21 世纪高等学校电子信息类专业规划教材

汇编语言程序设计

殷肖川　主　编
秦　莲　孙　鹏　赵雪岩　姬伟锋　副主编

清 华 大 学 出 版 社
北京交通大学出版社
北京

内 容 简 介

汇编语言程序设计是高校计算机专业的经典课程之一。本书系统介绍了基于 80x86 的汇编语言程序设计方法和技术。主要内容包括: 80x86 指令系统、寻址方式、宏指令与伪指令、汇编语言格式与程序结构、分支程序设计、循环程序设计、子程序设计、宏汇编技术、系统功能调用与使用方法、I/O 程序设计方法与中断程序设计、C/C++ 语言与汇编语言混合编程技术、基于 Win32 的汇编程序开发技术等。全书深入讨论了各种实际应用问题和解决问题的方法,并给出了大量的实例。各章均附有习题,便于学生课后练习。附录部分给出了 80x86 指令表、BIOS 中断调用和 debug 命令表。

本书遵循理论与实践相结合的原则,系统地介绍了汇编语言程序设计的方法和技术,便于组织教学。此外,考虑到目前大部分应用系统都是基于 Windows 系统之上的原因,传统的基于 DOS 的应用平台已不能满足现实应用的需要,因此在内容安排上较为详细地介绍了 Win32 汇编编程技术及 Win32 汇编语言与 C/C++ 语言的混合编程方法,使读者能够对 Win32 汇编编程有一个初步的认识。

本书可作为高校计算机专业、自动化控制专业及相关专业本科生汇编语言程序设计课程的教科书,也可作为相关领域工程技术人员的参考用书。

图书在版编目(CIP)数据

汇编语言程序设计/殷肖川主编. —北京:清华大学出版社;北京交通大学出版社, 2005.1
(21 世纪高等学校电子信息类专业规划教材)
ISBN 7-81082-473-2

Ⅰ.汇… Ⅱ.殷… Ⅲ.汇编语言-程序设计-高等学校-教材 Ⅳ. TP313

中国版本图书馆 CIP 数据核字(2005)第 005350 号

责任编辑:彭立辉
出 版 者:清华大学出版社 邮编:100084 电话:010-62776969
北京交通大学出版社 邮编:100044 电话:010-51686414
印 刷 者:北京鑫海金澳胶印有限公司
发 行 者:新华书店总店北京发行所
开 本:185× 260 印张:19.25 字数:465 千字
版 次:2005 年 1 月第 1 版 2005 年 1 月第 1 次印刷
书 号:ISBN 7-81082-473-2/TP·175
印 数:1~4 000 册 定价:25.00 元

前　言

汇编语言是提供给用户直接访问计算机系统最快而又最为有效的一种编程语言。使用汇编语言编写程序能够充分发挥计算机硬件系统的功能,具有占用存储空间少、运行速度快及代码质量高等优点。那些需要对计算机硬件进行控制或对运行时间和效率有较高要求的系统软件或应用软件,通常都是用汇编语言编写的。此外,通过汇编语言程序设计的学习,学生能够对计算机系统的组成及工作原理有更深刻的理解。因此,汇编语言程序设计始终是作为高校计算机及相关学科的经典课程之一。

面对计算机技术的迅猛发展和操作系统的更新换代,传统的基于 DOS 平台的汇编程序设计已不能满足需要。本书从便于教学和注重实际应用出发,在内容编排上既兼顾以传统的 Intel 8086/8088 为代表的 16 位汇编程序设计,同时又以较大篇幅介绍了以 80386/80486/Pentium 为代表的 32 位汇编程序设计、Win32 汇编技术及 32 位汇编与 C/C++混合编程的方法。

本书正文共 9 章,从内容上可分 3 部分。

第一部分(第 1 ~6 章)主要介绍 8086/8088 指令系统及其汇编语言程序设计方法。

第 1 章介绍了学习汇编语言程序的基础知识,包括微型计算机系统的组成与发展概况、数据信息的表示及运算方法。

第 2 章介绍了微型计算机系统的组织结构,包括 Intel 8086/8088 微处理器、存储器及 I/O接口等内容。重点讨论了 Intel 8086 的内部结构和外部引脚功能,对其内部寄存器组的设置与功能进行了较详细的说明。

第 3 章针对 8086/8088 微处理器,详细介绍了其指令系统和寻址方式,对各类指令的指令格式、指令功能和使用方法进行了重点阐述。

第 4 章系统介绍了 MASM 汇编语言的语句格式、汇编程序结构及上机调试方法。介绍了常用伪指令、宏指令的使用方法。

第 5 章针对汇编语言的特点,详细介绍了汇编语言程序的设计方法。重点介绍了顺序结构、分支结构、循环结构、子程序及模块化程序设计技术,并结合大量实例对汇编语言程序的编程方法进行了详尽的说明。

第 6 章重点阐述了 I/O 程序和中断程序的设计方法。介绍了 I/O 的基本概念、I/O 控制方式、I/O 指令,叙述了中断的概念及其工作过程,列举出计算机系统中若干个常用的中断及其功能调用方法,详细介绍了 I/O 程序的编程设计方法,并给出了典型 I/O 程序的实例。

第二部分(第 7 章和第 9 章)主要介绍 32 位汇编及 Win32 汇编方法。

第 7 章简单介绍了常用 32 位 CPU 的处理器结构、寄存器组及工作方式,在此基础上重点阐述了常用 32 位扩展指令的功能及应用,最后介绍了 32 位程序的设计方法。

第 9 章介绍了汇编语言配合 API 开发基于 Win32 平台的应用程序的方法。包含程序框架、资源文件的使用和 Windows 消息处理 3 部分内容,借助 API 编写 32 位 Windows 程序,可

以充分利用 Windows 的高级特性，得到短小精悍的可执行文件。

第三部分（第 8 章）主要介绍汇编语言与 C/C++ 混合编程技术。

首先介绍了高级语言与汇编语言的混合编程方式和实际应用。在嵌入式汇编方式中，详细阐述了嵌入式汇编的编程方法和相关约定，介绍了利用嵌入式汇编语言编写 C/C++ 函数的具体方法，对嵌入式汇编指令如何访问 C/C++ 语言程序中的常量、变量和函数进行了描述，并给出了设计实例。

在模块调用方式中，阐述了模块间的连接方式，重点讨论了 C/C++ 语言程序调用汇编模块的方法，对调用接口、参数传递、返回值处理、寄存器的使用、变量的引用等进行了深入的分析，并结合应用示例进行了说明。

本书由殷肖川同志负责组织编写。具体编写分工是：第 1、3、4 章由秦莲编写；第 5、6、8 章由殷肖川编写；第 7、9 章由孙鹏编写；第 2 章由赵雪岩编写；姬伟锋同志参加了部分章节的编写和程序调试工作。

在本书的编写过程中，还得到了许多老师和研究生的支持：蔡飞华、李嘉生同志审阅了本书初稿，并提出宝贵意见；高丁、吴传芝、王欣同志对书中的实例及图表做了大量的工作。在此对他们的辛勤付出表示感谢。

由于编者水平所限，书中难免存在错误和不妥之处，敬请广大读者批评指正。

作 者

2005 年 1 月

目　　录

第 1 章　汇编语言基础知识 …… (1)
1.1　概述 …… (1)
1.1.1　微型计算机的发展与应用 …… (1)
1.1.2　微型计算机系统的主要性能指标 …… (2)
1.2　进位计数制及相互转换 …… (3)
1.2.1　进位计数制 …… (3)
1.2.2　进位计数制的相互转换 …… (4)
1.3　数值信息表示 …… (5)
1.3.1　计算机中数的表示方法 …… (5)
1.3.2　微型计算机的算术运算 …… (7)
1.4　字符表示法 …… (9)
1.5　基本逻辑运算 …… (9)
1.6　程序设计语言 …… (11)
1.6.1　机器语言 …… (11)
1.6.2　汇编语言 …… (11)
1.6.3　高级语言 …… (11)
习题 …… (12)
第 2 章　微型计算机系统组成 …… (14)
2.1　微型计算机系统硬件结构 …… (14)
2.1.1　结构特点与框图 …… (14)
2.1.2　主要组成部分及功能 …… (15)
2.2　8086/8088 微处理器 …… (17)
2.2.1　内部结构 …… (17)
2.2.2　引脚及功能 …… (19)
2.3　存储器组成 …… (23)
2.4　系统总线 …… (24)
2.5　输入/输出接口 …… (25)
2.5.1　I/O 接口概述 …… (25)
2.5.2　I/O 端口的编址方式 …… (25)
2.5.3　I/O 同步控制方式 …… (26)
2.6　80x86 系列微处理器简介 …… (27)
2.7　微型计算机软件系统 …… (29)

习题 …… (29)
第3章 8086寻址方式与指令系统 …… (31)
3.1 8086的寻址方式 …… (31)
3.1.1 有效地址EA …… (31)
3.1.2 段约定和段更换 …… (32)
3.1.3 立即寻址 …… (32)
3.1.4 寄存器寻址 …… (32)
3.1.5 存储器寻址方式 …… (33)
3.2 8086指令系统 …… (35)
3.2.1 传送指令 …… (35)
3.2.2 算术运算指令 …… (38)
3.2.3 逻辑运算指令 …… (45)
3.2.4 移位指令 …… (47)
3.2.5 串操作指令 …… (48)
3.2.6 控制转移指令 …… (51)
3.2.7 转移指令 …… (52)
3.2.8 调用和返回指令 …… (54)
3.3 处理器控制指令 …… (55)
3.3.1 标志操作指令 …… (55)
3.3.2 其他控制指令 …… (55)
习题 …… (56)
第4章 8086汇编语言 …… (58)
4.1 汇编语言源程序格式 …… (58)
4.2 伪指令语句 …… (59)
4.2.1 程序结构伪指令语句 …… (59)
4.2.2 过程和宏定义伪指令语句 …… (70)
4.2.3 条件汇编伪指令语言 …… (71)
4.2.4 列表伪指令语句 …… (73)
4.3 汇编语言程序的调试与运行 …… (74)
4.3.1 上机调试过程 …… (74)
4.3.2 常用DEBUG命令 …… (77)
习题 …… (81)
第5章 汇编程序设计 …… (82)
5.1 程序设计方法 …… (82)
5.2 顺序程序设计 …… (82)
5.3 分支程序设计 …… (84)
5.3.1 分支结构 …… (84)

5.3.2 用分支指令实现分支结构程序 …… (86)
5.3.3 用伪指令实现分支结构 …… (90)
5.4 循环程序设计 …… (92)
5.4.1 循环结构 …… (92)
5.4.2 单循环程序设计 …… (94)
5.4.3 多重循环程序设计 …… (97)
5.4.4 用伪指令实现循环结构 …… (100)
5.5 子程序设计 …… (101)
5.5.1 子程序定义 …… (102)
5.5.2 子程序的调用和返回 …… (103)
5.5.3 子程序的参数传递 …… (104)
5.5.4 子程序嵌套与递归 …… (110)
5.6 模块化程序设计 …… (112)
5.6.1 模块划分 …… (112)
5.6.2 源程序文件包含 …… (113)
5.6.3 模块间的连接 …… (114)
习题 …… (115)
第 6 章 输入/输出与中断控制 …… (116)
6.1 I/O 概述 …… (116)
6.1.1 I/O 接口 …… (116)
6.1.2 端口编址方式 …… (116)
6.1.3 I/O 指令 …… (117)
6.1.4 I/O 控制方式 …… (118)
6.2 简单 I/O 程序举例 …… (119)
6.3 中断系统 …… (121)
6.3.1 中断和中断源 …… (122)
6.3.2 中断向量表 …… (122)
6.3.3 中断服务程序 …… (123)
6.3.4 设置中断向量 …… (124)
6.3.5 中断功能分类 …… (125)
6.4 系统功能调用与 BIOS 中断 …… (126)
6.4.1 调用方式 …… (126)
6.4.2 系统功能调用 …… (126)
6.4.3 BIOS 中断调用 …… (128)
6.5 软中断开发 …… (143)
6.5.1 软中断开发方法 …… (143)
6.5.2 中断重定向 …… (145)

6.5.3 驻留中断程序 …… (146)
习题 …… (148)
第7章 32位指令及其编程 …… (150)
7.1 32位微处理器结构 …… (150)
7.1.1 80386微处理器结构 …… (150)
7.1.2 Pentium微处理器结构 …… (151)
7.1.3 Pentium微处理器基本寄存器组 …… (154)
7.1.4 Pentium微处理器系统级寄存器组 …… (156)
7.2 80x86 CPU的工作方式 …… (158)
7.3 32位扩展指令 …… (161)
7.3.1 新增的寻址方式 …… (161)
7.3.2 常用32位扩展指令 …… (162)
7.4 32位程序设计 …… (168)
7.4.1 32位汇编开发环境 …… (168)
7.4.2 实模式下的编程 …… (170)
7.4.3 保护模式下的编程 …… (177)
7.4.4 程序实例 …… (179)
习题 …… (181)
第8章 汇编语言与C/C++混合编程 …… (183)
8.1 混合编程方式 …… (183)
8.2 C/C++的嵌入式汇编 …… (183)
8.2.1 在C/C++程序嵌入汇编语句 …… (184)
8.2.2 在嵌入式汇编中访问C/C++的数据 …… (187)
8.2.3 用汇编程序段编写C函数 …… (190)
8.2.4 在嵌入式汇编中调用C/C++函数 …… (191)
8.3 用C/C++调用汇编模块 …… (194)
8.3.1 接口约定 …… (194)
8.3.2 调用汇编模块 …… (196)
习题 …… (200)
第9章 Win32程序设计 …… (201)
9.1 汇编语言Win32程序简介 …… (201)
9.1.1 汇编语言Win32程序框架 …… (201)
9.1.2 简单Win32应用程序设计 …… (204)
9.2 资源文件的使用 …… (207)
9.2.1 资源文件的作用 …… (207)
9.2.2 资源文件在汇编中的应用 …… (209)
9.2.3 编程实例 …… (210)

9.3 Win32 程序设计实例 …… (219)
9.3.1 WM_PAINT 消息的处理 …… (219)
9.3.2 键盘消息处理 …… (224)
9.3.3 鼠标消息处理 …… (227)
习题 …… (230)
附录 A ASCII 码表 …… (231)
附录 B DOS 和 BIOS 的宏定义 …… (232)
附录 C DEBUG 命令表 …… (241)
附录 D 中断列表 …… (242)
附录 E Pentium 指令的执行周期数 …… (285)
参考文献 …… (297)

第 1 章　汇编语言基础知识

计算机系统分为硬件和软件两大部分，硬件（Hardware）是计算机的物理实体；软件（Software）则是为了运行、管理和维护计算机而编制的各种程序的集合。计算机程序设计语言是开发软件的工具，它的发展经历了由低级语言到高级语言的过程，而汇编语言是一种面向机器的低级语言。汇编语言程序运行速度快，占用内存容量少，这些都是高级语言无法代替的，作为一个计算机专业人员，必须掌握好汇编语言程序设计方法。本章重点介绍学习汇编语言程序需要掌握的基础知识；微型计算机系统的产生与发展概况；微型计算机内部基本数制及其相互转换的方法；原码、反码、补码的定义与运算规则；编码的使用与运算规则；逻辑与、或、非、异或运算规则。

1.1　概　　述

自 1946 世界上第一台数字电子计算机 ENIAC（Electronic Numerical Integrator And Calculator）诞生以来，计算机的发展突飞猛进，短短几十年中，已经历了电子管计算机、晶体管计算机、集成电路计算机和大规模/超大规模集成电路计算机等 4 代的发展历程。作为第四代计算机的一个重要分支，微型计算机于 20 世纪 70 年代初诞生了。微型计算机（Microcomputer）与其他大、中、小型计算机的区别，主要在于其中央处理器 CPU（Central Processing Unit）采用了大规模、超大规模集成电路技术，而其他类型计算机的 CPU 则是由相当多的分立元件电路或集成电路组成。通常把微型计算机的 CPU 芯片称为微处理器 MPU（Micro Processing Unit 或 Microprocessor）。

1.1.1　微型计算机的发展与应用

微处理器的发展决定着微型计算机的更新换代，微处理器的集成度几乎每两年增加一倍，产品每 2～4 年更新换代一次，现已进入第 5 代。各代的划分通常以 MPU 的字长和速度为主要依据。

第一代 4 位微处理器以 Intel 公司的 4004 为代表，它虽然简单，运算能力不强，速度不快，但它的问世标志着计算机的发展进入了一个新纪元；第二代 8 位处理器的典型产品有 Intel8008/8080、Motorola 的 MC6800、Zilog 的 Z80 等；第三代 16 位微处理器的典型产品有 Intel8086/8088、MC68000/68010、Z8000 等；第四代 32 位微处理器的典型产品有 Intel80386/80486、MC68020/68030 等；第五代 64 位微处理器则以 Intel 公司先后推出的 Pentium/PentiumPro、AMD、Cgrix 公司的 6X86/MediaGX/6X86M 和 IBM, Apple, Motorola 三大公司共同开发的 Power PC 等为代表产品。

随着微型计算机的发展，其应用范围不断扩大，归纳起来应用领域主要包含以下几个方面。

（1）数据处理

在科学研究、工程设计和经济管理中，存在大量复杂的数学计算问题，利用微型计算机

可快速得到较理想的结果。

(2) 过程控制

在农业、国防、交通等领域,利用微型计算机对生产和试验过程进行自动实时监测、控制和管理,可提高效率、提高质量、降低成本、缩短生产和试验周期。

(3) 信息管理

采用目前迅猛发展的计算机网络技术,可实现信息管理的自动化和办公自动化。常用于财务管理、人事档案管理、情报资料管理等。

(4) 辅助设计与仿真

为了提高自动化水平,目前普遍借助计算机进行设计,即计算机辅助设计(Computer Aided Design)、计算机辅助测试(Computer Aided Test)、计算机辅助制造(Computer Aided Manufacture)、计算机集成制造系统(Computer Aided and Manufacture System)、计算机辅助教学(Computer Aided Instruction)等。

(5) 人工智能

用微型计算机系统来模拟人类某些智能行为,包括声音、图像、文字等模式识别、自然语言理解等。

微型计算机的分类方法有多种。

单片机是最简单的微型计算机,它仅由一块超大规模集成电路组成,CPU、存储器、I/O接口电路和总线制作在一块很小的芯片上。

单板机规模比单片机大,它的 CPU 是一块单独的大规模集成电路芯片,存储器和I/O接口电路也各是一块或几块大规模集成电路芯片。这些芯片加上单板机若干附加逻辑电路和简单的键盘/数码显示器装在一块印刷电路板上,构成一个单板机。

多板机即通常所说的台式微型计算机,指由 CPU 芯片、存储器芯片、I/O 接口电路、I/O 适配器和必要的外部设备(键盘、CRT 显示器、磁盘/光盘驱动器等)组成的整机系统。

1.1.2　微型计算机系统的主要性能指标

衡量微型计算机系统的主要性能指标与其他大、中、小型计算机系统的衡量指标基本相同,主要有字长、存储器容量、运算速度、外设扩展、软件配置等方面。

1. 字长

字长是计算机内部一次可以处理的二进制数码的位数。一般一台计算机的字长决定于它的 CPU 内部通用寄存器的位数、内部存储器的位数、算术逻辑运算单元(ALU)的位数和数据总线的宽度。字长越长,一个字所能表示的数据精度就越高,在完成同样精度的运算时,其运算速度越快。如 8086 的数据总线为 16 位,字长为 16,称之为 16 位机;80486 的数据总线为 32 位,字长为 32,称之为 32 位机。

2. 存储器容量

存储器容量是衡量计算机存储二进制信息量大小的一个重要指标。微型计算机一般以字节 B 为单位表示存储器容量,通常将 1 024 B 简称为 1 KB,1 024 KB 简称为1 MB,1 024 MB 简称为 1 GB,1 024 GB 简称为 1 TB。

存储器容量包括内存容量和外存容量。内存容量又分为最大容量和实际装机容量。最大容量由 CPU 的地址总线的位数决定,如 8080 的地址总线为 16 位,其内存最大容量为

64 KB;8086 的地址总线为 20 位,其内存最大容量为 1 MB;80286 的地址总线为 24 位,其内存最大容量为 16 MB;80386/80486 的地址总线为 32 位,其内存最大容量为 4 GB 等。

3. 运算速度

计算机的运算速度一般用每秒钟所能执行的指令条数表示。常用的衡量标准是:直接给出的 CPU 主频和每条指令执行所需的时钟周期,主频越高执行速度越快。主频一般以 MHz 为单位。例如,Intel 8086 的主频为 4.77 MHz, Intel 80286 的主频为 6 MHz, Intel 80386 的主频为 16 MHz, Intel 80486 的主频为 50 MHz, PentiumⅢ的主频为 600 MHz。

4. 外设扩展能力

这主要指计算机系统配接各种外部设备的可能性、灵活性和适应性。一台计算机允许配接多少外部设备,对于系统接口和软件研制都有重大影响。在微型计算机系统中,外存容量、USB 接口、多媒体设备配置等都是外设配置中需要考虑的问题。

5. 软件配置情况

软件是计算机系统必不可少的重要组成部分,它配置是否齐全,直接关系到计算机性能的好坏和效率的高低。软件主要分为系统软件和应用软件,它包括操作系统软件(如 DOS, Windows, UNIX, Windows NT 等)、应用软件(如 C++, SQL Server, VB 等)、文字处理软件(如 Office, WPS 等)、图形处理软件(Flash, AutoCAD 等)。软件配置情况是关系到微型计算机性能好坏的重要因素之一。

1.2　进位计数制及相互转换

1.2.1　进位计数制

人们已经习惯了十进制数,但是计算机则用二进制形式表达数值,为了便于表示二进制数,还经常使用到十六进制数。常用进位计数制如下:

二进制　　R = 2　　D_i = 0, 1

八进制　　R = 8　　D_i = 0, 1, 2, 3, 4, 5, 6, 7

十进制　　R = 10　　D_i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

十六进制　　R = 16　　D_i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

其中,R 表示基数,D_i 为基本数字,可见,二进制数只有两个基本数字,逢二进位;八进制数只有 8 个基本数字,逢八进位;十进制数只有 10 个基本数字,逢十进位;十六进制数只有 10 个基本数字,6 个基本字母,逢十六进位。通常,在二进制数尾加 B 作为标识,十进制数尾加 D 标识(可省略),八进制数加 Q 标识,十六进制数加 H 标识。

例如,分别写出 63 和 232 的二进制数、八进制数、十六进制数。

63 = 00111111B = 77Q = 3FH

232 = 11101000B = 350Q = E8H

对于各种进制的算术运算规则,可以用实例来说明。例如,完成 11011101B + 01111011B 和 11110001B - 00101011B 二进制数运算。

```
    11011101B                    11110001B
+   01111011B                -   00101011B
  101011000B                     11000110B
```

1.2.2　进位计数制的相互转换

由于计算机内部是由一系列逻辑电路组成的,因此在计算机内部的运算与存储都是采用二进制数,而二进制数对于人们日常的书写、记忆都很不方便。虽然十进制数是我们常用的数制,但计算机又不能直接使用,所以需要对各种数制进行转换。

1. 二进制与十进制的相互转换

(1) 十进制转换为二进制

十进制转换为二进制的规则如下。

整数部分:将要转换的十进制数的整数部分用 2 除,取其余数,直至商为零,将其余数按由下向上的顺序排列。小数部分:将要转换的十进制数的小数部分用乘 2 取整法,取其整数,直至小数部分为 0,将整数按由上向下的顺序排列即得转换后的二进制数。

【例 1.1】　将十进制数 X = 107.125D 转换为二进制数。

X 整数部分为 107D,其转换步骤如下:

```
107/2 = 53      (D0 = 1)
53/2 = 26       (D1 = 1)
26/2 = 13       (D2 = 0)
13/2 = 6        (D3 = 1)
6/2 = 3         (D4 = 0)
3/2 = 1         (D5 = 1)
1/2 = 0         (D6 = 1)
```

X 小数部分为 0.125D,其转换步骤如下:

```
0.125× 2 = 0.25    (D0 = 0)
0.25× 2 = 0.5      (D1 = 0)
0.5× 2 = 1         (D2 = 1)
```

X = 1101011.001B

(2) 二进制转换为十进制

其转换规则是将要转换的二进制数按权展开计算。

【例 1.2】　将二进制数 X = 10011001.10101B 转换为十进制数。

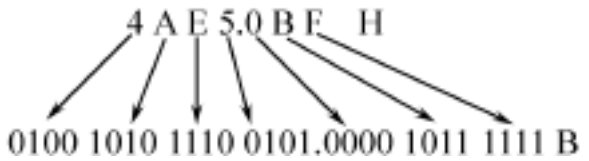
10011001.10101B =

$$1\times 2^{7}+1\times 2^{4}+1\times 2^{3}+1\times 2^{0}+1\times 2^{-1}+1\times 2^{-3}+1\times 2^{-5}=153.65625D$$

2. 十六进制与二进制的相互转换

(1) 十六进制转换为二进制

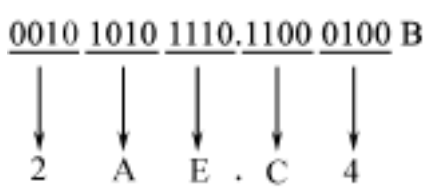
用 4 位二进制数取代每一位十六进制数,即得对应的十六进制数。例如,



(2) 二进制转换为十六进制

首先从小数点开始分别向左和向右把整数和小数部分每 4 位分成一组。若整数最高位的一组不足 4 位, 则在其左边补 0; 若小数最低位一组不足 4 位, 则在其右边补 0。然后用十六进制数取代每组的 4 位二进制数, 即得对应的十六进制数。

例如:



1.3　数值信息表示

计算机中的数是用二进制表示的, 数的符号也只能用 0 和 1 来表示, 一般用最高有效位表示数的符号, 正数用 0 表示, 负数用 1 表示。对于数值信息的表示方法, 常用的有原码、反码和补码, 各种码制有不同的定义方法和表示范围。

1.3.1　计算机中数的表示方法

1. 机器数和真值

计算机中使用的包含符号位的数, 称为机器数。机器数所表示的真实值叫真值。例如,

10110101(机器数)　　真值: - 53D 或 - 0110101B

00101010(机器数)　　真值: + 42D 或 + 0101010B

2. 有符号数的表示方法

对有符号数, 机器数常用的表示方法有原码、反码、补码 3 种。

(1) 原码

最高位表示符号, 数值为二进制绝对值表示的方法, 即为原码表示方法。例如,

$(+64D)_{原} = 01000000B$

$(-64D)_{原} = 11000000B$

注意:

n 位原码表示数值的范围是 $-(2^{n-1}-1) \sim +(2^{n-1}-1)$。

0 的原码有两种不同形式, 即 $(+0)_{原} = 00000000$ 和 $(-0)_{原} = 10000000$。

原码表示简单、直观、且与真值间转换方便, 但用它进行加减运算不方便。

(2) 反码

正数的反码与原码表示相同。负数的反码是将其对应的正数各位(连同符号位)取反得到, 或将其原码除符号位外各位取反得到。例如,

$(+64D)_{反} = (+64D)_{原}$

$(-64D)_{反}=10111111B$

注意:

n 位反码表示数值的范围是 $-(2^{n-1}-1)\sim+(2^{n-1}-1)$。

0 的反码也有两种形式,即 $(+0)_{反}=00000000$　$(-0)_{反}=11111111$。

将反码还原为真值的方法是反码 →原码 → 真值,而 $(X)_{原}=((X)_{反})_{反}$。最高位为 0 时,后面的二进制序列值即为真值,且为正;最高位为 1 时,则为负数,后面的数值要按位求反才为真值。

(3) 补码

正数的补码表示与原码相同。负数的补码是将其对应的正数各位(连同符号位)取反加 1(最低位加 1)而得到,或将其原码除符号位外各位取反加 1 而得到。例如:

$(+64D)_{补}=(+64D)_{反}=(+64D)_{原}$

$(-64D)_{补}=11000000B$

注意:

n 位补码表示数值的范围是 $-2^{n-1}\sim+(2^{n-1}-1)$。

它对应于补码 100…0 ~011…1。

0 的补码只有一个,即 $(+0)_{补}=(-0)_{补}=000\cdots0$(全 0)。

将补码还原为真值的方法是补码 → 原码 → 真值,而 $(X)_{原}=((X)_{补})_{补}$。若补码的符号位为 0 时,则其后的数值即为真值,且为正;若符号位为 1,则应将其后的数值位按位取反加 1,所得结果才是真值,且为负。

综上所述,可以得出以下几点结论。

① 原码、反码、补码的最高位都是符号位。符号位为 0 时,表示真值为正数,其余位为真值;符号位为 1 时,表示真值为负,其余位除原码外不再是真值。对于反码,需按位取反才是真值;对于补码,则需按位取反加 1 才是真值。

② 对于正数,3 种编码都是一样的,即 $(X)_{原}=(X)_{反}=(X)_{补}$;对于负数,3 种编码互不相同。所以,原码、反码、补码本质上是用来解决负数在机器中表示的 3 种不同的编码方法。

③ 二进制位数相同的原码、反码、补码所能表示的数值范围不完全相同。以 8 位为例,它们表示的真值范围分别为:

原码 -127 ~+127

反码 -127 ~+127

补码 -128 ~+127

④ 上面讨论的原码、反码、补码都是针对真值 X 为整数而言的。若真值 X 为小数(纯小数),则以 -0.75 为例(n=8),其原码、反码、补码的定义应为:

$(0.75)_{原}=1-(-0.1100000)=1.11000000$

$(-0.75)_{反}=2-2^{-7}-0.1100000=1.0011111$

$(-0.75)_{补}=2-0.1100000=1.0100000$

最后需要说明的是,当计算机采用不同的码制时,运算器和控制器的结构将不同。采用原码形式的计算机称为原码机,类似的有反码机和补码机。目前以补码机居多,各种微型计算机基本上都是以补码作为机器码,原因是补码的加减法运算简单,减法运算可变为加法运算,可省掉减法器电路;而且它是符号位与数值位一起参加运算,运算后能直接获得正确结果。

3．数的定点和浮点表示

当所要处理的数含有小数部分时，就有一个如何表示小数点的问题。在计算机中并不用某个二进制位来表示小数点，而是隐含规定小数点的位置。根据小数点的位置是否固定，数的表示方法可分为定点表示和浮点表示，相应的机器数就叫定点数和浮点数。在计算机中，小数点的位置是固定不变的，称之为定点数。小数点的位置是浮动的，称之为浮点数。根据小数点固定的位置不同，定点数有定点(纯)整数和定点(纯)小数两种。当要处理的数是既有整数又有小数的混合数时，采用定点数格式很不方便。为此，人们一般采用浮点数进行运算。

4．无符号数的表示方法

无符号数在计算机中通常有 3 种表示方法，位数不等的二进制码、BCD 码、ASCII 码，其中 BCD 码的表示形式一般又有两种，压缩 BCD 码(或叫组合 BCD 码、紧凑 BCD 码)和非压缩 BCD 码(或叫非组合 BCD 码、非紧凑 BCD 码)。前者每位 BCD 码用 4 位二进制数表示，一个字节(8 位二进制)表示 2 位 BCD 码，如 10010011B 表示十进制数 93；后者每位 BCD 码用一个字节表示，高 4 位总是 0000，低 4 位的 0000～1001 表示 0～9。例如，93 用非压缩 BCD 码表示时，需要两个字节(16 位二进制数)，00001001 | 00000011。

ASCII 码表示与非压缩 BCD 码表示很相似，低 4 位都是用 0000～1001 表示 0～9，差别仅在高 4 位，ASCII 码不是 0000，而是 0011。ASCII 码一般在计算机的输入、输出设备中使用，而二进制码和 BCD 码则在运算、处理过程中使用。因此，在应用计算机解决实际问题时，常常需要在这几种机器码之间进行转换。

1.3.2　微型计算机的算术运算

1．补码的加减法运算

(1) 加减法运算规则

补码的加减法运算规则可用下式表示：

$$(X \pm Y)_{补} = (X)_{补} + (\pm Y)_{补}$$

其中，X、Y 为正或负数均可。

例如，已知 X = 33，Y = 45，求 X + Y，X - Y。

解　$(X)_{补} = 00100001$

$(Y)_{补} = 00101101$

$(-Y)_{补} = 11010011$

所以，

$$X + Y = ((X)_{补} + (Y)_{补})_{补} = 01001110 = (+78)_{10}$$

$$X - Y = ((X)_{补} + (-Y)_{补})_{补} = 10001100 = (-12)_{10}$$

显然，上述结果是正确的。从上述补码运算规则和举例可看出，用补码表示计算机中的有符号数优点明显。第一，负数的补码与对应正数的补码之间的转换可用同一方法——求补运算实现，因而可简化硬件；第二，可将减法变为加法运算，从而省去减法器；第三，有符号数和无符号数的加法运算可用同一加法器完成，结果都是正确的。

(2) 溢出与溢出判断

当结果超出补码表示的数值范围时，上述补码运算就不正确了，计算机怎样判断是否产

生溢出呢? 通常有 3 种方法。

① 符号比较法: 两个同符号数相加, 若“和数”符号与加数符号不同, 或者两个异符号数相减, 若“差数”符号与被减数符号不同, 则表示产生了溢出。两个异符号数相加或两个同符号数相减, 是不可能产生溢出的。

② 双符号位法(也叫变形补码法): 对参与运算的数在运算过程中采用两个符号位(正数由 0 扩展为 00, 负数由 1 扩展为 11), 若运算结果的“和”或“差”的两个符号位不同, 表示有溢出, 相同表示无溢出。

③ 双进位法: 加减运算后,“和数”与“差数”中的符号位的进位输入 Cin(即最高数值位向符号位的进位) 与进位输出 Cout(即符号位向进借位的进位) 若不同, 说明有溢出; 若相同, 说明无溢出。

例如, 求 55 +66。

$$\begin{array}{rl} & [55]_{补} = 00110111 \\ + & [66]_{补} = 01000010 \\ \hline & 01111001 = [121]_{补} \end{array}$$

用双进位法判断溢出因为 C0 =0, Ci = 0, OF = Ci 异或 C0 = 0, 所以无溢出, 结果正确。

Cout = C0　　Cin = Ci

例如, 求 98 +45。

$$\begin{array}{rl} & [98]_{补} = 01100010 \\ + & [45]_{补} = 00101101 \\ \hline & 10001111 \end{array}$$

用双进位法判断溢出因为 C0 =0, Ci = 1, OF = Ci 异或 C0 =1, 所以有溢出, 结果错误。

2. BCD 码运算及其十进制调整

进行 BCD 码加减法运算时, 每组 4 位二进制码表示的十进制数之间应该遵循“逢十进一”和“借一当十”的规则。但是, 由于计算机总是将数作为二进制数来处理的, 即每 4 位之间总是按“逢十六进一”和“借一当十六”来处理, 所以当 BCD 码运算出现进位和借位时, 结果将出错, 应对其进行十进制调整。

(1) 十进制加法调整规则

若两个一位 BCD 数相加结果大于 9(即 1001) 则修正。和数大于 9 时, 说明有进位, 而 4 位二进制数相加只有结果超过 15 才会进位, 所以要做加 6 修正。

若两个 BCD 数相加结果在本位并不大于 9, 但产生了进位, 这相当于十进制运算大于等于 15, 所以也应在本位做加 6 修正。

例如, 求 BCD 码 57 +65。

$$\begin{array}{rll} & 0101 & 0111 \\ + & 0110 & 0101 \\ \hline & 1011 & 1100 \\ + & 0110 & 0110 \\ \hline & 0001 & 0010 \end{array}$$

(2) 十进制减法调整规则

两个 BCD 数相减, 若出现本位差超过 9, 或虽不超过 9 但向高位有借位, 则说明必然是借了 16, 多借了 6, 所以应在本位做减 6 修正。

例如, 求 BCD 码 71 - 29。

```
    0111  0001
-   0010  1001
-------------
    0100  1000
-           0110
-------------
    0100  0010
```

1.4　字符表示法

计算机中处理的信息并非全是数, 经常需要处理字符或字符串, 例如从键盘输入的信息或打印机输出的信息等都是用字符方式输入输出的, 所以, 在计算机内部对字符的表示通常使用的是美国信息交换标准代码 ASCII 来表示。这种代码用一个字节来表示一个字符, 其中低 7 位为字符的 ASCII 值, 最高一位一般作为校验位, 校验位为 0 是基本的 ASCII 码, 校验位为 1 是扩展的 ASCII 码。ASCII 码字符如下。

字母: A, B, …, Z; a, b, …, z。

数字: 0, 1, 2, …, 9。

专用字符: + , - , * , /, SP 等。

非打印字符: BEL(响铃) 、LF(换行) 、CR(回车) 等。

ASCII 码详见附录 A。

1.5　基本逻辑运算

在计算机中, 一切运算最终都归结为对 0 和 1 的运算, 实质是逻辑代数运算。逻辑代数(也称布尔代数、开关代数) 的应用为计算机提供了判断、状态测试和数字逻辑运算的能力, 并为计算机设计提供了重要的工具。逻辑代数用二进制数字 1 和 0 确定逻辑判断, 即真与假、是与非等。

逻辑或(OR) 、逻辑与(AND) 、逻辑非(NOT) 是逻辑代数中 3 种最基本的运算。任何二进制数通过这 3 种运算及其组合都可以得到一个二进制数字结果, 和普通代数一样, 用字母表示逻辑变量, 但变量取值只有 1 或 0。

1. “ 或 ”运算

逻辑“ 或 ”运算规则如下。

如果逻辑变量 A 或 B 等于 1, 或两者都等于 1, 则结果为 1, 否则为 0。通常用加号“ + ”或符号“ ∨ ”表示“ 或 ”, 运算规则为:

$0 \vee 0 = 0$

$0 \vee 1 = 1$

$1 \vee 0 = 1$

$1 \vee 1 = 1$

由此可见,“ 或 ”运算与二进制加法的惟一差别在于 1 + 1 = 1 而不是等于 10, 无进位关系。“ 或 ”运算又称为逻辑加。

2. “ 与 ”运算

逻辑“ 与 ”运算规则如下。

如果逻辑变量 A 或 B 都等于 1, 则结果为 1, 否则为 0。通常用乘号“ * ”或符号“ ∧ ”表示“ 与 ”, 运算规则为:

0 ∧ 0 = 0

0 ∧ 1 = 0

1 ∧ 0 = 0

1 ∧ 1 = 1

可以看出其运算规则与代数乘法相同, 故逻辑“ 与 ”也称为逻辑乘。只有当逻辑变量 A 和 B 都为 1 时, 结果才为 1。

3. “ 非 ”运算

逻辑“ 非 ”运算规则如下。

输出结果与输入值相反, 记为 $F = \overline{A}$, 上面的横杠表示非, 读作“ A 非 ”。逻辑非运算可用三极管构成的反相器来实现, 因此也称非门逻辑为反相器。

以上讨论了一位逻辑数的 3 种基本运算规则, 下面对两个 n 位逻辑数给出 3 种基本运算示例。

例如, 有两个 8 位逻辑数 X, Y, 其中 X = 10101101 Y = 11001011, 求 X ∨ Y, X ∧ Y, $\overline{X}$。

10101101	10101101	X = 10101101
∨ 11001011	∧ 11001011	$\overline{X}$ = 01010010
11101111	10001001	

从上例中可以看出两个 n 位数进行逻辑或运算, 凡数位中有 1 者, 结果对应位均为 1; 而两个 n 位数进行逻辑与运算, 凡数位中有 0 者。结果位均为 0; 对 n 位数进行逻辑非运算就是按位求其反值。

4. “ 异或 ”运算

逻辑“ 或 ”、“ 与 ”、“ 非 ”3 种基本逻辑运算, 可以构成一个完备的逻辑代数系统。在实际问题中, 这些基本逻辑运算很少单独出现, 而是通过这 3 种基本运算组成更复杂的逻辑运算, 其函数关系可用逻辑表达式反映。例如,“ 异或 ”是一种很有用的复合逻辑, 它实现的逻辑功能为, $A\overline{\oplus}B = A\overline{B} \vee \overline{A}B$。

该逻辑运算称为异或运算, 也称按位模 2 加法, 简称按位加, 通常用符号“ $\overline{\oplus}$ ”表示异或运算, 运算规则为:

$0\overline{\oplus}0 = 0$　$1\overline{\oplus}0 = 1$

$0\overline{\oplus}1 = 1$　$1\overline{\oplus}1 = 0$

可以看出, 异或运算就是一位二进制数不考虑进位的加法运算。

例如, X = 11001001　Y = 10101011, 求 $X\overline{\oplus}Y$。

解　列出按位加式

$$\begin{array}{rr} & 11001001 \\ \overline{\oplus} & 10101011 \\ \hline & 01100010 \end{array}$$

所以 $X\overline{\oplus}Y = 01100010$

1.6　程序设计语言

程序设计语言的种类很多,总体可分成低级语言和高级语言。低级语言有机器语言和汇编语言,高级语言有 C/C++, Pascal, Basic 等。

1.6.1　机器语言

计算机能够直接识别的是二进制数 0 和 1 组成的代码。机器指令(Instruction) 就是二进制编码指令,一条机器指令控制计算机完成一个操作。每种处理器都有各自的机器指令集,某处理器支持的所有指令的集合(Instruction Set) 就是该处理器的指令系统。指令集及使用它们编写程序的规则被称为机器语言(Maching Language) 。用机器语言编写的程序是计算机惟一能够直接识别并执行的程序,而用其他语言程序编写的程序必须经过翻译变换成机器语言程序,所以机器语言程序常被称为目标程序(或目的程序) 。

机器指令一般由操作码(Opcode) 和操作数(Operand) 构成。操作码是表明操作性质的代码,操作数则表明参加操作的数或数所在的地址。一条机器指令是一组二进制代码,一个机器语言程序就是一段二进制代码序列。

用机器语言编写程序的最大缺点是难以理解,因而极易出错,也难以发现错误。所以,只是在计算机发展的早期或不得已的情况下,才用机器语言编写程序。现在,除了有时在程序某处需要直接采用机器指令填充外,几乎没有人采用机器语言编写程序。

1.6.2　汇编语言

为了克服机器语言的缺点,人们采用便于记忆并能描述指令功能的符号来表达机器指令。表示指令操作码的符号称为指令助记符,简称助记符,助记符一般是表明指令功能的英语单词或其缩写。指令操作数同样也可以用易于记忆的符号表示。

用助记符表示的指令就是汇编格式指令。汇编格式指令以及使用它们编写程序的规则就形成汇编语言(Assembly Language) 。用汇编语言编写的程序就是汇编语言程序,或称汇编语言源程序。汇编语言是一种符号语言,它用助记符表示操作码,比机器语言容易理解和掌握,编写的程序也容易调试和维护。但是,汇编语言源程序要翻译成机器语言程序才可以由处理器执行。这个翻译的过程称为“汇编”,完成汇编工作的程序就是汇编程序(Assembler) 。

1.6.3　高级语言

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

高级语言简单、易学,而汇编语言复杂、难懂,那么是否就没有必要再采用汇编语言了

呢? 首先比较一下汇编语言和高级语言的特点。汇编语言与处理器密切相关,每种处理器都有自己的指令系统,相应的汇编语言各不相同。所以,汇编语言程序的通用性、可移植性较差。相对来说,高级语言与具体计算机无关,高级语言程序可以在多种计算机上编译执行。汇编语言功能有限,又涉及寄存器、主存单元等硬件细节,所以编写程序比较烦琐,调试起来也比较困难。高级语言提供了强大的功能,采取类似自然语言的语法,所以容易被掌握和应用,也不必关心诸如标志、堆栈等琐碎问题。汇编语言本质上就是机器语言,它可直接地、有效地控制计算机硬件,因而容易产生运行速度快、指令序列小的高效率目标程序。高级语言不易直接控制计算机的各种操作,编译程序产生的目标程序往往比较庞大,程序难以优化,所以运行速度较慢。

可见汇编语言的主要优点就是可以直接控制计算机硬件,可以编写在“ 时间 ”和“ 空间 ”两方面最有效的程序。这些优点使得汇编语言在程序设计中占有重要的位置,是不可取代的。汇编语言的特点也是明显的,它与处理器密切相关,要求程序员比较熟悉计算机硬件系统,需要考虑许多细节问题,导致编写程序烦琐,调试、维护、交流和移植困难。因此,有时可以采用高级语言和汇编语言混合的办法,互相取长补短,更好地解决实际问题。汇编语言的主要应用场合如下。

① 程序要求具有较快的执行时间,或者只能占用较小的存储容量。例如,操作系统的核心程序段、实时控制系统的软件、智能仪器仪表的控制程序等。

② 程序与计算机硬件密切相关,要直接、有效地控制硬件。例如,I/O 接口电路的初始化程序段、外部设备的低层驱动程序等。

③ 大型软件需要提高性能、优化处理的部分。例如,计算机系统频繁调用的子程序、动态连接库等。

④ 没有合适的高级语言或只能采用汇编语言的时候。例如,开发最新的处理器程序时,暂时没有支持新指令的编译程序。

⑤ 汇编语言还有许多实际应用,例如分析具体系统尤其是该系统的低层软件、加密解密软件、分析和防治计算机病毒等。

习　　题

1. 简述计算机和微型计算机经历了哪些主要发展阶段,它们的发展是以什么为主要技术特征的?
2. 微型计算机与其他大、中、小型计算机有什么共性和区别?
3. 试述微型计算机的主要应用领域。
4. 什么叫微处理器,什么叫微型计算机,什么叫微型计算机系统?
5. 什么叫单片机,什么叫单板机?
6. 衡量微机系统性能的主要指标有哪些?
7. 简述补码运算中判断溢出的方法。
8. 什么叫机器数,什么叫真值?
9. 什么叫指令和指令系统?
10. 指令由几部分组成,它们各起什么作用?
11. 在汇编语言中,如何表示二进制、八进制、十进制和十六进制的数值?
12. 在计算机中,如何表示正、负数,在保持数值大小不变的情况下,如何把位数少的二进制数值扩展

成位数较多的二进制数值?

13. 已知 X = - 74, Y = - 85, 试用补码完成下列运算, 并判断有无溢出产生(设字长为 8 位)。

(1) X + Y　　(2) X - Y　　(3) - X + Y　　(4) - X - Y

14. 简述汇编程序、连接程序和汇编语言源程序的概念。

第 2 章　微型计算机系统组成

微型计算机系统硬件由微处理器、存储器、I/O 接口等部分组成，了解它们的内部结构和工作方式将有助于学习后续的内容。汇编语言以助记符的形式表示计算机指令，每条指令对应着计算机硬件的一个具体操作，利用汇编语言编写的程序与计算机硬件密切相关，程序员可直接对处理器内的寄存器、主存储器的存储单元以及外设的端口等进行操作，从而能够有效地控制计算机各个部件。因此，要学好汇编语言程序设计，必须对微型计算机的硬件部分有所了解。

本章重点讨论 Intel 8086 芯片的内部结构和外部引脚，对其内部寄存器组的设置与功能进行了较详细的说明；介绍了存储器系统的基本组成、I/O 接口的基本工作方式以及微处理器芯片的发展。

2.1　微型计算机系统硬件结构

2.1.1　结构特点与框图

目前各种微型计算机系统，无论是简单的单片机、还是较复杂的个人计算机系统，以至超级微机和巨型微机系统，从硬件体系结构来看，采用的基本上是计算机的经典结构—冯·诺依曼结构。这种结构的特点如下。

由运算器、控制器、存储器、输入设备和输出设备 5 大部分组成。

数据和程序以二进制代码形式不加区别地存放在存储器中，存放位置由地址指定，地址码也为二进制。

控制器是根据存放在存储器中的指令序列即程序来工作的，并由一个程序计数器控制命令的执行。控制器具有判断能力，可以计算结果为基础，选择不同的工作流程。

由此可见，任何一个微型机系统都是由硬件和软件组成。硬件又由运算器、控制器、存储器、输入设备和输出设备 5 部分组成。图 2 - 1 给出了具有这种结构特点的微型计算机典型硬件组成框图。各组成部分之间通过地址总线 AB、数据总线 DB、控制总线 CB 联系在一

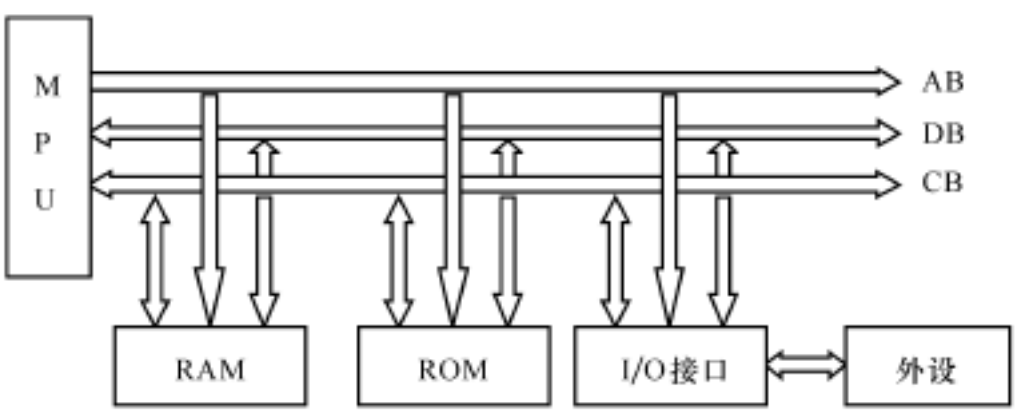


图 2 - 1　微型计算机的结构框图

起。采用总线结构,可使微型计算机的系统结构比较简洁,并且具有更大的灵活性和更好的可扩展性、可维修性。

2.1.2　主要组成部分及功能

1. 微处理器(MPU)

微处理器是微型计算机的运算和指挥中心。不同型号的微型计算机,其性能的差别首先在于其微处理器性能的不同。而微处理器性能又与它的内部结构、硬件配置有关。每种微处理器都有其特有的指令系统,但无论哪种微处理器,其内部结构总是相同的,都有控制器、运算器、内部总线及缓冲器 4 大部分,每部分又各由一些基本部件组成,如图 2 - 2 所示。

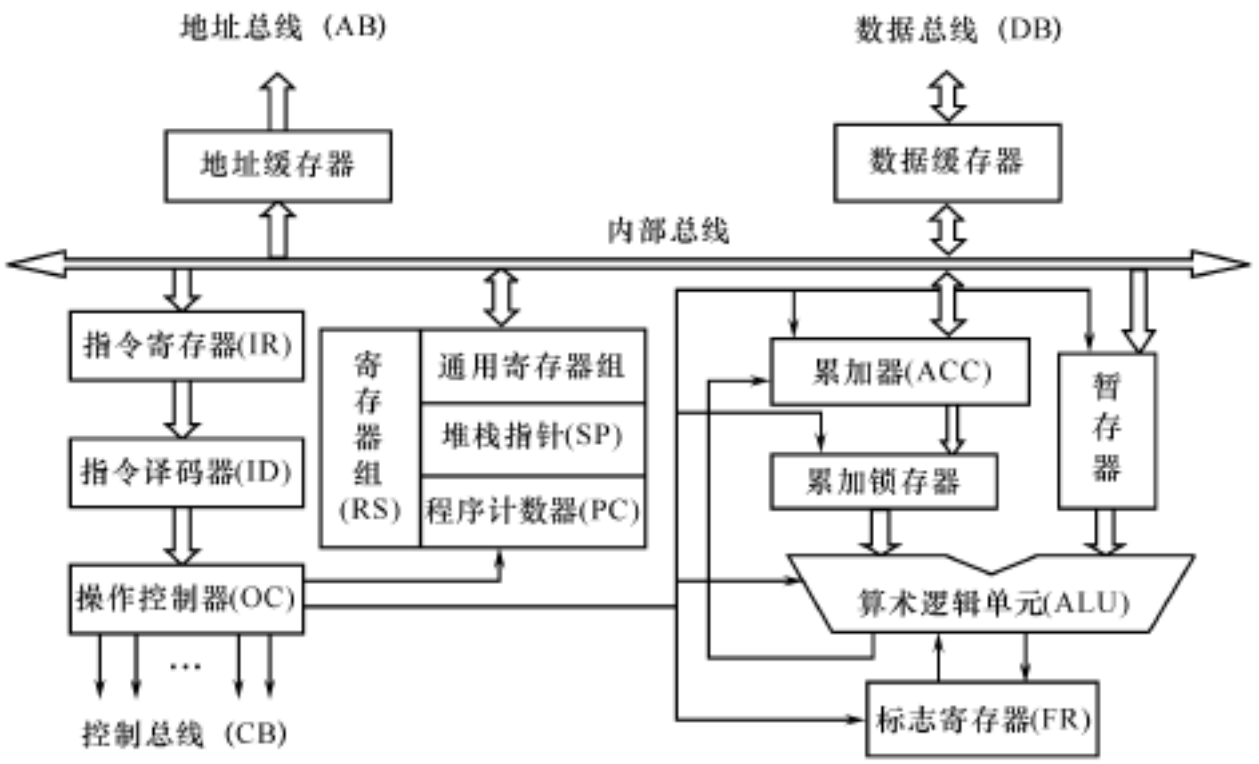


图 2 - 2　微处理器典型结构示意图

主要部件的功能如下。

(1) 算术逻辑单元 ALU(Arithmatic Logic Unit)

ALU 是运算器的核心,它是以全加器为基础,辅之以移位寄存器及相应控制逻辑组合而成的电路,在控制信号的作用下可完成加、减、乘、除四则运算和各种逻辑运算。

(2) 累加器 ACC

累加器 ACC(Accumulator)通常简称为累加器 A,它实际上是通用寄存器中的一个。

(3) 标志寄存器 FR(Flags Register)

FR 用于寄存 ALU 操作结果的某些重要状态或特征,如是否溢出,是否为零,是否为负,是否有进位,是否有偶数个“1”等。

(4) 寄存器组 RS(Register Set 或 Registers)

RS 实质上是微处理器的内部 RAM,因受芯片面积和集成度所限,其容量(即寄存器数目)不可能很多。寄存器组可分为专用寄存器和通用寄存器,其中专用寄存器的作用是固定的。图 2 - 2 中的堆栈指针 SP、程序计数器 PC、标志寄存器 FR 即为专用寄存器。通用寄存器可由程序员规定其用途。

(5) 堆栈指针 SP(Stack Pointer)

堆栈(Stack) 是一组寄存器或存储器开辟的一个特定区域。由内部寄存器组构成的堆栈叫硬件堆栈; 由软件开辟的存储器区域叫软件堆栈。目前, 绝大多数微型计算机中都是采用软件堆栈。堆栈在计算机中是作为一种数据结构或数据暂存方式引入的。

堆栈指针 SP 就是用来指示栈顶的寄存器。SP 的初值由程序员设定, 一旦设定初值后, 便意味着栈底在内存储器中的位置已经确定, 此后 SP 的内容即栈顶位置便由 CPU 自动管理。一般来说, 对于栈底地址高、栈顶地址低的向下增长型堆栈, 将新数据压入其中时, 每压入一个字节, SP 自动减 1, 向上浮动而指向新的栈顶; 当数据从栈中弹出时, 每弹出一个字节, SP 自动加 1, 向下浮动而指向新的栈顶。反之, 对于栈底地址低、栈顶地址高的向上增长型堆栈, 随着数据的压入或弹出, 指针 PC 的浮动方向则正好相反。堆栈主要用于中断处理与过程(子程序) 调用 /返回(特别是多重中断与多重调用) 。

(6) 程序计数器 PC(Porgram Counter)

PC 用于存放下一条要执行的指令的地址码, 即程序中的各条指令所在的地址编号。

(7) 指令寄存器 IR(Instruction Register) 、指令译码器 ID(Instruction Decoder) 和操作控制器 OC(Operation Controller)

这 3 个部件是整个微处理器的指挥控制中心, 对协调整个微型计算机有序工作极为重要。MPU 根据用户预先编好的程序, 依次从存储器中取出各条指令, 放在指令寄存器 IR 中, 通过指令译码(分析) 确定应该进行什么操作, 然后通过操作控制器 OC, 按确定的时序, 向相应的部件发出控制信号。操作控制器 OC 中主要包括节拍脉冲发生器、控制矩阵、时钟脉冲发生器、复位电路和启停电路等控制逻辑。

2. 存储器

存储器又叫内存或主存, 是微型计算机的存储和记忆部件, 用以存放数据(包括原始数据、中间结果和最终结果) 和程序。微型计算机的内存都是采用半导体存储器。

(1) 内存单元的地址和内容

内存中存放的数据和程序, 从形式上看都是二进制数。内存是由若干个内存单元组成的, 每一个内存单元中可存放一个字节(8 位) 的二进制信息。内存单元的总数叫内存容量。微型计算机通过给各个内存单元规定不同地址来管理内存。这样, CPU 便能识别不同的内存单元, 正确地对它们进行操作。注意, 内存单元的地址和内存单元的内容是两个完全不同的概念。图 2 - 3 给出了这两个概念的示意图。

地址	内容
00000H	10110010
00001H	11000011
00002H	00001001
F0000H	00111110
FFFFFH	00000000

图 2 - 3　内存单元地址和内容

(2) 内存操作

CPU 对内存的操作有读、写两种。读操作将内存单元的内容送入 CPU 内部, 而写操作是将 CPU 内部信息传送到内存单元保存起来。显然, 写操作的结果改变了被写单元的内容, 而读操作则不改变被读单元中原有的内容。

(3) 内存分类

按工作方式的不同,内存可分为随机存取存储器 RAM(Random Access Memory) 和只读存储器 ROM(Read - Only Memory) 两大类。RAM 可以被 CPU 随机地读和写,所以又称为读写存储器。这种存储器用于存放用户装入的程序、数据及部分系统信息。当机器断电后,所存信息消失。ROM 中的信息只能被 CPU 随机读取,而不能由 CPU 随机写入。

3. 输入/输出(I/O) 接口

I/O 接口是微型计算机系统的重要组成部分,微型计算机通过它与外部交换信息,完成实际工作任务。

4. 三总线

总线实际上是一组导线,是各种公共信号线的集合,用于微型计算机中所有各组成部分传输信息。

(1) 数据总线 DB(Data Bus)

数据总线用来传输数据信息,是双向总线,CPU 既可通过 DB 从内存或输入设备读入数据,又可通过 DB 将内部数据送往内存或输出设备。

(2) 地址总线 AB(Address Bus)

地址总线用于传送 CPU 发出的地址信息,是单向总线。目的是指明与 CPU 交换信息的内存单元或 I/O 设备。

(3) 控制总线 CB(Control Bus)

控制总线用来传送控制信号、时序信号和状态信息等。其中有的是 CPU 向内存和外设发出的信号,有的则是内存或外设向 CPU 发出的信息。可见,CB 中每一根线的方向是一定的、单向的,但作为一个整体则是双向的,所以在各种结构框图中,凡涉及控制总线 CB,均以双向线表示。

2.2　8086/8088 微处理器

2.2.1　内部结构

8086 是全 16 位微处理器,内、外数据总线都为 16 位。8088 是准 16 位微处理器,内数据总线为 16 位,外数据总线为 8 位。

1. 内部结构

8086/8088 微处理器从功能上可分为两个独立的处理单元: 执行单元 EU(Execution Unit) 和总线接口单元 BIU(Bus Interface Unit) ,其内部结构如图 2 - 4 所示。

执行单元 EU 由 8 个 16 位的通用寄存器、1 个 16 位的标志寄存器、1 个 16 位的暂存寄存器、1 个 16 位的算术逻辑单元 ALU 及 EU 控制电路组成。总线接口单元 BIU 由 4 个 16 位的段寄存器、1 个 16 位的指令指针寄存器、1 个与 EU 通信的内部暂存器、1 个指令队列、1 个计算 20 位物理地址加法器∑及总线控制电路组成。其内部寄存器组如图 2 - 5 所示。

(1) 通用寄存器

8 个通用寄存器中, AX, BX, CX, DX 为数据寄存器,用于存放参与运算的数据或运算的结果,它们中的每一个既可以作为一个 16 位寄存器使用,又可以将高、低 8 位分别作为两个

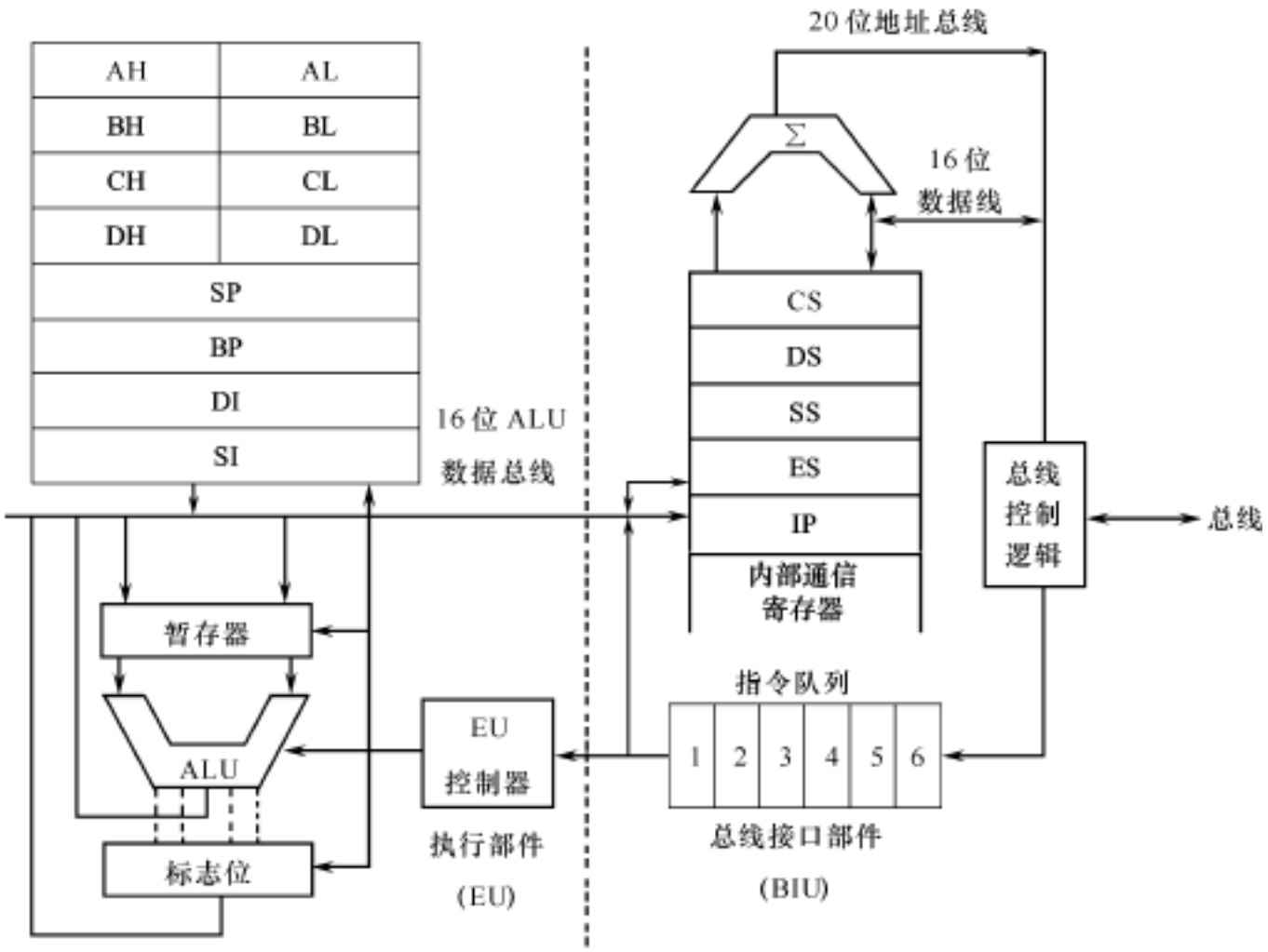


图 2 - 4　8086/8088 微处理器内部结构如图

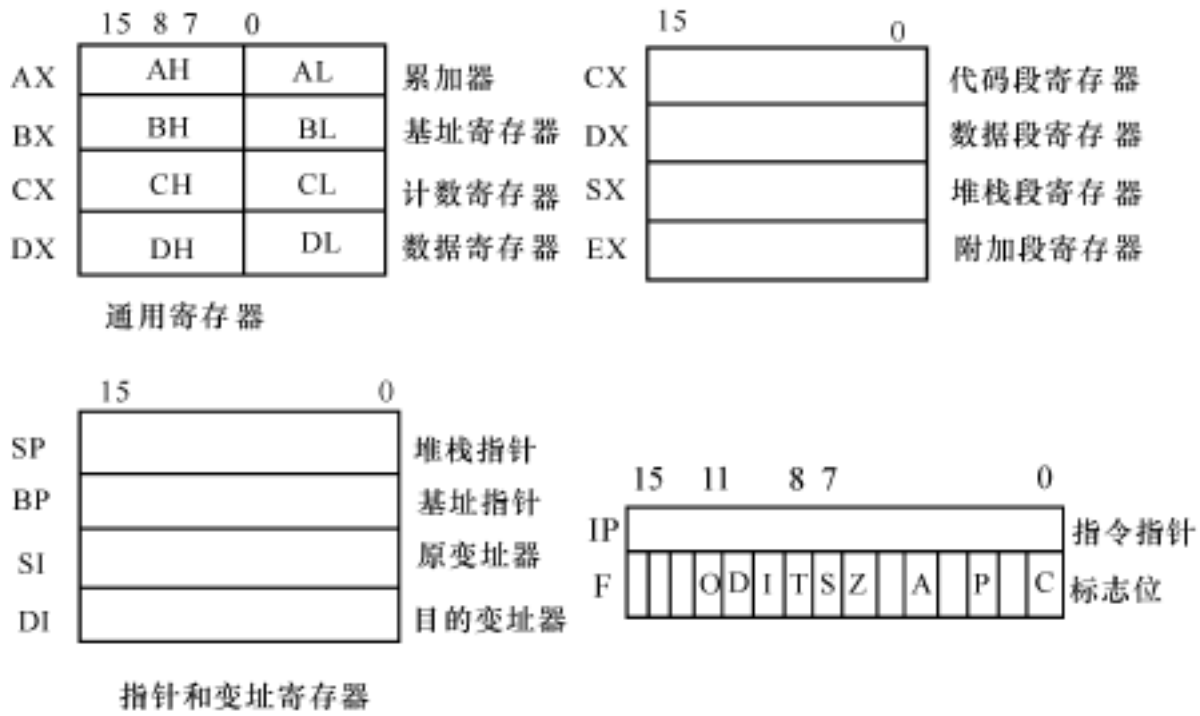


图 2 - 5　8086/8088CPU 内部寄器

独立的 8 位寄存器使用。作为 8 位寄存器时，它们的名称分别为 AH, AL, BH, BL, CH, CL, DH, DL。**指针寄存器** SP **和** BP **分别为堆栈指针寄存器和基址指针寄存器，作为通用寄存器的一种，它们既可以存放数据，也可以存放内存单元的偏移地址。而** SI **和** DI **则主要用于变址寻址方式的源变址和目的变址。**

(2) 段寄存器

4 个段寄存器分别用于存放代码段(CS)、数据段(DS)、堆栈段(SS)、和附加段(ES)的段基址的高 16 位。地址加法器∑用于将段基址与偏移量按一定的规则相加,形成系统所需的 20 位物理地址,它的输出直接送往地址总线。

物理地址 = 段基址× 10H + 偏移量

(3) 指令指针寄存器 IP

指令指针寄存器 IP 类似于 8080 和 8085 中的程序计数器 PC,不同的是,8080 和 8085 中的 PC 总是指向下一条即将执行的指令的地址,而 8086 中的 IP 则总是指向下一条待取指令在现行代码段中相对于代码段基址的偏移量。

(4) 标志寄存器 F

16 位标志寄存器 F 中包含两种标志,状态标志(6 位)和控制标志(3 位)。

状态标志包含以下几种。

进位标志 CF(位 0):当运算结果的最高位产生了进位或借位时,CF = 1;否则,CF = 0。

偶标志 PF(位 2):若运算结果中"1"的个数为偶数,则 PF = 1;否则,PF = 0。

辅助进位标志 AF(位 4):若运算结果导致低 4 位向前位有进位或借位,则 AF = 1;否则,AF = 0。

零标志 ZF(位 6):若运算的所有位全为 0,则 ZF = 1;否则,ZF = 0。

符号标志 SF(位 7):SF 的值与运算结果的最高位相同。即结果的最高位为 1,SF = 1;否则,SF = 0。对于用补码表示的符号数,SF = 1 表示结果为负数。

溢出标志 OF(位 11):OF 反映了有符号数的运算是否产生了溢出。OF = 1 表示运算结果的数值超过了可表示的范围。

控制标志包含以下几种。

自陷标志 TF(位 8):TF = 1 表示 CPU 将进入单步执行方式,即 CPU 每执行一条指令后,自动产生一个内部中断。

中断允许标志 IF(位 9):IF = 1 表示允许 CPU 去接外部的可屏蔽中断请求,否则,将屏蔽外部可屏蔽中断。

方向标志(位 10):DF = 1 表示在串操作指令执行期间,地址指针 EDI(DI)和 ESI(SI)的变化方向为减量,否则为增量。

2.2.2　引脚及功能

8086 CPU 芯片是双列直插式结构,共有 40 个引脚(如图 2 - 6 所示),主要包括以下几部分。

1. 地址/状态线

A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3(输出、三态)为地址/状态线,均是多路开关输出。一般 CPU 对存储器或 I/O 端口完成一次读写操作需要 4 个时钟周期,即 T_1, T_2, T_3, T_4,这些线只是在存储器操作时的 T1 状态作为最高 4 位地址线,其余情况大都用作状态线或不用。作为状态线,S_6 始终为 0,表示 CPU 当前连在总线上;S_5 是中断允许标志的状态,S_5 = 1 表示当前允许可屏蔽中断请求,S_5 = 0 则禁止一切可屏蔽中断。S_4 和 S_3 编码指示哪一个段寄存器正在被使用,00(ES),01(SS),10(CS),11(DS) ES 为附加数据段寄存器,SS 为堆栈段寄存器,

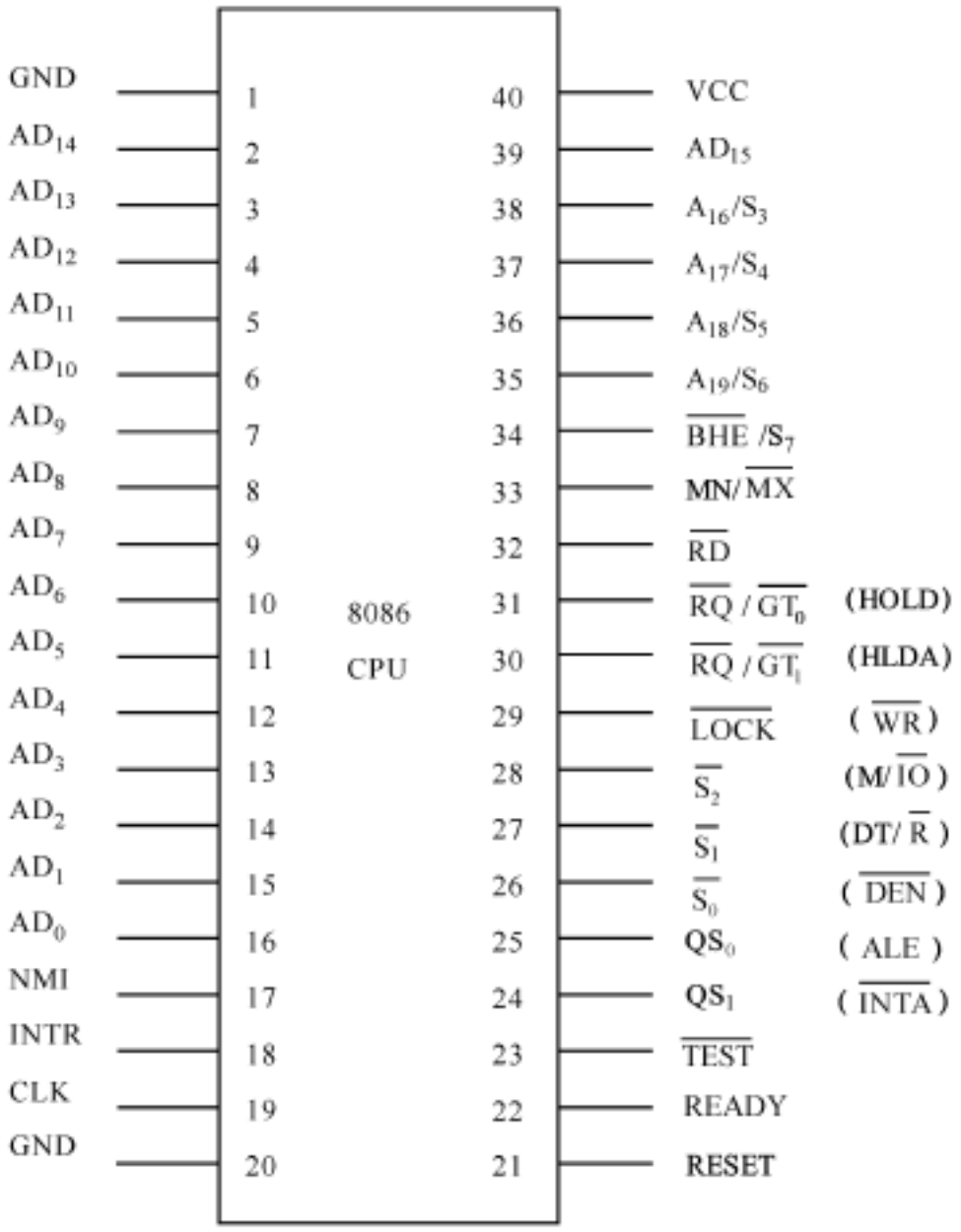


图 2 - 6　8086 CPU 引脚图

CS为代码段寄存器, DS 为数据段寄存器。在直接存储器存取式控制 DMA(Direct Memory - Access) 方式时, 这些引脚悬空。

2. 地址/数据线

AD_{15}～AD_0(输入/输出、三态) 为地址/数据线, 这些引线用于构成分时复用的存储器、I/O地址总线和数据总线。通常, CPU 访问存储器或外设时, 先要给出所访问单元(或端口) 地址, 然后才给出读写所需的数据。在 DMA 方式时, 这些引脚悬空。显然, 8086 CPU 的地址总线为 20 根(A_{19}～A_0) , 可访问的物理地址空间为 2^{20} = 1 MB, 地址范围为 00000H ～FFFFFH; 其数据总线为 16 根(D_{15}～D_0) , 占据两个字节宽度。

3. 公用控制引线

(1) $\overline{BHE}/S_7$(输出、三态)

在 T_1 状态时, 作为总线高半部分的 D_{15}～D_8 允许信号, 低电平有效。S_7 是备用状态信号, 在 T_2 ～T_4 状态时输出。在 DMA 方式, 此线悬空。

(2) $\overline{RD}$(输出、三态)

读选通信号, 低电平有效, 与 $M/\overline{IO}$信号配合实现对存储器或 I/O 读操作, DMA 时悬空。

(3) READY(输入)

准备就绪信号,高电平有效,表示将完成数据传送,否则在 T_3 周期结束后插入一个或若干个等待周期 T_W,直至 READY 有效,进入 T_4 周期,完成数据传送。

(4) INTR(输入)

可屏蔽中断请求信号。

(5) NMI(输入)

非屏蔽中断请求信号。

(6) $\overline{\text{TEST}}$(输入)

当 CPU 在执行 WAIT 指令时,它每隔 5 个时钟周期对$\overline{\text{TEST}}$进行一次测试。若$\overline{\text{TEST}}$ = 1, CPU 进入等待状态; 若$\overline{\text{TEST}}$ = 0, CPU 继续往下执行被暂停的指令。

(7) RESET(输入)

复位输入,该信号必须保持高电平有效至少 4 个时钟周期,以完成内部的复位过程,并对 FLAG, IP, DS, SS, ES 及指令队列清零,而将 CS 设置为 FFFFH。当变为低电平时, CPU 从 FFFF0H 单元开始执行程序。

(8) CLK(输入)

时钟输入信号(主频),提供处理器和总线控制器的定时操作。

(9) VCC 和 GND

+5V 的电源引脚和地线。

4. 由 MN/$\overline{\text{MX}}$定义的引线信号

MN/$\overline{\text{MX}}$引脚为最小工作模式和最大工作模式的选择控制端,如果此引脚固定接为 +5V,则 CPU 处于最小工作模式;如果接地,则 CPU 处于最大工作模式。8086 CPU 第 24 ～31 引脚在最小工作模式和最大工作模式下有不同的名称和定义。

系统在最小工作模式下引脚 24 ～31 功能如下。

(1) M/$\overline{\text{IO}}$(输出、三态)

若此引线为高电平,访问存储器;若此引线为低电平,访问 I/O。在 DMA 方式时,此线悬空。

(2) $\overline{\text{WR}}$(输出)

低电平有效,表示 CPU 是处在存储器写或 I/O 写(取决于 M/$\overline{\text{IO}}$)状态。到底为哪种写操作,则由 M/$\overline{\text{IO}}$信号决定。在 DMA 方式时,此线悬空。

(3) $\overline{\text{INTA}}$(输出、三态)

中断响应信号,低电平有效。它在每一个中断响应周期有效,可用作读外设端口送来的中断类型码的选通信号。

(4) ALE(输出)

地址锁存允许信号,高电平有效。这是一个由处理器提供的,把在地址/数据线(AD_0 ～AD_{15})上出现的地址信号锁存到接口芯片 8282/8283 的地址锁存器中时所用的触发信号。

(5) DT/$\overline{\text{R}}$(输出、三态)

数据发送/接收信号,在最小工作模式时,若为了增加数据总线的驱动能力而用总线发送/接收接口芯片 8286/8287 时,需要这个信号确定数据传送的方向。此线为高电平时,为数据发送;此线为低电平时,为数据接收。在 DMA 方式时,此线悬空。

(6) $\overline{DEN}$(输出、三态)

数据允许信号,低电平有效。在使用接口芯片 8286/8287 的最小工作模式系统中,此信号作为 8286/8287 芯片的输出允许信号。它在每一次存储访问或 I/O 访问或中断响应周期有效。在 DMA 方式时,此线悬空。

(7) HOLD(输入)、HLDA(输出)

HOLD 是当其他主设备(如 DMA 方式)要求占用总线时,向 CPU 发出的总线请求信号,高电平有效。

系统在最大工作模式下 8086 的引脚 24～31 的含义如下。

(1) $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$(输出、三态)

这些总线周期状态信号线的编码及 8288 产生的控制信号如表 2 - 1 所示。这些信号组合起来可以指出当前总线周期中所进行的数据传输过程的类型。最大工作方式系统中的总线控制器 8288 就是利用这些信号来产生对存储器和 I/O 的控制信号,这在 DMA 方式时处在悬空状态。

表 2 - 1　$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$的组合所对应的操作及 8288 产生的控制信号

S_2	S_1	S_0	对应的操作	8288 产生的控制信号
0	0	0	发中断响应信号	$\overline{INTA}$
0	0	1	读 I/O 端口	$\overline{IORC}$
0	1	0	写 I/O 端口	$\overline{IOWC}$和$\overline{AOWC}$
0	1	1	暂停	无
1	0	0	取指令	$\overline{WROC}$
1	0	1	读内存	$\overline{WROC}$
1	1	0	写内存	$\overline{MWTC}$和$\overline{AMWC}$
1	1	1	无效	无

(2) $\overline{RQ}/\overline{GT0}$, $\overline{RQ}/\overline{GT1}$(输入/输出)

这两个信号端可供 CPU 以外的处理器用来发出使用总线的请求信号和接收 CPU 对总线请求信号的回答信号。$\overline{RQ}/\overline{GT0}$和$\overline{RQ}/\overline{GT1}$都是双向的,总线请求信号和允许信号在同一引脚上传输,但方向相反。其中$\overline{RQ}/\overline{GT0}$比$\overline{RQ}/\overline{GT1}$的优先级要高。

(3) $\overline{LOCK}$(输出、三态)

低电平有效,当$\overline{LOCK}$有效时,其他总线主设备不能对系统总线进行控制。$\overline{LOCK}$信号由指令前缀 LOCK 使其有效,且在下一个指令完成以前保持有效。在 DMA 方式时,此线处于悬空状态。

(4) QS_1 和 QS_0(输出)

QS_0 和 QS_0 这两个信号组合起来,提供了前一个时钟周期(指总线周期的前一个状态)中指令序列的状态,以便于外部对 CPU 内部指令队列的动作进行跟踪,其编码含义如表 2 - 2所示。

表 2 - 2　队列状态位的编码

QS_1	QS_0	队 列 状 态
0	0	无操作, 队列中的指令没有被取走
0	1	指令的第一字节, 从队列中取出
1	0	队列已空, 在执行一条传送指令时, 队列被重新初始化
1	1	除第一字节外, 还取走了后续字节中的代码

2.3　存储器组成

存储器是计算机的基本组成部分, 用于存储计算机工作中所必需的数据和程序。处理器实际运行时的大部分总线周期都是对存储器的读/写进行访问。所以存储器性能的好坏在很大程度上影响着计算机系统的性能。

存储器系统设计的首要目的是使存储器在工作速度上很好地与处理器匹配, 并满足各种存取的需要。在早期的 8 位微型计算机中, 微处理器的读写速度不高, 实现与存储器的匹配没有很大的矛盾。随着微电子技术的发展, 微处理器的工作速度有很大的提高。在实际微型计算机系统中, 总是采用分级的方法来设计整个存储器系统, 如图 2 - 7 所示。

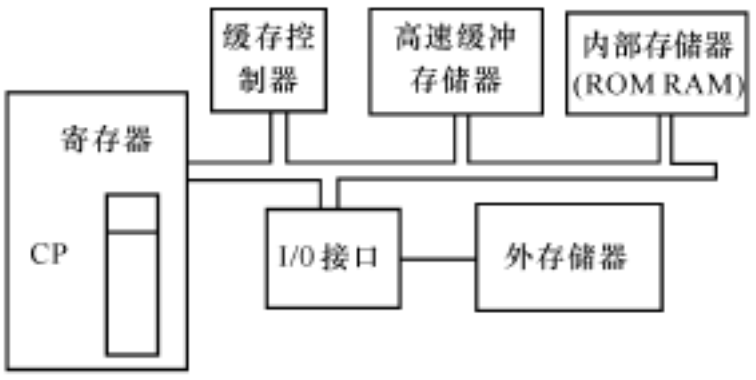


图 2 - 7　分级存储器系统

寄存器组是最高一级的存储器。在微型计算机中, 寄存器组一般是微处理器内含的。有些待处理的数据, 或者运算的中间结果可以暂存在这些寄存器中, 微处理器在对本芯片内的寄存器进行读写时, 其速度很快, 一般在一个时钟周期中完成。

第二级存储器是高速缓冲存储器(Cache) 。这一级存储器中一般只装载当前用得最多的程序或数据, 使微处理器能以自己最高的速度工作。设置高速缓冲存储器是高档微型计算机中很常用的一种方法, 目前一般也将它们或它们的一部分制作在 CPU 芯片中(如 80486 DX, 80486 DX2 和 Pentium 等) 。

第三级存储器是内存储器, 运行的程序和数据都放在其中。由于微处理器的寻址大部分落在高速缓冲存储器上, 因此内存就可以采用速度稍慢的存储器芯片, 对系统性能的影响不会太大。

最低一级存储器是大容量的外存, 如磁带 、软盘 、硬盘 、光盘等, 这种外存容量可达几百兆至成千上百万兆字节, 在存取速度上比内存要慢得多。如果加上寻找数据位置所需的时间, 速度更慢。

2.4 系 统 总 线

所谓总线,就是在模块与模块之间、设备与设备之间传送信息的一组共用信号线,是系统在主控器(模块或设备)的控制下,将发送器(模块或设备)发送的信息准确地传送给某个接收器(模块或设备)的信号通路。

总线按其信号线性质不同一般可分为 3 组,即数据总线 DB、地址总线 AB、控制总线 CB。总线还有另一种分法,即基本信息总线、判决总线、数据握手总线。基本信息总线包括数据总线、地址总线和存储器与 I/O 读写控制线;判决总线则包括总线判决、中断和 DMA 判决几类控制线;数据握手总线用于启动和停止总线操作,控制每个操作周期中数据传送的开始和结束。

总线作为微型计算机系统的组成基础和重要资源,各微型计算机生产公司从一开始就十分重视它的设计并制定了自己的总线标准,以支持本公司的产品设计和开发。

采用通用总线标准,可以为不同模块、设备的互连提供一个标准的界面。这个界面对两端的模块和设备来说都是透明的,界面的任一边只需根据总线标准的要求来实现接口的功能,而不必考虑另一边的接口方式。这就为接口的硬件、软件设计提供了方便,并且使设计出的接口也具有通用性。总线按其在系统中的位置及功能不同,一般分为 3 级。

① 芯片级总线:利用它把 CPU 芯片与其他芯片连成微处理机模块(主板)。芯片级总线因 CPU 不同而异,所以实质上就是 CPU 总线。

② 模块级总线:利用它把主板和主板上各模块连成微型计算机。模块级总线是微型计算机内部的总线,所以也叫微型计算机总线或内总线。

③ 系统级总线:利用它把多台微型计算机或设备连成微型计算机系统,所以也叫外总线。通常系统级总线又分为并行总线和串行总线两种。

PC 系列微型计算机系统中广泛使用的系统级总线标准,主要有以下几种。

① Centronics 总线标准:它是一种早期得到工业界大力支持的并行总线接口标准,主要用于打印机接口。

② IEEE-488 总线标准:它是由美国 HP 公司于 20 世纪 70 年代初首先提出的接口标准,以后相继为 IEEE(Institute of Electrical & Electronic Engineers)和 IEC(International Electrotechnical Commission)所承认,成为国际上最流行的并行总线标准,主要适用于各种中小规模仪器系统。

③ CAMAC 总线标准:它是国际上最早推出的比较有影响的通用仪器接口标准,主要用在原子能和核物理研究领域,以及一些规模较大的计算机自动测量与控制系统中。

④ ATA/IDE 总线标准:这是 IDE(Integrated Drive Electronics)类硬盘的一种 AT 嵌入式(AT Attachment)接口标准,其接口信号是 AT 总线的子集。

⑤ SCSI 总线标准:这是一种小型计算机系统接口标准,支持高速智能并行 I/O 接口。利用它可以混接各种磁盘、光盘、磁带机、打印机、扫描仪、条码阅读器以及通信设备等。

⑥ RS-232-C 总线标准:它是美国 EIA 推荐的串行通信总线标准,应用极广。目前,每种微型计算机都配有这种总线接口,用于与 CRT 显示终端、鼠标器、Modem 等串行外设连接。

⑦ USB 总线标准: 这是由 Intel, Compag 等 7 家公司联合制定, 于 1996 年公开推出的新型通用串行总线标准。它主要用于低速设备(如键盘、鼠标器等) 和中速设备(如扫描仪、Modem 和数字相机等) 与 PC 的连接, 具有支持即插即用和热插拔的功能, 目前已在高档 Pentium PC 中普遍采用。

⑧ IEEE - 1394 总线标准: 这是一种最新的高性能串行总线标准。相当于 USB 标准的高速版本, 除具有 USB 的各种特点和优越性外, 主要在数据传送的高速性、实时性方面进行了改进, 使其数据传输速率可达 100 Mbps ~400 Mbps, 为 USB 最高速率的 8 倍以上。

⑨ 各种总线标准之间尽管在设计细节上有很多不同, 且各有特点。但从总体原则上看, 每种总线设计所要解决的问题是大体相同的, 如信号分类、传输应答同步控制和资源共享分配等。

2.5　输入/输出接口

2.5.1　I/O 接口概述

为了完成一定的实际任务, 微型计算机都必须与各种外部设备相联系, 与它们交换信息。微型计算机与外部设备间交换的信息通常有 4 种类型。

数字量信息: 一种二进制的数据或经过编码的二进制数据, 最小单位为“ 位 ”(bit) , 8 位称为一个字节。

模拟量信息: 用模拟电压或模拟电流的幅值大小表示的物理量。

开关量信息: 只有“ 开 ”或“ 关 ”两种状态的信息。

脉冲量信息: 以脉冲形式表示的信息。

接口是处理器与外部设备连接的通道, 从而实现主机与外设之间的数据交换。处理器通过总线与接口电路连接, 接口电路与外部设备连接, 接口技术是组成任何实用计算机系统的关键技术, 任何微型计算机应用开发工作都离不开接口的设计、选用和连接。因此, 微型计算机接口技术是一种综合软件、硬件设计完成某一特定任务的技术。

2.5.2　I/O 端口的编址方式

微处理器与指定外设间的信息交换是通过访问与该外设相对应的端口来实现的, 如何实现对相关端口的访问, 取决于这些端口的编址方式。通常有两种编址方式, 存储器映像方式和隔离 I/O 方式。

1. 存储器映像方式

这种编址方式是将 I/O 端口和存储器单元一起编址, 相当于给每个 I/O 端口分配一个存储器地址。

存储器映像式编址的主要优点。

① 对 I/O 接口的操作与对存储器的操作完全相同, 任何存储器操作指令都可用来操作 I/O接口, 而不必使用专用的 I/O 指令。

② 可以使外设数目或 I/O 寄存器数目除只受总存储容量的限制外几乎不受限制, 从而大大提高了系统的信息处理能力。

③ 使微型计算机系统的读写控制逻辑较为简单。

存储器映像方式的主要缺点。

① 占用了存储器的一部分地址空间,使可用的内存空间减少。

② 为了识别一个 I/O 端口,必须对全部地址线译码,这样不但增加了地址译码电路的复杂性,而且使执行外设寻址的操作时间相对增长。

2. 隔离 I/O 方式

这种编址方式是对 I/O 端口和存储器进行不同的处理,分开编址,即两者的地址空间是互相“隔离”的,I/O 结构不会影响存储器的地址空间。

这种编址方式的优点如下。

① I/O 端口地址不占用存储器地址空间,或者说存储器全部地址空间都不受 I/O 寻址的影响。

② 由于 I/O 地址线较少,所以 I/O 端口地址译码较简单,寻址速度较快。

③ 使用专用 I/O 指令和真正的存储器访问指令有明显区别,可使程序编制得清晰,便于理解和检查。

这种方式的缺点如下。

① 专用 I/O 指令类型少,远不如存储器访问指令丰富,使程序设计灵活性差。使用 I/O 指令只能在累加器 A 和 I/O 端口间交换信息,处理能力不如存储器映像方式强。

② 要求处理器能提供存储器读/写、I/O 端口读/写两组控制信号,这不仅增加了控制逻辑的复杂性,而且对于引脚线本来就紧张的 MPU 芯片来说,不能不说是一个负担。

2.5.3　I/O 同步控制方式

I/O 同步控制方式是微型计算机基本系统与 I/O 外设之间数据传送的管理方法,是微型计算机系统的一种调度策略。I/O 同步控制的目的就是要实现 CPU 与 I/O 设备之间操作的同步,以实现两者之间正确有效地传送数据。I/O 外设与存储器不同,其数据传送速度比 MPU 相差很多,而且各种外设之间的数据传送速度也相差悬殊,加上 I/O 工作过程定时完全地独立于微处理器,因此它们之间的数据传送必须通过接口中的数据缓冲寄存器来进行,如图 2 - 8 所示。

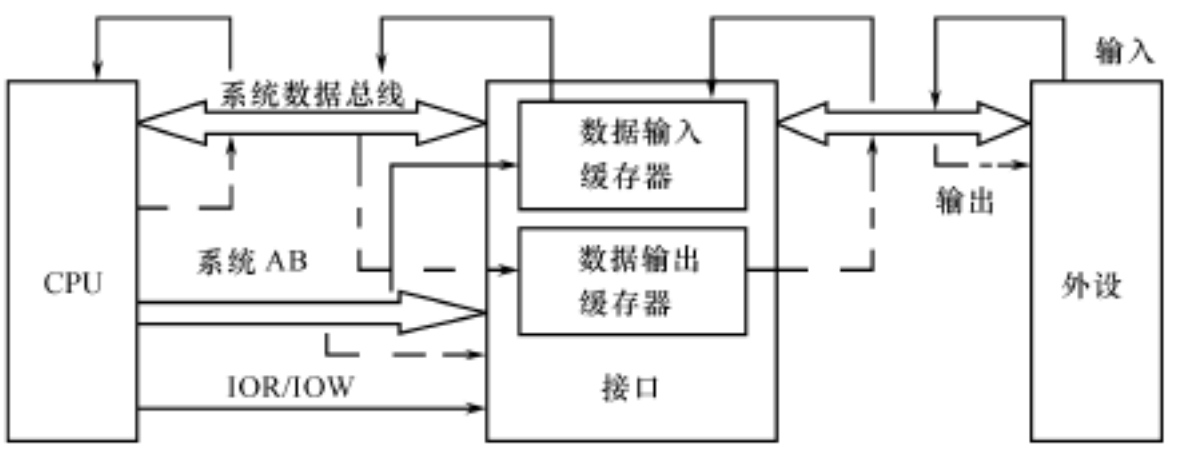


图 2 - 8　CPU 与 I/O 外设之间的数据传送示意

输入一个数据的过程是如下。(以实线表示)

① 外设把数据送入 I/O 接口的数据输入缓存器。

② 数据处理器发出地址码, 通过地址总线寻址该输入缓存器。

③ 把该输入缓存器的数据送上系统数据总线, 处理器从系统数据总线上读取数据, 存入相应寄存器(一般为 A 累加器) 中。

输出一个数据的过程如下。(以虚线表示)

① 处理器把输出接口的地址放在系统地址总线上, 选择某一输出缓存器。

② 处理器把需输出的数据放在系统数据总线上, 并送入接口的相应数据输出缓存器。

③ 处理器确认数据有效后, 从该数据输出缓存器取走数据。

I/O 设备的同步控制方式通常有 4 种, 程序查询式、中断驱动式、直接存储器存取式和专用 I/O 处理机式。当然, 从 I/O 与 CPU 间交换信息来说, 还有一种最简单的无条件传送方式, 但这是一种不需要控制的 I/O 操作方式, 只有在外部控制过程的各种动作时间是固定的, 且是在已知的条件下才能使用, 否则就会出错, 所以不把它作为一种 I/O 控制方式来考虑。

1. 程序查询式控制

在这种 I/O 控制方式中, I/O 设备与 CPU 之间的数据传送完全由 CPU 通过程序查询来实现。CPU 通过反复查询 I/O 设备的状态, 了解哪个设备准备好了, 即转入相应的设备服务程序; 如果外设未准备好, 则 CPU 继续查询。这种控制方式的特点是 I/O 操作由 CPU 启动, 即 CPU 主动, I/O 被动。

2. 中断驱动式控制

在这种控制方式中, I/O 设备与 CPU 之间的数据传送是 CPU 通过响应 I/O 设备发出的中断请求来实现的, CPU 和 I/O 设备的关系是 CPU 被动, I/O 主动, 即 I/O 操作是由 I/O 设备启动的。当 I/O 设备需要 CPU 服务时, 通过其接口发出中断请求信号; CPU 在收到中断请求后, 中断正在执行的程序, 保护断点, 转去为相应外设服务, 执行一个相应的中断服务子程序; 服务完毕后恢复断点, 返回原来被中断的程序继续执行。如果 CPU 没收到中断请求, 则根本不理睬 I/O 设备。

3. 直接存储器存取式控制

在这种控制方式下, I/O 设备是和存储器直接交换信息的, 不需 CPU 介入。这和前两种方式有根本的不同, 前两种方式都是解决 CPU 与 I/O 设备之间的信息交换问题, 它们的特点是对外设的服务都由软件来完成, 外设的数据都需经过 CPU 才能和存储器交换, 这就必然使传输速度受到限制。而 DMA 方式无需 CPU 介入, 外设与存储器之间的数据传输是在硬件的作用下完成的, 因此, 可使传输速度大大提高。

4. 专用 I/O 处理机控制

对于有大量 I/O 设备的微型计算机系统, 前 3 种 I/O 同步控制方式都难以满足需要, 于是, 人们又提出并在实际上广泛采用了一种专用 I/O 处理机控制方式, 把原来由 CPU 完成的各种 I/O 操作与控制全部交给 I/O 处理机(IOP) 去完成。I/O 处理器能够直接存取系统主存储器中的数据, 能够中断 CPU 或被 CPU 查询, 并能直接执行 I/O 程序和数据预处理程序。I/O 处理器完成 I/O 操作和预处理后, 再以查询或中断方式与 CPU 交换数据。

2.6　80x86 系列微处理器简介

美国 Intel 公司是目前世界上最有影响的微处理器生产厂家, 也是世界上第一个微处理

器生产厂家,所生产的 80x86 系列微处理器一直是个人微型计算机的主流 CPU。1971 年,Intel 公司生产的 4 位微处理芯片 4004 宣告了微型计算机时代的到来。1972 年,Intel 公司又开发了 8 位微处理器 8008 芯片;1974 年接着生产了 8080 CPU;1977 年,Intel 将 8080 微处理器及其支持电路集成在一块集成电路芯片上,形成了性能更高的 8 位微处理器 8085。

1. 80286 CPU

1982 年,Intel 推出 80286 CPU,时钟频率 6 MHz~20 MHz,16 位内、外数据总线,地址总线 24 位,物理存储器容量 16 MB。80286 有实模式和保护模式两种工作方式。在实模式下,80286 的工作方式相当于一个快速 8086。在保护模式下,80286 提供了虚拟存储器,提供保护机制。80286 指令系统包括全部 80186 指令及新增的保护方式指令 15 条,其中有些保护方式指令在实模式下也可使用。

8086/80186/80286(包括 8088/80188) CPU 的内部结构都是 16 位的,本书将统称它们为 16 位 Intel 80x86 CPU,简称 16 位 x86 CPU。而下面将要介绍的 80386/80486/Pentium/Pentium II/Pentium III/Pentium 4 等 CPU 的内部结构都是 32 位的,统称它们为 32 位 Intel 80x86 CPU,简称 32 位 x86 CPU。

2. 80386 CPU

1985 年,Intel 80x86 微处理器成为第三代 80386 CPU。80386 CPU 采用 32 位结构,数据总线 32 位,地址总线也是 32 位,可寻址 4 GB 内存和 64 TB 虚拟内存,时钟频率 16/25/33 MHz。80386 除保持与 80286 完全兼容外,又提供了虚拟 8086 工作模式,可同时模拟多个 8086 处理器。80386 指令系统在兼容原来 16 位指令系统的基础上,全面升级为 32 位,还新增了有关位操作、条件设置指令及控制、调试和测试寄存器的传送指令等。

3. 80486 CPU

1989 年,Intel 推出 80486 CPU。80486 微处理器把 80386 CPU、80387 FPU 及 8 KB 高速缓冲存储器 Cache 集成到一个芯片上,它的最高内部时钟频率达到 100 MHz。80486 将浮点处理单元 FPU 集成进来,所以可以直接执行 80387 的所有浮点指令;另外,80486 采用了精简指令集计算机技术 RISC 和指令流水线方式,使它的性能得到提高。80486 指令系统新增了用于多处理器和内部 Cache 操作的 6 条指令。

4. 奔腾系列 CPU

1993 年,Intel 公司推出了 Pentium(奔腾)微处理器。Pentium 仍为 32 位结构,地址总线为 32 位,但外部数据线为 64 位,内部时钟频率 60 MHz~200 MHz。Pentium 对浮点处理单元进行重大改进,如包含了专用的加法、乘法和除法单元;采用具有两条整数流水线的超标量技术;对常用的简单指令用硬件实现,重新设计指令的微代码等。所有这些都大大提高了 Pentium 的整体性能。Pentium 新增了一条 8 字节的比较交换指令、一条处理器识别指令以及 4 条系统专用指令。

1995 年,Intel 推出 Pentium Pro(高能奔腾)即所谓 686,可以寻址主存 64 GB 容量。Pentium Pro 由 16 KB CPU 和 256/512 KB 的二级 Cache 芯片组成。Pentium Pro 扩展了超标量技术,具有 3 个整数处理单元和一个浮点处理单元,能同时执行 3 条指令并对 32 位指令进行优化处理。

1996 年,为了适应多媒体数据的处理需要,Intel 将多媒体扩展 MMX 技术融入 Pentium 形成了 MMX Pentium(多能奔腾)。它的引脚与 Pentium 兼容,时钟频率为 166/200/233

MHz, 内部 Cache 为 32 KB, 可直接用于 Pentium 系统中, MMX Pentium 新增了 57 条多媒体指令, 可用这些指令对图像、音频、视频和通信方面的程序进行优化, 提高了微型计算机对媒体软件的执行速度。

同样, 为了增强对多媒体数据的处理能力, 1997 年, Intel 在 Pentium Pro 中也采用 MMX 技术, 推出了 Pentium II。Pentium II 继承了 MMX 技术和 Pentium Pro 的动态执行技术, 内部一级 Cache 增为 64 KB, 二级 Cache 为 512 KB, 时钟频率进一步提高。

1999 年, 针对互联网和三维多媒体程序的应用要求, Intel 又采用数据流 SIMD 扩展 SSE 技术(原称为 MMX - 2) 开发了 Pentium III。Pentium III 在 Pentium II 的基础上又新增了 70 条 SSE 指令, 极大地提高了浮点 3D 数据的处理能力。

2000 年底, Intel 又推出新增 76 条 SSE2 指令的 Pentium 4。SSE2 指令系统侧重于增强浮点运算能力, 加上 Pentium 4 的 32 位 Net Burst 微结构以及 400 MHz 外部时钟, 使 32 位微处理器的性能更高。

2.7　微型计算机软件系统

软件是计算机系统的重要组成部分, 它可以使机器更好地发挥其功能。软件可分为系统软件和应用软件。

1. 系统软件

系统软件是为了方便使用、维护和管理计算机系统而编制的一类软件及其文档, 它包括操作系统、语言翻译程序等。系统软件是面向计算机系统的, 通常有计算机厂家提供的程序及其文档, 它是用户使用计算机时为产生、准备和执行用户程序所必需的程序。在系统软件中, 最重要的软件是操作系统。操作系统负责管理整个系统的软硬件资源, 向用户提供交互的界面, 为所有其他程序的运行打下基础。用户借助操作系统使用计算机系统, 程序员也要采用操作系统提供的驱动程序编写用户程序。程序员采用某种程序设计语言写源程序, 利用语言翻译程序将源程序转变成可运行的程序。例如, 本书介绍用汇编语言设计源程序的方法, 就必须利用“ 汇编程序 ”完成源程序的翻译工作。高级语言则采用编译类程序来完成这个工作。

2. 应用软件

应用软件是解决某一问题的程序及其文档。它覆盖了计算机应用的所有方面, 每一个应用都有相应的应用程序。微型计算机系统具有多种多样的应用软件。例如, 进行程序设计时要采用文本编辑软件编写源程序, 带有丰富格式的字处理软件用于书写文章, 排版软件则用于书刊出版。大型的程序设计项目往往要借助软件开发工具(包), 这个开发工具是进行程序设计所用到的各种软件的有机集合, 所以也被称为集成开发环境。其中, 有文本编辑器、语言翻译程序和用于形成可执行文件的连接程序。

习　　题

1. 简述微型计算机的硬件结构。
2. 什么是存储单元, 什么是存储单元的地址?

3. 什么是堆栈,什么是堆栈指针?
4. 简述 8086 CPU 的内部寄存器组。
5. 简述 8086 CPU 芯片引脚中的关于存储器操作和 I/O 接口操作的引脚。
6. 简述 I/O 同步控制方式。
7. 简述 80x86CPU 的发展概况
8. 什么是微型计算机软件系统?
9. 什么叫冯·诺依曼计算机?
10. 为什么把微型计算机的基本结构说成是总线结构? 试简述总线结构的分类及其优缺点。

第 3 章　8086/8088 寻址方式与指令系统

本章针对 8086/8088 微处理器，详细介绍其指令系统和寻址方式，对各类指令的指令格式、指令功能和使用方法进行重点阐述，为进一步学习汇编语言程序设计奠定基础。

3.1　8086/8088 的寻址方式

所谓寻址方式，就是寻找指令中操作数所在地址的方式。8086/8808 微处理器可采用许多不同的方法来存取指令操作数，其操作数所在地址有 3 种可能。

① 直接包含在指令中，即指令的操作数部分就是操作数本身。

② 包含在 CPU 的某个内部寄存器中，这时指令中的操作数是 CPU 的一个内部寄存器内容。

③ 在内存储器中，这时指令中的操作数部分包含着该操作数所在的内存地址。

3.1.1　有效地址 EA

1．存储器分段管理机制

由于 8086/8088 CPU 有 20 根地址线，故可寻址的地址空间为 2^{20} 即 1 MB，而 CPU 内部寄存器均为 16 位，故可寻址的地址空间为 2^{16} 即 64 KB。为了解决这个问题，8086/8088 CPU 采用了存储器分段管理的方法，即将整个 1 MB 的物理存储空间分成若干个逻辑段，每个逻辑段的最大长度为 64 KB。这样，对于一个具体的存储单元，即由此单元所在段的起始地址和段内偏移地址来确定。采用存储器分段管理后，存储器地址就有物理地址和逻辑地址之分。物理地址是指内存单元通过地址总线 AB 确定的 20 位实际地址，即某一地址可惟一表示 1 MB 存储空间的某一个存储单元，其编码范围是 00000H～FFFFFH；逻辑地址则供编程时使用，由段的起始地址和段内偏移地址两部分组成，两者都是 16 位。由 16 位逻辑地址转换为 20 位物理地址的关系如下。

物理地址 = 段地址× 10H + 偏移地址

物理地址的生成是在 CPU 内部 BIU 的地址加法器中完成的，其形成过程如图 3 - 1 所示。

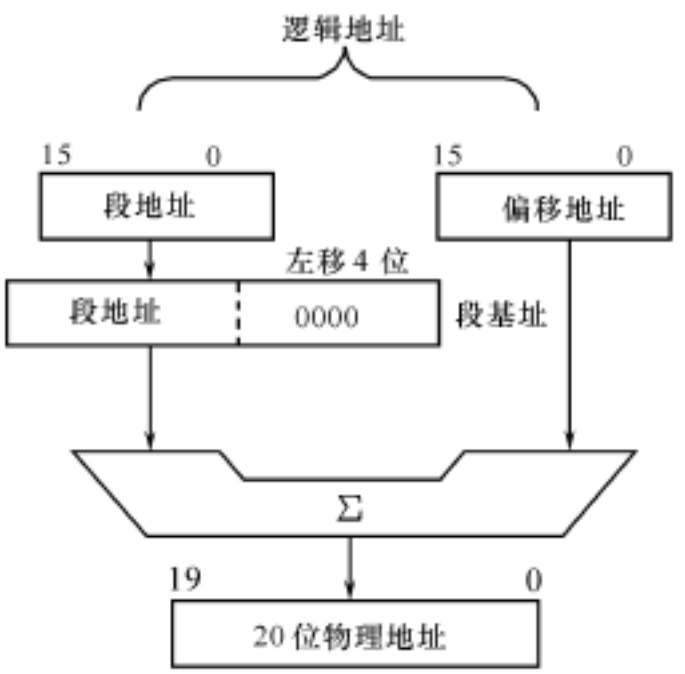


图 3 - 1　物理地址形成示意图

例如，MOV AL, ES: [2000H]

假设　ES = 3000H

则逻辑地址为 3000H: 2000H，物理地址为 3000H× 10H + 2000H = 32000H

2．有效地址 EA

为了适应处理各种数据结构的需要，段内偏移地址可以由以下几个基本部分组合而成。

基址寄存器内容。

变址寄存器内容。

位移量。

通常将这 3 种元素有效组合形成的偏移地址称为有效地址 EA(Effective Address) , 即

EA = 基址 + 变址 + 位移量

其中, 基址为 BX, BP; 变址为 SI, DI; 位移量为 0, 8, 16。

例如, OV AL, DS: 80H[BX]

则: EA = BX + 80H

3.1.2 段约定和段更换

若指令中的操作数不在约定的段区域内, 则必须在指令中指定其所在的段寄存器, 这就是段超越。例如,

MOV AL, [2000H]　　约定段为 DS　　物理地址 = 16× DS + 2000H

MOV AL, ES: [2000H]　　段更换为 ES　　物理地址 = 16× ES + 2000H

段地址的基本规定(或约定) 和允许更换的情况如表 3 - 1 所示。

表 3 - 1　存储器存取时的约定段

存储器存取方式	约定段	可修改段	偏移
取指令	CS	无	IP
堆栈操作	SS	无	SP、BP
源串	DS	CS、ES、SS	SI
目的串	ES	无	DI
BP 作为地址	SS	CS、DS、ES	有效地址 EA
通用数据读写	DS	CS、ES、SS	有效地址 EA

3.1.3 立即寻址

在这种寻址方式下, 操作数作为立即数直接存放在指令中, 即操作数紧跟在操作码之后, 操作数可为 8 位或 16 位。

例如, MOV CX, 2129H

该指令的功能是将立即数 29H 送 CL, 21H 送 CH, 其执行过程如图 3 - 2 所示。

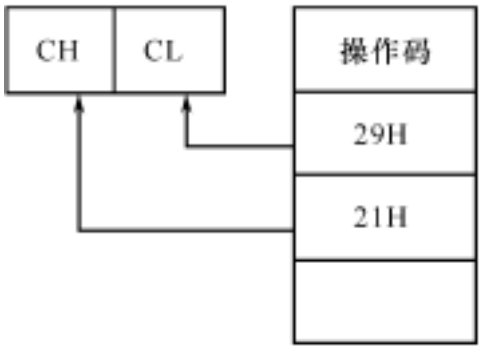


图 3 - 2　立即数寻址示意图

3.1.4 寄存器寻址

寄存器寻址也叫寄存器直接寻址。由于该方式的操作数存放于 CPU 的某个内部寄存器中, 不需要访问存储器, 因此执行速度较快, 是经常使用的方法。

例如, MOV DS, AX

该指令的功能是将寄存器 AX 的内容直接送到段寄存器 DS, 其执行过程如图 3 - 3 所

示, 即将 8756H 送 DS。

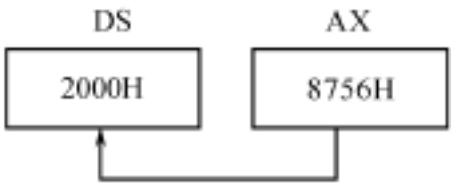


图 3 - 3　寄存器寻址示意

3.1.5　存储器寻址方式

寄存器寻址虽然速度较快, 但 CPU 中寄存器数目有限, 不可能把所有参与运算的数据都存放在寄存器中。多数情况下, 操作数还是存于内存中。在内存中寻找操作数的方式统称为存储器寻址方式, 具体包含 5 种, 下面分别进行介绍。

1. 直接寻址方式

直接寻址实质上是存储器直接寻址的简称。指令中直接给出操作数的有效地址 EA, 即操作码后紧跟着操作数的有效地址。

例如, MOV AX, [2000H]

假设

DS = 1492H

PA = 1492H× 10H + 2000H = 16920H

则该指令是将 16920H 单元的内容 78H 送到 AL 寄存器中, 将 16921H 单元的内容 5BH 送到 AH 寄存器中, 其执行过程如图 3 - 4 所示。

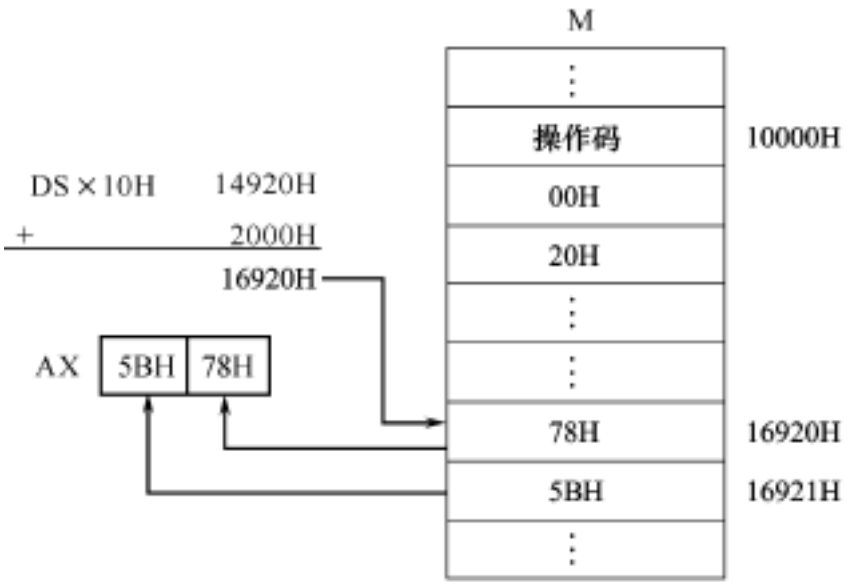


图 3 - 4　直接寻址示意图

2. 寄存器间接寻址方式

寄存器间接寻址中寄存器的内容是操作数的有效地址, 即 EA = [寄存器] 。

例如, MOV AX, [BP]

假设　SS = 6000H, BP = 3000H

PA = 6000H× 10H + 3000H = 63000H

则该指令是将 63000H 单元的内容 20H 送到 AL 寄存器中, 将 63001H 单元的内容 88H 送到 AH 寄存器中, 其执行过程如图 3 - 5 所示。

3. 寄存器相对寻址方式

在这种寻址方式下, 寄存器(BX、BP、SI、DI) 的内容加位移量(8 位或 16 位) 构成有效地址, 即 EA = [寄存器] + 位移量。

例如, MOV CX, [BX + 1000H]

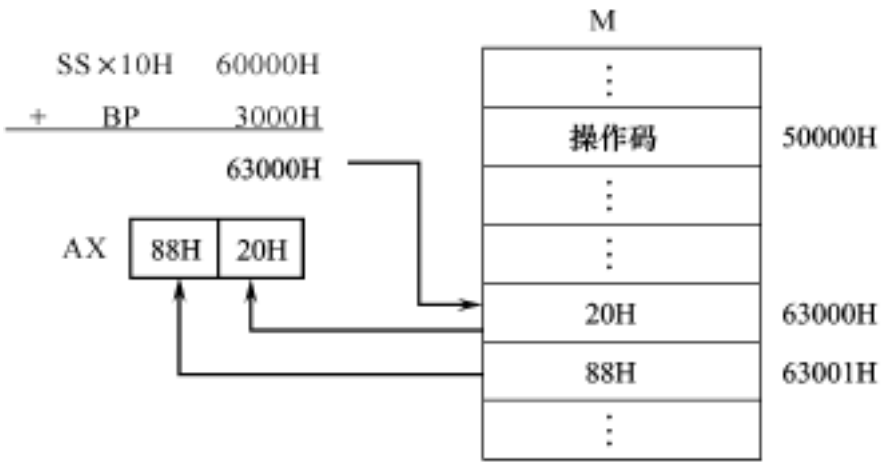


图 3 - 5　寄存器间接寻址示意图

假设　DS = 5000H, BX = 2000H

PA = 5000H× 10H + 2000H + 1000H = 53000H

则该指令是将 53000H **单元的内容** 70H **送到** CL **寄存器中, 将** 53001H **单元的内容** 58H **送到** CH **寄存器中, 其执行过程如图** 3 - 6 **所示。**

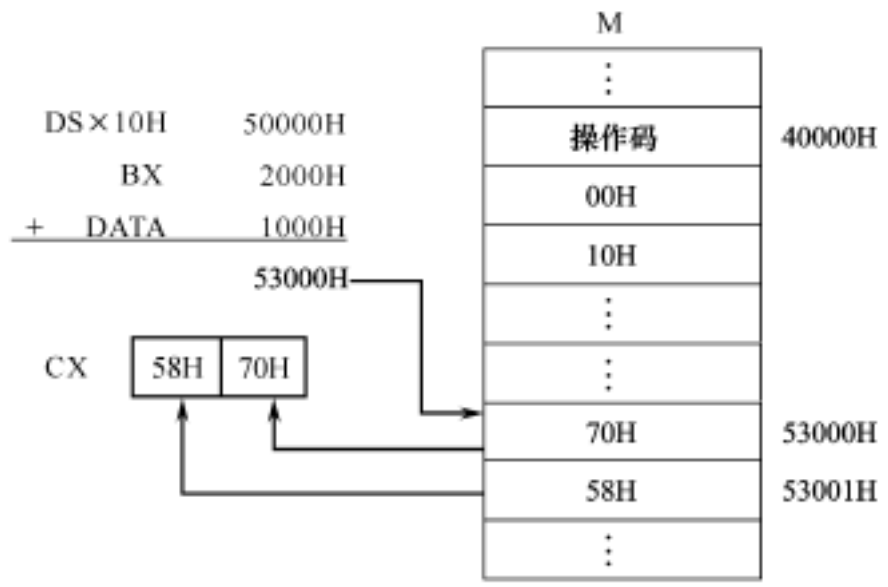


图 3 - 6　寄存器相对寻址示意图

4. **基址变址寻址方式**

这种寻址方式是把一个基址寄存器(BX **或** BP**) 的内容加上一个变址寄存器(** SI **或** DI**) 的内容构成有效地址, 即**

EA = [**基址寄存器**] + [**变址寄存器**] 。

例如, MOV CX, [BX + SI]

假设

DS = 6000H, BX = 2500H, SI = 3000H

PA = 6000H× 10H + 2500H + 3000H = 65500H

则该指令是将 65500H **单元的内容** F0H **送到** AL **寄存器中, 将** 65501H **单元的内容** 08H **送到** AH **寄存器中, 其执行过程如图** 3 - 7 **所示。**

5. **相对基址变址寻址方式**

相对基址变址寻址方式, 是用基址寄存器和变寄存器及指令中指定一个位移量三者之

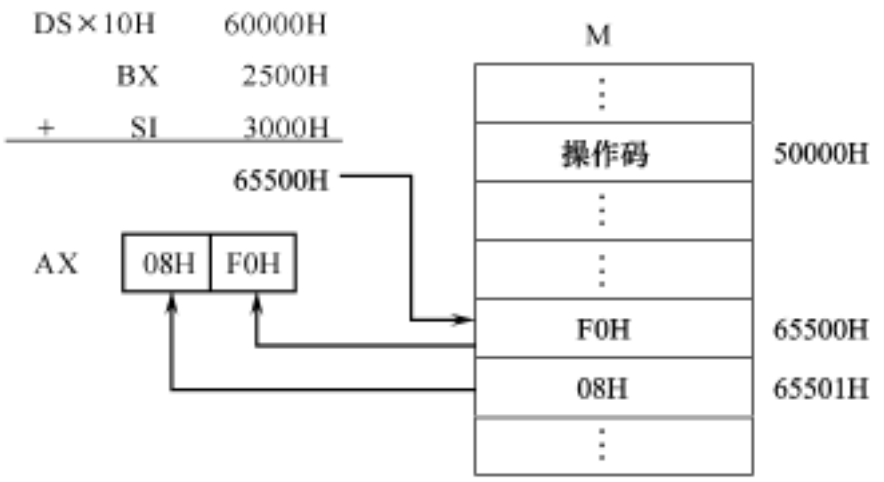


图 3 - 7　基址变址寻址示意图

和构成操作数的有效地址, 即 EA = [基址寄存器] + [变址寄存器] + 位移量

例如, MOV BX, [BP + SI + DATA]

假设

SS = 4000H, BP = 2000H

SI = 3500H, DATA = 3000H

PA = 4000H× 10H + 2000H + 3500H + 3000H = 48500H

则该指令是将 48500H 单元的内容 80H 送到 BL 寄存器中, 将 48501H 单元的内容 2BH 送到 BH 寄存器中, 其执行过程如图 3 - 8 所示。

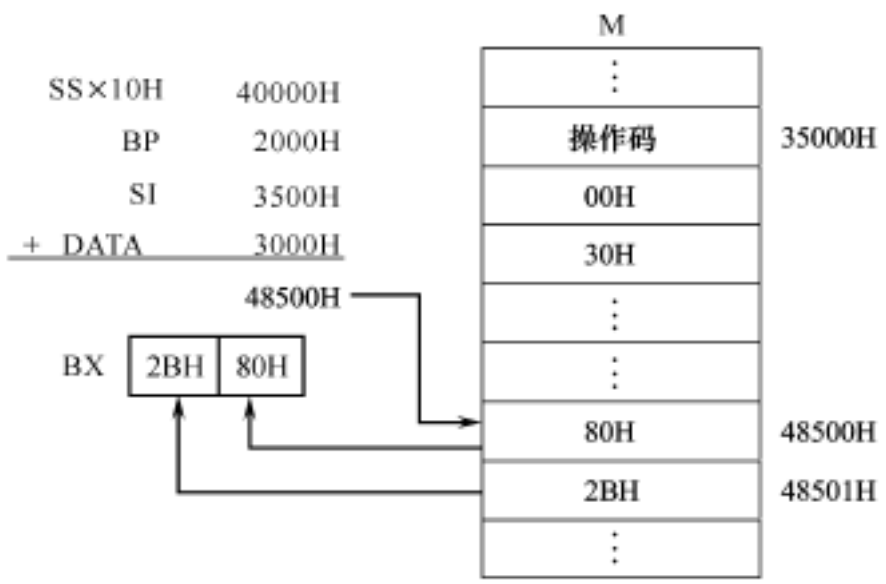


图 3 - 8　相对基址变址寻址示意图

3.2　8086 指令系统

3.2.1　传送指令

1. 数据传送指令(MOV)

指令格式: MOV OP1, OP2

其中, OP1 为目的操作数, 可以为通用寄存器、段寄存器、存储器。OP2 为源操作数, 可以为

通用寄存器、存储器、立即数。

功能描述: MOV 指令的功能是把 OP2 的内容传送到 OP1 中, OP1 和 OP2 可为字节、字或双字, 但两者的数据位数必须等长。例如,

```
MOV AL, BL              ; 将 BL 寄存器的内容送 AL 寄存器
MOV [2000H], 80H        ; 将 80H 送[2000H] 单元
MOV CX, 3000H           ; 将 3000H 送 CX 寄存器
```

要注意, 下列情况是错误的。

```
MOV 80H, AL             ; 立即数不能作为目的操作数
MOV [2000H], [SI]       ; 存储单元内容不能直接传送
MOV DS, 8000H           ; 立即数不能直接送到段寄存器
MOV CS, AX              ; CS 不能作为目的操作数
MOV DS, ES              ; 段寄存器之间不能直接传送
```

2. 堆栈操作指令(PUSH / POP)

指令格式: PUSH OP1

功能描述: 压栈指令, 即 OP1 压入堆栈, 同时(SP - 2) 送 SP。

指令格式: POP OP1

功能描述: 出栈指令, 即 OP1 弹出堆栈, 同时(SP + 2) 送 SP。

【例 3. 1】 利用堆栈完成数据传送, 程序段如下。

```
MOV     AX, 8060H
MOV     BX, 1234H
PUSH    AX
POP     BX
INT
```

执行后 AX = BX = 8060H, SP 的内容不变。

3. 标志传送指令(LAHF / SAHF)

指令格式:　AHF

　　SAHF

功能描述: LAHF 为标志寄存器读指令, 用于将 FLAGS 寄存器的低字节读出后传送到 AH 寄存器的指定位。SAHF 为标志寄存器写指令, 它正好与 LAHF 指令的操作相反, 用于将 AH 寄存器的内容写入到 FLAGS 寄存器的低字节中。

4. 地址传送指令

(1) 装入有效地址指令(LEA)

指令格式: LEA OP1, OP2

功能描述: 将 OP2 的有效(偏移) 地址送给 OP1。

例如, EA SI, FIRST ; SI = FIRST

　　LEA SI, DATA[BX] ; SI = DATA + BX

(2) 装入全地址指针指令(LDS / LES / LSS)

指令格式: LDS / LES / LSS OP1 , OP2

功能描述: 将一个 32 位地址装入一个段寄存器和一个通用寄存器中。

例如，ABLE DD 12345678H

LDS SI, TABLE

指令执行后, DS = 1234H SI = 5678H

5．**输入/输出指令**

(1) **输入指令**

指令格式: IN OP1, OP2

功能描述: **将** I/O **端口的内容输入到** AL **或** AX **累加器, 其中** OP1 **为** AL(AX) **寄存器,** OP2 **为端口地址。**

(2) **输出指令**

指令格式: OUT OP1, OP2

功能描述: **将** AL **或** AX **累加器的内容通过** I/O **端口输出, 其中** OP1 **为端口地址,** OP2 **为** AL(AX) **寄存器。**

通常, 访问 I/O **端口的方法有两种, 用** 8 **位立即数指定端口地址**(00H ~FFH) , **即直接寻址方式; 或采用** DX **寄存器间接寻址方法, 可寻址** 64 KB **个端口**(0000H ~FFFFH) **中的任一端口。**

例如，
```
IN    AL, 80H     ; 将 80H 端口的内容送 AL 寄存器
OUT   30H, AL     ; 将 AL 寄存器的内容送 30H 端口
MOV   DX, 30AFH   ; 将端口地址 30AFH 送 DX 寄存器
IN    AL, DX      ; 将 DX 端口的内容送 AL 寄存器
OUT   DX, AL      ; 将 AL 寄存器的内容送 DX 端口
```

6．**查表转换指令**(XLAT)

指令格式: XLAT

功能描述: XLAT **指令是字节查表转换指令, 它可以根据表中元素的下标, 查出表中相应元素的内容。在使用该指令以前, 应先建立一个字节表格, 并将表的首地址送给** BX, **表中元素的无符号下标送** AL **寄存器, 执行** XLAT **指令后, 即完成** [BX + AL] →AL。

【例 3. 2】 **内存的数据段中存有一张十六进制的** ASCII **码表, 其首地址为** TABLE, **如图** 3 - 9 **所示。若需要查寻第** 10 **个元素的** ASCII **码, 则程序段如下。**

```
TABLE  DB   30H, 31H, 32H, 33H, 34H, …; 46H
       …
       MOV BX, OFFSET TABLE
       MOV AL, 09H
       XLAT
       HLT
```

执行后 AL = 39H。

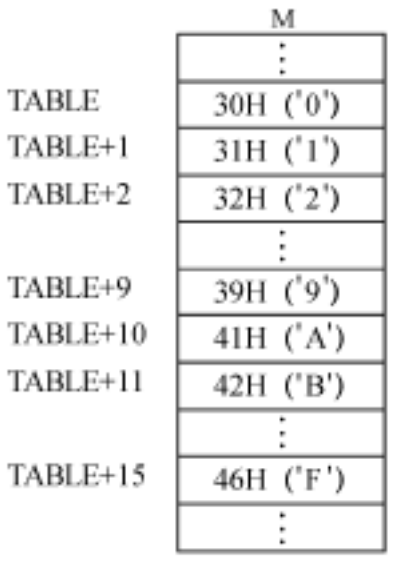


图 3 - 9　**十六进制的** ASCII **码表**

7．**交换指令**(XCHG)

指令格式: XCHG OP1, OP2

功能描述: **用于将操作数** OP1 **与** OP2 **的内容互相交换。**

例如，MOV AX, 1000H

```
MOV BX, 2000H
XCHG AX, BX
INT
```

执行后, AX = 2000H　BX = 1000H

3.2.2　算术运算指令

1. 加法指令

加法指令包括 ADD, ADC, INC, AAA, DAA **指令, 它们可完成单字节或多字节的加法运算。**

(1) 加法指令(ADD)

指令格式: ADD　OP1, OP2

功能描述: **将** OP1 **和** OP2 **两数相加, 结果存放在** OP1 **中。**

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF	AF	PF	CF
X	-	-	-	X	X	-	X	-	X	-	X

例如,
```
ADD　AL, 08H　　; AL 加 08H, 结果存入 AL
ADD　AX, BX　　; AX 加 BX, 结果存入 AX
ADD　[SI], CL　　; [SI] 加 CL, 结果存入[SI]
```

(2) 带进位加法指令(ADC)

指令格式: ADC OP1 , OP2

功能描述: **将** OP1、OP2 **和进位标志位** CF **相加, 结果存放在** OP1 **中。**

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF	AF	PF	CF
X	-	-	-	X	X	-	X	-	X	-	X

【例 3.3**】　编程实现** 12345678H + 87654321H。

程序段如下:

```
FIRST　　DD　12345678H　; 自 FIRST 开始存放数据, 一个 32 位数
SECOND　DD　87654321H　; 自 SECOND 开始存放数据, 一个 32 位数
　　…
LEA　　SI, FIRST
LEA　　DI, SECOND
MOV　　AX, [SI]
ADD　　AX, [DI]
MOV　　BX, [SI +2]
ADC　　BX, [DI +2]
INT
```

(3) 加 1 **指令**(INC)

指令格式: INC　OP1

功能描述: **对** OP1 **指定的操作数加** 1, **结果再返回此操作数。**

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	-

例如,　INC　SI　　;SI 内容增 1

　　INC　[DI]　　;[DI]的内容增 1

(4) BCD 码(十进制数)加法调整指令(AAA/DAA)

指令格式: AAA

功能描述: AAA 为未压缩 BCD 码的加法调整指令,用于对两个未压缩 BCD 码数相加后存放于 AL 中的和进行调整,以获得正确的未压缩 BCD 码结果。程序中 AAA 指令紧跟在 ADD、ADC 指令之后。

指令格式: DAA

功能描述: DAA 为压缩 BCD 码的加法调整指令,用于对两个压缩 BCD 码数相加后的结果(同样存放在 AL 中)进行调整,以获得正确的压缩 BCD 码结果。在程序中 DAA 指令也是紧跟在加法指令 ADD, ADC 之后。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
U	-	-	-	U	U	-	X	-	U	-	X
U	-	-	-	X	X	-	X	-	X	-	X

【例 3.4】　有两个十进制数以压缩 BCD 码的形式分别存放在数据段中从 FIRST 和 SECOND 开始的存储区中,低字节在地址的低处,相加结果存入以 SUM 开始的存储区中。

程序段如下:

```
FIRST    DW    1234H      ;自 FIRST 开始存放数据,一个 16 位数
SECOND   DW    5678H      ;自 SECOND 开始存放数据,一个 16 位数
SUM      DW    ?          ;自 SUM开始定义一个字的空间
    …
LEA      SI,FIRST
LEA      DI,SECOND
LEA      BX,SUM
MOV      AL,[SI]
ADD      AL,[DI]
DAA
MOV      [BX],AL
MOV      AL,[SI+1]
ADC      AL,[DI+1]
DAA
MOV      [BX+1],AL
HLT
```

2. 减法指令

减法指令包括 SUB, SBB, DEC, AAS, DAS, CMP 指令, 它们可完成单字节或多字节的减法运算。

(1) 减法指令(SUB)

指令格式: SUB OP1, OP2

功能描述: 将 OP1 减去 OP2 的结果存放在 OP1 中。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	X

例如,
```
SUB   AL, F3H        ; AL 减 F3H, 结果存入 AL
SUB   BX, CX         ; BX 减 CX, 结果存入 BX
```

(2) 带借位减法指令(SBB)

指令格式: SBB OP1, OP2

功能描述: SBB 是带借位减法指令, 其功能是用 OP1 减去 OP2, 再减去借位标志 CF, 结果存于 OP1 中。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	X

【例 3.5】 编程实现 56781234H - 12348765H。

程序段如下:

```
FIRST     DD  12345678H
SECOND    DD  87654321H
          …
LEA       SI, FIRST
LEA       DI, SECOND
MOV       AX, [SI]
SUB       AX, [DI]
MOV       BX, [SI+2]
SBB       BX, [DI+2]
INT
```

(3) 减 1 指令(DCE)

指令格式: DEC　OP1

功能描述: 对 OP1 指定的操作数减 1, 结果返回送 OP1。

对标志位的影响:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	-

【例 3.6】 从 100H 端口输入 50 个字节的数据存入 FIRST 开始的存储区中。

程序段如下:

```
          FIRST  B   50DUP(?)        ;定义 50 个字节的存储空间
                 …
          LEA    SI,FIRST
          MOV    CL,50
          MOV    DX,0100H
NEXT:     IN     AL,DX
          MOV    [SI],AL
          INC    SI
          DEC    CL
          JNZ    NEXT                ; 若 CL 不为零,则转到 NEXT 标号指令处
          HLT
```

(4) 十进制数(BCD)减法调整指令(AAS/DAS)

指令格式: AAS

功能描述: AAS 指令为未压缩 BCD 码的减法调整指令,其功能与 AAA 指令类似,用于对两个未压缩的 BCD 码相减后存放于 AL 中的结果进行调整,以获得正确的未压缩 BCD 码结果。

指令格式: DAS

功能描述: DAS 指令为压缩 BCD 码的减法调整指令,其功能与 DAA 指令类似,用于对两个压缩 BCD 码数相减后的差(同样存放在 AL)进行调整,以获得正确的压缩 BCD 码结果。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
U	-	-	-	U	U	-	X	-	U	-	X
U	-	-	-	X	X	-	X	-	X	-	X

(5) 整数取负指令(NEG)

指令格式:　NEG OP1

功能描述: 对操作数取负,即用零减去 OP1,其结果将使正数变负数或负数变正数,但绝对值不变。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	X

例如,

```
NEG    AL      ;0 - AL 送 AL
NEG    [SI]    ;0 - [SI] 送[SI]
```

(6) 比较指令(CMP)

指令格式: CMP OP1, OP2

功能描述: CMP 指令和 SUB 指令类似,也是将 OP1 减去 OP2。与 SUB 指令不同的是,CMP 指令不将相减结果送回 OP1,只是使结果影响标志位。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	X

CMP 指令主要用于比较两个数之间的关系,即两者是否相等,或两者哪个大。表 3 - 2 给出了无符号数比较和有符号数比较两种情况下,其状态标志反映的两数大小关系。

如果直接按照表 3 - 2 进行两个数的比较,显得比较烦琐。在实际编程时,通常是通过比较指令和条件转移指令来实现两个数的比较,但要注意参与比较的两个数是带符号数还是无符号数。

表 3 - 2　状态标志反映两数大小关系

两数比较关系 (A 比 B)		受影响标志			
		CF	ZF	SF	OF
A = B		0	1	0	0
无符号数	A < B	1	0	-	-
	A > B	0	0	-	-
有符号数	A < B	-	0	0	1
		-	0	1	0
	A > B	-	0	0	0
		-	0	1	1

【例 3. 7】 在 FIRST 开始的内存区域存有两个 16 位带符号数,试比较它们的大小,将大数存入 MAX 中。

程序段如下:

```
FIRST    DW    890EH,78BCH
           …
         LEA SI,FIRST
         MOV AX,[SI]
         MOV BX,[SI+2]
         CMP AX,BX
         JGE NEXT              ;若 AX ≥ BX 则转到 NEXT 标号(带符号数比较,用 JGE)
         XCHG AX,BX
NEXT:    MOV [MAX],AX
         HLT
```

如果要求比较无符号数,只需将上述程序段中的指令 JGE NEXT 换为用 JAE NEXT 指令即可。

3. 乘法指令

乘法指令包括 MUL, IMUL, AAM 指令,它们可完成单字节或多字节的乘法运算。

(1) 无符号数乘法指令(MUL)

指令格式: MUL OP1

功能描述: MUL 指令用于完成无符号整数乘法,被乘数隐含在 A 累加器(AL, AX) 中,

操作数 OP1 为乘数。字节操作时乘积返回到 AX, 字操作时乘积返回到 DX: AX。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	U	U	-	U	-	U	-	X

例如,
```
MUL BL                  ; AL* BL 结果送 AX
MUL WORD PTR [DI]       ; AX* [DI] 结果送 DX, AX
```

(2) 带符号数乘法指令(IMUL)

指令格式: MUL OP1

IMUL OP1, OP2

功能描述: IMUL 指令用于完成带符号数的乘法运算。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	U	U	-	U	-	U	-	X

在单操作数格式时, 被乘数存放在累加器(AL、AX)中, 乘数存放在由 OP1 指定的寄存器或存储器中。字节运算时乘积返回 AX, 字运算时乘积返回到 DX, AX。

在双字节格式时, 被乘数存放在由 OP1 指定的寄存器中, 乘数由 OP2 指定, 可存放在寄存器或存储器中, 也可为立即数, 返回的乘积存放在 OP1 指定的寄存器中。例如,
```
IMUL    AL              ; AL* AL 结果送 AX
IMUL    AL, BL          ; AL* BL 结果送 AX
```

(3) BCD 码(十进制数)乘法调整指令(AAM)

指令格式: AAM

功能描述: 指令为未压缩 BCD 码的乘法调整指令, 用于对两个未压缩的 BCD 码相乘后存放于 AX 中的乘积(高位在 AH 中, 低位在 AL 中)进行调整, 以获得正确的结果。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
U	-	-	-	X	X	-	U	-	X	-	U

【例 3.8】 实现两个未压缩的 BCD 码操作数的乘法运算程序。

```
MOV     AL, 06      ; 被乘数
MOV     BL, 08      ; 乘数
MUL     BL
AAM                 ; 十进制乘法调整
HLT
```

4. 除法指令

除法指令包括 DIV, IDIV, AAD 指令, 它们可完成单字节或多字节的除法运算。

(1) 无符号数除法指令(DIV)

指令格式: DIV OP1

功能描述: DIV 为无符号数的除法指令。除数由 OP1 指定, 被除数由累加器 A 隐含决定, OP1 为字节数据时使用 AX, OP1 为字数据时使用 DX, AX; 商则根据数据宽度分别返回到 AL 或 AX 中, 余数则相应存入 AH 或 DX 中。

对标志位的影响:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
U	-	-	-	U	U	-	U	-	U	-	U

(2) **带符号数除法指令**(IDIV)

指令格式: IDIV OP1

功能描述: IDIV 为带符号数的除法指令。除数由 OP1 指定, 被除数由累加器 A 隐含决定, OP1 为字节数据时使用 AX, OP1 为字数据时使用 DX: AX; 商则根据数据宽度分别返回到 AL 或 AX 中, 余数则相应存入 AH 或 DX 中。

对标志位的影响:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
U	-	-	-	U	U	-	U	-	U	-	U

例如,　　DIV　BL　　; AX/BL 结果送 AL

　　IDIV CX　　; DX: AX/CX 结果送 AX

(3) BCD 码(十进制) 除法调整指令(AAD)

指令格式: AAD

功能描述: AAD 指令用于调整除法运算前 AX 中的被除数内容(将它由未压缩 BCD 码调整为二进制数) , 以使除法所得的商(放在 AL 中) 和余数(放在 AH) 都为有效的未压缩 BCD 码。

对标志位的影响:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
U	-	-	-	X	X	-	U	-	X	-	U

【例 3. 9】　实现两个未压缩的 BCD 码操作数的除法运算。

程序段如下:

```
MOV　　AX, 0302H
MOV　　AH, 09
AAD
DIV　　BL
HLT
```

(4) **符号位扩展指令**(CBW /CWD)

指令格式:　CBW

　　CWD

功能描述: CBW, CWD 指令用于将原操作数的宽度加倍, 所以也叫数据宽度变换指令。CBW 指令将 AL 中的 8 位有符号数带符号扩展为 16 位放入 AX 中。AH 中各位取与 AL 中

数的符号位相同的值,即对负数进行 1 位扩展,对正数进行 0 扩展; CWD 指令将 AX 中的 16 位有符号数带符号扩展成 32 位,扩展后的高 16 位存放在 DX 中,各位值与原 AX 中符号位相同。该指令对标志位无影响。

【例 3.10】 假设自 FIRST 开始的内存中存有 X1, X2, X3,计算表达式 (X1 + X2) /X3 的值,设 X1、X2、X3 均为 8 位带符号数,要求将结果存入 RESULT 单元。

程序段如下:

```
FIRST   DB    X1, X2, X3        ;自 FIRST 开始存放一组字节数
RESULT  DB    ?
        …
        LEA   SI, FIRST
        MOV   AL, [SI]
        ADD   AL, [SI +1]
        CBW
        IDIV  [SI +2]
        MOV   [SI +3], AX
        HLT
```

3.2.3 逻辑运算指令

(1) 逻辑“与”指令(AND)

指令格式: AND OP1, OP2

功能描述: 将 OP1 和 OP2 按位进行逻辑“与”运算,并将结果送回 OP1。

对标志位的影响如下:

OF	DF	IF	TF	SF	ZF		AF		PF		CF
0	-	-	-	X	X	-	U	-	X	-	0

例如:
```
AND   AL, 80H
AND   AX, BX
AND   [SI +10H], AL
```

(2) 逻辑“或”指令(OR)

指令格式: OR OP1, OP2

功能描述: OR 指令用于将 OP1 和 OP2 按位进行逻辑“或”运算,并将结果送回 OP1。

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
0	-	-	-	X	X	-	U	-	X	-	0

(3) 逻辑“非”指令(NOT)

指令格式: NOT OP1

功能描述: NOT 指令的功能是使 8 位、16 位或 32 位寄存器或是存储器中的操作数各位都求反(0 变 1, 1 变 0)。

(4) XOR 逻辑“异或”指令

指令格式: XOR OP1, OP2

功能描述: XOR 指令用于将 OP1 和 OP2 按位进行逻辑“异或”运算, 并将结果送回 OP1。

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
0	-	-	-	X	X	-	U	-	X	-	0

【例 3.11】 从偏移地址 DATA 开始的内存区中, 存放着 100 个字节的十六进制数, 试将这些数进行累加, 并将累加和的低位存入 SUM 单元, 高位存入 SUM + 1 单元。

程序段如下:

```
        DATA    DB      12H, 23H, 67H, 78H, …
        SUM     DW      ?
                …
        LEA     BX, DATA
        MOV     CL, 100
        XOR     AX, AX
LOOP:   ADD     AL, [BX]
        JNC     NEXT
        INC     AH
NEXT:   INC     BX
        DEC     CL
        JNZ     LOOP
        MOV     SUM, AX
HLT
```

(5) **测试指令**(TEST)

指令格式: TEST OP1, OP2

功能描述: 将 OP1 和 OP2 按位进行逻辑“与”运算, 结果反映在状态标志位上。

对标志位的影响。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
0	-	-	-	X	X	-	U	-	X	-	0

【例 3.12】 实现对 AL 中数据的变补码操作。

程序段如下:

```
        TEST    AL, 80H
        JZ      NEXT        ; 若为正数, 则转到 NEXT
        AND     AL, 3FH     ; 若为负数, 则对 AL 变补码
        INC     AL
        …
NEXT:
```

3.2.4　移位指令

1. 算术逻辑移位指令

移位指令包括算术左移指令、逻辑左移指令、算术右移指令和逻辑右移指令 4 条。这些指令都可以对寄存器操作数和存储器操作数进行指定的移位。

(1) 算术/逻辑左移指令(SAL/SHL)

指令格式: SAL/SHL　OP1, OP2

功能描述: SAL 是算术左移指令, SHL 是逻辑左移指令。OP1 每左移一位在最低有效位补 0, 把最高有效位移进 CF。OP2 为移位次数(1 或 CL, CL 的最大值为 31)。执行过程如图 3 - 10 所示。

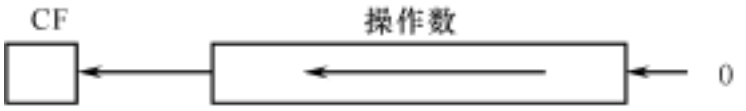


图 3 - 10　SAL/SHL 指令操作示意图

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	U	-	X	-	X

【例 3.13】　计算 4× 8。

```
MOV     AL, 04H
MOV     CL, 03H
SAL     AL, CL
INT
```

(2) 算术/逻辑右移指令(SAR/SHR)

指令格式:　SAR/SHR　OP1 , OP2

功能描述: SAR 是算术右移指令, 用于对有符号数右移; SHR 是逻辑右移指令, 用于对无符号数右移。操作过程如图 3 - 11 所示。

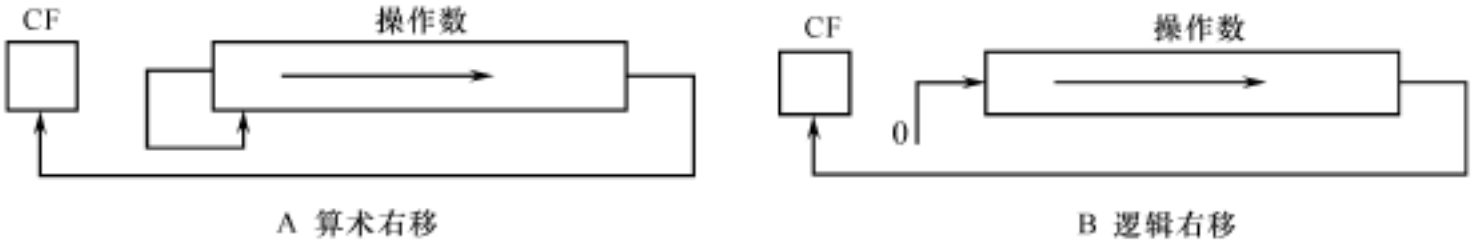


图 3 - 11　SAR/SHR 指令操作示意图

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	U	-	X	-	X

【例 3. 14】　将 AX 的内容除以 4。

若 AX 的内容为带符号数,则程序段如下。

```
MOV     CL, 2
SAR     AX, CL
HLT
```

若 AX 的内容为无符号数,则程序段如下。

```
MOV     CL, 2
SHR     AX, CL
HLT
```

2. 循环移位指令

循环移位指令(ROL/ROR/RCL/RCR) 有 4 条。

指令格式: ROL/ROR/RCL/RCR OP1, OP2

功能描述: ROL 是循环左移指令, ROR 是循环右移指令, RCL 和 RCR 则分别是带进位的循环左移指令和循环右移指令。ROR, RCR 的操作过程如图 3 - 12 所示。

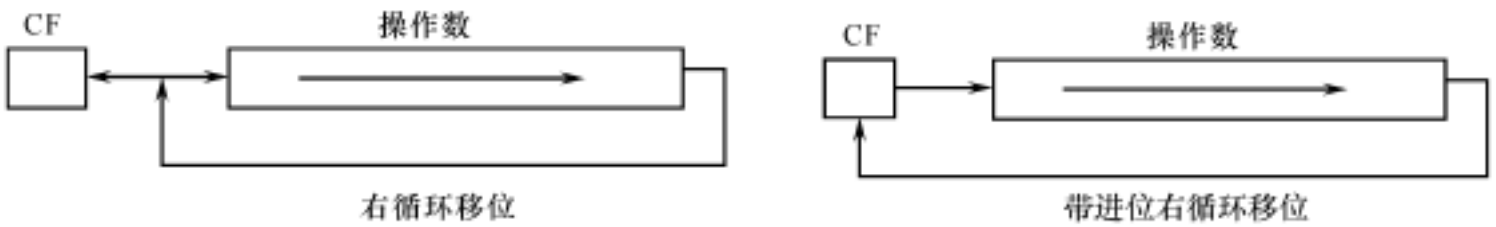


图 3 - 12　ROR、RCR 操作过程示意图

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	-	-	-	-	-	-	-	X

3.2.5　串操作指令

串操作指令是惟一可以实现内存单元之间的数据传送指令,能对存储器中的一个或多个长度为字节或字的字符串进行各种基本的操作。

1. 串传送指令(MOVS/MOVSB/MOVSW)

指令格式:　OVSB　　; 传送字节串

MOVSW　　; 传送字串

MOVS　　OP1 , OP2　　; 传送字节串或字串,取决于 OP1 和 OP2 的长度

功能描述: DS: SI 指定的源串中的一个字节或字复制到 ES: [DI] 指定的目的串中,然后根据方向标志 DF 的值自动修改地址指针 SI 和 DI。

【例 3. 15】　要将数据段中首地址为 BUFFER1 的 200 个字节传送到附加段首地址为 BUFFER2 的内存区中,使用字节串传送指令的程序段如下:

```
BUFFER    DB    12H,34H,56H,…  ;200 个字节
BUFFER2   DB    200DUP(?)
          …
          LEA   SI,BUFFER1
          LEA   DI,BUFFER2
          MOV   CX,200
          CLD
NEXT:     MOVSB
          DEC CX
          JNZ NEXT
          HLT
```

2．**串装入指令**(LODS/LODSB/LODSW/LODSD)

指令格式:　ODSB　　;**装入字节串**

　　LODSW　　;**装入字串**

　　LODS　　OP1　　;**装入字节串或字串,取决于** OP1 **长度**

功能描述:**将** DS: SI **指定的源串中的一个字节或字传送到累加器** AL **或** AX **中,同时修改源串指针** SI。

【例 3. 16**】　在内存中以** FIRST **为首地址的区域内存有一组字符串,试将其显示在屏幕上。在屏幕上显示一个字符的方法如下。**

```
MOV AH,20H          ;AH 存入 DOS 系统功能号
MOV DL, Y           ;DL 显示字符 Y
INT 21H             ;调用 DOS 的 21H 中断(在屏幕上显示)
```

程序段如下:

```
FIRST     DB    GOOD  AFTERNOON
          …
          LEA   SI,FIRST
          MOV   CX,COUNT
          CLD
          MOV   AH,02H
NEXT:     LODSB
          MOV   DL, AL
          INT   21H
          DEC   CX
          JNZ   NEXT
          HLT
```

3．**串存储指令**(STOS/STOSB/STOSW/STOSD)

指令格式:　TOSB　　;**存字节串**

　　STOSW　　;**存字串**

　　STOS　　OP1　　;**存字节串或字串,取决于** OP1 **长度**

功能描述:**把累加器** AL **或** AX **的内容传送到由** ES: DI **指定的字节或字串中去,然后根据方向标志** DF **的值自动修改地址指令** DI。

4. **串比较指令**(CMPS/CMPSB/CMPSW/CMPSD)

指令格式:　MPSB　　;**比较字节串**

CMPSW　　;**比较字串**

CMP　OP1, OP2　;**比较字节串或字串, 取决于** OP1 **和** OP2 **的长度**

功能描述: **将** DS: SI **指定的源串中的内容与** ES: DI **指定的目的串中的元素进行比较, 并根据其结果置标志位, 但不改变这两个存储单元中的内容, 同时自动修改源串指针和目的串指针。**

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	X

【例 3. 17**】　下列程序段用于比较两个字符串字符。找出其中第一个不相等的字符。如果两个字符串完全相同, 则转到** MATCH **进行处理。这两个字符串的长度为** 30, **首地址分别为** STRING1 **和** STRING2**。**

程序段如下:

```
STRING1    DB     ABCDEF…
STRING2    DB     DASDUHDLW…
             …
           LEA    SI, STRING1
           LEA    DI, STRING2
           MOV    CX, 30
           CLD
           REPE   CMPOSB
           JCXZ   MATCH
           DEC    SI
           HLT
MATCH:     MOV    SI, 0
           MOV    DI, 0
           HLT
```

5. **串搜索指令**(SCAS/SCASB/SCASW)

指令格式:　CASB　　;**扫描字节串**

SCASW　　;**扫描字串**

SCAS　OP1　;**扫描字节串或字串, 取决于** OP1 **长度**

功能描述: **将在一个字符串中搜索特定的关键字。字符串的起始地址只能放在** ES: DI **中, 不允许段超越。待搜索的关键字必须放在累加器** AL **或** AX **中, 它完成的操作是将累加器(** AL **或** AX) **的内容减去** ES: DI **指定的目的串元素, 并根据其结果置标志位, 同时自动修改目的串指针** DI, **但不改变累加器和目的串元素的内容。**

对标志位的影响如下。

OF	DF	IF	TF	SF	ZF		AF		PF		CF
X	-	-	-	X	X	-	X	-	X	-	X

【例 3.18】　在字符串中搜索关键字* 。

程序段如下:

```
BLOCK    DB       DSGFJD* H…
COUNT    EQU      $ - BLOCK
              …
         LEA      DI, OFFSET BLOCK
         MOV      CX, COUNT
         CLD
         MOV      AL, *
         REPNE SCASB
         JE MATCH
              …
MATCH:   HLT
```

6. 重复前缀指令(REP)

指令格式: REP

功能描述: 重复其后的字符串操作指令, 重复的次数由 CX 来决定。重复前缀指令的执行过程为, 判断 CX 是否为零, 若为零则结束重复操作, 执行该串操作指令的下一条指令, 否则, 对 CX 减 1, 继续执行其后的字符串操作指令, 直到 CX 等于零为止。

7. 相等重复前缀指令(REPE/REPZ)

指令格式: REPE/REPZ

功能描述: 重复其后的字符串操作指令, 每重复一次则 CX 减 1(不影响有关标志位), 重复操作的结束条件为 CX = 0 或 ZF = 1。

8. 不等重复前缀指令(REPNE/REPNZ)

指令格式: REPNE/REPNZ

功能描述: 重复其后的字符串操作指令, 每重复一次则 CX 减 1(不影响有关标志位), 重复操作的结束条件为 CX = 0 或 ZF = 0。

例如, 将例 3.15 使用重复前缀的字节串传送指令实现如下。

程序段如下:

```
BUFFER1    DB     12H,34H,56H, …     ;200 个字节
BUFFER2    DB     200 DUP(?)
           …
           LEA    SI, BUFFER1
           LEA    DI, BUFFER2
           MOV    CX,200
           CLD
           REP MOVSB
           …
```

3.2.6　控制转移指令

控制转移指令包括无条件转移指令(JMP)、条件转移指令(JCC)、过程调用/返回指令(CALL/RET)、循环控制指令(LOOP)和中断指令(INT)5 类。它们的共同特点是可以改变

程序的正常执行顺序,使之发生转移。而要改变程序的执行顺序,本质上就是要改变 CS: IP 的内容。

3.2.7　转移指令

转移指令是汇编语言程序员经常使用的一组指令。在高级语言中,时常有“尽量不要使用转移语句”的说明,但如果在汇编语言的程序中也尽量不用转移语句,那么该程序要么无法编写,要么没有多少功能。所以,在汇编语言中,不但要使用转移指令,而且还要灵活运用,因为指令系统中有大量的转移指令。

1. 无条件转移指令(JMP)

指令格式: JMP OP1

功能描述: JMP 指令是从程序当前执行的地方无条件转移到另一个地方执行。这种转移可以是一个短转移,偏移量在(- 128 ~127) 范围内,或近转移,偏移量在(- 32 KB ~ 32 KB) 范围内,或远转移(在不同的代码段之间转移) 。短转移和近转移是段内转移, JMP 指令只把目标指令位置的偏移量赋值给指令指针寄存器 IP,从而实现转移功能。但远转移是段间转移, JMP 指令不仅会改变指令指针寄存器 IP 的值,而且还会改变代码段寄存器 CS 的值。其中, OP1 是要转向的目标地址,可以是 16 位或 32 位的积寄存器或存储器,也可以是标号名或过程名。

例如:

```
        LOP1: …
              JMP     LOP1      ; 向前转移,偏移量之差为负数
              …
              JMP     LOP2      ;向后转移,偏移量之差为正数
              …
        LOP2: …
```

又例如:

```
        …
        MOV     AL, 80H
        MOV     BL, 72H
        ADD     AL, BL
        JO      NEXT1
        JMP     NEXT2
        …
        NEXT1: …
               …
        NEXT2: …
```

2. 条件转移指令(JCC)

指令格式:　JCC　OP1

功能描述: 根据 CPU 的标志状态组成的条件 CC,决定程序的执行方向。若条件 CC 成立,则控制程序转移到 OP1(标号) 所给出的转移目标地址;若不成立,程序将顺序执行。条件转移指令共有 18 条,它们可分为检测单个标志无符号数和有符号数的条件转移 3 类,如

表 3 - 3 所示。

表 3 - 3　条件转移指令表

	指令助记符	转移条件	说　明
检测单个标志	JC	CF = 1	有进位转移(与 JB/JNAE 重叠)
	JNC	CF = 0	无进位转移(与 JAE /JNB 重叠)
	JO	OF = 1	溢出转移
	JNO	OF = 0	无溢出转移
	JP / JPE	PF = 1	校验为偶转移
	JNP / JPO	PF = 0	校验为奇转移
	JS	SF = 1	为负数转移
	JNS	SF = 0	为正数转移
无符号数比较	JA / JNBE	CF = ZF = 0	高于 / 不低于或等于转移
	JAE / JNB	CF = 0	高于或等于 / 不低于转移
	JB / JNAE	CF = 1	低于 / 不高于或等于转移
	JBE / JNA	CF = 1 或 ZF = 1	低于或等于 / 不高于转移
	JE / JZ	ZF = 1	等于 / 为零转移
	JNE / JNZ	ZF = 0	不等于 / 非零转移
带符号数比较	JG / JNLE	ZF = 1 且 SF = OF	大于 / 不小于或等于转移
	JGE / JNL	SF = OF	大于或等于 / 不小于转移
	JL / JNGE	SF ≠ OF	小于 / 不大于或等于转移
	JLE / JNG	ZF = 1 或 SF ≠ OF	小于或等于 / 不大于转移

【例 3. 19】　比较无符号数大小, 将较大数存放在 BX 中。

程序段如下:

```
        CMP     BX, AX
        JNB     NEXT
        XCHG    BX, AX
NEXT: ...
```

【例 3. 20】　统计 X 字单元中 1 的个数, 将统计结果保存在 COUNT 字节单元中。

程序段如下:

```
        XOR     CL, CL
        MOV     AX, X
NEXT:
        TEST    AX, 0FFFFH
        JZ      DONE
        SHL     AX, 1
        JNC     NEXT
        INC     CX
        JMP     NEXT
DONE:
        MOV     COUNT, CL
        …
```

3. **循环控制指令**(LOOP / LOOPCC)

指令格式:　OOP　lable

LOOPE / LOOPZ　lable

LOOPNE / LOOPNZ　lable

功能描述: 以 CX 寄存器作为计数器, 指令的操作将 CX 的内容减 1, 结果不等于零, 则转到指令中指定的短标号处; 否则, 顺序执行下一条指令。

【例 3. 21**】**　有一串 N 个字符的字符串存放在首地址为 ASCII_STR 的存储区中。要求在字符串中查找字符 A, 找到则继续执行, 如果未找到则转到 NOFOUND 去执行。

程序段如下:

```
        MOV       CX, N
        MOV       SI , - 1
        MOV       AL, A
NEXT:   INC       SI
        CMP       AL, ASCII _STR[ SI ]
        LOOPNE    NEXT
        JNZ       NOFOUND
        …
NOFOUNDHLT
```

在程序执行过程中, 有两种可能性, 一种是找到了 A, 此时, ZF = 1, 因此提前结束循环, 在执行 JNZ 指令时, 因不满足测试条件而按顺序继续执行; 另一种是未查找到字符 A, 此时 ZF = 0, 因而转移到 NOFOUND 去执行。

3.2.8　调用和返回指令

1. **过程调用 / 返回指令**(CALL / RET)

调用指令: CALL　OP1　lable / proc

功能描述: CALL 指令用于调用一个目标地址为 OP1 的子程序(或过程) 。其具体操作是先将下一条指令的地址(即断点) 保存在栈堆中, 然后控制程序转移到 OP1 指定的存储单元。段内调用时, 将当前的 IP 压栈, 将目标地址送给 IP; 跨段调用时, 当前的 CS 和 IP 压栈, 将目标地址的段基址和偏移量送给 CS 和 IP。

返回指令: RET [imm]

功能描述: 返回指令 RET 用于从被调用的子程序返回调用程序。RET 指令通过弹出堆栈操作恢复调用前的 IP 值和 CS 值。如果调用程序向被调子程序通过堆栈方式传递参数, 可直接使用无操作数的 RET 形式; 否则应采用带操作数的 RET 形式 RET imm。

返回时堆栈指针 SP 先自动加上 imm 值(以便丢弃已使用过的传递参数) , 再据此弹出返回地址至 CS 和 IP, imm 值为参数个数的 2 倍。

2. **中断指令**(INT / INTO / IRET)

指令格式:　NT　　imm8

INTO

IRET

功能描述: INT imm8 为软中断指令, 用于产生一个由 8 位立即数指定中断号的软中断。

INTO 为溢出中断指令,它实际上是软中断指令 INT 的特例,其中断号隐含为 4,所以 INTO 与 INT 4 等价,它只有当 OF 置 1 时才产生中断。IRET 为中断返回指令,用于从中断服务程序返回原程序,它执行的操作是从堆栈中弹出原入栈保护的 IP、CS 和标志寄存器值,并重新开始被中断程序的执行。

3.3　处理器控制指令

处理器控制指令用于对 CPU 进行控制,例如,对 CPU 中某些标志位的状态进行操作,以便使 CPU 暂停、等待等。

3.3.1　标志操作指令

这类指令共有 7 条,其中 3 条针对进位标志 CF,两条针对方向标志 DF,另外两条针对中断标志 IF。它们都是零操作数指令,其指令格式及功能如表 3 - 4 所示。

表 3 - 4　标志位操作指令及功能

指令格式	功　能	说　明
CLC	清进位标志(CF←0)	CF = 0
CTC	置进位标志(CF←1)	CF = 1
CMC	进位标志取反(CF←$\overline{CF}$)	CF = $\overline{CF}$
CLD	清方向标志 (DF←0)	DF = 0 使所有串指令的标志指针为增量
STD	置方向标志(DF←1)	DF = 1 使所有串指令的标志指针为减量
CLI	清中断允许标志(IF←0)	IF = 0 表示禁止可屏蔽中断(关中断)
STI	置中断允许标志(IF←1)	IF = 0 表示允许可屏蔽中断(开中断)

3.3.2　其他控制指令

1．HLT 处理器暂停指令

指令格式: HLT

功能描述: HLT 指令的功能是使程序停止执行,处理器进入暂停状态。

2．WAIT 处理器等待指令

指令格式: WAIT

功能描述: WAIT 指令的作用是使处理器处于等待状态,直到出现外部中断为止。

3．LOCK 总线锁定前缀

指令格式: LOCK

功能描述: LOCK 是一字节的指令前缀,用于产生有效的 LOCK 总线信号,锁住由一条指令目的操作数指定的存储器区域,使之在指令执行期间一直受到保护,而防止其他主控器的访问。这在采用共享的多处理机系统中是十分有用的。

4．空操作指令: NOP

功能描述: NOP 指令的作用是完成一次空操作。它仅影响(E) IP 寄存器,对标志位无影响。

习　　题

1. 什么是寻址方式,8086/8088 微处理器有哪几种寻址方式,各类寻址方式的基本特征是什么?

2. 对于 8086/8088 处理器,存储器寻址的有效地址 EA 和物理地址 PA 的区别是什么?

3. 有效地址 EA 由哪 4 个元素组成,它们可优化组合出哪些存储器寻址方式?

4. 假定(DS) = 1000H,(SI) = 007FH,(BX) = 0040H,(BP) = 0016H,变量 TABLE 的偏移地址为 0100H。试指出下列指令的源操作数字段的寻址方式,它的有效地址(EA)和物理地址(PA)分别是多少?

(1) MOV AX,[1234H]　　(2) MOV AX,TABLE

(3) MOV AX,[BX+100H]　　(4) MOV AX,TABLE[BP][SI]

5. 识别下列指令正确与否,对错误指令,说明出错的原因。

(1) MOV DS,100

(2) MOV [1000H],23H

(3) MOV [1000H],[2000H]

(4) MOV DATA,1133H

(5) MOV 1020H,DX

(6) MOV AX,[0100H+BX+BP]

(7) MOV CS,AX

(8) PUSH AL

(9) PUSH WORD PTR[SI]

(10) IN AL,80H

6. 执行下列 3 条指令后 AL 寄存器的值是多少?

```
MOV AL,58H
ADD AL,64H
DAA
```

7. 执行指令 ADD AL,72H 前,(AL) = 8EH,标志寄存器的状态标志 OF,SF,ZF,AF,PF 和 CF 全为 0。指出该指令执行后各状态标志的值。

8. 编写一段程序,比较 EAX,EAX,ECX 中带符号数的大小,将最大的数放在 EAX 中。

9. 两个 4 位压缩 BCD 码定义如下:

```
X    DW    3526H
Y    DW    1234H
```

试编写计算 X+Y 的程序段。

10. 编写一程序段把 FIRST 和 SECOND 中的 30 个字节数分别相加,结果存放到 THIRD 中。

(1) 假定数据为无符号数,如果结果大于 255 则结果为 255。

(2) 假定数据为带符号数,如果有溢出则保存结果为 0。

11. 字符串 STR1 保存着 100 个字节的 ASCII 码,试编写一个程序统计该字符串中空格(20H)的个数。

12. 写出下列程序段执行的结果,(AL) = ?　(DL) = ?　并指出此程序断完成的是什么功能。

```
MOV     CL,4
MOV     AL,87
MOV     DL,87
AND     AL,0FH
OR      AL,30H
SHR     DL,CL
```

```
OR      DL,30H
```

13. 写出能代替下列重复串操作指令完成同样功能的指令序列。

(1) REP MOVSW　　(2) REP CMPSB

(3) REP SCASB　　(4) REP LODSW

(5) REP STOSB

14. 欲将数据段中自 AREA1 开始的 100 个字数据搬到附加段中以 AREA2 开始的区中，用下面 3 种传送指令编写。

(1) MOV 指令　　(2) 基本串指令　　(3) 重复串指令

15. 试分析下列两个程序段执行的功能。

(1)
```
CLD
LEA     DI,[0404H]
MOV     CX,0080H
XOR     AX,AX
REP     STOSW
```

(2)
```
MOV     CX,10
LEA     SI,First
LEA     DI,Second
REP     MOVSB
```

16. 已知内存中起始地址为 BLOCK 的数据块(字节数为 COUNT)的字节数据有正有负。试编写一个程序，将其中的正、负分开，分别送至同一段中的两个缓冲区，设正、负缓冲区的首地址分别为 PLUS 和 MINUS。

第 4 章　8086 汇编语言

汇编语言源程序的编写格式及使用方法有许多规定，一个汇编语言源程序除了程序主体之外，还有伪指令、宏指令等，它们也是汇编语言源程序的组成部分。汇编语言程序从编写完成到投入运行，一般要在多种系统软件的支持下，经过编辑（如行编辑 EDLIN、全屏幕编辑 WORDSTAR、WPS 等）、汇编（如 MASM）、连接（如 LINK）、装入和调试（如 DEBUG）等处理过程。本章重点讨论汇编语言源程序的编写格式与调试过程。

4.1　汇编语言源程序格式

汇编语言程序设计与其他语言程序设计一样，需要多种系统软件的支持，要经过编辑、汇编、调试之后才能运行，汇编语言的翻译器（汇编程序）对源程序有严格的格式要求，因此，汇编语言格式就是汇编语言必须遵循的语法规则。

用汇编语言编写的源程序，其结构上具有一些特点，首先，它是由若干逻辑段组成，有伪指令语句定义和说明；其次，整个源程序以 END 伪指令结束；第三，每个逻辑段由语句序列组成。语句可以是如下几种形式。

① 指令语句：对应于 CPU 指令系统中的一条指令，因此是可执行语句，汇编时译成目标码。

② 伪指令语句：CPU 不执行的语句，只是汇编时给汇编程序提供汇编信息，并不产生目标代码。

③ 宏指令语句：实际上是一个指令序列，汇编时产生对应的目标代码序列。

④ 注释语句：以分号“；”开始的说明性语句，汇编程序不予处理，只起注释作用，使程序易于理解。

⑤ 空行语句：为保持程序书写清晰，仅包含回车换行符的语句行。

下面给出的是一个标准的，以 MASM 为基础的单模块汇编语言源程序的结构形式。

【例 4.1】　汇编语言源程序结构形式。

```
        .8086
        .model small
        .data
        ...             (数据定义伪指令序列)
        .stack
        ...             (数据定义伪指令序列)
        .code
        assume cs:code,ss:stack,ds:data
start:  ov      ax,@data
        mov     ds,ax
        ...(核心程序段)
        mov     ah,4ch          ;返回操作系统
```

```
int     21h
end     start
```

汇编语言程序由语句序列构成,汇编语言程序中的每条语句一般占一行,每行不超过 132 个字符(MASM6.0 以上版本可以是 512 个字符) ,语句一般是由分隔符分成的 4 个部分组成,它们有两种格式。

① 执行性语句:由硬指令构成的语句,它通常对应一条机器指令,出现在汇编语言源程序的代码段中,具体格式如下:

标号:助记符　操作数,操作数;注释

其中,标号反映该指令的逻辑地址,它为分支、循环等指令提供转移的目的地址。

② 说明性语句:由伪指令构成的语句,它通常指示汇编程序如何汇编源程序,具体格式如下:

名字 定义符　参数,参数,　;注释

其中,名字可以为变量名、段名、子程序名或宏名等,既反映逻辑地址又具有自身的各种属性。

4.2　伪指令语句

4.2.1　程序结构伪指令语句

这类语句是与程序结构紧密相关的基本伪指令语句,用于说明 CPU 的类型、段结构、源程序(或模块) 起止信息和段内存的安排等。

4.2.1.1　方式选择伪指令

由于汇编语言是在 8086/8088 的基础上发展起来的,因此汇编程序为区分当前的源程序是对于哪种 CPU 执行的,提供了处理器方式选择伪指令。方式选择伪指令的格式和功能如表 4 - 1 所示。

表 4 - 1　方式选择伪指令的格式和功能

伪指令格式	功　能
.8086	默认方式,告诉汇编程序只接受 8086/8088 指令
.286/.286C	告诉汇编程序只接受 8086/8088 及 80286 非保护方式(即实地址方式) 下的指令,用.8086 可删除该伪指令
.286P	允许汇编程序接受 8086/8088 及 80286 的所有指令(包括保护方式和非保护方式下的指令)
.386/.386C	允许汇编 8086/8088 及非保护方式下的 80286/80386 指令。与.286/.286C 类似,在此方式下禁止所有保护方式下的指令出现
.386P	除具有.386/.386C 功能外,还允许汇编保护方式下的 80286/80386 指令
.8087	选 8087 指令集并指定实数的二进制码为 IEEE 格式
.287	选 80287 指令集并指定实数的二进制码为 IEEE 格式
.387	选 80387 指令集并指定实数的二进制码为 IEEE 格式
.486/486C	与.386/.386C 类似,允许汇编 80486 非保护方式下的指令,MASM6.0 可用
.486P	与.386P 类似,允许汇编 80486 的全部指令,MASM6.0 可用

4.2.1.2　逻辑段定义伪指令

8086 按照逻辑段组织程序，具有代码段、数据段、附加段和堆栈段。因此，一个汇编语言源程序可以包括若干个代码段、数据段或堆栈段，段与段之间的顺序可随意排列。需独立运行的程序必须包含一个代码段并指示程序执行的起点，一个程序只有一个起点，所有的可执行性语句必须位于某一个代码段内。说明性语句可有两种逻辑段定义，完整段定义和简化段定义。

1. 完整段定义伪指令

采用完整段定义伪指令可具体控制汇编程序和连接程序在内存中组织代码和数据的方式。它包括以下 3 种伪指令语句。

(1) 段定义语句

格式：段名 SEGMENT[定位类型] [组合类型] [字长选择] [类别]

…

(段体)

…

段名　ENDS

功能：指出段名及段的各种属性并表示段的开始和结束位置。

说明：段名是用户定义的段的标识符，用于指明段的基址。SEGMENT 后面有 4 个可选参数，代表段的 4 种属性，分别说明如下。

定位类型：用于制定该段地址的 5 种可选类型，如表 4 - 2 所示。

表 4 - 2　定位类型

定位类型	含义
BYTE(字节)	段起始地址可任意
WORD(字)	段起始地址必须为偶数，即该地址的 D0 位应为 0
DWORD(双字)	段起始地址必须为 4 的倍数，即该地址的 D 和 D0 位应为 0
RARA(节)	一般用于 80386 的 32 位段中
PAEG(页)	段起始地址必须为 16 的倍数，即该地址的 D3 到 D。位应为 0

组合类型：指定多个逻辑段之间的关系。用于告诉 LINK 程序本段与其他模块中同名段的组合连接关系。主要有 4 种可选组合类型，如表 4 - 3 所示。若此属性是默认的，表示段是独立的。

字长选择：用于定义段中使用的偏移地址和寄存器的字长，有两种字长选择。

USE16：表示该段字长为 16 位，按 16 位方式寻址，最大段长为 64 KB。

USE32：表示该段字长为 32 位，按 32 位方式寻址，最大段长为 4 GB。

类别：用于控制段的存放次序。它可以是任何合法的名称，但必须用引号括起来。若“ 类别 ”选择项是默认的，则表明该段类别为空。

(2) 段寄存器说明语句

格式：ASSUME 段寄存器：段名 / 组名[, 段寄存器：段名 / 组名…]

功能：说明源程序中定义的段或组由哪个段寄存器去寻址。段寄存器可以为 CS, SS,

DS, ES, FS 或 GS。

表 4 - 3　组合类型

组合类型	含　义
PUBLIC	将不同模块中相同的段连接到同一物理段中，使它们公用一个段地址
STACK	与 PUBLIC 同样处理，只是连接后的段为堆栈段
COMMON	该段可与其他同名且同类型的段重叠在一起，而且长度等于最长的段的长度
AT 表达式	将具有 AT 类型的段装在表达式所指定的地址边界上

(3) 组定义语句

格式：组名 GROUP 段名[，段名，…]

功能：将 GROUP 定义符后指定的所有段分配在一个 64 KB 的物理段中，并赋予该段一个名字——组名。

说明：组名是用户定义的名字，指出组地址的一种符号必须是惟一的，不能与任何标号、段名及变量等同名。段名可以是用 SEGMENT 语句定义或者由 SEG 运算符得到的段名。组定义语句不影响各段的次序，因此组内各段不一定连续存放，但都必须包含在 64 KB 中。如果组名在 ASSUME 语句中已经说明，且相应段寄存器(DS, ES, FS 或 CS)有初始化语句，则系统把组内各段中的变量或标号的偏移地址调整为相对于组的起始地址。当程序结构需要多个逻辑段时，使用本语句可节省段寄存器的使用。实际应用中最好是代码段为一组，堆栈段为一组，数据段为一组或两组，但组中各段所占的内存总量不能超过 64 KB。

【例 4.2】　关于组定义，有下列程序段。

```
dgroup    group    aseg,cseg
          assum    ds:dgroup,cs:text
aseg      segment  word public  data
          x  dw  ?
aseg      ends
bseg      segment  word public  data
          y  dw?
bseg      ends
cseg      segment  word public  data
          z  dw?
cseg      ends
text      segment  word public  code
start     mov      ds,ax
          ...
          ...
text      ends
          end      start
```

该程序段被汇编、连接之后，装入内存时所在位置如图 4 - 1 所示。从图 4 - 1 中可以看出，虽然 ASEG 与 CSEG 点在内存中的位置不同，但由于它们被 GROUP 伪指令说明在同一组，因此它们使用相同的段起始地址(即 ASEG 段的起始地址)。而 BSEG 与 TEXT 段不是组中的一部分，因此它们各有自己的段起始地址。变量 Y 的偏移地址是从 BSEG 段起始地

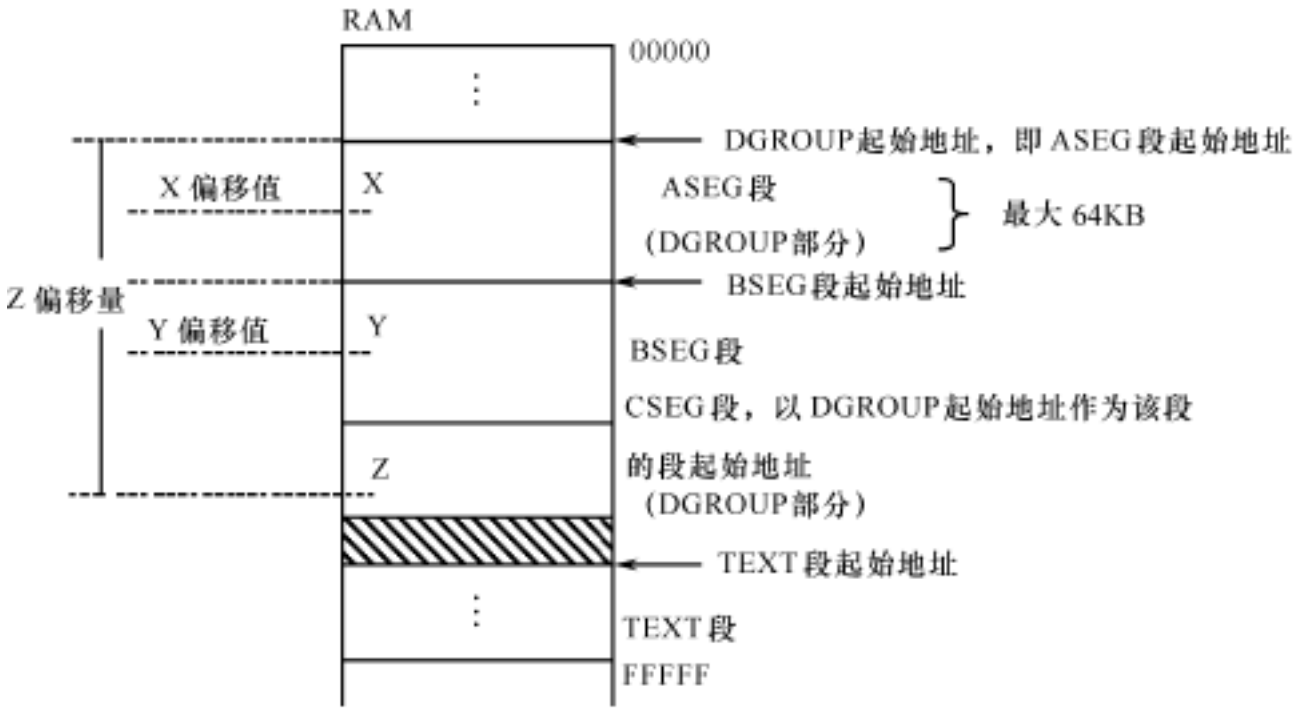


图 4 - 1　组内段结构

址算起的。该图只是说明 LINK 程序组织内各段的方法, 并不代表组的标准方法。

2. 简化段定义的伪指令

简化段有利于实现汇编语言程序模块与 Microsoft 高级语言程序模块的连接, 它可以由操作系统自动安排段序, 自动保证名字定义的一致性。

(1) 段次序语句(DOSSEG)

格式: DOSSEG

功能: 各段在内存的顺序按 DOS 段次序约定排列。

说明: 各段在内存的次序决定于很多因素, 多数程序对段次序无明确要求, 可由操作系统安排。本语句用于主模块前面, 其他模块不必使用。

(2) 内存模式语句(. MODEL)

格式: . MODEL 模式类型[, 高级语言]

功能: [高级语言] 是可选项, 可使用 C, BASIC, FORTRAN, PASCAL 等关键字来指定高级程序设计语言接口, 本语句一般放在段定义语句之前, 模式类型如下。

SMALL: 小模式, 数据、代码各存入一个物理段中, 均为近程。

MEDIUM: 中模式, 数据为近程, 代码允许为远程。

COMPACT: 压缩模式, 数据为近程, 代码允许为远程, 但任一数据段不可超过 64 KB。

LARGE: 大模式, 数据与代码均允许为远程, 但任一数据段不可超过 64 KB。

HUGE: 巨型模式, 数据与代码均允许为远程, 但任一数据段可超过 64 KB。

注意: 当独立的汇编语言源程序不与高级语言程序连接时, 多数情况下只用小模式即可。

(3) 段语句

简化段定义的段语句通常有以下几种形式。

CODE[名字] : 定义一个代码段, 若有多个代码段, 要用名字区别。

STACK[长度] : 定义一个堆栈段, (SP) = 长度, 若省略长度, 则(SP) = 1024。

DATA: 定义一个近程数据段, 其数据空间要赋初值。

DATA?: 定义一个近程数据段,其数据空间不赋初值。

CONST: 定义一个近程常数段。

FARDATA[名字] : 定义一个远程数据段,其数据空间要赋初值。

FARDATA? [名字] : 定义一个远程数据段,其数据空间不赋初值。

对于简化段定义,有以下两点需要说明。

① 凡是与高级语言程序连接的数据,必须把常数与变量分开,变量中又要把需赋初值的分开,并分别定义在. CONST, . DATA, . FARDATA, . DATA? 和. FARDATA? 中。远程数据段只能在压缩模式、大模式和巨型模式中使用,其他数据段和代码段可在任何模式下使用。

② 独立的汇编语言源程序(即不与高级语言连接的源程序) 只用 DOSSEG, . MODEL, . CODE, . STACK 和. DATA 五种简化语句,并不区分常数与变量以及赋初值与不赋初值。在. DATA 语句定义的段中,所有数据语句均可使用。

【例 4. 3】　简化段定义的一般格式。

```
dosseg
.model small
stack
.data
...
数据语句
...
.code
启动标号: mov ax, dgroup; 或 mov ax, @data
          mov ds, ax
          ...
          执行性语句
          ...
          end 启动标号
```

这种简化段的源程序结构中只有一个堆栈段、一个数据段和一个代码段。代码段长度可达 64 KB, 数据段与堆栈段为一个组, 其总长度可达 64 KB, 组名 DGROUP 和数据名@DATA 都代表组对应物理段的首地址,装入内存时系统给 CS 和 IP 赋初值使其指向代码段,同时系统还给 SS 和 SP 赋初值, 使(SS) = DGROUP, (SP) = 数据段长度 + 堆栈段长度, 从而使堆栈段为对应的物理段, 这样处理使堆栈元素也能用 DS 寄存器访问, 以便同高级语言程序连接。在代码段开始运行处(启动标号处) , 用户应设置 DS 指向组的段首地址。

3. 指定地址伪指令(ORG)

格式:　RG 偏移地址

ORG $ + 偏移地址

功能: 该伪指令以其指定的偏移地址或由$给出的当前地址加上指定的偏移地址作为当前开始分配和使用的偏移地址。

说明: 该伪指令语句不占内存, 它指定下一个占内存语句的偏移地址。偏移地址可写成表达式形式, 但其取值范围在 0 ~65535 之间。通常不必使用该语句, 只在需要指定存储空

间或保留一段存储空间时才使用它。该语句不能使用标号,否则语句无效。

4. 模块定义伪指令

一个可执行的汇编源程序可由多个模块组成,每个模块是一个独立的汇编单位。在操作系统中,汇编程序是一个* .ASM 源文件。汇编源程序的模块与汇编源文件是一一对应的。

(1) 模块开始语句(NAME)

格式: NAME[模块名]

功能: 表示源程序开始并指出模块名。

说明: 本语句一般可省略。省略时,模块名取源程序中 TITLE 语句的页标题的前 6 个字符。若没有 TITLE 语句,则取该模块的源程序文件名为模块名。

(2) 模块结束语句(END)

格式: END[标号/过程名]

功能: 模块结束语句表示源程序到此结束,并可指出程序的启动地址。

说明: 一个源程序必须有且只有一个 END 语句来指明源程序文件的结束。方括号中的标号或过程名是可选项,只有主模块才具有该项,其作用是指出该程序的第一条可执行指令的位置(为此,系统将给 CS, IP 赋初值) 。

5. 模块连接伪指令

模块连接伪指令又叫外部引用伪指令。该类伪指令用于解决多模块连接问题,实现多模块之间数据和过程的共享。这里介绍 4 种该类伪指令语句。

(1) 全局符说明语句 (PUBLIC)

格式: PUBLIC 符号名 1[,符号名 2, …]

功能: 将本文件中定义的符号名说明为全局符号,允许程序中其他模块使用。

说明: 本语句中的符号名可以是标号、变量名、过程名或由 EQU(或 =) 伪指令定义的名字。这些符号名必须是在当前源程序中定义的,其他模块不能再用它们去定义别的内容。未被说明的符号名不能被其他模块引用。需要注意的是, PUBLIC 伪指令与 SEGMENT 伪指令中的 PUBLIC 组合类型是两个不同的概念。

(2) 外部符说明语句(EXTRN)

格式: EXTRN 符号名 1: 类型[,符号名 2: 类型, …]

功能: 本语句指定的符号名是在其他模块中 PUBLIC 伪指令语句定义过的。类型可以是 NEAR, FAR, BYTE, WORD, DWORD, FWORD, QWORD, TBYTE 或 ABS(由 EQU 伪指令定义的常数符) ,具体类型必须与其他模块中定义的相同符号名的类型一致。

(3) 包含说明语句(INCLUDE)

格式: INCLUDE 文件名

功能: 将指定文件的内容完整地插入到本语句出现的位置。

说明: 使用本伪指令语句可避免重复书写多个模块都要使用的相同程序块。当 MASM 汇编某一源程序文件时,若遇到 INCLUDE 伪指令,就按文件说明打开磁盘上存在的相应文件,并将它插入到当前文件的该 INCLUDE 伪指令处,然后汇编插入进来的文件中的语句。当该文件中的所有语句处理完后, MASM 继续处理当前文件中 INCLUDE 语句后面的语句。

汇编语言程序设计时,常用 INCLUDE 伪指令语句将一个标准的宏定义插入到程序中。

(4) 公用符说明语句(COMM)

格式: COMM [NEAR/FAR] 符号名:尺寸[:元素数], …

功能: 将语句中的符号名说明为公用符号。公用符号既是全局的又是外部非初始化的。

说明: 语句中的符号名可以是近程(NEAR)或远程(FAR)数据段的符号, NEAR/FAR 默认时, 在完整段和简化段的中、小模式下为 NEAR, 其他模式下为 FAR; 尺寸可以是 BYTE, DWORD, QWORD 及 TBYTE; 元素数为符号的个数, 默认值为 1。一条语句中可说明多个公用符, 多个符号间用逗号隔开。本伪指令语句经常用在 INCLUDE 文件中将它说明为公用变量, 然后在每一个模块中都用 INCLUDE 指令嵌入这个 INCLUDE 文件。

4.2.1.3　符号/数据/标号伪指令语句

1. 符号定义伪指令

符号定义伪指令可用来为表达式赋予一个符号名。表达式可以是常数、变量、标号、指令语句和字符等。在程序中, 任何需要这种表达式的地方都可使用被赋予的符号名来代替它。符号定义伪指令语句有两种, 即等值语句和等号语句。

(1) 等值语句(EQU)

格式: 符号名 EQU 表达式

功能: 用符号名代替表达式的值。

说明: 在同一源程序中, EQU 语句不能重复定义同一符号名。

例如:

```
X   EQU    80
Y   EQU    X+20
```

(2) 等号语句(=)

格式: 符号名 = 表达式

功能: 等号语句的功能与 EQU 语句的功能相同, 在同一源程序中能重复定义同一符号名。

例如:

```
P1 = 55H
P1 = P1 + 90H
```

2. 数据定义伪指令

格式: [符号名] DB/DW/DD/DF/DQ/DT 初值序列

功能: 为数据项分配一个或多个字节、字、双字、长字、四字、十字节的存储空间, 给它们赋初值, 并用一个符号名与之相联系。

说明

本伪指令按数据长度可分为 DB, DW, DD, DF, DQ, DT 6 种类型, 分别定义 8 位、16 位、32 位、48 位、64 位、80 位数据。

与数据项相联系的符号名称为变量。经过定义的变量名有 3 个属性, 数据类型(字节、字、双字、长字、四字、十字节)、偏移量(可用 OFFSET 运算符获得)和段基址(可用 SEG 运算符求得)。符号名为可选项。

给变量赋初值可以赋确定的值, 也可以赋不确定的值(用“?”表示)。所谓赋不确定的值, 实质是不赋值, 而只预留规定长度的存储空间。初值序列可以是一个元素, 也可以是

用逗号分隔的多个元素,还可以是用 DUP 运算符建立单个值的多次复制;其中确定值可以是整数、浮点数(只允许 DD, DQ 和 DT 伪指令,并只用于 80486 和 80386/80287 协处理器上)、字符、字符串或表达式。例如:

```
x     db    12h,23h,34h
y     dw    89abh,0012h,4567h
z     db    ?
x1    dw    10h dup(0,5,6)
y1    dd    ?
z1    dt    1234h
w1    dq    5678h
```

3. 标号定义伪指令(LABEL)

格式:符号名 LABEL 类型

功能:为紧跟在本伪指令语句后的标号、操作码、过程或变量建立新的符号名,并刷新其类型属性。对标号、操作码或过程,其类型为 NEAR, FAR;对变量,其类型为 BYTE, WORD, DWORD, FWORD, QWORD 或 TBYTE。LABEL 伪指令提供了另一种定义标号或变量名的方法,但它并不为符号名分配存储空间。

例如:

```
SUBRF LABEL FAR          ;远调用入口
SUBRN:...                ;近调用入口
...
```

两个标号 SUBRF 与 SUBRN 均指向同一指令,但由于它们的类型不同(SUBRF 是 FAR,而 SUBRN 后有冒号,其类型隐含为 NEAR),所以可用不同的调用方法(远或近)来访问标号所指的程序段。其他代码段可通过远标号 SUBRF 来访问该程序段,当前代码段可通过近标号 SUBRN 来访问该程序段。例如:

```
BARRAY LABEL BYTE
ARRAY DW200 DUP(?)
...
MOV AL, BARRAY[99]       ;取数值的第 100 个字节值→AL
...
MOV AX, ARRAY[98]        ;取数组的第 99 个字值→AX
```

这里用两种方法定义了数组。由于 LABEL 伪指令不为变量分配存储空间,因此 BARRAY 与 ARRAY 的地址实际上是相同的,即它们指向同一个数组,只是 BARRAY 被定义为字节型,而 ARRAY 被定义为字型。这样,可根据需要的不同类型去存取数组中的数据。

4.2.1.4　结构性数据伪指令语句

一般数据语句只有几种固定的伪操作符,它们主要描述了几种计算型数据结构的形式。由于事物管理(如学籍、人事管理等)方面的数据结构比计算应用型的数据结构要复杂得多,为此,汇编语言提供了自定义数据结构的伪操作符,这些功能由结构型数据伪指令语句实现,常用的结构性数据伪指令包括以下 3 种。

1. 结构定义伪指令

把一个或多个数据定义语句结合在一起,分别以 STRUC 伪指令和 ENDS 伪指令作为起始语句和结束语句,便构成一个数据结构。定义了结构后,结构内各数据定义语句中的变量就成为结构变量,也叫结构字段名或域名。有关结构定义,有两种伪指令语句。

(1) 结构类型说明语句(STRUC/ENDS)

格式: 结构名 S RUC

…(结构体) ; 由数据定义语句构成

结构名 ENDS

功能: 定义一个结构的模板,但并不真正给结构分配存储空间,只说明结构的类型(包括结构名、域名及其数据类型) 。

说明: 保留字 STRUC 和 ENDS 是系统规定的,而结构名和结构体中的域名及长度、类型是由用户定义的。结构是多字节自定义数据结构,其最小域为字节,最大长度不限,可放在数据段前面。

(2) 结构变量说明与赋初值语句

格式: [变量名] 结构名〈[域值名] 〉

功能: 定义一个结构变量,并对其分配存储空间和赋初值。

说明: 本语句中的变量名是用户自定的,为任选项,默认时汇编程序照样分配存储空间; 结构名一定是曾经用 STRUC/ENDS 伪指令定义过的; 域值表用于给结构变量的各域赋初值,这些初值的类型、顺序必须与结构类型说明时的各域的类型、顺序一致,各初值间以逗号分隔。如果某域的初值与结构类型说明时的值相同,则相应位置可为空,但逗号不能省; 若所有域的初值都采用结构类型说明时的初值,则域值表可省略,只写一个尖括弧"〈 〉"即可。要注意,尖括弧在任何时候都不能省略。

【例 4.4】　描述学生基本情况的结构的定义和引用。

```
student     struc           ;定义数据结构
stu-no      dd 9703001h     ;学号
sname       db 王红
sex         db 0            ;性别:0,女;1,男
age         db 20           ;年龄
politica    db 团员         ;政治面貌
credit      dw 120          ;成绩
student     ends
dseg    segment     para public data
stu1    student     〈 〉          ;定义 6 个结构变量
stu2    student     〈9703002h, 张华 ,,,,108〉
stu3    student     〈9703010h, 孙涛 ,1,21, 群众 ,150〉
stu4    student     〈9703038h, 刘艳 ,,19, 群众 ,180〉
stu5    student     〈9703042h, 周松 ,1,, 党员 ,170〉
stu6    student     〈9703055h, 李丽 ,,, 党员 ,140〉
dseg    ends
code    segment     papr public code
        assume      cs:code,ds:dseg
start:    …
```

```
        mov ax,stu1.credit      ;比较王红和李丽的成绩
        cmp ax,stu6.credit
          …
code    ends
end     start
```

2. 记录定义伪指令

记录是与结构相似的一种数据组织工具,但两者不同的是,结构用于处理按字节计算的数据信息集合,而记录则用于处理按二进制位计算的数据信息集合。记录可用于定义一个字节、字或双字的记录变量集合,在这些集合中,同一字节、字或双字的不同位可能代表不同的意义,与结构定义相似,为了定义记录,也要使用两种伪指令语句。

(1) 记录类型说明语句(RECORD)

格式:记录名 RECORD 位域名:域宽 [= 表达式][, …]

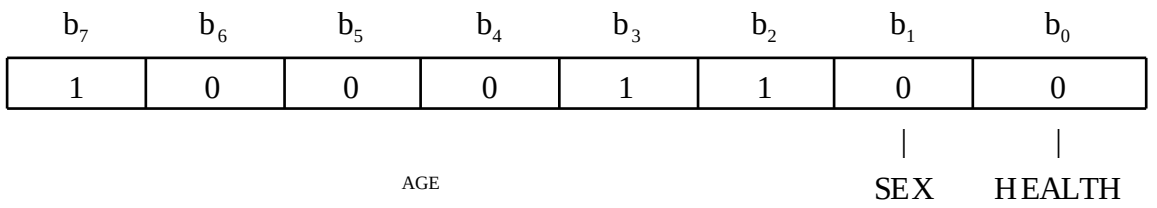
功能:定义位域集合的模板,但不实际分配存储空间。定义后可通过位域名单独访问。

说明:语句中,记录名和位域名是用户自定的,但不能与其他名相同;RECORD 是定义记录的伪指令;域宽表示相应位域的位数,但它必须是常数;表达式为任选项,若被选,则表示相应位域的初值,否则初值为 0。一个记录中可说明多个位域,相互间用逗号分隔。所有位域的宽度之和不能超过 32 位,小于等于 8 位时按字节处理;大于 8 位、小于等于 16 位时按字处理;大于 16 位、小于等于 32 位时按双字处理;凡不足 8 位、16 位、32 位时,则所有位域一律按其先后顺序从信息区的高位到低位排列,且向右对齐到字节、字或双字的最低有效位位置,高端补 0。

例如:

STATUS RECORD AGE:6 =35, SEX:1, HEALTH:1

该记录定义了一个占 1 个字节、有 3 个位域的记录类型。其中除 AGE 位域初值为 35 外,其余位域初值未指出,均被认为是 0。该记录各位域在 1 字节中的分配形式如下。



(2) 记录变量说明与赋初值语句

格式:[记录变量名] 记录名〈[域值表]〉

功能:定义一个记录变量并分配存储空间和赋初值。

说明:记录变量说明与赋初值语句同结构变量说明与赋初值语句相似,在此不再赘述。要特别说明的是,域值表为可选项,如省略时,其值为 0。

3. 联合定义伪指令

联合定义是 MASM 6.0 新增的一种结构性数据伪指令语句,它实质上是 STRUC 定义的一个补充,它与结构可同时使用,还可嵌套(对 MASM 6.0)。同结构数据语句一样,联合数据结构语句也必须先定义内存数据结构形式,然后按数据定义伪指令的使用方式,用它定义联合变量并赋初值。

(1) 联合类型说明语句(UNION/ENDS)

格式: 联　名 UNION

域 DB ?
域 DW ?
域 DD ?　} 联合体, 由数据定义语句构成
…

联合名 ENDS

功能: 定义一个联合的结构模块。

说明: UNION/ENDS 为系统规定的保留字, 联合名和域为用户定义的名字。每个域用一条数据伪指令定义, 一般不应有初值。从其格式可看出, UNION 与 STRUC 的定义类似, 只是域的偏移地址不同。STRUC 中的域是相对于结构顺序分配的; 而 UNION 中的域则是相对于联合重叠分配的, 其偏移地址均为 0, 所以域中不应有初值, 如果有初值, 汇编时只保留第一个域的初值。

例如, 下面是一个联合类型说明的例子。

```
UN  ANME        UNI ON
    XX          DB ?
    YY          DW?
    ZZ          DD ?
UNI ANME        ENDS
```

(2) 联合变量说明与赋初值语句

格式: 变量名 联合名〈域值表〉

功能: 定义一个联合变量, 分配存储空间并给它的各域变量赋初值。

说明: 联合变量及域值表的说明与结构变量的一样, 赋值规定也一致。

例如, 结合联合 UNIANME 的定义, 可定义下列联合变量。

```
VAR UNI ANME〈87654321〉
```

从而也定义了其中的各个域变量 VAR. XX, VAR. YY, VAR. ZZ, 并赋给了如下初值:

```
( VAR. XX) =21H
( VAR. YY) =2143H
( VAR. ZZ) =21436587H
```

要特别说明的是联合和结构均可以嵌套, 既可以结构套结构, 联合套联合, 又可以结构和联合相互嵌套。

【例 4. 5】　结构定义的嵌套, 即一个结构域中可有另一结构的变量。

```
FUN     STRUC
X       DW?
Y       DW200
FUN     ENDS
EXTR    STRUC
EXX     FUN
EYY     FUN<20,60>
MN      DB 5678
EXTR    ENDS
```

4.2.2　过程和宏定义伪指令语句

汇编语言中常用定义过程和宏的方法来实现模块管理程序代码的功能,因此过程和宏是进行模块化程序设计的基础。

1. 过程定义伪指令

过程是一段可由其他程序用 CALL 指令调用的程序,执行完后用 RET 指令从过程返回原调用处。过程相当于高级语言程序中的子程序,常用于代替完成特定任务的代码模块。

格式: 过程名　PROC [属性]　　;过程开始
　　...　　;过程体
　　[RET]
　　...
　　RET
　过程名　ENDP　　;过程结束

功能: 定义一个过程(子程序) 。

说明

① 过程名是过程的标识符(也可视为标号,当标号处理) 。

② 过程的属性可以是 NEAR 或 FAR。NEAR 类型为近,可在段内调用。FAR 类型为远,可跨段调用。默认时为 NEAR(近) 。如果使用简化段指令,则不需要用 NEAR/FAR 指定类型,这时过程的类型是由. MODEL 伪指令决定的。

③ RET 为返回指令,是过程的出口点,但不一定是过程的最后一条指令。一个过程可以有多条 RET 指令,但至少要执行到一条 RET 指令。

④ PROC/ENDP 必须成对出现。ENDP 指示过程结束,但它不会产生 HTL 或 RET 指令。

⑤ 过程可以嵌套,嵌套的深度(层数) 只受堆栈限制。过程和段也可以嵌套,但不能交叉覆盖。

【例 4.6】　下面程序段定义两个过程 P1 和 P2,其中 P1 又调用了 P2。

```
P1      PROC            ;定义过程 P1
CALL    P2              ;过程嵌套
        RET
P1      ENDP
        ...
P2      PROC            ;定义过程 P2
        ...
        RET
P2      ENDP
```

2. 宏定义伪指令

宏的概念与过程很相似,也是用一个宏名来代替源程序中经常需要用到的一个程序模块(代码段) ,宏定义语句格式与过程定义语句格式相似。

格式: 宏名　MACRO [形式参数表]
　　…　　;宏体
　　ENDM　　;宏定义结束

功能: 定义一个宏。

说明

① 宏名必须是惟一的, 它代表着所定义的宏体的内容, 在其后面的源程序中, 通过该名字来调用宏。

② 形式参数表是用逗号(或空格, 或制表符) 分隔的一个或多个形式参数。它是可选项。选用了形式参数时, 所定义的宏称为带参数的宏。当调用宏时, 需要对应的实际参数去取代, 以实现向宏中传递信息。

③ 宏体可以是汇编语言所允许的任意指令和伪指令语句序列, 它决定了宏的功能。在宏体中还可以定义或调用另一个宏(宏嵌套) 。

④ 宏一经定义, 就像为指令系统增加了新的指令一样, 在程序中就可像指令一样通过宏名对它进行任意次的调用, 故又称宏为宏指令或宏调用。要注意的是, 宏定义必须放在第一条调用它的指令之前, 一般都将它放在程序的开头。

【例 4.7】　以下程序段定义一个两参数相加并将结果送到第 3 个参数中的宏, 并调用它。

```
AD_UP    MACRO AD1, AD2, SUM     ; 定义一个带 3 个形参的宏
    MOV     AX, AD1
    MOV     AX, AD2
    ENDE
      …
ADDUP  BX, 24, DX       ; 宏调用, 用实际参数 BX、24 和 DX 分别取代形参 AD1 、
                        ; AD2 和 SUM
```

3. 宏和过程的比较

宏和过程都可用来简化源程序, 并可被程序多次调用, 从而使程序结构简洁清晰, 符合结构化程序设计风格。因此, 对于那些重复使用的程序模块, 既可用过程也可用宏来实现。

宏和过程的主要区别如下。

① 宏操作可以直接传递和接收参数, 它不需要通过堆栈等其他媒介来进行, 因此编程比较容易。而过程不能直接带有参数, 当过程之间需要传递参数时, 必须通过堆栈、寄存器和存储器来进行, 所以相对于宏而言, 它的编程要复杂一些。

② 宏调用只能简化源程序的书写, 缩短源程序的长度, 而并没有缩短目标代码的长度, 汇编程序处理宏指令时, 是把宏体插入到宏调用处, 所以目标程序占用内存空间并不因宏操作而减少。而过程(子程序) 调用却能缩短目标程序的长度, 因为过程在源程序的目标代码中只有一段, 无论主程序调用多少次, 除了增加 CALL 和 RET 指令的代码外, 并不增加子程序代码段。

③ 引入宏操作并不会在执行目标代码时增加额外的时间开销。相反, 过程调用由于需要保护和恢复现场及断点, 因而有额外的时间开销, 会延长目标程序的执行时间。

鉴于以上比较, 在程序设计中, 当执行速度比内存容量更为重要时, 或者要调用的例程较短且调用的次数不太频繁时, 适于选用宏调用技术; 反之, 则选用过程调用更合适。

4.2.3　条件汇编伪指令语言

条件汇编伪指令可使汇编程序根据某种条件对某部分源程序有选择地进行汇编。它在

形式上和高级语言中的条件语句类似,但实质上不同。条件汇编语句是一种说明性语句,其功能由汇编系统实现;而一般高级语言的条件语句是执行性语句,其功能由目标程序实现。条件汇编语句通常在宏定义中使用,使得宏定义的使用范围更广。一般情况下,使用条件汇编语句可使一个源文件产生几个不同的源程序,它们可有不同的功能。MASM 中共有 5 组条件汇编开始语句、1 种条件汇编结束语句、1 种否则语句。

格式:　　IF 条件

　　语句序列 1

[ELSE

　　语句序列 2]

ENDIF

功能:当条件为真(满足) 时执行汇编语句序列 1,否则执行汇编语句序列 2。

说明

“ 条件 ”为 IF 伪指令说明符的一部分。ELSE 伪指令及其后面的语句序列 2 是可选部分,表示条件为假(不满足) 时的情况,如果是非完全分支的判断,就不用这部分。整个条件汇编最后必须用 ENDIF 伪指令来结束。语句序列 1 和语句序列 2 中的语句是任意的,也可为条件汇编语句。条件汇编伪指令语句如表 4 - 4 所示。

表 4 - 4　条件汇编伪指令语句

伪指令语句		格　式	功　能	
是 0 否条件语句	IF	IF 表达式	表达式值非 0,则条件为真,汇编语句序列 1	
	IFE	IFE 表达式	表达式值为 0,则条件为真,汇编语句序列 1	
扫描 1 否条件语句	IF1	IF1	汇编处于第一次扫描时条件为真	
	IF2	IF2	汇编处于第二次扫描时条件为真	
符号有定义否条件语句	IFDEF	IFDEF 符号	符号已被定义或已由 EXTRN 伪指令说明,则条件为真	
	IFNDEF	IFNDEF 符号	符号未被定义或未由 EXTRN 伪指令说明,则条件为真	
空否条件语句	IFB	IFB(参数)	参数为空格,则条件为真,括弧不能省	
	IFNB	IFNB(参数)	参数不为空格,则条件为真,括弧不能省	
字符串比较条件语句	IFIDN	IFIDN〈字符串 1〉,〈字符串 2〉	字符串 1 与字符串 2 相同,则条件为真	这两条语句只能在宏定义中使用,用于检查传送给两个参数的实参是否相同
	IFDIF	IFDIF〈字符串 1〉,〈字符串 2〉	字符串 1 与字符串 2 不相同,则条件为真	

【例 4.8】　下列程序段将输入及输出字符的 DOS 功能调用放在一个宏定义中,通过判断参数为 0 还是非 0 值来选择是执行汇编输入还是输出 DOS 功能。

```
INOUT   ACRO X
        IF X
        MOV AH,2
        INT 21H             ;输出 DL 中的字符
        ELSE
        MOV AH,1
        INT 21H             ;输入一个字符到 AL
```

```
ENDI F
ENDM
```

当宏调用为 INOUT 0 时,表明传递给参数 X 的值为 0,此时 IF X 的条件为假,因此汇编程序只汇编 ELSE 与 ENDIF 之间的语句,这样对该宏调用来说,实际上是执行下面的两条指令:

```
MOV AH,1
I NT 21H
```

而当宏调用为 INOUT 1 时,实际上是执行这样两条指令:

```
MOV AH,2
I NT 21H
```

4.2.4　列表伪指令语句

汇编程序在对源程序进行汇编时,除可产生目标代码(.OBJ)外,还可产生一个列表文件(.LST)和一个交叉参考列表文件(.CRE)。它们都是能被显示或打印的文件,其中列表文件以源程序指令与其相应目标程序指令相对照的形式给出汇编结果,随后给出程序中所用符号(标号、变量名等)的符号表;交叉参考列表文件按字母顺序列出源程序中所用的符号清单及其使用情况,并给出它们在程序中使用的位置(行号)。这两种文件便于调试。

列表伪指令就是用来控制上述两种文件的输出格式和方式的。表 4-5 给出了各列表伪指令语句的格式及功能。

表 4-5　列表伪指令语句

伪指令名	语句格式	功　能	说　明
显示汇编信息	%OUT 信息	汇编时,将该伪指令语句给出的信息在屏幕上显示	“信息”可以是任意的 ASCII 码字符串。若要显示多行信息,每行开头都要显示本语句
控制列表文件显示页格式	TITLE 标题	指定列表文件每页的标题	“标题”可以是长度在 80 个字符内的字符串。若无本语句,则标题为空
	SUBTTL 子标题	指定列表文件每页的子标题	“子标题”可以是长度在 60 个字符内的字符串。若无本语句或虽有但未给出子标题,则子标题为空
	PAGE ([行数],[列数])	控制列表文件的分页及每页的行、列数	行数范围在 10～255 之间,默认值为 58 行;列数范围在 60～132 之间,默认值为 80 个字符。若没选行、列数这两个参数,则使打印机换页,走纸到下页的顶端,页号加 1,并打印新页的标题、子标题及文件的其余部分
	PAGE +	控制换页并开始新的一节,且使节号加 1,页号重置为 1	这里节数是对整个文件而言,而页号则是在节中的页数

续表

伪指令名	语句格式	功　能	说　明
控制列表文件输出内容	. LIST	控制列表的开始	两者一般成对出现，用来选择或禁止源程序中某些行的输出
	. XLIST	控制列表的结束	
	. LFCOND	允许其后条件为假的条件块被列表	与. LIST 和. XLIST 类似，一般成对出现，用于限制程序中条件块的列表输出
	. SFCOND	禁止其后条件为假的条件块被列表	
	. TECOND	改变对条件为假的条件块的现行列表状态，即原来列表改为不列表，原来不列表改为列表	该伪指令语句控制条件块列表的状态正好与. LFCOMD 和. SFCOND 相反
	. LALL	将宏展开中的所有内容列表输出	用于控制宏展开的内容是否包含在列表文件中
	. SALL	对所有宏展开内容均不列表输出	
	. XALL	只将宏展开产生数据或目标代码的语句列表输出	
控制交叉参考列表文件的输出	. CREF	控制交叉参考信息(以字母顺序排列的符号、变量、标号的清单及其使用情况)是否列在交叉参考列表文件(. CRF)中	恢复在. CRF 文件中产生交叉参考信息。汇编程序默认的状态是. CREF
	. XCREF		禁止在. CRF 文件中产生参考信息

4.3　汇编语言程序的调试与运行

4.3.1　上机调试过程

1. 编辑汇编源程序

上机调试的第一步是编辑源程序。编辑源程序可以通过任何一个文本编辑器实现。例如，Windows 系统中的记事本、写字版，DOS 中的全屏幕文本编辑器 EDIT，也可以采用 MASM 程序员工作平台 PWB 中的编辑环境。

例如，在 Windows 的记事本中编写查询学生成绩程序，代码如下。

```
.model small
.stack
.data        ; 为初始变量和查询结果分配单元
tab db 78H,84H,80H,85H,56H,77H,87H,56H
no          db 6
english     db ?
.code
.st rtup
```

```
    lea bx,tab          ; bx 指向 TAB 表首地址
    mov al,no           ; 学号送 al 寄存器
    dec al
    xlat tab            ; 用换码指令查表
    mov english,al      ;结果保存在 english 单元
    .exit 0             ; 结束退出
end
```

程序中学生成绩按学号(从 1 开始)从小到大的顺序排列在 tab 表中,被查询的学生学号放在变量 no 中,查表结果放在变量 english 中。

假定将源程序存储到 C:\masm32 目录下,命名为 query.asm。

2. 用 MASM 汇编

汇编是将源程序翻译成由机器代码组成的目标模块文件的过程。下面以宏汇编 masm.exe 进行汇编,假设 masm.exe 文件存储在 C:\masm32 目录下。

命令如下:

C:\masm32 > masm query.asm

如果源程序中没有语法错误,MASM 将自动生成一个目标模块文件 query.obj,否则 MASM 将给出相应的错误信息。这时应根据错误信息,重新编辑修改源程序后,再进行汇编。

3. 目标文件的连接

连接程序能把一个或多个目标文件和库文件合成一个可执行文件(.EXE,.COM 文件)。下面以 link.exe 进行连接,假设 link.exe 文件存储在 C:\masm32 目录下。

命令如下:

C:\masm32 > link query.obj

如果不带文件名,LINK 连接程序将提示输入 OBJ 文件名,它还会提示生成的可执行文件名以及列表文件名,一般采用默认文件名即可。如果没有严重错误,LINK 将生成一个可执行文件 query.exe,否则将提示相应的错误信息,这时需要根据错误信息重新修改源程序后再汇编、连接,直到生成可执行文件。

4. 程序调试

汇编程序源代码经过汇编和连接后就生成了.EXE 文件,.EXE 文件可以直接在 DOS 下执行,如下所示:

```
C:\masm32 >  query.exe
C:\masm32 >
```

此时程序运行结束并返回 DOS。如果程序已经把结果在终端上显示出来,那么利用以上方法就可以观察到程序的运行结果,但是有很多程序并不直接显示运行结果。例如,query.exe 查询学号为 6 的学生成绩,结果在内存单元中,并未显示到屏幕上,要想查看结果必须借助 DEBUG 的帮助。此外,大部分程序必须经过 DEBUG 的调试才能纠正程序执行中的错误,得到正确结果。

下面以 query.exe 说明利用 DEBUG 进行程序调试的过程。

(1) 进入 DEBUG 并装入程序

命令如下:

```
C: \masm32 > debug query. exe
-
```

DEBUG 以短划线“ - ”作为提示符, 在提示符下可键入 DEBUG 命令。

(2) 使用反汇编命令

程序中学生成绩按学号(从 1 开始) 从小到大的顺序排列在 tab 表中, 要查的学生学号放在变量 no 中, 查表结果放在变量 english 中。要查询运行结果必须知道变量 english 对应的存储单元的地址。另外, 查询运行结果时我们希望程序暂停在返回 DOS 之前, 因此要确定断点地址。

键入反汇编命令 U 后, 显示信息如下:

```
- U
0B84:0000 BA860B        MOV     DX,0B86
0B84:0003 8EDA          MOV     DS,DX
0B84:0005 8CD3          MOV     BX,SS
0B84:0007 2BDA          SUB     BX,DX
0B84:0009 D1E3          SHL     BX,1
0B84:000B D1E3          SHL     BX,1
0B84:000D D1E3          SHL     BX,1
0B84:000F D1E3          SHL     BX,1
0B84:0011 FA            CLI
0B84:0012 8ED2          MOV     SS,DX
0B84:0014 03E3          ADD     SP,BX
0B84:0016 FB            STI
0B84:0017 8D1E0A00      LEA     BX,[000A]
0B84:001B A01200        MOV     AL,[0012]
0B84:001E FEC8          DEC     AL
0B84:0020 D7            XLAT
0B84:0021 A21300        MOV     [0013],AL
0B84:0024 B8004C        MOV     AX,4C00
0B84:0027 CD21          INT     21
```

由反汇编结果可知 tab 对应的内存单元的起始地址为 000A, no 对应的内存单元的起始地址为 0012, english 对应的内存单元的起始地址为 0013, 程序执行到 0024 时调用 INT 21 返回 DOS, 因此执行断点设为 0024 可保证程序完整地执行。

由于程序中使用. startup 伪指令作为程序的默认启动地址, 程序汇编连接后会产生一个程序前缀, 因此程序代码段的第一条指令的偏移地址非 0, 例如本例中的 lea bx, tab 指令的偏移地址为 0017, 但是利用 G 命令执行程序时起始地址必须从 0000 开始。

(3) 断点执行

得到程序运行结果的存储地址及断点地址后, 就可以启动程序执行并使其暂停在断点 0024 处。

键入 G 0024, 信息如下:

```
- g 0024
AX=0077 BX=000A CX=0034 DX=0B86 SP=0420 BP=0000 SI=0000
DI=0000 DS=0B86 ES=0B74 SS=0B86 CS=0B84 IP=0024 NV UP EI PL NZ NA PE NC
```

```
0B84:0024 B8004C      MOV     AX,4C00
```

此时程序停在断点处,并显示出所有寄存器以及各标志位的当前值,最后一行给出了将要执行指令的地址、机器代码和对应的汇编语句。根据这些信息可以判断程序执行是否正确,例如由 AX = 0077 可知 AL = 77,正是要查找的学生的成绩。

(4) 查看运行结果

根据已知的 tab、no、english 的地址,利用显示内存单元内容命令 D 观察运行结果。键入 D 000A,信息如下:

```
-D 000A
0B86:0000                                78 84 80 85 56 77          x...Vw
0B86:0010  87 56 06 77 26 FF 36 84-13 26 FF 36 82 13 26 8E  .V.w&.6..&.6..&.
0B86:0020  34 7E 13 55 8B EC F7 46-06 00 02 5D 74 01 FB CF  4~.U...F...]t...
0B86:0030  68 01 80 60 1E 06 8B EC-E9 E0 00 68 02 80 60 1E  h..`.......h..`.
0B86:0040  06 8B EC E9 D5 00 68 03-80 60 1E 06 8B EC E9 CA  ......h..`......
```

其中左边给出数据的起始地址(格式、数据段地址、偏移地址),然后顺序给出每个字节单元的内容,中间用十六进制数表示内容,右边用字符显示内容。由以上内容可以看出从 000AH 开始的 8 个连续存储单元存储了成绩表,0012H 单元存储了学生学号 6,0013H 单元存储了学号为 6 的学生成绩 77,执行结果正确。

(5) 退出 DEBUG

程序调试结束,利用 Q 命令退出 DEBUG。键入 Q,信息如下:

```
-q
C:\masm32>
```

上面介绍了 DEBUG 调试执行程序的一般过程,读者可以结合程序调试的具体情况灵活运用其他 DEBUG 命令。DEBUG 是一个功能强大的调试工具,在程序调试工作中会提供很大的帮助。

4.3.2　常用 DEBUG 命令

DEBUG 程序是专门为分析、研制和开发汇编语言程序而设计的一种调试工具,具有跟踪程序执行、观察中间运行结果、显示和修改寄存器或存储单元内容等多种功能。它能使程序设计人员或用户触及机器内部,是学习汇编语言必须掌握的调试工具。

DEBUG 是 DOS 的一个外部命令,其命令格式如下:

[path] DEBUG [filename] [parm1] [parm2]

其中[path] 是 DEBUG 命令在磁盘上的路径,filename 是要用 DEBUG 来处理的文件的名字,它包括文件的盘符、路径、文件主名和扩展名。参数 parm1 和 parm2 是文件 filename 运行时使用的参数。

启动 DEBUG 时,将对 CPU 的各寄存器进行初始化,初始化操作将按以下几种方式进行。

① 如果启动时指定的 filename 是.EXE 文件,则 DEBUG 启动后将自动把指定的文件装入内存,并置:

CS 为程序代码段段地址;

IP 为第一条要执行指令的偏移地址;

SS 为堆栈段段地址;

SP 为堆栈底部 +1 单元的偏移地址;

DS 和 ES 是装入文件前第一个可用内存段的段地址(即 DEBUG 程序后的第一个段地址) ;

标志寄存器的所有标志位为 0;

BX(0) 和 CX 是装入的文件长度;

其余寄存器为 0。

② 如果启动 DEBUG 时指定的文件 filename 不是. EXE 文件, 则 DEBUG 将把文件装入内存, 并置:

4 个段寄存器为 DEBUG 程序后面的第一个段地址;

IP 指向 100H;

SP 指向这个段的段尾;

标志寄存器的所有标志位为 0;

BX 和 CX 是装入的文件长度;

其余寄存器为 0。

③ 如果启动 DEBUG 时不指定 filename, 则只是把 CPU 的各寄存器进行初始化, 初始化结果与上述的第②点相同。这时要想显示、修改文件, 可以用 DEBUG 的子命令装入文件。

DEBUG 的命令都用单个字母表示, 其后可跟一个或多个参数, 参数之间用空格或逗号分隔。

DEBUG 的命令参数大多数是地址或地址范围, 其地址书写格式为:

[段地址:] 偏移地址

其中的段地址可以用段寄存器名表示, 也可以用一个十六进制数表示。如: ES: 100 43A5: 200。

地址范围的书写格式为:

[段地址:] 起始偏移地址 终止偏移地址

[段地址:] 起始偏移地址 L 长度

如, CS: 100 10F 和 CS: 100 L10 所指的地址范围是一致的。

当输入的命令不正确时, DEBUG 将在该行下面指出错误所在。

注意: 在 DEBUG 下, 输入的数据和显示的数据都是十六进制数, 不用在数据后加 H。

1. 汇编与反汇编命令

(1) 汇编命令 A

格式: A [地址]

功能: 从键盘输入汇编程序, 并逐条地把汇编指令翻译成机器代码指令存入对应内存单元。

说明　如果不指定汇编地址, 则以 CS: IP 为地址。

(2) 反汇编命令 U

格式: U [地址] /[地址范围]

功能: 将指定地址范围内的机器代码翻译成汇编源程序指令显示出来, 并同时显示地址及代码。

注意: 反汇编时一定要确认指令的起始地址, 否则得不到正确的结果。

2. 显示、修改内存单元内容的命令

(1) 显示内存单元内容命令 D

格式 1: D [地址]

格式 2: D 地址范围

说明: D 命令在屏幕上显示的内容分为 3 部分, 左边是每一行存储单元的起始地址, 中间是各字节单元的内容, 右边是各单元内容对应的 ASCII 码字符(不可显示的字符用“.”代替)。

例如:

```
-D200
0B2E:0200  E8 DA E1 46 E8 AC DF 74-0D E8 45 00 AC E8 41 00  ...F...t..E...A.
0B2E:0210  81 CD 00 40 EB 34 3C 0D-75 09 B0 00 AA 81 CD 00  ...@.4<.u.......
0B2E:0220  40 EB CE E8 2B 00 E8 B4-DF 06 57 51 0E 07 BF A3  @...+.....WQ....
0B2E:0230  8F B9 06 00 81 CD 00 40-F2 AE 75 0B 81 E5 FF BF  .......@..u.....
0B2E:0240  B8 01 00 D3 E0 0B E8 59-5F 07 B0 00 AA 5F 9D F8  .......Y_...._..
0B2E:0250  C3 AA 41 FE 06 E8 99 C3-2E C7 06 55 91 00 00 2E  ..A........U....
0B2E:0260  89 0E DF 91 2E 89 26 E1-91 2E 89 36 E3 91 FC 2E  ......&....6....
0B2E:0270  89 0E 48 91 2E C7 06 4A-91 00 00 2E C7 06 5D 91  ..H....J......].
```

其中 0B2E 是段基址。

(2) 修改内存单元内容命令 E

格式 1: E 地址 内容表

说明: 内容表可以是以逗号或空格分隔的两位十六进制数, 也可以是用单引号或双引号括起来的字符串, 还可以是二者的组合。

格式 2: E 地址

说明　在修改数据时可用以下键进行不同操作。

键入空格键: 修改后一个字节单元的内容。

输入减号“-”: 另起一行, 修改前面一个字节单元的内容。

输入回车键: 结束内存单元的修改。

例如:

```
-E100  ABC 12 34      ;修改 DS:0100 为起始单元的连续 5 个字节单元的内容
-D100                 ;显示修改后的结果
0B2E:0100  41 42 43 12 34 61 2E 61-73 6D 41 96 2B 00 00 05  ABC.4a.asmA.+...
0B2E:0110  53 54 41 43 4B 05 5F 44-41 54 41 06 34 00 1D 0B  STACK._DATA.4...
```

(3) 填充内存命令 F

格式: F 地址范围 内容表

功能: 将内容表的值逐个填入指定地址范围, 内容表中的内容用完后再重复使用。

3. 显示、修改寄存器内容的 R 命令

格式 1: R

功能: 显示当前所有寄存器的内容、状态标志及将要执行的下一条指令的地址、代码和汇编指令形式。

格式 2: R 寄存器名

功能: 显示并修改指定寄存器的内容

例如:

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B2E ES=0B2E SS=0B2E CS=0B2E IP=0100 NV UP EI PL NZ NA PO NC
0B2E:0100 99        CWD
-R AX               ;输入命令
AX 0000             ;显示 AX 的内容
:                   ;供修改, 不修改按回车
```

4. 运行和跟踪命令

(1) 运行程序命令 G

格式: G [=起始地址] [断点地址]

功能: 从起始地址开始执行程序, 直到程序结束或遇到断点地址为止。

说明: 如果不指定起始地址, 则从 CS: IP 处开始执行。如果程序执行到结束, 则显示“Program terminated normally”(程序正常结束)。如果遇到断点, 则程序停止执行, 并显示当时各寄存器的内容和下一条要执行的指令。

(2) 跟踪运行命令 T

格式: T [=起始地址] [指令条数]

功能: 逐条跟踪程序的运行, 同时显示出各寄存器的内容、状态标志和下一条要执行的指令, 当执行够指定的指令数后就暂停程序的运行。

说明: 如果不指定起始地址, 则从 CS: IP 处开始执行。不指定指令条数时, 认为只执行一条指令。

(3) 继续命令 P

格式: P [=起始地址] [指令条数]

功能: 与 T 命令一样跟踪程序的运行, 但遇到子程序、中断程序、循环时并不跟踪下去, 而是把它们作为一条指令来执行。

5. 磁盘读写命令

(1) 文件命名命令 N

格式: N 文件名

功能: 指定要装入内存或写到磁盘的文件的名字(包括盘符和路径)。

(2) 装入命令 L

格式: L [地址] [驱动器号 扇区号 扇区数]

功能: 把指定文件或磁盘扇区的内容装入到内存指定地址。

说明: 地址的默认值为 CS: 100。驱动器号用 0 表示 A 盘, 1 表示 B 盘, 2 表示 C 盘。

(3) 写磁盘命令 W

格式: W [地址] [驱动器号 扇区号 扇区数]

功能: 将指定内存地址的一组单元内容写到磁盘中。

说明: 地址的默认值为 CS: 100。要将内存内容写入文件时, 必须先用 N 命令命名一个文件, 并置 BX 和 CX 为文件长度。

注意: W 命令不能写入以. EXE 和. HEX 为扩展名的文件。

6. DEBUG 的其他命令

(1) 移动内存命令

格式: M 源地址范围 目标起始地址

功能: 把源地址范围中的内容顺序移到从目标起始地址开始的一组连续内存单元。

注意: 源区域的数据不因移动而消失, 其内容仍保持不变。源、目标中的地址只要不指定段地址, 则都将隐含使用 DS 段。

(2) 比较命令 C

格式: C 源地址范围 目标起始地址

功能: 从源地址范围的起始地址单元开始, 逐个与目标起始地址后的单元的内容顺序进行比较, 直到源终止地址为止。遇到不相同时显示出它们的地址和内容, 源地址、源内容、目标内容、目标地址。

(3) 查找命令 S

格式: S 地址范围 要查找的内容

功能: 在指定的地址范围内查找指定的内容, 若找到则显示出它们所处的地址, 否则不显示任何信息。

(4) 十六进制算术运算命令 H

格式: H 值 1　值 2

功能: 显示十六进制数值 1 与值 2 的和、差的结果。

(5) 退出 DEBUG 命令 Q

格式: Q

功能: 结束 DEBUG 程序, 返回到 DOS 提示符下。

注意: Q 命令并不把内存中正在工作的文件存盘。

以上介绍的是 DEBUG 的常用命令, 其他命令可参阅本书附录 3。

习　题

1. 什么是伪指令, 伪指令的主要作用有哪些?

2. 什么是宏指令, 什么是宏调用?

3. 什么是过程, 什么是过程调用?

4. 宏调用与过程调用的主要区别有哪些?

5. 判断下列标识符的合法性。

(1) code　(2) ABCDH5　(3) eax　(4) @A? 9

(5) Data　(6) 0123　(7) ??? 01　(8) www@zsu

6. 从 FIRST 开始的 100 个单元中存放着一个字符串, 试编程统计该字符串中字母 A 的个数。

7. 从 BLOCK 开始, 存放着 256 个字节的带符号数, 试编程从这些数中找出绝对值最大的数, 将其存入 MAX 单元中。

8. 编写一个程序, 将一个 64 位二进制数转换成 ASCII 码字符串。

9. 试用 3 种方法定义字节变量(或字段名) B1 和字变量(或字段名) W1, 它们共享 20 个存储单元。

10. 用 DEBUG 调试汇编语言程序时, 显示某指令的地址是 2F80: F400, 此时段寄存器 CS 的值是多少?

第 5 章　汇编程序设计

汇编语言是一种易学,却很难精通的语言。对于一个大型软件而言,编写一个结构合理、快速高效和稳定可靠的程序是一件很困难的事情,必须经过长期的编程训练才能达到。针对汇编语言的特点,深入学习和掌握汇编语言程序的结构和编程方法,在软件工程的编码阶段是非常必要的。本章将详细介绍汇编语言程序的顺序、分支、循环、子程序及模块化程序设计技术。

5.1　程序设计方法

为了解决实际问题,就必须设计程序。对汇编程序而言,就是要编制一个汇编语言源程序,然后上机调试程序。汇编语言是一种最接近计算机核心的编码语言,编写出高质量的程序,它可以最大限度地发挥计算机硬件的性能。一般来说,编写汇编程序应遵循如下步骤。

① 分析问题,确定算法。这一步是能否编制出高质量程序的关键,在开始设计之前,应该仔细地分析和理解问题,确定需求。对于一些较为复杂的问题,还需要将问题进行分解,划分模块并确定各模块间的接口关系。然后根据要解决的问题确定合理的算法及适当的数据结构。

② 绘制流程图。流程图可以将算法逻辑控制结构以及数据流程形象地表示出来,是设计思想的直观表现形式。流程图是编写程序的指导性文件,认真绘制流程图可以有效地减少出错的可能性。这一点对初学者而言尤其重要。

③ 分配资源。汇编程序不同于高级语言程序,汇编程序的许多资源需要由程序员来分配和使用,而这些资源有时是非常有限的。因此,在开始编写代码前,要合理地分配和使用这些资源,例如为原始数据、中间结果和最后结果分配必要的存储空间或寄存器。

④ 根据流程图编写程序。在编写代码的过程中,要以程序流程图为主线,并根据预先确定的资源分配方案,选择合适的汇编语句进行编制并进行必要的注释。

⑤ 上机调试程序。调试程序的目的是查找和排除软件错误(Bug)的关键环节,任何程序都必须经过反复的上机调试才能验证设计思想是否正确,以及编写的程序是否符合设计思想。在调试程序的过程中应充分利用调试工具(如 DEBUG)来调试程序,发现并修正程序中存在的各种语法错误和逻辑错误。

从程序结构上看,汇编程序有顺序、分支、循环和子程序 4 种基本结构形式。为了提高编程能力,必须了解和掌握汇编程序的结构。

5.2　顺序程序设计

顺序程序是一种最简单也是最基本的结构形式,它的执行流程与指令的排列顺序完全一致,运行时从前至后逐条执行。对于一些复杂的问题,顺序结构往往是作为复杂程序结构

中的一部分;而对于一些简单的问题,我们可以依次写出相应的汇编指令,用顺序结构即可实现编程要求。下面通过几个例子说明。

【例 5.1】　已知某班学生的英语成绩按学号(从 1 开始)从小到大的顺序排列在 tab 表中,要查的学生学号放在变量 no 中,查表结果放在变量 english 中。

根据问题要求,可绘制出如图 5 - 1 所示的程序流程图。

汇编源程序如下:

```
.model small
.stack
.da a                    ; 为初始变量和查询结果分配单元
   tab db 68,78,42,84,80,85,56,77,87,56
   no       db 6
   english db ?
.code
.st rtup
   lea bx,tab       ; bx 指向 tab 表首地址
   mov al,no        ; 学号送 al 寄存器
   dec al
   xlat tab         ; 用换码指令查表
   mov english,al ;结果保存在 english 单元
.exit 0             ; 结束退出
end
```

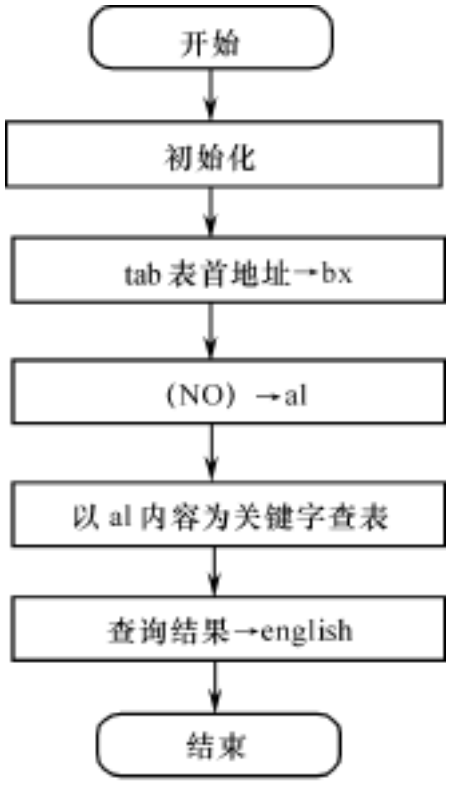


图 5 - 1　简单查表程序

【例 5.2】　把字单元 dat1 中的一个非压缩十进制数转换为一个压缩的十进制数,并将结果保存在字节单元 dat2 中。

根据问题要求,绘制流程图,如图 5 - 2 所示。

编源程序如下:

```
.model small
.stack 200h
.data
      dat1   dw0506h   ; 非压缩十进制数 56
      dat2   db ?
.star up
      mov ax,dat1   ;(ax) =0506h
      mov cl,4
      sal ah,cl      ;ah 左移 4 位后值为 50h
      rol ax,cl      ;ax 循环左移 4 位后值为 0065h
      rol al,cl      ;al 循环左移 4 位后值为 56h
      mov dat2,al   ;保存结果
.exit 0
end
```

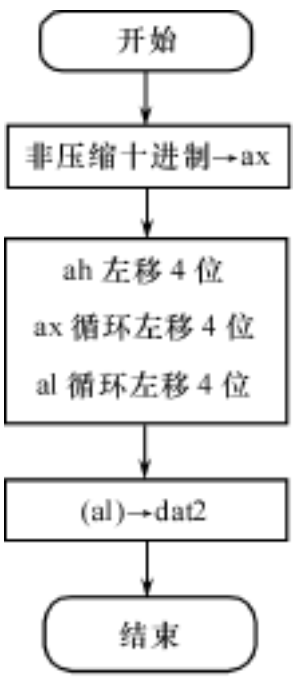


图 5 - 2　十进制数转换

【例 5.3】　编写一个程序,计算以下表达式的值:

w = (v - (x* y + z - 460)) / x

式中 x, y, z, v 均为带符号的字数据,要求结果存放在双字变量 w 之中。

程序的流程图如图 5 - 3 所示。

汇编源程序如下:

```
.model small
.stack
.data
    x       dw      200
    y       dw      300
    z       dw      4000
    v       dw      10000
    w       dd      ?
.code
.startup
    mov     ax,x
    imul    y           ;(x) * (y) →dx:ax
    mov     cx,ax
    mov     bx,dx       ;(dx:ax) →(bx:cx)
    mov     ax,z
    cwd                 ;(z) 符号扩展
    add     cx,ax
    adc     bx,dx       ;(bx:cx) +(dx:ax) →(bx:cx)
    sub     cx,460
    sbb     bx,0        ;(bx:cx) - 460 →(bx:cx)
    mov     ax,v
    cwd                 ;(v) 符号扩展
    sub     ax,cx
    sbb     dx,bx       ;(dx:ax) - (bx:cx) →(dx:ax)
    idiv    x           ;(dx:ax) /x
    mov     w,ax        ;商→w
    mov     w+2,dx      ;余数 dx→w+2
.exit 0
end
```

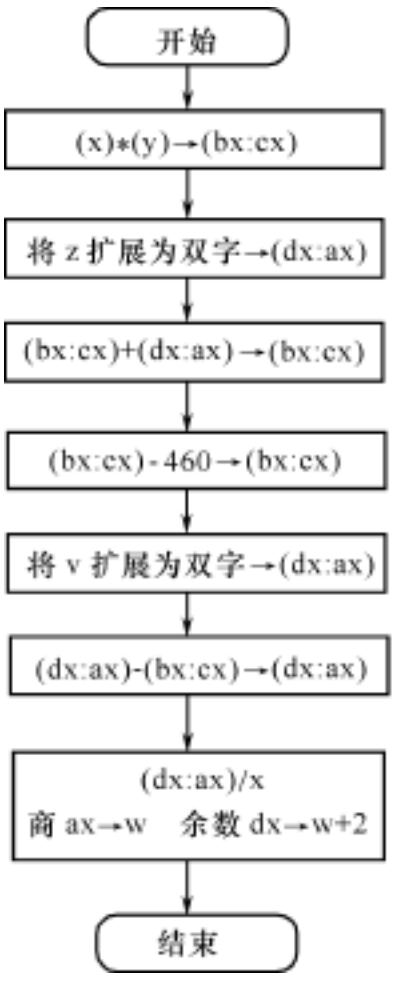


图 5 - 3　计算表达式的值

5.3　分支程序设计

5.3.1　分支结构

在许多实际问题中,往往需要对不同的情况或条件做出不同的处理,这就要用到分支结构程序。分支结构是利用条件转移指令或跳转表,使程序执行到某一指令后,根据运行结果是否满足一定条件来改变程序执行的顺序,然后去执行不同的分支语句。可以说,正是分支结构程序使计算机有了一定的分析和判断能力。

在程序中,当需要进行逻辑分支时,可用每次分两支的方法来达到程序最终分多支的要求,也可以用地址表的方法来达到分多支的目的。分支程序的结构通常有 3 种形式:不完全分支、完全分支和多分支,如图 5 - 4 所示。

条件转移指令 JCC 和无条件转移指令 JMP 用于实现程序的分支。JCC 指令是实现分

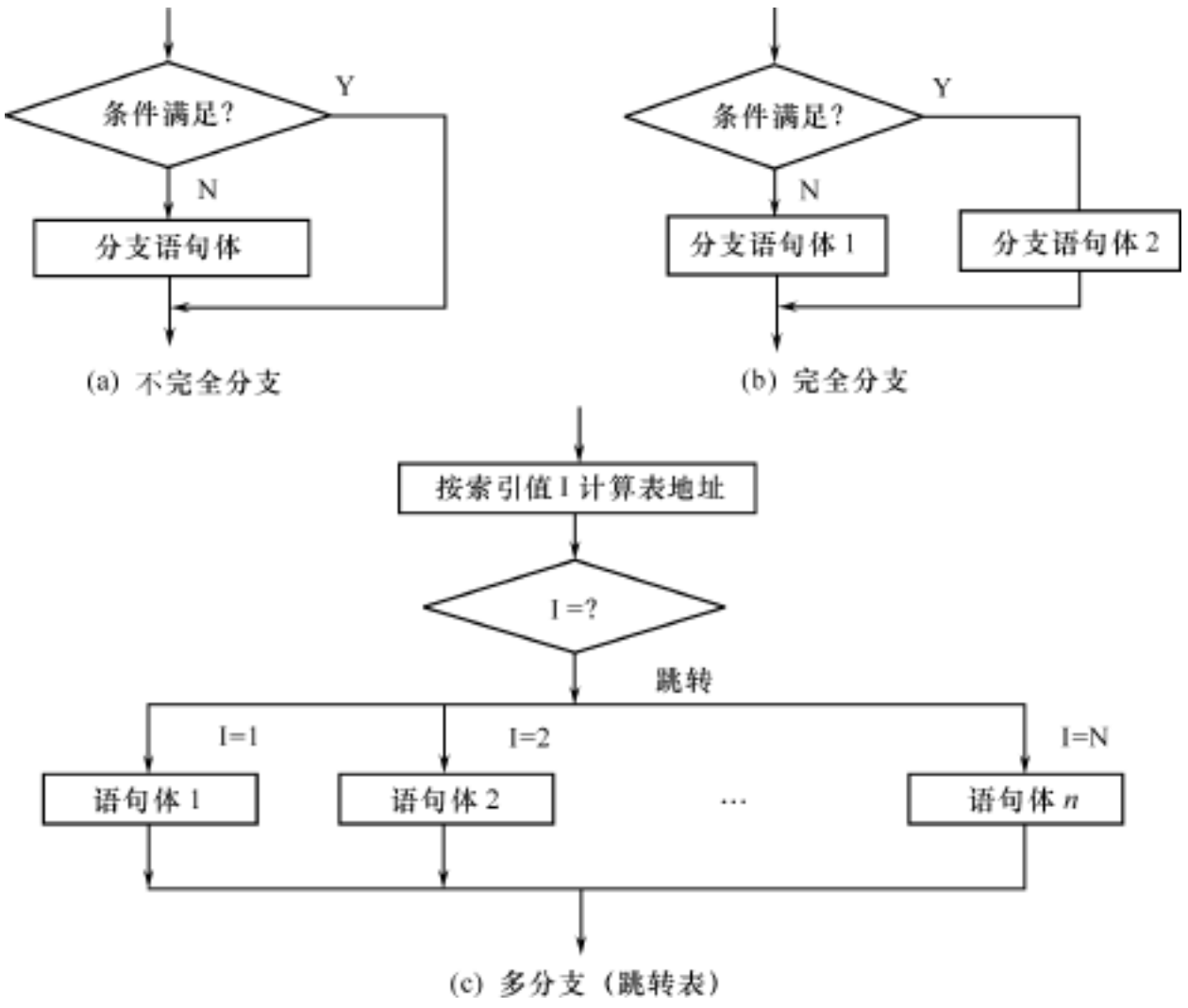


图 5 - 4　3 种典型分支结构

支结构最常用和最灵活的指令，该指令可根据当前某些标志位的状态实现转移或不转移。因此，在条件转移指令前，通常需要安排算术运算指令、比较指令或测试指令等能够影响标志位的相关指令。JMP 指令在分支程序中作为辅助性的指令，实现固定转移到某个指定的位置。

计算绝对值是一个典型的不完全分支结构，如下程序段用来计算 ax 的绝对值，并将计算结果保存在 result 单元中：

```
        cmp     ax,0        ; (ax) - 0,影响标志位
        jge     non_neg     ; ax≥0,转到 non_neg
        neg     ax          ; ax <0,求 ax 的补码
non_neg: mov result,ax      ; 保存 ax 的绝对值
```

再看一个完全分支结构的程序段，本例用来显示 bx 的最高位：

```
        shl     bx,1        ;bx 最高位移入到进位标志中
        jc      one         ;最高位为 1,转移到 one
        mov     dl, 0       ;最高位为 0, 0 →dl
        jmp     zero        ;跳过另一个分支体,执行显示
one :   mov     dl, 1       ;最高位为 1, 1 →dl
zero:   mov     ah,2        ;显示
        int 21h
```

下面这个程序段例子说明如何判断一个值是否在某个区间中。已知字节变量 char1，如

果是小写字母,则把它转换成大写字母。

```
        mov al,char1        ; 取字符的 ASCII 码值
        cmp al, a
        jb next             ; 字符值 <  a ,不处理
        cmp al, z
        ja next             ; 字符值 >  z ,不处理
        sub char1,20h       ;  a ≤ 字符值 ≤ z ,转换成大写字母
next: ...G
```

5.3.2　用分支指令实现分支结构程序

分支结构是一种非常重要的程序结构,也是实现程序功能选择所必要的程序结构。由于汇编语言需要书写转移指令来实现分支结构,而转移指令又会破坏程序的结构,所以,编写清晰的分支结构是掌握该结构的重点,也是用汇编语言编程的基本功。

在编写分支程序时,要尽可能避免编写“头重脚轻”的结构,即当前分支条件成立时,将执行一系列指令,而条件不成立时,所执行的指令很少。这样就使后一个分支离分支点较远,有时甚至会遗忘编写后一分支程序。这种分支方式不仅不利于程序的阅读,而且也不便于将来的维护。因此,在编写分支结构时,一般先处理简单的分支,再处理较复杂的分支。对多分支的情况,也可遵循“由易到难”的原则。因为简单的分支只需要较少的指令就能处理完,一旦处理完这种情况后,在后面的编程过程中就可集中考虑如何处理复杂的分支。下面通过一些例子来说明分支结构程序的设计方法。

【例 5.4】　设数据 X、Y 均为字节型变量,编写计算下面函数值的程序。

$$Y=\begin{cases}1 & X>0\\0 & X=0\\-1 & X<0\end{cases}$$

程序流程图如图 5-5 所示。

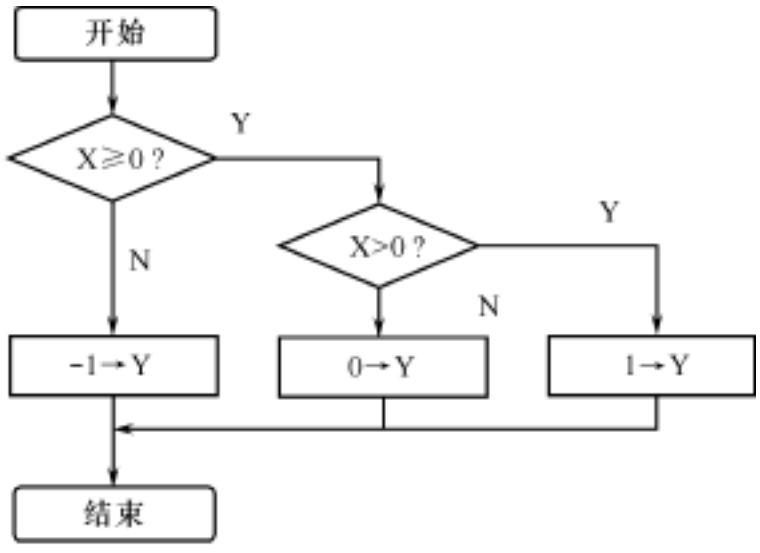


图 5-5　例 5.4 的程序流程图

汇编源程序如下:

```
.model small
.stack
.data
      x     db     -5
      y     db     ?
.code
.star up
      cmp       x,0
      jge       case1       ;当 x≥0 时,则转到 case1
      mov       y, -1       ;x <0 时, -1→y
      jmp       done
case1:
       jg       case2       ;x >0 时,则转到 case2
       mov      y,0         ;x =0 时,0→y
                jmp         done
case2:
```

```
        mov     y,1             ;x>0 时,1→y
done:
.exit 0
end
```

【例 5.5】　判断方程 $ax^2+6x+c=0$ 是否有实根,若有实根则将字节变量 tag 置 1,否则置 0(假设 a,b,c 均为字节变量),程序流程图如图 5-6 所示。

分析:二元一次方程有根的条件是:$b^2-4ac\geq 0$。根据题意,首先计算出 b^2 和 4ac,然后比较两者大小,然后根据比较结果分别给 tag 赋不同的值。

```
.model small
.stack
.data
        a       db      ?
        b       db      ?
        c       db      ?
.code
.start  p
        mov     al,b
        imul    al
        mov     bx,ax           ; b²→bx
        mov     al,a            ;
        imul    c
        mov     cx,4            ;
        imul    cx              ; 4ac→ax
        cmp     bx,ax           ; bx≥ax ?
        jge     yes             ; 满足条件,转到 yes
        mov     tag,0           ; 不满足条件时,0→tag
        jmp     done
yes:    mov tag,1               ;x>0 时,1→tag
done:
.exit   0
end
```

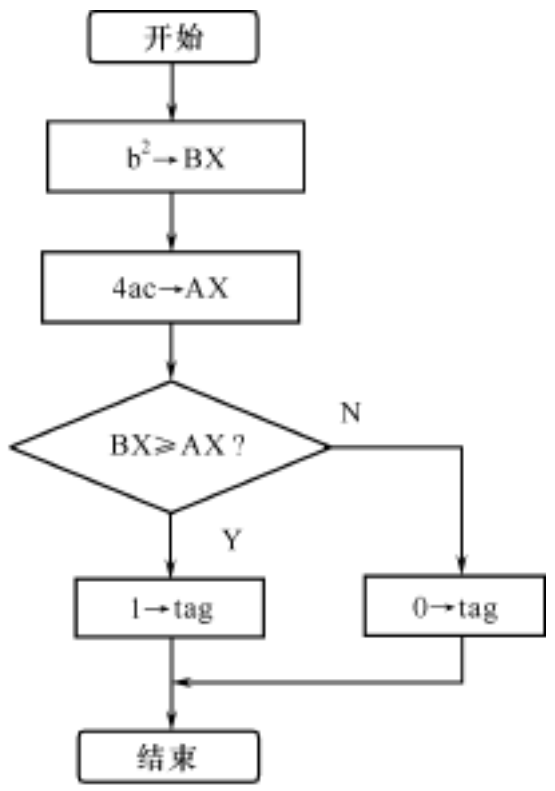


图 5-6　例 5.5 的程序流程图

【例 5.6】　设某程序有 8 路分支,试根据给定的 N 值(1~8),将程序的执行转移到其中的一路分支。

程序如下:

```
.model small
.stack
.data
        tab dw p1,p2,p3,p4,p5,p6,p7,p8
        n db 5
.code
.star up
        mov     al,n
        del     al
        add     al,al
        mov     bm,al
```

```
        mov     bh,0
        jmp     tab[bx]
p1:     …
        …
        jmp exit
p2:     …
        …
        jmp exit
        …
p8:     …
        …
exit:   mov ah,4ch
        int 21h
.exit   0
end
```

【例 5.7】　把下列 C 语言的语句改写成等价的汇编语言程序段(不考虑运算过程中的溢出)。

```
if (a+b > 0 && c%2 == 0)
    a = 62;
else
    a = 21;
```

其中变量 a, b 和 c 都是有符号的整型(int) 变量。

解

```
.model small
.stack
.data
        a   dw  ?
        b   dw  ?
        c   dw  ?
          ...
.code
        ...
        mov     ax,a
        add     ax,b        ; a+b,改变算术标志位
        jle     _else       ; 当 a+b<0 时,转到_else
        test    c,1         ; c%2 ==0,也就是看 c 的最低位是否为 0
        jnz     _else       ; 当 c%2 不0 时,转到_else
        mov     a,62d
        jmp     next
_else:
        mov a,21d
next : ...
```

【例 5.8】　用地址转移表实现下列 C 语言的 switch 语句, 其中变量 A 和 B 是带符号的整型(int) 变量。

```
switch (a%8)
{
```

```
    case 0:    = 32;
               break;
    case 1:
    case 2:    = a + 43;
               break;
    case 3:    = 2* a;
               break;
    case 4:    --;
               break;
    case 5:
    case 6:
    case 7:  rintf("Function5_6_7");
               break;
}
```

对应的汇编代码程序段如下:

```
.model small
.stack
.data
      a       dw ?
      b       dw ?
      table   dw case0, case12,case12,case3
              dw case4,case567,case567,case567
      msg     db function5_6_7 $
.code
.startup
          mov ax,a
          mov bx,ax
          and bx,7          ;得到 bx 的低三位, 实现 a%8 的计算
          shl bx,1          ;由于地址表是字类型, 其下标要乘 2
          jmp table[bx]     ;利用地址表实现多路转移
case0:    mov b,32d
          jmp next
case12:   add ax,43d
          mov b,ax
          jmp next
case3:    shl ax,1
          mov b,ax
          jmp next
case4:    dec b
          jmp next
case567:  lea dx,msg
          mov ah,9
          int 21h
          jmp next
next:     ...
```

用地址表实现多路转移的关键在于转移入口的地址表和转移情况可整数化。如果这两

个要求有一个不满足或很难构造,则无法使用该方法。

5.3.3　用伪指令实现分支结构

为了改善汇编语言源程序的结构,减少显式转移语句所带来的混乱,在宏汇编 MASM 6.11 系统中,增加了表达分支结构的伪指令。该伪指令的书写格式与高级语言的书写方式类似,汇编程序在汇编时会自动增加转移指令和相应的标号。

分支伪指令的具体格式如下。

格式1:

```
. IF c  ndition              ;以英文“ 句号 ”开头
      指令序列               ;条件“ condition ”成立时所执行的指令序列
. ENDIF
```

格式2:

```
. IF   ondition
    指令序列 1
. ELS
      指令序列 2             ;条件“ condition ”不成立时所执行的指令序列
. ENDIF
```

格式3:

```
. IF   ondition1
    指令序列 1
. ELS  IF condition2
      指令序列 2             ;条件“ condition2 ”成立时所执行的指令序列
. ENDIF
```

其中条件表达式“ condition ”的书写方式与 C 语言中条件表达式的书写方式相似,也可用括号来组成复杂的条件表达式。

条件表达式中可用的操作符有:

==	(等于)
!=	(不等)
>	(大于)
>=	(大于等于)
<	(小于)
<=	(小于等于)
&	(位操作与)
!	(逻辑非)
&&	(逻辑与)
||	(逻辑或)

若在条件表达式中检测标志位的信息,则可以使用的符号名有:

CARRY?	(相当于 CF == 1)
OVERFLOW?	(相当于 OF == 1)

PARITY?　　(相当于 PF ==1)
SIGN?　　(相当于 SF ==1)
ZERO?　　(相当于 ZF ==1)

例如:

```
.IF  ARRY? && AX != BX ;检测 CF ==1 且 AX!= BX 是否成立
     汇编语言指令序列
.ENDIF
```

在指令序列中,还可再含有其他的.IF 伪指令,即允许嵌套。伪指令.ELSEIF 引导出另一个二叉分支,但它不能作伪指令块的第一个伪指令。

汇编程序在对“条件表达式”进行代码转换时将进行代码优化处理,以便尽可能生成最好的指令代码。如:

```
.IF ax == 0
```

汇编程序会把它转换为指令“OR ax,ax”,而不是直接地转换成“CMP ax,0”,因为前者比后者更好。

用伪指令书写的分支结构类似于高级语言的表达式,结构简单明了,程序可读性强,例如已知字节变量 CHAR1,如果是小写字母,则把它转换成大写字母的程序段如下。

```
...
MOV     AL,CHAR1
.IF     AL >= a     && AL <= z     ;语句像 C 语言语句
        sub   char1,20h
.ENDIF
...
```

也可把例 5.4 的代码段部分写成如下程序段。

```
...
mov     ax,x
.IF     AX < 0
        mov y,-1
.ELSEIF  AX == 0
        mov y,0
.ELSE
        mov y,1
.ENDIF
...
```

【例 5.9】　根据当前计算机的时间和日期,显示上午(AM)或下午(PM)以及所在的季节。

```
.model small
.stack
.data
      msg     db   "time: "
      ampm    db   "am",13,10
              db   "season: $"
```

```
        winter   db    "winter $"
        spring   db    "spring $"
        summer   db    "summer $"
        autumn   db    "autumn $"
.code
.star up
      mov    ah,2ch                ;取当前系统时间
      int    21h
      .IF    ch >= 12              ;下午时间
             mov ampm, p           ;为显示 pm作安排
      .ENDIF
      mov    dx,offset msg
      mov    ah,09h
      int    21h                   ;显示字符串 msg,直到 $ 结束
      mov    ah,2ah
      int    21h                   ;取当前系统日期
      .IF     dh == 12) ||(dh < 3)     ;判断是否为12 月、1 月和 2 月
          mov dx,offset winter
      .ELSEIF (dh >= 3) &&(dh < 6)     ;判断是否为3、4 和5 月
          mov dx,offset spring
      .ELSEIF (dh >= 6) &&(dh < 9)     ;判断是否为6、7 和8 月
          mov dx,offset summer
      .ELSE  ; 9、10 和 11 月
          mov dx,offset autumn
      .ENDIF
      mov ah,09h                       ;显示季度名称
      int 21h
done:
.exit 0
end
```

5.4　循环程序设计

5.4.1　循环结构

当需要重复执行某段程序时,可以利用循环程序结构。循环结构一般是根据某一条件判断为真或假来确定是否重复执行循环体,条件永真或无条件的重复循环就是逻辑上的死循环(永真循环、无条件循环) 。

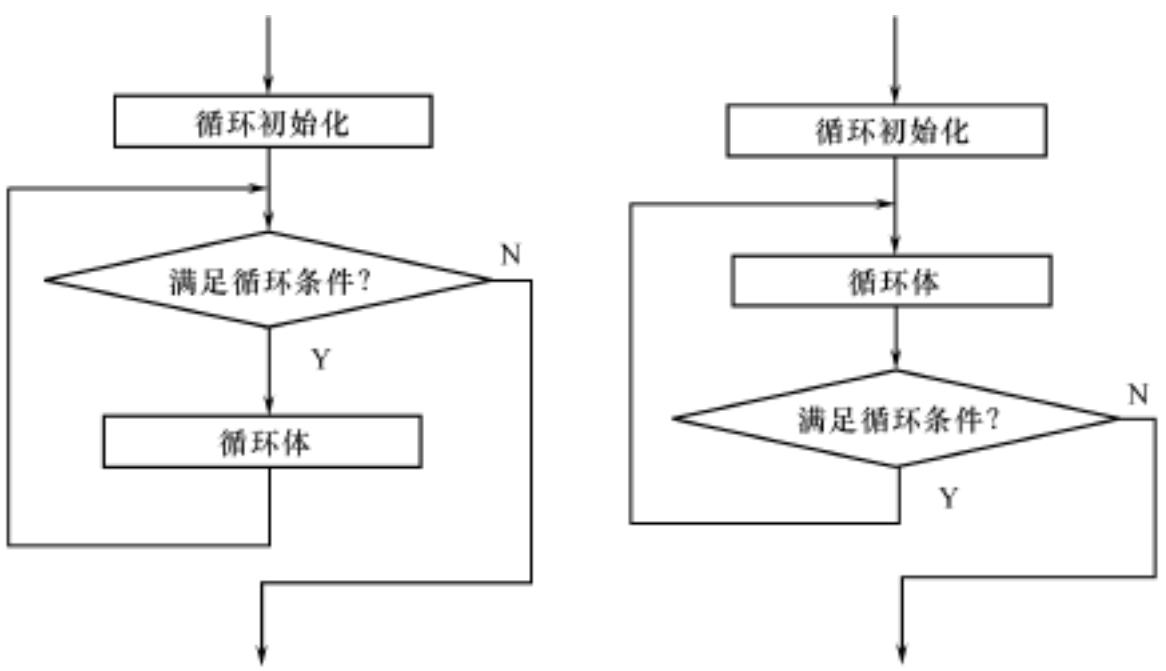
循环结构的程序通常由以下 3 个部分组成。

循环初始部分: 为开始循环准备必要的条件,如循环次数及为循环体正常工作而建立的初始状态等。

循环体部分: 重复执行的程序代码,这是循环工作的主体,它由循环的工作部分及修改部分组成。循环的工作部分是为完成程序功能而设计的主要程序段,循环的修改部分则是为保证每一次重复时,参加执行的信息能发生有规律的变化而建立的程序段。

循环控制部分: 判断循环条件是否成立,决定是否继续循环。

其中循环控制部分是编程的关键和难点。每个循环程序必须选择一个循环控制条件来控制循环的运行和结束,而合理地选择该控制条件就成为循环程序设计的关键问题。循环条件判断的循环控制可以在进入循环之前进行,即形成“先判断、后循环”的 WHILE 型循环程序结构,满足条件就执行循环体,否则退出循环,如图 5 - 7 (a) 所示。如果循环之后进行循环条件判断,即形成“先循环、后判断”的 DO_UNTIL 型循环程序结构,不满足条件则继续执行循环操作,一旦满足条件则退出循环,如图 5 - 7 (b) 所示。



(a) WHILE 型循环结构　　(b) DO_UNTIL 型循环结构

图 5 - 7　两种典型循环结构

循环控制本来应该属于循环体的一部分,由于它是循环程序的关键,所以要对它做专门的讨论。有时循环次数是已知的,此时可以用循环次数作为循环的控制条件,LMP 指令使这种循环程序设计能很容易地实现。有时循环次数是已知的,但有可能使用其他特征或条件来使循环提前结束。

但是,有时循环次数是未知的,那就需要根据具体情况找出控制循环结束的条件。循环控制条件的选择是很灵活的,有时可能的选择方案不止一种,此时就应分析比较选择一种效率最高的方案来实现。

循环控制方式通常有以下 4 种。

计数控制: 事先已知循环次数,设循环一次加 / 减 1。

条件控制: 事先不知循环次数,根据条件真假控制循环。

状态控制: 根据事先设置或是实时检测的状态来控制循环。

逻辑尺控制: 当循环条件不规则时,可通过位串(逻辑尺) 来控制循环。

无论采用哪种循环控制方式,最终都是要达到循环控制的目的。

8086 CPU 指令集中有一组专门用于循环控制的指令,它们是:

JCXZ

LOOP

LOOPE / LOOPZ

LOOPNE / LOOPNZ

从某种意义上讲，它们都是计数循环，即用于循环次数已知或最大循环次数已知的循环控制，且需预先将循环次数或最大循环次数置入 CX 寄存器；LOOPE / LOOPZ、LOOPNE / LOOPNZ 只是在计数循环的基础上增加了关于 ZF 标志位的测试，可根据标志位 ZF 值的当前状态提前退出计数循环或继续下一次循环。

在编写循环结构的程序时，我们可以采用多种方法。如循环次数是已知的，可用 LOOP 指令来构造循环；当循环次数未知或不定时，则可用条件转移或无条件转移来构造循环结构。循环程序分为单循环和多重循环。对于多重循环要求内外循环不能交叉，即内循必须完整地包含在外循环中。

5.4.2　单循环程序设计

所谓单循环，即其循环体内不再包含循环结构，这种循环结构比较简单。下面举例说明单循环设计的方法。

【例 5.10】　计算数组 score 的平均整数并存入内存字变量 Average 中，数组以 - 1 为结束标志。

程序流程图如图 5 - 8 所示。程序如下：

```
.model small
.stack
.data
      score dw90,95,54,65,36,78,66,0,99,50, - 1
      average dw0
.code
.star up
      xor   ax,ax
      xor   dx,dx          ;用(dx,ax)来保存数组元素之和
      xor   cx,cx          ;用 cx 来保存数组元素个数
      lea   si,score        用 si 来访问整个数组
again
      mov  bx,word ptr [si]
      cmp  bx,0
      jl   over
      add  ax,bx
      adc  dx,0            ;把当前数组元素之值加到(dx,ax)中
      inc  cx              ;数组元素个数加 1
      add  si,2
      jmp  again
over:
      jcxz exit            ;防止零作除数，即数组是空数组
      div  cx
      mov  average,ax
exit:
.exit 0
end
```

【例 5.11】　某采集系统采集的 12 个参数值存于 BUFFER 起始的缓冲区中，要求预处

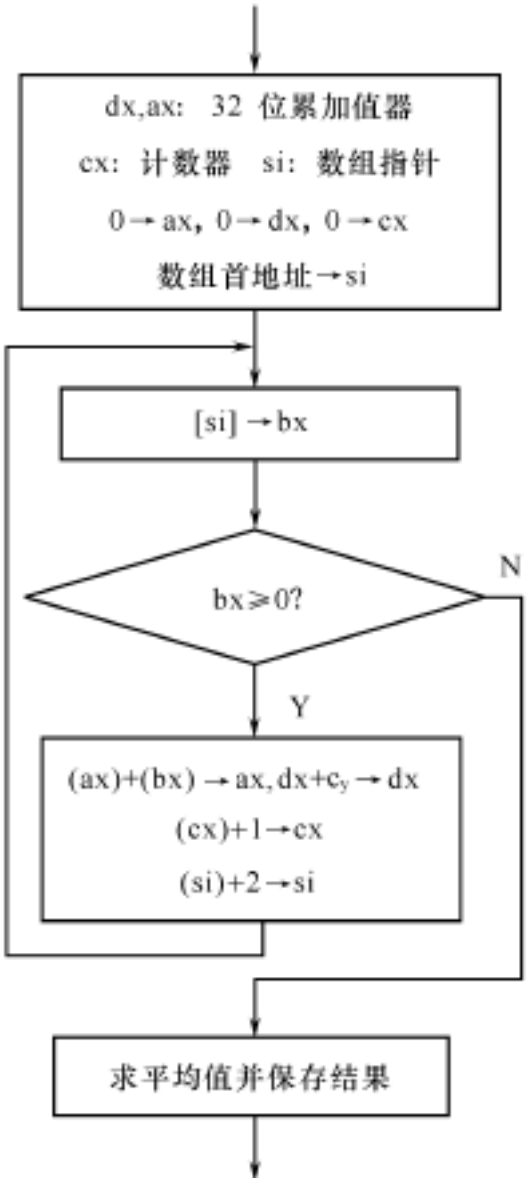


图 5 - 8　例 5. 10 的程序流程图

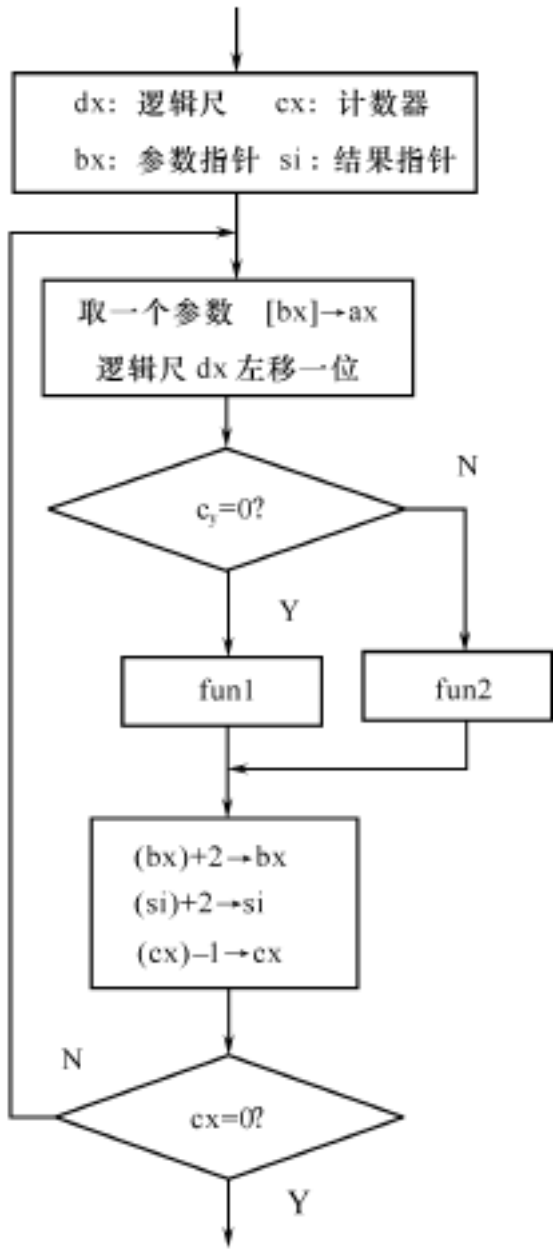


图 5 - 9　例 5. 11 的程序流程图

理对第 1, 2, 5, 7, 10 各参数值调用函数 1(Y = 2X) 处理, 对第 3, 4, 6, 8, 9, 11, 12 各参数值调用函数 2(Y = 4X) 处理, 预处理后的结果保存在 BLOCK 起始的缓冲区中。

分析: 由于分支条件不规则, 可建立一个与要求相对应的逻辑尺(位串) 来作为扩展控制条件。逻辑尺设为: 00110101110110000。

程序流程图如图 5 - 9 所示。程序如下:

```
.model small
.stack
.da a
    buffer     dw    11,22,33,44,55,66,77,88,99,10,20,30
    bl ock     dw    12 dup(?)
    count      equ   12
    logrul     equ   0011010110110000b
.cod
.st rtup
    ...
    mov        dx,   logrul
    mov        cx,   count
```

```
    lea     bx,   buffer
    lea     si,   block
again:  ov      ax,[bx]
        rcl     dx,1
        jc      anoth
        call    fun1
next:   mov     [si],ax
        inc     bx
        inc     bx
        inc     si
        inc     si
        loop    again
        ...
anoth: call     fun2
        jmp     next
fun1    proc
        add     ax,ax
        ret
fun1    endp
fun2    proc
        add     ax,ax
        add     ax,ax
        ret
fun2    endp
```

【例 5.12】　试编写一个程序,要求比较两个字符串 str1 和 str2 所含字符是否相同,若相同则显示“ match! ”,若不相同则显示“ no match! ”。

程序如下:

```
.model small
.stack
.data
      str1      db computer!
      n1 = $-str1
      str2      db computer!
      n2 = $-str2
      info1     db 0dh,0ah, match! $
      info2     db 0dh,0ah, no match! $
.code
.star up
     mov        al,n1
     cmp        al,n2
     jne        exit
     lea        si,str1
     lea         i,str2          ;初始化
     mov        cx,n1
lopa:
     mov        al,[si]
     cmp        al,[di]          ;工作部分
```

```
        jne     exit
        inc     si
                inc di          ;修改部分
                dec cx
        jnz     lopa            ;控制部分
        lea     dx,inf01
        mov     ah,9            ;显示信息“match!”
        int     21h
        jmp     retu
exit:
        lea     dx,inf02
        mov     ah,9            ;显示信息“no match!”
                int 21h
retu:
                .exit 0
end
```

5.4.3　多重循环程序设计

所谓多重循环,即循环体内再套循环,外层的循环称为外循环,内层的循环称为内循环。多重循环程序设计方法和单循环程序设计是一致的。

设计多重循环程序时,可从外循环到内循环一层一层地进行。在设计外层循环时,仅把内层循环看成一个处理框,当内层循环设计完之后,用其替换外层循环体中被视为一个处理框的对应部分,就可构成多重循环。下面举例说明多重循环程序的设计方法。

【例 5.13】　在以 BUF 为首址的字存储区中存放有 N 个有符号数,要求采用“冒泡法”将它们按从大到小的顺序排列在 BUF 存储区中,试编写其程序。

分析:冒泡排序算法从第一个数开始依次对相邻两个数进行比较,如次序对,则不交换两数位置;如次序不对则使这两个数交换位置。可以看出,第一遍需比较(N - 1)次,此时,最小的数已经放到了最后;第二遍比较只需考虑剩下的(N - 1)个数,即只需比较(N - 2)次;第三遍只需比较(N - 3)次,……整个排序过程最多需(N - 1)遍。

程序流程图如图 5 - 10 所示。程序如下:

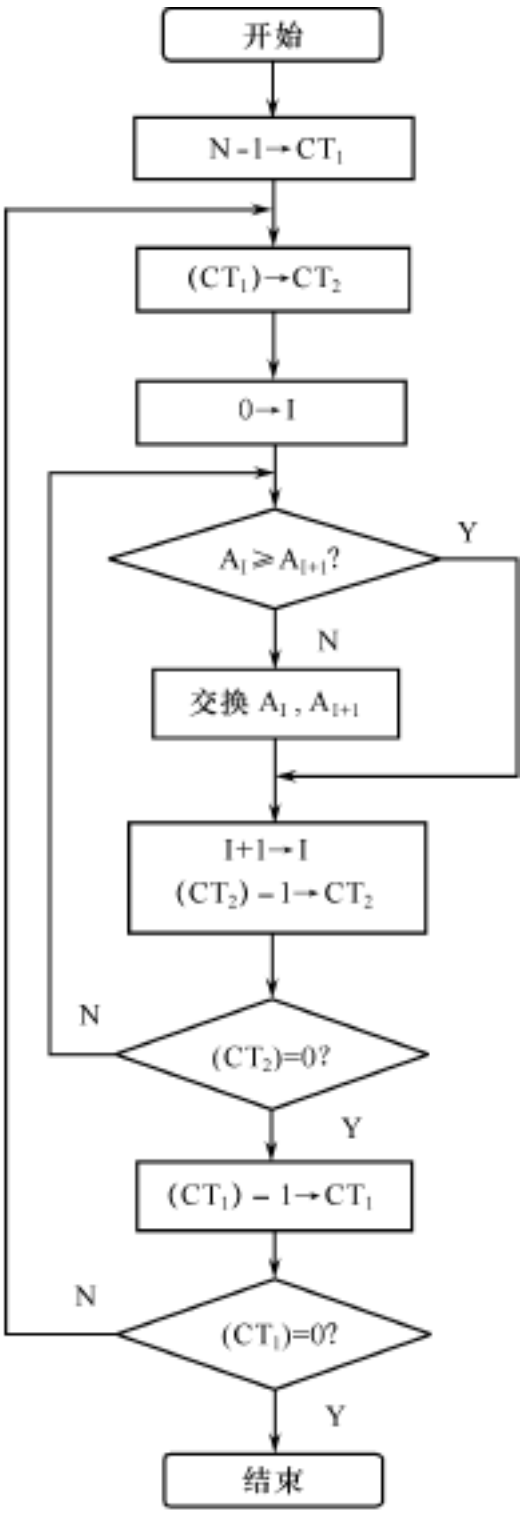


图 5 - 10　例 5.13 的程序流程图

```
.model small
.stack
.data
    buf dw13,-4,6,9,8,2,11,-8,-6,-20,30
    n=($-buf)/2
.code
```

```
.st rtup
    mov  cx,n
    dec  cx
loop1:
       mov  dx,cx
       mov  bx,0
loo 2:
    mov     ax,buf[bx]
    cmp     ax,buf[bx+2]
    jge     next
    xchg    ax,buf[bx+2]
    mov     buf[bx],ax
nex :
    add     bx,2
    dec     cx
    jne     loop2
    mov     cx,dx
    loop    loop1
.exit 0
end
```

【例 5.14】　编写如下矩阵相乘的程序。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 1 \\ b_4 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 1 \\ c_4 \end{bmatrix}$$

计算公式为:

$$c_i = \sum_{j=1}^{4} a_{ij} b_j \qquad (i = 1, 2, 3, 4)$$

展开后的形式为:

$$c_1 = a_{11} b_1 + a_{12} b_2 + a_{13} b_3 + a_{14} b_4$$
$$c_2 = a_{21} b_1 + a_{22} b_2 + a_{23} b_3 + a_{24} b_4$$
$$c_3 = a_{31} b_1 + a_{32} b_2 + a_{33} b_3 + a_{34} b_4$$
$$c_4 = a_{41} b_1 + a_{42} b_2 + a_{43} b_3 + a_{44} b_4$$

下面首先考虑编出计算一个 c_i(先固定 i = 1) 的程序。为了便于循环, 把矩阵 A 的元素按行依次相邻存放, 把向量 B 和 C 的元素也依次相邻存放, 在数据段里按如下方法定义数据项:

```
A    DB a11, a12, a13, a14
     DB a21, a22, a23, a24
     DB a31, a32, a33, a34
     DB a41, a42, a43, a44
B    DB b1, b2, b3, b4
C    DW 4 DUP( ?)
```

则它们的存储分配如下:

A[0] : a11	A[4] : a21	A[8] : a31	A[12] : a41
A[1] : a12	A[5] : a22	A[9] : a32	A[13] : a42
A[2] : a13	A[6] : a23	A[10] : a33	A[14] : a43
A[3] : a14	A[7] : a24	A[11] : a34	A[15] : a44
B[0] : b1	C[0] : c1		
B[1] : b2	C[2] : c2		
B[2] : b3	C[4] : c3		
B[3] : b4	C[6] : c4		

其中, 小写字母 a11, a12, . . . a44, b1, . . . b4 等均代表具体的无符号数。这样, 计算 c1 的框图如图 5 - 11 中的内层循环所示。紧接着应考虑 c2 的计算, 从框图上看, 计算 c2 和计算 c1 有两点不同: 一是数据 A 从 a21 开始而不是 a11 开始, 也就是从 A[4] 开始, SI 从 4 开始而不是从 0 开始; 二是结果存放在 C[2] 而不是 C[0] 。其中, 第一点不同, 由于在计算 c1 的过程中 SI 每次加 1, 到计算完 c1 时, SI 的值恰好是 4, 这一要求自然满足; 第二点不同, 可以利用基址寄存器 BX, 开始给 BX 送初值 0, 每循环一次加 2, 利用 C[BX] 间接寻址方式访问 ci 的方法解决。c3, c4 的计算也是一样的。这里出现了二重循环, 计算一个 ci 的循环为内循环、计算所有 ci 的循环为外循环。内、外循环都是用 CX 作为计数器, 因此需要将控制计算 4 个 ci 的 CX 的计数器暂时保存起来, 本程序通过堆栈保存和恢复。程序流程图如图 5 - 11 所示。

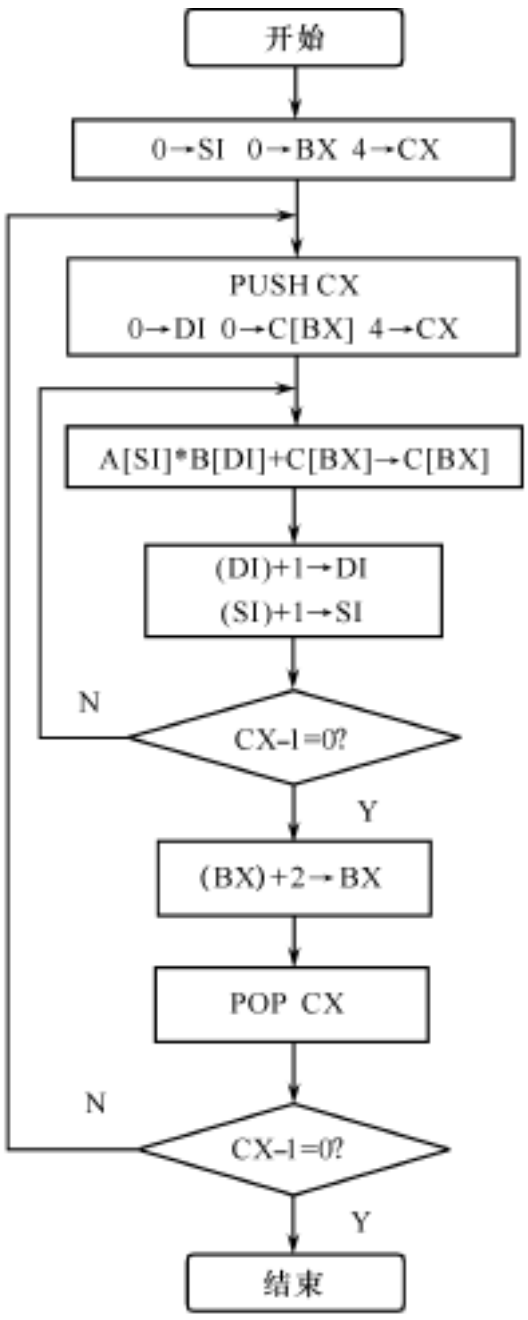


图 5 - 11　例 5.14 的程序流程图

根据框图很容易编写出如下的程序(对矩阵任意假定一组数据) :

```
.model small
.stac
      stapn  dw100 dup(?)
      top    equ length stapn
.data
       a     db    1,0,2,3
             db    0,1,1,0
             db    3,0,1,0
             db    4,2,0,1
       b     db    0,1,1,0
       c     dw    4 dup(?)
.code
.startup
       mov  ax,stack
```

```
        mov  ss,ax
        mov  sp,top
        mov  si,0                    ;si 作为 ai 指针,初值 0
        mov  bx,0                    ;bx 作为 ci 指针,初值 0
        mov  cx,4                    ;cx 作为外循环计数器,初值 0
loop1:
        push cx                      ;将 cx 压栈保存
        mov  di,0                    ;以 di 作为 bj 指针,赋值 0
        mov  word ptr c[bx],0
        mov  cx,4                    ;cx 也为内循环计数器
loop2:
        mov  ah,0
        mov  al,a[si]
        mul  b[di]
        add  c[bx],ax
        inc  si                      ;si 指针加 1
        inc  di                      ;di 指针加 1
        loop loop2                   ; 内循环结束
        add  bx,2                    ; bx 指针加 2
        pop  cx                      ;弹出 cx
        loop loop1                   ;外循环结束
.exit 0
end
```

5.4.4　用伪指令实现循环结构

循环控制结构也可以用伪指令来实现,其书写格式与高级语言的书写方式类似,循环伪指令的格式和含义如下。

(1) WHILE 型循环伪指令

```
.WH LE condition
        循环体的指令序列          ;条件“condition”成立时所执行的指令序列
.ENDW
```

其中.ENDW 与前面的.WHILE 相匹配,它标志着其循环体到此结束。

如果条件表达式 condition 在循环开始时就为假(false),那么,该循环体一次也不会被执行。

(2) REPEAT 型循环伪指令

格式 1:　REPEAT

```
        循环体的指令序列
.UNTIL condition
```

格式 2:　REPEAT

```
        循环体的指令序列
.UNTILCXZ [ condition]
```

REPEAT 型循环在执行完循环体后,才判定逻辑表达式 condition 的值。若该表达式的值为真,则终止该循环,并将执行伪指令.UNTIL[CXZ] 后面的指令,否则,将向上跳转到伪

指令. REPEAT 之后的指令,为继续执行其循环体做准备。

如果. UNTILCXZ 后面没有写逻辑表达式,那么,由. REPEAT-. UNTILCXZ 所构成的循环与用 LOOP 指令所构成的循环是一致的,它们都是以“ CX = 0 ”为循环终止条件。

如果. UNTILCXZ 后面书写了逻辑表达式,那么,该逻辑表达式的形式只能是:“ EXP1 == EXP2 ”或“ EXP1! = EXP2 ”。所以,这时由“ . REPEAT-. UNTILCXZ condition ”所构成的循环就与用 LOOPNE /LOOPE 指令所构成的循环是一致的,它们都是以“ condition || CX = 0 ”为循环终止条件。

和高级语言的 REPEAT 型的循环一样, . REPEAT-. UNTIL[CXZ] 的循环体也会至少被执行一次。

. WHILE-. ENDW 和. REPEAT-. UNTIL[CXZ] 的循环体内还可再含有循环伪指令,这样就构成了循环结构的嵌套。

(3) 终止循环伪指令

格式: BREAK

. BREAK . IF condition

该伪指令用来终止包含它的最内层循环。前者是无条件终止循环,后者是仅当逻辑表达式 condition 为真时,才终止循环。

```
. WH LE 1
    . . .
    . BREAK . IF condition
    . . .
ENDW
```

对于以上循环结构,当逻辑表达式 condition 为真时,该循环就会被终止。

(4) 循环继续伪指令

格式: CONTINUE

. CONTINUE . IF condition

该伪指令用于直接跳转到包含它的最内层循环的计算循环条件表达式的代码处。前者是无条件转移到计算循环条件表达式的代码处,后者是仅当条件表达式 condition 为真时,才进行这样的跳转。

```
. WH LE condition 1
    . . .
    . CONTINUE . IF condition2
    . . .
ENDW
```

对于以上循环结构,当逻辑表达式 condition2 为真时,则重新开始下一次的循环。

5.5　子程序设计

子程序是程序设计的基本概念。实际编程时,常把功能相对独立的程序段单独编写和调试,作为一个相对独立的模块供程序使用,这就是子程序,亦称过程,相当于高级语言中的

过程和函数,调用子程序的程序称为主程序(或称调用程序)。子程序可以简化程序结构,实现程序的模块化,缩短源程序长度,节省目标程序的存储空间,也可提高程序的可维护性和共享性。

汇编语言也提供了编写子程序的方法。与高级语言不同,在设计汇编子程序时,要考虑诸如参数传递、保护现场和恢复现场等实现细节问题。本节主要介绍子程序的定义、调用和返回以及子程序的参数传递方法等内容。

5.5.1　子程序定义

子程序的定义是由一对过程定义伪指令 PROC 和 ENDP 来完成的,其一般格式如下。

```
子程序    PROC [ NEAR | FAR]
          [保护现场]
          子程序体
          [恢复现场]
          RET
子程序名 ENDP
```

对子程序定义的具体规定如下。

"子程序名"必须是一个合法的标识符,并且二者要前后一致。

PROC 和 ENDP 必须是成对出现的关键字,它们分别表示子程序定义的开始和结束。

子程序的类型有近(NEAR)、远(FAR)之分,其默认的类型是近类型。

如果一个子程序要被另一段程序调用,那么,其类型应定义为 FAR,否则,其类型可以是 NEAR。显然,NEAR 类型的子程序只能被与其同段的程序所调用。

子程序至少要有一条返回指令,也可有多条返回指令。返回指令是子程序的出口语句,但它不一定是子程序的最后一条语句。

子程序名有段值、偏移量和类型 3 个属性。其段值和偏移量对应子程序的入口地址,其类型就是该子程序的类型。

例如,实现回车、换行功能的子程序,过程定义如下。

```
subr    proc
        push    ax          ;保护现场
        push    dx
        mov     dl,0dh      ; 回车控制字符0DH
        mov     ah,2        ;
        int     21h         ; 执行回车功能
        mov     dl,0ah      ; 换行控制字符0AH
        mov     ah,2        ;
        int     21h         ; 执行换行功能
        pop     dx          ;恢复现场
        pop     ax
        ret                 ;子程序返回
subr endp
```

【例 5.15】　生成 0 ~100 之间的随机数子程序。

随机数生成程序需要一个种子,通常可以借助机内的时钟计数来获得。通过 BIOS 的

INT 1AH 中断的功能 0, 可以在 CX: DX 中得到时钟计数值。

随机数生成的子程序如下:

```
; 子程序功能: 生成 0 ~100 之间的随机数
; 入口参数: 无
; 出口参数: BL 寄存器带回随机数
rand    proc
        push    cx
        push    dx
        push    ax
        sti
        mov     ah,0
        int     1ah         ; 读时钟计数器值送 CX: DX
        mov     ax,dx       ; 清高 6 位
and ah,3
        mov     dl,101      ; 除 101, 产生 0 ~100 之间的余数
        div     dl
        mov     bl,ah       ; 余数作随机数存入 bl 寄存器
        pop     ax
        pop     dx
        pop     cx
        ret
rand endp
```

编写子程序除了要考虑实现子程序功能的方法外, 还要养成书写子程序说明信息的好习惯。其说明信息一般包括以下几方面内容。

功能描述: 包括名称、功能、主要性能指标等。

入口及出口参数。

作用的寄存器和存储单元。

其中调用的子程序。

编者及编写日期等。

这些说明性信息虽然不是子程序功能的一部分, 但其他程序员可通过它们对该子程序的整体信息有一个较清晰的认识, 为准确地调用它们提供直接的帮助。与此同时, 也为实现子程序的共享提供了必要的资料。

5.5.2　子程序的调用和返回

子程序的调用和返回是由指令 CALL 和 RET 来完成的。为了保证子程序返回主程序时主程序能继续执行, 必须注意主程序现场的保护。现场是指主程序转到子程序前这一时刻主程序所使用的资源或状态, 如标志寄存器、通用寄存器及存储器单元的内容。通常在转到子程序前将它们压入堆栈, 以免子程序在执行时使用这些资源而发生冲突。而当子程序返回主程序时, 主程序的现场必须恢复, 即把它们从堆栈中弹出, 保持原来的内容不被改变, 主程序才能正确地继续执行。保护与恢复现场的工作通常安排在子程序中进行, 在子程序的开始处安排一串保护现场的语句, 在子程序返回前, 再恢复有关内容。

应特别注意的是, 为保证子程序的正确调用与返回, 除定义时需正确选择属性外, 还应

该注意子程序运行期间的堆栈状态。我们知道,当发生过程调用时,CALL 指令的功能之一是将返回地址压入堆栈;当子程序返回时,RET 则直接从当前栈顶取内容作为返回地址;而子程序中可能还有其他指令涉及到堆栈操作。因此,要保证 RET 指令执行前堆栈栈顶的内容刚好是过程返回的地址,即相应 CALL 指令压栈的内容,否则将造成不可预测的错误。

关于子程序的调用有两种特殊的情况,即子程序嵌套调用和子程序递归调用。在子程序调用其他子程序时,称为子程序嵌套,只要堆栈空间允许,嵌套层次不限。若子程序中又调用该子程序本身则称为递归调用,递归的深度亦与堆栈大小有关。

例如,实现把寄存器 AL 中存放的字符转换成大写的子程序 upper 如下。

```
;子程序功能:把 AL 中存放的字符变成大写
;入口参数:AL
;出口参数:AL
upper   proc
        cmp     al, a
        jb      over
        cmp     al, z
        ja      over
        sub     al,20h
over:   ret
upper   endp
```

主程序调用方式:

```
        mov     al, b
        call    upper       ;子程序返回时,(al) = B
        mov     al, 4
        call    upper       ;子程序返回时,al 的值保持不变
```

5.5.3　子程序的参数传递

子程序一般是完成某种特定功能的程序段。当一个程序调用一个子程序时,通常都向子程序传递若干个数据让它来处理;当子程序处理完后,一般也向调用它的程序传递处理结果,我们称这种在调用程序和子程序之间的信息传递为参数传递。调用程序向子程序传递的参数称为子程序的入口参数,子程序向调用它的程序传递的参数称为子程序的出口参数。

调用程序与子程序间通过参数传递建立联系,相互配合共同完成处理工作。传递参数的多少反映程序模块间的耦合程度。根据实际情况,子程序可以只有入口参数或只有出口参数,也可以同时存在入口参数和出口参数。参数的具体内容可以是数据本身(传数值)也可以是数据的存储地址(传地址)。程序和被调用子程序之间的参数传递方法是程序员自己或和别人事先约定好的信息传递方法。方便灵活的参数传递是子程序设计的关键环节之一。

在汇编语言中,常用的 3 种参数传递方法包括利用寄存器传递参数、约定存储单元传递参数和利用堆栈传递参数。下面分别进行讨论。

5.5.3.1　利用寄存器传递参数

由于 CPU 中的寄存器在任何程序中都是“可见”的,一个程序对某寄存器赋值后,在另

一个程序中就能直接使用,所以用寄存器来传递参数最直接、简便,也是最常用的参数传递方式。但由于 CPU 中寄存器的个数和容量都是非常有限的,所以该方法适用于传递较少的参数信息。

主程序在调用子程序前,先将需要传递的参数保存在某些通用寄存器中,然后再调用子程序,这样,子程序就可直接从寄存器中获得入口参数。同样,出口参数可以通过寄存器返回给主程序。

下面来看一个例子,要求通过编程将从键盘上输入的小写字母转换成大写后输出。为简化程序,我们将判断输入的字符是否为小写字母的工作编为一个子程序 compare,该子程序将判断的结果标志寄存器中的 CF 标志返回给主程序,CF = 0 表示是小写字母,CF = 1 表示不是小写字母;主程序通过 AL 寄存器将要判断的内容传递给子程序 compare。程序段如下。

```
.code
.startup
inpt:
    mov     ah,1
    int     21h             ;读入一个字符,送到 al,通过 al 传递参数
    call    compare         ;调用子程序 compare
    jc      input           ;cf =1,非小写字母,重新输入
    sub     al,32           ;cf =0,将小写字母转换为大写字母
    mov     dl,al
    mov     ah,02
    int     21h             ;输出显示转换后的大写字母
    mov     ah,4ch
    int     21h             ;返回 dos
;子程序功能:判断 AL 中存放的字符是否为小写字母
;入口参数:AL
;出口参数:CF 标志。AL 中的字符为小写字母时则 CF =0,否则 CF =1
compare:  cmp     al, a
          jb      setflag
          cmp     al, z
          ja      setflag
          clc
          ret
setflag:  stc
          ret
end
```

【例 5.16】　按 5 位十进制的形式显示寄存器 BX 中的内容,如果 BX 的值小于 0,则应在显示数值之前显示负号“ - ”。

定义 5 个字节的存储单元,用来存放转换后得到的 5 位十进制数。先判断 BX 是否小于零,如果是,则先显示负号“ - ”,再取 BX 的绝对值;采用除 10 得余数的方法,从低位向高位求出每位十进制位并保存在预先定义的存储单元中;最后输出数据的字符串。

```
;子程序功能:把寄存器 BX 的内容按十进制有符号数显示出来
;入口参数:BX
```

```
;出口参数:无
subda a segment
      db 5 dup( 0 ),0ah,0dh, $        ; 0ah、0dh:换行、回车
subdata       ends
dispbx        proc
              assume    ds:subdata
              push      ds
              push      dx
              push      cx
              push      ax
              mov       ax,subdata    ;取子程序所用的数据区段地址
              mov       ds,ax
              cmp       bx,0
              jge       next
              mov       dl,"-"
              mov       ah,2
              int       21h           ;显示负号"-"
              neg       bx            ;求-bx,使其值为正数
next:         mov       si,4
              mov       ax,bx
              mov       cx,10d
again:        xor       dx,dx
              idiv      cx            ;dx 存放余数,ax 存放商
              add       dl,0
              mov       [si],dl
              dec       si
              jge       again
              xor       dx,dx
              mov       ah,9
              int       21h           ; 显示 ds:dx 指向的字符串
              pop       ax
              pop       cx
              pop       dx
              pop       ds
              ret
    dispbx    endp
```

5.5.3.2　利用约定存储器传递参数

主程序与子程序之间可利用约定的一组存储单元来传递参数。在调用子程序时,当需要向子程序传递大量数据时,因受到寄存器容量的限制,就不能采用寄存器传递参数的方式,而要改用约定存储单元的传递方式。这种参数传递方式有点像情报人员和联络人员之间的传递信息方式,一个向指定地点放情报,另一个从指定地点取情报。这种方法适于参数较多的情况。

下面是采用约定存储单元传递参数的例子,所处理的数据不是直接传给子程序,而是把存储它们的地址告诉子程序。

【例 5.17】　编写一个子程序分类统计出一个字符串中的数字字符、字母和其他字符的

个数。该字符串的首地址用 DS:DX 来指定(以 0 为字符串结束),各类字符个数分别存放 BX, CX 和 DI 中。

```
;子程序功能:分类统计出字符串中的数字字符、字母和其他字符的个数
;入口参数: DS:DX 指向被统计的字符串
;出口参数: BX, CX 和 DI 分别保存数字字符、字母和其他字符的个数
count proc
        push    ax
        push    si
        xor     bx,bx
        xor     cx,cx
        xor     di,di          ;上三条指令使各类字符计数清零
        mov     si,dx
again:  mov     al,[si]
        inc     si
        cmp     al,0
        je      over
        cmp     al, 0
        jl      other
        cmp     al, 9
        jg      next
        inc     bx             ;数字字符个数加 1
        jmp     again
next:   call    upper          ;调用子程序把 al 中的字母变成大写字母
        cmp     al, a
        jl      other
        cmp     al, z
        jg      other
        inc     cx             ;字母个数加 1
        jmp     again
other:  inc     di             ;其他字符个数加 1
        jmp     again
over:   pop     si
        pop     ax
        ret
count   endp
```

主程序调用形式如下:

```
.model small
.data
    msg db  AfgBCDl0tt984/[y]3oiu(* &(5341 ,0
.code
.star up
     lea dx,msg             ;ds:dx 指向待统计的字符串
     call count             ;调用子程序统计出各类字符的个数
     …                      ;BX, CX 和 DI 分别保存数字字符、字母和其他字符的个数
.exit 0
end
```

5.5.3.3　利用堆栈传递参数

由于堆栈具有先进后出、后进先出的特性，故多重调用中各重参数的层次很分明，很适于参数多且子程序有嵌套、递归调用的情况。通常情况下，用堆栈传递入口参数，用寄存器传递出口参数。

当用堆栈传递入口参数时，要在调用子程序前把有关参数依次压栈，子程序从堆栈中取到入口参数。

(1) 主程序用堆栈传递入口参数

```
push    参数 1
push    参数 2
...
push    参数 n      ; 把 n 个字的参数压栈
call    subpro      ; 调用子程序 subpro
```

主程序将参数压栈后的堆栈状态如图 5 - 12 所示。

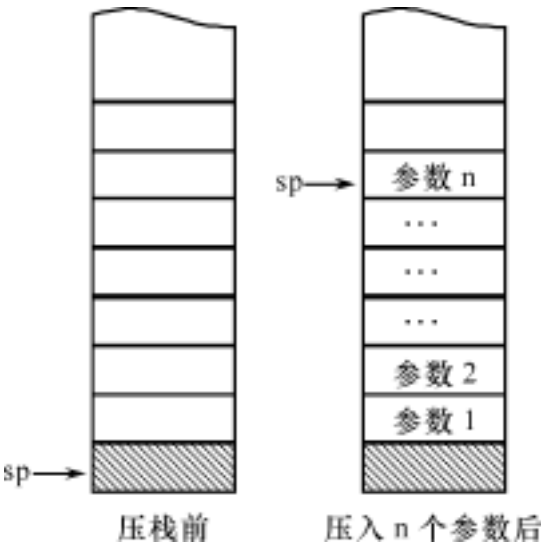


图 5 - 12　执行 CALL 之前堆栈

(2) 子程序在堆栈中取入口参数

子程序调用分为段内调用和段间调用两种方式，对于这两种不同的调用方式，获取入口参数的方法略有不同。

对于段内调用子程序，CALL 指令只把返回地址的偏移量 IP 压栈；对于段间调用子程序时，CALL 指令则会把返回地址的偏移量 IP 和段寄存器 CS 的内容都压栈。在进入子程序后，为了能读取传递过来的参数，需要用 BP 来访问堆栈，所以要先保护 BP 原来的值，然后再把当前 SP 的值传送给 BP。进入子程序后，堆栈的状态如图 5 - 13 所示。

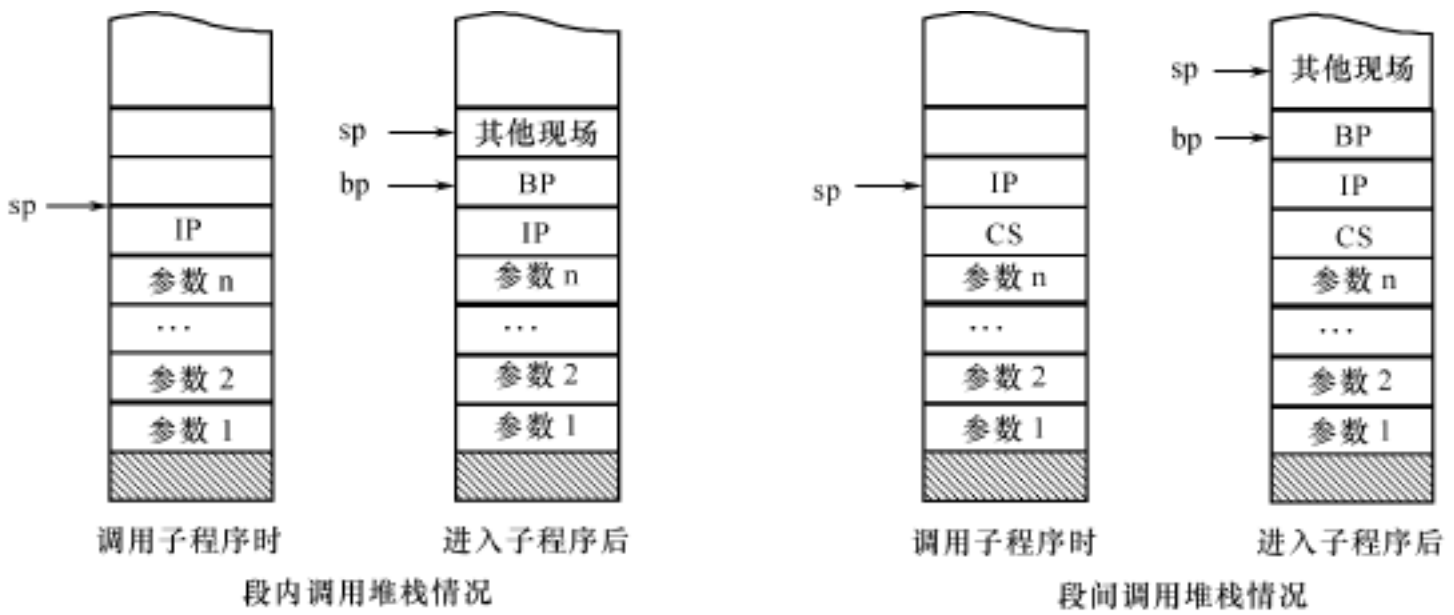


图 5 - 13　子程序看到的堆栈情况

显然，对于段内子程序调用来说，参数 i 在堆栈中的地址可以用[BP + 4 + 2* (n - i)] 来获得。而对于段间子程序调用，参数 i 的地址可用[BP + 6 + 2* (n - i)] 来表示。

```
subpro  proc    near
        push    bp              ; 保护寄存器 bp
        mov     bp, sp          ; 用寄存器 bp 来访问堆栈
```

```
            ...                                    ;保护其他寄存器的指令
            mov     ax, [bp +4 +2* (n - 1)]        ;读取参数 1
            mov     ax, [bp +4 +2* (n - 2)]        ;读取参数 2
            ...
            mov     ax, [bp +4]                    ;读取参数 n
            ...
subpro      endp
```

【例 5. 17】　两个数组位于数据段中, 实现两个数组的分别求和。

将求和程序定义为过程, 且主程序和过程分别安排在两个不同的代码段中(即过程是 FAR 类型的) 利用堆栈实现主程序向过程的参数传递, 要特别注意配合好子程序中参数的读取和返回。

程序如下:

```
.model small
.stack
      spae    dw20 dup(?)
      top     equ length spae
.data
      ary1    db 10 dup(?)          ;定义数组 1
      sum1    dw?
      ary2    db 100 dup(?)         ;定义数组 2
      sum2    dw?
.code
.startup
      str:    push ds
      mov     ax,0
      push    ax
      mov     ax,data
      mov     ds,ax
      mov     ax,size ary1
      push    ax                    ;sum过程的入口参数 1 进栈
      mov     ax,offset ary1
      push    ax                    ;sum过程的入口参数 2 进栈
      call    sum
      mov     ax,size ary2
      push    ax
      mov     ax,offset ary2
      push    ax
      call    sum
      hlt
sum   proc    far
      push    ax
      push    bx
      push    cx
      push    bp
      mov     bp,sp
      pushf
```

```
        mov     cx,[bp+14]          ;数组长度 size 由栈 → cx
        mov     bx,[bp+12]          ;数组存放的地址偏移量入栈到 bx
        mov     ax,0
and:    add     al,[bx]
        inc     bx
        adc     ah,0
        loop    and
        mov     [bx],ax             ;数组之和送到结果处
        popf                        ;恢复现场
        pop     bp
        pop     cx
        pop     bx
        pop     ax
        ret     4                   ;返回主程序并废除参数 1 和参数 2
    sumendp
end
```

本程序执行过程中,堆栈变化情况如图 5 - 14 所示。

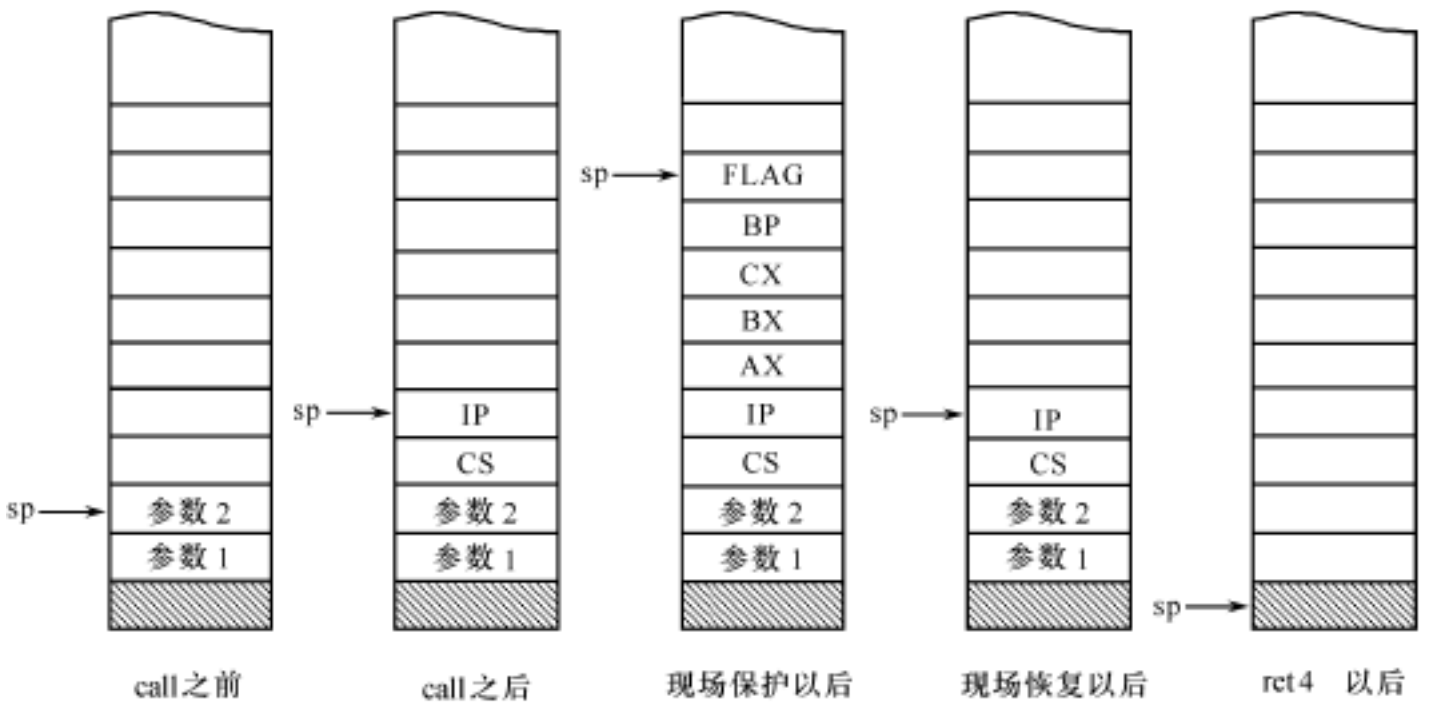


图 5 - 14　堆栈变化示意图

5.5.4　子程序嵌套与递归

1．子程序的嵌套

子程序不但可以被主程序调用,而且也可以被其他子程序调用。在一个子程序中调用另一个子程序被称为子程序的嵌套调用。只要堆栈空间允许,嵌套层次不限。

子程序的嵌套调用示意图如图 5 - 15 所示。

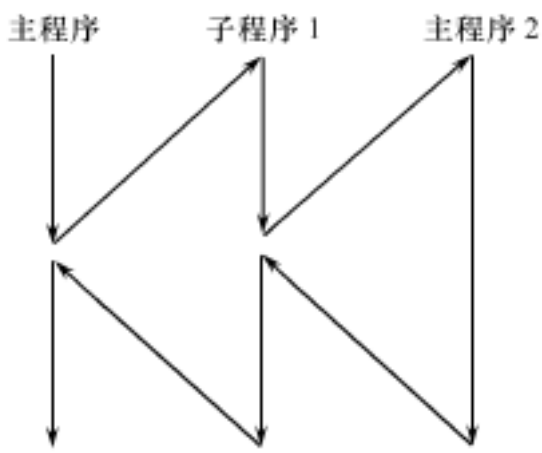


图 5 - 15　子程序嵌套

2．子程序的递归

子程序的递归调用是指一个子程序直接或间接地

调用自己。递归子程序一般对应于数学上对函数的递归定义,它往往能设计出效率较高的程序,完成相当复杂的计算,因而是很有用的。

子程序调用它自己时称为直接递归;当子程序 A 调用其他子程序时,这个被调用的子程序又调用子程序 A,此时,称为间接递归。递归的深度亦与堆栈大小有关。

下面以阶乘函数为例,说明递归子程序的设计方法。

【例 5.19】　要求编制计算 N! (N>0)的程序。N! 的递归定义可以表示如下:

$$0! = 1$$

$$N! = N * (N-1)! \qquad (N>0)$$

下面根据递归定义来设计该程序。求 N! 本身是一个子程序,由于 N! 是 N 和(N-1)! 的乘积,所以为求(N-1)! 必须递归调用求 N! 的子程序,但每次调用所使用的参数都不相同。递归子程序的设计必须保证每次调用都不破坏以前调用时所用的参数和中间结果,所以一般把每次调用的参数、寄存器内容以及所有的中间结果都存放在堆栈中。通常把一次调用所保存的信息称为帧,一般一帧包括所保存的寄存器内容、参数或参数地址和中间结果等,每次调用把一帧信息存入堆栈。递归子程序中还必须包括基数的设置,当调用参数达到基数时还必须有一个条件转移指令实现嵌套退出,保证能按反向次序退出并返回主程序。

根据这些要求可画出本例的程序框图,如图 5-16 所示。

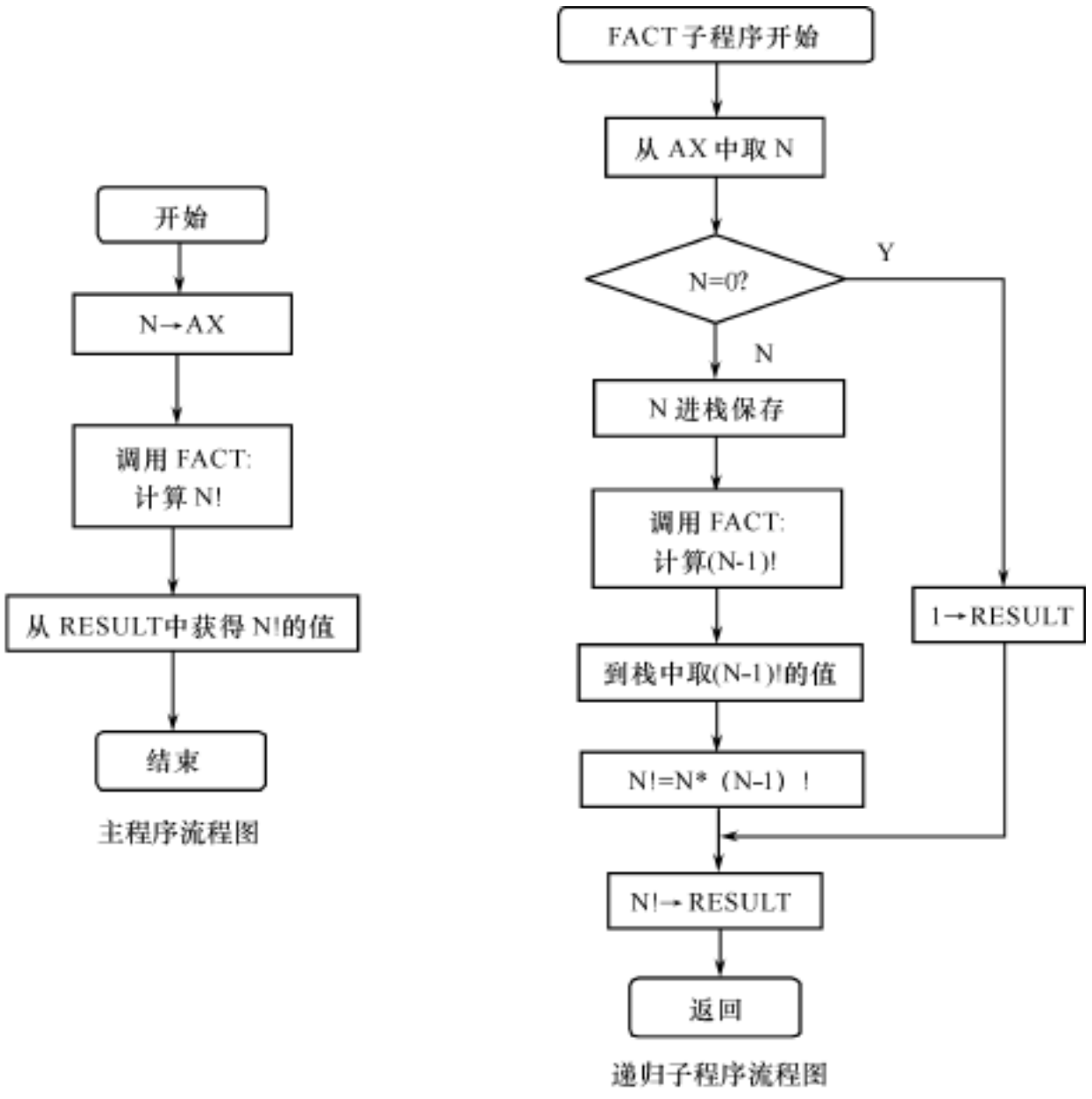


图 5-16　计算 N! 的流程图

程序如下:

```
.model small
.stack
.data
        n           dw5
result  dw?
.code
.startup
        mov         ax,n            ;N通过寄存器ax传递到子程序
        call        fact            ;调用求N! 的子程序
        mov         ax,result;
.exit 0
fact    proc
        cmp         ax,0
        jne         l1
        mov         result,1        ; 0! =1
        jmp         exit            ;
l1:     push        ax
        dec         ax
        call        fact            ;求(N-1)!
        pop         ax
                    mul result      ; 计算N*(N-1)!
                    mov             result,ax
exit:               ret
fact    endp
end
```

5.6　模块化程序设计

按照软件工程的思想,一个复杂的软件应该进行模块化设计,这样可以将一个复杂的问题分解为成若干相对简单的问题,便于软件的开发和管理,同时也有利于软件质量的提高。模块化程序设计就是将一个大的软件按功能划分为许多功能相对独立的模块,模块间按统一规范连接,每个模块分别编写和调试,最后连接成一个完整的软件。在编写大型的汇编语言程序时,采用模块化程序设计方法显得十分必要,本节将简要介绍汇编语言支持的模块化程序设计方法。

5.6.1　模块划分

模块化程序设计的首要问题是合理地划分模块,这就要求将一个复杂问题进行分解,确定功能模块和接口关系。模块划分的一般原则如下。

模块功能相对独立。每个模块的功能要明确,大小要适中,独立性要强,交换的接口信息要少,最好只有一个入口和一个出口。

模块间的关系要明确。各模块最好再分层,形成树型层次结构。即上层模块可调用下层模块,下层模块可返回上层模块,反之则不然。这就使得各层间不会构成循环。

程序中易变化的部分与不易变化的部分要分开,形成不同的模块,这样便于软件的升级。例如,系统软件中把与 CPU 有关的部分分出来形成一个专门模块。

MASM 汇编程序提供了两种模块划分的方法。一种是源程序级的模块划分,MASM 允许把一个大的源程序分别放在几个源程序文件中,但每个文件不能独立汇编和执行,汇编时必须通过包含伪指令(INCLUDE) 将这些源程序文件结合起来,统一汇编后形成一个目标文件。这样划分的好处是便于源程序的管理与维护,同时,也利于这些文件的重复应用。

另一种模块划分方法是目标代码级的,每个模块可以单独编写和调试,形成若干个目标文件,最后由连接程序将这些目标文件连接起来形成一个完整的可执行文件。通常所说的模块化程序设计指的是后一种方式。

5.6.2　源程序文件包含

MASM 汇编语言支持的源程序包含的伪指令(INCLUDE) 与 C 语言中包含语句的作用类似,即将 INCLUDE 指令指定的源程序文件的内容插入到该伪指令所在位置。包含伪指令的格式为:

INCLUDE 源程序文件名

假如一个汇编程序由 F. ASM、F1. ASM、F2. ASM 3 个源程序文件组成,其中 F. ASM 为主程序文件,其余两个分别是不同类型的子程序文件,我们就可以方便地将不同的子程序文件插入到主程序文件中,如图 5 - 17 所示。

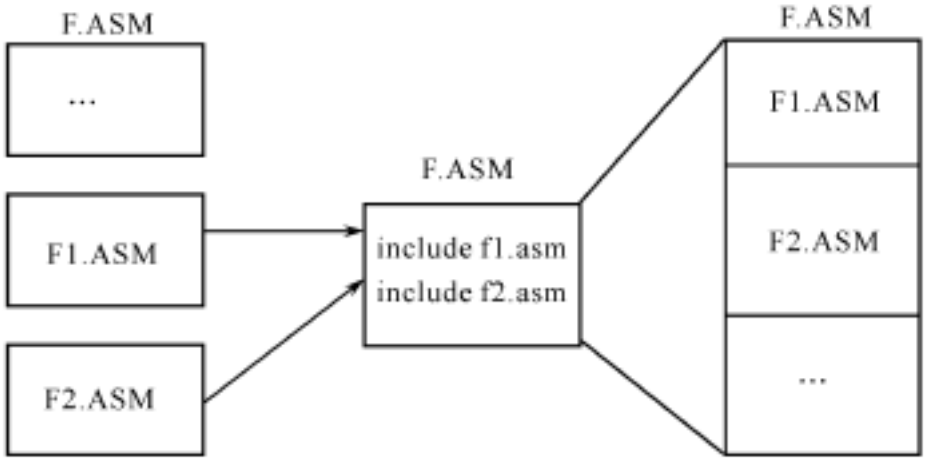


图 5 - 17　文件包含示意图

在 F. ASM 文件中,利用两条包含伪指令将 f1. asm、f2. asm 这两个文件包含到 F. ASM 文件中,此时只要对 F. ASM 进行汇编、连接后就能生成一个可执行程序 F. EXE。

在包含伪指令中,文件名可以含有路径,用来指明文件的存储位置,如果没有路径,MASM 则先在汇编命令行参数指定的目录下寻找,然后在当前目录下寻找,最后还会在环境参数 INCLUDE 指定的目录下寻找。

利用 INCLUDE 伪指令包含其他文件,其实质仍然是一个源程序,只是分成几个文件书写,被包含的文件不能独立汇编。因此,合并的源程序之间的各种标识符,如标号和名字等,应该统一规定,不能发生冲突。

5.6.3　模块间的连接

模块间的连接就是将多个相对独立的源程序文件分别单独汇编，形成若干个目标文件(.OBJ)，然后利用连接程序(LINK)将多个目标文件连接起来，生成一个可执行文件(.EXE)。连接程序的使用方法如下。

命令格式：LINK 目标文件1 + 目标文件2 + …

功能：实现对目标文件1、目标文件2的连接，生成一个可执行文件。

说明：多个目标文件中，只能有一个是主程序模块，其余的应是子程序模块，程序的执行总是从主程序模块开始。

例如，一个汇编程序由F.ASM、F1.ASM、F2.ASM 3个源程序文件组成，其中F.ASM为主程序文件，其余两个分别是不同类型的子程序文件。我们首先对这3个文件进行汇编，分别得到3个目标文件F.OBJ、F1.OBJ和F2.OBJ。然后利用LINK程序进行连接：

```
LINK F.OBJ + F1.OBJ + F2.OBJ
```

最后便可得到一个可执行文件F.EXE。

利用模块间连接方式开发源程序时，必须注意以下几个问题。

1．全局符号的使用

单个模块中使用的符号(变量、过程等)为局部符号，一个模块中定义的符号如不另加明，均为局部符号，局部符号只能在定义它的模块中使用。

多个模块间可共同使用的符号为全局符号。在大型程序开发过程中，一个文件可能要利用另一个文件定义的变量或过程，为了实现这样的调用，必须将相应的变量或过程声明为全局符号。

PUBLIC伪指令用于说明某个变量或过程可以被别的模块使用，其格式为：

```
PUBLIC　标识符[,标识符...]
```

EXTERN伪指令用于说明某个变量或过程是在其他模块中定义的，其格式为：

```
EXTERN　标识符:类型,[标识符:类型,...]
```

其中，标识符是变量名、过程名等；类型是byte/word/dword(变量)或near/far(过程)。在一个源程序中，public/extern语句可以有多条。各模块间的public/extern伪指令要互相配对，并且指明的类型互相一致。

2．段属性的匹配

由于各个文件独立汇编，所以子程序文件也必须定义在代码段中，也可以具有局部的数据变量。

采用简化段定义格式，因为默认的段名(如_TEXT))、类别(如CODE)相同，组合类型都是PUBLIC，所以只要采用相同的存储模式，很容易实现正确的调用。

完整段定义格式中，为了实现模块间的段内近调用(NEAR类型)，各自定义的段名、类别必须相同，组合类型都是PUBLIC，因为这是多个段能够组合成一个物理段的条件。在实际的程序开发中，各个模块往往由不同的程序员完成，不易实现段同名或类别相同，因此最好将所有的公用过程说明为远调用类型(即FAR类型)。

定义数据段时，同样也要注意这个问题。当各个模块的数据段不同时，要正确设置数据段DS寄存器的段基地址。

3．模块间的参数传递问题

模块间传递参数的基本方法与子程序间的参数传递方法相似。可以用寄存器或堆栈的方法传递数据或数据缓冲区指针，当然也可以用全局变量传递参数。

习　　题

1．编写汇编语言程序的一般步骤是什么?

2．从程序结构上看，汇编语言程序分为哪 4 种基本结构形式? 简要说明这 4 种结构的特点和用途。

3．编写一个程序段，将 BH 和 BL 中的内容互换。

4．编程实现比较两个带符号数的大小，并将较大的数送 MAX 单元保存。

5．把下列 C 语句改写成对应的汇编语言程序段(其中: 变量都为 16 位整型) 。

(1) h = (key & 0xFF00) >> 8　　(2) k = (k + 1 - 0xabcd) / 56

(3) for (s = 0, i = 100; i > 0; i++) s += i * 2

6．假设内存中有 3 个整型变量 a、b 和 c，试编写一个程序，判断它们能否构成一个三角形。

7．假设有 3 个无符号字存放在以 Buffer 为开始的缓冲区中，编写一个程序把它们按从低到高排序。

8．编写一个程序，将一个字符串(字符串以 0 结尾) 中的所有小写字母转换为对应的大写字母，并统计转换次数。

9．编写程序，找出自 BUF 存储单元开始的 50 个无符号数中最大者和最小者，分别存入 MAX 单元和 MIN 单元中。

10．编写一个程序，对从内存单元 BUFFER 开始的 100 个字求和，并把结果存入程序中的变量 SUM 中。

11．编写一个程序，假设从 Buff 开始存放了 100 个字，编写一个程序统计出其正数、0 和负数的个数，并把它们分别存入 N1、N2 和 N3 中。

12．子程序的参数传递有哪些方法? 试简单进行比较。

13．编写一个程序，从键盘上输入一个十进制数，然后把该数以十六进制形式显示在屏幕上。

14．编写程序，将一个包含有 20 个有符号数据的数组 arrayM 分成两个数组: 正数数组 arrayP 和负数数组 arrayN，并分别把这两个数组中的数据个数显示出来。

15．分别编写计算两个数 X 和 Y 的最小公倍数和最大公约数的子程序，并编写相应的调用程序。

16．分别编写求如下函数的子程序，并编写相应的调用程序。

(1) $SUM = \sum_{i=n}^{m} i$　　(2) $SUM = \sum_{n=1}^{20} (2n+1)^2$　　(3) $SUM = \sum_{n=1}^{6} n!$

17．分别编写与如下 C 语言函数功能相同的汇编子程序，并编写相应的调用程序。

(1) strcpy(buf1, buf2)　　(2) strcmp(str1, str2)　　(3) scanf("%d", &x)

18．编制一个子程序，把一个 16 位二进制数用十六进制形式在屏幕上显示出来，分别用寄存器传递、变量传递和堆栈方法传递实现，并用一个主程序验证它的正确性。

19．简述模块化程序设计的基本方法，不同模块间的连接应注意哪几方面的问题。

20．采用模块化的程序设计方法实现 18 题的功能。

第 6 章　输入/输出与中断控制

输入输出设备是计算机的重要组成部分,是人—机交互功能的主要承担者。在早期的计算机系统中,通常把输入输出设备或功能作为次要的部分,而把 CPU 作为主要研究对象。但现在随着输入输出设备的日益丰富,功能要求越来越复杂,输入输出部分在整个计算机系统中的地位也得到了进一步提高。

输入输出是一个完整应用程序的重要组成部分,是交互式应用程序不可缺少的组成部分。在用高级语言编程时,程序员可直接用输入输出语句来完成键盘输入、屏幕显示或打印输出等需求,而无需关心这些输入输出语句是如何实现的,因为编译程序会自动把这些语句转换成相应的输入输出指令。但如果用汇编语言编写程序情况就不同了,因为汇编语言是与机器有关的程序设计语言,要编写出具有输入输出功能的代码段就必须清楚 CPU 为输入输出提供了哪些指令,或计算机系统提供了哪些可直接使用的功能调用。

本章介绍了 I/O 的基本概念和 I/O 指令,叙述了中断的概念及其工作过程,列举了计算机系统中若干个常用的中断及其功能调用,详细介绍了 I/O 程序的编程设计方法并给出了典型的 I/O 程序的例。

6.1　I/O 概述

6.1.1　I/O 接口

对于系统中的每一台外设,都需要通过 I/O 接口实现与 CPU 的连接。I/O 接口通常含有设备状态寄存器、设备控制寄存器和设备数据寄存器 3 类寄存器。各寄存器的功能如下。

数据寄存器:保存处理器与外设间交换的数据。

控制寄存器:处理器通过它对外设进行控制,也称命令寄存器。

状态寄存器:外设的当前工作状态通过它向处理器提供。

I/O 接口实现了处理器与外设之间的连接。通过 I/O 接口,处理器可以接受从 I/O 设备输入的数据和状态信息,也可向 I/O 设备发送数据和控制信息。I/O 接口示意图如图 6 - 1 所示。

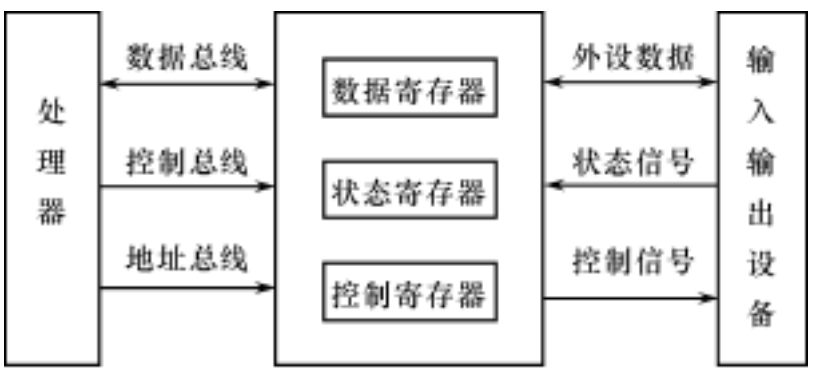


图 6 - 1　I/O 接口示意图

6.1.2　端口编址方式

I/O 接口呈现给程序员的是各种可编程寄存器。在涉及外设操作的输入输出程序中,各种设备寄存器以 I/O 地址(端口)体现;这些设备寄存器在 I/O 空间中均被指定一个固定

地址。CPU 在寻址外设时,便使用专门的 I/O 指令来访问端口,达到交换信息的目的。这种寻址方式被称为端口寻址。

I/O 端口是 CPU 与输入输出设备交换数据的场所,通过 I/O 端口,处理器可以接受从输入设备输入的信息,也可向输出设备发送信息。在计算机系统中,为了区分各类不同的 I/O 端口,采用不同的数字对其进行编号,这种对 I/O 端口的编号就称为 I/O 端口地址。每次可交换一个字节数据的端口称为字节端口,每次可交换一个字数据的端口称为字端口。表 6 - 1 列举了几个重要的 I/O 端口地址。

表 6 - 1　几个重要的 I/O 端口地址

端口地址	端口名称	端口地址	端口名称
020H～023H	中断屏蔽寄存器	378H～37FH	并行口 LPT2
040H～043H	时钟/计数器	3B0H～3BBH	单色显示器端口
060H	键盘输入端口	3BCH～3BFH	并行口 LPT1
061H	扬声器(0,1 位)	3C0H～3CFH	VGA/EGA
200H～20FH	游戏控制口	3D0H～3DFH	CGA
278H～27FH	并行口 LPT3	3F0H～3F7H	磁盘控制器
2F8H～2FFH	串行口 COM2	3F8H～3FFH	串行口 COM1

在 Intel 公司的 CPU 家族中,I/O 端口的地址空间可达 64 KB,即可有 65 536 个字节端口,或 32 768 个字端口。这些地址不是内存单元地址的一部分,不能用普通的访问内存指令来读取其信息,而要用专门的 I/O 指令才能访问它们。虽然 CPU 提供了很大的 I/O 地址空间,但目前大多数微机所用的端口地址都在 0～3FFH 范围之内,其所用的 I/O 地址空间只占整个 I/O 地址空间的很小部分。

6.1.3　I/O 指令

由于 I/O 端口地址和内存单元地址是相互独立的,这些端口地址不能用普通的访问内存指令来访问其信息,所以,在 CPU 的指令系统中就专门设置了 I/O 指令来存取 I/O 端口的信息。

1. 输入指令

格式: IN AL/AX, PortNo/DX

该指令的作用是从端口中读入一个字节或字并保存在寄存器 AL 或 AX 中。如果某输入设备的端口地址在 0～255 范围之内,那么,可在指令 IN 中直接给出,否则,要把该端口地址先存入寄存器 DX 中,然后在指令中由 DX 来给出其端口地址。

例如:

```
in      al,60h          ;从端口60h读入一个字节到al中
in      ax,20h          ;把端口20h、21h组成的字读入ax
mov     dx,2f8h
in      al,dx           ;从端口2f8h读入一个字节到al中
in      ax,dx           ;把端口2f8h、2f9h组成的字读入ax
```

2. 输出指令

格式: OUT PortNo/DX, AL/AX

该指令的作用是把寄存器 AL 或 AX 的内容输出到指定端口。如果某输出设备的端口地址在 0 ~255 范围之内, 那么, 可在指令 OUT 中直接给出, 否则, 要把该端口地址先存入寄存器 DX 中, 然后在指令中由 DX 来给出其端口地址。

例如:

```
out     61h,al      ;把 al 的内容输出到端口 61h 中
out     20h,ax      ;把 ax 的内容输出到端口 20h、21h 中
mov     dx,3c0h
out     dx,al       ;把 al 的内容输出到端口 3c0h 中
out     dx,ax       ;把 ax 的内容输出到端口 3c0h、3c1h 中
```

6.1.4　I/O 控制方式

由于外设的工作特点, 处理器与之交换数据就不像与存储器那样简单。具体的数据传送方式有主要由软件程序控制的直接查询和中断方法, 还有主要由硬件完成的 DMA 方法。

1. 无条件传送方式

对于一些接口电路较简单的外设, 不需查询外设的状态即可实现数据传送。对于这类外设, 在程序中可直接用 IN 或 OUT 指令实现数据传送, 这种方式称为无条件传送或称为程序直接控制输入输出方式。

2. 查询传送方式

在 CPU 与外设不能保持同步的情况下, 事先应查询输入设备是否准备好或输出设备是否忙。下面给出两个查询式传送的例子。若输入设备准备好, 则可输入, 否则继续等待并查询, 直到输入设备准备好, 才能进行数据输入, 如图 6 - 2(a) 所示; 若输出设备“ 忙 ”, 则等待并查询, 若不忙, 则可输出数据, 如图6 - 2(b) 所示。

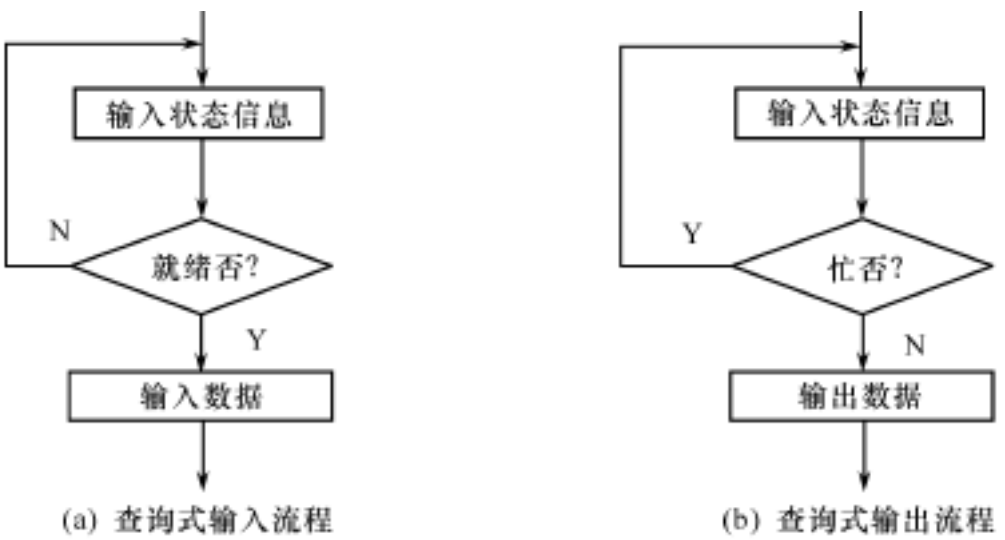


图 6 - 2　查询式 I/O 传送方式

3. 中断传送方式

在查询传送方式中, CPU 不停地查询外设, 实际上是一种时间的浪费。这是因为查询到外设未准备好时, CPU 要等待, 不能做别的工作, CPU 自始至终与外设串行工作。当 CPU 速度越高时, 与外设低速度的矛盾就越突出。

为提高 CPU 工作效率,可采用中断传送方式,即 CPU 启动外设后,不再等待外设工作的完成,而是继续执行主程序。使得 CPU 与外设可以并行工作,当输入设备已准备好或输出设备已空闲时,由外设向 CPU 发出中断请求,CPU 暂停原来执行的程序(实现中断),转去执行输入或输出操作(中断服务),待输入或输出完成,立即返回原程序并继续执行,从而提高了 CPU 的利用率。当然 CPU 还可以与多台外设并行工作,不过,每中断一次,传送一次数据,其效率并不高。因此,中断传送方式一般适用低速 I/O 设备。

4. DMA 方式

此方式适用于高速 I/O 设备(如软、硬磁盘机及高速通信机等)与内存储器直接交换数据的场合,数据是成批传送的。其过程是:先把数据在高速外设中存放的起始位置、数据在内存中的起始位置、待传送的数据个数等参数送给外设相应的寄存器,然后启动外设,准备直接传送数据,当高速的外设准备好时,便向 CPU 发一个直接传送的请求信号,CPU 收到信号时,便让出总线控制权,使外设与存储器在很短时间内完成批量数据交换。交换完毕后,CPU 又收回总线控制权,进行下一步有关处理。

6.2　简单 I/O 程序举例

在编写 I/O 控制程序时,程序员需要了解 I/O 接口电路的基本工作原理,I/O 接口呈现给程序员的是各种 I/O 端口。因此,程序员需要理解和掌握外设占用了哪些端口,各个端口用于交换什么信息,然后,根据具体情况采用不同的 I/O 控制方式设计 I/O 程序。下面通过一些例子来说明 I/O 程序的设计方法。

【例 6.1】　用一个 8 位输出锁存器控制 8 个发光二极管,使发光二极管依次逐个地闪亮,每个二极管在一段规定的时间内发光。设输出锁存器的地址为 84H。

分析:发光二极管是一种简单的外设,可以认为它是始终就绪的,只要程序将控制码写入到输出锁存器,发光二极管便能按输出锁存器中指定的控制码发光。因此,可以采用无条件传送方式编写控制程序如下:

```
start:   mov   al,01h          ;控制代码 01h→al
next:    out   84h,al          ;(al)→84h
         call  delay           ;延时一定时间
         rol   al,1            ;控制代码循环左移 1 位
         jmp   next            ;无条件转 next 继续程序循环
delay:   proc                  ;定义延时子程序
         push  ax
         mov   ax,count
time:    dec   ax
         jnz   time
         pop   ax
         ret                   ;返回调用程序
delay    endp                  ;子程序调用结束
```

【例 6.2】　PC 扬声器声音的控制。分别编写控制扬声器“响”与“不响”的子程序,主程序首先让扬声器声“响”,然后用户在键盘上按任何键后声音停止。

分析:在 PC 接口电路中,扬声器发声与否,是由 PB 端口(端口地址为 61H)的 PB_0 和

PB_1 两位控制的，PB_0 和 PB_1 对应数据位 D_0 和 D_1。$PB_0PB_1 = 11$ 的控制扬声器发声，而 $PB_0PB_1 = 00$ 时则扬声器不响。对扬声器的控制可以采用程序直接控制的方式，控制程序如下：

```
.model tiny                ;采用微型模式,形成com格式的程序
.code
.startup
call speaker_on            ;打开扬声器
mov ah,1                   ;等待按键
int 21h
call speaker_off           ;关闭扬声器
.exit 0
speaker_on proc            ;打开扬声器子程序
    push ax
    in al,61h              ;读取原来控制信息
    or al,03h              ;PB0PB1 =11,其他位不变
    out 61h,al             ;控制扬声器发声
    pop ax
    ret
speaker_on endp
speaker_off proc           ;关闭扬声器子程序
    push ax
    in al,61h              ;读取原来控制信息
    and al,0fch            ; PB0PB1 =00,其他位不变
    out 61h,al             ;关闭扬声器声音
    pop ax
    ret
speaker_off endp
end
```

【例 6.3】 假设从某输入设备上输入一组数据送缓冲区，若缓冲区已满则显示信息“buffer overflow”，然后结束。设该设备的端口地址如下。

启动端口地址：FCH

数据端口地址：F8H

状态端口地址：FAH

分析：对于这个输入设备，首先由程序控制启动，然后检查数据是否准备就绪，若输入数据准备好，则可输入，否则等待并查询，直到输入设备准备好时，才能进行数据输入。每输入一组数据后，应检查缓冲区是否满，不满则进行下一个数据的输入，若缓冲区已满，则显示信息“buffer overflow”，然后程序结束。查询式输入控制程序如下：

```
.model small
.stack
        stapn  dw    100 dup(?)
        top    equ   length stapn
.data
        messi  db    "buffer overflow","$"
        buff   db    60 dup(?)
```

```
.code
.startup
        mov     bx,offset buff          ;送缓冲区指针
        mov     cx,60                   ;送计数初值
        out     0fch,al                 ;启动设备
wait:
        in      al,0fah                 ;查询状态,ready=0,则等待
        test    al,01h
        jz      wait
        in      al,0f8h                 ;输入数据
        mov     [bx],al
        inc     bx
        loop    wait                    ;检测缓冲区是否满,不满再输入
        mov     dx,offset messi         ;缓冲区满,输出标志字符串
        mov     ah,09h
        int     21h
.exit 0
end
```

【例 6.4】　编写一个程序段,控制打印机输出字符。设该打印机的端口地址如下。

数据端口地址: 378H

状态端口地址: 379H

启动端口地址: 37AH

分析: 对于打印设备,只有当前的一个字符打印完成后,才能取出下一个字符进行打印。因此,应采用程序查询方式控制打印程序。打印程序段如下:

```
        mov     bx,offset buffer        ;置缓冲区偏移量
        mov     cx,chrtl                ;置输出字符计数器
bg:
        mov     al,[bx]                 ;取字符
        mov     dx,378h
        out     dx,al                   ;输出字符到打印控制器的数据端口
        mov     dx,379h
wt:
        in      al,dx                   ;读打印机状态
        test    al,80h                  ;忙否? al 最高位为1,表示忙
        jz      wt                      ;忙,重复查询
        mov     dx,37ah                 ;不忙,则形成选通信号
        mov     al,0dh
        out     dx,al                   ;令打印机打印字符
        mov     al,0ch
        out     dx,al
        inc     bx                      ;指向下一输出字符
        loop    bg
```

6.3　中 断 系 统

在计算机系统中,引入中断的最初目的是为了提高系统的输入输出性能。随着计算机

应用的发展,中断技术也应用到计算机系统的许多领域,如异常事件处理、多道程序、分时系统、实时处理、程序监视和跟踪等领域。因此,了解中断系统的工作原理和掌握中断程序的设计方法是非常必要的。

6.3.1　中断和中断源

所谓中断就是当某种紧急事件发生时,CPU 暂停当前程序的执行,转而执行处理紧急事务的程序(中断服务程序),并在该事务处理完后能自动恢复执行原先被中断的程序继续执行的过程。中断系统还根据紧急事务的紧急程度,把中断分为不同的优先级,并规定高优先级事务能中断较低优先级的中断服务程序的执行。

中断源是指引起紧急事务的事件。80x86 的中断源可分为外部中断和内部中断两大类。

外部中断:CPU 外部事件引起,通常是各种外设的输入输出请求,如键盘输入引起的中断、通信端口接受信息引起的中断等。外部中断源通过 CPU 的不可屏蔽中断请求线 NMI 或可屏蔽中断请求线 INTR 通知 CPU。

内部中断:由 CPU 内部原因引起,包括除数为 0 中断、溢出中断、单步中断等,也包括由程序设定的软中断指令 INT 引发的中断。

CPU 在执行程序时,是否响应中断要取决于以下 3 个条件能否同时满足。

① 有中断请求。

② 允许 CPU 接受中断请求。

③ 当前指令执行完毕。

条件①是响应中断的主体。除用指令 INT 所引起的软件中断之外,其他中断请求事件是随机产生的,程序员是无法预见的。

程序员可用程序部分地控制条件②是否满足,即可用指令 STI 和 CLI 来允许或不允许 CPU 响应可屏蔽的外部中断。而对于不可屏蔽中断和内部中断,程序员是无控制权的,CPU 一定会予以响应。

对于外部中断和内部中断,如果同时有多个中断请求信号,那么 CPU 如何响应它们呢?80x86 系统规定的各中断源的优先级顺序如下:

除法错→INTO →INT n→NMI→INTR→单步中断

当多个中断源发出中断请求信号时,CPU 根据其优先级次序,首先响应最高级的中断请求,完成该中断处理后,再响应次高级的中断请求。

6.3.2　中断向量表

对于不同的中断源,所要求的中断处理是不同的,换句话说,每一个中断源都对应着一个确定的中断处理程序。那么,当 CPU 响应某一中断源发出中断请求时,如何根据中断源找到要调用的中断处理程序的入口地址呢? 为此,PC 系统采用向量中断的方法,在向量中断中,中断系统正是根据中断向量表提供的信息,确定不同的中断服务程序的入口地址。

中断向量表是一个特殊的线性表,它保存着系统所有中断服务程序的入口地址(偏移量和段地址)。在微机系统中,该向量表有 256 个元素(0 ~0FFH),每个元素占 4 个字节,前两个字节用来存放入口地址的偏移地址,后两个字节用来存放入口地址的段地址,总共

1 KB。中断向量表在内存中的存储形式及其存储内容如图 6 - 3 所示。

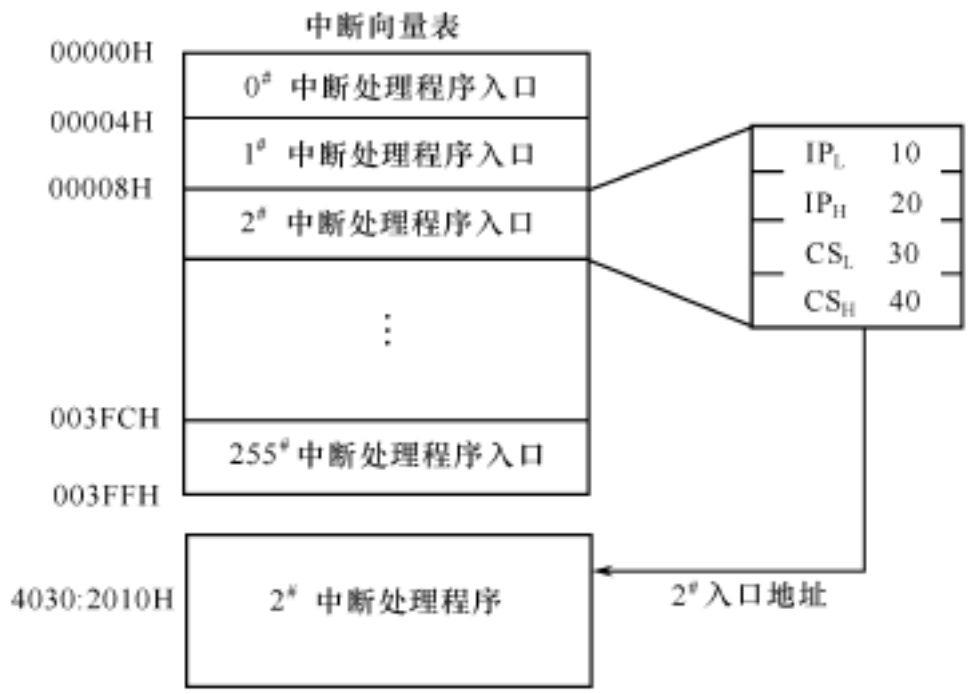


图 6 - 3　中断向量表结构

图 6 - 3 中的“ 中断偏移量 ”和“ 中断段地址 ”是指该中断服务程序入口单元的“ 偏移量 ”和“ 段地址 ”。由此不难看出, 假如中断号为 n, 那么, 在中断向量表中存储该中断处理程序的入口地址的单元地址即为 4× n。

表 6 - 2 中列出前 16 个中断向量表中的部分常用中断号。

表 6 - 2　部分常用的中断号及其含义

中断号	含　义	中　断　号	含　义
0	除法出错	8	定时器
1	单步	9	键盘
2	非屏蔽中断	A	未用
3	断点	B	COM2
4	溢出	C	COM1
5	打印屏幕	D	硬盘(并行口)
6	未用	E	软盘
7	未用	F	打印机

6.3.3　中断服务程序

中断服务程序亦称中断处理程序。处理机响应中断后, 便根据中断系统提供中断处理程序的入口地址进入中断服务程序。处理机在进入中断服务程序之前自动完成如下工作。

将标志寄存器压栈, 清除标志位 IF 和 TF。

将 CS 的内容压栈, 并把中断服务程序入口地址的高字部分送 CS。

将 IP 的内容压栈, 并把中断服务程序入口地址的低字部分送 IP。

从 8086 CPU 获得中断向量号开始到进入中断服务程序的过程, 不同类型的中断是没有差别的。各种中断服务程序的结构和编程原则都大致相同。中断服务程序的一般结构

如下。

保护现场→中断处理→恢复现场→开中断→中断返回

中断和子程序调用之间有其相似和不同之处。它们的工作过程非常相似,即暂停当前程序的执行,转而执行另一程序段,当该程序段执行完时,CPU 自动恢复原程序的执行。它们的主要差异如下。

子程序调用一定是程序员在编写源程序时事先安排好的,是可知的,而中断是由中断源产生的,是不可预见的(用指令 INT 引起的中断除外)。

子程序调用是用 CALL 指令来实现的,但没有调用中断的指令,只有发出中断请求的事件(指令 INT 是发出内部中断信号,而不要理解为调用中断服务程序)。

子程序的返回指令是 RET,而中断服务程序的返回指令是 IRET/IRETD。

在通常情况下,子程序是由应用系统的开发者编写的,而中断服务程序是由系统软件设计者编写的。

6.3.4　设置中断向量

无论是外部中断还是内部中断,当发生中断时,为了实现自动转入相应的中断服务程序进行处理,必须在中断向量表中预先装入该中断的中断服务程序的入口地址。中断向量的设置通常有两种方法,一种是通过系统功能调用方法设置中断向量,另一种是直接访问存储器去设置中断向量。

1. 利用系统功能调用(INT 21H)设置中断向量

中断 21H 的功能 25H 可为指定的中断号设置新的入口地址。其使用方法如下。

入口参数:　H = 中断号

　　DS: DX = 中断处理程序的入口地址

出口参数: 无

下面的例子给出了用功能调用的方法把子程序 newfunc 设置为中断 n 的中断处理程序的方法。

```
…
new unc proc
    …
    iret
newfunc endp
…
mov ax,seg newfunc
mov ds,ax                    ;设置段地址寄存器
mov dx,offset newfunc        ;设置偏移量
mov al,n                     ;这里的 n 要用具体的中断号来定
mov ah,25h
int 21h
…
```

2. 直接访问存储器设置中断向量

程序员可以通过直接访问存储器的方法来修改中断向量表,从而为中断处理程序设置中断向量。用直接访问存储单元的方法把子程序 newfunc 设置为中断 n 的中断处理程序。

下面的例子给出了用直接访问存储单元的方法把子程序 newfunc 设置为中断 n 的中断处理程序的方法。

```
…
new unc proc
    …
    iret
newfunc endp
…
mov ax,0h
mov ds,ax
mov bx,4* n
cli
mov word ptr [bx],offset newfunc      ;设置中断处理程序的偏移量
mov word ptr [bx+2],seg newfunc       ;设置中断处理程序的段地址
sti
…
```

在上面的程序段中,指令 CLI 是关中断指令,它用来确保随后两条 MOV 指令被连续执行而不被打断。这是因为执行其第一条 MOV 指令后,原中断向量表中的入口地址就被破坏了,这时,该入口地址既不指向原处理程序,也不指向新处理程序。如果此刻正巧发生了该类型的中断请求,那么,系统将转向一个非法的位置。后面的指令 STI 是开中断指令,它允许 CPU 响应其后的中断请求。

6.3.5　中断功能分类

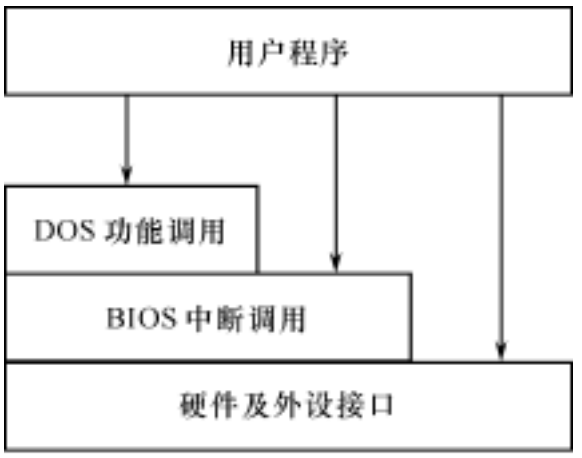


图 6 - 4　各类中断间的层次关系

中断可分为硬件中断和软件中断两大类。对于硬件中断,中断是随机产生的,程序员不能控制它,如硬件和外设的中断等。而对于软件中断,程序员可通过中断指令 INT 来有目的的安排中断的发生。常用的这类中断有 DOS 功能调用(INT 21H)、BIOS 中断等。各类系统中断之间的层次关系如图6 - 4所示。

在用户程序中,若直接通过端口来操作硬件或外设,那么,其处理过程没有额外的多余工作,处理速度显然是最快的,但这样做,无疑使用户程序具有了很大的局限性。硬件环境的改变将直接影响程序的正常运行。

若用户程序通过调用 DOS 功能来实现其所需功能,那么,应用程序与低层硬件相距较远,操作最终的对象需要经过中间环节,处理速度肯定受到一定的损失,但这种应用程序适应性强,应用范围广,对硬件的依赖性最小。

由于 BIOS 介于 DOS 和具体硬件之间,所以,调用 BIOS 中断功能是一个很好的折中方案。

综上所述,可以归纳出如下结论:使用中断的层次越高,它与硬件设备相关的程度就越低,处理速度也就越低,但用户程序的适用范围较广,反之亦然。

6.4　系统功能调用与 BIOS 中断

6.4.1　调用方式

汇编语言特别适合于编制直接管理计算机硬件设备的底层软件。要编制这类软件，就必须十分了解被管理的硬件设备。DOS 系统和 ROM BIOS 系统为用户提供了一组例行子程序，用于完成基本 I/O 设备(如 CRT 显示器、键盘、打印机、软盘、磁盘、磁带等) 的输入输出控制，需要特别指出的是，对这些例行子程序的调用都是通过软中断指令来实现的。

每个功能子程序都对应于一个子程序文件，其说明部分包括名称、功能、入口参数、出口参数和实例等内容。对所有功能子程序的调用，调用时一般均需经过以下 3 个步骤。

① 将入口参数送往指定的寄存器。

② 将功能调用号送往 AH 寄存器。

③ 执行软中断指令 INT n 调用相应的功能子程序。

其中软中断指令 INT 中的 n 值因子程序不同而可能不同。有的软中断命令只对应于一个子程序，这时的功能调用无需上述步骤②；有的软中断命令则对应很多子程序，例如 INT 21H 对应有 100 多个子程序，INT 2FH 对应有近 50 个子程序等，这时的功能调用就必须经过上述 3 个步骤。

6.4.2　系统功能调用

INT 21H 软中断命令(简称 DOS 功能调用) 实现最常用的输入/输出功能子程序调用。下面给出几种常用的系统功能调用方法。

(1) 输入并显示字符(01H 号功能调用)

功能描述：从标准输入设备(如键盘) 输入一个字符，并将其 ASCII 码值送入 AL 寄存器，同时将该字符显示在屏幕上。

入口参数：无。

调用方式：　OV AH,01H

　　INT 21H

出口参数：AL 中为输入字符的 ASCII 码。

如下程序段实现从键盘输入字符，判断输入的字符是否为“Y”或“y”，是则退出，否则，重新等待键盘输入。

```
again:
        mov ah,01h
        int 21h
        cmp al, Y          ;判断是否是大写字母“Y”
        je exit            ;判断是否是则退出
        cmp al, y          ;判断是否是小写字母“y”
        je exit            ;是则退出
        jmp again          ;不是 y 和 Y 继续等待输入
exit:   mov ah,4ch
```

```
        int 21h                ;退出本程序, 返回 dos
```

(2) 单字符输出(02H 号功能调用)

功能描述: 将 DL 寄存器中的字符输出到标准输出设备。

入口参数: DL 寄存器中存放要输出字符的 ASCII 码值。

调用方式:　OV AH,02H

　　INT 21H

出口参数: 无。

下面一段代码实现在屏幕上输出一个字符 C。

```
mov ah,02h
mov dl , C          ; 输出字符 C
int 21h
```

(3) 字符串输出(09H 号功能调用)

功能描述: 将一个以 $ 字符结尾的字符串输出到显示器。

入口参数: DS: DX 指向内存中一个以 $ 符结尾的字符串。

调用方式:　OV AH,09H

　　INT 21H

出口参数: 无。

下面一段代码实现在屏幕上输出一个字符串“ This is a test ! ”

```
.data
    str     db "this is a test !",0dh,0ah, $
.code
    mov     ax,@data
    mov     ds,ax
    mov     dx,offset str
    mov     ah,09h
    int     21h
    ……
```

(4) 字符串输入子程序(0AH 号功能调用)

功能描述: 从键盘读入字符串并将字符串存入缓冲区, 用回车键结束字符串。若字符串各数超过规定的长度, 则响铃并忽略超出长度的字符。

入口参数: DS: DX = 存放输入字符缓冲区的起始地址。

其中第一个字节存放缓冲区长度(等于实际字符加 1) ; 第二个字节用于存放实际输入的字符数; 从第三个字节开始存放从键盘上接受的字符。若实际输入的字符少于定义的字符, 则缓冲区内其余字节填 0。

调用方式:　OV AH,0AH

　　INT 21H

出口参数: 实际输入的字符数(不包括回车键) 保存在缓冲区的第二个字节; 输入的字符串保存在缓冲区第三个字节开始的地方, 最后一个字符总是回车键“ Enter ”。

调用实例:

```
.da a
   buff   db  40                ; 缓冲区大小(实际字符长度为39)
          db  ?                 ; 留作系统填入实际输入的字符个数
          db  40 dup(?)         ; 定义40 个字节的存储空间
   .co e
      .....
      mov  dx,offset buff
      mov  ah,0ah
      int  21h
```

(5) **获取系统日期**(2AH **号功能调用**)

入口参数: **无**。

调用方式:　OV AH,0AH

INT 21H

出口参数:　X = **年**(1980 ~2099), DH = **月**(1 ~12), DL = **日**(1 ~31),

AL = **星期几**(0 = Sunday, 1 = Monday, . . .)。

(6) **设置系统日期**(2BH **号功能调用**)

入口参数: CX = **年**(1980 ~2099), DH = **月**(1 ~12), DL = **日**(1 ~31)。

调用方式:　OV AH,0AH

INT 21H

出口参数: AL = 00H(**设置成功**), 0FFH(**设置失败**)。

(7) **获取系统时间**(2CH **号功能调用**)

入口参数: **无**。

调用方式:　OV AH,0AH

INT 21H

出口参数: CH = **时**(0 ~23), CL = **分**(0 ~59), DL = **秒**(0 ~59), AL = **百分秒**(0 ~99)。

(8) **设置系统时间**(2DH **号功能调用**)

入口参数: CH = **时**(0 ~23), CL = **分**(0 ~59), DL = **秒**(0 ~59), AL = **百分秒**(0 ~99)。

调用方式:　OV AH,0AH

INT 21H

出口参数: AL = 00H(**设置成功**), 0FFH(**设置失败**)。

6.4.3　BIOS **中断调用**

6.4.3.1　**键盘中断**(16H **中断**)

键盘是绝大多数程序的主要输入方式,学习和掌握有关键盘输入中断的使用方法对编写交互式程序是非常重要的,也能更进一步理解计算机是如何接受键盘输入的。

1. **键盘状态字**

计算机键盘上除了可输入各种字符(字母、数字和符号等)的按键之外,还有一些功能键(如 F1, F2 **等)、控制键(如** Ctrl, Alt, Shift **等)、双态键(如** Num Lock, Caps Lock **等)和特殊请求键(如** Print Screen, Scroll Lock **等)。**

键盘中的控制键和双态键是非打印按键,它们是起控制或转换作用的。当使用者按下

控制键或双态键时,系统要记住其所按下的按键。为此,在计算机系统中,特意安排一个字来标志这些按键的状态,我们称该字为键盘状态字。键盘状态字的各位含义如图 6 - 5 所示。

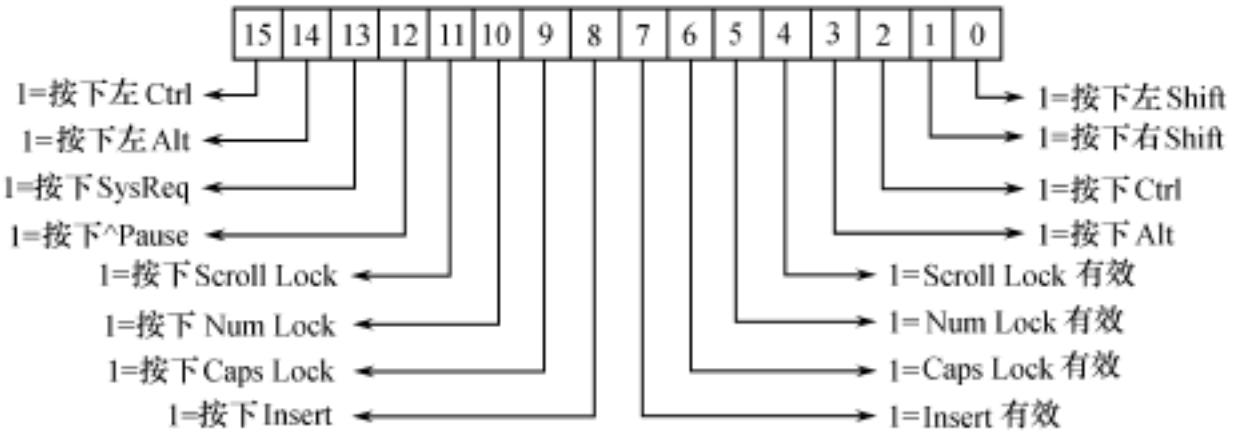


图 6 - 5　键盘状态字的各位含义

2. 键盘中断处理过程

当用户按键时,键盘接口会得到一个代表该按键的键盘扫描码,同时产生一个中断请求。键盘中断服务程序先从键盘接口取得按键的扫描码,然后根据其扫描码判断用户所按的键并作相应的处理,最后通知中断控制器本次中断结束并实现中断返回。

若用户按下双态键(如 Caps Lock, Num Lock 和 Scroll Lock 等)或按下控制键(如 Ctrl, Alt 和 Shift 等),则在键盘标志字中设置其标志位;若用户按下功能键(如 F1, F2 等),再根据当前是否又按下控制键来确定其系统扫描码,并把其系统扫描码和一个值为 0 的字节存入键盘缓冲区;若用户按下字符键(如 A、1、+ 等),此时,再根据当前是否又按下控制键来确定其系统扫描码,并得到该按键所对应的 ASCII 码,然后把其系统扫描码和 ASCII 码一起存入键盘缓冲区;若用户按下功能请求键(如 Print Screen 等),则系统直接产生一个具体的动作。

3. BIOS 中的键盘输入

在 BIOS 系统中,提供了中断 16H 来实现键盘输入功能。其具体的功能如下。

(1) 输入字符不回显(功能 00H 和 10H)

入口参数: AH = 00H 读普通键盘, AH = 10H 读扩展键盘。

出口参数:　H = 键盘的扫描码, AL = 字符的 ASCII 码。

(2) 读取键盘状态(功能 01H 和 11H)

入口参数: AH = 01H 检查普通键盘, AH = 11H 检查扩展键盘。

出口参数: 若 ZF = 1 无字符输入; 若 ZF = 0 则 AH = 键盘的扫描码, AL = 按键的 ASCII 码。

(3) 读取当前键盘状态字(功能 02H 和 12H)

入口参数: AH = 02H (普通键盘), AH = 12H (扩展键盘)。

出口参数:　L = 键盘标志(02H 和 12H 都有效), AH = 扩展键盘的标志(12H 有效)。

【例 6.5】　用键盘最多输入 10 个字符并存入内存变量 Buff 中,若按 Enter 键,则表示输入结束。

```
.model small
    cr      equ 0dh ;定义“回车”键
```

```
.data
    buff  db 10 dup(?)
.code
.startup
mov    cx,10
lea    bx,buff
.re eat
   mov  ah,0h
   int  16h                ;调用 bios 的键盘输入中断功能
   .break .if al ==cr      ;按下“ 回车 ”键则退出循环
   mov [bx],al
   inc  bx
.untilcxz                  ;缓冲区未满,则继续输入
.exit 0
end
```

6.4.3.2　鼠标中断(33H 中断)

鼠标是现在计算机系统中的不可缺的输入设备,鼠标的出现为使用计算机带来了极大的方便,利用鼠标的移动、点击和拖放等功能可以完成绝大多数的操作。同时,鼠标指针的各种形状还反映了系统当前的工作状态。

鼠标指针主要有两种表现形式:文本鼠标和图形鼠标。文本鼠标又分为软指针和硬指针。软指针是用各种字符来作鼠标指针,而硬指针是用方块光标的大小来表示鼠标指针。图形鼠标可用各种不同的指针形状来反映系统当前所处的工作状态和所能进行的操作。在 Windows 操作系统及其应用程序中使用了很多指针形状的变化来表达各种有用信息。

鼠标的文本软指针与图形指针的形成过程一致,它需要两部分信息:像素掩码和光标掩码。其指针形成过程如下。

① 像素掩码与当前鼠标所处位置的像素信息进行“ 逻辑与 ”运算。

② 光标掩码与步骤①的运算结果再进行“ 异或 ”操作,该操作所得到的 16× 16 位的0/1信息就构成了当前鼠标指针的形状。

假设像素掩码为全 0。全 0 的像素掩码与屏幕上的显示信息“ 逻辑与 ”后,所得结果仍为全 0,全 0 的运算结果再和“ 光标掩码 ”进行“ 异或 ”操作,这时,所得结果显然与“ 光标掩码 ”完全一样,所以,看到的鼠标指针形状就是光标掩码所表达的指针形状。

综合上述,可得结论:若像素掩码为全 0,那么,鼠标的形状就是 16× 16 位光标掩码所表示的指针形状,鼠标所到之处就看不到该区域内(16× 16 点阵范围)的其他显示信息。

在 Windows 操作系统及其应用程序中,在 16× 16 点阵范围内,除了看不见被各种形状指针覆盖的部分之外,还能看见其他区域,这是因为鼠标的“ 像素掩码 ”取其“ 光标掩码 ”的反相点阵所致。

BIOS 系统提供了用“ 中断 33H ”来实现鼠标中断的功能,下面介绍几种常用的鼠标中断功能。

(1) 设置文本指针(功能 0AH,分为软指针和硬指针设置)

软指针设置入口参数:

BX = 0 , CX = 像素掩码, DX = 光标掩码(位 7 ~0:鼠标指针符号的 ASCII 码,位 10 ～

8: 前景色, 位 11: 亮度, 位 14 ~12: 背景色, 位 15: 闪烁) 。

出口参数: 无

软指针设置示例如下:

```
mov     bx,0        ;软指针
mov     dl , s      ;用字符‘ s ”作为鼠标指针符号
mov     dh,07fh     ;置鼠标的颜色
mov     cx,0        ;置像素位掩码
mov     ax,0ah
int     33h         ;设置文本鼠标指针
```

硬指针设置入口参数:

BX = 1, CX = 光标的起始扫描线, DX = 光标的结束扫描线。

出口参数: 无。

硬指针设置示例如下:

```
mov     bx,1        ;硬指针
mov     cx,01h      ;鼠标硬指针的起始扫描线
mov     dx,0fh      ;鼠标硬指针的结束扫描线
mov     ax,0ah      ;设置文本鼠标指针
int     33h
```

【例 6. 6】　编写可随时修改文本鼠标指针符号的程序, 要求在程序运行过程中, 随时在键盘上按什么字符, 即以该字符为鼠标指针符号。

程序如下:

```
.model small
.code
.sta tup
    mov  ax,00
    int  33h                    ;初始化鼠标
    .if   ax! = 00
        mov  ax,01h
        int  33h
        mov  bx,0
        mov  dl, a              ;用字符‘ a ”作为鼠标指针符号
        mov  dh,07fh            ;设置鼠标的颜色
        mov  cx,0h
        mov  ax,0ah
        int  33h                ;设置文本鼠标指针
.repeat
        mov  ah,01h
        int  16h
        jz  next                ;无键盘按键, 则转到后面
        mov  ah,00h
        int  16h                ;读键盘按键
        mov  bx,0
        mov  dl,al              ;设置当前按键为鼠标指针符号
```

```
        mov  dh,7fh
        mov  cx,0
        mov  ax,0ah
        int  33h
    nex :
        mov  ax,03h
        int  33h
    .unt l  bx ==1
        mov  ax,02h
        int  33h
    .endif
.exit 0
end
```

(2) 设置图形指针(功能 09H)

入口参数: BX = 指针的水平位置, CX = 指针的垂直位置, ES: DX = 16× 16 位光标的映像地址。

出口参数: 无。

参数说明: (BX, CX) 是鼠标的指针在 16× 16 点阵中的位置, (0, 0) 是左上角; ES: DX 指向的存储单元内存放 16× 16 点阵的位映像隐码, 紧跟其后的是 16× 16 点阵的光标掩码。

鼠标指针的显示方法是先将位映像隐码与屏幕显示区的内容进行“逻辑与”运算, 然后再用光标掩码内容“异或”前面运算的结果。

下面的代码给出了设置图形指针的中断功能使用方式。

```
…
pmask    dw    16 dup(?)        ;像素位掩码
cmask    dw    16 dup(?)        ;光标掩码
…
mov      ax,ds
mov      es,ax
lea      dx,pmask              ;es:dx = 像素位掩码的起始地址
mov      bx,0
mov      cx,0                  ;在鼠标指针范围内,(0,0)点为指示点
mov      ax,09h
int      33h                   ;设置图形鼠标指针
```

(3) 获取鼠标位置及其按键状态(功能 03H)

入口参数: 无。

出口参数: CX = 水平位置, DX = 垂直位置, BX = 按键状态(位 0、1、2 分别表示按下左、右和中键, 其他位保留)。

(4) 设置鼠标指针位置(功能 04H)

入口参数: CX = 水平位置, DX = 垂直位置。

出口参数: 无。

(5) 获取鼠标按钮释放信息(功能 06H)

入口参数: BX = 指定的按键(0、1、2 分别表示左键、右键和中键)。

出口参数: AX = 按键状态(位 0、1、2 分别表示按下左、右和中键, 其他位保留) , BX = 释放的次数, CX = 最后释放时的水平位置。

【例 6.7】　在屏幕的右上角动态显示文本鼠标的位置, 即鼠标的任何移动都将马上显示其所处位置, 按鼠标左键结束程序的运行。

程序如下:

```
.model small,c
.data
    oldaddr  dd ?
    msg      label byte
    xmsg     db  x =                    ;显示方式: x =* *  y =* *
    xasc     dw?
             db
    ymsg     db  y =
    yasc     dw?
    count    equ $ - msg
.stack 128
.code
  clsscr  proc                          ;清除整个屏幕
          mov  ax,0600h
          mov  bh,30h
          mov  cx,0
          mov  dx,184fh
          int  10h
          ret
clsscr    endp
;把两位十进制数 data 转换成字符串放入以首地址 desc 开始的缓冲区之中
btoasc proc uses ax cx di data:word,desc:ptr byte
          mov  ax,data
          mov  di,desc
          mov  word ptr [di],2020h    ;赋两个空格
          mov  cl,10
          div  cl
          or   al,30h
          mov  [di],al
          inc  di
          or   ah,30h
          mov  [di],ah
          ret
btoasc    endp
;把当前鼠标位置( cx,dx) 显示在屏幕右上角, 该子程序被设置为鼠标移动事
;件的中断处理程序, 所以, 只要鼠标一移动, 该子程序马上就被执行
disppos proc far
          shr  cx,3
          shr  dx,3
invoke btoasc,cx,addr xasc
invoke btoasc,dx,addr yasc
          mov  ah,02h
```

```
            mov  bh,0
            mov  dx,046h
            int  10h                          ;设置字符串的显示位置
            mov  ah,40h
            mov  bx,01h
            mov  cx,count
            lea  dx,msg
            int  21h                          ;显示鼠标位置的字符串
ret
disppos endp
.startup
            call clsscr
            mov  ax,00
            int  33h
            .if  ax != 00
                 mov    ax,01h
                 int    33h                   ;显示鼠标指针
                 mov    ax,cs
                 mov    es,ax
                 lea    dx,disppos
                 mov    ax,14h
                 mov    cx,1                  ;把子程序 disppos 设置成鼠标
                 int    33h                   ;移动事件的中断服务程序
                 mov    word ptr oldaddr,dx
                 mov    word ptr oldaddr +2,es   ;保存原移动事件处理程序入口
                 .repeat
                   mov    ax,03h
                   int    33h
                 .until bx ==1                 ;按下鼠标左键
                 mov    ax,02h
                 int    33h                   ;隐藏鼠标指针
            .endif
            mov    dx,word ptr oldaddr
            mov    es,word ptr oldaddr +2
            mov    cx,1
            mov    ax,0ch
            int    33h                 ;恢复原鼠标移动的中断服务程序
            call   clsscr
.exit 0
end
```

6.4.3.3　屏幕中断(10H 中断)

显示器是计算机系统中一种重要的输出设备,有两种显示方式:文本显示方式和图形显示方式。常用的显示分辨率为 800× 600 像素或 1 024× 768 像素等。

BIOS 系统提供了中断 10H 来实现各种屏幕处理功能。不同的操作功能通过功能号来指定(放在 AH 寄存器中),其具体功能如表 6 - 3 所示。有关中断功能的详细描述和调用参数在此从略,需要查阅者可参阅附录 2。

表 6 - 3　屏幕中断功能

功能号	功能描述	功能号	功能描述
00H	设置显示器模式	0AH	在当前光标处显示字符
01H	设置光标形状	0BH	设置调色板、背景色或边框
02H	设置光标位置	0CH	写图形像素
03H	读取光标信息	0DH	读图形像素
04H	读取光笔位置	0EH	在 Teletype 模式下显示字符
05H	设置显示页	0FH	读取显示器模式
06H	清屏或屏幕上滚	10H	设置颜色
07H	屏幕下滚	11H	设置字体
08H	读光标处的字符及其属性	12H	配置显示器
09H	在光标处按指定属性显示字符	13H	在 Teletype 模式下显示字符串

1. 文本显示模式

在常用的文本显示模式(模式 3)下,屏幕被划分成 25 行,每行可显示 80 个字符,所以,每屏最多可显示 2 000(80× 25)个字符。为了便于标识屏幕上的每个显示位置,我们就用其所在行和列来表示之,并规定:屏幕的左上角坐标为(0,0),右下角坐标为(24,79)。在显示字符时,用一个字节存储该字符的 ASCII 码,用另一个字节存储该字符的显示属性,即显示颜色。彩色显示器的字符显示属性的定义如图 6 - 6 所示。

以下代码段可将显示模式设为 80× 25 彩色文本方式:

```
mov ah,0
mov al,3        ; 设置 80×25 彩色文本方式
int 10h
```

下面给出几个在文本显示方式常用的功能调用及参数说明。

(1) 设置光标位置(功能 02H)

入口参数:BH = 显示页码,DH = 行(Y 坐标),DL = 列(X 坐标)。

出口参数:无。

(2) 清屏或滚屏 (功能 06H、07H)

入口参数: AH = 06H 向上滚屏,AH07H 向下滚屏,AL = 滚动行数(0 清屏),BH = 空白区域的默认属性,(CH、CL) = 窗口的左上角位置(Y 坐标,X 坐标),(DH、DL) = 窗口的右下角位置(Y 坐标,X 坐标)。

出口参数: 无。

(3) 在当前光标处按指定属性显示字符(功能 09H)

入口参数:AL = 字符,BH = 显示页码,BL = 属性(文本模式)或颜色(图形模式),CX = 重复输出字符的次数。

出口参数:无。

(4) 在当前光标处按原有属性显示字符(功能 0AH)

入口参数:AL = 字符,BH = 显示页码,CX = 重复输出字符的次数。

出口参数:无。

闪烁	背景属性			亮度	前景属性		
7	6	5	4	3	2	1	0
Blink	Red	Green	Blue	Bright	Red	Green	Blue

7654	背景属性
0000	Black
0001	Blue
0010	Green
0011	Cyan
0100	Red
0101	Magenta
0110	Brown
0111	White
1000	Black Blink
1001	Blue Blink
1010	Green Blink
1011	Cyan Blink
1100	Red Blink
1101	Magenta Blink
1110	Brown Blink
1111	White Blink

3210	前景属性
0000	Black
0001	Blue
0010	Green
0011	Cyan
0100	Red
0101	Magenta
0110	Brown
0111	White
1000	Dark Gray
1001	Light Blue
1010	Light Green
1011	Light Cyan
1100	Light Red
1101	Light Magenta
1110	Yellow
1111	Bright White

图 6 - 6　字符显示属性的定义

(5) **在 Teletype 模式下显示字符串(功能** 13H)

入口参数: BH = **页码**, BL = **属性**, CX = **显示字符串长度**, (DH、DL) = **坐标(行、列)**, ES:BP = **显示字符串的地址**, AL = 0 **显示后光标位置不变**, AL = 1 **显示后光标位置改变**。

出口参数: **无**。

【例 6. 8**】　在屏幕第** 10 **行、第** 20 **列处显示** 5 **朵梅花, 要求颜色各异且中间一朵能够闪烁**。

源程序如下:

```
.model small
.data
    atri db 6eh,52h,94h,52h,6eh
.code
.startup
    mov     ah,0
```

```
    mov     al,3
    int     10h             ;设置 80× 25 彩色文本方式
    len     si,atri         ;属性字节值表首址存 si
    mov     di,5            ;显示 5 个字符
    mov     dx,0a13h        ;显示位置(10 行,20 列)
    mov     ah,15           ;取当前页号
    int     10h
lp:
    mov     ah,2            ;置光标位置
    inc     dl
    int     10h
    mov     al,5            ;显示梅花形字符
    mov     bl,[si]
    mov     cx,1
    mov     ah,9
    int     10h
    inc     si              ;指向下一属性字节
    dec     di              ;判断显示完否
    jnz     lp              ;未完转 lp 再显示
.exit 0
end
```

【例 6.9】　用“霓虹灯”的显示方式显示字符串“Hello”,按 Esc 键时结束程序的运行。本题可用显示颜色的变化来模拟霓虹灯的显示方式,即用颜色 15(亮白)作为字符的主要显示颜色,再用颜色 12(亮红)从左到右逐个扫描。

```
.model small,c
.data
      kbesc  equ  1bh
      buff   db    H,15, e ,15, l ,15, l ,15, o ,15
.code
clear  roc near uses ax bx cx dx     ;清屏并保护所用寄存器
      mov    cl,0
      mov    ch,0
      mov    dl,79
      mov    dh,24                   ;(0,0) - (24,79)是屏幕的左上角和右下角
      Wmov   bh,7
      mov    al,0
      mov    ah,6
      int    10h
      ret
clear endp
.startup
      call   clear
      mov    ax,ds
      mov    es,ax
      mov    si,9
again:
      mov    buff[si],15          ;把前一次的红色还原
```

```
        add     si,2
        .if     si > 9
                mov  si,1
        .endif
        mov     buff[si],12          ;把当前字符以红色显示
        mov     bh,0
        mov     cx,5
        mov     dh,5
        mov     dl,20                ;显示位置从(5,20)开始
        lea     bp,buff
        mov     al,2
        mov     ah,13h
        int     10h                  ;调用中断10h的功能13h
        mov     ah,1
        int     16h                  ;检查是否有按键
        jz      again                ;若无字符可读,则继续循环
        mov     ah,0
        int     16h
        cmp     al,kbesc
        jnz     again                ;若按键不是Esc,则继续循环
.exit 0
end
```

【例6.10】　编写一个输入口令的程序,该程序的具体要求如下。

① 每输入一个字符,屏幕显示字符“#”。

② 口令最多只有10个字符,多余的按键被丢弃。

③ 若输入的口令为“hello”,则以蓝色显示“welcome...”,否则,以闪烁、亮红色显示“invalid password”。

程序如下:

```
.model small
.data
    cr    equ  0dh
    msg1  db   "welcome..."
    msg2  db   "invalid password"
    psw1  db   "hello"
    buff  db   10 dup(?)
.code
.startup
    mov   ax,ds
    mov   es,ax
    xor   bx,bx
again:
    mov   ah,0h
    int   16h                        ;从键盘接受字符输入
    cmp   al,cr
    jz    next                       ;若按回车键,则结束密码输入
    cmp   bx,10
```

```
    jz      again                   ;若已接受了10 个字符,则丢弃随后的字符
    .if     al >= a && al <= z
            sub  al,20h
    .endif
    mov     buff[bx],al
    inc     bx                      ;保存当前输入并移动有关指针
    mov     dl, #
    mov     ah,2
    int     21h                     ;在屏幕上显示字符“ #”
    jmp     again
next:
    call    clear                   ;清屏,具体过程略去
    cmp     bx,5
    jnz     error                   ;若输入的字符串长度不为5,则密码错误
    lea     si,psw1
    lea     di,buff
    mov     cx,bx
    cld
    repe    cmpsb                   ;比较字符串 psw1 和 buff
    .if     zero?                   ;若字符串的相应字符相同
    mov     bp,offset msg1          ;显示字符串首地址
    mov     bl,09h                  ;显示字符颜色:蓝色
    mov     cx,0ah                  ;显示字符串长度:10
    .else
error:
    mov     bp,offset msg2          ;显示字符串首地址
    mov     bl,8ch                  ;显示字符颜色:闪烁、亮红色
    mov     cx,10h                  ;显示字符串长度:16
    .endif
    mov     al,0
    mov     bh,0
    mov     dh,5
    mov     dl,20                   ;在位置(5,20)的位置开始显示字符串
    mov     ah,13h
    int     10h                     ;使用中断 10h 的功能13h 来显示字符串
.exit 0
end
```

2. 图形显示模式

图形显示是目前最常用的一种显示方式,也是 Windows 操作系统的默认显示方式。在该显示方式下,我们可以看到优美的图像,浏览丰富多彩的网页等。图形显示的最小单位是像素,对每个像素可用不同的颜色来显示。所以,在显示缓冲区内记录的信息是屏幕中各像素的显示颜色。

由于各种图形显示模式所能显示的颜色和像素是不同的,因此它决定了显示缓冲区的存储方式也是不同的。各种显示模式在显示缓冲区存储方式上有明显差异,操作像素方法的难易程度相差也很大,所以,程序员不要用直接操作显示缓冲区的办法来达到改变显示像素的目的,最好是通过 BIOS 内的中断功能来实现相应的功能,这样,所编写的程序能很方

便地适应不同的图形显示模式。

【例 6.11】　在 256 色 320× 200 的图形显示模式下，从屏幕最左边向最右边，依次画竖线(从顶到底)，线的颜色从 1 依次加 1，要求用中断调用的方法来画线。

```
.model small
.data
  mode db ?                          ;保存当前显示模式
.code
  vline proc near uses ax bx dx
    mov   dx,0
    mov   bh,0
    mov   ah,0ch
draw:
    int   10h
    inc   dx
    cmp   dx,200
    jl    draw
    ret
vline endp
.startup
    mov   ah,0fh
    int   10h
    mov   mode,al                    ;保存当前显示模式,在程序结束前恢复
    mov   ah,0
    mov   al,13h
    int   10h                        ;设置 256 色 320× 200 的图形显示模式
    mov   cx,0
    mov   al,01h                     ;cx = 线所在列, al = 线的颜色
    call  vline
    inc   al
    inc   cx
    cmp   cx,320
    jl    draw                       ;从左到右画 320 条竖线
    mov   ah,0
    int   16h                        ;等待一个按键
    mov   al,mode
    mov   ah,0
    int   10h                        ;恢复原来的屏幕显示模式
.exit 0
end
```

6.4.3.4　打印机中断(17H 中断)

BIOS 17H 中断指令提供了由 AH 寄存器指定的 3 种不同的操作，即功能 0、功能 1 和功能 2，下面分别进行介绍。

(1) 向打印机输出字符(功能 00H)

入口参数：AL = 待打印的字符，DX = 打印机号(0—LPT1，1—LPT2，2—LPT3)。

出口参数：　H = 打印机状态。其各位为 1 时的含义如下：

位 7—空闲　　位 6—响应　　位 5—无纸　　位 4—被选(联机)
位 3—I/O 错误　位 2—保留　　位 1—保留　　位 0—超时

BIOS 中断 17H 的功能 0 是打印一个字符。要打印输出的字符放在 AL 中,打印机号放在 DX 中,BIOS 最多允许连接 3 台打印机,机号分别为 0、1 和 2。如果只有一台打印机,那么就是 0 号打印机,打印机的状态信息被送回到 AH 寄存器。

例如,通过 LPT1 打印字符“A”的程序段如下。

```
mov   al, A          ;打印字符“A”
mov   dx,0           ;选择 0#打印机
mov   ah,0           ;输出打印
int   17h
```

(2) 初始化打印机端口(功能 01H)

入口参数: DX = 打印机号。

出口参数: AH = 打印机状态。

对于大多数打印机,只要一接通电源,就会自动地初始化打印机。初始化打印机使打印机各部分复位到初始值,并送回打印机状态到 AH 寄存器。在每个程序的初始化部分可以用 17H 的功能 1 来初始化打印机。

```
mov   dx,0           ;选择 0#打印机
mov   ah,01h         ;初始化功能
int   17h
```

(3) 读取打印机状态(功能 02H)

入口参数: DX = 打印机号。

出口参数: AH = 打印机状态。

如果在打印程序中先安排指令测试打印机的状态,则 BIOS 操作就会返送回状态码,通过状态码,可以了解打印机的当前状态,进而做出下一步的处理。下面一段代码用来检查打印机是否联机,是则打印,否则等待。

```
…
select:
mov     dx,0          ;选择 0#打印机
mov     ah,02h        ;读取 0#打印机的状态
int     17h
and     ah,10h        ;检查联机状态
jz      select        ;未联机,跳转
mov     al, A         ;联机,输出字符打印字符 A
mov     ah,0          ;输出打印
int     17h
…
```

6.4.3.5　串行口中断(14H 中断)

串行接口主要用作微机与微机或微机与特定类型的设备之间的接口。由于计算机通信的广泛应用,串行接口已成为个人计算机必备的部件。最常用于个人计算机上的串行接口是标准的 RS232 串行接口。这个标准串行接口既可用于近程或远程的数据通信,又可用来

连接附加的一些外部设备。每个系统中可以有两个或多个串行控制器连接到不同的外设上,如 PC 可连接 COM1 和 COM2 两个串行接口,但程序每次只能对其中一个接口进行存取。两台 PC 机或设备进行近距离通信时,可直接将它们连接;当它们进行远距离通信时,则要使用调制解调器。

通过 INT 14H 调用的串行口中断调用,可以方便地实现串行口初始化、检测串行口控制器状态以及读写字符等功能。

(1) 初始化串行口(功能 00H)

入口参数: DX = 串行口号(0 = COM1, 1 = COM2, …); AL = 初始化参数,参数的说明如表 6 - 4 所示。

表 6 - 4　参数的说明

b7b6b5(波特率)	b4b3(奇偶位)	b2(停止位)	b1b0(编码位数)
000 = 110	X0 = None	0 = 1 bit	10 = 7 bits
001 = 150	01 = Odd	1 = 2 bits	11 = 8 bits
010 = 300	11 = Even		
011 = 600			
100 = 1200			
101 = 2400			
110 = 4800			
111 = 9600			

出口参数: AH = 串行口状态,各状态位为 1 时的含义如下:

位 7—超时　　位 6—传递移位寄存器为空

位 5—传递保持寄存器为空　　位 4—发现终止

位 3—发现帧错误　　位 2—发现奇偶错

位 1—发现越界错　　位 0—接收数据准备好

AL = Modem 状态,各状态位为 1 时的含义如下:

位 7—接受单线信号诊断　　位 6—环指示器

位 5—数据发送准备好　　位 4—清除数据,再发送

位 3—改变在接收线上的信号诊断　　位 2—后边界环指示器

位 1—改变“数据准备好”状态　　位 0—改变“清除 - 发送”状态

(2) 向串行口发送字符(功能 01H)

入口参数: AL = 发送字符, DX = 串行口号。

出口参数: AL 的值不变, AH 的位 7 = 0 表示发送成功,其余各位表示串行口状态。

(3) 从串行口接收字符(功能 02H)

入口参数: DX = 串行口号。

出口参数: AL = 接收的字符, AH 的位 7 = 0 表示接收成功,其余各位表示串行口状态。

(4) 检测串行口状态(功能 03H)

入口参数: DX = 串行口号。

出口参数: AH = 串行口状态, AL = Modem 状态。

【例 6.12】　从 COM1 接收字符并把它们显示出来,如果字符没有准备好则等待,如果传送有错则显示出错信息“？”。

程序段如下:

```
Check:  mov     ah,3
        mov     dx,0
        int     14h             ;检测 COM1 的状态
        and     ah,1            ;检测字符是否就绪
        jz      check           ;没有就绪,重新检测
        mov     ah,2            ;就绪,将字符读入
        mov     dx,0
        int     14h
        test    ah,80h          ;检测是否成功读入
        jnz     error           ;不成功,转出错处理
        and     al,7fh          ;成功,屏蔽奇偶校验位,获得该字符的 ASCII 码
        mov     bx,0            ;显示读入的字符
        mov     ah,0eh
        int     10h
        jmp     check           ;转去接收下一个字符
error:
        mov     al,'?'          ;显示错误信息
        mov     bx,0
        mov     ah,0eh
        int     10h
```

6.5　软中断开发

6.5.1　软中断开发方法

为了扩充系统的功能,用户有时需要开发自己的中断处理程序。对于外部硬中断而言,中断向量通常由中断控制器提供(例如 Intel 8259A 可编程中断控制器),中断系统能够根据中断控制器提供的中断号自动转入相应的处理程序,程序员只需将中断处理程序的入口地址填入中断向量表的指定位置即可。

对于软中断程序,则是通过软中断指令“INT n”来调用,为了定义一个用户自己的软中断指令,可以按以下步骤进行。

① 确定一个空闲的中断号 n。

② 明确入口参数和出口参数。

③ 编写相应的中断处理程序。

④ 设置中断向量,即将新编写的中断处理程序入口地址写入中断向量表的 4× n 开始的 4 个字节中。

在完成了上述工作后,便可使用软中断指令 INT n 实现 n 号中断调用,执行用户自己开发的软中断处理程序。

【例 6.13】　开发一个软中断处理程序,将输入的数字密码串置换加密后保存在自

KEYBUFF 开始的单元中。要求利用空闲中断号 65H 实现软中断设置，密码不超过 20 位，以 $ 符结尾。采用如下置换密码表：

原数字：0、1、2、3、4、5、6、7、8、9。

密码字：7、8、4、6、3、0、2、9、5、1。

例如，若输入的明码为“ 123456 ”，则置换后保存的密码为“ 846302 ”。

下面给出加密程序的示例：

```
; ------------ 软中断处理程序 ----------------
cr   equ  0dh                          ;定义“ 回车 ”键
.data
  mimatab  db   7846302951             ;定义置换密码表
  keybuff  db  21 dup( $ )             ;预留 20 位密码缓冲区
.code
  public   int65h
  int65h   proc far
        push    ax                     ;保护现场
        push    bx
        push    cx
        push    si
        lea     bx,mimatab             ;bx 指向置换密码表
        lea     si,keybuff             ;si 指向密码缓冲区
        mov     cx,20                  ;最多 20 位密码缓冲区
next:
        mov     ah,1
        int     21h                    ;输入一个字符
        cmp     al,cr                  ;检测是否是回车符
        jz      exit                   ;是则转跳
        and     al,0fh                 ;不是则将数字字符转换为相应的数字
        xlat    mimatab                ;明码转换为密码
        mov     [si],al                ;保存一位密码
        inc     si
        loop    next                   ;不满 20 位，则转下一位密码输入
exit:
        pop     si                     ;恢复现场
        pop     cx
        pop     bx
        pop     ax
        iret                           ;中断返回
int65h endp
end
; ------------ 主程序 ----------------
extrn int65h:far                       ;声明外部过程
.model small
.stack 64
.code
.startup
        mov     ax,seg int65h          ;取中断处理程序的段地址
        mov     ds,ax
```

```
        mov     dx,offset int65h      ;取中断处理程序的偏移地址
        mov     ah,25h
        mov     al,65h
        int     21h                   ;填写软中断号 65h 的中断向量
        int     65h                   ;调用 65h 软中断
.exit 0
end
```

解密是加密的逆过程,在已知密码表的情况下,可以方便地实现解密操作。对于本例采用的置换密码表,显然其解密表如下:

密码字:0、1、2、3、4、5、6、7、8、9。

原数字:5、9、6、4、2、8、3、0、1、7。

例如,密码“846302”对应的明码为“123456”。读者在上述程序的基础上稍作修改,便能实现相应的解密程序。

6.5.2　中断重定向

在实际应用中,程序员有时希望临时修改某个中断的中断向量,转去执行自己的中断处理程序,处理程序返回后,又能恢复原来的中断向量。这样,在其程序运行过程中,该指定的中断将按新的处理程序来处理,程序结束后,中断系统又恢复成原来的处理方式。所以,这种中断向量的改变对其他程序来说是透明的。要实现中断重定向,可以按以下步骤来完成。

①读取指定中断的中断向量,并把它保存在存储单元内。

② 把用户编写的程序设置为指定中断的新处理程序。

③ 在用户程序结束之前,恢复成处理该中断的入口地址。

下面程序段给出了程序示例:

```
…
intno   equ  40h                    ;假设被修改的中断号
oldaddr      dd     ?
…
newfunc      proc                   ;用户编写的新的中断处理程序
…
iret
newfunc      endp
…
;以下是用户编写的主程序
mov     al,intno
mov     ah,35h
int     21h
mov     word ptr oldaddr,bx         ;保存原入口地址
mov     word ptr oldaddr+2,es
mov     dx,offset newfunc           ;设置新的入口地址
mov     ax,seg newfunc
mov     ds,ax
mov     al,intno
mov     ah,25h
int     21h
```

```
…
int     40h                          ;执行用户的中断处理 newfunc
…
mov     dx,word ptr oldaddr          ;恢复原入口地址
mov     ds,word ptr oldaddr +2
mov     al,intno
mov     ah,25h
int     21h
…
int     40h                          ;执行原来的中断处理程序
…
```

利用中断重定向技术,可以实现对某些系统中断的改向,从而完成一些特殊的功能。用户可以利用中断重定向技术进行系统跟踪,可以通过截获系统中断来触发执行自己的程序,某些病毒程序正是利用截获系统中断并使之改向的方法达到感染系统的目的。

6.5.3　驻留中断程序

一般程序都是由系统加载到内存后执行,运行结束后又退出内存,这种程序称之为暂驻程序。而对于许多突发事件,如在不终止当前运行程序的情况下,想随时查看一下系统时间,或者定时时间已到,则暂时中断当前程序的运行,而转向事件处理程序进行处理等。这些突发事件的处理都是暂驻程序难以做到的。只有将这些对突发事件的特殊处理程序预先放置在内存的某个区域驻留下来,在执行暂驻程序过程中突发事件发生时,临时调用驻留内存的相应处理程序进行处理,处理完后再返回暂驻程序继续执行。这种内存驻留程序简称 TSR(Terminate and Stay Resident) 程序,它能快速、方便、及时地处理许多暂驻程序不能处理的事件。

TSR 程序的结构,从功能上可以分为常驻内存部分和自举部分,通常所说的 TSR 是指 TSR 常驻内存部分。而自举部分则完成 TSR 常驻内存部分接管中断向量并驻留内存的工作。实现 TSR 中断程序通常需要考虑以下几个环节。

程序自举: 完成 TSR 常驻内存部分接管中断向量并驻留内存。

获得控制: 激活 TSR 常驻内存部分。

执行处理程序: 执行 TSR 的常驻内存处理程序。

交出控制: 当 TSR 处理程序部分执行完毕时,即把控制权交还给被中断的进程。

1. 程序自举

程序自举部分要完成两项工作,一是让 TSR 常驻内存部分接管中断向量,以便激活 TSR;二是让 TSR 常驻内存部分驻留内存。INT 21H 的 25H 功能可为指定的中断号设置新的入口地址,其使用方法如下:

入口参数: AH = 中断号, DS: DX = 中断处理程序的入口地址。

出口参数: 无。

INT 21H 的 31H 功能可以完成 TSR 常驻内存部分驻留内存,而起安装作用的自举程序随之退出。其使用方法如下:

入口参数: AH = 31H , DX = 程序驻留的容量(单位为节, 1 节 =16 个字节) 。

出口参数: AL = 返回代码(0 表示没有错误) 。

2．获得控制

TSR 获得控制称为激活 TSR。目前普遍采用的两种激活方式是定时激活和“热键”激活。定时激活就是把 TSR 程序挂接到定时中断上，如 INT 08H 定时硬件中断上。而“热键”激活通常是把 TSR 程序挂接到键盘硬件中断 INT 09H 上，通过某个(些)特殊的按键来激活 TSR 运行。因此，实现 TSR 离不开中断。正是由于中断机制，TSR 程序才能在潜伏之后得以“唤醒”而投入执行。

3．执行处理程序

这一部分是 TSR 真正要做的事情，其余部分的工作都是在做准备工作。一旦激活 TSR 的条件满足，TSR 的处理部分就开始工作。

4．交出控制

当 TSR 处理部分执行完毕时，即把控制权交还给被中断的进程，这时 TSR 仍驻留在内存，等待再次激活。另一种交出控制的方式是撤离 TSR 功能，即结束 TSR，这种情况除了释放 TSR 所占的内存及恢复相关的中断向量外，还需交还 TSR 所曾占用过的一切资源。因此，这种复杂的撤离操作一般由 DOS 来做。

由此，我们得出一个 TSR 程序能够驻留内存并可以激活运行的一般方法。

① 获取一个原有中断向量的入口地址并将其保存起来。

② 修改这个中断向量入口地址使其指向准备驻留程序的开始处。

③ 完成程序驻留部分驻留内存。

其中，①是为了保持原有的中断调用功能而采取的接管方法，即在驻留内存的部分执行完成后仍能调用原来的中断处理程序。如果驻留部分根本不需要调用原来的中断处理程序，则可略去①。②是确保在发生该中断时能够调用执行这个 TSR 程序(被激活)，否则，这个 TSR 程序将无法得到执行。③是完成 TSR 程序驻留部分驻留内存，自举部分的主程序退出。

下面给出一个利用 80H 号中断的驻留中断服务程序的例子。为简化起见，本例不考虑调用 80H 原来的中断处理程序。

【例 6.14】　利用 80H 中断，设计一个 TSR 驻留中断服务程序，用来显示信息：“This is a TSR!”。

程序如下：

```
.model tiny
.code
.start p                              ;等价于 org 100h
       jmp   start                    ;跳转到主程序的开始执行位置
newint80h  proc                       ;驻留的中断服务程序
       sti                            ;开中断
       push  ax                       ;保护现场
       push  bx
       push  cx
       push  si
       mov   si,offset intmsg         ;得到显示字符串的首地址
       mov   cx,sizeof intmsg         ;计算显示字符串的长度
disp:
```

```
        mov   al,cs:[si]              ;取一个字符
        mov   bx,0
        mov   ah,0eh
        int   10h                     ;显示一个字符
        inc   si
        loop  disp                    ;取下一个字符
        pop   si                      ;恢复现场
        pop   cx
        pop   bx
        pop   ax
        iret                          ;中断返回
intmsg db  This is a TSR!  ,0dh,0ah     ; 中断服务程序显示的信息
newint80h  endp                       ;中断服务程序结束
start:
        mov   ax,cs                   ;主程序开始位置
        mov   ds,ax ;
        mov   dx,offset newint80h
        mov   al,80h
        mov   ah,25h
        int   21h                     ;设置80h 的中断向量
        int   80h                     ;调用一下查看
        mov   dx,offset istmsg        ;显示驻留成功
        mov   ah,9
        int   21h
        mov   dx,(offset start) +15   ;计算驻留程序的长度(需要多加15 个字节)
        mov   cl,4
        shr   dx,c1                   ;除以16 转换成“节”
        mov   ah,31h                  ;中断服务程序驻留后,主程序返回DOS 系统
        int   21h
istmsg db  INT80His installed!  ,0dh,0ah, $
end
```

该程序执行后,驻留的中断服务程序将常驻主存,而起安装作用的主程序随之从主存中消失。80H 中断服务程序驻留主存后,若在其他任何程序中执行 80H 号中断,就会在屏幕上显示“This is a TSR!”信息。

习　　题

1. 简述 I/O 端口的概念,I/O 端口的地址空间是如何划分的?

2. 简述不同的 I/O 控制方式的特点及主要应用场合。

3. 简述中断处理程序的结构框架,说明中断和子程序调用之间的主要区别是什么。为什么要区分 IRET 指令与 RET 指令?

4. 简述中断向量表的结构,说明中断系统是如何根据中断向量号获得中断处理程序入口地址的。

5. 编写程序段,轮流测试两个设备的状态寄存器,只要一个状态寄存器的第 0 位为 1,则与其相应的设备就输入一个字符;如果其中任一状态寄存器的第 3 位为 1,则整个输入过程结束。两个状态寄存器的端口地址分别是 24H 和 96H,与其相应的数据输入寄存器的端口则为 26H 和 98H,输入字符分别存入首地

址为 BUF1 和 BUF2 开始的存储区中。

6. 设中断类型 9 的中断处理程序的首地址为 INT9PRO, 给出为中断类型 9 设置中断向量的程序段。

7. 简述系统功能调用和 BIOS 中断的作用和一般调用方法。

8. 编写一个子程序, 它可显示以 0 结尾的字符串。子程序的入口参数 DS: DX 为待输出字符串的首地址。

9. 编写一个子程序, 用来读入一个键, 并在屏幕上按十六进制的形式显示按键的扩展 ASCII 码, 如果按键为普通字符, 则不显示。

10. 编写程序, 显示鼠标的按键状态, 若按左键, 显示 Left, 若按右键, 显示 Right, 按 Esc 键, 程序结束。

11. 编写一个程序, 在屏幕的右下角闪烁显示编程者自己的姓名, 显示颜色自定。

12. 假设显示器的显示模式设置为 12H, 编写实现下列功能的程序。

(1) 在屏幕中间从上到下显示一条明亮的蓝色线, 线宽为 1 个像素。

(2) 在屏幕底下横向画一条绿色线, 线宽为 2 个像素。

(3) 在屏幕上垂直显示 16 种颜色, 每种颜色宽 40 个像素。

(4) 设置屏幕背景为白色, 在屏幕中间画一条青色线, 线宽为 10 个像素。

13. 编写程序, 检测计算机是否已安装了鼠标, 并以显示 Yes/No 来表示检测结果。

14. 简述软中断的开发方法和应该注意的问题。

15. 编写一个 INT 95H 的软中断处理程序, 实现响铃功能, 并设置相应的中断向量。

16. 编制一个改写 INT 95H 的重定向程序, 实现显示 Welcome! 的功能。

第 7 章　32 位指令及其编程

由于高级语言都具有向程序员隐藏许多普通的和重复性细节这样一个非常好的特点，从而使程序员可以专注于他们所要实现的目标，因此目前高级语言在程序设计中得到了广泛的应用，汇编语言的使用则相对少一些。然而，当编写直接处理硬件的代码或编写对性能要求极高的代码时，程序员就必须使用比较低级的语言。汇编语言是最接近硬件的编程语言，这就很自然地使它成为上述情况下最常使用的一种编程语言。

由于 32 位 CPU 的系统结构、寄存器组、工作方式以及寻址方式与前面学习的 16 位微处理器有很大的区别，所以在本章中，我们首先学习 32 位 CPU 新增的寻址方式和 32 位寄存器组，理解用 16 位指令如何实现 32 位扩展，熟悉常用 16 位指令的 32 位扩展功能及应用，同时要对新增指令 PUSHA, POPA, MOVSX, MOVZX, INS, OUTS, JECXZ, BT, BTC, BTR, BTS, BSWAP, CMPXCHG 等指令的功能有详细的了解；然后学习 32 位 CPU 的 3 种工作方式；最后介绍 32 位程序的编程方法。

7.1　32 位微处理器结构

7.1.1　80386 微处理器结构

80386 是一种典型的 32 位微处理器，具有较强的功能。下面首先介绍一下 80386 微处理器的结构。80386 微处理器内部采用流水线结构，使用二级存储器管理方式，支持虚拟存储器的使用；具有 9 种寻址方式和丰富的指令系统，可支持多任务/多用户操作系统的运行；可处理无符号/带符号的 8 位、16 位、32 位、64 位数据以及 BCD 码、ASCII 码数据、1～32 位的位操作；支持 80387 数值协处理器，可进行浮点及函数运算。

80386 微处理器的内部组成如图 7 - 1 所示，可分为执行部件(EU)、存储器管理部件(MMU)和总线控制接口部件(BIU)三大部分。采用流水线工作方式，形成指令预取、指令译码、地址生成、取操作数、执行指令、存储结果等 6 级流水段。一般情况下，任一流水线由前一指令使用后，进入下一流水段时，后一条指令即可使用该流水段。由此形成 6 级流水作业式的多条指令的并行过程。

执行部件 EU 由预取指令部件、指令译码部件、控制部件、算术/逻辑部件 ALU、寄存器组、乘/除硬件、桶形移位器及保护检测部件组成。其中 ALU、乘/除硬件、寄存器组、桶形移位器统称为数据部件，执行部件由总线接口部件获得指令，并将指令送至预取指令队列中排队等候执行。执行时，指令被逐条取出送指令译码部件。译码结果送到已译码指令队列中，根据其代码控制微程序控制器(控制 ROM)执行其微指令。ALU 用来进行算术/逻辑运算，乘/除硬件和桶形移位器主要进行快速乘/除法运算。对于转移/调用指令，由指令译码部件译码后将转移/调用地址送存储器管理部件转换成物理地址，然后再送总线接口部件提取转移/调用的指令序列并送入预取指令队列。保护检测部件在程序执行过程中由微指令代码

控制,用来检测静态并且与段有关的错误,即检测是否段越界。

存储器管理部件包括分段部件、分页部件和保护检测部件,实现对存储器的段页式管理,把指令中给出的逻辑地址转换成存储单元的物理地址,每页 4 KB 单元,每段最多 1 024× l024个页,因此最大段存储空间为 4 096 MB。若采用虚拟存储器技术,则最大存储空间为 64 TB。为了提高存储器的访问速度,在构成微机系统时可配置高速缓冲存储器(Cache),这样可构成完整的 Cache—主存—外存三级存储体系。

总线接口部件包括地址驱动器、流水线/总线宽度控制逻辑电路、MUX/收发器。其作用是与外部联系,访问存储器,预取指令,读/写数据,访问 I/O 设备等。指令取出后送预取指令队列排队。另外,在总线接口部件中还设有总线请求判优器,用以控制与其他主设备连接,构成多机系统。

图 7 - 1 右侧的信号为 80386 CPU 的外部总线信号,这些信号经过隔离、驱动及有机的组合,构成了系统总线信号。有许多信号也可直接作为局部总线信号使用,比如 M/$\overline{IO}$信号可区分 CPU 的操作是存储器操作(=1) 还是 I/O 操作(=0) 。M/$\overline{R}$ 信号可区分是写操作(=1) 还是读操作(=0) 等。

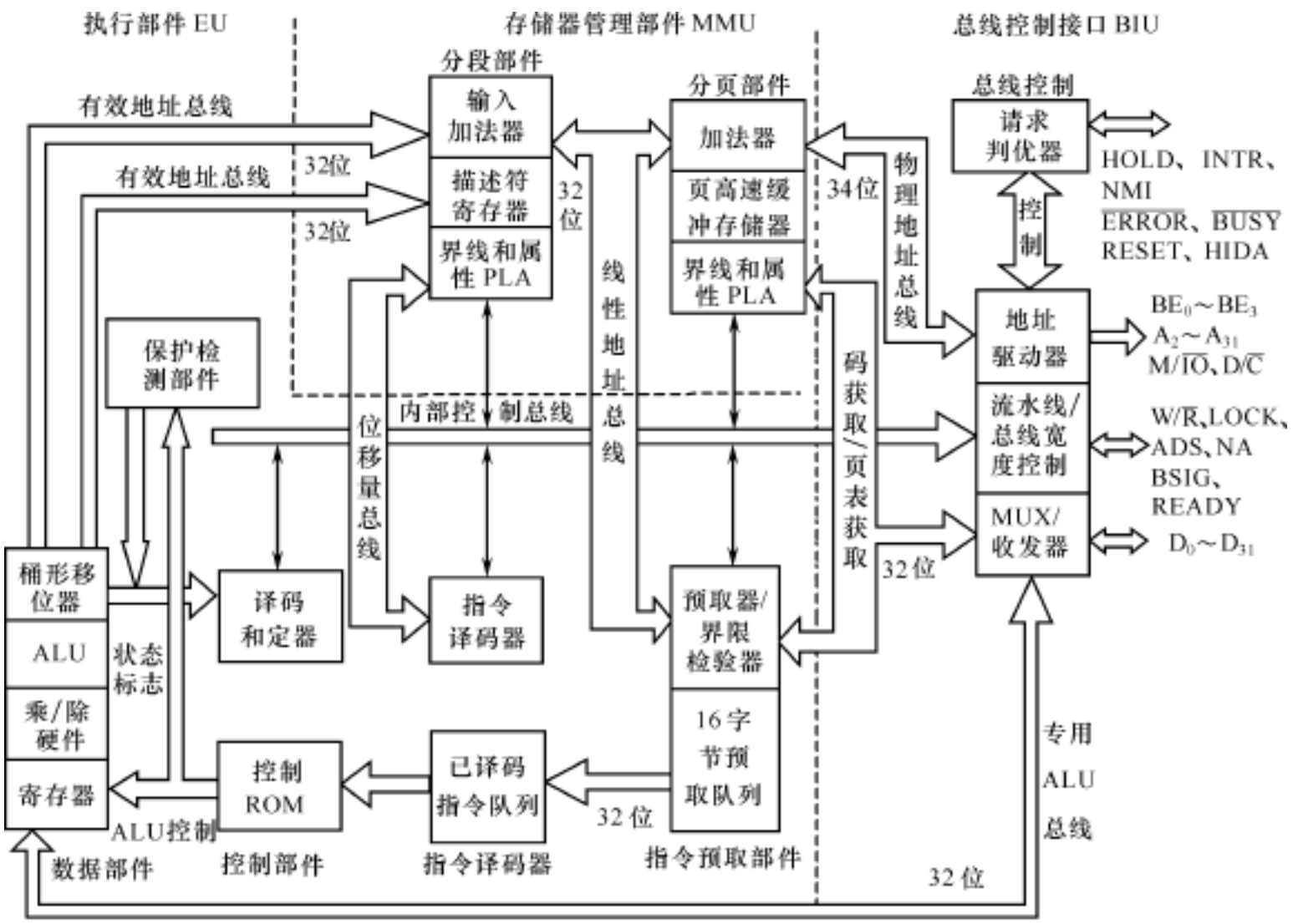


图 7 - 1　80386 内部结构框图

7.1.2　Pentium 微处理器结构

相对于 80386 微处理器而言, Pentium 微处理器在结构上又进行了一定的改进,其结构图如图 7 - 2 所示。下面简要介绍一下 Pentium 处理器在超标量流水线、分立的指令 Cache

和数据 Cache、重新设计的浮点单元及转移预测等 4 个方面的新型体系结构的特点。

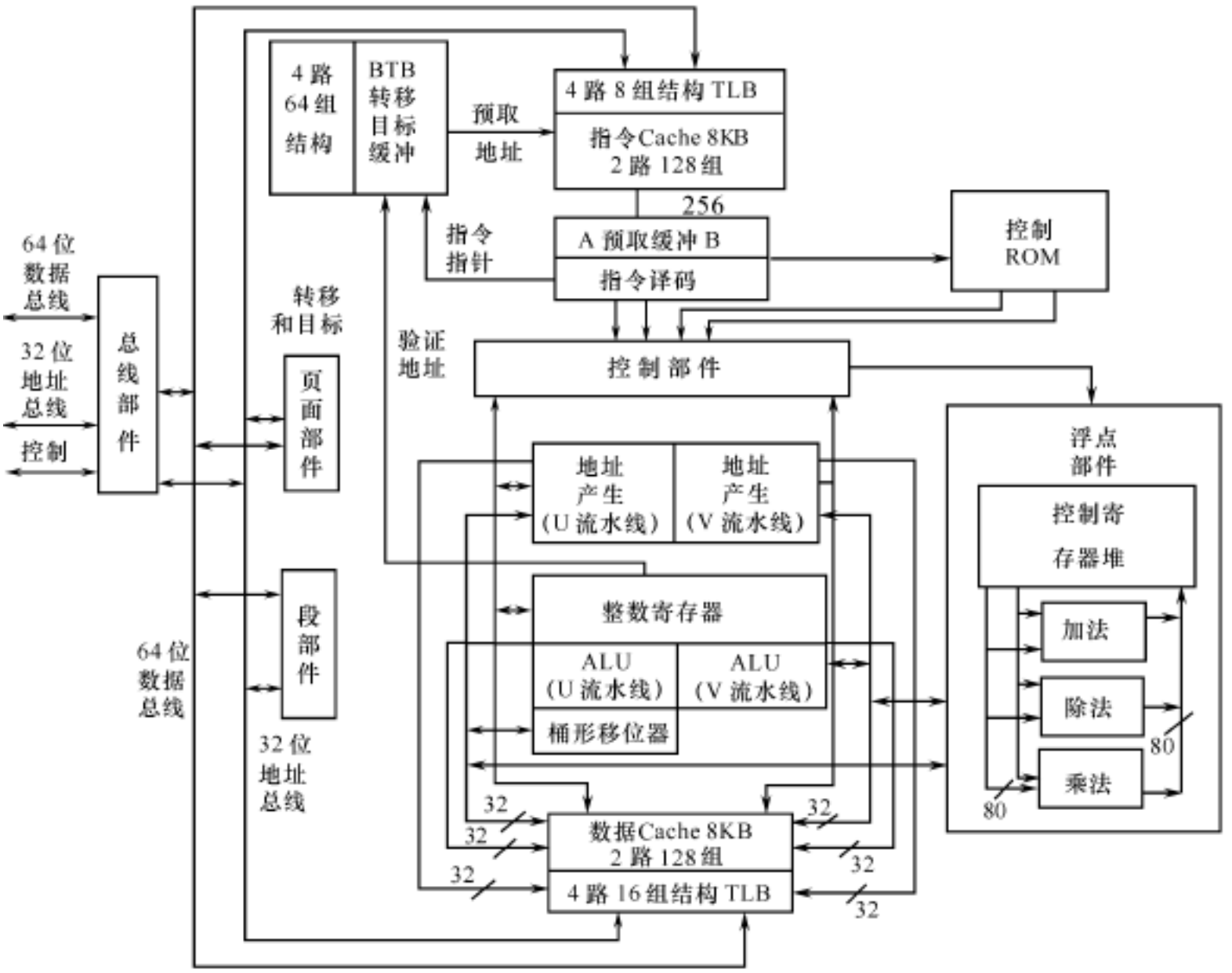


图 7 - 2　Pentium 微处理器结构框图

1. 超标量流水线

超标量流水线是 Pentium 系统结构的核心。它由 U 和 V 两条指令流水线构成，每条流水线都有自己的 ALU、地址生成电路和与数据 Cache 的接口。因而，有可能在 1 个时钟周期内执行两条整数指令，每条流水线执行 1 个指令。这种指令并行方式要求指令必须是简单指令，且 V 流水线总是接受 U 流水线的下一条指令。U 流水线能执行指令集的任何指令，包括指令前缀；而 V 流水线却不能，它只能执行“指令配对法则”中规定的简单指令。若连续的两条指令不能配对，则只能使用 U 流水线先后执行这两条指令。U，V 流水线采用的是按序发射、按序完成的调度策略。注意，在这种策略下只要指令 Cache 及其 TLB 和数据 Cache 及其 TLB 连续命中，就可每时钟连续完成两条可匹配指令。

与 486 流水线相类似，Pentium 双流水线中每一条也分为 5 段，即指令预取(PF) 、指令译码(D1) 、地址生成(D2) 、指令执行(EX) 和结果写回(WB) 。D1 段除完成指令译码外，还要完成指令配对检查和条件转移预测。带有前缀的指令，要求有一个额外的 D1 周期。经检查合格的配对指令，由 D1 段同时发射给下一段时，才真正开始两条流水线的并行工作。

2．分立的指令 Cache 和数据 Cache

80486 片内有 8 KB 的指令和数据合一的 Cache，而 Pentium 则是将指令 Cache 和数据 Cache 分立，各为 8 KB。指令 Cache 是只读的（应用程序不能改写，但可由操作系统实行替换算法进行新旧行的更替）指令代码。数据 Cache 是可读写的，双端口每端口 32 位，与 U，V 两条流水线交换整数数据，或组合成一个 64 位端口与浮点运算部件交换浮点数据。两个 Cache 与 64 位数据、32 位地址的 CPU 内部总线相接。

指令 Cache 和数据 Cache 都是两路相连的结构，每行 32 字节。数据 Cache 可设置成行写回或写直达方式，并且遵守 MESI 协议来维护一级、二级 Cache 的一致性。个别页面的访问可用硬件或软件设置成是否可以高速缓冲，也可用硬件或软件来禁止或使用 Pentium 的 Cache 功能。

它们都是物理地址（实地址）Cache。每个 Cache 都有一个转换后援缓冲器 TLB，负责将 TLB 中的线性地址转换成 32 位物理地址。

分立的指令和数据 Cache 是对 Pentium 超标量结构的有力支持。它不仅使指令预取和数据读写能无冲突地同时完成，而且可同时与 U，V 两条流水线分别交换数据。

3．重新设计的浮点运算部件

同 80486 DX 一样，Pentium 也将浮点运算器包含于芯片内，但 Pentium 的浮点运算部件进行了重新设计，执行过程是分为 8 段的流水线。前 4 段为指令预取（PF）、指令译码（D1）、地址生成（D2）、取操作数（EX），在 U，V 流水线完成；后 4 段为执行 1（X1）、执行 2（X2）、结果写回寄存器堆（WF）、错误报告（ER），在浮点运算部件中完成。一般情况下，只能由 U 流水线完成一条浮点操作指令。少数情况下，V 流水线也能同时完成一条指令，例如浮点数交换指令可利用 V 流水线同时完成。

浮点部件内有浮点专用的加法器、乘法器和除法器，有 8 个 80 位寄存器组成的寄存器堆，内部的数据总线为 80 位宽。支持 IEEE 754 标准的单、双精度格式的浮点数，另外还使用一种称为临时实数的 80 位浮点数。对于浮点数的常用指令如 LOAD，ADD，MUL 等采用了新的算法并予以固化，用硬件来实现，其执行速度是 80486 的 10 倍多。

4．以 BTB 实现的动态转移预测

Pentium 采用动态转移预测技术，来减少由于过程相关性引起的流水线性能损失。它提供的转移目标缓冲器 BTB 是个小容量的 Cache，当一条指令导致程序转移时，BTB 记住这条指令及其转移目标地址。以后遇到这条转移指令时，BTB 会依据前面转移发生的历史来预测这次是转移取还是顺序取，若是预测为转移取则记录的转移目标地址将被立即送出。

Pentium 的超标量流水线有两个指令预取缓冲器，当前总是使用其中一个。当在指令译码（D1）段译出一条转移指令时立即检索 BTB。若预测为顺序取则继续以当前预取缓冲器取指令，若预测为转移取则立即冻结当前预取缓冲器，启动另一个预取缓冲器，由给出的转移目标地址处开始取分支程序的指令序列。因而保证了流水线的指令预取步骤永远不会空置，而且预测转移取不正确时，正确路径的指令已在指令预取缓冲器中，也使因预测错误而蒙受的性能损失减小。

除了上述特点外，Pentium 在数据完整性、容错性和节电等方面也采取了一些特殊的设计方法。

7.1.3　Pentium 微处理器基本寄存器组

Pentium 的基本结构寄存器组的组织结构如图 7 - 3 所示, 下面分三部分进行介绍。

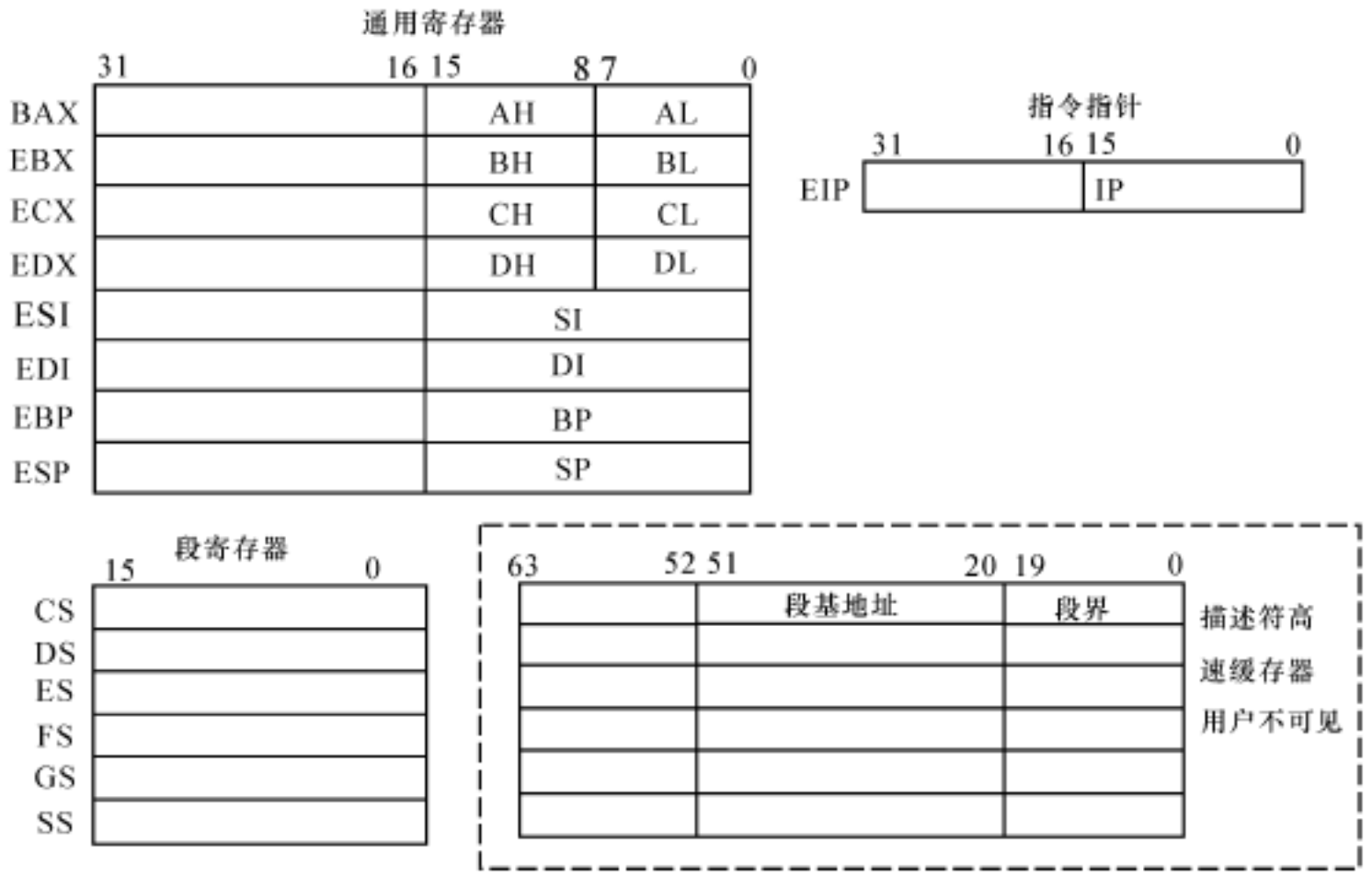


图 7 - 3　Pentium 基本结构寄存器组

1. 通用寄存器

8 个 32 位通用寄存器, 用于保存数据和地址, 它们命名为 EAX, EBX, ECX, EDX, ESI, EDI, EBP 和 ESP。这些寄存器的低 16 位可作为 16 位寄存器单独使用, 它们是 AX, BX, CX, DX, SI, DI, BP 和 SP, 使用时不影响高 16 位的值。AX, BX, CX, DX 四个寄存器的高、低 8 位可以单独使用来保存 8 位数据, 它们是 AH, AL, BH, BL, CH, CL, DH, DL, 但它们不能用于有效地址计算。

2. 段寄存器

Pentium 有 6 个段寄存器, 每个都是 16 位长, CS 是代码段寄存器, DS, ES, FS, GS 是数据段寄存器, SS 是堆栈段寄存器, 每个段寄存器都有一个相应的 64 位描述符高速缓存器, 这些高速缓存器是用户不可见的。

在保护模式下, 段寄存器的内容是选择符。其中, b1, b0 位为请求特权级 RPL, b2 为表指示位 TI, 为 1 时访问局部描述符表 LDT, 为 0 时访问全局描述符表 GDT; b3 ~b15 为选择子, 这 13 位选择子(× 8) 为访问描述符表的索引值。不论什么时候改变了段寄存器的内容, 与该选择符对应的 8 字节(64 位) 描述符就被分段部件自动从位于内存的描述符表取出, 装载到相应的描述符高速缓存器中。一旦被装入, 对那个段的全部访问都使用此高速缓存器中的描述符信息, 直到下次改变。这样, 不用每次的段访问都去查表, 从而加快了访问速度。

在实模式下,段寄存器的内容左移 4 位是段基址的高 16 位(低 4 位全为 0),其对应的描述符高速缓存器继续起作用,只不过这 64 位不是从描述符表取来的而是自动设定的。其中,段基地址为段寄存器值× 16,段界为 0FFFFH,段属性为 16 位寻址时各段的存取属性(如可执行否、可读否、可写否、段扩展方向等),值得注意的是此时特权级为 0(最高)。

3. 指令指针和标志寄存器

指令指针是一个 32 位寄存器,名为 EIP,它保存下一条待取出指令的代码段的段内偏移值,即总是相对于 CS 段基地址的值。EIP 的低 16 位名为 IP,是 16 位的指令指针,供 16 位寻址时使用。

标志寄存器 EFLAGS 是一个 32 位寄存器,低 16 位名为 FLAGS,也是在 16 位寻址方式时使用。EFLAGS 的第 1、3、5 和 22 ~31 位,Intel 公司未定义,其余的 19 位大多反映操作结果的状态(S),但也有少数位起着控制作用(C),其余为系统标志(X)。下面结合图 7 - 4 介绍各标志位的意义。

31 ~22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IOPL	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	0	CF

图 7 - 4　标志寄存器 EFLAGS

CF = 进位标志(S),PF = 奇偶校验标志(S),AF = 辅助进位标志(S),ZF = 零标志(S),SF = 符号标志(S),TF = 陷阱标志(X),IF = 中断允许标志(X),DF = 方向标志(C),OF = 溢出标志(S0)。这 9 个标志是 8086 处理器标志,一直被继承下来。

IOPL(b13,b12) = I/O 特权值(X),用于指明在保护模式下不产生异常中断 13 而执行 I/O 指令所要求的最大 CPL(当前特权级)的允许值。

NT (b14) = 嵌套任务标志(X),该位置位指明当前任务嵌套在另一个任务中执行,用于保护模式。NT = 1 时执行 IRAT 指令引起 TSS 反向链装入 TR,返回父任务,否则返回同任务。

RF (b16) = 恢复标志(X) Z,是与调试寄存器的断点操作一起使用的标志。在该位置位时即使遇到断点或调试故障,也不产生异常中断 1。在成功地执行每条指令时,RF 自动被复位。一进入断点处理程序先将此位置 1,然后压入堆栈,断点处理完毕时,将其弹出,使以后的指令不按断点指令执行,即断点指令只实行一次。

VM (b17) = V86 模式标志(X),在该位置位时表明当前工作在虚拟 86(V86)模式下。

AC (b18) = 对准检查标志(X),所谓对准是访问字数据(16bit)时地址应为偶数,访问双字数据(32bit)时地址应为 4 的倍数,访问 4 字数据(64bit)时地址应为 8 的倍数。若该位置位,进行未对准地址访问时将产生异常中断 17。只有在特权级 3 时此位有效。

VIF(b19) = 虚拟中断标志(X)。

VIP(b20) = 虚拟中断挂起标志(X)。

虚拟中断(Virtual Interrupt)用于多任务环境中,虚拟中断标志是所用中断标志的虚拟映像,虚拟中断挂起标志指示虚拟中断是否被挂起。

ID(b21) = 识别标志(X),若此位能被置位和复位,则指明这个处理器能支持 CPUID 指令。该指令能提供处理器的厂商、系列和模式等信息。

VIF, VIP 和 ID 这 3 个标志位是 Pentium 比 80486 新增加的标志位。

7.1.4　Pentium 微处理器系统级寄存器组

系统级寄存器组包括 4 个表(或段) 的基地址寄存器 GDTR, IDTR, LDTR, TR 和 5 个控制寄存器 CR0, CR1, CR2, CR3, CR4。所谓系统级寄存器组是指这些寄存器只能由特权级 0 的系统程序访问。换句话说, 这些寄存器由操作系统核心维护和使用。系统级寄存器组的框图如图 7 - 5 所示。下面分两部分来介绍。

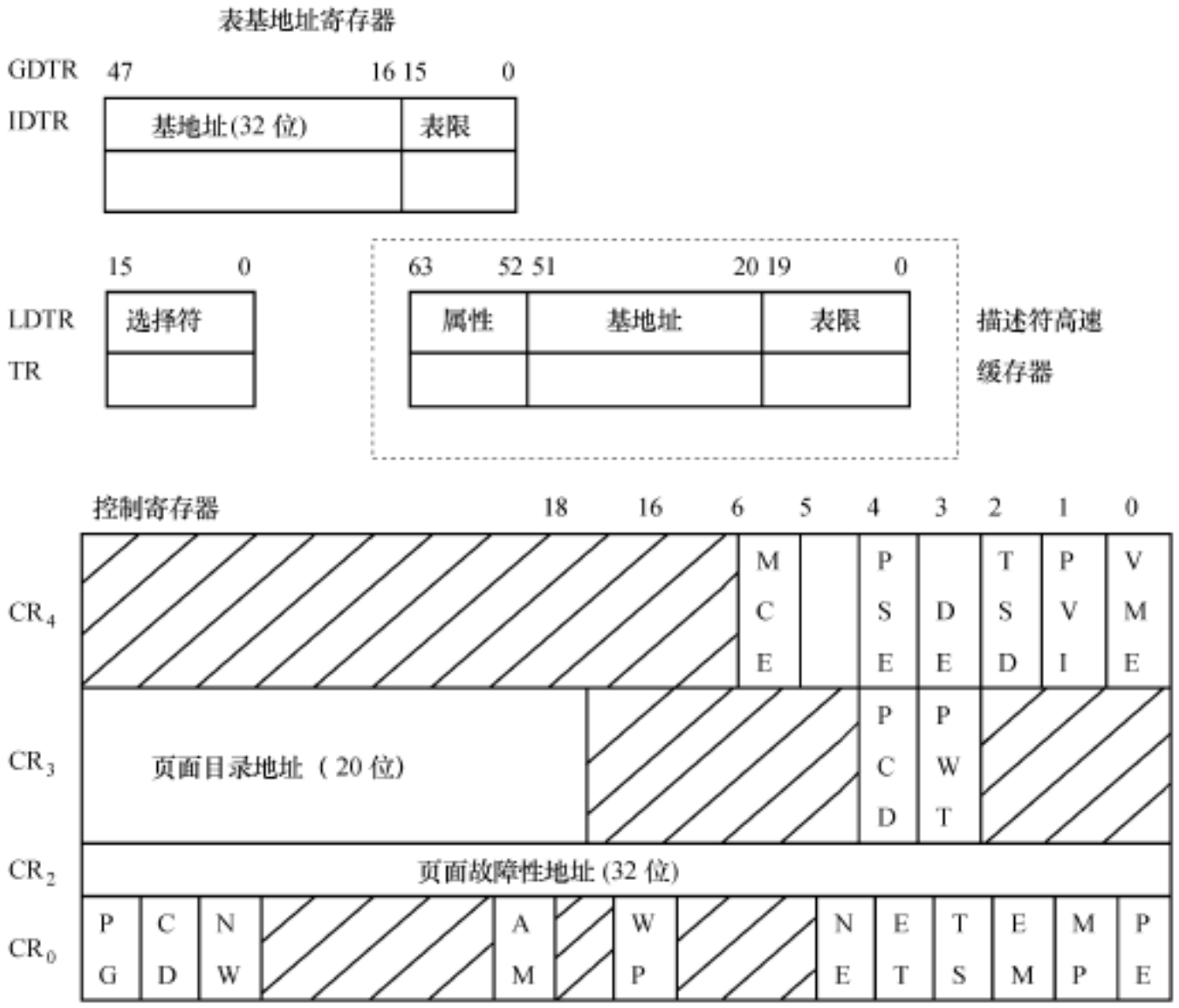


图 7 - 5　Pentium 系统级寄存器

1. 表基地址寄存器

Pentium CPU 维护和管理位于主存的 4 类系统表(或段) , 它们是全局描述符表 GDT、中断描述符表 IDT、局部描述符表 LDT 和任务状态段 TSS。TSS 保存有任务的环境, 默认时只有 104 字节, IDT 存放中断门描述符、陷阱门描述符和任务门描述符。LDT 和 TSS 每个任务各一个, 系统中可以存在多个 LDT 和 TSS。任何时候系统只有一个 GDT 和一个 IDT。

GDT 的存储位置由全局描述表寄存器 GDTR 指向, IDT 的存储位置由中断描述表寄存器 IDTR 指向, 两个表基地址寄存器还有 16 位的表限字段, 即表最大长度为 64 KB。而多个 LDT 和 TSS 表(或段) 的基地址、表(或段) 限、属性等又以描述符的形式登记在 GDT 中。当前使用的 LDT 和 TSS, 分别由局部描述符表寄存器 LDTR 和任务寄存器 TR(各为 16 位长) 以选择符形式指出, 即以指定的选择符查找 GDT, 读取相应的描述符并装入各自相应的描

述符高速缓存器, 才得到当前 LDT 和 TSS 的基地址及界限和属性等信息。

2. 控制寄存器

80286 CPU 只有机器状态字 MSW, 即 CR0 的低 16 位。80386 CPU 开始具有 CR0 ~CR3 四个控制器, 到 Pentium 又增加了 1 个控制寄存器 CR4, 并对 CR0 的 CD, NW 位进行重新定义。其中, CR1 被 Intel 公司保留, 未使用。

(1) CR0 寄存器控制位

CR0 寄存器的系统控制位用于控制操作模式或指示处理器正常工作的状态。各位含义如下:

PE = 保护模式允许位, 允许/禁止保护模式。

MP = 监督协处理器(Momtor Coprocessor) , 该位为 1/0, 表示有无协处理器, 仅当在 Pentium 上运行 80x86 的程序时才对此位感兴趣。

EM = 仿真协处理器控制位, 在该位置位时表示用软件仿真协处理器, 此时, CPU 遇到 ESC(协处理器指令标志) 指令, 产生中断 7, 即可用中断处理程序进行仿真操作。

TS = 任务切换位, 每当任务切换时由硬件将其自动置位, 以后由软件复位。与 EFLAGS 寄存器的 NT 标志位共同管理任务切换。

ET = 扩展类型控制位, 在 80386 机器上用于支持数字协处理器指令, Pentium 不使用。

NE = 数值错处理控制位, 执行浮点指令发生故障时, 若在该位置位则用异常中断 16 进行处理, 否则以外部中断处理。

WP = 写保护控制位, 当此位清除时监控程序能向用户级的只读页进行写入。这个特征可用来支持进程生成, 对某些操作系统是有用的。

AM = 对准检查允许控制位, 该位置位时, 地址对准检查有效。注意对准检查只是在保护模式下起作用。

NW = 非写直达控制位, 在该位置位时, 数据 Cache 写直达操作被禁止。

CD = Cache 禁止控制位, 该位为 1/0, 表示禁止/允许内部 Cache 的填充操作。

PG = 分页控制位, 该位为 1/0, 表示允许/禁止分页工作模式。

(2) CR2 和 CR3 控制寄存器

当 CR0 的 PG 位为 1(即允许分页) 时, CR2 和 CR3 控制寄存器有效。在产生页故障中断(中断 13, 缺页中断) 后, 最后访问页面的 32 位线性地址保存于 CR2 控制寄存器中。

CR3 控制寄存器的高 20 位是页目录表基地址的高 20 位。因页目录表的大小为 1 页, 故页目录表基地址必以 4 KB 为边界, 此基地址的低 12 位为全 0, CR3 的页目录表指针作用参见图 7 - 5, CR3 还有两个控制位 PCD 和 PWT, 用于控制 CPU 的两个同名输出引脚。

PCD = 页面 Cache 禁止控制位, 该位为 1 禁止页面 Cache 操作, 该位为 0 允许 Cache 操作。

PWT = 页面 Cache 写直达控制位, 该位为 1 允许页面采用写直达法的 Cache 写策略, 该位为 0 允许的 Cache 写策略。

当然, 这两位的作用一定要在 CR0 的 CD = 0(Cache 允许) 和 PG = 1(分页方式) 的前提下。另外, CPU 的 PCD 和 PWT 输出引脚, 既受 CR3 的 PCD, PWT 控制位影响, 也受页目录项和页表项中的 PCD, PWT 位的影响。当前 PCD 和 PWT 输出引脚, 按照 Pentium 手册规定

是由页面地址转换控制,即由当前页面的页表项 PCD, PWT 控制。

(3) CR4 控制寄存器

CR4 控制寄存器增添了 6 个控制位,即 VME, PVI, TSD, DE, PSE, MCE 控制位。

VME = V86 模式扩充允许位,该位为 1 时允许虚拟 86 模式扩充,否则禁止扩充。

PVI = 保护模式虚拟中断允许位,该位为 1 时允许,否则禁止虚拟中断。

TSD = 时间标记禁止位,该位为 1 而且当前特权级不为 0 时,禁止 RDTSC(读时间标记计数器)指令,该位为 0 时,允许在所有特权级上执行 RDTSC 指令。

DE = 调试扩充允许位,该位为 1 时允许 I/O 断点,这使处理器能在 I/O 读写时中断,该位为 0 时禁止此调试扩充能力。

PSE = 页面大小扩充允许位,该位为 1 时允许使用 4 MB 分页方式。

MCE = 机器检查中断允许位,当一个总线周期不能成功完成或在一个读总线周期出现数据奇偶错时,若该位为 1 则产生机器检查中断。

系统级寄存器的设置充分体现了 CPU 对软件、操作系统和虚拟存储器段、页式映像的支持。

7.2　80x86 CPU 的工作方式

自从微处理器发展到 32 位结构,它已采用了过去只有在大、中型计算机系统才能见到的先进存储管理策略,而且,在多数情况下,微处理器的存储管理策略要优于这些大、中型系统最初采用的策略。

80x86 有 3 种工作方式:实模式、保护模式和虚拟 8086 模式。下面具体介绍这 3 种工作模式。

1. 实模式

实地址模式也简称为实模式,这是自 8086 一直延续继承下来的 16 位模式。逻辑地址形式为段地址:偏移地址,二者均为 16 位。将段名所指定的段寄存器的内容乘以 16(即左移 4 位),得到 20 位段基址,加上 16 位偏移即得 20 位物理地址。

实模式使用 A0 ~A19 的 20 根地址线,最大物理地址空间为 1 MB。最高物理地址为 FFFFF(H),若段基址加上偏移地址计算出的物理地址超过 20 位,则超出位被丢弃,即出现地址环绕现象。例如 FFFF: FFFF 计算出的地址为 10FFEF,实际送出的物理地址为 0FFEF。

MS-DOS 操作系统、运行于实模式时的 Windows 3. x 和它们的 16 位应用程序采用实模式。

2. 保护模式

受保护的虚拟地址模式(Protected Virtual Address Mode)简称为保护模式。这是 80386 才具备并一直延续下来的 32 位模式。Pentium 的存储管理部件 MMU 设有分段部件 SU 和分页部件 PU,允许 SU, PU 单独工作或同时工作。于是保护模式又细分为如下 3 种模式。

(1) 分段不分页模式

本模式下的虚拟地址(或逻辑地址)采用段式映像产生,虚拟地址由一个 16 位的段参照和一个 32 位的偏移组成。段参照的最低 2 位与保护机构相关,高 14 位用于指定具体的段。一个进程可拥有的最大虚拟地址空间是 $2^{14+32}=2^{46}=64$ TB。

由分段部件 SU 将二维的分段虚拟地址转换成一维的 32 位线性地址。对于分段不分页模式,这也就是它的物理地址。分段不分页的好处是无需访问页目录表和页表,地址转换速度快,另外,对段提供的一些保护定义可以一直贯通到段的单个字节级。缺点是涉及大容量段的滚进滚出,耗时大,内存管理粗糙,有失灵活性。

(2) 分段分页模式

这是一种在分段基础上添加分页存储管理的模式。即将分段部件 SU 转换后的 32 位线性地址看成由页目录、页表和页内偏移 3 个字段组成,由分页部件 PU 完成两级页表的查找并将其转换成 32 位物理地址。此模式下一个进程可拥有的最大虚拟地址空间等同于分段不分页模式,也是 64 TB。这是一种兼顾分段分页两种优点的虚拟地址模式,受到 UNIX System V 和 OS/2 操作系统的‘偏爱’。

(3) 不分段分页模式

这个模式下分段部件 SU 不工作,只是分页部件 PU 工作。程序也不提供段选择符,以寄存器提供的 32 位地址,该地址被看成是由页目录、页表和页内偏移 3 个字段组成。由分页部件 PU 完成虚拟地址到物理地址的转换。进程所拥有的最大虚拟空间是 2^{32} = 4 GB,虽然虚拟空间减小了,但是对于一般系统而言也是够用的。这种纯分页的虚拟地址模式,也称为平展地址模式(Flat Address Mode)。它也能提供保护机制,而且将虚拟存储器看成是线性分页地址空间,比分段模式具有更大的灵活性。Windows NT, Windows 95 操作系统采用了这种模式来支持 32 位应用程序的运行。

3. 平展模式及其线性空间

下面解释平展模式的线性地址分配,如图 7 - 6 所示。

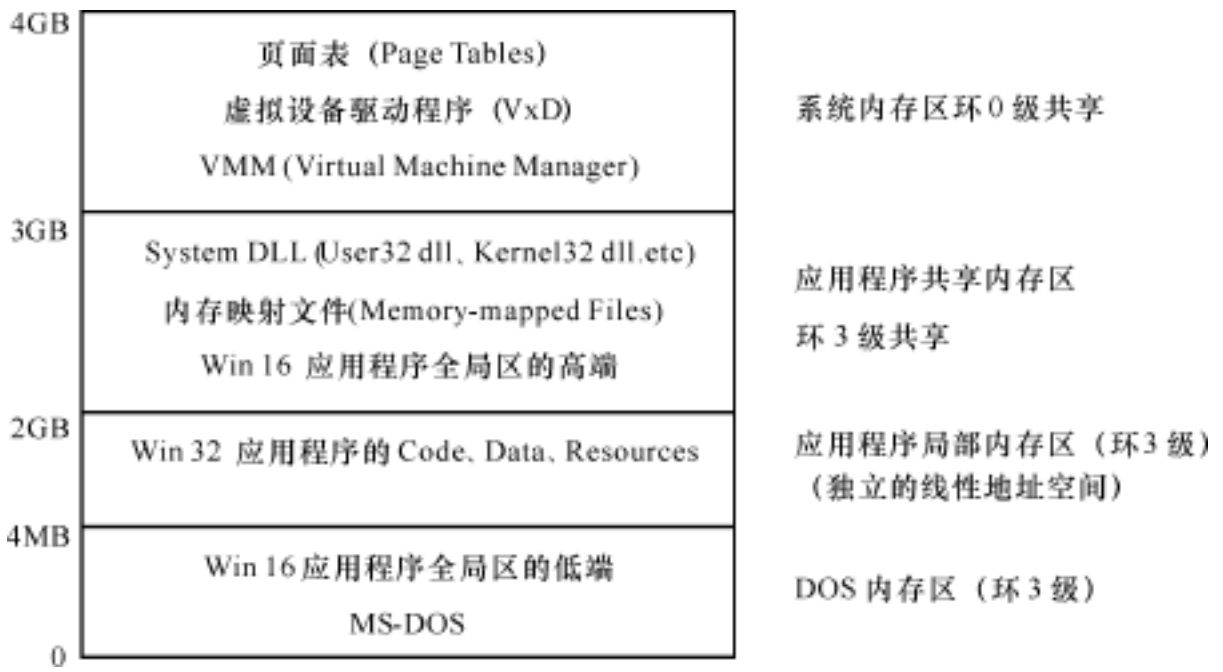


图 7 - 6　Windows 95 线性地址空间

(1) DOS 内存区(0 ~4 MB)

这部分在图上标的是 DOS 内存区。其实 DOS 只能利用到 0 MB ~1 MB 的内存空间,其余部分是空的。Windows 95 把 1 MB ~4 MB 的线性地址空间也当作 DOS 内存区,这样 Windows 95 在 DOS 虚拟机(VM)之间切换时,就能够以页目录项(Page Table Directory Entry)为单位来进行。这样虽说浪费了 3 MB 的地址,却换来了 DOS VM 切换的高效率。为

了使系统、Win16 应用程序能与 DOS 应用程序互相协作, 紧挨着 1 MB 的下面, 放了一个 Win16 全局区的低端部分。同时要注意图中还有一个 Win16 应用程序全局区的高端部分。其作用是为了高效率地切换 DOS VM。每一个 DOS VM 在大于 3 GB 的地址空间都有一个备份, Windows 95 在 DOS VM 之间进行切换时, 只是简单切换一下页目录的页表基地址即可。所以说, 如果一个 VxD 想访问某个 DOS VM, 没有必要一定要等到该 DOS VM 成为当前 VM 才能访问, 它可以直接去访问那个 DOS VM 的备份。这个 DOS VM 备份的地址称为 High - linear address。

(2) Win32 应用程序的代码(4 MB ~2 GB)

Win32 应用程序的代码、数据和资源都存放在这段内存中, 这部分内存, 对于每个 Win32 应用程序来说都是私有的, 这里是最能体现保护模式分页机制作用的地方。两个 Win32 应用程序, 对相同的线性地址(4 MB ~2 GB 范围内) 进行读写, 实际上, 它们是在对不同的物理地址进行读写。也就是说 Windows 95 通过操纵这 511(4 m 一段, 4 m ~2 GB 共分为 511 段) 个 Page Directory Entry 实现了 Win32 应用程序“独立”的线性地址空间。

(3) 共享内存区(2 GB ~3 GB)

这部分对应的内存空间称之为“应用程序共享内存区”, 这里存放着环 3 级应用程序需要共享的数据和代码。其中包括 Windows 95 系统的 DLL(如 User32. dll, Kernel32. dll 等)、内存映射文件、Win16 应用程序以及 DPMI 调用分配的内存。通过把要共享的数据映射到 2 GB ~3 GB 之间的线性内存空间, 可以实现 Win32 应用程序间的数据共享。Win16 应用程序是需要共享线性地址空间的, 这是 Windows 3. 1 的历史遗留问题。为了使以前运行于 Windows 3. 1 的 Win16 应用程序能在 Windows 95 下有同样的运行效果, Microsoft 决定把 Win16 应用程序放到这段共享内存区来运行。应该说这是比较合理的决策, 既省时省力, 又保证了对上一代产品的良好兼容性。再从分页机制的角度上来思考一下这段“共享内存区”是如何实现的。2 GB ~3 GB 的线性空间对应着 256 个 Page Directory Entry。无论谁在运行, Windows 95 都不会改变这 256 个 Page Directory Entry, 也就是说 2 GB ~3 GB 的线性地址空间对应的物理地址是一样的。这样, Windows 95 什么也不用做就实现了 2 GB ~3 GB 线性地址空间的内存共享。

(4) 系统内存区(3 GB ~4 GB)

这部分内存空间称为“系统内存区”。只有环 0 级的 VMM 和 VxD 可以访问这部分内存空间。这部分内存也是共享的。虽说环 3 级的应用程序无法直接共享这部分内存空间(也就是说 SDK 中讲述的方法无法做到), 但我们还是称之为共享内存区, 至少从分页机制的角度上说, 对应于这部分内存的 Page Directory Entry 是不变的。其实 Win32 应用程序还是有办法访问这部分内存空间的。一般作法是, 在 VxD 中分配一块内存, 然后把指向那块内存的 32 位地址指针传给 Win32 应用程序, 这样就可以在 Win32 应用程序中直接访问那块内存。前面在讲 0 ~4 MB 内存空间时, 提到 High - linear address, 即 DOS VM 备份的地方, 在 3 GB ~4 GB 内存空间中。

4. 虚拟 86 模式

这是一种在 32 位保护模式下支持 16 位实模式应用程序运行的特殊保护模式, 简称 V86 模式。自 80386 开始具备并一直延续到 Pentium。在这种模式下系统可建立多个 8086 虚拟机, 每个虚拟机都认为自己是惟一运行的机器, 安全地运行以实模式编写的 16 位应用

程序。这样在 32 位保护模式的操作系统管理下,系统可同时运行 32 位应用程序和 16 位应用程序。当然,这种“同时”是以 CPU 切换完成的。CPU 中 EFLAG 寄存器的 VM 位(b17 位)即为 V86 模式位,已经工作在保护模式下的 CPU,若 VM = 1,则 CPU 运行 V86 模式,否则运行一般保护模式。这种相互切换由任务转换或中断来完成。

V86 模式也是将段寄存器的内容乘以 16 作为段的基地址,再加上 16 位偏移量而得到访问存储器的线性地址,这与实模式形成物理地址的方式相同。但此时没有实模式的地址环绕现象,即允许 10FFEF 这样的线性地址出现。换句话说,V86 模式具有 1 MB + 64 KB 的线性存储空间。另外要注意,V86 模式是一种具有最低特权级(级 3)的保护模式。

5. 系统管理模式

Pentium 除了上述 3 种主要工作模式外,还从 80486 继承下来一种称为系统管理模式 SMM (System Management Mode)的新模式。通过软件检测到某种硬件条件时,可由其他模式进入 SMM。SMM 对其他模式总是隐藏的,从而允许处理器在 SMM 模式下以软件完成某种功能,而这些对应用软件甚至对操作系统都是隐藏的,最早引进这个模式是为了笔记本电脑的电源管理模式,在 CPU 没有实质工作进行时系统降低电源功耗,此时就是以 SMM 模式来保护现场。现在 SMM 又增添了新功能,能对实际不存在于系统中的硬件予以虚拟化。

保护方式下的 80x86 及相关的程序设计内容除特别说明外其寄存器、寻址方式和指令等基本概念与实模式下保持一致。

尽管实模式下 80386 以及后续的 80x86 的功能要大大超过其先前的处理器(8086/8088、80186、80286),但只有在保护方式下,80x86 才能真正发挥更大的作用。主要表现在以下几个方面。

在保护方式下,80x86 的全部 32 条地址线都有效,可寻址高达 4 GB 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和数据的安全和保密以及任务的隔离。

支持虚拟 8086 方式,便于执行 8086 程序。

7.3　32 位扩展指令

7.3.1　新增的寻址方式

80486/80586 指令在寻址时,逻辑地址(即偏移地址或有效地址)可由两种方法产生:

由基址 + (变址× 比例) + 位移量计算;

基址、变址、比例及位移量中部分项组合计算。

其中基址是任何通用寄存器的内容,包括 EAX, EBX, ECX, EDX, ESI, EDI, EBP 及 ESP,这些寄存器可以存放数据,也可以存放地址;变址是除 ESP 以外的任何通用寄存器的内容,通常用来访问阵列中的元素或字符串;比例是与变址寄存器相乘的因子 1、2、4 或 8;位移量

是 8 位、16 位、32 位的立即数。

在形成物理地址时应考虑相应的段寄存器。段寄存器在 8086 下存放的是形成物理地址的段地址，而在 80x86 下存放的是段选择符，段选择符在实模式下就是段址，在保护模式下是段描述符的索引号。

80486/80586 的寻址方式除了与 8086/8088 相同的寻址方式外，还增加了几种新的寻址方式。

(1) 比例变址寻址

汇编格式：X [EIR× 比例]

功能：逻辑地址 = X + (EIR) × 比例。

说明：X 为位移量，EIR 代表变址寄存器。

例如：MOV EAX, 6 [EDI× 8]

(2) 基址比例变址寻址

汇编格式：[EBR] [EIR× 比例]

功能：逻辑地址 = (KBR) + (E1R) × 比例。

说明：EBR 代表基址寄存器。

例如：INC [EBX] [ESI× 8]

(3) 带位移的基址比例变址寻址

汇编格式：X [XBR] [EIR× 比例]

功能：逻辑地址 = (EBR) + (EIR) × 比例 + X。

例如：MOV ECX, X[EBP] [ESI× 2]

7.3.2　常用 32 位扩展指令

80x86 的指令系统是向下兼容的，都是在 8086 指令系统的基础上扩充而来的，Intel 在 80386 以后新增了不少指令，下面介绍一下常用的 32 位扩展指令，包括这些指令在原指令基础上增强的功能。以下带* 号的指令为增加的新指令，未带* 号者为功能有增强的指令，其原功能仍然保持。

7.3.2.1　数据传送

1. 数据传送指令

格式：MOVSX/MOVZX opt1, opt2

功能：将 8 位或 16 位源操作数的值传送到一个寄存器中，这个寄存器要扩充到 16 位或 32 位。

说明：MOVSX 实现有符号数扩展传送，用源操作数的符号位填充高位，MOVZX 实现无符号数扩展传送(用 0 填充高位) 。

如果两操作数位数相等，则仅实现传送。

opt1 为 16 位或 32 位寄存器，opt2 为 8 位 16 位存储器或 8 位 16 位寄存器，opt2 的位数要小于 opt1 的位数(opt2 的位数为 16 位时，opt1 的位数也可为 16 位) 。

实例：将一个 8 位或 16 位寄存器/存储器的值传送到一个寄存器中，这个寄存器要用符号扩充到 16 位或 32 位。执行前 BH 的值为 8F(符号为 1) , CL 的值为 0F8H。

```
MOVSX     AX,  BH          ; AX = FF8F
MOVZX     DX, CL           ; DX = 00F8H
MOVSX     EBX, CL          ; EBX = 0FFFFFFF8H
MOVSX     EBX, DX          ; EBX = 0000F8H
MOVSX     AX, DX           ; AX = 00F8H (仅传送)
```

2．地址传送指令

(1) LEA 指令

格式: LEA opt1, opt2

功能: 将存储器的偏移地址送 32 位寄存器。

说明: opt1 为 32 位寄存器, opt2 为存储器。

(2) LDS 指令

格式: LDS opt1, opt2

功能: 48 位存储器→32 位寄存器, 48 位存储器 + 4→DS。

说明: 将 48 位存储器相继 6 个字节内容送到指定的 32 位寄存器和 DS。opt1 为 32 位寄存器, opt2 为 48 位存储器。

(3) LES 指令

格式: LES opt1, opt2

功能: 与 LDS 类似, 只是将段址送 ES。

说明: opt1 为 32 位寄存器, opt2 为 48 位存储器。

3．堆栈操作

(1) PUSH opt

功能: 将 l6 位或 32 位立即数压人堆栈, 非 16 位或 32 位数据将自动作相应扩展。

说明: opt 为 16 位立即数或 PUSH 32 位立即数。

(2) PUSHA

功能: 将通用寄存器 AX, CX, DX, BX, SP, BP, SI 及 DI 顺序压入堆栈。

(3) PUSHAD

功能: 将通用寄存器 EAX, ECX, EDX, EBX, ESP, EBP, ESI 及 EDI 顺序压入堆栈。

(4) POPA

功能: 将 SP 指示的内容顺序弹至通用寄存器 DI, SI, BP, SP, BX, DX, CX, AX 中。

(5) POPAD

功能: 将 ESP 指示内容顺序弹出至通用寄存器 EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX 中。

实例: 因为终端是随机发生的, 为了不破坏正在运行的程序, 在中断服务程序中, 一般都必须要保护中断程序要用到的寄存器(即保护现场)。

```
Int  andler proc far              ;定义一个过程
    Sti
    Pusha
    ...
    popa
    iret
```

Int Handler endp

7.3.2.2　算术运算

(1) XADD(交换加)

格式: XADD opt1, opt2

功能: opt1←opt1 + opt2, opt2←原 opt1。

说明: opt1 可以是存储器与寄存器, opt2 只能是寄存器。

例如: 若(AH) = 10H, (AL) = 01H, 则执行 XADD AH. AL 后(AH) = 11H, (AL) = 10H。

(2) CMPXCHG(比较传送)

格式: CMPXCHG opt1, opt2

功能: 由 opt1, opt2 置 ZF 位, 若 ZF = 1, 则 opt1←opt2; 若 ZF = 0, 则 opt1 送 AL/AX/EAX。

说明: opt1 可以是寄存器或存储器, opt2 只能是寄存器。

(3) IMUL 有符号数乘法

格式一: IMUL opt1, opt2

功能: 寄存器←寄存器× 立即数, 乘后的结果限制为 16 位的有符号数, 如果溢出, 则丢掉超过部分并置 CF = OF = 1。

说明: opt1 为位寄存器, opt2 为立即数。

例如: IMUL CX, 4 ; CX←CX × 4。

格式二: MUL opt1, opt2, opt3

或

IMUL opt1, opt2, opt3

功能: 寄存器←存储器× 立即数或寄存器←寄存器× 立即数。

说明: opt1 是寄存器, opt2 是存储器, opt3 是立即数。

例如: IMUL BX, BUF, 2; BX←BUF× 2。

(4) CWDE 与 CDQ 有符号数扩展指令

功能: CWDE 将 AX 中有符号数的符号扩展到 EAX 的高位部分。CDQ 将 EAX 中有符号数的符号扩展到 EDX。

7.3.2.3　移位指令

(1) SAL 算术左移

格式: SAL opt, COUNT

功能: 与 SAL opt、CL 相同。

说明: 当 COUNT = 1 时, 使用 SAL opt, 1 称为算术左移 1 位指令。

(2) SAR 算术右移

格式: SAR opt, COUNT

功能; 与 SAR opt, CL 相同。

说明: 当 COUNT = 1 时, 使用 SAR opt, 1 称为算术右移 1 位指令。

(3) SHL 逻辑左移

格式: SHL opt, COUNT

功能: 与 SHL opt, COUNT 相同。

说明: 当 COUNT = 1 时, 使用 SHL opt, 1 称为逻辑左移 1 位指令。

(4) SHR 逻辑右移

格式: SHR opt, COUNT

功能: 与 SHR opt, CL 相同。

说明: 当 COUNT = 1 时使用 SHR opt, 1 称为逻辑右移 1 位指令。

(5) RCL 进位循环左移

格式; ECL opt, COUNT

功能: opt 内容连同 CF 左移 COUNT 位。

说明: CF 有变化, OF 不确定, opt 可以是寄存器, 也可以是存储器; 当 COUNT = l 时, 称进位循环左移 1 位指令, 即使用 RCR opt, 1 格式。如果移位前后最高位不等, 则 OF = 1, 否则 OF = 0, CF、OF 均有变化。

(6) RCR 进位循环右移

格式: RCR opt, COUNT

功能: opt 连同 OF 右移 COUNT 位

说明: CF 有变化, OF 不确定; 当 COUNT = 1 时, 称进位循环右移 1 位指令, 即使用 RCR opt, 1。如果移位前后的最高 2 位不等, 则 OF = 1, 否则 OF = 0, 此时 CF, OF 有变化。

(7) SHLD 联合左移

格式: SHLD opt1, opt 2, COUNT

功能: CF 与 opt l 联合左移 COUNT 次, 右边移空的位用 opt 2 从左边开始顺序提供。

说明: AF 不确定, CF, OF, PF, SF, ZF 有变化。

(8) SHRD 联合右移

格式; SHRD opt l, opt 2, COUNT

功能: 将 opt 1 右移 COUNT 次, 左边空出的位用 opt 2 从右边顺序读出补充, 第二操作数不变。

说明: AF、OF 不确定, CF, PF, SF, ZF 有变化。

7.3.2.4　字符串操作

在此类操作中, 源串指针放在 SI 或 ESI 中, 目的串指针应放在 DI 或 EDI 中。对双字操作时, 指针增减 4。

(1) MOVSD 串传送

功能: 将 SI 或 ESI 所指源串中的双字传送到 DI 或 EDI 所指的目的串中, 此操作不影响标志位。

(2) LODSD 串装入

功能: 将 SI/ESI 所指源串中的一个双字元素送 EAX 中, 该指令对标志位无影响。

(3) STOSD 串存储

功能: 将 EAX 中的内容存入 DI/EDI 所指目的串中, 对标志位无影响。

(4) CMPSD 串比较

功能: 将 SI/ESI 所指源串中的双字元素的值与 DI/EDI 所指目的串中的元素进行比较, 对标志位的影响同 CMP。

(5) SCASD 串扫描

功能: 在目的串中扫描 EAX 中的内容, 对标志位的影响同 CMP。

格式: NSB

INSW

INS　opt, DX

实例: ES　　DI，DEST

MOV　　DX，301H

MOV　　ECX, 8

REP　　INSB

(6) INSD 串输入

功能: 从 DX 给出的端口地址读入一个字符或双字, 并送到 DI/EDI 所指向的目的串中, 对标志位无影响。

格式:　UTS　Dx, opt

OUTSB

OUTSW

实例:　EA　　ESI，SOURCE

MOV　　DX，300H

MOV　　ECX, 8

REP　　OUTSB

(7) OUTSD 串输出

功能: 将 SI/ESI 所指源串的一个字符/字/双字元素从 DX 寄存器给出端口地址中输出。该指令对标志位无影响。

7.3.2.5　转移指令

子程序的调用和返回是一种特殊的无条件转移指令。

(1) 段内直接调用

格式: CALL　32 位地址的近标号

功能: 对 32 位地址, ESP←ESP - 4, SS: ESP←EIP, EIP←OFFSET 近标号。

(2) 段内间接调用

格式: CALL OPD32

功能: 对 32 位的地址, ESP←ESP - 4, SS: ESP←EIP, EIP←32 位地址变量。

说明: OPD32 可以是 32 位的寄存器或存储器。

(3) 段间直接调用

格式: 对 32 位地址,　SP←ESP - 4, SS: ESP←CS;

ESP←ESP - 4, SS: ESP←EIP;

CS←SEG 远标号, EIP←OFFSET 远标号。

(4) 段间间接调用

格式: CALL 32 位的存储器

功能: 对 32 位地址,　SP←ESP - 4, ES: ESP←CS;

ESP←ESP - 4, SS: ESP←EIP;

CS←48 位的存储器 - 4, EIP←48 位的存储器。

(5) 段内返回

格式: RETN/RET 空/16 位立即数/32 位立即数

功能: 对 16 位操作数, IP←SS: SP, SP←SP + 2 或 SPf←SP + 2 + 16 位立即数。

对 32 位操作数, EIP←SS: ESP, ESP←ESP + 4 或 ESP←ESP + 4 + 32 位立即数。

(6) 段间返回

格式: RERF/RET 空/16 位立即数/32 位立即数

功能: 对 16 位操作数, P←SS: SP, SP←SP + 2 或 SP←SP + 2 + 16 位立即数;

CS←SS: SP, SP←SP + 2 或 SP←SP + 2 + 16 位立即数。

对 32 位操作数: IP←SS: ESP, ESP← ESP + 4 或 ESP←ESP + 4 + 32 位立即数;

CS←SS: ESP, ESP←ESP + 4 或 ESP←ESP + 4 + 32 位立即数。

注意: 对于 8086 系统条件转移指令来说, 其后操作数均为短标号, 而在 80486/80586 系统中, 在原条件转移指令后, 还可为近标号。

在原 8086 近间无条件转移指令 JMP 后跟 16 位的寄存器或 16 位存储器, 而在 80486/80586 系统中, JMP 后还可允许为 32 位寄存器或存储器。

在原 8086 远间无条件转移指令 JMP 跟 32 位存储器操作数在 80486/80586 系统中, 其后允许跟 48 位存储器操作数。

对上述功能增强的转移指令在此不作详细介绍。

7.3.2.6　移位指令

以下介绍的各指令其后均可以是 8 位的寄存器或 8 位的存储器操作数。

(1) SETA/SETNBE 高于设置

功能: 如果 CF = 0, ZF = 0, 即高于或不低于等于, 则地址变量 = 1, 否则地址变量为 0。

(2) SETAE/SETNB, 高于等于设置

功能: 如果 CF = 0, 即高于等于或不低于, 则地址变量 = 1、否则地址变量为 0。

(3) SET/SETNAE 低于设置

功能: 如果 CF = 1, 即低于或不高于等于, 则地址变量 = 1, 否则地址变量 = 0。

(4) SETBE/SETNA 低于等于

功能: 如果 CF = 1 或 ZF = 1, 即低于等于或不高于, 则地址变量 = 1, 否则地址变量为 0。

(5) SETNE/SETNZ 不等于设置

功能: 如果 ZF = 0, 即不等于或不为 0, 则地址变量 = 1, 否则地址变量 = 0。

(6) SETG/SETNLE 大于设置

功能: 如果 ZF = 0 或 SF = 0, 即大于或不小于等于, 则地址变量 = 1, 否则地址变量 = 0。

(7) SETGE/SETNL 大于等于设置

功能: 如果 SF = OF, 即大于等于或不小于, 则地址变量 = 1, 否则地址变量 = 0。

(8) SETL/SETNGE 小于设置

功能: 如果 SF ≠ OF, 即小于或不大于等于, 则地址变量 = 1, 否则地址变量 = 0。

(9) SETLE/SETGE 小于等于设置

功能: 如果 ZF = 1 或 SF < > OF, 即小于等于或不大于, 则地址变量 = 1, 否则地址变量 = 0。

(10) SETNO 无溢出设置

功能: 如果 OF = 0, 即无溢出, 则地址变量 = 1, 否则地址变量 = 0。

(11) SETNS 正号设置

功能: 如果 SF = 0, 即为正号, 则地址变量 = 1, 否则地址变量 = 0。

(12) SETO 溢出设置

功能: 如果 OF = 1, 即溢出, 则地址变量 = 1, 否则地址变量 = 0。

(13) SETP/SETPE 偶校验设置

功能: 如果 PF = 1, 即偶校验, 则地址变量 = 1, 否则地址 = 0。

(14) 负号设置

功能: 如果 SF = 1, 即负号, 则地址变量 = 1, 否则地址变量 = 0。

7.4　32 位程序设计

利用 32 位指令进行汇编语言程序设计的方法和前面章节中介绍的 16 位指令程序设计基本相同, 但有下面几点需要注意。

1. 指定汇编程序识别新指令

MASM 默认情况下只接受 8086 指令集, 要使用 80186 及其以后的微处理器指令, 必须使用处理器选择伪指令。例如, 处理器选择伪指令. 386 指明 MASM 接受 80386 指令(特权指令除外)。

2. 处理 16 位段和 32 位段

针对 32 位 80x86 CPU 编写 DOS 环境(实模式)的可执行程序, 尽管可以利用处理器的 32 位寄存器、32 位寻址方式, 但程序的逻辑段必须是 16 位段, 即最大 64 KB 的物理段。只有进入保护方式, 才可以使用 32 位段。采用. 386 及以上的处理器选择伪指令, 在简化段定义格式中, 应注意它的位置。当处理器选择伪指令在. model 语句之后时, 程序采用 16 位段模式; 当处理器选择伪指令在. model 语句之前时, 程序采用 32 位段模式。在完整段定义格式中, 段字属性 USE16 和 USE32 分别确定 16 位和 32 位段模式。

3. 16 位段和 32 位段中指令有差别

由于 16 位段和 32 位段的属性不同, 有些指令在 16 位段和 32 位段的操作会有差别。例如, 串操作指令在 16 位段采用 SI/DI 指示地址, CX 表达个数; 而在 32 位段采用 ESI/EDI 指示地址, ECX 表达个数。另外, 由于处理 32 位数据, 变量定义伪指令常采用 DWORD 属性。

7.4.1　32 位汇编开发环境

利用汇编语言编写的 Windows 源程序要经过汇编、链接才能生成可执行文件。当前使用的此类软件开发包主要是 Steve Hutchesson 提供的一个免费软件包。该软件包中包括编辑器、MASM 汇编程序和链接程序, 还有相当完整的 Win32 包含文件、库文件以及教程和事例等。

目前 Steve Hutchesson 提供的 MASM 开发环境已经到了 8.2 版, 此处以 8.2 版(m32v82r. zip, 下载地址: http://www. movsd. com) 为例介绍 MASM32 开发环境。另外, Visual C++ 也中包含 MASM, 并且该 MASM 会随着 VC 的 Service Pack 一同升级。为了支持最新的指令集(例如 Pentium 4), 除了 Service Pack 5 之外, 还应安装 Processor Pack。同时

Microsoft 提供了 MASM 的下载(http: / /www. microsoft. com/ddk /ddk98. asp) 。

利用 MASM32 编制 32 位汇编程序时有以下两种工作方式。

1. MASM32 编辑器

MASM32 的编辑器是 qeditor. exe 文件(MASM32 \目录下) , 除用于编辑源程序外, 也可以汇编、链接, 还可以直接创建(Build) 可执行文件。所有包含文件在 MASM32 \Include 目录下, MASM32 自身的库文件在 MASM32 \LIB 目录下。

汇编程序在 qeditor 中编辑完成并保存后, 可利用“ 工程(Project) ”菜单下的“ 汇编和链接(Assemble&Link) ”命令生成可执行文件。

2. 命令行方式

在命令行方式下, 可以使用任意编辑器编写汇编程序, 编写完成后将其存储为文本文件。然后利用 ml. exe 和 link. exe 对其进行汇编和链接, 这两个文件在 MASM32 \bin 目录下。

汇编命令的一般格式为:

```
ml /c /coff filename. asm
```

其中参数/c 告诉 MASM 只汇编不链接。参数/coff 告诉 MASM 生成 coff 格式的目标代码。当汇编成功后会产生 filename. obj 文件。

链接命令是:

```
link /SUBSYSTEM: WINDOWS /LIBPATH: \MASM32 \LIB filename. obj
```

其中参数/SUBSYSTEM: WINDOWS 告诉链接程序可执行文件的运行平台; 参数/LIBPATH: \MASM32 \LIB 告诉链接程序引入库文件的路径。链接程序库文件向目标文件中加入重定位信息, 最后生成可执行文件。为了不在每次链接时输入太多的信息, 可在 DOS 提示符下键入如下命令行:

```
Set Lib = d: \masm32 \lib
```

这样“ 链接 ”可简单写成:

```
link /SUBSYSTEM: WINDOWS filename. obj
```

3. ml 命令选项介绍

ml 命令是 MASM32 的汇编命令, 其完整格式为:

```
ML [ /options ] filelist [ /link linkoptions ]
```

“ /options ”参数有:

/AT	允许微小模式(即生成. COM 文件)
/Bl < linker >	使用预备链接
/c	只汇编不链接
/Cp	保护用户定义的符号, 主要起到区分大小写的作用
/Cu	将所有定义的符号转换为大写字母
/Cx	保护公共及外部符号
/coff	生成 coff 格式的目标代码
/D < name > [= text]	定义文本宏
/EP	由标准输出设备输出预处理列表
/F < hex >	设置堆栈大小(单位: 字节)
/Fe < file >	指定可执行文件名

/Fl[file]	产生汇编列表信息到指定文件
/Fm[file]	产生映像图到指定文件
/Fo < file >	指定目标文件名
/Fpi	生成 80x87 仿真代码
/Fr[file]	产生受限浏览信息到指定文件
/FR[file]	产生完整浏览信息
/G < c | d | z >	使用 Pascal, C, 或 Stdcall 调用
/H < number >	设置最大外部名长度
/I < name >	添加库文件路径
/link < linker options and libraries >	汇编后同时进行链接
/nologo	屏蔽著作权信息
/Sa	最大化源码表
/Sc	在列表中产生时间信息
/Sf	产生第一次通过的列表
/Sl < width >	设置行宽
/Sn	禁止符号表列表
/Sp < length >	设置页长度
/Ss < string >	设置副标题
/St < string >	设置标题
/Sx	列举条件错误
/Ta < file >	汇编非. ASM 文件
/WX	将警告视为错误
/W < number >	设置警告级别
/w	与 /W0, /WX 相同
/X I	忽略库文件路径
/Zd	在 debug 信息中添加行号
/Zf	建立所有的公共符号
/Zi	添加符号化的 debug 信息
/Zm	与 MASM 5. 10 兼容
/Zp[n]	设置结构队列
/Zs	只进行语法检查

最后强调一点: ml 命令中“ /options ”区分大小写。

7.4.2 实模式下的编程

80x86 在实模式下是一个更快的 8086, 它不但可以进行 32 位操作、32 位寻址, 并且还可以使用 80x86 的扩展指令。不过, 由于是在实模下, 寻址的最大空间为 1 MB, 在一个段内, 段的最大长度不超过 64 KB, 否则就会发生异常。

1. 实模式下的段定义格式

80x86 下的段定义格式与 8086 的段定义格式非常类似, 仅仅是在 8086 的段完整格式后

面添加了属性类型。

在 8086 下定义一个段的完整格式是:

段名 [定位类型] [组合类型] [　类别　]

80x86 下定义一个段的完整格式是:

段名 [定位类型] [组合类型] [　类别　] [属性类型]

说明: 在 80x86 下属性类型有两种: USE32 和 USE16, USE32 表示 32 位段, USE16 表示 16 位段。如果程序中用到伪指令.386, 那么默认的属性类型就是 USE32(32 位段), 如果没有用伪指令指定 CPU 的类型, 那么默认的属性类型就是 USE16, 在实模式下只能使用 16 位段, 即用 USE16。

例如:
```
CSEG   PARA   PUBLIC   USE32   ;定义一个 32 位的段
AA     DW   ?
BB     DD   ?
CC     DB   ?
DD     DW   ?
EE     DW   0,0,0.…
CSEG   ENDS
```

由于在 80x86 中用到了 66H 操作前缀和 67H 地址前缀, 因此尽管在实模式下, 只要设定的 CPU 类型是 80386, 仍然可以进行 32 位操作, 可以进行 32 位寻址。66H, 67H 这两个前缀无需在程序中写出, 汇编程序会自动添加。只要在程序中对 32 位操作数进行访问或进行 32 位寻址, 那么系统就会加上操作数前缀 66H 和地址前缀 67H。相反, 如果在 32 位段中进行 16 位或 8 位的访问, 汇编程序中也会加上这两个前缀。

2. 编程实例

下面给出一个实例程序, 演示在 80x86 的实模式下编程的方法与技巧, 并与 8086 下的程序设计进行比较。

本实例实现的功能为: 用十进制、十六进制、二进制 3 种形式显示双字存储单元 F000:1234 中的内容。实例由主程序框架和 todec, tobin, tohex, toasc, newline, echo 六个过程构成。

```
------------------ MAIN PROC ------------
.386
code segment para public  code use16
  assume cs:code
begin:
  mov    ax,0f000h
  mov    fs,ax
  mov    eax,fs:[1234H]
  call   todec
  call   newline
  call   tohex
  mov    al, H
  call   echo
  call   newline
  call   tobin
  mov    al, B
  call   echo
```

```
    call    newline
    mov     ah,4ch
    int     21h
;定义 todec 子过程
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
;定义 tobin 子过程
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
```

```
;定义 tohex 子过程
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
;定义 toasc 子过程
toasc proc near
  and    al,0fh
  cmp    al, 0
  cmp    al, 9
  seta   dl
  movzx  dx,dl
  imul   dx,7
  add    al,dl
toasc1:ret
toasc endp
;定义 newline 子过程
newline proc near
  push   dx
  push   ax
  mov    dl,0dh
  mov    ah,2
  int    21
```

```
    mov     dl,0ah
    int     21
    pop     ax
    pop     dx
    ret
newline endp
;定义 echo 子过程
echo proc near
    push    ax
    push    dx
    mov     dl,al
    mov     ah,2
    int     21h
    pop     dx
    pop     ax
echo endp
```

3. 实例分析

先来看主程序框架，下面就是 MAIN PROC:

```
------------------- MAIN PROC --------------------------------
    .386            ;定义处理器的类型为 386 表示可以使用所有 80386 指令
code segment para public  code use16
  assume cs:code
begin:
  mov     ax,0f000h
  mov     fs,ax                 ;将 f000h 装入段寄存器 fs
  mov     eax,fs:[1234H]        ;将 1234H 内存单元中的双字送给寄存器 EAX
  call    todec                 ;调用子过程 todec
  call    newline               ;调用子过程 newline 进行回车换行
  mov     eax,fs:[1234h]
  call    tohex                 ;调用子过程 tohex
mov al, H
  call    echo                  ;显示字符 H
  call    newline;
  mov     eax,fs:[1234H]
  call    tobin                 ;调用子过程 tobin
  mov     al, B
  call    echo
  call    newline
  mov     ah,4ch
  int     21h
```

主程序中的内容很简单。和 8086 下惟一不同的是就是要用伪指令定义 CPU 的类型，并且段寄存器的定义多了一个属性类型 USE16，还有一点就是使用 80386 的指令进行 32 位操作，除此之外其他的和 8086 下的程序没有什么区别。

这里重点要分析下面几个过程：todec，tobin 和 tohex。

(1) 子过程 todec

这个子过程的主要功能是将 f000:1234 双字单元的内容用十进制显示，下面就来看每一行代码：

```
todec proc near
  pushad                    ;将 8 个 32 位通用寄存器全部入栈
  mov   ebx,10              ;10→ebx
  xor   cx,cx               ;cx 清 0
dec1:
  xor   edx,edx             ;edx 清 0
  div   ebx  ;eax 内容即 ffff:[1234]双字的内容除以 10,商放在 eax,余数放在 edx 中
  push  dx                  ;将 edx 的低 16 位 dx 入栈
  inc   cx                  ;cx +1→cx
  or    eax,eax             ;对 eax 进行或操作,判断 eax 是否为 0
  jnz   dec1
dec2:
  pop   ax                  ;将放在堆栈中的余数逐个弹出到 ax 中
  call  toasc               ;显示 ax 的内容
  call  echo
  loop  dec2                ;将所有的余数显示完毕
  popad                     ;8 个 32 位通用寄存器全部出栈
  ret
todec endp
```

分析：将一个数用十进制数来表示，要它对它进行除以 10 的运算，得到商和余数。再将商除以 10，如此循环直到商为 0 为止，在这个过程中得到的一系列的模(余数)就是十进制数系列。在主程序中，已经将 f000:1234 双字单元的内容放到 EAX 寄存器中，由于后来要用十六进制数和二进制数显示，所以 EAX 寄存器中的内容不允许改变，因此在子过程的一开始，要将 EAX 的内容先入栈，所以子过程的一开始就用 PUSHAD 将 8 个 32 位通用寄存器的内容全部入栈。在标号 dec1 下不断地进行除以 10 运算，将所得到的余数全部入栈，同时用 cx 进行计数。在标号 dec2 中，逐个弹出在标号 dec1 中得到的余数，然后分别将它们显示出来，这样就可以将该存储单元中的内容用十进数表示。

(2) 子过程 tohex

```
tohex proc near
  countb =8                 ;伪指令定义一局部变量 countb,其值为 8
  enter  countb,0           ;建立堆栈框架指令
  movzx  ebp,bp             ;对 bp 进行零扩展
  mov    ecx,countb         ;8→ecx
  mov    edx,eax            ;eax→edx
hex1:
  mov    al,dl
  and    al,0fh             ;取低 4 位
  mov    [ebp-countb+ecx-1],al
  ror    edx,4              ;对 edx 进行循环右移,每次移动 4 位
  loop   hex1
  mov    cx,countb
```

```
    xor     ebx,ebx                 ;ebx 清 0
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                           ;释放堆栈框架
    ret
tohex endp
```

分析: 该子过程的功能是将 f000:1234 双字单元的内容以 16 进制数显示出来, 首先来考虑一下将一个数以 16 进制数表示出来的算法, 事实上在汇编语言中操作数一直都是以十六进制表示的, 因此, 只需要将 32 位操作数以每半个字节(4 位)的内容显示出来即可, 循环 8 次就可以显示出 32 位的 EAX, 所以这里用 ror 指令来不断循环移位, 每次右移 4 位放到 dl 的低 4 位中。这 8 个半字节分别放在 ebp - 1 ~ebp - 8 的存储单元中。不过, 存储的顺序是由低位到高位, 如果这样显示将导致显示结果变反。标号 hex2, hex3 的主要功能是用来判断 f000:1234 双字单元的内容是否为 0, 如果为 0, 只需要将最后结果显示为一个 0 即可, 否则就显示出 8 位内容。

(3) 子过程 tobin

```
tobin proc near
    push    eax                 ;eax 入栈
    push    ecx                 ;ecx 入栈
    push    edx                 ;edx 入栈
    bsr     edx,eax             ;对 eax 进行扫描,并把第一个为 1 的位号送给 edx
    jnz     bin1                ;如果 eax 不为 0,就跳到 bin1 执行
    xor     dx,dx               ;如果 eax 为 0,就将 dx 清 0
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                 ;edx 出栈
    pop     ecx                 ;ecx 出栈
    pop     eax                 ;eax 出栈
    ret
tobin endp
```

分析: 将一个数用二进制数显示出来, 只需要用 ROL 指令就可实现。在标号 bin1 中判断 f000:1234 单元的内容是否为 0, 如果为 0, 那么只需要在屏幕上显示 0。否则, 就用 ROL 指令对源操作数移位 32 位, 从最高位 31 位到最低位 0 位逐一显示出来。

7.4.3　保护模式下的编程

保护模式与实模式编程有着显著的区别, 它是 80386 才具备的编程模式, 保护模式下能够支持多任务, 同时 80386 CPU 为系统进行快速任务切换和保护任务环境提供了硬件支持。80386 及其以后的处理器都支持该模式。下面具体介绍一下保护模式下的汇编语言程序设计。

1. 保护模式下的内存管理方式

在保护模式下能够提供对不同任务和同一个任务不同段的保护。在内存空间中程序的代码和数据是分段存储, 因此每个段都应该有各段的起始地址、段界限和自己的属性。这样计算机才能控制哪些操作能访问哪些段, 而哪些段不能访问。所以每个段就应该有段起始地址、段界限和段属性。每一个任务都由许多不同的段构成, 有些任务有共同的数据段或者代码段, 为了节约内存空间, 就需要将共同的代码段或者数据段共享, 使得多个任务都能访问共享数据。这就要求系统能够决定哪些段是共享的, 哪些段是私有的, 所以每个段都有一个特权级数。i386 将段分为 4 个级别: 0、1、2、3, 最常用的是 0 级, 它代表内核模式, 3 级代表用户模式, 1 级和 2 级通常是不用的。

因为可以同时运行多个任务, 每个任务又有很多段, 这样就给计算机对存取段的操作带来了麻烦, 也不利后面的分页机制, 因此用一个局部描述表(LDT) 来描述一个任务。一个局部描述表由许多段的描述符构成, 即所谓的段描述符。段描述符是一个占 8 个字节的存储空间, 它用来存储一个段的起始地址、段界限和段属性。LDT 表由许多段描述符构成, 每个段描述符指向它对应的段, 一个 LDT 可以构成一个段, 每个系统还存在一个全局描述表(GDT) , 它是由共享段的段描述符和许多指向局部描述表段的描述符构成的, 这样就可以把内存中的所有段抽象出来, 用几张表来表示, 如图 7 - 7 所示。

内存中段的查找通过段选择子来实现, 段选择子包含 16 位二进制信息, 低两位是用来表示特权级, 第 3 位用 TI 来表示, TI 用来确定是从 GDT 还是从 LDT 中查找所需要的段, TI = 0指示从全局描述表 GDT 中读取描述符, TI = 1 指示从局部描述表 LDT 中读取描述符。段选择子的高 13 位是描述符索引, 所谓描述符索引是指描述符在描述符表中的序号。通过这种方法就可以像使用目录查找书本中的内容一样利用段选择子来查找程序需要的段。

2. 段选择子管理

段选择子在保护模式程序设计中起着非常关键的作用, 段选择子的管理是通过段寄存器来完成的。

CPU 中包含有许多段寄存器, 每个段寄存器后面都有一个高速缓冲器, 当将段选择子

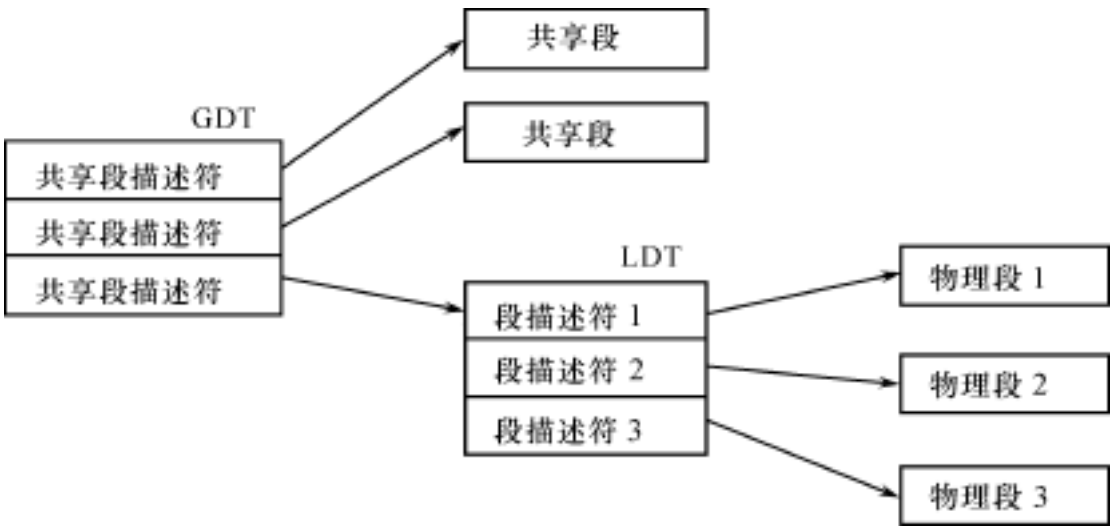


图 7 - 7　保护模式中内存段的意图

存入段寄存器时,系统自动将对应的段描述符存入高速缓冲器,然后系统根据描述符的信息去访问对应的段。这和实模式下的编程是完全不同的,在实模式下,段寄存器装的是段的地址,而在保护模式下,段寄存器中装的是段选择子。

(1) 系统地址寄存器

系统地址寄存器包括: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。全局描述表自己构成了一个段,我们用一个伪描述符描述它。它被装入 GDTR 中,通常用如下结构描述:

```
pdesc    struc
limit    dw   0
base     dd   0
pdesc    ends
```

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。LDTR 类似于段寄存器,由程序员可见的 16 位的寄存器和程序员不可见的高速缓冲寄存器组成。在初始化或任务切换过程中,把描述符对应任务 LDT 的描述符的选择子装入 LDTR,处理器根据装入 LDTR 可见部分的段选择子,从 GDT 中取出对应的描述符,并把 LDT 的基地址、界限和属性等信息保存到 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 的段选择子,所以装入到 LDTR 的选择子必须确定一个位于 GDT 中的类型为 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为 LDT。

(2) 控制寄存器

控制寄存器有 CR0, CR1, CR2, CR3,其中 CR0 跟分段和分页有重要的联系,CR1 是保留的,CR2 和 CR3 此处不做介绍。

控制寄存器的 CR0 的第一位用 PE 表示,它用来控制分段,当 PE = 0 时,处理器运行在实模式下,当 PE = 1 时,处理器运行在保护模式下。在代码中可以通过设置 CR0 的 PE 位,进行保护模式和实模式的切换。CR0 的 31 位,是用来控制分页的,用 PG 表示,当 PG = 0 时禁用分页机制,当 PG = 1 时启用分页机制。

7.4.4　程序实例

下面看一个例子，本实例的功能是实现实模式和保护模式的切换。当在实模式下输出一个 B 时，在保护模式下输出一个 A。

```
;-------------------------------------------------------------
;打开 A20 地址线
;-------------------------------------------------------------
ea20   macro
            push    ax
            in      al,92h
            or      al,00000010b
            out     92h,al
            pop     ax
       endm
;-------------------------------------------------------------
;关闭 A20 地址线
;-------------------------------------------------------------
da20   macro
            push    ax
            in      al,92h
            and     al,11111101b
            out     92h,al
            pop     ax
       endm
jump   macro  selector,offsetv
         db   0EAh
         dw   offsetv
         dw   selector
       endm
;---------------------
descriptor  struc                ;描述符的结构
  limitl    dw   0
  basel     dw   0
  basem     db   0
  attributes     dw   0
  baseh     db   0
descriptor ends
;---------------------
pdesc  struc
  limit     dw   0
  base      dd   0
pdesc ends
atdw=92h                                    ; 存在可读写数据段类型
atce=98h                                    ; 存在只执行代码段类型
atcer=9ah                                   ; 存在可执行可读写的代码段类型
.386P
dseg  segment  use16
```

```
;-----------------------------------------
gdt   label byte                                    ;全局描述符表
dummy descriptor <>
code    descriptor   <0FFFFh,,,atce,>               ;代码段描述符
code_sel =code - gdt                                ;选择子
data    descriptor   <0FFFFh,,,atdw,>               ;数据段描述符
data_sel =data - gdt                                ;对应的选择子
;注意:代码段有代码段的属性,数据段有数据段的属性,不要混淆,否则无法正常运行
vcode    descriptor     <0FFFFh,,,atce,>            ;显示字符 A 的代码段描述符
vcode_sel =vcode - gdt                              ;对应选择子
vbuf    descriptor     <0FFFFh,8000h,0bh,atdw,>
vbuf_sel =vbuf - gdt
normal  descriptor     <0FFFFh,,,atdw,>             ;描述符
normal_sel =normal - gdt                            ;选择子
;-----------------------------------------
gdtlen = $ - gdt
vgdtr  pdesc       <gdtlen - 1,>
dseg   ends
;-----------------------------------------
;代码段
vcseg  egment use16  vcode
       assume cs:vcseg
vstart:  ov       ax,0B800h                         ;直接写屏,输出 A
         mov      ds,ax
         mov      bx,0
         mov      al, A
         mov      ah,07h
         mov      [bx],ax
         jump     <code_sel >, <offset toreal >
vcseg ends
         end      vstart
cseg     segment use16  code                        ;实模式
assume   cs:cseg,ds:dseg
start:
         mov      ax,dseg
         mov      ds,ax
         mov      bx,16                             ;16 转化成32 位
         mul      bx
         add      ax,offset gdt
         adc      dx,0
    mov    word    ptr    vgdtr.base,ax
    mov    word    ptr    vgdtr.base +2,dx
    mov ax,cseg
    mul bx
    mov    word    ptr    code.basel,ax
    mov    byte    ptr    code.basem,dl
    mov    byte    ptr    code.baseh,dh
    mov ax,dseg
    mul bx
```

```
    mov     word    ptr     data.basel,ax
    mov     byte    ptr     data.basem,dl
    mov     byte    ptr     data.baseh,dh
    mov ax,vcseg
    mul bx
    mov     word    ptr     vcode.basel,ax
    mov     byte    ptr     vcode.basem,dl
    mov     byte    ptr     vcode.baseh,dh
    lgdt    qword ptr vgdtr                     ;将伪描述符导入到 GDTR
    cli
    ea20                                        ;打开 A20 时能够寻址 4GB 的空间
    mov     eax,cr0                             ;设置 CR0,进入保护模式
    or      eax,1                               ;将 CR0 设置为 1 进入保护模式
    mov   cr0,eax
    jump   <vcode_sel >, <offset vstart >       ;跳转到显示字符 A 的代码段
toreal:
    mov     ax,normal_sel
    mov     ds,ax
    mov     eax,cr0                             ;设置 CR0,进入保护实模式
    and     eax,0FFFFFFFEH                      ;将 CR0 设置为 0,返回到实模式
    mov     cr0,eax
    jump     <seg real >, <offset real >        ;跳转到显示字符 B 的代码段
real:
    da20                                        ;关闭 A20
    sti
    mov   ax,dseg                               ;显示字符 B
    mov   ds,ax
    mov   dl , B
    mov   ah,2
    int   21h
    mov   ah,4Ch
    int   21h
cseg ends
end start
```

本程序比较简单,在这里要注意实模式与保护模式的切换是通过对 CR0 的设置来实现的,当 CR0 为 1 时,程序进入保护模式,当 CR0 为 0 时,程序进入实模式。另外要注意,当程序切换到保护模式时,A20 一定要被打开,这样才能实现对 4 GB 空间的寻址,程序中打开和关闭 A20 是通过 ea20 和 da20 这两个宏来实现的。进入保护模式还需要将伪描述符导入到 GDTR 中,以便实现对多任务和每个任务的多段内存空间的管理。

习　　题

1. 80386 微处理器的内部组成可分为_____、_____和_____三大部分。采用流水线工作方式,形成_____、_____、_____、_____、_____、_____等 6 级流水段。

2. Pentium 微处理器共有_____个 32 位通用寄存器,用于保存数据和地址,它们命名为_____、_____、_____、_____、_____、_____、_____和_____。这些寄存器的低 16

位可作为 16 位寄存器单独使用，它们是_____、_____、_____、_____、_____、_____、_____和_____，使用时不影响高 16 位的值。AX, BX, CX, DX 四个寄存器的高、低 8 位可以单独使用来保存 8 位数据，它们是_____、_____、_____、_____、_____、_____、_____、_____，但它们不能用于有效地址计算。

3. 80x86 有 3 种工作方式，它们是：_____、_____和_____模式。

4. 段描述符是一个占_____个字节的存储空间，它用来存储一个段的_____、_____和_____。

5. 80486/80586 的寻址方式除了与 8086/8088 相同的寻址方式外，还增加了哪几种新的寻址方式？

6. POPA 和 POPAD 指令执行后，SP 和 ESP 的值为多少，为什么？

7. 利用 32 位指令编程实现将一个 64 位数据算术左移 8 位。

8. 简述保护模式下内存的管理方法。

9. 编写一个识别 Intel 80x86 系列微处理器的过程。

第 8 章　汇编语言与 C/C++ 混合编程

高级语言是面向用户和算法的语言，对于程序员来说，编写高级语言程序不必考虑具体的机器，其主要精力是放在算法描述和程序实现上，编程及调试效率高。而汇编语言则是一种面向机器的语言，它的特点是运行速度快，空间利用率高，直接控制硬件能力强，但编写及调试汇编语言程序相对高级语言程序来说要困难得多，程序员必须熟悉机器的内部结构及其指令系统。此外，不同系列机器的汇编程序不能通用，程序可移植性差。

高级语言编程方便简洁，汇编语言编程能充分发挥计算机硬件性能且程序运行效率高，因此，实际程序设计中常将两者结合使用，取长补短，发挥各自的优势。这种组合多种程序设计语言的编程方法就是混合编程。C/C++ 语言是一个被广泛使用的程序设计语言，它不仅具有良好的高级语言特征，而且还具有一些低级语言的特点，如寄存器变量、位操作等。所以，C/C++ 语言的程序与汇编语言程序之间能很平滑地衔接。

本章主要介绍汇编语言和 C/C++ 语言的混合编程。

8.1　混合编程方式

有两种方法可以实现汇编语言与 C/C++ 语言的混合程序设计。一种方法是在 C/C++ 语言中直接嵌入汇编语言语句，即嵌入式汇编。这种方法比较简洁直观，但功能较弱，此外，由于在其源程序中嵌入一段汇编语言程序段，降低了程序的可移植性。另一种方法是，两种语言分别编写独立的程序模块，分别产生目标代码 OBJ 文件，然后进行连接，形成一个完整的程序。这种方法使用灵活、功能强，但需要解决好混合编程中不同语言间的接口问题。

在混合编程方式中，最常用的做法是主程序模块用高级语言编制，子程序模块用汇编语言编写，它们统一使用高级语言程序中的堆栈，主程序模块和子模块间采用近调用或远调用实现程序转移，相应汇编语言程序设计成近过程或远过程且不设堆栈。模块间的参数传递多使用堆栈方式，有时也用寄存器返回运算结果。这种高级语言程序调用汇编语言子程序的问题，实质上就是一个不同语言的多模块间的程序设计问题。

8.2　C/C++ 的嵌入式汇编

嵌入式汇编又称行内(in - line) 汇编。它们允许在 C/C++ 源程序中直接插入汇编语言指令的语句。嵌入式汇编语言指令可以直接访问 C/C++ 语言程序中定义的常量、变量、函数，而不必考虑二者之间的接口。这样，可以避免汇编语言与 C/C++ 语言间复杂的接口问题，从而提高程序设计的效率。

8.2.1　在 C/C++程序嵌入汇编语句

1. 嵌入方式

在 VC 中,嵌入汇编语言指令是在汇编语句前加一个_asm 关键字(asm 前面是两个下划线),格式如下:

格式 1: _asm 操作码 操作数 <; 或换行 >

格式 2: _asm {
　　汇编指令 1
　　汇编指令 2
　　…
　　汇编指令 n
　}

例如,要嵌入 mov ax, 01h 和 int 10h 两条汇编语句,可以有如下 3 种方式:

方式 1:

```
_asm                                    // 成组嵌入
{
    mov ax, 01h
    int 10h
}
```

方式 2:

```
_asm mov ax, 01h                        // 逐条嵌入
_asm int 10h
```

方式 3:

```
_asm mov ax, 01h _asm int 10h           // 在一行中嵌入多条
```

在格式 1 中,内嵌的汇编语句可以用分号“;”结束,也可以用换行符结束;一行中可以写多个汇编语句,相互间用分号分隔,但不能跨行书写。注意:这里的分号“;”不是汇编语言中起注释作用的分号,而是作为语句的分隔符,若要对语句进行注释,应使用 C 语言的注释方法,如/* ……* /或//……。

若希望在 C 语言源程序中嵌入一条汇编语句,可按下列方式来做:

```
_asm mov ax, data
```

在格式 2 中,一次可以嵌入一组汇编指令,这种方式结构清晰,汇编指令与 C 语句相对独立,程序可读性好。尽管格式 1 也允许在一行内嵌入多条汇编指令,但其可读性较差且不能跨行。因此,在嵌入多条汇编指令时,通常采用格式 2。

若要嵌入一组汇编语句,通常采用如下方式:

```
_a m {                                  // 实现整型变量 x 和 y 之值的交换
   mov ax, x
   xchg ax, y
   mov x, ax
}
```

C/C++语言编译系统一般都提供了嵌入式汇编功能，如 TurboC, BorlandC++, Mirosoft C/C++, Visual C++等。下面给出两个在 VC 中嵌入汇编语句的实例。

【例 8.1】　在 C 语言程序中嵌入汇编语句，实现赋值 SUM = A + B + C。其中：A, B, C 及 SUM 均为 16 位整型变量。

```
#include <stdio.h>
#include <windows>
main()
{
    short a,b,c,sum;                    // 变量 A, B, C 及 SUM均为 16 位整型变量
    a=10;
    b=20;
    c=30;
    _asm                                // 实现 16 位整型变量相加 SUM= A+B+C
        mov   ax,a
        add   ax,b
        add   ax,c
        mov   sum,ax
    }
    printf("SUM=%d",sum);
}
```

上面的程序可在 VC 环境下直接编译、连接生成可执行文件。

值得注意的是，在 C/C++源程序中嵌入汇编语言语句时，一定要注意变量的数据类型及长度。在例 8.1 中，变量 a, b, c 均为 16 位整型变量，因此，其类型定义应为 short，若定义为 int，则在 VC 中默认为 32 位整型，如果仍然采用 16 位的 AX 寄存器，VC 在编译时会报类型冲突错误，此时，必须采用 32 位寄存器，下面是 32 位运算的例子。

【例 8.2】　在 C 语言程序中嵌入汇编语句，实现赋值 SUM = A + B + C。其中 A, B, C 及 SUM 均为 32 位整型变量。

```
#include <stdio.h>
#include <windows>
main()
{
    int a,b,c,sum;              // 变量 A, B, C 及 SUM均为 32 位整型变量
    a=10;
    b=20;
    c=30;
    _asm {                      // 实现 32 位整型变量相加 SUM= A+B+C
        mov         eax,a
        add         eax,b
        add         eax,c
        mov         sum,eax
        pop         eax
    }
    printf("SUM=%d",sum);
}
```

2. 约定和限制

嵌入式汇编代码可以使用汇编表达式,这个表达式是操作数和操作符的组合,产生一个数值或地址。

嵌入式汇编语言语句中,可以使用汇编语言格式表示整数常量(如 378h),也可以采用 C++的格式(如 0x378)。

嵌入式汇编语言语句中,注释可以使用 C 注释风格(如/* ……* /和//),也可以使用汇编语言的注释风格。

嵌入式汇编语句不能使用 C 的专用操作符,如 <<、++、--等。对两种语言都有的操作符,在汇编语句中作为汇编语言操作符,如* 、[ ]等。在 C 语言中,[ ]表示数组的某个元素,而在汇编语言中,[ ]表示字节偏移量,其意义是不一样的。例如:

```
int array[6];
//下面两条意义不同
array[6] = 0;                          //将0 存储在 array +24
_asmmov array[6],ebx                   //将 ebx 的值存储在 array +6
// 下面两条意义相同
array[6] = 0;                          //将0 存储在 array +24
_asmmov array[6* TYPE int],ebx         //将 ebx 的值存储在 array +24
```

在上面的例子中,TYPE 是一个运算符,用来获得变量或类型的字节数,由于 int 占 4 个字节,因此 TYPE int 的值为 4。

嵌入式汇编代码不支持汇编伪指令定义的数据(如 DB, DW, DD, DT, DF, DUP, THIS 等定义的操作数)。

在嵌入式汇编代码中,不能使用汇编的结构和记录(如 STRUCT, RECORD, WIDTH 及 MASK 等),也不能使用宏伪指令(如 MACRO, ENDM, REPEAT/FOR/FORC 等) 和宏操作符(如!、&、% 等)。

虽然嵌入式汇编不支持大部分汇编伪指令,但它支持 EVEN 和 ALIGN。这些指令将 NOP 指令放在汇编代码中以便对齐边界。对有些处理器来说,这样可以更有效地读取指令。

嵌入汇编引用段时只能通过寄存器而不能通过段名,段超越时,必须明确地用段寄存器来说明。例如:

```
MOV  AX, ES: [ EBX]
```

在用汇编语言编写的函数中,不必保存 EAX, EBX, ECX, EDX, ESl 和 EDI 寄存器,但必须保存函数中使用的其他寄存器(如 DS, SS, ESP, EBP 和整数标志寄存器),例如,用 STD 和 CLD 改变方向标志位,就必须保存标志寄存器的值。

嵌入式汇编支持 80x86 的全部指令系统。对于还不能支持的指令, VC 提供了_emit 伪指令用于机器指令(以字节为单位) 的扩展。

格式: _asm _emit < 机器码 >

若希望扩展一条操作码为 0xAE46 的机器指令,由于该指令为 2 字节指令,而_emit 伪指令一次只能定义一个字节的内容,因此,可以采用 C 程序中的宏进行定义,定义方式如下:

```
# define new_instruct _asm _emit 0xAE _asm _emit 0x46
```

宏调用方式如下:

```
_asm {new_instruct}
```

上面的宏调用展开后即为该指令的机器码。

8.2.2　在嵌入式汇编中访问 C/C++的数据

内嵌的汇编语句除可以使用指令允许的立即数、寄存器名外,还可以使用 C 程序中的任何符号,包括变量、常量、标号、寄存器变量、函数名、函数参数等。C 编译程序自动将它们转换成相应汇编语言指令的操作数,并在标识符名前加下划线。一般来说,只要汇编语句能够使用存储器操作数(地址操作数),就可以采用一个 C 语言程序中的符号;同样,只要汇编语句可以用寄存器作为合法的操作数,就可以使用一个寄存器变量。

1. 使用 C 程序的变量

在嵌入式汇编语句中,可以使用 C 程序中的变量或常量。

【例 8.3】　用嵌入式汇编语句访问 C 程序中的变量。

```
main()
{
    int a,b,sum;
    a =10;
    b =20;
    _a  m{
      mov eax,a              // 将变量 a,b 作为汇编程序的输入数据
      add eax,b
      mov sum,eax            // 将结果保存在 C 程序的 sum 变量中
    }
    printf("a +b =%d",sum);
}
```

在使用 C 程序的数据时,有时需要 C 数据类型及大小,以便与汇编指令匹配,例如两个 16 位的 C 数据进行相加,则汇编指令应采用 16 位寄存器;若是两个 32 位的 C 数据进行相加,则汇编指令应采用 32 位寄存器来完成。为此,在嵌入式汇编代码中,可以使用 LENGTH, SIZE, TYPE 操作符来获取 C 变量和类型的大小。LENGTH 用来返回数组元素的个数,对非数组变量返回值为 1; TYPE 返回 C 类型或变量的大小,如果变量是一个数组,它返回数组单个元素的大小。SIZE 返回 C 变量的大小,即 LENGTH 和 TYPE 的乘积。

【例 8.4】　获取 C 变量和类型的大小。

```
#include <stdio.h>
#include <windows.h>
main()
{
    int array[8];
    int a,b,c;
    _a  m{
      mov eax,LENGTH array
      mov ebx,TYPE array[0]
```

```
        mov ecx,SIZE array
        mov a,eax
        mov b,ebx
        mov c,ecx
    }
    printf("LENGTH =%d TYPE =%d SIZE =%d",a,b,c);
    getchar();
}
```

程序运行的结果为:

```
LENGTH =8 TYPE =4 SIZE =32
```

从上面例子可以看出,对于一个 C **语言中的整型数组** int array[8] **来说,数组共有** 8 **个** int **元素,由于** int **类型是** 32 **位,占** 4 **个字节,则有**:

LENGTH array **返回值为** 8(**相当于** C **的** sizeof(array) / sizeof(array[0]))。

TYPE array **返回值为** 4(**相当于** C **的** sizeof(array[0]))。

SIZE array **返回值为** 32(**相当于** C **的** sizeof(array))。

下面给出一个上述操作符的应用实例。

【例 8. 5】 **实现对** array **数组元素自动求和。**

```
main()
{
    int   array[ ] ={1,13,5,10, - 1,9,4,2, - 21,32};
    int   sum;
    _asm                                    // 实现对 array 数组元素求和
        mov   ecx,LENGTH array              // 用 LENGTH 操作符获取数组元素的个数
        mov   eax,0
        mov   esi,0
    next:
        add   eax,array[esi]
        add   esi,4
        loop  next
        mov   sum,eax
    }
    printf("sum=%d",sum);                   // 打印汇编代码的求和结果
    getchar();
}
```

本例利用 LENGTH **操作符来获取** array **数组元素的个数,并将其保存在作为循环计数的** ECX **寄存器中,从而实现对数组元素的自动求和。**

2. **标号的使用**

标号在两种语言中都用来标识程序语句的位置,汇编语句可以转到 C **程序的标号位置,**C **程序也能转到汇编程序的标号位置。需要特别注意的是:**C **程序区分大小写,而汇编程序则对大小写不加区分。下面是使用示例**:

```
void func( void )
{
    goto    C_Dest;        // 正确
```

```
    goto    c_dest;          // 错误
    goto    A_Dest;          // 正确
    goto    a_dest;          // 正确
    _asm
    {
       jmp C_Dest ; 正确
       jmp c_dest ; 错误
       jmp A_Dest ; 正确
       jmp a_dest ; 正确
       a_dest: ; 汇编标号
    }
    C_Dest:                  // C 标号
    return;
}
```

3. 结构成员的引用

嵌入式汇编语句可以方便地访问 C 语言结构中的某个成员，通常有两种引用方法。一种是通过变量名引用，即采用结构变量名加成员名的方法：

结构变量名.成员名

另一种是通过变量地址引用，即把结构变量的首地址送往某一地址寄存器，然后用该寄存器名(加方括号)再加成员名，中间用圆点隔开。

[地址寄存器].成员名

例如：

```
stru t score{
     int  a;
     int  b;
     int  c;
)s1,s2;
example()
{
   …
   asm  mov  eax,s1.a              // 取结构变量 s1 的成员 a
   asm  mov  eax,s2.c              // 取结构变量 s2 的成员 c
   asm  mov  ebx,offset sl         // 取结构变量 s1 的主存地址
   asm  mov  ecx,[ebx].b           // 取结构变量 s1 的成员 b
   …
}
```

如果类、结构、联合的成员名字惟一，可不说明变量或类型名就可以引用成员名，否则必须说明。例如：

```
struct first_type
{
   char  * weasel;
   int   same_name;
};
struct second_type
```

```
{
    int   wonton;
    long  same_name;
};
struct first_type hal;
struct second_type oat;
_asm
{
    mov  ebx,OFFSET hal
    mov  ecx,[ebx]hal.same_name    ; // 必须使用“hal”
    mov  esi,[ebx].weasel          ; // 可以省略“hal”
}
```

4. 使用 C 语言的宏指令

用 C/C++宏可以方便地将一段汇编代码插入到源程序中。C/C++宏将扩展成为一个逻辑行,所以书写具有嵌入汇编的 C/C++宏时,应遵循下列规则:将汇编程序段放在括号中,每一个汇编语言指令前必须有_asm 标志,应该使用 C 的多行注释风格(/* ……* /),不要使用单行注释(//)和汇编语言的分号注释方式。例如:

```
#define PORTIO_asm \
/* 端口输出 */ \
{ \
_asm mov dx,0xD007 \
_asm out al,dx \
}
```

宏展开为:

```
_asm /* 端口输出* / { _asm mov dx,0xD007 _asm out al, dx }
```

8.2.3　用汇编程序段编写 C 函数

利用嵌入式汇编语言可以方便地编写 C 程序中的函数,函数的定义和函数调用与 C 程序相同。与模块调用方式相比,这种方式更加灵活方便,因为这不需要独立的汇编程序,无需考虑复杂的接口问题,参数传递和从函数返回的值也非常简单。下面通过例子加以说明。

【例 8.6】　用嵌入式汇编语句编写 C 函数。

```
#include <stdio.h>
int power2(int num,int power);
void main( void )
{
    printf( "2 的 5 次方乘以 3 等于:%d\n",power2(3,5));// 显示函数返回值
    getchar();
}
int power2( int num,int power )
{
    _asm
    {
        mov  eax,num              ; 第一个参数 3 送 eax
```

```
        mov   ecx,power          ; 第二个参数5 送 ecx
        shl   eax,cl             ; eax 左移5 次,EAX = 3* 2^5
    }
    /*  寄存器 EAX 的内容作为函数的默认返回值 * /
}
```

该程序的运行结果如下:

```
2 的5 次方乘以3 等于:96
```

本例中没有用 return 语句给出明确的函数返回值,其值采用默认方式返回。函数返回值的约定是:对于小于等于 32 位的数据扩展为 32 位,存放在 EAX 寄存器中返回,4～8 字节的返回值存放在 EDX:EAX 寄存器对中返回;更大字节数据则将它们的地址指针存放在 EAX 中返回。

【例 8.7】　用嵌入汇编方式实现取两数较大值的函数 GetMax。

```
#include <stdio.h>
int GetMax(int x,int y)                    // 嵌入汇编实现求较大值
{
    int max;
    _asm{
            mov   eax,x                    // 第一个参数 x 送 eax
            cmp   eax,y                    // 比较第一个参数 x 与第二个参数 y
            jge   maxexit
            mov   eax,y
            mov   max,eax
    maxexit:
             mov max,eax                   // 将寄存器 eax 的内容赋值给变量 max
    }
    return(max);                           // 将变量 max 的内容作为函数的返回值
}
void main()
{
    printf("max = %d\n",GetMax(10,20));    // 显示比较结果
    getchar();
}
```

8.2.4　在嵌入式汇编中调用 C/C++函数

嵌入式汇编不仅可以编写 C/C++函数,还可以调用 C 函数和非重载的全局 C++函数,也可以调用任何用 extern“C”说明的函数,嵌入式汇编不支持对 C++成员函数的调用。

1. 参数传递

参数的正确传递是调用 C 函数的关键。汇编程序向 C 函数传递参数的方法是通过堆栈。由于 C 程序是按参数顺序的相反顺序压栈,所以在汇编程序中参数的压栈顺序要与 C 程序接收参数的顺序相反。若汇编程序希望把参数 a, b, c 依次传送给 C 函数的 3 个形式参数 x, y, z,则汇编程序的压栈顺序应为 c, b, a。在汇编语言程序调用 C 函数完成后,应该立即平衡堆栈,即清除堆栈里的参数,恢复堆栈到调用前的情形,通常可以利用一组 POP 指令来完成堆栈的平衡。下面是一个参数传递的示例:

【例 8.8】 汇编代码传递参数给 C 函数的压栈次序。

```
c_func(int x,int y,int z)      // C函数c_func的参数次序为x,y,z
{
  …
}
_asm{                          // 调用C函数c_func(a,b,c)
  …
  mov eax,c                    // 按c,b,a的次序压栈,将a,b,c依次传送给参数x,y,z
  push eax
  mov eax,b
  push eax
  mov eax,a
  push eax
  call c_func                  // 调用C函数c_func
  pop ebx                      // 平衡堆栈,恢复堆栈到调用前的情形
  pop ebx
  pop ebx
}
```

若参数传递采用传值方式,汇编语言程序就把参数值或已赋值的变量压入堆栈;若汇编语言程序向 C 函数以传址的方式传递数据,则应该把参数的地址压入堆栈。看下面的例子,比较它们的参数传递方式有何不同。

【例 8.9】 汇编代码按值和按址传递参数给 C 函数。

```
#include <stdio.h>
#include <string.h>
int x;
char st1[80];
void display(int x,char * s)        // 显示整数和字符串
{
   printf("%d\n",x);
   printf("%s\n",s);
   getchar();
}
void main()
{
   x=30;
   strcpy(st1,"hello");
   _asm{
     mov eax,offset st1             // 传送变量的地址
     push eax
     mov eax,x                      // 传送变量的值
     push eax
     call display                   // 调用C函数display
     pop ebx
     pop ebx
   }
}
```

2. 使用函数的返回值

若 C 函数需要向汇编语言程序送返回值，则 C 语言函数体必须用 return 返回。返回值的传递约定同前所述，即对于小于等于 32 位的数据扩展为 32 位，存放在 EAX 寄存器中返回，5～8 字节的返回值存放在 EDX: EAX 寄存器对中返回；更大字节数据则将它们的地址指针存放在 EAX 中返回。

【例 8.10】　调用带返回值的函数。

```
#include <stdio.h>
#include <string.h>
int getmax(int x,int y)                    // 比较两个整数的大小,返回较大的整数
{
   int max;
   if (x >= y)
     max =x;
   else
     max =y;
   return max;
}
void main()
{
   int x,y,m;
   x =20;
   y =50;
   _asm{
     mov    eax,y              // 传递参数
     push   eax
     mov    eax,x
     push   eax
     call   getmax             // 调用 C 函数 getmax,返回值默认保存在 EAX 寄存器中
     mov    m,eax
     pop    ebx
     pop    ebx
   }
   printf("max =%d",m);
   getchar();
}
```

3. 调用 C 的库函数

因为 C 的所有的标准头文件都采用 extern“C”说明库函数，并且已经通过头包含文件加入到 C/C++的源程序中，所以 C/C++程序中的嵌入式汇编也可以调用 C 的库函数。

【例 8.11】　调用 C 的库函数 printf，实现 printf("%s %s\n","Hello","China") 的功能。

程序如下：

```
// 汇编调用 printf 库函数的程序示例
#include <stdio.h>
char format[] = "%s %s\n";
```

```
char hello[] = "Hello";
char china[] = "China";
void main( void )
{
   _asm
   {
      mov    eax,offset china        // 利用堆栈为 printf 函数传递参数
      push   eax
      mov    eax,offset hello
      push   eax
      mov    eax,offset format
      push   eax
      call   printf                  // 调用 C 的库函数 printf
      pop    ebx                     // 清除压入的参数,恢复调用前的堆栈内容
      pop    ebx
      pop    ebx
   }
}
```

程序的运行结果如下:

```
Hello
China
```

在调用 C 的库函数时,一定要注意各种库函数的调用方式和参数类型,否则,会产生意想不到的结果。

8.3　用 C/C++ 调用汇编模块

模块连接方式是不同程序设计语言之间混合编程经常使用的方法。各种语言的程序分别编写,利用各自的开发环境编译形成目标文件(.obj)模块文件,然后将它们连接在一起,最终生成可执行文件(.exe)。但是,为了保证各种语言的模块文件的正确连接,必须对它们的接口、参数传递、返回值处理及寄存器的使用、变量的引用等做出约定,以保证连接程序能得到必要的信息。

8.3.1　接口约定

混合编程的关键是建立不同语言之间的接口及相互调用方式。即在不同格式的两种语言间提供有效的通信方式,做出符合两种语言调用约定的某种形式说明,实现两种语言间的程序模块互相调用、变量的相互传送以及参数和返回值的正确使用。下面以 VC 调用汇编子程序为例,介绍有关约定。

1. 名字约定

C 语言区分大小写,它的外部名可大小写混合使用,每个外部名隐含地使用一个下划线做前缀,因而外部名在汇编语言程序中要用下划线做前缀。例如,C 语言可用下列调用语句:

```
MyProc( int a, int b)
```

相应汇编过程的名字则为 _MyProc。汇编时,对此名字必须使用选项/MX(对

MASM5.0)或/CX(对 MASM6.0),以便 MASM 保持公用名字中的字母不转换为大写,而连接命令要使用选项/NOI,使 LINK 不忽略字母的大小写。

2. 调用规范

C/C++与汇编语言混合编程的参数传递通常利用堆栈,调用规范决定利用堆栈的方法和命名约定。VC 提供了 3 种调用规范:_cdecl,_stdcall 和_fastcall。其中_cdecl 是 VC 采用的默认调用规范。它在名字前自动加一个下划线,从右到左将实参压入堆栈,由调用程序进行堆栈的平衡。

Windows 图形用户界面过程和 APl 函数等采用_stdcall 调用规范,它在名字前自动加一个下划线,名字后跟@和表示参数所占字节数的十进制数值,从右到左将实参压入堆栈,由被调用程序平衡堆栈。

_fastcall 调用规范是在名字前、后都加一个@,后再跟表示参数所占字节数的十进制数值。首先利用寄存器 ECX,EDX 传递前两个双字参数,其他参数再通过堆栈传递(从右到左),由被调用程序平衡堆栈。在与其他语言进行混合编程时不要使用_fastcall 规范。

为了与 C/C++混合编程,汇编语言必须采用与 C/C++程序一致的调用规范。MASM 汇编语言利用“语言类型”确定调用规范和命名约定,支持的语言类型有:C,SYSCALL,STDCAL,PASCAL,BASIC 和 FORTRAN,通常我们采用 C 语言规范(即_cdecl 规范)。可用.model伪指令说明汇编程序所采用的调用规范。

例如:.model flat,c

说明汇编程序采用平展模式、C 语言规范。

3. 声明函数和变量

汇编程序中供外部调用的标识符应具有 PUBLIC 属性,例如:

```
PUBLIC    asmproc        ;说明汇编过程 asmproc 可供外部调用
.code
asmproc   proc           ;定义 asmproc 过程
…
ret                      ;返回,结果在 EAX中
asmproc   endp
```

为了 EXTERN 使 C/C++函数对汇编语言程序可见,汇编语言程序需要对所调用的 C 函数、变量用关键字 EXTERN 进行说明,形式如下:

EXTERN　被调用函数名

EXTERN　变量名:变量属性

变量属性可以是 byte,word,dword,qword 和 tbyte,它们的大小分别是 1、2、4、8、10 个字节。VC 与 MASM 数据类型的对应关系如表 8-1 所示。

表 8-1　VC 与 MASM 数据类型的对应关系

VC 数据类型	MASM 数据类型	VC 数据类型	MASM 数据类型	字节数
unsigned char	byte	char	sbyte	1
unsigned short	word	short	sword	2
unsigned long [int]	dword	long [int]	sdword	4

续表

VC 数据类型	MASM 数据类型	VC 数据类型	MASM 数据类型	字节数
float	real4			4
double	real8			8
long double	real10			10

不论何种整数类型,在进行参数传递时,都将扩展成 32 位。另外,32 位 VC 中没有近、远调用之分,所有调用都是 32 位的偏移地址,所有的地址参数也都是 32 位偏移地址,在堆栈中占 4 个字节。

在 C/C++ 程序中,采用 extern“C”{...}对所要调用的外部过程、函数、变量予以说明,说明形式如下:

extern“C”{返回值类型,调用规范,函数名(参数类型表);}

extern“C”{变量类型 变量名;}

例如,在 C/C++ 程序中有如下说明:

```
short   i,array[10];
char    ch;
int     result;
```

在汇编语言程序中,应说明为:

```
EXTERN i:word,arrray:word,ch:sbyte,result:dword
```

4. 参数及返回值约定

参数传递分传递值及传递指针两种情况。前者是把参数值直接入栈,后者是把参数的地址压入栈。在 C/C++ 语言中,不论采用何种调用规范,除了数组变量传递指针外,其他都是“传值”,参数“传址”应利用指针数据类型。当然还可用“&变量”表示变量的地址,用“*指针变量”表示值。

无论是 C/C++ 函数返回,还是汇编过程返回,其返回值的约定为:对于小于等于 32 位的数据扩展为 32 位,存放在 EAX 寄存器中返回;4 ~8 字节的返回值存放在 EDX:EAX 寄存器对中返回,更大字节数据则将它们的地址指针存放在 EAX 中返回。

8.3.2　调用汇编模块

在明确了接口关系和相关约定后,就可以编写与 VC 混合编程的汇编语言过程。程序员必须明确这是一个 32 位的编程环境。程序员可以采用全部 32 位 Intel 80x86 CPU 指令,但必须首先注意 32 位指令程序设计的问题,例如用.386p 等处理器伪指令说明采用的指令集。32 位编程环境下的寄存器是 32 位的,所以用汇编语言存取堆栈要使用 32 位寄存器 EBP 进行相对寻址,如 mov eax,[ebp + 8],而不能采用 BP。当然,Win32 环境下也能运行 16 位的汇编程序,用的 VC 开发 C/C++ 程序也能调用 16 位的汇编模块,但这样做显然不能充分发挥 32 位编程环境的优势。

对于 VC 的 32 位程序来说,没有存储模式的选择,因此,汇编语言简化段定义格式应该采用平展模式(flat),并且汇编时采用选项/coff。ML 命令行的选项/coff 使得产生的.obi 模块文件采用与 Win32 兼容的 COFF(Common Object File Format)格式。

调用步骤如下:

① 将汇编语言过程汇编为符合 COFF 的 obj 文件。

② 在 VC 编译环境下创建一个项目。

③ 将汇编过程的 obj 文件插入到 VC 的项目中。

④ 编译连接该项目,生成可执行文件。

【例 8.12】　VC 下调用汇编过程示例。

```
;汇编源程序文件名:hbgc.asm
;模块名 :power2
;入口参数:num:dword,power:dword
;模块功能:计算 num* 2^power,返回值放在 EAX 寄存器中
.386P
.model flat,c
public power2
.code
power2 proc num:dword,power:dword        ;定义过程 power2
       mov eax,num                       ; 第一个参数 num 送 eax
       mov ecx,power                     ; 第二个参数 power 送 ecx
       shl eax,cl                        ; eax 左移 5 次,EAX = num* 2^power
       ret                               ; 寄存器 EAX 的内容作为返回值
power2 endp
end
```

用以下命令对 hbgc.asm 进行汇编:

```
ML /c/coff hbgc.asm
```

得到名为 hbgc.obj 的目标文件。

```
// C++源程序文件名:cgc.cpp
// 使用外部过程 power2,汇编目标文件名为:hbgc.obj
#include <stdio.h>
extern "C" {int power2(int,int);}
void main( void )
{
    int num,power;
    num=3;
    power =5;
    printf("%d* 2^%d = %d\n",num,power,power2(num,5)); // 显示结果
    getchar();
}
```

VC 环境下具体的调用步骤如下:

① 在 VC 编译环境下创建一个项目 cgc,并插入 cgc.cpp 源文件。

② 将汇编过程的 hbgc.obj 目标文件插入到 VC 的项目中。

③ 对该项目编译连接,生成可执行文件 cgc.exe。

④ 执行 cgc.exe。

该程序的运行结果如下:

```
3* 2^5 =96
```

【例 8.13】 假设有 C++主程序 MSORT.CPP 实现从键盘输入任意多个 4 位十进制整数(先输入整数的个数),然后调用汇编过程对这些整数进行升序排列后返回主程序,最后再由主程序输出排序后的整数。

C++程序中定义一个数组 b[100]及一个整型变量 a,它们都作为参数传递给汇编过程,本例的参数传递通过堆栈来实现。

MSORT.CPP 的程序如下:

```
// C++源程序文件名:msort.cpp
// 使用外部过程 cprotc,汇编目标文件名为:sort.obj
#include <stdio.h>
extern "C" {int cprotc(int *,int );}
void main()
{
   int a,i,b[100];
   printf("please input a=\n");
   scanf("%d",&a);
   for(i=0;i<=a-1;i++)
   {
     scanf("%d",&b[i]);
   }
   cprotc(b,a);
   for(i=0;i<=a-1;i++)
   {
     printf("%d ",b[i]);
   }
   getchar();
}
```

汇编源程序如下:

```
;汇编源程序文件名:sort.asm
;模块名:cprotc
;入口参数:a:dword(值参),b:dword(地址参数)
;模块功能:对以 b 为始址,数组元素个数为 a 的整型数组按升序排序

.386P
.model flat,c
PUBLIC  cprotc
.code
cprotc  proc
la0:    push    ebp
        mov     ebp,esp
        sub     esp,4
        push    esi
        mov     ebx,[ebp+8]             ;取参数 b,得数组 b 的始址
        mov     ecx,[ebp+12]            ;取参数 a
sorta:  mov     esi,0
        mov     dword ptr[ebp-4],0      ;局部参数"0"
```

```
        dec     ecx                     ;元素个数减"1"为循环次数
        jcxz    finish
comp:   mov     eax,[ebx+esi]
        cmp     eax,[ebx+esi+4]         ;比较
        jle     incs                    ;小于等于,转 incs
        xchg    eax,[ebx+esi+4]         ;否则交换
        mov     [ebx+esi],eax
        mov     dword ptr[ebp-4],1      ;局部参数置"1"
incs:   add     esi,4
        loop    comp
        test    dword ptr[ebp-4],1      ;测试局部参数是否为"0"
        jz      finish                  ;是"0",转结束
        shr     esi,1                   ;除以 4 为下次元素个数
        shr     esi,1
        mov     ecx,esi
        jmp     sorta
finish: pop     esi
        mov     esp,ebp
        pop     ebp
        ret
cprotc endp
end
```

汇编过程的源文件为 SORT. ASM,采用冒包排序法,并使用 1 个局部变量[BP - 4] 作为交换标志,控制是否继续外层循环。运行的结果放在主程序提供的数组中,数组 b 的始址作为指针参数传到堆栈中,数组的元素个数 a 为值参,也传到堆栈中。VC 在调用外部过程时,将参数按自右至左的次序压入堆栈,最后再压入指令指针 EIP,即先压入参数 a 再压入参数 b,最后是 EIP,C 程序调用汇编过程后的堆栈情况如图 8 - 1(a) 所示。

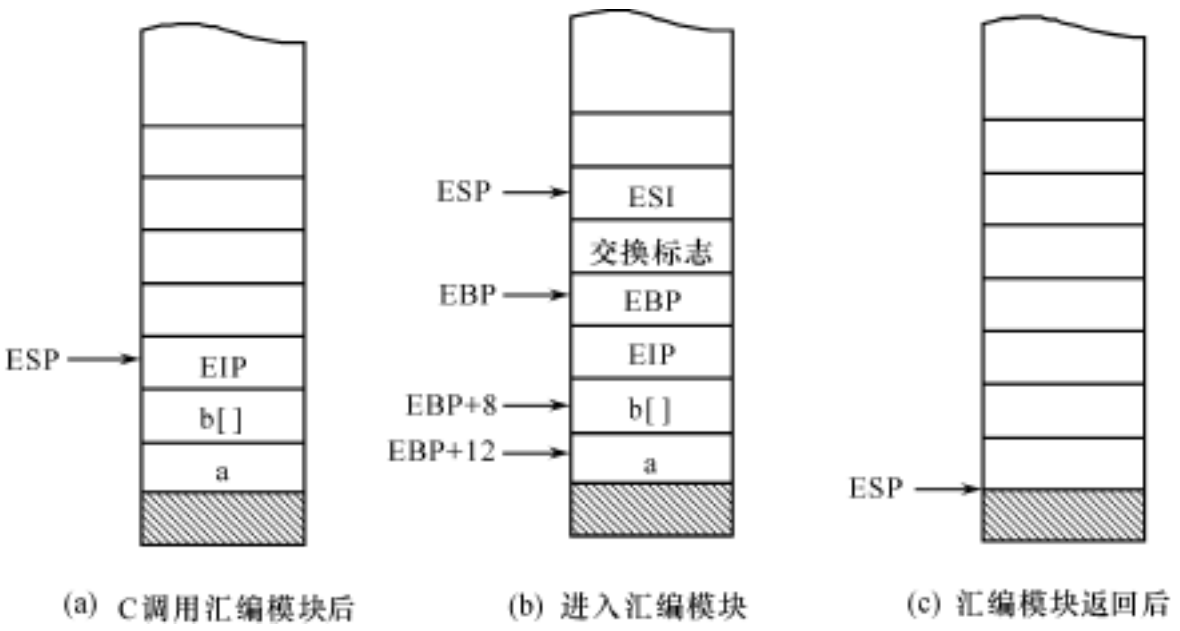


图 8 - 1　C 调用汇编模块 cprotc(b, a) 的堆栈情况

进入汇编过程后,汇编过程将 EBP 和 EIP 寄存器的内容压栈保存,并为交换标志预留 4 个字节,堆栈变化情况如图 8 - 1(b) 所示。显然,参数 a 的地址为[EBP +12] ,占 4 个字节,数组 b 的始址的存放地址为[EBP +8] ,占 4 个字节。交换标志的地址为[EBP +8] 。

汇编过程返回后，堆栈必须恢复，其中 EBP 和 EIP 寄存器通过 POP 指令恢复，而参数 a，b 及 EIP 则是通过 RET 指令自动恢复并返回 C 程序，此时堆栈恢复为调用前的情况，如图 8 - 1(c) 所示。

习　　题

1. 什么是混合编程，汇编语言与 C/C++ 语言在混合编程时应注意什么问题?

2. 汇编语言与 C/C++ 语言混合编程有哪两种方式，各自的特点是什么?

3. 说明在 VC 环境下嵌入汇编语言指令的方式。

4. 在嵌入汇编语言中能否访问 C/C++ 的数据? 说明在嵌入汇编语言中访问 C/C++ 数据的一般方法。

5. 说明嵌入汇编语言编写 C/C++ 函数的一般方法，并指出参数传递和函数值的返回方式。

6. 在 VC 环境下利用嵌入式汇编指令，完成对两个 C 变量的求和，结果由 C 程序显示。

7. 在 VC 程序中输入两个整数，然后调用汇编子程序对这两个数求和，结果在主程序中打印显示。

8. 在 VC 程序中编写一个嵌入式汇编排序函数，C 语言主程序提供待排序的数据并显示排序后的结果。

9. 用嵌入汇编指令编写一个字符转换函数，实现将 C 语言主程序中的一个字符串内的所有小写字母转换为大写字母。转换前后的字符串内容由 C 语言主程序打印显示。

10. 在嵌入式汇编指令中调用一个 C 函数。

11. 进行 32 位混合编程时，如何编写 VC++ 主程序和汇编语言过程?

12. 说明 VC 程序调用外部汇编模块的具体方法，并总结参数传递和汇编模块返回值的接口约定。分析 VC 程序调用汇编模块前后的堆栈变化情况。

13. 编写两个数求和的汇编模块，两个操作数通过参数名接收，在 VC 程序中输入两个整数，调用汇编模块求和，结果在主程序中打印显示。

14. 编写两个数求和的汇编模块，两个操作数通过堆栈传递，在 VC 程序中输入两个整数，调用汇编模块结果求和，结果在主程序中打印显示。画图说明调用汇编模块前后的堆栈变化情况。

15. 编写一个汇编排序程序模块，该模块可被 VC 调用，然后编写一个 VC 主程序，提供待排序的数据并显示排序后的结果。画图说明调用汇编模块前后的堆栈变化情况。

第 9 章　Win32 程序设计

开发基于 Win32 平台的应用程序, 程序员主要利用简单实用、功能强大的多种可视化开发环境, 例如 Visual Basic, Visual C++等, 同时汇编语言配合 API 提供了另一种方法。借助 API 编写 32 位 Windows 程序, 既可以充分利用 Windows 的高级特性, 又可以得到短小精悍的可执行文件, 同时不需要其他外部库文件(Windows 本身的动态链接库除外) 。

利用汇编语言, 可以开发出更高性能的可执行文件、动态链接库 DLL, 也有利于我们从更深层次理解 Windows 运行机制与程序设计思想。

9.1　汇编语言 Win32 程序简介

Win32 程序运行在保护模式下, 保护模式的历史可以追溯到 80286。Windows 系统把每一个 Win32 应用程序放到分开的虚拟地址空间中去运行, 也就是说每一个应用程序都各自拥有其相互独立的 4 GB 地址空间, 当然这并不是说它们都拥有 4 GB 的物理地址空间, 而只是说能够在 4 GB 的范围内寻址。操作系统将会在应用程序运行时完成 4 GB 的虚拟地址和物理内存地址间的转换。这就要求编写应用程序时必须遵守 Windows 的规范, 否则极易引起内存的保护模式错误。而过去的 Win16 内存模式下, 所有的应用程序都运行于同一个 4 GB地址空间, 它们可以彼此“看”到其他程序的内容, 这极易导致一个应用程序破坏另一个应用程序甚至操作系统的数据或代码。

和 16 位 Windows 下的把代码分成 DATA, CODE 等段的内存模式不同, Win32 只有一种内存模式, 即 FLAT 模式, 意思是“平坦”的内存模式, 再没有 64 KB 的段大小限制, 所有的 Win32 应用程序运行在一个连续、平坦、巨大的 4 GB 空间中。这同时也意味着无须和段寄存器打交道, 就可以用任意的段寄存器寻址任意的地址空间, 这对于程序设计来说是非常方便的, 这也使得用 32 位汇编语言和用 C 语言一样方便。在 Win32 下编程, 有许多重要的规则需要遵守。有一条很重要的是: Windows 在内部频繁使用 ESI, EDI, EBP, EBX 寄存器, 而且并不去检测这些寄存器的值是否被更改, 这样当程序要使用这些寄存器时必须先保存它们的值, 使用完后再恢复它们。

9.1.1　汇编语言 Win32 程序框架

编写 Win32 应用程序时必须遵守 Windows 的规范, 其中重要的一点就是必须按照指定的框架进行编写。

Win32 应用程序框架如下:

```
.386
.MODEL Flat, STDCALL
include          windows.inc
include          kernel32.inc
```

```
includelib          kernel32. lib
. DATA
 < Your   initialized data >
   …
. DATA?
    < Your uninitialized data >
    …
. CONST
    < Your constants >
    …
. CODE
    < label >
       < Your code >
      …
end  < label >
```

框架中各部分的含义如下:

(1) **处理器选择伪指令**. 386

这是一个汇编语言伪指令, 它告诉编译器程序使用 80386 **指令集编写。还可以使用** . 486、. 586 **等处理器选择伪指令。对于每一种** CPU **有两套几乎功能相同的伪指令**: . 386 / . 386P、. 486 / . 486P、. 586 / . 586P。**带** P **的指令表明程序中可以用特权级指令。特权级指令是保留给操作系统的, 如虚拟设备驱动程序。在大多数时间, 程序都无须运行在** RING0 **层, 故用不带后缀** P **的伪指令已足够。**

(2) **存储模式伪指令**. MODEL FLAT、STDCALL

. MODEL **是用来指定内存模式的伪指令, 在** Win32 **下, 只有一种内存模型, 那就是** FLAT。STDCALL **告诉编译器参数的传递约定。参数的传递约定是指参数传递时的顺序(从左到右或从右到左) 和由谁恢复堆栈指针(调用者或被调用者)。在** Win16 **下有两种约定**: C **和** PASCAL, C **约定规定参数传递顺序是从右到左, 即最右边的参数最先压栈, 由调用者恢复堆栈指针。**

例如, 调用函数 functionC (int first_param, int second_param, int third_param) ; **按** C **约定的汇编代码应该是这样的**:

```
push   [third_param]
push   [second_param]
push   [first_param]
call   functionC
add   esp,3 * 4                    ;调用者自己恢复堆栈指针
```

PASCAL **约定和** C **约定正好相反, 它规定参数从左向右传递。**

Win16 **采用了** PASCAL **约定, 因为** PASCAL **约定产生的代码量要小。当不知道参数的个数时**, C **约定特别有用, 如在函数** wsprintf ( ) **中**, wsprintf **预先并不知道要传递几个参数, 所以它不知道如何恢复堆栈。**STDCALL **是** C **约定和** PASCAL **约定的混合体, 它规定参数的传递是从右到左, 恢复堆栈的工作交由被调用者。**Win32 **只用** STDCALL **约定, 但除了一个**

特例, 即 wsprintf。

(3) 包含伪指令 include 语句

include 语句包含了一些系统的定义和 API 函说明, 其中所有的 Windows 数据结构定义和常量定义都包含在 windows. inc 中, 例如 Win32 程序中常用的两个常量 NULL 和 MB_OK 在 windows. inc 中定义如下:

```
NULL        equ        0
MB_OK       equ        0
```

windows. inc 中包含了所有 Windows 数据结构的描述, 而 Windows API 在 kernel32. inc, user32. inc, gdi32. inc 等文件中进行声明。大多数常用 API 函数都存在于 kernel32. dll, user32. dll 和 gdi32. dll 三个大的系统动态链接库 DLL 文件中, kernel32. dll 中的函数主要处理内存管理和进度调度, user32. dll 中的函数主要控制用户界面, gdi32. dll 中的函数则负责图形方面的操作。对应它们的 3 个包含文件是: kernel32. inc 包含文件声明操作系统核心函数, user32. inc 包含文件声明用户接口和其他函数, gdi32. inc 包含文件声明图形有关的函数。它们各自的库文件依次是 kernel32. lib, user32. lib, gdi32. lib。

如需查找函数处于哪个库文件中, 可查看 Microsoft Win32 Programmer s Reference。例如, 查找下面实例程序中用到的 ExitProcess 函数的位置, 通过查看 Microsoft Win32 Programmer s Reference 可知道 ExitProcess 包含在 kernel32. dll 中。此时, 就要在程序中包括 include kernel32. inc 和 includelib kernel32. lib 语句, 否则在编译时会出现 API 函数未定义的错误。而实例程序中用到的 MessageBox 函数在 user32. dll 中, 那么只有在程序中包括 include user32. inc 和 includelib user32. lib 语句时才能保证程序的正常运行。

(4) 数据段和代码段定义伪指令

. DATA, . DATA?, . CONST 和. CODE 四个伪指令是“ 分段 ”(SECTION) 伪指令。前面介绍过 Win32 下没有“ 段 ”(SEGMENT) 的概念, 但是可以把程序分成不同的“ 分段 ”, 一个“ 分段 ”的开始即是上一个“ 分段 ”的结束。WIN32 中只有两种性质的“ 分段 ”: DATA 和 CODE。

其中 DATA“ 分段 ”又分为 3 种:

. DATA 中包括已初始化的数据。

. DATA? 中包括未初始化的数据。比如有时程序只想预先分配一些内存但并不想指定初始值。使用未初始化的数据的优点是它不占据可执行文件的大小, 如: 若要在 . DATA?段中分配 10 000 字节的空间, 可执行文件的大小无须增加 10 000 字节, 而仅仅是要告诉编译器在装载可执行文件时分配所需字节。

. CONST 中包括常量定义。这些常量在程序运行过程中是不能更改的。

应用程序并不需要以上所有的 3 个“ 分段 ”, 而是根据需要进行定义, . CODE 是代码“ 分段 ”。

实际上, 这里所说的分段并不是像在 DOS 中那样为不同的段分别指出不同的段寄存器, 因为 Windows 下只有一个 4 GB 的段, Windows 程序中的分段表现在当程序装载时, 赋予不同的分段不同的属性, 比如说当程序加载时, 对于 Ring3 程序来说, . code 段是不可写的, 而. data 段是可写的, 如果像在 DOS 下一样编写代码部分, 将会得到一个蓝屏错误。

(5) 代码范围标签

< label >

```
...
end  <label>
```

是用来惟一标识代码范围的标签,两个标签必须相同,应用程序的所有可执行代码必须在两个标签之间。end <label>表示程序汇编到此结束。

9.1.2 简单 Win32 应用程序设计

下面看一个最简单的 Win32 汇编语言应用程序实例,程序运行将弹出一个消息框并显示 Welcome Win32!,标题为 MessageBox。

1. 程序源代码

【例 9.1】 依据 Win32 程序框架,编写这个程序。

```
.386
.model flat,stdcall
NULL   equ 0
MB_OK  equ 0
ExitProcess PROTO:DWORD
MessageBoxA PROTO:DWORD,:DWORD,:DWORD,:DWORD
includelib kernel32.lib
includelib user32.lib
.data
szText     db "Welcome Win32!",0
szCaption db "MessageBox",0
.code
start:
        push    MB_OK
        lea     eax,szCaption
        push    eax
        lea     eax,szText
        push    eax
        push    NULL
        call    MessageBoxA
        xor     eax,eax
        push    eax
        call    ExitProcess
        end     start
```

2. 汇编链接

对 Win32 程序进行汇编链接,汇编链接前首先要存储上述代码为一个文件,假设文件名为 mesgbox.asm。

汇编链接分下面两步进行:

```
ml /c /coff mesgbox.asm
link /subsystem:windows /libpath:c:\masm32\lib mesgbox.obj
```

第一步编译生成.obj 文件。

/c 表示只编译,不链接。

/coff 表示生成 COFF 格式的目标文件。

第二步链接生成. exe 文件。

/subsystem: windows 表示生成 windows 文件

/libpath: c: \masm32\lib 表示引入库的路径为: c: \masm32\lib。

在安装 Masm32 后,引入库位于 Masm32\Lib 目录下。

也可按上节中介绍的方法设置环境变量 Lib 的值使“链接”简单写成:

```
link/subsystem: windows mesgbox. obj
```

3. 程序执行

汇编链接完成后,当前目录下会出现一个名为 mesgbox. exe 的可执行文件,在 DOS 提示符下键入 mesgbox,回车,屏幕中央将出现一个标题为“MessageBox”,显示信息为“Welcome Win32!”的带一个“确定”按钮的消息框,如图 8 - 2 所示。

图 8 - 2　程序执行结果

4. 实例程序分析

实例程序的语句非常简单,仅仅设计到了 PUSH(压栈)、LEA(有效地址传送)、XOR(异或)和 CALL(子程序调用)4 条指令。消息框的生成是通过调用 API 函数实现的。下面着重介绍一下 Win32 程序中 API 函数的调用。

程序中有如下两条语句:

```
call MessageBoxA
call ExitProcess
```

以前的章节中已经学过,CALL 命令是子程序调用命令,但是程序中并没编写 MessageBoxA 和 ExitProcess 这两个子程序,事实上,这些是 API 函数。作为函数,在调用时需要为函数传送一些参数,传送的参数通过函数原型定义,如下所示:

```
ExitProcess PROTO : DWORD
MessageBoxA PROTO : DWORD, : DWORD, : DWORD, : DWORD
```

PROTO 是过程的原型说明伪指令,其作用与 PROC 伪指令类似,但是由 PROTO 说明的过程除了可用 CALL 调用外,还可以用 INVOKE 调用。和 CALL 相比,用 INVOKE 调用过程的优点是能自动将各参数入栈,而当从过程返回时被自动清除,并且在需要时,也能根据原型说明进行参数类型的转换。

可以看出,ExitProcess 有一个参数,MessageBoxA 有 4 个参数,这些参数都是 DWORD 类型。

在 Win32 中,参数的传递都是通过堆栈来完成的,堆栈的压栈顺序十分关键。程序的第二行. model flat, stdcall 中的 stdcall 指定参数是从右到左压入堆栈的,且调整堆栈是在子程序返回时完成的。所以 MessageBoxA 函数中的 4 个参数按照从右到左的顺序入栈。由于调整堆栈在子程序返回时完成,所以在源程序中不需要用“add sp,的移量”来保持堆栈平衡。对 MessageBox,在 API 手册中是这样定义的:

```
int MessageBox(
    HWND hWnd,              // handle of owner window
    LPCTSTR lpText,         // address of text in message box
    LPCTSTR lpCaption,      // address of title of message box
    UINT uType              // style of message box
    );
```

所以程序中会有如下语句:

```
push MB_OK
lea eax,szCaption
push eax
lea eax,szText
push eax
push NULL
call MessageBoxA
```

通过上面的程序不难想到,假如在写程序时,少往堆栈里压入一个数据,那将是一个致命的错误。这种检查参数个数是否匹配的工作可以交给计算机来完成,可以通过 INVOKE 指令由计算机完成压栈工作。假如程序中参数个数不正确,连接器将给出错误提示。所以,建议使用 INVOKE 代替 CALL 来调用子程序。

5. 实例程序的改进

1. 程序源代码

【例 9.2】 使用 INVOKE 指令由计算机完成压栈工作,程序可进行大幅度的简化。

```
.386                                    ;表示要用到386 指令
.model  flat,stdcall                    ;32 位程序,要用 flat
option  casemap:none                    ;区别大小写
include     windows.inc                 ;常量及结构定义
include     kernel32.inc                ;函数原型声明
include     user32.inc
includelib  kernel32.lib                ;用到的引入库
includelib  user32.lib
.data
    szText db " Welcome Win32!",0
    szCaption db " MessageBox",0
.code
start:
    invoke MessageBoxA,NULL,addr szText,addr szCaption,MB_OK
    invoke ExitProcess,NULL             ;程序退出
end start
```

2. 代码分析

API 函数中包含大量的常量和结构,如果在编制程序中对这些常量和结构进行定义,将对程序的编制增加极大的工作量,更不便于看程序的主要部分。MASM32 开发环境中的 windows.inc 包含了 Win32 编程所需要的常量和结构体的定义,程序中可简单地用一个 include 指令将这些常量和结构的定义插入到文件中:

```
include c:\masm32\include\windows.inc
```

这样一来 MessageBoxA 函数中的 NULL, addr szText, addr szCaption 和 MB_OK 参数就不需重新定义了。但要注意, windows.inc 文件中的语句是区分大小写的,为了保证程序编译的正常进行必须使用如下语句:

```
option casemap:none
```

该语句的功能是告诉 MASM 要区分符号的大小写，和汇编程序 ml.exe 的参数“/Cp”有相同的效果。

但是 windows.inc 中并不包含函数原型的声明，还要从其他的头文件中得到函数原型的声明，比如：MessageBoxA 的原型声明在 user32.inc 文件中，ExitProcess 在 kernel32.inc 文件中。这些头文件都放在 \masm32\include 文件夹下，并在程序中使用如下语句：

```
include     kernel32.inc
include     user32.inc
```

有了上述 3 条语句，就可使用 invoke 指令实现 API 函数的调用。语句如下：

```
invoke MessageBoxA,NULL,addr szText,addr szCaption,MB_OK
```

3．汇编链接

使用 invoke 指令实现 API 函数的调用时，由于使用了引用库，所以程序汇编时需要指明引入库的位置。汇编命令如下：（假设上述文件存储为 mesgbox1.asm）

```
ml /c /coff /I c:\masm32\include mesgbox1.asm
```

/I c:\masm32\include 表示 *.inc 文件的位置，也可设置环境变量 Set include = c:\masm32\include 来简化操作，也可在程序中明确指出 *.inc 的位置。上述语句可简化为：

```
ml /c /coff mesgbox1.asm
```

链接操作同上例。

9.2　资源文件的使用

在上一节改进的实例程序中，我们利用“include”命令引入了一些资源文件，实际上不管在 DOS 下编程还是在 Windows 下编程，总是要用到除了可执行文件外的很多其他数据，如声音数据、图形数据、文本等。在 DOS 下编程，可以自己定义这些文件的格式，但这样一来就造成了很多资源不能共享的问题。虽然在 Win32 编程中，仍然可以定义自己的资源格式，但 Win32 编程为我们提供了一种格式统一的资源文件，此时可以把字符串、图形、对话框包括上面的按钮、文本等信息定义到一个资源文件中。由于具有统一的格式，我们可以方便地在不同的文件中使用它。最重要的是，如果我们用自己设计的文件格式，使用时就要涉及这些文件的读写操作，比较复杂，但使用格式统一的资源文件时，Windows 提供了一系列的 API 来装入资源，使用起来非常方便。

9.2.1　资源文件的作用

Windows 为编写应用程序提供了大量的资源。其中最重要的是 Windows API（Application Programming Interface）。Windows API 是一组功能强大的函数，它们本身驻留在 Windows 中供人们随时调用。上一节中已经提到 API 函数的大部分被包含在几个动态链接库（DLL）中，例如 kernel32.dll，user32.dll 和 gdi32.dll。Kernel32.dll 中的函数主要处理内存管理和进程调度，user32.dll 中的函数主要控制用户界面，gdi32.dll 中的函数则负责图形方面的操作。除了上面主要的 3 个动态链接库，我们还可以调用包含在其他动态链接库中的函数。

1. 动态链接库的定义与加载方法

动态链接库,顾名思义,这些 API 的代码本身并不包含在 Windows 可执行文件中,而是当程序要使用它们时才被加载。为了让应用程序在运行时能找到这些函数,就必须事先把有关的重定位信息嵌入到应用程序的可执行文件中。这些信息存在于导入库中,由链接器把相关信息从导入库中找出插入到可执行文件中。程序中必须指定正确的导入库,因为只有正确的导入库才会有正确的重定位信息。

当应用程序被加载时 Windows 会检查这些信息,这些信息包括动态链接库的名字和其中被调用的函数的名字。若检查到这样的信息,Windows 就会加载相应的动态链接库,并且重定位调用的函数语句的入口地址,以便在调用函数时控制权能转移到函数内部。

2. API 分类

如果从和字符集的相关性来分,API 共有两类:一类是处理 ANSI 字符集的,另一类是处理 UNICODE 字符集的。处理 ANSI 字符集的函数名字的尾部带一个“A”字符,处理 UNICODE 的则带一个“W”字符。我们比较熟悉的 ANSI 字符串是以 NULL 结尾的一串字符数组,每一个 ANSI 字符是一个字节宽。对于欧洲语言体系,ANSI 字符集已足够了,但对于有成千上万个惟一字符的几种东方语言体系来说就只有用 UNICODE 字符集才能满足需求。每一个 UNICODE 字符占有两个字节宽,这样一来就可以在一个字符串中使用 65 336 个不同字符,这也是为什么引进 UNICODE 的原因。

在大多数情况下我们都可以用一个包含头文件,在其中定义一个宏,然后在实际调用函数时,函数名后不需要加后缀‘A’或‘W’。

```
#ifdef UNICODE
#define foo() fooW()
#else
#define foo() fooA()
#endif
```

3. 资源文件的定义

下面让我们看一个很简单的资源文件的源文件,它的扩展名是.rc,当它用资源编译器编译以后产生.res 文件就可以在连接的时候连入.exe 文件中:

```
#include     <Resource.h>
#define     DLG_MAIN 1
DLG_MAIN     DIALOGEX 0,0,236,185
STYLE  DS_MODALFRAME  WS_POPUP  WS_VISIBLE  WS_CAPTION  WS_SYSMENU
CAPTION       "对话框"
FONT       9,"宋体"
BEGIN
    DEFPUSHBUTTON       "退出",IDOK,177,163,50,14
    CONTROL             "",-1,"Static",SS_ETCHEDHORZ,7,155,222,1
END
```

现在简单解释一下 .rc 文件的语法:

#include <Resource.h>:Resource.h 文件包括资源文件的一些常量定义,如下面的 WS_POPUP,WS_VISIBLE 等窗口的风格。

#define DLG_MAIN 1:类似于. asm 文件的 equ 语句,和汇编源程序一样,这些定义是为了程序的可读性。

DLG_MAIN DIALOGEX 0, 0, 236, 185:定义对话框,Windows 的. rc 文件可以定义 BITMAP(位图)、CURSOR(光标)、ICON(图标)、ACCELERATORS(加速键)、DIALOG(对话框)、MENU(菜单)、STRINGTABLE(字符串表)、RCDATA(自定义资源)等 8 种资源,详细的描述可以参考有关 MFC 的书籍,在 Win32ASM 资源编译器的语法中,这些资源的定义方法是:

```
位图定义: nameID BITMAP [load - mem] filename
光标定义: nameID CURSOR [load - mem] filename
图标定义: nameID ICON [load - mem] filename
加速键定  :
        acctablename ACCELERATORS [optional - statements]
        BEGIN event,idvalue,[type] [options]
        . . .
        END
```

具体的定义和参数可以参考 MASM 开发环境 8.2 版中的 Rc. hlp 帮助文件(masm32\bin 目录下)。我们可以用资源编辑器来所见即所得地编辑资源,也可以在文本编辑器中用上面这些语句自己定义资源。

9.2.2　资源文件在汇编中的应用

在程序中,要使用的资源必须先装入内存。本节一开始就介绍了使用资源文件时,Windows 提供了一系列的 API 函数来装入资源,例如 LoadMenu(加载菜单资源)、LoadString(加载字符串资源)、LoadBitmap(加载位图资源)等。

1. 资源加载 API 函数的定义

下面以 LoadBitmap API 函数为例介绍一下资源加载 API 函数的定义与工作原理。

```
LoadBitmap 函数定义:
  HBITMAP LoadBitmap(
    HINSTANCE hInstance,          // 应用实例句柄
    LPCTSTR lpBitmapName          // 位图资源名地址
    );
```

这类 Load 函数的返回值是一个句柄,调用参数中一般至少为两项: hInstance 和 ResouceName,这个 ResouceName(如 BitmapName, MenuName)就是在资源文件中的#define 指定的值,如果使用#define MY_ICON 10 / MY_ICON ICON "Main. ico" 定义了一个图标,那么在程序中要使用 Main. ico 图标就可以用 LoadIcon(hInstance, 10) 来装入已经定义为 10 号的图标文件。另一个参数 hInstance 是执行文件的句柄,它对应资源所在的文件名,可以在程序开始执行时用如下语句获得 hInstance:

```
invoke GetModuleHandle, NULL
```

2. 资源文件的加载

Win32 程序设计中,大部分资源使用 Load 类函数显式地装入,语句如下:

```
invoke LoadCursor, NULL, IDC_ARROW
```

该语句的作用是加载资源名为 IDC_ARROW 的光标。

另外一些资源并不需要使用 Load 类函数显式地装入，如对话框资源，它是在建立对话框的函数中由 Windows 自己装入的，如下面语句所示：

```
invoke DialogBoxParam,hInstance,DLG_MAIN,NULL,offset _ProcDlgMain,0
```

该语句的作用是在屏幕上显示一个资源文件中已经定义好了的对话框，因此程序中并不存在 LoadDialogBox 之类的 API 函数对话框进行先行装入的语句。

9.2.3　编程实例

下面利用 Windows 操作系统提供的资源在桌面上显示一个窗口，窗口标题为“The First Window”，程序中用到了图标资源和光标资源。

1．桌面显示一个窗口的步骤

在桌面显示一个窗口包括以下几个步骤：

① 得到应用程序的句柄(必需)；

② 得到命令行参数(如果程序要从命令行得到参数，可选)；

③ 注册窗口类(必需，除非程序使用 Windows 预定义的窗口类，如 MessageBox 或 dialog box)；

④ 产生窗口(必需)；

⑤ 在桌面显示窗口(必需，除非程序不想立即显示它)；

⑥ 刷新窗口客户区；

⑦ 进入获取窗口消息的循环；

⑧ 如果有消息到达，由窗口回调函数处理；

⑨ 如果用户关闭窗口，进行退出处理。

2．Win32 编程注意事项

相对于单用户的 DOS 下的编程来说，Windows 下的程序框架结构是相当复杂的。这是因为 Windows 和 DOS 在系统架构上截然不同：Windows 是一个多任务的操作系统，故系统中同时有多个应用程序彼此协同运行。这就要求程序员在编写 Windows 程序时必须严格遵守编程规范，并养成良好的编程风格。

在具体介绍窗口程序的源代码之前，首先指出 Win32 程序设计的几个要点。

应当把程序中要用到的所有常量和结构体的声明放到一个头文件中，并且在源程序的开始处包含这个头文件。这么做将会节省大量的时间，避免大量的重复输入。我们也可以定义自己的常量和结构体，但最好把它们放到独立的头文件中。

用 includelib 指令包含程序要引用的库文件，例如，若程序要调用“MessageBox”，就应当在源文件中加入如下一行：

```
includelib user32.lib
```

这条语句告诉 MASM 本程序将要用到一个引用库。如果程序需要若干个引用库，只要简单地加入 includelib 语句，然后在链接时用链接开关 /LIBPATH 指明库所在的路径即可。

在其他地方运用头文件中定义的函数原型、常数和结构体时，要严格保持和头文件中的定义一致，包括大小写。在查询函数定义时，这将节约大量的时间。

3. 窗口程序源代码

【例 9.3】　窗口程序源代码如下:

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib          ; 调用 user32.lib 和
                                           ; kernel32.lib 中的函数
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
.DATA                                      ; 初始化数据
ClassName db "SimpleWinClass",0            ; 为窗口类命名
AppName db "The First Window",0            ; 指定标题名
.DATA?                                     ; 初始化数据
hInstance HINSTANCE ?                      ; 建立程序实例句柄
CommandLine LPSTR ?
.CODE                                      ; 代码段
start:
invoke GetModuleHandle,NULL                ; 得到程序实例句柄
                                           ; 在 Win32 下,hmodule == hinstance
mov hInstance,eax
invoke GetCommandLine                      ; 得到命令行。如果程序没有处理命令;
                                           行,将不能处理这个函数
mov CommandLine,eax
invoke WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
                                           ; 调用主函数
invoke ExitProcess,eax          ; 退出程序,退出码由 WinMain 返回到 eax 中
WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,\
                CmdLine:LPSTR,CmdShow:DWORD
    LOCAL    wc:WNDCLASSEX                 ; 在堆栈中建立本地变量
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov      wc.cbSize,SIZEOF WNDCLASSEX   ; 为 wc 的成员赋值
    mov      wc.style,CS_HREDRAW or CS_VREDRAW
    mov      wc.lpfnWndProc,OFFSET WndProc
    mov      wc.cbClsExtra,NULL
    mov      wc.cbWndExtra,NULL
    push     hInstance
    pop      wc.hInstance
    mov      wc.hbrBackground,COLOR_WINDOW+1
    mov      wc.lpszMenuName,NULL
    mov      wc.lpszClassName,OFFSET ClassName
    invoke   LoadIcon,NULL,IDI_APPLICATION
    mov      wc.hIcon,eax
    mov      wc.hIconSm,eax
    invoke   LoadCursor,NULL,IDC_ARROW
```

```
mov         wc.hCursor,eax
invoke      RegisterClassEx,addr wc          ; 注册窗口类
invoke      CreateWindowEx,NULL,\            ;“\”换行符
                ADDR ClassName,\
                ADDR AppName,\
                WS_OVERLAPPEDWINDOW,\
                CW_USEDEFAULT,\
                CW_USEDEFAULT,\
                CW_USEDEFAULT,\
                CW_USEDEFAULT,\
                NULL,\
                NULL,\
                hInst,\
                NULL
mov         hwnd,eax
invoke      ShowWindow,hwnd,CmdShow          ; 在桌面上显示窗口
invoke      UpdateWindow,hwnd                ; 刷新客户区
.WHILE      TRUE                             ; 进入消息循环
invoke      GetMessage,ADDR msg,NULL,0,0
.BREAK      .IF (! eax)
invoke      TranslateMessage,ADDR msg
invoke      DispatchMessage,ADDR msg
.ENDW
mov         eax,msg.wParam                   ; 在 eax 中返回退出码
    ret
WinMain endp
WndProc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM
      .IF       uMsg==WM_DESTROY             ;如果用户关闭窗口
        invoke PostQuitMessage,NULL          ; 退出应用
      .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
                                             ; 默认的信息处理
        ret
      .ENDIF
      xor eax,eax
      ret
WndProc endp
end start
```

(4) 代码分析

上面的 Windows 程序比较长,但是我们必须要知道上面的大多数代码都是 Win32 程序的模板,模板的意思是指这些代码对差不多所有标准 Windows 程序来说都是相同的。在写 Windows 程序时可以对这些代码进行复制,或者把这些重复的代码写到一个库中。事实上,程序中真正要写的代码集中在 WinMain 中。这和一些 C 编译器一样,不需要关心其他杂务,集中精力于 WinMain 函数。惟一不同的是 C 编译器要求源代码必须有一个函数叫 WinMain,否则 C 不知道将哪个函数和有关的前后代码链接。相对 C,汇编语言提供了较大的灵活性,它不强行要求一个叫 WinMain 的函数。

下面具体分析一下上面的程序。

```
.386
.model flat,stdcall
option casemap:none
```

这三行是一般 Windows 程序必须有的。

.386 告诉 MASN 要使用 80386 指令集。

. model flat, stdcall 语句中 flat 告诉 MASM 程序使用的内存寻址模式, stdcall 告诉 MASM 所用的参数传递约定。

option casemap: none 告诉 MASM 要区分符号的大小写。

```
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
```

这一行是函数 WinMain 的原型申明, 因为程序稍后要用到该函数, 故必须先声明。

```
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

程序必须包含 windows. inc 文件, 因为其中包含大量要用到的常量和结构的定义, 该文件是一个文本文件, 可以用任何文本编辑器打开它。

由于程序调用驻留在 user32. dll (例如, CreateWindowEx, RegisterWindowClassEx) 和 kernel32. dll(ExitProcess) 中的函数, 所以必须链接这两个库。程序中的函数所在的库都必须包含进来。例如, 若要调用的函数在 gdi32. dll 中, 就要包含 gdi32. inc 头文件。

```
.DATA
ClassName db "SimpleWinClass",0
AppName db "Our First Window",0
.DATA?
hInstance HINSTANCE ?
CommandLine LPSTR ?
```

这几行定义了 DATA“分段”。在. DATA 中定义了两个以 NULL 结尾的字符串, 其中 ClassName 是 Windows 类名, AppName 是窗口的名字, 这两个变量都已被初始化。未进行初始化的两个变量放在. DATA? “分段”中, hInstance 代表应用程序的句柄, CommandLine 保存从命令行传入的参数。HINSTACE 和 LPSTR 是两个数据类型名, 它们在头文件中定义, 可以看做是 DWORD 的别名, 之所以要这么重新定义仅仅是为了容易记忆, 相关信息可以查看 windows. inc 文件。在. DATA? 中的变量都是未经初始化的, 这也就是说在程序刚启动时它们的值是什么无关紧要, 只不过占有了一块内存, 以后可以再利用而已。

```
.CODE
start:
invoke GetModuleHandle,NULL
mov hInstance,eax
invoke GetCommandLine
mov CommandLine,eax
```

```
invoke WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
invoke ExitProcess,eax
…
end start
```

程序剩下的部分都是.CODE“分段”,该段包含了应用程序的所有代码,这些代码必须都在.CODE 和 end 之间。至于 label 的命名只要遵循 Windows 规范并且保证其惟一即可。

.CODE“分段”中的第一条语句是调用 GetModuleHandle 去查找应用程序的句柄。在 Win32 下,应用程序的句柄和模块的句柄是一样的,即 hmodule == hinstance。大家可以把实例句柄看成是应用程序的 ID 号。由于程序中调用的几个函数都是把它作为参数来进行传递,所以在程序一开始就要得到并保存它,以便以后程序的设计。需要特别注意的是:Win32 下的实例句柄实际上是该应用程序在内存中的线性地址。

Win32 中函数的返回值是通过 EAX 寄存器来传递的。其他的值可以通过传递进来的参数地址进行返回。一个 Win32 函数被调用时总会保存好段寄存器和 EBX, EDI, ESI, EBP 寄存器,而 ECX 和 EDX 中的值总是不定的,不能在返回中应用。

特别注意:从 Windows API 函数中返回后,EAX, ECX, EDX 中的值和调用前不一定相同。当函数返回时,返回值放在 EAX 中。如果应用程序中的函数提供给 Windows 调用时,也必须遵守这一点,即在函数入口处保存段寄存器和 EBX, ESP, ESI, EDI 的值并在函数返回时恢复。如果不这样做,应用程序会由于现场得不到保护而很快会崩溃。从程序中提供给 Windows 调用的函数大体上有两种:Windows 窗口过程和 Callback 函数。

如果应用程序不处理命令行,就无须调用 GetCommandLine。

接下来的一行是调用 WinMain 函数。该函数共有 4 个参数,这 4 个参数分别包含:应用程序的实例句柄、该应用程序的前一实例句柄、命令行参数串指针和窗口如何显示。Win32 没有前一实例句柄的概念,所以第二个参数总为 0。之所以保留它是出于和 Win16 兼容的考虑,在 Win16 下,如果 hPrevInst 是 NULL,则该函数是第一次运行。

特别注意:在 Win32 编程中不必声明一个名为 WinMain 的函数,只要把 WinMain 中的代码复制到 GetCommandLine 之后即可,所实现的功能完全相同。在 WinMain 返回时,把返回码放到 EAX 中。然后在应用程序结束时通过 ExitProcess 函数把该返回码传递给 Windows。

```
WinMain proc Inst:HINSTANCE,hPrevInst:HINSTANCE,\
CmdLine:LPSTR,CmdShow:DWORD
```

上面一行是 WinMain 的定义。注意跟在 proc 指令后的“参数:类型”形式的参数,它们是由调用者传给 WinMain 的,引用时直接用参数名即可。至于压栈和退栈时的平衡堆栈工作由 MASM 在编译时加入相关的前序和后序汇编指令来进行。

```
LOCAL wc:WNDCLASSEX
LOCAL msg:MSG
LOCAL hwnd:HWND
```

这 3 条 LOCAL 伪指令为局部变量在栈中分配内存空间,所有的 LOCAL 指令必须紧跟在 proc 指令之后。LOCAL 后跟声明的变量,其形式是:

变量名:变量类型

例如, LOCAL wc: WNDCLASSEX 即是告诉 MASM 名字为 wc 的局部变量在栈中分配长度为 WNDCLASSEX 结构体长度的内存空间, 然后程序中再使用该局部变量时就无须考虑堆栈的问题, 不过这种方法存在一些缺陷: 第一, 要求声明的局部变量在函数结束时释放栈空间(也就是不能在函数体外被引用) ; 第二, 不能同时初始化局部变量, 因此不得不在稍后另外再对其赋值。

```
mov wc.cbSize, SIZEOF WNDCLASSEX
mov wc.style, CS_HREDRAWor CS_VREDRAW
mov wc.lpfnWndProc, OFFSET WndProc
mov wc.cbClsExtra, NULL
mov wc.cbWndExtra, NULL
push hInstance
pop wc.hInstance
mov wc.hbrBackground, COLOR_WINDOW+1
mov wc.lpszMenuName, NULL
mov wc.lpszClassName, OFFSET ClassName
invoke LoadIcon, NULL, IDI_APPLICATION
mov wc.hIcon, eax
mov wc.hIconSm, eax
invoke LoadCursor, NULL, IDC_ARROW
mov wc.hCursor, eax
invoke RegisterClassEx, addr wc
```

上面几行完成窗口类(Window Class) 的注册。其中涉及的主要概念是窗口类, 一个窗口类就是一个有关窗口的规范, 这个规范定义了窗口中包含的几个主要元素, 如图标、光标、背景色和负责处理该窗口的函数。产生一个窗口时就必须要有这样的一个窗口类。如果要产生不止一个同种类型的窗口时, 最好的方法就是把这个窗口类存储起来, 这种方法可以节约许多的内存空间。如果要定义自己的创建窗口类就必须在一个 WINDCLASS 或 WINDOWCLASSEXE 结构体中指明窗口的组成元素, 然后调用 RegisterClass 或 RegisterClassEx 根据该窗口类产生窗口。对不同特色的窗口必须定义不同的窗口类。Windows 有几个预定义的窗口类, 例如按钮、编辑框等。要产生该种风格的窗口无须再预先定义窗口类, 只要将包含预定义类的类名作为参数调用 CreateWindowEx 即可。

LpfnWndProc 是窗口处理函数的指针, 它是 WNDCLASSEX 中最重要的成员。前缀 Lpfn 表示该成员是一个指向函数的长指针。注意, 在 Win32 中由于内存模式是 FLAT 型, 所以没有 near 或 far 的区别。每一个窗口类必须有一个窗口过程, 当 Windows 把属于特定窗口的消息发送给该窗口时, 该窗口的窗口类负责处理所有的消息, 如键盘消息或鼠标消息。由于窗口过程差不多智能地处理了所有的窗口消息循环, 所以程序只要在其中加入消息处理过程即可。

下面具体介绍 WNDCLASSEX 的每一个成员。

```
WNDCLASSEX STRUCT DWORD
    cbSize          DWORD       \          ;“ \”换行符
    style           DWORD       \
    lpfnWndProc     DWORD       \
    cbClsExtra      DWORD       \
```

```
cbWndExtra      DWORD       \
hInstance       DWORD       \
hIcon           DWORD       \
hCursor         DWORD       \
hbrBackground   DWORD       \
lpszMenuName    DWORD       \
lpszClassName   DWORD       \
hIconSm         DWORD       \
WNDCLASSEX ENDS
```

cbSize: WNDCLASSEX 的大小, 可以用 sizeof(WNDCLASSEX) 来获得准确的值。

style: 从这个窗口类派生的窗口具有的风格, 可以用“ or ”操作符来把几个风格结合到一起。

lpfnWndProc: 窗口处理函数的指针。

cbClsExtra: 指定紧跟在窗口类结构后的附加字节数。

cbWndExtra: 指定紧跟在窗口事例后的附加字节数。如果一个应用程序在资源中用 CLASS 伪指令注册一个对话框类时, 则必须把这个成员设成 DLGWINDOWEXTRA。

hInstance: 本模块的事例句柄。

hIcon: 图标的句柄。

hCursor: 光标的句柄。

hbrBackground: 背景画刷的句柄。

lpszMenuName: 指向菜单的指针。

lp szClassName: 指向类名称的指针。

hIconSm: 和窗口类关联的小图标。如果该值为 NULL, 则把 hCursor 中的图标转换成大小合适的小图标。

```
invoke CreateWindowEx, NULL, \
                ADDR ClassName, \
                ADDR AppName, \
                WS_OVERLAPPEDWINDOW, \
                CW_USEDEFAULT, \
                CW_USEDEFAULT, \
                CW_USEDEFAULT, \
                CW_USEDEFAULT, \
                NULL, \
                NULL, \
                hInst, \
                NULL
```

注册窗口类后, 将调用 CreateWindowEx 来产生实际的窗口。要注意该函数有 12 个参数(以 ANSI 字符集函数为例) 。

```
CreateWindowExA proto dwExStyle: DWORD, \
lpClassName: DWORD, \
lpWindowName: DWORD, \
dwStyle: DWORD, \
```

```
X: DWORD, \
Y: DWORD, \
nWidth: DWORD, \
nHeight: DWORD, \
hWndParent: DWORD, \
hMenu: DWORD, \
hInstance: DWORD, \
lpParam: DWORD
```

下面分析一下这些参数。

dwExStyle: 附加的窗口风格, 相对于旧的 CreateWindow 这是一个新的参数。在 Windws 9x/NT 中可以使用新的窗口风格。可以在 Style 中指定一般的窗口风格, 但是一些特殊的窗口风格, 如顶层窗口则必须在此参数中指定。如果不想指定任何特别的风格, 则把此参数设为 NULL。

lpClassName: ASCIIZ(以 0 或空字节结尾的, ASCII 码字符串) 形式的窗口类名称的地址。可以是自定义的类, 也可以是预定义的类名。如上面所说, 每一个应用程序都必须有一个窗口类。

lpWindowName: ASCIIZ 形式的窗口名称的地址, 该名称会显示在标题条上。如果该参数为空白, 则标题条上什么都没有。

dwStyle: 窗口的风格, 在此可以指定窗口的外观。可以指定该参数为零, 但那样该窗口就没有系统菜单, 也没有最大化、最小化按钮和关闭按钮, 那样将不得不按 Alt + F4 来关闭它。最为普遍的窗口类风格是 WS_OVERLAPPEDWINDOW。一种窗口风格是一种按位的掩码, 这样可以用‘ or ’把希望的窗口风格结合起来。像 WS_OVERLAPPEDWINDOW 就是由几种最为普遍的风格结合起来的。

X, Y: 指定窗口左上角的以像素为单位的屏幕坐标位置。默认时可指定为 CW_USEDEFAULT, 这样 Windows 会自动为窗口指定最合适的位置。

nWidth, nHeight: 以像素为单位的窗口大小。默认时可指定为 CW_USEDEFAULT, 这样 Windows 会自动为窗口指定最合适的大小。

hWndParent: 父窗口的句柄(如果有的话) , 此参数告诉 Windows 这是一个子窗口以及它的父窗口是谁。这和 MDI(多文档结构) 不同, 此处的子窗口并不会局限在父窗口的客户区内, 它只是用来告诉 Windows 各个窗口之间的父子关系, 以便在父窗口销毁时一同把其子窗口销毁。在本例子的程序中因为只有一个窗口, 故把该参数设为 NULL。

hMenu: Windows 菜单的句柄, 如果只用系统菜单则指定该参数为 NULL。前面 WNDCLASSEX 结构中的 lpszMenuName 参数, 它也指定一个菜单, 这是一个默认菜单, 任何从该窗口类派生的窗口若想用其他的菜单都需使用 hMenu 参数重新指定。其实该参数有双重意义: 一方面若这是一个自定义窗口时该参数代表菜单句柄, 另一方面, 若这是一个预定义窗口时, 该参数代表该窗口的 ID 号。Windows 是根据 lpClassName 参数来区分是自定义窗口还是预定义窗口的。

hInstance: 产生该窗口的应用程序的实例句柄。

lpParam: (可选) 指向欲传给窗口的结构体数据类型参数的指针。如在 MDI 中在产生窗口时传递 CLIENTCREATESTRUCT 结构的参数。一般情况下, 该值总为零, 表示没有参

数传递给窗口。可以通过 GetWindowLong 函数检索该值。

```
mov hwnd,eax
invoke ShowWindow,hwnd,CmdShow
invoke UpdateWindow,hwnd
```

调用 CreateWindowEx 成功后,窗口句柄在 EAX 中,程序必须保存该值以备后用。程序刚刚产生的窗口不会自动显示,所以必须调用 ShowWindow 按照我们希望的方式来显示该窗口。接下来调用 UpdateWindow 来更新客户区。

```
.WHILE TRUE
    invoke GetMessage,ADDR msg,NULL,0,0
    .BREAK .IF (! eax)
    invoke TranslateMessage,ADDR msg
    invoke DispatchMessage,ADDR msg
.ENDW
```

这时候程序产生的窗口已显示在屏幕上了,但是它还不能从外界接收消息,所以程序中还必须给它提供相关的消息。程序是通过一个消息循环来完成该项工作的,每一个模块仅有一个消息循环,程序不断地调用 GetMessage 从 Windows 中获得消息。GetMessage 传递一个 MSG 结构体给 Windows,然后 Windows 在该函数中填充有关的消息,一直到 Windows 找到并填充好消息后 GetMessage 才会返回。在这段时间内系统控制权可能会转移给其他的应用程序。这样就构成了 Win16 下的多任务结构。如果 GetMessage 接收到 WM_QUIT 消息后就会返回 FALSE,使循环结束并退出应用程序。TranslateMessage 函数是一个是实用函数,它从键盘接受原始按键消息,然后解释成 WM_CHAR,在把 WM_CHAR 放入消息队列。由于经过解释后的消息中含有按键的 ASCII 码,因此比原始的扫描码好理解得多。如果应用程序不处理按键消息,就可以不调用该函数。DispatchMessage 会把消息发送给负责该窗口过程的函数。

```
    mov eax,msg.wParam
    ret
WinMain endp
```

如果消息循环结束了,退出码存放在 MSG 中的 wParam 中,就可以通过把它放到 EAX 寄存器中传给 Windows,目前 Windows 没有利用到这个结束码,但程序设计时应该遵循 Windows 规范已防意外。

```
WndProc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM
```

这行代码定义了窗口处理函数,可以随便给该函数命名。其中第一个参数 hWnd 是接收消息的窗口的句柄,uMsg 是接收到的消息。注意 uMsg 不是一个 MSG 结构,实际只是一个 DWORD 类型的数据。Windows 定义了成百上千个消息,大多数应用程序仅仅处理其中的一小部分。当有该窗口的消息发生时,Windows 会发送一个相关消息给该窗口。其窗口过程处理函数会智能地处理这些消息。wParam 和 lParam 只是附加参数,以方便传递更多的和该消息有关的数据。

```
.IF uMsg == WM_DESTROY
```

```
    invoke PostQuitMessage,NULL
.ELSE
    invoke DefWindowProc,hWnd,uMsg,wParam,lParam
    ret
.ENDIF
    xor eax,eax
    ret
WndProc endp
```

上面是窗口处理函数,这是编写 Windows 程序时需要修改的主要部分。此处程序检查 Windows 传递过来的消息,如果是程序感兴趣的消息则加以处理,处理完后,在 EAX 寄存器中传递 0,否则必须调用 DefWindowProc,把该窗口过程接收到的参数传递给默认的窗口处理函数。所有消息中必须处理的是 WM_DESTROY,当应用程序结束时 Windows 把这个消息传递进来,当应用程序运行到该消息时它已经在屏幕上消失,这仅是通知应用程序窗口已销毁,必须自己准备返回 Windows。在此消息中可以做一些清理工作,但无法阻止退出应用程序。如果需要阻止退出应用程序,可以处理 WM_CLOSE 消息。在处理完清理工作后,必须调用 PostQuitMessage,该函数会把 WM_QUIT 消息传回应用程序,而该消息会使得 GetMessage 返回,并在 EAX 寄存器中放入 0,然后会结束消息循环并退回 Windows。可以在程序中调用 DestroyWindow 函数,它会发送一个 WM_DESTROY 消息给应用程序,从而迫使它退出。

9.3　Win32 程序设计实例

上一节中介绍了资源文件的使用,事实上 Win32 程序设计中除了涉及大量的资源信息以外,仍需对大量的 Windows 消息进行处理。例 9.3 实例程序仅仅是在桌面上显示了一个窗口,窗口只能处理 WM_DESTROY 消息。本节将在上节实例的基础上为其增加一些消息处理功能,主要介绍 WM_PAINT 消息、WM_CHAR 消息和 WM_LBUTTONDOWN 消息的处理,关于 Windows 消息方面的其他内容本书不进行详细介绍,大家可参考有关 Windows 程序设计的书籍资料。

9.3.1　WM_PAINT 消息的处理

在窗口的客户区显示字符信息是 Windows 程序最常用的功能之一。简单地说,客户区显示字符信息可以通过响应 WM_PAINT 消息来实现。

1. Windows 系统文本显示的特点

Windows 中的文本是一个 GUI(图形用户界面)对象。每一个字符实际上是由许多的像素点组成,这些点在有笔画的地方显示出来,这样就会出现字符。通常程序都是在应用程序的客户区“绘制”字符串。

Windows 下的“绘制”字符串方法和 DOS 下的字符串显示方法截然不同,在 DOS 下,我们可以把屏幕想像成一个 85× 25 的平面,而 Windows 下由于屏幕上同时有几个应用程序的画面,所以必须严格遵守 Windows 的规范。Windows 是通过把每一个应用程序限制在其各自的客户区来实现这一功能的。注意各程序的客户区大小是可变的,程序运行过程中可被

调整。

在程序的客户区“绘制”字符串前，必须从 Windows 那里得到当前程序客户区的大小。程序在绘制前必须得到 Windows 的允许，当程序被允许在客户区中绘制文本信息后，Windows 会告诉当前客户区的大小、字体、颜色和其他 GUI 对象的属性，这些信息被存储于设备环境中。程序可以直接用这些信息在客户区中绘制信息。

2．设备环境(DC)

“设备环境”其实是由 Windows 内部维护的一个数据结构。一个“设备环境”和一个特定的设备相连，例如打印机和显示器。对于显示器来说，“设备环境”和桌面上一个个特定的窗口相连。

“设备环境”中的有些属性和绘图有关，如颜色、字体等。在程序中可以随时改动那些默认值。我们可以把“设备环境”想像成是 Windows 系统为程序准备的一个绘图环境，而程序可以随时根据需要改变某些默认属性。

3．设备环境的获取

当应用程序需要绘制信息时，程序必须得到一个“设备环境”的句柄。通常有以下几种方法：

在 WM_PAINT 消息中使用 call BeginPaint。

在其他消息中使用 call GetDC。

利用 call CreateDC 建立自己的 DC。

必须注意的是：

在处理单个消息后程序必须释放“设备环境”句柄，不要在一个消息处理中获得“设备环境”句柄，而在另一个消息处理中再释放它。

在 Windows 发送 WM_PAINT 消息时处理绘制客户区，Windows 不会保存客户区的内容，它用的方法是“重绘”机制(例如，当客户区刚被另一个应用程序的客户区覆盖)，Windows 会把 WM_PAINT 消息放入该应用程序的消息队列。

重绘窗口的客户区是各个窗口自己的责任，程序要做的是在窗口过程处理 WM_PAINT 的部分知道绘制什么和如何绘制。

4．无效区域

Windows 系统中把一个最小的需要重绘的正方形区域叫做“无效区域”。当 Windows 发现了一个“无效区域”后，它就会向该应用程序发送一个 WM_PAINT 消息，在 WM_PAINT 的处理过程中，窗口首先得到一个有关绘图的结构体，里面包括无效区的坐标位置等。程序可以通过调用 BeginPaint 让“无效区”有效。注意：如果程序中不处理 WM_PAINT 消息，至少要调用默认的窗口处理函数 DefWindowProc，或者调用 ValidateRect 让“无效区”有效。否则应用程序将会收到无穷无尽的 WM_PAINT 消息。

5．WM_PAINT 消息的响应步骤

下面是响应该消息的步骤：

① 取得“设备环境”句柄；

② 绘制客户区；

③ 释放“设备环境”句柄。

注意，程序中无须显式地让“无效区”有效，这个动作由 BeginPaint 自动完成。程序可以

在 BeginPaint 和 Endpaint 之间调用所有的绘制函数。几乎所有的 GDI 函数都需要“设备环境”的句柄作为参数。

6. 程序代码

【例 9.4】　写一个应用程序,使其在客户区的中心显示一行“欢迎进入 Win32 汇编!”。

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
.DATA
ClassName db "SimpleWinClass",0
AppName db "The First Window",0
OurText db "欢迎进入 Win32 汇编!",0
.DATA?
hInstance HINSTANCE ?
CommandLine LPSTR ?
.CODE
start:
    invoke  GetModuleHandle,NULL
    mov     hInstance,eax
    invoke  GetCommandLine
    mov     CommandLine,eax
    invoke  WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
    invoke  ExitProcess,eax
WinMain proc  Inst:HINSTANCE,hPrevInst:HINSTANCE,\
              CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL  msg:MSG
    LOCAL  hwnd:HWND
    mov    wc.cbSize,SIZEOF WNDCLASSEX
    mov    wc.style,CS_HREDRAWor CS_VREDRAW
    mov    wc.lpfnWndProc,OFFSET WndProc
    mov    wc.cbClsExtra,NULL
    mov    wc.cbWndExtra,NULL
    push   hInst
    pop    wc.hInstance
    mov    wc.hbrBackground,COLOR_WINDOW+1
    mov    wc.lpszMenuName,NULL
    mov    wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov    wc.hIcon,eax
    mov    wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
```

```
        mov     wc.hCursor,eax
        invoke RegisterClassEx,addr wc
        invoke CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
                WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
                CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,\
                hInst,NULL
        mov     hwnd,eax
        invoke  ShowWindow,hwnd,SW_SHOWNORMAL
        invoke  UpdateWindow,hwnd
        .WHILE  TRUE
            invoke GetMessage,ADDR msg,NULL,0,0
            .BREAK .IF (! eax)
            invoke TranslateMessage,ADDR msg
            invoke DispatchMessage,ADDR msg
            .ENDW
            mov eax,msg.wParam
            ret
WinMain endp
WndProc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM
    LOCAL hdc:HDC
    LOCAL ps:PAINTSTRUCT
    LOCAL rect:RECT
    .IF uMsg == WM_DESTROY
      invoke PostQuitMessage,NULL
    .ELSEIF uMsg == WM_PAINT
        invoke  BeginPaint,hWnd,ADDR ps
        mov h   dc,eax
        invoke  GetClientRect,hWnd,ADDR rect
        invoke  DrawText,hdc,ADDR OurText,-1,ADDR rect,\
        DT_SINGLELINE or DT_CENTER or DT_VCENTER
        invoke  EndPaint,hWnd,ADDR ps
    .ELSE
    invoke      DefWindowProc,hWnd,uMsg,wParam,lParam
    ret
    .ENDIF
    xor     eax,eax
    ret
WndProc  endp
end  start
```

7. 代码分析

```
LOCAL  hdc:HDC
LOCAL  ps:PAINTSTRUCT
LOCAL  rect:RECT
```

这些局部变量由处理 WM_PAINT 消息中的 GDI 函数调用。hdc 用来存放调用 BeginPaint 返回的“设备环境”句柄。ps 是一个 PAINTSTRUCT 数据类型的变量。通常程序只会用到其中的部分值，这些值由 Windows 传递给 BeginPaint，在结束绘制后再原封不动地传递给 EndPaint。rect 是一个 RECT 结构体类型参数，它的定义如下：

```
RECT    Struct left LONG ?
top     LONG ?
right   LONG ?
bottom  LONG ?
RECT    ends
```

left 和 top 是正方形左上角的坐标。right 和 bottom 是正方形右下角的坐标。客户区的左上角的坐标是 x = 0、y = 0, 这样对于 x = 0、y = 10 的坐标点就在它的下面。

```
invoke BeginPaint, hWnd, ADDR ps
mov hdc, eax
invoke GetClientRect, hWnd, ADDR rect
invoke DrawText, hdc, ADDR OurText, -1, ADDR rect, \
DT_SINGLELINE or DT_CENTER or DT_VCENTER
invoke EndPaint, hWnd, ADDR ps
```

在处理 WM_PAINT 消息时, 程序调用 BeginPaint 函数, 传给 BeginPaint 函数一个窗口句柄和未初始化的 PAINTSTRUCT 型参数。调用成功后在 eax 中返回“设备环境”的句柄。下一次, 调用 GetClientRec 以得到客户区的大小并放在 rect 中, 然后把它传给 DrawText。DrawText 的语法如下:

```
DrawText proto hdc: HDC, lpString: DWORD, \
nCount: DWORD, lpRect: DWORD, uFormat: DWORD
```

DrawText 是一个高层的调用函数。它能自动处理类似于换行、把文本放到客户区中间等这些操作。所以程序只管集中精力“绘制”字符串即可。下面看一下 DrawText 函数的参数。

hdc: “设备环境”的句柄。

lpString: 要显示的文本串, 该文本串要么以 NULL 结尾, 要么在 nCount 中指出它的长短。

nCount: 要输出的文本的长度。若以 NULL 结尾, 该参数必须是 - 1。

lpRect: 指向要输出文本串的正方形区域的指针, 该方形必须是一个裁剪区, 也就是说超过该区域的字符将不能显示。

uFormat: 指定如何显示。我们可以用 or 把以下标志或到一块。

DT_SINGLELINE: 是否单行显示。

DT_CENTER: 是否水平居中。

DT_VCENTER : 是否垂直居中。

结束绘制后, 必须调用 EndPaint 释放“设备环境”的句柄。

8. 要点小结

字符串绘制过程以下几点是必须注意的。

① 必须在开始和结束处分别调用 BeginPaint 和 EndPaint。

② 在 BeginPaint 和 EndPaint 之间调用所有的绘制函数。

③ 如果在其他的消息处理中重新绘制客户区, 可以有两种选择:

用 GetDC 和 ReleaseDC 代替 BeginPaint 和 EndPaint;

调用 InvalidateRect 或 UpdateWindow 让客户区无效, 这将迫使 WINDOWS 把 WM_

PAINT 放入应用程序消息队列,从而使得客户区重绘。

9.3.2　键盘消息处理

响应按键信息并在窗口的客户区显示字符信息是 Windows 程序最常用的功能之一。这个功能可通过响应 WM_CHAR 和 WM_PAINT 消息来实现。

1. Windows 系统对键盘消息响应的原理

因为大多数的 PC 机只有一个键盘,所以所有运行中的 Windows 程序必须共用它。Windows 将负责把击键消息送到具有输入焦点的应用程序中去。需要注意的是:尽管屏幕上可能同时有几个应用程序窗口,但一个时刻仅有一个窗口有输入焦点。有输入焦点的那个应用程序的标题条总是高亮度显示的。

2. 键盘消息的分类

Windows 系统中把键盘消息分为两类。

一类是把键盘消息看成是很多按键消息的集合,在这种情况下,当按下一个键时,Windows 就会发送一个 WM_KEYDOWN 给有输入焦点的那个应用程序,提醒它有一个键被按下;当释放键时,Windows 又会发送一个 WM_KYEUP 消息,告诉有一个键被释放。这种情况下把每一个键当成是一个按钮。

另一类是把键盘看成是字符输入设备。当按下“A”键时,Windows 发送一个 WM_CHAR 消息给有输入焦点的应用程序,告诉它“A”键被按下。实际上 Windows 内部发送 WM_KEYDOWN 和 WWM_KEYUP 消息给有输入焦点的应用程序,而这些消息将通过调用 TranslateMessage 翻译成 WM_CHAR 消息。

Windows 窗口过程函数将决定是否处理所收到的消息,一般来说程序很少去处理 WM_KEYDOWN, WM_KEYUP 消息,在消息循环中 TranslateMessage 函数会把上述消息转换成 WM_CHAR 消息。下面将具体介绍如何处理 WM_CHAR。

3. 程序代码

【例 9.5】　编写一个应用程序,使其在客户区的左上角显示键盘输入的信息。

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
.data
ClassName db "SimpleWinClass",0
AppName db "The First Window",0
char WPARAM20h
.data?
hInstance HINSTANCE ?
```

```
CommandLine LPSTR ?
.code
sta t:
    invoke  Get ModuleHandle,NULL
    mov     hInstance,eax
        invoke GetCommandLine
        mov CommandLine,eax
        invoke WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
        invoke ExitProcess,eax
WinMain  roc hInst:HINSTANCE,hPrevInst:HINSTANCE,\
         CmdLine:LPSTR,CmdShow:DWORD
         LOCAL    wc:WNDCLASSEX
         LOCAL    msg:MSG
         LOCAL    hwnd:HWND
         mov      wc.cbSize,SIZEOF WNDCLASSEX
         mov      wc.style,CS_HREDRAWor CS_VREDRAW
         mov      wc.lpfnWndProc,OFFSET WndProc
         mov      wc.cbClsExtra,NULL
         mov      wc.cbWndExtra,NULL
         push     hInst
         pop      wc.hInstance
         mov      wc.hbrBackground,COLOR_WINDOW+1
         mov      wc.lpszMenuName,NULL
         mov      wc.lpszClassName,OFFSET ClassName
         invoke   LoadIcon,NULL,IDI_APPLICATION
         mov      wc.hIcon,eax
         mov      wc.hIconSm,eax
         invoke   LoadCursor,NULL,IDC_ARROW
         mov      wc.hCursor,eax
         invoke   RegisterClassEx,addr wc
         inv ke   CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
             WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
             CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,\
             hInst,NULL
             mov      hwnd,eax
             invoke   ShowWindow,hwnd,SW_SHOWNORMAL
             invoke   UpdateWindow,hwnd
             .WHILE   TRUE
                 invoke     GetMessage,ADDR msg,NULL,0,0
                 .BREAK     .IF (! eax)
                 invoke     TranslateMessage,ADDR msg
                 invoke     DispatchMessage,ADDR msg
             .ENDW
             mov     eax,msg.wParam
             ret
WinMain endp
WndP oc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,l Param:LPARAM
      LOCAL hdc:HDC
      LOCAL ps:PAINTSTRUCT
```

```
        .IF uMsg==WM_DESTROY
          invoke PostQuitMessage,NULL
        .ELSEIF uMsg==WM_CHAR
            push    wParam
            pop     char
            invoke  InvalidateRect,hWnd,NULL,TRUE
        .ELSEIF uMsg==WM_PAINT
            invoke  BeginPaint,hWnd,ADDR ps
            mov     hdc,eax
    invoke TextOut,hdc,0,0,ADDR char,1
    invoke EndPaint,hWnd,ADDR ps
    .ELSE
    invoke DefWindowProc,hWnd,uMsg,wParam,lParam
    ret
    .ENDIF
    xor     eax,eax
    ret
WndProc endp
end start
```

4. **代码分析**

```
char WPARAM20h
```

这个变量将保存从键盘接收到的字符。因为它是在窗口过程中通过 WPARAM **型变量传送的,所以我们简单地把它定义为** WPARAM **型。由于程序的窗口在初次刷新时(也即刚被创建的那一次)是没有键盘输入的,所以将其初始化为空格符(**20h**),这样开始显示时就什么都看不见。**

```
.EL  EIF uMsg==WM_CHAR
    push wParam
    pop char
    invoke InvalidateRect,hWnd,NULL,TRUE
```

这一段是用来处理 WM_CHAR **消息的。它把接收到的字符放入变量** char **中,接着调用** InvalidateRect,**而** InvalidateRect **使得窗口的客户区无效,这样它会发出** WM_PAINT **消息,而** WM_PAINT **消息迫使** Windows **重新绘制它的客户区。该函数的语法如下:**

```
InvalidateRect   roto hWnd: HWND, \          ;“ \ ”换行符
                 lpRect: DWORD, \
                 bErase: DWORD
```

lpRect **是指向客户区程序想要使其无效的一个正方形结构体的指针。如果该值等于** NULL,**则整个客户区都无效;布尔值** bErase **告诉** Windows **是否擦除背景,如果是** TRUE,**则** Windows **在调用** BeginPaint **函数时把背景擦掉。所以,此处的做法是:保存所有有关重绘客户区的数据,然后发送** WM_PAINT **消息,处理该消息的程序段再根据相关数据重新绘制客户区。尽管这么做似乎像走了弯路,但** Windows **要处理非常庞大的消息群,必须遵循一定的规则。实际上程序中完全可以通过调用** GetDC **获得设备上下文句柄,然后绘制字符,然后再调用** ReleaseDC **释放设备上下文句柄,这样也能在客户区绘制出正确的字符。但是如果**

在这之后接收到 WM_PAINT 消息要处理时，客户区会重新刷新，而程序在这之前所绘制的字符就会消失掉。因此，为了让字符一直正确地显示，就必须把它们放到 WM_PAINT 的处理过程中处理。

```
invoke TextOut,hdc,0,0,ADDR char,1
```

在调用 InvalidateRect 时，WM_PAINT 消息被发送到了 Windows 窗口处理过程，程序流程转移到处理 WM_PAINT 消息的程序段，然后调用 BeginPaint 得到设备上下文的句柄，再调用 TextOut 在客户区的(0,0)处输出保存的按键字符。这样无论在键盘上按什么键都能在客户区的左上角显示，不仅如此，而且无论怎么缩放窗口(迫使 Windows 重新绘制它的客户区)，字符都会在正确的地方显示，这也是为什么必须把所有重要的绘制动作都放到处理 WM_PAINT 消息的程序段中去的原因。

9.3.3　鼠标消息处理

响应鼠标消息是 Windows 程序最常用的功能之一。下面的示例程序演示了如何等待左键按下消息，并在按下的位置显示一个字符串。这个功能可通过响应 WM_LBUTTONDOWN 和 WM_PAINT 消息来实现。

1．Windows 系统对鼠标消息响应的原理

和处理键盘输入一样，Windows 将捕捉鼠标动作并把它们发送到相关窗口。这些活动包括鼠标键按下、移动、双击以及滚轮消息。Windows 并不像处理键盘输入那样把所有的鼠标消息都导向有输入焦点的窗口，无论窗口是否有输入焦点，任何鼠标经过的窗口都将接收到鼠标消息。另外，窗口还会接收到鼠标在非客户区移动的消息(WM_NCMOVE)，但大多数的情况下这个消息会被忽略。

2．鼠标消息种类

Windows 系统中鼠标消息分为按键消息和移动消息。

Windows 系统对鼠标的按键操作会产生两个消息：WM_xBUTTONDOWN 和 WM_xBUTTONDOWN，其中 x 对应 L，R，对于三键鼠标还会有 WM_MBUTTONDOWN 和 WM_MBUTTONUP 消息。一个窗口若想处理 WM_LBUTTONDBCLK 或 WM_RBUTTONDBCLK，那么它的窗口类必须有 CS_DBLCLKS 风格，否则它就会接收到很多按键起落(WM_xBUTTONDOWN或 WM_xBUTTONUP)的消息。

当鼠标在某窗口客户区移动时，该窗口将接收到 WM_MOUSEMOVE 消息。

3．鼠标消息参数的传递

对于所有类型的鼠标消息其参数都是通过 lParam 和 wParam 传递的。窗口过程函数传入的参数 lParam 包含了鼠标的位置，其中低位为 x 坐标，高位为 y 坐标，这些坐标值都是相对于窗口客户区的左上角的值，wParam 中则包含了鼠标按钮的状态。

4．程序代码

下面将编写一个应用程序，它会在鼠标左键按下的位置显示“The First Window”字符串。

【例 9.6】　在屏幕上显示“The First Window”字符串。

```
.386
.model    flat,stdcall
```

```
option    casemap:none
WinMain   proto :DWORD,:DWORD,:DWORD,:DWORD
include   \masm32\include\windows.inc
include   \masm32\include\user32.inc
include   \masm32\include\kernel32.inc
include   \masm32\include\gdi32.inc
includelib\masm32\lib\user32.lib
includelib\masm32\lib\kernel32.lib
includelib\masm32\lib\gdi32.lib
.data
ClassName db "SimpleWinClass",0
AppName db "The First Window",0
MouseClick db 0                          ;0 = 没有鼠标按下
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hitpoint POINT < >
.code
sta t:
    invoke    GetModuleHandle,NULL
    mov       hInstance,eax
    invoke    GetCommandLine
    mov       CommandLine,eax
    invoke    WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
    invoke    ExitProcess,eax
Win ain proc  Inst:HINSTANCE,hPrevInst:HINSTANCE,\
              CmdLine:LPSTR,CmdShow:DWORD
    LOCAL     wc:WNDCLASSEX
    LOCAL     msg:MSG
    LOCAL     hwnd:HWND
    mov       wc.cbSize,SIZEOF WNDCLASSEX
    mov       wc.style,CS_HREDRAW or CS_VREDRAW
    mov       wc.lpfnWndProc,OFFSET WndProc
    mov       wc.cbClsExtra,NULL
    mov       wc.cbWndExtra,NULL
    push      hInst
    pop       wc.hInstance
    mov       wc.hbrBackground,COLOR_WINDOW+1
    mov       wc.lpszMenuName,NULL
    mov       wc.lpszClassName,OFFSET ClassName
    invoke    LoadIcon,NULL,IDI_APPLICATION
    mov       wc.hIcon,eax
    mov       wc.hIconSm,eax
    invoke    LoadCursor,NULL,IDC_ARROW
    mov       wc.hCursor,eax
    invoke    RegisterClassEx,addr wc
    inv ke    CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
        WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
        CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,\
```

```
        hInst,NULL
    mov         hwnd,eax
    invoke      ShowWindow,hwnd,SW_SHOWNORMAL
    invoke      UpdateWindow,hwnd
    .WHILE      TRUE
        invoke  GetMessage,ADDR msg,NULL,0,0
        .BREAK  .IF (! eax)
        invoke  DispatchMessage,ADDR msg
    .ENDW
    mov         eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM
    LOCAL hdc:HDC
    LOCAL ps:PAINTSTRUCT
    .IF uMsg == WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg == WM_LBUTTONDOWN
    mov  eax,lParam
    and  eax,0FFFFh
    mov  hitpoint.x,eax
    mov  eax,lParam
    shr  eax,16
    mov  hitpoint.y,eax
    mov  MouseClick,TRUE
    invoke  InvalidateRect,hWnd,NULL,TRUE
    .ELSEIF uMsg == WM_PAINT
        invoke  BeginPaint,hWnd,ADDR ps
        mov     hdc,eax
        .IF  ouseClick
            invoke  lstrlen,ADDR AppName
            invoke  TextOut,hdc,hitpoint.x,hitpoint.y,ADDR AppName,eax
        .ENDIF
        invoke  EndPaint,hWnd,ADDR ps
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor     eax,eax
    ret
WndProc endp
end start
```

5. 代码分析

```
.ELSEIF uMsg == WM_LBUTTONDOWN
```

窗口过程处理了 WM_LBUTTONDOWN **消息,当接收到该消息时,**lParam **中包含了相对于窗口客户区左上角的坐标,程序将它保存下来,放到一个结构体变量(**POINT**)中,该结构体变量的定义如下:**

```
POI  T STRUCT
    x dd ?
    y dd ?
POINT ENDS
```

由于 lParam 是一个 32 位长的数, 其中高、低 16 位分别包括了 x, y 坐标, 要得到 x, y 坐标必须对 lParam 做一些处理, 以便保存它们, 处理过程如下:

```
mov eax,lParam
and eax,0FFFFh
mov hitpoint.x,eax
shr eax,16
mov hitpoint.y,eax
```

保存完坐标后设标志 MouseClick 为 TRUE, 这是在处理 WM_PAINT 时用来判断是否有鼠标左键按下消息的标志。

```
mov MouseClick,TRUE
```

然后调用 InvalidateRect( ) 函数迫使 Windows 重新绘制客户区。

```
invoke InvalidateRect,hWnd,NULL,TRUE
```

绘制客户区的代码首先检测 MouseClick 标志位, 再决定是否重绘。

```
.IF MouseClick
invoke lstrlen,ADDR AppName
invoke TextOut,hdc,hitpoint.x,hitpoint.y,ADDR AppName,eax
.ENDIF
```

因为在首次显示窗口时还没有左键按下的消息, 所以在初始时把该标志设为 FALSE, 告诉 Windows 不要重绘客户区, 当有左键按下的消息时, 它会在鼠标按下的位置绘制字符串。注意在调用 TextOut( ) 函数时, 其关于字符串长度的参数是调用 lstrlen( ) 函数来计算的。

习　　题

1. 简述汇编语言 Win32 程序框架以及框架的各部分功能。
2. 简述例 9.1 为什么没有使用函数原型声明语句 include kernel32.inc 和 include user32.inc。
3. 简述如何利用宏实现用相同函数名完成对 ANSI 字符集和 UNICODE 字符集的处理。
4. 对包含资源文件的汇编语言程序进行汇编时应注意哪些问题?
5. 加载资源文件的 Load 类 API 函数的返回值是一个_____, 调用参数中一般至少为两项: _____和_____。
6. 简述资源文件的加载方法。
7. 编写程序: 利用 32 位汇编语言编程实现单行字符串的输入。
8. 编写程序, 利用资源文件编程实现更改应用程序图标。

附录 A　ASCII 码表

基本的 ASCII 字符集共有 128 个字符，其中有 96 个可打印字符，包括常用的字母、数字、标点符号等，另外还有 32 个控制字符。标准 ASCII 码使用 7 个二进位对字符进行编码，对应的 ISO 标准为 ISO646 标准。下表显示了基本 ASCII 字符集及其编码。

H / L	0000	0001	0010	0011	0100	0101	0110	0111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB		7	G	W	g	w
1000	BS	CAN	)	8	H	X	h	x
1001	HT	EM	(	9	H	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	～
1111	SI	US	/	?	O	_	o	DEL

标准 ASCII 字符集字符数目有限，在实际应用中往往无法满足要求。为此，国际标准化组织又制定了 ISO2022 标准，它规定了在保持与 ISO646 兼容的前提下将 ASCII 字符集扩充为 8 位代码的统一方法。ISO 陆续制定了一批适用于不同地区的扩充 ASCII 字符集，每种扩充 ASCII 字符集分别可以扩充 128 个字符，这些扩充字符的编码均为高位为 1 的 8 位代码（即十进制数 128 ～255），称为扩展 ASCII 码。关于扩展 ASCII 字符集及其编码可参阅相关资料。

附录 B　DOS 和 BIOS 的宏定义

1．控制台输入和输出(Console Input and Output)

@Ge Char　　从键盘读字符

使用语法:　@GetChar [echo] [, [break] [, clearbuf]]

参数说明:　< echo >　　常量,非零表示“ 回显 ”,默认值为“ 回显 ”

< break >　　常量,非零表示接受“ ^C ”,默认值为“ 接受 ”

< clearbuf >　　常量,非零表示清键盘缓冲区,默认值为“ 不清 ”

返回参数:　AL = ASCII 码

内容破坏:　AX, DL(若回显,且不接受^C)

参见内容:　INT 21H—01H, 07H, 08H 和 0CH, @GetStr

@Ge Str　　从键盘读字符串

使用语法:　@GetStr buffer [, [terminator] [, [limit] [, segment]]]

参数说明:　< buffer >　　字符串的偏移量,必须是偏移量地址

字节 1——调用前,字符串的最大长度

字节 2——调用后,字符串的实际长度

字节 3——存放着字符串

< terminator >　　常量/寄存器,不能是存储单元,终止字节为 0 或 $

< limit >　　常量,字符串的最大长度。若未指定该参数,则调用前它必须在缓冲区

< segment >　　缓冲区的段地址,若未指定,默认值是 DS

返回参数:　SI = 指向字符串, BX = 字符串长度

内容破坏:　AX, DX, BX 和 SI

参见内容:　INT 21H — 0AH, @GetChar

@Sh wChar　　在屏幕上显示若干个字符

使用语法:　@ShowChar char [, char] . . .

参数说明:　< char >　　ASCII 码

返回参数:　无

内容破坏:　AX 和 DL

参见内容:　INT 21H — 02H

@Pr Char　　向 LPT1 输出若干个字符

使用语法:　@PrtChar char [, char] . . .

参数说明:　< char >　　ASCII 码

返回参数:　无

内容破坏:　AX 和 DL

参见内容:　INT 21H — 05H, @ShowStr

@Sh wStr　　显示用“ $ ”结束的字符串

使用语法:　@ShowStr address [, segment]

参数说明:　< address >　　字符串的偏移量,该字符串由“ $ ”结束

< segment >　字符串的段,若未指定,则其默认值为 DS

返回参数:　无

内容破坏:　AX, DX, DS(若段改变了)

参见内容:　INT 21H—09H

2. 设备和文件控制(Device and File Control)

@Re d: 从文件或设备读数据

使用语法:　@Read buffer, length [, [handle] [, segment]]

参数说明:　< buffer >　存放数据的缓冲区偏移量

< length >　数据字节的长度

< handle >　文件或设备的句柄,默认值为 0(键盘)

< segment >　字符串的段地址,默认值为 DS

返回参数:　若 CF = 0, 则 AX = 读入的字节数

内容破坏:　AX, BX, CX, DX 和 DS(若段改变了)

参见内容:　INT 21H—3FH, @Write

@Wr te: 向文件或设备写数据

使用语法:　@Write buffer, length [, [handle] [, segment]]

参数说明:　< buffer >　存放数据的缓冲区偏移量

< length >　数据字节的长度

< handle >　文件或设备的句柄,默认值为 1(屏幕)

< segment >　字符串的段地址,默认值为 DS

返回参数:　若 CF = 0, 则 AX = 输出的字节数

内容破坏:　AX, BX, CX, DX 和 DS(若段改变了)

参见内容:　INT 21H—40H, @Read

@Ma eFile: 创建一个文件

使用语法:　@MakeFile path [, [attrib] [, [segment] [, kind]]]

参数说明:　< path >　文件名的地址

< attrib >　文件的属性,默认值为 0

< segment >　字符串的段地址,默认值为 DS

< kind >　可用" tmp" /" new",若不指定,则创建一个新文件(即使文件已存在)

" tmp" 创建一个惟一文件

" new" 创建一个新文件(在原文件不存在时)

返回参数:　若 CF = 0, 则 AX = 文件的句柄

内容破坏:　AX, CX, DX 和 DS(若段改变了)

参见内容:　INT 21H—3CH, 5AH 和 5B, @OpenFile, @CloseFile

@Op nFile: 为输入/输出而打开文件

使用语法:　@OpenFile path, access [, segment]

参数说明:　< path >　文件名的地址

< access >　常量, 文件访问代码,默认值为 0(可读/写)

< segment >　字符串的段地址,默认值为 DS

返回参数:　若 CF = 1, 则 AX = 错误代码

内容破坏:　AX, DX 和 DS(若段改变了)

参见内容:　INT 21H—3DH, @MakeFile

@Cl seFile: 关闭文件句柄

使用语法: @CloseFile handle

参数说明: < handle > 先前打开的文件句柄

返回参数: 若 CF = 1, 则 AX = 错误代码

内容破坏: AX 和 BX

参见内容: INT 21H—3EH, @OpenFile, @MakeFile

@De File: 删除一个指定的文件

使用语法: @DelFile path [, segment]

参数说明: < path > 文件名字符串的偏移量

< segment > 路径的段地址, 默认值为 DS

返回参数: 若 CF = 1, 则 AX = 错误代码

内容破坏: AX, DX 和 DS(若段改变了)

参见内容: INT 21H—41H

@Mo eFile: 文件移动或换名

使用语法: @MoveFile old, new [, [segold] [, segnew]]

参数说明: < old > 被换名的文件名字符串的偏移量

< new > 新文件名字符串的偏移量

< segold > 旧文件名的段地址, 默认值为 DS

< segnew > 新文件名的段地址, 默认值为 ES

返回参数: 若 CF = 1, 则 AX = 错误代码

内容破坏: AX, DX, DI, DS 和 ES(若相应段改变了)

参见内容: INT 21H—56H

@GetFirst: 读取第一个匹配的文件

@Ge Next: 读取其他的匹配文件

使用语法: @GetFirst path [, [attribute] [, segment]]

@GetNext

参数说明: < path > 指定文件的偏移量, 可含通配符

< attribute > 被搜索的文件属性, 默认值为 0

< segment > 路径的段地址, 默认值为 DS

返回参数: 若 CF = 1, 则 AX = 错误代码

内容破坏: AX(二者) , 对@GetFirst, CX, DX 和 DS(若段改变了)

参见内容: INT 21H—4EH 和 4FH, @SetDTA, @GetDTA

@GetDTA: 读取 DTA(Disk Transfer Address)

@Se DTA: 设置 DTA

使用语法: @SetDTA buffer [, segment]

@GetDTA

参数说明: < buffer > 新 DTA 缓冲区的偏移量

< segment > 新 DTA 缓冲区的段地址, 默认值为 DS

返回参数: 对@GetDTA, ES: BX = 指向 DTA 的指针

内容破坏: 对@GetDTA, AX, ES 和 BX

对@SetDTA, AX, DS 和 DX

参见内容: INT 21H—2FH 和 1AH, @GetNext

@Ge FileSize: 读取文件大小

使用语法:　@GetFileSize handle

参数说明:　< handle >　先前打开的文件句柄

返回参数:　若 CF = 0, 则, DX: AX = 文件的大小

内容破坏:　AX, BX, CX 和 DX

注意: 该宏将把文件指针复位到 0, 所以在文件操作过程中, 不要使用该宏

参见内容:　INT 21H—42H

@MovePtrAbs: 移动文件指针到一个绝对位置

@Mo ePtrRel: 移动文件指针到一个相对位置

使用语法:　@MovePtrAbs handle [, distance]

@MovePtrRel handle [, distance]

参数说明:　< handle >　先前打开的文件句柄

< distance >　16 位常量或 16 /32 位变量, 默认值是 CX: DX

返回参数:　若 CF = 0, 则, DX: AX = 文件指针的位置

内容破坏:　AX, BX, CX 和 DX

参见内容:　INT 21H—42H

3. 目录和驱动器控制(Directory & Drive Control)

@MkDir: 创建一个目录

@RmDir: 删除一个目录

@Ch ir: 改变当前目录

使用语法:　@MkDir path [, segment]

@RmDir path [, segment]

@ChDir path [, segment]

参数说明:　< path >　目录路径字符串的地址

< segment >　路径的段地址, 默认值为 DS

返回参数:　若 CF = 1, 则 AX = 错误代码

内容破坏:　AX, DX, DS(若段改变了)

参见内容:　INT 21H—39H, 3AH 和 38H, @GetDir

@Ge Dir: 获取指定驱动器的当前目录

使用语法:　@GetDir buffer [, [drive] [, segment]]

参数说明:　< path >　接受目录路径的缓冲区地址

< drive >　驱动器号(一个字节) (0 = 当前, 1 = A, 2 = B, . . .) , 默认值为 0

< segment >　路径的段地址, 默认值为 DS

返回参数:　若 CF = 1, 则 AX = 错误代码

内容破坏:　AX, SI, DL, DS(若段改变了)

参见内容:　INT 21H—47H, @ChDir, @GetDrv

@GetDrv: 获取当前驱动器

@Se Drv: 设置当前驱动器

使用语法:　@GetDrv

@SetDrv drive

参数说明:　< drive >　驱动器号(一个字节) (0 = A, 1 = B, . . .)

返回参数:　对@GetDrv, AL = 驱动器号(0 = A, 1 = B, . . .)

对@SetDrv, AL = 驱动器数

内容破坏:　AX(二者) , DL(在@ SetDrv 中)

参见内容:　INT 21H—19H 和 0EH, @GetDir, @ ChkDrv

@Ch Drv: **读取磁盘的信息**

使用语法:　@ChkDrv [drive]

参数说明:　< drive >　驱动器号(一个字节) (0 = 当前, 1 = A, 2 = B, . . .) , 默认值为 0

返回参数:　AX = 每柱面中的扇区数, - 1 - 非法的驱动器

BX = 可用柱面

CX = 每扇区的字节数

DX = 驱动器中的柱面数

内容破坏:　AX, BX, CX 和 DX

参见内容:　INT 21H—36H, @GetDrv

4. 内存控制(Memory Control)

@Fr eBlock: **释放内存块**

使用语法:　@FreeBlock [segment]

参数说明:　< segment >　要释放内存的起始地址, 其默认值为 ES

返回参数:　若 CF = 1, 则 AX = 错误代码

内容破坏:　AX, ES(若有段参数)

参见内容:　INT 21H—49H, @GetBlock, @ModBlock

@Ge Block: **分配内存块**

使用语法:　@GetBlock paragraphs [, retry]

参数说明:　< paragraphs >　需要内存的段落数

< retry >　若非零, 则分配满足要求的最大块

返回参数:　若 CF = 1, 则 AX = 错误代码, 否则, AX = 被分配内存的段地址, BX = 实际分配的段落数

内容破坏:　AX 和 BX

参见内容:　INT 21H—48H, @FreeBlock, @ModBlock

@Mo Block:　**修改一个已分配的内存块**

使用语法:　@ModBlock paragraphs [, segment]

参数说明:　< paragraphs >　需要内存的段落数

< segment >　释放内存单元的起始地址, 默认值为 ES

返回参数:　若 CF = 1, 则 AX = 错误代码, 否则, ES = 被分配内存的段地址, BX = 实际分配的段落数

内容破坏:　AX, BX, ES(若指定了段)

参见内容:　INT 21H—4AH, @GetBlock, @FreeBlock

5. DOS 杂项(Miscellaneous DOS)

@Ge Date: **读取系统日期**

使用语法:　@GetDate

参数说明:　无

返回参数:　AL = 一周内的日期(0 - Sunday, 1 - Monday, . . .)

CX = 年(1980 ~2099)

DH = 月

DL = 日

内容破坏:　AX, CX 和 DX

参见内容:　INT 21H—2AH, @SetDate, @GetTime

@Se Date: **设置系统日期**

使用语法:　@SetDate month, day, year

参数说明:　< month >　**月份**(1 - 12)

< day >　**日**(1 - 31)

< year >　**年**(1980 - 2099)

返回参数:　**若日期合法, 则** AL = 0, **否则**, AL = - 1

内容破坏:　AX, CX 和 DX

参见内容:　INT 21H—2BH, @GetDate, @SetTime

@Ge Time: **读取系统时间**

使用语法:　@GetTime

参数说明:　**无**

返回参数:　CH = **小时**(0～23)

CL = **分钟**(0～59)

DH = **秒**(0～59)

DL = **百分秒**(0～99)

内容破坏:　AX, CX 和 DX

参见内容:　INT 21H—2CH, @GetDate, @SetTime

@Se Time: **设置系统时间**

使用语法:　@SetTime hour, minute, second, hundredth

参数说明:　< hour >　**小时**(0～23)

< minute >　**分钟**(0～59)

< second >　**秒**(0～59)

< hundredth >　**百分秒**(0～99)

返回参数:　**若时间合法, 则** AL = 0, **否则**, AL = - 1

内容破坏:　AX, CX 和 DX

参见内容:　INT 21H—2DH, @SetDate, @GetTime

@Ge Ver: **读取 DOS 版本**

使用语法:　@GetVer

参数说明:　**无**

返回参数:　AL = **主要版本号**, AH = **次要版本号**

BH = OEM **序列号**, BL: CX = 24 **位用户号**

内容破坏:　AX, BX 和 CX

参见内容:　INT 21H—30H, @SetDate, @GetTime

@GetInt: **读取指定中断的入口地址**

@Se Int: **设置指定中断的入口地址**

使用语法:　@GetInt interrupt

@SetInt interrupt, vector [, segment]

参数说明:　< interrupt >　**常量, 中断号**(8 **位**)

< vector >　**中断处理程序的偏移量**

< segment >　**中断处理程序的段地址, 默认值** DS **对数据区**

返回参数:　**对**@GetInt, ES: BX = **指向中断服务程序**

内容破坏:　**改变**@GetInt, AX, ES **和** BX

改变@SetInt, AX, DS **和** DX

参见内容:　INT 21H—35H, 25H

@Ex t: **带返回码返回到** DOS

使用语法:　@Exit [return]

参数说明:　< return >　　**常量**(8 **位**) , **默认值为** AL

返回参数:　**无**

内容破坏:　AX

参见内容:　INT 21H—4CH, . EXIT

@TS　: **终止程序运行, 并驻留**

使用语法:　@TSR paragraphs [, return]

参数说明:　< paragraphs >　　**驻留程序需要分配的内存段落数**

< return >　　**常量**(8 **位**) , **默认值为** AL

返回参数:　**无**

内容破坏:　AX **和** DX

参见内容:　INT 21H—31H

6. 模式、页和颜色等控制(Mode, Page & Color Control)

@Ge Mode: **读取当前显示模式和显示页**

使用语法:　@GetMode

参数说明:　**无**

返回参数:　AL = **显示模式**

AH = **屏幕宽度**(**字符数**)

BH = **显示页号**

内容破坏:　AX **和** BH

参见内容:　INT 10H—0Fh, @SetMode

@Se Mode: **设置当前显示模式和显示页**

使用语法:　@SetMode mode

参数说明:　< mode >　　**显示模式**(**一个字节**)

返回参数:　**无**

内容破坏:　AX

参见内容:　INT 10H—00H, @GetMode

@Se Color: **设置背景色**

使用语法:　@SetColor color

参数说明:　< color >　　**背景色**(0 ~15, **一个字节**)

返回参数:　**无**

内容破坏:　AX **和** BX

参见内容:　INT 10H—0BH

@Se Palette: **设置调色板**

使用语法:　@SetPalette color

参数说明:　< color >　　**调色板颜色**(**一个字节**)

返回参数:　**无**

内容破坏:　AX **和** BX

参见内容:　INT 10H—0BH

@Se Page: **设置显示页**

使用语法:　@SetPage page

参数说明:　< page >　　**页码(一个字节)**

返回参数:　**无**

内容破坏:　AX

参见内容:　INT 10H—05H

7. 字符和光标控制(Character and Cursor Control)

@Ge Csr: **读取光标的位置和大小**

使用语法:　@GetCsr [page]

参数说明:　< page >　　**显示页(一个字节),默认值为** 0

< segment >　　**字符串的段,若未指定,则其默认值为** DS

返回参数:　DL = **列**

DH = **行**

CL = **光标起起行**

CH = **光标结束行**

内容破坏:　AX, DX, CX **和** BH

参见内容:　INT 10H—03h, @SetCsrPos, @SetCsrSize

@Se CsrPos: **设置光标位置**

使用语法:　@SetCsrPos [column] [, [row] [, page]]

参数说明:　< column >　　**列(一个字节),默认值为** DL

< row >　　**行(一个字节),默认值为** DH

< page >　　**光标所在的页(一个字节),默认值为** 0

返回参数:　**无**

内容破坏:　AX, DX **和** BH

参见内容:　INT 10H—02H, @GetCsr

@Se CsrSize: **设置光标的大小和形状**

使用语法:　@SetCsrSize startline, endline

参数说明:　< startline >　　**起始线(一个字节),默认值为** 6(CGA) /12

< endline >　　**结束线(一个字节),默认值为** 7(CGA) /13

返回参数:　**无**

内容破坏:　AX **和** CX

参见内容:　INT 10H—01H, @GetCsr

@Ge CharAtr: **读取光标处的字符及其属性**

使用语法:　@GetCharAtr [page]

参数说明:　< page >　　**页号(一个字节),默认值为** 0

返回参数:　AH = **属性**, AL = **字符** ASCII **码**

内容破坏:　AX **和** BH

参见内容:　INT 10H—08H, @PutCharAtr

@Pu CharAtr: **在光标处显示指定属性的字符**

使用语法:　@PutCharAtr [character] [, [attrib] [, [page] [, count]]]

@PutChar [character] [, [page] [, count]]

参数说明:　< character >　　**显示字符, 其默认值为** AL
< attrib >　　**显示属性(一个字节) , 默认值为** BL
< page >　　**页号(一个字节) , 默认值为** 0
< count >　　**显示次数, 默认值为** 1

返回参数:　**无**

内容破坏:　AX, BX **和** CX

参见内容:　INT 10H—09H **和** 0AH, @GetCharAtr

@Sc oll: **向上 / 向下滚动指定的窗口**

使用语法:　@Scroll dist [, [attr] [, [upcol [, [uprow [, [dncol] [, dnrow]]]]]]]

参数说明:　< dist >　　**滚动方向, 正 - 向下滚, 负 - 向下滚,** 0 **- 清屏**
< attr >　　**空白行属性(一个字节) , 默认值为** 7(**黑底白字**)
< upcol >　　**左上角列, 默认值为** CL
< uprow >　　**左上角行, 默认值为** CH
< dncol >　　**右下角列, 默认值为** DL
< dnrow >　　**右下角列, 默认值为** DH

返回参数:　**无**

内容破坏:　AX, CX, DX **和** BH

参见内容:　INT 10H—06H **和** 07H

@Cl : **清屏**

使用语法:　@Cls [page]

参数说明:　< page >　　**页号(一个字节) , 默认值为** 0

返回参数:　**无**

内容破坏:　AX, BX, CX **和** DX

参见内容:　INT 10H—06H **和** 07H

附录 C　DEBUG 命令表

命令	用　途	格　式
A	汇编语句	A[<地址>]
C	比较存储器内容	C <源地址范围> <目的地址范围>
D	显示存储器内容	D[<地址>] 或 D[<起始地址>] [<目的地址>]
E	修改存储器内容	E <地址> [<字节串>]
F	填充存储器内容	F <地址> <要填入的字节或字节串>
G	运行程序	G[= <起始地址>] [<断点> …]
H	计算十六进制的和与差	H 数 1, 数 2
I	从指定端口输入并显示	I <端口地址>
L	装入文件或磁盘扇区	L[<地址>] [<盘号> <逻辑扇区号> <扇区数>]
M	移动存储器内容	M <源地址范围> <目的地址>
N	定义文件和参数	N <文件名> [<文件名> …]
O	向指定的端口输出字节	O <端口地址> <字节>
Q	结束 DEBUG 运行	Q
R	显示和修改寄存器	R[<寄存器号>]
S	搜索字符合字符串	S <地址范围> <要查找的字节和字节串>
T	跟踪运行并显示	T[= <地址>] [<跟踪条数>]
U	对指令进行反汇编	U[<地址范围>]
W	写磁盘文件或磁盘扇区	W[<地址> [<盘号> <逻辑扇区号> <扇区数>]]]
XA	分配扩展内存	XA[#页面号]
XD	释放扩展内存	XD[句柄]
XM	影像扩展内存页	XM[Lpage] [Ppage] [Handle]
XS	显示扩展内存状态	XS

附录 D　中 断 列 表

1．中断 INT 21H

(1) 字符功能调用类(Character - Oriented Function)

功能01H、07H **和** 08H

功能描述: 从标准输入设备(如键盘) 读入一个字符。该中断在处理过程中将一直处于等待状态, 直到有字符可读为止。该输入还可被重定向, 如果这样做, 则无法判断文件是否已到文件尾

入口参数: AH = 01H, 过滤掉控制字符, 回显
= 07H, 不过滤掉控制字符, 不回显
= 08H, 过滤掉控制字符, 不回显

出口参数: AL = 输入字符的 ASCII 码

功能02H

功能描述: 向标准输出设备(如屏幕) 输出一个字符。该输出还可被重定向, 如果这样做, 则将无法判断磁盘是否满

入口参数: AH = 02H
DL = 待输出字符的 ASCII 码

出口参数: 无

功能03H

功能描述: 从辅助设备读入一个字符, 该辅助设备的默认值为 COM1

入口参数: AH = 03H

出口参数: AL = 读入字符的 ASCII 码

功能04H

功能描述: 向辅助设备输出一个字符, 该辅助设备的默认值为 COM1

入口参数: AH = 04H
DL = 待输出字符的 ASCII 码

出口参数: 无

功能05H

功能描述: 向标准的输出设备输出一个字符。该默认的输出设备为 LPT1 端口的打印机, 除非用 MODE 命令来改变

入口参数: AH = 05H
DL = 待输出字符的 ASCII 码

出口参数: 无

功能06H

功能描述: 控制台(如键盘、屏幕) 输入/输出。如果输入/输出操作被重定向, 那么, 将无法判断文件是否已到文件尾或磁盘已满

入口参数: AH = 06H, DL = 输入/输出功能选择

出口参数: 若 DL = 00H - FEH, 则此功能为输出, DL 为待输出字符的 ASCII 码;
若 DL = 0FFH, 则此功能为输入, 此时: 若 ZF = 1, 则无字符可读, 否则, AL = 读入字符

的 ASCII 码

功能09H

功能描述: 输出一个字符串到标准输出设备上。如果输出操作被重定向,那么,将无法判断磁盘已满

入口参数: AH = 09H

DS: DX = 待输出字符的地址

说明: 待显示的字符串以 $ 作为其结束标志

出口参数: 无

功能0AH

功能描述: 从标准输入设备上读入一个字节字符串,遇到回车键结束输入(输入的字符在标准的输出设备上有回显) 。如果该输入操作被重定向,那么,将无法判断文件是否已到文件尾

入口参数: AH = 0AH

DS: DX = 存放输入字符的起始地址

接受输入字符串缓冲区的定义说明:

第一个字节为缓冲区的最大容量,可认为是入口参数。

第二个字节为实际输入的字符数(不包括回车键) ,可看作出口参数。

从第三个字节开始存放实际输入的字符串。

字符串以回车键结束,回车符是接受的最后一个字符。

若输入的字符数超过缓冲区的最大容量,则多出的部分被丢弃,系统发出响铃,直到按回车键才结束输入。

例如:

```
BUFF   80, ?, 80   DUP( ?)      ;最多接受 80 个字符
```

出口参数: 无

功能0BH

功能描述: 检查标准输入设备上是否有字符可读。该输入操作可被重定向

入口参数: AH = 0BH

出口参数: AL = 00H——无字符可读; FFH——有字符可读

功能0CH

功能描述: 清空当前的标准输入缓冲区,再读入字符。其输入操作可被重定向

入口参数: AH = 0CH

AL = 01H, 06H, 07H, 08H 或 0AH

出口参数: 若入口参数 AL 为 0AH, 则 DS: DX = 存放输入字符的起始地址, 否则, 出口参数 AL = 输入字符的 ASCII 码

(2) 目录控制功能(Directory - Control Function)

功能39H

功能描述: 用指定的驱动器和路径创建一个新目录

入口参数: AH = 39H

DS: DX = 指定路径的字符串地址(以 0 为字符串的结束标志)

出口参数: CF = 0——创建成功, 否则, AX = 错误号(03H 或 05H)

功能3AH

功能描述: 删除指定的驱动器和路径的目录

入口参数: AH = 3AH

DS: DX = 指定路径的字符串地址(以 0 为字符串的结束标志)

出口参数:　CF = 0——删除成功, 否则, AX = 错误号(03H 或 05H)

功能3BH

功能描述:　用指定的驱动器和路径设置当前目录

入口参数:　AH = 3BH

DS: DX = 指定路径的字符串地址(以 0 为字符串的结束标志)

出口参数:　CF = 0——设置成功, 否则, AX = 错误号(03H)

功能47H

功能描述:　取当前目录的完全路径字符串

入口参数:　AH = 47H

DL = 驱动器号(0 = 默认, 1 = A, . . .)

DS: SI = 存放当前目录字符串的地址

出口参数:　CF = 0——读取成功, 否则, AX = 错误号(0FH)

(3) 磁盘管理功能(Disk-Management Function)

功能0DH

功能描述:　清空当前的文件缓冲区, 但在 MS-DOS 内, 暂时写入缓冲区的数据将写入磁盘

入口参数:　AH = 0DH

出口参数:　无

功能0EH

功能描述:　指定当前驱动器

入口参数:　AH = 0EH

DL = 驱动器号(0 = A, 1 = B, . . .)

出口参数:　AL = 系统中当前的驱动器号

功能19H

功能描述:　取当前默认驱动器号

入口参数:　AH = 19H

出口参数:　AL = 驱动器号(0 = A, 1 = B, . . .)

功能1BH 和 1CH

功能描述:　获得驱动器的分配信息

入口参数:　AH = 1BH——为默认驱动器

AH = 1CH——为任意驱动器, DL = 驱动器号(0 = 默认, 1 = A, . . .)

出口参数:　AL = 0FFH——失败, 否则,

AL = 每簇的扇区数

DS: BX = ID 字节的地址

CX = 物理扇区的大小(字节数)

DX = 驱动器的簇数

功能2EH

功能描述:　设置/清除操作系统自动读取检验标志

入口参数:　AH = 2EH

DL = 00H

AL = 00H——清除该标志, 01H——设置该标志

出口参数:　无

功能36H

功能描述:　取选定驱动器的信息

入口参数:　AH = 36H

DL = 驱动器号(0 - 默认, 1 = A, 2 = B, . . .)

出口参数: 若功　调用失败, AX = 0FFFFH, 否则,

AX = 每簇的扇区数

BX = 可用的簇数

CX = 物理扇区的大小(字节数)

DX = 驱动器中的簇数

功能54H

功能描述:　读取校验标志

入口参数:　AH = 54H

出口参数:　AL = 当前检验标志值: 00H - 关检验、01H - 开检验

(4) 文件操作功能(File Operation Function)

功能3CH

功能描述:　用指定的文件名创建一个新文件。如果指定的文件已存在, 则设置其长度为 0。创建后, 该文件是打开的并返回其句柄

入口参数:　AH = 3CH

DS: DX = 指定文件名字符串的地址(以 0 为字符串的结束标志)

CX = 文件属性(这些标志位可以组合)

位 0 = 1——只读　　位 3 = 1——卷标号

位 2 = 1——系统　　位 1 = 1——隐含

位 5 = 1——归档　　其他位保留不用并置为 0

出口参数:　CF = 0——创建成功, AX = 文件句柄, 否则, AX = 错误号(03H, 04H 或 05H)

功能3DH

功能描述:　打开指定的驱动器、路径和文件名并返回其文件句柄

入口参数:　AH = 3DH

DS: DX = 表明文件的字符串(以 0 为字符串的结束标志)

AL 为打开方式:

位 0 ~2000——只读方式　　001——写方式　　010——读/写方式

位 3 保留, 其值为 0

位4 ~6 共享模式

000——兼容模式　　001——不共享　　010——拒绝写

011——拒绝读　　100——不拒绝任何操作

位 7 继承标志——0 /1: 子进程继承或不继承句柄

出口参数:　CF = 0——打开成功, AX = 文件句柄, 否则, AX = 错误号(02H, 03H, 04H, 05H 或 0CH)

功能3EH

功能描述:　关闭指定句柄的文件

入口参数:　AH = 3EH

BX = 文件句柄

出口参数:　CF = 0——关闭成功, 否则, AX = 错误号(06H)

功能41H

功能描述:　删除指定的文件

入口参数: AH = 41H

DS: DX = 文件名字符串的地址

出口参数: CF = 0——删除成功, 否则, AX = 错误号(02H, 03H 或 05H)

功能43H

功能描述: 读取或设置指定文件的属性

入口参数: AH = 43H

BX = 文件句柄

DS: DX = 文件名字符串的地址

AL = 00H /01H——读取 /设置文件属性

CX = 文件属性:

位 0 = 1——只读　　位 1 = 1——隐含

位 2 = 1——系统　　位 3 = 1——卷标号

位 5 = 1——归档　　其他位保留不用, 并置为 0

出口参数: CF = 0——关闭成功, CX = 文件属性, 否则, AX = 错误号(01H, 02H, 03H 或 05H)

功能45H

功能描述: 复制当前打开设备或文件的句柄, 该句柄对应同样设备或文件的相同位置

入口参数: AH = 45H

BX = 待复制的文件句柄

出口参数: CF = 0——复制成功, AX = 新句柄, 否则, AX = 错误号(04H 或 06H) , 其含义见错误代码表

功能46H

功能描述: 指定两个句柄, 把第二个句柄指向第一个句柄, 即第二个句柄被重定向

入口参数: AH = 46H

BX = 文件或设备的句柄

CX = 待重定向的文件句柄

出口参数: CF = 0——重定向成功, 否则, AX = 错误号(04H 或 06H)

功能4EH

功能描述: 获取第一个与给定的文件名相匹配的文件

入口参数: AH = 4EH

DS: DX = 给定文件名的字符串

CX = 搜索时使用的文件属性:

位 0 = 1——只读　　位 1 = 1——隐含

位 2 = 1——系统　　位 3 = 1——卷标号

位 4 = 1——目录　　位 5 = 1——归档

其他位保留不用, 并置为 0

出口参数: CF = 1——操作失败, AX = 错误号(02H, 03H 或 12H) , 否则, 操作成功, DTA(Disk Transfer Area) 按下列方式填入数据:

字节 00 ~14H　　保留

字节 15H　　匹配的文件属性

字节 16 ~17H　　压缩的文件名

字节 18 ~19H　　压缩的文件日期

字节 1A ~1DH　　文件大小

字节 1E ~2AH　　文件名字符串

功能4FH

功能描述:　在中断 21H 的功能 4EH 成功使用之后,再搜索下一个文件名

入口参数:　AH = 4FH

AL = 返回的代码

出口参数:　CF = 1——操作失败, AX = 错误号(12H) , 否则, 操作成功, DTA 中的数据如前面功能 4EH 所示

功能56H

功能描述:　文件换名

入口参数:　AH = 56H

DS: DX = 当前文件名字符串地址

ES: DI = 新文件名字符串地址

出口参数:　CF = 0——操作成功, 否则, AX = 错误号(02H, 03H, 05H, 11H)

功能57H

功能描述:　读取/设置文件的日期和时间

入口参数:　AH = 57H

BX = 文件句柄

读取日期和时间　AL = 00H

设置日期和时间　AL = 01H

CX = 时间(0F ~0BH: 小时, 0AH ~05H: 分钟, 04H ~00H: 2 秒的个数)

DX = 日期(0F ~09H: 年(相对 1980 年) , 08H ~05H: 月, 04H ~00H: 日)

出口参数:　CF = 1——操作失败, AX = 错误号(01H, 06H) , 否则, 若是读文件信息, 则, CX = 时间, DX = 日期

功能5AH

功能描述:　创建临时文件

入口参数:　AH = 5AH

DS: DX = 路径名的地址

CX = 文件属性(位可组合) , 其定义如下:

位 0 = 1	只读	位 3 - 4 = 0	保留
位 1 = 1	隐含	位 5 = 1	归档
位 2 = 1	系统	位 6 - 15 = 0	保留

出口参数:　CF = 0——操作成功, AX = 文件句柄, DS: DX = 完整的路径文件地址, 否则, AX = 错误号(03H, 04H 或 05H)

功能5BH

功能描述:　创建新文件

入口参数:　AH = 5BH

DS: DX = 路径名的地址

CX = 文件属性(位可组合) , 其定义如下:

位 0 = 1	只读	位 4 = 0	保留
位 1 = 1	隐含	位 5 = 1	归档
位 2 = 1	系统	位 6 - 15 = 0	保留
位 3 = 1	卷标号		

出口参数: CF = 0——操作成功, AX = 文件句柄, 否则, AX = 错误号(03H, 04H, 05H 或 50H) , 其含义见错误代码表

功能67H

功能描述: 设置文件句柄数(最多文件数)

入口参数: AH = 67H

BX = 句柄的数量

出口参数: CF = 0——操作成功, 否则, AX = 错误号

功能6CH

功能描述: 扩展的打开文件功能(打开、创建或替换文件)

入口参数: AH = 6CH

AL = 00H

DS: SI = 路径名的地址

BX = 打开方式

位 2～0　000——只读　001——只写　010——可读、写位 3　保留(0)

位 6～4　000——兼容　001——拒绝读写　010——拒绝写

011——拒绝读　100——不拒绝任何操作

位 7　0——子进程继承句柄, 1——子进程不继承句柄

位 12～8　保留(0)

位 13　致命错误处理程序, 0——执行 INT 24H, 否则, 返回错误代码给进程

位 14　写入方式: 0——写入缓冲区, 1——直接写入文件

位 15　保留(0)

CX = 文件属性

位 0 = 1	只读	位 4 = 0	保留
位 1 = 1	隐含	位 5 = 1	归档
位 2 = 1	系统	位 6 - 15 = 0	保留
位 3 = 1	卷标签		

DX = 打开标志

位 3～0　0——打开失败, 1——打开文件, 2——替换文件

位 7～4　0——打开失败, 1——创建文件

位 15～8　0——保留

出口参数: CF = 1——操作失败, AX = 错误号, 否则, AX = 文件句柄

CX = 1——文件存在, 打开文件

= 2——文件不存在, 创建文件

(5) 文件操作功能文件控制块(File Operation Function)

功能0FH

功能描述: 打开文件并使之为顺序读/写做好准备

入口参数: AH = 0FH

DS: DX = 文件控制块的地址

出口参数: AL = 00H——打开成功, 否则, AL = FFH(如文件找不到)

在 MS-DOS 操作系统中, 文件控制块的字段如下表所示。

字段名	偏移量	字段含义
驱动器字	00H	1 for drive A, 2 for drive B, . . .
当前块字段	0CH	00H
记录大小字段	0EH	0080H
文件长度字段	10H	文件字节数
日期字段	14H	日期
时间字段	16H	时间

功能10H

功能描述:　关闭文件

入口参数:　AH = 10H

DS: DX = 文件控制块的地址

出口参数:　AL = 00H——关闭成功, 否则, AL = FFH

功能11H 和 12H

功能描述:　查找第一个或下一个相匹配的文件

入口参数:　AH = 11H——第一个相匹配的文件

= 12H——下一个相匹配的文件

DS: DX = 文件控制块的地址

出口参数:　AL = 00H——查找到, 否则, AL = FFH

功能13H

功能描述:　在指定(或默认) 的驱动器中, 删除所有相匹配的文件

入口参数:　AH = 13H

DS: DX = 文件控制块的地址

出口参数:　AL = 00H——删除成功, 否则, AL = FFH

功能16H

功能描述:　在当前目录中创建一个文件, 其文件长度为 0。打开该文件, 为随后的读/写操作做好必要的准备

入口参数:　AH = 16H

DS: DX = 未打开的文件控制块的地址

出口参数:　AL = 00H——创建成功, 否则, AL = FFH(如磁盘满)

功能 17H

功能描述:　在指定的驱动器的当前目录中, 把所有相匹配的文件换名

入口参数:　AH = 17H

DS: DX = 指定文件控制块的地址　口参数:　AL = 00H——换名成功, 否则, AL = FFH

功能23H

功能描述:　在当前目录中查找一个相匹配的文件。如果发现, 则用其记录数来更新其文件大小

入口参数:　AH = 23H

DS: DX = 未打开的文件控制块的地址

出口参数:　AL = 00H——匹配成功, FCB 中偏移量为 21H 的字段被设置为其记录数, 否则, AL = 0FFH

功能29H

功能描述:　分析一个字符串(文件名) 置入 FCB 表中的不同字段

入口参数: AH = 29H

CX = 要写入的记录数

DS: SI = 字符串段的地址

ES: DI = FCB 的地址

AL = 分析的控制标志位

位 3 = 1——若字符串中有文件后缀, 则 FCB 中的文件后缀将改变

= 0——若后缀忽略修改, 或若分析后无后缀, 则 FCB 中后缀字段被置为“空”

位 2 = 1——若字符串中有文件名, 则 FCB 中的文件名将改变

= 0——若文件名忽略修改或分析后无文件名, 则 FCB 中文件名字段被置为“空”

位 1 = 1——若字符串中指定了驱动器号, 则 FCB 中的 ID 字节被修改

= 0——若 ID 字节忽略修改或分析后没有指定驱动器号, 则 FCB 中驱动器字段被置为 0(默认值)

位 0 = 1——忽略前导分割符

= 0——不忽略前导分割符

出口参数: AL = 00H——没有通配字符　AL = 01H——有通配字符　AL = FFH——驱动器号非法

DS: SI = 分析后文件名第一个字符的地址

ES: DI = 格式化后的、未打开的 FCB 地址

(6) 记录功能(Record Function)

功能1AH

功能描述: 设置磁盘传送数据区地址, 为随后 FCB 的相关操作做准备

入口参数: AH = 1AH

DS: DX = 指定文件控制块的地址

出口参数: 无

功能2FH

功能描述: 为 FCB 读/写操作而获取 DTA 的当前地址

入口参数: AH = 2FH

出口参数: ES: BX = DTA 的段地址和偏移量

功能3FH

功能描述: 从先前打开的文件中读出指定数目的字节并移动文件指针

入口参数: AH = 3FH

BX = 文件句柄

CX = 将要读出的字节数

DS: DX = 存放字符的缓冲区地址

出口参数: CF = 0——读取成功, AX = 读取的字符数, 否则, AX = 错误号(05H 或 06H)

功能40H

功能描述: 向先前打开的文件写入指定数量的字节, 并修改相应的文件指针

入口参数: AH = 40H

BX = 文件句柄

CX = 写入的字节数

DS: DX = 存放数据的缓冲区地址

出口参数: CF = 0——关闭成功, AX = 写入的字节数, 否则, AX = 错误号(05H 或 06H), 其含义见错误代码表

功能42H

功能描述:　设置文件指针的相对位置(相对与文件头、文件尾和当前位置)

入口参数:　AH = 42H

BX = 文件句柄

CX = 偏移量的高位

DX = 偏移量的低位

AL = 00H——从文件头开始的绝对偏移量

= 01H——从当前文件指针开始的偏移量(可带符号)

= 02H——从文件尾开始的偏移量(可带符号)

出口参数:　CF = 0——设置成功, DX 是指针的高位, AX 是其低位, 否则, AX = 错误号(01H 和 06H)

功能5CH

功能描述:　文件区域加锁或解锁

入口参数:　AH = 5CH

AL = 00H——区域加锁　　01H——区域解锁

BX = 文件句柄

CX: DX = 区域偏移量

SI: DI = 区域长度

DS: DX = 路径名的地址

出口参数:　CF = 0——操作成功, 否则, AX = 错误号(01H、06H、21H 或 24H)

功能68H

功能描述:　提交文件缓冲区数据

入口参数:　AH = 68H

BX = 文件句柄

出口参数:　CF = 0——操作成功, 否则, AX = 错误号, 其含义见错误代码表

(7) 记录功能文件控制块(Record Function)

功能14H

功能描述:　从文件中读出下一个顺序块, 并相应增加文件的指针

入口参数:　AH = 14H

DS: DX = 先前打开文件控制块的地址

出口参数:　AL = 00H——读取成功　　01H——文件尾

02H——段缠绕　　03H——部分记录在文件尾

功能15H

功能描述:　向文件写入下一个顺序数据块并相应增加文件的指针

入口参数:　AH = 15H

DS: DX = 先前打开文件控制块的地址

出口参数:　AL = 00H——写入成功　01H——磁盘满　02H——段缠绕

功能21H

功能描述:　从文件中读出当前选定的记录

入口参数:　AH = 21H

DS: DX = 先前打开文件控制块的地址

出口参数:　AL = 00H——读取成功　　01H——文件尾

02H——取消读操作　　03H——部分记录在文件尾

功能22H

功能描述:　**把内存中的数据写入到文件中当前选定的记录**

入口参数:　AH = 22H

DS: DX = **先前打开文件控制块的地址**

出口参数:　AL = 00H—**写入成功**　01H—**磁盘满**　02H—**取消写操作**

功能24H

功能描述:　**设置** FCB **中相对记录数作为被打开** FCB **中的记录数**

入口参数:　AH = 24H

DS: DX = **先前打开文件控制块的地址**

出口参数:　AL **的值被破坏, 其他寄存器不受影响,** FCB **中偏移量** 21H **单元被修改**

功能27H

功能描述:　**从文件中读出若干个记录到内存中**

入口参数:　AH = 27H

DS: DX = **先前打开文件控制块的地址**

出口参数:　AL = 00H——**读取成功**　01H——**文件尾**

02H——**取消读操作**　03H——**部分记录在文件尾**

CX = **实际读出的记录数**

功能28H

功能描述:　**从内存向文件中写入若干个记录**

入口参数:　AH = 28H

CX = **要写入的记录数**

DS: DX = **先前打开文件控制块的地址**

出口参数:　AL = 00H——**写入成功**　01H——**磁盘满**　02H——**段缠绕**

CX = **实际写入的记录数**

(8) **内存分配功能**(Memory-Allocation Function)

功能48H

功能描述:　**分配一块内存单元并返回该块内存单元的首地址**

入口参数:　AH = 48H

BX = **需要申请的内存单元字节数**

出口参数:　CF = 0——**分配成功**, AX = **存储单元的首地址**, **否则**, AX = **错误号**(07H **或** 08H), BX = **还可用的最大块数**

功能49H

功能描述:　**释放内存单元块以便为其他程序使用**

入口参数:　AH = 49H

ES = **被申请块的段地址**

出口参数:　CF = 0——**释放成功**, **否则**, AX = **错误号**(07H **或** 09H)

功能4AH

功能描述:　**根据程序的需要, 动态地改变一个内存块**

入口参数:　AH = 4AH

BX = **需要一个新存储块的大小**

ES = **被修改块的段地址**

出口参数:　CF = 0——**修改成功**, **否则**, AX = **错误号**(07H、08H **或** 09H), **其含义见错误代码表**

BX = **可用最大块的大小**

功能58H

功能描述:　**读取/设置内存分配策略**

入口参数:　AH = 58H

读取内存分配策略　AL = 00H

设置内存分配策略　AL = 01H

BX = **内存分配策略代码**:

00H——**第一满足**

01H——**最好满足**

02H——**最后满足**

出口参数:　CF = 0——**操作成功**, AX = **已选用的内存分配策略代码**(**含义如上说明**), **否则**, AX = **错误号**(01H)

(9) **系统功能**(System Function)

功能25H

功能描述:　**设置中断向量表**

入口参数:　AH = **中断号**

DS: DX = **中断处理程序的入口地址**

出口参数:　**无**

功能30H

功能描述:　**取** MS-DOS **操作系统的版本号**

入口参数:　AH = 30H

出口参数:　AL = 0——V 1. 0; **对其他高版本有**: AL = **主要版本号**

AH = **次版本号**(MS-DOS 3. 1 = 0AH, . . .)

BH = OEM **的序列号**(Original Equipment Manufacturer)

BL: CX = 24 **位用户序列号**

功能33H

功能描述:　**获得或改变操作系统中断的状态。在功能调用期间, 将影响**^C **的检测**

入口参数:　AH = 33H

AL = 00H——**取状态**

= 01H——**设置状态**, DL = 00 /01 **表示置该状态** OFF / ON

出口参数:　**取状态时**, DL = 00 /01——**分别表示** OFF / ON

说明:

若 AL **中存入其他的功能号, 则返回时**, AL **的值为** 0FFH

若置 AL **为** 5, **则启动驱动器号返回在** DL **中**(1 - A, 2 - B, . . .)

功能34H

功能描述:

获得 InDos **标志的远地址, 它由** DOS **系统维护, 表示** DOS **是否处于活跃状态**

入口参数:　AH = 34H

出口参数:　ES: BX = InDos **标志的远地址**

若该单元值为 1, **表示** DOS **功能在执行, 否则, 则不是**

功能35H

功能描述:　**取指定中断号的入口地址**

入口参数:　AH = 35H

AL = **中断号**

出口参数: ES: BX = 中断处理程序的入口地址

功能38H

功能描述: 读取或设置国家信息

入口参数: AH = 38H

当读　国家信息时

DS: DX = 存放返回信息的地址

AL = 0——取当前国家信息

AL = 1～0FEH——取国家代码小于 255 的国家信息

AL = 0FFH——取代码大于等于 255 的国家信息, BX = 国家代码

设置　家信息时

DX = 0FFFFH

AL = 0～0FEH——设置国家代码小于 255 的国家信息

AL = 0FFH——置代码大于等于 255 的国家信息, BX = 国家代码

出口参数: CF = 0——调用成功, BX = 国家代码, 否则, AX = 错误代码(02H)

说明国家信息如下所示:

字节 0～1H:　日期格式: 0—mdy、1—dmy、2—ymd

字节 2～6H:　货币字符

字节 7～8H:　数值千位分割符

字节 9～0AH:　数值精度分割符

字节 0B～0CH:　日期间隔符

字节 0D～0EH:　时间间隔符

字节 0FH:　货币格式

位 0 = 0——货币符号在前, 否则, 货币符号在后

位 1 = 0——货币符号和数据之间无空格, 否则, 二者之间有一个空格

位 2 = 0——货币符号和小数点分开, 否则, 货币符号代替小数点

字节 10H:　货币的小数位数

字节 11H:　时间格式。位 0 = 0——12 小时制, 否则, 24 小时制

字节 12～15H:　Case - Map 调用地址

字节 16～17H:　字符串分割符

字节 18～21H:　保留

功能44H

功能描述: 输入/输出控制, 其子功能描述:

00H——取设备信息　　01H——取设备信息

02H——从字符设备驱动器接受控制数据　03H——发送控制数据到字符设备驱动器

04H——从块设备驱动器接受控制数据　05H——发送控制数据到块设备驱动器

06H——检查输入状态　　07H——检查输出状态

08H——检查块设备是否为可拆卸设备　09H——检查设备是否为远程设备

0AH——检查句柄是否为远程对象　　0BH——改变共享访问入口数

0CH——字符设备的一般 I/O 控制信息　0DH——块设备的一般 I/O 控制信息

0EH——读取逻辑驱动器映射关系　　0FH——设置逻辑驱动器映射关系

说明: 输入/输出子功能中的 00H, 06H 和 07H 仅针对文件句柄, 子功能 00H～08H 不支持网络设备

功能50H

功能描述:　设置程序段前缀(PSP) 地址

入口参数:　AH = 50H

BX = 新的 PSP 地址

出口参数:　无

功能51H

功能描述:　读取程序段前缀(PSP) 地址

入口参数:　AH = 51H

出口参数:　BX = PSP 地址

功能59H

功能描述:　读取扩展的错误信息

入口参数:　AH = 59H

BX = 00H

出口参数:　AX = 扩展的错误代码, 其含义见错误代码表

BH = 错误类型, 其定义如下:

01h——资源短缺　　02h——处于临时状态而非错误

03h——权限问题　　04h——系统软件内部错误

05h——硬件失败　　06h——系统软件失败, 但不是活跃进程失败

07h——应用程序错　　08h——文件或数据项未发现

09h——文件或数据项类型或格式错　0Ah—文件或数据项相互加锁

0Bh——驱动器中坏磁盘、磁盘中坏区域或存储问题

0Ch——其他错误

BL = 建议采用的措施, 其定义如下:

01h —— 重试若干次后, 再选“ 终止 ”或“ 忽略 ”

02h —— 重试若干次(二次之间要等待) 后, 再选“ 终止 ”或“ 忽略 ”

03h —— 从用户获取正确的信息

04h —— 终止应用程序并清除其所使用资源

05h —— 立即终止程序, 但没有清除其资源

06h —— 忽略错误

07h —— 消除错误原因, 再重试

CH = 错误地点, 其定义如下:

01h —— 不知道

02h —— 块设备(磁盘或磁盘模拟器)

03h —— 网络

04h —— 串行设备

05h —— 内存

ES: DI = 插入磁盘标签的字符串, 若 AX = 0022h 表示非法改变磁盘

功能5EH

功能描述:　读取机器名, 读取/设置打印机配置

子功能号	功能描述
00h	读取机器名
02h	设置打印机安装字符串
03h	读取打印机安装字符串

① 子　能 1

入口参数: AH = 5EH
AL = 00H
DS: DX = 接受字符串缓冲区的地址
出口参数: CF = 1——操作失败, AX = 错误号(01H) , 否则, 判断 CH
CH = 00H——机器名未定义, 否则, 机器名已定义
CL = NetBIOS 名称号(当 CH ≠00H 时)
DS: DX = 标识符地址(当 CH ≠00H 时)

② 子　能 2

入口参数: AH = 5EH
AL = 02H
BX = 重定向列表索引
CX = 安装字符串的长度
DS: SI = 安装字符串的地址
出口参数: CF = 0——操作成功, 否则, AX = 错误号(01H)

③ 子　能 3

入口参数: AH = 5EH
AL = 03H
BX = 重定向列表索引
ES: DI = 接受字符串缓冲区的地址
出口参数: CF = 0——操作成功, CX = 接受字符串的长度, 否则, AX = 错误号(01H) , 其含义见错误代码表

功能5FH

功能描述: 设备重定向

子功能号	功能描述
02h	读取重定向列表索引
03h	重定向设备

① 子　能 1

入口参数: AH = 5FH
AL = 02H
BX = 重定向列表索引
DS: SI = 接受本地设备名的 16 字节存储区地址
ES: DI = 接受网络名的 128 字节存储区地址
出口参数: CF = 1——操作失败, AX = 错误号(01H 或 12H) , 否则, 判断 BH 的第 0 位, 位 0 = 0H——设备合法, 否则, 设备非法
BL = 设备类型——03H: 打印机, 04H: 驱动器
CX = 存储参数值
DX = 被破坏
BP = 被破坏
DS: SI = 存放本地设备名的地址
ES: DI = 存放网络名的地址

② 子　能 2

入口参数: AH = 5FH
AL = 03H

BL = 设备类型——03H: 打印机, 04H: 驱动器

CX = 调用者保存的参数

DS: SI = 本地设备名的 16 字节存储区地址

ES: DI = 网络名的 128 字节存储区地址, 紧跟其后是密码

出口参数: CF = 0——操作成功, 否则, AX = 错误号(01H, 03H, 05H, 08H, 0FH 或 12H)

功能63H

功能描述: 读取前导字节表

入口参数: AH = 63H

AL = 子功能

= 00H——读取系统前导字节表地址

= 01H——设置/清除临时控制台标志(DL = 00H/01H——清除/设置标志)

= 02H——读取临时控制台标志值

出口参数: BX = 1——操作失败, AX = 错误号(01H) , 否则, 调用时,

若 AL = 00H, 则, DS: SI = 系统前导字节表地址

若 AL = 02H, 则, DL = 临时控制台标志值

功能65H

功能描述: 读取扩展的国家信息

入口参数: AH = 65H

BX = 代码页(- 1 = 活跃的 CON 设备)

CX = 接受信息的缓冲区大小

DX = 国家标识(- 1 = 默认)

ES: DI = 接受信息的缓冲区地址

AL = 子功能

= 01H——读取一般的国家信息

= 02H——读取指向大写字母表的指针

= 04H——读取指向文件名大写字母表的指针

= 06H——读取指向校对表的指针

= 07H——读取指向 DBCS 向量的指针

出口参数: CF = 0——操作成功, 需要的数据存入调用的缓冲区, 否则, AX = 错误号(02H)

功能66H

功能描述: 读取/设置代码页

入口参数: AH = 66H

AL = 子功能号: 01H——读取代码页, 02H——选择代码页

BX = 选择的代码页(当 AL = 02H)

出口参数: CF = 0——操作成功, 当调用子功能 01H 时, BX = 活跃的代码页, DX = 默认的代码页, 否则, AX = 错误号(02H 或 65H) ,

功能5D0AH

功能描述: 设置扩展的错误信息

入口参数: AX = 5D0AH

DS: DX = 扩展错误结构的地址, 该结构的说明如下:

```
EXTEND_ERR    STRUCT
                RAx     WORD    ?       ; AX
                RBx     WORD    ?       ; BX
```

```
RCx     WORD    ?       ; CX
RDx     WORD    ?       ; DX
RSi     WORD    ?       ; SI
RDi     WORD    ?       ; DI
RDs     WORD    ?       ; DS
REs     WORD    ?       ; ES
Pad     WORD    3       DUP(0)
EXTEND_ERR ENDS
```

出口参数: 无

(10) **进程控制功能**(Process - Control Function)

功能00H

功能描述: 终止进程,这是程序可以使用的终止进程的方法之一

入口参数: AH = 00H

CS = **代码段地址**

出口参数: 无

功能26H

功能描述: 把当前正在执行程序的程序段前缀(PSP)复制到内存的指定地址中,并可对其改变为其他程序所使用

入口参数: AH = 26H

DX = **新程序段前缀的段地址**

出口参数: 无

功能31H

功能描述: 终止程序的运行,传递一个返回代码给其父进程,但该程序部分或全部驻留在内存中

入口参数: AH = 31H

AL = **返回代码号**

DX = **驻留在内存中的字节数**

出口参数: 无

功能4BH

功能描述: 执行程序(EXEC)

入口参数: AH = 4BH

ES: BX = **参数块的地址**

DS: DX = **程序的入口地址**

AL = 00H——**装入并执行程序**, 03H——**以覆盖的形式装入**

出口参数: CF = 0——**操作成功**,**除** CS **和** IP **之外**,**其他寄存器的值都被破坏**,**否则**, AX = **错误号**(01H, 02H, 03H, 05H, 08H, 0AH **或** 0BH)

功能4CH

功能描述: 终止程序的执行并可返回一个代码

入口参数: AH = 4CH

AL = **返回的代码**

出口参数: 无

功能4DH

功能描述: 父进程获取子进程的返回代码

入口参数:　AH = 4DH

出口参数:　AH = 00H——用中断 20H、中断 21H 的功能 00H 或 4CH 正常终止

= 01H——用户按^C 终止

= 02H——因致命错误而终止

= 03H——用中断 21H 的功能 31H 或中断 27H 终止

AL = 子进程的返回码: 00H——子进程由中断 20H、中断 21H 的功能 0 或 4CH 终止

功能62H

功能描述:　读取 PSP 地址

入口参数:　AH = 62H

出口参数:　BX = PSP 的偏移量

(11) 时间和日期功能(Time and Date Function)

功能2AH

功能描述:　取系统日期

入口参数:　AH = 2AH

出口参数:　CX = 年(1980～2099), DH = 月(1～12), DL = 日(1～31)

AL = 星期几(0 = Sunday, 1 = Monday, . . .)

功能2BH

功能描述:　置系统日期

入口参数:　AH = 2BH

CX = 年(1980 - 2099), DH = 月(1～12), DL = 日(1～31)

出口参数:　AL = 00H——设置成功, 0FFH——设置失败

功能2CH

功能描述:　取系统时间

入口参数:　AH = 2CH

出口参数:　CH = 时(0～23), CL = 分(0～59), DL = 秒(0～59), AL = 百分秒(0～99)

功能2DH

功能描述:　置系统时间

入口参数:　AH = 2DH

CH = 时(0～23), CL = 分(0～59), DL = 秒(0～59), AL = 百分秒(0～99)

出口参数:　出口参数: AL = 00H——设置成功, 0FFH——设置失败

2. 鼠标功能中断 INT 33H

功能00H

功能描述:　初始化鼠标, 该操作只需要执行一次

入口参数:　AX = 00H

出口参数:　AX = 0000H——不支持鼠标功能, FFFFH——支持鼠标功能

BX = 鼠标按钮个数(在支持鼠标功能时)

在支持鼠标功能的情况下, 鼠标还可设置如下参数:

鼠标指针放在屏幕中央

如果当前鼠标指针是显示的, 则操作后, 鼠标指针被隐藏

鼠标指针的显示页为 0

根据屏幕的显示模式显示鼠标指针: 文本——反向显示矩形块, 图形——尖头形状

水平像素比 = 8: 8, 垂直像素比 = 16: 8
设置水平和垂直的显示边界为当前显示模式的最大边界
允许光笔仿真
双速门槛值 = 64

功能01H

功能描述: 显示鼠标指针, 通常在鼠标初始化后, 用此功能显示其指针
入口参数: AX = 01H
出口参数: 无

功能02H

功能描述: 隐藏鼠标指针, 一般在程序结束时, 调用此功能
入口参数: AX = 02H
出口参数: 无

功能03H

功能描述: 读取鼠标位置及其按钮状态
入口参数: AX = 03H
出口参数: BX = 按键状态: 位 0 = 1——按下左键
位 1 = 1——按下右键
位 2 = 1——按下中键
其他位——保留, 内部使用
CX = 水平位置
DX = 垂直位置

功能04H

功能描述: 设置鼠标指针位置
入口参数: AX = 04H, CX = 水平位置, DX = 垂直位置
出口参数: 无

功能05H

功能描述: 读取鼠标按键信息
入口参数: AX = 05H, BX = 指定的按键: 0——左键, 1——右键, 2——中键
出口参数: AX = 按键状态, 参见功能 03H 中 BX 的说明
BX = 按键次数
CX = 水平位置(最后按键时)
DX = 垂直位置(最后按键时)

功能06H

功能描述: 读取鼠标按钮释放信息
入口参数: AX = 06H, BX = 指定的按键: 0——左键, 1——右键, 2——中键
出口参数: AX = 按键状态, 参见功能 03H 中 BX 的说明
BX = 释放的次数
CX = 水平位置(最后释放时)
DX = 垂直位置(最后释放时)

功能07H

功能描述: 设置鼠标水平边界
入口参数: AX = 07H
CX = 最小水平位置

DX = 最大水平位置

出口参数:　无,鼠标有可能因新区域变小而自动移进新区域内

功能08H

功能描述:　设置鼠标垂直边界

入口参数:　AX = 08H

CX = 最小垂直位置

DX = 最大垂直位置

出口参数:　无,鼠标有可能因新区域变小而自动移进新区域内

功能09H

功能描述:　设置图形区鼠标形状

入口参数:　AX = 09H

BX = 指针的水平位置

CX = 指针的垂直位置

ES: DX = 16× 16 位光标的映像地址

参数说明:

(BX, CX) 是鼠标的指针在 16× 16 点阵中的位置, (0, 0) 是左上角

ES: DX 指向的存储单元内存放 16× 16 点阵的位映像隐码,紧跟其后的是 16× 16 点阵的光标掩码

鼠标指针的　示方法:

位映像隐码“ 逻辑与 ”上屏幕显示区的内容,然后再用光标掩码内容“ 异或 ”前面运算的结果

出口参数:　无

功能0AH

功能描述:　设置本文区鼠标形状

入口参数:　AX = 0AH

BX = 光标类型:

0——CX 和 DX 的各位含义如下:

位 7～0　　鼠标指针符号

位 10～8　字符前景色

位 11　　亮度

位 14～12　字符背景色

位 15　　闪烁

1—　X = 光标的起始扫描线

DX = 光标的结束扫描线

出口参数:　无

功能0BH

功能描述:　读取鼠标移动计数

入口参数:　AX = 0BH

出口参数:　CX = 水平移动距离: 正数——向右移, 负数——向左移

DX = 垂直移动距离: 正数——向下移, 负数——向上移

功能0CH

功能描述:　为鼠标事件设置处理程序

入口参数:　AX = 0CH

CX = 中断掩码

位 0 = 1——鼠标指针位置发生变化

位 1 = 1——按下左按钮

位 2 = 1——释放左按钮

位 3 = 1——按下右按钮

位 4 = 1——释放右按钮

位 5 = 1——按下中间按钮

位 6 = 1——释放中间按钮

位 7～15 = 0——保留

ES: DX = 中断处理程序的地址

在进　中断处理程序时, 有关寄存器值的含义:

AX = 中断掩码

BX = 按键状态

CX = 鼠标指针的水平位置

DX = 鼠标指针的垂直位置

SI = 水平位置的变化量

DI = 垂直位置的变化量

出口参数:　无

功能0DH

功能描述:　允许光笔仿真

入口参数:　AX = 0DH

出口参数:　无

功能0EH

功能描述:　关闭光笔仿真

入口参数:　AX = 0EH

出口参数:　无

功能0FH

功能描述:　设置鼠标计数与像素比

入口参数:　AX = 0FH

CX = 水平比例

DX = 垂直比例

出口参数:　无

功能10H

功能描述:　设置鼠标指针隐藏区域

入口参数:　AX = 10H

CX = 左上角 X 坐标

DX = 左上角 Y 坐标

SI = 右下角 X 坐标

DI = 右下角 Y 坐标

出口参数:　无

功能13H

功能描述:　设置倍速的阈值, 其默认值为 64

入口参数:　AX = 13H

DX = 阀值

出口参数:　无

功能14H

功能描述:　替换鼠标事件中断

入口参数:　AX = 14H

CX = 中断掩码

ES: DX = 中断处理程序的地址

出口参数:　CX = 旧的中断掩码

ES: DX = 旧的中断处理程序地址

功能15H

功能描述:　读取鼠标驱动器状态的缓冲区大小

入口参数:　AX = 15H

出口参数:　BX = 存放鼠标驱动器状态所需缓冲区的大小

功能16H

功能描述:　存储鼠标驱动器状态

入口参数:　AX = 16H

ES: DX = 存储鼠标驱动器状态的地址

出口参数:　无

功能17H

功能描述:　重装鼠标驱动器状态

入口参数:　AX = 17H

ES: DX = 鼠标驱动器状态的地址

出口参数:　无

功能18H

功能描述:　为鼠标事件设置可选的处理程序

入口参数:　AX = 18H

CX = 替换中断掩码

ES: DX = 替换中断处理程序的地址

CF = 0

出口参数:　无

功能19H

功能描述:　读取替换处理程序的地址

入口参数:　AX = 19H

CX = 替换中断掩码

出口参数:　若 AX = - 1——不成功, 否则, ES: DX = 中断处理程序的地址

功能1AH

功能描述:　设置鼠标的灵敏度, 其取值 1 ~100

入口参数:　AX = 1AH

BX = 水平灵敏度(每 8 个像素鼠标需要移动的数量, 一般为 8)

CX = 垂直灵敏度(每 8 个像素鼠标需要移动的数量, 一般为 16)

DX = 倍速阀值

出口参数:　无

功能1BH

功能描述: 读取鼠标的灵敏度

入口参数: AX = 1BH

出口参数: BX = 水平灵敏度

CX = 垂直灵敏度

DX = 倍速阀值

功能1CH

功能描述: 设置鼠标中断速率

入口参数: AX = 1CH

BX = 每秒钟中断的次数: 0——关中断, 1～30/s, 2～50/s, 3～100/s, 4～200/s

出口参数: 无

功能1DH

功能描述: 为鼠标指针选择显示页

入口参数: AX = 1DH

BX = 显示页

出口参数: 无

功能1EH

功能描述: 读取鼠标指针的显示页

入口参数: AX = 1EH

出口参数: BX = 显示页

功能1FH

功能描述: 禁止鼠标驱动程序

入口参数: AX = 1FH

出口参数: 若 AX = -1——不成功, 否则, ES: BX = 鼠标驱动程序的地址

功能20H

功能描述: 启动鼠标驱动程序

入口参数: AX = 20H

出口参数: 无

功能21H

功能描述: 鼠标驱动程序复位

入口参数: AX = 21H

出口参数: 若 AX = -1——不成功, 否则, BX = 2

功能22H

功能描述: 设置鼠标驱动程序信息语言

入口参数: AX = 22H

BX = 语言代码: 0——英语、1——法语、2——荷兰语、3——德语、4——瑞典语、5——芬兰语、6——西班牙语、7——葡萄牙语、8——意大利语

出口参数: 无

功能23H

功能描述: 读取语种

入口参数: AX = 23H

出口参数: BX = 语言代码

功能24H

功能描述: 读取鼠标信息

入口参数:　AX = 24H

出口参数:　BH = 主版本号, BL = 辅版本号

CL = 中断请求号

CH = 鼠标类型: 1——Bus Mouse, 2——Serial Mouse, 3——InPort Mouse, 4——PS/2 Mouse, 5——HP Mouse

功能25H

功能描述:　读取鼠标驱动程序信息

入口参数:　AX = 25H

出口参数:　AX = 鼠标驱动程序信息:

位 15 ——0: 驱动程序是. SYS 文件, 否则, 为. COM 文件

位 14——0: 不完全鼠标显示驱动程序, 否则, 为完全的

位 13、12——00: 软件文本光标

01: 硬件文本光标

1X: 图形光标

功能26H

功能描述:　读取最大有效坐标

入口参数:　AX = 26H

出口参数:　BX = 鼠标驱动程序状态

CX = 最大水平坐标

DX = 最大垂直坐标

3. 其他 DOS 中断

中断INT 20H

功能描述:　终止当前正在运行的程序, 它是几种终止程序的运行方法之一

入口参数:　CS = PSP 的段地址

出口参数:　无

中断INT 22H

功能描述:　终止处理程序的地址, 该地址在程序装入内存运行前被放入 PSP 的 0AH ~0DH 的单元内。该中断指令从不直接书写在程序之中

中断INT 23H

功能描述:　Ctrl + C 处理程序。该中断指令从不直接书写在程序之中

中断INT 24H

功能描述:　致命错误处理程序。该中断指令从不直接书写在程序之中

中断INT 25H

功能描述:　绝对读磁盘, 直接从逻辑设备中读出数据到内存单元中

入口参数:　AL = 驱动器号(0 = A, 1 = B, …)

分区容量　≤32 MB, 有:　X = 读出的扇区数

DX = 起始扇区数

DS: BX = 存放数据缓冲区的地址

否则, 有:　X = - 1

DS: BX = 参数块缓冲区的地址, 该参数块的结构如下:

字节	描述
00 ~03H	32 位扇区数

04～05H　　将被读出的扇区数
06～07H　　存放数据的缓冲区的偏移量
08～09H　　存放数据的缓冲区的段地址

出口参数:　CF = 0——操作成功, 否则, AX = 错误号, 其含义见如下错误代码表

错误代码	错误含义
80H	附件响应失败
40H	定位操作失败
20H	设备控制器失败
10H	数据错(错误的 CRC)
08H	DMA 失败
04H	需要的扇区未发现
02H	错误的地址标志
01H	错误命令

INT 26H

功能描述:　绝对写磁盘, 直接把内存单元中的内容写入逻辑设备

入口参数:　与前面的 INT 25H 相一致

出口参数:　与前面的 INT 25H 相一致

INT 27H

功能描述:　终止并驻留在内存

入口参数:　CS = PSP 的段值
　　DX = 被保护程序最后一个字节的偏移量再加 1

出口参数:　无

INT 28H

功能描述:　DOS 空闲中断

入口参数:　无

出口参数:　无

INT 2FH

功能描述:　多重中断服务, 允许多个驻留程序通过单个中断与其他进程通信

入口参数:　AH = 标识号, AL = 功能号, 功能号及其含义如下:

功能号	功能描述
01H	假脱机打印
06H	驻留 ASSIGN 命令
10H	驻留 SHARE 命令
B7H	驻留 APPEND 命令

出口参数:　若入口 AL 为 0, 则出口 AL = 0FFH, 否则, 其值取决于处理程序

4. BIOS 中断

(1) 显示服务(Video Service——INT 10H)

功能00H

功能描述:　设置显示器模式

入口参数:　AH = 00H
　　AL = 显示器模式, 见下表

出口参数:　无

可用的显示模式如下所列:

显示模式		显示模式属性	显示模式		显示模式属性
00H	40× 25	16 色 文本	01H	40× 25	16 色 文本
02H	80× 25	16 色 文本	03H	80× 25	16 色 文本
04H	320× 200	4 色	05H	320× 200	4 色
06H	640× 200	2 色	07H	80× 25	2 色 文本
08H	160× 200	16 色	09H	320× 200	16 色
0AH	640× 200	4 色	0BH	保留	
0CH	保留		0DH	320× 200	16 色
0EH	640× 200	16 色	0FH	640× 350	2(单色)
10H	640× 350	4 色	10H	640× 350	16 色
11H	640× 480	2 色	12H	640× 480	16 色
13H	640× 480	256 色			

对于超级 VGA 显示卡, 可用 AX = 4F02H 和下列 BX 的值来设置其显示模式。

BX		显示模式属性	BX		显示模式属性
100H	640× 400	256 色	101H	640× 480	256
102H	800× 600	16 色	103H	800× 600	256 色
104H	1024× 768	16 色	105H	1024× 768	256 色
106H	1280× 1024	16 色	107H	1280× 1024	256 色
108H	80× 60	文本模式	109H	132× 25	文本模式
10AH	132× 43	文本模式	10BH	132× 50	文本模式
10CH	132× 60	文本模式			

功能01H

功能描述: 设置光标形状

入口参数: AH = 01H

CH 低四位 = 光标的起始行

CL 低四位 = 光标的终止行

出口参数: 无

功能02H

功能描述: 用文本坐标下设置光标位置

入口参数: AH = 02H

BH = 显示页码

DH = 行(Y 坐标)

DL = 列(X 坐标)

出口参数: 无

功能03H

功能描述: 在文本坐标下, 读取光标各种信息

入口参数: AH = 03H

BH = 显示页码

出口参数: CH = 光标的起始行

CL = 光标的终止行

DH = 行(Y 坐标)

DL = 列(X 坐标)

功能04H

功能描述: 获取当前状态和光笔位置

入口参数: AH = 04H

出口参数: AH = 00h——光笔未按下/未触发, 01h——光笔已按下/已触发

BX = 像素列(图形 X 坐标)

CH = 像素行(图形 Y 坐标, 显示模式: 04H ~06H)

CX = 像素行(图形 Y 坐标, 显示模式: 0DH ~10H)

DH = 字符行(文本 Y 坐标)

DL = 字符列(文本 X 坐标)

功能05H

功能描述: 设置显示页, 即选择活动的显示页

入口参数: AH = 05H

AL = 显示页

对于 CGA, EGA, MCGA 和 VGA, 其显示页如下表所列:

模式	页数	显示器类型
00H、01H	0 ~7	CGA, EGA, MCGA, VGA
02H、03H	0 ~3	CGA
02H、03H	0 ~7	EGA, MCGA, VGA
07H	0 ~7	EGA, VGA
0DH	0 ~7	EGA, VGA
0EH	0 ~3	EGA, VGA
0FH	0 ~1	EGA, VGA
10H	0 ~1	EGA, VGA

对于 PCjr:

AL = 80H——读取 CRT/CPU 页寄存器

81H——设置 CPU 页寄存器

82H——设置 CRT 页寄存器

83H——设置 CRT/CPU 页寄存器

BH = CRT 页(子功能号 82H 和 83H)

BL = CPU 页(子功能号 81H 和 83H)

出口参数: 对于前者, 无出口参数, 但对 PCjr 在子功能 80H ~83H 调用下, 有: BH = CRT 页寄存器, BL = CPU 页寄存器

功能06H 和 07H

功能描述: 初始化屏幕或滚屏

入口参数: AH = 06H——向上滚屏, 07H——向下滚屏

AL = 滚动行数(0——清窗口)

BH = 空白区域的默认属性

(CH、CL) = 窗口的左上角位置(Y 坐标, X 坐标)

(DH、DL) = 窗口的右下角位置(Y 坐标, X 坐标)

出口参数: 无

功能08H

功能描述: 读光标处的字符及其属性

入口参数: AH = 08H
BH = 显示页码
出口参数: AH = 属性
AL = 字符

功能09H
功能描述: 在当前光标处按指定属性显示字符
入口参数: AH = 09H
AL = 字符
BH = 显示页码
BL = 属性(文本模式) 或颜色(图形模式)
CX = 重复输出字符的次数
出口参数: 无

功能0AH
功能描述: 在当前光标处按原有属性显示字符
入口参数: AH = 0AH
AL = 字符
BH = 显示页码
BL = 颜色(图形模式, 仅适用于 PCjr)
CX = 重复输出字符的次数
出口参数: 无

功能0BH
功能描述: 设置调色板、背景色或边框
入口参数: AH = 0BH
设置颜色: BH = 00H, BL = 颜色
选择调色板: BH = 01H, BL = 调色板(320× 200、4 种颜色的图形模式)
出口参数: 无

功能0CH
功能描述: 写图形像素
入口参数: AH = 0CH
AL = 像素值
BH = 页码
(CX、DX) = 图形坐标列(X) 、行(Y)
出口参数: 无

功能0DH
功能描述: 读图形像素
入口参数: AH = 0DH
BH = 页码
(CX、DX) = 图形坐标列(X) 、行(Y)
出口参数: AL = 像素值

功能0EH
功能描述: 在 Teletype 模式下显示字符
入口参数: AH = 0EH
AL = 字符

BH = 页码

BL = 前景色(图形模式)

出口参数: 无

功能0FH

功能描述: 读取显示器模式

入口参数: AH = 0FH

出口参数: AH = 屏幕字符的列数

AL = 显示模式(参见功能 00H 中的说明)

BH = 页码

功能10H

功能描述: 颜色中断。其子功能说明如下:

功能号	子功能名称	功能号	子功能名称
00H	设置调色板寄存器	01H	设置边框颜色
02H	设置调色板和边框	03H	触发闪烁/亮显位
07H	读取调色板寄存器	08H	读取边框颜色
09H	读取调色板和边框	10H	设置颜色寄存器
12H	设置颜色寄存器块	13H	设置颜色页状态
15H	读取颜色寄存器	17H	读取颜色寄存器块
1AH	读取颜色页状态	1BH	设置灰度值

功能11H

功能描述: 字体中断。其子功能说明如下:

子功能号	子功能名称
00H	装入用户字体和可编程控制器
10H	装入用户字体和可编程控制器
01H	装入 8× 14 ROM 字体和可编程控制器
11H	装入 8× 14 ROM 字体和可编程控制器
02H	装入 8× 8 ROM 字体和可编程控制器
12H	装入 8× 8 ROM 字体和可编程控制器
03H	设置块指示器
04H	装入 8× 16 ROM 字体和可编程控制器
14H	装入 8× 16 ROM 字体和可编程控制器
20H	设置 INT 1FH 字体指针
21H	为用户字体设置 INT 43H
22H	为 8× 14 ROM 字体设置 INT 43H
23H	为 8× 8 ROM 字体设置 INT 43H
24H	为 8× 16 ROM 字体设置 INT 43H
30H	读取字体信息

功能12H

功能描述: 显示器的配置中断。其子功能说明如下:

功能号	功能名称	功能号	功能名称
10H	读取配置信息	20H	选择屏幕打印
30H	设置扫描行	31H	允许/禁止装入默认调色板
32H	允许/禁止显示	33H	允许/禁止灰度求和

34H —— 允许/禁止光标模拟　　35H —— 切换活动显示

36H —— 允许/禁止屏幕刷新

功能13H

功能描述: 在 Teletype 模式下显示字符串

入口参数: AH = 13H

BH = 页码

BL = 属性(若 AL = 00H 或 01H)

CX = 显示字符串长度

(DH、DL) = 坐标(行、列)

ES: BP = 显示字符串的地址

AL = 显示输出方式

0——字符串中只含显示字符,其显示属性在 BL 中。显示后,光标位置不变

1——字符串中只含显示字符,其显示属性在 BL 中。显示后,光标位置改变

2——字符串中含显示字符和显示属性。显示后,光标位置不变

3——字符串中含显示字符和显示属性。显示后,光标位置改变

出口参数: 无

功能1AH

功能描述: 读取/设置显示组合编码,仅 PS/2 有效,在此从略

功能1BH

功能描述: 读取功能/状态信息,仅 PS/2 有效,在此从略

功能1CH

功能描述: 保存/恢复显示器状态,仅 PS/2 有效,在此从略

(2) 直接磁盘服务(Direct Disk Service——INT 13H)

功能00H

功能描述: 磁盘系统复位

入口参数: AH = 00H

DL = 驱动器,00H～7FH:软盘;80H～0FFH:硬盘

出口参数: CF = 0——操作成功,AH = 00H,否则,AH = 状态代码,参见功能号 01H 中的说明

功能01H

功能描述: 读取磁盘系统状态

入口参数: AH = 01H

DL = 驱动器,00H～7FH:软盘;80H～0FFH:硬盘

出口参数: AH = 00H,AL = 状态代码,其定义如下:

00H —— 无错　　01H — 非法命令

02H ——地址目标未发现　　03H —— 磁盘写保护(软盘)

04H —— 扇区未发现　　05H —— 复位失败(硬盘)

06H —— 软盘取出(软盘)　　07H —— 错误的参数表(硬盘)

08H —— DMA 越界(软盘)　　09H —— DMA 超过 64 KB 界限

0AH —— 错误的扇区标志(硬盘)　　0BH—— 错误的磁道标志(硬盘)

0CH —— 介质类型未发现(软盘)　　0DH —— 格式化时非法扇区号(硬盘)

0EH—— 控制数据地址目标被发现(硬盘)　　0FH—— DMA 仲裁越界(硬盘)

10H —— 不正确的 CRC 或 ECC 编码　　11H —— ECC 校正数据错(硬盘)

CRC: Cyclic Redundancy Check code

ECC: Error Checking & Correcting code

20H —— 控制器失败　　40H —— 查找失败

80H —— 磁盘超时(未响应)　　AAH —— 驱动器未准备好(硬盘)

BBH —— 未定义的错误(硬盘)　　CCH —— 写错误(硬盘)

E0H —— 状态寄存器错(硬盘)　　FFH —— 检测操作失败(硬盘)

功能02H

功能描述: 读扇区

入口参数: AH = 02H

AL = 扇区数

CH = 柱面

CL = 扇区

DH = 磁头

DL = 驱动器, 00H ~7FH: 软盘; 80H ~0FFH: 硬盘

ES: BX = 缓冲区的地址

出口参数: CF = 0——操作成功, AH = 00H, AL = 传输的扇区数, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能03H

功能描述: 写扇区

入口参数: AH = 03H

AL = 扇区数

CH = 柱面

CL = 扇区

DH = 磁头

DL = 驱动器, 00H ~7FH: 软盘; 80H ~0FFH: 硬盘

ES: BX = 缓冲区的地址

出口参数: CF = 0——操作成功, AH = 00H, AL = 传输的扇区数, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能04H

功能描述: 检验扇区

入口参数: AH = 04H

AL = 扇区数

CH = 柱面

CL = 扇区

DH = 磁头

DL = 驱动器, 00H ~7FH: 软盘; 80H ~0FFH: 硬盘

ES: BX = 缓冲区的地址

出口参数: CF = 0——操作成功, AH = 00H, AL = 被检验的扇区数, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能05H

功能描述: 格式化磁道

入口参数: AH = 05H

AL = 交替(Interleave)

CH = 柱面

DH = 磁头

DL = 驱动器, 00H ~7FH: 软盘; 80H ~0FFH: 硬盘

ES: BX = 地址域列表的地址

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能06H

功能描述: 格式化坏磁道

入口参数: AH = 06H

AL = 交替

CH = 柱面

DH = 磁头

DL = 80H ~0FFH: 硬盘

ES: BX = 地址域列表的地址

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能07H

功能描述: 格式化驱动器

入口参数: AH = 07H

AL = 交替

CH = 柱面

DL = 80H ~0FFH: 硬盘

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能08H

功能描述: 读取驱动器参数

入口参数: AH = 08H

DL = 驱动器, 00H ~7FH: 软盘; 80H ~0FFH: 硬盘

出口参数: CF = 1——操作失败, AH = 状态代码, 参见功能号 01H 中的说明, 否则,

BL = 01H——360 KB

= 02H——1.2 MB

= 03H——720 KB

= 04H——1.44 MB

CH = 柱面数的低 8 位

CL 的位 7 - 6 = 柱面数的该 2 位

CL 的位 5 - 0 = 扇区数

DH = 磁头数

DL = 驱动器数

ES: DI = 磁盘驱动器参数表地址

功能09H

功能描述: 初始化硬盘参数

入口参数: AH = 09H

DL = 80H ~0FFH: 硬盘(还有有关参数表问题, 在此从略)

出口参数: CF = 0—操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能0AH

功能描述: 读长扇区, 每个扇区随带 4 个字节的 ECC 编码

入口参数: AH = 0AH

AL = 扇区数

CH = 柱面

CL = 扇区

DH = 磁头

DL = 80H ~0FFH: 硬盘

ES: BX = 缓冲区的地址

出口参数: CF = 0——操作成功, AH = 00H, AL = 传输的扇区数, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能0BH

功能描述: 写长扇区, 每个扇区随带 4 个字节的 ECC 编码

入口参数: AH = 0BH

AL = 扇区数

CH = 柱面

CL = 扇区

DH = 磁头

DL = 80H ~0FFH: 硬盘

ES: BX = 缓冲区的地址

出口参数: CF = 0——操作成功, AH = 00H, AL = 传输的扇区数, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能0CH

功能描述: 查寻

入口参数: AH = 0CH

CH = 柱面的低 8 位

CL(7 - 6 位) = 柱面的高 2 位

DH = 磁头

DL = 80H ~0FFH: 硬盘

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能0DH

功能描述: 硬盘系统复位

入口参数: AH = 0DH

DL = 80H ~0FFH: 硬盘

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能0EH

功能描述: 读扇区缓冲区

入口参数: AH = 0EH

ES: BX = 缓冲区的地址

出口参数: CF = 0——操作成功, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能0FH

功能描述: 写扇区缓冲区

入口参数: AH = 0FH

ES: BX = 缓冲区的地址

出口参数: CF =0——操作成功, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能10H

功能描述:　读取驱动器状态

入口参数:　AH = 10H

DL = 80H～0FFH: 硬盘

出口参数:　CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能11H

功能描述:　校准驱动器

入口参数:　AH = 11H

DL = 80H～0FFH: 硬盘

出口参数:　CF = 0——操作成功, AH = 00H, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能12H

功能描述:　控制器 RAM 诊断

入口参数:　AH = 12H

出口参数:　CF = 0——操作成功, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能13H

功能描述:　控制器驱动诊断

入口参数:　AH = 13H

出口参数:　CF = 0——操作成功, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能14H

功能描述:　控制器内部诊断

入口参数:　AH = 14H

出口参数:　CF = 0——操作成功, 否则, AH = 状态代码, 参见功能号 01H 中的说明

功能15H

功能描述:　读取磁盘类型

入口参数:　AH = 15H

DL = 驱动器, 00H～7FH: 软盘; 80H～0FFH: 硬盘

出口参数:　CF = 1——操作失败, AH = 状态代码, 参见功能号 01H 中的说明,

否则, AH = 00H—— 未安装驱动器

= 01H —— 无改变线支持的软盘驱动器

= 02H —— 带有改变线支持的软盘驱动器

= 03H —— 硬盘, CX: DX = 512 字节的扇区数

功能16H

功能描述:　读取磁盘变化状态

入口参数:　AH = 16H

DL = 00H～7FH: 软盘

出口参数:　CF = 0——磁盘未改变, AH = 00H, 否则, AH = 06H, 参见功能号 01H 中的说明

功能17H

功能描述:　设置磁盘类型

入口参数:　AH = 17H

DL = 00H～7FH: 软盘

AL = 00H — 未用

= 01H —— 360 KB 软盘在 360 KB 驱动器中

= 02H —— 360 KB 软盘在 1. 2 MB 驱动器中

= 03H —— 1. 2 MB 软盘在 1. 2 MB 驱动器中

= 04H —— 720 KB 软盘在 720 KB 软盘驱动器中

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态编码, 参见功能号 01H 中的说明

功能18H

功能描述: 设置格式化媒体类型

入口参数: AH = 18H

CH = 柱面数

CL = 每磁道的扇区数

DL = 00H～7FH: 软盘

出口参数: CF = 0——操作成功, AH = 00H, ES: DI = 介质类型参数表地址, 否则, AH = 状态编码, 参见功能号 01H 中的说明

功能19H

功能描述: 磁头保护, 仅在 PS/2 中有效, 在此从略

功能1AH

功能描述: 格式化 ESDI 驱动器, 仅在 PS/2 中有效, 在此从略

(3) 串行口服务(Serial Port Service——INT 14H)

功能00H

功能描述: 初始化通信口

入口参数: AH = 00H

DX = 初始化通信口号(0 = COM1, 1 = COM2, …)

AL = 初始化参数, 参数的说明如下:

波特率	奇偶位	停止位	字的位数
765	43	2	10
000 = 110	X0 = None	0 = 1 bit	10 = 7 bits
001 = 150	01 = Odd	1 = 2 bits	11 = 8 bits
010 = 300	11 = Even		
011 = 600			
100 = 1200			
101 = 2400			
110 = 4800			
111 = 9600			

对于 PS/2, 可用 INT 14H 的功能 04H 和 05H 来初始化其通信速率大于 9 600。

出口参数: AH = 通信口状态, 各状态位为 1 时的含义如下:

位 7——超时　　位 6——传递移位寄存器为空

位 5——传递保持寄存器为空　　位 4——发现终止

位 3——发现帧错误　　位 2——发现奇偶错

位 1——发现越界错　　位 0——接受数据准备好

AL = Modem 状态

位 7——接受单线信号诊断　　位 6——环指示器

位 5——数据发送准备好　　位 4——清除数据, 再发送

位 3——改变在接受线上的信号诊断　　位 2——后边界环指示器

位 1——改变“数据准备好”状态　　位 0——改变“清除—发送”状态

功能01H

功能描述: 向通信口输出字符

入口参数:　AH = 01H

AL = 字符

DX = 初始化通信口号(0 = COM1, 1 = COM2, …)

出口参数:　AL 的值不变

AH 的位 7 = 0——操作成功, 通信口状态, AH 的位 6～0 是其状态位

功能02H

功能描述:　从通信口读入字符

入口参数:　AH = 02H

DX = 初始化通信口号(0 = COM1, 1 = COM2, …)

出口参数:　AL = 接受的字符

AH 的位 7 = 0——操作成功, 通信口状态, AH 的位 6～0 是其状态位

功能03H

功能描述:　读取通信口状态

入口参数:　AH = 03H

DX = 初始化通信口号(0 = COM1, 1 = COM2, …)

出口参数:　AH = 通信口状态, AL = Modem 状态, 参见功能号 00H 中的说明

功能04H

功能描述:　扩充初始化通信口, 仅在 PS/2 中有效, 在此从略

(4) 杂项系统服务(Miscellaneous System Service——INT 15H)

功能00H

功能描述:　开盒式磁带机马达

入口参数:　AH = 00H

出口参数:　CF = 0——操作成功, 否则, AH = 状态(86H, 若未安装盒式磁带机)

功能01H

功能描述:　关盒式磁带机马达

入口参数:　AH = 01H

出口参数:　CF = 0——操作成功, 否则, AH = 状态(86H, 若未安装盒式磁带机)

功能02H

功能描述:　读盒式磁带机

入口参数:　AH = 02H

CX = 读入的字节数

ES: BX = 存放数据的缓冲区地址

出口参数:　CF = 0——操作成功, DX = 实际读入的字节数, ES: BX 指向最后一个字节的后面地址, 否则, AH = 状态码, 其值含义如下:

01H —— CRC 校验码错　　80H —— 非法命令

02H —— 位信号混乱　　86H —— 未安装盒式磁带机

04H —— 无发现数据

功能03H

功能描述:　写盒式磁带机

入口参数:　AH = 03H

CX = 要写入的字节数

ES: BX = 已存数据的缓冲区地址

出口参数:　CF = 0——操作成功, CX = 00H, ES: BX 指向最后一个字节的后面地址, 否则, AH =

状态码, 其值含义如下:

80H —— 非法命令　　86H —— 未安装盒式磁带机

功能0FH

功能描述: 格式化 ESDI 驱动器定期中断, 仅在 PS/2 中有效, 在此从略

功能21H

功能描述: 读/写自检(POST) 错误记录, 仅在 PS/2 中有效, 在此从略

功能4FH

功能描述: 键盘截听, 仅在 PS/2 中有效, 在此从略

功能80H

功能描述: 打开设备

入口参数: AH = 80H

BX = 设备号

CX = 进程号

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态码

功能81H

功能描述: 关闭设备

入口参数: AH = 81H

BX = 设备号

CX = 进程号

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态码

功能82H

功能描述: 进程终止

入口参数: AH = 81H

BX = 进程号

出口参数: CF = 0——操作成功, AH = 00H, 否则, AH = 状态码

功能83H

功能描述: 事件等待

入口参数: AH = 83H

若需要事件等待, 则:

AL = 00H

CX: DX = 千分秒

ES: BX = 信号量字节的地址

否则, 调用参数为

AL = 01H

出口参数: 若调用时, AL = 00H, 操作成功——CF = 0, 否则, CF = 1

功能84H

功能描述: 读游戏杆

入口参数: AH = 84H

DX = 00H——读取开关设置

= 01H——读取阻力输入

出口参数: CF = 1H——操作失败, 否则,

DX = 00H 时, AL = 开关设置(位 7 ~4)

DX = 01H 时, AX, BX, CX 和 DX 分别为 A(x) , A(y) , B(x) 和 B(y) 的值

功能85H

功能描述:　**系统请求**(SysReq) **键**

入口参数:　AH = 85H

AL = 00H——**键按下**

= 01H——**键放开**

出口参数:　CF = 0——**操作成功**, AH = 00H, **否则**, AH = **状态码**

功能86H

功能描述:　**延迟**

入口参数:　AH = 86H

CX: DX = **千分秒**

出口参数:　CF = 0——**操作成功**, AH = 00H

功能87H

功能描述:　**从常规内存和扩展内存之间移动扩展内存块**

入口参数:　AH = 87H

CX = **移动的字数**

ES: SI = GDT(Global Descriptor Table) **的地址, 其结构定义如下**:

偏移量	**存储的信息**
00h ~0Fh	**保留, 但是全为** 0
10h ~11h	**段的长度**(2CX - 1 **或更大**)
12h ~14h	24 **位源地址**
15h	**访问权限字节**(**其值为** 93h)
16h ~17h	**保留, 但是全为** 0
18h ~19h	**段的长度**(2CX - 1 **或更大**)
1Ah ~1Ch	24 **位目标源地址**
1Dh	**访问权限字节**(**其值为** 93h)
1Eh ~2Fh	**保留, 但是全为** 0

出口参数:　CF = 0——**操作成功**, AH = 00H, **否则**, AH = **状态码, 其含义如下**:

01H —— RAM **奇偶错**

02H —— **异常中断错**

03H —— 20 **号线门地址失败**

功能88H

功能描述:　**读取扩展内存大小**

入口参数:　AH = 88H

出口参数:　AX = **扩展内存字节数**(**以** KB **为单位**)

功能89H

功能描述:　**进入保护模式**, CPU **从实模式进入保护模式**

入口参数:　AH = 89H

BH = IRQ0 **的中断号**

BL = IRQ8 **的中断号**

ES: SI = GDT **的地址**(**参见功能号** 87H)

出口参数:　CF = 1——**操作失败**, AH = 0FFH, **否则**, AH = 00H。CS, DS, ES **和** SS **都是用户定义的选择器**

功能90H

功能描述: 设备等待

入口参数: AH = 90H

AL = 驱动器类型, 具体的驱动器类型定义如下:

= 00H ~7FH——串行再重用设备

= 80H ~0BFH——可重入式设备

= 0C0H ~0FFH——等待访问设备, 没有自检功能

00h —— 磁盘　　02h —— 键盘

80h —— 网络　　FDh —— 软盘马达启动

01h—— 软盘　　03h —— 点设备(Pointing Device)

FCh —— 硬盘复位　　FEh —— 打印机

ES: BX = 对驱动器类型 80H ~0FFH 的请求块地址

出口参数: CF = 1——操作失败, 否则, AH = 00H

功能91H

功能描述: 设备加电自检

入口参数: AH = 91H

AL = 00H ~7FH——串行再重用设备

= 80H ~0BFH——可重入式设备

出口参数: AH = 00H

功能0C0H

功能描述: 读取系统环境

入口参数: AH = 0C0H

出口参数: ES: BX = 配置表地址, 配置表的定义如下:

偏移量	含义说明
00h ~01h	表的大小(字节数)
02h	系统模型
03h	系统子模型
04h	BIOS 版本号
05h	配置标志, 其各位为 1 时的说明如下:
	位 7——DMA 通道 3 使用
	位 6——存在从属 8259
	位 5——实时时钟有效
	位 4——键盘截听有效
	位 3——等待外部事件有效
	位 2——扩展 BIOS 数据区
	位 1——微通道设施
	位 0——保留
06h ~09h	保留

功能C1H

功能描述: 读取扩展 BIOS 数据区地址, 仅在 PS/2 中有效, 在此从略

功能C2H

功能描述: 鼠标图形, 仅在 PS/2 中有效, 在此从略

功能C3H

功能描述: 设置 WatcHdog 超时, 仅在 PS/2 中有效, 在此从略

功能C4H

功能描述:　可编程选项选择,仅在 PS/2 中有效,在此从略

(5) **键盘服务**(Keyboard Service——INT 16H)

功能00H **和** 10H

功能描述:　从键盘读入字符

入口参数: AH = 00H——读键盘

= 10H——读扩展键盘,可根据 0000:0496H 单元的内容判断:扩展键盘是否有效

出口参数:　AH = 键盘的扫描码

AL = 字符的 ASCII 码

功能01H **和** 11H

功能描述:　读取键盘状态

入口参数: AH = 01H——检查普通键盘

= 11H——检查扩展键盘

出口参数:　ZF = 1——无字符输入,否则,AH = 键盘的扫描码,AL = ASCII 码。

功能02H **和** 12H

功能描述:　读取键盘标志

入口参数: AH = 02H——普通键盘的移位标志

= 12H——扩展键盘的移位标志

出口参数:　AL = 键盘标志(02H 和 12H 都有效),其各位的值为 1 时的含义如下:

位 7——Ins 开状态	位 3——Alt 键按下
位 6——Caps Lock 开状态	位 2——Ctrl 键按下
位 5——Num Lock 开状态	位 1——左 Shift 键按下
位 4——Scroll Lock 开状态	位 0——右 Shift 键按下

AH = 扩展键盘的标志(12H 有效),其各位的值为 1 时的含义如下:

位 7——SysReq 键按下	位 3——右 Alt 键按下
位 6——Caps Lock 键按下	位 2——右 Ctrl 键按下
位 5——Num Lock 键按下	位 1——左 Alt 键按下
位 4——Scroll 键按下	位 0——左 Ctrl 键按下

功能03H

功能描述:　设置重复率

入口参数:　AH = 03H

对于 PC/AT 和 PS/2:　L = 05H

BH = 重复延迟

BL = 重复率

对于 PCjr:　AL = 00H——装入默认的速率和延迟

= 01H——增加初始延迟

= 02H——重复频率降低一半

= 03H——增加延迟和降低一半重复频率

= 04H——关闭键盘重复功能

出口参数:　无

功能04H

功能描述:　设置键盘点击

入口参数:　AH = 04H

AL = 00H——关闭键盘点击功能

= 01H——打开键盘点击功能

出口参数: 无

功能05H

功能描述: 字符及其扫描码进栈

入口参数: AH = 05H

CH = 字符的描述码

CL = 字符的 ASCII 码

出口参数: CF = 1——操作成功, AL = 00H, 否则, AL = 01H

(6) 并行口服务(Parallel Port Service——INT 17H)

功能00H

功能描述: 向打印机输出字符

入口参数: AH = 00H

AL = 输出的字符

DX = 打印机号(0 - LPT1, 1 - LPT2, 2 - LPT3, …)

出口参数: AH = 打印机状态。其各位为 1 时的含义如下:

位 7——打印机空闲　位 3——I/O 错误

位 6——打印机响应　位 2——保留

位 5——无纸　　位 1——保留

位 4——打印机被选　位 0——打印机超时

功能01H

功能描述: 初始化打印机端口

入口参数:

AH = 01H

DX = 打印机号(0 - LPT1, 1 - LPT2, 2 - LPT3, …)

出口参数: AH = 打印机状态, 各位定义如下功能 00H 所示

功能02H

功能描述: 读取打印机状态

入口参数: AH = 02H

DX = 打印机号(0 - LPT1, 1 - LPT2, 2 - LPT3, …)

出口参数: AH = 打印机状态, 各位定义如下功能 00H 所示

(7) 时钟服务(Clock Service——INT 1AH)

功能00H

功能描述: 读取时钟“滴答”计数

入口参数: AH = 00H

出口参数: AL = 00H——未过午夜, 否则, 表示已过午夜

CX: DX = 时钟“滴答”计数

功能01H

功能描述: 设置时钟“滴答”计数

入口参数: AH = 01H

CX: DX = 时钟“滴答”计数

出口参数: 无

功能02H

功能描述:　读取时间

入口参数:　AH = 02H

出口参数:　CH = BCD 码格式的小时

CL = BCD 码格式的分钟

DH = BCD 码格式的秒

DL = 00H——标准时间, 否则, 夏令时

CF = 0——时钟在走, 否则, 时钟停止

功能03H

功能描述:　设置时间

入口参数:　AH = 03H

CH = BCD 码格式的小时

CL = BCD 码格式的分钟

DH = BCD 码格式的秒

DL = 00H——标准时间, 否则, 夏令时

出口参数:　无

功能04H

功能描述:　读取日期

入口参数:　AH = 04H

出口参数:　CH = BCD 码格式的世纪

CL = BCD 码格式的年

DH = BCD 码格式的月

DL = BCD 码格式的日

CF = 0——时钟在走, 否则, 时钟停止

功能05H

功能描述:　设置日期

入口参数:　AH = 05H

CH = BCD 码格式的世纪

CL = BCD 码格式的年

DH = BCD 码格式的月

DL = BCD 码格式的日

出口参数:　无

功能06H

功能描述:　设置闹钟

入口参数:　AH = 06H

CH = BCD 码格式的小时

CL = BCD 码格式的分钟

DH = BCD 码格式的秒

出口参数:　CF = 0——操作成功, 否则, 闹钟已设置或时钟已停止

功能07H

功能描述:　闹钟复位

入口参数:　AH = 07H

出口参数:　无

功能0AH

功能描述: 读取天数计数, 仅在 PS/2 有效, 在此从略

功能0BH

功能描述: 设置天数计数, 仅在 PS/2 有效, 在此从略

功能80H

功能描述: 设置声音源信息

入口参数: AH = 80H

AL = 声音源

= 00H——8253 可编程计时器, 通道 2

= 01H——盒式磁带输入

= 02H——I/O 通道上的“ Audio In ”

= 03H——声音产生芯片

出口参数: 无

(8) **直接系统服务**(Direct System Service)

INT 00H —“ 0 ”作除数

INT 01H — 单步中断

INT 02H —非屏蔽中断(NMI)

INT 03H — 断点中断

INT 04H — 算术溢出错误

INT 05H — 打印屏幕和 BOUND 越界

INT 06H — 非法指令错误

INT 07H — 处理器扩展无效

INT 08H — 时钟中断

INT 09H — 键盘输入

INT 0BH — 通信口(COM2:)

INT 0CH — 通信口(COM1:)

INT 0EH — 磁盘驱动器输入/输出

INT 11H — 读取设备配置

INT 12H — 读取常规内存大小(返回值 AX 为内存容量, 以 KB 为单位)

INT 18H — ROM BASIC

INT 19H — 重启动系统

INT 1BH — Ctrl + Break 处理程序

INT 1CH — 用户时钟服务

INT 1DH — 指向显示器参数表指针

INT 1EH — 指向磁盘驱动器参数表指针

INT 1FH — 指向图形字符模式表指针

附录 E　Pentium 指令的执行周期数

1. 数据传送指令

(1) 传送指令

指令的语法	举例	周期数
MOV reg, reg	mov bp, sp	1
MOV mem, reg	mov array[di] , bx	1
MOV reg, mem	mov bx, pointer	1
MOV mem, immed	mov [bx] ,15	1
MOV reg, immed	mov cx,256	1
MOV mem, accum	mov total, ax	1
MOV accum, mem	mov al, string	1
MOV segreg, reg16	mov ds, ax	2,3
MOV segreg, mem16	mov es, psp	2,3
MOV reg16, segreg	mov ax, ds	1
MOV mem16, segreg	mov stack_save, ss	1
MOV reg32, controlreg	mov eax, cr0	22
	mov eax, cr2	12
	mov eax, cr3	21, 46
	mov eax, cr4	14
MOV controlreg, reg32	mov cr0, eax	4
MOV reg32, debugreg	mov edx, dr0	DR0 - DR3, DR6, DR7 = 11
		DR4, DR5 = 12
MOV debugreg, reg32	mov dr0, ecx	DR0 - DR3,
	DR4, DR5 = 12	
	DR6, DR7 = 11	

(2) 传送 - 填充指令

MOVSX reg, reg	movsx bx, al	3
MOVSX reg, mem	movsx eax, bsign	3
MOVZX reg, reg	movzx bx, al	3
MOVZX reg, mem	movzx eax, bunsign	3

(3) 交换指令

XCHG reg, reg	xchg cx, dx	3
XCHG reg, mem	xchg bx, pointer	3
XCHG mem, reg	xchg [bx] , ax	3
XCHG accum, reg	xchg ax, cx	2
XCHG reg, accum	xchg cx, ax	2

(4) **取段地址和有效地址指令**

LDS reg, mem	lds si, fpointer	4
LES reg, mem	les di, fpointer	4
LFS reg, mem	lfs edi, fpointer	4
LGS reg, mem	lgs bx, fpointer	4
LSS reg, mem	lss bp, fpointer	4, pm = 8
LEA reg, mem	lea bx, npointer	1

(5) **进栈指令**

PUSH reg	push dx	1
PUSH mem	push [di]	2
PUSH segreg	push es	1
PUSH immed	push 15000	1
PUSHA	pusha	5
PUSHAD	pushad	5
PUSHF	pushf	4, pm = 3
PUSHFD	pushfd	4, pm = 3

(6) **出栈指令**

POP reg	pop cx	1
POP mem	pop param	3
POP segreg	pop es	3
POPA	popa	5
POPAD	popad	5
POPF	popf	6, pm = 4
POPFD	popfd	6, pm = 4

(7) **转换指令**

XLAT [[segreg:] mem]	xlat	4
XLATB [[segreg:] mem]	xlatb es: table	4

(8) **输入指令**

IN accum, immed	in ax, 60h	7, pm = 4, 21* (注) , vm = 19
IN accum, DX	in ax, dx	7, pm = 4, 21* , vm = 19
INS [ES:] dest, DX	ins es: instr, dx	9, pm = 6, 24* , vm = 22
INSB	Insb	9, pm = 6, 24* , vm = 22
INSW	Insw	9, pm = 6, 24* , vm = 22
INSD	Insd	9, pm = 6, 24* , vm = 22

注: 当 CPL≤IOPL 时, 执行时间是第一个时钟周期, 否则是第二个时钟周期。

(9) 输出指令

OUT immed8, accum	out 60h, al	12, pm = 9, 26, VM = 24
OUT DX, accum	out dx, ax	12, pm = 9, 25 VM = 24
OUTS DX, [segreg:] src	outs dx, buffer	13, pm = 10, 27, VM = 24
OUTSB [DX, [segreg:] src]	outsb	13, pm = 10, 27, VM = 24
OUTSW [DX, [segreg:] src]	outsw	13, pm = 10, 27, VM = 24
OUTSD [DX, [segreg:] src]	outsd	13, pm = 10, 27, VM = 24

2. 标志位操作指令

(1) 标志位操作指令

指令的语法	举例	周期数
CLC	clc	2
CMC	cmc	2
STC	stc	2
CLD	cld	2
STD	std	2
CLI	cli	7
STI	sti	7

(2) 标志位存取操作指令

SAHF	Sahf	2
LAHF	lahf	2

(3) 标志位堆栈操作指令

PUSHF	pushf	4, pm = 3
PUSHFD	pushfd	4, pm = 3
POPF	popf	6, pm = 4
POPFD	popfd	6, pm = 4

3. 算术运算指令

(1) 加法指令

指令的语法	举例	周期数
ADC reg, reg	adc dx, cx	1
ADC mem, reg	adc word ptr m16[2] , dx	3
ADC reg, mem	adc dx, dword ptr m32[2]	2
ADD reg, reg	add ax, bx	1
ADD mem, reg	add total, cx	3

续表

指令的语法	举例	周期数
ADD reg, mem	add cx, incr	2
ADD reg, immed	add bx, 6	1
ADD mem, immed	add pointers[bx] [si] , 6	3
ADD accum, immed	add ax, 10	1
INC reg	inc bx	1
INC mem	inc vpage	3
XADD reg, reg	xadd dl, al	3
XADD mem, reg	xadd string, bl	4

(2) **减法指令**

SUB reg, reg	sub ax, bx	1
SUB mem, reg	sub array[di] , bi	3
SUB reg, mem	sub al, [bx]	2
SUB reg, immed	sub bl, 7	1
SUB mem, immed	sub total, 4000	3
SUB accum, immed	sub ax, 32000	1
SBB accum, immed	sbb ax, 320	1
SBB reg, immed	sbb dx, 45	1
SBB mem, immed	sbb word ptr m32[2] , 40	3
SBB reg, reg	sbb dx, cx	1
SBB mem, reg	sbb word ptr m32[2] , dx	3
SBB reg, mem	sbb dx, word ptr m32[2]	2
DEC reg	dec ax	1
DEC mem	dec counter	3
NEG reg	neg ax	1
NEG mem	neg balance	3

(3) **乘法指令**

MUL reg	mul bx	
MUL memX	mul word ptr [bx]	8, 16 - bit = 11
		32 - bit = 10
IMUL reg	imul dx	11
IMUL mem	imul factor	11
IMUL reg, immed	imul cx, 25	10
IMUL reg, reg, immed	imul dx, ax, 18	10
IMUL reg, mem, immed	imul bx, [si] , 60	10
IMUL reg, reg	imul cx, ax	10
IMUL reg, mem	imul dx, [si]	10

(4) 除法指令

DIV reg	div cx	byte = 17 word = 25
DIV mem	div [bx]	dword = 41
IDIV reg	idiv dl	8 - bit = 22; 16 - bit = 30
IDIV mem	idiv itemp	32 - bit = 46

(5) 类型转换指令

CBW	cbw	3
CWD	cwd	2
CWDE	cwde	3
CDQ	cdq	2

4. 逻辑运算指令

(1) 逻辑与操作指令

指令的语法	举例	周期数
AND reg, reg	and dx, bx	1
AND mem, reg	and bitmask, bx	3
AND reg, mem	and bx, masker	2
AND reg, immed	and dx, 0F7h	1
AND mem, immed	and masker, 100lb	3
AND accum, immed	and ax, 0B6h	1

(2) 逻辑或操作指令

OR reg, reg	or ax, dx	1
OR mem, reg	or bits, dx	3
OR reg, mem	or dx, color[di]	2
OR reg, immed	or dx, 110110b	1
OR mem, immed	or flag_rec, 8	3
OR accum, immed	or ax, 40h	1

(3) 逻辑非操作指令

NOT reg	not ax	1
NOT mem	not masker	3

(4) 逻辑异或操作指令

XOR reg, reg	xor cx, bx	1
XOR reg, mem	xor cx, flags	2
XOR mem, reg	xor [bp + 10] , cx	3
XOR reg, immed	xor bl, 1	1
XOR mem, immed	xor switches[bx] , 101b	3
XOR accum, immed	xor ax, 01010101b	1

5. 移位操作指令

(1) 算术左移指令

指令的语法	举例	周期数
SAL reg, 1	sal bx, 1	1
SAL mem, 1	sal word ptr m32[0] , 1	3
SAL reg, CL	sal ah, cl	4
SAL mem, CL	sal BYTE PTR [di] , cl	4
SAL reg, immed	sal cx, 6	1
SAL mem, immed	sal array[bx + di] , 14	3

(2) 算术右移指令

SAR reg, 1	sar di, 1	1
SAR mem, 1	sar count, 1	3
SAR reg, CL	sar bx, cl	4
SAR mem, CL	sar sign, cl	4
SAR reg, immed	sar bx, 5	1
SAR mem, immed	sar sign_count, 3	3

(3) 逻辑左移指令

SHL reg, 1	shl si, 1	1
SHL mem, 1	shl index, 1	3
SHL reg, CL	shl di, cl	4
SHL mem, CL	shl index, cl	4
SHL reg, immed	shl di, 2	1
SHL mem, immed	shl unsign, 4	3

(4) 逻辑右移指令

SHR reg, 1	shr dh, 1	1
SHR mem, 1	shr unsign[di] , 1	3
SHR reg, CL	shr dx, cl	4
SHR mem, CL	shr word ptr m32[2] , cl	4
SHR reg, immed	shr bx, 8	1
SHR mem, immed	shr mem16, 11	3

(5) 双精度左移指令

SHLD reg16, reg16, immed8	shld ax, dx, 10	4
SHLD reg32, reg32, immed8		
SHLD mem16, reg16, immed8	shld bits, cx, 5	4
SHLD mem32, reg32, immed8		
SHLD reg16, reg16, CL	shld ax, dx, cl	4
SHLD reg32, reg32, CL		
SHLD mem16, reg16, CL	shld masker, ax, cl	5
SHLD mem32, reg32, CL		

(6) **双精度右移指令**

SHRD reg16, reg16, immed8	shrd cx, si, 3	4
SHRD reg32, reg32, immed8		
SHRD mem16, reg16, immed8	shrd [di], dx, 5	4
SHRD mem32, reg32, immed8		
SHRD reg16, reg16, CL	shrd ax, dx, cl	4
SHRD reg32, reg32, CL		
SHRD mem16, reg16, CL	shrd [bx], ax, cl	5
SHRD mem32, reg32, CL		

6. 位操作指令

(1) **正向位扫描指令** BSF

指令的语法	**举例**	**周期数**
BSF reg16, reg16	bsf cx, bx	6～34
BSF reg32, reg32	bsf cx, bx	6～42
BSF reg16, mem16	bsf ecx, bitmask	6～35
BSF reg32, mem32	bsf ecx, bitmask	6～43

(2) **正向位扫描指令** BSR

BSR reg16, reg16	bsr cx, dx	7～39
BSR reg32, reg32	bsr ecx, edx	7～71
BSR reg16, mem16	bsr ax, bitmask	7～40
BSR reg32, mem32	bsr eax, bitmask	7～72

(3) **正向位扫描指令** BT

BT reg16, immed8* (注)	bt ax, 4	4
BT mem16, immed8	bt [bx], 4	4
BT reg16, reg16	bt ax, bx	4
BT mem16, reg16	bt [bx], dx	9

注: **操作数也可以是** 32 **位数**。

(4) **正向位扫描指令** BCT

BTC reg16, immed8*	btc edi, 4	7
BTC mem16, immed8*	btc color[di], 4	8
BTC reg16, reg16*	btc eax, ebx	7
BTC mem16, reg16*	btc [bp+8], si	13

(5) **正向位扫描指令** BTR

BTR reg16, immed8*	btr bx, 17	7
BTR mem16, immed8*	btr [bx] , 27	8
BTR reg16, reg16*	btr cx, di	7
BTR mem16, reg16*	btr rotate, cx	13

(6) **正向位扫描指令** BTS

BTS reg16, immed8*	bts ax, 4	7
BTS mem16, immed8*	bts maskit, 4	8
BTS reg16, reg16*	bts bx, ax	7
BTS mem16, reg16*	bts flags[bx] , cx	13

(7) **正向位扫描指令** TZST

TEST reg, reg	test dx, bx	1
TEST mem, reg	test flags, dx	2
TEST reg, immed	test cx, 30h	1
TEST mem, immed	test masker, 1	2
TEST accum, immed	test ax, 90h	1

7. 比较运算指令

(1) **比较指令**

指令的语法	**举例**	**周期数**
CMP reg, reg	cmp dl, cl	1
CMP mem, reg	cmp array[si] , bl	2
CMP reg, mem	cmp bh, array[si]	2
CMP reg, immed	cmp bx, 24	1
CMP mem, immed	cmp tester, 4000	2
CMP accum, immed	cmp ax, 1000	1

(2) **比较交换指令**

CMPXCHG mem, reg	cmpxchg string, bl	6
CMPXCHG reg, reg	cmpxchg bx, cx	6
CMPXCHG8B reg, mem64	cmpxchg8b ax, [bx]	10

8. 循环指令

(1) 循环指令

LOOP label	loop wend	5,6
LOOPE label	loope again	7,8
LOOPZ label	loopz again	7,8
LOOPNE label	loopne for_next	7,8
LOOPNZ label	loopnz for_next	7,8
JCXZ label	jcxz notfound	6,5
JECXZ label	jecxz notfound	6,5

9. 转移指令

(1) 无条件转移指令

指令的语法	举例	周期数
JMP label	jmp NEAR PTR distant	1
	jmp distant	3
JMP reg16	jmp ax	2
JMP mem16	jmp table[di]	2
JMP reg32	jmp eax	3
JMP mem32	jmp fpointer[si]	2
JMP mem48	jmp FWORD PTR [di]	4

(2) 条件转移指令

Jcondition label	je next	1

(3) 子程序调用指令

CALL label	call upcase	1
	call distant	4
CALL reg	call ax	2
CALL mem32	call [bx]	2
CALL mem32	call dword ptr [bx]	5

(4) 子程序返回指令

RETN	retn	2
RETN immed16	retn 8	3
RETF	retf	4,23
RETF immed16	retf 32	4,23

(5) 中断指令

INT immed8	int 25h	16, pm = 31, 48* (注)
INT 3	int 3	13, pm = 27, 44*
INTO	Into	13, pm = 27, 44*

注: 第一时间是同等优先级的中断时间, 第二时间为高优先级的中断时间。

(6) 中断返回指令

IRET	Iret	8* (注), 10, pm = 27
IRETD	Iretd	10, pm = 27
IRETF	Iretf	
IRETDF	Iretdf	

注: 实方式或虚拟 8086 方式。

10. 条件设置字节指令

SETcondition reg8	setc dh	1
SETcondition mem8	setle flag	2

11. 字符串操作指令

(1) 取字符串数据指令

指令的语法	举例	周期数
LODS [segreg:] src	lods es: source	2
LODSB [[segreg:] src]	Lodsb	2
LODSW [[segreg:] src]	Lodsw	2
LODSD [[segreg:] src]	Lodsd	2

(2) 置字符串数据指令

STOS [ES:] dest	stor es: dstring	3
STOSB [[ES:] dest]	stosb	3
STOSW [[ES:] dest]	stosw	3
STOSD [[ES:] dest]	stosd	3

(3) 字符串传送指令

MOVS [es:] dest, [segreg:] src	movs dest, es: source	4
MOVSB [[es:] dest, [segreg:] src]	movsb	4
MOVSW [[es:] dest, [segreg:] src]	movsw	4
MOVSD [[es:] dest, [segreg:] src]	movsd	4

(4) 输入字符串指令

INS [ES:] dest, DX	ins es: instr, dx	9, pm = 6, 24* , vm = 22
INSB	Insb	9, pm = 6, 24* , vm = 22
INSW	Insw	9, pm = 6, 24* , vm = 22
INSD	Insd	9, pm = 6, 24* , vm = 22

(5) 输出字符串指令

OUTS DX, [segreg:] src	outs dx, buffer	13, pm = 10, 27, VM = 24
OUTSB [DX, [segreg:] src]	outsb	13, pm = 10, 27, VM = 24
OUTSW [DX, [segreg:] src]	outsw	13, pm = 10, 27, VM = 24
OUTSD [DX, [segreg:] src]	outsd	13, pm = 10, 27, VM = 24

(6) 字符串比较指令

CMPS [segreg:] src, [ES:] dest	cmps source, es: dest	5
CMPSB [[segreg: [src,] ES:] dest]	cmpsb	5
CMPSW [[segreg: [src,] ES:] dest]	cmpsw	5
CMPSD [[segreg: [src,] ES:] dest]	cmpsd	5

(7) 字符串扫描指令

SCAS [ES] : dest	scas es: destin	4
SCASB	Scasb	4
SCASW	Scasw	4
SCASD	Scasd	4

(8) 重复前缀指令

REP INS dest, DX	rep ins dest, dx	11 + 3n, pm = (8, 25) + 3n*
REP MOVS dest, src	rep movs dest, source	6, 13n
REP OUTS DX, src	rep outs dx, source	13 + 4n, pm = (10, 27) + 4n*
REP LODS dest	rep lods dest	7, 7 + 3n
REP STOS dest	rep stos dest	6, 9 + 3n

注: 当 CPL≤IOPL 时, 执行时间是第一个时钟周期, 否则是第二个时钟周期。

(9) 相等重复前缀指令

REPE CMPS src, dest	repe cmps src, dest	7, 9 + 4n
REPE SCAS dest	repe scas dest	7, 9 + 4n

(10) 不相等重复前缀指令

REPNE CMPS src, dest	repne cmps src, dest	7,8 + 4n
REPNE SCAS dest	repne scas dest	7,9 + 4n

12. ASCII - BCD 码运算调整指令

指令的语法	举例	周期数
AAA	aaa	3
AAD	aad	10
AAM	aam	18
AAS	aas	3
DAA	daa	3
DAS	das	3

13. 处理器指令

HLT	Hlt	12
NOP	nop	1
WAIT	wait	1
LOCK	lock	1

14. 协处理器指令

指令的语法	举例	周期数
FBLD membcd	fbld packbck	48 - 58
FBSTP membcd	fbstp bcds[bx]	148 - 154
FLD reg	fld st(3)	1
FLD mem32real	fld longreal	1
FLD mem64real		1
FLD mem80real		3
FST reg	fst st	1
FST memreal	fst longs[bx]	2
FSTP reg	fstp st(3)	1
FSTP mem32real	fstp longreal	2
FSTP mem64real		2
FSTP mem80real		3
FXCH [reg]	fxchg st(3)	1
FILD memint	fild quads[si]	3,1
FIST memint	fist doubles[8]	6
FISTP memint	fistp longint	6

参 考 文 献

[1] 钱晓捷. 汇编语言程序设计. 第2版. 西安: 西安电子科技大学出版社, 2003

[2] 扬季文. 80x86 汇编语言程序设计教程. 北京: 清华大学出版社, 1998

[3] 沈美明. IBM - PC 汇编语言程序设计. 北京: 清华大学出版社, 1991

[4] Abel P. IBM PC Assembly Langeuage and Programming. 4th ed. 北京: 清华大学出版社; Prentice - Hall, 1998

[5] 胡元义. 汇编语言实践教程. 西安: 西安电子科技大学出版社, 2003

[6] 邹逢兴. 计算机硬件技术及应用基础. 长沙: 国防科技大学出版社, 2002

[7] Brey B B. The INTEL Microprocessors. 5th ed. 影印版. 北京: 高等教育出版社, 2001

[8] Triebel W A. 80X86 / Pentium 处理器硬件、软件及接口技术教程. 王克义等译. 北京: 清华大学出版社, 1998

[9] 罗云彬. Windows 环境下32 位汇编语言程序设计. 北京: 电子工业出版社, 2002

[10] 罗省贤. 汇编语言程序设计教程. 北京: 电子工业出版社, 2004

[11] 王成耀. 80x86 汇编语言程序设计. 北京: 人民邮电出版社, 2003