序执行特权指令或 I/O 访问时发生，支持虚拟 8086 程序的系统或支持虚拟 I/O 访问的系统，需要模拟这些指令，并在模拟完成产生故障的指令后，重新执行被中断的程序。

② 违反保护方式，并导致程序终止的异常。这类故障提供的出错码可能为 0，也可能不为 0(能确定选择子时)。引起这类故障的一些原因如下：

向某个只读数据段或代码段写；

从某个只能执行的代码段读出；

将某个系统段描述符装入到数据段寄存器 DS、ES、FS、GS 或 SS；

将控制转移到一个不可执行的段；

在通过段寄存器 CS、DS、ES、FS 或 GS 访问内存时，偏移越出段界限；

当访问某个描述符表时，超过描述符表段界限；

把 PG 位为 1 但 PE 位为 0 的控制信息装入到 CR0；

切换到一个正忙的任务。

对上述两类通用保护故障的辨别，可通过检查引起故障的指令和出错码进行。如果出错码非 0，那么肯定是第二类通用保护故障。如果出错码是 0，那么需进一步检查引起故障的指令，以确定它是否是系统支持的可以模拟的指令。

(9) 页故障(异常 0EH)

关于页故障的详细说明见 10.10.4 节。

(10) 协处理器出错(异常 10H)

协处理器出错故障指示协处理器发生了未被屏蔽的数字错误，如上溢或下溢。在引起

故障的浮点指令之后的下一条浮点指令或 WAIT 指令，把协处理器出错作为一个故障通知给系统。协处理器出错故障不提供出错码。

2. 陷阱类异常

(1) 调试陷阱(异常 1)

调试异常有故障类型，也有陷阱类型。调试程序可以访问调试寄存器 DR6，以确定调试异常的原因和类型。调试异常不提供出错码。

(2) 单字节 INT3(异常 3)

INT3 是一条特别的一字节"INT n"指令。调试程序可利用该指令支持程序断点。INT3 指令被看成是一种陷阱，而不是一个中断。当由于执行 INT3 指令进入异常 3 处理程序时，被保存的 CS 和 EIP 指向紧跟 INT3 的指令，也即 INT3 指令后面的一个字节。INT3 陷阱不提供出错码。

(3) 溢出(异常 4)

INTO 指令提供条件陷阱。如果 OF 标志为 1，那么 INTO 指令产生陷阱；否则不产生陷阱，继续执行 INTO 后面的指令。在进入溢出处理程序时，被保存的 CS 和 EIP 指向 INTO 指令的下一条指令。溢出陷阱不提供出错码。

3. 中止类异常

(1) 双重故障异常(异常 8)

当系统正在处理一个异常时，如果又检测到一个异常，处理器试图向系统通知一个双重故障，而不是通知第二个异常。双重故障被分在中止异常那一类，所以在转入双重故障处理程序时，被保存的 CS 和 EIP 可能不指向引起双重故障的指令，而且指令的重新启动不支持双重故障。双重故障提供的出错码是 0。

当正处理一个段故障异常时，有可能又产生一个页故障。在这种情况下，通知给系统的是页故障异常而不是双重故障异常。但是，如果正处理一个段或页故障时，又一个段故障被检测到；或者如果正处理一个页故障时，又一个页故障被检测到，那么就引起双重故障。

当正处理一个双重故障时，又一个段或页故障被检测到，那么处理器暂停执行指令，并进入关机方式。关机方式类似于处理器执行一条 HLT 指令后的状态：处理器空转，并维持到处理器接收到一个 NMI 中断请求或者被重新启动为止。在关机方式下，处理器不响应 INTR 中断请求。

双重故障通常指示系统表出现严重的问题，例如段描述符表、页表或中断描述符表出现问题。双重故障处理程序在重建系统表后，可能不得不重新启动操作系统。

(2) 协处理器段越界(异常 9)

协处理器段越界异常被分在中止异常这一类。当浮点指令操作数超出段界限时，产生该中止异常。协处理器段越界异常不提供出错码。

10.7.3 中断和异常的转移方法

80386 在实方式下的中断和异常转移方法与第 5 章所介绍的 8086/8088 响应中断的方法相同。这里介绍的中断和异常的转移方法是指 80386 在保护方式下响应中断和处理

异常时所采用的转移方法，

1. 中断描述符表IDT

与8086/8088一样，在响应中断或者处理异常时，80386根据中断向量号转对应的处理程序。但是，在保护方式下80386不再使用实方式下的中断向量表，而是使用中断描述符表IDT(Interrupt Descriptor Table)。在保护方式下，80386把中断向量号作为中断描述符表IDT中描述符的索引，而不再是中断向量表中的中断向量的索引。

像全局描述符表GDT一样，在整个系统中，中断描述符表IDT只有一个。中断描述符表寄存器IDTR指示IDT在内存中的位置，这也与如图10.10所示的GDTR指示GDT相似。由于80386只识别256个中断向量号，所以IDT最大长度是2K。

中断描述符表IDT所含的描述符只能是中断门、陷阱门和任务门。也就是说，在保护方式下，80386只有通过中断门、陷阱门或任务门才能转移到对应的中断或异常处理程序。

图10.13给出了门描述符的格式。从中可见门描述符包含由选择子和偏移量构成的48位全指针。另外，双字计数字段对中断门、陷阱门和任务门而言无意义。

2. 中断响应和异常处理的步骤

由硬件自动实现的中断响应和异常处理的步骤如下：

首先，判中断向量号要索引的门描述符是否超出IDT的界限。如果超出界限，就引起通用保护故障，出错码是中断向量号乘8再加2。

其次，从IDT中取得对应的门描述符，分解出选择子、偏移量和描述符属性类型，并进行有关检查。描述符只能是任务门、286中断门、286陷阱门、386中断门或386陷阱门，否则，就引起通用保护故障，出错码是中断向量号乘8再加2。如果是由于INT n指令或者INTO指令引起转移，还要检查中断门或陷阱门描述符中的DPL是否满足CPL<=DPL。这种检查可以避免应用程序执行INT n指令时，使用分配给各种设备用的中断向量号。如果检查不通过，就引起通用保护故障，出错码是中断向量号乘8再加2。门描述符中的P位必须是1，表示门描述符是一个有效项，否则就引起段不存在故障，出错码是中断向量号乘8再加2。

最后，根据门描述符类型，分情况转入中断或异常处理程序。

对于异常处理，在开始上述步骤之前，还要根据异常类型确定返回点；如果有出错码，则形成符合出错码格式的出错码。对于异常处理，如果有出错码，在实际执行异常处理程序之前，还要把出错码压入堆栈。为了保证堆栈的双字边界对齐，16位的出错码以32位的值压入，其中高16位的值未作定义。

3. 通过中断门或陷阱门的转移

如果中断向量号所指示的门描述符是386中断门或386陷阱门，那么控制转移到当前任务的一个处理程序过程，并且可以变换特权级。与通过调用门的CALL指令一样，从中断门或陷阱门中获取指向处理程序的48位全指针。其中，16位选择子是对应处理程序代码段的选择子，它指示GDT或LDT中的描述符；32位偏移指示处理程序入口点在代码段内的偏移。

通过中断门或陷阱门的转移过程如图10.20所示。该过程由硬件自动进行。图中“开

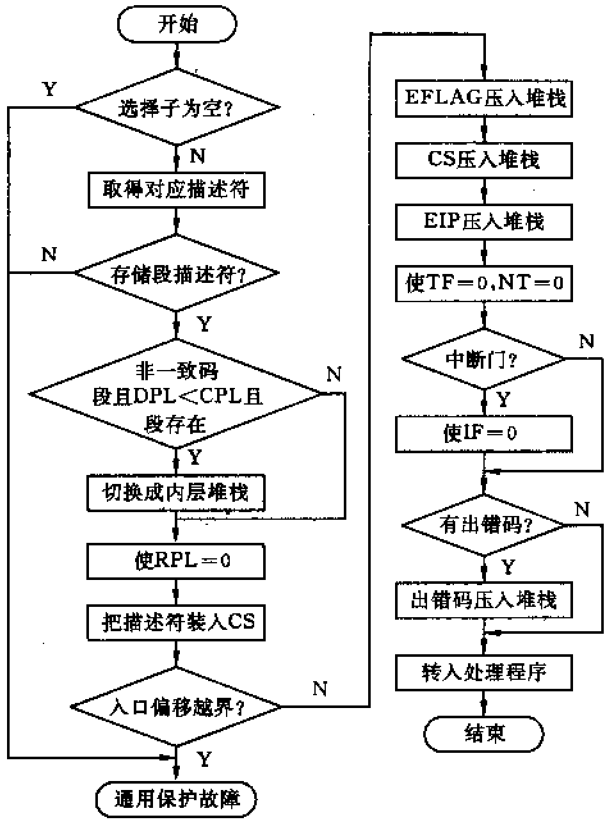


图 10.20　通过中断门或陷阱门的转移过程

始"处表示接上述分情况转移，所以此时已对由中断向量号所索引的 IDT 中的中断门或陷阱门描述符进行过必要的检查，并从中取得指示处理程序的由选择子和偏移构成的 48 位全指针。"结束"处表示转入实际的中断或陷阱处理程序。

图 10.20 所示通过中断门或陷阱门的转移与通过调用门的转移很相似。

从图 10.20 可见，中断门或陷阱门中指示处理程序的选择子必须指向描述一个可执行代码段的描述符。如果选择子为空，就引起通用保护故障，出错码是 0。如果描述符不是代码段描述符，就引起通用保护故障，出错码含选择子。

中断或异常可以转移到同一特权级或内层特权级。上述指定处理程序段的描述符中的类型及 DPL 字段，决定了这种同一任务内的转移是否要发生特权级的变换。如果是一个非一致的代码段，并且 DPL<CPL，那么要发生特权级的变换，堆栈也要切换成内层堆栈。但不复制堆栈中的参数。

图 10.20 中的"把描述符装入 CS"是指把上述指定处理程序段的描述符装入 CS 的高速缓冲寄存器中，在这一步骤中要对描述符进行如 10.6.1 节所述的其他检查，包括是否是代码段描述符和代码段是否存在等，因此可能再发生异常。在对该描述符进行检查时，通过调整 RPL=0 的方法，实现只考虑 DPL，而不考虑门中选择子的 RPL。在把描述符装

入 CS 之后，还要检查门描述符中给出的表示处理程序代码段入口的偏移是否越界，即是否超出段界限。如果越界，就引起出错码为 0 的通用保护故障。

从图 10.20 可见，把标志寄存器和断点压入堆栈的做法和顺序与实方式是相同的，但这里每一次堆栈操作是一个双字，CS 被扩展成 32 位。

把 TF 置成 0，表示不允许处理程序单步执行。把 NT 置成 0，表示处理程序在利用中断返回指令 IRET 返回时，返回到同一任务而不是一个嵌套任务。

从图 10.20 可见，通过中断门的转移和通过陷阱门的转移之间的差别只是对 IF 标志的处理。对于中断门，在转移过程中，把 IF 置成 0，使得在处理程序执行期间，屏蔽掉 INTR 中断；对于陷阱门，在转移过程中，保持 IF 位不变，即如果 IF 位原是 1，那么通过陷阱门转移到处理程序之后仍允许 INTR 中断。因此，中断门最适宜于处理中断，而陷阱门适宜于处理异常。

从图 10.20 可见，在有出错码的情况下，在转入处理程序之前，还要把出错码压入堆栈。只有异常处理才可能有出错码。

图 10.21 给出了通过中断门或陷阱门转移时的堆栈情况。(a)是没有变换特权级和没有出错码的情形；(b)是没有变换特权级和有出错码的情形；(c)是变换特权级和没有出错码的内层堆栈情形；(d)是变换特权级和有出错码的内层堆栈情形。注意，图中每一项为双字。

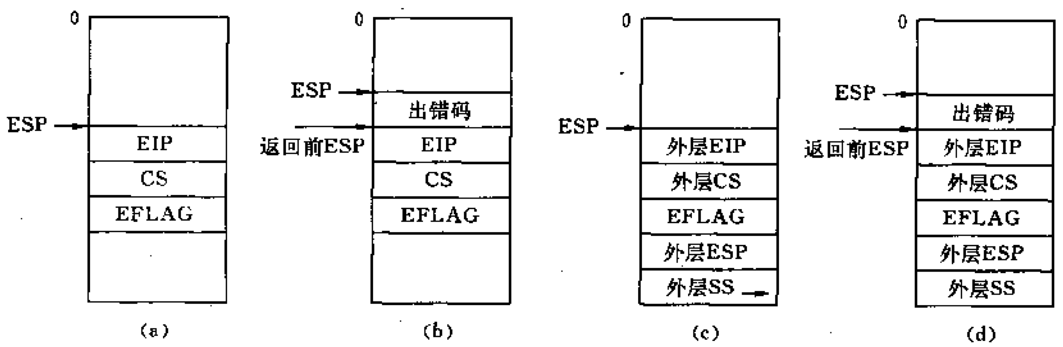


图 10.21 通过中断门或陷阱门转移时的堆栈

4. 通过任务门的转移

如果中断向量号所指示的门描述符是任务门描述符，那么控制转移到一个作为独立的任务方式出现的处理程序。如图 10.13 所示，任务门中含 48 位全指针。这时，16 位选择子是指向描述对应处理程序任务的 TSS 段的选择子，也即该选择子指示一个可用的 286TSS，或 386TSS。通过任务门的转移与通过任务门到一个可用的 386TSS 的 CALL 指令的转移很相似，主要的区别是，对于提供出错码的异常处理，在完成任务切换之后，把出错码压入新任务的堆栈中。

通过任务门的转移，在进入中断或异常处理程序时，标志寄存器 EFLAGS 中的 NT 位被置 1，表示是嵌套任务。

在响应中断或处理异常时，使用任务门可提供一个处理程序任务的自动调度。这种任

务调度由硬件直接执行，并且越过包含在操作系统中的软件任务切换，这就为处理程序提供了一个快速的任务切换。

5. 转移方法的比较

对中断的响应和异常的处理，80386允许通过使用中断门或陷阱门实现由当前任务之内的一个过程进行处理；也允许通过使用任务门实现由另外一个任务进行处理。在当前任务内的处理程序较为简单，并可以很快转移到处理程序，但处理程序要负责保存及恢复处理器的寄存器等内容。转到不同任务的处理程序要花费较长时间，保存及恢复处理器寄存器内容的开销作为任务切换的一部分。使用当前任务内的处理程序的方法，在响应中断或处理异常时，对正执行任务的状态可直接进行访问，但是，这样就要求每一个任务之内都包含一个处理程序。使用独立任务的处理方法，使处理程序得到较好的隔离，但在响应中断或处理异常时，对原任务状态的访问变得较为复杂。

无效TSS异常必须使用任务门进行处理，以保证处理程序有一个有效的任务环境。其他的异常通常在任务环境之内进行处理。在任务内，异常被检测并且不必屏蔽中断，所以使用陷阱门。由陷阱门指示的异常处理程序，是一个由所有任务共享的过程，所以该处理程序最好置于全局地址空间之内。如果各个任务要求有不同的处理程序，那么全局异常处理程序可保存一个各处理程序的入口表，并为引起异常的任务调用相应的处理程序。

中断通常与正执行的任务没有关系，并可能从使用任务门提供的隔离中获得好处。要求较快响应的中断，通过中断门可以得到较好的处理。因为中断随时都可能发生，所以，通过中断门访问的中断处理程序，必须置于全局地址空间中，以便对所有的任务都有效。

6. 中断或异常处理后的返回

中断返回指令IRET用于从中断或异常处理程序的返回。该指令的执行根据任务嵌套标志NT是否为1，分两种情形。

NT为1，表示是嵌套任务的返回。当前TSS中的链接字段保存由前一任务的TSS的选择子，取出该选择子，进行任务切换就完成了返回。这种情形在由通过任务门转入的中断或异常处理程序返回时出现，因为在由中断门或陷阱门转入处理程序时，NT位已被清0。

NT为0，表示当前任务内的返回。这种情形在由通过中断门或陷阱门转入的中断或异常处理程序返回时出现。具体进行的操作包括：从堆栈顶弹出返回指针EIP及CS，然后弹出EFLAG值。弹出的CS选择子的RPL字段，确定返回后的特权级。如果返回选择子的RPL与CPL相同，则不进行特权级的改变。若RPL规定了一个外层特权级，则需要特权级改变，从内层堆栈中弹出外层堆栈的ESP及SS的值，参见图10.21。这些做法与RET指令的实现相似。例如，使用返回CS选择子的RPL，而不是由选择子标识的段的DPL，是为了返回到不在DPL给定的级执行的一致代码段。

对于提供出错代码的异常的处理程序，必须先从堆栈中弹出出错代码，然后再执行IRET指令。

中断返回指令IRET不仅能够用于由中断/异常引起的嵌套任务的返回，而且也适用于由段间调用指令CALL通过任务门引起的嵌套任务的返回。如10.6.5节所述，在执行通过任务门进行任务切换的CALL指令时，标志寄存器中的NT被置1，表示任务嵌套。

10.7.4 演示中断处理的实例(实例六)

下面给出一个用于演示中断处理的实例。该实例的逻辑功能是,在屏幕的左上角以倒计时方式显示秒为单位的时间,在时间用完后结束。该实例演示内容包括:外部中断处理程序,陷阱处理程序。

1. 源程序组织和清单

本实例有如下几部分组成:

(1) 全局描述符表 GDT。GDT 中除了含有常见的几个描述符外,含有描述时钟中断处理程序所使用的代码段和数据段,还含有描述显示程序所使用的代码段和数据段。

(2) 中断描述符表 IDT。为了在保护方式下响应中断和处理异常,必须有 IDT。IDT 含有 256 个门描述符。8H 号安排的是一个通向时钟中断处理程序的中断门,0FEH 号安排的是通向显示处理程序的陷阱门,其他均安排成通向其他中断或异常处理程序的陷阱门。

(3) 时钟中断处理程序的代码段和数据段。

(4) 实现直接填显示缓冲区进行显示的显示程序的代码段和数据段。

(5) 处理其他中断或异常的处理程序的代码段。

(6) 演示程序的代码段、数据段和堆栈段等。

(7) 实方式下执行的启动和结束程序代码段和数据段。

源程序清单如下:

```
;程序名: T10-6.ASM
;功  能: 演示中断处理的实现
;
        INCLUDE     386SCD.ASM            ;参见实例三
;
        .386P
;---------------------------------------------------------------------
;部分常量定义
EOICOM      =    20H                      ;外部中断处理结束命令
ICREGP      =    20H                      ;中断控制寄存器端口地址
IMREGP      =    21H                      ;中断屏蔽寄存器端口地址
;---------------------------------------------------------------------
;全局描述符表 GDT
GDTSEG      SEGMENT  PARA  USE16
GDT         LABEL  BYTE
DUMMY       DESCRIPTOR  <>
NORMAL      DESCRIPTOR  <0FFFFH,0,0,ATDW,0>
Normal_SEL    =  NORMAL_GDT
EFFGDT      LABEL  BYTE
;临时代码段描述符
```

```
TEMPCODE     DESCRIPTOR   <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL   =   TEMPCODE_GDT
;演示任务代码段描述符
DEMOCODE   DESCRIPTOR   <DemoCodeLEN-1,DemoCodeSEG,,ATCE,>
DemoCode_SEL   =   DEMOCODE_GDT
;演示任务数据段描述符
DEMODATA   DESCRIPTOR   <DemoDataLEN-1,DemoDataSEG,,ATDW,>
DemoData_SEL   =   DEMODATA—GDT
;演示任务堆栈段描述符
DEMOSTACK    DESCRIPTOR   <DemoStackLEN-1,DemoStackSEG,,ATDWA,>
DemoStack_SEL   =   DEMOSTACK_GDT
;FEH 号中断处理程序(显示程序)代码段描述符
ECHOCODE   DESCRIPTOR   <EchoCodeLEN-1,EchoCodeSEG,,ATCE,>
EchoCode_SEL   =   ECHOCODE_GDT
;FEH 号中断处理程序(显示程序)数据段描述符
ECHODATA   DESCRIPTOR   <EchoDataLEN-1,EchoDataSEG,,ATDW,>
EchoData_SEL   =   ECHODATA_GDT
;视频缓冲区描述符(B8000H)
VIDEOBUFF   DESCRIPTOR   <80*25*2-1,0B800H,,ATDW,>
VideoBuff_SEL   =   VIDEOBUFF_GDT
;8H 号中断处理程序代码段描述符
TICODE        DESCRIPTOR   <TICodeLEN-1,TICodeSEG,,ATCE,>
TI_SEL           =   TICODE_GDT
;8H 号中断处理程序数据段描述符
TIDATA        DESCRIPTOR   <TiDataLEN-1,TiDataSEG,,ATDW,>
TIData_SEL       =   TIDATA_GDT
;其他中断或异常处理程序代码段描述符
OTHER         DESCRIPTOR   <OTHERCodeLEN-1,OTHERCodeSEG,,ATCE,>
OTHER_SEL   =   OTHER_GDT
;GDT 中的需要进行基地址初始化的描述符个数
GDNUM        =   ($-EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN       =   $-GDT
GDTSEG       ENDS
;-----------------------------------------------------------------
;中断描述符表 IDT
IDTSEG    SEGMENT    PARA USE16
IDT       LABEL   BYTE
;从 00~07 的 8 个陷阱门描述符
          REPT    8
          GATE    <OtherBegin,OTHER_SEL,0,AT386TGAT,0>
          ENDM
```

```
;对应 8H 号(时钟)中断处理程序的中断门描述符
INT08     GATE    <TIBegin,TI_SEL,0,AT386IGAT,0>
;从 09～FDH 的 245 个陷阱门描述符
          REPT    254 -- 9
          GATE    <OtherBegin,OTHER_SEL,0,AT386TGAT,0>
          ENDM
;对应 0FEH 号中断处理程序的陷阱门描述符
INTFE     GATE     <EchoBegin,EchoCode_SEL,0,AT386TGAT,>
;对应 0FFH 号中断处理程序的陷阱门描述符
          GATE     <OtherBegin,OTHER_SEL,0,AT386TGAT,0>
IDTLEN    =   $_IDT
IDTSEG        ENDS
;-----------------------------------------------------------
;其他中断或异常处理程序的代码段
OtherCodeSEG    SEGMENT    PARA    USE16
        ASSUME      CS:OtherCodeSEG
OtherBegin:
        MOV     AX,VideoBuff_SEL
        MOV     ES,AX
        MOV     AH,17H
        MOV     AL,'!'
        MOV     ES:[0],AX          ;在屏幕左上角显示兰底白色符号"!"
        JMP     $                  ;无限循环
OtherCodeLEN    =     $
OtherCodeSEG    ENDS
;-----------------------------------------------------------
;8H 号(时钟)中断处理程序的数据段
TIDataSEG    SEGMENT  PARA  USE16
COUNT        DB  0                 ;中断发生的计数器
TIDataLEN    =   $
TIDataSEG    ENDS
;-----------------------------------------------------------
;8H 号(时钟)中断处理程序的代码段
TICodeSEG  SEGMENT  PARA  USE16
        ASSUME   CS:TICodeSEG,DS:TIDataSEG
TIBegin:
        PUSH    EAX
        PUSH    DS
        PUSH    FS                  ;保护现场
        PUSH    GS
        MOV     AX,TIData_SEL
        MOV     DS,AX               ;置中断处理程序数据段
```

```
        MOV     AX,EchoData_SEL
        MOV     FS,AX                   ;置显示过程数据段
        MOV     AX,DemoData_SEL
        MOV     GS,AX                   ;置演示程序数据段
        ;
        CMP     COUNT,0
        JNZ     TI2                     ;计数非 0 表示未到一秒
        MOV     COUNT,18                ;每秒约 18 次
        INT     0FEH                    ;调用 0FEH 号中断处理程序显示
        CMP     FS:MESS,'0'
        JNZ     TI1
        MOV     GS:FLAG,1               ;显示符号 '0' 时置标记
TI1:    DEC     FS:MESS                 ;调整显示符号
TI2:    DEC     COUNT                   ;调整计数
        POP     GS
        POP     FS                      ;恢复现场
        POP     DS
        MOV     AL,EOICOM
        OUT     ICREGP,AL               ;通知中断控制器中断处理结束
        POP     EAX
        IRETD                           ;中断返回
TICodeLEN   =   $
TICodeSEG   ENDS
;------------------------------------------------------------------
;0FEH 号中断处理程序的数据段
EchoDataSEG  SEGMENT  PARA  USE16
MESS            DB      '8',07H
EchoDataLEN     =   $
EchoDataSEG     ENDS
;------------------------------------------------------------------
;0FEH 号中断处理程序(显示程序)的代码段
EchoCodeSEG   SEGMENT    PARA   USE16
        ASSUME  CS:EchoCodeSEG,DS:EchoDataSEG
EchoBegin:
        PUSH    AX
        PUSH    DS                      ;保护现场
        PUSH    ES
        MOV     AX,EchoData_SEL
        MOV     DS,AX                   ;置显示过程数据段
        MOV     AX,VideoBuff_SEL
        MOV     ES,AX                   ;置视频数据段
        MOV     AX,WORD  PTR  MESS
```

```
        MOV     ES:[0],AX               ;显示符号
        POP     ES
        POP     DS                      ;恢复现场
        POP     AX
        IRETD                           ;中断返回
EchoCodeLEN     =   $
EchoCodeSEG     ENDS
;------------------------------------------------------------
;演示任务的堆栈段
DemoStackSEG    SEGMENT   PARA   USE16
DemoStackLEN    =    1024
        DB      DemoStackLEN   DUP   (0)
DemoStackSEG    ENDS
;------------------------------------------------------------
;演示任务的数据段
DemoDataSEG     SEGMENT   PARA   USE16
FLAG            DB      0
DemoDataLEN     =   $
DemoDataSEG     ENDS
;------------------------------------------------------------
;演示任务的代码段
DemoCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS:DemoCodeSEG,DS:DemoDataSEG
DemoBegin:
        MOV     AX,DemoStack_SEL
        MOV     SS,AX                   ;置堆栈
        MOV     SP,DemoStackLEN
        MOV     AX,DemoData_SEL
        MOV     DS,AX                   ;置数据段
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        ;
        MOV     AL,11111110B            ;置中断屏蔽寄存器
        OUT     IMREGP,AL               ;仅开放时钟中断
        STI                             ;开中断
DemoConti:
        CMP     FLAG,0                  ;判标志
        JZ      DemoConti               ;为0继续
        ;
        CLI                             ;关中断
        ;转回临时代码段,准备回实方式
```

```
OVER: JUMP16 TempCode_SEL,<OFFSET ToDOS>
DemoCodeLEN     =    $
DemoCodeSEG     ENDS
;-----------------------------------------------------------------
;临时代码段
TempCodeSEG     SEGMENT   PARA   USE16
        ASSUME   CS : TempCodeSEG
Virtual: ;转演示程序
        JUMP16      DemoCode_SEL,DemoBegin
ToDOS: ;准备返回实方式
        MOV      AX,Normal_SEL
        MOV      DS,AX
        MOV      ES,AX
        MOV      FS,AX
        MOV      GS,AX
        MOV      SS,AX
        MOV      EAX,CR0
        AND      EAX,0FFFFFFFEH
        MOV      CR0,EAX                  ;返回实方式
        JUMP16   <SEG Real>,<OFFSET Real>
TempCodeSEG     ENDS
;=================================================================
;实方式下的数据段
RDataSEG        SEGMENT    PARA   USE16
VGDTR           PDESC      <GDTLEN-1,>  ;GDT 伪描述符
VIDTR           PDESC      <IDTLEN-1,>  ;IDT 伪描述符
NORVIDTR        PDESC      <3FFH,0>     ;用于保存原 IDTR 值
SPVAR           DW   ?                  ;用于保存原堆栈指针
SSVAR           DW   ?
IMASKREGV       DB   ?                  ;用于保存原中断屏蔽寄存器值
RDataSEG        ENDS
;-----------------------------------------------------------------
;实方式下的代码段
RCodeSEG        SEGMENT    PARA   USE16
        ASSUME    CS : RCodeSEG,DS : RDataSEG
Start:
        MOV      AX,RDataSEG
        MOV      DS,AX
        CLD
        CALL     INIT_GDT                 ;初始化 GDT
        CALL     INIT_IDT                 ;初始化 IDT
        ;
```

```
        MOV     SSVAR,SS                ;保存堆栈指针
        MOV     SPVAR,SP
        SIDT    NORVIDTR                ;保存 IDTR 值
        IN      AL,IMREGP
        MOV     IMASKREGV,AL            ;保存中断屏蔽字节
        ;
        LGDT    QWORD  PTR  VGDTR       ;置 GDTR
        CLI                             ;关中断
        LIDT    QWORD  PTR  VIDTR       ;置 IDTR
        ;
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX                 ;转保护方式下的临时代码段
        JUMP16  <TempCode_SEL>,<OFFSET Virtual>
Real:   ;又回到实方式
        MOV     AX,RDataSEG
        MOV     DS,AX                   ;置实方式数据段
        LSS     SP,DWORD PTR SPVAR      ;恢复堆栈指针
        LIDT    NORVIDTR                ;恢复 IDTR
        MOV     AL,IMASKREGV            ;恢复中断屏蔽字节
        OUT     IMREGP,AL
        STI                             ;开中断
        MOV     AX,4C00H                ;返回 DOS
        INT     21H
;----------------------------------------------------------------
;初始化过程
INIT_GDT    PROC  NEAR
;同实例四
INIT_GDT    ENDP
;
;初始化 IDTR 伪描述符子程序
INIT_IDT    PROC
        MOV     BX,16
        MOV     AX,IDTSEG
        MUL     BX
        MOV     WORD  PTR  VIDTR.BASE,AX
        MOV     WORD  PTR  VIDTR.BASE+2,DX
        RET
INIT_IDT    ENDP
RCodeSEG    ENDS
            END     Start
```

2. 关于实例六的说明

下面再对上述演示程序作些说明：

(1) 时钟中断仍使用 8H 号中断向量

为了既简单又清楚地演示在保护方式下响应外部中断并进行处理，实例使用了时钟中断源，但没有通过重新设置中断控制器的方法改变对应的中断向量。所以，时钟中断使用的 8H 号中断向量号就与双重故障异常使用的中断向量号发生冲突。但实例仅是演示程序，所以只要保证不发生双重故障异常，就可避免冲突，就不会影响演示。

设置中断屏蔽寄存器，仅开放时钟中断。所以，在开中断状态下，也只可能发生时钟中断，而不会发生其他外部中断。

(2) 时钟中断处理程序的设计

由于通过中断门转时钟中断处理程序，所以在控制转移时不发生任务切换。但作为外部中断，随时可能发生，因此中断处理程序必须采取保护现场等措施。作为演示程序，该中断处理程序检查和调整在其数据段中的计数器；在满 18 次后，就认为已满一秒，再调整用于显示的倒计数信息；如果倒计数信息为 0，那么就设置演示程序数据段中的时间为 0 标志。该中断处理程序通过约定的数据区与显示程序及演示程序交换信息。

(3) 利用一个"软中断"(陷阱处理)程序实现显示

为了演示陷阱及其处理，把显示过程安排成陷阱处理程序，简称为显示程序。上述时钟中断处理程序，通过软中断指令 INT 调用该显示程序，显示倒计时数。在控制转移时，也没有任务切换。该陷阱处理程序相当于一个"软中断"处理程序。

(4) 对其他中断或异常的响应

为了简单，除了 8H 号和 0FEH 号外，IDT 中其他的门均通向同一个处理程序。该处理程序用于处理其他中断或异常。处理过程也极其简单，在屏幕左上角显示兰底白色的符号"!"，然后进入无限循环。实际上，按演示程序现在的安排，不可能发生这种情况。

(5) 没有特权级变换

为了简单，实例涉及的中断处理程序和异常处理程序都保持特权级 0。所以，控制转移时不发生特权级变换。因此，没有使用其他堆栈。

(6) 对 IDT 的初始化

由于 IDT 中门描述符没有 32 位段基地址，并且入口点偏移较小，所以就直接填门描述符结构变量，没有额外再初始化。过程 INIT_IDT 只是设置 IDT 伪描述符。

(7) 装载和保存 IDTR 寄存器

在使 IDT 发挥作用之前，还要装载中断描述符表寄存器 IDTR；但为了回到实方式后，恢复原 IDTR 之内容，所以先保存 IDTR 的内容。实例使用如下指令保存 IDTR：

```
SIDT    NORVIDTR
```

该指令的功能是把 IDTR 的内容保存到存储器中的伪描述符 NORVIDTR。伪描述符 NORVIDTR 的结构如前述结构类型 PDESC 所示，低字是以字节为单位的界限，高双字是基地址。在 10.8 节中对 SIDT 指令作详细说明。

本实例使用如下指令装载 IDTR：

```
LIDT    QWORD  PTR  VIDTR
```

```
LIDT     QWORD   PTR   NORVIDTR
```

LIDT 指令类似于 LGDT 指令，在 10.8 节中对 LIDT 指令再作详细说明。

10.7.5 演示异常处理的实例(实例七)

下面给出一个用于模拟异常和演示异常处理的实例。该实例的逻辑功能是，在屏幕上显示一条提示用户击键方式选择模拟异常类型的字符，然后模拟指定的异常。该实例演示内容包括：除法出错故障处理、溢出陷阱处理、段不存在故障处理、堆栈段出错故障处理和通用保护故障处理；还有作为一个独立任务方式出现的陷阱处理程序。

1. 源程序组织和清单

为了演示以独立任务方式出现的陷阱处理程序，实例含有两个任务：演示任务和读键盘任务。实例有如下几部分组成：

(1) 全局描述符表 GDT 和中断描述符表 IDT；

(2) 读键盘任务局部描述符表、任务状态段、堆栈段和代码段等；

(3) 演示任务的局部描述符表、任务状态段、堆栈段、代码段和数据段等；

(4) 作为演示任务一部分的有关陷阱处理和故障处理程序的代码段；

(5) 作为演示任务一部分的显示出错码过程代码段；

(6) 实方式下执行的启动和结束程序代码段和数据段。

在切换到保护方式后，就进入临时代码段。为了简单，演示不发生特权级的变换。演示步骤如下：

(1) 从临时代码段转移到演示代码段。

(2) 做演示准备。把演示任务的 LDT 选择子装入 LDTR，并填入 TSS，装载任务寄存器 TR，建立演示任务堆栈，设置其他数据段寄存器。

(3) 接收要模拟的异常类型号。通过软中断指令 INT 调用读键盘任务完成该步骤。读键盘任务只有在接收到指定的字符后才结束。接收的字符是 0、4、B、C 和 D。

(4) 按接收的字符模拟异常。也即根据键入的字符，执行有关片段。在这些片段中，有意安排了能引起有关故障或陷阱的指令。

(5) 结束演示，转临时代码段，返回 DOS。

源程序清单如下：

```
;程序名：T10-7.ASM
;功  能：模拟异常和演示异常处理
;
        INCLUDE    386SCD.ASM        ;参见实例三
;
        .386P
;-------------------------------------------------------------------
;全局描述符表 GDT
GDTSEG      SEGMENT   PARA   USE16
GDT         LABEL     BYTE
```

```
DUMMY         DESCRIPTOR     <>
NORMAL        DESCRIPTOR     <0FFFFH,0,0,ATDW,0>
Normal_SEL          =    NORMAL_GDT
EFFGDT        LABEL      BYTE
;临时代码段描述符
TEMPCODE      DESCRIPTOR     <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL        =    TEMPCODE_GDT
;演示任务代码段描述符
DEMOCODE      DESCRIPTOR     <DemoCodeLEN-1,DemoCodeSEG,,ATCE,>
DemoCode_SEL        =    DEMOCODE_GDT
;演示任务局部描述符表段描述符
DEMOLDT       DESCRIPTOR     <DemoLDTLEN-1,DemoLDTSEG,,ATLDT,>
DemoLDT_SEL         =    DEMOLDT_GDT
;演示任务 TSS 段描述符
DEMOTSS       DESCRIPTOR     <DemoTSSLEN-1,DemoTSSSEG,,AT386TSS,>
DemoTSS_SEL         =    DEMOTSS_GDT
;缓冲数据段描述符
XBUFFER       DESCRIPTOR     <BufferLEN-1,BufferSEG,,ATDW,>
XBuffer_SEL         =    XBUFFER_GDT
;读键盘任务局部描述符表段描述符
GKEYLDT       DESCRIPTOR     <GKeyLDTLEN-1,GKeyLDTSEG,,ATLDT,>
GKeyLDT_SEL         =    GKEYLDT_GDT
;读键盘任务 TSS 段描述符
GKEYTSS       DESCRIPTOR     <GKeyTSSLEN-1,GKeyTSSSEG,,AT386TSS,>
GKeyTSS_SEL         =    GKEYTSS_GDT
;视频缓冲区段描述符(B8000H)
VIDEOBUFF     DESCRIPTOR     <80*25*2-1,0B800H,,ATDW,>
VideoBuff_SEL       =    VIDEOBUFF_GDT
;显示陷阱处理程序代码段描述符
ECHOCODE      DESCRIPTOR     <EchoCodeLEN-1,EchoCodeSEG,,ATCE,>
EchoCode_SEL        =    ECHOCODE_GDT
;显示出错码过程代码段描述符
SUBCODE       DESCRIPTOR     <SUBCodeLEN-1,SUBCodeSEG,,ATCE,>
SUBCode_SEL         =    SUBCODE_GDT
;其他中断或异常处理程序代码段描述符
OTHER   DESCRIPTOR           <OTHERCodeLEN-1,OTHERCodeSEG,,ATCE,>
OTHER_SEL           =    OTHER_GDT
;GDT 中需要初始化基地址的描述符个数
GDNUM               =    ($-EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN              =    $-GDT
GDTSEG        ENDS
```

```
;----------------------------------------------------------------
;中断描述符表 IDT
IDTSEG      SEGMENT   PARA   USE16
IDT         LABEL     BYTE
;00H 号陷阱门描述符(对应除法出错故障)
INT00       GATE      <DIVBegin,Divide_SEL,0,AT386TGAT,0>
;从 01~03 的 3 个陷阱门描述符
            REPT      3
            GATE      <OtherBegin,OTHER_SEL,0,AT386TGAT,0>
            ENDM
;04H 号陷阱门描述符(对应溢出陷阱)
INT04       GATE      <OFBegin,OF_SEL,0,AT386TGAT,0>
;从 05~0AH 的 6 个陷阱门描述符
            REPT      6
            GATE      <OtherBegin,OTHER_SEL,0,AT386TGAT,0>
            ENDM
;0BH 号陷阱门描述符(对应段不存在故障)
INT0B       GATE      <SNPBegin,SNP_SEL,0,AT386TGAT,0>
;0CH 号陷阱门描述符(对应堆栈段故障)
INT0C       GATE      <SSEBegin,SSE_SEL,0,AT386TGAT,0>
;0DH 号陷阱门描述符(对应通用保护故障)
INT0D       GATE      <GPBegin,GP_SEL,0,AT386TGAT,0>
;从 0E~EDH 的 240 个陷阱门描述符
            REPT      254-14
            GATE      <OtherBegin,OTHER_SEL,0,AT386TGAT,0>
            ENDM
;0FEH 号陷阱门描述符(对应显示中断处理程序)
INTFE       GATE      <EchoBegin,EchoCode_SEL,0,AT386TGAT,>
;0FFH 号任务门描述符(对应读键盘中断处理任务)
INTFF       GATE      <,GKeyTSS_SEL,0,ATTASKGAT,>
IDTLEN      =   $-IDT
IDTSEG      ENDS
;----------------------------------------------------------------
;读键盘任务局部描述符表段
GKeyLDTSEG    SEGMENT   PARA   USE16
GLDT     LABEL   BYTE
;代码段描述符
GKEYCODE        DESCRIPTOR     <0FFFFH,GKeyCodeSEG,,ATCE,>
GKeyCode_SEL        =   (GKEYCODE-GLDT) + TIL
;堆栈段描述符
GKEYSTACK       DESCRIPTOR  <GKeyStackLEN-1,GKeyStackSEG,,ATDWA,>
GKeyStack_SEL       =   (GKEYSTACK-GLDT) + TIL
```

```
;该 LDT 中需要初始化基地址的描述符个数
GKeyLDNUM         =   ($-GLDT)/(SIZE DESCRIPTOR)
GKeyLDTLEN        =   $
GKeyLDTSEG      ENDS
;-------------------------------------------------------------------
;读键盘任务 TSS 段
GKeyTSSSEG    SEGMENT    PARA    USE 16
        DD      0                       ;链接字
        DD      ?                       ;0 级堆栈指针
        DW      ?,?
        DD      ?                       ;1 级堆栈指针
        DW      ?,?
        DD      ?                       ;2 级堆栈指针
        DW      ?,?
        DD      0                       ;CR3
        DW      GKeyBegin,0             ;EIP
        DD      0                       ;EFLAGS
        DD      0                       ;EAX
        DD      0                       ;ECX
        DD      0                       ;EDX
        DD      0                       ;EBX
        DW      GKeyStackLEN,0          ;ESP
        DD      0                       ;EBP
        DD      0                       ;ESI
        DD      0                       ;EDI
        DW      Normal_SEL,0            ;ES
        DW      GKeyCode_SEL,0          ;CS
        DW      GKeyStack_SEL,0         ;SS
        DW      Normal_SEL,0            ;DS
        DW      Normal_SEL,0            ;FS
        DW      Normal_SEL,0            ;GS
        DW      GKeyLDT_SEL,0           ;LDT
        DW      0                       ;TSS 的特别属性字
        DW      $+2                     ;指向 I/O 许可位图区的指针
        DB      0FFH                    ;I/O 许可位图结束字节
GKeyTSSLEN    =    $
GKeyTSSSEG    ENDS
;-------------------------------------------------------------------
;读键盘任务堆栈段
GKeyStackSEG    SEGMENT    PARA    USE16
GKeyStackLEN    =    1024
                DB    GKeyStackLEN    DUP (0)
```

```
GKeyStackSEG    ENDS
;------------------------------------------------------------------
;读键盘任务代码段
GKeyCodeSEG     SEGMENT   PARA   USE16
        ASSUME CS:GKeyCodeSEG,DS:RDataSEG,ES:BufferSEG
GKeyBegin:
        PUSH    DS
        PUSH    ES
        PUSH    FS
        PUSH    GS
        MOV     AX,Normal_SEL
        MOV     SS,AX                   ;准备转实方式
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH
        MOV     CR0,EAX                 ;转实方式
        JUMP16 <SEG  GetKey>,<OFFSET  GetKey>
GetKey: ;实方式
        MOV     AX,RDataSEG
        MOV     DS,AX
        MOV     EBP,ESP                 ;恢复实方式部分现场
        LSS     SP,DWORD  PTR  SPVAR
        LIDT    NORVIDTR
        STI
        MOV     DX,OFFSET  MESS
        MOV     AH,9
        INT     21H                     ;显示提示信息
GetKey1:MOV     AH,0
        INT     16H                     ;读键盘
        CMP     AL,'0'
        JZ      GetKey2
        CMP     AL,'4'                  ;只有[0,4,B,C,D]有效
        JZ      GetKey2
        AND     AL,11011111B            ;小写转大写
        CMP     AL,'B'
        JB      GetKey1
        CMP     AL,'D'
        JA      GetKey1
GetKey2:MOV     DL,AL
        MOV     AH,2
        INT     21H                     ;显示所按字符
        MOV     AX,BufferSEG
        MOV     ES,AX
```

```
        MOV    ES：KeyASCII,DL          ;保存到缓冲数据段
        CLI                             ;准备返回保护方式
        LIDT   QWORD  PTR  VIDTR
        MOV    EAX,CR0
        OR     EAX,1
        MOV    CR0,EAX                  ;返回保护方式
        JUMP16 <GKeyCode_SEL>,<OFFSET GetKeyV>
GetKeyV：;又进保护方式
        MOV    AX,GKeyStack_SEL
        MOV    SS,AX
        MOV    ESP,EBP
        POP    GS
        POP    FS
        POP    ES
        POP    DS
        IRETD
        JMP    GKeyBegin
GKeyCodeLEN    =   $
GKeyCodeSEG    ENDS
;-----------------------------------------------------------------
;其他中断或异常处理程序代码段
OtherCodeSEG    SEGMENT   PARA   USE16
        ASSUME  CS：OtherCodeSEG
OtherBegin：;
        MOV    SI,OFFSET MESSOTHER
        INT    0FEH                     ;显示提示信息
        JMP    $                        ;进入无限循环
OtherCodeLEN   =   $
OtherCodeSEG   ENDS
;-----------------------------------------------------------------
;除法出错故障处理程序代码段
DIVCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS：DIVCodeSEG
DIVBegin：
        MOV    SI,OFFSET MESS0
        MOV    DI,0
        INT    0FEH                     ;显示提示信息
        SHR    AX,1                     ;处理模拟的除法错误
        IRETD                           ;返回
DIVCodeLEN     =   $
DIVCodeSEG     ENDS
;-----------------------------------------------------------------
```

```
;溢出陷阱处理程序代码段
OFCodeSEG       SEGMENT    PARA   USE16
        ASSUME   CS : OFCodeSEG
OFBegin:
        MOV     SI,OFFSET   MESS4
        MOV     DI,0
        INT     0FEH                        ;显示提示信息
        IRETD                               ;返回
OFCodeLEN       =    $
OFCodeSEG       ENDS
;----------------------------------------------------------------
;段不存在故障处理程序代码段
SNPCodeSEG      SEGMENT    PARA   USE16
        ASSUME   CS : SNPCodeSEG
SNPBegin:
        MOV     SI,OFFSET   MESSB
        MOV     DI,0
        INT     0FEH                        ;显示提示信息
        ;
        POP     EAX                         ;弹出出错代码
        CALL16 SUBCode_SEL,SUBBegin         ;显示出错代码
        ;
        POP     EAX
        ADD     EAX,2                       ;按模拟的引起段不存在指令
        PUSH    EAX                         ;调整返回地址
        IRETD
SNPCodeLEN      =    $
SNPCodeSEG      ENDS
;----------------------------------------------------------------
;堆栈段故障处理程序代码段
SSECodeSEG      SEGMENT    PARA   USE16
        ASSUME   CS : SSECodeSEG
SSEBegin:
        MOV     SI,OFFSET   MESSC
        MOV     DI,0
        INT     0FEH                        ;显示提示信息
        POP     EAX                         ;弹出出错代码
        CALL16 SUBCode_SEL,SUBBegin         ;显示出错代码
        POP     EAX
        ADD     EAX,4                       ;按模拟的引起堆栈段错误指令
        PUSH    EAX                         ;调整返回地址
        IRETD
```

```
SSECodeLEN      =    $
SSECodeSEG      ENDS
;-----------------------------------------------------------------
;通用保护故障处理程序代码段
GPCodeSEG       SEGMENT    PARA    USE16
        ASSUME    CS:GPCodeSEG
GPBegin:
        PUSH    EBP
        MOV     EBP,ESP
        PUSH    EAX
        PUSH    ESI                         ;保护现场
        PUSH    EDI
        ;
        MOV     SI,OFFSET    MESSD
        MOV     DI,0
        INT     0FEH                        ;显示提示信息
        ;
        MOV     EAX,[BP+4]                  ;从堆栈中取出错代码
        CALL16 SUBCode_SEL,SUBBegin         ;显示出错代码
        ;
        POP     EDI
        POP     ESI                         ;恢复部分现场
        POP     EAX
        ADD     DWORD    PTR [EBP+8],2;按模拟的故障指令调整返回地址
        POP     EBP
        ADD     SP,4                        ;废除堆栈中的出错代码
        IRETD
GPCodeLEN       =    $
GPCodeSEG       ENDS
;-----------------------------------------------------------------
;显示出错码过程代码段
SUBCodeSEG      SEGMENT    PARA    USE16
        ASSUME    CS:SUBCodeSEG
SUBBegin:
        PUSH    AX                          ;AX 含出错代码
        PUSH    CX
        PUSH    DX                          ;保护现场
        PUSH    SI
        PUSH    DI
        MOV     SI,OFFSET ERRCODE
        MOV     DX,AX
        MOV     CX,4
```

```
SUBR1:  ROL     DX,4                    ;把16位出错代码
        MOV     AL,DL                   ;转成4位十六进制数的ASCII码
        AND     AL,0FH                  ;并保存
        ADD     AL,30H
        CMP     AL,'9'
        JBE     SUBR2
        ADD     AL,7
SUBR2:  MOV     [SI],AL
        INC     SI
        LOOP    SUBR1
        ;
        MOV     SI,OFFSET ERRMESS
        MOV     DI,80*2                 ;在第二行首开始
        INT     0FEH                    ;显示出错代码
        POP     DI
        POP     SI
        POP     DX                      ;恢复部分现场
        POP     CX
        POP     AX
        RETF
SUBCodeLEN      =   $
SUBCodeSEG      ENDS
;-----------------------------------------------------------------
;实现显示的陷阱处理程序代码段
EchoCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS:EchoCodeSEG
EchoBegin:  ;DS:SI指向显示信息串,ES:DI指向显示缓冲区
        PUSHAD                          ;保护现场
        CLD
        MOV     AH,7
        MOV     AL,20H
        MOV     CX,80
        PUSH    DI
        REP     STOSW                   ;清所在显示行
        POP     DI
Echo1:  LODSB
        OR      AL,AL
        JZ      Echo2
        STOSW                           ;显示指定信息串
        JMP     Echo1
Echo2:  POPAD                           ;恢复现场
        IRETD
```

```
EchoCodeLEN     =   $
EchoCodeSEG     ENDS
;-----------------------------------------------------------------
;缓冲区数据段
BufferSEG       SEGMENT   PARA   USE16
KeyASCII        DB   ?
Buffer          DB   128   DUP (?)
BufferLEN       =   $
BufferSEG       ENDS
;-----------------------------------------------------------------
;演示任务局部描述符表段
DemoLDTSEG      SEGMENT     PARA   USE16
DLDT            LABEL       BYTE
;演示任务 TSS 段描述符
TODEMOTSS       DESCRIPTOR    <DemoTSSLEN-1,DemoTSSSEG,,ATDW,>
ToDemoTSS_SEL   =    (TODEMOTSS_DLDT) + TIL
;演示任务堆栈段描述符
DEMOSTACK       DESCRIPTOR    <DemoStackLEN-1,DemoStackSEG,,ATDWA,>
DemoStack_SEL   =    (DEMOSTACK_DLDT) + TIL
;演示任务数据段描述符
DEMODATA        DESCRIPTOR    <DemoDataLEN-1,DemoDataSEG,,ATDW,>
DemoData_SEL    =    (DEMODATA_DLDT) + TIL
;除法出错故障处理程序代码段描述符
DIVIDE          DESCRIPTOR    <DIVCodeLEN-1,DIVCodeSEG,,ATCE,>
Divide_SEL      =    (DIVIDE_DLDT) + TIL
;溢出陷阱处理程序代码段描述符
OVERFLOW        DESCRIPTOR    <OFCodeLEN-1,OFCodeSEG,,ATCE,>
OF_SEL          =    (OVERFLOW_DLDT) + TIL
;段不存在故障处理程序代码段描述符
SNPCODE         DESCRIPTOR    <SNPCodeLEN-1,SNPCodeSEG,,ATCE,>
SNP_SEL         =    (SNPCODE_DLDT) + TIL
;堆栈段出错故障处理程序代码段描述符
SSECODE         DESCRIPTOR    <SSECodeLEN-1,SSECodeSEG,,ATCE,>
SSE_SEL         =    (SSECODE_DLDT) + TIL
;通用保护故障处理程序代码段描述符
GPCODE          DESCRIPTOR    <GPCodeLEN-1,GPCodeSEG,,ATCE,>
GP_SEL          =    (GPCODE_DLDT) + TIL
;为模拟段不存在故障而安排的数据段描述符
TESTNPS         DESCRIPTOR    <0FFFFH,0,,ATDW-80H,>
TestNPS_SEL     =    (TESTNPS_DLDT) + TIL
;该 LDT 中需要初始化基地址的描述符个数
DemoLDNUM       =    ($_DLDT)/(SIZE DESCRIPTOR)
```

```
DemoLDTLEN      =   $
DemoLDTSEG      ENDS
;-----------------------------------------------------------------
;演示任务 TSS 段
DemoTSSSEG        SEGMENT  PARA  USE16
DemoTaskSS        TASKSS   < >
                  DB       0FFH
DemoTSSLEN        =  $
DemoTSSSEG        ENDS
;-----------------------------------------------------------------
;演示任务堆栈段
DemoStackSEG      SEGMENT  PARA  USE16
DemoStackLEN      1024
                  DB 1024  DUP  (0)
DemoStackSEG      ENDS
;-----------------------------------------------------------------
;演示任务数据段
DemoDataSEG     SEGMENT   PARA  USE16
MESS0           DB        'Divide Error (Exception 0)',0
MESS4           DB        'Overflow (Exception 4)',0
MESSB           DB        'Segment Not Present (Exception 11)',0
MESSC           DB        'Stack Segment (Exception 12)',0
MESSD           DB        'General Protection (Exception 13)',0
MESSOTHER       DB        'Other Execption',0
ERRMESS         DB        'Error Code = '
ERRCODE         DB        4 DUP (0),'H',0
DemoDataLEN     =         $
DemoDataSEG     ENDS
;-----------------------------------------------------------------
;演示任务代码段
DemoCodeSEG     SEGMENT   PARA  USE16
        ASSUME  CS : DemoCodeSEG
DemoBegin:
        MOV     AX,DemoLDT_SEL
        LLDT    AX                      ;装载 LDTR
        MOV     AX,DemoStack_SEL
        MOV     SS,AX                   ;置堆栈指针
        MOV     ESP,DemoStackLEN
        MOV     AX,ToDemoTSS_SEL
        MOV     GS,AX                   ;把演示任务 LDT 选择子填入 TSS
        MOV     GS : DemoTaskSS. TRLDT,DemoLDT_SEL
        ;装载 TR
```

```
        MOV     AX,DemoTSS_SEL
        LTR     AX
        ;装载其他数据段寄存器
        MOV     AX,DemoData_SEL
        MOV     DS,AX
        MOV     AX,VideoBuff_SEL
        MOV     ES,AX
        MOV     AX,XBuffer_SEL
        MOV     FS,AX
        MOV     AX,XBuffer_SEL
        MOV     GS,AX
        ;接收要模拟的异常类型号
        INT     0FFH
        ;按接收的字符模拟异常
        MOV     AL,FS:KeyASCII
        CMP     AL,'0'
        JNZ     Demo4
Exception0:
        MOV     AX,1000
        MOV     CL,2                    ;模拟除法出错故障
        DIV     CL                      ;该指令长 2 字节
        JMP     OVER
Demo4:
        CMP     AL,'4'
        JNZ     Demo11
Exception4:
        MOV     AL,100
        ADD     AL,50
        INTO                            ;模拟溢出陷阱
        JMP     OVER
Demo11:
        CMP     AL,'B'
        JNZ     Demo12
Exception11:
        MOV     AX,TestNPS_SEL          ;模拟段不存在故障
        MOV     GS,AX                   ;该指令长 2 字节
        JMP     OVER
Demo12:
        CMP     AL,'C'
        JNZ     Demo13
Exception12:
        MOV     EBP,ESP                 ;模拟堆栈出错故障
```

```
        MOV     AL,[EBP]                ;该指令长 4 字节
        JMP     OVER
Demo13:
Exception13:
        MOV     AX,DemoTSS_SEL          ;模拟通用保护故障
        MOV     GS,AX                   ;该指令长 2 字节
        JMP     OVER
;转临时代码段
OVER:   JUMP16  TempCode_SEL,<OFFSET ToDOS>
DemoCodeLEN     =   $
DemoCodeSEG     ENDS
;--------------------------------------------------------------
;临时代码段
TempCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS:TempCodeSEG
Virtual: ;转演示代码段
        JUMP16  DemoCode_SEL, 0
ToDOS: ;演示结束后准备返回实方式
        MOV     AX,Normal_SEL
        MOV     DS,AX
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     SS,AX
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH
        MOV     CR0,EAX                 ;返回实方式
        JUMP16  <SEG  Real>,<OFFSET  Real>
TempCodeSEG     ENDS
;==============================================================
;实方式下的数据段
RDataSEG   SEGMENT   PARA   USE16
VGDTR      PDESC     <GDTLEN-1,>        ;GDT 伪描述符
VIDTR      PDESC     <IDTLEN-1,>        ;IDT 伪描述符
NORVIDTR   PDESC     <3FFH,0>           ;保存 IDTR 原值
SPVAR      DW        ?                  ;保存原堆栈指针
SSVAR      DW        ?
MESS       DB        "Strike a key [0,4,B,C,D]: $"    ;提示信息
RDataSEG   ENDS
;--------------------------------------------------------------
;实方式下的代码段
RCodeSEG   SEGMENT   PARA   USE16
```

```
        ASSUME  CS : RCodeSEG,DS : RDataSEG
Start:
        MOV     AX,RDataSEG
        MOV     DS,AX
        CLD
        CALL    INIT_GDT                        ;初始化 GDT
        CALL    INIT_IDT                        ;初始化 IDT
        MOV     AX,GKeyLDTSEG
        MOV     FS,AX
        MOV     CX,GKeyLDNUM
        MOV     SI,OFFSET GLDT
        CALL    INIT_LDT                        ;初始化读键盘任务 LDT
        MOV     AX,DemoLDTSEG
        MOV     FS,AX
        MOV     CX,DemoLDNUM
        MOV     SI,OFFSET DLDT
        CALL    INIT_LDT                        ;初始化演示任务 LDT
        MOV     SSVAR,SS
        MOV     SPVAR,SP
        LGDT    QWORD  PTR  VGDTR
        SIDT    NORVIDTR
        CLI
        LIDT    QWORD  PTR  VIDTR
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX
        JUMP16  <TempCode_SEL>,<OFFSET Virtual>
Real:
        MOV     AX,RDataSEG
        MOV     DS,AX
        LSS     SP,DWORD  PTR  SPVAR
        LIDT    NORVIDTR
        STI
        MOV     AX,4C00H
        INT     21H
;-----------------------------------------------------------------
;实方式下的初始化过程
INIT_GDT    PROC    NEAR
        ;同实例四中的过程内容
INIT_GDT    ENDP
;
INIT_IDT    PROC
```

```
        ;同实例六过程内容
INIT_IDT    ENDP
;
INIT_LDT    PROC
        ;同实例四过程内容
INIT_LDT    ENDP
RCodeSEG    ENDS
            END     Start
```

2. 关于实例七的说明

上述模拟与演示程序的许多内容与实例六相同，下面就各异常处理程序和读键盘任务的实现作些说明：

(1) 除法出错故障处理程序的实现

从源程序可见，除法出错是在执行故意安排的被除数为1000，而除数为2的无符号除指令时引起。作为演示，除法出错故障处理程序先显示一条提示信息，然后把存放被除数的AX内容右移一位，然后就返回。由于除法出错归入故障异常这一类，所以，在故障处理结束返回后，仍执行该无符号除指令。显然，将再次引起同样的故障，仍把被除数右移一位。但由于每次处理时都把被除数减半，所以几次故障后就不发生该故障。

(2) 溢出陷阱处理程序的实现

作为演示的溢出陷阱处理程序较简单。先显示一条提示信息，然后就返回。因为溢出异常归入陷阱这一类，所以在陷阱处理结束后，就直接返回到引起陷阱的指令的下一条指令。

(3) 段不存在故障处理程序的实现

从源程序可见，段不存在故障是在执行故意安排的把一个选择子送段寄存器GS的指令时引起。该选择子索引的描述符中的存在位P被故意置为0，表示对应段不在内存。在正常情况下，段不存在故障处理程序要把对应的段装入内存，再把描述符内的P位修改为1，于是，在故障处理结束后，引起故障的指令可得到顺利执行。为了简单，这里安排的故障处理程序先显示一条提示信息，然后显示出错码，最后调整堆栈中的返回地址并返回。段不存在故障提供一个出错码，该故障处理程序利用POP指令把它从堆栈中弹出，这样堆栈指针就指向返回地址。由于段不存在异常归入故障这一类，所以返回点仍是引起故障的指令。因此，作为演示程序调整了堆栈中的返回地址。

(4) 堆栈段出错故障处理程序的实现

引起堆栈出错故障的原因有多种，实例通过执行故意安排的偏移超过段界限的堆栈段访问指令来模拟堆栈段出错故障的产生。作为演示的堆栈出错故障处理程序比较简单，先显示一条提示信息，然后显示出错码，最后调整堆栈中的返回地址并返回。

(5) 通用保护故障处理程序的实现

引起通用保护故障的原因有多种，实例通过把一个指向系统段描述符的选择子装入数据段寄存器GS来模拟通用保护故障的产生。作为演示的通用保护故障处理程序，象上述两个故障处理一样比较简单，先显示一条提示信息，然后显示出错码，最后调整堆栈中

的返回地址并返回，但在废除堆栈中的出错码和调整堆栈中的返回地址时采用了其他方法。

(6) 异常处理程序的一般说明

在实例中，通向上述各种异常处理程序的门都是陷阱门。所以，在发生异常而转入这些异常处理程序时，都不发生任务切换。于是，这些异常处理仍作为演示任务的一部分。

正常情况下，异常处理程序应该注意现场的保护和恢复，但为了简单，作为演示的异常处理程序没有能够切实地保护现场。注意，这些异常处理程序所采用的处理方法与所模拟的指令有关，不适用于一般情况。

(7) 显示出错代码的过程

实例采用一个过程用于显示出错码，该过程的入口参数是 AX 含出错码。利用该过程不仅缩短程序，而且也用于表现异常处理程序的实现。

(8) 读键盘任务的实现

在实例的 IDT 中，0FFH 号门描述符是任务门，指向一个独立的任务。该任务的功能是读键盘，接收一个指定范围内的字符。演示任务通过指令“INT 0FFH”指令调用它，接收一个代表需要模拟异常的字符。

为了简单，该任务在实方式下读键盘，接收指定范围内的字符。为此，该任务每次经历如下步骤：① 转回到实方式。此前要做必要的准备，回到实方式后，要恢复必须的实方式下的部分现场。② 接收指定的字符。调用 DOS 功能显示提示信息，调用 BIOS 读键盘，如在指定范围内那么就显示，并保存在约定的数据段中。③转回到保护方式，此前也要做必要的准备。

尽管在任务切换时，自动利用 TSS 保护和恢复现场，但由于该任务相当于一个读键盘的过程，所以在开始任务时，还通过堆栈保护必要的现场，在结束任务时恢复现场。请特别注意，安排在该任务代码段中的 IRETD 指令之后的转移指令的作用。

10.7.6 各种转移途径小结

如上所述，中断/异常可引起任务切换、任务内特权级变换和任务内无特权级变换的转移。至此，任务切换、任务内特权级变换和任务内无特权级变换转移的各种途径已全部列出。

1. 任务切换的途径

任务之间切换的途径如图 10.22 所示。段间转移指令 JMP、段间调用指令 CALL、软中断指令 INT 和中断返回指令 IRET 引起的任务切换是主动的任务切换，或者说是当前任务要求的任务切换。中断和异常(不包括软中断指令)引起的任务切换是被动的任务切换，或者说是不受当前任务左右的任务的切换。

伴随着任务切换，特权级当然可能发生变换。只要任务切换发生，这种特权级的变换取决于目标任务，而与当前特权级无关。

2. 任务内特权级变换的途径

任务内特权级变换的途径如图 10.23 所示。图中特权级 m 是外层特权级，特权级 n 是内层特权级。通常 RET 与 CALL 对应；IRET 与 INT、中断/异常对应。但也可以通过

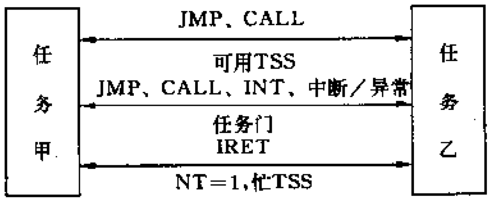


图 10.22　任务切换的途径

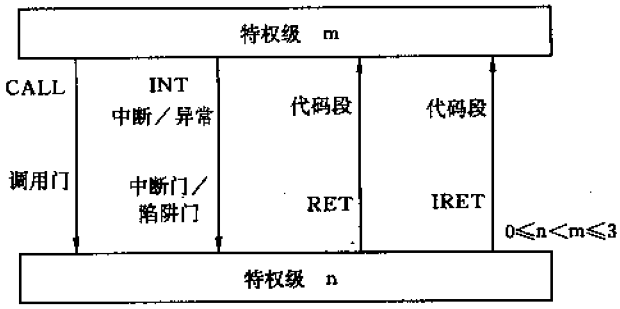


图 10.23　任务内特权级变换的途径

在堆栈中建立合适环境的手段，使用 RET 或 IRET 从内层特权级变换到外层特权级。

3. 任务内相同特权级转移的途径

任务内相同特权级转移的途径如图 10.24 所示。由图可见，任务内相同特权级转移的途径多种多样。

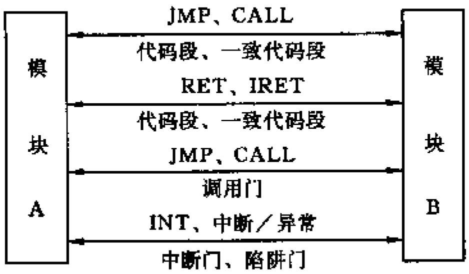


图 10.24　任务内相同特权级转移的途径

10.8　操作系统类指令

在第 9 章已介绍了 80386 新增的普通指令，本节介绍操作系统类指令。其中的某些指令始于 80286。通常只在操作系统代码中使用这些指令，而不在应用程序代码中使用这些指令。这是把它们称为操作系统类指令的原因。为了保证操作系统的安全，保护方式下的 80386 支持四个特权级。相应地，这些操作系统类指令也可分为三种：实方式和任何特权级下可执行的指令、实方式及特权级 0 下可执行的指令和仅在保护方式下执行的指令。

10.8.1 实方式和任何特权级下可执行的指令

1. 存储全局和中断描述符表寄存器指令

全局描述符表 GDT 和中断描述符表 IDT 包含着系统的重要数据，对应的两个描述符表寄存器 GDTR 和 IDTR 含有这两张表的定位信息。利用存储描述符表寄存器指令能把描述符表寄存器的内容保存到指定的存储单元。这样，访问这些存储单元就可获得描述符表的定位信息。与 GDT 和 IDT 被所有任务共享不同，LDT 是每个任务私有，所以存储局部描述符表寄存器 LDTR 的指令不在所列。

(1) 存储全局描述符表寄存器指令

存储全局描述符表寄存器指令的格式如下：

```
SGDT    DST
```

其中，操作数 DST 是 48 位(6 字节)的存储器操作数。该指令的功能是把全局描述符表寄存器 GDTR 的内容存储到存储单元 DST。GDTR 中的 16 位界限存入 DST 的低字，GDTR 中的 32 位基地址存入 DST 的高双字。

该指令对标志没有影响。

例如：如下指令把 GDTR 保存到由 BX 所指向的存储单元中：

```
SGDT    [BX]
```

(2) 存储中断描述符表寄存器指令

存储中断描述符表寄存器指令的格式如下：

```
SIDT    DST
```

其中，操作数 DST 是 48 位(6 字节)的存储器操作数。该指令的功能是把中断描述符表寄存器 IDTR 的内容存储到存储单元 DST。IDTR 中的 16 位界限存入 DST 的低字，IDTR 中的 32 位基地址存入 DST 的高双字。该指令与上面的存储全局描述符表寄存器指令 SGDT 相似。

该指令对标志没有影响。

例如：如下指令把 IDTR 保存到由 EDX 所指向的存储单元中：

```
SIDT    [EDX]
```

2. 存储机器状态字指令

存储机器状态字指令的格式如下：

```
SMSW    DST
```

其中，操作数 DST 可以是 16 位通用寄存器或存储单元。该指令的功能是把机器状态字存储到 DST。

该指令对标志没有影响。

例如：如下指令把机器状态字存储到 AX：

```
SMSW    AX
```

80386有此指令是为了包含80286的指令集。由于80386的控制寄存器CR0的低16位等同于80286的机器状态字，所以在为80386编程时，如果需要存储机器状态字，那么应该使用存储CR0寄存器的指令。

10.8.2 实方式及特权级0下可执行的指令

下列指令涉及设置最关键的寄存器，所以只能在实方式和保护方式的特权级0下执行。为了从初始时的实方式转入保护方式，必须作基本的准备工作。例如，设置妥全局描述符表寄存器GDTR等。这是允许下列指令在实方式下执行的原因。

在保护方式下，如当前特权级(CPL)不为0，执行这些指令将引起错误码为0的通用保护故障。在虚拟8086方式下，因为CPL为3，所以执行这些指令也就会引起错误码为0的通用保护故障。

1. 清任务切换标志指令

每当任务切换时，控制寄存器CR0中的任务切换标志TS被自动置1。这样安排的原因已在前面说明过。

清任务切换标志指令的功能是把TS标志清0。该指令的格式如下：

```
CLTS
```

该指令仅影响TS标志，对其他标志没有影响。

2. 暂停指令

暂停指令的格式如下：

```
HLT
```

该指令使处理机暂停执行。暂停之后的系统，只有在接受一个已经启用的中断，或者让系统复位，才能重新启动。

该指令对标志没有影响。

3. 装载全局描述符表和中断描述符表寄存器的指令

(1) 装载全局描述符表寄存器指令

装载全局描述符表寄存器指令的格式如下：

```
LGDT    SRC
```

其中，操作数SRC是48位(6字节)的存储器操作数。该指令的功能是把存储器中的伪描述符装入到全局描述符表寄存器GDTR。伪描述符SRC的结构如前述结构类型PDESC所示，低字是以字节为单位的段界限，高双字是段基地址。

该指令对标志没有影响。

例如：如下指令从BX所指向的存储单元中装载GDTR：

```
LGDT    [BX]
```

在前几节所举实例中，已多次使用了该指令。

(2)装载中断描述符表寄存器指令

装载中断描述符表寄存器指令的格式如下：

```
LIDT     SRC
```

其中，操作数 SRC 是 48 位(6 字节)的存储器操作数。该指令的功能是把存储器中的伪描述符装入到中断描述符表寄存器 IDTR。

该指令的格式与功能与装载全局描述符表寄存器指令的格式与功能类似。请参见实例六和七。

4. 装载机器状态字指令

装载机器状态字指令的格式如下：

```
LMSW     SRC
```

其中，操作数 SRC 可以是 16 位通用寄存器或存储单元。该指令的功能是将 SRC 装入机器状态字(也就是 CR0 的低 16 位)。

该指令对标志寄存器中的标志没有影响。

例如：如下指令把 AX 的内容装载到机器状态字：

```
LMSW     AX
```

将 PE 位置 1，便进入保护方式。在 80286 中，没有控制寄存器，为进入保护方式需要通过该指令把 MSW 中的 PE 位置 1。如果的确是这样，那么在 LMSW 指令后面必须紧跟一条转移指令。

80386 提供此指令是为了包含 80286 的指令集。由于 80386 的控制寄存器 CR0 的低 16 位等同于 80286 的机器状态字，所以在为 80386 编程时，如果需要装载机器状态字，那么应该使用控制寄存器传送指令。

5. 控制寄存器数据传送指令

控制寄存器数据传送指令的一般格式如下：

```
MOV     DST,SRC
```

控制寄存器数据传送指令实现 80386 的控制寄存器和 32 位通用寄存器之间的数据传送。所以，操作数 SRC 和 DST 可以是 80386 使用的三个控制寄存器和任一 32 位通用寄存器，但不能同时是控制寄存器。

该指令对标志没有影响。

例如：

```
MOV     EAX,CR2     ;把 CR2 送 EAX
MOV     CR0,EAX     ;把 EAX 送 CR0
```

6. 调试寄存器数据传送指令

调试寄存器数据传送指令的一般格式与上面的控制寄存器数据传送指令的格式相同。功能是实现 80386 的调试寄存器和 32 位通用寄存器之间的数据传送。操作数 SRC 和 DST 可以是 80386 使用的 6 个调试寄存器和任一 32 位通用寄存器，但不能同时是调试

寄存器。

80386 可使用的 6 个调试寄存器记为：DR0、DR1、DR2、DR3、DR6 和 DR7。

其他说明与控制寄存器数据传送指令相同。

7. 测试寄存器数据传送指令

测试寄存器数据传送指令的一般格式与上面的控制寄存器数据传送指令的格式相同。功能是实现 80386 的测试寄存器和 32 位通用寄存器之间的数据传送。80386 使用的 2 个测试寄存器是 TR6 和 TR7。其他说明与控制寄存器数据传送指令相同。

10.8.3 只能在保护方式下执行的指令

下面介绍的指令只能在保护方式下执行。如果在实方式下执行这些指令，将引起非法操作码故障（向量号 6）。

1. 装载和存储局部描述符表寄存器指令

(1) 装载局部描述符表寄存器指令

装载局部描述符表寄存器指令的格式如下：

```
LLDT    SRC
```

其中，操作数 SRC 可以是 16 位通用寄存器或存储单元。该指令的功能是把 SRC 中的内容作为指示局部描述符表 LDT 的选择子装入到 LDTR。

该指令不影响标志。

例如：如下指令把 CX 的内容作为指向 LDT 的选择子装入 LDTR：

```
LLDT    CX
```

操作数 SRC 给定的选择子应该指示 GDT 中的类型为 LDT 的描述符。但 SRC 也可以是一个空的选择子，如果这样的话，表示暂时不使用局部描述符表 LDT。

若 CPL 不为 0，那么执行该指令将产生错误码为 0 的通用保护故障。若被装载的选择子不指示 GDT 中描述符，或者描述符类型不是 LDT 描述符，那么产生通用保护故障，错误码由该选择子构成。

从 10.3 节可知，象段寄存器那样，LDTR 也有两部分。在把指示 LDT 的选择子装入到 LDTR 可见部分时，处理器自动把选择子所索引的 LDT 描述符中的段基地址等信息保存到不可见的高速缓冲寄存器中。

(2) 存储局部描述符表寄存器指令

存储局部描述符表寄存器指令的格式如下：

```
SLDT    DST
```

其中，操作数 DST 可以是 16 位通用寄存器或存储单元。该指令的功能是把局部描述符表寄存器 LDTR 的内容存储到存储单元 DST，也就是把指向当前任务 LDT 的选择子存储到 DST。

例如：如下指令把局部描述符表寄存器 LDTR 的内容保存到寄存器 DX：

```
SLDT    DX
```

2. 装载和存储任务寄存器指令

任务寄存器 TR 指示当前任务的任务状态段 TSS。随着任务的切换，TR 的内容也随之改变；如果任务嵌套，那么 TR 的原值作为链接字保存到新任务的 TSS 中。但有时候需要直接地装载 TR，或者保存 TR，这就需要使用装载 TR 指令和存储 TR 指令。

（1）装载任务寄存器指令

装载任务寄存器指令的格式如下：

```
LTR    SRC
```

其中，操作数 SRC 可以是 16 位通用寄存器或存储单元。该指令的功能是将 SRC 作为指示 TSS 描述符的选择子装载到任务寄存器 TR。从 10.3 节可知，象 LDTR 那样，TR 也有两部分。在把指示 TSS 的选择子装入到 TR 可见部分时，处理器自动把选择子所索引的描述符中的段基地址等信息保存到不可见的高速缓冲寄存器中。所以，SRC 表示的选择子不能为空，必须索引位于 GDT 中的描述符，并且描述符的类型必须是 TSS。

该指令对标志没有影响。

例如：如下指令把 AX 的内容作为指示 TSS 描述符的选择子装载到 TR：

```
LTR    AX
```

若 CPL 不为 0，那么执行该指令将产生错误码为 0 的通用保护故障。若被加载的选择子不指示 GDT 中的 TSS 描述符，那么产生通用保护故障，错误码由该选择子构成。

（2）存储任务寄存器指令

存储任务寄存器指令的格式如下：

```
STR    DST
```

其中，操作数 DST 可以是 16 位通用寄存器或存储单元。该指令的功能是把 TR 所含的指示当前任务 TSS 描述符的选择子存储到 DST。

该指令对标志没有影响。

例如：如下指令把 TR 的内容存储到 AX 中：

```
STR    AX
```

3. 调整申请特权级指令

调整申请特权级指令的格式如下：

```
ARPL    OPRD1,OPRD2
```

其中，操作数 OPRD1 可以是 16 位通用寄存器或存储单元，操作数 OPRD2 是 16 位通用寄存器。该指令把操作数 OPRD1 和 OPRD2 视为两个选择子，用 OPRD2 的申请特权级(RPL)去检查 OPRD1 的 RPL。选择子 OPRD1 和 OPRD2 的 RPL 分别由它们的最低 2 个位规定。如果 OPRD1 的 RPL 小于 OPRD2 的 RPL，那么零标志 ZF 被置 1，并把 OPRD2 的 RPL 值赋予 OPRD1 的 RPL(使它们最低 2 位相等)；否则，ZF 被清 0。

OPRD1 和 OPRD2 都可以是空的选择子。

该指令影响 ZF，不影响其他标志。

请参见本节的示例程序。

4. 装载存取权指令

装载存取权指令的格式如下：

```
LAR    OPRD1,OPRD2
```

其中，操作数 OPRD1 可以是 16 位通用寄存器或 32 位通用寄存器；操作数 OPRD2 可以是 16 位通用寄存器或存储单元，也可以是 32 位通用寄存器或存储单元。操作数 OPRD1 和操作数 OPRD2 的尺寸应该一致。该指令把操作数 OPRD2 视为选择子（当 32 位时仅使用低 16 位），如果 OPRD2 所指示的描述符满足如下条件，那么零标志 ZF 被置 1，并把描述符内的属性字段装入 OPRD1；否则，ZF 被清 0，OPRD1 不变。

① 在描述符表的范围内；

② 是存储段描述符或系统段描述符，或者任务门描述符和调用门描述符；

③ CPL 和 OPRD2 的 RPL 都不大于 DPL。

在满足条件的情况下，装入到 OPRD1 的由 OPRD2 所指示的描述符中的属性字段是指描述符的高 4 字节和 00FXFF00H 相与的结果，其中 X 表示第 16 位到第 19 位无定义。注意，如果指令使用 16 位操作数，那么只有上述高 4 字节中的低字被装入到 OPRD1，也即装入到 OPRD1 的属性字段不包括 G 位和 AVL 位等。参见本节给出的示例程序。

该指令影响 ZF，不影响其他标志。

5. 装载段界限指令

装载界限指令的格式如下：

```
LSL    OPRD1,OPRD2
```

其中，操作数 OPRD1 可以是 16 位通用寄存器或 32 位通用寄存器；操作数 OPRD2 可以是 16 位通用寄存器或存储单元，也可以是 32 位通用寄存器或存储单元。操作数 OPRD1 和操作数 OPRD2 的尺寸应该一致。该指令把操作数 OPRD2 视为选择子（当 32 位时仅使用低 16 位），如果 OPRD2 所指示的描述符满足如下条件，那么零标志 ZF 被置 1，并把描述符内的界限字段装入 OPRD1；否则，ZF 被清 0，OPRD1 不变。

① 在描述符表的范围内；

② 是存储段描述符或系统段描述符，而非门描述符；

③ CPL 和 OPRD2 的 RPL 都不大于 DPL。

在满足条件的情况下，装入到 OPRD1 的由 OPRD2 所指示的描述符中的界限字段以字节为单位。如果描述符中的界限字段以 4K 字节为单位（G=1），那么在装入到 OPRD1 时被左移 12 位，空出的低位全部填成 1。注意，如果指令使用 16 位操作数，那么只有段界限的低 16 位被装入到 OPRD1。参见本节给出的示例程序。

该指令影响 ZF，不影响其他标志。

6. 读写检验指令

利用读检验指令和写检验指令可分别检查在当前特权级上指定的段能否读或写，从而避免引起不必要的异常。

（1）读检验指令

读检验指令的一般格式如下：

```
VERR    OPRD
```

其中，操作数 OPRD 可以是 16 位通用寄存器或存储单元，也可以是 32 位通用寄存器或存储单元。该指令的功能是把 OPRD 的内容作为一个选择子(当 32 位时仅使用低 16 位)，判断在当前特权级上该选择子所指示的段是否能读。如果该选择子指示合法的一个存储段描述符，并且在当前特权级上可写所描述的段，那么零标志 ZF 被置 1，否则 ZF 被清 0。

该指令只影响零标志 ZF，而不影响其他标志。

(2) 写检验指令

写检验指令的一般格式如下：

```
VERW    OPRD
```

其中，操作数 OPRD 可以是 16 位通用寄存器或存储单元，也可以是 32 位通用寄存器或存储单元。该指令的功能是把 OPRD 的内容作为一个选择子(当 32 位时仅使用低 16 位)，判断在当前特权级上该选择子所指示的段是否能写。如果该选择子指示合法的一个存储段描述符，并且在当前特权级上可读所描述的段，那么零标志 ZF 被置 1，否则 ZF 被清 0。

该指令只影响零标志 ZF，而不影响其他标志。

10.8.4 显示关键寄存器内容的实例(实例八)

下面结合说明部分操作系统类指令的使用，给出一个显示 80386 关键寄存器内容的实例。该实例的逻辑功能，是显示系统中 GDTR、IDTR、LDTR 和 DR 等关键寄存器的当前内容。

实例八源程序清单如下：

```
;程序名：T10-8.ASM
;功  能：显示关键寄存器内容兼说明操作系统类指令的使用
;
        INCLUDE     386SCD.ASM          ;参见实例三
;
        .386P
;-------------------------------------------------------------
;全局描述符表 GDT
GDTSEG      SEGMENT   PARA   USE16
GDT         LABEL BYTE
DUMMY       DESCRIPTOR   <>
NORMAL      DESCRIPTOR   <0FFFFH,0,0,ATDW,0>
Normal_SEL      =     NORMAL_GDT
EFFGDT      LABEL BYTE
;临时代码段描述符
```

```
TEMPCODE     DESCRIPTOR   <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL       =      TEMPCODE_GDT
;缓冲区段描述符
BUFFER       DESCRIPTOR   <BufferLEN-1,BufferSEG,,ATDW,>
Buffer_SEL         =      BUFFER_GDT
;测试描述符1
TEST1        DESCRIPTOR   <1111H,0,,8792H,>
Test1_SEL          =      TEST1_GDT
TestR_SEL          =      TEST1_GDT + RPL3
;测试描述符2
TEST2        DESCRIPTOR   <2222H,0,,1782H,>
Test2_SEL      =      TEST2_GDT
GDNUM          =      ($-EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN         =      $-GDT
GDTSEG       ENDS
;----------------------------------------------------------------
;缓冲区段
BufferSEG    SEGMENT   PARA   USE16
GDTR_V       PDESC    <>              ;存放GDTR
IDTR_V       PDESC    <>              ;存放IDTR
MSW_V        DW       ?               ;存放机器状态字
LDTR_V       DW       ?               ;存放LDTR选择子
TR_V         DW       ?               ;存放TR选择子
CR0_V        DD       ?               ;存放控制寄存器CR0
CR3_V        DD       ?               ;存放控制寄存器CR3
DR7_V        DD       ?               ;存放调试寄存器DR7
TEST_RPL     DW       ?
;
TEST1_SLD    DD       ?               ;演示用变量
TEST1_ARD    DD       ?
TEST1_SLW    DW       ?
TEST1_ARW    DW       ?
TEST1_RF     DW       ?
TEST1_WF     DW       ?
;
TEST2_SLD    DD       ?               ;演示用变量
TEST2_ARD    DD       ?
TEST2_SLW    DW       ?
TEST2_ARW    DW       ?
TEST2_RF     DW       ?
TEST2_WF     DW       ?
```

```
BufferLEN     =        $
BufferSEG     ENDS
;----------------------------------------------------------------
;临时代码段
TempCodeSEG  SEGMENT    PARA   USE16
           ASSUME CS : TempCodeSEG,DS : BufferSEG
Virtual:
           MOV     AX,Buffer_SEL
           MOV     DS,AX
           MOV     EAX,CR0              ;存储 CR0
           MOV     CR0_V,EAX
           MOV     EAX,CR3              ;存储 CR3
           MOV     CR3_V,EAX
           MOV     EAX,DR7              ;存储 DR7
           MOV     DR7_V,EAX
           STR     TR_V                 ;存储 TR
           SLDT    LDTR_V               ;存储 LDTR
           ;
           MOV     TEST_RPL,Test1_SEL
           MOV     AX,TestR_SEL
           ARPL    TEST_RPL,AX          ;说明调整申请特权级指令
           ;
           MOV     BX,0
           MOV     AX,TEST1_SEL
LAB1:      MOV     EDX,0
           MOV     CX,0
           LSL     EDX,EAX              ;说明装载段界限指令
           LSL     CX,AX
           MOV     TEST1_SLD[BX],EDX
           MOV     TEST1_SLW[BX],CX
           MOV     EDX,0
           MOV     CX,0
           LAR     EDX,EAX              ;说明装载存取权指令
           LAR     CX,AX
           MOV     TEST1_ARD[BX],EDX
           MOV     TEST1_ARW[BX],CX
           MOV     TEST1_RF[BX],0
           VERR    AX                   ;说明读检验指令
           JNZ     LAB2
           MOV     TEST1_RF[BX],1
LAB2:      MOV     TEST1_WF[BX],0
           VERW    AX                   ;说明写检验指令
```

```
            JNZ      LAB3
            MOV      TEST1_WF[BX],1
LAB3:       ADD      BX,16
            MOV      AX,TEST2_SEL
            CMP      BX,32
            JB       LAB1
Over:       ; 准备返回实方式
            MOV      AX,Normal_SEL
            MOV      DS,AX
            MOV      EAX,CR0
            AND      EAX,0FFFFFFFEH
            MOV      CR0,EAX                ;返回实方式
            JUMP16   <SEG Real>,<OFFSET Real>
TempCodeSEG     ENDS
;-----------------------------
;实方式下的数据段和代码段
RCodeSEG     SEGMENT     PARA   USE16
VGDTR       PDESC  <GDTLEN-1,>
            ASSUME      CS : RCodeSEG,DS : BufferSEG
Start:      ;实方式
            MOV      AX,BufferSEG
            MOV      DS,AX
            SGDT     GDTR_V                 ;存储 GDTR
            SIDT     IDTR_V                 ;存储 IDTR
            SMSW     MSW_V                  ;存储机器状态字
            ;准备转入保护方式
            PUSH     CS
            POP      DS
            CLD
            CALL     INIT_GDT
            MOV      BX,OFFSET VGDTR
            LGDT     [BX]
            CLI
            MOV      EAX,CR0
            OR       EAX,1
            ;转入保护方式
            MOV      CR0,EAX
            JUMP16     <TempCode_SEL>,<OFFSET Virtual>
Real:       ; 回到实方式
            STI
            ; (略去显示相关变量内容的部分代码)
            MOV      AX,4C00H
```

```
        INT     21H
;-------------------------------
;实方式下的初始化过程
INIT_GDT    PROC   NEAR
            ;同实例四中的过程内容
INIT_GDT    ENDP
RCodeSEG    ENDS
            END     Start
```

10.8.5 特权指令

所谓特权指令是指保护方式下只有当前特权级 CPL=0 时，才可以执行的指令。如果 CPL 不等于 0 而执行它们，那么会引起通用保护异常。从上面介绍的操作系统类指令可归纳出如表 10.8 所列的 80386 特权指令。这些特权指令在构成完善的保护机制方面起了重要作用。

表 10.8 特权指令

指令	功能	指令	功能
CLTS	清除 CR0 中的 TS 位	LTR	装入 TR
HLT	停机	MOV CRn,reg	装入控制寄存器
LGDT	装入 GDTR	MOV reg,CRn	保存控制寄存器
LIDT	装入 IDTR	MOV DRn,reg	装入调试寄存器
LLDT	装入 LDTR	MOV reg,DRn	保存调试寄存器
LMSW	装入 MSW(CR0 低 16 位)		

从表 10.8 可见，装入 GDTR、IDTR、LDTR、TR 和 MSW 的指令都是特权指令，而存储上述寄存器的指令不是特权指令。这表示，保护方式下任何程序可获取这些寄存器的值，但只有具有特权级 0 的程序才能够改变这些寄存器的值。从表 10.8 还可见，设置和存储控制寄存器及调试寄存器的指令都是特权指令。

10.9 输入/输出保护

为了支持多任务，80386 不仅要有效地实现任务隔离，而且还要有效地控制各任务的输入/输出，避免输入/输出冲突。本节介绍输入/输出保护。

10.9.1 输入/输出保护

80386 采用 I/O 特权级 IOPL 和 I/O 许可位图的方法来控制输入/输出，实现输入/输出的保护。

1. I/O 敏感指令

输入/输出特权级(I/O Privilege Level)规定了可以执行所有与 I/O 相关的指令和访问 I/O 空间中所有地址的最外层特权级。IOPL 值在如图 9.2 所示的标志寄存器中。I/O

许可位图规定了 I/O 空间中的哪些地址可以由在任何特权级执行的程序所访问。I/O 许可位图在任务状态段 TSS 中。

表 10.9 I/O 敏感指令

指令	功能	保护方式下执行条件
CLI	清除 EFLAGS 中的 IF 位	CPL<= IOPL
STI	设置 EFLAGS 中的 IF 位	CPL<= IOPL
IN	从 I/O 地址读出数据	CPL<= IOPL 或 I/O 位图允许
INS	从 I/O 地址读出字符串	CPL<= IOPL 或 I/O 位图允许
OUT	向 I/O 地址写数据	CPL<= IOPL 或 I/O 位图允许
OUTS	向 I/O 地址写字符串	CPL<= IOPL 或 I/O 位图允许

表 10.9 所列的指令称为 I/O 敏感指令。由于这些指令与 I/O 有关，并且只有在满足所列条件时才可以执行，所以把它们称为 I/O 敏感指令。从表 10.9 可见，当前特权级不在 I/O 特权级外层时，可以正常执行所列的全部 I/O 敏感指令；当前特权级在 I/O 特权级外层时，执行 CLI 和 STI 指令将引起通用保护异常；当前特权级在 I/O 特权级外层时，其他四条输入/输出指令是否能够执行要根据访问的 I/O 地址及 I/O 许可位图情况而定（在下面细述），如果条件不满足而执行，那么将引起通用保护异常。

由于每个任务使用各自的 EFLAGS 值和拥有自己的 TSS，所以每个任务可以有不同的 IOPL，并且可以定义不同的 I/O 许可位图。

注意，这些 I/O 敏感指令在实方式下总是可执行的。

2. I/O 许可位图

如果只用 IOPL 限制 I/O 指令的执行是很不方便的，不能满足实际需要。因为这样做会使得在特权级 3 执行的应用程序，要么可访问所有 I/O 地址，要么不可访问所有 I/O 地址。实际需要与此刚好相反，只允许任务甲的应用程序访问部分 I/O 地址，只允许任务乙的应用程序访问另一部分的 I/O 地址，以避免任务甲和任务乙在访问 I/O 地址时发生冲突，从而避免任务甲和任务乙使用独享设备时发生冲突。

所以，在 IOPL 的基础上，再采用了 I/O 许可位图。I/O 许可位图由二进制位串组成。位串中的每一位依次对应一个 I/O 地址，位串的第 0 位，对应 I/O 地址 0，位串的第 n 位对应 I/O 地址 n。如果位串中的 m 位为 0，那么对应的 I/O 地址 m 可以由在任何特权级执行的程序访问；否则对应的 I/O 地址 m 只能由在 IOPL 特权级或更内层特权级执行的程序访问。如果在 IOPL 外层特权级执行的程序访问位串中位值为 1 的位所对应的 I/O 地址，那么引起通用保护异常。

I/O 地址空间按字节进行编址。一条 I/O 指令最多可涉及四个 I/O 地址。例如，如下指令读取 I/O 端口 71H～74H 内的四个字节至 EAX：

```
IN    EAX,71H
```

在需要根据 I/O 位图决定是否可访问 I/O 地址的情况下，当一条 I/O 指令涉及多个 I/O 地址时，只有这多个 I/O 地址所对应的 I/O 位图中的位都为 0 时，该 I/O 指令才能正

常执行；如果对应位中任一位为 1，就会引起通用保护异常。

80386 支持的 I/O 地址空间大小是 64K，所以构成 I/O 许可位图的二进制位串最大长度是 64K 个位，也即 I/O 许可位图有效部分最大是 8K 字节。一个任务实际需要使用的 I/O 许可位图大小通常要远小于这个数目。

当前任务使用的 I/O 许可位图存储在当前任务 TSS 中低端的 64K 字节内。I/O 许可位图总以字节为单位存储，所以位串所含的位数总被认为是 8 的倍数。从如图 10.14 所示的 TSS 格式可见，TSS 内偏移 66H 的字确定 I/O 许可位图的开始偏移。由于 I/O 许可位图最长可达 8K 字节，所以开始偏移应该小于 56K。

3. I/O 访问许可检查细节

保护方式下处理器在执行 I/O 指令时进行许可检查的细节如图 10.25 所示。图中“字节偏移”是 I/O 指令访问的 I/O 地址对应位所在字节在 I/O 许可位图内的偏移，把 I/O 地址右移 3 位就得该字节偏移；“位偏移”是上述对应位在字节内的位数，根据它计算出屏蔽码字。把字节偏移加上位图开始偏移，再加上 1，所得值与 TSS 界限比较，如果越界，那么就引起通用保护异常。如果不越界，那么就从位图中读出对应字节及下一字节，并和屏蔽字进行与运算，结果为 0 表示检查通过可以进行 I/O 访问，结果非 0 就引起通用保护异常。

设任务甲的 TSS 段如下：

```
TSSSEGA   SEGMENT  PARA  USE16
          TASKSS  < >                   ;TSS 低端固定格式部分
          DB  8  DUP (0)                ;对应 I/O 端口 0～3FH
          DB  10000000B                 ;对应 I/O 端口 40H～47H
          DB  01100000B                 ;对应 I/O 端口 48H～4FH
          DB  8182  DUP (0FFH)          ;对应 I/O 端口 50H～FFFFH
          DB  0FFH                      ;位图结束字节
TSSLENA   =    $
TSSSEGA    ENDS
```

再假设 IOPL＝1，CPL＝3。那么如下 I/O 指令有些能正常执行，有些会引起通用保护异常：

```
IN      AL,21H          ;(1) 正常执行
IN      AL,47H          ;(2) 引起异常
OUT     20H,AL          ;(3) 正常执行
OUT     4EH,AL          ;(4) 引起异常
IN      AX,20H          ;(5) 正常执行
OUT     20H,EAX         ;(6) 正常执行
OUT     4CH,AX          ;(7) 引起异常
IN      AX,46H          ;(8) 引起异常
IN      EAX,42H         ;(9) 正常执行
```

从图 10.25 所示的细节可见，不论是否有必要，当进行许可位检查时，80386 总是从

I/O 许可位图中读取两个字节。目的是为了尽快地进行 I/O 许可检查。一方面，常常要读取 I/O 许可位图中的两个字节。例如：上面的第(8)条指令要对 I/O 位图中的两个位进行检查，其低位是某字节的最高位，高位是下一字节的最低位。可见即使只要检查两个位，也可能需要读取两个字节。另一方面，最多检查四个连续的位，也即最多也只需读取两个字节。所以每次都读取两个字节。这也是在判别是否越界时再加 1 的缘故。为此，为了避免在读取 I/O 许可位图内的最高字节时产生越界，必须在 I/O 许可位图的最后添加一个全 1 的字节。

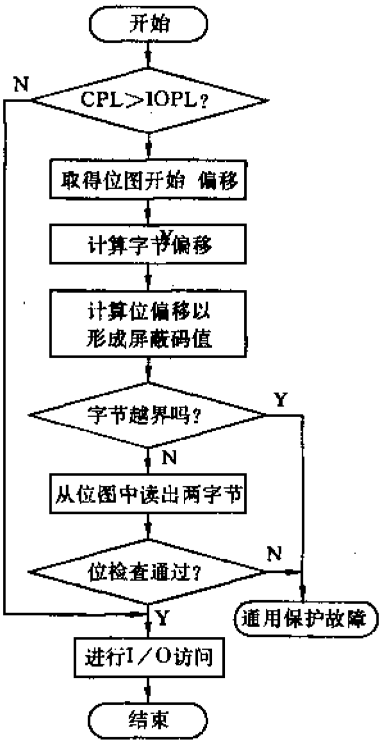


图 10.25 I/O 许可检查细节

从图 10.25 及其关于字节越界的说明可知，I/O 许可位图开始偏移加 8K 所得值与 TSS 界限值二者中较小的值决定 I/O 许可位图的末端。当 TSS 的界限大于 I/O 许可位图开始偏移加 8K 时，I/O 许可位图有效部分就有 8K 字节，I/O 许可检查全部根据该位图进行。当 TSS 的界限不大于 I/O 许可位图开始偏移加 8K 时，I/O 许可位图有效部分就不到 8K 字节，于是对较小 I/O 地址访问的许可检查根据位图进行，而对较大 I/O 地址访问的许可检查总被认为不可访问而引起通用保护异常，因为这时会发生字节越界而引起通用保护异常，所以在这种情况下，可认为不足的 I/O 许可位图的高端部分是全 1。利用这一特点，可大大节约 TSS 中 I/O 许可位图占用的存储单元，也就大大减小了 TSS。

设任务乙的 TSS 段如下：

```
TSSSEGB     SEGMENT     PARA   USE16
            TASKSS        < >                ;TSS 的低端部分
            DB  100H/8  DUP (0FFH)           ;对应 I/O 端口 0～FFH
            DB  100H/8  DUP (0)              ;对应 I/O 端口 100H～1FFH
            DB  0FFH
TSSLENB     =       $
TSSSEGB     ENDS
```

上面任务乙的 TSS 内只含 I/O 许可位图的最初 200H 位，对应 I/O 端口地址 0——1FFH，而其他位都可以被认为是 1。

设任务丙的 TSS 段如下：

```
TSSSEGC     SEGMENT     PARA   USE16
            TASKSS      < >                  ;TSS 的低端部分
            DB  100H/8  DUP (0)              ;对应 I/O 端口 0～FFH
            DB  0FFH
```

```
TSSLENC    =    $
TSSSEGC    ENDS
```

上面任务丙的 TSS 内只含 I/O 许可位图的最初 100H 位，对应 I/O 端口地址 0～0FFH，而其他位都可以被认为是 1。

设任务丁的 TSS 段如下：

```
TSSSEGD    SEGMENT   PARA   USE16
           TASKSS    < >
           DB  0FFH
TSSLEND    =    $
TSSSEGD    ENDS
```

上面任务丁的 TSS 内只有 I/O 许可位图结束字节，而无有效的 I/O 许可位图。

综上所述，使用 I/O 许可位图可以较好地满足实际应用需要。在方便应用程序进行输入/输出的同时，限制输入/输出，避免任务间在使用外设时发生冲突。

10.9.2 重要标志保护

输入/输出的保护与存储在标志寄存器 EFLAGS 中的 IOPL 密切相关，显然不能允许随便地改变 IOPL，否则就不能有效地实现输入/输出保护。类似地，对 EFLAGS 中的 IF 位也必须加以保护，否则 CLI 和 STI 作为敏感指令对待是无意义的。此外，EFLAGS 中的 VM 位决定着处理器是否按虚拟 8086 方式工作。

80386 对 EFLAGS 中的这三个字段的处理比较特殊，只有在较高特权级执行的程序才能执行 IRET、POPF、CLI 和 STI 等指令来改变它们。表 10.10 列出了不同特权级下对这三个字段的处理情况。

表 10.10 不同特权级对 EFLAGS 特殊字段处理

特权级	标志字段		
	VM	IOPL	IF
CPL = 0	可变(除 POPF 指令外)	可变	可变
0< CPL <= IOPL	不变	不变	可变
CPL > IOPL	不变	不变	不变

从表 10.10 可见，只有在特权级 0 执行的程序才可以修改 IOPL 位及 VM 位；只能由相对于 IOPL 同级或更内层特权级执行的程序才可以修改 IF 位。与 CLI 和 STI 指令不同，在特权级不满足上述条件的情况下，当执行 POPF 指令和 IRET 指令时，如果试图修改这些字段中的任何一个字段，并不引起异常，但试图要修改的字段也未被修改，也不给出任何特别的信息。

此外，指令 POPF 总不能改变 VM 位，而 PUSHF 指令总把 0 压入到 VM 位。

10.9.3 演示输入/输出保护的实例(实例九)

下面给出一个用于演示输入/输出保护的实例。演示内容包括：I/O 许可位图的作用、

I/O 敏感指令引起的异常和特权指令引起的异常；使用段间调用指令 CALL 通过任务门调用任务，实现任务嵌套等。

1. 演示步骤

实例演示的内容比较丰富，具体演示步骤如下：

(1) 在实方式下作必要准备后，切换到保护方式。

(2) 进入保护方式的临时代码段后，把演示任务的 TSS 段描述符装入 TR，设置演示任务的堆栈。

(3) 进入演示代码段，演示代码段的特权级是 0。

(4) 通过任务门调用测试任务 1。测试任务 1 能够顺利执行。

(5) 通过任务门调用测试任务 2。测试任务 2 演示由于违反 I/O 许可位图规定而导致通用保护异常。

(6) 通过任务门调用测试任务 3。测试任务 3 演示 I/O 敏感指令如何引起通用保护异常。

(7) 通过任务门调用测试任务 4。测试任务 4 演示特权指令如何引起通用保护异常。

(8) 从演示代码转临时代码，准备返回实方式。

(9) 返回实方式，并作结束处理。

2. 源程序组织和清单

为了达到演示目的，实例除了演示任务外，还涉及四个测试任务和一个通用保护故障处理任务。实例有如下几部分组成：

(1) 全局描述符表 GDT 和中断描述符表 IDT；

(2) 其他中断/异常处理程序代码段；

(3) 通用保护故障处理任务的任务状态段、堆栈段和代码段；

(4) 四个测试任务合用的任务状态段、堆栈段和代码段；

(5) 演示任务的任务状态段、堆栈段和代码段；

(6) 演示任务的临时代码段；

(7) 实方式下执行的启动和结束程序代码段和数据段。

实例九源程序清单如下：

```
;程序名：T10-9.ASM
;功  能：演示 I/O 保护及 I/O 敏感指令的作用
;
        INCLUDE     386SCD.ASM        ;参见实例三说明
;
        .386P
;------------------------------
;全局描述符表 GDT
GDTSEG      SEGMENT   PARA   USE16
GDT         LABEL BYTE
DUMMY       DESCRIPTOR      <>
NORMAL      DESCRIPTOR      <0FFFFH,0,0,ATDW,0>
```

```
Normal_SEL      =    NORMAL - GDT
EFFGDT       LABEL BYTE
;显示缓冲区段描述符(任何特权级可写)
VIDEOBUFF   DESCRIPTOR    <80*25*2-1,0B800H,,ATDW+DPL3,>
VideoBuff_SEL     =    VIDEOBUFF - GDT
;演示任务 TSS 段描述符
DEMOTSS     DESCRIPTOR    <DemoTSSLEN-1,DemoTSSSEG,,AT386TSS,>
DemoTSS_SEL     =    DEMOTSS - GDT
;演示任务堆栈段描述符
DEMOSTACK    DESCRIPTOR <DemoStackLEN-1,DemoStackSEG,,ATDW+D32,>
DemoStack_SEL     =    DEMOSTACK - GDT
;演示任务代码段描述符
DEMOCODE    DESCRIPTOR    <DemoCodeLEN-1,DemoCodeSEG,,ATCE+D32,>
DemoCode_SEL     =     DEMOCODE - GDT
;属于演示任务的临时代码段描述符
TEMPCODE    DESCRIPTOR    <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL     =     TEMPCODE - GDT
;指向 GDT 的存储段描述符
TOGDT        DESCRIPTOR    <GDTLEN-1,GDTSEG,,ATDW,>
ToGDT_SEL       =    TOGDT - GDT
;指向通用保护故障处理任务 TSS 的存储段描述符
TOGPTSS      DESCRIPTOR    <GPTSSLEN-1,GPTSSSEG,,ATDW,>
ToGPTSS_SEL      =    TOGPTSS - GDT
;指向测试任务 TSS 的存储段描述符
TOTESTTSS   DESCRIPTOR    <TestTSSLEN-1,TestTSSSEG,,ATDW,>
ToTestTSS_SEL     =    TOTESTTSS - GDT
;测试任务 TSS 段描述符
TESTTSS      DESCRIPTOR    <TestTSSLEN-1,TestTSSSEG,,AT386TSS,>
TestTSS_SEL      =    TESTTSS - GDT
;测试任务 1 堆栈段描述符(DPL=1)
TEST1STACK      DESCRIPTOR     <TestStackLEN-1,TestStackSEG
                                ,,ATDW+D32+DPL1,>

Test1Stack_SEL     =    (TEST1STACK - GDT) + RPL1
;测试任务 1 代码段描述符(DPL=1)
TEST1CODE   DESCRIPTOR   <TestCodeLEN-1,TestCodeSEG
                          ,,ATCE+D32+DPL1,>
Test1Code_SEL     =    (TEST1CODE - GDT) + RPL1
;测试任务 2 堆栈段描述符(DPL=2)
TEST2STACK      DESCRIPTOR   <TestStackLEN-1,TestStackSEG
                              ,,ATDW+D32+DPL2,>
```

```
Test2Stack_SEL      =   (TEST2STACK - GDT) + RPL2
;测试任务 2 代码段描述符(DPL=2)
TEST2CODE       DESCRIPTOR  <TestCodeLEN-1,TestCodeSEG,
                            ,ATCE+D32+DPL2,>
Test2Code_SEL       =   (TEST2CODE - GDT) + RPL2
;测试任务 3 堆栈段描述符(DPL=3)
TEST3STACK      DESCRIPTOR  <TestStackLEN-1,TestStackSEG
                            ,,ATDW+D32+DPL3,>
Test3Stack_SEL      =   (TEST3STACK - GDT) + RPL3
;测试任务 3 代码段描述符(DPL=3)
TEST3CODE   DESCRIPTOR  <TestCodeLEN-1,TestCodeSEG
                        ,,ATCE+D32+DPL3,>
Test3Code_SEL       =   (TEST3CODE - GDT) + RPL3
;通用保护故障处理任务的 TSS 段描述符
GPTSS       DESCRIPTOR  <GPTSSLEN-1,GPTSSSEG,,AT386TSS,>
GPTSS_SEL       =   GPTSS - GDT
;通用保护故障处理任务的堆栈段描述符
GPSTACK     DESCRIPTOR  <GPStackLEN-1,GPStackSEG,,ATDW+D32,>
GPStack_SEL     =   GPSTACK - GDT
;通用保护故障处理任务的代码段描述符
GPCODE      DESCRIPTOR  <GPCodeLEN-1,GPCodeSEG,,ATCE+D32,>
GPCode_SEL      =   GPCODE - GDT
;其他中断/异常处理程序代码段描述符(一致可读代码段)
ERRCODE     DESCRIPTOR  <ERRCodeLEN-1,ERRCodeSEG,,ATCCOR+D32,>
ERRCode_SEL     = ERRCODE - GDT
GDNUM       =  ($ - EFFGDT)/(SIZE DESCRIPTOR)     ;要初始化的描述符个数
;指向测试任务的任务门
TESTTASK    GATE    <,TestTSS_SEL,0,ATTASKGAT,>
Test_SEL    =   TESTTASK - GDT
GDTLEN      =   $ -GDT
GDTSEG      ENDS
;--------------------------------
;中断描述符表 IDT
IDTSEG      SEGMENT     PARA    USE16
IDT     LABEL   BYTE
        REPT    13
        GATE    <ERRBegin,ERRCode_SEL,0,AT386TGAT,0>
        ENDM
INT0D   GATE    <,GPTSS_SEL,0,ATTASKGAT,>       ; 通用故障处理门描述符
        REPT    254-14
        GATE    <ERRBegin,ERRCode_SEL,0,AT386TGAT,0>
        ENDM
```

```
IDTLEN      =       $ -IDT
IDTSEG    ENDS
;-----------------------------------------------------------
;其他中断/异常处理程序代码段(一致的可读代码段)
ERRCodeSEG       SEGMENT PARA USE32
ERRMESS          DB 'Error ...............'
ERRMESSLEN       =    $ - ERRMESS
        ASSUME  CS:ERRCodeSEG
ERRBegin:
        CLD
        MOV     AX,ERRCode_SEL
        MOV     DS,AX               ;可读代码段
        LEA     ESI,ERRMESS
        MOV     AX,VideoBuff_SEL
        MOV     ES,AX
        XOR     EDI,EDI
        MOV     ECX,ERRMESSLEN
        MOV     AH,17H
ERR1:   LODSB                       ;显示提示信息
        STOSW
        LOOP    ERR1
        JMP     $                   ;循环等待
ERRCodeLEN      =     $
ERRCodeSEG      ENDS
;-----------------------------------------------------------
;通用保护故障处理任务的 TSS 段
GPTSSSEG   SEGMENT  PARA  USE16
GPTaskSS     LABEL  BYTE
             DD     0                   ;任务嵌套时的链接指针
             DD     ?                   ;0 级堆栈指针
             DW     ?,?
             DD     ?                   ;1 级堆栈指针
             DW     ?,?
             DD     ?                   ;2 级堆栈指针
             DW     ?,?
             DD     0                   ;CR3
             DW     GPBegin,0           ;EIP
             DD     0                   ;EFLAGS
             DD     0                   ;EAX
             DD     0                   ;ECX
             DD     0                   ;EDX
             DD     0                   ;EBX
```

```
                DW      GPStackLEN,0            ; ESP
                DD      0                       ; EBP
                DD      0                       ; ESI
                DD      0                       ; EDI
                DW      VideoBuff_SEL,0         ; ES
                DW      GPCode_SEL,0            ; CS
                DW      GPStack_SEL,0           ; SS
                DW      ToTestTSS_SEL,0         ; DS
                DW      ToGPTSS_SEL,0           ; FS
                DW      0,0                     ; GS
                DW      0,0                     ; LDT
                DW      0
                DW      $+2                     ; 指向I/O许可位图区的指针
                DB      0FFH                    ; I/O许可位图结束标志
GPTSSLEN    =   $
GPTSSSEG    ENDS
;-------------------------------
;通用保护故障处理任务的堆栈段
GPStackSEG  SEGMENT  PARA  USE32
GPStackLEN  =   512
            DB  GPStackLEN  DUP (0)
GPStackSEG  ENDS
;-------------------------------
;通用保护故障处理任务的代码段
GPCodeSEG   SEGMENT  PARA  USE32
        ASSUME  CS : GPCodeSEG
GPStart:  ;在屏幕左上角显示故障点
        XOR     EDI,EDI
        MOV     EBX,OFFSET  TestTaskSS
        MOV     EDX,DWORD  PTR  [EBX].TRCS
        CALL    EchoEDX
        MOV     AX,(17H  SHL  8)+':'
        STOSW
        MOV     EDX,[EBX].TREIP
        CALL    EchoEDX
        ;延时以便看清故障点
        MOV     ECX,1234567H
        LOOP    $
        ;调整任务链接指针,终止故障任务
        MOV     EBX,OFFSET GPTaskSS
        MOV     AX,DemoTSS_SEL
        MOV     FS:[EBX].TRLINK,AX
```

```
        IRETD
GPBegin: ;通用故障处理任务开始点
        JMP     GPStart
;显示 EDX 内容的子程序
EchoEDX PROC
        MOV     AH,17H
        MOV     ECX,8
EchoEDX1:
        ROL     EDX,4
        MOV     AL,DL
        CALL    HTOASC
        STOSW
        LOOP    EchoEDX1
        RET
EchoEDX     ENDP
;把 4 位二进制数转换成对应 ASCII 码
HTOASC      PROC
            ;内容略
HTOASC      ENDP
GPCodeLEN   =   $
GPCodeSEG   ENDS
;----------------------------------------------------------------
;测试任务的 TSS 段
TestTSSSEG  SEGMENT   PARA  USE16
TestTaskSS  TASKSS  <>                      ; TSS 的固定格式部分
IOMAP   LABEL   BYTE                        ; I/O 许可位图
        DB      8  DUP (0)                  ; 端口 0000H~003FH
        DB      11000100B                   ; 端口 0040H~0047H
        DB      00111011B                   ; 端口 0048H~004FH
        DB      0,0                         ; 端口 0050H~005FH
        DB      11110010B                   ; 端口 0060H~0067H
        DB      0                           ; 端口 0068H~006FH
        DB      0FFH                        ; I/O 许可位图结束标志
TestTSSLEN      =   $
TestTSSSEG    ENDS
;------------------------------
;测试任务的堆栈段
TestStackSEG        SEGMENT  PARA  USE32
TestStackLEN        =   1024
                    DB    TestStackLEN   DUP (0)
TestStackSEG        ENDS
;------------------------------
```

```
;测试任务的代码段
TestCodeSEG      SEGMENT   PARA   USE32
          ASSUME   CS:TestCodeSEG
Test3Begin:
          CLI                                  ;I/O敏感指令
          CLTS                                 ;特权指令
          IRETD
TestBegin:
          MOV      AL,0B6H                     ;使扬声器发出一长声
          OUT      43H,AL
          MOV      AL,2
          OUT      42H,AL
          MOV      AL,34H
          OUT      42H,AL
          IN       AL,61H
          MOV      AH,AL
          OR       AL,3
          OUT      61H,AL
          MOV      ECX,1234567H
          LOOP     $                           ;延时
          MOV      AL,AH
          OUT      61H,AL
          IRETD
TestCodeLEN      =     $
TestCodeSEG      ENDS
;--------------------------------
;演示任务的TSS段
DemoTSSSEG       SEGMENT   PARA   USE16
DemoTaskSS       TASKSS    <>
                 DB  0FFH
DemoTSSLEN       =     $
DemoTSSSEG       ENDS
;--------------------------------
;演示任务的堆栈段
DemoStackSEG     SEGMENT   PARA   USE32
DemoStackLEN     =   1024
                 DB      DemoStackLEN   DUP (0)
DemoStackSEG     ENDS
;--------------------------------
;演示任务的代码段
DemoCodeSEG      SEGMENT   PARA   USE32
          ASSUME   CS:DemoCodeSEG
```

```
DemoBegin:
        MOV     AX,ToTestTSS_SEL
        MOV     DS,AX
        MOV     EBX,OFFSET TestTaskSS
        ;把测试任务1的入口点、堆栈指针和标志值(含IOPL)填入测试任务TSS
        MOV     WORD  PTR  [EBX].TRSS,Test1Stack_SEL
        MOV     DWORD  PTR  [EBX].TRESP,TestStackLEN
        MOV     WORD  PTR  [EBX].TRCS,Test1Code_SEL
        MOV     DWORD  PTR  [EBX].TREIP,OFFSET TestBegin
        MOV     DWORD  PTR  [EBX].TREFLAG,IOPL1
        ;通过任务门调用测试任务
        CALL32  Test_SEL,0
        ;
        ;把测试任务2的入口点、堆栈指针和标志值(含IOPL)填入测试任务TSS
        MOV     WORD  PTR  [EBX].TRSS,Test2Stack_SEL
        MOV     DWORD  PTR  [EBX].TRESP,TestStackLEN
        MOV     WORD  PTR  [EBX].TRCS,Test2Code_SEL
        MOV     DWORD  PTR  [EBX].TREIP,OFFSET TestBegin
        MOV     DWORD  PTR  [EBX].TREFLAG,IOPL1
        ;通过任务门调用测试任务
        CALL32  Test_SEL,0
        ;
        ;把测试任务TSS描述符内的属性置为"可用"
        MOV     AX,ToGDT_SEL
        MOV     FS,AX
        MOV     FS:TESTTSS.ATTRIBUTES,AT386TSS
        ;把测试任务3的入口点、堆栈指针和标志值(含IOPL)填入测试任务TSS
        MOV     WORD  PTR  [EBX].TRSS,Test3Stack_SEL
        MOV     DWORD  PTR  [EBX].TRESP,TestStackLEN
        MOV     WORD  PTR  [EBX].TRCS,Test3Code_SEL
        MOV     DWORD  PTR  [EBX].TREIP,OFFSET Test3Begin
        MOV     DWORD  PTR  [EBX].TREFLAG,IOPL2
        ;通过任务门调用测试任务
        CALL32  Test_SEL,0
        ;
        ;把测试任务TSS描述符内的属性置为"可用"
        MOV     AX,ToGDT_SEL
        MOV     FS,AX
        MOV     FS:TESTTSS.ATTRIBUTES,AT386TSS
        ;把测试任务4的入口点、堆栈指针和标志值(含IOPL)填入测试任务TSS
        MOV     WORD  PTR  [EBX].TRSS,Test3Stack_SEL
        MOV     DWORD  PTR  [EBX].TRESP,TestStackLEN
```

```
        MOV     WORD  PTR  [EBX].TRCS,Test3Code_SEL
        MOV     DWORD  PTR  [EBX].TREIP,OFFSET Test3Begin
        MOV     DWORD  PTR  [EBX].TREFLAG,IOPL3
        ;通过任务门调用测试任务
        CALL32  Test_SEL,0
        ;
OVER:   JUMP32  TempCode_SEL,<OFFSET ToDOS>
DemoCodeLEN     =   $
DemoCodeSEG     ENDS
;--------------------------------
;演示任务的临时代码段
TempCodeSEG     SEGMENT   PARA   USE16
        ASSUME    CS:TempCodeSEG
Virtual:
        ;置数据段寄存器为空
        MOV     AX,0
        MOV     DS,AX
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        ;置堆栈指针
        MOV     AX,DemoStack_SEL
        MOV     SS,AX
        MOV     ESP,DemoStackLEN
        ;置任务寄存器 TR
        MOV     AX,DemoTSS_SEL
        LTR     AX
        ;转演示代码段
        JUMP16  DemoCode_SEL,DemoBegin
ToDOS:
        CLTS                            ;清任务切换标志 TS
        MOV     AX,Normal_SEL
        MOV     DS,AX                   ;把规范数据段描述符装入段寄存器
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     SS,AX
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH
        MOV     CR0,EAX                 ;切换到实方式
        JUMP16  <SEG Real>,<OFFSET Real>
TempCodeSEG     ENDS
```

```
;=============================================
;实方式数据段
RDataSEG     SEGMENT   PARA   USE16
VGDTR        PDESC <GDTLEN-1,>
VIDTR        PDESC <IDTLEN-1,>
NORVIDTR     PDESC <3FFH,0>
SPVAR        DW        ?
SSVAR        DW        ?
RDataSEG     ENDS
;------------------------------
;实方式代码段
RCodeSEG     SEGMENT   PARA   USE16
        ASSUME   CS:RCodeSEG,DS:RDataSEG
Start:
        MOV     AX,RDataSEG
        MOV     DS,AX
        CLD
        CALL    INIT_GDT
        CALL    INIT_IDT
        LGDT    QWORD  PTR  VGDTR
        MOV     SSVAR,SS                ;保存实方式堆栈指针
        MOV     SPVAR,SP
        SIDT    NORVIDTR                ;保存 IDTR
        CLI
        LIDT    QWORD  PTR  VIDTR
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX                 ;切换到保护方式
        JUMP16  <TempCode_SEL>,<OFFSET Virtual>
Real:   ;实方式
        MOV     AX,RDataSEG
        MOV     DS,AX
        LSS     SP,DWORD  PTR  SPVAR
        LIDT    NORVIDTR                ;恢复 IDTR
        STI
        MOV     AX,4C00H                ;返回 DOS
        INT     21H
;------------------------------
INIT_GDT     PROC   NEAR
        ;同实例四中的过程内容
INIT_GDT     ENDP
;
```

```
INIT_IDT     PROC
             ;同实例六中的过程内容
INIT_IDTENDP
RCodeSEG     ENDS
             END  Start
```

3. 关于实例九的说明

为了节省篇幅，同时也反映任务状态段的作用，实例通过任务门调用的四个测试任务合用一个任务状态段。从源程序可见，这种合用，实际上是串行的。先把测试任务1的入口点填入测试任务状态段，同时填入堆栈指针，然后调用测试任务1。在测试任务1完成后，再填入测试任务2的入口点，也填入堆栈指针，然后调用测试任务2。依次类推。

通用保护故障处理程序作为一个独立的任务出现。在发生通用保护故障后，通用保护故障处理程序在屏幕的左上角显示故障点的选择子和偏移，然后通过调整存放在任务状态段内的任务链接指针的方法，终止引起故障的测试任务。

实例的其他实现细节作为习题留给读者思考。

10.10 分页管理机制

80386开始支持存储器分页管理机制。分页机制是存储器管理机制的第二部分。段管理机制实现虚拟地址（由段和偏移构成的逻辑地址）到线性地址的转换，分页管理机制实现线性地址到物理地址的转换。如果不启用分页管理机制，那么线性地址就作为物理地址。本节介绍80386的存储器分页管理机制和线性地址如何转换为物理地址。

10.10.1 存储器分页管理机制

在保护方式下，控制寄存器CR0中的最高位PG位控制分页管理机制是否生效。如果PG=1，分页机制生效，把线性地址转换为物理地址。如果PG=0，分页机制无效，线性地址就直接作为物理地址。请参见图10.1。必须注意，只有在保护方式下分页机制才可能生效。只有在保证使PE位为1的前提下，才能够使PG位为1，否则将引起通用保护异常。请参见表10.3。

分页机制把线性地址空间和物理地址空间分别划分为大小相同的块。这样的块称之为页。通过在线性地址空间的页与物理地址空间的页之间建立的映射，分页机制实现线性地址到物理地址的转换。线性地址空间的页与物理地址空间的页之间的映射可根据需要而确立，可根据需要而改变。图10.26反映了线性地址空间的部分页与物理地址空间的部分页之间的映射关系，线性地址空间的任何一页，可以映射为物理地址空间中的任何一页。

采用分页管理机制实现线性地址到物理地址转换映射的主要目的是便于实现虚拟存储器。不像段的大小可变，页的大小是相等并固定的。根据程序的逻辑划分段，而根据实现虚拟存储的方便划分页。

在80386中，页的大小固定为4K字节，每一页的边界地址必须是4K的倍数。因此，

4G 大小的地址空间被划分为 1M 个页，页的开始地址具有“XXXXX000H”的形式。为此，我们把页开始地址的高 20 位 XXXXXH 称为页码。线性地址空间页的页码也就是页开始边界线性地址的高 20 位；物理地址空间页的页码也就是页开始边界物理地址的高 20 位。可见，页码左移 12 位就是页开始地址，所以页码规定了页。

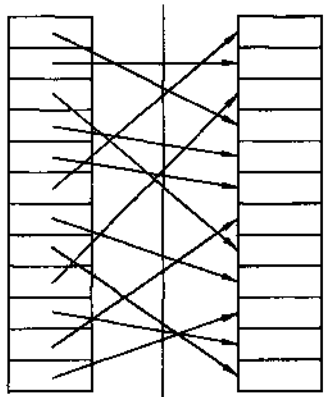


图 10.26 线性地址空间页与物理地址空间页之间的映射示意

由于页的大小固定为 4K 字节，且页的边界是 4K 的倍数，所以在把 32 位线性地址转换成 32 位物理地址的过程中，低 12 位地址可以保持不变。也就是说，线性地址的低 12 位就是物理地址的低 12 位。假设分页机制采用的转换映射把线性地址空间的 XXXXXH 页映射到物理地址空间的 YYYYYH 页，那么线性地址 XXXXXxxxH 被转换为 YYYYYxxxH。因此，线性地址到物理地址的转换要解决的是线性地址空间页到物理地址空间页的映射，也就是线性地址高 20 位到物理地址高 20 位的转换。

10.10.2 线性地址到物理地址的转换

1. 映射表结构

线性地址空间页到物理地址空间页之间的映射用表来描述。由于 4G 的地址空间划分为 1M 个页，因此，如果用一张表来描述这种映射，那么该映射表就要有 1M 个表项，如果每个表项占用 4 个字节，那么该映射表就要占用 4M 字节。为避免映射表占用如此巨大的存储器资源，所以 80386 把页映射表分为两级。

页映射表的第一级称为页目录表，存储在一个 4K 字节的物理页中。页目录表共有 1K 个表项，其中，每个表项为 4 字节长，包含对应第二级表所在物理地址空间页的页码。页映射表的第二级称为页表，每张页表也安排在一个 4K 字节的页中。每张页表都有 1K 个表项，每个表项为 4 字节长，包含对应物理地址空间页的页码。由于页目录表和页表均由 1K 个表项组成，所以使用 10 位就能指定表项。

图 10.27 示出了由页目录表和页表构成的页映射表结构。从图 10.27 中可见，控制寄存器 CR3 指定页目录表；页目录表可以指定 1K 个页表，这些页表可以分散存放在任意的物理页中，而不需要连续存放；每张页表可以指定 1K 个物理地址空间页，这些物理地址空间页可以任意地分散在物理地址空间中。

2. 表项格式

页目录表和页表中的表项都采用如图 10.28 所示的格式。从图可见，最高 20 位(位 12 至位 31)包含物理地址空间页的页码，也就是物理地址的高 20 位。低 12 位包含页的属性。

图 10.28 所示属性中内容为 0 的位是 Intel 公司为 80486 等处理器留下的保留位，在为 80386 编程使用到它们时必须设置成 0。在位 9 至位 11 的 AVL 字段供软件使用。

表项的最低位是存在属性位，记作 P。P 位表示该表项是否有效。P=1 表项有效；P=

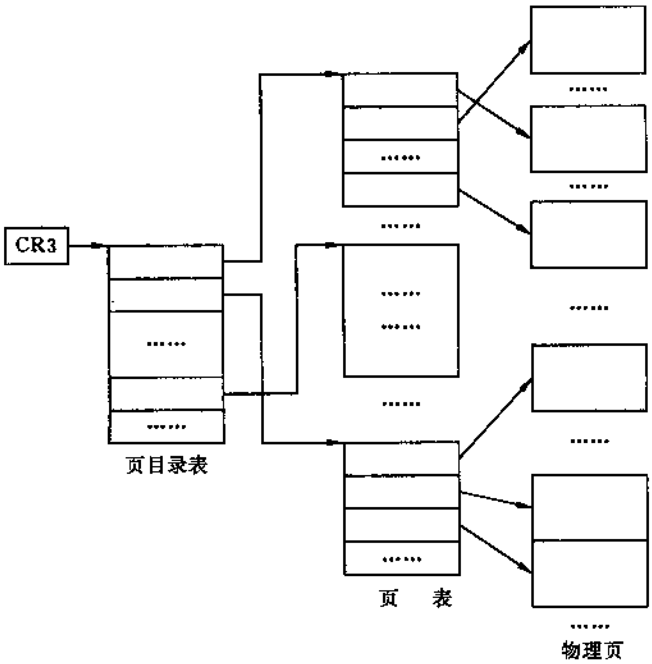


图 10.27　页映射表结构

31～12	11～9	8	7	6	5	4	3	2	1	0
物理页码	AVL	0	0	D	A	0	0	U/S	R/W	P

图 10.28　页目录表或页表的表项格式

0 表项无效，表项中其余各位均可供软件使用，80386 不解释 P＝0 的表项中的任何其他的位。在通过页目录表和页表进行的线性地址到物理地址的转换过程中，无论在页目录表或页表中遇到无效表项，都会引起页故障。

其他属性位的作用在下一小节中介绍。

3. 线性地址到物理地址的转换

分页管理机制通过上述页目录表和页表实现 32 位线性地址到 32 位物理地址的转换。控制寄存器 CR3 的高 20 位作为页目录表所在目录页的页码。首先，把线性地址的最高 10 位(即位 22 至位 31)作为页目录表的索引，对应表项所包含的页码指定页表；然后，再把线性地址的中间 10 位(即位 12 至位 21)作为已指定页表的索引，对应表项所包含的页码指定物理地址空间中的一页；最后，把已指定物理页的页码作为高 20 位，把线性地址的低 12 不加改变直接作为低 12 位，构成 32 位物理地址。

图 10.29 示出了分页管理机制通过页目录表和页表实现 32 位线性地址到 32 位物理地址的转换过程。设分页管理机制有效，CR3 的内容是 00200000H，部分页目录表项和对应的部分页表项如图 10.30 所示，线性地址 00402567H 被转换成物理地址 00303567H。

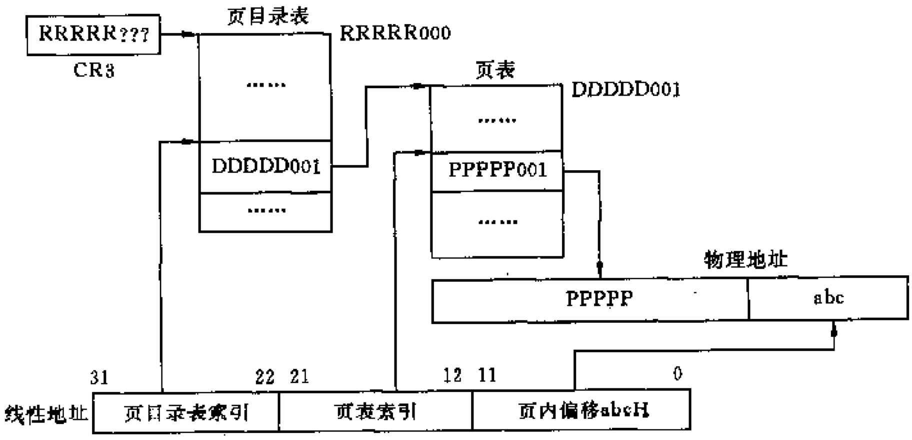


图 10.29 线性地址到物理地址的转换过程

由 CR3 得到页目录表的基地址是 00200000H，线性地址 00402567H 的高 10 位是 001H，作为页目录表中的索引，所以对应表项的物理地址是 00200004H；从该表项得到页表所在物理页的页码是 00201H，也即页表所在物理页的基地址是 00201000H，线性地址的中间 10 位是 002H，作为页表中的索引，所以对应表项的物理地址是 00201008H；从该表项得到物理页的页码是 00303H；线性地址的低 12 位是 567H，直接作为物理地址的低 12 位，于是得物理地址是 00303567H。

基于上述假设，线性地址 000F0123H 被转换成物理地址 000B8123H；线性地址 00000987H 被转换成物理地址 00000987H，与线性地址相同。

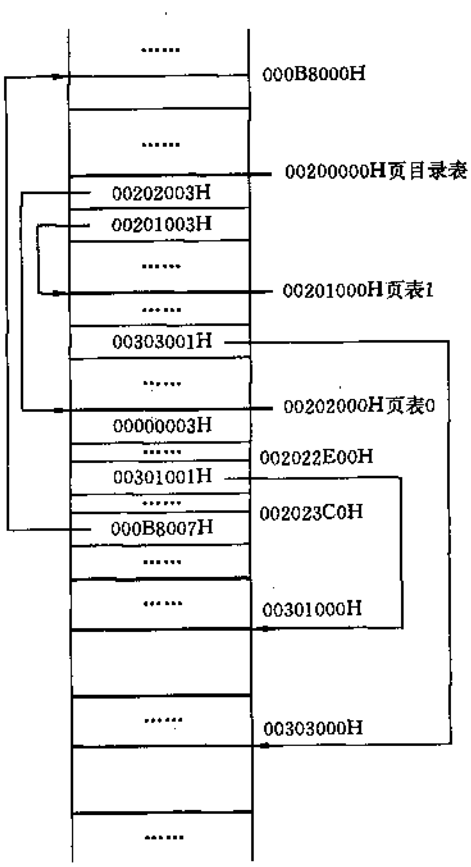


图 10.30 关于页目录表和页表的一个例子

4. 不存在的页表

采用如图 10.27 所示结构的页映射表，存储全部 1K 张页表需要 4M 字节，此外还需要 4K 字节用于存储页目录表。这样的两级页映射表似乎反而要比单一的整张页映射表多占用 4K 字节。其实不然，事实上不需要在内存存储完整的两级页映射表。两级页映射表结构中对于线性地址空间中不存在的或未使用的部分不必分配页表。除必须给页目录表分配物理页外，仅当在需要时才给页表分配物理页，于是页映射表的大小就对应于实际使用的线性地址空间大小。因为任何一个实际运行的程序

使用的线性地址空间，都远小于4G字节，所以用于分配给页表的物理页也远小于4M字节。

页目录表项中的存在位P表明对应页表是否有效。如果P=1，表明对应页表有效，可利用它进行地址转换；如果P=0，表明对应页表无效。如果试图通过无效的页表进行线性地址到物理地址的转换，那么将引起页故障。因此，页目录表项中的属性位P使得操作系统只要给覆盖实际使用的线性地址范围的页表分配物理页。

页目录表项中的属性位P也可用于把页表存储在虚拟存储器中。当发生由于所需页表无效而引起的页故障时，页故障处理程序再申请物理页，从磁盘上把对应的页表读入，把对应页目录表项中的P位置1。换句话说，可以当需要时才为所要的页表分配物理页。这样，页表占用的物理页数量可降到最小。

5. 页的共享

从如图10.27所示的页映射表结构可见，分页机制没有全局页和局部页的规定。每一个任务可使用自己的页映射表独立地实现线性地址到物理地址的转换。但是，如果使每一个任务所用的页映射表具有部分相同的映射，那么也就可以实现部分页的共享。

常用的实现页共享的方法是线性地址空间的共享。也就是不同任务的部分相同的线性地址空间的映射信息相同，具体表现为部分页表相同或页表内的部分表项的页码相同。例如：如果任务A和任务B分别使用的页目录表A和页目录表B内的第0项中的页码相同，也就是页表0相同，那么任务A和任务B的00000000H至003FFFFFH线性地址空间就映射到相同的物理页。再如，任务A和任务B使用的页表0不同，但这两张页表内第0至第0FFH项的页码对应相同，那么任务A和任务B的00000000H至000FFFFFH线性地址空间就映射到相同的物理页。

10.10.3 页级保护和虚拟存储器支持

在如图10.28所示格式的表项中，安排了用于页级保护的属性位和用于支持虚拟存储器的属性位。

1. 页级保护

80386不仅提供段级保护，也提供页级保护。分页机制只区分两种特权级。特权级0、1和2统称为系统特权级，特权级3称为用户特权级。在如图10.28所示页目录表和页表的表项中保护属性位R/W和U/S就是用于对页的保护。

表项的位1是读/写属性位，记作R/W。R/W位指示该表项所指定的页是否可读、写或执行。如R/W=1，对表项所指定页可进行读、写或执行；如R/W=0，对表项所指定页可读或执行，但不能对该指定页写。但是，R/W位对页的写保护只在处理器处于用户特权级时发挥作用；当处理器处于系统特权级时，R/W位被忽略，也即总可以读、写或执行。

表项的位2是用户/系统属性位，记作U/S。U/S位指示该表项所指定的页是否是用户级页。如U/S=1，表项所指定页是用户级页，可由任何特权级下执行的程序访问；如U/S=0，表项所指定页是系统级页，只能由在系统特权级下执行的程序访问。

表 10.11 页级保护属性

U/S	R/W	用户级访问权限	系统级访问权限
0	0	无	读/写/执行
0	1	无	读/写/执行
1	0	读/执行	读/写/执行
1	1	读/写/执行	读/写/执行

表 10.11 列出了在上述属性位 R/W 和 U/S 所确定的页级保护下，用户级程序和系统级程序分别具有的对用户级页和系统级页进行操作的权限。用户级页可以规定为只允许读/执行或者规定为读/写/执行。系统级页对于系统级程序总是可读/写/执行，而对用户程序级程序总是不可访问的。与分段机制一样，外层用户级执行的程序只能访问用户级的页，而内层系统级执行的程序，既可访问系统级页，也可访问用户级页。与分段机制不同的是，在内层系统级执行的程序，对任何页都有读/写/执行访问权，即使规定为只允许读/执行的用户页，内层系统级程序也对该页有写访问权。

页目录表项中的保护属性位 R/W 和 U/S 对由该表项指定页表所指定的全部 1K 个页起到保护作用。所以，对页访问时引用的保护属性位 R/W 和 U/S 的值是组合计算页目录表项和页表项中的保护属性位的值所得。表 10.12 列出了组合计算前后的保护属性位值，组合计算是“与”操作。例如：假设某页表中的某项的 R/W=1 和 U/S=1，表示所指定页是可由用户级程序读/写/执行的用户级页。如果指定该页表的页目录项中的 R/W=0，U/S=1，那么用户级程序实际上可对该页的访问被限制为读/执行；如果指定该页表的页目录项中的 R/W=1，U/S=0，那么实际上用户级程序没有对该页的访问权。

表 10.12 组合页保护属性

目录表项 U/S	页表项 U/S	组合 U/S	目录表项 R/W	页表项 R/W	组合 R/W
0	0	0	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	1	1	1

正如在 80386 地址转换机制中分页机制在分段机制之后起作用一样，由分页机制支持的页级保护也在由分段机制支持的段级保护之后起作用。先测试有关的段级保护，如果启用分页机制，那么在检查通过后，再测试页级保护。例如，设启用分页机制和当前特权级是 3，那么，对于一个存储单元，仅当其所在段及页都允许写入时，该存储单元才是可写的；如果段的类型为读/写，而页规定为只允许读/执行，那么不允许写；如果段的类型为只读/执行，那么不论页保护如何，也不允许写。

页级保护的检查是在线性地址转换成物理地址的过程中进行的，如果违反页保护属性的规定，对页进行访问（读/写/执行），那么将引起页异常。

2. 对虚拟存储器的支持

页表项中的 P 位是支持采用分页机制虚拟存储器的关键。P=1，表示表项指定的页

存在于物理存储器中，并且表项的高 20 位是物理页的页码；P＝0，表示该线性地址空间中的页所对应的物理地址空间中的页不在物理存储器中。如果程序访问不存在的页，会引起页异常，这样操作系统可把该不存在的页从磁盘上读入，把所在物理页的页码填入对应表项和把表项中的 P 置为 1，然后使引起异常的程序恢复运行。

此外，在如图 10.28 所示表项中的访问位 A 和写标志位 D 也用于支持有效地实现虚拟存储器。

表项的位 5 是访问属性位，记作 A。在为了访问某存储单元而进行线性地址到物理地址的转换过程中，处理器总要把页目录表内的对应表项和其所指定页表内的对应表项中的 A 位置 1，除非页表或页不存在，或者访问违反保护属性规定。所以，A＝1 表示已访问过对应的物理页。处理器永不清除 A 位。通过周期地检测及清除 A 位，操作系统就可确定哪些页在最近一段时间未被访问过。当存储器资源紧缺时，这些最近未被访问的页很可能就被选择出来，将它们从内存换出到磁盘上去。

表项的位 6 是写标志位，记作 D。在为了访问某存储单元而进行线性地址到物理地址的转换过程中，如果是写访问并且可以写访问，处理器就把页表内对应表项中的 D 位置 1，但并不把页目录表内对应表项中的 D 置 1。当某页从磁盘上读入内存时，页表中对应表项的 D 位被清 0。所以，D＝1 表示已写过对应的物理页。当某页需要从内存换出到磁盘上去，如果该页的 D 位为 1，那么必须进行写操作。但是，如果要写到磁盘上的页的 D 位为 0，那么不需要实际的磁盘写操作，而只要简单地放弃内存中的该页即可。因为内存中的页与磁盘中的页具有完全相同的内容。

10.10.4 页异常

启用分页机制后，线性地址不再直接等于物理地址，线性地址要经过分页机制转换才成为物理地址。在转换过程中，如果出现下列情况之一就会引起页异常：

(1) 涉及的页目录表内的表项或者页表内的表项中的 P＝0，也即涉及的页不在内存；

(2) 发现试图违反页保护属性的规定而对页进行访问

报告页异常的中断向量号是 14(0EH)。页异常属于故障类异常。在进入故障处理程序时，保存的 CS 及 EIP 指向发生故障的指令。一旦引起页故障的原因被排除后，即可从页故障处理程序通过执行一条 IRET 指令，直接地重新执行产生故障的指令。

当页故障发生时，处理器把引起页故障的线性地址装入控制寄存器 CR2。页故障处理程序可以利用该线性地址确定对应的页目录项和页表项。

页故障还在堆栈中提供一个出错码，出错码的格式如图 10.31 所示。页故障的响应处理模式同其他故障一样。

15　　　　　　　　　　　　3	2	1	0
未使用	U	W	P

图 10.31　页故障出错码格式

在如图 10.31 所示的页故障出错码中，U 位表示引起故障程序的特权级，U＝1 表示用户特权级(特权级 3)，U＝0 表示系统特权级(特权级 0、1 和 2)；W 位表示访问类型，W＝0 表示读/执行，W＝1 表示写；P 位表示异常类型，P＝0 表示页不存在故障，P＝1 表示保护故障。

10.10.5　演示分页机制的实例(实例十)

下面给出一个演示如何启用分页管理机制的实例。该实例的逻辑功能是，在屏幕上显示一条表示已启用分页管理机制的提示信息。该实例演示内容包括：初始化页目录表和部分页表；启用分页管理机制；关闭分页管理机制等。该实例假设系统有 4M 字节物理内存。

1. 演示步骤和源程序清单

为了简单化，实例只有一个任务，并且没有局部描述符表和中断描述符表，不允许中断，也不考虑发生异常，甚至没有使用堆栈。实例执行步骤如下：

(1) 在实方式下为进入保护方式作初始化；

(2) 切换到保护方式后进入临时代码段，把部分演示代码传送到预定的内存，然后转演示代码段；

(3) 建立页目录表，如图 10.30 所示；

(4) 建立页表，如图 10.30 所示；

(5) 启用分页管理机制；

(6) 演示在分页管理机制启用后的程序执行和数据存取；

(7) 关闭分页管理机制；

(8) 退出保护方式，结束。

实例十源程序清单如下：

```
;程序名：T10-10.ASM
;功  能：演示使用分页管理机制
;
        INCLUDE     386SCD.ASM
;
PL          =   1               ;存在属性位P值
RWR         =   0               ;R/W属性位值,读/执行
RWW         =   2               ;R/W属性值,读/写/执行
USU         =   0               ;U/S属性值,系统级
USS         =   4               ;U/S属性值,用户级
;
PDT_AD      =   200000H         ;页目录表所在物理页的地址
PT0_AD      =   202000H         ;页表0所在物理页的地址
PT1_AD      =   201000H         ;页表1所在物理页的地址
;
PhVB_AD     =   0B8000H         ;物理视频缓冲区地址
LoVB_AD     =   0F0000H         ;程序使用的逻辑视频缓冲区地址
```

```
MPVB_AD   =   301000H          ;线性地址 0B8000H 所映射的物理地址
PhSC_AD   =   303000H          ;部分演示代码所在内存的物理地址
LoSC_AD   =   402000H          ;部分演示代码的逻辑地址
;
        .386P
;-----------------------------------------------------------------
;全局描述符表 GDT
GDTSEG    SEGMENT   PARA   USE16
GDT       LABEL   BYTE
DUMMY     DESCRIPTOR       <>
NORMAL    DESCRIPTOR       <0FFFFH,0,0,ATDW,0>
Normal_SEL    =   NORMAL - GDT
;页目录表所在段描述符(在保护方式下初始化时用)
PDTable   DESCRIPTOR        <0FFFH,PDT_AD AND 0FFFFH,
                                   PDT_AD SHR 16,ATDW,0>
PDT_SEL      =    PDTable - GDT
;页表 0 所在段描述符(在保护方式下初始化时用)
PTable0   DESCRIPTOR       <0FFFH,PT0_AD AND 0FFFFH,
                                  PT0_AD SHR 16,ATDW,0>
PT0_SEL      =    PTable0 - GDT
;页表 1 所在段描述符(在保护方式下初始化时用)
PTable1   DESCRIPTOR       <0FFFH,PT1_AD AND 0FFFFH,
                                  PT1_AD SHR 16,ATDW,0>
PT1_SEL      =    PTable1 - GDT
;逻辑上的显示缓冲区所在段描述符
LOVIDEOB    DESCRIPTOR    <3999,LoVB_AD AND 0FFFFH,
                                  LoVB_AD SHR 16,ATDW,0>
LoVideoB_SEL      =   LOVIDEOB - GDT
;逻辑上的部分演示代码所在段描述符
LOCODE      DESCRIPTOR    <SCodeLEN-1,LoSC_AD AND 0FFFFH,
                                  LoSC_AD SHR 16,ATCE,0>
LoCode_SEL        =   LOCODE - GDT
;预定内存区域(用于部分演示代码)所在段的描述符
TPSCODE     DESCRIPTOR    <SCodeLEN-1,PhSC_AD AND 0FFFFH,
                                  PhSC_AD SHR 16,ATDW,>
TPSCode_SEL       =   TPSCODE - GDT
EFFGDT      LABEL BYTE    ;以下是需要额外初始化的描述符
;临时代码段描述符
TEMPCODE  DESCRIPTOR      <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL      =    TEMPCODE - GDT
;演示任务代码段描述符
```

```
DEMOCODE DESCRIPTOR   <DemoCodeLEN-1,DemoCodeSEG,,ATCE,>
DemoCode_SEL     =   DEMOCODE - GDT
;演示任务数据段描述符
DEMODATA DESCRIPTOR   <DemoDataLEN-1,DemoDataSEG,,ATDW,>
DemoData_SEL     =   DEMODATA - GDT
;在初始化时要移动的代码段描述符(移动时作为数据对待)
SCODE        DESCRIPTOR   <SCodeLEN-1,SCodeSEG,,ATDR,>
SCode_SEL    =   SCODE - GDT
GDNUM        =   ($ - EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN       =   $ -GDT
GDTSEG       ENDS
;-----------------------------------------------------------------
;这部分代码在初始化时被复制到预定内存区域
;其功能是在屏幕上显示提示信息
SCodeSEG     SEGMENT  PARA  USE16
        ASSUME  CS:SCodeSEG,DS:DemoDataSEG
SBegin:
        MOV     AX,LoVideoB_SEL
        MOV     ES,AX
        MOV     DI,0
        MOV     AH,17H
        MOV     CX,MESSLEN
S1:     LODSB                                   ;显示信息
        STOSW
        LOOP    S1
        JUMP16  DemoCode_SEL,Demo3
MLEN     =      $ - SBegin
SCodeLEN =      $
SCodeSEG  ENDS
;-----------------------------------------------------------------
;演示任务数据段
DemoDataSEG     SEGMENT   PARA   USE16
MESS            DB   'Page  is  ok!'
MESSLEN         =   $ - MESS
DemoDataLEN     =   $
DemoDataSEG     ENDS
;-----------------------------------------------------------------

;演示任务代码段
DemoCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS:DemoCodeSEG
DemoBegin:
```

```
        MOV     AX,PDT_SEL                  ;初始化页目录表
        MOV     ES,AX
        XOR     DI,DI
        MOV     CX,1024
        XOR     EAX,EAX                     ;先把全部表项置成无效
        REP     STOSD                       ;再置表项0和表项1
        MOV     DWORD  PTR  ES:[0],PT0_AD OR (USU+RWW+PL)
        MOV     DWORD  PTR  ES:[4],PT1_AD OR (USU+RWW+PL)
        ;
        MOV     AX,PT0_SEL                  ;初始化页表0
        MOV     ES,AX
        XOR     DI,DI
        MOV     CX,1024
        XOR     EAX,EAX
        OR      EAX,USU+RWW+PL
Demo1:  STOSD
        ADD     EAX,1000H                   ;先全部置成直接对应等地址的物理页
        LOOP    Demo1                       ;再特别设置两个表项
        MOV     DI,(PhVB_AD SHR 12) * 4
        MOV     DWORD PTR ES:[DI],MPVB_AD+USS+RWW+PL
        MOV     DI,(LoVB_AD  SHR 12) * 4
        MOV     DWORD  PTR  ES:[DI],PhVB_AD+USU+RWR+PL
        ;
        MOV     AX,PT1_SEL                  ;初始化页表1
        MOV     ES,AX
        XOR     DI,DI
        MOV     CX,1024
        MOV     EAX,400000H
Demo2:  STOSD                               ;先把全部表项置成无效
        ADD     EAX,1000H
        LOOP    Demo2                       ;再特别设置一项
        MOV     DI,((LoSC_AD SHR 12) AND 3FFH) * 4
        MOV     DWORD  PTR  ES:[DI],PhSC_AD+USU+RWR+PL
        ;
        MOV     EAX,PDT_AD
        MOV     CR3,EAX
        MOV     EAX,CR0
        OR      EAX,80000000H
        MOV     CR0,EAX                     ;启用分页机制
        JMP     SHORT  PageE
PageE:
        MOV     AX,DemoData_SEL
```

```
        MOV     DS,AX
        MOV     SI,OFFSET   MESS
        JUMP16  LoCode_SEL,Sbegin            ;转位于较大线性地址处的代码执行
Demo3:
        MOV     EAX,CR0
        AND     EAX,7FFFFFFFH                ;关闭分页机制
        MOV     CR0,EAX
        JMP     SHORT   PageD
PageD:
        MOV     AX,Normal_SEL
        JUMP16  TempCode_SEL,ToDOS
DemoCodeLEN     =   $
DemoCodeSEG     ENDS
;---------------------------------------------------------------------
;临时代码段
TempCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS:TempCodeSEG
Virtual:
        CLD                                  ;为演示在启用分页机制后执行位于
        MOV     AX,SCode_SEL                 ;较高线性地址空间中的代码作准备
        MOV     DS,AX
        MOV     AX,TPSCode_SEL
        MOV     ES,AX
        MOV     SI,OFFSET SBegin
        MOV     DI,SI
        MOV     CX,MLEN                      ;把部分演示代码复制到预定的内存
        REP     MOVSB                        ;
        JUMP16  DemoCode_SEL,DemoBegin
        ;
ToDOS:
        MOV     DS,AX                        ;准备返回实方式
        MOV     ES,AX
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH
        MOV     CR0,EAX                      ;返回实方式
        JUMP16  <SEG Real>,<OFFSET Real>
TempCodeSEG     ENDS
;=====================================================================
;实方式下的初始化代码和数据
RCodeSEG        SEGMENT   PARA   USE16
                ASSUME   CS:RCodeSEG,DS:RCodeSEG
VGDTR           PDESC       <GDTLEN-1,>
```

```
Start:  PUSH    CS
        POP     DS
        CLD
        CALL    INIT_GDT                    ;初始化全局描述符表 GDT
        CALL    EA20                        ;打开地址线 A20
        LGDT    QWORD  PTR  VGDTR
        CLI
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX                     ;转保护方式
        JUMP16  <TempCode_SEL>,<OFFSET Virtual>
Real:                                       ;回到实方式
        CALL    DA20                        ;关闭地址线 A20
        STI
        MOV     AX,4C00H                    ;返回 DOS
        INT     21H
;------------------------------------------------------------
;实方式下的初始化过程
INIT_GDT   PROC   NEAR
        ;同实例四中的过程内容
INIT_GDT   ENDP
;
EA20    PROC
        ;同实例一
EA20    ENDP
;
DA20    PROC
        ;同实例一
DA20    ENDP
RCodeSEG        ENDS
                END     Start
```

2. 关于实例十的说明

上述演示程序的许多内容与其他实例相同，下面仅就演示分页管理机制方面的内容作些说明：

(1) 部分演示代码的移动

为了充分说明分页机制所实现的线性地址到物理地址的转换，在初始化时把部分演示代码移动到预定的内存区域。预定的内存区域从 00303000H 开始，也即是页码为 00303H 的物理页。该部分演示代码的功能是显示指定的字符串。在进入保护方式后做这初始化工作的原因是预定的内存区域在扩展内存中，注意初始化时还没有启用分页机制。

(2) 页映射表的初始化

实例按如图 10.30 所示安排映射表。页目录表安排在页码为 00200H 的物理页中，页表 0 安排在页码为 00202H 的物理页中，页表 1 安排在页码为 00201H 的物理页中。演示程序涉及的线性地址空间不超出 007FFFFFH，所以只使用两张页表，为此页目录表中的其他项被置为无效(P=0)。

页表 0 把线性地址空间中的 00000000H～003FFFFFH 映射到物理地址空间中。实例在初始化页表 0 时，使该线性地址空间直接映射到相同地址的物理地址空间，除线性地址空间中页码为 000B8H 和 000F0H 这两页以外。000B8H 页被映射到页码为 00301H 的物理页，而 000F0H 页被映射到页码为 000B8H 的物理页。

页表 1 把线性地址空间中的 00400000H～007FFFFFH 映射到物理地址空间中。实例在初始化页表 1 时，似乎使该线性地址空间直接映射到相同地址的物理地址空间，但是除了对应线性地址空间中 00402H 页的表项被另外设置外，其他表项中的 P 位为 0，也即表示对应物理页不存在。初始化后，页表 1 的第 2 项把线性地址空间中的 00402H 页，映射到页码为 00303H 的物理页，也就是存放部分演示代码的指定内存区域。

(3) 启用分页管理机制

在建立好页映射表后，启用分页机制所要做的操作是简单的，只要把控制寄存器中的最高位，也就是 PG 位置 1。具体指令如下：

```
        MOV     EAX,CR0
        OR      EAX,80000000H
        MOV     CR0,EAX
        JMP     SHORT  PageE
PageE:
```

在启用分页机制前，线性地址就是物理地址；在启用分页机制后，线性地址要通过分页机制的转换，才成为物理地址。尽管使用一条转移指令，可清除预取的指令，但随后在取指令时使用的线性地址就要经过转换才成为物理地址。为了保证顺利过渡，在启用分页机制之后的过渡阶段，仍要维持线性地址等同于物理地址。为了做到这一点，在建立页映射表时，必须使实现过渡的代码所在的线性地址空间页映射到具有相同地址的物理地址空间页。实例中页表 0 就做到了这一点。

(4) 关闭分页管理机制

只要把控制寄存器 CR0 中的 PG 位清 0，便关闭分页机制。在这一过渡阶段，也要保持地址转换前后的一致。

(5) 地址转换的演示

在启用分页机制之后，就转移到位于线性地址空间中 00402000H 处开始的代码，该部分代码的功能是显示提示信息“Page is ok!”。实际上这部分代码存放在从物理地址 00303000H 开始的内存区域中，是在初始化时被移到此区域的。

在显示提示信息时，把要显示的字符 ASCII 和显示属性填到线性地址空间中 000F0000H 开始的区域中，而不是 000B8000H 开始的区域。从在初始化时建立的映射表可见，线性地址空间中的 000F0H 页，被映射到物理地址空间中的 000B8H 页。所以，向线性地址空间中的 000F0H 页写，实际上是向物理地址空间中的 000B8H 页写，也就是真正

显示。

(6) 页级保护的演示

在进入保护方式之后,特权级一直是0级。所以,无论系统级和用户级页,无论只能读/执行,还是读/执行/写,总是可进行各种形式的访问。

10.11 虚拟8086方式

继推出80386之后,Intel又推出了80486、Pentium和Pentium PRO。这些处理器都具有实方式和保护方式两种工作模式。前面已介绍过,实方式与8086方式兼容,可以运行DOS及以其为平台的几乎所有软件;但在实方式下,处理器不能发挥自身的优越性能,不能支持多用户、多任务操作系统的运行。为了充分发挥处理器的功能,同时使DOS及以其为平台的软件继续有效地运行,从80386开始增加了虚拟8086方式。本节介绍虚拟8086方式。

10.11.1 V86方式

1. V86方式

虚拟8086方式是保护方式下的一种工作方式,也称为V8086方式,或者简称为V86方式。在虚拟8086方式下,处理器类似于8086。寻址的地址空间是1M字节;段寄存器的内容作为段值解释;20位存储单元地址由段值乘16加偏移构成。所以,在虚拟8086方式下,可以运行DOS及以其为平台的软件。但V86方式毕竟是虚拟8086的一种方式,所以不完全等同于8086。

当如图9.2所示的标志寄存器中的标志VM为1时,处理器就处于V86方式。当处理器处于V86方式时,其当前特权级必定是3。

2. V86任务

8086程序可直接在V86方式下运行,而V86方式受到称为V86监控程序的控制。V86监控程序和在V86方式下的8086程序构成的任务称为虚拟8086任务,或者简称为V86任务。V86任务形成一个由处理器硬件和属于系统软件的监控程序组成的"虚拟8086机"。V86监控程序控制V86外部界面、中断和I/O。硬件提供该任务最低端1M字节线性地址空间的虚拟存储空间,包含虚拟寄存器的TSS,并执行处理这些寄存器和地址空间的指令。

80386把V86任务作为与其他任务具有同等地位的一个任务。它可以支持多个V86任务,每个V86任务是相对独立的。所以,通过V86方式这种形式,运行8086程序可充分发挥处理器的能力和充分利用系统资源。

10.11.2 进入和离开V86方式

保护方式和V86方式之间的切换情形如图10.32所示。图中左面部分为V86任务。从图10.32可见,V86方式与保护方式的切换可发生在V86任务之内,这种切换是V86方式下的8086程序与保护方式下的监控程序之间的转换;V86方式与保护方式的切换可

发生在任务之间，这种切换是V86任务与其他任务的切换。此外，V86监控程序与其他任务之间的切换是普通的任务切换。

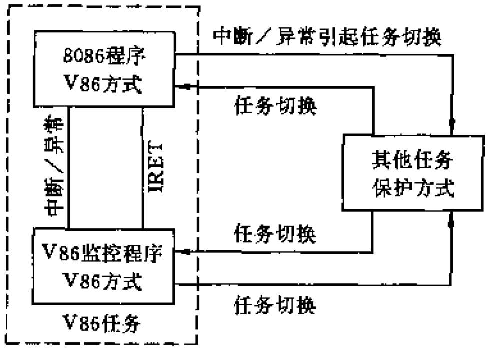


图10.32 进入和离开V86方式的情形

由于80386没有提供直接改变VM标志的指令，并且只有当前特权级CPL＝0时，对VM的改变才有效，所以V86方式与保护方式的切换不能简单地通过设置或改变VM而进行。下面介绍V86方式与保护方式之间的切换，也就是如何进入和离开V86方式。为了方便，先介绍如何离开V86方式。

1. 离开V86方式

在V86方式下，如果处理器响应中断/异常，那么就会退出当前V86任务的V86方式。

在V86方式下，处理器对中断/异常的响应处理不同于真正的8086，而仍然采用保护方式下对中断/异常响应处理的方法。所以，在V86方式下，不是根据位于线性地址空间最低端的中断向量表内的对应中断向量转入处理程序，而是根据中断描述符表IDT内的对应门描述符的指示转入处理程序。

(1) 在V86任务内离开V86方式

如果对应的门描述符是386中断门或386陷阱门，那么就发生在当前V86任务内从V86方式到保护方式的转换。80386要求执行这种中断/异常处理程序时的CPL必须等于0。

由于V86方式下的CPL＝3，而转换到保护方式后的CPL＝0，所以这种转换包含了特权级的变换。在按10.7节介绍的方法转入处理程序之前，处理器先将V86方式下的段寄存器GS、FS、DS及ES压入0级堆栈，并装入空选择子。为保持使堆栈对齐，在把段寄存器压入堆栈时，一律按32位值压入，低16位是段寄存器的值，高16位为空。于是，转换后的0级堆栈如图10.33所示。其中，段寄存器SS和CS的值也是V86方式下的段值。图(a)是没有出错码的情形；图(b)是有出错码的情形。请与图10.21作比较。

在这种V86任务内从V86方式转换到保护方式的过程中，为了保证中断/异常处理程序工作于特权级0，对目标代码段描述符特权级进行检查，如果由目标代码段描述符特权级决定的CPL不等于0，将引起通用保护异常。此外，标志寄存器EFLAGS中的VM位被清0，从而使得中断/异常处理在保护方式下进行，也即离开V86方式。

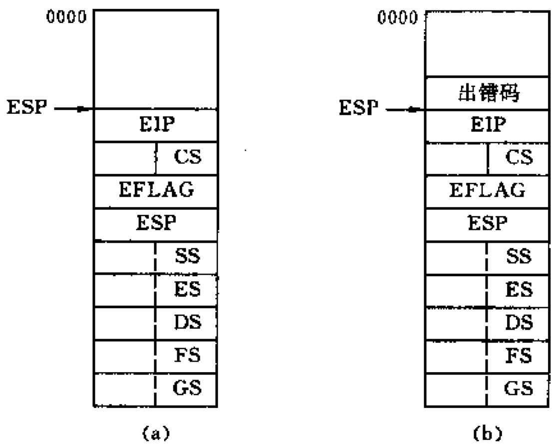


图 10.33 通过中断门/陷阱门从 V86 方式到保护方式转换时的 0 级堆栈

这种情况下，相应的中断/异常处理在当前 V86 任务之内进行。中断/异常处理程序可以检查保存在堆栈中的 EFLAGS 映象，根据 VM 位的值来确定被中断程序的工作方式。如果 VM＝1，那么被中断的程序工作于 V86 方式，是 8086 程序；否则，被中断的程序工作于保护方式，是 V86 监控程序。

(2) 任务切换离开 V86 方式

如果对应的门描述符是任务门，那么就发生从当前 V86 任务到其他任务的切换，也就离开当前 V86 任务的 V86 方式。象普通任务切换一样，V86 方式的各通用寄存器、段寄存器、指令指针和标志寄存器 EFLAGS 等保存到原 V86 任务的 386TSS 中。被保存的段寄存器的内容是 V86 方式下的段值。被保存的 EFLAGS 内的 VM＝1。

这种情况下，相应的中断/异常处理在另一个任务内进行。目标任务可以是普通任务，也可以是另一个 V86 任务。如果目标任务 TSS 内的 EFLAGS 字段内的 VM＝1，那么就转入另一个 V86 任务的 V86 方式。

2. 进入 V86 方式

与离开 V86 方式的两条途径相对应，有两条进入 V86 方式的途径。

(1) 通过 IRET 指令进入 V86 方式

通常在中断/异常处理结束时使用 IRET 指令返回被中断的程序继续执行。指令 IRET 的执行流程如图 10.34 所示，尽管它不够细致和没包括异常情况，但还是体现了指令 IRET 执行时所处理的三种情形。第一种情形是当前 EFLAGS 中的 NT＝1，也即嵌套任务返回，那么就进行任务切换，指向目标任务 TSS 的选择子在当前任务 TSS 的链接字段。NT＝0 表示当前中断/异常处理程序与被中断程序属于同一任务，于是就从堆栈弹出 EIP、CS 和 EFLAGS。第二和第三种情形是在 NT＝0 的条件下产生。第二种情形是弹出的 EFLAGS 中 VM＝0，表示被中断的程序是普通保护方式程序，那么就考虑特权级变换、如果向外层返回，那么就恢复外层堆栈指针，不允许向内层返回。在 10.7.3 中介绍的

指令 IRET 的动作只考虑情形一和情形二，并不是指令 IRET 的完整动作。

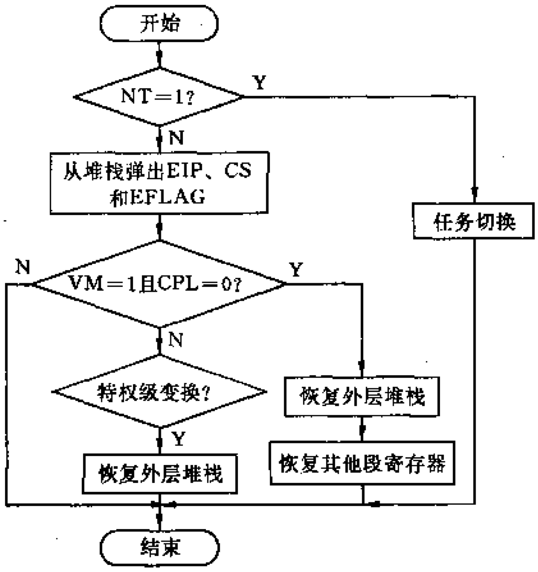


图 10.34　指令 IRET 的执行动作流程

第三种情形是弹出的 EFLAGS 中 VM=1 且 CPL=0，表示被中断的程序是 V86 方式下的 8086 程序，当前是从同一 V86 任务下的中断/异常处理程序返回。由于 V86 方式的特权级是 3，所以要进行堆栈切换，也即从堆栈中弹出 3 级堆栈的指针(ESP 和 SS)。此外，还从堆栈中弹出段寄存器 ES、DS、FS 和 GS。在这种情形下，弹到各段寄存器(包括 CS 和 SS)的内容都作为段值，而非选择子。这种处理动作对应于上述第一种离开 V86 方式的情形，有关堆栈操作也与图 10.33 所示的堆栈内容相符。当然，如果产生异常时提供出错码，那么异常处理程序在利用 IRET 指令返回时，必须确保堆栈指针指向图 10.33 所示保存 EIP 的单元。简单的实现方法是，异常处理程序在执行 IRET 前，先从堆栈弹出出错码。

利用指令 IRET 处理的这第三种情形，可以方便地从 V86 任务下的中断/异常处理程序返回到 V86 方式下的 8086 程序。利用这条途径还可以直接进入 V86 方式。为此，先在 0 级堆栈中形成如图 10.33(a)所示的栈顶。对应 EIP 值是 V86 方式下要执行的 8086 程序入口点的 16 位偏移；对应 CS 值是 V86 方式下要执行的 8086 程序入口点的段值；对应 EFLAGS 值中的 VM 位必须是 1；对应 SS 和 ESP 的值是要执行的 8086 程序的堆栈指针；对应 ES、DS、FS 和 GS 的值是相应的段值。然后，在 CPL=0 和 NT=0 的情况下，执行 IRET 指令。实际上，这种进入 V86 方式的途径是，先建立一个 V86 方式下执行的 8086 程序被中断而离开 V86 方式的环境，然后再返回。

(2) 通过任务切换进入 V86 方式

通过任务切换的途径，可以从其他任务进入 V86 任务内的 V86 方式。

利用在前面几节介绍的任务切换方法可以进行任务切换。如果目标任务由 386TSS 描述，并且其中 EFLAGS 字段内的 VM 位为 1，那么在切换到目标任务时，也就进入 V86

方式。在切换到 V86 方式时，CPL 被规定为 3。目标任务 TSS 中的各段寄存器字段被解释为 8086 可以接受的段值，而不是选择子。任务切换时也将装载 LDTR 和 CR3。

如果利用这条途径建立 V86 任务并进入 V86 方式，那么主要是把对应 386TSS 中 EFLAGS 字段内的 VM 位置 1，把 8086 程序的有关段值填入对应 386TSS 中的相应段寄存器字段。此外，如果 V86 监控程序需要用到 LDT，那么还要填 LDTR 字段；如果需要采用分页机制，那么还要填 CR3 字段。

10.11.3 演示进入和离开 V86 方式的实例(实例十一)

下面给出一个用于演示进入和离开 V86 方式的实例。该实例的逻辑功能是，以驻留方式结束程序，退出时已处于 V86 方式。该实例演示内容包括：两种方式进入 V86 方式和两种方式离开 V86 方式；V86 方式下的 8086 程序如何调用实方式下的软中断处理程序。

1. 演示步骤和源程序清单

为了便于演示，本实例含有三个任务：临时任务、V86 任务和 INTFF 任务。在实方式下作必要的初始化工作后切换到保护方式，也即进入临时任务，开始演示。演示分两个阶段：第一阶段进入 V86 任务的 V86 方式，并驻留退出；第二阶段进入 INTFF 任务，切换到临时任务，并返回实方式。

第一阶段的演示步骤如下：

(1) 开始临时任务后，作切换到 V86 任务的准备；

(2) 切换到 V86 任务，由于 V86 任务 TSS 中的 EFLAGS 字段内的 VM＝1，所以伴随着任务切换，就进入 V86 方式；

(3) 进入 V86 任务的 V86 方式后，显示提示信息，驻留结束，出现 DOS 提示符，第一阶段至此结束。

在 V86 方式下，可进行各种操作，运行其他 8086 程序。如果 8086 程序引起通用保护异常，那么在屏幕第一行显示提示信息，并中止该 8086 程序。如果在 8086 程序中执行“INT FF”指令，开始第二阶段。

第二阶段的演示步骤如下：

(1) 进入 INTFF 任务后，显示提示信息，切换到临时任务；

(2) 在临时任务内切换回到实方式；

(3) 在实方式下，中止发出“INT FF”的程序。

源程序有如下几部分组成：

(1) 全局描述符表 GDT；

(2) 中断描述符表 IDT(只适用于 V86 任务)；

(3) INTFF 任务的 TSS 段、LDT 段、0 级堆栈段和代码段；

(4) V86 任务的 TSS 段、LDT 段、0 级堆栈段、3 级堆栈段及数据段，通用保护异常处理程序段和其他中断/异常处理程序段，V86 方式下的 8086 程序段；

(5) 临时任务的 TSS 段和代码段；

(6) 实方式下的初始化代码段及有关过程。

源程序清单如下：

```
;程序名：T10-11.ASM
;功  能：演示进入和离开 V86 方式
;
        INCLUDE     386SCD.ASM
;
    .386P
;-----------------------------------------------------------------
;全局描述符表 GDT
GDTSEG      SEGMENT   PARA   USE16
GDT         LABEL   BYTE
DUMMY       DESCRIPTOR   <>
NORMAL      DESCRIPTOR   <0FFFFH,0,0,ATDW,0>
Normal_SEL      =     NORMAL - GDT
EFFGDT      LABEL   BYTE          ;以下是需要额外初始化的全局描述符
;V86 任务 TSS 段描述符
V86TSS      DESCRIPTOR   <V86TSSLEN-1,V86TSSSEG,,AT386TSS,>
V86TSS_SEL      =     V86TSS - GDT
;V86 任务局部描述符表的描述符
V86LDT      DESCRIPTOR   <V86LDTLEN-1,V86LDTSEG,,ATLDT,>
V86LDT_SEL      =     V86LDT - GDT
;INTFF 任务 TSS 段描述符
INTFFTSS    DESCRIPTOR   <INTFFTSSLEN-1,INTFFTSSSEG,,AT386TSS,>
INTFF_TSS_SEL=      INTFFTSS - GDT
;INTFF 任务局部描述符表的描述符
INTFFLDT    DESCRIPTOR   <INTFFLDTLEN-1,INTFFLDTSEG,,ATLDT,>
INTFFLDT_SEL =      INTFFLDT - GDT
;临时任务 TSS 段描述符
TEMPTSS     DESCRIPTOR   <TempTSSLEN-1,TempTSSSEG,,AT386TSS,>
TempTSS_SEL     =     TEMPTSS - GDT
;临时任务代码段描述符
TEMPCODE DESCRIPTOR      <0FFFFH,TempCodeSEG,,ATCE,>
TempCode_SEL    =     TEMPCODE - GDT
;显示缓冲区描述符
VIDEOBUFF DESCRIPTOR     <80*25*2-1,0B800H,,ATDW,>
VideoBuff_SEL   =     VIDEOBUFF - GDT
GDNUM           =     ($ - EFFGDT)/(SIZE DESCRIPTOR)
GDTLEN          =     $ - GDT
GDTSEG      ENDS
;-----------------------------------------------------------------
;V86 任务使用的中断描述符表 IDT
IDTSEG      SEGMENT   PARA   USE16
```

```
IDT     LABEL   BYTE
;对应 0~12 号中断/异常的中断门描述符
        REPT    13
        GATE    <?,TPCode_SEL,0,AT386IGAT+DPL3,0>
        ENDM
;对应一般保护异常的陷阱门描述符
        GATE    <GPBegin,GPCode_SEL,0,AT386TGAT+DPL3,0>
;对应 15-254 号中断/异常的中断门描述符
        REPT    256 -1 -14
        GATE    <?,TPCode_SEL,0,AT386IGAT+DPL3,0>
        ENDM
;对应 255(0FFH)号中断的任务门描述符
        GATE    <?,INTFF_TSS_SEL,0,ATTASKGAT+DPL3,? >
IDTLEN  =       $ - IDT
IDTSEG  ENDS
;-------------------------------------------------------------
;INTFF 任务的 TSS 段
INTFFTSSSEG     SEGMENT   PARA   USE16
          DD    0                       ; 链接字
          DD    ?                       ; 0 级堆栈指针
          DW    ?,?
          DD    ?                       ; 1 级堆栈指针
          DW    ?,?                     ; 特权级不会变换,无需初始化
          DD    ?                       ; 2 级堆栈指针
          DW    ?,?
          DD    0                       ; CR3
          DW    INTFFBegin,0            ; EIP
          DW    0,0                     ; EFLAGS
          DD    0                       ; EAX
          DD    0                       ; ECX
          DD    0                       ; EDX
          DD    0                       ; EBX
          DW    INTFFStack0LEN,0        ; ESP
          DD    0                       ; EBP
          DD    0                       ; ESI
          DD    0                       ; EDI
          DW    Normal_SEL,0            ; ES(段寄存器已初始化妥)
          DW    INTFFCode_SEL,0         ; CS
          DW    INTFFStack0_SEL,0       ; SS
          DW    Normal_SEL,0            ; DS
          DW    Normal_SEL,0            ; FS
          DW    Normal_SEL,0            ; GS
```

```
        DW      INTFFLDT_SEL,0          ; LDT
        DW      0
        DW      $+2                     ; 指向I/O许可位图区的指针
        DB      0FFH                    ; I/O许可位图结束字节
INTFFTSSLEN     =   $
INTFFTSSSEG     ENDS
;----------------------------------------------------------------
;INTFF任务的LDT段
INTFFLDTSEG     SEGMENT   PARA   USE16
FLDT        LABEL BYTE
;0级堆栈段描述符
INTFFSTACK0     DESCRIPTOR    <INTFFStack0LEN-1,INTFFStack0SEG,,
                                        ATDWA,>
INTFFStack0_SEL   =  (INTFFSTACK0 - FLDT) + TIL
;代码段描述符
INTFFCODE       DESCRIPTOR    <INTFFCodeLEN-1,INTFFCodeSEG
                                        ,,ATCER,>
INTFFCode_SEL     =  (INTFFCODE - FLDT) + TIL
INTFFLDNUM        =  ($ - FLDT)/(SIZE  DESCRIPTOR)
INTFFLDTLEN       =  $                ;LDT段长度
INTFFLDTSEG     ENDS
;----------------------------------------------------------------
;INTFF任务的0级堆栈
INTFFStack0SEG  SEGMENT   PARA   USE16
INTFFStack0LEN    =   512
        DB INTFFStack0LEN   DUP   (0)
INTFFStack0SEG  ENDS
;----------------------------------------------------------------
;INTFF任务的代码段
INTFFCodeSEG    SEGMENT   PARA   USE16
        ASSUME  CS:INTFFCodeSEG
INTFFMESS       DB  'Return  to  real  mode.'
INTFFMESSLEN      =     $ - INTFFMESS
INTFFBegin:
        MOV     SI,OFFSET GPERRMESS
        MOV     AX,VideoBuff_sel
        MOV     ES,AX                   ; 置显示缓冲区选择子
        MOV     DI,0                    ; 从屏幕左上角开始显示
        MOV     AH,17H                  ; 置显示属性
        MOV     CX,INTFFMESSLEN         ; 置提示信息长度
        CLD
INEXT:  MOV     AL,CS:[SI]              ; 从代码段取显示信息
```

```
        INC     SI
        STOSW                               ；显示返回实方式的提示信息
        LOOP    INEXT
        JUMP16  TempTSS_SEL,0               ；切换到临时任务
INTFFCodeLEN    =   $
INTFFCodeSEG    ENDS
;--------------------------------------------------------------------
;V86 任务的 TSS 段
V86TSSSEG     SEGMENT   PARA   USE16
        DD      0                           ；链接字
        DD      V86Stack0LEN                ；0 级堆栈指针
        DW      V86Stack0_SEL,?
        DD      ?                           ；1 级堆栈指针
        DW      ?,?
        DD      ?                           ；2 级堆栈指针
        DW      ?,?
        DD      0                           ；CR3
        DW      V86Begin,0                  ；EIP
        DW      IOPL3,VMFL                  ；EFLAGS(IO 特权级为 3,VM=1)
        DD      0                           ；EAX
        DD      0                           ；ECX
        DD      0                           ；EDX
        DD      0                           ；EBX
        DW      V86Stack3LEN,0              ；ESP
        DD      0                           ；EBP
        DD      0                           ；ESI
        DD      0                           ；EDI
        DW      V86CodeSEG,0                ；ES(V86 方式下的段值)
        DW      V86CodeSEG,0                ；CS
        DW      V86Stack3SEG,0              ；SS
        DW      V86CodeSEG,0                ；DS
        DW      V86CodeSEG,0                ；FS
        DW      V86CodeSEG,0                ；GS
        DW      V86LDT_SEL,0                ；V86 任务的局部描述符表选择子
        DW      0
        DW      $+2                         ；指向 I/O 许可位图区的指针
        DB      4000H/8 DUP (0)             ；I/O 许可位图
        DB      0FFH                        ；I/O 许可位图结束字节
V86TSSLEN       =   $
V86TSSSEG       ENDS
;--------------------------------------------------------------------
; V86 任务的 LDT 段
```

```
V86LDTSEG     SEGMENT   PARA   USE16
VLDT     LABEL  BYTE
; V86 任务线性地址空间中最低端 1M 字节段的描述符
VALLMEM  DESCRIPTOR     <0FFFFH,0,,8F00H+ATDWA,>
VAllMEM_SEL   =    (VALLMEM - VLDT) + TIL
;V86 任务 0 级堆栈段描述符
V86STACK0 DESCRIPTOR    <V86Stack0LEN-1,V86Stack0SEG,,ATDWA,>
V86Stack0_SEL   =    (V86STACK0 - VLDT) + TIL
;V86 任务数据段描述符
V86DATA   DESCRIPTOR    <V86DataLEN-1,V86DataSEG,,ATDR,>
V86Data_SEL     =    (V86DATA - VLDT) + TIL
;V86 任务中断/异常处理程序代码段描述符
TPCODE    DESCRIPTOR    <TPCodeLEN-1,TPCodeSEG,,ATCE,>
TPCode_SEL      =    (TPCODE - VLDT) + TIL
;V86 任务通用保护异常处理程序代码段描述符
GPCODE    DESCRIPTOR    <GPCodeLEN-1,GPCodeSEG,,ATCE,>
GPCode_SEL      =    (GPCODE - VLDT) + TIL
V86LDNUM        =    ($ - VLDT)/(SIZE  DESCRIPTOR)
V86LDTLEN       =    $
V86LDTSEG     ENDS
;-------------------------------------------------------------------
;V86 任务的 0 级堆栈
V86Stack0SEG    SEGMENT   PARA   USE16
V86Stack0LEN     =    512
         DB  V86Stack0LEN   DUP (0)
V86Stack0SEG    ENDS
;-------------------------------------------------------------------
;V86 任务的 3 级堆栈
V86Stack3SEG    SEGMENT   PARA   USE16
V86Stack3LEN     =    1024
         DB  V86Stack3LEN   DUP (0)
V86Stack3SEG    ENDS
;-------------------------------------------------------------------
;V86 数据段
V86DataSEG      SEGMENT   PARA   USE16
GPERRMESS       DB  '……General  Protection  Error…'
GPERRMESSLEN =   $ - GPERRMESS
V86DataLEN       =   $
V86DataSEG      ENDS
;-------------------------------------------------------------------
;定义部分代表堆栈单元的符号(参见图 10.33)
Perr      EQU     WORD  PTR    [BP+0]
```

```
Pip     EQU   WORD  PTR   [BP+4]
Pcs     EQU   WORD  PTR   [BP+8]
Pflag   EQU   WORD  PTR   [BP+12]
Psp     EQU   WORD  PTR   [BP+16]
Pss     EQU   WORD  PTR   [BP+20]
Pes     EQU   WORD  PTR   [BP+24]
Pds     EQU   WORD  PTR   [BP+28]
Pfs     EQU   WORD  PTR   [BP+32]
Pgs     EQU   WORD  PTR   [BP+36]
;-----------------------------------------------------------------
;V86 任务下的中断/异常处理程序代码段
TPCodeSEG    SEGMENT  PARA  USE16
        ASSUME  CS:TPCodeSEG
TPBegin:
COUNT   =       0
        REPT    256                          ;对应 256 个入口
        IF  COUNT  EQ  21H
ENT21H     LABEL  BYTE                       ;在第 21H 项处定义标号 ENT21H
        ENDIF
        PUSH    BP
        MOV     BP,COUNT                     ;置中断向量号到 BP
        JMP     PROCESS                      ;都转统一的处理程序
COUNT =         COUNT+1
        ENDM
PROCESS:
        PUSH    BP                           ;保存 BP
        MOV     BP,SP                        ;堆栈指针送 BP
        PUSH    EAX
        PUSH    EBX                          ;保存 EAX、EBX
        ;①在 V86 堆栈顶形成返回点的现场
        MOV     AX,VAllMEM_SEL               ;装载描述最低 1M 字节线性地址空间
        MOV     DS,AX                        ;的描述符选择子
        XOR     EAX,EAX
        MOV     AX,Psp                       ;修改在 V86 任务 0 级堆栈中保存的
        SUB     AX,3*2                       ;3 级堆栈指针,减 3 个字
        MOV     Psp,AX                       ;即在 V86 方式下的堆栈顶空出 3 个字
        XOR     EBX,EBX
        MOV     BX,Pss                       ;使 EBX 指向 V86 堆栈顶
        SHL     EBX,4
        ADD     EBX,EAX
        MOV     AX,Pip                       ;把保存在 0 级堆栈中的返回地址的
        MOV     [EBX],AX                     ;偏移部分送 V86 堆栈
```

```
        MOV     AX,Pcs
        MOV     [EBX+2],AX              ; 段值部分送 V86 堆栈
        MOV     [AX,Pflag
        MOV     [EBX+4],AX              ; 标志值送 V86 堆栈
        ;②用对应的中断向量值代替返回地址
        MOV     BX,[BP]                 ; 取中断号
        SHL     BX,2                    ; 乘 4
        MOV     AX,[BX]                 ; 取实方式下对应中断向量的偏移
        MOV     Pip,AX                  ; 代替 0 级堆栈中的 EIP
        MOV     AX,[BX+2]               ; 取实方式下对应中断向量的段值
        MOV     Pcs,AX                  ; 代替 0 级堆栈中的 CS
        ;
        POP     EBX                     ; 恢复 EBX、EAX 等
        POP     EAX
        POP     BP
        POP     BP
        ;③从保护方式返回 V86 方式
        ;先转入对应中断处理程序,再返回中断发生处
        IRETD
TPCodeLEN     =   $
TPCodeSEG     ENDS
;-----------------------------------------------------------------
;V86 任务下的通用保护异常处理程序代码段
GPCodeSEG     SEGMENT   PARA   USE16
      ASSUME    CS:GPCodeSEG
GPBegin:
        MOV     AX,V86Data_SEL
        MOV     DS,AX                   ; 装载 V86 任务的数据段
        MOV     SI,OFFSET  GPERRMESS
        MOV     AX,VideoBuff_sel
        MOV     ES,AX
        MOV     DI,0
        MOV     AH,17H                  ; 显示属性值
        MOV     CX,GPERRMESSLEN
        CLD
GNEXT:  LODSB
        STOSW                           ; 显示发生通用保护异常的提示信息
        LOOP    GNEXT
        ; 利用 DOS 的 21H 功能调用终止引起该异常的程序
        ADD     ESP,4                   ; 废除堆栈中的出错代码
        MOV     AX,4C01H
        JUMP16  TPCode_SEL,ENT21H       ; 转 21H 号中断处理程序
```

```
GPCodeLEN     =   $
GPCodeSEG     ENDS
;-------------------------------------------------------------------------------
;V86 方式执行的 8086 程序段
V86CodeSEG    SEGMENT   PARA   USE16
        ASSUME    CS：V86CODESEG,DS：V86CODESEG
Message             DB   'V86 is OK.',0DH,0AH,24H
V86Begin:                                      ；处于 V86 方式
          MOV     AH,9                         ；显示进入 V86 方式的提示信息
          MOV     DX,OFFSET   Message
          INT     21H
          ；驻留内存方式返回到 DOS
          MOV     AX,RCodeSEG
          SUB     AX,GDTSEG                    ；计算驻留的长度
          MOV     DX,OFFSET TSRLINE +16
          SHR     DX,4                         ；以“节”为单位
          ADD     DX,AX
          ADD     DX,10H                       ；含 PSP 的节数
          MOV     AX,3100H
          INT     21H
V86CodeSEG        ENDS
;-------------------------------------------------------------------------------
;临时任务的 TSS 段
TempTSSSEG      SEGMENT    PARA   USE16
          DD      0                            ；链接字
          DD      ?                            ；0 级堆栈指针
          DW      ?,?                          ；总是特权级 0,无堆栈切换
          DD      ?                            ；1 级堆栈指针
          DW      ?,?                          ；
          DD      ?                            ；2 级堆栈指针
          DW      ?,?
          DD      0                            ；CR3
          DW      ?,0                          ；EIP
          DD      0                            ；EFLAGS
          DD      0                            ；EAX
          DD      0                            ；ECX
          DD      0                            ；EDX
          DD      0                            ；EBX
          DW      ?,0                          ；ESP
          DD      0                            ；EBP
          DD      0                            ；ESI
          DD      0                            ；EDI
```

```
        DW      ?,0                         ;ES
        DW      TempCode_SEL,0              ;CS
        DW      ?,0                         ;SS
        DW      ?,0                         ;DS
        DW      ?,0                         ;FS
        DW      ?,0                         ;GS
        DW      0,0                         ;LDT(临时任务不使用LDT)
        DW      0
        DW      $+2                         ;指向I/O许可位图区的指针
        DB      0FFH                        ;I/O许可位图结束字节
TempTSSLEN=     $
TempTSSSEG      ENDS
;------------------------------------------------------------
;临时任务的代码段
TempCodeSEG     SEGMENT   PARA   USE16
        ASSUME  CS:TempCodeSEG
Virtual:  进入保护方式后的入口点
        MOV     AX,TempTSS_SEL
        LTR     AX                          ;加载TR指向临时任务的TSS
        MOV     AX,Normal_SEL               ;准备切换到V86任务
        MOV     DS,AX                       ;给各段寄存器赋适当的选择子
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     SS,AX
        JUMP16  V86TSS_SEL,0                ;转移到V86任务(V86方式)
;从INTFF任务回到临时任务的入口点
ToReal: CLTS
        MOV     EAX,CR0
        AND     EAX,0FFFFFFFEH
        MOV     CR0,EAX                     ;返回实方式
        JUMP16  <SEG  Real>,<OFFSET Real>
TempCodeSEG     ENDS
;============================================================
;实方式下的初始化代码和数据
RCodeSEG    SEGMENT    PARA    USE16
VGDTR       PDESC      <GDTLEN-1,>          ;伪GDTR
VIDTR       PDESC      <IDTLEN-1,>          ;伪IDTR
NORVIDTR    PDESC      <3FFH,0>             ;保存实方式下的IDTR
SPVAR       DW         ?                    ;保存实方式下的堆栈指针
SSVAR       DW         ?
        ASSUME  CS:RCodeSEG,DS:RCodeSEG
```

```
Start:
        MOV     AX,RCodeSEG
        MOV     DS,AX
        CLD
        CALL    INIT_GDT                ;初始化 GDT
        CALL    INIT_IDT                ;初始化 IDT
        MOV     AX,V86LDTSEG
        MOV     FS,AX
        MOV     CX,V86LDNUM
        MOV     SI,OFFSET  VLDT
        CALL    INIT_LDT                ;初始化 V86 任务的 LDT
        MOV     AX,INTFFLDTSEG
        MOV     FS,AX
        MOV     CX,INTFFLDNUM
        MOV     SI,OFFSET  FLDT
        CALL    INIT_LDT                ;初始化 INTFF 任务的 LDT
        MOV     SSVAR,SS                ;保存实方式下的堆栈指针
        MOV     SPVAR,SP
        LGDT    QWORD  PTR  VGDTR       ;装载 GDTR
        SIDT    NORVIDTR                ;保存 IDTR
        CLI
        LIDT    QWORD  PTR  VIDTR       ;装载 IDTR
        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX                 ;转保护方式下的临时任务
        JUMP16  <TempCode_SEL>,<OFFSET Virtual>
        ;
Real:   ;从保护方式回到实方式时的入口点
        MOV     AX,CS
        MOV     DS,AX
        LIDT    NORVIDTR                ;恢复 IDTR
        LSS     SP,DWORD  PTR  SPVAR
        STI
        MOV     AX,4C00H                ;结束发出 INT FF 指令的 DOS 程序
        INT     21H
TSRLINE     LABEL  BYTE
;----------------------------------------------------------------
;实方式下的初始化过程
INIT_GDT     PROC   NEAR
        ;同实例四
INIT_GDT     ENDP
;初始化局部描述符表的过程
```

```
INIT_LDT     PROC
        ;同实例四
INIT_LDT     ENDP
;初始化 IDT 表及伪 IDTR 的过程
INIT_IDT     PROC
        PUSH    DS
        MOV     AX,IDTSEG
        MOV     DS,AX
        MOV     CX,256-1                    ;对 FFH 号特殊处理
        MOV     SI,OFFSET  IDT
        MOV     AX,OFFSET  TPBegin
IIDT1:  CMP     CX,256-1-13
        JZ      IIDT2                       ;对 13 号特殊处理
        MOV     [SI],AX
IIDT2:  ADD     SI,8                        ;每个门描述符 8 字节
        ADD     AX,7                        ;处理程序开始部分长 7 字节
        LOOP    IIDT1
        POP     DS
        ;
        MOV     BX,16
        MOV     AX,IDTSEG
        MUL     BX                          ;设置伪 IDTR
        MOV     WORD  PTR  VIDTR.BASE,AX
        MOV     WORD  PTR  VIDTR.BASE+2,DX
        RET
INIT_IDT     ENDP
RCodeSEG     ENDS
        END     Start
```

2. 说明

(1) 对 IDT 表的初始化

为了方便地书写 IDT 表，采用了重复汇编。从采用重复汇编方式定义的 IDT 表可见，除对应通用保护异常的 13 号陷阱门描述符和 255 号任务门描述符外，其他中断门描述内的偏移均未设定。为此，在实方式下初始化时，把相应的入口偏移填入这些门描述符。从源程序可见，这些处理程序的入口片段也用重复汇编书写，并且字节数相同，所以间隔等长。

(2) 任务切换方式进入 V86 方式

实例从临时任务切换到 V86 任务时进入 V86 方式。在 V86 任务的 TSS 中，EFLAGS 字段内的 VM=1，所以随着任务切换，就进入 V86 方式。为此，TSS 中对应各段寄存器字段内的初始值都是 V86 方式下要执行的 8086 程序的各段值，而非选择子。由于在发生中断/异常时要进入 V86 任务的特权级 0，所以初始化了 0 级堆栈指针。V86 任务使用局部

描述表 LDT，所以 TSS 中对应字段填有相应的选择子。

(3) V86 方式下对中断的处理

实例对 V86 方式下响应中断和执行软中断指令“INT n”的处理方法是，转实方式下的对应中断处理程序。具体步骤如下：①在 V86 方式堆栈(V86 任务的 3 级堆栈)顶形成返回点的现场；②用实方式下对应的中断向量值代替返回地址；③从保护方式返回 V86 方式。由于堆栈中保存的 EFLAGS 内的 VM＝1，所以在保护方式下执行 IRET 指令时，返回 V86 方式；由于在 0 级堆栈中保存的返回地址(段值和偏移)已被修改成实方式下的中断向量，所以这时的返回也就是转入实方式下的对应中断处理程序；由于在 V86 堆栈顶已安排了返回地址，所以在实方式下执行对应中断处理程序时，遇到 IRET 指令就返回到 V86 任务的被中断处。这种处理方法似乎绕了个弯。但就是利用这个方法有效地调用了 DOS 功能，方便地显示了提示信息“V86 is ok.”，顺利地实现了驻留退出。

所以，除为了确定中断向量号在中断处理程序之初的代码不同外，其他代码可重复利用。这也是可用重复汇编和循环填写偏移初始化的条件。

这种处理方法没有充分考虑异常，更没有考虑异常时的出错码。

(4) V86 任务的通用保护异常处理

在演示的第一阶段和第二阶段，不会发生任何异常。但在这两个阶段之间，由于允许执行其他程序，可能引起异常。为了简单，实例只考虑了通用保护异常，而未考虑其他异常。该可能发生异常的阶段是在 V86 方式下。如果发生通用保护异常，那么就转入 V86 任务的通用保护异常处理程序。通用保护异常处理程序在保护方式下显示提示信息“……General Protection Error……”，然后再转 21H 号中断处理程序，通过设置入口参数 AX＝4C01H，中止引起通用保护异常的程序。

(5) INTFF 任务

为了演示以任务切换方式离开 V86 方式，在实例中安排了这一任务，并称之为 INTFF 任务。由于 IDT 表中的 255(0FFH)号描述符是任务门描述符，所以当在 V86 方式下执行“INT FFH”时便从 V86 方式切换到 INTFF 任务。INTFF 任务的 TSS 是初始化好的，在 INTFF 任务下不发生特权级变换，不使用局部描述表 LDT。INTFF 任务先显示提示信息“Return to real mode.”，然后再切换到临时任务。

10.11.4 V86 方式下的敏感指令

在 V86 方式下，CPL＝3，执行 10.8.5 节所述特权指令或者要引起出错码为 0 的通用保护故障，或者要引起非法操作码故障。

在 V86 方式下，表 10.9 所列的指令仍是 I/O 敏感指令，但输入/输出指令的敏感条件有所变化。指令 CLI 和 STI 的敏感条件不变，由于 CPL＝3，所以如果 IOPL＜3，那么执行 CLI 或 STI 指令引起通用保护故障。输入/输出指令 IN、INS、OUT 或 OUTS 的敏感条件仅仅是当前 V86 任务 TSS 内的 I/O 许可位图，而忽略 EFLAGS 中的 IOPL。输入/输出指令 IN、INS、OUT 或 OUTS 是否可以执行与 CPL 是否小于 IOPL 无关，而直接由 I/O 许可位图对应位决定，如果输入/输出指令所使用 I/O 地址对应的 I/O 许可位图内的各位都为 0 时，输入/输出指令可正常执行，否则引起通用保护故障。

此外，在 V86 方式下，指令 PUSHF、POPF、INT n 和 IRET 却对 IOPL 敏感。也就是说，在 V86 方式下，当 IOPL<3 时，执行指令 PUSHF、POPF、INT n 及 IRET 会引起出错码为 0 的通用保护故障，并非象 10.9.2 节所述保持沉默。

采取这些措施的目的是使操作系统软件可以支持一个"虚拟 EFALG"寄存器。

10.12 习　　题

题 10.1　80386 的哪些功能只有在保护方式下才能起作用？

题 10.2　80386 的物理地址空间有多大？虚拟地址空间有多大？是如何计算的？

题 10.3　有选择子和偏移构成的逻辑地址如何转换成物理地址？

题 10.4　80386 的四个特权级是如何划分的？哪级最高？哪级最低？

题 10.5　在保护方式下，80386 如何定义一个段？

题 10.6　按描述符所描述的对象来划分，80386 有哪几类描述符？

题 10.7　有哪些门描述符？

题 10.8　长度最大的段可达多少？如何表示？

题 10.9　符号 CPL、RPL、DPL 代表什么？它们之间有何关系？

题 10.10　在 80386 的某个时刻，全局描述符表 GDT、局部描述符表 LDT 和中断描述符表 IDT 各有几张？

题 10.11　描述符表的最大有效段界限是多少？

题 10.12　请写出在 10.2.3 节中所列描述符表 DESTAB 中的 6 个描述符所描述的各个段的段基地址、段界限和段属性。

题 10.13　如何实现某个段被两个任务共享，但又不被第三个任务所共享？

题 10.14　选择子与段值有何区别？

题 10.15　80386 控制寄存器的作用是什么？系统地址寄存器的作用是什么？

题 10.16　段描述符高速缓冲寄存器有何作用？

题 10.17　在从保护方式切换到实方式时要注意什么？

题 10.18　在从实方式切换到保护方式时要做哪些准备工作？

题 10.19　32 位段与 16 位段的区别是什么？代码段描述符如何描述 32 位代码段和 16 位代码段？

题 10.20　任务状态段的作用是什么？

题 10.21　在保护方式下，控制转移有哪些情形？通过转移指令 JMP 的转移与通过调用指令 CALL 的转移有和区别？

题 10.22　如何体现特权级变换？特权级变换必须满足什么条件？

题 10.23　特权级变换时为什么要切换堆栈？如何切换堆栈？

题 10.24　如何体现任务切换？

题 10.25　什么时候发生任务切换？简述任务切换过程。

题 10.26　如何实现任务嵌套？

题 10.27　别名技术是指什么？

题 10.28 80386 的中断和异常有何异同？

题 10.29 80386 异常分为哪三类？各有什么特点？

题 10.30 在发生中断和异常后，如何转入对应的处理程序？

题 10.31 如何防止应用程序执行“INT n”指令时，使用了分配给各种设备用的中断向量号？

题 10.32 特权指令的特权指的是什么？为什么要有特权指令？

题 10.33 为什么要实施输入/输出保护？80386 如何实现输入/输出保护。

题 10.34 请说明线性地址到物理地址的转换过程。

题 10.35 80386 如何对页面进行保护？

题 10.36 V86 方式是指什么？为什么需要 V86 方式？

题 10.37 在 V86 方式下，是否可利用 IRET 指令离开 V86 方式？

题 10.38 在保护方式下，80386 提供了哪些保护措施？

题 10.39 80386 对实现虚拟存储器有何支持？

题 10.41 请画出各实例的内存映象。

题 10.42 请实际调试各实例。

第 11 章　80486 及 Pentium 程序设计基础

Intel 的 80486 和 Pentium 是 80x86 家族的新成员，它们保持与 80386 的兼容。本章在前两章的基础上介绍 80486 和 Pentium。本章不涉及浮点处理部件方面的内容

11.1　80486 程序设计基础

80486 是 80x86 家族中继 80386 之后又一种功能更强大的 32 位微处理器。80486 有 80486DX 和 80486SX 两款，80486DX 是在 80386 的基础上集成浮点处理部件和超高速缓存而构成的，80486SX 不包含浮点处理部件。由于我们几乎不涉及浮点处理方面的内容，所以把它们简单地统称为 80486，而不加区分。

从程序设计的角度看，80486 只比 80386 多了几个控制位和 6 条指令，所以第 9 章和第 10 章介绍的内容对 80486 仍有效。

11.1.1　寄存器

80486 不仅兼容 80386，而且还在片上集成了相当于 80387 的浮点处理部件，所以 80486 含有 80386 和 80387 所拥有的全部 32 位寄存器。这些寄存器可分为如下几类：(1)基本结构寄存器，包括通用寄存器、段寄存器、指令指针和标志寄存器。(2)系统级寄存器，包括控制寄存器和系统地址寄存器。(3)浮点处理部件 FPU 寄存器，包括数据寄存器、标志字、状态字、控制字、指令和数据指示字。(4)调试和测试寄存器。

1. 基本结构寄存器及标志寄存器

80486 的 8 个 32 位的通用寄存器与 80386 相同，6 个段寄存器与 80386 相同，指令指针也与 80386 相同。请参见图 9.1。

80486 的标志寄存器 EFLAGS 仍是 32 位，如图 11.1 所示。与图 9.2 所示的 80386 标志寄存器相比，80486 的标志寄存器新增了一个对齐检查标志 AC。其他标志位的位置及意义保持与 80386 相同。

31　…　19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留为0	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

图 11.1　80486 标志寄存器

标志位 AC 参与控制地址不对齐异常的发生。所谓地址不对齐是指如下情形：访问一个奇地址的字，或访问地址不是 4 的倍数的双字等等。如果 AC 置 1，那么当出现地址不对齐情形时，引起地址对齐异常。但在特权级 0、1 和 2 运行时，忽略 AC 位的设置，在 CR0

中的 AM 位为 1 时也忽略 AC 位的设置。

地址对齐异常是 80486 新设置的异常，属于故障类异常，向量号规定为 11H。80486 地址对齐异常提供出错码 0。只有在特权级 3 运行的应用程序才可能引起地址对齐故障。

2. 系统级寄存器及控制寄存器

系统地址寄存器是指全局描述符表寄存器 GDTR、局部描述符表寄存器 LDTR、中断描述符表寄存器 IDTR 和任务状态段寄存器 TR。80486 的这些系统地址寄存器与 80386 对应的系统地址寄存器相同。请参见图 10.9。

80486 仍只包括 3 个控制寄存器 CR0、CR2 和 CR3。CR2 用于指示引起页面故障的线性地址。CR0 新设了 5 个控制位。CR3 新设了 2 个控制位。

(1) 控制寄存器 CR0

图 11.2 给出了 80486 控制寄存器 CR0 各位的定义。与如图 10.9 所示的 80386 控制寄存器 CR0 相比较，80486 的 CR0 新定义了如下控制位：用于控制片上超高速缓存工作方式的 CD 位和 NW 位；对齐屏蔽位 AM；页面写保护位 WP；数字异常位 NE。

31	30	29	28 19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PG	CD	NW	保留为0	AM	0	WP	0	0	0	0	0	0	0	0	0	0	NE	ET	TS	EM	MP	PE

图 11.2 80486 控制寄存器 CR0

PE 位控制 80486 工作于实方式还是保护方式，PG 位控制是否启用分页机制，它们的作用与 80386 保持兼容，它们组合定义的处理器工作方式如表 10.3 所列。

由于 80486 含浮点处理部件，所以处理器扩展类型位 ET 总是 1。

位 TS、EM 和 MP 的作用与 80386 保持兼容。只是它们控制片上浮点处理部件。

新设的数字异常位 NE 控制通过哪种方式报告未屏蔽的浮点部件出错故障。NE=0，采用外部中断方式报告。这种方式是系统复位时的缺省方式，保持与先前微机系统的处理方式相一致。也即当浮点部件出错时，导致中断向量号为 0DH 的外部中断。NE=1，通过引起浮点部件出错故障报告，对应中断向量号为 10H。请参见 10.7.2 节。

新设的对齐屏蔽位 AM 控制标志寄存器 EFLAGS 中的对齐检查标志 AC 是否有效。AM=0，忽略 AC 位。这是系统复位时的缺省状态，以便保持与 80386 兼容。AM=1，考虑 AC 位，这时才可能引起地址对齐异常。

新设的页面写保护位 WP 控制系统级程序写访问只读页面。80386 允许系统特权级(0 级、1 级和 2 级)程序写访问只读页面，请参见 10.10.3 节。在 80486 中，这种情况受到 WP 位的控制。WP=0，保持与 80386 兼容，这是系统复位时的缺省状态。WP=1，任何特权级程序向只读页面写访问，都将引起页故障。

新设的片上超高速缓存控制位 CD 控制是否允许超高速缓存填充。CD=0，允许片上超高速缓存填充。CD=1，禁止片上超高速缓存填充。

新设的片上超高速缓存直写方式控制位 NW 控制是否采用直写方式。NW=1，采用直写方式和允许使无效，这是系统复位时的缺省状态。NW=0，禁止直写方式及使无效。

关于这两个控制位的详细说明请参见 11.1.4 节。

为了严格与先前的处理器兼容，装入机器状态字指令 LMSW，不能改变 ET 位和 NE 位。

(2) 控制寄存器 CR3

如图 10.9 所示，CR3 的高 20 位是页目录表所在物理页的页码。在 80386 中，CR3 的低 12 位保留未用。80486 在 CR3 的低 12 位中定义了 2 个新的控制位 PCD 和 PWT，如图 11.3 所示。

31 … 12	11							4	3			0
页目录表所在物理的页码	0	0	0	0	0	0	0	PCD	PWT	0	0	0

图 11.3　80486 的控制寄存器 CR3

80486 的页目录表和页表项也在位 4 和位 3 的位置定义了 PCD 和 PWT 位，也就是说在如图 10.28 所示的表项格式的位 4 是 PCD 位，位 3 是 PWT 位。

控制寄存器 CR3、页目录表和页表中的这两个控制位参与控制页面可超高速缓存性。

3. 调试和测试寄存器

80486 象 80386 一样含有 6 个调试寄存器，它们分别是 DR0、DR1、DR2、DR3、DR6 和 DR7。在 11.2 节介绍调试寄存器的使用。

80486 含有 5 个测试寄存器。TR6 和 TR7 用于支持转换后援缓冲器 TLB 的测试，这与 80386 相同。TR3、TR4 和 TR5 用于测试片上超高速缓存。

11.1.2　指令系统

80486 的指令集是在 80386 指令集的基础上增加了 6 条新指令。所以，80486 的指令集包含了 80386 的指令集，并保持与 80386 兼容。新增的指令主要用于片上高速缓存的清洗和对多处理器系统的支持。下面就只介绍新增的 6 条指令。

1. 字节交换指令 BSWAP

字节交换指令的格式如下：

```
BSWAP   OPRD
```

其中操作数 OPRD 是任一 32 位通用寄存器。字节交换指令 BSWAP 的功能是在操作数 OPRD 内交换 4 个字节的顺序。交换对应关系是：第 0 字节与第 3 字节交换，第 1 字节与第 2 字节交换，如图 11.4 所示。

例如：

```
BSWAP   EAX        ;设 EAX=11223344H,执行后 EAX=44332211H
BSWAP   ESI        ;设 ESI=87654321H,执行后 ESI=21436587H
```

该指令不影响各标志。

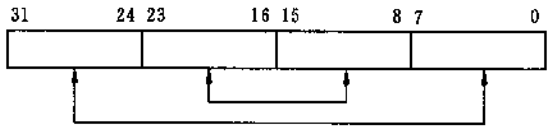


图 11.4 指令 BSWAP 字节交换顺序

80x86 系列处理器按“高高低低”的原则存储多字节数据，但某些处理器按“低低高高”原则存储数据。BSWAP 指令特别适宜于这两种数据格式之间的转换。

2. 交换加指令

交换加指令的格式如下：

```
XADD    OPRD1,OPRD2
```

交换加指令 XADD 的功能是交换操作数 OPRD1 和 OPRD2 的内容，并把两个操作数相加结果送到操作数 OPRD1 中。其中操作数 OPRD1 可以是 8 位、16 位或 32 位通用寄存器或者存储单元，操作数 OPRD2 只能是 8 位、16 位或 32 位通用寄存器。两个操作数的尺寸必须一致。

例如：

```
XADD    AL,AH           ;设 AX=1122H,执行后 AX=2233H
XADD    [BX],ESI
XADD    [ECX+3],AL
```

XADD 指令按照一条相当于 ADD 指令的操作设置标志寄存器中的各运算结果标志。

XADD 指令的功能相当于连续的交换指令 XCHG 和加运算指令 ADD 的功能。但该指令能更好地实现信号量操作。例如：

```
MOV     AL,1            ;信号量增值
XADD    Sema,AL         ;增加
JC      Failed          ;未到转,但已取得原信号量值
```

当在 XADD 指令前加 LOCK 前缀时，能方便地实现多处理器场合的信号量操作。

3. 比较交换指令 CMPXCHG

比较交换指令的格式如下：

```
CMPXCHG    OPRD1,OPRD2
```

其中操作数 OPRD1 可以是 8 位、16 位或 32 位通用寄存器或者存储单元；操作数 OPRD2 只能是通用寄存器，并且尺寸必须与 OPRD1 相一致。比较交换指令的功能是把对应尺寸的累加器(EAX、AX、AL)与操作数 OPRD1 比较，如果相等，把操作数 OPRD2 的内容送操作数 OPRD1，并置零标志 ZF；如果不等，把操作数 OPRD1 的内容送累加器，并清零标志 ZF。

例如：

```
CMPXCHG    BL,CL
CMPXCHG    [EDX],SI
CMPXCHG    [SI],EAX
```

该指令按比较结果影响有关标志。

在加上 LOCK 前缀后，该指令对多处理器情况下的信号量操作非常有用。

4. 使超高速缓存无效指令 INVD

使超高速缓存无效指令的格式如下：

```
INVD
```

该指令使片上超高速缓存无效，也即清洗片上超高速缓存。该指令也产生一个特殊的总线周期，它可以用于使外部(二级)超高速缓存无效。注意，该指令不把片上超高速缓存中的内容写回主存，所以使用时必须十分小心，通常应使用 WBINVD 指令。

该指令不影响各标志。

该指令是特权指令。只有在实方式和保护方式的特权级 0 下，才可执行该指令。

我们在 11.1.3 节介绍片上超高速缓存的有关内容，和举例说明 INVD 指令的使用。

5. 写回并使超高速缓存无效指令 WBINVD

写回并使超高速缓存无效指令的格式如下：

```
WBINVD
```

该指令使片上超高速缓存无效，也即清洗片上超高速缓存，但在清洗前把片上超高速缓存中更改的内容写回主存。该指令会产生特殊的总线周期，指示把外部超高速缓存中更改的内容写回主存和指示外部超高速缓存无效。

通常应该使用该指令清洗片上超高速缓存。

该指令不影响各标志。

该指令是特权指令。只有在实方式和保护方式的特权级 0 下，才可执行该指令。

6. 使 TLB 项无效指令 INVLPG

我们知道，在启用分页机制的情况下，分页部件利用页目录表和页表把线性地址转换成物理地址。为了加快转换速度，80386 和 80486 片上都有转换后援缓冲器 TLB。TLB 含 32 个项，用于存放当前最常使用的物理页的页码。

使 TLB 项无效指令的格式如下：

```
INVLPG    OPRD
```

其中操作数 OPRD 必须是存储器操作数。该指令的功能是，如果存储器操作数 OPRD 能通过 TLB 中的某项转换成物理地址，那么使 TLB 内对应项无效。

例如：

```
INVLPG    [BX]
INVLPG    [ECX*2+30H]
```

该指令不影响各标志。

该指令是特权指令。只有在实方式和保护方式的特权级 0 下,才可执行该指令。

只有在特殊情况下才使用该指令,强制更新某些 TLB 项。

11.1.3 片上超高速缓存

为了进一步提高性能,80486 在片上带有 8K 字节的超高速缓存器。它对软件是透明的,以保证与 80x86 系列先前的处理器兼容。尽管超高速缓存器是透明的,但了解它的组织和工作方式,对优化程序很有益。下面简单介绍片上超高速缓存的组织和工作方式,同时说明控制寄存器 CR0 中控制位 CD 和 NW 等的作用。

1. 片上超高速缓存及其工作方式的控制

80486 的 8K 字节片上超高速缓存既能存储数据又能存储代码(指令)。8K 字节容量指能用于存储数据或指令的容量,而不包括用于存储地址标记等的容量。

片上超高速缓存采用 4 路组相关联结构,在物理上分成 4 个 2K 字节的块。每块由 128 行构成,每行 16 字节宽。在逻辑上,分成 128 组,每组 4 行。每行都有一个 21 位的标记相关联,记录每行与主存储器中存储单元的对应关系,这 21 位的标记相当于主存储器中存储单元物理地址的高端部分。每一行都有一个有效位相关联,每行不是有效就是无效,没有部分有效的行。每行还有用于记录最近最少使用情况的 LRU 位相关联。

超高速缓存命中是指欲访问的存储单元地址作为有效标记部分出现在超高速缓存中。如果是读命中,那么直接从片上超高速缓存中读出,从而大大提高速度。如果读未命中,那么通常会把该存储单元所在行填入超高速缓存。如果写命中超高速缓存,那么 80486 通常(采用直写方式时)不仅向超高速缓存相应单元写,同时也向主存储器相应单元写。如果写未命中,那么直接写入主存储器相应单元,不影响超高速缓存。

片上超高速缓存的工作方式由控制寄存器 CR0 中的 CD 位和 NW 位控制。其中,CD 位允许和禁止填充片上超高速缓存,NW 位控制直写和使无效。由 CD 位和 NW 位组合定义的片上超高速缓存的工作方式比较灵活,列于表 11.1。尽管两位可表示 4 种方式,但 CD=0 和 NW=1 的组合是无效组合,会引起通用保护故障。在 RESET 后的缺省组合是 CD=1 和 NW=1,超高速缓存为空。

表 11.1 片上超高速缓存的工作方式

CD	NW	片上超高速缓存工作方式
0	0	允许超高速缓存填充,允许直写和使无效
0	1	无效组合,导致出错码为 0 的通用保护故障
1	0	禁止超高速缓存填充,允许直写和使无效
1	1	禁止超高速缓存填充,禁止直写和使无效

CD=0 允许超高速缓存填充。当读访问存储单元时,如果未命中片上超高速缓存,那么所读存储单元所在行通常就会被填充到超高速缓存,并且物理地址的高端部分就作为该行的标记。在处理器外部,通过其他途径可以阻止某些存储单元不被超高速缓存,这是不通常情况。在把某行填充到超高速缓存时,首先检查该组中所有的 4 行是否都有效,如

果有无效行，则更新该行，如果都有效，那么采用最近最不常使用算法确定更新哪一行。

CD＝1 禁止超高速缓存填充。当读访问存储单元未命中片上超高速缓存时，不把所在行填充到超高速缓存。不影响读命中超高速缓存，不影响写命中超高速缓存和写未命中超高速缓存的动作。

NW＝0 允许直写和使无效。当写访问命中超高速缓存时，所写信息写入片上超高速缓存，同时驱动一个外部写总线周期，更新外部对应存储单元。80486 集成有监视其他外部系统写入主存储器的逻辑，当它检查到其他外部系统写入主存储器的存储单元命中片上超高速缓存时，就使片上超高速缓存的对应行无效。这就是使无效。它在保证超高速缓存的内容与对应主存储器的内容保持一致方面起重要作用。

NW＝1 禁止直写和使无效。当写访问命中超高速缓存时，更新片上超高速缓存相应行，但不更新外部对应存储单元。当其他外部系统更新主存储器的存储单元命中片上超高速缓存时，并不使片上超高速缓存对应行无效。这种处理方法会导致片上超高速缓存内容与对应主存储器的内容不一致。

2. 演示片上超高速缓存作用的实例

下面给出一个演示片上超高速缓存如何提高速度性能的演示程序。演示程序含有一个循环访问某些存储单元的测试子程序，该测试子程序还能测定运行所耗时间。演示程序在允许和禁止片上超高速缓存两种情况下，分别调用该测试子程序，然后显示两种情况下运行同一子程序所耗时间。该演示程序在 80486 实方式下运行。源程序清单如下：

```
;程序名：T11-1.ASM
;功  能：演示片上高速缓存的作用
;说  明：仅在 80486 实方式下运行(用 TASM 汇编，用 TLINK 连接)
CDBIT     =   30                          ;CR0 种的 CD 位位置
COUNT     =   100
          .486P                           ;识别 486 指令集
CSEG      SEGMENT   PARA   USE16
CacheD    DB      'Cache  Disable: $'
CacheE    DB      'Cache  Enable: $'
          ASSUME    CS:CSEG
BEGIN:    PUSH    CS
          POP     DS                      ;使数据段同代码段
Step1:    ;在禁止片上超高速缓存的情况下调用测试子程序
          CLI
          MOV     EAX,CR0
          BTS     EAX,CDBIT
          MOV     CR0,EAX                 ;置 CD 位，禁止片上超高速缓存
          INVD                            ;清洗片上超高速缓存
          CALL    Access                  ;调用测试子程序
          STI
          MOV     ESI,EDX                 ;保存所耗时间参数
```

```
Step2:    ;在允许片上超高速缓存的情况下调用测试子程序
          CLI
          MOV    EAX,CR0
          BTR    EAX,CDBIT
          MOV    CR0,EAX                 ;清 CD 位,允许片上超高速缓存
          INVD                           ;清洗片上超高速缓存
          CALL   Access                  ;调用测试子程序
          STI
          MOV    EDI,EDX                 ;保存所耗时间参数
Step3:                                   ;显示两种情况下的所耗时间表示值
          MOV    DX,OFFSET  CacheD
          MOV    ECX,ESI
          CALL   DMESS
          MOV    DX,OFFSET  CacheE
          MOV    ECX,EDI
          CALL   DMESS
Over:     ;结束
          MOV    AH,4CH
          INT    21H
;
;过程名称: Access
;功    能: 测试允许和禁止片上超高速缓存效率
;入口参数: 无
;出口参数: EDX 含所耗时间参数
;             高 16 位是开始时间表示值,低 16 位是结束时间表示值
Access    PROC
          MOV    CX,COUNT
          MOV    EBX,16
          CALL   GetCount                ;取时间表示值
          SHL    EDX,16                  ;保存到 EDX 高 16 位
ACC1:     MOV    EAX,[EBX]
          MOV    EAX,[EBX * 2]
          MOV    EAX,[EBX+EBX * 2]
          LOOP   ACC1
          CALL   GetCount                ;再取时间表示值,保存在 DX 中
          RET
Access    ENDP
;
;过程名称: GetCount
;功    能: 读系统定时计数器 0,取得时间表示值
;入口参数: 无
;出口参数:DX=时间表示值
```

```
GetCount      PROC
          MOV    AL,0
          OUT    43H,AL                   ;选定系统定时器 0
          CALL   DELAY
          IN     AL,40H                   ;读计数值低 8 位
          MOV    DL,AL
          CALL   DELAY
          IN     AL,40H                   ;读计数值高 8 位
          MOV    DH,AL
DELAY:    RET
GetCount      ENDP
;
;过程名称:DMESS
;功    能:显示说明信息和所用时间表示值
;入口参数:DX=提示信息开始地址偏移
          ECX=含时间表示值(高 16 位是开始表示值,低 16 位是结束表示值)
;出口参数:无
DMESS     PROC
          MOV    AH,9
          INT    21H                      ;显示提示信息
          SHLD   EDX,ECX,16
          SUB    DX,CX                    ;得所耗时间表示值
DMESS1:MOV       AX,DX
          CALL   DHEX                     ;以 16 进制数形式显示
          CALL   NEWLINE                  ;回车换行
          RET
DMESS     ENDP
;略去按十六进制数形式显示 DX 之内容的过程 DHEX
;略去形成回车换行的过程 NEWLINE
CSEG      ENDS
          END    BEGIN
```

上述演示程序的演示过程是清楚的。现就测试子程序所耗时间表示值的测定作些说明。PC 及其兼容机系统中的定时计数器芯片支持多个独立的计数器,其中 0 号计数器用于形成系统软时钟。计数器采用减计数方式,每当计数到 0 时就自动重新按初始化时设置的计数初值开始下一轮计数。计数器 0 的计数初值是 65536。读取计数器 0 的方法是把计数器号送到定时计数器控制端口 43H,然后分两次从 40H 端口读取计数器计数值。测试子程序在开始循环前读取计数值,在结束循环后再读取计数值。这两个计数值可认为是开始和结束的时间表示,它们的差能够反映测试子程序所耗的时间,但不等于所耗时间,所以我们称之为时间表示值。通常定时计数器的输入频率固定为 1193180,根据该计数频率可计算出测试子程序所耗的时间。顺便说一下,时钟中断产生的间隔时间大约 55ms 就是

根据该输入频率计算出的。

请注意，由于 Pentium 片上超高速缓存通常情况下不采用直写方式，而采用回写方式，所以上述演示程序在以 Pentium 为处理器的系统上运行可能要出问题。如要在 Pentium 上运行，那么把程序中的清洗片上超高速缓存的指令 INVD，改成指令 WBINVD。

3. 演示控制位 NW 作用的子程序

上述演示程序清楚地说明了 CR0 中控制位 CD 的作用，下面的程序片段能说明控制位 NW 的作用，也反映禁止直写可能导致超高速缓存与主存储器相应存储单元不一致的情况。所以，在禁止超高速缓存和禁止直写和使无效时，应清空超高速缓存。

```
        ;……
        ;只能在 80486 实方式下调试
        ENTER   2,0                      ;在堆栈中安排一个临时字变量
        CLI
        INVD                             ;清洗超高速缓存
LINE1:  MOV     BL,[BP-2]                ;读访问该临时字变量(未命中而被填充)
        MOV     BYTE PTR [BP-2],1        ;命中,也写入主存储器
        MOV     EAX,CR0
        BTS     EAX,30
LINE2:  BTS     EAX,29
        MOV     CR0,EAX                  ;使 CD=1 和 NW=1
        MOV     BYTE PTR [BP-2],5        ;命中,但不写入主存储器
        MOV     CL,[BP-2]                ;命中,CL=5
LINE3:  INVD                             ;清洗超高速缓存
        MOV     BL,[BP-2]                ;没有命中,BL=1
        BTR     EAX,30
        BTR     EAX,29
        MOV     CR0,EAX                  ;使 CD=0 和 NW=0
        STI
        LEAVE
        ……
```

请考虑分别删除上述程序片段中 LINE1、LINE2 或 LINE3 后的执行情况。

4. 页面可超高速缓存性

在处理器外部，通过对 80486 的输入引脚 KEN# 的控制，可以使访问的存储单元不被超高速缓存。软件可以设置控制位 CD 禁止超高速缓存填充。

在启用分页机制时，软件可以设置页表项中的 PCD 位阻止页面被超高速缓存。设置页目录表中的 PCD 位和 CR3 中的 PCD 位阻止页表和页目录表被超高速缓存。因为存在转换后援缓冲器 TLB，所以通常页表和页目录表不需要超高速缓存。

80486 的输出引脚 PCD 和 PWT 能够用于控制外部高速缓存。引脚 PCD 和 PWT 受页表项、页目录项或者 CR3 中 PCD 位和 PWT 位的驱动。但 CD 能够屏蔽 PCD。总之，

PCD 位能够用于控制页面可超高速缓存性；PWT 位控制外部高速缓存的直写策略，但不影响片上超高速缓存的直写策略。

11.2　80486 对调试的支持

8086/8088 提供断点指令 INT3 和单步标志 TF，调试工具利用它们可以设置断点和实现单步。从 80386 开始，在片上集成了调试寄存器。利用这些调试寄存器不仅可以设置代码执行断点，而且还可以设置数据访问断点；不仅可以把断点设置在 RAM 中，也可以把断点设置在 ROM 中。所以说，这些调试寄存器能提供调试便利和简化调试过程。

80486 的调试功能包括了 80386 的调试功能并稍有扩充。关于断点指令和单步与 8086/8088 基本相一致。本节主要介绍 80486 通过调试寄存器提供的支持调试能力。

11.2.1　调试寄存器

80386 和 80486 都支持 6 个调试寄存器，如图 11.5 所示。它们分别是断点地址寄存器 DR0、DR1、DR2 和 DR3，调试状态寄存器 DR6 和调试控制寄存器 DR7。这些断点寄存器都是 32 位寄存器。利用在 10.8.2 节介绍的调试寄存器数据传送指令，可以存取这些调试寄存器。但必须注意，调试寄存器数据传送指令只能在实方式和保护方式的特权级 0 下执行，所以只有在实方式或特权级 0 执行的程序才能设置断点和进行断点处理。

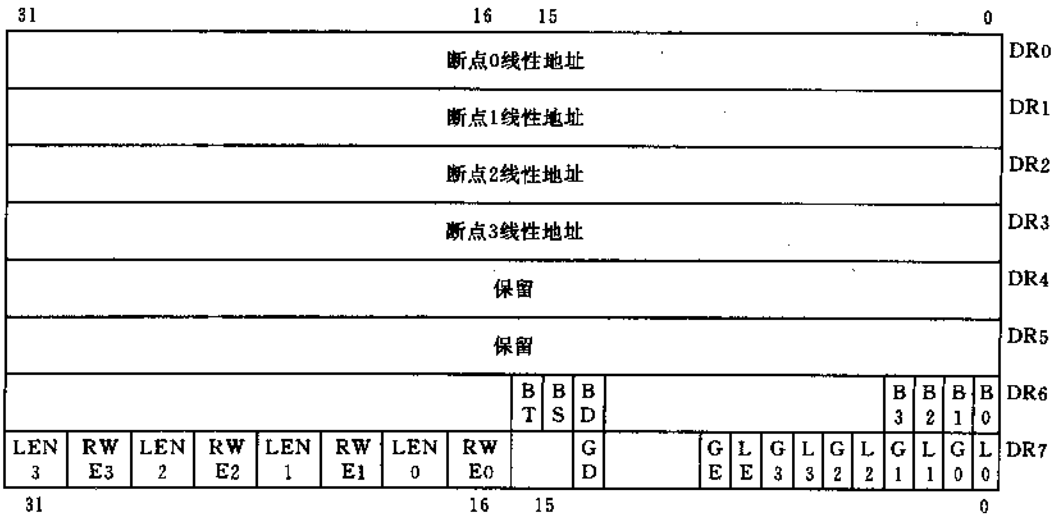


图 11.5　80386/80486 调试寄存器

1. 断点地址寄存器

断点地址寄存器用于保存断点处的线性地址，也即指示断点位置。这些寄存器长 32 位，与 32 位线性地址长度相符。处理器硬件把执行指令所涉及的线性地址和断点地址寄存器内的线性地址进行比较，判别执行指令是否触及断点。处理器具有 4 个断点地址寄存

器 DR0、DR1、DR2 和 DR3，所以可以同时支持 4 个这样的“硬”断点。

因为由片上寄存器指示断点位置，而非断点中断指令指示断点位置，所以这些“硬”断点可以设置在 ROM 中或者几个任务共享的代码中。

不论是否启用分页机制，断点地址寄存器内保存的总是线性地址。我们知道，如果不启用分页机制，那么线性地址就等于物理地址；如果启用分页机制，那么线性地址经过分页部件转换成物理地址。根据线性地址设置断点保证使断点与分页无关。这种断点位置由线性地址表示的做法便于断点的表示，也符合实际调试的需要。

2. 调试控制寄存器

调试寄存器 DR7 也称为调试控制寄存器，其各字段的意义如图 11.5 所示。它不仅控制是否允许断点，还控制各断点是代码执行断点还是数据访问断点。DR7 所含各字段作用如下：

(1) 断点类型说明字段 RWE

DR7 有 4 个 RWE 字段，依次分别对应 4 个断点。RWEi 字段说明 DRi 寄存器所指示断点的类型。每一个 RWE 占两位，所表示的类型列于表 11.2。“00”表示指令执行断点，当执行对应断点地址寄存器所含地址处的指令时，满足断点条件。注意，指令执行断点地址必须等于指令开始的字节地址(包括前缀)。“01”和“11”表示数据访问断点，当按所示读写方式访问对应断点地址寄存器所含地址处的存储单元时，满足断点条件。

表 11.2 RWE 字段说明的断点类型

RWE 字段取值	断点类型(断点条件)	RWE 字段取值	断点类型(断点条件)
0 0	只执行	1 0	未定义(不能用该值)
0 1	只写入数据	1 1	只读或写数据

(2) 断点长度说明字段

DR7 有 4 个 LEN 字段，依次分别对应 4 个断点。LENi 字段说明 DRi 寄存器所指示断点的长度(范围)。每一个 LEN 占两位，所表示的长度列于表 11.3。指令执行断点的断点长度必须为 1 字节。数据访问断点的断点长度可以是 1 字节、2 字节或 4 字节。

表 11.3 LEN 字段说明的断点长度

LEN 字段取值	断点长度(范围)	断点地址寄存器指示的断点地址
0 0	1 字节	全部 32 位指示一个单字节的断点
0 1	2 字节	最低 1 位被忽略，确定字对齐地址开始的 2 字节断点
1 0	未定义(不能取该值)	
1 1	4 字节	最低 2 位被忽略，确定双字对齐地址开始的 4 字节断点

对于数据访问断点而言，如表 11.3 所示，实际上，断点长度说明字段 LEN 和对应断点寄存器规定了断点的区域范围。例如，设 DR1 内的断点线性地址是 XXXXXXX5H，当

LEN1＝00 时，断点的区域范围只有 1 字节，地址就是 XXXXXXX5H；当 LEN1＝01 时，断点的区域范围是地址 XXXXXXX4H 到 XXXXXXX5H 的 2 字节；当 LEN1＝11 时，断点的区域范围是地址 XXXXXXX4H 到 XXXXXXX7H 的 4 字节。当按 RWE 所示数据访问断点方式访问存储单元触及断点范围内的字节时，就满足断点条件。

(3) 全局和局部断点允许位

DR7 有 4 个 Gi 和 Li，分别对应 4 个断点。Gi 和 Li 控制 DRi 所指示的断点 i 在断点条件满足时，是否引起断点异常。当 Gi 或 Li 为 1 时，如果 DRi 所指示数据访问断点条件满足，那么导致进入向量号为 1 的调试陷阱，如果 DRi 所指示指令执行断点条件满足，那么导致引发向量号为 1 的调试故障。

Gi 和 Li 分别称为全局断点允许位和局部断点允许位。在任务切换时，处理器清各 Li 位，所以 Li 位只支持一个任务范围内的断点。任务切换并不影响 Gi 位，所以 Gi 支持系统内各任务的断点。

(4) 精确数据访问断点相符位

DR7 还有 GE 位和 LE 位，用于指示是否要求数据访问断点精确相符。80486 无论 GE 或 LE 是否置位，总是断点精确相符的。在 80386 中，可能出现数据访问断点不精确相符，通过设置 GE 位或 LE 位可指示 80386 数据访问断点精确相符。在任务切换时，LE 位被自动清除，所以 LE 位是局部于任务的；任务切换时 GE 位不受影响，所以 GE 位是全局的。

(5) 全局调试寄存器访问检测位

DR7 的位 13 是调试寄存器访问检测位 GD。尽管只能在实方式或保护方式的特权级 0 时才能访问调试寄存器，但 GD 位还提供对调试寄存器的特别保护。在 GD＝1 的情况下，即使在实方式或保护方式的特权级 0 时，访问任何调试寄存器都会引起向量号为 1 的调试故障。这种附加的保护特性保证调试程序在需要的时候可以完全控制调试寄存器资源。在进入向量号为 1 的异常处理程序时，自动地清除 GD 位，以便异常 1 处理程序能自由地访问调试寄存器。

3. 调试状态寄存器

调试寄存器 DR6 也称为调试状态寄存器，其各位的定义如图 11.5 所示。它指示断点原因，也即指示进入异常 1 处理程序的原因。异常 1 处理程序可根据 DR6 的有关位，确定是数据访问断点、指令执行断点、单步或其他原因。

DR6 内的各指示位为 1 时所表示的意义如下：

Bi＝1(i＝0,1,2,3)表示由 DRi 所指示的断点引起指令执行调试故障或进入数据访问调试陷阱。每当处理器在某个允许的断点 i 处发现断点条件满足，就设置 DR6 中的 Bi，然后转入异常 1 处理程序。要特别注意，在某个允许的断点 i 处断点条件满足而设置 Bi 时，同时设置所有在那瞬间断点条件满足的各对应 Bj 位，而不论是否允许。因此，异常 1 处理程序可能看到多个 Bi 被置位，但可以通过判断断点允许位 Gi 和 Li 位的方法来确定真实的原因。

BD＝1 表示在 GD 置位的情况下访问调试寄存器，从而引起调试故障。在进入异常 1 处理程序时，自动清除 GD 位。

BS＝1 表示由于单步原因进入调试陷阱。当标志寄存器 EFLAGS 中的单步标志 TF 置位时，一般每执行完一条指令后就进入异常 1 处理程序，这就是单步。由于在进入异常处理程序时，自动清 TF 标志。实方式下的情形请参见图 5.6，保护方式下的情形请参见图 10.20，所以，异常 1 处理程序的执行不会再单步。在异常处理程序结束时，中断返回指令 IRET 从堆栈中弹出原标志寄存器内容到标志寄存器，如果 TF 为 1 的话，那么又会产生单步。实际上，在执行置位 TF 指令的下一条指令(通常是 IRET)之后才产生单步。还请参见 5.3.5 节。

BT＝1 表示刚切换到任务状态段 TSS 中的调试陷阱标志 T 置位的任务。任务状态段 TSS 内安排了一个调试陷阱标志 T，从图 10.14 可见，该标志 T 是任务状态段 TSS 内偏移 64H 处的字的最低位。在任务切换时，如果进入任务的 T 位为 1，那么通常在任务切换完成之后，新任务的第一条指令执行之前进入调试陷阱，也即进入异常 1 处理程序。这也称为任务切换自陷。

请注意，硬件根据上述各种情况设置调试状态寄存器 DR6 中的相应标志，但硬件并不自动清除它们。所以，异常 1 处理程序一般应该在返回前清除 DR6，以免发生混淆。

4. 调试故障和调试陷阱的区别

调试异常分为调试故障和调试陷阱两类。数据访问断点、单步和任务切换自陷属于调试陷阱。调试陷阱是在执行引起异常的指令之后发生，进入调试陷阱时，保存在堆栈中的返回地址指向引起陷阱的指令的下一条要执行指令。调试故障是在引起异常的指令之前发生，进入调试故障时，堆栈中的返回地址指向引起故障的指令。

标志寄存器 EFLAGS 中的重启动标志 RF 能控制是否产生调试故障。在把 RF 置成 1 后，下一条指令的任何调试故障被忽略。通常每当成功地执行完一条指令，那么 RF 被清 0。但 IRETD 指令例外，它能根据堆栈中标志寄存器映象的 RF 位值设置 RF。异常 1 处理程序能利用这一特性，在返回断点处时不再重复产生断点故障。

11.2.2 演示调试故障/陷阱的实例

下面给出一个可在实方式下运行的演示调试故障和调试陷阱的实例。

实例程序由两部分组成：异常 1 处理程序和演示程序，分别安排在两个段中。

异常 1 处理程序的处理步骤如下：(1)在屏幕的左上角区域以二进制数的形式显示调试状态寄存器 DR6、指令指针 EIP 的低 16 位部分 IP 和 32 位寄存器 EAX 的内容。(2)调用 BIOS 键盘管理程序，等待按键。该步是让用户能够看清上述显示内容。(3)根据 DR6 和 DR7 判断进入异常 1 处理程序的原因。分调试陷阱和调试故障两种情形结束异常 1 处理程序。对于调试陷阱，直接用中断返回指令 IRET 返回。对于调试故障，通过能够设置标志寄存器中 RF 标志的 IRETD 指令返回，为此还必须在堆栈中形成由二个双字的返回地址和 32 位 EFLAGS 映象的断点现场。

演示程序的执行步骤如下：(1)把异常 1 处理程序的入口填入中断向量表。由于实例在实方式下运行，所以仍使用中断向量表。(2)模拟单步陷阱和访问调试寄存器故障。(3)把演示的断点线性地址装入 DR0、DR2 和 DR3，并设置 DR7。(4)模拟指令执行断点和数据访问断点。

源程序清单如下：

```
;程序名：T11-2.ASM
;功  能：演示调试故障/陷阱的发生及处理
;说  明：用 TASM 汇编，用"TLINK  /3"命令连接。
COLOR   =       17H                     ;显示字符属性值(兰底白字)
        .486P
;调试故障/陷阱处理程序代码段
CSEGD   SEGMENT  PARA   USE16
        ASSUME   CS:CSEGD
Debug:  PUSHAD                          ;保存现场
        PUSH    ES
        CLD
        MOV     DI,0B800H
        MOV     ES,DI                   ;置显示缓冲区段值
        ;在屏幕第一行显示 DR6 之内容
        XOR     DI,DI
        MOV     EAX,(COLOR SHL 8 +'R') SHL 16 + (COLOR SHL 8 +'D')
        STOSD
        MOV     EAX,(COLOR SHL 8 +'=') SHL 16 + (COLOR SHL 8 +'6')
        STOSD
        MOV     EDX,DR6
        BSWAP   EDX
        XCHG    DH,DL
        CALL    ECHOB                   ;显示 DR6 之高 16 位
        BSWAP   EDX
        CALL    ECHOB                   ;显示 DR6 之低 16 位
        ;在屏幕第二行显示 IP 之内容
        MOV     DI,160
        MOV     AX,COLOR SHL 8 +' '
        STOSW
        MOV     EAX,(COLOR SHL 8 +'P') SHL 16 + (COLOR SHL 8 +'I')
        STOSD
        MOV     AX,COLOR SHL 8 +'='
        STOSW
        MOV     DX,[ESP+4*8+2]
        CALL    ECHOB
        ;在屏幕第三行显示 EAX 之内容
        MOV     DI,320
        MOV     EAX,(COLOR SHL 8 +'A') SHL 16 + (COLOR SHL 8 +'E')
        STOSD
        MOV     EAX,(COLOR SHL 8 +'=') SHL 16 + (COLOR SHL 8 +'X')
```

```
        STOSD
        MOV     EDX,[ESP+4*8+2-4]
        ROL     EDX,16
        CALL    ECHOB
        ROL     EDX,16
        CALL    ECHOB
        ;为了演示而设的等待按键
        mov     ah,0
        int     16h
        ;取调试状态寄存器 DR6 和清 DR6
        MOV     EDX,DR6
        XOR     EAX,EAX
        MOV     DR6,EAX
        ;取调试控制寄存器 DR7 及分析调试故障/陷阱原因
        MOV     EAX,DR7
        XOR     EBX,EBX
        MOV     CX,4                    ;4 个断点可能
Debug1: MOV     BP,CX
        DEC     BP
        BT      DX,BP                   ;DR6 中的 Bi 置 1?
        JNC     Debug2                  ;否,转
        ADD     BP,BP
        INC     BP
        BTS     BX,BP
        DEC     BP
        BTS     BX,BP
        TEST    AX,BX                   ;判 Gi/Li
        JZ      Debug2
        SHL     EBX,16
        TEST    EAX,EBX                 ;判是数据访问断点还是指令执行断点
        JNZ     Debug3
        JMP     SHORT   Debug5
Debug2: LOOP    Debug1
        BT      DX,14                   ;判 BS=1?
        JNC     Debug4
        BTR     WORD    PTR  [ESP+4*8+2+4],8
Debug3: ;调试陷阱处理
        POP     ES                      ;恢复现场
        POPAD
        IRET                            ;返回
Debug4:
        BT      DX,13                   ;判 BD=1?
```

```
        JNC     Debug3
Debug5: ;调试故障处理
        POP     ES
        POPAD                               ;恢复现场
        SUB     ESP,6
        PUSH    EAX                         ;在堆栈中形成返回地址
        MOV     AX,[ESP+10]
        MOVZX EAX,AX
        MOV     [ESP+4],EAX
        MOV     AX,[ESP+12]
        MOV     [ESP+8],EAX
        PUSHFD
        POP     EAX
        MOV     AX,[ESP+14]
        MOV     [ESP+12],EAX
        BTS     DWORD PTR [ESP+12],16
        POP     EAX
        IRETD                               ;返回
;以二进制数形式显示 DX 之内容
ECHOB:  MOV     AH,COLOR
        MOV     CX,16
ECHOB1: RCL     DX,1
        SETC    AL
        ADD     AL,'0'
        STOSW
        LOOP    ECHOB1
        RET
CSEGD   ENDS
;
;演示程序代码段
CSEGM   SEGMENT  PARA  USE16
        ASSUME   CS:CSEGM
VARA    DW      0
VARB    DW      4                           ;假设的数据变量
VARC    DW      0
VARD    DB      0
BREAKD    LABEL   BYTE
VARE    DD      0
Begin:  ;把异常 1 处理程序入口填入中断向量表(实方式)
        XOR     AX,AX
        MOV     FS,AX
        MOV     AX,CSEGD
```

```
        SHL     EAX,16
        MOV     AX,OFFSET  Debug
        XCHG    FS:[1*4],EAX
        MOV     ESI,EAX
Step1:  ;模拟单步陷阱
        PUSHF
        BTS     WORD  PTR  [ESP],8
        POPF
        MOV     EAX,1
Step2:  ;模拟访问调试寄存器故障
        MOV     EAX,DR7
        BTS     EAX,13
        MOV     DR7,EAX
        MOV     EAX,2
        MOV     EAX,DR3
Step3:  ;设置 DR0 为指令执行断点
        MOV     AX,CS
        MOVZX EAX,AX
        SHL     EAX,4
        MOV     EBX,OFFSET BRK0
        XADD    EAX,EBX
        MOV     DR0,EAX
        XOR     EDX,EDX
        BTS     EDX,0                   ;L0=1,RWE0=00,LEN0=00
        ;设置 DR2 为数据访问断点
        MOV     EAX,OFFSET  BREAKD
        ADD     EAX,EBX
        MOV     DR2,EAX
        BTS     EDX,4                   ;L2=1
        BTS     EDX,24
        BTS     EDX,25                  ;RWE2=11
        BTS     EDX,26
        BTS     EDX,27                  ;LEN2=11
        ;设置 DR3 为数据访问断点
        MOV     DR3,EAX
        BTS     EDX,7                   ;G3=1
        BTS     EDX,28                  ;RWE3=01
        BTS     EDX,30                  ;LEN3=01
        MOV     DR7,EDX
Step4:  ;模拟指令执行断点
        MOV     EAX,3
BRK0:   MOV     EAX,0
```

```
Step5:  ;模拟数据访问断点(读访问)
        MOV    EAX,DWORD  PTR  CS:VARB
Step6:  ;模拟数据访问断点(写访问)
        PUSH   CS
        POP    DS
        MOV    EAX,5
        MOV    WORD PTR VARD,AX
Step7:  ;清 DR7
        XOR    EAX,EAX
        MOV    DR7,EAX
Over:   ;恢复 1 号中断向量
        MOV    FS:[1*4],ESI
        ;结束
        MOV    AH,4CH
        INT    21H
CSEGM   ENDS
        END    Begin
```

11.3 Pentium 程序设计基础

Pentium 是继 80486 之后的新一代微处理器,它保持与先前各代微处理器百分之百二进制兼容。它不仅采用更先进的技术实现先前微处理器的功能,而且增加了许多新功能,因此它的性能要大大优于 80486。Pentium 实现超标量体系结构,支持分别称为 U 和 V 的两条流水线,理想情况下,在一个时钟周期内可以执行两条指令。Pentium 通过在片内的分支目标缓冲器(BTB),实现动态分支预测,有效地提高分支处理执行速度。Pentium 增加了片上超高速缓存的容量,8K 字节用于数据,8K 字节用于代码。此外,Pentium 还实现了流水线浮点部件,加强了错误检测和报告功能,完善和扩充了虚拟 8086 模式,提供多处理器支持。

我们在先前各章节介绍的内容几乎都适用于 Pentium。本节从程序设计的角度简单介绍 Pentium 新增、扩充或改进的内容。

11.3.1 寄存器

Pentium 维持 80486 绝大部分寄存器的作用和使用方法。但新增了若干 Pentium 模型专用寄存器,这些模型专用寄存器主要用于系统测试和运行性能检测。把它们称为模型专用寄存器的原因是它们与处理器关系密切,可能在以后的处理器中不会以相同的方法继续使用。80486 所含的测试寄存器的功能,在 Pentium 上由模型专用寄存器实现。

1. 基本结构寄存器及其标志寄存器

Pentium 的 8 个 32 位的通用寄存器与 80386 相同,6 个段寄存器也与 80386 相同,指令指针也与 80386 相同。请参见图 9.1。

Pentium 的标志寄存器 EFLAGS 仍是 32 位，如图 11.6 所示。与图 11.1 所示的 80486 标志寄存器相比，Pentium 的标志寄存器新增了三个标志：虚拟中断标志 VIF、虚拟中断挂起标志 VIP 和标识标志 ID。其他标志位的位置及意义保持与 80486 相同。

31 … 22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留为0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

图 11.6 Pentium 标志寄存器

虚拟中断标志 VIF 占用标志寄存器的位 19。当允许虚拟 8086 方式扩充或者允许保护方式虚拟中断时，VIF 是中断标志的虚拟映象。当禁止虚拟 8086 方式扩充和禁止保护方式虚拟中断时，VIF 被强制为 0。

虚拟中断挂起标志 VIP 占用标志寄存器的位 20。当允许虚拟 8086 方式扩充或者允许保护方式虚拟中断时，VIP 指示虚拟中断是否挂起。当禁止虚拟 8086 方式扩充和禁止保护方式虚拟中断时，VIP 被强制为 0。

标识标志 ID 占用标志寄存器的位 21。如果可以设置或清除该标志，那么表示处理器支持 CPUID 指令。利用该指令可获得处理器类型等信息。请参见 T11-3.ASM。

2. 系统级寄存器及控制寄存器

Pentium 的系统地址寄存器(全局描述符表寄存器 GDTR、局部描述符表寄存器 LDTR、中断描述符表寄存器 IDTR 和任务状态段寄存器 TR)与 80386 的系统地址寄存器相同。请参见图 10.9。

Pentium 支持 4 个控制寄存器 CR0、CR2、CR3 和 CR4。与 80486 相比，多了 CR4。

控制寄存器 CR0 各位的安排与 80486 相同，如图 11.2 所示，但其中的 CD 位和 NW 位重新定义了片上超高速数据缓存的工作方式。请参见 11.3.4 节关于片上超高速缓存的说明。

控制寄存器 CR2 用于指示引起页面故障的线性地址，这与 80486 相同。

控制寄存器 CR3 高 20 位含有页目录表所在物理页的页码，低 12 位中定义了控制位 PCD 和 PWT。这与 80486 相同，如图 11.3 所示。

CR4 是 Pentium 新增的控制寄存器，其各位的定义如图 11.7 所示。

31 … 7	6	5	4	3	2	1	0
保留为0	MCE	0	PSE	DE	TSD	PVI	VME

图 11.7 Pentium 控制寄存器 CR4

位 2 是读时间标记计数器指令 RDTSC 使用控制位 TSD。当 TSD 为 0 时，可在任一特权级上执行读时间标记计数器指令 RDTSC。当 TSD 为 1 时，只有在当前特权级为 0 时，可执行指令 RDTSC，否则将导致出错码为 0 的通用保护异常。RESET 后，TSD 为 0，任一特权级执行的程序都可使用指令 RDTSC，读取时间标记计数器。

位 3 是调试扩充控制位 DE。当 DE 为 0 时，禁止调试扩充，也即不支持 I/O 断点。当 DE 为 1 时，允许调试扩充，也即支持 I/O 断点。RESET 后，DE 为 0，禁止调试扩充，这样 Pentium 就保持 80486 原有调试功能。请参见下面关于调试寄存器的说明。

位 0 是虚拟 8086 方式扩充控制位 VME。当 VME 为 0 时，禁止虚拟 8086 方式扩充。位 1 是保护方式虚拟中断控制位 PVI。当 PVI 为 0 时，禁止保护方式虚拟中断。位 4 是页面大小扩充控制位 PSE。当 PSE 为 0 时，禁止页面大小扩充。位 6 是机器检查异常控制位 MCE。当 MCE 为 0 时，禁止机器检查异常。在 RESET 后，这些控制位都是 0，从而保持与 80486 相一致。

3. 调试寄存器

Pentium 不仅支持如图 11.5 所示的调试寄存器，而且调试功能还有所扩充。Pentium 所支持的调试功能扩充是指支持 I/O 断点。80486 调试控制寄存器 DR7 中的各断点类型说明字段 RWEi 尽管是 2 位，但该字段取值“10”的情况被保留。在允许调试扩充时，Pentium 调试控制寄存器 DR7 中的各断点类型说明字段 RWEi，可以取值“10”，所表示的断点条件是 I/O 端口读或者写。在允许调试扩充时，表 11.2 所列出的断点类型扩充为表 11.4 所列的断点类型。对应的断点地址寄存器内存放的是扩展成 32 位的 I/O 端口地址。

表 11.4 调试功能扩充时的断点类型

RWE 字段取值	断点类型(断点条件)	RWE 字段取值	断点类型(断点条件)
0 0	只执行	1 0	读写 I/O 端口
0 1	只写入数据	1 1	只读或写数据

控制寄存器 CR4 中的 DE 位决定是否允许调试功能扩充。当 DE 为 1 时，允许调试功能扩充。

11.3.2 指令系统

Pentium 指令集应该说是 80486 指令集的超集。由于在 Pentium 中测试寄存器归属于 Pentium 模型专用寄存器，所以 Pentium 不再支持测试寄存器数据传送指令。除此之外，Pentium 支持其他 80486 的全部指令，并保持与 80486 兼容。与 80486 相比，Pentium 还新增了几条指令。但某些新增的指令是否有效与 Pentium 的型号有关，可利用处理器特征识别指令 CPUID 判别处理器是否支持某些新增指令。

Pentium 支持控制寄存器 CR4，所以 Pentium 的控制寄存器传送指令的操作数还可以是控制寄存器 CR4。Pentium 还提供从系统管理方式返回的指令 RSM。

1. 8 字节比较交换指令 CMPXCHG8B

8 字节比较交换指令的格式如下：

```
CMPXCHG8B    OPRD
```

其中操作数 OPRD 是 64 位存储器操作数。该指令的功能是把 EDX：EAX 内的 64 位值与存储器操作数 OPRD 相比较，如果相等，把 ECX：EBX 内的 64 位值存入存储器操作数 OPRD；如果不等，把存储器操作数 OPRD 内的 64 位值装入 EDX：EAX。EDX 和

ECX 分别是 64 位值的高 32 位。

例如：

```
CMPXCHG8B   [BX]
CMPXCHG8B   [EDX]
```

如果 EDX：EAX 的值与存储器操作数 OPRD 的值相等，那么置 ZF，否则清 ZF。该指令不影响其他标志。

在加上 LOCK 前缀后，该指令对多处理器情况下的信号量操作有用。

2. 处理器特征识别指令 CPUID

处理器特征识别指令的一般格式如下：

```
CPUID
```

利用该指令能够方便地获得包括处理器类型在内的若干处理器特征信息。CPUID 指令返回由 EAX 指定的某方面的处理器特征信息。表 11.5 列出了指令 CPUID 根据 EAX 返回特征信息的定义。在 Pentium 第 1 型中，指令参数可取的最大值是 1。在新一代处理器中，可能提供更丰富的特征信息。

表 11.5 CPUID 返回的特征信息

参　　数	CUPID 返回信息
EAX=0	EAX = 最大值 EBX：EDX：ECX=厂商识别标识串
EAX=1	EAX=CPU 说明信息 EDX = 特征标志字
1<EAX≤最大值	可能在未来的处理器中定义
EAX>最大值	未定义

例如，如下指令可取得最大特征参数值和厂商识别标识串：

```
MOV   EAX,0
CPUID
```

Pentium 处理器的厂商识别标识串是“Genuine Intel”。

关于由 EAX 返回的 CPU 说明信息和由 EDX 返回的特征标志字的格式及说明请参见 11.3.3 节。

该指令不影响各标志。

从后期的 80486 开始，就支持该指令。软件可通过判断标志寄存器中的 ID 位是否可设置和清除来判断处理器是否支持该指令。如果 ID 可设置和清除，那么可使用 CPUID 指令。

3. 读时间标记计数器指令 RDTSC

Pentium 含有一个 64 位的时间标记计数器。该计数器随每一时钟周期递增。在 RESET 后，该计数器被清 0。利用该计数器可检测程序运行性能。

读时间标记计数器指令的一般格式如下：

RDTSC

该指令把时间标记计数器的高 32 位复制到 EDX，把低 32 位复制到 EAX。

控制寄存器 CR4 的 TSD 位限制该指令的使用。当 TSD 为 0 时，可在任一特权级上执行 RDTSC 指令。当 TSD 为 1 时，只有当特权级为 0 时，可执行 RDTSC 指令，否则将导致出错码为 0 的通用保护异常。

该指令不影响各标志。

4. 读模型专用寄存器指令 RDMSR

读模型专用寄存器指令的一般格式如下：

RDMSR

该指令把由 ECX 寄存器指定的模型专用寄存器的内容送到 EDX：EAX，EDX 含高 32 位，EAX 含低 32 位。如果所指定的模型寄存器不足 64 位，那么在 EDX：EAX 中的对应位未定义。

Pentium 提供一组模型专用寄存器，在利用 RDMSR 读某个模型专用寄存器时，必须先把欲读模型专用寄存器的编号送到 ECX 寄存器。如果在 ECX 中指定的编号未定义或被保留，将导致通用保护异常。

该指令不影响各标志。

该指令只能在实方式或者保护方式的特权级 0 下执行，否则将导致通用保护异常。

在使用该指令前，应该利用 CPUID 指令取得处理器特征标志字，以判别是否可使用该指令，否则可能导致无效操作码异常。

Pentium 的上述时间标记计数器是模型专用寄存器之一。利用访问模型专用寄存器的指令也可读取时间标记计数器值。但不提倡应用程序通过 RDMSR 指令读取时间标记计数器。

5. 写模型专用寄存器指令 WRMSR

写模型专用寄存器指令的一般格式如下：

WRMSR

该指令把 EDX：EAX 的内容送到由 ECX 寄存器指定的模型专用寄存器，EDX 送到高 32 位，EAX 送到低 32 位。如果指定的模型寄存器有未定义的或者被保留的位，那么这些位的内容不变。象利用 RDMSR 指令读模型专用寄存器那样，如果在 ECX 中指定的编号未定义或被保留，将导致通用保护异常。

该指令不影响各标志。

该指令只能在实方式或者保护方式的特权级 0 下执行，否则将导致通用保护异常。

在使用该指令前，应该利用 CPUID 指令取得处理器特征标志字，以判别是否可使用该指令，否则可能导致无效操作码异常。

系统程序利用 WRMSR 指令可设置上述时间标记计数器。

6. 用宏定义新增指令

低版的汇编器不识别 Pentium 的新增指令。可利用如下定义的宏来使用这些新增指令。请注意，其中的 8 字节比较交换指令宏没有考虑可能的地址扩展前缀，所以如果在 16

位段代码中采用该宏指令时，不能使用 32 位地址，如果在 32 位代码中采用该宏指令时，不能使用 16 位地址。

```
;文件名：PENTIUM.INC
;包含为 Pentium 新增指令定义的宏
CMPXCHG8B     MACRO    MQ                    ;8 字节比较交换指令，该宏指令
         LOCAL  LAB1,LAB2                    ;在 16 位段中使用时，不支持 32 位地址
LAB1     =      $
         STR    MQ
LAB2     =      $
         ORG    LAB1+1
         DB     0C7H
         ORG    LAB2
         ENDM
;
CPUID    MACRO                               ;处理器特征识别指令
         DB     0FH,0A2H
         ENDM
;
RDTSC    MACRO                               ;读时间标记计数器指令
         DB     0FH,31H
         ENDM
;
RDMSR    MACRO                               ;读模型专用寄存器指令
         DB     0FH,32H
         ENDM
;
         WRMSR MACRO                         ;写模型专用寄存器指令
         DB     0FH,30H
         ENDM
;
RSM      MACRO                               ;从系统管理方式返回指令
         DB     0FH,0AAH
         ENDM
```

11.3.3 处理器的识别

随着处理器不断升级换代，处理器的功能越来越强，80x86 系列新型号处理器具有老型号处理器的功能。通常，在老型号处理器上可运行的程序，在新型号处理器上也可运行。这称为向前兼容，或者称为向上兼容。另一方面，程序只有利用处理器提供的新特性，才能充分发挥处理器的能力。所以，一个良好的程序应该能够根据不同的处理器采取不同的方法，实现相同的功能。那么如何识别处理器型号(类型)呢？对前几代处理器的识别，可通

过判断标志寄存器中某些标志位的设置情况来进行。从后期的 80486 开始，处理器提供了处理器特征识别指令 CPUID，利用该指令可方便地获得处理器特征信息。

1. 处理器说明信息和特征标志字

如果处理器支持 CPUID 指令，那么利用如下指令可取得 CPU 说明信息和特征标志字：

```
MOV   EAX,1
CPUID
```

执行上述指令后，EAX 含有 CPU 说明信息，EDX 含有特征标志字。

CPU 说明信息的格式如图 11.8 所示。类别字段(Processor Type)占用 2 位，说明处理器的种类信息。家族代号(Family)字段占用 4 位，80486 处理器该字段值是 4，Pentium 处理器该字段值是 5，Pentium Pro 处理器该字段值是 6。型号(Model)字段占用 4 位，反映型号值。系列号(Stepping ID)字段占用 4 位。

31　　　　14	13　12	11　　8	7　　4	3　　0
保留	类别	家族代号	型号	系列号

图 11.8　CPU 说明信息格式

特征标志字的每一位可用于指示一个特性是否存在。Pentium 特征标志字中的基本特征标志如下：

位 0 是片上浮点处理单元特征位 FPU。该位为 1 表示处理器含有浮点处理单元，可执行 387 指令集中的浮点处理指令。

位 1 是虚拟 8086 方式扩充特征位 VME。该位为 1 表示处理器支持虚拟 8086 方式扩充，这种情况下，控制寄存器 CR4 中的 VME 位可以控制虚拟 8086 方式扩充是否允许；PVI 位可以控制保护方式虚拟中断；TSS 可以支持软件间接位图；标志寄存器中的 VIF 和 VIP 标志位有效。

位 2 是调试功能扩充特征位 DE。该位为 1 表示处理器支持调试功能扩充，这种情况下，控制寄存器 CR4 中的 DE 位可以控制调试功能扩充是否允许，也即 I/O 断点是否允许。

位 3 是页面大小扩充特征位 PSE。该位为 1 表示处理器支持页面大小扩充，这种情况下，控制寄存器 CR4 中的 PSE 位可以控制页面大小扩充是否允许。

位 4 是时间标记计数器特征位 TSC。该位为 1 表示处理器支持读时间标志计数器指令 RDTSC，控制寄存器 CR4 中的 TSD 位才可以控制允许使用 RDTSC 指令的特权级。

位 5 是模型专用寄存器特征位 MSR。该位为 1 表示处理器支持读和写模型专用寄存器指令 RDMSR 和 WRMSR。

位 7 是机器检查异常特征位 MCE。该位为 1 表示处理器支持控制寄存器 CR4 中的 MCE 位，MCE 位可以控制是否允许机器检查异常。

位 8 是 CMPXCHG8B 指令特征位 CX8。该位为 1 表示处理器支持 8 字节比较交换指令 CMPXCHG8B。

位 9 是特征标志位 APIC。该位为 1 表示处理器含有高级可编程中断控制器 APIC，并可以使用 APIC。

位 23 是特征标志位 MMX。该位为 1 表示处理器支持 MMX 指令集。

2. 识别处理器类型的实例

下面给出一个识别处理类型的实例。该实例通过判别标志寄存器中有关标志位区分 8086、80286、80386 和 80486，在确定处理器至少是 80486 之后，根据判别 ID 标志来判断是否可执行处理器特征识别指令 CPUID，Pentium 总是支持 CPUID 的。如果可以执行 CPUID 指令，那么利用该指令进一步获取处理器的特征信息。但该实例没有识别浮点处理部件。

```
;程序名：T11-3.ASM
;功  能：识别处理器类型(没有显示功能)
;说  明：该实例应在实方式下执行
          .386P
          INCLUDE    PENTIUM.INC      ;包含定义 Pentium 新增指令的宏文件
CSEG      SEGMENT  USE16
          ASSUME   CS:CSEG,DS:CSEG
CPUTYPE     DB  ?                  ;处理器类型(0:86,2:286,3:386,4:486,5:
Pentium)
CPUIDF      DB  0                ;1 表示使用 CPUID 指令
INTELF      DB  0                ;Intel 产品标识
FAMILY      DB  0                ;家族代号信息
CMODEL      DB  0                ;型号信息
STEPID      DB  0                ;系列号信息
PROPF       DD  0                ;特征标志字
;获取处理器类型的过程
GetCPUI   ROC
Check8086:
;8086 标志寄存器的最高 4 位总为 1，根据此特性判断是否是 8086
          PUSHF
          POP     AX
          AND     AX,0FFFH
          PUSH    AX
          POPF
          PUSHF
          POP     AX
          AND     AX,0F000H
          CMP     AX,0F000H
          MOV     CPUTYPE,0             ;假设是 8086
          JNZ     SHORT   Check286
          RET
```

```
Check286:
;实方式下 80286 标志寄存器中的 IOPL 总为 0,根据此特性判断是否是 80286
          PUSHF
          POP     AX
          OR      AX,3000H
          PUSH    AX
          POPF
          PUSHF
          POP     AX
          TEST    AX,3000H
          MOV     CPUTYPE,2            ;假设是 80286
          JNZ     SHORT  Check386
          RET
Check386:
;80386 标志寄存器中的位 18 不能设置,而 80486 中该位定义成 AC 标志
;根据此特性判断是否是 80386
;现在可以使用 386 指令和 32 位操作数
          MOV     BP,SP
          AND     SP,NOT  3            ;避免堆栈出现不对齐现象
          PUSHFD
          POP     EAX
          MOV     EDX,EAX
          BTS     EAX,18
          PUSH    EAX
          POPFD
          PUSHFD
          POP     EAX
          BT      EAX,18
          JC      SHORT  A386
          MOV     CPUTYPE,3            ;是 80386
          MOV     SP,BP
          RET
A386:     PUSH    EDX                  ;恢复
          POPFD
          MOV     SP,BP
Check486:
;对于后期的 80486 以及 Pentium,可利用 CPUID 指令获取处理器类型信息
;标志寄存器中的位 21 定义成 ID 标志
;能否改变 ID 标志,反映是否可使用 CPUID 指令
          MOV     EAX,EDX
          BTS     EAX,21
          PUSH    EAX
```

```
        POPFD
        PUSHFD
        POP     EAX
        PUSH    EDX
        POPFD
        BT      EAX,21
        MOV     CPUTYPE,4               ;至少是 80486
        JNC     SHORT  IDOK
AE486:  MOV     CPUIDF,1                ;可使用 CPUID 指令
        XOR     EAX,EAX
        CPUID                           ;取厂商识别标识串
        CMP     EBX,"uneG"
        JNE     SHORT  IDOK
        CMP     EDX,'Ieni'
        JNE     SHORT  IDOK
        CMP     ECX,'letn'
        JNE     SHORT  IDOK
YINTEL:
        MOV     INTELF,1                ;是 Intel 产品
        CMP     EAX,1
        JB      SHORT  IDOK
        MOV     EAX,1
        CPUID                           ;进一步取得特征信息
        MOV     PROPF,EDX
        MOV     BL,AL
        AND     AX,0F0FH
        MOV     STEPID,AL               ;保存系列号信息
        MOV     FAMILY,AH               ;保存家族代号信息
        SHR     BL,4
        MOV     CMODEL,BL               ;保存型号信息
        MOV     CPUTYPE,AH
IDOK:   RET
GetCPUID    ENDP
;
Begin:  PUSH    CS
        POP     DS
        CALL    GetCPUID
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     Begin
```

为了简单,上述实例没有提供显示输出功能。读者可完善该实例,使其能够显示输出处理器类型信息和有关特征信息。

11.3.4 片上超高速缓存

Pentium 在片上包含了独立的指令(代码)和数据超高速缓存器,分别为 8K 字节。指令和数据超高速缓存可以同时被访问。可以利用软件和硬件的方法控制片上超高速缓存的工作方式,也可以利用软件和硬件的方法控制可被超高速缓存的内存区域。为了适应多处理器环境的需要,对于数据超高速缓存,采用一种称为 MESI 的协议,保持超高速缓存数据一致;对于指令超高速缓存,则采用 MESI 协议的子集 SI 来保持一致。下面简单介绍 Pentium 片上超高速缓存的组织。尽管可以认为超高速缓存器是透明的,但事实上了解它的组织对编写高效的程序有益。

1. 片上超高速缓存及其工作方式的控制

Pentium 片上的数据超高速缓存和指令超高速缓存的容量皆为 8K 字节,而且都采用二路组相关联结构。每个超高速缓存有 128 组,每组 2 行。每行 32 字节宽。每个超高速缓存都使用物理地址来访问,并且每个超高速缓存都有自己的 TLB 将线性地址转换为物理地址。

数据超高速缓存由在 4 字节边界处交错的 8 个体构成。它在同一个时钟内可以支持两次数据访问。在两条流水线中执行的指令可以同时访问数据超高速缓存,但如果访问对同一个体进行,那么会有延时。Pentium 的数据超高速缓存是一种回写式超高速缓存,填充和替换以行为单位进行。每一行有用于支持 MESI 协议的两位相关联,可表示 4 种状态:M 状态(Modified)、E 状态(Exclusive)、S 状态(Shared)和 I 状态(Invalid)。另外,每一行还有用于记录最近最少使用情况的 LRU 位相关联。

指令超高速缓存在一个时钟内可提供多达 32 字节的原始代码。由于指令超高速缓存无需回写,所以每一行有用于支持 SI 协议的一位相关联,可表示 S 和 I 这 2 种状态。每一行也有用于记录最近最少使用情况的 LRU 位相关联。

控制寄存器 CR0 中的 CD 位和 NW 位可控制片上超高速缓存的工作方式,它们定义的工作方式如表 11.6 所列。由于 Pentium 的超高速缓存支持回写方式,所以与 80486 相比,这两个控制位的定义有所不同。

表 11.6 控制超高速缓存工作方式的 CD 和 NW 位的说明

CD	NW	说　　明
0	0	读命中访问超高速缓存 读未命中可能引起置换 写命中更新超高速缓存 仅在写到共享行和写未命中时向外写出 在 WB/WT# 引脚控制下,写命中能把共享行变换到互斥状态 允许使无效
0	1	无效组合,导致出错码为 0 的通用保护故障

续表

CD	NW	说　明
1	0	读命中访问超高速缓存 读未命中不引起置换 写命中更新超高速缓存 仅在写到共享行和写未命中时更新存储器单元 在 WB/WT#引脚控制下,写命中能把共享行变换到互斥状态 允许使无效
1	1	读命中访问超高速缓存 读未命中不引起置换 写命中更新超高速缓存,但不更新存储器单元 写命中将使互斥状态行变换到已修改状态 共享行在写命中后保留共享状态 写未命中访问存储器单元 禁止使无效

CD=0 和 NW=0 能使片上超高速缓存发挥最高性能。把 CD 和 NW 置 1 能禁止超高速缓存,但为了完全禁止超高速缓存,在把 CD 和 NW 置 1 后,还应该清洗超高速缓存。在 RESET 后,CD=1 和 NW=1。

2. 演示片上超高速缓存作用的实例

下面给出一个演示程序,该程序演示片上指令超高速缓存和数据超高速缓存如何提高处理性能。演示程序含有以循环方式访问某些存储单元的两个测试子程序,测试子程序能测定运行所耗时钟数。演示程序在禁止和允许超高速缓存两种情况下,分别调用测试子程序 1,然后在允许超高速缓存的情况下再调用子程序 2,最后分别显示所用时钟数。该演示程序在 Pentium 实方式下运行。源程序清单如下:

```
;程序名: T11-4.ASM
;功  能: 演示 Pentium 片上高速缓存的作用
;说  明: 仅在 Pentium 的实方式下运行
CDBIT           =   30                        ;CR0 种的 CD 位位置
COUNTV          =   10
;
                .486P                         ;识别 486 指令集
;
                INCLUDE   PENTIUM.INC         ;包含定义 Pentium 新增指令的宏文件
;
DSEG            SEGMENT  PARA  USE16
Mess1           DB  'Clock1: $'
Mess2           DB  'Clock2: $'
Mess3           DB  'Clock3: $'
COUNT1          DD  ?
COUNT2          DD  ?
```

```
COUNT3        DD ?
DSEG    ENDS
CSEG    SEGMENT   PARA   USE16
        ASSUME    CS:CSEG,DS:DSEG
BEGIN:  MOV    AX,DSEG
        MOV    DS,AX
Step1:
        CLI
        MOV    EAX,CR0
        BTS    EAX,CDBIT
        MOV    CR0,EAX              ;禁止超高速缓存
        WBINVD                      ;清洗超高速缓存
        CALL   Access1              ;调用测试子程序 1
        STI
        MOV    COUNT1,EAX           ;保存时钟数
Step2:
        CLI
        MOV    EAX,CR0
        BTR    EAX,CDBIT
        MOV    CR0,EAX              ;允许超高速缓存
        WBINVD                      ;清洗超高速缓存
        CALL   Access1              ;调用测试子程序 1
        STI
        MOV    COUNT2,EAX           ;保存时钟数
Step3:
        CLI
        WBINVD                      ;清洗超高速缓存
        CALL   Access2              ;调用测试子程序 2
        STI
        MOV    COUNT3,EAX           ;保存时钟数
Step4:
        MOV    DX,OFFSET Mess1
        MOV    ECX,COUNT1
        CALL   DMESS                ;显示提示信息和所用时钟数 1
        MOV    DX,OFFSET Mess2
        MOV    ECX,COUNT2
        CALL   DMESS                ;显示提示信息和所用时钟数 2
        MOV    DX,OFFSET Mess3
        MOV    ECX,COUNT3
        CALL   DMESS                ;显示提示信息和所用时钟数 3
        MOV    AX,4C00H
        INT    21H
```

```
;测试子程序 1
;EAX 返回运行所用时钟数
Access1  PROC
         MOV    CX,COUNTV
         MOV    EBX,1024
         RDTSC                          ;开始时读时间标记计数器
         MOV    ESI,EDX                 ;保存
         MOV    EDI,EAX
ACC1:    MOV    EAX,[EBX]
         MOV    EAX,[EBX+1024 * 4]
         MOV    EAX,[EBX+1024 * 2]
         MOV    EAX,[EBX+1024 * 6]
         LOOP   ACC1
         RDTSC                          ;结束时再读时间标记计数器
         SUB    EAX,EDI
         SBB    EDX,ESI                 ;得所用时钟数
         RET
Access1  ENDP
;测试子程序 2
;EAX 返回运行所用时钟数
;与测试子程序 1 的不同之处是访问的存储单元
Access2  PROC
         MOV    CX,COUNTV
         MOV    EBX,1024
         RDTSC
         MOV    ESI,EDX
         MOV    EDI,EAX
ACC2:    MOV    EAX,[EBX]               ;这种安排使数据超高速缓存
         MOV    EAX,[EBX+1024 * 4]      ;不断被置换填充,也即仍然每次
         MOV    EAX,[EBX+1024 * 8]      ;访问都不命中
         MOV    EAX,[EBX+1024 * 12]
         LOOP   ACC2
         RDTSC
         SUB    EAX,EDI
         SBB    EDX,ESI
         RET
Access2  ENDP
;略去过程 DMESS
CSEG     ENDS
         END    BEGIN
```

上述演示程序利用了读时间标记计数器指令 RDTSC,在循环开始时读取时间标记

值，在循环结束时再读取时间标记值，它们的差可认为是循环所花的时钟数。

11.4 基于 Pentium 的程序优化技术

Pentium 处理器具有双整型流水线、指令和数据独立的超高速缓存、动态分支预测机制和流水线浮点处理部件等多种新特性。Pentium 的这些新特性能够大大提高处理性能。尽管这些新特性的实现对程序员而言是透明的，但如果程序员能够在理解这些新特性后注意发挥它们的作用，那么程序运行效率就可更高。本节简单介绍有助于提高程序执行速度的优化技术，这些优化技术以 Pentium 处理器为基础，这些技术的实质就是如何较好地利用 Pentium 的上述新特性。注意，在 Pentium Pro 和 Pentium II 及支持 MMX 的 Pentium 上可能略有差异。

本节介绍的例子都假设是在某个 32 位代码段中的片段。

11.4.1 流水线优化技术

Pentium 是一个高级的超标量处理器，它拥有两条并行的整型流水线。流水线是指，每条指令的执行过程分成若干阶段，每个阶段都有独立的部件来处理，当一条指令的某个处理阶段完成后，它就进入到下一处理阶段，而独立的处理部件就可立即处理下一条指令。采用流水线方式执行指令，能提高指令执行的并行程度，有效地提高机器处理性能。Pentium 的两条流水线能够同时执行指令，所以 Pentium 最多能在一个时钟内执行两条整型指令。

1. U 流水线和 V 流水线

Pentium 的两条流水线分别称为 U 流水线和 V 流水线。U 流水线是主流水线，它能够执行指令集中的所有指令。V 流水线在可执行的指令方面有限制，它只能执行大多数常用指令。

例如，如下程序片段在理想情况下执行时只要花 2 个时钟：

```
MOV     EAX,EBX                 ;在U流水线执行
MOV     EDX,ECX                 ;在V流水线执行
MOV     EBX,12345678H           ;在U流水线执行
MOV     CX,[ESI]                ;在V流水线执行
```

Pentium 的整型流水线含有五步：预取(PreFetch)、译码 1(Decode stage 1)、译码 2(Decode stage 2)、执行(Execute)、回写(WriteBack)。整型指令的执行要经过流水线中这五步。与 80486 的流水线相比，Pentium 的流水线经过了优化，可达到更高的性能。另一方面，如果要执行的两条指令符合“指令配对规则”能够配对，那么一条指令经由 U 流水线执行，另一条指令经由 V 流水线执行。Pentium 处理器以流水线方式执行指令的过程如图 11.9 所示，其中 in 表示指令 n。尽管指令在两条流水线中并行执行，但执行指令的功能与指令顺序执行时是完全一致的。

流水线的第一步是预取(PF)，在该阶段，从指令超高速缓存或存储器预取指令。两条流水线的预取阶段是合在一起的，由指令预取器完成。指令预取器具有两对 32 字节长的

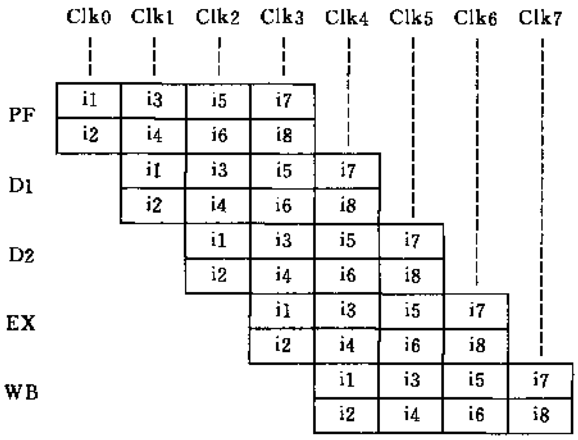


图 11.9 Pentium 流水线执行指令示意图

预取缓冲区，一对用于顺序预取，一对用于预测分支执行预取(从分支预测机制预测执行的分支处预取)。由于 Pentium 具有独立的指令超高速缓存，所以从超高速缓存预取指令不会与数据访问相冲突。

流水线的第二步是第一阶段译码(D1)。在该阶段之初，译码器根据指令配对规则判别随后的两条指令是否配对。如果配对，那么把两条指令依次发往 U 流水线和 V 流水线；如果不配对，那么仅一条指令交 U 流水线。

流水线的第三步是第二阶段译码(D2)。在该阶段，计算存储器操作数的有效地址。通常该阶段仅需一个时钟。

流水线的第四步是执行(EX)。在该阶段，进行算术逻辑运算和数据存取访问。因此，那些既要算术逻辑运算，又要数据存取访问的指令在该步中需要化去多个时钟。例如，指令“ADD [EBX],EAX”就是这样的指令，在该阶段要化 3 个时钟。

流水线的第五步是回写(WB)。在该阶段，改变处理器状态，最终完成指令的执行。

在依次经过流水线各步的过程中，由于某些情况，指令执行可能被拖延。U 流水线和 V 流水线中的指令一起配对进入和离开 D2 和 EX 阶段。如果在某条流水线中的指令在某阶段被耽搁，那么在另一条流水线中的配对指令也在同一阶段被耽搁，所以 U 流水线和 V 流水线中的配对指令同时进入 EX 阶段。在 EX 阶段，如果 U 流水线中的指令被拖延，那么 V 流水线中可能有的配对指令也被耽搁；如果 V 流水线中的指令被拖延，那么 U 流水线中的配对指令也被耽搁。只有当处于两条流水线 EX 阶段的配对指令都到达 WB 阶段，那么两条流水线上的随后指令才能够进入 EX 阶段。在两条流水线并行执行过程中，对存储器的访问仍保持顺序执行时的次序，所以只有在 U 流水线中的指令完成需要的存储器访问后，V 流水线中可能有的配对指令才能够进行存储器访问，这也可能会导致某些执行被拖延。

2. 整型指令配对规则

指令集中的指令根据如何进行流水线配对可分为如下四类：

(1) UV 类指令，这类指令既可发到 U 流水线又可发到 V 流水线。大多数算术逻辑

运算指令、全部比较指令和全部以通用寄存器为操作对象的堆栈操作指令属于 UV 类指令。

(2) PU 类指令，这类指令在配对时只能发到 U 流水线。有前缀的指令属于 PU 类指令；带进位或借位操作指令、移位位数由立即数决定的移位指令属于 PU 类指令。

(3) PV 类指令，这类指令在配对时只能发到 V 流水线。简单控制转移指令(如近调用 CALL，近转移 JMP，条件转移指令)属于 PV 类指令。

(4) NP 类指令，这类指令不能配对，只能单独在 U 流水线中执行。移位位数由 CL 确定的移位指令、乘除指令(如 MUL，DIV)、扩充的指令(如 PUSHA，ENTER，MOVS，LOOPNZ，MOVSX)、涉及段寄存器的指令(如远调用指令 CALL，PUSH　DS)属于 NP 类指令。

表 11.7 列出了主要可配对的整型指令，其中 U 流水线列的指令是在配对时可进入 U 流水线的指令(即 UV 类指令和 PU 类指令)，V 流水线列的指令是在配对时可进入 V 流水线的指令(即 UV 类指令和 PV 类指令)。表中的操作符 alu 表示算术逻辑运算操作符，shift/rot 表示移位或者循环操作符，jcc　near 表示条件转移指令，0F jcc 表示转移范围扩充的条件转移指令，acc 表示累加器，reg 和 r 表示通用寄存器，m 表示存储器操作数，r/m 表示寄存器操作数或者存储器操作数，imm 表示立即操作数。

表 11.7　整型指令配对

U 流水线可配对指令			V 流水线可配对指令		
mov r,r	alu m,imm	push imm	mov r,r	alu m,imm	push imm
mov r,m	alu m,r	pop r	mov r,m	alu m,r	pop r
mov m,r	alu r,m	nop	mov m,r	alu r,m	jmp near
mov r,imm	inc/dec r	shift/rot by1	mov r,imm	inc/dec r	jcc near
mov m,imm	inc/dec m	shift by imm	mov m,imm	inc/dec m	call near
alu r,r	lea r,m	test reg,r/m	alu r,r	lea r,m	0F jcc
alu r,imm	push r	test acc,imm	alu r,imm	push r	nop
				test reg,r/m	test acc,imm

整型指令配对的基本规则(必要条件)如下：

(1) 配对的两条指令都不能是 NP 类指令，前一条指令不能是 PV 类指令，后一条指令不能是 PU 类指令。

(2) 配对的两条指令必须已在指令超高速缓存中，而且已执行过。但如果前一条指令是单字节指令，那么该指令不受该项约束。例如，指令“PUSH　EAX”就是单字节指令。

(3) 除某些特别的配对指令外，配对的指令之间不能有隐式或显式的寄存器争用。稍后再介绍寄存器争用的概念和如何避免寄存器争用。

(4) 如果指令的一个操作数由立即寻址方式确定，并且另一个操作数的寻址方式中使用了相对偏移，那么这样的指令不能参与配对。立即数和地址偏移可能是 8 位、16 位或者 32 位。例如，指令“MOV　BYTE PTR [EBX+1],12”，不能参与配对，属于 NP 类指

令。所以，如果要多次使用某个立即数，那么可先将其装入某个通用寄存器。

(5) 指令基本长度(不包括前缀)超过 7 字节的指令不能参与配对。

如果随后的两条指令能够配对，那么前一条指令发到 U 流水线执行，后一条指令发到 V 流水线执行；如果不能配对，那么只有前一条指令发到 U 流水线执行，后一条指令将作为下次配对的前一条指令。

例如，设某个程序片段中的指令分类情况是：PV1、PU2、PU3、UV4 和 NP5，并设这 5 条指令不受上述规则其他条款的限制，那么流水线配对情况如下：PV1 指令没有配对进入 U 流水线；PU2 没有配对进入 U 流水线；PU3 和 UV4 配对，PU3 进入 U 流水线，UV4 进入 V 流水线；NP 没有配对，进入 U 流水线。

那么，如何提高配对机会？如何减少指令在流水线中的耽搁时间呢？下面就简单介绍这方面的技巧和方法。

3. 减少寄存器争用

数据相关能够导致寄存器争用。除某些特别的配对外，如果前后指令之间发生寄存器争用，那么就不能配对。显然，减少寄存器争用可提高配对机会。下列情况被视为寄存器争用：

(1) 当前一条指令以某个寄存器为目标写，而后一条指令要引用该寄存器，就引起寄存器争用。例如：

```
MOV     EAX,8
MOV     EBX,EAX
```

(2) 当前后两条指令都以某个寄存器为目标写，也引起寄存器争用。例如：

```
MOV     EAX,[EBX]
ADD     EAX,5
```

在判定寄存器争用时，访问 32 位寄存器的部分(字节或者字)被认为是对 32 位寄存器的访问。例如，如下两条指令被认为是争用寄存器 EAX：

```
MOV     AL,0
MOV     AH,1
```

但是，由于改变标志而影响标志寄存器，不认为是对标志寄存器的争用。

另外，如果前一条指令以某个寄存器为源读，而后一条指令以该寄存器为目标写，这种情况不认为是寄存器争用。例如：

```
MOV     EAX,EBX
MOV     EBX,ECX
```

为了提高可配对的机会，Pentium 对某些寄存器争用情况作了特殊处理，把这些经特殊处理的争用配对称为特别配对。特别配对主要涉及隐式读写堆栈指针寄存器 ESP 或者隐式写标志寄存器。如下指令对是特别配对的(以行为单位)：

```
PUSH    reg/imm         PUSH    reg/imm
PUSH    reg/imm         CALL
```

```
POP    reg          POP  reg
CMP                 Jcc              ;Jcc表示条件转移指令
ADD                 JNE
```

减少寄存器争用的一个方法，是在保持程序原有逻辑的前提下调整指令次序。请考虑如下片段(假设是32位代码段，并且符合其他配对规则)。其中，L1至L4的4条指令争用寄存器ECX，导致它们不能配对；L5至L8的4条指令争用寄存器EDX，导致它们不能配对；只有L4和L5处的两条指令可配对。

```
L1:     MOV     ECX,[EAX+40]
L2:     ADD     ECX,EBX
L3:     AND     ECX,0FFFF0000H
L4:     MOV     [EAX+40],ECX
L5:     MOV     EDX,[EAX+80]
L6:     ADD     EDX,EBX
L7:     AND     EDX,0FFFF0000H
L8:     MOV     [EAX+80],EDX
```

现在调整指令次序如下，那么它们就能依次两两配对(假设符合其他配对规则)：

```
L1:     MOV     ECX,[EAX+40]
L5:     MOV     EDX,[EAX+80]
L2:     ADD     ECX,EBX
L6:     ADD     EDX,EBX
L3:     AND     ECX,0XFFFF0000H
L7:     AND     EDX,0XFFFF0000H
L4:     MOV     [EAX+40],ECX
L8:     MOV     [EAX+80],EDX
```

充分利用其他暂时不用的通用寄存器，也能够减少寄存器争用。

4. 减少地址形成相关现象

在某个通用寄存器作为基址寄存器或者变址寄存器用于计算有效地址时，如果该寄存器又是刚刚执行(发出)指令的目的寄存器，那么就出现地址形成相关(Address Generate Interlock)。由于在一个时钟内最多可执行两条指令，所以上述"刚刚执行"的指令可能间隔2条指令。例如，在执行如下两条指令时就会出现地址形成相关(AGI)现象：

```
MOV     EBX,4
MOV     ECX,ESI
MOV     EAX,[EBX]
```

当出现地址相关现象时，那么在Pentium流水线的D2阶段会耽搁一个时钟。所以，为了提高流水线的效率，应该减少和避免地址形成相关现象的发生。

与调整指令次序可减少寄存器争用一样，通过调整指令次序的方法也可减少地址形成相关现象的发生。如下片段中的4条指令都是UV类指令，但由于L4指令把EBX作为

基址寄存器，而 L1 指令以 EBX 为目标，发生地址形成相关，所以在不考虑其他因素的情况下，执行时要化 3 个时钟。

```
L1:      ADD     EBX,4
L2:      MOV     EAX,[ESI]
L3:      ADD     ECX,8
L4:      MOV     [EBX],EAX
```

现在调整指令次序如下，避免了地址形成相关现象的发生，执行时只要化 2 个时钟：

```
L2:      MOV     EAX,[ESI]
L3:      ADD     ECX,8
L4:      MOV     [EBX+4],EAX       ;注意该指令增加了相对偏移
L1:      ADD     EBX,4
```

5. 减少超高速缓存体的冲突

Pentium 的数据超高速缓存由 8 个体构成，每个体宽 4 字节。它支持在两条流水线中执行的指令可以同时访问。在两条流水线中执行的配对指令，同时对同一个体的访问称为超高速缓存体冲突。如果出现超高速缓存体冲突，那么 V 流水中执行的指令在 D2 阶段被耽搁一个时钟。

例如，下面片段中的 mem 表示存储器操作数，L1 指令和 L2 指令配对，被发到 U 流水线和 V 流水线并行执行。但 L1 指令和 L2 指令引起超高速缓存体冲突，在 V 流水线执行的 L2 指令在 D2 阶段被耽搁一个时钟。由于配对指令要一起离开 D2 阶段进入 EX 阶段，所以执行将被耽搁一个时钟。

```
L1:      MOV     ECX,[EBX]
L2:      MOV     EDX,[EBX+32]
L3:      ADD     EAX,4
```

我们同样可采用调整指令次序的方法减少超高速缓存体冲突。例如，把上面的片段作如下调整，这样，L1 指令和 L3 指令配对，就避免了超高速缓存体冲突：

```
L1:      MOV     ECX,[EBX]
L3:      ADD     EAX,4
L2:      MOV     EDX,[EBX+32]
```

6. 选择合适的指令

80x86 系列处理器的指令集越来越丰富，在实现某个功能时往往有多种选择，现在举例说明如何选择合适的指令，以增加指令配对的机会。

(1) 充分利用单字节指令

在指令超高速缓存中的指令才能够配对这条规则有个例外，也即如果前一条指令是单字节指令，那么该指令可不在超高速缓存中。所以，要多采用单字节指令。请比较如下左右两边的指令（以行位单位）：

```
XCHG    ECX,EBX                              XCHG    EAX,EBX
```

```
ADD     ESI,1                    INC   ESI
SUB     EDX,1                    DEC   EDX
```

(2) 考虑使用多条单时钟指令代替一条多时钟指令

所谓单时钟指令是指只要化一个时钟就可执行的指令,许多指令是单时钟指令。所谓多时钟指令是指执行时要化多个时钟的指令,以存储器操作数为目标操作数的算术逻辑运算指令不是单时钟指令,执行这样的指令往往要化 3 个时钟。尽管用多条单时钟指令代替一条多时钟指令一般会增加代码长度,可能还要使用更多的寄存器,但有时用这种方法可增加指令配对机会,提高流水线效率。

例如,如下右边的 3 条单时钟指令可代替左边的一条 3 时钟指令:

```
ADD     [EBX],ECX                MOV  EAX,[EBX]
                                 ADD  EAX,ECX
                                 MOV  [EBX],EAX
```

尽管右边的 3 条指令之间由于寄存器争用不能配对,但首末 2 条指令可能和上下指令配对,即使仍不配对都在 U 流水线执行,也只要花 3 个时钟。另外,两条流水线中的指令要同时退出 EX 阶段,所以某条流水线被耽搁,会影响到另一条流水线;由于 Pentium 对存储器的访问仍保持顺序执行时的次序,只有在 U 流水线中的指令完成需要的存储器访问后,V 流水线中可能有的配对指令才能够进行存储器访问。因此,上述左边指令可能多耽搁流水线超过 2 个时钟,而右边的片段可避免由于存储访问而耽搁流水线。

如下左边的 2 条指令用右边 6 条指令代替后,可收到较好的效果,实际上可避免流水线被耽搁:

```
INC     DWORD PTR  [EBX]         MOV   ECX,[EBX]
INC     DWORD PTR  [EBX+4]       MOV   EDX,[EBX+4]
                                 INC   ECX
                                 INC   EDX
                                 MOV   [EBX],ECX
                                 MOV   [EBX+4],EDX
```

以存储器操作位目标操作数的算术逻辑运算指令一般都可采用上述方法。但必须注意,这种方法的代价是增加目标代码长度和占用其他寄存器。

(3) 避免使用 NP 类指令

有某些指令不仅是 NP 类指令,而且执行时要花多个时钟。为了提高指令配对机会,提高流水线效率,可考虑用多条指令代替它。这类似于用多条单时钟指令代替一条多时钟指令,这种方法通常会增加目标代码长度。

例如,循环指令 LOOP,要花 5 至 8 个时钟,而且属于 NP 类指令。用如下两条指令代替指令 LOOP 后,可减少分支指令执行时钟,而且增加指令配对机会:

```
DEC     ECX
JNZ     NEXT                     ;NEXT 是标号
```

例如,建立堆栈框架指令 ENTER,要花 10 个以上的时钟,而且属于 NP 类指令。用如

下 3 条指令代替指令“ENTER　BCOUNT,0”后，可提高流水线效率，而且增加指令配对机会：

```
PUSH    EBP
MOV     EBP,ESP
SUB     ESP,BCOUNT                ;BCOUNT 是常数
```

例如，符号扩展指令 CDQ，属于 NP 类指令。用如下 2 条指令代替指令 CDQ 后，可增加指令配对机会。虽然 CDQ 指令花 2 个时钟，如下两条指令也要化 2 个时钟，但增加了与前后指令的配对机会：

```
MOV     EDX,EAX
SAR     EDX,31                    ;该指令属于 PU 类指令
```

7. 由指令前缀带来的副作用

80x86 系列处理器，通过在正常指令之前添加前缀的方法，有效地改变或者扩充了指令原有功能。例如，使用段超越前缀可方便地实现跨段访问；使用操作数尺寸前缀和地址长度前缀可灵活地实现 16 位代码和 32 位代码的混合。可能的前缀有段超越前缀、重复前缀、封锁前缀、操作数尺寸前缀(66H)、地址长度前缀(67H)和操作码扩充前缀等。

原本属于 UV 类的指令，由于添加了前缀，只能发到 U 流水线，也就是说在配对时只能作为 PU 类指令。实际上，指令的各个前缀依次都发到 U 流水线，然后基本指令再参加可能的配对发到 U 流水线。只有扩充的增加转移范围的条件转移指令所具有的前缀(0FH)是例外。

在 16 位代码段(实方式下的代码段是 16 位代码段)中，如果使用 32 位操作数或者 32 位地址，那么指令就带有操作数前缀或者地址长度前缀；在 32 位代码段中，如果使用 16 位操作数或者 16 位地址，那么指令就带有操作数前缀或者地址长度前缀。因此，要减少 16 位代码和 32 位代码的混用，特别是在 32 位代码段中，要减少使用 16 位寄存器和减少 16 位存储器操作数地址。

尽管利用段超越前缀的方法可方便地访问多个段，但这会带来前缀，所以要尽量通过 DS 访问数据段。

11.4.2 分支优化技术

Pentium 具有动态分支预测能力。在采用流水线方式执行指令后，如果不能有效地预测分支转移目标，那么转移就会导致流水线被冲洗，严重降低流水线处理效率。

1. 动态分支预测

Pentium 利用一个称为分支目标缓存器(Branch Target Buffer)的超高速缓存，实现动态分支预测。分支目标缓存器(BTB)是一个 4 路组相关联的超高速缓存，共有 256 项，分为 64 组。访问 BTB 时使用的标记是分支指令的地址。BTB 项的主要内容是对应位置处转移指令的转移目标地址，该地址就作为预测的转移目标地址。

在流水线的开始阶段对转移指令的转移目标地址进行预测。在流水线的后期阶段，根据实际转移地址修正预测地址。但并非每当根据 BTB 项作出的预测不正确时，就一定用

实际转移地址来修正预测地址。BTB 项除记录转移目标地址外，还有 2 位用于记录对应转移指令的转移历史信息。在修正预测地址时，还根据历史信息位作出是否要改变 BTB 中记录的转移目标地址的判断。这可避免因一次偶尔的转移变化，导致预测不正确。在连续两次 BTB 预测不正确的情况下，那么改变 BTB 中记录的该分支指令转移目标地址。表 11.8 列出了对在某个循环中的如下转移指令的动态预测情况，设 EBX 的初值是 26H：

```
        SHR     EBX,1               ;EBX 初值是 00100110B
        JC      NEXT
                ......
NEXT:           ......
```

表 11.8 对某分支转移指令的动态预测情况

循环序号	转移情况	命中 BTB 情况	流水线被冲洗情况
1	否，顺序	0	0
2	是，转移	0	1
3	是，转移	1	0
4	否，顺序	1	1
5	否，顺序	1	1
6	是，转移	1	1
7	否，顺序	1	1
8	否，顺序	1	0

如果 BTB 已记录了预取的转移指令的转移目标地址，那么 Pentium 就预测该地址是要转移的目标地址。如果 BTB 还没有记录预取转移指令的转移目标地址，也即无法根据 BTB 作出预测，那么 Pentium 就预测为不发生转移。在不是根据 BTB 作出预测的情况下，如果预测正确，那么不把该顺序地址保存到对应的 BTB 项；如果预测不正确，那么就把实际转移地址保存到对应的 BTB 项。请参见表 11.8，最初情况是 BTB 没有记录预取转移指令的转移目标地址。

如果预测正确，Pentium 流水线并未被耽搁；如果预测不正确，那么流水线将被冲洗，为此要耽搁 3 或 4 个时钟。

为了提高流水线的效率，应该减少分支转移，提高预测正确程度。下面就简单介绍这方面的技巧和方法。

2. 调整基本片段位置

所谓基本片段是指只有一个入口和出口的顺序执行的片段。所谓调整基本片段的位置是指：把使用频率高的基本片段安排在分支转移指令之后的位置处，也即使得最经常的分支是直行的；把使用频率低的基本片段安排得尽可能离开使用频率高的基本片段。这样的安排，不仅提高可预测程度，而且可减少把很少执行的代码预取入超高速缓存，还可减少对 BTB 项的占用。

图 11.10 给出了一个简单分支所涉及的两个基本片段的调整情况。设 BB3 是经常使

用的片段，而 BB2 是不太使用的基本片段。图的左边是调整前的情况，右边是调整后的情况。尽管调整后的目标代码可能增长，而且还增加了转移指令，但当 BB3 片段的使用频率明显超过 BB2 片段使用频率时，调整后的执行效率会较好。因为，在多数情况下，将直接执行 BB3 片段。

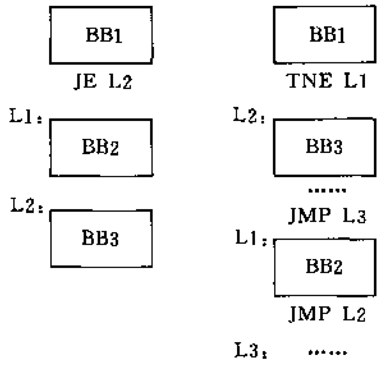


图 11.10　基本片段调整举例示意图

3. 代替条件转移指令

有时可用无分支转移的程序片段代替含分支转移的程序片段，从而消除分支。有两种常用的代替方法：一种是用条件字节设置指令 SETcc 代替条件转移指令 Jcc；另一种方法是用带进位 CF 操作的 ADC 或 SBB 指令代替条件转移指令 Jcc。

如下两个并列的程序片段，完成相同的功能(r3 = r1 ae r2 ? 1 : 0)，其中的 r1、r2 和 r3 分别表示通用寄存器。左边的片段含转移指令；右边的片段没有转移指令，效率要高于左边的片段。所以，通常情况下应该使用右边的片段代替左边的片段。

```
          CMP    r1,r2                    XOR    r3,r3
          MOV    r3,1                     CMP    r1,r2
          Jae    NEXT                     SETae  r3
          MOV    r3,0
NEXT:
```

如下两个并列的程序片段，都能完成更一般化的功能，r3 = r1 cc r2 ? CONST1: CONST2，其中的 r1、r2 和 r3 分别表示通用寄存器，cc 表示条件，CONST1 和 CONST2 分别表示常数。

```
          CMP    r1,r2             XOR    r3,r3
          MOV    r3,CONST1         CMP    r1,r2
          Jcc    NEXT              SETcc  r3L       ;r3L 表示 r3 低 8 位寄存器
          MOV    r3,CONST2         DEC    r3
NEXT:                              AND    r3,CONST2-CONST1
                                   ADD    r3,CONST1   ;如 CONST1 为 0,可省
```

上述左边的片段含有条件转移指令，右边的片段没有转移指令。是否要用右边的片段代替左边的片段，取决于左边片段的平均效率。

如下程序片段是利用带 CF 位的 SBB 指令代替条件转移指令的一个例子。程序片段的功能是根据 EBX 内容的正负设置 EDX 的值(正时 EDX 为 0，负时 EDX 为－1)。右边片段的效率比左边片段的效率要好。

```
CMP    EAX,0                SHL    EAX,1
MOV    EDX,-1               SBB    EDX,EDX
JS     NEXT
```

```
MOV     EDX,0
```

如下程序片段是利用带 CF 的 ADC 指令代替条件转移指令的一个例子。其中，r1，r2，r3 是寄存器，CONST 是常数。片段的功能是实现 r3＝(r1 ne r2) ? const : 0。但在某个程序中是否要用右边的片段代替左边的片段，要视具体情况而定：

```
        CMP    r1,r2                SUB  r1,r2
        MOV    r3,CONST             CMP  r1,1
        JNE    NEXT                 MOV  r3,0
        MOV    r3,0                 ADC  r3,-1
NEXT:                               AND  r3,CONST
```

另一个更一般的例子如下，它们的功能是实现 r3 ＝ (r1 e r2) ? CONST1 : CONST2。

```
        CMP    r1,r2                SUB  r1,r2
        MOV    r3,CONST1            CMP  r1,1
        JE     NEXT                 SBB  r3,r3
        MOV    r3,CONST2            AND  r3,CONST1-CONST2
NEXT:                               ADD  r3,CONST2
```

11.4.3 超高速缓存优化技术

Pentium 的指令超高速缓存和数据超高速缓存是独立的，各有 8K 字节；都采用二路组相关联结构，都分成 128 组，每组 2 行，每行宽 32 字节；都使用物理地址访问；都采用最近最少使用淘汰算法。Pentium 片上超高速缓存能够支持在一个时钟内进行 2 次数据访问和获取 32 字节的原始代码。但是，如果不能命中片上超高速缓存，那么至少要耽搁 3 个时钟；如果还没有命中机器系统的 2 级高速缓存，或者机器没有 2 级高速缓存，那么要耽搁的时间更多，达 7 个时钟。理想情况下，Pentium 在 7 个时钟内可能执行 14 条指令。所以，根据超高速缓存的特点，安排数据和组织代码对提高程序的效率是极其重要的。

1. 对齐访问

尽管 Pentium 能够存取任何字节边界处的数据，但对数据超高速缓存不对齐的访问要多花 3 个时钟。所以，应该尽量实现对齐访问。也就是说，对 2 字节数据的访问应在 2 字节边界处进行，至少不跨越 4 字节边界；对 4 字节数据的访问应在 4 字节边界处进行；对 8 字节数据的访问应在 8 字节边界处进行。

例如，如下两条指令左边的访问不对齐，右边的访问对齐，设数据已在超高速缓存中，那么左边的不对齐访问要比右边的对齐访问多花 3 个时钟：

```
MOV     EAX,[0005H]            MOV    EAX,[0004H]
```

例如，设 EBX 的内容是 xxxx0H，在不考虑其他因素影响的情况下，那么如下三条指令的访问要比指令“MOV   AX,[EBX+3]”快：

```
MOV    AX,[EBX]
```

```
MOV    AX,[EBX+1]
MOV    AX,[EBX+2]
```

因此，在定义变量和缓冲区时，要注意数据对齐。在定义结构类型时，要注意各字段位置的安排，以便结构变量字段的对齐和结构数组的对齐。

例如，如下两个结构类型的定义，结构 STRUBB 比 STRUAA 要好。当然，假设利用这些结构类型定义的结构变量或者结构数组是对齐的。

```
STRUAA STRUC
       VARB1  DB  ?
       VARW   DW  ?
       VARD   DD  ?
       VARB2  DB  ?
STRUAA ENDS
;
STRUBB STRUC
       VARB1  DB  ?
       VARB2  DB  ?        ;即使无需 VARB2 字段，也宜添一个字节
       VARW   DW  ?        ;以便对齐
       VARD   DD  ?
STRUBB ENDS
```

注意，为了做到对齐访问，可能要浪费部分存储单元。

2. 访问数据相对集中

通常程序运行时所执行的指令和所访问的数据各自相对集中，而且越是被多次访问的数据越有可能再被访问。高速缓存正是根据这些特点设计的。为了充分利用超高速缓存，在安排数据和组织代码时要注意相对集中。具体地说是数据相对集中和访问相同数据的代码相对集中。相对集中可使得被访问的数据尽量已出现在超高速缓存中。

例如，假设要对一组数据进行两项操作，如果分别用两个循环来处理，那么把这两个循环安排得较近为好。如果数据组较大，那么应该考虑把这两个循环合为一个循环。

3. 缩短代码长度

尽量使当前活跃的代码长度在 8K 字节范围内，以适合指令缓冲区。算法的改进是缩短代码长度的主要手段。此外还有一些选择指令的小技巧。

采用一条多时钟的指令代替多条单时钟的指令能有效地缩短目标代码长度。但前面已提及，这种方法会减少指令配对机会，在取舍时要综合考虑。

充分利用 32 位寻址方式。例如，如下的指令片段可用一条指令“MOV   ECX,[EBX+EDX*8+32]”代替：

```
SHL    EDX,3
ADD    EBX,EDX
ADD    EBX,32
MOV    ECX,[EBX]
```

利用 LEA 指令往往可方便地实现多个操作数相加。并且 LEA 指令是 UV 类指令，执行时只要花一个时钟。但要注意，LEA 指令可能会增加 AGI 现象的发生。

例如，如下的程序片段，可用一条指令“LEA　ECX，[EAX+EBX * 4+CONSTV]”代替，其中 CONSTV 是常数：

```
MOV    ECX,EBX
SHL    ECX,2
ADD    ECX,EAX
ADD    ECX,CONSTV
```

注意采用较短的指令。在 11.4.1 指出了单字节指令便于配对的优点，显然单字节指令也能够缩短目标代码，但多数情况是无法用单字节指令实现的。在使用多字节指令时，要注意选择较短的多字节指令。例如，如下左右两个片段多根据 EBX 的内容是否为 0 进行分支，但右边的片段较短：

```
CMP  EBX,0                OR   EBX,EBX
JNZ  NEXT                 JNZ  NEXT
```

当条件转移指令和无条件转移指令的转移范围较小时，要用“SHORT”明确说明，这样可只用 1 字节表示地址差。例如，如果上述条件转移指令中的标号 NEXT 较近，并且是向前引用，那么在其前加上“SHORT”可节省 3 个字节（设 32 位代码）。

11.5 习　题

题 11.1　80486 有哪些新特点？Pentium 有哪些新特点？

题 11.2　80486 标志寄存器比 80386 标志寄存器多定义了哪些标志？这些标志有何作用？

题 11.3　Pentium 标志寄存器中的标志 ID 有何作用？

题 11.4　80486 控制寄存器 CR0 比 80386 的控制寄存器 CR0 多定义了哪些控制位？

题 11.5　Pentium 控制寄存器 CR0 比 80486 控制寄存器多定义了哪些控制位？

题 11.6　80486 指令集比 80386 指令集增加了哪些指令？

题 11.7　Pentium 指令集比 80486 指令集增加了哪些指令？

题 11.8　80486 片上超高速缓存有哪些特点？

题 11.9　Pentium 片上超高速缓存有哪些特点？

题 11.10　在 11.1.3 中有一个演示 NW 位作用的程序片段。请说明在分别删除该程序片段中 LINE1、LINE2 或 LINE3 行后，该片段的执行情况。

题 11.11　80386 和 80486 等增加了哪些调试功能？它们如何实现数据访问断点？

题 11.12　在 80386 和 80486 等微处理器上通过哪些途径可实现程序执行断点？

题 11.13　请画出程序 T11-2.ASM 为结束调试故障处理而安排的堆栈。

题 11.14　Pentium 的调试功能扩充指什么？

题 11.15　请完善程序 T11-3.ASM，使其具有显示输出功能。

题 11.16　在基于 Pentium 优化程序时，要考虑哪些方面？

题 11.17　请用一条指令代替如下由于寄存器争用而不能配对的两条指令：

```
MOV     ECX,EBX
ADD     ECX,EAX
```

题 11.18 考虑如下程序片段，请调整指令次序，以增加指令配对机会：

```
MOV     EAX,ECX
ADD     EAX,4
ADD     EDX,EAX
MOV     EBX,3
ADD     ECX,EBX
TEST    EBX,EBX
```

题 11.19 考虑如下程序片段，请通过调整指令次序的方法，避免 AGI 现象：

```
ADD     ECX,4
MOV     [ECX],EAX
INC     EAX
CMP     EAX,100
JL      NEXT
```

题 11.20 请考虑如何用多条单时钟指令代替如下指令片段：

```
ADD     DWORD PTR [EBX],123456H
SUB     DWORD PTR [EBX+8],EAX
```

题 11.21 请考虑如何用多条指令代替如下属于 NP 类的指令：

(1) LEAVE　　(2) XCHG

(3) MOVSX　　(4) MOVZX

题 11.22 请比较如下左右两边指令片段的效率：

```
(1) IMUL    ECX,16          SHL     ECX,4
(2) IMUL    ECX,4           LEA     ECX,[ECX*4]
(3) IMUL    EAX,7           MOV     EBX,EAX
                            SHL     EAX,3
                            SUB     EAX,EBX
```

题 11.23 请考虑如何调整"IF THEN ELSE"结构所涉及的基本片段。

题 11.24 如下两个程序片段实现相同的功能，r3=(r1　e　r2)？ b+1：b。其中，b 是常数，r3L 表示 r3 的低 8 位寄存器。请对它们作比较：

```
CMP     r1,r2           XOR     r3,r3
MOV     r3,b+1          CMP     r1,r2
Je      NEXT            SETe    r3L
MOV     r3,b            ADD     r3,b
```

题 11.25 请考虑如下程序片段所能完成的功能，其中 d 是常数(1、2、4、8)，CONST1 和 CONST2 也是常数。

```
XOR     ECX,ECX
CMP     EAX,EBX
SETGE   CL
LEA     ECX,[ECX*d+ECX+CONST1-CONST2]
```

题 11.26 请利用含有 ADC 或者 SBB 指令的片段代替如下含条件转移指令的片段：

```
        CMP     r1,r2
        MOV     r3,CONST+1
        JL      NEXT
        MOV     r3,CONST
NEXT:
```

题 11.27 如何尽可能实现数据对齐？为何追求对齐访问可能要浪费部分存储单元？

题 11.28 请简化如下程序片段：

```
        ADD     EAX,4
        SHL     EAX,2
        ADD     EBX,EAX
        MOV     ECX,[EBX]
```

题 11.29 哪些优化方法总是可取的？哪些优化方法要根据具体情况而定，为什么？

题 11.30 请在 80486 或者 Pentium 上调试第 9 和第 10 章的例题，并作优化。

第三部分 上机实验部分

第12章 实验指导

上机实际操作是学习汇编语言程序设计的重要步骤。由于汇编语言固有的特点，调试汇编语言程序要比调试高级语言程序困难得多。

本章介绍的实验指导内容基于DOS平台，如果使用的是Windows平台，那么可转换到MS-DOS程序方式，即建立DOS平台。

12.1 实验的一般步骤

通常程序设计的开始工作是对需求进行分析，根据要求和规模等因素划分模块，设计各功能模块的实现算法。在完成这些工作之后，进行编程和调试。就汇编语言程序设计或者汇编语言和高级语言的混合编程而言，需求分析、模块划分和算法设计等工作与高级语言程序设计是相似的。从根据数据结构和算法进行编码到形成可试用程序的过程如图12.1所示。这个过程主要由编辑、汇编、连接和调试四个步骤构成。

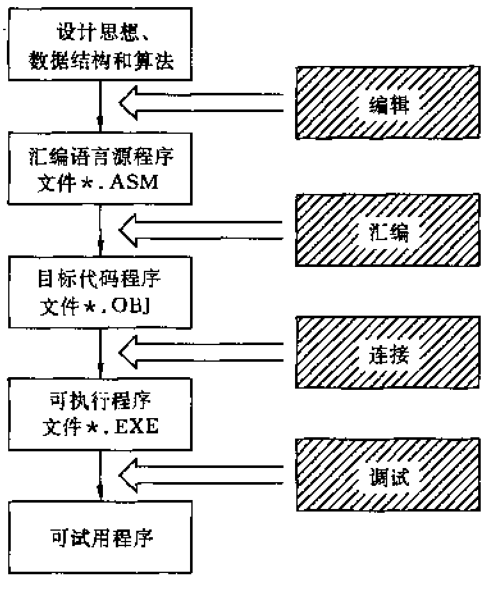


图12.1 汇编语言程序设计编程调试过程

通常，编辑、汇编、连接和调试这四步不在一个集成环境的支持下进行，而是分步独立进行。

第一步是编辑源程序。利用文本编辑工具（如 EDIT）编辑源程序，生成一个汇编语言源程序的纯文本文件。通常汇编语言源程序文件的扩展名是 ASM。汇编语言源程序一行安排一条语句，不采用类似 C 或者 PASCAL 等高级语言源程序那样的分层次缩进格式。如下是汇编语言源程序的一般格式。请注意上下行之间的指令助记符及第一个操作数首字符的对齐，利用制表符能较好地实现对齐格式。

```
;源程序名：HELLO.ASM
;功    能：显示一个字符串
;数据段
DSEG     SEGMENT
MESSAGE DB   'How do you do.',0DH,0AH,24H
DSEG     ENDS
;代码段
CSEG     SEGMENT
         ASSUME  CS:CSEG,DS:DSEG
BEGIN:
        MOV     AX,DSEG
        MOV     DS,AX
        MOV     DX,OFFSET  MESSAGE
        MOV     AH,9
        INT     21H
        MOV     AH,4CH
        INT     21H
CSEG    ENDS
        END     BEGIN
```

第二步是汇编源程序。利用汇编器（如 MASM 或者 TASM）汇编源程序生成目标代码文件。通常目标代码文件的扩展名是 OBJ。汇编器还可以生成列表文件和交叉参考文件。汇编器相当于高级语言程序设计中的编译器。汇编器按汇编语言的语法检查源程序，如果源程序中有语法错误的行，那么汇编器就不生成目标代码文件。这种情况下，必须回到第一步，重新编辑源程序，修改语法错误的行。当发现源程序中的某些行含可疑成分或不确定因素时，汇编器会给出警告信息，但仍按缺省处理办法生成目标代码文件。这种情况下，可以重新编辑源程序，消除可疑成分或不确定因素。

第三步是连接目标程序。利用连接器（如 LINK 或者 TLINK）连接目标代码程序和库函数代码生成可执行程序文件。通常 DOS 平台上的可执行程序文件的扩展名是 EXE。一般单个模块的连接不会发生连接错误，总可以顺利地生成可执行程序文件。当多个模块连接，或者与库中的函数连接时，如果在目标代码文件或者库中找不到所需的连接信息，连接器就会发出错误提示信息，而不生成可执行程序文件。这就需要修改源程序，使得汇编器生成的目标代码文件含有连接器需要的信息。这样的修改主要是对伪指令和汇编语言

操作符的修改，或者是对名字符号的修改。如果出现这种情况，那么就要回到第一步编辑源程序，还要重新进行第二步汇编源程序。

第四步是调试可执行程序。程序的动态调试是在形成可执行程序文件后，针对可执行程序进行的。DEBUG 是简单而有效的动态调试工具，Turbo Debug 是功能较强的调试工具。利用调试工具动态地调试程序，找出程序中的“臭虫”。如果发现程序中有“臭虫”，那么必须回到第一步，重新开始。当然，如果问题是算法不正确造成的，那么还得修正算法。

上述四个步骤不包括函数库的建立和管理。

12.2 汇编器和连接器的使用

在 DOS 平台上使用得较普遍的汇编器是 MASM 和 TASM，连接器是 LINK 和 TLINK。MASM 和 LINK 由 Microsoft 公司出品，TASM 和 TLINK 由 Borland 公司出品。

12.2.1 MASM 的使用

下面介绍的宏汇编器 MASM 5.0 版本使用比较普遍。命令格式如下：

MASM [/options] [source(.asm)],[out(.obj)],[list(.lst)],[cref(.crf)] [;]

可选的命令动作选项由符号“/”引导。利用命令“MASM /HELP”可获得有关命令动作选项及其说明信息。

source(.asm)指定源程序，缺省的扩展名是 ASM。

out(.obj)指定输出的目标代码文件。缺省的文件名同源程序文件名，缺省的扩展名是 OBJ。

list(.lst)指定输出的列表文件，缺省的扩展名是 LST。缺省情况是不生成列表文件。

cref(.crf)指定输出的交叉参考文件，缺省的扩展名是 CRF。缺省情况是不生成交叉参考文件。

命令行最后的分号表示其后的缺省项，按缺省设置处理。

例如：利用 MASM 汇编 12.1 节的源程序 HELLO.ASM，生成的目标代码存放在文件 TEST.OBJ 中，操作命令如下：

```
C>MASM
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: HELLO.ASM          指定源文件名为 HELLO.ASM
Object filename [HELLO.OBJ]: TEST          指定目标代码文件名为 TEST.OBJ
Source listing  [NUL.LST]:                 按回车键，表示不要生成列表文件
Cross-reference [NUL.CRF]:                 按回车键，表示不要生成交叉参考文件

51568 + 426896 Bytes symbol space free
```

```
    0  Warning Errors
    0  Severe  Errors
```

如果有警告提示信息或者错误提示信息，那么屏幕上就会出现对应的提示信息行，而且最后的统计结果就不等于 0。当有错误提示信息时，不生成目标代码文件。

再如，也可以用如下命令汇编源程序 HELLO. ASM，目标代码文件存放在 HELLO. OBJ 文件中：

```
C>MASM  HELLO                      缺省的扩展名是 ASM
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Object filename [HELLO.OBJ]:  按回车键，同意目标代码文件名 HELLO.OBJ
Source listing  [NUL.LST]: ;  按分号键，按缺省设置处理，不生成列表文件
                                                不生成交叉参考文件
51568 + 426896  Bytes  symbol  space  free

      0  Warning Errors
      0  Severe  Errors
```

如下操作更简单，指定源程序文件 HELLO. ASM，生成的目标文件以 HELLO. OBJ 命名，不生成列表文件和交叉参考文件：

```
C>MASM  HELLO ;
```

命令行尾的分号表示其他参数按缺省设置处理，也即目标文件名同源文件名（扩展名为 OBJ），不生成列表文件和交叉参考文件。

12.2.2 LINK 的使用

下面介绍的连接器 LINK 3.0 版本较低。命令格式如下：

LINK [/options] [source(.obj)...],[out(.exe)],[mapfile(.map)],[library(.lib)...][;]

可选的命令动作选项由符号“/”引导。利用命令“LINK /HELP”可获得有关命令动作选项及其说明信息。

source(.obj)指定目标代码文件，缺省的扩展名是 OBJ。可以有多个目标程序代码文件，文件标识间用加号间隔或者用空格间隔。

out(.exe)指定输出的可执行程序文件，缺省的文件名同第一个目标代码模块的文件名。缺省的扩展名是 EXE。

mapfile(.map)指定输出的定位图文件，缺省的扩展名是 MAP。缺省情况下不生成定位图文件。

library(.lib)指定连接时使用的库文件，缺省的扩展名是 LIB。可以有多个库，库文件标识间用加号间隔或者用空格间隔。缺省情况下不使用库。

命令行最后的分号表示其后的缺省项，按缺省设置处理。

例如：利用 LINK 汇编独立的目标代码模块 HELLO. OBJ，生成的可执行程序存放在文件 TEST. EXE 中，操作命令如下：

```
C>LINK
Microsoft (R) 8086 Object Linker    Version 3.05
Copyright (C) Microsoft Corp 1983, 1984, 1985.    All rights reserved.

Object Modules [.OBJ]: HELLO           指定目标代码模块文件标识，扩展名是 OBJ
Run File [HELLO.EXE]: TEST             指定可执行文件标识，缺省扩展名是 EXE
List File [NUL.MAP]:                   按回车键，不生成定位图文件
Libraries [.LIB]:                      按回车键，不使用库
Warning: no stack segment
```

上面最后一行的信息是 LINK 给出的警告信息，表示没有堆栈段。该警告信息不影响可执行程序文件的生成，但生成的可执行程序使用缺省的堆栈。一般情况下，普通的上机实验程序可不安排堆栈段，可以使用缺省的堆栈。缺省的堆栈在哪里？利用动态调试工具便能清楚地看到缺省堆栈的位置。

再如，可用如下的连接命令连接独立的目标代码模块 HELLO. OBJ，生成可执行程序 HELLO. EXE：

```
C>LINK  HELLO                          缺省的扩展名是 OBJ
Microsoft (R) 8086 Object Linker       Version 3.05
Copyright (C) Microsoft Corp 1983, 1984, 1985.    All rights reserved.

Run File [HELLO.EXE]: ;                按分号键，缺省项按缺省设置处理，
                                       可执行文件 HELLO.EXE，不生成定位图文件
                                       不使用库
```

如下命令实现同样的功能，但更加简便：

```
C>LINK  HELLO;
```

连接器 LINK 能够方便地实现多个模块的连接，也能实现目标模块与库函数的连接，最终生成可执行程序。多个模块中应有一个是主模块。

如下命令把两个目标代码模块 TEST1. OBJ 和 TEST2. OBJ 连接，生成的可执行程序存放在文件 TEST. EXE 中：

```
C>LINK  TEST1+TEST2 ,TEST;
```

如下命令把主目标代码模块 ABC. OBJ 与库 DEF. LIB 内的函数（过程）连接，生成的可执行程序文件存放在文件 ABC. EXE 中：

```
C>LINK  ABC,,,DEF
```

如下命令把主目标代码模块 TEST1. OBJ、TEST2. OBJ 与库 DEF. LIB 内的函数（过程）连接，生成的可执行程序文件存放在文件 ABC. EXE 中，生成定位图文件 GHI. MAP：

```
C>LINK  TEST1  TEST2,ABC,GHI,DEF
```

12.2.3 TASM 的使用

下面介绍的汇编器 TASM 是 Borland C++的一部分。命令格式如下：

```
TASM [options] source [,object] [,listing] [,xref]
```

该命令格式与上述 MASM 的格式类似。如果简单地键入命令 TASM，那么就可得到 TASM 的关于可选项和参数的说明信息。

source 指定源程序文件，缺省的扩展名是 ASM。命令行应该有该参数。

object 指定生成目标文件的标识，可以缺省。缺省时，文件名同源程序文件名，扩展名是 OBJ。

listing 指定生成列表文件的标识，可以缺省。缺省时，一般表示不生成列表文件。但如果使用/l 选项或者/la 选项，那么生成列表文件，列表文件的文件名同源程序文件名。列表文件的缺省扩展名是 LST。

xref 指定生成交叉参考文件的标识，可以缺省。缺省时，表示不生成交叉参考文件。生成的交叉参考文件的缺省扩展名是 XRF。

例如：如下命令汇编 HELLO.ASM，生成目标代码文件 HELLO.OBJ：

```
C>TASM  HELLO
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:      HELLO.ASM
Error messages:       None
Warning messages:     None
Passes:               1
Remaining  memory:      437k
```

再如，如下命令汇编 HELLO.ASM，生成对应的目标文件 TEST.OBJ 和列表文件 ABC.LST：

```
C>TASM  HELLO,TEST,ABC
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:      HELLO.ASM  to  TEST.OBJ
Error messages:       None
Warning messages:     None
Passes:               1
Remaining memory:     432k
```

如下命令汇编 HELLO.ASM，生成目标代码文件 HELLO.OBJ，还生成对应的列表文件 HELLO.LST：

```
C>TASM  /l  HELLO
```

12.2.4 TLINK 的使用

下面介绍的连接器 TLINK 也是 Borland C++的一部分，它的使用方法与 LINK 类似，但由于命令选项较多，所以功能比 LINK 要强。命令格式如下：

```
TLINK  objfiles[, exefile][, mapfile][, libfiles]
```

如果简单地键入命令 TLINK，那么就可得到 TLINK 的关于可选项和参数的说明信息。

objfiles 指定欲连接的目标代码文件，缺省的扩展名是 OBJ。如要连接多个目标代码文件，那么文件标识间用加号间隔或者用空格间隔。

exefile 指定输出的可执行程序文件，缺省的文件名同第一个目标代码模块的文件名。缺省的扩展名一般是 EXE。

mapfile 指定输出的定位图文件，缺省的扩展名是 MAP。缺省情况下生成定位图文件，文件名与可执行程序文件名相同。

libfiles 指定连接时使用的库文件，缺省的扩展名是 LIB。可以有多个库，库文件标识间用加号间隔或者用空格间隔。缺省情况下不使用库。

例如：如下命令连接单个目标代码模块 HELLO.OBJ，生成可执行程序 HELLO.EXE：

```
C>TLINK  HELLO
Turbo Link  Version 7.00 Copyright (c) 1987, 1994 Borland International
Warning: No stack
```

再如，如下命令连接目标代码模块 TEST1、TEST2、TEST3，生成可执行程序文件 TEST.EXE：

```
C>TLINK  TEST1+TEST2+TEST3,TEST
```

如下命令连接目标代码模块 ABC 和库 DEF 中的相应过程，生成可执行程序 ABC.EXE：

```
C>TLINK  ABC,,,DEF
```

TLINK 支持 32 位目标代码的连接。如果被连接的目标代码模块含 32 位项，那么就必须按 32 位方式连接。这可通过使用选项/3 表示。

例如：如下命令按 32 位方式连接目标代码模块 TEST：

```
C>TLINK  /3  TEST
```

第 10 章的若干实例就要使用 32 位方式连接。

12.3 调试器 DEBUG 的使用

每个版本的 DOS 都带有动态调试器 DEBUG。原因是 DEBUG 不仅是动态调试器，

也是二进制文件编辑器，还是简单的系统维护工具。DEBUG 能提供一个动态调试程序的环境，程序员利用这个环境，可方便地调试目标代码程序。本节先介绍 DEBUG 命令的使用，举例说明如何利用 DEBUG 调试程序。

为了方便地区分键盘输入和 DEBUG 显示输出，我们用黑体表示键盘输入。

12.3.1 启动和退出 DEBUG

DEBUG 作为 DOS 的一个外部命令，必须与 DOS 版本相符。

1. 启动 DEBUG

DEBUG 的启动与其它外部命令程序一样简单，在 DOS 系统提示符下发出 DEBUG 命令即可。一般格式如下：

DEBUG [文件标识符 [参数表]]

其中，可选的文件标识符指定要调试的程序，参数表给出被调试程序所要用的命令行参数。文件标识符必须指定一个文件，也只能指定一个文件。DEBUG 根据文件标识符中的后缀是. EXE 还是. COM 判定被调试程序的类型。其他类型的文件被认为是数据文件。如果缺省文件标识符，那么认为暂时没有指定被调试对象。

DEBUG 在启动成功后，将给出 DEBUG 的提示符，即连接符"-"。通常只要有 DOS 就能找到版本相符的 DEBUG，DEBUG 就能启动成功。

例如，发出如下命令：

C>**DEBUG**

虽然，没有指定被调试程序，但 DEBUG 仍在可用内存区建立一个程序段前缀，做好准备工作，显示其提示符。这时用户可发出 DEBUG 提供的各命令。各段寄存器的初值是相同的，等于 PSP 的段值，该值与可用内存区位置有关。段内偏移 0 至 FFH 的区域作为 PSP，指令指针 IP 的值定为 100H，这种安排与调试. COM 类型的可执行文件相同。

例如，发出如下命令：

C>**DEBUG HELLO. EXE**

假设指定的被调试程序是前两节介绍的程序 HELLO. ASM 所生成的目标程序。如果找不到指定的命令文件，那么 DEBUG 将显示提示信息"File not found"，但仍给出 DEBUG 提示符，结果就象没有指定被调试程序那样。如果找到 HELLO. EXE，那么 DEBUG 就象 DOS 命令解释器 COMMAND 装载. EXE 类型命令文件那样，为 HELLO. EXE 建立程序段前缀，装载 HELLO. EXE，装载过程包括可能需要的重定位，作好运行 HELLO. EXE 的所有准备工作，显示其提示符。DS 和 ES 都等于 PSP 段值。CS 等于代码段段值，IP 等于启动地址的偏移。SS 等于堆栈段段值，SP 等于堆栈顶偏移。BX : CX 为实际装载长度(字节数)。

2. 退出 DEBUG

为了退出 DEBUG ，只需在 DEBUG 提示符下发出退出命令 Q 即可。退出命令的使用格式如下：

-Q

在发出 Q 命令后，DEBUG 就终止，控制将转回到 DOS。请注意，Q 命令不保存正在被调试的文件，为保存被调试的内容必须使用其他的 DEBUG 命令。

12.3.2 命令一览

1. 命令一览表

DEBUG 的主要命令列于表 12.1。

表 12.1 DEBUG 命令一览表

命令格式	功能说明
A [地址]	汇编
C [范围] 地址	内存区域比较
D [范围]	显示内存单元内容
E 地址 [字节值表]	修改内存单元内容
F 范围 字节值表	填充内存区域
G [=起始地址] [断点地址表]	断点执行
H 数值 数值	十六进制数加减
I 端口地址	从端口输入
L [地址 [驱动器号 扇区号 扇区数]]	从磁盘读
M 范围 地址	内存区域传送
N 文件标识符 [文件标识符...]	指定文件
O 端口 字节值	向端口输出
P [=地址] [数值]	执行过程
Q	退出 DEBUG
R [寄存器名]	显示和修改寄存器内容
S 范围 字节值表	在内存区域搜索
T [=地址] [数值]	跟踪执行
U [范围]	反汇编
W [地址 [驱动器号 扇区号 扇区数]]	向磁盘写

2. 通用说明

(1) DEBUG 接受和显示的数都用 16 进制表示。

(2) 命令都是一个字母，命令参数随命令而异。

(3) 命令和参数可以用大写或小写字母或混合方式输入。

(4) 命令和参数间，可以用定界符分隔(空格、制表符、逗号等)。但是，定界符只是在

两个相邻接的 16 进制数之间是必需的。因此下面的命令是等效的：

```
DCS : 100  110
D  CS : 100,  110
D,CS : 100,  110
```

(5) 在提示符出现时，可键入 DEBUG 命令，只有在按回车键后，命令才开始执行。

(6) 若 DEBUG 检查出一个命令的语法错误，则 DEBUG 将用“^ Error”指出。例如：

```
DCS : 100 CS : 110
          ^ Error
```

(7) 在输入 DEBUG 的命令行时，可以用常用的编辑键。

(8) 可以用 Ctrl+Break 键或 Ctrl+C 键来打断一个命令的执行，返回到 DEBUG 的提示符。

(9) 若一个命令产生相当多的输出行时，为了能看清屏幕上的显示内容，可按 Ctrl+S 键，暂停显示。

3. 命令参数的说明

除了退出命令 Q 外，其它 DEBUG 命令都可带有参数。主要参数的表示方法如下说明：

(1) 地址，地址参数通常表示一个内存区域(或缓冲区)的开始地址，它由段值和偏移两部分组成。段值可用一个段寄存器表示，也可用 4 位 16 进制数表示。偏移用 4 位 16 进制数表示。段值和偏移间必须有冒号作为分隔。段值部分是可省的，在段值缺省的情况下，除了 A，G，L，T，U 和 W 命令隐含使用 CS 寄存器之值外，其他命令隐含使用 DS 寄存器之值。

(2) 端口地址，端口地址使用于输入输出命令，端口地址是一个两位 16 进制数。

(3) 范围，范围用于指定内存区域(缓冲区)，由两种表示方式：第一种是用起始地址和结束地址表示，第二种是用起始地址和长度表示。长度必须以字母 L 引导。范围最大是 64K，即 0 至 0FFFFH。例如：

```
CS : 100  110
CS : 100  L10
```

但下面的地址是无效的：

```
CS : 100  CS : 110
          ^ Error
```

(4) 数值，数值参数一般表示命令重复的次数，最多可以是 4 位 16 进制数。

(5) 字节值，字节值参数表示输出到端口的值，最多可以是 2 位 16 进制数。

(6) 字节值表，字节值表参数表示要替换或查找的若干个以字节为单位的值。各值间由空格等间隔符分隔。字节值表可以含字符串，字符串必须用引号括起。

(7) 驱动器号，驱动器号参数表示要读写扇区所在的驱动器。0 代表 A 驱动器，1 代表 B 驱动器，2 代表 C 驱动器，3 代表 D 驱动器。

(8) 扇区号，扇区号表示 DOS 逻辑扇区号，最多可以是 3 位 16 进制数。

12.3.3 利用 DEBUG 调试程序

我们举例说明如何利用 DEBUG 调试程序，同时说明 DEBUG 常用命令的使用。

下面分析执行程序 HELLO。先装载 HELLO.EXE：

```
C>DEBUG  HELLO.EXE
```

设 DEBUG 装载 HELLO.EXE 成功。现在可发出 DEBUG 的各种命令。

利用 R 命令可显示 8086/8088 各寄存器的内容和下一条将要执行的指令。现在看看程序在装入后，执行前各寄存器的内容：

```
-R
AX=0000  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
DS=1086  ES=1086  SS=1096  CS=1098  IP=0000   NV UP EI PL NZ NA PO NC
1098:0000 B89610          MOV    AX,1096
```

各段寄存器的内容与存储器的实际使用情形有关。

不带任何参数的 R 命令，DEBUG 按如上格式显示各寄存器内容。

每一个标志的状态分别用两个字母表示，表示 8 个标志的状态的符号列于表 12.2。DEBUG 采用显示标志状态符号的方法反映标志值。

表 12.2 标志状态的符号表示

标志名称	溢出 OF	方向 DF	中断 IF	符号 SF	零 ZF	辅助进位 AF	奇偶 PF	进位 CF
置位状态	OV	DN	EI	NG	ZR	AC	PE	CY
复位状态	NV	UP	DI	PL	NZ	NA	PO	NC

利用反汇编命令 U 可把内存单元的内容作为机器指令，用助记符的形式显示出来。反汇编命令 U 显示的信息包括内存单元的地址、机器指令码和对应汇编格式指令。现在看看将要执行的代码：

```
-U
1098:0000 B89610          MOV    AX,1096
1098:0003 8ED8            MOV    DS,AX
1098:0005 BA0000          MOV    DX,0000
1098:0008 B409            MOV    AH,09
1098:000A CD21            INT    21
1098:000C B44C            MOV    AH,4C
1098:000E CD21            INT    21
1098:0010 8B4502          MOV    AX,[DI+02]
1098:0013 48              DEC    AX
1098:0014 3B4606          CMP    AX,[BP+06]
1098:0017 7703            JA     001C
```

```
1098 : 0019 E92E01          JMP     014A
1098 : 001C 897EFE          MOV     [BP-02],DI
1098 : 001F 8BDF            MOV     BX,DI
```

没有指定范围的反汇编命令 U，从当时 CS：IP 所指处开始，或者紧接着上次反汇编结束地址处开始反汇编，长度约为 32 字节。对照源程序 HELLO. ASM，程序代码到偏移 000FH 为止。所列出的从偏移 0010H 到 0020H 的指令似乎无意义，这是因为没有对应程序，但 DEBUG 仍将其视为指令反汇编。

反汇编命令所指定的地址是很重要的，一般应在一条有效指令的开始处：

```
-U 0  L5
1098 : 0000 B89610          MOV     AX,1096
1098 : 0003 8ED8            MOV     DS,AX
-U 1  L5
1098 : 0001 96              XCHG    SI,AX
1098 : 0002 108ED8BA        ADC     [BP+BAD8],CL
```

偏移 0000H 处的指令"MOV　AX，1096"对应源程序中标号 BEGIN 行的语句"MOV　AX,DSEG"，由此可确定数据段 DSEG 的实际段值是 1096H。

利用显示内存单元命令 D 可显示最低端 1M 范围内的任一内存单元的内容。现在可看一下数据段的内容(段值须根据实际装入的地址而定)：

```
-D 1096 : 00  L30
1096 : 0000  48 6F 77 20 64 6F 20 79-6F 75 20 64 6F 2E 0D 0A   How do you do...
1096 : 0010  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   $......
1096 : 0020  B8 96 10 8E D8 BA 00 00-B4 09 CD 21 B4 4C CD 21   ......!.L.!
```

如上格式的信息中，左边是所显示内存单元的地址，中间部分是字节值，右边把字节值作为 ASCII 码所对应的符号，对于非 ASCII 码，或者非显示符号，用点或者空格表示。

程序 HELLO 的数据段占 32 字节。如上列出的 48 字节中，前 32 字节是数据段的内容，后 16 字节实际上是代码段的内容。段 1096H 的偏移 0020H，相当于段 1098H 的偏移 0000H。由于 D 命令要求列出 48 字节的内容，所以 DEBUG 还是把其视为数据显示。

利用跟踪执行命令 T 可跟踪执行一条或多条指令。现在我们跟踪执行 HELLO：

```
-T
AX=1096  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
DS=1086  ES=1086  SS=1096  CS=1098  IP=0003  NV UP EI PL NZ NA PO NC
1098 : 0003 8ED8            MOV     DS,AX
-T = 0003  2
AX=1096  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=0005  NV UP EI PL NZ NA PO NC
1098 : 0005 BA0000          MOV     DX,0000

AX=1096  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
```

```
DS=1096  ES=1086  SS=1096  CS=1098  IP=0008  NV UP EI PL NZ NA PO NC
1098:0008 B409             MOV    AH,09
```

跟踪执行命令 T 可指定起始执行地址，地址参数以等号引导，如地址参数中无段值，那么就以 CS 为段值。必须注意，起始地址处必须是可执行指令。如果无起始地址，那么跟踪执行从 CS:IP 所指处开始。如果不指定跟踪执行指令的条数，那么就跟踪执行一条指令。

```
-T2
AX=0996  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=000A  NV UP EI PL NZ NA PO NC
1098:000A CD21             INT    21

AX=0996  BX=0000  CX=0030  DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=1096  ES=1086  SS=1096  CS=02A6 IP=04A0  NV UP DI PL NZ NA PO NC
02A6:04A0 80FC72           CMP    AH,72
```

从上可见，T 命令跟踪进入了 DOS 功能调用程序。请注意，一般情况下不要进入 DOS 功能调用程序和 BIOS 程序。

利用执行过程命令 P 可步进执行一条或多条指令。现在我们从偏移 0005H 处重新开始执行(段值须根据实际装入地址而定)：

```
-P = 1098:0005
AX=0996  BX=0000  CX=0030  DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=0008  NV UP DI PL NZ NA PO NC
1098:0008 B409             MOV    AH,09
-P2
AX=0996  BX=0000  CX=0030  DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=000A  NV UP DI PL NZ NA PO NC
1098:000A CD21             INT    21
How do you do.

AX=0924  BX=0000  CX=0030  DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=000C  NV UP DI PL NZ NA PO NC
1098:000C B44C             MOV    AH,4C
```

请注意在执行 DOS 功能调用后所显示的信息"How do you do."。

从上可见 P 命令和 T 命令的区别，P 命令不会由于过程调用或软中断调用而跟踪进入被调用程序。实际上 P 命令是步进跟踪，所以也能一次执行完 LOOP 指令，或者一次执行完重复的串操作指令。但 P 命令不适用于执行 ROM 中的程序。

利用执行命令 G 可以设置断点执行被调试程序。现在我们执行完被调试程序：

```
-G
Program  terminated  normally
```

没有指定开始地址的 G 命令从当前 CS：IP 处开始执行，直到遇断点或程序正常终止而结束。如上的 G 命令没有指定开始地址，也没有指定断点，所以从偏移 000CH 处开始执行，直到程序正常终止。HELLO 调用 DOS 的 4CH 号功能终止，所以 DEBUG 显示提示信息"Program terminated normally"而报告被调试程序执行完。

HELLO 的执行与装载时的初值无关。我们重新执行已被装载的程序 HELLO：

```
-G = 0000  0C
How do you do.
AX=0924  BX=0000  CX=0030 DX=0000  SP=0000  BP=0000 SI=0000 DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=000C  NV UP EI PL NZ NA PO NC
1098:000C B44C              MOV     AH,4C
```

如果地址参数无段值，那么就缺省认为是当前代码段，即由 CS 而定。由等号引导的开始地址处必须是有效的指令，断点处必须是有效的指令。断点不能设置在 ROM 中。

利用修改内存单元命令 E 可方便地修改任一 RAM 单元的内容。现在我们把 HELLO 程序数据段中的信息"How do you do."修改为"How are you."：

```
-E 1096:0  "How are you."
```

重新执行，看看显示的是什么信息：

```
-G = 00  000c
How are you.o.
AX=0924  BX=0000  CX=0030  DX=0000  SP=0000 BP=0000 SI=0000 DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=000C  NV UP EI PL NZ NA PO NC
1098:000C B44C              MOV     AH,4C
```

所显示的信息尾部有符号"o"，这是 9H 号 DOS 功能调用显示字符串信息以符号"$"结尾的缘故。再看数据段内的实际信息：

```
-D 0  L20
1096:0000  48 6F 77 20 61 72 65 20-79 6F 75 2E 6F 2E 0D 0A    How are you.o...
1096:0010  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    $......
```

如果 D 命令的范围参数中没有段值，那么默认为段值在 DS 寄存器内。由于没有重新装载，所以 DS 已含段值 1096H，而不是装载后的初值 1086H。现在修改数据段内的字符串信息的结尾，其中 0DH 和 0AH 是控制符回车和换行：

```
-E 00C  0D  0A  '$'
-D 0  F
1096:0000  48 6F 77 20 61 72 65 20-79 6F 75 2E 0D 0A 24 0A  How are you...$.
```

R 命令不仅能显示各寄存器内容，还可修改各通用寄存器和段寄存器内容，还包括指令指针 IP 和标志寄存器。现在修改指令指针寄存器 IP，使其指向启动地址处：

```
-R IP
IP  000C
```

```
:00
```

现在再看看各寄存器内容:

```
-R
AX=0924  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=0000 NV UP EI PL NZ NA PO NC
1098:0000 B89610              MOV     AX,1096
```

再执行,断点设在偏移 000CH 处:

```
-G  0C  010
How are you.

AX=0924  BX=0000  CX=0030  DX=0000 SP=0000 BP=0000 SI=0000  DI=0000
DS=1096  ES=1086  SS=1096  CS=1098  IP=000C  NV UP EI PL NZ NA PO NC
1098:000C B44C                MOV     AH,4C
```

尽管设置了两个断点,但由于先遇到断点 000CH,所以就停在该处。

利用 R 命令还可设置有关标志:

```
-R F
NV UP EI PL NZ NA PO NC  - ZR  CY
```

再看看 ZF 和 CF 是否确已改变:

```
-R F
NV UP EI PL ZR NA PO CY  -按回车键
```

现在退出 DEBUG:

```
-Q
```

利用 R 命令可修改各寄存器之内容,利用 E 命令可修改内存单元的内容。DEBUG 还提供汇编命令 A,直接修改内存中的代码。下面利用 A 命令改变了偏移 0005H 处的指令“MOV DX,0000”为“MOV DX,0007”,这样在执行时,显示的提示信息仅是“you do.”:

```
C>DEBUG  HELLO.EXE
-A 05
1098:0005  MOV  DX,7
1098:0008  按回车键
```

现在看看代码是否确已被修改:

```
-U 03  0A
1098:0003 8ED8                MOV     DS,AX
1098:0005 BA0700              MOV     DX,0007
1098:0008 B409                MOV     AH,09
```

```
1098 : 000A CD21              INT     21
```

执行它：

```
-G
you do.

Program terminated normally
-Q
```

12.4 Turbo Debugger 的使用

Turbo Debugger 是一个比较先进的源代码级调试器，它可以调试多种语言编写成的程序。Turbo Debugger 的重叠式窗口、下拉式和弹出式菜单以及鼠标器支持等，为用户提供了一个快速、方便和交互式的环境。此外，联机帮助还可以在操作的每个阶段提供相关的帮助。以下我们把 Turbo Debugger 简称为 TD。

尽管 TD 能够支持源代码级的程序调试，但我们在本节简单介绍如何利用 TD 调试汇编语言程序，也即机器指令级调试。TD 的详细使用细节，可通过联机帮助获得。

12.4.1 启动和退出 TD

TD 需要 DOS3.1 或更高版本的 DOS 操作系统支持。如果使用 TD 进行源代码级的调试，那么必须事先把源文件编译连接成带有全部调试信息的可执行程序文件，而且同时存在源程序文件。

1. 启动 TD

TD 的启动与其他外部命令程序一样简单，在 DOS 系统提示符下发出 TD 命令即可。一般格式如下：

TD ［可选项］ ［文件标识符［ 参数表 ］］

TD 的可选项以符号“-”引导，利用命令“TD -?”可获得关于可选项的使用说明信息。TD 命令使用的可选项会被保存到配置文件

可选的文件标识符指定要调试的程序，可不使用扩展名。参数表给出被调试程序所要用的命令行参数。

如下命令启动 TD，并装载被调试程序 HELLO：

C>**TD HELLO**

设仅有可执行程序，那么 TD 就直接给出布置如图 12.2 所示的机器指令级调试界面。可能会叠加一个报告无符号表的对话框，按 Esc 键就能关闭该对话框。

主菜单条内包含有 Run 和 Breakpoints 等菜单项。按热键 F10 可激活主菜单条，然后按左右箭头键选择菜单项，按回车键可下拉出对应的子菜单窗口。在出现子菜单窗口后，按上下箭头键和回车键选择某个动作。我们采用“xxx|yyy”表示 xxx 下拉菜单中的 yyy 项。

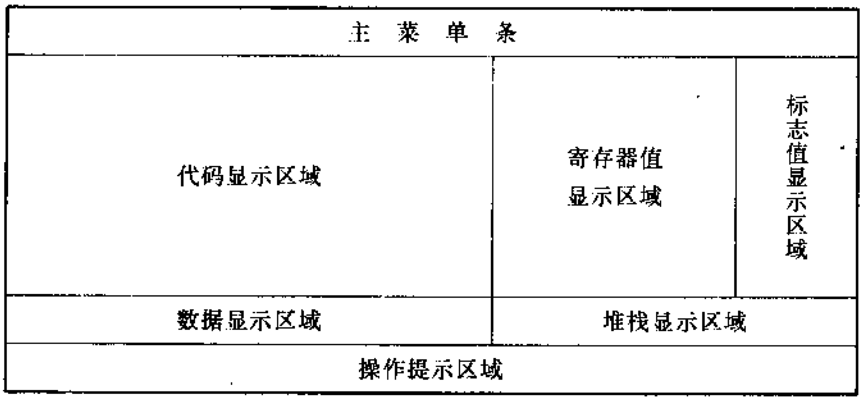


图 12.2　TD 的机器指令级调试界面

操作提示区域给出当前可使用的主要热键及其功能，随着活动区域的变化和功能切换键 Ctrl 或 Alt 的按下，该操作提示区域内的提示信息发生相应变换。

在代码显示区域，以行为单位显示汇编格式表示的机器指令和十六进制数形式表示的机器码，每一行首标出地址。在数据显示区域，通常以十六进制数形式和对应 ASCII 码符号形式显示某内存区域的内容。在堆栈显示区域，以十六进制数形式显示堆栈段内某区域的内容。在寄存器值显示区域，通常以 4 位十六进制数形式显示 8 个通用寄存器之内容。在标志值显示区域，显示各常用标志之值。

在这 5 个显示区域中，光条或光标出现的区域称为焦点区域。而且仅有一个焦点区域。利用 Tab 键、Shift＋Tab 键或者 Shift＋箭头键可方便地切换焦点区域。通常按方向箭头键可移动焦点区域中的光条或光标。利用鼠标不仅能够方便地指定焦点区域，而且还能方便地指定光条或光标的位置。

2. 退出 TD

当需要退出 TD 时，通常只要按组合键 Alt＋X 就行。如果有一个对话框处于活动状态，那么应该先按 Esc 键关闭对话框。此外，也可以选择 File|Quit 菜单动作退出 TD，返回操作系统。

12.4.2　利用 TD 调试汇编程序

在 TD 的控制下，可以采用多种方法跟踪程序的执行，随时查看内存单元、堆栈和各寄存器之内容，根据需要修改寄存器、内存单元或堆栈的内容。

由于 TD 支持源代码级调试，所以当欲输入的十六进制数以字母开头时，必须表示成以 0 开头，有时还要加后缀字母 H。

1. 查看和修改寄存器内容

从寄存器值显示区域可看到各通用寄存器和段寄存器的当前内容，包括指令指针 IP 之内容。现在把焦点区域定到寄存器值显示区域，那么在寄存器值显示区域就出现一个光条，指示某个寄存器，使用上下箭头键可移动该光条。按 Alt＋F10 可激活一个弹出式的操作菜单，利用其中的 Registers 32-bit 项可进行 16 位通用寄存器和 32 位通过用寄存器的切换，在 80386 及以上的系统中，指定 32 位通用寄存器可方便地查看和修改这些 32 位寄

存器的内容。按下 Ctrl 键，屏幕底部的操作提示区域就出现适用于寄存器值显示区域的有关组合键及其动作功能。按组合键 Ctrl+R 就可实现 16 位通用寄存器和 32 位通用寄存器的切换。

修改寄存器的内容是方便的。设焦点区域是寄存器值显示区域。按组合键 Ctrl+Z 就可方便地把光条所在寄存器的内容清 0；按组合键 Ctrl+I 使光条所在寄存器的内容递增 1；按组合键 Ctrl+D 使光条所在寄存器的内容递减 1。按组合键 Ctrl+C 键将弹出输入新值的对话框，输入的值就直接作为光条所在寄存器的新内容。此外，按字符或数值键，也直接弹出输入新值的对话框。

2. 查看和修改常用标志状态

从标志值显示区域可看到常用标志的当前状态。标志用一个字符表示，标志的当前状态用数值 1 或 0 表示。现在把焦点区域定到标志值显示区域，那么在标志值显示区域就出现一个光条，指示某个标志，使用上下箭头键可移动该光条。按下组合键 Ctrl+T，可翻转光条所在标志的状态。

3. 查看和修改数据区内容

从数据显示区域可看到部分内存区域的当前内容。通常用十六进制数和对应 ASCII 码字符两种形式显示内存区域的内容，并在每一行首给出由段值和偏移构成的内存单元地址，当段值与某个段寄存器的内容相等时，段值部分用段寄存器表示。当焦点区域定到数据显示区域时，按组合键 Ctrl+D 会弹出一个用于指定显示格式的菜单，可从中选择需要的内存区域数据显示格式。

现在把焦点区域定到数据显示区域，那么在数据显示区域就出现一个光标，指示某个内存单元，使用方向箭头键可移动该光标。通过上下翻页键和上下箭头键可调整所显示的内存区域，这种调整只是段内偏移的调整。可直接指定需要显示的内存区域首地址。按组合键 Ctrl+G，那么就会弹出一个用于输入定位地址的对话框。输入的内存区域首地址可包含段值部分，可以用段寄存器代表段值部分。如果地址表示成段值和偏移，那么段值和偏移需用冒号间隔。

可以在当前数据区域内查找指定的字节值表。设焦点区域是数据显示区域。那么按组合键 Ctrl+S 会弹出一个用于输入欲查找字节值表的对话框。

修改内存单元的内容是方便的。设焦点区域是数据显示区域。按组合键 Ctrl+C 会弹出输入数值的对话框。直接按字符或数字键，也会弹出输入数值的对话框。输入的有效数值将作为当前光标所对应存储单元的内容。如果输入多个数值，那么它们就依次作为当前光标所对应存储单元开始的若干存储单元的内容。如果欲修改的内存单元不在当前显示范围内，那么可先改变显示范围。

当焦点区域是数据显示区域时，按组合键 Alt+F10 会弹出一个适用于数据显示区域的操作菜单，选择其中的相应项，也可实现上述有关功能。

4. 查看和修改堆栈内容

尽管堆栈也是内存区域，但它又不同于普通数据区，所以 TD 还专门提供堆栈显示区域来显示当前堆栈的内容。从堆栈显示区域可看到堆栈顶的部分内容。

现在把焦点区域定到堆栈显示区域，那么在堆栈显示区域就出现一个光条，指示某个

内存单元，使用上下箭头键可移动该光条。通过上下翻页键和上下箭头键可调整所显示的堆栈范围。可以直接指定堆栈范围。按组合键 Ctrl+G，那么就会弹出一个用于输入定位地址的对话框。输入的内存区域首地址可包含段值部分。

修改堆栈的内容是方便的。设焦点区域是堆栈显示区域。按组合键 Ctrl+C 会弹出输入数值的对话框。直接按字符或数字键，也会弹出输入数值的对话框。输入的有效数值将作为当前光标所对应堆栈单元的内容。如果欲修改的堆栈单元不在当前显示范围内，那么可先改变显示范围。

5. 查看和修改代码区内容

在代码显示区域，可看到部分内存单元的内容作为机器指令反汇编的结果。每一行首由段值和偏移表示的地址，当段值与某个段寄存器内的内容相等时，段值部分用段寄存器表示。

现在把焦点区域定到代码显示区域，那么在代码显示区域就出现一个光条，指示某条指令，使用上下箭头键可移动该光标。通过上下翻页键和上下箭头键可调整所显示的代码区域，这种调整只是段内偏移的调整。可直接指定需要查看的代码区域首地址。按组合键 Ctrl+G，那么就会弹出一个用于输入定位地址的对话框。输入的代码区域首地址可包含段值部分，可以用段寄存器代表段值部分。如果地址表示成段值和偏移，那么段值和偏移需用冒号间隔。

可以在当前代码区域内查找指定的指令。设焦点区域是代码显示区域。那么按组合键 Ctrl+S 会弹出一个用于输入欲查找指令或者字节值表的对话框。

可方便地修改代码区域内的指令。设焦点区域是代码显示区域。直接按字符或数字键，就会弹出一个用于输入汇编格式指令的对话框，输入的指令被存放当前光条所指示地址开始的存储单元中。可连续输入多条汇编格式指令，它们被依次存放。

设焦点区域是代码显示区域。按组合键 Alt+F10 会弹出一个适用于代码显示区域的操作菜单。其中的 Assemble 项就用于汇编指令。另外，I/O 项可用于 I/O 端口操作。

6. 执行

按 F9 执行被调试程序。激活主菜单中的 Run 下拉菜单，选择 Run 项，执行被调试程序。在代码显示区域，有一个箭头符号指示当前指令位置，有时可能当前指令位置不在代码显示区域所显示的范围内。被调试程序从当前指令位置处开始执行，直到下列事件之一发生为止：程序结束；遇到断点；按组合键 Ctrl+Break 中止执行。

当出现某些意想不到的情形，程序似乎失去控制时，可试着按组合键 Ctrl+Break，使 TD 重新获得控制。只要一按 Ctrl+Break 键，它就立即起作用，因此程序有可能被中断在预想不到的代码位置上。Ctrl+Break 在下面两种情况下无法发挥作用：被调试程序已陷入禁止中断的循环中；由于执行了错误代码系统已崩溃。

按组合键 Ctrl+F2 键，可重新装载被调试程序。

按组合键 Alt+F5 键，可观察程序执行结果，再按任一键返回到调试界面。

7. 单步跟踪执行

TD 提供三个单步跟踪执行热键：F7、F8 和 Alt+F7。在汇编级调试时，按这三个热键之一通常都跟踪执行一指令。但遇下列指令时，这三个热键有差异。

当执行的指令是过程调用指令 CALL 时，按 F7 或 Alt＋F7 会跟踪进入过程，而按 F8 则直接执行完对应过程。类似地，当遇中断调用指令 INT、循环指令 LOOP 和串操作指令时，按 F8 就执行完对应指令。按 Alt＋F7 可跟踪进入中断处理程序。

在单步跟踪执行结束时，寄存器值显示区域和标志值显示区域就立即反映出寄存器内容变化情况，堆栈显示区域和数据显示区域也有相应改变。在代码显示区域，当前指令位置也就相应调整，有时代码显示区域所显示的范围也发生相应变化。

8. 执行到指定位置

如果想要从当前指令位置处开始，执行到某条指令时止，可利用 F4 键实现。先使代码显示区域成为焦点区域，那么在代码显示区域就出现一光条；然后利用上下箭头键或上下翻页键把光条定到希望停止执行的指令上；最后按 F4 键。如果没有遇到断点，而且执行经过指定的指令，那么执行就会在指定指令处停止。

此外，当代码显示区域是焦点区域时，按组合键 Alt＋F9 可直接指定执行停止位置。通常被调试从当前指令位置处开始执行，直到指定位置处为为止。但遇到断点或者不经过指定位置是例外。

9. 设置断点

当代码显示区域是焦点区域时，可很方便地设置断点。利用上下箭头键或上下翻页键把光条定到希望设置断点的指令上，然后按 F2 键就可在该处设置断点。设置断点的指令上出现一红色光条。按组合键 Alt＋F2 键会弹出一个用于输入断点位置的对话框。如果光条所在指令处已设置了断点，那么按 F2 键就取消该处的断点。选择主菜单 Breakpoints 中的 Delete all 项，可删除全部已设置的断点。

在设置断点后运行被调试程序，那么当执行到断点处时，就停止执行。寄存器值显示区域、标志值显示区域和其他区域都会及时反映有关变化。

参考文献

[1] 周明德主编. 保护方式下的 80386 及其编程. 北京：清华大学出版社. 1993 年 12 月

[2] 周明德等编著. 80x86 的结构与汇编语言程序设计. 北京：清华大学出版社. 1993 年 12 月

[3] 孟昭光等编著. 高档微机组成原理及接口技术. 北京：学苑出版社. 1993 年 11 月

[4] 沈美明等编著. IBM-PC 汇编语言程序设计. 北京：清华大学出版社. 1991 年 6 月

[5] 张载鸿编著. PC 系列机系统开发与应用(上). 北京：国防工业出版社. 1991 年 11 月

[6] 姚万生等编. IBM-PC 宏汇编语言程序设计. 哈尔滨：哈尔滨工业大学出版社. 1992 年 8 月

[7] 杨振生等编著, 8088/8086 汇编语言程序设计. 合肥：中国科学技术大学出版社. 1994 年 1 月

[8] 张怀莲等编, IBM-PC(8086/8088)宏汇编语言程序设计. 北京：电子工业出版社. 1991 年 3 月

[9] 吕强等编著. C 语言的 DOS 系统程序设计. 北京：清华大学出版社. 1994 年 6 月

[10] 白晓笛译. 80386/80286 汇编语言程序设计. 北京：电子工业出版社. 1988 年 6 月

[11] 求伯君等编. 新编深入 DOS 编程. 北京：学苑出版社. 1994 年 6 月。

[12] 田学锋等译. 80386/80486 编程指南. 北京：电子工业出版社. 1994 年 11 月。

[13] 索梅等编著. 80386/80286 汇编语言程序设计. 北京：清华大学出版社. 1994 年 6 月

[14] 邓洪涛编著. 80386/80486 汇编语言精要. 北京：清华大学出版社. 1995 年 6 月

[15] 钱培德等编著. CCDOS 操作系统技术大全. 北京：清华大学出版社. 1992 年 5 月

[16] 周明德等编著. 高档微型计算机(下册). 北京：清华大学出版社. 1989 年 5 月

[17] 朱巧明等编. 汉字系统使用大全. 长春：吉林大学出版社. 1994 年 12 月

[18] 李宝山编. ASM386 汇编语言. 北京：中国铁道出版社. 1989 年 5 月

[19] 王元珍等编. IBM-PC 宏汇编语言程序设计. 武汉：华中理工大学出版社. 1990 年 12 月

[20] John H. Crawford. Programming the 80386. SYBEX Inc. , 1987 年

[21] Intel Architecture Software Developer's Manual (Volume 1～3). 1997 年

[22] Pentium Processor Family Developer's Manual (Volume 1～3). 1997 年

附录　Pentium 指令与标志参考表

指令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT
AAA	—	—	—	TM	—	M				
AAD	—	M	M	—	M	—				
AAM	—	M	M	—	M	—				
AAS	—	—	—	TM	—	M				
ADC	M	M	M	M	M	TM				
ADD	M	M	M	M	M	M				
AND	0	M	M	—	M	0				
ARPL			M							
BOUND										
BSF/BSR	—	—	M	—	—	—				
BSWAP										
BT/BTS/BTR/BTC	—	—	—	—	—	M				
CALL										
CBW										
CLC										
CLD						0			0	
CLI								0		
CLTS										
CMC						M				
CMP	M	M	M	M	M	M				
CMPS	M	M	M	M	M	M		T		
CMPXCHG	M	M	M	M	M	M				
CMPXCHG8B			M							
CPUID										
CWD										
DAA	—	M	M	TM	M	TM				
DAS	—	M	M	TM	M	TM				
DEC	M	M	M	M	M					
DIV	—	—	—	—	—	—				
ENTER										
ESC										
HLT										
IDIV	—	—	—	—	—	—				
IMUL	M	—	—	—	—	M				
IN										
INC	M	M	M	M	M					

续表

指令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT
INS									T	
INT							0			0
INTO	T						0			0
INVD										
INVLPG										
IRET	R	R	R	R	R	R	R	R	R	T
Jcc	T	T	T		T	T				
JCXZ										
JMP										
LAHF										
LAR			M							
LDS/LES/LSS/LFS/LGS										
LEA										
LEAVE										
LGDT/LIDT/LLDT/LMSW										
LOCK										
LODS									T	
LOOP										
LOOPE/LOOPNE			T							
LSL			M							
LTR										
MOV										
MOV control,debug,test	—	—	—	—	—	—				
MOVS									T	
MOVSX/MOVZX										
MUL	M	—	—	—	—	M				
NEG	M	M	M	M	M	M				
NOP										
NOT										
OR	0	M	M	—	M	0				
OUT										
OUTS									T	
POP/POPA										
POPF	R	R	R	R	R	R	R	R	R	R
PUSH/PUSHA/PUSHF										
RCL/RCR 1	M					TM				
RCL/RCR count	—					TM				
RDMSR										
RDTSC										
REP/REPE/REPNE										

续表

指令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT
RET										
ROL/ROR 1	M					M				
ROL/ROR count	—					M				
RSM	M	M	M	M	M	M	M	M	M	M
SAHF		R	R	R	R	R				
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M				
SAL/SAR/SHL/SHR count	—	M	M	—	M	M				
SBB	M	M	M	M	M	TM				
SCAS	M	M	M	M	M	M			T	
SETcc	T	T	T		T	T				
SGDT/SIDT/SLDT/SMSW										
SHLD/SHRD	—	M	M	—	M	M				
STC						1				
STD									1	
STI								1		
STOS									T	
STR										
SUB	M	M	M	M	M	M				
TEST	0	M	M	—	M	0				
VERR/VERW			M							
WAIT										
WBINVD										
WRMSR										
XADD	M	M	M	M	M	M				
XCHG										
XLAT										
XOR	0	M	M	—	M	0				

注:M 表示指令影响标志;R 表示指令影响标志(根据操作数恢复标志);T 表示指令测试标志;0 表示清标志;1 表示置标志;—表示指令对标志的影响无定义;空表示指令不影响标志。